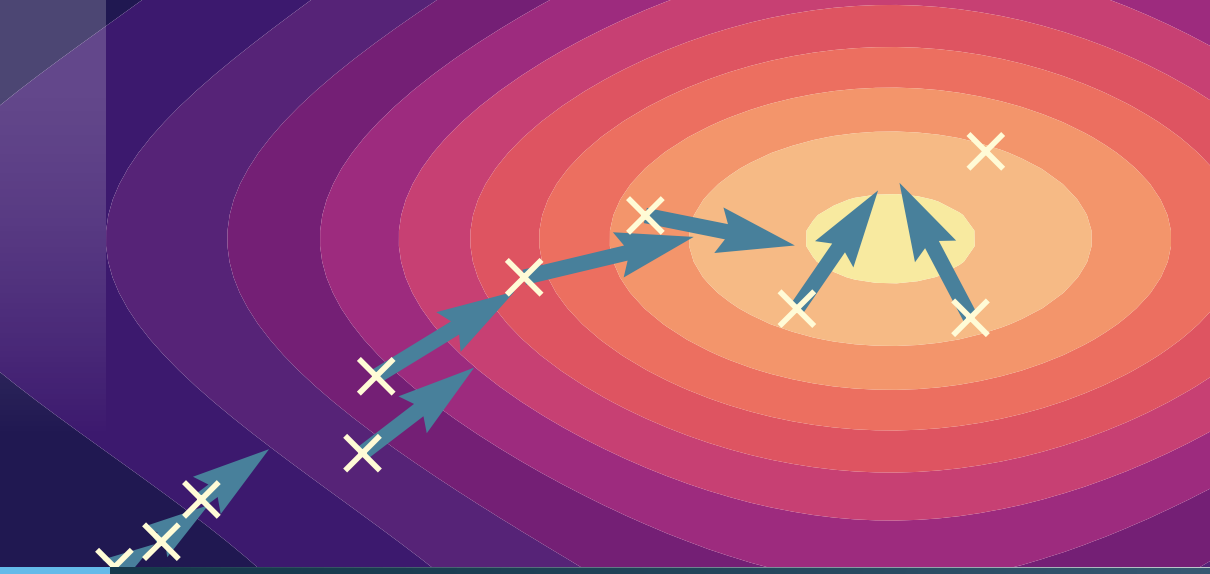


Jakub M. Tomczak

# Deep Generative Modeling

*Second Edition*

Springer

# Deep Generative Modeling

Jakub M. Tomczak

# Deep Generative Modeling

Second Edition

Springer

Jakub M. Tomczak
Eindhoven University of Technology
Eindhoven, The Netherlands

ISBN 978-3-031-64086-5 ISBN 978-3-031-64087-2 (eBook)
https://doi.org/10.1007/978-3-031-64087-2



This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

*To my beloved wife Ewelina,*
*my parents, and my brother.*

# Foreword

In the last decade, with the advance of deep learning, machine learning has made enormous progress. It has completely changed entire subfields of AI such as computer vision, speech recognition, and natural language processing. And more fields are being disrupted as we speak, including robotics, wireless communication, and the natural sciences.

Most advances have come from supervised learning, where the input (e.g., an image) and the target label (e.g., a "cat") are available for training. Deep neural networks have become uncannily good at predicting objects in visuals scenes and translating between languages. But obtaining labels to train such models is often time consuming, expensive, unethical, or simply impossible. That's why the field has come to the realization that unsupervised (or self-supervised) methods are key to make further progress.

This is no different for human learning: when human children grow up the amount of information that is consumed to learn about the world is mostly unlabeled. How often does anyone really what you see or hear in the world? We must learn the regularities of the world unsupervised, and we do this by searching for patterns and structure in the data.

And there is lots of structure to be learned! To illustrate this, imagine that we choose the three colors of each pixel of an image uniformly at random. The result will be an image that with overwhelmingly large probability will look like gibberish. The vast majority of image-space is filled with images that do not look like anything we see when we open our eyes. This means that there is a huge amount of structure that can be discovered, and so there is a lot to learn for children!

Of course, kids do not just stare into the world. Instead, they constantly interact with it. When children play, they test their hypotheses about the laws of physics, sociology, and psychology. When predictions are wrong, they are surprised and presumably update their internal models to make better predictions next time. It is reasonable to assume that this interactive play of an embodied intelligence is key to at least arrive at the type of human intelligence we are used to. This type of learning has clear parallels with reinforcement learning, where machine make plans, say to

play a game of chess, observe if they win or lose, and update their models of the world and strategies to act in them.

But it's difficult to make robots move around in the world to test hypotheses and actively acquire their own annotations. So, the more practical approach to learning with lots of data is unsupervised learning. This field has gained a huge amount of attention and has seen stunning progress recently. One only needs to look at the kind of images of non-existent human faces that we can now generate effortlessly to experience the uncanny sense of progress the field has made.

Unsupervised learning comes in many flavors. This book is about the kind we call probabilistic generative modeling. The goal of this subfield is to estimate a probabilistic model of the input data. Once we have such a model, we can generate new samples from it (i.e., new images of faces of people that do not exist).

A second goal is to learn abstract representations of the input. This latter field is called representation learning. The high-level representations self-organize the input into "disentangled" concepts, which could be the objects we are familiar with, such as cars and cats, and their relationships.

While disentangling has a clear intuitive meaning, it has proven to be a rather slippery concept to properly define. In the 1990s, people were thinking of statistically independent latent variables. The goal of the brain was to transform the highly dependent pixel representation into a much more efficient and less redundant representation of independent latent variables, which compresses the input and makes the brain more energy and information efficient.

Learning and compression are deeply connected concepts. Learning requires lossy compression of data because we are interested in generalization and not in storing the data. At the level of datasets, machine learning itself is about transferring a tiny fraction of the information present in a dataset into the parameters of a model and forgetting everything else.

Similarly, at the level of a single datapoint, when we process for example an input image, we are ultimately interested in the abstract high-level concepts present in that image, such as objects and their relations, and not in detailed, pixel level information. With our internal models, we can reason about these objects, manipulate them in our head, and imagine possible counterfactual futures for them. Intelligence is about squeezing out the relevant predictive information form the correlated soup pixel-level information that hits our senses and representing that information in a useful manner that facilitates mental manipulation.

But the objects that we are familiar with in our everyday lives are not really all that independent. A cat that is chasing a bird is not statistically independent of it. And so, people also made attempts to define disentangling in terms of (subspaces of variables) that exhibit certain simple transformation properties when we transform the input (a.k.a. equivariant representations), or as variables that one can independently control in order to manipulate the world around us, or as causal variables that are activating certain independent mechanisms that describe the world, and so on.

The simplest way to train a model without labels is to learn a probabilistic generative model (or density) of the input data. There are a number of techniques in the field of probabilistic generative models that focus directly on maximizing the log-

probability (or a bound on the log probability) of the data under the generative model. Besides VAEs and GANs, this book explains normalizing flows, autoregressive models, energy-based models, and the latest cool kid on the block: deep diffusion models.

One can also learn representations that are good for a broad range of subsequent prediction tasks without ever training a generative model. The idea is to design tasks for the representation to solve that do not require one to acquire annotations. For instance, when considering time varying data, one can simply predict the future, which is fortunately always there for you. Or one can invent more exotic tasks such as predicting whether a patch was to the right of the left of another patch, or whether a movie is playing forward or backward, or predicting a word in the middle of a sentence from the words around it. This type of unsupervised learning is often called self-supervised learning, although I should admit that also this term seems to be used in different ways by different people.

Many approaches can indeed be understood in this "auxiliary tasks" view of unsupervised learning, including some probabilistic generative models. For instance, a Variational Autoencoder (VAE) can be understood as predicting its own input back by first pushing the information through an information bottleneck. A GAN can be understood as predicting whether a presented input is a real image (datapoint) or a fake (self-generated) one. Noise contrastive estimation can be seen as predicting in latent space whether the embedding of an input patch was close or far in space and/or time.

This book discusses the latest advances in deep probabilistic generative models. And it does so in a very accessible way. What makes this book special is that, like the child who is building a tower of bricks to understand the laws of physics, the student who uses this book can learn about deep probabilistic generative models by playing with code. And it really helps that the author has earned his spurs by having published extensively in this field. It is a great to tool to teach this topic in the classroom.

What will the future of our field bring? It seems obvious that progress toward AGI will heavily rely on unsupervised learning. It's interesting to see that the scientific community seems to be divided into two camps: the "scaling camp" believes that we achieve AGI by scaling our current technology to ever larger models trained with more data and more compute power. Intelligence will automatically emerge from this scaling. The other camp believes we need new theory and new ideas to makes further progress, such as the manipulation of discrete symbols (a.k.a. reasoning), causality, and the explicit incorporation of common-sense knowledge.

And then there is of course the increasingly important and urgent discussion of how humans will interact with these models: can they still understand what is happening under the hood or should we simply give up on interpretability? How will our lives change by models that understand us better than we do, and where humans who follow the recommendations of algorithms are more successful than those who resist? Or what information can we still trust if deepfakes become so realistic that we cannot distinguish them anymore from the real thing? Will democracy still be able function under this barrage of fake news? One thing is certain, this field is one of

the hottest in town, and this book is an excellent introduction to start engaging with it. But everyone should be keenly aware that mastering this technology comes with new responsibilities toward society. Let's progress the field with caution.

October 30, 2021 Max Welling

# Preface

We live in a world where Artificial Intelligence (AI) has become a widely used term: There are movies about AI, journalists writing about AI, CEOs talking about AI. Most importantly, there is AI in our daily lives, turning our phones, TVs, fridges, or vacuum cleaners into smartphones, smart TVs, smart fridges, and vacuum robots. We use AI; however, we still do not fully understand what "AI" is and how to formulate it, even though AI was established as a separate field in the 1950s. Since then, many researchers have pursued the holy grail of creating an artificial intelligence system that is capable of mimicking, understanding, and aiding humans through processing data and knowledge. In many cases, we have succeeded in outperforming human beings on particular tasks in terms of speed and accuracy! Current AI methods do not necessarily imitate human processing (neither biologically nor cognitively) but rather are aimed at making quick and accurate decisions like navigating in cleaning a room or enhancing the quality of a displayed movie. In such tasks, *probability theory* is key since limited or poor quality of data or intrinsic behavior of a system forces us to quantify uncertainty. Moreover, *deep learning* has become a leading learning paradigm that allows learning hierarchical data representations. It draws its motivation from biological neural networks; however, the correspondence between deep learning and biological neurons is rather far-fetched. Nevertheless, deep learning has brought AI to the next level, achieving state-of-the-art performance in many decision-making tasks. The next step seems to be a combination of these two paradigms, probability theory and deep learning, to obtain powerful AI systems that are able to quantify their uncertainties about the environments they operate in.

*This new edition*. Since the publication of the previous edition of this book in February 2022, the world of AI, and Generative AI (GenAI) in particular, has changed dramatically. Nowadays, every company wants GenAI in their portfolio, and the need for new models, especially in production, is enormous. HugginFace, one of the leading platforms of open-sourced models, hosts over 600k models (Accessed on May 2, 2024) while they hosted less than 70k models in August 2022. This huge jump perfectly shows the needs of the market. Moreover, we see two leading trends in GenAI: Score-based Generative Models, and Large Language Models. Since these two topics were not covered in the first edition, we close this gap by adding new

chapters. Moreover, multiple errors were spotted by my curious readers, and they were corrected. With a high probability, there are new mistakes, but as one of my professors used to say: No text, no error.

*What is this book about then?* This book tackles the problem of formulating AI systems by combining probabilistic modeling and deep learning. Moreover, it goes beyond the typical predictive modeling and brings together supervised learning and unsupervised learning. The resulting paradigm, called *deep generative modeling*, utilizes the generative perspective on perceiving the surrounding world. It assumes that each phenomenon is driven by an underlying generative process that defines a joint distribution over random variables and their stochastic interactions, i.e., how events occur and in what order. The adjective "deep" comes from the fact that the distribution is parameterized using deep neural networks. There are two distinct traits of deep generative modeling. First, the application of deep neural networks allows rich and flexible parameterization of distributions. Second, the principled manner of modeling stochastic dependencies using probability theory ensures rigorous formulation and prevents potential flaws in reasoning. Moreover, probability theory provides a unified framework where the likelihood function plays a crucial role in quantifying uncertainty and defining objective functions.

*Who is this book for then?* The book is designed to appeal to curious students, engineers, and researchers with a modest mathematical background in undergraduate calculus, linear algebra, probability theory, and the basics of machine learning, deep learning, and programming in Python and PyTorch (or other deep learning libraries). It should appeal to students and researchers from a variety of backgrounds, including computer science, engineering, data science, physics, and bioinformatics who wish to get familiar with deep generative modeling. In order to engage with a reader, the book introduces fundamental concepts with specific examples and code snippets. The full code accompanying the book is available online at:

https://github.com/jmtomczak/intro_dgm

The ultimate aim of the book is to outline the most important techniques in deep generative modeling and, eventually, enable readers to formulate new models and implement them.

*The structure of the book.* This edition of the book contains updated content. The first version of the book consisted of eight chapters that can be read separately and in (almost) any order. Chapter 1 introduces the topic and highlights important classes of deep generative models and general concepts. Chapters 3, 4, and 5 discuss modeling of *marginal* distributions while Chaps. 6 and 7 outline the material on modeling of *joint* distributions. Chapter 8 presents a class of latent variable models that are not learned through the likelihood-based objective. Chapter 10 indicates how deep generative modeling could be used in the fast-growing field of neural compression. This new edition contains a new section on transformers (Sects. 3.3 and 5.5.3.5), and new chapters: Chaps. 2, 9 on score-based generative models, and Chap. 11 on Large Language Models and Generative AI Systems. All chapters are accompanied by code snippets to help to understand how the presented methods could be implemented. The references are generally and to indicate the original source of the presented

material and provide further reading. Deep generating modeling is a broad field of study and including all fantastic ideas is nearly impossible. Therefore, I would like to apologize for missing any paper. If anyone feels left out, it was not intentional from my side.

In the end, I would like to thank my wife, Ewelina, for her help and presence which gave me the strength to carry on with writing this book. I am also grateful to my parents for always supporting me, and my brother who spent a lot of time checking the first version of the book and the code.

Eindhoven, The Netherlands Jakub M. Tomczak
May 2, 2024

# Acknowledgments

This book, like many other books, would not have been possible without the contribution and help of many people. During my career, I was extremely privileged and lucky to work on deep generative modeling with an amazing set of people whom I would like to thank here (in alphabetical order): Erik Bekkers, Rianne van den Berg, Francesco Paolo Casale, Taco Cohen, Tim Davidson, Nicola De Cao, Kamil Deja, Babak Esmaeili, Luka Falorsi, Eliseo Ferrante, Patrick Forré, Ioannis Gatopoulos, Efstratios Gavves, Adam Gonczarek, Amirhossein Habibian, Leonard Hasenclever, Emiel Hoogeboom, Maximilian Ilse, Adam Izdebski, Thomas Kipf, Anna Kuzina, Christos Louizos, Yura Perugachi-Diaz, Ties van Rozendaal, Victor Satorras, Ewa Szczurek, Jerzy Świątek, Tomasz Trzciński, Sharvaree Vadgama, Max Welling, Szymon Zaręba, and Maciej Zięba.

I would like to thank other colleagues with whom I worked on AI and had plenty of fascinating discussions (in alphabetical order): Davide Abati, Tameem Adel, Ilze Auzina, Babak Ehteshami Bejnordi, Erik Bekkers, Sandjai Bhulai, Tijmen Blankevoort, Johannes Brandstetter, Matteo De Carlo, Haotian Chen, Fuda van Diggelen, A.E. Eiben, Ali El Hassouni, Jacintha Ellers, Jan Engelmann, Jes Frellsen, Arkadiusz Gertych, Russ Greiner, Albert Gu, Frank van Harmelen, Mark Hoogendoorn, Emile van Krieken, David Knigge, Gongjin Lan, Falko Lavitt, Romain Lepert, Jie Luo, ChangYong Oh, Piotr Miłoś, Karine Miras, Alessandro Palma, Siamak Ravanbakhsh, Diederik Roijers, David W. Romero, Florian Shkurti, Jan-Jakob Sonke, Aart Stuurman, Annette ten Teije, Thiviyan Thanapalasingam, Fabian Theis, Auke Wiggers, Michał Zając, and Alessandro Zonta.

I am especially thankful to my brother, Kasper, who patiently read all the sections and ran and checked every single line of code in the first edition of this book. You can't even imagine my gratitude for that!

I would like to thank my wife, Ewelina, for supporting me all the time and giving me the strength to finish the first and the second versions of the book. Without her help and understanding, it would be nearly impossible to accomplish this project. I would like to also express my gratitude to my parents, Elżbieta and Ryszard, for their support at different stages of my life because without them I would never be who I am now.

# Contents

# Chapter 1
# Why Deep Generative Modeling?

## 1.1 AI Is Not Only About Decision Making

Before we start thinking about (deep) generative modeling, let us consider a simple example. Imagine we have trained a deep neural network that classifies images ($\mathbf{x} \in \mathbb{Z}^D$) of animals ($y \in \mathcal{Y}$, and $\mathcal{Y} = \{cat, dog, horse\}$). Further, let us assume that this neural network is trained really well so that it always classifies a proper class with a high probability $p(y|\mathbf{x})$. So far so good, right? The problem could occur though. As pointed out in [1], adding noise to images could result in completely false classification. An example of such a situation is presented in Fig. 1.1 where adding noise could shift predicted probabilities of labels; however, the image is barely changed (at least to us, human beings).

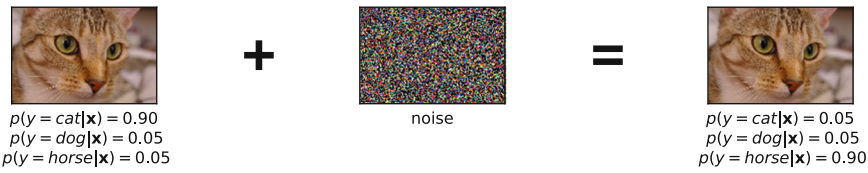



**Fig. 1.1** An example of adding noise to an almost perfectly classified image that results in a shift of predicted label.

This example indicates that neural networks that are used to parameterize the conditional distribution $p(y|\mathbf{x})$ seem to lack a semantic understanding of images. Further, we even hypothesize that learning *discriminative models* is not enough for proper decision-making and creating AI. A machine learning system cannot rely on learning how to make a decision without *understanding* the reality and being able to express *uncertainty* about the surrounding world. How can we trust such a system if even a small amount of noise could change its internal beliefs and also shift its certainty from one decision to the other? How can we communicate with such a



J. M. Tomczak, *Deep Generative Modeling*,
https://doi.org/10.1007/978-3-031-64087-2_1

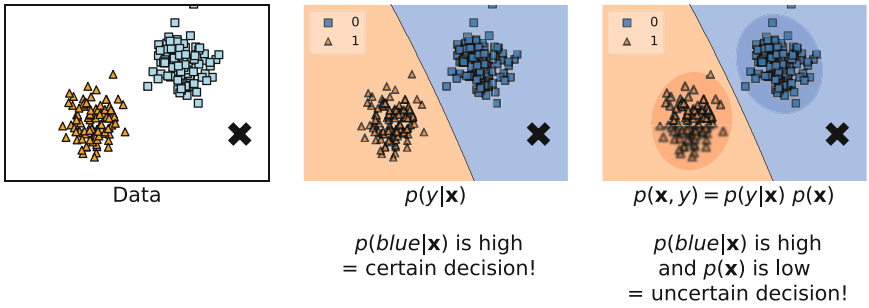



**Fig. 1.2** And example of data (*left*) and two approaches to decision-making: (*middle*) a discriminative approach, (*right*) a generative approach.

system if it is unable to properly express its opinion about whether its surroundings are new or not?

To motivate the importance of concepts like *uncertainty* and *understanding* in decision-making, let us consider a system that classifies objects, but this time into two classes: orange and blue. We assume we have some two-dimensional data (Fig. 1.2, left) and a new datapoint to be classified (a black cross in Fig. 1.2). We can make decisions using two approaches. First, a classifier could be formulated explicitly by modeling the conditional distribution $p(y|\mathbf{x})$ (Fig. 1.2, middle). Second, we can consider a joint distribution $p(\mathbf{x}, y)$ that could be further decomposed as $p(\mathbf{x}, y) = p(y|\mathbf{x})\ p(\mathbf{x})$ (Fig. 1.2, right).

After training a model using the discriminative approach, namely, the conditional distribution $p(y|\mathbf{x})$, we obtain a clear decision boundary. Then, we see that the black cross is farther away from the orange region; thus, the classifier assigns a higher probability to the blue label. As a result, the classifier is certain about the decision!

On the other hand, if we additionally fit a distribution $p(\mathbf{x})$, we observe that the black cross is not only farther away from the decision boundary, but also it is distant from the region where the blue datapoints lie. In other words, the black point is far away from the region of high probability mass. As a result, the (marginal) probability of the black cross, $p(\mathbf{x} = black\ cross)$ is low, and the joint distribution $p(\mathbf{x} = black\ cross, y = blue)$ will be low as well, and, thus, the decision is uncertain!

This simple example clearly indicates that if we want to build AI systems that make reliable decisions and can communicate with us, human beings, they must *understand* the environment first. For this purpose, they cannot simply learn how to make decisions, but they should be able to quantify their beliefs about their surroundings using the language of probability [2, 3]. In order to do that, we claim that estimating the distribution over objects, $p(\mathbf{x})$, is **crucial**.

From the *generative* perspective, knowing the distribution $p(\mathbf{x})$ is essential because:

- it could be used to assess whether a given object has been observed in the past or not;
- it could help to properly weigh the decision;

- it could be used to assess uncertainty about the environment;
- it could be used to actively learn by interacting with the environment (e.g., by asking for labeling objects with low $p(\mathbf{x})$);
- and, eventually, it could be used to generate (synthesize) new objects.

Typically, in the literature of deep learning, *generative* models are treated as generators of new data. However, here we try to convey a new perspective where having $p(\mathbf{x})$ has much broader applicability, and this could be essential for building successful AI systems. Lastly, we would like to also make an obvious connection to *generative modeling* in machine learning, where formulating a proper *generative process* is crucial for understanding the phenomena of interest [3, 4]. However, in many cases, it is easier to focus on the other factorization, namely, $p(\mathbf{x}, y) = p(\mathbf{x}|y)\, p(y)$. We claim that considering $p(\mathbf{x}, y) = p(y|\mathbf{x})\, p(\mathbf{x})$ has clear advantages as mentioned before.

## 1.2 Where Can We Use (Deep) Generative Modeling?

With the development of neural networks and the increased computational power, deep generative modeling becomes one of the leading directions in AI. Its applications vary from typical modalities considered in machine learning, i.e., text analysis (e.g., [5]), image analysis (e.g., [6]), and audio analysis (e.g., [7]), to problems in active learning (e.g., [8]), reinforcement learning (e.g., [9]), graph analysis (e.g., [10]), and medical imaging (e.g., [11]). In Fig. 1.3, we present graphically potential applications of deep generative modeling.

In some applications, it is indeed important to generate (synthesize) objects or modify features of objects to create new ones (e.g., an app turns a young person

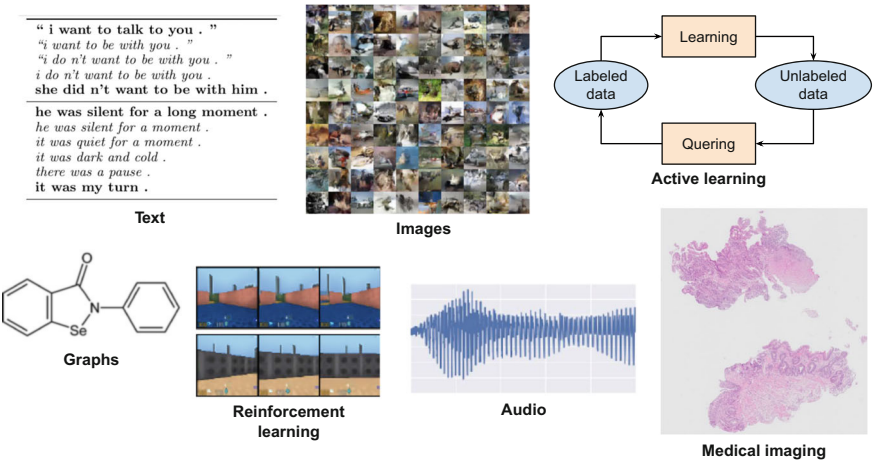



**Fig. 1.3** Various potential applications of deep generative modeling.

into an old one). However, in others like active learning, it is important to ask for *uncertain* objects, i.e., objects with low $p(\mathbf{x})$ that should be labeled by an oracle. In reinforcement learning, on the other hand, generating the next most likely situation (states) is crucial for taking action by an agent. For medical applications, explaining a decision, e.g., in terms of the probability of the label **and** the object, is definitely more informative to a human doctor than simply assisting with a diagnosis label. If an AI system would be able to indicate how certain it is, and also quantify whether the object is suspicious (i.e., low $p(\mathbf{x})$) or not, then it might be used as an independent specialist that outlines its own opinion.

These examples clearly indicate that many fields, if not all, could highly benefit from (deep) generative modeling. Obviously, there are many mechanisms that AI systems should be equipped with. However, we claim that the generative modeling capability is definitely one of the most important ones, as outlined in the abovementioned cases.

## 1.3 How to Formulate (Deep) Generative Modeling?

At this point, after highlighting the importance and wide applicability of (deep) generative modeling, we should ask ourselves how to formulate (deep) generative models. In other words, how to express $p(\mathbf{x})$ that we mentioned already multiple times.

We can divide (deep) generative modeling into four main groups (see Fig. 1.4):

- autoregressive generative models (ARM);
- flow-based models;
- latent variable models;
- energy-based models.

We use *deep* in brackets because most of what we have discussed so far could be modeled without using neural networks. However, neural networks are flexible and powerful; therefore, they are widely used to parameterize generative models. From now on, we focus entirely on deep generative models.

As a side note, please treat this taxonomy as a guideline that helps us to navigate through this book, not something written in stone. Personally, I am not a big fan of spending too much time categorizing and labeling science, because it very often results in antagonizing and gatekeeping. Anyway, there is also a group of models based on the score matching principle [12–14] that do not necessarily fit our simple taxonomy. However, as pointed out in [14], these models share a lot of similarities with latent variable models (if we treat consecutive steps of a stochastic process as latent variables), and, thus, we treat them as such.


**Fig. 1.4** A taxonomy of deep generative models.

### 1.3.1 Autoregressive Models

The first group of deep generative models utilizes the idea of **autoregressive modeling** (ARM). In other words, the distribution over $\mathbf{x}$ is represented in an autoregressive manner:

$$p(\mathbf{x}) = p(x_0) \prod_{i=1}^{D} p(x_i|\mathbf{x}_{<i}), \tag{1.1}$$

where $\mathbf{x}_{<i}$ denotes all $\mathbf{x}$'s up to the index $i$.

Modeling all conditional distributions $p(x_i|\mathbf{x}_{<i})$ would be computationally inefficient. However, we can take advantage of *causal convolutions* as presented in [7] for audio and in [15, 16] for images. We will discuss ARMs more in-depth in Chap. 3.

### 1.3.2 Flow-Based Models

The change of variables formula provides a principled manner of expressing the density of a random variable by transforming it with an invertible transformation $f$ [17]:

$$p(\mathbf{x}) = p\big(\mathbf{z} = f(\mathbf{x})\big)|\mathbf{J}_{f(\mathbf{x})}|, \tag{1.2}$$

where $\mathbf{J}_{f(\mathbf{x})}$ denotes the Jacobian matrix.

We can parameterize $f$ using deep neural networks, however, it cannot be any arbitrary neural networks, because we must be able to calculate the Jacobian matrix. First ideas of using the change of variable formulate focused on linear, volume-preserving transformations that yield $|\mathbf{J}_{f(\mathbf{x})}| = 1$ [18, 19]. Further attempts utilized theorems on matrix determinants that resulted in specific nonlinear transformations, namely, planar flows [20], and Sylvester flows [21, 22]. A different approach focuses on formulating invertible transformations for which the Jacobian determinant could be calculated easily like for coupling layers in RealNVP [23]. Recently, arbitrary neural networks are constrained in such a way they are invertible, and the Jacobian determinant is approximated [24–26].

In the case of the discrete distributions (e.g., integers), for the probability mass functions, there is no change of volume; therefore, the change of variable formulate takes the following form:

$$p(\mathbf{x}) = p\big(\mathbf{z} = f(\mathbf{x})\big). \tag{1.3}$$

Integer discrete flows propose to use affine coupling layers with rounding operators to ensure the integer-valued output [27]. A generalization of the affine coupling layer was further investigated in [28].

All generative models that take advantage of the change of variables formula are referred to as **flow-based models** or *flows* for short. We will discuss flows in Chap. 4.

### 1.3.3 Latent Variable Models

The idea behind **latent variable models** is to assume a lower-dimensional latent space and the following generative process:

$$\begin{aligned}\mathbf{z} &\sim p(\mathbf{z})\\ \mathbf{x} &\sim p(\mathbf{x}|\mathbf{z}).\end{aligned}$$

In other words, the latent variables correspond to hidden factors in data, and the conditional distribution $p(\mathbf{x}|\mathbf{z})$ could be treated as a *generator*.

The most widely known latent variable model is the **probabilistic Principal Component Analysis** (pPCA) [29] where $p(\mathbf{z})$ and $p(\mathbf{x}|\mathbf{z})$ are Gaussian distributions, and the dependency between $\mathbf{z}$ and $\mathbf{x}$ is linear.

A nonlinear extension of the pPCA with arbitrary distributions is the **Variational Auto-Encoder** (VAE) framework [30, 31]. To make the inference tractable, the variational inference is utilized to approximate the posterior $p(\mathbf{z}|\mathbf{x})$, and neural networks are used to parameterize the distributions. Since the publication of the seminal papers by [30, 31], there were multiple extensions of this framework, including working on more powerful variational posteriors [19, 21, 22, 32], priors [33, 34], and decoders [35]. Interesting directions include considering different topologies of the latent space, e.g., the hyperspherical latent space [36]. In VAEs and the pPCA, all distributions must be defined upfront; therefore, they are called *prescribed models*. We will pay special attention to this group of deep generative models in Chap. 5.

So far, ARMs, flows, the pPCA, and VAEs are probabilistic models with the objective function being the *log-likelihood function* that is closely related to using the Kullback-Leibler divergence between the data distribution and the model distribution. A different approach utilizes an *adversarial loss* in which a discriminator $D(\cdot)$ determines a difference between real data and synthetic data provided by a generator in the implicit form, namely, $p(\mathbf{x}|\mathbf{z}) = \delta\big(\mathbf{x} - G(\mathbf{z})\big)$, where $\delta(\cdot)$ is the Dirac delta. This group of models is called *implicit models*, and Generative Adversarial Networks (GANs) [6] become one of the first successful deep generative models for synthesizing realistic-looking objects (e.g., images). See Chap. 8 for more details.

### 1.3.4 Energy-Based Models

Physics provides an interesting perspective on defining a group of generative models through defining an *energy function*, $E(\mathbf{x})$, and, eventually, the Boltzmann distribution:

$$p(\mathbf{x}) = \frac{\exp\{-E(\mathbf{x})\}}{Z}, \tag{1.4}$$

where $Z = \sum_{\mathbf{x}} \exp\{-E(\mathbf{x})\}$ is the partition function.

In other words, the distribution is defined by the exponentiated energy function that is further normalized to obtain values between 0 and 1 (i.e., probabilities). There is much more to that if we think about physics, but we do not require delving into that. I refer to [37] as a great starting point for that.

Models defined by an energy function are referred to as *energy-based models* (EBMs) [38]. The main idea behind EBMs is to formulate the energy function and calculate (or rather approximate) the partition function. The largest group of EBMs consists of *Boltzmann Machines* that entangle $\mathbf{x}$'s through a bilinear form, i.e., $E(\mathbf{x}) = \mathbf{x}^\top \mathbf{W}\mathbf{x}$ [39, 40]. Introducing latent variables and taking $E(\mathbf{x}, \mathbf{z}) = \mathbf{x}^\top \mathbf{W}\mathbf{z}$ result in *Restricted Boltzmann Machines* [41]. The idea of Boltzmann machines

could be further extended to the joint distribution over $\mathbf{x}$ and $y$ as it is done, e.g., in classification Restricted Boltzmann Machines [42]. Recently, it has been shown that an arbitrary neural network could be used to define the joint distribution [43]. We will discuss how this could be accomplished in Chap. 7.

### 1.3.5 Score-Based Generative Models

Instead of matching distributions, we can match a *score function*, $\nabla_{\mathbf{x}} \ln p(\mathbf{x})$, and its model, $s_\theta(\mathbf{x})$, using the second norm (a.k.a. the mean squared error loss). However, since we do not have access to the *true* distribution, we can use a noisy version of the empirical distribution by adding small Gaussian noise, $\tilde{\mathbf{x}}_n = \mathbf{x}_n + \sigma \cdot \epsilon$, that yields:

$$q_{data}(\tilde{\mathbf{x}}_n) = \frac{1}{N} \sum_{n=1}^{N} \mathcal{N}(\tilde{\mathbf{x}}_n | \mathbf{x}_n, \sigma^2). \tag{1.5}$$

Since for Gaussian noise, the score function is analytically tractable, i.e., $\nabla_{\tilde{\mathbf{x}}} \ln \mathcal{N}(\tilde{\mathbf{x}}_n | \mathbf{x}_n, \sigma^2) = -\frac{1}{\sigma}\epsilon$, the final objective is the following:

$$\mathcal{L}(\theta) = \frac{1}{2N} \sum_{n=1}^{N} \mathbb{E}_{\mathcal{N}(\epsilon|0,\mathbf{I})} \left[ \| s_\theta(\tilde{\mathbf{x}}) + \frac{1}{\sigma}\epsilon \|^2 \right]. \tag{1.6}$$

Optimizing this objective leads to a **score model**, $s_\theta(\tilde{\mathbf{x}})$, and the approach is called **score matching** [13, 14]. Sampling from the model requires running an auxiliary procedure, e.g., Langevin dynamics [44].

The idea behind score matching has been further used to train generative models formulated as stochastic differential equations, a continuous generalization of diffusion-based models. Then, sampling results in applying numerical methods for solving differential equations like backward Euler's method. A similar approach to score matching with differential equations is called **flow matching** [45]. We will discuss all three frameworks in Chap. 9.

### 1.3.6 Overview

In Table 1.1, we compared all four groups of models (with a distinction between implicit latent variable models and prescribed latent variable models) using arbitrary criteria like:

- whether training is typically stable,
- whether it is possible to calculate the likelihood function,
- whether one can use a model for lossy or lossless compression;
- whether a model could be used for representation learning.

**Table 1.1** A comparison of deep generative models.

| Generative models | Training | Likelihood | Sampling | Compression | Representation |
|---|---|---|---|---|---|
| Autoregressive models | Stable | Exact | Slow | Lossless | No |
| Flow-based models | Stable | Exact | Fast/slow | Lossless | Yes |
| Implicit models | Unstable | No | Fast | No | No |
| Prescribed models | Stable | Approximate | Fast | Lossy | Yes |
| Energy-based models | Stable | Unnormalized | Slow | Rather not | Yes |
| Score-based models | Stable | Approximate | Fast/slow | No | No |

All likelihood-based models (i.e., ARMs, flows, EBMs, prescribed models like VAEs, and score-based models) can be trained in a stable manner, while implicit models like GANs suffer from instabilities. In the case of nonlinear prescribed models like VAEs, we must remember that the likelihood function cannot be exactly calculated, and only a lower bound can be provided. Similarly, EBMs require calculating the partition function which is an analytically intractable problem. As a result, we can get the unnormalized probability or an approximation at best. ARMs constitute one of the best likelihood-based models; however, their sampling process is extremely slow due to the autoregressive manner of generating new content. EBMs require running a Monte Carlo method to receive a sample. Since we operate on high-dimensional objects, this is a great obstacle to using EBMs widely in practice. All other approaches are relatively fast. In the case of compression, VAEs are models that allow us to use a bottleneck (the latent space). On the other hand, ARMs and flows could be used for lossless compression, since they are density estimators and provide the exact likelihood value. Implicit models cannot be directly used for compression; however, recent works use GANs to improve image compression [46]. Flows, prescribed models, and EBMs (if they use latents) could be used for representation learning, namely, learning a set of random variables that summarize data in some way and/or disentangle factors in data. The question about what is a good representation is a different story, and we refer a curious reader to literature, e.g., [47]. Nowadays, state-of-the-art performance is achieved by score-based generative models, but they do not provide representation or cannot be used for compression purposes.

## 1.4 Purpose and Content of This Book

This book is intended as an introduction to the field of deep generative modeling. Its goal is to convince you, dear reader, of the philosophy of generative modeling and show you its beauty! Deep generative modeling is an interesting hybrid that combines probability theory, statistics, probabilistic machine learning, and deep learning in a single framework. However, to be able to follow the ideas presented in this book,

it is advised to possess knowledge in algebra and calculus, probability theory and statistics, the basics of machine learning and deep learning, and programming with Python. Knowing PyTorch[1] is highly recommended, since all code snippets are written in PyTorch. However, knowing other deep learning frameworks like Keras, Tensorflow, or JAX should be sufficient to understand the code.

In this book, we will not review machine learning concepts or building blocks in deep learning unless it is essential to comprehend a given topic. Instead, we will delve into models and training algorithms of deep generative models. We will discuss the marginal models, such as mixture models and probabilistic circuits (Chap. 2), autoregressive models (Chap. 3), flow-based models (Chap. 4): RealNVP, Integer Discrete Flows, and residual and densenet flows, latent variable models (Chap. 5): Variational Auto-Encoders and its components, hierarchical VAEs, and diffusion-based deep generative models, or frameworks for modeling the joint distribution like hybrid modeling (Chap. 6) and energy-based models (Chap. 7). After that, we will outline GANs (Chap. 8). Next, we will delve into score-based generative models including score matching, score-based stochastic differential equations, and flow matching (Chap. 9). After that, we will present how deep generative modeling could be useful for data compression within the neural compression framework (Chap. 10). Eventually, we will discuss large language models and generative AI systems (Chap. 11). In general, the book is organized in such a way that each chapter can be followed independently from the others and in an order that suits a reader best.

So who is the target audience of this book? Well, hopefully, everybody who is interested in AI, but there are two groups that could definitely benefit from the presented content. The first target audience is university students who want to go beyond standard courses in machine learning and deep learning. The second group is research engineers who want to broaden their knowledge of AI or prefer to take the next step in their careers and learn about the next generation of AI systems. Either way, the book is intended for curious minds who want to understand AI and learn not only about theory but also how to implement the discussed material. For this purpose, each topic is associated with a general discussion and introduction that is further followed by formal formulations and a piece of code (in PyTorch). The intention of this book is to truly understand deep generative modeling that, in the humble opinion of the author of this book, is only possible if one can not only derive a model but also implement it. Therefore, this book is accompanied by the following code repository:

https://github.com/jmtomczak/intro_dgm

## References

1. Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural net-

[1] https://pytorch.org/

works. In *2nd International Conference on Learning Representations, ICLR 2014*, 2014.
2. Christopher M Bishop. Model-based machine learning. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 371(1984):20120222, 2013.
3. Zoubin Ghahramani. Probabilistic machine learning and artificial intelligence. *Nature*, 521(7553):452–459, 2015.
4. Julia A Lasserre, Christopher M Bishop, and Thomas P Minka. Principled hybrids of generative and discriminative models. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 1, pages 87–94. IEEE, 2006.
5. Samuel Bowman, Luke Vilnis, Oriol Vinyals, Andrew Dai, Rafal Jozefowicz, and Samy Bengio. Generating sentences from a continuous space. In *Proceedings of The 20th SIGNLL Conference on Computational Natural Language Learning*, pages 10–21, 2016.
6. Ian J Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *arXiv preprint arXiv:1406.2661*, 2014.
7. Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.
8. Samarth Sinha, Sayna Ebrahimi, and Trevor Darrell. Variational adversarial active learning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 5972–5981, 2019.
9. David Ha and Jürgen Schmidhuber. World models. *arXiv preprint arXiv:1803.10122*, 2018.
10. Martin Simonovsky and Nikos Komodakis. GraphVAE: Towards generation of small graphs using variational autoencoders. In *International Conference on Artificial Neural Networks*, pages 412–422. Springer, 2018.
11. Maximilian Ilse, Jakub M Tomczak, Christos Louizos, and Max Welling. DIVA: Domain invariant variational autoencoders. In *Medical Imaging with Deep Learning*, pages 322–348. PMLR, 2020.
12. Aapo Hyvärinen and Peter Dayan. Estimation of non-normalized statistical models by score matching. *Journal of Machine Learning Research*, 6(4), 2005.
13. Yang Song and Stefano Ermon. Generative modeling by estimating gradients of the data distribution. *arXiv preprint arXiv:1907.05600*, 2019.
14. Yang Song, Jascha Sohl-Dickstein, Diederik P Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. Score-based generative modeling through stochastic differential equations. In *International Conference on Learning Representations*, 2020.
15. Aaron Van Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel recurrent neural networks. In *International Conference on Machine Learning*, pages 1747–1756. PMLR, 2016.

16. Aäron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, and Koray Kavukcuoglu. Conditional image generation with pixelcnn decoders. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, pages 4797–4805, 2016.
17. Oren Rippel and Ryan Prescott Adams. High-dimensional probability estimation with deep density models. *arXiv preprint arXiv:1302.5125*, 2013.
18. Laurent Dinh, David Krueger, and Yoshua Bengio. NICE: Non-linear independent components estimation. *arXiv preprint arXiv:1410.8516*, 2014.
19. Jakub M Tomczak and Max Welling. Improving variational auto-encoders using householder flow. *arXiv preprint arXiv:1611.09630*, 2016.
20. Danilo Rezende and Shakir Mohamed. Variational inference with normalizing flows. In *International Conference on Machine Learning*, pages 1530–1538. PMLR, 2015.
21. Rianne Van Den Berg, Leonard Hasenclever, Jakub M Tomczak, and Max Welling. Sylvester normalizing flows for variational inference. In *34th Conference on Uncertainty in Artificial Intelligence 2018, UAI 2018*, pages 393–402. Association For Uncertainty in Artificial Intelligence (AUAI), 2018.
22. Emiel Hoogeboom, Victor Garcia Satorras, Jakub M Tomczak, and Max Welling. The convolution exponential and generalized sylvester flows. *arXiv preprint arXiv:2006.01910*, 2020.
23. Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. Density estimation using Real NVP. *arXiv preprint arXiv:1605.08803*, 2016.
24. Jens Behrmann, Will Grathwohl, Ricky TQ Chen, David Duvenaud, and Jörn-Henrik Jacobsen. Invertible residual networks. In *International Conference on Machine Learning*, pages 573–582. PMLR, 2019.
25. Ricky TQ Chen, Jens Behrmann, David Duvenaud, and Jörn-Henrik Jacobsen. Residual flows for invertible generative modeling. *arXiv preprint arXiv:1906.02735*, 2019.
26. Yura Perugachi-Diaz, Jakub M Tomczak, and Sandjai Bhulai. Invertible densenets with concatenated lipswish. *Advances in Neural Information Processing Systems*, 2021.
27. Emiel Hoogeboom, Jorn WT Peters, Rianne van den Berg, and Max Welling. Integer discrete flows and lossless compression. *arXiv preprint arXiv:1905.07376*, 2019.
28. Jakub M Tomczak. General invertible transformations for flow-based generative modeling. *INNF+*, 2021.
29. Michael E Tipping and Christopher M Bishop. Probabilistic principal component analysis. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 61(3):611–622, 1999.
30. Diederik P Kingma and Max Welling. Auto-encoding variational Bayes. *arXiv preprint arXiv:1312.6114*, 2013.
31. Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic backpropagation and approximate inference in deep generative models. In *International conference on machine learning*, pages 1278–1286. PMLR, 2014.

32. Durk P Kingma, Tim Salimans, Rafal Jozefowicz, Xi Chen, Ilya Sutskever, and Max Welling. Improved variational inference with inverse autoregressive flow. *Advances in Neural Information Processing Systems*, 29:4743–4751, 2016.
33. Xi Chen, Diederik P Kingma, Tim Salimans, Yan Duan, Prafulla Dhariwal, John Schulman, Ilya Sutskever, and Pieter Abbeel. Variational lossy autoencoder. *arXiv preprint arXiv:1611.02731*, 2016.
34. Jakub Tomczak and Max Welling. VAE with a VampPrior. In *International Conference on Artificial Intelligence and Statistics*, pages 1214–1223. PMLR, 2018.
35. Ishaan Gulrajani, Kundan Kumar, Faruk Ahmed, Adrien Ali Taiga, Francesco Visin, David Vazquez, and Aaron Courville. PixelVAE: A latent variable model for natural images. *arXiv preprint arXiv:1611.05013*, 2016.
36. Tim R Davidson, Luca Falorsi, Nicola De Cao, Thomas Kipf, and Jakub M Tomczak. Hyperspherical variational auto-encoders. In *34th Conference on Uncertainty in Artificial Intelligence 2018, UAI 2018*, pages 856–865. Association For Uncertainty in Artificial Intelligence (AUAI), 2018.
37. Edwin T Jaynes. *Probability theory: The logic of science*. Cambridge university press, 2003.
38. Yann LeCun, Sumit Chopra, Raia Hadsell, M Ranzato, and F Huang. A tutorial on energy-based learning. *Predicting structured data*, 1(0), 2006.
39. David H Ackley, Geoffrey E Hinton, and Terrence J Sejnowski. A learning algorithm for Boltzmann machines. *Cognitive science*, 9(1):147–169, 1985.
40. Geoffrey E Hinton, Terrence J Sejnowski, et al. Learning and relearning in Boltzmann machines. *Parallel distributed processing: Explorations in the microstructure of cognition*, 1(282–317):2, 1986.
41. Geoffrey E Hinton. A practical guide to training restricted Boltzmann machines. In *Neural networks: Tricks of the trade*, pages 599–619. Springer, 2012.
42. Hugo Larochelle and Yoshua Bengio. Classification using discriminative restricted Boltzmann machines. In *Proceedings of the 25th international conference on Machine learning*, pages 536–543, 2008.
43. Will Grathwohl, Kuan-Chieh Wang, Joern-Henrik Jacobsen, David Duvenaud, Mohammad Norouzi, and Kevin Swersky. Your classifier is secretly an energy-based model and you should treat it like one. In *International Conference on Learning Representations*, 2019.
44. Max Welling and Yee W Teh. Bayesian learning via stochastic gradient Langevin dynamics. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 681–688. Citeseer, 2011.
45. Yaron Lipman, Ricky TQ Chen, Heli Ben-Hamu, Maximilian Nickel, and Matt Le. Flow matching for generative modeling. *arXiv preprint arXiv:2210.02747*, 2022.
46. Fabian Mentzer, George D Toderici, Michael Tschannen, and Eirikur Agustsson. High-fidelity generative image compression. *Advances in Neural Information Processing Systems*, 33, 2020.
47. Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

# Chapter 2
# Probabilistic Modeling: From Mixture Models to Probabilistic Circuits


## 2.1 A Probabilistic Perspective on Modeling: Random Variables, Learning, Generalization and Inference

### 2.1.1 Random Variables and Probability Distributions

Let us imagine cats. Most people like cats, and some people are crazy in love with cats. There are ginger cats, black cats, big cats, small cats, puffy cats, and furless cats. In fact, there are many different kinds of cats. However, when I say this word: "a cat", everyone has some kind of a cat in their mind. One can close eyes and *generate* a picture of a cat, either their own cat or a cat of a neighbor. Further, this *generated* cat is located somewhere, e.g., sleeping on a couch or in a garden chasing a fly, during the night or during the day, and so on. Probably we can agree at this point that there are infinitely many possible scenarios of cats in some environments.

Why to think of cats? Actually, we can think of any other object in the world. Formally, we can define a space of all objects in the world as $\omega \in \Omega$. For simplicity, we consider only physical objects that an AI system (either embodied or not) could perceive through its sensors. Without a loss of generality, I would claim that even an AI-powered app deals with physical objects through, e.g., a camera, or images taken by a user through a camera. The AI system processes the input signals through some functions that return, e.g., continuous values like analog signals, discrete values like text from a conversation, or bits like digital signals. These functions are called **random variables**. For instance, a robot with a camera can take an RGB photo (e.g., of a cat) of the height $H$ and the width $W$ that defines a random variable. Typically, a random variable is denoted by $\mathbf{x}(\omega)$ or $\mathbf{x}$ for short. In our example, for images taken by a digital camera, we get $\mathbf{x} : \Omega \rightarrow \{0, \ldots, 255\}^{3 \times H \times W}$. In general, $\mathbf{x} : \Omega \rightarrow \mathcal{E}$ where $\mathcal{E}$ is a $D$-dimensional measurable space (in the sense of measure theory), e.g., $\mathcal{E} = \mathbb{R}^D$, $\mathcal{E} = \{0, 1\}^D$.

Random variables are called *random*, because their outcomes are not necessarily predictable due to unobserved factors.



J. M. Tomczak, *Deep Generative Modeling*,
https://doi.org/10.1007/978-3-031-64087-2_2

**A side note**

If we think about *randomness*, this simple introduction could very quickly turn into a very deep philosophical discourse. If you disagree with me, my curious reader, it is ok! But I hope that we can agree with these statements provided above and move on.

In other words, if one takes their cat and takes a series of photos, the resulting pixels could take various values due to lighting, the breathing of their cat, shaking hand, etc.; in general, some factors that are neither controlled nor explicitly measured. This series of photos, taken at different times of the day, or even on different days, we call **observations**, $\mathcal{D} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\}$. There is some randomness in the process of taking photos, but still, some pixel values are highly improbable or even impossible. For instance, a cat has four legs, a single head, or its fur cannot be naturally pink, and so on. As a result, we can say that there exists a *distribution* of pixel values (values of random variables) that correspond to less likely values or more likely values. Formally, we say that there exists some *probability distribution* of a random variable $\mathbf{x}$, denoted as $p(\mathbf{x})$, such that: $\forall_{\mathbf{x}} p(\mathbf{x}) \in [0, 1]$ and (discrete case) $\sum_{\mathbf{x}} p(\mathbf{x}) = 1$, (continuous case) $\int_{\mathbf{x}} p(\mathbf{x}) \mathrm{d}\mathbf{x} = 1$. There are multiple nice properties of probability distributions, and two of them are especially useful in modeling and inference (for simplicity, for two random variables $\mathbf{x} = (x_1, x_2)$):

- the **sum rule**: $p(x_1) = \sum_{x_2} p(x_1, x_2)$,
- the **product rule**: $p(x_1, x_2) = p(x_1|x_2)p(x_2)$ or $p(x_1, x_2) = p(x_2|x_1)p(x_1)$.

The sum rule shows us how to "get rid of" some random variable in a probabilistic manner. The product rule indicates how to express a joint distribution using conditional distributions.

In the product rule, we see a *conditional distribution*, i.e., a distribution over one random variable given information about the other random variable. The conditional distribution is a distribution, namely, it sums to 1 and returns values between 0 and 1. If two random variables are independent (stochastically), e.g., $x_1$ is independent of $x_2$, then $p(x_1|x_2) = p(x_1)$. In plain words, no matter what information we provide about $x_2$, there is no change in the distribution of $x_1$. As a result, the product rule results in $p(x_1, x_2) = p(x_1)p(x_2)$.

### 2.1.2 Modeling

In general, this is useful to think of some *true* probability distribution for a given random variable. However, this could be also misleading. Again, we can turn this dry sentence into a long discussion. Therefore, from time to time, we will refer to some abstract *true* distribution if needed, but practically, we will use some *empirical distribution* defined by observations $\mathcal{D}$ instead. Formally, we define the empirical distribution as a sum of *peaks*:

$$p_{data}(\mathbf{x}) = \frac{1}{N}\sum_{n=1}^{N}\delta(\mathbf{x}-\mathbf{x}_n), \tag{2.1}$$

where $\delta(\mathbf{x}-\mathbf{x}_n)$ for $\mathbf{x} \neq \mathbf{x}_n$ equals 0 and for $\mathbf{x} = \mathbf{x}_n$ equals $+\infty$ in the continuous case (Dirac's delta), and 1 in the discrete case (Kronecker's delta).

Please remember, my curious reader, that this is mathematics; we need that to communicate easily and to express our thoughts crisply. It is not magic, even though, sometimes, it sounds complicated. If you are not familiar with these terms, no worries. You will get used to those, and, as you will see, they are quite useful.

Very often we assume that observations come from the same *true* distributions and that all observations were achieved independently. This is referred to as *identically and independently distributed* (*iid*) data. These two statements mean that: (i) all $\mathbf{x}_n \in \mathcal{D}$ follow the same distribution, and (ii) all $\mathbf{x}_n$'s are independent from each other, i.e., $p(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N) = \prod_{n=1}^{N} p(\mathbf{x}_n)$.

Alright, but why do introduce all these concepts? When does it become interesting? These are the questions that pinch your brain, my curious readers, aren't they? They are fascinating if we could formulate a model distribution (or simply a model) of the world (or the tiny piece of the world we consider), namely, $p(\mathbf{x};\theta)$. As you can notice, we introduced a new quantity $\theta$ that denotes some **parameters** of the model. But what is this model, what is its form? It depends on the character of the observed data. For instance, for real-valued data, an appropriate distribution can be a **Gaussian (normal) distribution**:

$$p(\mathbf{x};\theta) = \frac{1}{\sqrt{(2\pi)^D|\Sigma|}}\exp\left(-\frac{1}{2}(\mathbf{x}-\mu)^\top\Sigma^{-1}(\mathbf{x}-\mu)\right), \tag{2.2}$$

$$\stackrel{df}{=} \mathcal{N}(\mathbf{x}|\mu,\Sigma), \tag{2.3}$$

where $\theta = (\mu, \Sigma)$, $\mu$ denotes location parameters (a.k.a. means), and $\Sigma$ denotes scale parameters (a.k.a. the covariance matrix with variances on the diagonal).

However, if our data is discrete, we use a **Bernoulli distribution** for binary data (i.e., $x_d \in \{0, 1\}$):

$$p(\mathbf{x};\theta) = \prod_{d=1}^{D}\theta_d^{x_d}(1-\theta_d)^{1-x_d}, \tag{2.4}$$

$$\stackrel{df}{=} \text{Bern}(\mathbf{x}|\theta), \tag{2.5}$$

or a **Categorical distribution** for categorical data (i.e., discrete and unordered, $x_d \in \{0, 1, \ldots, K-1\}$):

$$p(\mathbf{x};\theta) = \prod_{d=1}^{D}\prod_{k=1}^{K}\theta_{dk}^{\mathbb{1}[x_d=k]}, \tag{2.6}$$

$$\stackrel{df}{=} \text{Cat}(\mathbf{x}|\theta), \tag{2.7}$$

where $\mathbb{1}[x = k]$ is the indicator function that returns 1 if $x$ equals $k$ and 0 - otherwise. Interestingly, $\theta$'s in a Bernoulli distribution and a categorical distribution have very specific meanings, respectively, $\theta_d = p(x_d = 1; \theta)$ and $\theta_{dk} = p(x_d = k; \theta)$.

### 2.1.3 Learning

Once we pick a distribution for our model, we can now ask the fundamental question: How to obtain the values of parameters $\theta$, i.e., how to *estimate* $\theta$? This is what AI people call **learning** or **training**. In the probabilistic setting, it is typically done by *fitting* our model to the empirical distribution by minimizing some distance metric. For instance, it can be accomplished by minimizing the Kullback-Leibler divergence with respect to the parameters $\theta$:

$$KL\left[p_{data}(\mathbf{x})||p(\mathbf{x};\theta)\right] = \sum_{\mathbf{x}} p_{data}(\mathbf{x}) \log \frac{p_{data}(\mathbf{x})}{p(\mathbf{x};\theta)} \tag{2.8}$$

$$= \sum_{\mathbf{x}} p_{data}(\mathbf{x}) \log p_{data}(\mathbf{x}) - \sum_{\mathbf{x}} p_{data}(\mathbf{x}) \log p(\mathbf{x};\theta) \tag{2.9}$$

$$= C - \sum_{\mathbf{x}} p_{data}(\mathbf{x}) \log p(\mathbf{x};\theta) \tag{2.10}$$

$$= C - \frac{1}{N} \sum_{\mathbf{x}} \sum_{n=1}^{N} \delta(\mathbf{x} - \mathbf{x}_n) \log p(\mathbf{x};\theta) \tag{2.11}$$

$$= C - \frac{1}{N} \sum_{n=1}^{N} \log p(\mathbf{x}_n;\theta) \tag{2.12}$$

where $C$ is the negative entropy of the empirical distribution, which is independent of the parameters $\theta$, thus, irrelevant for learning the model, and the last step comes from a property of the delta function.

**A side note**

Please remember, my curious reader, that the Kullback-Leibler divergence is asymmetric, thus, where we place the empirical distribution and the model matters!

For the *iid* data, the term $\sum_{n=1}^{n} \log p(\mathbf{x}_n;\theta)$ corresponds to the **log-likelihood function**, namely:

$$\log p(\mathcal{D};\theta) \overset{df}{=} \log p(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N;\theta) \tag{2.13}$$

$$\overset{iid}{=} \log \prod_{n=1}^{N} p(\mathbf{x}_n;\theta) \tag{2.14}$$

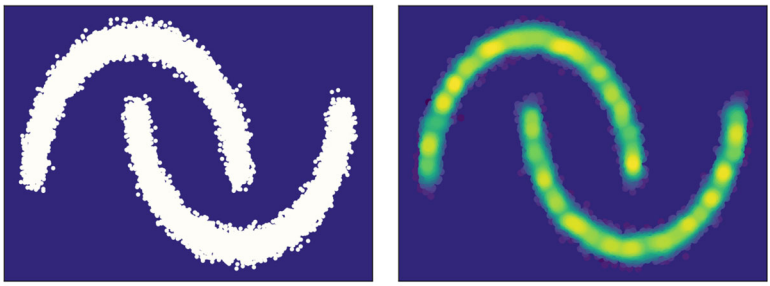
$$= \sum_{n=1}^{N} \log p(\mathbf{x}_n;\theta). \tag{2.15}$$

In the second step, we use the fact that the observations come from the same distribution and all observations are independent from each other. The likelihood function plays a special role in probabilistic modeling and serves as a starting point for many deep generative models.

### 2.1.4 Generalization

As a quote from a famous British statistician named George Box indicates, all models are wrong but some are useful. They are wrong, fine. But what does it mean that some are useful?! Let us consider the following example in Fig. 2.1. We have some observed data from a *true* distribution (here we use the so-called two-moon problem) like in Fig. 2.1a. After fitting a model to those data, we end up with a density estimator in Fig. 2.1b. As you can notice, the model distribution assigns probability mass mainly where the data lies; however, it also assigns some probability mass in regions where little to no data is observed. Is it a good feature? My answer is yes, because this is what we expect from a model to **generalize** to regions where no or little data was observed.

In machine learning, generalization is understood as a model's ability to adapt properly to new, previously unseen data, drawn from the same distribution as the one used to create the model [1]. Here, *adapting* means to sample or to assign probability/likelihood. Hence, to be able to generalize, having a probabilistic model (i.e., a model distribution) is key! Now we can see that some models are more *useful*



**A**. Observed data. **B**. A model

**Fig. 2.1** An example of a probabilistic model (**b**), i.e., a model distribution, fit to data (**a**).

than others, because they can generalize better. Please keep it in mind while thinking of various deep generative models.

### 2.1.5 Inference

After fitting a model to data, we can (finally!) use our model! The *simplest* uses of probabilistic models (either conditional or joint) are obtaining new data and calculating the likelihood function (also referred to as *evidence*), $p(\mathbf{x})$. I call these *simple*, however, these are quite powerful, if you think about it, my curious reader. But wait a second, are conditional models useful for *generating* new data? For instance, if $x_2$ is the weight of a cat and $x_1$ is the height of a cat, then one can say that sampling from $p(x_2|x_1)$ is... generating new data, namely, pairs $(x_1, x_2)$. There is a thin and blurry line between discriminative (predictive) models and generative models, especially from a probabilistic perspective. After all, we deal with probability distributions, and calling some models "discriminative" and others "generative" is not necessarily constructive. But, again, we will not delve into that discussion here; instead, we will focus on **inference**.

In the probabilistic modeling world, inference is the key to draw conclusions about stochastic dependencies and quantities. We can define inference as *answering probabilities queries*. Following [2], we can distinguish the following classes of probabilistic queries:

- (EVI) Calculating evidence $p(\mathbf{x})$.
- (MAR) Calculating marginals $p(\mathbf{x}_{-d}) = \int p(\mathbf{x})\, \mathrm{d}x_d$.
- (MAP) Calculating the most probable value, $\arg\max_{\mathbf{x}} p(\mathbf{x})$.
- Calculating advanced queries, e.g., about logical events.

Interestingly, not all classes of contemporary deep generative models can do EVI, and only a few provide MAR and/or MAP. Later, we will briefly discuss a class of models called probabilistic circuits [2] that can do EVI, MAR, and MAP. However, the rest of the deep generative models can calculate EVI at best.

## 2.2 Interlude: Probabilistic Graphical Models

As you can imagine, my curious reader, working with probabilistic models can be sometimes quite difficult, especially if the number of variables $D$ is large, and there is a specific structure among random variables, i.e., complicated stochastic dependencies. For this purpose, to represent probability distributions, **probabilistic graphical models** (PGMs) were proposed [3]. In a nutshell, a PGM is a probabilistic model for which a graph expresses the conditional dependence structure between random variables. There are **undirected graphical models** in which nodes are random variables and edges imply some sort of dependence among them, a.k.a.

Markov networks. A more *straightforward* class of PGMs is a class of **Bayesian networks** that are directed acyclic graphs with directed edges denoting conditional dependencies and random variables being nodes.

Here, we will not go into details of various PGMs, interested (and curious!) readers can check [3] as a fantastic primer. It is good to know though that for generative modeling, Bayesian networks are useful in thinking in a generative manner. Why? Because they are *acyclic* meaning that there are no cycles, i.e., a path starting and ending in the same random variable. Clearly, a cycle disallows a proper generative process (if a random variable $x_A$ depends on $x_B$ and the other way around, how we can generate $x_A$ and $x_B$?).

**A side note**

Not all probability distributions could be represented as Bayesian networks. Please be aware of that! Therefore, there are some generative models that go beyond Bayesian networks. However, to get familiar with generative processes, Bayesian networks give a good starting point for that.

**Example**

An example of a Bayesian network is presented in Fig. 2.2. A gray node denotes an observable variable, a white node is a latent variable, and arrows determine conditional dependencies. The corresponding joint distribution in the example in Fig. 2.2 is the following:

$$p(A, B, C) = p(C|A, B)\, p(A)\, p(B). \tag{2.16}$$

If we would like to calculate the marginal distribution over observable variables, we need to sum out the latent variable, namely:

$$p(A, C) = \sum_B p(A, B, C) \tag{2.17}$$

$$= \sum_B p(C|A, B)\, p(A)\, p(B). \tag{2.18}$$

## 2.3 Mixture Models

### 2.3.1 Modeling with Mixture of Gaussians (MoG)

Take a deep breath. A lot of information so far, I am aware of that, my curious reader. Take a short break if needed, and make a coffee. And buckle up, we are going to get all the information together and propose one of the simplest yet very effective generative models: a **mixture model**. Here we go!

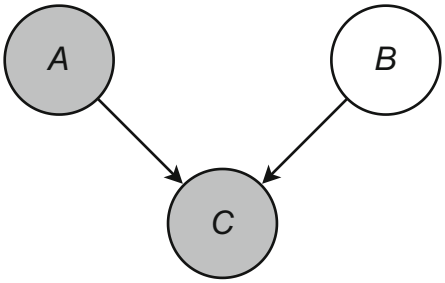



**Fig. 2.2** An example of a Bayesian network for three random variables (observable variables are gray nodes and a latent variable is a white node).

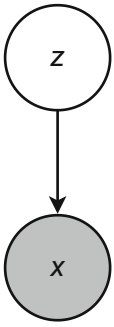



**Fig. 2.3** An example of a latent variable model.

Let us consider the following situation. We have a random variable $x$ denoting the weight of cats. There are female cats and male cats, and they differ in weight. We model the *true* distribution for female cats using a Gaussian, and another Gaussian for modeling the *true* distribution for male cats. Then someone asks us what is the *overall* distribution over weights of cats (no distinction between female and male cats). We are given some *iid* data of weights of cats but with no information about the gender of observed cats, just pure numbers of their weights. How can we approach this problem?

We can start with random variables. OK, $x$ denotes weight, and it is an observable random variable. With a small abuse of reality, we assume that $x \in \mathbb{R}$. Next, we know that there is the biological sex of cats $z \in \{0, 1\}$, where $z = 0$ for female cats and $z = 1$ for male cats. However, $z$ is an unobserved (latent) variable. Further, we know that the sex, $z$, influences the weight, $x$. The resulting probabilistic graphical model is presented in Fig. 2.3.

Let us write the joint distribution and use the product rule according to the PGM in Fig. 2.3:

$$p(x, z) = p(x|z)\, p(z). \tag{2.19}$$

So far, so good! Since $x$ is real-valued and $z$ is binary, we can use a Gaussian distribution and a Bernoulli distribution, respectively, to model them:

$$p(x, z) = \mathcal{N}(x|\mu(z), \sigma^2(z)) \, \text{Bern}(z|\omega), \tag{2.20}$$

where we indicate the dependence between $x$ and $z$ through the mean and the variance, i.e., the parameters of the Gaussian are different for different values of $z$.

Now, please remember, my curious reader, that we have some observations $\mathcal{D} = \{x_1, x_2, \ldots, x_N\}$. Since we do not observe $z$, we should marginalize it out using the sum rule, namely:

$$\begin{aligned} p(x) &= \sum_z p(x, z) && (2.21)\\ &= \sum_z \mathcal{N}(x|\mu(z), \sigma^2(z)) \, \text{Bern}(z|\omega) && (2.22)\\ &= \mathcal{N}(x|\mu(z=0), \sigma^2(z=0)) \, \text{Bern}(z=0|\omega) + \\ &\quad + \mathcal{N}(x|\mu(z=1), \sigma^2(z=1)) \, \text{Bern}(z=1|\omega) && (2.23)\\ &= (1-\omega) \, \mathcal{N}(x|\mu_0, \sigma_0^2) + \omega \, \mathcal{N}(x|\mu_1, \sigma_1^2), && (2.24) \end{aligned}$$

where we used $\mu_i \equiv \mu(z = i)$ and $\sigma_i^2 \equiv \sigma^2(z = i)$ to keep the notation uncluttered.

Let us get back to our cats and their weights (a disclaimer: all values are made up and they serve educational purposes!). An example of a mixture of two Gaussians is presented in Fig. 2.4. In Fig. 2.4a, the two mixture components are visualized separately, while in Fig. 2.4b the marginal distribution, $p(x)$, is presented. Note that the two components have different component probabilities $\omega_0$ and $\omega_1 = 1 - \omega_0$. Moreover, in Fig. 2.4b, the region where the two components overlap is summed.

In general, we define a mixture of Gaussians (MoG) with $K$ components as follows:

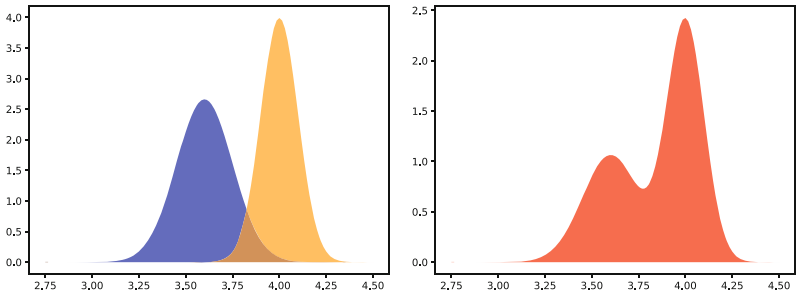



**Fig. 2.4** An example of a distribution of weights of cats: (**a**) Gaussians (separate components) for weights of female cats (blue, on the left) and male cats (orange, on the right). (**b**) The marginal distribution of weights of cats, $p(x)$.

$$p(x;\theta) = \sum_{z} \mathcal{N}(x|\mu(z), \sigma^2(z))\ \mathrm{Cat}(z|\omega) \tag{2.25}$$

$$= \sum_{k=0}^{K=1} \omega_k\ \mathcal{N}(x|\mu_k, \sigma_k^2). \tag{2.26}$$

### 2.3.2 Training: The Log-Likelihood Function

Alright, let us continue our example with two components and think of how we can train a mixture of Gaussians. First, we need to set up the stage, namely, calculate the log-likelihood function. For given data $\mathcal{D}$, the likelihood function is the following:

$$p(\mathcal{D};\mu_0,\mu_1,\sigma_0^2,\sigma_1^2,\omega) = \prod_{n=1}^{N} p(x_n;\mu,\sigma^2,\omega) \tag{2.27}$$

$$= \prod_{n=1}^{N} \Big((1-\omega)\ \mathcal{N}(x_n|\mu_0,\sigma_0^2) + \omega\ \mathcal{N}(x_n|\mu_1,\sigma_1^2)\Big) \tag{2.28}$$

and the log-likelihood function is the following (we take the logarithm to work with sums instead of products; the logarithm does not change the position of the optimum):

$$\log p(\mathcal{D};\mu_0,\mu_1,\sigma_0^2,\sigma_1^2,\omega) = \sum_{n=1}^{N} \log\Big((1-\omega)\ \mathcal{N}(x_n|\mu_0,\sigma_0^2) + \omega\ \mathcal{N}(x_n|\mu_1,\sigma_1^2)\Big). \tag{2.29}$$

A few comments here:

1. The parameters of our model are $\theta = (\mu_0, \mu_1, \sigma_0^2, \sigma_1^2, \omega)$.
2. The (log-)likelihood function has a nice interpretation: How *likely* are the observed data for given values of parameters $\theta$?
3. The log-likelihood function can be taken as the objective/training function and optimized with respect to the parameters $\theta$ to find the most *likely* parameters that generated the observed data.

Further, we (yes, you and me, my curious reader!) notice that it would be easier to deal with $\log \mathcal{N}(x|\mu,\sigma^2)$ instead of $\mathcal{N}(x|\mu,\sigma^2)$ because of the cumbersome exp in Gaussians. Taking the log gives:

$$\log \mathcal{N}(x|\mu,\sigma^2) = \log\left(\frac{1}{\sigma\sqrt{2\pi}} \exp\left\{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right\}\right) \tag{2.30}$$

$$= -\frac{1}{2}\log 2\pi - \frac{1}{2}\log\sigma^2 - \frac{1}{2\sigma^2}(x-\mu)^2. \tag{2.31}$$

Now, if only we could get the logarithm inside the sum, it would be easier! But we cannot... However, do we need to do that? We can use the property of exp and log: $\exp(\log(x)) = x$. Let us use that for the log-likelihood function:

$$\log p(\mathcal{D}; \mu_0, \mu_1, \sigma_0^2, \sigma_1^2, \omega) = \sum_{n=1}^{N} \log \left( \exp \left( \log \left( (1-\omega)\, \mathcal{N}(x_n|\mu_0, \sigma_0^2) \right) \right) + \right.$$
$$\left. + \exp \left( \log \left( \omega\, \mathcal{N}(x_n|\mu_1, \sigma_1^2) \right) \right) \right) \tag{2.32}$$
$$= \sum_{n=1}^{N} \log \left( \exp \left( \log(1-\omega) + \log \mathcal{N}(x_n|\mu_0, \sigma_0^2) \right) + \right.$$
$$\left. + \exp \left( \log(\omega) + \log \mathcal{N}(x_n|\mu_1, \sigma_1^2) \right) \right) \tag{2.33}$$
$$= \sum_{n=1}^{N} \log \left( \sum_{i=0}^{1} \exp \left( \log(\omega_i) + \log \mathcal{N}(x_n|\mu_i, \sigma_i^2) \right) \right) \tag{2.34}$$
$$= \sum_{n=1}^{N} \log \left( \sum_{i=0}^{1} \exp(a_i) \right). \tag{2.35}$$

A few comments:

1. As you noticed, my curious reader, we introduced $a_i \stackrel{df}{=} \log(\omega_i) + \log \mathcal{N}(x_n|\mu_i, \sigma_i^2)$. It will be clear why in a second.
2. We quite artificially introduced another sum so that you can get familiar with notation but also to see that we can deal with more than two components in the same manner.
3. In ML, the function $\log \left( \sum_{i=1}^{I-1} \exp(a_i) \right)$ has a specific name, and it is called... the log-sum-exp function (LogSumExp, LSE). This function, in general, could result in underflow/overflow (i.e., it can be numerically unstable). To avoid that, we can use the following trick:

$$LSE(a_0, \ldots, a_{I-1}) = a^* + LSE(a_0 - a^*, \ldots, a_{I-1} - a^*), \tag{2.36}$$

where $a^* = \max\{a_0, \ldots, a_{I-1}\}$. Please remember this function, because it plays a crucial role in many topics in AI. And if you do not want to make a mistake, always check first if a library you use provides the LSE function. Most of them do!

In general, the log-likelihood function for a MoG with $K$ components is defined as follows:

$$\log p(\mathcal{D}; \theta) = \sum_{n=1}^{N} \log \left( \sum_{k=0}^{K-1} \exp(\log \omega_k + \log \mathcal{N}(x|\mu_k, \Sigma_k)) \right). \tag{2.37}$$

### 2.3.3 Training: Algorithms

Now we are ready to implement a training algorithm. Since we know the form of the log-likelihood function, we can pick any optimization algorithm we want. In practice, there are two options:

1. We can use a famous algorithm called **Expectation-Maximization** (EM) that carries out training in two stages: (i) First, calculate expected probabilities for points belonging to specific components. (ii) Second, update the parameters of components. We will not discuss this algorithm here since we are lazy deep-learning bastards. However, I highly recommend looking up the details of the EM algorithm in one of the textbooks, e.g., [1, 4, 5], and implementing it.
2. The optimization can be done using gradient-based methods.

Here, we follow the second path, because it is easier, since we can use any library with Autograd. The procedure is straightforward: Calculate gradients of the log-likelihood with respect to the parameters, and update the parameters. That is all! The only thing we need to remember is to use the LSE function.

Some examples of fitting a mixture of Gaussians (for different numbers of components) to the half-moon data are presented in Fig. 2.5. As you can notice, my curious reader, fitting too *few* components (Fig. 2.5a) results in a very poorly generalizable model while taking too *many* components could lead to numerical instabilities (e.g., if a component is located on a datapoint and the variance collapses to 0). In general, taking more components (e.g., see Fig. 2.5d) results in a better fit and a better log-likelihood value. Touching upon this topic, sometimes we need a regularization of some sort, but we will not discuss it further. You can read about it, my curious reader, in one of the textbooks (e.g., about a Bayesian treatment of a mixture of Gaussians [1]).

### 2.3.4 Other Mixture Models

Here, we discussed mixtures of Gaussians, however, we can formulate any mixture model, depending on the character of observable data. For instance, for binary data, we get a mixture of Bernoullis:

$$p(x;\theta) = \sum_{k=1}^{K} \pi_k \mathrm{Bern}(x|\theta). \tag{2.38}$$

Similarly, we can build other mixture models. In general, building a mixture of distributions from the exponential family of distributions is straightforward due to the fact that we can use the LSE trick to formulate a tractable log-likelihood-based training objective.

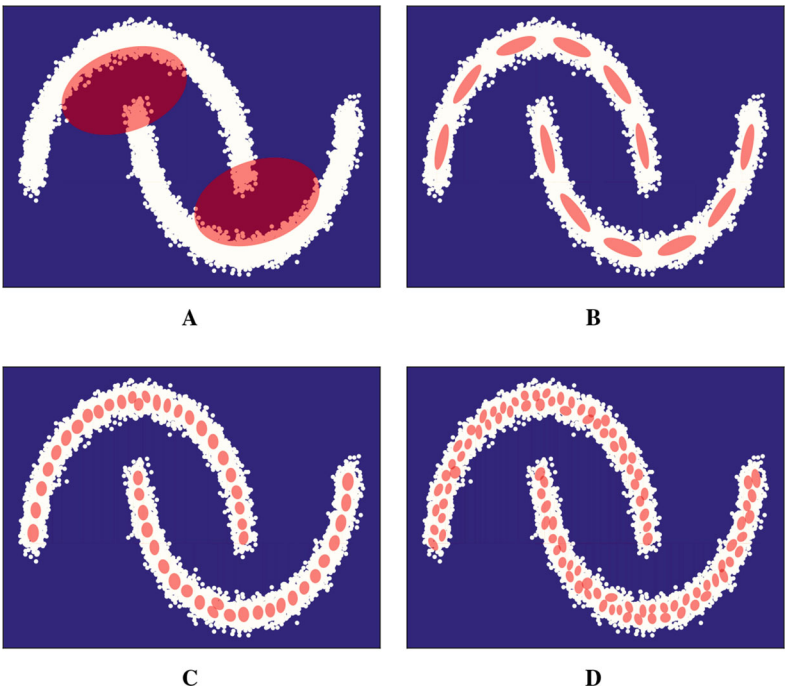



**Fig. 2.5** Several examples of MoGs with varying number of components: (**a**) $K = 2$, (**b**) $K = 12$, (**c**) $K = 50$, (**d**) $K = 100$.

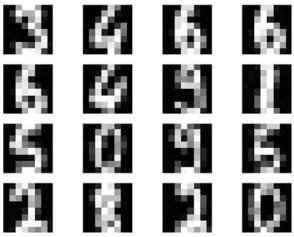


**Fig. 2.6** A few examples of handwritten, tiny images ($8 \times 8$ pixels).

### 2.3.5 Coding Mixture of Gaussians (MoG)

Alright, my curious reader, it is enough of a chit-chat. I know you, and I feel you, we should code a MoG up! We will train our MoG on simple data (tiny handwritten images). Some examples are presented in Fig. 2.6. These are not necessarily complicated, and very small, but still training a model to generate such objects is nontrivial, as we will see.

But back to work! An implementation of an MoG is presented in the code snippet below. Please take a careful look, my curious reader, and try to understand every single line. For instance, we use Gaussian components parameterized with means

and diagonal covariance matrices (in fact, we use log variances for numerical convenience). Moreover, we allow two options for component probabilities: $\omega$'s are either trainable or fixed and equiprobable, i.e., $\omega_k = \frac{1}{K}$ for all $k = 0, \ldots, K-1$. Note that we must keep $\sum_k \omega_k = 1$ and each $\omega_k \in [0, 1]$. If we train $\omega$'s with a gradient-based method, it is hard to ensure that. Therefore, we introduce parameters $w$'s in the code and then use the softmax function to turn them into $\omega$'s. A simple trick that always works.

Alright, we are good to go. Buckle up and code it up!

```
class MoG(nn.Module):
    def __init__(self, D, K, uniform=False):
        super(MoG, self).__init__()

        print('MoG by JT.')

        # hyperparams
        self.uniform = uniform
        self.D = D  # the dimensionality of the input
        self.K = K  # the number of components

        # params
        self.mu = nn.Parameter(torch.randn(1, self.K, self.D) *
    0.25 + 0.5)
        self.log_var = nn.Parameter(-3. * torch.ones(1, self.K,
    self.D))

        if self.uniform:
            self.w = torch.zeros(1, self.K)
            self.w.requires_grad = False
        else:
            self.w = nn.Parameter(torch.zeros(1, self.K))

        # other
        self.PI = torch.from_numpy(np.asarray(np.pi))

    def log_diag_normal(self, x, mu, log_var, reduction='sum',
    dim=1):
        log_p = -0.5 * torch.log(2. * self.PI) - 0.5 * log_var -
    0.5 * torch.exp(-log_var) * (x.unsqueeze(1) - mu)**2.
        return log_p

    def forward(self, x, reduction='mean'):
        # calculate components
        log_pi = torch.log(F.softmax(self.w, 1))  # B x K,
    softmax is used for R^K -> [0,1]^K s.t. sum(pi) = 1
        log_N = torch.sum(self.log_diag_normal(x, self.mu, self.
    log_var), 2)  # B x K, log-diag-Normal for K components

        # =====LOSS: Negative Log-Likelihood
        NLL_loss = -torch.logsumexp(log_pi + log_N,  1)  # B

```

```
        # Final LOSS
        if reduction == 'sum':
            return NLL_loss.sum()
        elif reduction == 'mean':
            return NLL_loss.mean()
        else:
            raise ValueError('Either ‘sum‘ or ‘mean‘.')

    def sample(self, batch_size=64):
        # init an empty tensor
        x_sample = torch.empty(batch_size, self.D)

        # sample components
        pi = F.softmax(self.w, 1)  # B x K, softmax is used for R^K -> [0,1]^K s.t. sum(pi) = 1

        indices = torch.multinomial(pi, batch_size, replacement=True).squeeze()

        for n in range(batch_size):
            indx = indices[n]  # pick the n-th component
            x_sample[n] = self.mu[0,indx] + torch.exp(0.5*self.log_var[0,indx]) * torch.randn(self.D)

        return x_sample

    def log_prob(self, x, reduction='mean'):
        with torch.no_grad():
            # calculate components
            log_pi = torch.log(F.softmax(self.w, 1))  # B x K, softmax is used for R^K -> [0,1]^K s.t. sum(pi) = 1
            log_N = torch.sum(self.log_diag_normal(x, self.mu, self.log_var), 2)  # B x K, log-diag-Normal for K components

            # log_prob
            log_prob = torch.logsumexp(log_pi + log_N,  1)  # B

            if reduction == 'sum':
                return log_prob.sum()
            elif reduction == 'mean':
                return log_prob.mean()
            else:
                raise ValueError('Either ‘sum‘ or ‘mean‘.')
```

**Listing 2.1** An example of an implementation of MoG.

After running our MoG with $K = 25$ and trainable components probabilities, we should obtain results like in Fig. 2.7. A few notes here:

1. The generated images are not fantastic (Fig. 2.7a); however, they resemble images (if you close your eyes a bit, perhaps). The point here is not to get state of the art; we are here to learn the principles. If you wish to get better results, go ahead and play with the code!

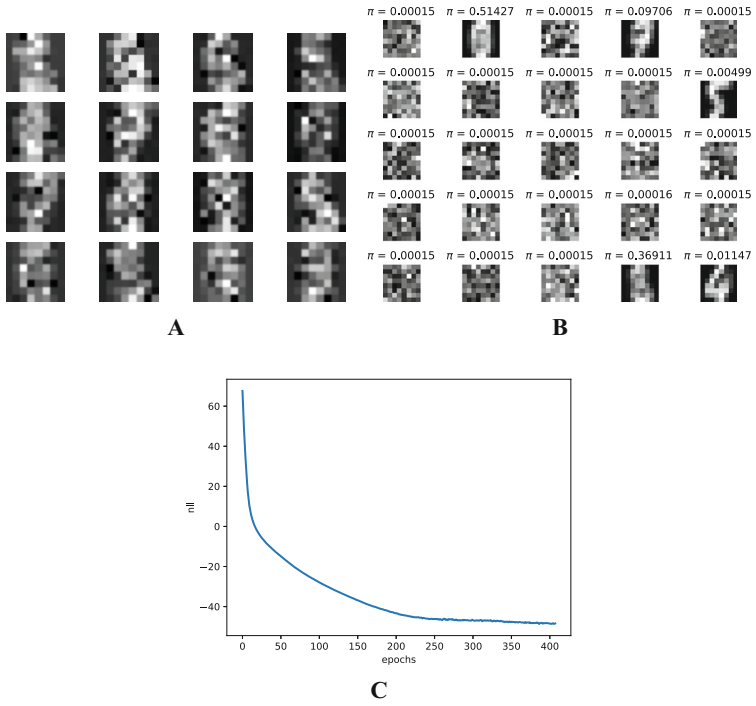



**Fig. 2.7** (**a**) A sample of generated images from a trained MoG. (**b**) Examples of means of a trained MoG. (**c**) An example of the negative log-like trained MoG.

2. Take a look at the means (Fig. 2.7b)! It is really interesting what happens there. First, there are many collapsed components for which $\omega_k$ is close to 0 (component probabilities are presented above each mean in Fig. 2.7b). This means that they are basically useless and we could even remove them from the mixture model. Second, the components with nonzero probability are either quite specific (e.g., a component for 5s or 4s) or very generic (e.g., the two means with high $\omega$'s).
3. The training is very smooth and stable (see Fig. 2.7c). The negative log-likelihood is dropping nicely. However, it does not mean we should not use additional tricks and/or play with hyperparameter values to get even better results! Please remember, my curious reader, never be fooled by nicely looking curves or loss values in generative AI. Always try to inspect what is generated by your model.
4. Please be aware that the solution we found here is local, not global! It means that there exist multiple non-collapsed solutions and finding the best one (global) is hard if not (almost) impossible. Moreover, if a component collapses (i.e., $\sigma_k^2 = 0$), then we get the unbounded likelihood. To mitigate this issue, one can add a regularization term [6].

## 2.4 Probabilistic Circuits

Mixture models are latent variables models (since we have latent variables representing component numbers). Here, we discuss **finite** mixture models with $z$ being a categorical variable. There are also **infinite** mixture models with $z$ being continuous, for example. We will look into those models on another occasion. Moreover, the presented mixture models are not very **deep**, since there is only a single level of latent variables. In the following, we will briefly look into a very broad class of hierarchical mixture models: **Probabilistic Circuits**. Before we do that, we will say a few words about the simplest model possible, namely, fully factorized models.

### 2.4.1 Fully Factorized Models

An MoG is a quite powerful model that allows calculating the evidence. However, calculating MAP for MoGs is intractable because:

$$\max_x p(x) = \max_x \sum_k \omega_k p_k(x) \tag{2.39}$$

$$\neq \sum_k \max_x \omega_k p_k(x), \tag{2.40}$$

in other words, we cannot swap the sum and the max.

However, if we take a very simple model that is fully factorized (i.e., all variables are stochastically independent from each other):

$$p(\mathbf{x}) = \prod_i p_i(x_i), \tag{2.41}$$

then calculating EVI, MAR, and MAP is linear! But it is not very expressive... However, here is a question: Can we combine somehow a mixture model and a fully factorized model to get the best of the two worlds? The answer is... YES!

### 2.4.2 Hierarchical Mixture Models a.k.a Probabilistic Circuits

#### 2.4.2.1 Building Blocks

Let us introduce the following semantics [7]:

- Nodes correspond either to distributions or an operation (a summation or a product);
- Edges define directions of probability flows;

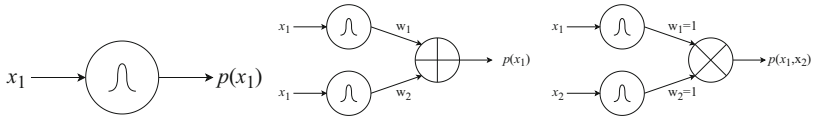



**Fig. 2.8** Basic probabilistic units: (**a**) An example of a base unit. (**b**) An example of a sum unit. (**c**) An example of a product unit.

- Inputs to nodes are random variables (their values) or probabilities;
- Outputs are probabilities.

Further, we formulate the following basic units (see Fig. 2.8):

- A base unit (Fig. 2.8a): the input is a value of a random variable and the output is its probability (in other words, this unit implements a probability distribution);
- A sum unit (Fig. 2.8b): inputs are values of a random variable and the output is a probability of a mixture model;
- A product unit (Fig. 2.8c): inputs are values of different random variables and the output is a probability of a fully factorized distribution.

To build tractable probability distributions out of these units, we need to impose a few constraints:

- **Decomposability**: A product unit is decomposable if its children depend on disjoint sets of variables.
- **Smoothness**: A sum unit is smooth if its children depend on the same variables.

Fulfilling these two constraints gives us tractable MAR and CON (for tractable MAP we need other properties, see [2] for details).

#### 2.4.2.2 Building Probabilistic Circuits

Now we have simple building blocks that define proper distributions. Similarly to neural networks, we can build complex distributions that, in fact, define hierarchical mixture models (i.e., mixtures of mixtures of . . . and so on). In principle, for sufficiently large $K$, a mixture model can model any distribution. However, we know that if $K > N$ (i.e., there are more components than datapoints), the model collapses and is not generalizable. Therefore, having a hierarchical mixture model could help to keep expressiveness while increasing generalizability. In Fig. 2.9, we present a few examples of probabilistic circuits (PCs) consisting of simple building units.

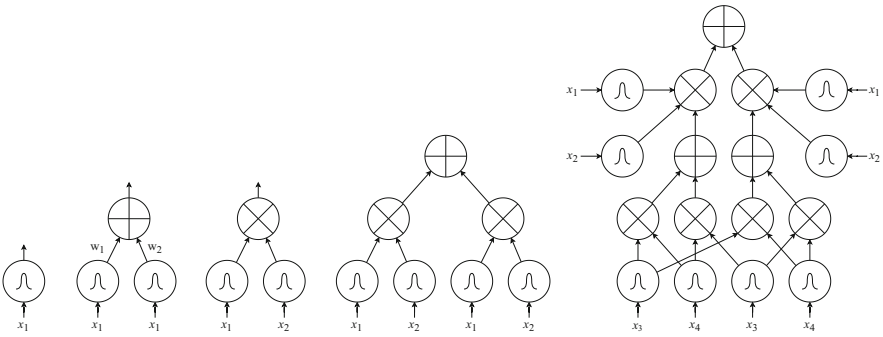



**Fig. 2.9** A few examples of probabilistic circuits, starting from simple units (left) to more complex networks (right).

**Example**

We quickly present nice properties of PCs based on an example presented in a fantastic tutorial [8]. If you are interested in PCs, this is a great start for you, my curious reader! Here, we briefly present the power of PCs!

Let us say that we have a PC like in Fig. 2.10a, defining the probability distribution over four random variables, $p(x_1, x_2, x_3, x_4)$, in which all base units are known distributions. Our goal is to calculate a marginal distribution $p(x_2, x_4)$. How to do that then? It is easy and requires the following procedure: 1. For variables we marginalize over, we set the values of based units to 1 (green nodes in Fig. 2.10b). 2. For given values of $x_2$ and $x_4$, we calculate the probabilities (orange nodes in Fig. 2.10c). 3. We carry out a feedforward (bottom-up, i.e., from all leaves to the root) evaluation of all units.

The final value of the MAR distribution for given values of $x_2$ and $x_4$ is in the root. The beauty of PCs is that all operations are tractable, they are very basic and the final result is given in time linear in circuit size!

### 2.4.2.3 Final Comments on PCs

The world of PCs is very rich and gets extra traction nowadays, mainly due to their tractability. Here are some useful pointers and thoughts:

- PCs are closely related to logical circuits [9]. For instance, sum units can be seen as probabilistic generalizations of OR units, while product units are closely related to AND units.
- If we know the architecture of PCs, training (i.e., estimating parameters of base units) can be accomplished by applying expectation-maximization [10, 11] or a gradient-based method [12].

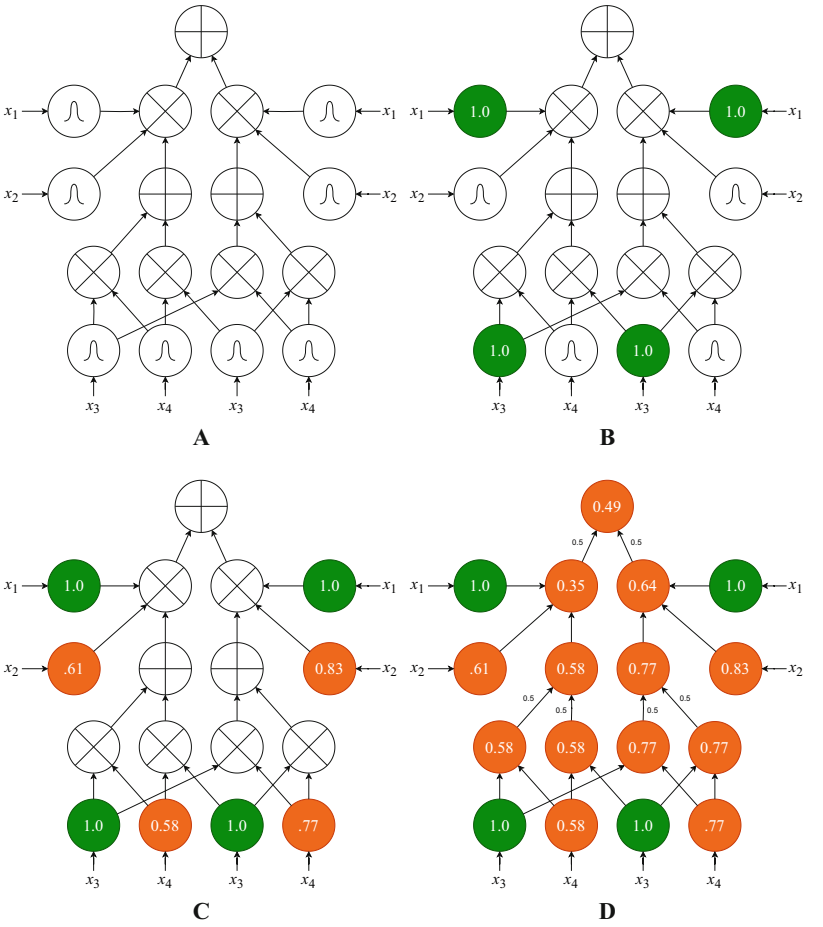



**Fig. 2.10** An example of how a model of the vector field (blue arrows) changes over time around datapoints (orange two moons).

- Learning a PC structure from data is more challenging and typically requires additional constraints or assumptions [13].
- There are specialized packages and repos that can help you to build and learn PCs, e.g., Einsum Networks in PyTorch [12], Juice in Julia [14], and SPFlow in Python [15].
- Interestingly, PCs could generalize well-known methods like decision trees or random forests. In [16], it was shown that decision trees and random forests can be turned into PCs and, additionally, treated as generative models!

I give only a tiny-weeny piece of a fantastic world of PCs here! I hope you got interested, my curious reader, and you will delve further into this class of deep (hierarchical), tractable probabilistic models.

## References


1. Christopher M Bishop. *Pattern recognition and machine learning*. Springer, 2006.
2. Y Choi, Antonio Vergari, and Guy Van den Broeck. Probabilistic circuits: A unifying framework for tractable probabilistic models. *UCLA. URL: http://starai.cs.ucla.edu/papers/ProbCirc20.pdf*, page 6, 2020.
3. Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.
4. David Barber. *Bayesian reasoning and machine learning*. Cambridge University Press, 2012.
5. Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT Press, 2012.
6. Pierre-Alexandre Mattei and Jes Frellsen. Leveraging the exact likelihood of deep latent variable models. *Advances in Neural Information Processing Systems*, 31, 2018.
7. Antonio Vergari, YooJung Choi, Anji Liu, Stefano Teso, and Guy Van den Broeck. A compositional atlas of tractable circuit operations for probabilistic inference. *Advances in Neural Information Processing Systems*, 34:13189–13201, 2021.
8. Antonio Vergari, YooJung Choi, Robert Peharz, and Guy Van den Broeck. Probabilistic circuits: Representations, inference, learning and applications. In *Tutorial at the The 34th AAAI Conference on Artificial Intelligence*, 2020.
9. Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.
10. Mattia Desana and Christoph Schnörr. Learning arbitrary sum-product network leaves with expectation-maximization. *arXiv preprint arXiv:1604.07243*, 2016.
11. Robert Peharz, Robert Gens, Franz Pernkopf, and Pedro Domingos. On the latent variable interpretation in sum-product networks. *IEEE transactions on pattern analysis and machine intelligence*, 39(10):2030–2044, 2016.
12. Robert Peharz, Steven Lang, Antonio Vergari, Karl Stelzner, Alejandro Molina, Martin Trapp, Guy Van den Broeck, Kristian Kersting, and Zoubin Ghahramani. Einsum networks: Fast and scalable learning of tractable probabilistic circuits. In *International Conference on Machine Learning*, pages 7563–7574. PMLR, 2020.
13. Robert Gens and Domingos Pedro. Learning the structure of sum-product networks. In *International conference on machine learning*, pages 873–880. PMLR, 2013.
14. Meihua Dang, Pasha Khosravi, Yitao Liang, Antonio Vergari, and Guy Van den Broeck. Juice: A Julia package for logic and probabilistic circuits. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 16020–16023, 2021.

15. Alejandro Molina, Antonio Vergari, Karl Stelzner, Robert Peharz, Pranav Subramani, Nicola Di Mauro, Pascal Poupart, and Kristian Kersting. Spflow: An easy and extensible library for deep probabilistic learning using sum-product networks. *arXiv preprint arXiv:1901.03704*, 2019.
16. Alvaro Correia, Robert Peharz, and Cassio P de Campos. Joints in random forests. *Advances in neural information processing systems*, 33:11404–11415, 2020.

# Chapter 3
# Autoregressive Models


## 3.1 Introduction

Before we start discussing how we can model the distribution $p(\mathbf{x})$, we refresh our memory about the core rules of probability theory, namely, the **sum rule** and the **product rule**. Let us introduce two random variables $\mathbf{x}$ and $\mathbf{y}$. Their joint distribution is $p(\mathbf{x}, \mathbf{y})$. The **product rule** allows us to *factorize* the joint distribution in two manners, namely:

$$p(\mathbf{x}, \mathbf{y}) = p(\mathbf{x}|\mathbf{y})p(\mathbf{y}) \tag{3.1}$$
$$= p(\mathbf{y}|\mathbf{x})p(\mathbf{x}). \tag{3.2}$$

In other words, the joint distribution could be represented as a product of a marginal distribution and a conditional distribution. The **sum rule** tells us that if we want to calculate the marginal distribution over one of the variables, we must integrate out (or sum out) the other variable, that is:

$$p(\mathbf{x}) = \sum_{\mathbf{y}} p(\mathbf{x}, \mathbf{y}). \tag{3.3}$$

These two rules will play a crucial role in probability theory and statistics and, in particular, in formulating deep generative models.

Now, let us consider a high-dimensional random variable $\mathbf{x} \in \mathcal{X}^D$ where $\mathcal{X} = \{0, 1, \ldots, 255\}$ (e.g., pixel values) or $\mathcal{X} = \mathbb{R}$. Our goal is to model $p(\mathbf{x})$. Before we jump into thinking of specific parameterization, let us first apply the product rule to express the joint distribution in a different manner:

$$p(\mathbf{x}) = p(x_1) \prod_{d=2}^{D} p(x_d|\mathbf{x}_{<d}), \tag{3.4}$$

where $\mathbf{x}_{<d} = [x_1, x_2, \ldots, x_{d-1}]^\top$. For instance, for $\mathbf{x} = [x_1, x_2, x_3]^\top$, we have $p(\mathbf{x}) = p(x_1)p(x_2|x_1)p(x_3|x_1, x_2)$.



J. M. Tomczak, *Deep Generative Modeling*,
https://doi.org/10.1007/978-3-031-64087-2_3

As we can see, the product rule applied multiple times to the joint distribution provides a principled manner of factorizing the joint distribution into many conditional distributions. That's great news! However, modeling all conditional distributions $p(x_d|\mathbf{x}_{<d})$ separately is simply infeasible! If we did that, we would obtain $D$ separate models, and the complexity of each model would grow due to varying conditioning. A natural question is whether we can do better, and the answer is yes.

## 3.2 Autoregressive Models Parameterized by Neural Networks

As mentioned earlier, we aim to model the joint distribution $p(\mathbf{x})$ using conditional distributions. A potential solution to the issue of using $D$ separate model is utilizing a single, shared model for the conditional distribution. However, we need to make some assumptions to use such a shared model. In other words, we look for an **autoregressive model** (ARM). In the next subsection, we outline ARMs parameterized with various *neural networks*. After all, we are talking about deep generative models, so using a neural network is not surprising, isn't it?

### 3.2.1 Finite Memory

The first attempt to limit the complexity of a conditional model is to assume a *finite memory*. For instance, we can assume that each variable is dependent on no more than two other variables, namely:

$$p(\mathbf{x}) = p(x_1)p(x_2|x_1)\prod_{d=3}^{D} p(x_d|x_{d-1}, x_{d-2}). \tag{3.5}$$

Then, we can use a small neural network, e.g., multilayered perceptrons (MLP), to predict the distribution of $x_d$. If $\mathcal{X} = \{0, 1, \ldots, 255\}$, the MLP takes $x_{d-1}, x_{d-2}$ and outputs probabilities for the categorical distribution of $x_d$, $\theta_d$. An example of this approach is depicted in Fig. 3.1.

**A side note**

The MLP could be of the following form:

$$[x_{d-1}, x_{d-2}] \rightarrow \text{Linear}(2, M) \rightarrow \text{ReLU} \rightarrow \text{Linear}(M, 256) \rightarrow \text{softmax} \rightarrow \theta_d$$

where $M$ denotes the number of hidden units, e.g., $M = 300$.

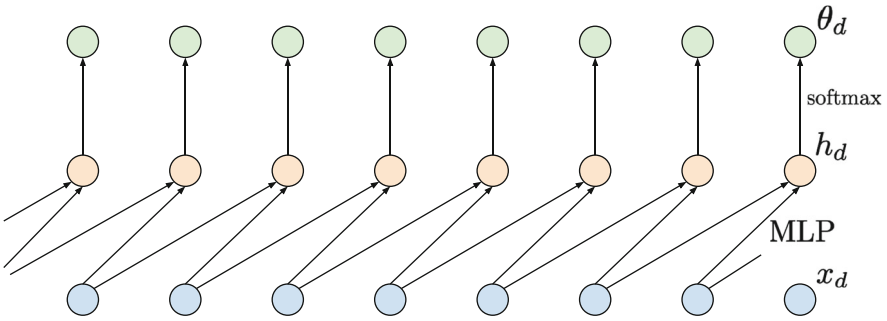



**Fig. 3.1** An example of applying a shared MLP depending on two last inputs. Inputs are denoted by blue nodes (bottom), intermediate representations are denoted by orange nodes (middle), and output probabilities are denoted by green nodes (top). Notice that a probability $\theta_d$ is **not** dependent on $x_d$.

It is important to notice that now we use a single, shared MLP to predict probabilities for $x_d$. Such a model is not only nonlinear, but also its parameterization is convenient due to a relatively small number of weights to be trained. However, the obvious drawback of this approach is a **limited memory** (i.e., only two last variables in our example). Moreover, it is unclear a priori how many variables we should use in conditioning. In many problems, e.g., image processing, learning *long-range statistics* is crucial to understand complex patterns in data; therefore, having long-range memory is essential.

### 3.2.2 Long-Range Memory Through RNNs

A possible solution to the problem of a short-range memory modeled by an MLP relies on applying a recurrent neural network (RNN) [1, 2]. In other words, we can model the conditional distributions as follows [3]:

$$p(x_d|\mathbf{x}_{<d}) = p\left(x_d|\text{RNN}(x_{d-1}, h_{d-1})\right), \tag{3.6}$$

where $h_d = \text{RNN}(x_{d-1}, h_{d-1})$ and $h_d$ is a hidden context that acts as a *memory* that allows learning long-range dependencies. An example of using an RNN is presented in Fig. 3.2.

This approach gives a single parameterization; thus, it is efficient and also solves the problem of a finite memory. So far so good! Unfortunately, RNNs suffer from other issues, namely:

- they are sequential, hence, slow,
- if they are badly conditioned (i.e., the eigenvalues of a weight matrix are larger or smaller than 1), then they suffer from exploding or vanishing gradients, respectively, that hinder learning long-range dependencies.

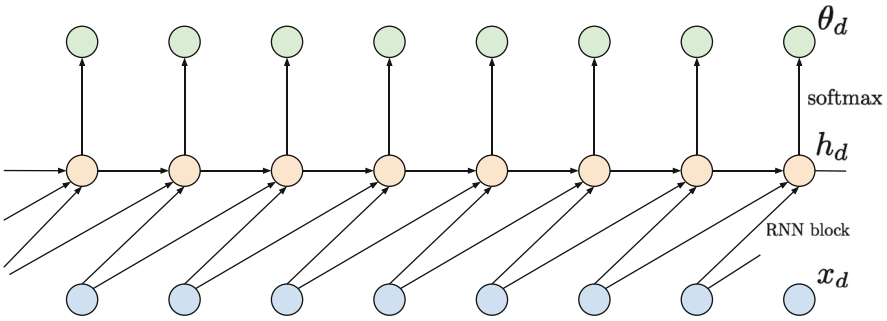



**Fig. 3.2** An example of applying an RNN depending on two last inputs. Inputs are denoted by blue nodes (bottom), intermediate representations are denoted by orange nodes (middle), and output probabilities are denoted by green nodes (top). Notice that compared to the approach with a shared MLP, there is an additional dependency between intermediate nodes $h_d$.

There exist methods to help train RNNs like gradient clipping or, more generally, gradient regularization [4] or orthogonal weights [5]. However, here we are not interested in looking into rather specific solutions to new problems. We seek for a different parameterization that could solve our original problem, namely, modeling long-range dependencies in an ARM.

### 3.2.3 Long-Range Memory Through Convolutional Nets

In [6, 7], it was noticed that convolutional neural networks (CNNs) could be used instead of RNNs to model long-range dependencies. To be more precise, one-dimensional convolutional layers (Conv1D) could be stacked together to process sequential data. The advantages of such an approach are the following:

- kernels are shared (i.e., an efficient parameterization),
- the processing is done in parallel, which greatly speeds up computations,
- by stacking more layers, the effective kernel size grows with the network depth.

These three traits seem to place Conv1D-based neural networks as a perfect solution to our problem. However, can we indeed use them straight away?

A Conv1D can be applied to calculate embeddings like in [7], but it cannot be used for autoregressive models. Why? Because we need convolutions to be **causal** [8]. *Causal* in this context means that a Conv1D layer is dependent on the last $k$ inputs but the current one (*option A*) or with the current one (*option B*). In other words, we must "cut" the kernel in half and forbid it to look into the next variables (look into the future). Importantly, option A is required in the first layer because the final output (i.e., the probabilities $\theta_d$) cannot be dependent on $x_d$. Additionally, if we are concerned about the effective kernel size, we can use *dilation* larger than 1.

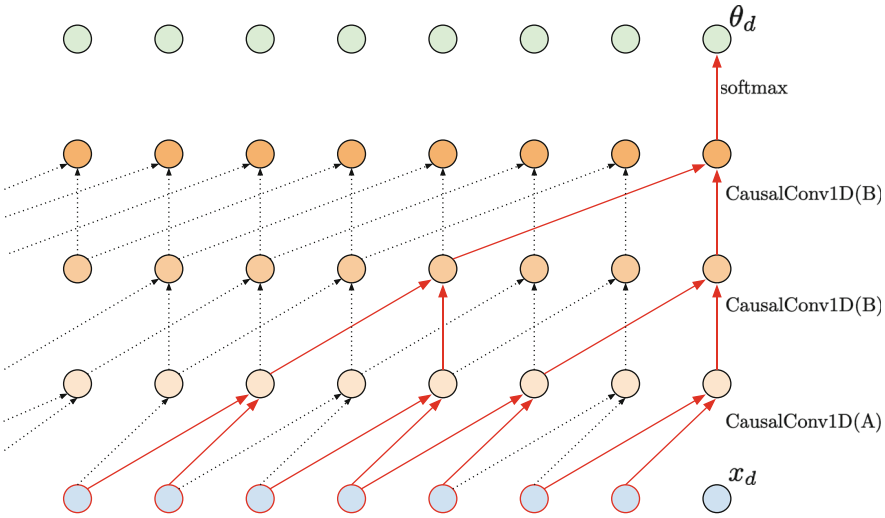



**Fig. 3.3** An example of applying causal convolutions. The kernel size is 2, but by applying dilation in higher layers, a much larger input could be processed (red edges); thus, a larger memory is utilized. Notice that the first layers must be option A to ensure proper processing.

In Fig. 3.3, we present an example of a neural network consisting of three *causal Conv1D* layers. The first CausalConv1D is of type A, i.e., it does not take into account only the last $k$ inputs without the current one. Then, in the next two layers, we use CausalConv1D (option B) with dilation 2 and 3. Typically, the dilation values are 1, 2, 4, and 8 [9]; however, taking 2 and 4 would not nicely fit in a figure. We highlight in red all connections that go from the output layer to the input layer. As we can notice, stacking CausalConv1D layers with a dilation larger than one allows us to learn long-range dependencies (in this example, by looking at seven last inputs).

An example of an implementation of CausalConv1D layer is presented below. If you are still confused about option A and option B, please analyze the code snippet step by step.

```
class CausalConv1d(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size,
    dilation, A=False, **kwargs):
        super(CausalConv1d, self).__init__()

        # The general idea is the following: We take the built-in
     PyTorch Conv1D. Then, we must pick a proper padding, because
     we must ensure the convolutional is causal. Eventually, we
    must remove some final elements of the output, because we
    simply don't need them! Since CausalConv1D is still a
    convolution, we must define the kernel size, dilation, and
    whether it is option A (A=True) or option B (A=False).
    Remember that by playing with dilation we can enlarge the
    size of the memory.

```

```
        # attributes:
        self.kernel_size = kernel_size
        self.dilation = dilation
        self.A = A # whether option A (A=True) or B (A=False)
        self.padding = (kernel_size - 1) * dilation + A * 1

        # we will do padding by ourselves in the forward pass!
        self.conv1d = torch.nn.Conv1d(in_channels, out_channels,
                                      kernel_size, stride=1,
                                      padding=0,
                                      dilation=dilation,**kwargs)

    def forward(self, x):
        # We do padding only from the left! This is more
    efficient implementation.
        x = torch.nn.functional.pad(x, (self.padding, 0))
        conv1d_out = self.conv1d(x)
        if self.A:
            # Remember, we cannot be dependent on the current
    component, therefore, the last element is removed.
            return conv1d_out[:, :, : -1]
        else:
            return conv1d_out
```

**Listing 3.1** Causal Convolution 1D.

The CausalConv1D layers are better suited to modeling sequential data than RNNs. They obtain not only better results (e.g., classification accuracy) but also allow learning long-range dependencies more efficiently than RNNs [8]. Moreover, they do not suffer from exploding/vanishing gradient issues. As a result, they seem to be a perfect parameterization for autoregressive models! Their supremacy has been proven in many cases, including audio processing by WaveNet, a neural network consisting of CausalConv1D layers [9], or image processing by PixelCNN, a model with CausalConv2D components [10].

Then, is there any drawback of applying autoregressive models parameterized by causal convolutions? Unfortunately, yes, there is, and it is connected with sampling. If we want to evaluate probabilities for given inputs, we need to calculate the forward pass where all calculations are done in parallel. However, if we want to sample new objects, we must iterate through all positions (think of a big for-loop, from the first variable to the last one) and iteratively predict probabilities and sample new values. Since we use convolutions to parameterize the model, we must do $D$ full forward passes to get the final sample. That is a big waste, but, unfortunately, that is the price we must pay for all "goodies" coming from the convolutional-based parameterization of the ARM. Fortunately, there is ongoing research on speeding up computations; e.g., see [11].

### 3.2.4 Deep Generative Autoregressive Model in Action!

Alright, let us talk more about the details and how to implement an ARM. Here, and in the whole book, we focus on images, e.g., $\mathbf{x} \in \{0, 1, \ldots, 15\}^{64}$. Since images are represented by integers, we will use the categorical distribution to represent them (in the next chapters, we will comment on the choice of distribution for images and present some alternatives). We model $p(\mathbf{x})$ using an ARM parameterized by CausalConv1D layers. As a result, each conditional is the following:

$$p(x_d|\mathbf{x}_{<d}) = \text{Categorical}\left(x_d|\theta_d\left(\mathbf{x}_{<d}\right)\right) \tag{3.7}$$

$$= \prod_{l=1}^{L} \left(\theta_{d,l}\right)^{[x_d=l]}, \tag{3.8}$$

where $[a = b]$ is the Iverson bracket (i.e., $[a = b] = 1$ if $a = b$, and $[a = b] = 0$ if $a \neq b$), $\theta_d\left(\mathbf{x}_{<d}\right) \in [0, 1]^{16}$ is the output of the CausalConv1D-based neural network with the softmax in the last layer, so $\sum_{l=1}^{L} \theta_{d,l} = 1$. To be very clear, the last layer must have 16 output channels (because there are 16 possible values per pixel), and the softmax is taken over these 16 values. We stack CausalConv1D layers with nonlinear activation functions in between (e.g., LeakyRELU). Of course, we must remember to take the option A CausalConv1D as the first layer! Otherwise, we break the assumption about taking into account $x_d$ in predicting $\theta_d$.

What about the objective function? ARMs are the likelihood-based models, so for given $N$ i.i.d. datapoints $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$, we aim at maximizing the logarithm of the likelihood function, that is (we will use the product and sum rules again):

$$\ln p(\mathcal{D}) = \ln \prod_n p(\mathbf{x}_n) \tag{3.9}$$

$$= \sum_n \ln p(\mathbf{x}_n) \tag{3.10}$$

$$= \sum_n \ln \prod_d p(x_{n,d}|\mathbf{x}_{n,<d}) \tag{3.11}$$

$$= \sum_n \left( \sum_d \ln p(x_{n,d}|\mathbf{x}_{n,<d}) \right) \tag{3.12}$$

$$= \sum_n \left( \sum_d \ln \text{Categorical}\left(x_d|\theta_d\left(\mathbf{x}_{<d}\right)\right) \right) \tag{3.13}$$

$$= \sum_n \left( \sum_d \left( \sum_{l=1}^{L} [x_d = l] \ln \theta_d\left(\mathbf{x}_{<d}\right) \right) \right). \tag{3.14}$$

For simplicity, we assumed that $\mathbf{x}_{<1} = \emptyset$, i.e., no conditioning. As we can notice, the objective function takes a very nice form! First, the logarithm over the i.i.d. data $\mathcal{D}$ results in a sum over data points of the logarithm of individual distributions $p(\mathbf{x}_n)$. Second, applying the product rule, together with the logarithm, results in another

sum, this time over dimensions. Eventually, by parameterizing the conditionals by CausalConv1D, we can calculate all $\theta_d$ in one forward pass and then check the pixel value (see the last line of $\ln p(\mathcal{D})$). Ideally, we want $\theta_{d,l}$ to be as close to 1 as possible if $x_d = l$.

### 3.2.5 Code

Uff... Alright, let's take a look at some code. The full code is available under the following: https://github.com/jmtomczak/intro_dgm. Here, we focus only on the code for the model. We provide details in the comments.

```
class ARM(nn.Module):
    def __init__(self, net, D=2, num_vals=256):
        super(ARM, self).__init__()

        # Remember, always credit the author, even if it's you ;)
        print('ARM by JT.')

        # This is a definition of a network. See the next cell.
        self.net = net
        # This is how many values a pixel can take.
        self.num_vals = num_vals
        # This is the problem dimensionality (the number of
    pixels)
        self.D = D

    # This function calculates the arm output.
    def f(self, x):
        # First, we apply causal convolutions.
        h = self.net(x.unsqueeze(1))
        # In channels, we have the number of values. Therefore,
    we change the order of dims.
        h = h.permute(0, 2, 1)
        # We apply softmax to calculate probabilities.
        p = torch.softmax(h, 2)
        return p

    # The forward pass calculates the log-probability of an image
    .
    def forward(self, x, reduction='avg'):
        if reduction == 'avg':
            return -(self.log_prob(x).mean())
        elif reduction == 'sum':
            return -(self.log_prob(x).sum())
        else:
            raise ValueError('reduction could be either `avg` or
    `sum`.')

    # This function calculates the log-probability (log-
    categorical).
```

```
    # See the full code in the separate file for details.
    def log_prob(self, x):
        mu_d = self.f(x)
        log_p = log_categorical(x, mu_d, num_classes=self.
    num_vals, reduction='sum', dim=-1).sum(-1)

        return log_p

    # This function implements a sampling procedure.
    def sample(self, batch_size):
        # As you can notice, we first initialize a tensor with
    zeros.
        x_new = torch.zeros((batch_size, self.D))

        # Then, iteratively, we sample a value for a pixel.
        for d in range(self.D):
            p = self.f(x_new)
            x_new_d = torch.multinomial(p[:, d, :], num_samples
    =1)
            x_new[:, d] = x_new_d[:,0]

        return x_new
```

**Listing 3.2** Autoregressive Model parameterized by Causal Convolutions 1D.

```
# An example of a network. NOTICE: The first layer is A=True,
    while all other layers are A=False.
# At this point we should know already why :)
M = 256

net = nn.Sequential(
    CausalConv1d(in_channels=1, out_channels=MM, dilation=1,
    kernel_size=kernel, A=True, bias=True),
    nn.LeakyReLU(),
    CausalConv1d(in_channels=MM, out_channels=MM, dilation=1,
    kernel_size=kernel, A=False, bias=True),
    nn.LeakyReLU(),
    CausalConv1d(in_channels=MM, out_channels=MM, dilation=1,
    kernel_size=kernel, A=False, bias=True),
    nn.LeakyReLU(),
    CausalConv1d(in_channels=MM, out_channels=num_vals, dilation
    =1, kernel_size=kernel, A=False, bias=True))
```

**Listing 3.3** An example of a network.

Perfect! Now we are ready to run the full code. After training our ARM, we should obtain results similar to those in Fig. 3.4.

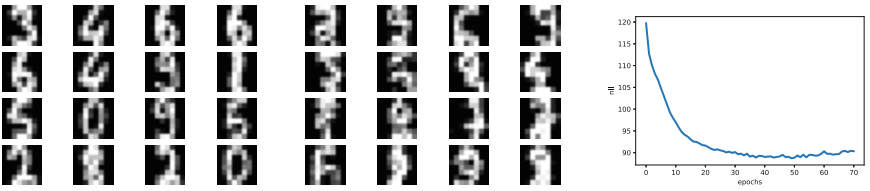



**Fig. 3.4** An example of outcomes after the training: (**a**) Randomly selected real images. (**b**) Unconditional generations from the ARM. (**c**) The validation curve during training.

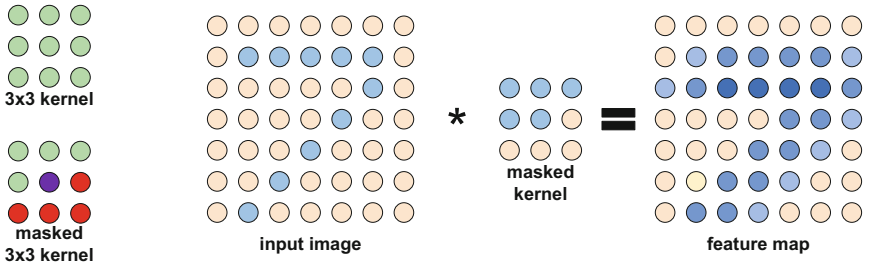



**Fig. 3.5** An example of a masked $3 \times 3$ kernel (i.e., a causal 2D kernel): (*left*) A difference between a standard kernel (all weights are used; denoted by green), and a masked kernel (some weights are masked, i.e., not used; in red). For the masked kernel, we denoted the node (pixel) in the middle in violet, because it is either masked (option A) or not (option B). (*middle*) An example of an image (light orange nodes, zeros; light blue nodes, ones) and a masked kernel (option A). (*right*) The result of applying the masked kernel to the image (with padding equal to 1).

### 3.2.6 Is It All? No!

First of all, we discussed one-dimensional causal convolutions that are typically insufficient for modeling images due to their spatial dependencies in 2D (or 3D if we consider more than 1 channel; for simplicity, we focus on a 2D case). In [10], a CausalConv2D was proposed. The idea is similar to that discussed so far, but now we need to ensure that the kernel will not look into future pixels in both the x-axis and y-axis. In Fig. 3.5, we present the difference between a standard kernel where all kernel weights are used and a masked kernel with some weights zeroed out (or masked). Notice that in CausalConv2D, we must also use option A for the first layer (i.e., we skip the pixel in the middle), and we can pick option B for the remaining layers. In Fig. 3.6, we present the same example as in Fig. 3.5 but using numeric values.

In [12], the authors propose a further improvement on the causal convolutions. The main idea relies on creating a block that consists of vertical and horizontal convolutional layers. Moreover, they use the gated non-linearity function, namely:

$$\mathbf{h} = \tanh(\mathbf{W}\mathbf{x}) \odot \sigma(\mathbf{V}\mathbf{x}). \tag{3.15}$$

$$\begin{bmatrix} 0&0&0&0&0&0&0 \\ 0&1&1&1&1&1&0 \\ 0&0&0&0&0&1&0 \\ 0&0&0&0&1&0&0 \\ 0&0&0&1&0&0&0 \\ 0&0&1&0&0&0&0 \\ 0&1&0&0&0&0&0 \end{bmatrix} * \begin{bmatrix} 1&1&1 \\ 1&1&0 \\ 0&0&0 \end{bmatrix} = \begin{bmatrix} 0&0&0&0&0&0&0 \\ 0&1&2&2&2&2&1 \\ 1&2&3&3&3&3&2 \\ 0&0&0&0&2&2&1 \\ 0&0&0&2&2&1&0 \\ 0&0&2&2&1&0&0 \\ 0&2&2&1&0&0&0 \end{bmatrix}$$

**Fig. 3.6** The same example as in Fig. 3.5 but with numeric values.

See Figure 2 in [12] for details.

Further improvements on ARMs applied to images are presented in [13]. Therein, the authors propose to replace the categorical distribution used for modeling pixel values with the discretized logistic distribution. Moreover, they suggest using a mixture of discretized logistic distributions to further increase the flexibility of their ARMs.

The introduction of the causal convolution opened multiple opportunities for deep generative modeling and allowed obtaining state-of-the-art generations and density estimations. It is impossible to review all papers here; we just name a few interesting directions/applications that are worth remembering:

- An alternative ordering of pixels was proposed in [14]. Instead of using the ordering from left to right, a "zig-zag" pattern was proposed that allows pixels to depend on pixels previously sampled to the left and above.
- ARMs could be used as stand-alone models, or they can be used in combination with other approaches. For instance, they can be used for modeling a prior in the (Variational) Auto-Encoders [15].
- ARMs could be also used to model videos [16]. Factorization of sequential data like video is very natural; ARMs fit this scenario perfectly.
- A possible drawback of ARMs is a lack of latent representation, because all conditionals are modeled explicitly from data. To overcome this issue, [17] proposed to use a PixelCNN-based decoder in a variational auto-encoder.
- An interesting and important research direction is about proposing new architectures/components of ARMs or speeding them up. As mentioned earlier, sampling from ARMs could be slow, but there are ideas to improve on that by predictive sampling [11, 18].
- Alternatively, we can replace the likelihood function with other similarity metrics, e.g., the Wasserstein distance between distributions as in quantile regression. In the context of ARMS, quantile regression was applied in [19], requiring only minor architectural changes, that resulted in improved quality scores.
- An important class of models constitutes *transformers* [20]. These models use self-attention layers instead of causal convolutions.
- Multi-scale ARMs were proposed to scale high-quality images logarithmically instead of quadratically. The idea is to make local independence assumptions [21] or by imposing a partitioning on the spatial dimensions [22]. Even though these ideas allow for lowering the memory requirements, sampling remains rather slow.

## 3.3 Autoregressive Models with Transformers

### 3.3.1 Introduction

As presented previously, the idea behind autoregressive models (ARMs) is relatively simple, since it utilizes the product rule multiple times to factorize the joint distribution, namely:

$$p(\mathbf{x}) = \left( \prod_{d=1}^{D} p(x_d|\mathbf{x}_{<d}) \right) p(x_1), \tag{3.16}$$

where $\mathbf{x}_{<d} = [x_1, x_2, \ldots, x_{d-1}]$.

We highlighted that the whole difficulty in parameterizing such a model lies in having a flexible model that can easily deal with conditionals $p(x_d|\mathbf{x}_{<d})$. By *easily* we mean having a single model that outputs probabilities for whatever the inputs $\mathbf{x}_{<d}$ are. In general, it is a nontrivial requirement. However, we showed that parameterizing ARMs using neural networks could help us in this task. Particularly, **causal convolutions** can greatly simplify the implementation of conditional distributions, and, additionally, they can speed up calculations by computing the conditional probability for a given $\mathbf{x}$ in a single forward pass. Moreover, they allow learning long-range dependencies! This last feature is incredibly important for modeling such complex objects as images. Do we need more? Are we done?

In some way, the answer is "yes." Autoregressive models parameterized by causal convolutions are extremely powerful, and they have set SOTA performance on many tasks for a long time! However, a curious researcher cannot simply accept that a problem is solved and stop looking further. Especially when there is a new sheriff in town: the **attention mechanism**! Originally, attention was utilized in neural machine translation [23, 24], but later was applied to processing sets of images [25], among many other applications. In general, it has been noticed that convolutional layers operate on a fixed grid of neighbors, while in NLP, there are irregular connections among words. For instance, Dutch grammar requires placing words in a specific order, while Polish grammar is much more flexible. As a result, using a predefined neighborhood seems to be highly limiting, and the attention mechanism, which allows *learning* long-range connections, looks like a much better approach. The idea of the attention mechanism was further extended for processing sets and sequences in the seminal paper of [20] that introduced **transformers**, an architecture built of three core components: attention layers, fully connected layers, and layer normalization. Dropout could be useful too, and other tricks as well (after all, it is deep learning, tricks, and hacks are everything!), but these three elements are crucial, and they have changed the game in many applications (especially in NLP). Let us dive into the world of transformers and how they could be used for parameterizing ARMs!

### 3.3.2 Transformers, Because Attention Is All You Need!

As the title of [20] claimed, "attention is all you need." As already mentioned, the attention mechanism could be thought of as some sort of *generalization* of convolutional layers. Assuming for a second that it is the case, then indeed we should be able to build neural networks by using the attention mechanism alone; thus, attention is the only component we need. Well, as we will see shortly, it is not entirely true, but almost.

#### 3.3.2.1 Self-Attention

The attention mechanism and the **self-attention** mechanism specifically constitute the core of many architectures, and transformers are based on them. How is this defined then? To better understand how self-attention works, please keep in mind that kernels in convolutional layers move along an object (e.g., a sequence, an image) and calculate responses (outputs) based on a fixed neighborhood. In self-attention, however, we will learn about this neighborhood. Moreover, similarly to multiple kernels in CNNs, we can have multiple versions of self-attention, called **heads** (thus, **multi-head self-attention**).

First, we focus on **single-head self-attention**, because extending it to multiple heads is really straightforward. In all of our discussions, we will assume that we have well-prepared data. This means we assume having a **tokenizer** that takes an input (e.g., a text, a molecule, an image) and returns a tensor $\mathbf{X} \in \mathcal{X}^{B \times T \times D}$, where $B$, the batch size; $T$, the number of tokens (e.g., characters in a sentence); and $D$, the dimensionality of an embedding of a token, which could be further processed by any contemporary neural networks. Formulating a tokenizer is an extremely important element of the whole system, and it could be thought of as a featurizer or a preprocessing step. A lot of the success of an AI system based on transformers comes from the fact of a powerful tokenizer. But going back to our train of thought, we assume we have a tokenizer, and later on, we will briefly discuss one for molecules.

Alright, self-attention then! In fact, self-attention is relatively simple and requires the basic knowledge of matrix calculus or, in terms of deep learning, the knowledge of how linear layers work. It works in the following way: We first transform $\mathbf{X}$ using linear layers to obtain three new tensors: **values**, **keys**, and **queries**, namely (to be consistent with code, we introduce $\otimes$ to denote batch matrix multiplication, i.e., a for-loop over the batch dimension and applying matrix multiplication to each element in a batch; this is equivalent to the `bmm` function in PyTorch):

- values: $\mathbf{V} = \mathbf{X} \otimes \mathbf{W}_v + \mathbf{b}_v$;
- keys: $\mathbf{K} = \mathbf{X} \otimes \mathbf{W}_k + \mathbf{b}_k$;
- queries: $\mathbf{Q} = \mathbf{X} \otimes \mathbf{W}_q + \mathbf{b}_q$.

**A side note**

To get a good feeling about the batch matrix multiplication $\otimes$. For a tensor $\mathbf{X}$ of size $B \times T \times D$ (or in PyTorch or Numpy: `X.shape = (B, T, D)`), and a matrix $\mathbf{W}$ of size $D \times M$, the resulting tensor of $\mathbf{X} \otimes \mathbf{W}$ is of size $B \times T \times M$. In other words, the operation $\otimes$ calculates the matrix multiplication of $\mathbf{X}_b\mathbf{W}$ for $b = 1, 2, \ldots, B$ and then concatenates all outputs. If $\mathbf{W}$ is a tensor $B \times D \times M$, then we concatenate all outputs $\mathbf{X}_b\mathbf{W}_b$ for $b = 1, 2, \ldots, B$.

Since we talk about **self**-attention, we transform the same input $\mathbf{X}$. However, in general, we are not constrained to do that. In fact, the idea of attention is highly influenced by information retrieval, hence the names of the three tensors. For instance, a search text is mapped to queries, then the information in a database is mapped to keys, and the best-matched objects are values. Then the attention mechanism could be thought of as a retrieval system. It might be confusing; therefore, it is enough to say that we call the three linear layers by these names and that's it.

Once we apply these linear transformations to $\mathbf{X}$, we can move on to the core of the self-**attention**, namely, **attention weights**, $\mathbf{A} \in [0, 1]^{B \times T \times T}$. Attention weights define the strengths of connections (relations) among tokens. In some way, we can think of $\mathbf{A}$ as a soft adjacency matrix of a graph that determines edges and their strengths. Thus my claim before about a possible advantage of self-attention over convolutional layers because they can learn long-range dependencies without stacking multiple layers. We can define attention weights in the following manner:

$$\mathbf{A} = \text{softmax}(\mathbf{Q} \otimes \mathbf{K}^\top), \tag{3.17}$$

where softmax is calculated over the last dimension (i.e., 2), and the transposition is applied to the first dimension and the second dimension (in PyTorch: `X.transpose(1, 2)`). Additionally, to counteract possible large magnitudes in calculating the matrix multiplication, the argument of the softmax is scaled by the square root of the dimension $D$, namely:

$$\mathbf{A} = \text{softmax}\left(\frac{\mathbf{Q} \otimes \mathbf{K}^\top}{\sqrt{D}}\right). \tag{3.18}$$

Lastly, to get the outputs of the self-attention, we have to multiply values and attention weights (scaling is used almost always by default):

$$\mathbf{Y} = \text{softmax}\left(\frac{\mathbf{Q} \otimes \mathbf{K}^\top}{\sqrt{D}}\right) \otimes \mathbf{V}, \tag{3.19}$$

and $\mathbf{Y}$ is of size $B \times T \times D$. If you do not see it, please go through all the steps again. The whole procedure is schematically presented in Fig. 3.7.

At this point, we know how self-attention works! Moreover, in Fig. 3.1, we have a diagram of all operations, and it is basically ready to be translated to code! The only missing piece of the puzzle is multiple heads. How do we define it? Well, it is

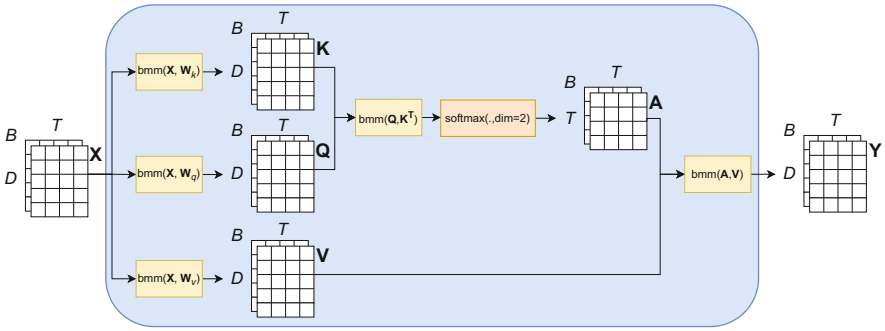



**Fig. 3.7** A schematic representation of operations in self-attention.

really simple: We define self-attention $H$-times, $\mathbf{Y}_h = \text{softmax}\left(\frac{\mathbf{Q}_h \otimes \mathbf{K}_h^\top}{\sqrt{D}}\right) \otimes \mathbf{V}$, where index $h$ indicates a separate set of weights and then concatenate them along the first dimension, $\mathbf{Y} = \mathbf{Y}_1 \oplus \ldots \oplus \mathbf{Y}_H$, where $\oplus$ denotes the concatenation that gives $\mathbf{Y} \in \mathbb{R}^{B \times T \times H \cdot D}$. Eventually, we multiply $\mathbf{Y}$ with a matrix of weights $\mathbf{W}_c \in \mathbb{R}^{H \cdot D \times D}$ that yields the output of multi-head self-attention:

$$\mathbf{M} = \mathbf{Y} \otimes \mathbf{W}_c .$$

Now we are really done! It was not so difficult, wasn't it? It is all about linear layers! So one may ask what is the whole fuzz about? Again, the brilliancy of self-attention comes from the fact that we learn the soft adjacency matrix $\mathbf{A}$. It allows the model to *figure out* how some quantities are related to each other, and using a single step, it learns long-range dependencies. I like to think of attention as some sort of learning a *knowledge graph*. I know, it is far-fetched, but if you think about it, a single multi-head self-attention determines patterns in input data (each head is a set of concepts), and then, by stacking multiple multi-head self-attentions, we obtain high-level patterns in data. You must say, my curious reader, that it is fantastic in its simplicity!

#### 3.3.2.2 Toward Implementing Transformers

Multi-head Self-Attention

The way we presented self-attention was very close to the code. However, to implement self-attention, we can use a few simple trick to speed up computations. The most important one is about calculating keys, queries, and values. We do not need to define separate single-head self-attentions; instead, we can define linear layers that output $D \times H$ outputs, and then we can just reshape tensors to $B \times T \times H \times D$ for batch matrix multiplication. The full code is presented in Listing 3.4.

```python
class MultiHeadSelfAttention(nn.Module):
    def __init__(self, num_emb, num_heads=8):
        super().__init__()

        # hyperparams
        self.D = num_emb
        self.H = num_heads

        # weights for self-attention
        self.w_k = nn.Linear(self.D, self.D * self.H)
        self.w_q = nn.Linear(self.D, self.D * self.H)
        self.w_v = nn.Linear(self.D, self.D * self.H)

        # weights for a combination of multiple heads
        self.w_c = nn.Linear(self.D * self.H, self.D)

    def forward(self, x, causal=True):
        # x: B(atch) x T(okens) x D(imensionality)
        B, T, D = x.size()

        # keys, queries, values
        k = self.w_k(x).view(B, T, self.H, D) # B x T x H x D
        q = self.w_q(x).view(B, T, self.H, D) # B x T x H x D
        v = self.w_v(x).view(B, T, self.H, D) # B x T x H x D

        k = k.transpose(1, 2).contiguous().view(B * self.H, T, D)
     # B*H x T x D
        q = q.transpose(1, 2).contiguous().view(B * self.H, T, D)
     # B*H x T x D
        v = v.transpose(1, 2).contiguous().view(B * self.H, T, D)
     # B*H x T x D

        k = k / (D**0.25) # scaling
        q = q / (D**0.25) # scaling

        # kq
        kq = torch.bmm(q, k.transpose(1, 2)) # B*H x T x T

        # if causal
        if causal:
            mask = torch.triu_indices(T, T, offset=1)
            kq[..., mask[0], mask[1]] = float('-inf')

        # softmax
        skq = F.softmax(kq, dim=2)

        # self-attention
        sa = torch.bmm(skq, v) # B*H x T x D
        sa = sa.view(B, self.H, T, D) # B x H x T x D
        sa = sa.transpose(1, 2) # B x T x H x D
        sa = sa.contiguous().view(B, T, D * self.H) # B x T x D*H


        out = self.w_c(sa) # B x T x D
```

```
52
53         return out
```

**Listing 3.4** Multi-head self-attention.

### Causality

Is everything clear? It should be! But wait, what is this flag `causal`? Yes, you are right, my careful reader; we did not discuss it at all, but after all, we want to use self-attention as a replacement for **causal** convolutional layers! To be able to parameterize ARMs with self-attention, we need to ensure that we do not look into the future. Fortunately, it is relatively simple to achieve self-attention, namely, we need to **mask out** the multiplication of keys and queries to ensure causal calculations. For instance, assuming that the first token is *dummy* (i.e., it corresponds to the Beginning Of Sequence, `BOS`), we mask values above the diagonal of the product of keys and queries. Then we ensure that there is no *leakage* of future tokens in the attention weights.

### Positional Encodings

Another point that you probably noticed, my bright reader, is that if we want to apply self-attention to sequences, we must ensure that we are not permutation equivariant. Yes, self-attention is permutation equivariant which means that permuting input tokens results in permuting (in the same way) output tokens. To prevent this, typically position information is added to $\mathbf{X}$. The simplest approach is to encode absolute positions using an embedding layer (i.e., each integer is represented by a real-valued vector). This **positional encoding**, $\mathbf{P}$, is then added to inputs, namely, $\mathbf{X} + \mathbf{P}$. Now, by adding $\mathbf{P}$, we destroy permutation equivariance.

### A Transformer Block: Putting It All Together

As we can imagine, transformers like any other deep learning architectures consist of simpler building components called **transformer blocks**. A single transformer block consists of the following operations:

**Transformer Block**

- A **multi-head self-attention**: $f_{mhsa}(\mathbf{X}) = \mathbf{Y} \otimes \mathbf{W}_c$, where $\mathbf{Y} = \mathbf{Y}_1 \oplus \ldots \oplus \mathbf{Y}_H$ and $\mathbf{Y}_h = \mathbf{V}_h \otimes \text{softmax}\left(\frac{\mathbf{Q}_h \otimes \mathbf{K}_h^\top}{\sqrt{D}}\right)$.
- **Layer Normalization**: $f_{ln}(\mathbf{X}) = \gamma \frac{\mathbf{X}-\mathbf{m}}{\mathbf{s}+\epsilon} + \delta$, where all parameters, i.e., $\mathbf{m}, \mathbf{s}, \gamma, \delta$ are calculated per layer (not batch!), and $\epsilon > 0$ is to ensure numerical stability.
- **MLP** (an example architecture we used in the code later on): $f_{mlp}(\mathbf{X}) = linear(GELU(linear(\mathbf{X})))$.

A single transformer block calculates output tokens in the following manner:

$$1.\ \mathbf{M} = f_{mhsa}(\mathbf{X}) \tag{3.20}$$
$$2.\ \mathbf{U} = f_{ln1}(\mathbf{X} + \mathbf{M}) \tag{3.21}$$
$$3.\ \mathbf{Z} = f_{mlp}(\mathbf{U}) \tag{3.22}$$
$$4.\ \mathbf{X} = f_{ln2}(\mathbf{Z} + \mathbf{U}) \tag{3.23}$$

The code for an example of a transformer block is presented in Listing 3.5.

```
class TransformerBlock(nn.Module):
    def __init__(self, num_emb, num_neurons, num_heads=4):
        super().__init__()

        # hyperparams
        self.D = num_emb
        self.H = num_heads
        self.neurons = num_neurons

        # components
        self.msha = MultiHeadSelfAttention(num_emb=self.D,
    num_heads=self.H)
        self.layer_norm1 = nn.LayerNorm(self.D)
        self.layer_norm2 = nn.LayerNorm(self.D)

        self.mlp = nn.Sequential(nn.Linear(self.D, self.neurons *
     self.D),
                                 nn.GELU(),
                                 nn.Linear(self.neurons * self.D,
    self.D))

    def forward(self, x, causal=True):
        # Multi-Head Self-Attention
        x_attn = self.msha(x, causal)
        # LayerNorm
        x = self.layer_norm1(x_attn + x)
        # MLP
        x_mlp = self.mlp(x)
        # LayerNorm
```

```
        x = self.layer_norm2(x_mlp + x)

        return x
```

**Listing 3.5** Transformer block.

The rationale behind using **Layer Normalization** is twofold. First, it parallelizes better than batch normalization, and it is typically preferred in NLP applications. Second, as discussed in-depth in [26], using layer normalization plays a crucial role in controlling gradient scales. As a result, layer normalization is essential for proper training of transformers.

#### 3.3.2.3 Implementing ARMs with Transformers

Initial Remarks

Now, having a piece of code for the transformer block, we are ready to put all the pieces together and define a transformer architecture. In the code below, we use the following:

- We assume that inputs (i.e., tokens) are integers (but the order plays no role), and the first token is "dummy" (i.e., it only indicates the beginning of a sequence).
- We use an embedding layer to turn input integers into real-valued vectors. We use another embedding layer for positional encoding. Both embedding layers are learnable.
- Before applying transformer blocks, we use a dropout layer with the dropping probability equal to 0.1. This forces the model to cope with missing tokens. In many transformer architectures, dropout is used in all transformer blocks. To keep our code simple, we do not follow this trend.
- We assume that tokens are represented by integers, but please keep in mind that the order is not important. As a result, we use categorical distribution to model all conditionals $p(x_t|\mathbf{x}_{<t})$. As a result, our objective is a sum of conditional likelihoods defined by categorical distributions.
- Since the first token is "dummy," we need to remember to shift the outputs of the transformer and disregard the first input for calculating the loss function.

Sampling

We must remember that we use transformers here to parameterize ARMs. As a result, the sampling procedure is very similar to ARMs with causal convolutions, namely, we have a for-loop that starts with a sequence with the "dummy" token and samples one token at a time.

### Transformer ARM

You can find the full code for a transformer-based ARM in Listing 3.6.

```
class Transformer(nn.Module):
    def __init__(self, num_tokens, num_token_vals, num_emb,
    num_neurons, num_heads=2, dropout_prob=0.1, num_blocks=10,
    device='cpu'):
        super().__init__()

        # Remember, always credit the author, even if it's you ;)
        print('Transformer by JT.')

        # hyperparams
        self.device = device
        self.num_tokens = num_tokens
        self.num_token_vals = num_token_vals
        self.num_emb = num_emb
        self.num_blocks = num_blocks

        # embedding layer
        self.embedding = torch.nn.Embedding(num_token_vals,
    num_emb)

        # positional embedding
        self.positional_embedding = nn.Embedding(num_tokens,
    num_emb)

        # transformer blocks
        self.transformer_blocks = nn.ModuleList()
        for _ in range(num_blocks):
            self.transformer_blocks.append(TransformerBlock(
    num_emb=num_emb, num_neurons=num_neurons, num_heads=num_heads
    ))

        # output layer (logits + softmax)
        self.logits = nn.Sequential(nn.Linear(num_emb,
    num_token_vals))

        # dropout layer
        self.dropout = nn.Dropout(dropout_prob)

        # loss function
        self.loss_fun = LossFun()

    def transformer_forward(self, x, causal=True, temperature
    =1.0):
        # x: B(atch) x T(okens)
        # embedding of tokens
        x = self.embedding(x) # B x T x D
        # embedding of positions
        pos = torch.arange(0, x.shape[1], dtype=torch.long).
    unsqueeze(0).to(self.device)
        pos_emb = self.positional_embedding(pos)
```

```
        # dropout of embedding of inputs
        x = self.dropout(x + pos_emb)

        # transformer blocks
        for i in range(self.num_blocks):
            x = self.transformer_blocks[i](x)

        # output logits
        out = self.logits(x)

        return F.log_softmax(out/temperature, 2)

    @torch.no_grad()
    def sample(self, batch_size=4, temperature=1.0):
        x_seq = np.asarray([[self.num_token_vals - 1] for i in
    range(batch_size)])

        # sample next tokens
        for i in range(self.num_tokens-1):
            xx = torch.tensor(x_seq, dtype=torch.long, device=
    self.device)
            # process x and calculate log_softmax
            x_log_probs = self.transformer_forward(xx,
    temperature=temperature)
            # sample i-th tokens
            x_i_sample = torch.multinomial(torch.exp(x_log_probs
    [:,i]), 1).to(self.device)
            # update the batch with new samples
            x_seq = np.concatenate((x_seq, x_i_sample.to('cpu').
    detach().numpy()), 1)

        return x_seq

    @torch.no_grad()
    def top1_rec(self, x, causal=True):
        x_prob = torch.exp(self.transformer_forward(x, causal=
    True))[:,:-1,:].contiguous()
        _, x_rec_max = torch.max(x_prob, dim=2)
        return torch.sum(torch.mean((x_rec_max.float() == x
    [:,1:].float().to(device)).float(), 1).float())

    def forward(self, x, causal=True, temperature=1.0, reduction=
    'mean'):
        # get log-probabilities
        log_prob = self.transformer_forward(x, causal=causal,
    temperature=temperature)

        return self.loss_fun(log_prob[:,:-1].contiguous(), x
    [:,1:].contiguous(), reduction=reduction)
```

**Listing 3.6** Transformer-based ARM.

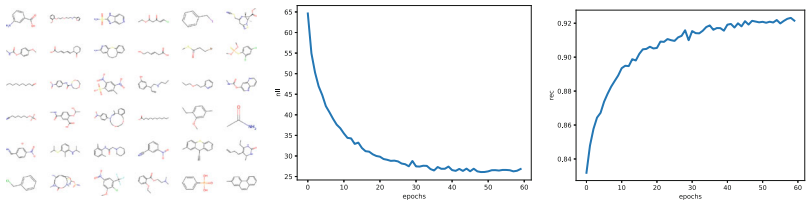



**Fig. 3.8** Examples of results after running the code: (**a**) An example of a sample from a trained transformer. (**b**) The negative log-likelihood. (**c**) The reconstruction accuracy.

Results

In the experiments, we consider the task of learning a generative model for molecules. Interestingly, molecules could be represented as strings (a sequence of atoms and additional characters like brackets, among others) called SMILES [27]. After using a specialized tokenizer (e.g., see DeepChem [28]), we can represent molecules as a sequence of integers, where each integer encodes either an atom or a character needed to represent a structure. The process of representing molecules as SMILES is reversible (i.e., for given SMILEs, we can recover a graphical representation), and, similarly, encoding SMILES using integers is also invertible. Moreover, we used a simple dataset of about 7.4K molecules called `Tox21` that we divided into a training set (6.6k molecules), a validation set (400 molecules), and a test set (a bit more than 400 molecules). We can monitor multiple metrics, but here we focused on two: the value of the negative log-likelihood (nll) and the reconstruction accuracy calculated by taking the most probable outputs of the transformer instead of a sample (rec).

After running a relatively simple transformer architecture (about 1M weights), you can expect generations similar to those in Fig. 3.8a, and validation nll and validation rec in Fig. 3.8b and c, respectively.

### 3.3.2.4 Transformers Constitute Their Own Field (Almost)

Nowadays, transformers are behind many, many successes of generative AI! Probably you heard about GPT (generative pre-trained transformer) [29] and ChatGPT; yes, they all use transformers. I think it is safe to say that the boom for generative AI is due to GPT-based models (among others, but they contributed a lot to that!). Since the success of transformers, many researchers have used a single term for deep neural networks (mainly transformer-based architectures) in NLP: **Large Language Models** (LLMs). Since LLMs (or other **big** transformers) are typically first pretrained, and then fine-tuned on a downstream task or specific data, they are called **foundation models**.

Let me indicate only a few, really a tiny fraction of interesting reads, because otherwise the list could be almost endless:

- I highly recommend reading a chapter on transformers in [30]. This fantastic book has an amazing introduction to transformers. You can find a plethora of extensions and applications of transformers therein.
- Here, we focused on ARMs; however, transformers could be used for processing sequences and then for a downstream task (e.g., a property prediction or classification). One of the most famous LLMs is Bidirectional Encoder Representations from Transformers or BERT for short [31]. It does not use causal masks and learns a conditional distribution $p(\mathbf{x}'|\mathbf{x})$. It corresponds to an auto-encoding kind of learning in which given tokens are reconstructed first (*pretraining phase*), and then, using the first output token corresponding to the "dummy" input token, a predictor is trained (*finetuning phase*). To avoid potential overfitting, dropout is applied to input tokens.
- There are three general groups of transformers: (i) *Decoders*, they are ARMs that use causal masks, e.g., GPT; (ii) *Encoders*, they do not use causal masks, e.g., BERT; and (iii) *Encoder-decoder*, a conditional ARM that uses cross-attention instead of self-attention in which queries come from conditioning processed by an encoder-transformer. Encoder-decoders are widely used in machine translation.
- Interestingly, in the NLP task, it is possible to play with prompts (i.e., input texts) to obtain specific outputs. For instance, as discussed in [32], LLMs are capable of zero-shot learning by presenting new examples in a prompt. This leads to an interesting research direction that focuses on modifying prompts for controlling LLMs behavior to get a specific outcome without updating the model weights. You can read a great overview provided by [33].
- For people interested in molecule generation, I highly suggest getting familiar with the following paper (and code) on drug discovery [34]. Then, you can look up some papers that use transformers for molecule generation [35] and downstream tasks including property prediction and de novo drug design [36].
- Transformers are also used for image processing. In a nutshell, an image is divided into small patches (e.g., 16-by-16 pixels) and then processed in a similar manner to processing sequences. The resulting approach is dubbed visual transformer or ViT for short [37].

As promised, I keep the list very short on purpose, my curious reader. I hope you got excited about transformers and their capabilities (here, using less than a thousand lines of code, we were able to generate new molecules!), and with only these pointers, you are ready to delve into a fantastic world of transformers!

## References

1. Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.
2. Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

3. Ilya Sutskever, James Martens, and Geoffrey E Hinton. Generating text with recurrent neural networks. In *ICML*, 2011.
4. Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318. PMLR, 2013.
5. Martin Arjovsky, Amar Shah, and Yoshua Bengio. Unitary evolution recurrent neural networks. In *International Conference on Machine Learning*, pages 1120–1128. PMLR, 2016.
6. Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167, 2008.
7. Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. A convolutional neural network for modelling sentences. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*, pages 212–217. Association for Computational Linguistics, 2014.
8. Shaojie Bai, J Zico Kolter, and Vladlen Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *arXiv preprint arXiv:1803.01271*, 2018.
9. Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.
10. Aaron Van Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel recurrent neural networks. In *International Conference on Machine Learning*, pages 1747–1756. PMLR, 2016.
11. Auke Wiggers and Emiel Hoogeboom. Predictive sampling with forecasting autoregressive models. In *International Conference on Machine Learning*, pages 10260–10269. PMLR, 2020.
12. Aäron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, and Koray Kavukcuoglu. Conditional image generation with pixelcnn decoders. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, pages 4797–4805, 2016.
13. Tim Salimans, Andrej Karpathy, Xi Chen, and Diederik P Kingma. Pixelcnn++: Improving the pixelcnn with discretized logistic mixture likelihood and other modifications. *arXiv preprint arXiv:1701.05517*, 2017.
14. Xi Chen, Nikhil Mishra, Mostafa Rohaninejad, and Pieter Abbeel. Pixelsnail: An improved autoregressive generative model. In *International Conference on Machine Learning*, pages 864–872. PMLR, 2018.
15. Amirhossein Habibian, Ties van Rozendaal, Jakub M Tomczak, and Taco S Cohen. Video compression with rate-distortion autoencoders. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 7033–7042, 2019.
16. Nal Kalchbrenner, Aäron Oord, Karen Simonyan, Ivo Danihelka, Oriol Vinyals, Alex Graves, and Koray Kavukcuoglu. Video pixel networks. In *International Conference on Machine Learning*, pages 1771–1779. PMLR, 2017.

17. Ishaan Gulrajani, Kundan Kumar, Faruk Ahmed, Adrien Ali Taiga, Francesco Visin, David Vazquez, and Aaron Courville. PixelVAE: A latent variable model for natural images. *arXiv preprint arXiv:1611.05013*, 2016.
18. Yang Song, Chenlin Meng, Renjie Liao, and Stefano Ermon. Accelerating feedforward computation via parallel nonlinear equation solving. In *International Conference on Machine Learning*, pages 9791–9800. PMLR, 2021.
19. Georg Ostrovski, Will Dabney, and Rémi Munos. Autoregressive quantile networks for generative modeling. In *International Conference on Machine Learning*, pages 3936–3945. PMLR, 2018.
20. Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.
21. Scott Reed, Aäron Oord, Nal Kalchbrenner, Sergio Gómez Colmenarejo, Ziyu Wang, Yutian Chen, Dan Belov, and Nando Freitas. Parallel multiscale autoregressive density estimation. In *International Conference on Machine Learning*, pages 2912–2921. PMLR, 2017.
22. Jacob Menick and Nal Kalchbrenner. Generating high fidelity images with subscale pixel networks and multidimensional upscaling. In *International Conference on Learning Representations*, 2018.
23. Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.
24. Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.
25. Maximilian Ilse, Jakub Tomczak, and Max Welling. Attention-based deep multiple instance learning. In *International conference on machine learning*, pages 2127–2136. PMLR, 2018.
26. Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tieyan Liu. On layer normalization in the transformer architecture. In *International Conference on Machine Learning*, pages 10524–10533. PMLR, 2020.
27. David Weininger. Smiles, a chemical language and information system. 1. introduction to methodology and encoding rules. *Journal of Chemical Information and Computer Sciences*, 28(1):31–36, 1988.
28. Bharath Ramsundar, Peter Eastman, Patrick Walters, Vijay Pande, Karl Leswing, and Zhenqin Wu. *Deep Learning for the Life Sciences*. O'Reilly Media, 2019. https://www.amazon.com/Deep-Learning-Life-Sciences-Microscopy/dp/1492039837.
29. Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.
30. Simon JD Prince. *Understanding Deep Learning*. MIT press, 2023.

31. Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.
32. Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213, 2022.
33. Lilian Weng. Prompt engineering. *lilianweng.github.io*, Mar 2023.
34. Nathan Brown, Marco Fiscato, Marwin HS Segler, and Alain C Vaucher. Guacamol: benchmarking models for de novo molecular design. *Journal of chemical information and modeling*, 59(3):1096–1108, 2019.
35. Viraj Bagal, Rishal Aggarwal, PK Vinod, and U Deva Priyakumar. Molgpt: molecular generation using a transformer-decoder model. *Journal of Chemical Information and Modeling*, 62(9):2064–2076, 2021.
36. Adam Izdebski, Ewelina Weglarz-Tomczak, Ewa Szczurek, and Jakub M Tomczak. De novo drug design with joint transformers. *arXiv preprint arXiv:2310.02066*, 2023.
37. Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16*times*16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

# Chapter 4
# Flow-Based Models


## 4.1 Flows for Continuous Random Variables

### 4.1.1 Introduction

So far, we have discussed a class of deep generative models that model the distribution $p(\mathbf{x})$ directly in an autoregressive manner. The main advantage of ARMs is that they can learn long-range statistics and, as a consequence, powerful density estimators. However, their drawback is that they are parameterized in an autoregressive manner; hence, sampling is rather a slow process. Moreover, they lack a latent representation; therefore, it is not obvious how to manipulate their internal data representation which makes it less appealing for tasks like compression or metric learning. In this chapter, we present a different approach to direct modeling of $p(\mathbf{x})$. However, before we start our considerations, we will discuss a simple example.

**Example**

Let us take a random variable $z \in \mathbb{R}$ with $\pi(z) = \mathcal{N}(z|0, 1)$. Now, we consider a new random variable after applying some linear transformation to $z$, namely, $x = 0.75z + 1$. Now the question is the following:

*What is the distribution of $x$, $p(x)$?*

We can guess the solution by using properties of Gaussians, or dig in our memory about the **change of variables formula** to calculate this distribution, that is:

$$p(x) = \pi\left(z = f^{-1}(x)\right)\left|\frac{\partial f^{-1}(x)}{\partial x}\right|, \tag{4.1}$$

where $f$ is an invertible function (a bijection). What does it mean? It means that the function maps one point to another, distinctive point, and we can always invert the function to obtain the original point.



J. M. Tomczak, *Deep Generative Modeling*,
https://doi.org/10.1007/978-3-031-64087-2_4

In Fig. 4.1, we have an example of a bijection. Notice that the volumes of the domains do not need to be the same! Keep it in mind and think about it in the context of $\left|\frac{\partial f^{-1}(x)}{\partial x}\right|$.

Coming back to our example, we have:

$$f(z) = 0.75z + 1, \tag{4.2}$$

and the inverse of $f$ is:

$$f^{-1}(x) = \frac{x-1}{0.75}. \tag{4.3}$$

Then, the derivative of the **change of volume** is:

$$\left|\frac{\partial f^{-1}(x)}{\partial x}\right| = \frac{4}{3}. \tag{4.4}$$

Putting all information so far together yields:

$$p(x) = \pi\left(z = \frac{x-1}{0.75}\right)\frac{4}{3} = \frac{1}{\sqrt{2\pi\, 0.75^2}} \exp\left\{-(x-1)^2/0.75^2\right\}. \tag{4.5}$$

We immediately realize that we end up with the Gaussian distribution again:

$$p(x) = \mathcal{N}(x|1, 0.75). \tag{4.6}$$

Moreover, we see that the part $\left|\frac{\partial f^{-1}(x)}{\partial x}\right|$ is responsible to **normalize** the distribution $\pi(z)$ after applying the transformation $f$. In other words, $\left|\frac{\partial f^{-1}(x)}{\partial x}\right|$ counteracts a possible *change of volume* caused by $f$.

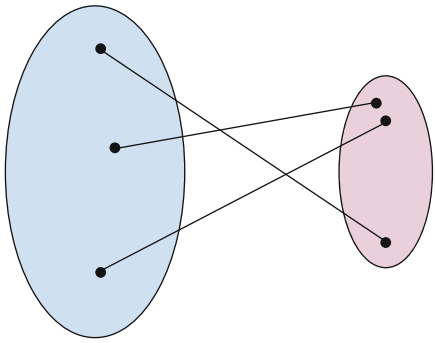


**Fig. 4.1** An example of a bijection where for each point in the blue set there is precisely one corresponding point in the purple set (and *vice versa*).

First of all, this example indicates that we can calculate a new distribution of a continuous random variable by applying a known bijective transformation $f$ to a random variable with a known distribution, $z \sim p(z)$. The same holds for multiple variables $\mathbf{x}, \mathbf{z} \in \mathbb{R}^D$:

$$p(\mathbf{x}) = p\left(\mathbf{z} = f^{-1}(\mathbf{x})\right) \left| \frac{\partial f^{-1}(\mathbf{x})}{\partial \mathbf{x}} \right|, \tag{4.7}$$

where:

$$\left| \frac{\partial f^{-1}(\mathbf{x})}{\partial \mathbf{x}} \right| = \left| \det \mathbf{J}_{f^{-1}}(\mathbf{x}) \right| \tag{4.8}$$

is the Jacobian matrix $\mathbf{J}_{f^{-1}}$ that is defined as follows:

$$\mathbf{J}_{f^{-1}} = \begin{bmatrix} \frac{\partial f_1^{-1}}{\partial x_1} & \cdots & \frac{\partial f_1^{-1}}{\partial x_D} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_D^{-1}}{\partial x_1} & \cdots & \frac{\partial f_D^{-1}}{\partial x_D} \end{bmatrix}. \tag{4.9}$$

Moreover, we can also use the **inverse function theorem** that yields:

$$\left| \mathbf{J}_{f^{-1}}(\mathbf{x}) \right| = \left| \mathbf{J}_f(\mathbf{z}) \right|^{-1}. \tag{4.10}$$

Since $f$ is invertible, we can use the inverse function theorem to rewrite (4.7) as follows:

$$p(\mathbf{x}) = p\left(\mathbf{z} = f^{-1}(\mathbf{x})\right) \left| \mathbf{J}_f(\mathbf{z}) \right|^{-1}. \tag{4.11}$$

To get some insight into the role of the Jacobian-determinant, take a look at Fig. 4.2. Here, there are three cases of invertible transformations that play around with a uniform distribution defined over a square.

In the case on top, the transformation turns a square into a rhombus without changing its volume. As a result, the Jacobian-determinant of this transformation is 1. Such transformations are called **volume-preserving**. Notice that the resulting distribution is still uniform and since there is no change of volume, it is defined over the same volume as the original one, thus, the color is the same.

In the middle, the transformation shrinks the volume, therefore, the resulting uniform distribution is "denser" (a darker color in Fig. 4.2). Additionally, the Jacobian-determinant is smaller than 1.

In the last situation, the transformation enlarges the volume, hence, the uniform distribution is defined over a larger area (a lighter color in Fig. 4.2). Since the volume is larger, the Jacobian-determinant is larger than 1.

Notice that the shifting operator is volume-preserving. To see that imagine adding an arbitrary value (e.g., 5) to all points of the square. Does it change the volume? Not at all! Thus, the Jacobian-determinant equals 1.

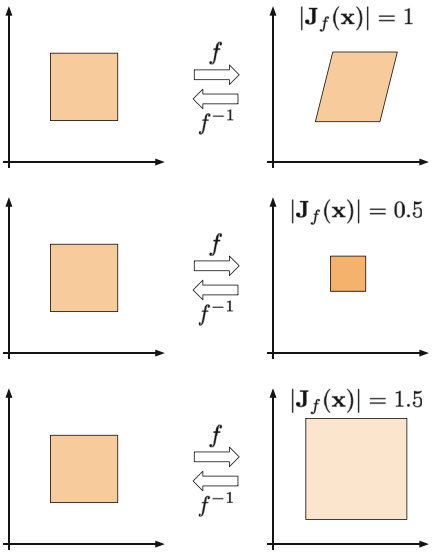



**Fig. 4.2** Three examples of invertible transformations: (*top*) a volume-preserving bijection, (*middle*) a bijection that shrinks the original area, (*bottom*) a bijection that enlarges the original area.

### 4.1.2 Change of Variables for Deep Generative Modeling

A natural question is whether we can utilize the idea of the change of variables to model a complex and high-dimensional distribution over images, audio, or other data sources. Let us consider a hierarchical model, or, equivalently, a sequence of invertible transformations, $f_k : \mathbb{R}^D \rightarrow \mathbb{R}^D$. We start with a known distribution $\pi(\mathbf{z}_0) = \mathcal{N}(\mathbf{z}_0|0, \mathbf{I})$. Then, we can sequentially apply the invertible transformations to obtain a flexible distribution [1, 2]:

$$p(\mathbf{x}) = \pi\left(\mathbf{z}_0 = f^{-1}(\mathbf{x})\right) \prod_{i=1}^{K} \left|\det \frac{\partial f_i\left(\mathbf{z}_{i-1}\right)}{\partial \mathbf{z}_{i-1}}\right|^{-1}, \tag{4.12}$$

or by using the notation of a Jacobian for the $i$-th transformation:

$$p(\mathbf{x}) = \pi\left(\mathbf{z}_0 = f^{-1}(\mathbf{x})\right) \prod_{i=1}^{K} \left|\mathbf{J}_{f_i}(\mathbf{z}_{i-1})\right|^{-1}. \tag{4.13}$$

An example of transforming a unimodal base distribution like Gaussian into a multimodal distribution through invertible transformations is presented in Fig. 4.3. In principle, we should be able to get almost any arbitrarily complex distribution and revert to a *simple* one.

Let $\pi(\mathbf{z}_0)$ be $\mathcal{N}(\mathbf{z}_0|0, \mathbf{I})$. Then, the logarithm of $p(\mathbf{x})$ is the following:

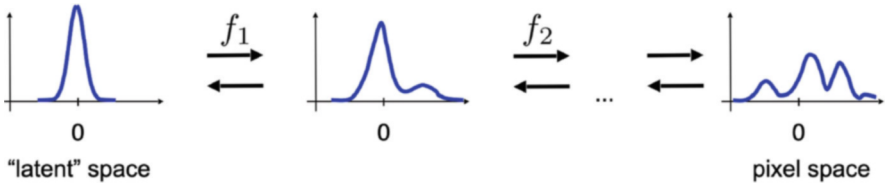



**Fig. 4.3** An example of transforming a unimodal distribution (the latent space) to a multimodal distribution (the data space, e.g., the pixel space) through a series of invertible transformations $f_i$.

$$\ln p(\mathbf{x}) = \ln \mathcal{N}\left(\mathbf{z}_0 = f^{-1}(\mathbf{x})|0, \mathbf{I}\right) - \sum_{i=1}^{K} \ln \left|\mathbf{J}_{f_i}(\mathbf{z}_{i-1})\right|. \tag{4.14}$$

Interestingly, we see that the first part, namely, $\ln \mathcal{N}\left(\mathbf{z}_0 = f^{-1}(\mathbf{x})|0, \mathbf{I}\right)$ corresponds to the *Mean Square Error* loss function between 0 and $f^{-1}(\mathbf{x})$ plus a constant. The second part, $\sum_{i=1}^{K} \ln \left|\mathbf{J}_{f_i}(\mathbf{z}_{i-1})\right|$, as in our example, ensures that the distribution is properly normalized. However, since it penalizes the change of volume (take a look again at the example above!), we can think of it as a kind of a *regularizer* for the invertible transformations $\{f_i\}$.

Once we have laid down the foundations of the change of variables for expressing density functions, now we must face two questions:

- How to model the invertible transformations?
- What is the difficulty here?

The answer to the first question could be neural networks, because they are flexible and easy to train. However, we cannot take **any** neural network because of two reasons. First, the transformation must be **invertible**; thus, we must pick an **invertible neural network**. Second, even if a neural network is invertible, we face the problem of calculating the second part of (4.14), i.e., $\sum_{i=1}^{K} \ln \left|\mathbf{J}_{f_i}(\mathbf{z}_{i-1})\right|$, that is nontrivial and computationally intractable for an arbitrary sequence of invertible transformations. As a result, we seek for such neural networks that are both invertible, and the logarithm of a Jacobian determinant is (relatively) easy to calculate. The resulting model that consists of invertible transformations (neural networks) with tractable Jacobian determinants is referred to as *normalizing flows* or *flow-based models*.

There are various possible invertible neural networks with tractable Jacobian determinants, e.g., planar normalizing flows [1], Sylvester normalizing flows [3], residual flows [4, 5], and invertible DenseNets [6]. However, here we focus on a very important class of models: **RealNVP**, *Real-valued Non-Volume Preserving* flows [7] that serve as a starting point for many other flow-based generative models (e.g., GLOW [8]).

### 4.1.3 Building Blocks of RealNVP

#### 4.1.3.1 Coupling Layers

The main component of RealNVP is a *coupling layer*. The idea behind this transformation is the following. Let us consider an input to the layer that is divided into two parts: $\mathbf{x} = [\mathbf{x}_a, \mathbf{x}_b]$. The division into two parts could be done by dividing the vector $\mathbf{x}$ into $\mathbf{x}_{1:d}$ and $\mathbf{x}_{d+1:D}$ or according to a more sophisticated manner, e.g., a *checkerboard pattern* [7]. Then, the transformation is defined as follows:

$$\mathbf{y}_a = \mathbf{x}_a \tag{4.15}$$

$$\mathbf{y}_b = \exp\left(s\left(\mathbf{x}_a\right)\right) \odot \mathbf{x}_b + t\left(\mathbf{x}_a\right), \tag{4.16}$$

where $s(\cdot)$ and $t(\cdot)$ are **arbitrary neural networks** called *scaling* and *translation*, respectively.

This transformation is invertible by design, namely:

$$\mathbf{x}_b = (\mathbf{y}_b - t(\mathbf{y}_a)) \odot \exp\left(-s(\mathbf{y}_a)\right) \tag{4.17}$$

$$\mathbf{x}_a = \mathbf{y}_a. \tag{4.18}$$

Importantly, the logarithm of the Jacobian determinant is easy to calculate, because:

$$\mathbf{J} = \begin{bmatrix} \mathbf{I}_{d\times d} & \mathbf{0}_{d\times(D-d)} \\ \frac{\partial \mathbf{y}_b}{\partial \mathbf{x}_a} & \operatorname{diag}\left(\exp\left(s\left(\mathbf{x}_a\right)\right)\right) \end{bmatrix} \tag{4.19}$$

that yields:

$$\det(\mathbf{J}) = \prod_{j=1}^{D-d} \exp\left(s\left(\mathbf{x}_a\right)\right)_j = \exp\left(\sum_{j=1}^{D-d} s\left(\mathbf{x}_a\right)_j\right). \tag{4.20}$$

Eventually, coupling layers seem to be flexible and powerful transformations with tractable Jacobian determinants! However, we process only half of the input; therefore, we must think of an appropriate additional transformation a coupling layer could be combined with.

#### 4.1.3.2 Permutation Layers

A simple yet effective transformation that could be combined with a coupling layer is a **permutation layer**. Since permutation is *volume-preserving*, i.e., its Jacobian determinant is equal to 1, we can apply it each time after the coupling layer. For instance, we can reverse the order of variables.

An example of an invertible block, i.e., a combination of a coupling layer with a permutation layer, is schematically presented in Fig. 4.4.

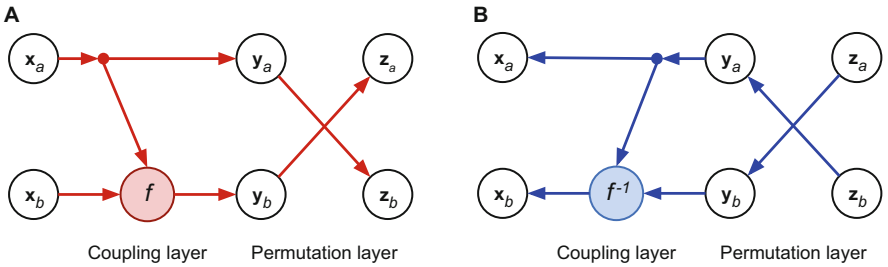



**Fig. 4.4** A combination of a coupling layer and a permutation layer that transforms $[\mathbf{x}_a, \mathbf{x}_b]$ to $[\mathbf{z}_a, \mathbf{z}_b]$. (**a**) A forward pass through the block. (**b**) An inverse pass through the block.

#### 4.1.3.3 Dequantization

As discussed so far, flow-based models assume that $\mathbf{x}$ is a vector of real-valued random variables. However, in practice, many objects are discrete. For instance, images are typically represented as integers taking values in $\{0, 1, \ldots, 255\}^D$. In [9], it has been outlined that adding a uniform noise, $\mathbf{u} \in [-0.5, 0.5]^D$, to original data, $\mathbf{y} \in \{0, 1, \ldots, 255\}^D$, allows applying density estimation to $\mathbf{x} = \mathbf{y} + \mathbf{u}$. This procedure is known as *uniform dequantization*. Recently, there were different schemas of dequantization proposed; you can read more on that in [10].

An example of two binary random variables and the uniform dequantization is depicted in Fig. 4.5. After adding $\mathbf{u} \in [-0.5, 0.5]^2$ to each discrete value, we obtain a continuous space, and now probabilities originally associated with volumeless points are "spread" across small square regions.

### 4.1.4 Flows in Action!

Let us turn math into code! We will first discuss the log-likelihood function (i.e., the learning objective) and how mathematical formulas correspond to the code. First, it is extremely important to know what is our learning objective, i.e., the log-likelihood

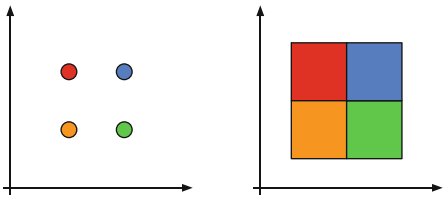


**Fig. 4.5** A schematic representation of the uniform dequantization for two binary random variables: (*left*) the probability mass is assigned to points, (*right*) after the uniform dequantization, the probability mass is assigned to square areas. Colors correspond to probability values.

function. In the example, we use coupling layers as described earlier, together with permutation layers. Then, we can plug the logarithm of the Jacobian determinant for the coupling layers (for the permutation layers, it is equal to 1, so $\ln(1) = 0$) in Eq. 4.14 that yields:

$$\ln p(\mathbf{x}) = \ln \mathcal{N}\left(\mathbf{z}_0 = f^{-1}(\mathbf{x})|0, \mathbf{I}\right) - \sum_{i=1}^{K}\left(\sum_{j=1}^{D-d} s_k\left(\mathbf{x}_a^k\right)_j\right), \tag{4.21}$$

where $s_k$ is the scale network in the $k$-th coupling layer and $\mathbf{x}_a^k$ denotes the input to the $k$-th coupling layer. Notice that exp in the log-Jacobian determinant is canceled by applying the logarithm.

Let us think again about the learning objective from the implementation perspective. First, we definitely need to obtain $\mathbf{z}$ by calculating $f^{-1}(\mathbf{x})$, and then we can calculate $\ln \mathcal{N}\left(\mathbf{z}_0 = f^{-1}(\mathbf{x})|0, \mathbf{I}\right)$. That is actually easy, and we get:

$$\ln \mathcal{N}\left(\mathbf{z}_0 = f^{-1}(\mathbf{x})|0, \mathbf{I}\right) = -const - \frac{1}{2}\|f^{-1}(\mathbf{x})\|^2, \tag{4.22}$$

where $const = \frac{D}{2}\ln(2\pi)$ is the normalizing constant of the standard Gaussian and $\frac{1}{2}\|f^{-1}(\mathbf{x})\|^2 = MSE(0, f^{-1}(\mathbf{x}))$.

Alright, now we should look into the second part of the objective, i.e., the log-Jacobian determinants. As we can see, we have a sum over transformations, and for each coupling layer, we consider only the outputs of the scale nets. Hence, the only thing we must remember during implementing the coupling layers is to return not only output but also the outcome of the scale layer too.

### 4.1.5 Code

Now, we have all the components to implement our own RealNVP! Below, there is a code with a lot of comments that should help to understand every single line of it.

```
class RealNVP(nn.Module):
    def __init__(self, nets, nett, num_flows, prior, D=2,
    dequantization=True):
        super(RealNVP, self).__init__()

        # Well, it's always good to brag about yourself.
        print('RealNVP by JT.')

        # We need to dequantize discrete data. This attribute is
    used during training to dequantize integer data.
        self.dequantization = dequantization

        # An object of a prior (here: torch.distribution of
    multivariate normal distribution)
        self.prior = prior
```

```
        # A module list for translation networks
        self.t = torch.nn.ModuleList([nett() for _ in range(num_flows)])
        # A module list for scale networks
        self.s = torch.nn.ModuleList([nets() for _ in range(num_flows)])
        # The number of transformations, in our equations it is denoted by K.
        self.num_flows = num_flows

        # The dimensionality of the input. It is used for sampling.
        self.D = D

    # This is the coupling layer, the core of the RealNVP model.
    def coupling(self, x, index, forward=True):
        # x: input, either images (for the first transformation) or outputs from the previous transformation
        # index: it determines the index of the transformation
        # forward: whether it is a pass from x to y (forward=True), or from y to x (forward=False)

        # We chunk the input into two parts: x_a, x_b
        (xa, xb) = torch.chunk(x, 2, 1)

        # We calculate s(xa), but without exp!
        s = self.s[index](xa)
        # We calculate t(xa)
        t = self.t[index](xa)

        # Calculate either the forward pass (x -> z) or the inverse pass (z -> x)
        # Note that we use the exp here!
        if forward:
            #yb = f^{-1}(x)
            yb = (xb - t) * torch.exp(-s)
        else:
            #xb = f(y)
            yb = torch.exp(s) * xb + t

        # We return the output y = [ya, yb], but also s for calculating the log-Jacobian-determinant
        return torch.cat((xa, yb), 1), s

    # An implementation of the permutation layer
    def permute(self, x):
        # Simply flip the order.
        return x.flip(1)

    def f(self, x):
        # This is a function that calculates the full forward pass through the coupling+permutation layers.
        # We initialize the log-Jacobian-det
        log_det_J, z = x.new_zeros(x.shape[0]), x
```

```
        # We iterate through all layers
        for i in range(self.num_flows):
            # First, do coupling layer,
            z, s = self.coupling(z, i, forward=True)
            # then permute.
            z = self.permute(z)
            # To calculate the log-Jacobian-determinant of the sequence of transformations we sum over all of them.
            # As a result, we can simply accumulate individual log-Jacobian determinants.
            log_det_J = log_det_J - s.sum(dim=1)
        # We return both z and the log-Jacobian-determinant, because we need z to feed into the logarightm of the Norma;
        return z, log_det_J

    def f_inv(self, z):
        # The inverse path: from z to x.
        # We apply all transformations in the reversed order.
        x = z
        for i in reversed(range(self.num_flows)):
            x = self.permute(x)
            x, _ = self.coupling(x, i, forward=False)
        # Since we use this function for sampling, we don't need to return anything else than x.
        return x

    def forward(self, x, reduction='avg'):
        # This function is essential for PyTorch.
        # First, we calculate the forward part: from x to z, and also we need the log-Jacobian-determinant.
        z, log_det_J = self.f(x)
        # We can use either sum or average as the output.
        # Either way, we calculate the learning objective: self.prior.log_prob(z) + log_det_J.
        # NOTE: Mind the minus sign! We need it, because, by default, we consider the minimization problem,
        # but normally we look for the maximum likelihood estimate. Therefore, we use:
        # max F(x) <=> min -F(x)
        if reduction == 'sum':
            return -(self.prior.log_prob(z) + log_det_J).sum()
        else:
            return -(self.prior.log_prob(z) + log_det_J).mean()

    def sample(self, batchSize):
        # First, we sample from the prior, z ~ p(z) = Normal(z|0,1)
        z = self.prior.sample((batchSize, self.D))
        z = z[:, 0, :]
        # Second, we go from z to x.
        x = self.f_inv(z)
        return x.view(-1, self.D)
```

**Listing 4.1** An example of an implementation of RealNVP.

```
# The number of flows
num_flows = 8

# Neural networks for a single transformation (a single flow).
nets = lambda: nn.Sequential(nn.Linear(D//2, M), nn.LeakyReLU(),
                             nn.Linear(M, M), nn.LeakyReLU(),
                             nn.Linear(M, D//2), nn.Tanh())

nett = lambda: nn.Sequential(nn.Linear(D//2, M), nn.LeakyReLU(),
                             nn.Linear(M, M), nn.LeakyReLU(),
                             nn.Linear(M, D//2))

# For the prior, we can use the built-in PyTorch distribution.
prior = torch.distributions.MultivariateNormal(torch.zeros(D),
    torch.eye(D))

# Init of the RealNVP. Please note that we need to dequantize the
     data (i.e., uniform dequantization).
model = RealNVP(nets, nett, num_flows, prior, D=D, dequantization
    =True)
```

**Listing 4.2** An example of networks.

*Et voila*! Now we are ready to run the full code. After training our RealNVP, we should obtain results resembling those in Fig. 4.6.

### 4.1.6 Is It All? Really?

Yes and no. Yes in the sense it is the minimalistic example of an implementation of the RealNVP. No, because there are many improvements over the instance of the RealNVP presented here, namely:

- *Factoring out* [7]: During the forward pass (from $\mathbf{x}$ to $\mathbf{z}$), we can split the variables and proceed with processing only a subset of them. This could help to parameterize the base distribution by using the outputs of intermediate layers. In other words, we can obtain an autoregressive base distribution.

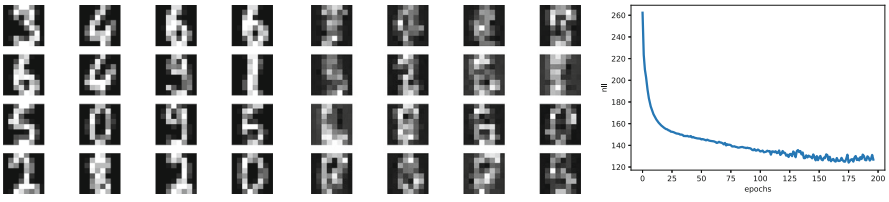



**Fig. 4.6** An example of outcomes after the training: (**a**) Randomly selected real images. (**b**) Unconditional generations from the RealNVP. (**c**) The validation curve during training.

- *Rezero trick* [11]: Introducing additional parameters to the coupling layer, e.g., $\mathbf{y}_b = \exp(\alpha s(\mathbf{x}_a)) \odot \mathbf{x}_b + \beta t(\mathbf{x}_a)$ and $\alpha, \beta$ is initialized with 0's. This helps to ensure that the transformations act as identity maps in the beginning. It is shown in [12] that this trick helps to learn better transformations by maintaining information about the input through all layers in the beginning of the training process.
- *Masking* or *Checkerboard pattern* [7]: We can use a checkerboard pattern instead of dividing an input into two parts like $[\mathbf{x}_{1:D/2}, \mathbf{x}_{D/2+1:D}]$. This encourages learning local statistics better.
- *Squeezing* [7]: We can also play around with "squeezing" some dimensions. For instance, an image consists of $C$ channels, width $W$, and height $H$, which could be turned into $4C$ channels, width $W/2$, and height $H/2$.
- *Learnable base distributions*: instead of using a standard Gaussian base distribution, we can consider another model for that, e.g., an autoregressive model.
- *Invertible 1x1 convolution* [8]: A fixed permutation could be replaced with a (learned) invertible 1x1 convolution as in the GLOW model [8].
- *Variational dequantization* [13]: We can also pick a different dequantization scheme, e.g., variational dequantization. This allows to obtain much better scores. However, it is not for free, because it leads to a lower bound to the log-likelihood function.

Moreover, there are many new fascinating research directions! I will name them here and point to papers where you can find more details:

- *Data compression with flows* [14]: Flow-based models are perfect candidates for compression, since they allow us to calculate the exact likelihood. [14] proposed a scheme that allows using flows in the bit-back-like compression scheme.
- *Conditional flows* [15–17]: Here, we present the unconditional RealNVP. However, we can used a flow-based model for conditional distributions. For instance, we can use the conditioning as an input to the scale network and the translation network.
- *Variational inference with flows* [1, 3, 18–21]: Conditional flow-based models could be used to form a flexible family of variational posteriors. Then, the lower bound to the log-likelihood function could be tighter. We will come back to that in Chap. 5, Sect. 5.4.2.
- *Integer discrete flows* [12, 22, 23]: Another interesting direction is a version of the RealNVP for integer-valued data. We will explain this idea in Sect. 4.2.
- *Flows on manifolds* [24]: Typically, flow-based models are considered in the Euclidean space. However, they could be considered in non-Euclidean spaces, resulting in new properties of (partially) invertible transformations.
- *Flows for ABC* [25]: Approximate Bayesian computation (ABC) assumes that the posterior over quantities of interest is intractable. One possible approach to mitigate this issue is to approximate it using flow-based models, e.g., masked autoregressive flows [26], as presented in [25].

Much other interesting information on flow-based models could be found in a fantastic review by [27].

### 4.1.7 ResNet Flows and DenseNet Flows

#### 4.1.7.1 ResNet Flows [4, 5]

In the previous sections, we discussed flow-based models with predesigned architectures (i.e., blocks consisting of coupling layers and permutation layers) that allow easy calculation of the Jacobian determinant. However, we can take a different approach and think of how we can approximate the Jacobian determinant for an almost arbitrary architecture. And, additionally, what kind of requirements we must impose to make the architecture invertible?

In [4], the authors consider widely used residual neural networks (ResNets) and construct an invertible ResNet layer which is only constrained in Lipschitz continuity. A ResNet is defined as $F(\mathbf{x}) = \mathbf{x} + g(\mathbf{x})$, where $g$ is modeled by a (convolutional) neural network and $F$ represents a ResNet layer which is in general not invertible. However, $g$ is constructed in such a way that it satisfies the Lipschitz constant being strictly lower than 1, Lip(g) < 1, by using spectral normalization of [28, 29]:

$$\text{Lip}(g) < 1, \quad \text{if} \quad ||W_i||_2 < 1, \tag{4.23}$$

where $||\cdot||_2$ is the $\ell_2$ matrix norm. Then $\text{Lip}(g) = \text{K} < 1$ and $\text{Lip}(F) < 1 + \text{K}$. Only in this specific case, the Banach fixed-point theorem holds and ResNet layer $F$ has a unique inverse. As a result, the inverse can be approximated by fixed-point iterations [4].

Estimating the log determinant is, especially for high-dimensional spaces, computationally intractable due to expensive computations. Since ResNet blocks have a constrained Lipschitz constant, the logarithm of the Jacobian determinant is cheaper to compute, tractable, and approximated with guaranteed convergence [4]:

$$\ln p(\mathbf{x}) = \ln p(f(\mathbf{x})) + \text{tr}\left(\sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{k} [J_g(\mathbf{x})]^k\right), \tag{4.24}$$

where $J_g(\mathbf{x})$ is the Jacobian of $g$ at $\mathbf{x}$ that satisfies $||J_g|| < 1$. The Skilling-Hutchinson trace estimator [30, 31] is used to compute the trace at a lower cost than to fully compute the trace of the Jacobian. Residual flows [5] use an improved method to estimate the power series at an even lower cost with an unbiased estimator based on "Russian roulette" of [32]. Intuitively, the method estimates the infinite sum of the power series by evaluating a finite amount of terms. In return, this leads to less computation of terms compared to invertible residual networks. To avoid derivative saturation, which occurs when the second derivative is zero in large regions, the LipSwish activation is proposed [4].

#### 4.1.7.2 DenseNet Flows [6]

Since it is possible to formulate a flow for a ResNet architecture, a natural question is whether it could be accomplished for densely connected networks (DensNets) [33]. In [6], it was shown that indeed it is possible!

The main component of DenseNet flows is a DenseBlock that is defined as a function $F : \mathbb{R}^d \to \mathbb{R}^d$ with $F(\mathbf{x}) = \mathbf{x} + g(\mathbf{x})$, where $g$ consists of dense layers $\{h_i\}_{i=1}^n$. Note that an important modification to make the model invertible is to output $\mathbf{x} + g(\mathbf{x})$ whereas a standard DenseBlock would only output $g(\mathbf{x})$. The function $g$ is expressed as follows:

$$g(\mathbf{x}) = h_{n+1} \circ h_n \circ \cdots \circ h_1(\mathbf{x}), \tag{4.25}$$

where $h_{n+1}$ represents a $1 \times 1$ convolution to match the output size of $\mathbb{R}^d$. A layer $h_i$ consists of two parts concatenated to each other. The upper part is a copy of the input signal. The lower part consists of the transformed input, where the transformation is a multiplication of (convolutional) weights $W_i$ with the input signal, followed by a nonlinearity $\phi$ having $\text{Lip}(\phi) \leq 1$, such as ReLU, ELU, LipSwish, or tanh. As an example, a dense layer $h_2$ can be composed as follows:

$$h_1(x) = \begin{bmatrix} x \\ \phi(W_1 x) \end{bmatrix}, \; h_2(h_1(x)) = \begin{bmatrix} h_1(x) \\ \phi(W_2 h_1(x)) \end{bmatrix}. \tag{4.26}$$

The DenseNet flows [6] rely on the same techniques for approximating the Jacobian determinant as in the ResNet flows. The main difference between the DenseNet flows and the ResNet flows lies in normalizing weights, so that the Lipschitz constant of the transformation is smaller than 1 and, thus, the transformation is invertible. Formally, to satisfy $\text{Lip}(g) < 1$, we need to enforce $\text{Lip}(h_i) < 1$ for all $n$ layers, since $\text{Lip}(g) \leq \text{Lip}(h_{n+1}) \cdot \ldots \cdot \text{Lip}(h_1)$. Therefore, we first need to determine the Lipschitz constant for a dense layer $h_i$. We know that a function $f$ is K-Lipschitz if for all points $v$ and $w$ the following holds :

$$d_Y(f(v), f(w)) \leq \text{K} d_X(v, w), \tag{4.27}$$

where we assume that the distance metrics $d_X = d_Y = d$ are chosen to be the $\ell_2$-norm. Further, let two functions $f_1$ and $f_2$ be concatenated in $h$:

$$h_v = \begin{bmatrix} f_1(v) \\ f_2(v) \end{bmatrix}, \quad h_w = \begin{bmatrix} f_1(w) \\ f_2(w) \end{bmatrix}, \tag{4.28}$$

where function $f_1$ is the upper part and $f_2$ is the lower part. We can now find an analytical form to express a limit on K for the dense layer in the form of Equation (4.27):

$$\begin{aligned} d(h_v, h_w)^2 &= d(f_1(v), f_1(w))^2 + d(f_2(v), f_2(w))^2, \\ d(h_v, h_w)^2 &\leq (\text{K}_1^2 + \text{K}_2^2) d(v, w)^2, \end{aligned} \tag{4.29}$$

where we know that the Lipschitz constant of $h$ consist of two parts, namely, $\text{Lip}(f_1) = \text{K}_1$ and $\text{Lip}(f_2) = \text{K}_2$. Therefore, the Lipschitz constant of layer $h$ can be expressed

as:

$$\text{Lip}(h) = \sqrt{(\text{K}_1^2 + \text{K}_2^2)}. \tag{4.30}$$

With spectral normalization of Equation (4.23), we know that we can enforce (convolutional) weights $W_i$ to be at most 1-Lipschitz. Hence, for all $n$ dense layers, we apply the spectral normalization on the lower part which locally enforces $\text{Lip}(f_2) = \text{K}_2 < 1$. Further, since we enforce each layer $h_i$ to be at most 1-Lipschitz and we start with $h_1$, where $f_1(x) = x$, we know that $\text{Lip}(f_1) = 1$. Therefore, the Lipschitz constant of an entire layer can be at most $\text{Lip}(h) = \sqrt{1^2 + 1^2} = \sqrt{2}$, thus dividing by this limit enforces each layer to be at most 1-Lipschitz. To read more about DenseNet flows and further improvements, please see the original paper [6].

## 4.2 Flows for Discrete Random Variables

### 4.2.1 Introduction

While discussing flow-based models in the previous section, we presented them as *density estimators*, namely, models that represent stochastic dependencies among continuous random variables. We introduced the *change of variables* formula that helps to express a random variable by transforming it using invertible maps (bijections) $f$ to a random variable with a known probability density function. Formally, it is defined as follows:

$$p(\mathbf{x}) = p\left(\mathbf{z} = f^{-1}(\mathbf{x})\right) \left|\mathbf{J}_f(\mathbf{z})\right|^{-1}, \tag{4.31}$$

where $\mathbf{J}_f(\mathbf{z})$ is the Jacobian of $f$ at $\mathbf{z}$.

However, there are potential issues with such an approach. First of all, in many problems (e.g., image processing), the considered random variables (objects) are discrete. For instance, images typically take values in $\{0, 1, \ldots, 255\} \subset \mathbb{Z}$. In order to apply flows, we must apply *dequantization* [10] that results in a lower bound to the original probability distribution.

A continuous space possesses various potential pitfalls. One of them is that if a transformation is a bijection (as in flows), not all continuous deformations are possible. It is tightly connected with *topology* and, more precisely, homeomorphisms, i.e., a continuous function between topological spaces that has a continuous inverse function, and diffeomorphisms, i.e., invertible functions that map one differentiable manifold to another such that both the function and its inverse, are smooth. It is not crucial to know topology, but a curious reader may take a detour and read on that; it is definitely a fascinating field, and I wish to know more about it! Anyway, let us consider three examples.

Imagine we want to transform a square into a circle (Fig. 4.7a). It is possible to find a homeomorphism (i.e., a bijection) that turns the square into a circle and back. Imagine you have a hammer and an iron square. If you start hitting the square

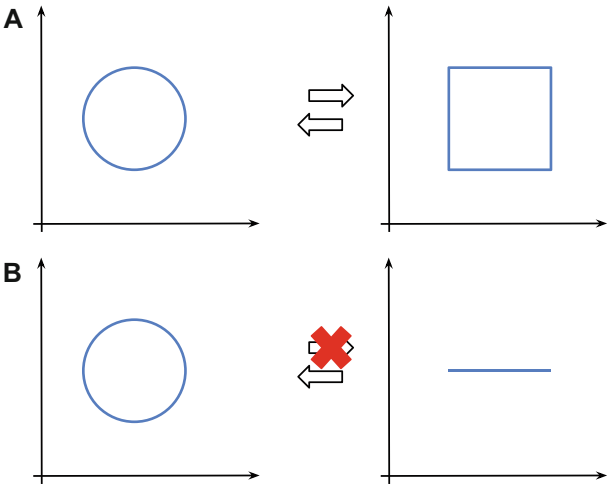



**Fig. 4.7** Examples of: (**a**) homeomorphic spaces, and (**b**) non-homeomorphic spaces. The red cross indicates it is impossible invert the transformation.

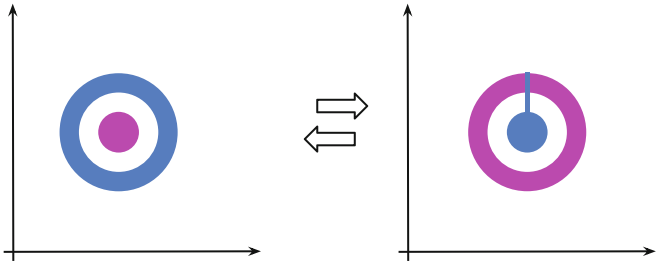


**Fig. 4.8** An example of "replacing" a ring (in blue) with a ball (in magenta).

infinitely many times, you can get an iron circle. Then, you can do it "backward" to get the square back. I know, it is unrealistic but hey, we talking about math here!

However, if we consider a line segment and a circle (Fig. 4.7b), the situation is a bit more complicated. It is possible to transform the line segment into a circle, but not the other way around. Why? Because while transforming the circle to the line segment, it is unclear which point of the circle corresponds to the beginning (or the end) of the line segment. That is why we cannot invert the transformation!

Another example that I really like, and which is closer to the potential issues of continuous flows, is transforming a ring into a ball as in Fig. 4.8. The goal is to replace the blue ring with the magenta ball. In order to make the transformation bijective, while transforming the blue ring in place of the magenta ball, we must ensure that the new magenta "ring" is in fact "broken" so that the new blue "ball" can get inside! Again, why? If the magenta ring is not broken, then we cannot say how the blue ball got inside that destroys bijectivity! In the language of topology, it is impossible because the two spaces are non-homeomorphic.

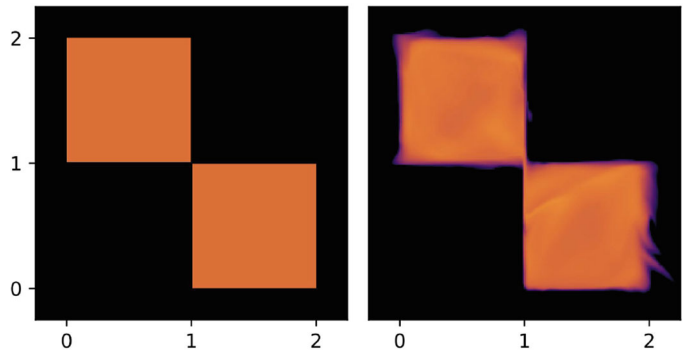



**Fig. 4.9** An example of uniformly dequantized discrete random variables (*left*) and a flow-based model (*right*). Notice that in these examples, the true distribution assigns equal probability mass to the two regions in orange, and zero probability mass to the remaining two regions (in black). However, the flow-based model assigns probability mass outside the original nonzero probability regions.

Alright, but how this affects the flow-based models? I hope that some of you asked this question, or maybe even imagined possible cases where this might hinder learning flows. In general, I would say it is fine, and we should not look for faults where there are none or almost none. However, if you work with flows that require dequantization, then you can spot cases like the one in Fig. 4.9. In this simple example, we have two discrete random variables that after uniform dequantization have two regions with equal probability mass and the remaining two regions with zero probability mass [10]. After training a flow-based model, we have a density estimator that assigns nonzero probability mass where the true distribution has zero density! Moreover, the transformation in the flow must be a bijection; therefore, there is a continuity between the two squares (see Fig. 4.9, right). Where did we see that? Yes, in Fig. 4.8! We must know how to invert the transformation; thus, there must be a "trace" of how the probability mass moves between the regions.

Again, we can ask ourselves if it is bad. Well, I would say not really, but if we think of a case with more random variables, and there is always some little error here and there, this causes a *probability mass leakage* that could result in a far-from-perfect model. And, overall, the model could err in proper probability assignment.

### 4.2.2 Flows in $\mathbb{R}$ or Maybe Rather in $\mathbb{Z}$?

Before we consider any specific cases and discuss discrete flows, first we need to answer whether there is a change of variables formula for discrete random variables. The answer, fortunately, is yes! Let us consider $\mathbf{x} \in \mathcal{X}^D$ where $\mathcal{X}$ is a discrete space, e.g., $\mathcal{X} = \{0, 1\}$ or $\mathcal{X} = \mathbb{Z}$. Then the change of variables takes the following form:

$$p(\mathbf{x}) = \pi\left(\mathbf{z}_0 = f^{-1}(\mathbf{x})\right), \tag{4.32}$$

where $f$ is an invertible transformation and $\pi(\cdot)$ is a base distribution. Immediately we can spot a "missing" Jacobian determinant. This is correct! Why? Because now we live in a discrete world where the probability mass is assigned to points that are "shapeless" and the bijection cannot change the volume. Thus, the Jacobian determinant is always equal to 1! That seems to be good news, isn't it? We can take any bijective transformations, and we do not need to bother about the Jacobian. That is obviously true; however, we need to remember that the output of the transformation must be still discrete, i.e., $z \in \mathcal{X}^D$. As a result, we cannot use any arbitrary invertible neural network. We will discuss it in a minute; however, before we do that, it is worth discussing the expressivity of discrete flows.

Let us assume that we have an invertible transformation $f : \mathcal{X}^D \to \mathcal{X}^D$. Moreover, we have $\mathcal{X} = \{0, 1\}$. As noted by [27], a discrete flow can only permute probability masses. Since there is no Jacobian (or, rather, the Jacobian determinant is equal to 1), there is no chance to decrease or increase the probability for specific values. We depict it in Fig. 4.10. You can easily imagine the situation as the space is the Rubik's cube and your hands are the flows. If you record your moves, you can always play the video backward; thus, it is invertible. However, you can only shuffle the colors around! As a result, we do not gain anything by applying the discrete flow and learning the discrete flow is equivalent to learning the base distribution $\pi$.[1] So we are back to square one.

However, as pointed out by [12], the situation looks different if we consider an extended space (or infinite space like $\mathbb{Z}$). The discrete flow can still only shuffle the probabilities, but now it can reorganize them in such a way that the probabilities can be factorized! In other words, it can help the base distribution to be a product of marginals, $\pi(\mathbf{z}) = \prod_{d=1}^{D} \pi_d(z_d|\theta_d)$, and the dependencies among variables are now encoded in the invertible transformations. An example of this case is presented in Fig. 4.11. We refer to [12] for a more thorough discussion with an appropriate lemma.

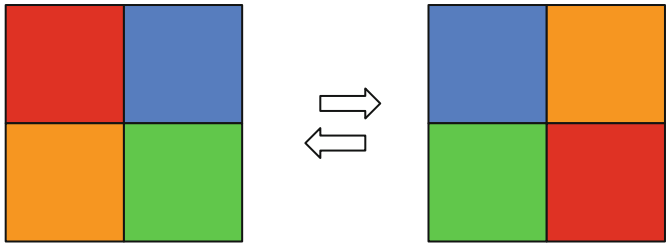


**Fig. 4.10** An example of a discrete flow for two binary random variables. Colors represent various probabilities (i.e., the sum of all squares is 1).

[1] Well, this is not entirely true; we can still learn some correlations, but it is definitely highly limited.

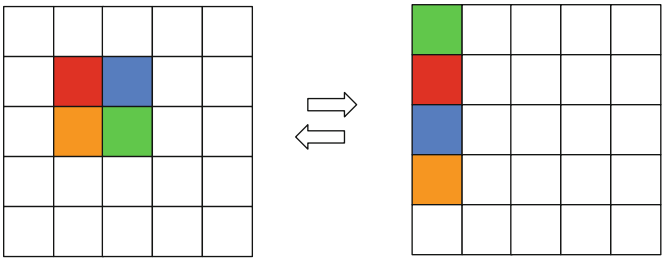


**Fig. 4.11** An example of a discrete flow for two binary random variables but in the extended space. Colors represent various probabilities (i.e., the sum of all squares is 1).

This is amazing information! It means that building a flow-based model in the discrete space makes sense. Now we can think of how to build an invertible neural network in discrete spaces, and we have it!

### 4.2.3 Integer Discrete Flows

We know now that it makes sense to work with discrete flows and that they are flexible as long as we use extended spaces or infinite spaces like $\mathbb{Z}$. However, the question is how to formulate an invertible transformation (or rather: an invertible neural network) that will output discrete values.

Hoogeboom et al. [22] proposed to focus on integers, since they can be seen as discretized continuous values. As such, we consider coupling layers [7] and modify them accordingly. Let us remind ourselves the definition of bipartite coupling layers for $\mathbf{x} \in \mathbb{R}^D$:

$$\mathbf{y}_a = \mathbf{x}_a \tag{4.33}$$

$$\mathbf{y}_b = \exp\left(s\left(\mathbf{x}_a\right)\right) \odot \mathbf{x}_b + t\left(\mathbf{x}_a\right), \tag{4.34}$$

where $s(\cdot)$ and $t(\cdot)$ are arbitrary neural networks called *scaling* and *transition*, respectively.

Considering integer-valued variables, $\mathbf{x} \in \mathbb{Z}^D$ requires modifying this transformation. First, using scaling might be troublesome, because multiplying by integers is still possible, but when we invert the transformation, we divide by integers, and dividing an integer by an integer does not necessarily result in an integer. Therefore, we must remove scaling just in case. Second, we use an arbitrary neural network for the transition. However, this network must return integers! [22] utilize a relatively simple trick; namely, they say that we can round the output of $t(\cdot)$ to the closest integer. As a result, we add (in the forward) or subtract (in the inverse) integers from integers that is perfectly fine (the outcome is still integer-valued). Eventually, we get the following bipartite coupling layer:

$$\mathbf{y}_a = \mathbf{x}_a \tag{4.35}$$

$$\mathbf{y}_b = \mathbf{x}_b + \lfloor t(\mathbf{x}_a) \rceil, \tag{4.36}$$

where $\lfloor \cdot \rceil$ is the rounding operator. An inquisitive reader could ask at this point whether the rounding operator still allows using the backpropagation algorithm, in other words, whether the rounding operator is differentiable. The answer is **no**, but [22] showed that using the straight-through estimator (STE) of a gradient is sufficient. As a side note, the STE in this case uses the rounding in the forward pass of the network, $\lfloor t(\mathbf{x}_a) \rceil$, but it utilizes $t(\mathbf{x}_a)$ in the backward pass (to calculate gradients). [12] further indicated that indeed the STE works well and the bias does not hinder training too much. The implementation of the rounding operator using the STE is presented below.

```
# We need to turn torch.round (i.e., the rounding operator) into
    a differentiable function. For this purpose, we use the
    rounding in the forward pass, but the original input for the
    backward pass. This is nothing else than the STE.
class RoundStraightThrough(torch.autograd.Function):

    def __init__(self):
        super().__init__()

    @staticmethod
    def forward(ctx, input):
        rounded = torch.round(input, out=None)
        return rounded

    @staticmethod
    def backward(ctx, grad_output):
        grad_input = grad_output.clone()
        return grad_input
```

**Listing 4.3** An implementation of the rounding operator using the STE.

In [23], it has been shown how to generalize invertible transformations like bipartite coupling layers, among others, namely ($\mathcal{X}_{i:j}$ denotes a subset of $\mathcal{X}$ corresponding to variables from the $i$-th dimension to the $j$-th dimension, $\mathbf{x}_{i:j}$, we assume that $\mathcal{X}_{1:0} = \emptyset$ and $\mathcal{X}_{n+1:n} = \emptyset$):

Proposition [23]

**Proposition 4.1** *Let us take* $\mathbf{x}, \mathbf{y} \in \mathcal{X}$. *If binary transformations* $\circ$ *and* $\triangleright$ *have inverses* $\bullet$ *and* $\blacktriangleleft$, *respectively, and* $g_2, \ldots, g_D$ *and* $f_1, \ldots, f_D$ *are arbitrary functions, where* $g_i : \mathcal{X}_{1:i-1} \rightarrow \mathcal{X}_i$, $f_i : \mathcal{X}_{1:i-1} \times \mathcal{X}_{i+1:n} \rightarrow \mathcal{X}_i$, *then the following transformation from* $\mathbf{x}$ *to* $\mathbf{y}$:

$$
\begin{aligned}
y_1 &= x_1 \circ f_1(\emptyset, \mathbf{x}_{2:D}) \\
y_2 &= (g_2(y_1) \triangleright x_2) \circ f_2(y_1, \mathbf{x}_{3:D}) \\
&\ldots \\
y_d &= (g_d(\mathbf{y}_{1:d-1}) \triangleright x_d) \circ f_d(\mathbf{y}_{1:d-1}, \mathbf{x}_{d+1:D}) \\
&\ldots \\
y_D &= (g_D(\mathbf{y}_{1:D-1}) \triangleright x_D) \circ f_D(\mathbf{y}_{1:D-1}, \emptyset)
\end{aligned}
$$

*is invertible.*

***Proof*** In order to inverse $\mathbf{y}$ to $\mathbf{x}$ we start from the last element to obtain the following:

$$
x_D = g_D(\mathbf{y}_{1:D-1}) \blacktriangleleft (y_D \bullet f_D(\mathbf{y}_{1:D-1}, \emptyset)) .
$$

Then, we can proceed with the next expressions in the decreasing order (*i.e.*, from $D - 1$ to 1) to eventually obtain:

$$
\begin{aligned}
x_{D-1} &= g_{D-1}(\mathbf{y}_{1:D-2}) \blacktriangleleft (y_{D-1} \bullet f_{D-1}(\mathbf{y}_{1:D-2}, x_D)) \\
&\ldots \\
x_d &= g_d(\mathbf{y}_{1:d-1}) \blacktriangleleft (y_d \bullet f_d(\mathbf{y}_{1:d-1}, \mathbf{x}_{d+1:D})) \\
&\ldots \\
x_2 &= g_2(y_1) \blacktriangleleft (y_2 \bullet f_2(y_1, \mathbf{x}_{3:D})) \\
x_1 &= y_1 \bullet f_1(\emptyset, \mathbf{x}_{2:D}).
\end{aligned}
$$

□

For instance, we can divide $\mathbf{x}$ into four parts, $\mathbf{x} = [\mathbf{x}_a, \mathbf{x}_b, \mathbf{x}_c, \mathbf{x}_d]$, and the following transformation (a quadripartite coupling layer) is invertible [23]:

$$
\mathbf{y}_a = \mathbf{x}_a + \lfloor t\,(\mathbf{x}_b, \mathbf{x}_c, \mathbf{x}_d) \rceil \tag{4.37}
$$

$$
\mathbf{y}_b = \mathbf{x}_b + \lfloor t\,(\mathbf{y}_a, \mathbf{x}_c, \mathbf{x}_d) \rceil \tag{4.38}
$$

$$
\mathbf{y}_c = \mathbf{x}_c + \lfloor t\,(\mathbf{y}_a, \mathbf{y}_b, \mathbf{x}_d) \rceil \tag{4.39}
$$

$$
\mathbf{y}_d = \mathbf{x}_d + \lfloor t\,(\mathbf{y}_a, \mathbf{y}_b, \mathbf{y}_c) \rceil . \tag{4.40}
$$

This new invertible transformation could be seen as a kind of autoregressive processing, since $\mathbf{y}_a$ is used to calculate $\mathbf{y}_b$, and then both $\mathbf{y}_a$ and $\mathbf{y}_b$ are used for

obtaining $\mathbf{y}_c$ and so on. As a result, we get a more powerful transformation than the bipartite coupling layer.

If we stick to a coupling layer, we need to remember to use a permutation layer to reverse the order of variables. Otherwise, some inputs would be only partially processed. This holds true for any coupling layer, either they are used for continuous flows or integer-valued flows.

The last component we need to think of is the base distribution. Similarly to flow-based models, we can use various tricks to boost the performance of the model. For instance, we can consider squeezing, factoring out, and a mixture model for the base distribution [22]. However, in this section, we try to keep the model as simple as possible; therefore, we use the product of marginals as the base distribution. For images represented as integers, we use the following:

$$\pi(\mathbf{z}) = \prod_{d=1}^{D} \pi_d(z_d) \tag{4.41}$$

$$= \prod_{d=1}^{D} \mathrm{DL}(z_d|\mu_d, \nu_d), \tag{4.42}$$

where $\pi_d(z_d) = \mathrm{DL}(z_d|\mu_d, \nu_d)$ is the discretized logistic distribution that is defined as a difference of CDFs of the logistic distribution as follows [34]:

$$\pi(z) = \mathrm{sigm}\left((z + 0.5 - \mu)/\nu\right) - \mathrm{sigm}\left((z - 0.5 - \mu)/\nu\right), \tag{4.43}$$

where $\mu \in \mathbb{R}$ and $\nu > 0$ denote the mean and the scale, respectively, and $\mathrm{sigm}(\cdot)$ is the sigmoid function. Notice that this is equivalent to calculating the probability of $z$ falling into a bin of length 1; therefore, we add 0.5 in the first CDF and subtract 0.5 from the second CDF. An example of the discretized distribution is presented in Fig. 4.12, and the implementation follows. Interestingly, we can use this distribution

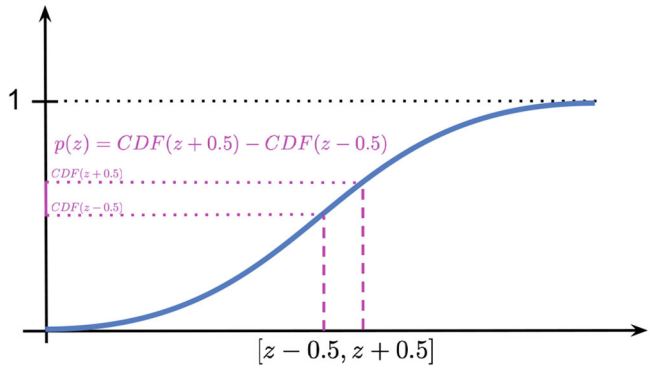



**Fig. 4.12** An example of the discretized logistic distribution with $\mu = 0$ and $\nu = 1$ The magenta area on the y-axis corresponds to the probability mass of a bin of size 1.

to replace the categorical distribution in Chap. 3, as it was done in [18]. We can even use a mixture of discretized logistic distribution to further improve the final performance [22, 35].

```
# This function implements the log of the discretized logistic
    distribution.
def log_integer_probability(x, mean, logscale):
    scale = torch.exp(logscale)

    logp = log_min_exp(
        F.logsigmoid((x + 0.5 - mean) / scale),
        F.logsigmoid((x - 0.5 - mean) / scale))

    return logp
```

**Listing 4.4** The logarithm of the discretized logistic distribution. [34]

Eventually, our log-likelihood function takes the following form:

$$\ln p(\mathbf{x}) = \sum_{d=1}^{D} \ln \mathrm{DL}(z_d = f^{-1}(\mathbf{x})|\mu_d, \nu_d) \tag{4.44}$$

$$= \sum_{d=1}^{D} \ln \left(\mathrm{sigm}\left((z_d + 0.5 - \mu_d)/\nu_d\right) - \mathrm{sigm}\left((z_d - 0.5 - \mu_d)/\nu_d\right)\right), \tag{4.45}$$

where we make all $\mu_d$ and $\nu_d$ learnable parameters. Notice that $\nu_d$ must be positive (strictly larger than 0); therefore, in the implementation, we will consider the logarithm of the scale, because taking exp of the log-scale ensures having strictly positive values.

### 4.2.4 Code

Now, we have all components to implement our own integer discrete flow (IDF)! Below, there is a code with a lot of comments that should help to understand every single line of it.

```
# That's the class of the Integer Discrete Flows (IDFs).
# There are two options implemented:
# Option 1: The bipartite coupling layers as in (Hoogeboom et al
    ., 2019).
# Option 2: A new coupling layer with 4 parts as in (Tomczak,
    2021).
# We implement the second option explicitly, without any loop, so
     that it is very clear how it works.
class IDF(nn.Module):
    def __init__(self, netts, num_flows, D=2):
        super(IDF, self).__init__()

        print('IDF by JT.')
```

```
11
12          # Option 1:
13          if len(netts) == 1:
14              self.t = torch.nn.ModuleList([netts[0]() for _ in
    range(num_flows)])
15              self.idf_git = 1
16
17          # Option 2:
18          elif len(netts) == 4:
19              self.t_a = torch.nn.ModuleList([netts[0]() for _ in
    range(num_flows)])
20              self.t_b = torch.nn.ModuleList([netts[1]() for _ in
    range(num_flows)])
21              self.t_c = torch.nn.ModuleList([netts[2]() for _ in
    range(num_flows)])
22              self.t_d = torch.nn.ModuleList([netts[3]() for _ in
    range(num_flows)])
23              self.idf_git = 4
24
25          else:
26              raise ValueError('You can provide either 1 or 4
    translation nets.')
27
28          # The number of flows (i.e., invertible transformations).
29          self.num_flows = num_flows
30
31          # The rounding operator
32          self.round = RoundStraightThrough.apply
33
34          # Initialization of the parameters of the base
    distribution.
35          # Notice they are parameters, so they are trained
    alongside the weights of neural networks.
36          self.mean = nn.Parameter(torch.zeros(1, D)) #mean
37          self.logscale = nn.Parameter(torch.ones(1, D)) #log-scale
38
39          # The dimensionality of the problem.
40          self.D = D
41
42      # The coupling layer.
43      def coupling(self, x, index, forward=True):
44
45          # Option 1:
46          if self.idf_git == 1:
47              (xa, xb) = torch.chunk(x, 2, 1)
48
49              if forward:
50                  yb = xb + self.round(self.t[index](xa))
51              else:
52                  yb = xb - self.round(self.t[index](xa))
53
54              return torch.cat((xa, yb), 1)
55
56          # Option 2:
```

```
        elif self.idf_git == 4:
            (xa, xb, xc, xd) = torch.chunk(x, 4, 1)

            if forward:
                ya = xa + self.round(self.t_a[index](torch.cat((
    xb, xc, xd), 1)))
                yb = xb + self.round(self.t_b[index](torch.cat((
    ya, xc, xd), 1)))
                yc = xc + self.round(self.t_c[index](torch.cat((
    ya, yb, xd), 1)))
                yd = xd + self.round(self.t_d[index](torch.cat((
    ya, yb, yc), 1)))
            else:
                yd = xd - self.round(self.t_d[index](torch.cat((
    xa, xb, xc), 1)))
                yc = xc - self.round(self.t_c[index](torch.cat((
    xa, xb, yd), 1)))
                yb = xb - self.round(self.t_b[index](torch.cat((
    xa, yc, yd), 1)))
                ya = xa - self.round(self.t_a[index](torch.cat((
    yb, yc, yd), 1)))

            return torch.cat((ya, yb, yc, yd), 1)

    # Similar to RealNVP, we have also the permute layer.
    def permute(self, x):
        return x.flip(1)

    # The main function of the IDF: forward pass from x to z.
    def f(self, x):
        z = x
        for i in range(self.num_flows):
            z = self.coupling(z, i, forward=True)
            z = self.permute(z)

        return z

    # The function for inverting z to x.
    def f_inv(self, z):
        x = z
        for i in reversed(range(self.num_flows)):
            x = self.permute(x)
            x = self.coupling(x, i, forward=False)

        return x

    # The PyTorch forward function. It returns the log-
    probability.
    def forward(self, x, reduction='avg'):
        z = self.f(x)
        if reduction == 'sum':
            return -self.log_prior(z).sum()
        else:
            return -self.log_prior(z).mean()
```

```
102
103     # The function for sampling:
104     # First we sample from the base distribution.
105     # Second, we invert z.
106     def sample(self, batchSize, intMax=100):
107         # sample z:
108         z = self.prior_sample(batchSize=batchSize, D=self.D,
    intMax=intMax)
109         # x = f^-1(z)
110         x = self.f_inv(z)
111         return x.view(batchSize, 1, self.D)
112
113     # The function for calculating the logarithm of the base
    distribution.
114     def log_prior(self, x):
115         log_p = log_integer_probability(x, self.mean, self.
    logscale)
116         return log_p.sum(1)
117
118     # A function for sampling integers from the base distribution
    .
119     def prior_sample(self, batchSize, D=2):
120         # Sample from logistic
121         y = torch.rand(batchSize, self.D)
122         # Here we use a property of the logistic distribution:
123         # In order to sample from a logistic distribution, first
    sample y ~ Uniform[0,1].
124         # Then, calculate log(y / (1.-y)), scale is with the
    scale, and add the mean.
125         x = torch.exp(self.logscale) * torch.log(y / (1. - y)) +
    self.mean
126         # And then round it to an integer.
127         return torch.round(x)
```

**Listing 4.5** An example of networks.

Below, we provide examples of neural networks that could be used to run the IDFs.

```
1  # The number of invertible transformations
2  num_flows = 8
3
4  # This variable defines whether we use:
5  #   Option 1: 1 - the classic coupling layer proposed in (
       Hogeboom et al., 2019)
6  #   Option 2: 4 - the general invertible transformation in (
       Tomczak, 2021) with 4 partitions
7  idf_git = 1
8
9  if idf_git == 1:
10     nett = lambda: nn.Sequential(
11                         nn.Linear(D//2, M), nn.LeakyReLU(),
12                         nn.Linear(M, M), nn.LeakyReLU(),
13                         nn.Linear(M, D//2))
14     netts = [nett]
```

```
elif idf_git == 4:
    nett_a = lambda: nn.Sequential(
                        nn.Linear(3 * (D//4), M), nn.LeakyReLU(),
                        nn.Linear(M, M), nn.LeakyReLU(),
                        nn.Linear(M, D//4))

    nett_b = lambda: nn.Sequential(
                        nn.Linear(3 * (D//4), M), nn.LeakyReLU(),
                        nn.Linear(M, M), nn.LeakyReLU(),
                        nn.Linear(M, D//4))

    nett_c = lambda: nn.Sequential(
                        nn.Linear(3 * (D//4), M), nn.LeakyReLU(),
                        nn.Linear(M, M), nn.LeakyReLU(),
                        nn.Linear(M, D//4))

    nett_d = lambda: nn.Sequential(
                        nn.Linear(3 * (D//4), M), nn.LeakyReLU(),
                        nn.Linear(M, M), nn.LeakyReLU(),
                        nn.Linear(M, D//4))

    netts = [nett_a, nett_b, nett_c, nett_d]

# Init IDF
model = IDF(netts, num_flows, D=D)
# Print the summary (like in Keras)
print(summary(model, torch.zeros(1, 64), show_input=False,
    show_hierarchical=False))
```

**Listing 4.6** An example of networks.

And we are done; this is all we need to have! After running the code and training the IDFs, we should obtain results similar to those in Fig. 4.13.

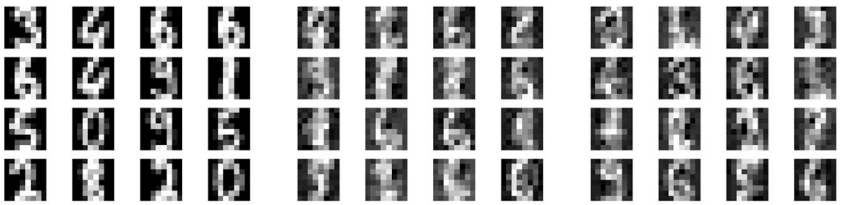



**Fig. 4.13** An example of outcomes after the training: (**a**) Randomly selected real images. (**b**) Unconditional generations from the IDF with bipartite coupling layers. (**c**) Unconditional generations from the IDF with quadripartite coupling layers.

### 4.2.5 What's Next?

Similarly to our example of RealNVP, here we present rather a simplified implementation of IDFs. We can use many of the tricks presented in the section on RealNVP (see Sect. 4.1.6. On recent developments on IDFs, please see also [12].

Integer discrete flows have great potential in data compression. Since IDFs learn the distribution $p(\mathbf{x})$ directly on the integer-valued objects, they are excellent candidates for lossless compression. As presented in [22], they are competitive with other codecs for lossless compression of images.

The paper by [12] further shows that the potential bias following from the STE of the gradients is not as dramatic as originally thought [22], and they can learn flexible distributions. This result suggests that IDFs require special attention, especially for real-life applications like data compression.

It seems that the next step would be to think of more powerful transformations for discrete variables, e.g., see [23], and develop powerful architectures. Another interesting direction is utilizing alternative learning algorithms in which gradients could be better estimated [36] or even replaced [37].

## References

1. Danilo Rezende and Shakir Mohamed. Variational inference with normalizing flows. In *International Conference on Machine Learning*, pages 1530–1538. PMLR, 2015.
2. Oren Rippel and Ryan Prescott Adams. High-dimensional probability estimation with deep density models. *arXiv preprint arXiv:1302.5125*, 2013.
3. Rianne Van Den Berg, Leonard Hasenclever, Jakub M Tomczak, and Max Welling. Sylvester normalizing flows for variational inference. In *34th Conference on Uncertainty in Artificial Intelligence 2018, UAI 2018*, pages 393–402. Association For Uncertainty in Artificial Intelligence (AUAI), 2018.
4. Jens Behrmann, Will Grathwohl, Ricky TQ Chen, David Duvenaud, and Jörn-Henrik Jacobsen. Invertible residual networks. In *International Conference on Machine Learning*, pages 573–582. PMLR, 2019.
5. Ricky TQ Chen, Jens Behrmann, David Duvenaud, and Jörn-Henrik Jacobsen. Residual flows for invertible generative modeling. *arXiv preprint arXiv:1906.02735*, 2019.
6. Yura Perugachi-Diaz, Jakub M Tomczak, and Sandjai Bhulai. Invertible densenets with concatenated lipswish. *Advances in Neural Information Processing Systems*, 2021.
7. Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. Density estimation using Real NVP. *arXiv preprint arXiv:1605.08803*, 2016.
8. Diederik P Kingma and Prafulla Dhariwal. Glow: generative flow with invertible 1× 1 convolutions. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 10236–10245, 2018.

9. Lucas Theis, Aäron van den Oord, and Matthias Bethge. A note on the evaluation of generative models. *arXiv preprint arXiv:1511.01844*, 2015.
10. Emiel Hoogeboom, Taco S Cohen, and Jakub M Tomczak. Learning discrete distributions by dequantization. *arXiv preprint arXiv:2001.11235*, 2020.
11. Thomas Bachlechner, Bodhisattwa Prasad Majumder, Huanru Henry Mao, Garrison W Cottrell, and Julian McAuley. Rezero is all you need: Fast convergence at large depth. *arXiv preprint arXiv:2003.04887*, 2020.
12. Rianne van den Berg, Alexey A Gritsenko, Mostafa Dehghani, Casper Kaae Sønderby, and Tim Salimans. Idf++: Analyzing and improving integer discrete flows for lossless compression. *arXiv e-prints*, pages arXiv–2006, 2020.
13. Jonathan Ho, Xi Chen, Aravind Srinivas, Yan Duan, and Pieter Abbeel. Flow++: Improving flow-based generative models with variational dequantization and architecture design. In *International Conference on Machine Learning*, pages 2722–2730. PMLR, 2019.
14. Jonathan Ho, Evan Lohn, and Pieter Abbeel. Compression with flows via local bits-back coding. *arXiv preprint arXiv:1905.08500*, 2019.
15. Michał Stypułkowski, Kacper Kania, Maciej Zamorski, Maciej Zięba, Tomasz Trzciński, and Jan Chorowski. Representing point clouds with generative conditional invertible flow networks. *arXiv preprint arXiv:2010.11087*, 2020.
16. Christina Winkler, Daniel Worrall, Emiel Hoogeboom, and Max Welling. Learning likelihoods with conditional normalizing flows. *arXiv preprint arXiv:1912.00042*, 2019.
17. Valentin Wolf, Andreas Lugmayr, Martin Danelljan, Luc Van Gool, and Radu Timofte. Deflow: Learning complex image degradations from unpaired data with conditional flows. *arXiv preprint arXiv:2101.05796*, 2021.
18. Durk P Kingma, Tim Salimans, Rafal Jozefowicz, Xi Chen, Ilya Sutskever, and Max Welling. Improved variational inference with inverse autoregressive flow. *Advances in Neural Information Processing Systems*, 29:4743–4751, 2016.
19. Emiel Hoogeboom, Victor Garcia Satorras, Jakub M Tomczak, and Max Welling. The convolution exponential and generalized sylvester flows. *arXiv preprint arXiv:2006.01910*, 2020.
20. Jakub M Tomczak and Max Welling. Improving variational auto-encoders using householder flow. *arXiv preprint arXiv:1611.09630*, 2016.
21. Jakub M Tomczak and Max Welling. Improving variational auto-encoders using convex combination linear inverse autoregressive flow. *arXiv preprint arXiv:1706.02326*, 2017.
22. Emiel Hoogeboom, Jorn WT Peters, Rianne van den Berg, and Max Welling. Integer discrete flows and lossless compression. *arXiv preprint arXiv:1905.07376*, 2019.
23. Jakub M Tomczak. General invertible transformations for flow-based generative modeling. *INNF+*, 2021.
24. Johann Brehmer and Kyle Cranmer. Flows for simultaneous manifold learning and density estimation. *arXiv preprint arXiv:2003.13913*, 2020.

25. George Papamakarios, David Sterratt, and Iain Murray. Sequential neural likelihood: Fast likelihood-free inference with autoregressive flows. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 837–848. PMLR, 2019.
26. George Papamakarios, Theo Pavlakou, and Iain Murray. Masked autoregressive flow for density estimation. *arXiv preprint arXiv:1705.07057*, 2017.
27. George Papamakarios, Eric Nalisnick, Danilo Jimenez Rezende, Shakir Mohamed, and Balaji Lakshminarayanan. Normalizing flows for probabilistic modeling and inference. *arXiv preprint arXiv:1912.02762*, 2019.
28. Henry Gouk, Eibe Frank, Bernhard Pfahringer, and Michael Cree. Regularisation of neural networks by enforcing Lipschitz continuity. *arXiv preprint arXiv:1804.04368*, 2018.
29. Takeru Miyato, Toshiki Kataoka, Masanori Koyama, and Yuichi Yoshida. Spectral normalization for generative adversarial networks. *arXiv preprint arXiv:1802.05957*, 2018.
30. John Skilling. The eigenvalues of mega-dimensional matrices. In *Maximum Entropy and Bayesian Methods*, pages 455–466. Springer, 1989.
31. Michael F Hutchinson. A stochastic estimator of the trace of the influence matrix for laplacian smoothing splines. *Communications in Statistics-Simulation and Computation*, 19(2):433–450, 1990.
32. Herman Kahn. Use of different Monte Carlo sampling techniques. *Proceedings of Symposium on Monte Carlo Methods*, 1955.
33. Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2017.
34. Subrata Chakraborty and Dhrubajyoti Chakravarty. A new discrete probability distribution with integer support on $(-\infty, \infty)$. *Communications in Statistics-Theory and Methods*, 45(2):492–505, 2016.
35. Tim Salimans, Andrej Karpathy, Xi Chen, and Diederik P Kingma. Pixelcnn++: Improving the pixelcnn with discretized logistic mixture likelihood and other modifications. *arXiv preprint arXiv:1701.05517*, 2017.
36. Emile van Krieken, Jakub M Tomczak, and Annette ten Teije. Storchastic: A framework for general stochastic automatic differentiation. *Advances in Neural Information Processing Systems*, 2021.
37. Niru Maheswaranathan, Luke Metz, George Tucker, Dami Choi, and Jascha Sohl-Dickstein. Guided evolutionary strategies: Augmenting random search with surrogate gradients. In *International Conference on Machine Learning*, pages 4264–4273. PMLR, 2019.

# Chapter 5
# Latent Variable Models


## 5.1 Introduction

In the previous sections, we discussed two approaches to learning $p(\mathbf{x})$: autoregressive models (ARMs) in Chap. 3 and flow-based models (or flows for short) in Chap. 4. Both ARMs and flows model the likelihood function directly, that is, either by factorizing the distribution and parameterizing conditional distributions $p(x_d|\mathbf{x}_{<d})$ as in ARMs or by utilizing invertible transformations (neural networks) for the change of variables formula as in flows. Now, we will discuss a third approach that introduces **latent variables**.

Let us briefly discuss the following scenario. We have a collection of images with horses. We want to learn $p(\mathbf{x})$ for, e.g., generating new images. Before we do that, we can ask ourselves how we should generate a horse, or, in other words, if we were such a generative model, how we would do that. Maybe we would first sketch the general silhouette of a horse, its size and shape; then add hooves; fill in details of a head; color it; etc. In the end, we may consider the background. In general, we can say that there are some *factors* in data (e.g., a silhouette, a color, a background) that are crucial for generating an object (here, a horse). Once we decide about these factors, we can generate them by adding details. I do not want to delve into a philosophical/cognitive discourse, but I hope that we all agree that when we paint something, this is more or less our procedure of generating a painting.

We use mathematics now to express this *generative process*. Namely, we have our high-dimensional objects of interest, $\mathbf{x} \in \mathcal{X}^D$ (e.g., for images, $\mathcal{X} \in \{0, 1, \ldots, 255\}$), and a **low-dimensional latent variables**, $\mathbf{z} \in \mathcal{Z}^M$ (e.g., $\mathcal{Z} = \mathbb{R}$), that we can call hidden factors in data. In mathematical words, we can refer to $\mathcal{Z}^M$ as a low-dimensional *manifold*. Then, the generative process could be expressed as follows:

1. $\mathbf{z} \sim p(\mathbf{z})$ (Fig. 5.1, in red);
2. $\mathbf{x} \sim p(\mathbf{x}|\mathbf{z})$ (Fig. 5.1, in blue).



J. M. Tomczak, *Deep Generative Modeling*,
https://doi.org/10.1007/978-3-031-64087-2_5

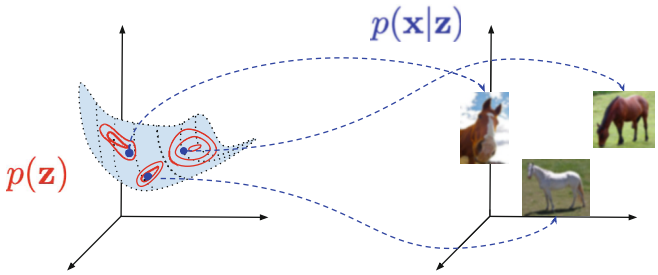



**Fig. 5.1** A diagram presenting a latent variable model and a generative process. Notice the low-dimensional manifold (here 2D) embedded in the high-dimensional space (here 3D).

In plain words, we first sample $\mathbf{z}$ (e.g., we imagine the size, the shape, and the color of a horse) and then create an image with all necessary details, i.e., we sample $\mathbf{x}$ from the conditional distribution $p(\mathbf{x}|\mathbf{z})$. One can ask whether we need probabilities here but try to create *precisely the same* image at least two times. Due to many various external factors, it is almost impossible to create two identical images. That is why probability theory is so beautiful and allows us to describe reality!

The idea behind **latent variable models** is that we introduce the latent variables $\mathbf{z}$, and the joint distribution is factorized as follows: $p(\mathbf{x}, \mathbf{z}) = p(\mathbf{x}|\mathbf{z})p(\mathbf{z})$. This naturally expressed the generative process described above. However, for training, we have access only to $\mathbf{x}$. Therefore, according to probabilistic inference, we should *sum out* (or *marginalize out*) the unknown, namely, $\mathbf{z}$. As a result, the (marginal) likelihood function is the following:

$$p(\mathbf{x}) = \int p(\mathbf{x}|\mathbf{z})p(\mathbf{z})\,\mathrm{d}\mathbf{z}. \tag{5.1}$$

A natural question now is how to calculate this integral. In general, it is a difficult task. There are two possible directions. First, the integral **is** tractable. We will briefly discuss it before we jump into the second approach that utilizes a specific **approximate inference**, namely, **variational inference**.

## 5.2 Probabilistic Principal Component Analysis

Let us discuss the following situation:

- We consider continuous random variables only, i.e., $\mathbf{z} \in \mathbb{R}^M$ and $\mathbf{x} \in \mathbb{R}^D$.
- The distribution of $\mathbf{z}$ is the standard Gaussian, i.e., $p(\mathbf{z}) = \mathcal{N}(\mathbf{z}|0, \mathbf{I})$.
- The dependency between $\mathbf{z}$ and $\mathbf{x}$ is linear, and we assume a Gaussian additive noise:

$$\mathbf{x} = \mathbf{W}\mathbf{z} + \mathbf{b} + \varepsilon, \tag{5.2}$$

where $\varepsilon \sim \mathcal{N}(\varepsilon|0, \sigma^2\mathbf{I})$. The property of the Gaussian distribution yields [1]:

$$p(\mathbf{x}|\mathbf{z}) = \mathcal{N}\left(\mathbf{x}|\mathbf{W}\mathbf{z} + \mathbf{b}, \sigma^2\mathbf{I}\right). \tag{5.3}$$

This model is known as the *probabilistic Principal Component Analysis* (pPCA) [2].

Next, we can take advantage of the properties of a linear combination of two vectors of normally distributed random variables to calculate the integral explicitly [1]:

$$p(\mathbf{x}) = \int p(\mathbf{x}|\mathbf{z})\, p(\mathbf{z})\, \mathrm{d}\mathbf{z} \tag{5.4}$$

$$= \int \mathcal{N}\left(\mathbf{x}|\mathbf{W}\mathbf{z} + \mathbf{b}, \sigma^2\mathbf{I}\right) \mathcal{N}\left(\mathbf{z}|0, \mathbf{I}\right)\, \mathrm{d}\mathbf{z} \tag{5.5}$$

$$= \mathcal{N}\left(\mathbf{x}|\mathbf{b}, \mathbf{W}\mathbf{W}^\top + \sigma^2\mathbf{I}\right). \tag{5.6}$$

Now, we are able to calculate the logarithm of the (marginal) likelihood function $\ln p(\mathbf{x})$! We refer to [1, 2] for more details on learning the parameters in the pPCA model. Moreover, what is interesting about the pPCA is that, due to the properties of Gaussians, we can also calculate the *true* posterior over $\mathbf{z}$ analytically:

$$p(\mathbf{z}|\mathbf{x}) = \mathcal{N}\left(\mathbf{M}^{-1}\mathbf{W}^\top(\mathbf{x} - \mathbf{b}), \sigma^{-2}\mathbf{M}\right), \tag{5.7}$$

where $\mathbf{M} = \mathbf{W}^\top\mathbf{W} + \sigma^2\mathbf{I}$. Once we find $\mathbf{W}$ that maximizes the log-likelihood function, and the dimensionality of the matrix $\mathbf{W}$ is computationally tractable, we can calculate $p(\mathbf{z}|\mathbf{x})$. This is a big thing! Why? Because for a given observation $\mathbf{x}$, we can calculate the distribution over the *latent factors*!

**A side note**

In my opinion, the probabilistic PCA is an extremely important latent variable model for two reasons. First, we can calculate everything *by hand*, thus, it is a great exercise to develop an intuition about the latent variable models. Second, it is a linear model, therefore, a curious reader like you should feel tingling in their head already and ask herself the following questions: What would happen if we take non-linear dependencies? And what would happen if we use other distributions than Gaussians? In both cases, the answer is the same: We would not be able to calculate the integral exactly, and some sort of approximation would be necessary. Anyhow, pPCA is a model that everyone interested in latent variable models should study in depth to create an intuition about probabilistic modeling.

## 5.3 Variational Auto-encoders: Variational Inference for Nonlinear Latent Variable Models

### 5.3.1 The Model and the Objective

Let us take a look at the integral one more time and think of a general case where we cannot calculate it analytically. The simplest approach would be to use the Monte Carlo approximation:

$$p(\mathbf{x}) = \int p(\mathbf{x}|\mathbf{z})\, p(\mathbf{z})\, \mathrm{d}\mathbf{z} \tag{5.8}$$

$$= \mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}\left[p(\mathbf{x}|\mathbf{z})\right] \tag{5.9}$$

$$\approx \frac{1}{K}\sum_k p(\mathbf{x}|\mathbf{z}_k), \tag{5.10}$$

where, in the last line, we use samples from the prior over latents, $\mathbf{z}_k \sim p(\mathbf{z})$. Such an approach is relatively easy, and since our computational power grows so fast, we can sample a lot of points in a reasonably short time. However, as we know from statistics, if $\mathbf{z}$ is multidimensional, and $M$ is relatively large, we get into a trap of the *curse of dimensionality*, and to cover the space properly, the number of samples grows exponentially with respect to $M$. If we take too few samples, then the approximation is simply very poor.

We can use more advanced Monte Carlo techniques [3]; however, they still suffer from issues associated with the curse of dimensionality. An alternative approach is an application of *variational inference* [4]. Let us consider a family of variational distributions parameterized by $\phi$, $\{q_\phi(\mathbf{z})\}_\phi$. For instance, we can consider Gaussians with means and variances, $\phi = \{\mu, \sigma^2\}$. We know the form of these distributions, and we assume that they assign nonzero probability mass to all $\mathbf{z} \in \mathcal{Z}^M$. Then, the logarithm of the marginal distribution could be approximated as follows:

$$\ln p(\mathbf{x}) = \ln \int p(\mathbf{x}|\mathbf{z})p(\mathbf{z})\, \mathrm{d}\mathbf{z} \tag{5.11}$$

$$= \ln \int \frac{q_\phi(\mathbf{z})}{q_\phi(\mathbf{z})} p(\mathbf{x}|\mathbf{z})p(\mathbf{z})\, \mathrm{d}\mathbf{z} \tag{5.12}$$

$$= \ln \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z})}\left[\frac{p(\mathbf{x}|\mathbf{z})p(\mathbf{z})}{q_\phi(\mathbf{z})}\right] \tag{5.13}$$

$$\geq \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z})} \ln\left[\frac{p(\mathbf{x}|\mathbf{z})p(\mathbf{z})}{q_\phi(\mathbf{z})}\right] \tag{5.14}$$

$$= \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z})}\left[\ln p(\mathbf{x}|\mathbf{z}) + \ln p(\mathbf{z}) - \ln q_\phi(\mathbf{z})\right] \tag{5.15}$$

$$= \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z})}\left[\ln p(\mathbf{x}|\mathbf{z})\right] - \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z})}\left[\ln q_\phi(\mathbf{z}) - \ln p(\mathbf{z})\right]. \tag{5.16}$$

In the fourth line, we used *Jensen's inequality*.

If we consider an *amortized variational posterior*, namely, $q_\phi(\mathbf{z}|\mathbf{x})$ instead of $q_\phi(\mathbf{z})$ for each $\mathbf{x}$, then we get:

$$\ln p(\mathbf{x}) \geq \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\ln p(\mathbf{x}|\mathbf{z})\right] - \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\ln q_\phi(\mathbf{z}|\mathbf{x}) - \ln p(\mathbf{z})\right]. \tag{5.17}$$

Amortization could be extremely useful, because we train a single model (e.g., a neural network with some weights), and it returns the parameters of distribution for a given input. From now on, we will assume that we use amortized variational posteriors; however, please remember that we do not need to do that! Please take a look at [5] where a semi-amortized variational inference is considered.

As a result, we obtain an auto-encoder-like model, with a *stochastic encoder*, $q_\phi(\mathbf{z}|\mathbf{x})$, and a *stochastic decoder*, $p(\mathbf{x}|\mathbf{z})$. We use *stochastic* to highlight that the encoder and the decoder are probability distributions and to stress out a difference to a deterministic auto-encoder. This model, with the amortized variational posterior, is called a **Variational Auto-Encoder** [6, 7]. The lower bound of the log-likelihood function is called the evidence lower bound (**ELBO**).

The first part of the ELBO, $\mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\ln p(\mathbf{x}|\mathbf{z})\right]$, is referred to as the (negative) *reconstruction error*, because $\mathbf{x}$ is encoded to $\mathbf{z}$ and then decoded back. The second part of the ELBO, $\mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\ln q_\phi(\mathbf{z}|\mathbf{x}) - \ln p(\mathbf{z})\right]$, could be seen as a *regularizer*, and it coincides with the Kullback-Leibler divergence (KL). Please keep in mind that for more complex models (e.g., hierarchical models), the regularizer(s) may not be interpreted as the KL term. Therefore, we prefer to use the term *the regularizer*, because it is more general.

### 5.3.2 A Different Perspective on the ELBO

For completeness, we provide also a different derivation of the ELBO that will help us to understand why the lower bound might be tricky sometimes:

$$\ln p(\mathbf{x}) = \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\ln p(\mathbf{x})\right] \tag{5.18}$$

$$= \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\ln \frac{p(\mathbf{z}|\mathbf{x})p(\mathbf{x})}{p(\mathbf{z}|\mathbf{x})}\right] \tag{5.19}$$

$$= \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\ln \frac{p(\mathbf{x}|\mathbf{z})p(\mathbf{z})}{p(\mathbf{z}|\mathbf{x})}\right] \tag{5.20}$$

$$= \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\ln \frac{p(\mathbf{x}|\mathbf{z})p(\mathbf{z})}{p(\mathbf{z}|\mathbf{x})}\frac{q_\phi(\mathbf{z}|\mathbf{x})}{q_\phi(\mathbf{z}|\mathbf{x})}\right] \tag{5.21}$$

$$= \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\ln p(\mathbf{x}|\mathbf{z})\frac{p(\mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x})}\frac{q_\phi(\mathbf{z}|\mathbf{x})}{p(\mathbf{z}|\mathbf{x})}\right] \tag{5.22}$$

$$= \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\ln p(\mathbf{x}|\mathbf{z}) - \ln \frac{q_\phi(\mathbf{z}|\mathbf{x})}{p(\mathbf{z})} + \ln \frac{q_\phi(\mathbf{z}|\mathbf{x})}{p(\mathbf{z}|\mathbf{x})}\right] \tag{5.23}$$

$$= \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\ln p(\mathbf{x}|\mathbf{z})\right] - KL\left[q_\phi(\mathbf{z}|\mathbf{x})\|p(\mathbf{z})\right] + KL\left[q_\phi(\mathbf{z}|\mathbf{x})\|p(\mathbf{z}|\mathbf{x})\right]. \tag{5.24}$$

Please note that in the derivation above, we use the sum and the product rules together with multiplying by $1 = \frac{q_\phi(\mathbf{z}|\mathbf{x})}{q_\phi(\mathbf{z}|\mathbf{x})}$, nothing else, no dirty tricks here! Please try to replicate this by yourself, step by step. If you understand this derivation well, it would greatly help you to see where potential problems of the VAEs (and the latent variable models in general) lie.

Once you analyzed this derivation, let us take a closer look at it:

$$\ln p(\mathbf{x}) = \underbrace{\mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\ln p(\mathbf{x}|\mathbf{z})\right] - KL\left[q_\phi(\mathbf{z}|\mathbf{x})\|p(\mathbf{z})\right]}_{ELBO} + \underbrace{KL\left[q_\phi(\mathbf{z}|\mathbf{x})\|p(\mathbf{z}|\mathbf{x})\right]}_{\geq 0}. \tag{5.25}$$

The last component, $KL\left[q_\phi(\mathbf{z}|\mathbf{x})\|p(\mathbf{z}|\mathbf{x})\right]$, measures the difference between the variational posterior and the *real* posterior, but we do not know what the real posterior is! However, we can skip this part, since the Kullback-Leibler divergence is always equal or greater than 0 (from its definition); thus, we are left with the ELBO. We can think of $KL\left[q_\phi(\mathbf{z}|\mathbf{x})\|p(\mathbf{z}|\mathbf{x})\right]$ as a gap between the ELBO and the true log-likelihood.

Beautiful! But ok, why this is so important? Well, if we take $q_\phi(\mathbf{z}|\mathbf{x})$ that is a bad approximation of $p(\mathbf{z}|\mathbf{x})$, then the KL term will be larger, and even if the ELBO is optimized well, the gap between the ELBO and the true log-likelihood could be huge! In plain words, if we take a too simplistic posterior, we can end up with a bad VAE anyway. What is "bad" in this context? Let us take a look at Fig. 5.2. If the ELBO is a loose lower bound of the log-likelihood, then the optimal solution of the ELBO could be completely different than the solution of the log-likelihood. We will comment on how to deal with that later on; for now, it is enough to be aware of that issue.

### 5.3.3 Components of VAEs

Let us wrap up what we know right now. First of all, we consider a class of amortized variational posteriors $\{q_\phi(\mathbf{z}|\mathbf{x})\}_\phi$ that approximate the true posterior $p(\mathbf{z}|\mathbf{x})$. We can

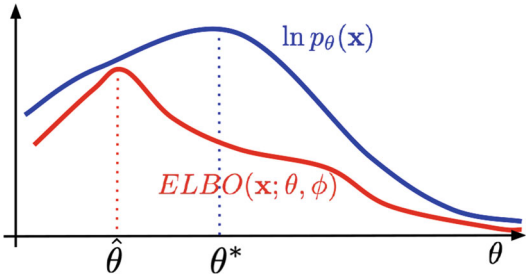



**Fig. 5.2** The ELBO is a lower bound on the log-likelihood. As a result, $\hat{\theta}$ maximizing the ELBO does not necessarily coincides with $\theta^*$ that maximizes $\ln p(\mathbf{x})$. The looser the ELBO is, the more this can bias maximum likelihood estimates of the model parameters.

see them as **stochastic encoders**. Second, the conditional likelihood $p(\mathbf{x}|\mathbf{z})$ could be seen as a **stochastic decoder**. Third, the last component, $p(\mathbf{z})$, is the **marginal distribution**, also referred to as a **prior**. Lastly, the objective is the ELBO, a lower bound to the log-likelihood function:

$$\ln p(\mathbf{x}) \geq \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\ln p(\mathbf{x}|\mathbf{z})\right] - \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\ln q_\phi(\mathbf{z}|\mathbf{x}) - \ln p(\mathbf{z})\right]. \tag{5.26}$$

There are two questions left to get the full picture of the VAEs:

1. How to parameterize the distributions?
2. How to calculate the expected values? After all, these integrals have not disappeared!

#### 5.3.3.1 Parameterization of Distributions

As you can probably guess by now, we use neural networks to parameterize the encoders and the decoders. But before we use the neural networks, we should know *what* distributions we use! Fortunately, in the VAE framework, we are almost free to choose any distribution! However, we must remember that they should make sense for a considered problem. So far, we have explained everything through images, so let us continue that. If $\mathbf{x} \in \{0, 1, \ldots, 255\}^D$, then we *cannot* use a normal distribution, because its support is totally different than the support of discrete-valued images. A possible distribution we can use is the *categorical distribution*, that is:

$$p_\theta(\mathbf{x}|\mathbf{z}) = \text{Categorical}\left(\mathbf{x}|\theta(\mathbf{z})\right), \tag{5.27}$$

where the probabilities are given by a neural network NN, namely, $\theta(\mathbf{z}) = \text{softmax}\left(\text{NN}(\mathbf{z})\right)$. The neural network NN could be an MLP, a convolutional neural network, RNNs, etc.

The choice of a distribution for the latent variables depends on how we want to express the latent factors in data. For convenience, typically $\mathbf{z}$ is taken as a vector of continuous random variables, $\mathbf{z} \in \mathbb{R}^M$. Then, we can use Gaussians for both the variational posterior and the prior:

$$q_\phi(\mathbf{z}|\mathbf{x}) = \mathcal{N}\left(\mathbf{z}|\mu_\phi(\mathbf{x}), \text{diag}\left[\sigma^2_\phi(\mathbf{x})\right]\right) \tag{5.28}$$

$$p(\mathbf{z}) = \mathcal{N}\left(\mathbf{z}|0, \mathbf{I}\right), \tag{5.29}$$

where $\mu_\phi(\mathbf{x})$ and $\sigma^2_\phi(\mathbf{x})$ are outputs of a neural network, similarly to the case of the decoder. In practice, we can have a shared neural network $\text{NN}(\mathbf{x})$ that outputs $2M$ values that are further split into $M$ values for the mean $\mu$ and $M$ values for the variance $\sigma^2$. For convenience, we consider a diagonal covariance matrix. We could use flexible posteriors (see Sect. 5.4.2). Moreover, here we take the standard Gaussian prior. We will comment on that later (see Sect. 5.4.1).

#### 5.3.3.2 Reparameterization Trick

So far, we played around with the log-likelihood, and we ended up with the ELBO. However, there is still a problem with calculating the expected value, because it contains an integral! Therefore, the question is how we can calculate it and why it is better than the MC approximation of the log-likelihood without the variational posterior. In fact, we will use the MC approximation, but now, instead of sampling from the prior $p(\mathbf{z})$, we will sample from the variational posterior $q_\phi(\mathbf{z}|\mathbf{x})$. Is it better? Yes, because the variational posterior assigns typically more probability mass to a smaller region than the prior. If you play around with your code of a VAE and examine the variance, you will probably notice that the variational posteriors are almost deterministic (whether it is good or bad is rather an open question). As a result, we should get a better approximation! However, there is still an issue with the variance of the approximation. Simply speaking, if we sample $\mathbf{z}$ from $q_\phi(\mathbf{z}|\mathbf{x})$, plug them into the ELBO, and calculate gradients with respect to the parameters of a neural network $\phi$, the variance of the gradient may still be pretty large! A possible solution to that, first noticed by statisticians (e.g., see [8]), is the approach of **reparameterizing** the distribution. The idea is to realize that we can express a random variable as a composition of primitive transformations (e.g., arithmetic operations, logarithm, etc.) of an independent random variable with a simple distribution. For instance, if we consider a Gaussian random variable $z$ with a mean $\mu$ and a variance $\sigma^2$, and an independent random variable $\epsilon \sim \mathcal{N}(\epsilon|0, 1)$, then the following holds (see Fig. 5.3):

$$z = \mu + \sigma \cdot \epsilon. \tag{5.30}$$

Now, if we start sampling $\epsilon$ from the standard Gaussian, and apply the above transformation, then we get a sample from $\mathcal{N}(z|\mu, \sigma)$!

If you do not remember this fact from statistics, or you simply do not believe me, write a simple code for that and play around with it. In fact, this idea could be applied to many more distributions [9].

The **reparameterization trick** could be used in the encoder $q_\phi(\mathbf{z}|\mathbf{x})$. As observed by [6, 7], we can drastically reduce the variance of the gradient by using this repa-

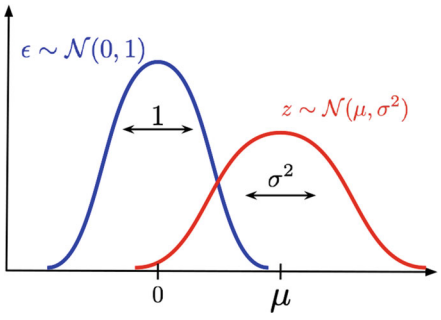



**Fig. 5.3** An example of reparameterizing a Gaussian distribution: We scale $\epsilon$ distributed according to the standard Gaussian by $\sigma$, and shift it by $\mu$.

rameterization of the Gaussian distribution. Why? Because the randomness comes from the independent source $p(\epsilon)$, and we calculate gradient with respect to a deterministic function (i.e., a neural network), not random objects. Even better, since we learn the VAE using stochastic gradient descent, it is enough to sample $\mathbf{z}$ only once during training!

### 5.3.4 VAE in Action!

We went through a lot of theory and discussions, and you might think it is impossible to implement a VAE. However, it is actually simpler than it might look. Let us sum up what we know so far and focus on very specific distributions and neural networks.

First of all, we will use the following distributions:

- $q_\phi(\mathbf{z}|\mathbf{x}) = \mathcal{N}\left(\mathbf{z}|\mu_\phi(\mathbf{x}), \sigma^2_\phi(\mathbf{x})\right)$;
- $p(\mathbf{z}) = \mathcal{N}(\mathbf{z}|0, \mathbf{I})$;
- $p_\theta(\mathbf{x}|\mathbf{z}) = \text{Categorical}(\mathbf{x}|\theta(\mathbf{z}))$.

We assume that $x_d \in \mathcal{X} = \{0, 1, \ldots, L-1\}$.

Next, we will use the following networks:

- The *encoder network*:

$$\begin{aligned}\mathbf{x} \in \mathcal{X}^D \rightarrow &\text{Linear}(D, 256) \rightarrow \text{LeakyReLU} \rightarrow \\ &\text{Linear}(256, 2 \cdot M) \rightarrow \text{split} \rightarrow \mu \in \mathbb{R}^M,\ \log \sigma^2 \in \mathbb{R}^M\end{aligned}$$

  Notice that the last layer outputs $2M$ values, because we must have $M$ values for the mean and $M$ values for the (log-)variance. Moreover, a variance must be positive; therefore, instead, we consider the logarithm of the variance, because it can take real values then. As a result, we do not need to bother about variances being always positive. An alternative is to apply a nonlinearity like softplus.
- The *decoder network*:

$$\begin{aligned}\mathbf{z} \in \mathbb{R}^M \rightarrow &\text{Linear}(M, 256) \rightarrow \text{LeakyReLU} \rightarrow \\ &\text{Linear}(256, D \cdot L) \rightarrow \text{reshape} \rightarrow \text{softmax} \rightarrow \theta \in [0, 1]^{D \times L}\end{aligned}$$

Since we use the categorical distribution for $\mathbf{x}$, the outputs of the decoder network are probabilities. Thus, the last layer must output $D \cdot L$ values, where $D$ is the number of pixels, and $L$ is the number of possible values of a pixel. Then, we must reshape the output to a tensor of the following shape: $(B, D, L)$, where $B$ is the batch size. Afterward, we can apply the softmax activation function to obtain probabilities.

Finally, for a given dataset $\mathcal{D} = \{\mathbf{x}_n\}_{n=1}^N$, the training objective is the ELBO where we use a single sample from the variational posterior $\mathbf{z}_{\phi,n} = \mu_\phi(\mathbf{x}_n) + \sigma_\phi(\mathbf{x}_n) \odot \epsilon$. We must remember that in almost any available package, we minimize by default, so we must take the negative sign, namely:

$$-ELBO(\mathcal{D};\theta,\phi) = \sum_{n=1}^{N} -\left\{\ln \text{Categorical}\left(\mathbf{x}_n|\theta\left(\mathbf{z}_{\phi,n}\right)\right) + \left[\ln \mathcal{N}\left(\mathbf{z}_{\phi,n}|\mu_\phi(\mathbf{x}_n),\sigma^2_\phi(\mathbf{x}_n)\right) - \ln \mathcal{N}\left(\mathbf{z}_{\phi,n}|0,\mathbf{I}\right)\right]\right\}. \tag{5.31}$$

So as you can see, the whole math boils down to a relatively simple learning procedure:

1. Take $\mathbf{x}_n$ and apply the encoder network to get $\mu_\phi(\mathbf{x}_n)$ and $\ln \sigma^2_\phi(\mathbf{x}_n)$.
2. Calculate $\mathbf{z}_{\phi,n}$ by applying the reparameterization trick, $\mathbf{z}_{\phi,n} = \mu_\phi(\mathbf{x}_n) + \sigma_\phi(\mathbf{x}_n) \odot \epsilon$, where $\epsilon \sim \mathcal{N}(0,\mathbf{I})$.
3. Apply the decoder network to $\mathbf{z}_{\phi,n}$ to get the probabilities $\theta(\mathbf{z}_{\phi,n})$.
4. Calculate the ELBO by plugging in $\mathbf{x}_n$, $\mathbf{z}_{\phi,n}$, $\mu_\phi(\mathbf{x}_n)$ and $\ln \sigma^2_\phi(\mathbf{x}_n)$.

### 5.3.5 Code

Now, all components are ready to be turned into a code! Here, we focus only on the code for the VAE model. We provide details in the comments. We divide the code into four classes: Encoder, Decoder, Prior, and VAE. It might look like overkill, but it may help you to think of the VAE as a composition of three parts and better comprehend the whole approach.

```
class Encoder(nn.Module):
    def __init__(self, encoder_net):
        super(Encoder, self).__init__()

        # The init of the encoder network.
        self.encoder = encoder_net

    # The reparameterization trick for Gaussians.
    @staticmethod
    def reparameterization(mu, log_var):
        # The formula is the following:
        # z = mu + std * epsilon
        # epsilon ~ Normal(0,1)

        # First, we need to get std from log-variance.
        std = torch.exp(0.5*log_var)

        # Second, we sample epsilon from Normal(0,1).
        eps = torch.randn_like(std)

        # The final output
        return mu + std * eps

    # This function implements the output of the encoder network
    (i.e., parameters of a Gaussian).
```

```
    def encode(self, x):
        # First, we calculate the output of the encoder network
    of size 2M.
        h_e = self.encoder(x)
        # Second, we must divide the output into the mean and the
     log-variance.
        mu_e, log_var_e = torch.chunk(h_e, 2, dim=1)
        return mu_e, log_var_e

    # Sampling procedure.
    def sample(self, x=None, mu_e=None, log_var_e=None):
        #If we don't provide a mean and a log-variance, we must
    first calcuate it:
        if (mu_e is None) and (log_var_e is None):
            mu_e, log_var_e = self.encode(x)
        # Or the final sample
        else:
        # Otherwise, we can simply apply the reparameterization
    trick!
            if (mu_e is None) or (log_var_e is None):
                raise ValueError('mu and log-var can't be None!')
        z = self.reparameterization(mu_e, log_var_e)
        return z

    # This function calculates the log-probability that is later
    used for calculating the ELBO.
    def log_prob(self, x=None, mu_e=None, log_var_e=None, z=None)
    :
        # If we provide x alone, then we can calculate a
    corresponding sample:
        if x is not None:
            mu_e, log_var_e = self.encode(x)
            z = self.sample(mu_e=mu_e, log_var_e=log_var_e)
        else:
        # Otherwise, we should provide mu, log-var, and z!
            if (mu_e is None) or (log_var_e is None) or (z is
    None):
                raise ValueError('mu, log-var, z can't be None')

        return log_normal_diag(z, mu_e, log_var_e)

    # PyTorch forward pass: it is either log-probability (by
    default) or sampling.
    def forward(self, x, type='log_prob'):
        assert type in ['encode', 'log_prob'], 'Type could be
    either encode or log_prob'
        if type == 'log_prob':
            return self.log_prob(x)
        else:
            return self.sample(x)
```

**Listing 5.1** An encoder class.

```
class Decoder(nn.Module):
    def __init__(self, decoder_net, distribution='categorical',
    num_vals=None):
        super(Decoder, self).__init__()

        # The decoder network.
        self.decoder = decoder_net
        # The distribution used for the decoder (it is
    categorical by default, as discussed above).
        self.distribution = distribution
        # The number of possible values. This is important for
    the categorical distribution.
        self.num_vals=num_vals

    # This function calculates parameters of the likelihood
    function p(x|z)
    def decode(self, z):
        # First, we apply the decoder network.
        h_d = self.decoder(z)

        # In this example, we use only the categorical
    distribution...
        if self.distribution == 'categorical':
            # We save the shapes: batch size
            b = h_d.shape[0]
            # and the dimensionality of x.
            d = h_d.shape[1]//self.num_vals
            # Then we reshape to (Batch size, Dimensionality,
    Number of Values).
            h_d = h_d.view(b, d, self.num_vals)
            # To get probabilities, we apply softmax.
            mu_d = torch.softmax(h_d, 2)
            return [mu_d]
        # ... however, we also present the Bernoulli distribution
    . We are nice, aren't we?
        elif self.distribution == 'bernoulli':
            # In the Bernoulli case, we have x_d \in {0,1}.
    Therefore, it is enough to output a single probability,
            # because p(x_d=1|z) = \theta and p(x_d=0|z) = 1 - \
    theta
            mu_d = torch.sigmoid(h_d)
            return [mu_d]

        else:
            raise ValueError('Only: `categorical`, `bernoulli`')

    # This function implements sampling from the decoder.
    def sample(self, z):
        outs = self.decode(z)

        if self.distribution == 'categorical':
            # We take the output of the decoder
            mu_d = outs[0]
            # and save shapes (we will need that for reshaping).
```

```
            b = mu_d.shape[0]
            m = mu_d.shape[1]
            # Here we use reshaping
            mu_d = mu_d.view(mu_d.shape[0], -1, self.num_vals)
            p = mu_d.view(-1, self.num_vals)
            # Eventually, we sample from the categorical (the
    built-in PyTorch function).
            x_new = torch.multinomial(p, num_samples=1).view(b,m)

        elif self.distribution == 'bernoulli':
            # In the case of Bernoulli, we don't need any
    reshaping
            mu_d = outs[0]
            # and we can use the built-in PyTorch sampler!
            x_new = torch.bernoulli(mu_d)

        else:
            raise ValueError('Only: ‘categorical‘, ‘bernoulli‘')

        return x_new

    # This function calculates the conditional log-likelihood
    function.
    def log_prob(self, x, z):
        outs = self.decode(z)

        if self.distribution == 'categorical':
            mu_d = outs[0]
            log_p = log_categorical(x, mu_d, num_classes=self.
    num_vals, reduction='sum', dim=-1).sum(-1)

        elif self.distribution == 'bernoulli':
            mu_d = outs[0]
            log_p = log_bernoulli(x, mu_d, reduction='sum', dim
    =-1)

        else:
            raise ValueError('Only: ‘categorical‘, ‘bernoulli‘')

        return log_p

    # The forward pass is either a log-prob or a sample.
    def forward(self, z, x=None, type='log_prob'):
        assert type in ['decoder', 'log_prob'], 'Type could be
    either decode or log_prob'
        if type == 'log_prob':
            return self.log_prob(x, z)
        else:
            return self.sample(z)
```

**Listing 5.2** A decoder class.

```
# The current implementation of the prior is very simple, namely, it is a standard Gaussian.
# We could have used a built-in PuTorch distribution. However, we didn't do that for two reasons:
# (i) It is important to think of the prior as a crucial component in VAEs.
# (ii) We can implement a learnable prior (e.g., a flow-based prior, VampPrior, a muxture of distributions).
class Prior(nn.Module):
    def __init__(self, L):
        super(Prior, self).__init__()
        self.L = L

    def sample(self, batch_size):
        z = torch.randn((batch_size, self.L))
        return z

    def log_prob(self, z):
        return log_standard_normal(z)
```

**Listing 5.3** A prior class.

```
class VAE(nn.Module):
    def __init__(self, encoder_net, decoder_net, num_vals=256, L=16, likelihood_type='categorical'):
        super(VAE, self).__init__()

        print('VAE by JT.')

        self.encoder = Encoder(encoder_net=encoder_net)
        self.decoder = Decoder(distribution=likelihood_type, decoder_net=decoder_net, num_vals=num_vals)
        self.prior = Prior(L=L)

        self.num_vals = num_vals

        self.likelihood_type = likelihood_type

    def forward(self, x, reduction='avg'):
        # encoder
        mu_e, log_var_e = self.encoder.encode(x)
        z = self.encoder.sample(mu_e=mu_e, log_var_e=log_var_e)

        # ELBO
        RE = self.decoder.log_prob(x, z)
        KL = (self.prior.log_prob(z) - self.encoder.log_prob(mu_e=mu_e, log_var_e=log_var_e, z=z)).sum(-1)

        if reduction == 'sum':
            return -(RE + KL).sum()
        else:
            return -(RE + KL).mean()

    def sample(self, batch_size=64):
```

```
        z = self.prior.sample(batch_size=batch_size)
        return self.decoder.sample(z)
```

**Listing 5.4** A VAE class.

```
# Examples of neural networks used for parameterizing the encoder and the decoder.

# Remember that the encoder outputs 2 times more values because we need L means and L log-variances for a Gaussian.
encoder = nn.Sequential(nn.Linear(D, M), nn.LeakyReLU(),
                        nn.Linear(M, M), nn.LeakyReLU(),
                        nn.Linear(M, 2 * L))

# Here we must remember that if we use the categorical distribution, we must output num_vals per each pixel.
decoder = nn.Sequential(nn.Linear(L, M), nn.LeakyReLU(),
                        nn.Linear(M, M), nn.LeakyReLU(),
                        nn.Linear(M, num_vals * D))
```

**Listing 5.5** Examples of networks.

Perfect! Now we are ready to run the full code, and after training our VAE, we should obtain results similar to those in Fig. 5.4.

### 5.3.6 Typical Issues with VAEs

VAEs constitute a very powerful class of models, mainly due to their flexibility. Unlike flow-based models, they do not require the invertibility of neural networks; thus, we can use any arbitrary architecture for encoders and decoders. In contrast to ARMs, they learn a low-dimensional data representation, and we can control the bottleneck (i.e., the dimensionality of the latent space). However, they also suffer from several issues. Except for the ones mentioned before (i.e., a necessity of an efficient integral estimation, a gap between the ELBO and the log-likelihood function for too simplistic variational posteriors), the potential problems are the following:

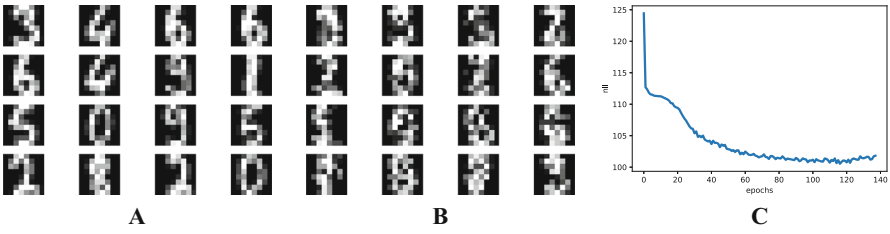



**Fig. 5.4** An example of outcomes after the training: (**a**) Randomly selected real images. (**b**) Unconditional generations from the VAE. (**c**) The validation curve during training.

- Let us take a look at the ELBO and the regularization term. For a non-trainable prior like the standard Gaussian, the regularization term will be minimized if $\forall_{\mathbf{x}} q_\phi(\mathbf{z}|\mathbf{x}) = p(\mathbf{z})$. This may happen if the decoder is so powerful that it treats $\mathbf{z}$ as a noise, e.g., a decoder is expressed by an ARM [10]. This issue is known as the *posterior collapse* [11].
- Another issue is associated with a mismatch between the aggregated posterior, $q_\phi(\mathbf{z}) = \frac{1}{N}\sum_n q_\phi(\mathbf{z}|\mathbf{x}_n)$, and the prior $p(\mathbf{z})$. Imagine that we have the standard Gaussian prior and the aggregated posterior (i.e., an average of variational posteriors over all training data). As a result, there are regions where there prior assigns a high probability, but the aggregated posterior assigns a low probability, or another way around. Then, sampling from these holes provides unrealistic latent values, and the decoder produces images of very low quality. This problem is referred to as the *hole problem* [12].
- The last problem we want to discuss is more general, and, in fact, it affects all deep generative models. As it was noticed in [13], the deep generative models (including VAEs) fail to properly detect out-of-distribution examples. Out-of-distribution datapoints are examples that follow a totally different distribution than the one a model was trained on. For instance, let us assume that our model is trained on MNIST, and then FashionMNIST examples are out-of-distribution. Thus, intuition tells us that a properly trained deep generative model should assign a high probability to in-distribution examples and a low probability to out-of-distribution points. Unfortunately, as shown in [13], this is not the case. The *out-of-distribution problem* remains one of the main unsolved problems in deep generative modeling [14].

### 5.3.7 There Is More!

There are a plethora of papers that extend VAEs and apply them to many problems. Below, we will list out selected papers and only touch upon the vast literature on the topic!

**Estimation of the log-likelihood using importance weighting** As we indicated multiple times, the ELBO is the lower bound to the log-likelihood, and it rather should not be used as a good estimate of the log-likelihood. In [7, 15], an *importance weighting* procedure is advocated to better approximate the log-likelihood, namely:

$$\ln p(\mathbf{x}) \approx \ln \frac{1}{K}\sum_{k=1}^{K} \frac{p(\mathbf{x}, \mathbf{z}_k)}{q_\phi(\mathbf{z}_k|\mathbf{x})}, \tag{5.32}$$

where $\mathbf{z}_k \sim q_\phi(\mathbf{z}_k|\mathbf{x})$. Notice that the logarithm is **outside** the expected value. As shown in [15], using importance weighting with sufficiently large $K$ gives a good estimate of the log-likelihood. In practice, $K$ is taken to be 512 or more if the computational budget allows.

**Enhancing VAEs: Better encoders** After introducing the idea of VAEs, many papers focused on proposing a flexible family of variational posteriors. The most prominent direction is based on utilizing conditional flow-based models [16–21]. We discuss this topic more in Sect. 5.4.2.

**Enhancing VAEs: Better decoders** VAEs allow using any neural network to parameterize the decoder. Therefore, we can use fully connected networks, fully convolutional networks, ResNets, or ARMs. For instance, in [22], a PixelCNN-based decoder was used utilized in a VAE.

**Enhancing VAEs: Better priors** As mentioned before, that could be a serious issue if there is a big mismatch between the aggregated posterior and the prior. There are many papers that try to alleviate this issue by using a multimodal prior mimicking the aggregated posterior (known as the VampPrior) [23], or a flow-based prior (e.g., [24, 25]), an ARM-based prior [26] or using an idea of resampling [27]. We present various priors in Sect. 5.4.1.

**Extending VAEs** Here, we present the unsupervised version of VAEs. However, there is no restriction to that, and we can introduce labels or other variables. In [28] a semi-supervised VAE was proposed. This idea was further extended to the concept of fair representations [29, 30]. In [30], the authors proposed a specific latent representation that allows domain generalization in VAEs. In [31] variational inference and the reparameterization trick were used for Bayesian Neural Nets. Blundell et al. [31] is not necessarily introducing a VAE, but a VAE-like way of dealing with Bayesian neural nets.

**VAEs for non-image data** So far, we explain everything on images. However, there is no restriction on that! In [11], a VAE was proposed to deal with sequential data (e.g., text). The encoder and the decoder were parameterized by LSTMs. An interesting application of the VAE framework was also presented in [32] where VAEs were used for molecular graph generation. In [26], the authors proposed a VAE-like for video compression.

**Different latent spaces** Typically, the Euclidean latent space is considered. However, the VAE framework allows us to think of other spaces. For instance, in [33, 34], a hyperspherical latent space was used, and in [35], the hyperbolic latent space was utilized. More details about hyperspherical VAEs can be found in Sect. 5.4.2.3. In [36], hyperspherical latent spaces were used for learning of geometrically meaningful representations via equivariant VAEs.

**Discrete latent spaces** We discuss the VAE framework with continuous latent variables. However, an interesting question is how to deal with discrete latent variables. The problem here is that we cannot use the reparameterization trick anymore. There are two potential solutions to that. First, a relaxation to the discrete variables could be used like the *Gumbel-Softmax trick* [37, 38]. Second, a method for a gradient estimation could be used [39].

**The posterior collapse** There were many ideas proposed to deal with the posterior collapse. For instance, [40] proposes to update variational posteriors more often than the decoder. In [41], a new architecture of the decoder is proposed by introducing *skip connections* to allow a better flow of information (thus, the gradients) in the decoder.

**Various perspectives on the objective** The core of the VAE is the ELBO. However, we can consider different objectives. For instance, [42] proposes an upper bound to the log-likelihood that is based on the chi-square divergence (CUBO). In [10], an information-theoretic perspective on the ELBO is presented. Higgins et al. [43] introduced the $\beta$-VAE where the regularization term is weighted by a fudge factor $\beta$. The objective does not correspond to the lower bound of the log-likelihood though.

**Deterministic regularized auto-encoders** We can take a look at the VAE and the objective, as mentioned before, and think of it as a regularized version of an auto-encoder with a stochastic encoder and a stochastic decoder. Ghosh et al. [44] "peeled off" VAEs from all stochasticity and indicated similarities between deterministic regularized auto-encoders and VAEs and highlighted potential issues with VAEs. Moreover, they brilliantly pointed out that even with deterministic encoders, due to the stochasticity of the empirical distribution, we can fit a model to the aggregated posterior. As a result, the deterministic (regularized) auto-encoder could be turned into a generative model by sampling from a model of the aggregated posterior, $p_\lambda(\mathbf{z})$, and then, deterministically, mapping $\mathbf{z}$ to the space of observable $\mathbf{x}$. In my opinion, this direction should be further explored, and an important question is whether we indeed need any stochasticity at all.

**Hierarchical VAEs** Very recently, there have been many VAEs with a deep, hierarchical structure of latent variables that achieved remarkable results! The most important ones are definitely BIVA [45], NVAE [46], and very deep VAEs [47]. Another interesting perspective on a deep, hierarchical VAE was presented in [25] where, additionally, a series of deterministic functions were used. We delve into that topic in Sect. 5.5.

**Adversarial auto-encoders** Another interesting perspective on VAEs is presented in [48]. Since learning the aggregated posterior as the prior is an important component mentioned in some papers (e.g., [23, 49]), a different approach would be to train the prior with an adversarial loss. Further, [48] presents various ideas on how auto-encoders could benefit from adversarial learning.

**Adversarial attacks** It is a well-known fact that VAEs are susceptible to adversarial attacks, i.e., a small change to the latent space results in enormous errors in reconstructions. There exist a possible remedy to that by applying MCMC technique at the inference time [50].

## 5.4 Improving Variational Auto-encoders

### 5.4.1 Priors

#### 5.4.1.1 Insights from Rewriting the ELBO

One of the crucial components of VAEs is the marginal distribution over $\mathbf{z}$'s. Now, we will take a closer look at this distribution, also called the *prior*. Before we start thinking about improving it, we inspect the ELBO one more time. We can write

ELBO as follows:

$$\mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}[\ln p(\mathbf{x})] \geq \mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}\left[\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}\left[\ln p_\theta(\mathbf{x}|\mathbf{z}) + \ln p_\lambda(\mathbf{z}) - \ln q_\phi(\mathbf{z}|\mathbf{x})\right]\right], \tag{5.33}$$

where we explicitly highlight the summation over training data, namely, the expected value with respect to $\mathbf{x}$'s from the empirical distribution $p_{data}(\mathbf{x}) = \frac{1}{N}\sum_{n=1}^{N}\delta(\mathbf{x}-\mathbf{x}_n)$, and $\delta(\cdot)$ is the Dirac delta.

The ELBO consists of two parts, namely, the reconstruction error:

$$RE \triangleq \mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}\left[\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}\left[\ln p_\theta(\mathbf{x}|\mathbf{z})\right]\right], \tag{5.34}$$

and the regularization term between the encoder and the prior:

$$\Omega \triangleq \mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}\left[\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}\left[\ln p_\lambda(\mathbf{z}) - \ln q_\phi(\mathbf{z}|\mathbf{x})\right]\right]. \tag{5.35}$$

Further, let us play a little bit with the regularization term $\Omega$:

$$\begin{aligned}
\Omega &= \mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}\left[\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}\left[\ln p_\lambda(\mathbf{z}) - \ln q_\phi(\mathbf{z}|\mathbf{x})\right]\right] && (5.36)\\
&= \int p_{data}(\mathbf{x}) \int q_\phi(\mathbf{z}|\mathbf{x})\left[\ln p_\lambda(\mathbf{z}) - \ln q_\phi(\mathbf{z}|\mathbf{x})\right] \mathrm{d}\mathbf{z}\,\mathrm{d}\mathbf{x} && (5.37)\\
&= \iint p_{data}(\mathbf{x}) q_\phi(\mathbf{z}|\mathbf{x})\left[\ln p_\lambda(\mathbf{z}) - \ln q_\phi(\mathbf{z}|\mathbf{x})\right] \mathrm{d}\mathbf{z}\,\mathrm{d}\mathbf{x} && (5.38)\\
&= \iint \frac{1}{N}\sum_n \delta(\mathbf{x}-\mathbf{x}_n) q_\phi(\mathbf{z}|\mathbf{x})\left[\ln p_\lambda(\mathbf{z}) - \ln q_\phi(\mathbf{z}|\mathbf{x})\right] \mathrm{d}\mathbf{z}\,\mathrm{d}\mathbf{x} && (5.39)\\
&= \int \frac{1}{N}\sum_{n=1}^{N} q_\phi(\mathbf{z}|\mathbf{x}_n)\left[\ln p_\lambda(\mathbf{z}) - \ln q_\phi(\mathbf{z}|\mathbf{x}_n)\right] \mathrm{d}\mathbf{z} && (5.40)\\
&= \int \frac{1}{N}\sum_{n=1}^{N} q_\phi(\mathbf{z}|\mathbf{x}_n) \ln p_\lambda(\mathbf{z})\mathrm{d}\mathbf{z} - \int \frac{1}{N}\sum_{n=1}^{N} q_\phi(\mathbf{z}|\mathbf{x}_n) \ln q_\phi(\mathbf{z}|\mathbf{x}_n)\mathrm{d}\mathbf{z} && (5.41)\\
&= \int q_\phi(\mathbf{z}) \ln p_\lambda(\mathbf{z})\mathrm{d}\mathbf{z} - \int \sum_{n=1}^{N}\frac{1}{N} q_\phi(\mathbf{z}|\mathbf{x}_n) \ln q_\phi(\mathbf{z}|\mathbf{x}_n)\mathrm{d}\mathbf{z} && (5.42)\\
&= -\mathbb{CE}\left[q_\phi(\mathbf{z})||p_\lambda(\mathbf{z})\right] + \mathbb{H}\left[q_\phi(\mathbf{z}|\mathbf{x})\right], && (5.43)
\end{aligned}$$

where we use the property of the Dirac delta: $\int \delta(a-a')f(a)\mathrm{d}a = f(a')$, and we use the notion of the **aggregated posterior** [48, 49] defined as follows:

$$q(\mathbf{z}) = \frac{1}{N}\sum_{n=1}^{N} q_\phi\left(\mathbf{z}|\mathbf{x}_n\right). \tag{5.44}$$

An example of the aggregated posterior is schematically depicted in Fig. 5.5.

Eventually, we obtain two terms:

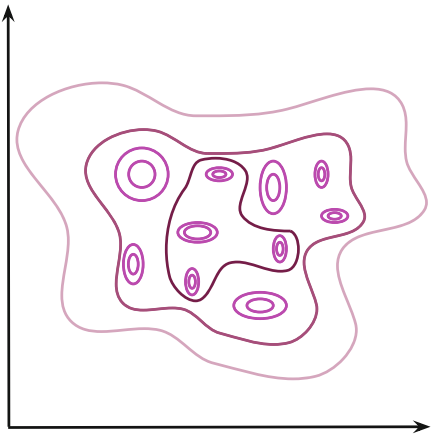


**Fig. 5.5** An example of the aggregated posterior. Individual points are encoded as Gaussians in the 2D latent space (magenta) and the mixture of variational posteriors (the aggregated posterior) is presented by contours.

(i) the first one, $\mathbb{CE}\left[q_\phi(\mathbf{z})||p_\lambda(\mathbf{z})\right]$, is the cross-entropy between the aggregated posterior and the prior,
(ii)the second term, $\mathbb{H}\left[q_\phi(\mathbf{z}|\mathbf{x})\right]$, is the conditional entropy of $q_\phi(\mathbf{z}|\mathbf{x})$ with the empirical distribution $p_{data}(\mathbf{x})$.

I highly recommend doing this derivation step by step; it helps a lot in understanding what is going on here. Interestingly, there is another possibility to rewrite $\Omega$ using three terms, with the total correlation [51]. We will not use it here, so it is left as a "homework."

Anyway, one may ask why is it useful to rewrite the ELBO? The answer is rather straightforward: We can analyze it from a different perspective! In this section, we will focus on the **prior**, an important component in the generative part that is very often neglected. Many Bayesianists are stating that a prior should not be learned. But VAEs are not Bayesian models; please remember that! Besides, who says we cannot learn the prior? As we will see shortly, a non-learnable prior could be pretty annoying, especially for the generation process.

#### 5.4.1.2 What Does the ELBO Tell Us About the Prior?

Alright, we see that $\Omega$ consists of the cross-entropy and the entropy. Let us start with the entropy, since it is easier to be analyzed. While optimizing, we want to maximize the ELBO; hence, we maximize the entropy:

$$\mathbb{H}\left[q_\phi(\mathbf{z}|\mathbf{x})\right] = -\int \sum_{n=1}^{N} \frac{1}{N} q_\phi(\mathbf{z}|\mathbf{x}_n) \ln q_\phi(\mathbf{z}|\mathbf{x}_n) \mathrm{d}\mathbf{z}. \tag{5.45}$$

Before we make any conclusions, we should remember that we consider Gaussian encoders, $q_\phi(\mathbf{z}|\mathbf{x}) = \mathcal{N}\left(\mathbf{z}|\mu(\mathbf{x}), \sigma^2(\mathbf{x})\right)$. The entropy of a Gaussian distribution with a diagonal covariance matrix is equal $\frac{1}{2}\sum_i \ln\left(2e\pi\sigma_i^2\right)$. Then, the question is when this quantity is maximized. The answer is: $\sigma_i^2 \rightarrow +\infty$. In other words, the entropy terms tries to stretch the encoders as much as possible by enlarging their variances! Of course, this does not happen in practice, because we use the encoder together with the decoder in the *RE* term, and the decoder tries to make the encoder as peaky as possible (i.e., ideally one $\mathbf{x}$ for one $\mathbf{z}$, like in the non-stochastic auto-encoder).

The second term in $\Omega$ is the cross-entropy:

$$\mathbb{CE}\left[q_\phi(\mathbf{z})||p_\lambda(\mathbf{z})\right] = -\int q_\phi(\mathbf{z}) \ln p_\lambda(\mathbf{z}) \mathrm{d}\mathbf{z}. \tag{5.46}$$

The cross-entropy term influences the VAE in a different manner. First, we can ask the question of how to interpret the cross-entropy between $q_\phi(\mathbf{z})$ and $p_\lambda(\mathbf{z})$. In general, the cross-entropy tells us the average number of bits (or rather nats because we use the natural logarithm) needed to identify an event drawn from $q_\phi(\mathbf{z})$ if a coding scheme used for it is $p_\lambda(\mathbf{z})$. Notice that in $\Omega$, we have the negative cross-entropy. Since we maximize the ELBO, it means we aim for minimizing $\mathbb{CE}\left[q_\phi(\mathbf{z})||p_\lambda(\mathbf{z})\right]$. This makes sense, because we would like $q_\phi(\mathbf{z})$ to match $p_\lambda(\mathbf{z})$. And we have accidentally touched upon the most important issue here: What do we really want here? The cross-entropy forces the aggregated posterior to **match** the prior! That is the reason why we have this term here. If you think about it, it is a beautiful result that gives another connection between VAEs and information theory.

Alright, so we see what the cross-entropy does, but there are two possibilities here. First, the prior is fixed (**non-learnable**), e.g., the standard Gaussian prior. Then, optimizing the cross-entropy *pushes* the aggregated posterior to match the prior. It is schematically depicted in Fig. 5.6. The prior acts like an anchor, and the *amoeba* of the aggregated posterior moves so that to fit the prior. In practice, this optimization process is troublesome, because the decoded forces the encoder to be peaked and, in the end, it is almost impossible to match a fixed-shaped prior. As a result, we obtain **holes**, namely, regions in the latent space where the aggregated posterior assigns low probability, while the prior assigns (relatively) high probability (see a dark gray ellipse in Fig. 5.6). This issue is especially apparent in generations, because sampling from the prior, from the hole, may result in a sample that is of an extremely low quality. You can read more about it in [12].

On the other hand, if we consider a learnable **prior**, the situation looks different. The optimization allows changing the aggregated posterior **and** the prior. As a consequence, both distributions try to match each other (see Fig. 5.7). The problem of holes is then less apparent, especially if the prior is flexible enough. However, we can face other optimization issues when the prior and the aggregated posteriors chase each other. In practice, the learnable prior seems to be a better option, but it is still an open question whether training all components at once is the best approach. Moreover, the learnable prior does not impose any specific constraint on the latent

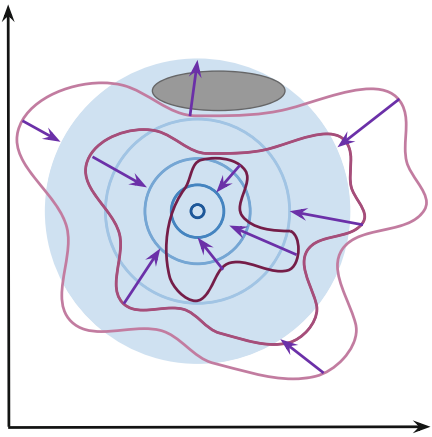


**Fig. 5.6** An example of the effect of the cross-entropy optimization with a non-learnable prior. The aggregated posterior (purple contours) tries to match the non-learnable prior (in blue). The purple arrows indicate the change of the aggregated posterior. An example of a hole is presented as a dark gray ellipse.

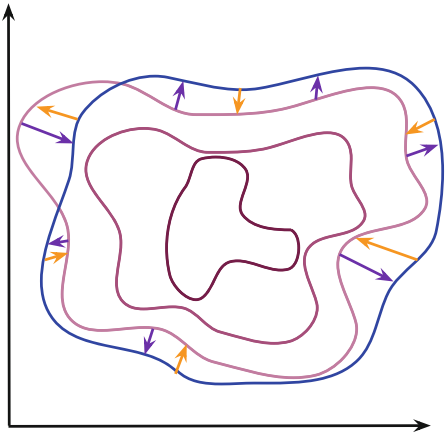


**Fig. 5.7** An example of the effect of the cross-entropy optimization with a learnable prior. The aggregated posterior (purple contours) tries to match the learnable prior (blue contours). Notice that the aggregated posterior is modified to fit the prior (purple arrows), but also the prior is updated to cover the aggregated posterior (orange arrows).

representation, e.g., sparsity. This could be another problem that would result in undesirable problems (e.g., non-smooth encoders).

Eventually, we can ask the fundamental question: What is the *best* prior then?! The answer is already known and is hidden in the cross-entropy term: It is the aggregated posterior. If we take $p_\lambda(\mathbf{z}) = \sum_{n=1}^{N} \frac{1}{N} q_\phi(\mathbf{z}|\mathbf{x}_n)$, then, theoretically, the cross-entropy equals the entropy of $q_\phi(\mathbf{z})$, and the regularization term $\Omega$ is smallest. However, in practice, this is infeasible because:

- we cannot sum over tens of thousands of points and backpropagate through them,
- this result is fine from the theoretical point of view; however, the optimization process is stochastic and could cause additional errors,
- as mentioned earlier, choosing the aggregated posterior as the prior does not constrain the latent representation in any obvious manner, and, thus, the encoder could behave unpredictably,
- the aggregated posterior may work well if it gets $N \rightarrow +\infty$ points, because then we can get any distribution; however, this is not the case in practice, and it contradicts also the first bullet.

As a result, we can keep this theoretical solution in mind and formulate **approximations** to it that are computationally tractable. In the next sections, we will discuss a few of them.

#### 5.4.1.3 Standard Gaussian

The vanilla implementation of the VAE assumes a standard Gaussian marginal (prior) over $\mathbf{z}$, $p_\lambda(\mathbf{z}) = \mathcal{N}(\mathbf{z}|0, \mathrm{I})$. This prior is simple, non-trainable (i.e., no extra parameters to learn), and easy to implement. In other words, it is amazing! However, as discussed above, the standard normal distribution could lead to very poor hidden representations with holes resulting from the mismatch between the aggregated posterior and the prior.

To strengthen our discussion, we trained a small VAE with the standard Gaussian prior and two-dimensional latent space. In Fig. 5.8, we present samples from the encoder for the test data (black dots) and the contour plot for the standard prior. We can spot holes where the aggregated posterior does not assign any points (i.e., the mismatch between the prior and the aggregated posterior).

The code for the standard Gaussian prior is presented below.

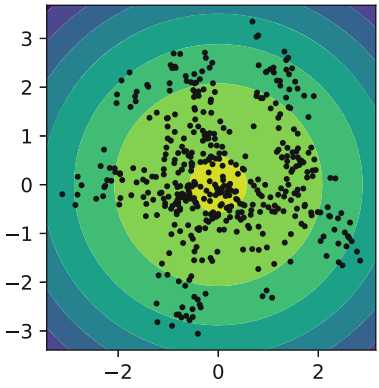



**Fig. 5.8** An example of the standard Gaussian prior (contours) and the samples from the aggregated posterior (black dots).

```
class StandardPrior(nn.Module):
    def __init__(self, L=2):
        super(StandardPrior, self).__init__()

        self.L = L

        # params weights
        self.means = torch.zeros(1, L)
        self.logvars = torch.zeros(1, L)

    def get_params(self):
        return self.means, self.logvars

    def sample(self, batch_size):
        return torch.randn(batch_size, self.L)

    def log_prob(self, z):
        return log_standard_normal(z)
```

**Listing 5.6** A standard Gaussian prior class.

#### 5.4.1.4 Mixture of Gaussians

If we take a closer look at the aggregated posterior, we immediately notice that it is a mixture model, and a mixture of Gaussians, to be more precise. Therefore, we can use the mixture of Gaussians (MoG) prior with $K$ components:

$$p_\lambda(\mathbf{z}) = \sum_{k=1}^{K} w_k \mathcal{N}(\mathbf{z}|\mu_k, \sigma_k^2), \tag{5.47}$$

where $\lambda = \left\{\{w_k\}, \{\mu_k\}, \{\sigma_k^2\}\right\}$ are trainable parameters.

Similarly to the standard Gaussian prior, we trained a small VAE with the mixture of Gaussians prior (with $K = 16$) and two-dimensional latent space. In Fig. 5.9, we present samples from the encoder for the test data (black dots) and the contour plot for the MoG prior. Compared to the standard Gaussian prior, the MoG prior fits better the aggregated posterior, allowing to *patch* holes.

An example of the code is presented below.

```
class MoGPrior(nn.Module):
    def __init__(self, L, num_components):
        super(MoGPrior, self).__init__()

        self.L = L
        self.num_components = num_components

        # params
        self.means = nn.Parameter(torch.randn(num_components,
    self.L)*multiplier)
```

```
        self.logvars = nn.Parameter(torch.randn(num_components, self.L))

        # mixing weights
        self.w = nn.Parameter(torch.zeros(num_components, 1, 1))

    def get_params(self):
        return self.means, self.logvars

    def sample(self, batch_size):
        # mu, lof_var
        means, logvars = self.get_params()

        # mixing probabilities
        w = F.softmax(self.w, dim=0)
        w = w.squeeze()

        # pick components
        indexes = torch.multinomial(w, batch_size, replacement=True)

        # means and logvars
        eps = torch.randn(batch_size, self.L)
        for i in range(batch_size):
            indx = indexes[i]
            if i == 0:
                z = means[[indx]] + eps[[i]] * torch.exp(logvars[[indx]])
            else:
                z = torch.cat((z, means[[indx]] + eps[[i]] * torch.exp(logvars[[indx]])), 0)
        return z

    def log_prob(self, z):
        # mu, lof_var
        means, logvars = self.get_params()

        # mixing probabilities
        w = F.softmax(self.w, dim=0)

        # log-mixture-of-Gaussians
        z = z.unsqueeze(0) # 1 x B x L
        means = means.unsqueeze(1) # K x 1 x L
        logvars = logvars.unsqueeze(1) # K x 1 x L

        log_p = log_normal_diag(z, means, logvars) + torch.log(w) # K x B x L
        log_prob = torch.logsumexp(log_p, dim=0, keepdim=False) # B x L

        return log_prob
```

**Listing 5.7** A mixture of Gaussians prior class.

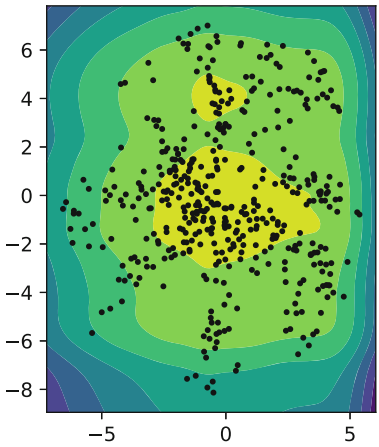



**Fig. 5.9** An example of the MoG prior (contours) and the samples from the aggregated posterior (black dots).

#### 5.4.1.5 VampPrior: Variational Mixture of Posteriors Prior

In [23], it was noticed that we can improve on the MoG prior and approximate the aggregated posterior by introducing *pseudo-inputs*:

$$p_\lambda(\mathbf{z}) = \frac{1}{K}\sum_{k=1}^{K} q_\phi(\mathbf{z}|\mathbf{u}_k), \tag{5.48}$$

where $\lambda = \left\{\phi, \{\mathbf{u}_k^2\}\right\}$ are trainable parameters and $\mathbf{u}_k \in \mathcal{X}^D$ is a pseudo-input. Notice that $\phi$ is a part of the trainable parameters. The idea of pseudo-input is to randomly initialize objects that mimic observable variables (e.g., images) and learn them by backpropagation.

This approximation to the aggregated posterior is called the **variational mixture of posterior prior**, VampPrior for short. In [23], you can find some interesting properties and further analysis of the Vampprior. The main drawback of the VampPrior lies in initializing the pseudo-inputs; however, it serves as a good proxy to the aggregated posterior that improves the generative quality of the VAE, e.g., [10, 52, 53].

Alemi et al. [10] presented a nice connection of the VampPrior with the information-theoretic perspective on the VAE. They further proposed to introduce learnable probabilities of the components:

$$p_\lambda(\mathbf{z}) = \sum_{k=1}^{K} w_k q_\phi(\mathbf{z}|\mathbf{u}_k), \tag{5.49}$$

to allow the VampPrior to select more relevant components (i.e., pseudo-inputs).

As in the previous cases, we train a small VAE with the VampPrior (with $K = 16$) and two-dimensional latent space. In Fig. 5.10, we present samples from the encoder

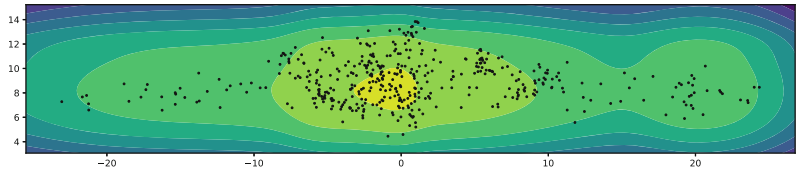



**Fig. 5.10** An example of the VampPrior (contours) and the samples from the aggregated posterior (black dots).

for the test data (black dots) and the contour plot for the VampPrior. Similar to the MoG prior, the VampPrior fits better the aggregated posterior and has fewer holes. In this case, we can see that the VampPrior allows the encoders to spread across the latent space (notice the values).

An example of an implementation of the VampPrior is presented below.

```
class VampPrior(nn.Module):
    def __init__(self, L, D, num_vals, encoder, num_components,
    data=None):
        super(VampPrior, self).__init__()

        self.L = L
        self.D = D
        self.num_vals = num_vals

        self.encoder = encoder

        # pseudoinputs
        u = torch.rand(num_components, D) * self.num_vals
        self.u = nn.Parameter(u)

        # mixing weights
        self.w = nn.Parameter(torch.zeros(self.u.shape[0], 1, 1))
     # K x 1 x 1

    def get_params(self):
        # u->encoder->mu, lof_var
        mean_vampprior, logvar_vampprior = self.encoder.encode(
    self.u) #(K x L), (K x L)
        return mean_vampprior, logvar_vampprior

    def sample(self, batch_size):
        # u->encoder->mu, lof_var
        mean_vampprior, logvar_vampprior = self.get_params()

        # mixing probabilities
        w = F.softmax(self.w, dim=0) # K x 1 x 1
        w = w.squeeze()

        # pick components
        indexes = torch.multinomial(w, batch_size, replacement=
    True)
```

```
33
34          # means and logvars
35          eps = torch.randn(batch_size, self.L)
36          for i in range(batch_size):
37              indx = indexes[i]
38              if i == 0:
39                  z = mean_vampprior[[indx]] + eps[[i]] * torch.exp
    (logvar_vampprior[[indx]])
40              else:
41                  z = torch.cat((z, mean_vampprior[[indx]] + eps[[i
    ]] * torch.exp(logvar_vampprior[[indx]])), 0)
42          return z
43
44      def log_prob(self, z):
45          # u->encoder->mu, lof_var
46          mean_vampprior, logvar_vampprior = self.get_params() # (K
     x L) & (K x L)
47
48          # mixing probabilities
49          w = F.softmax(self.w, dim=0) # K x 1 x 1
50
51          # log-mixture-of-Gaussians
52          z = z.unsqueeze(0) # 1 x B x L
53          mean_vampprior = mean_vampprior.unsqueeze(1) # K x 1 x L
54          logvar_vampprior = logvar_vampprior.unsqueeze(1) # K x 1
    x L
55
56          log_p = log_normal_diag(z, mean_vampprior,
    logvar_vampprior) + torch.log(w) # K x B x L
57          log_prob = torch.logsumexp(log_p, dim=0, keepdim=False) #
     B x L
58
59          return log_prob
```

**Listing 5.8** A VampPrior class.

#### 5.4.1.6 GTM: Generative Topographic Mapping

In fact, we can use any density estimator to model the prior. In [54] a density estimator called **generative topographic mapping** (GTM) was proposed that defines a grid of $K$ points in a low-dimensional space, $\mathbf{u} \in \mathbb{R}^C$, namely:

$$p(\mathbf{u}) = \sum_{k=1}^{K} w_k \delta(\mathbf{u} - \mathbf{u}_k) \tag{5.50}$$

that is further transformed to a higher-dimensional space by a transformation $g_\gamma$. The transformation $g_\gamma$ predicts parameters of a distribution, e.g., the Gaussian distribution, thus, $g_\gamma : \mathbb{R}^C \rightarrow \mathbb{R}^{2\times M}$. Eventually, we can define the distribution as follows:

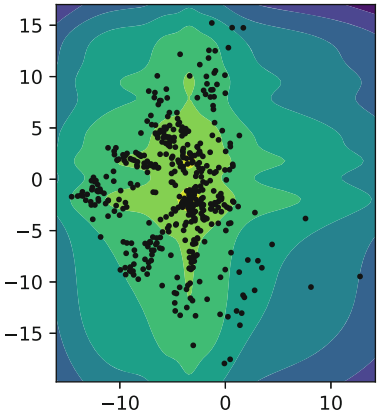



**Fig. 5.11** An example of the GTM-based prior (contours) and the samples from the aggregated posterior (black dots).

$$p_\lambda(\mathbf{z}) = \int p(\mathbf{u}) \mathcal{N}\left(\mathbf{z}|\mu_g(\mathbf{u}), \sigma_g^2(\mathbf{u})\right) \mathrm{d}\mathbf{u} \tag{5.51}$$

$$= \sum_{k=1}^{K} w_k \mathcal{N}\left(\mathbf{z}|\mu_g(\mathbf{u}_k), \sigma_g^2(\mathbf{u}_k)\right), \tag{5.52}$$

where $\mu_g(\mathbf{u})$ and $\sigma_g^2$ rare outputs of the transformation $g_\gamma(\mathbf{u})$.

For instance, for $C = 2$ and $K = 3$, we can define the following grid: $\mathbf{u} \in \{[-1,-1],[-1,0],[-1,1],[0,-1],[0,1],[1,-1],[1,0],[1,-1]\}$. Notice that the grid is fixed and only the transformation (e.g., a neural network) $g_\gamma$ is trained.

As in the previous cases, we train a small VAE with the GTM-based prior (with $K = 16$, i.e., a $4\times4$ grid) and a two-dimensional latent space. In Fig. 5.11, we present samples from the encoder for the test data (black dots) and the contour plot for the GTM-based prior. Similar to the MoG prior and the VampPrior, the GTM-based prior learns a pretty flexible distribution.

An example of an implementation of the GTM-based prior is presented below.

```
class GTMPrior(nn.Module):
    def __init__(self, L, gtm_net, num_components, u_min=-1.,
    u_max=1.):
    super(GTMPrior, self).__init__()

    self.L = L

    # 2D manifold
    self.u = torch.zeros(num_components**2, 2) # K**2 x 2
    u1 = torch.linspace(u_min, u_max, steps=num_components)
    u2 = torch.linspace(u_min, u_max, steps=num_components)

    k = 0
    for i in range(num_components):
        for j in range(num_components):
```

```
            self.u[k,0] = u1[i]
            self.u[k,1] = u2[j]
            k = k + 1

    # gtm network: u -> z
    self.gtm_net = gtm_net

    # mixing weights
    self.w = nn.Parameter(torch.zeros(num_components**2, 1, 1))

    def get_params(self):
        # u->z
        h_gtm = self.gtm_net(self.u) #K**2 x 2L
        mean_gtm, logvar_gtm = torch.chunk(h_gtm, 2, dim=1) # K
    **2 x L and K**2 x L
        return mean_gtm, logvar_gtm

    def sample(self, batch_size):
        # u->z
        mean_gtm, logvar_gtm = self.get_params()

        # mixing probabilities
        w = F.softmax(self.w, dim=0)
        w = w.squeeze()

        # pick components
        indexes = torch.multinomial(w, batch_size, replacement=
    True)

        # means and logvars
        eps = torch.randn(batch_size, self.L)
        for i in range(batch_size):
            indx = indexes[i]
            if i == 0:
                z = mean_gtm[[indx]] + eps[[i]] * torch.exp(
    logvar_gtm[[indx]])
            else:
                z = torch.cat((z, mean_gtm[[indx]] + eps[[i]] *
    torch.exp(logvar_gtm[[indx]])), 0)
        return z

    def log_prob(self, z):
        # u->z
        mean_gtm, logvar_gtm = self.get_params()

        # log-mixture-of-Gaussians
        z = z.unsqueeze(0) # 1 x B x L
        mean_gtm = mean_gtm.unsqueeze(1) # K**2 x 1 x L
        logvar_gtm = logvar_gtm.unsqueeze(1) # K**2 x 1 x L

        w = F.softmax(self.w, dim=0)

        log_p = log_normal_diag(z, mean_gtm, logvar_gtm) + torch.
    log(w) # K**2 x B x L
```

```
        log_prob = torch.logsumexp(log_p, dim=0, keepdim=False) # B x L

        return log_prob
```

**Listing 5.9** A GTM-based prior class.

#### 5.4.1.7 GTM-VampPrior

As mentioned earlier, the main issue with the VampPrior is the initialization of the pseudo-inputs. Instead, we can use the idea of the GTM to learn the pseudo-inputs. Combining these two approaches, we get the following prior:

$$p_\lambda(\mathbf{z}) = \sum_{k=1}^{K} w_k q_\phi \left(\mathbf{z}|g_\gamma(\mathbf{u}_k)\right), \tag{5.53}$$

where we first define a grid in a low-dimensional space, $\{\mathbf{u}_k\}$, and then transform them to $\mathcal{X}^D$ using the transformation $g_\gamma$.

Now, we train a small VAE with the GTM-VampPrior (with $K = 16$, i.e., a $4 \times 4$ grid) and a two-dimensional latent space. In Fig. 5.12, we present samples from the encoder for the test data (black dots) and the contour plot for the GTM-VampPrior. Again, this mixture-based prior allows wrapping the points (the aggregated posterior) and assigning the probability to proper regions.

An example of an implementation of the GTM-VampPrior is presented below.

```
class GTMVampPrior(nn.Module):
    def __init__(self, L, D, gtm_net, encoder, num_points, u_min
    =-10., u_max=10., num_vals=255):
    super(GTMVampPrior, self).__init__()

    self.L = L
    self.D = D
    self.num_vals = num_vals

    self.encoder = encoder

    # 2D manifold
    self.u = torch.zeros(num_points**2, 2) # K**2 x 2
    u1 = torch.linspace(u_min, u_max, steps=num_points)
    u2 = torch.linspace(u_min, u_max, steps=num_points)

    k = 0
    for i in range(num_points):
        for j in range(num_points):
            self.u[k,0] = u1[i]
            self.u[k,1] = u2[j]
            k = k + 1

    # gtm network: u -> x
```

```
    self.gtm_net = gtm_net

    # mixing weights
    self.w = nn.Parameter(torch.zeros(num_points**2, 1, 1))

    def get_params(self):
        # u->gtm_net->u_x
        h_gtm = self.gtm_net(self.u) #K x D
        h_gtm = h_gtm * self.num_vals
        # u_x->encoder->mu, lof_var
        mean_vampprior, logvar_vampprior = self.encoder.encode(
    h_gtm) #(K x L), (K x L)
        return mean_vampprior, logvar_vampprior

    def sample(self, batch_size):
    # u->encoder->mu, lof_var
    mean_vampprior, logvar_vampprior = self.get_params()

    # mixing probabilities
    w = F.softmax(self.w, dim=0)
    w = w.squeeze()

        # pick components
        indexes = torch.multinomial(w, batch_size, replacement=
    True)

        # means and logvars
        eps = torch.randn(batch_size, self.L)
        for i in range(batch_size):
            indx = indexes[i]
            if i == 0:
                z = mean_vampprior[[indx]] + eps[[i]] * torch.exp
    (logvar_vampprior[[indx]])
            else:
                z = torch.cat((z, mean_vampprior[[indx]] + eps[[i
    ]] * torch.exp(logvar_vampprior[[indx]])), 0)
        return z

    def log_prob(self, z):
        # u->encoder->mu, lof_var
        mean_vampprior, logvar_vampprior = self.get_params()

        # mixing probabilities
        w = F.softmax(self.w, dim=0)

        # log-mixture-of-Gaussians
        z = z.unsqueeze(0) # 1 x B x L
        mean_vampprior = mean_vampprior.unsqueeze(1) # K x 1 x L
        logvar_vampprior = logvar_vampprior.unsqueeze(1) # K x 1
    x L

        log_p = log_normal_diag(z, mean_vampprior,
    logvar_vampprior) + torch.log(w) # K x B x L
```

```
        log_prob = torch.logsumexp(log_p, dim=0, keepdim=False) # B x L

        return log_prob
```

**Listing 5.10** A GTM-VampPrior prior class.

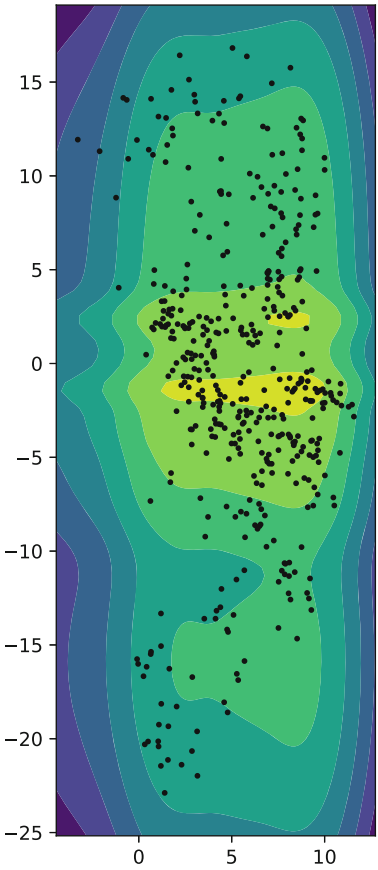



**Fig. 5.12** An example of the GTM-VampPrior (contours) and the samples from the aggregated posterior (black dots).

#### 5.4.1.8 Flow-Based Prior

The last distribution we want to discuss here is a flow-based prior. Since flow-based models can be used to estimate any distribution, it is almost obvious to use them for approximating the aggregated posterior. Here, we use the implementation of the RealNVP presented before (see Chap. 4 for details).

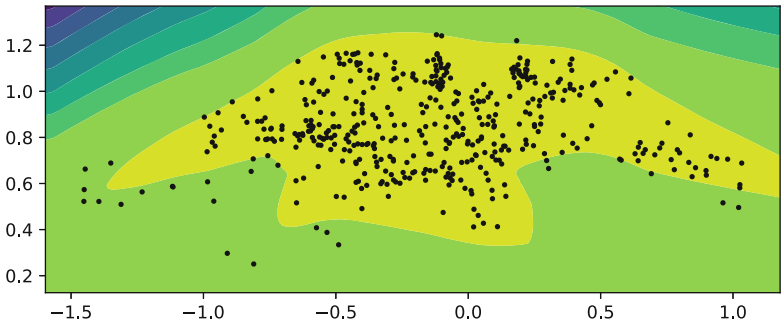



**Fig. 5.13** An example of the flow-based prior (contours) and the samples from the aggregated posterior (black dots).

As in the previous cases, we train a small VAE with the flow-based prior and two-dimensional latent space. In Fig. 5.13, we present samples from the encoder for the test data (black dots) and the contour plot for the flow-based prior. Similar to the previous mixture-based priors, the flow-based prior allows approximating the aggregated posterior very well. This is in line with many papers using flows as the prior in the VAE [24, 25]; however, we must remember that the flexibility of the flow-based prior comes with the cost of an increased number of parameters and potential training issues inherited from the flows.

An example of an implementation of the flow-based prior is presented below.

```
class FlowPrior(nn.Module):
    def __init__(self, nets, nett, num_flows, D=2):
        super(FlowPrior, self).__init__()

        self.D = D

        self.t = torch.nn.ModuleList([nett() for _ in range(
    num_flows)])
        self.s = torch.nn.ModuleList([nets() for _ in range(
    num_flows)])
        self.num_flows = num_flows

    def coupling(self, x, index, forward=True):
        (xa, xb) = torch.chunk(x, 2, 1)

        s = self.s[index](xa)
        t = self.t[index](xa)

        if forward:
            #yb = f^{-1}(x)
            yb = (xb - t) * torch.exp(-s)
        else:
            #xb = f(y)
            yb = torch.exp(s) * xb + t

        return torch.cat((xa, yb), 1), s
```

```
    def permute(self, x):
        return x.flip(1)

    def f(self, x):
        log_det_J, z = x.new_zeros(x.shape[0]), x
        for i in range(self.num_flows):
            z, s = self.coupling(z, i, forward=True)
            z = self.permute(z)
            log_det_J = log_det_J - s.sum(dim=1)

        return z, log_det_J

    def f_inv(self, z):
        x = z
        for i in reversed(range(self.num_flows)):
            x = self.permute(x)
            x, _ = self.coupling(x, i, forward=False)

        return x

    def sample(self, batch_size):
        z = torch.randn(batch_size, self.D)
        x = self.f_inv(z)
        return x.view(-1, self.D)

    def log_prob(self, x):
        z, log_det_J = self.f(x)
        log_p = (log_standard_normal(z) + log_det_J.unsqueeze(1))
        return log_p
```

**Listing 5.11** A flow-based prior class.

#### 5.4.1.9 Remarks

In practice, we can use any density estimator to model $p_\lambda(\mathbf{z})$. For instance, we can use an autoregressive model [26] or more advanced approaches like resampled priors [27] or hierarchical priors [53]. Therefore, there are many options! However, there is still an open question **how** to do that and **what** role the prior (the marginal) should play. As I mentioned in the beginning, Bayesianists would say that the marginal should impose some constraints on the latent space or, in other words, our prior knowledge about it. I am a Bayesiast deep down in my heart, and this way of thinking is very appealing to me. However, it is still unclear what is a good latent representation. This question is as old as mathematical modeling. I think that it would be interesting to look at optimization techniques, maybe applying a gradient-based method to all parameters/weights at once is not the best solution. Anyhow, I am pretty sure that modeling the prior is more important than many people think and plays a crucial role in VAEs.

### 5.4.2 Variational Posteriors

In general, variational inference searches for the best posterior approximation within a parametric family of distributions. Hence, recovering the true posterior is possible only if it happens to be in the chosen family. In particular, with widely used variational families such as diagonal covariance Gaussian distributions, the variational approximation is likely to be insufficient. Therefore, designing tractable and more expressive variational families is an important problem in VAEs. Here, we present two families of conditional normalizing flows that can be used for that purpose, namely, Householder flows [20] and Sylvester flows [16]. There are other interesting families, and we refer the reader to the original papers, e.g., the generalized Sylvester flows [17] and the inverse autoregressive flows [18].

The general idea about using the normalizing flows to parameterize the variational posteriors is to start with a relatively simple distribution like the Gaussian with the diagonal covariance matrix and then transform it to a complex distribution through a series of invertible transformations [19]. Formally speaking, we start with the latents $\mathbf{z}^{(0)}$ distributed according to $\mathcal{N}(\mathbf{z}^{(0)}|\mu(\mathbf{x}, \sigma^2(\mathbf{x}))$ and then, after applying a series of invertible transformations $\mathbf{f}^{(t)}$, for $t = 1, \ldots, T$, the last iterate gives a random variable $\mathbf{z}^{(T)}$ that has a more flexible distribution. Once we choose transformations $\mathbf{f}^{(t)}$ for which the Jacobian determinant can be computed, we aim at optimizing the following objective:

$$\ln p(\mathbf{x}) \geq \mathbb{E}_{q(\mathbf{z}^{(0)}|\mathbf{x})}\left[ \ln p(\mathbf{x}|\mathbf{z}^{(T)}) + \sum_{t=1}^{T} \ln \left| \det \frac{\partial \mathbf{f}^{(t)}}{\partial \mathbf{z}^{(t-1)}} \right| \right] - \mathrm{KL}\big(q(\mathbf{z}^{(0)}|\mathbf{x})||p(\mathbf{z}^{(T)})\big). \tag{5.54}$$

In fact, the normalizing flow can be used to enrich the posterior of the VAE with small or even no modifications in the architecture of the encoder and the decoder.

#### 5.4.2.1 Variational Posteriors with Householder Flows [20]

Motivation

First, we notice that any full-covariance matrix $\mathbf{\Sigma}$ can be represented by the eigenvalue decomposition using eigenvectors and eigenvalues:

$$\mathbf{\Sigma} = \mathbf{U}\mathbf{D}\mathbf{U}^{\top}, \tag{5.55}$$

where $\mathbf{U}$ is an orthogonal matrix with eigenvectors in columns and $\mathbf{D}$ is a diagonal matrix with eigenvalues. In the case of the vanilla VAE, it would be tempting to model the matrix $\mathbf{U}$ to obtain a full-covariance matrix. The procedure would require a linear transformation of a random variable using an orthogonal matrix $\mathbf{U}$. Since the absolute value of the Jacobian determinant of an orthogonal matrix is 1, for $\mathbf{z}^{(1)} = \mathbf{U}\mathbf{z}^{(0)}$, one gets $\mathbf{z}^{(1)} \sim \mathcal{N}(\mathbf{U}\boldsymbol{\mu}, \mathbf{U}\,\mathrm{diag}(\boldsymbol{\sigma}^2)\,\mathbf{U}^{\top})$. If $\mathrm{diag}(\boldsymbol{\sigma}^2)$ coincides with true

**D**, then it would be possible to resemble the true full-covariance matrix. Hence, the main goal would be to model the orthogonal matrix of eigenvectors.

Generally, the task of modeling an orthogonal matrix in a principled manner is rather nontrivial. However, first, we notice that any orthogonal matrix can be represented in the following form [55, 56]:

Theorem

**Theorem 5.1 (The Basis-Kernel Representation of Orthogonal Matrices)** *For any $M \times M$ orthogonal matrix $\mathbf{U}$ there exist a full-rank $M \times K$ matrix $\mathbf{Y}$ (the basis) and a nonsingular (triangular) $K \times K$ matrix $\mathbf{S}$ (the kernel), $K \leq M$, such that:*

$$\mathbf{U} = \mathbf{I} - \mathbf{Y}\mathbf{S}\mathbf{Y}^{\top}. \tag{5.56}$$

The value $K$ is called the *degree* of the orthogonal matrix. Further, it can be shown that any orthogonal matrix of degree $K$ can be expressed using the product of Householder transformations [55, 56], namely:

Theorem

**Theorem 5.2** *Any orthogonal matrix with the basis acting on the $K$-dimensional subspace can be expressed as a product of exactly $K$ Householder matrices:*

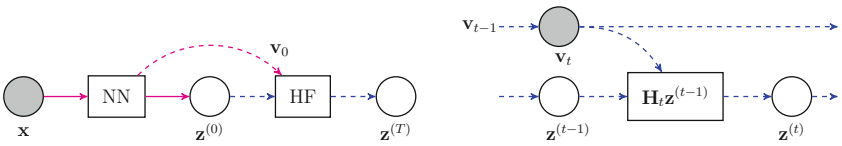
$$\mathbf{U} = \mathbf{H}_K \mathbf{H}_{K-1} \cdots \mathbf{H}_1, \tag{5.57}$$

*where $\mathbf{H}_k = \mathbf{I} - \mathbf{S}_{kk}\mathbf{Y}_{\cdot k}(\mathbf{Y}_{\cdot k})^{\top}$, for $k = 1, \ldots, K$.*

Theoretically, Theorem 5.2 shows that we can model any orthogonal matrix in a principled fashion using $K$ Householder transformations. Moreover, the Householder matrix $\mathbf{H}_k$ is *orthogonal* matrix itself [57]. Therefore, this property and the Theorem 5.2 put the Householder transformation as a perfect candidate for formulating a volume-preserving flow that allows to approximate (or even capture) the true full-covariance matrix.

### Householder Flows

The *Householder transformation* is defined as follows. For a given vector $\mathbf{z}^{(t-1)}$, the reflection hyperplane can be defined by a vector (a *Householder vector*) $\mathbf{v}_t \in \mathbb{R}^M$ that is orthogonal to the hyperplane, and the reflection of this point about the hyperplane is [57]:




(a) encoder network + Householder Flow (b) one step of the Householder Flow

**Fig. 5.14** A schematic representation of the encoder network with the Householder flow. (**a**) The general architecture of the VAE+HF: The encoder returns means and variances for the posterior and the first Householder vector that is further used to formulate the Householder flow. (**b**) A single step of the Householder flow that uses linear Householder transformation. In both figures, solid lines correspond to the encoder network, and the dashed lines are additional quantities required by the HF.

$$\mathbf{z}^{(t)} = \left(\mathbf{I} - 2\frac{\mathbf{v}_t\mathbf{v}_t^\top}{||\mathbf{v}_t||^2}\right)\mathbf{z}^{(t-1)} \tag{5.58}$$

$$= \mathbf{H}_t\mathbf{z}^{(t-1)}, \tag{5.59}$$

where $\mathbf{H}_t = \mathbf{I} - 2\frac{\mathbf{v}_t\mathbf{v}_t^\top}{||\mathbf{v}_t||^2}$ is called the *Householder matrix*.

The most important property of $\mathbf{H}_t$ is that it is an orthogonal matrix, and hence the absolute value of the Jacobian determinant is equal to 1. This fact significantly simplifies the objective (5.54) because $\ln\left|\det\frac{\partial\mathbf{H}_t\mathbf{z}^{(t-1)}}{\partial\mathbf{z}^{(t-1)}}\right| = 0$, for $t = 1, \ldots, T$. Starting from a simple posterior with the diagonal covariance matrix for $\mathbf{z}^{(0)}$, the series of $T$ linear transformations given by (5.58) defines a new type of volume-preserving flow that we refer to as the *Householder flow* (HF). The vectors $\mathbf{v}_t$, $t = 1, \ldots, T$, are produced by the encoder network along with means and variances using a linear layer with the input $\mathbf{v}_{t-1}$, where $\mathbf{v}_0 = \mathbf{h}$ is the last hidden layer of the encoder network. The idea of the Householder flow is schematically presented in Fig. 5.14. Once the encoder returns the first Householder vector, the Householder flow requires $T$ linear operations to produce a sample from a more flexible posterior with an approximate full-covariance matrix.

#### 5.4.2.2 Variational Posteriors with Sylvester Flows [16]

Motivation

The Householder flow can model only full-covariance Gaussians that is still not necessarily a rich family of distributions. Now, we will look into a generalization of the Householder flows. For this purpose, let us consider the following transformation similar to a single-layer MLP with $M$ hidden units and a residual connection:

$$\mathbf{z}^{(t)} = \mathbf{z}^{(t-1)} + \mathbf{A}h(\mathbf{B}\mathbf{z}^{(t-1)} + \mathbf{b}), \tag{5.60}$$

with $\mathbf{A} \in \mathbb{R}^{D\times M}, \mathbf{B} \in \mathbb{R}^{M\times D}$, $\mathbf{b} \in \mathbb{R}^{M}$, and $M \leq D$. The Jacobian determinant of this transformation can be obtained using *Sylvester's determinant identity*, which is a generalization of the matrix determinant lemma.

Theorem

**Theorem 5.3 (Sylvester's determinant identity)** *For all* $\mathbf{A} \in \mathbb{R}^{D\times M}, \mathbf{B} \in \mathbb{R}^{M\times D}$,

$$\det\left(\mathbf{I}_D + \mathbf{AB}\right) = \det\left(\mathbf{I}_M + \mathbf{BA}\right), \tag{5.61}$$

*where* $\mathbf{I}_M$ *and* $\mathbf{I}_D$ *are* $M$ *and* $D$*-dimensional identity matrices, respectively.*

When $M < D$, the computation of the determinant of a $D \times D$ matrix is thus reduced to the computation of the determinant of an $M \times M$ matrix.

Using Sylvester's determinant identity, the Jacobian determinant of the transformation in Eq. (5.60) is given by:

$$\det\left(\frac{\partial \mathbf{z}^{(t)}}{\partial \mathbf{z}^{(t-1)}}\right) = \det\left(\mathbf{I}_M + \text{diag}\left(h'(\mathbf{B}\mathbf{z}^{(t-1)} + \mathbf{b})\right)\mathbf{BA}\right). \tag{5.62}$$

Since Sylvester's determinant identity plays a crucial role in the proposed family of normalizing flows, we will refer to them as *Sylvester normalizing flows*.

Parameterization of $\mathbf{A}$ and $\mathbf{B}$

In general, the transformation in (5.60) will not be invertible. Therefore, we propose the following special case of the above transformation:

$$\mathbf{z}^{(t)} = \mathbf{z}^{(t-1)} + \mathbf{QR}h(\tilde{\mathbf{R}}\mathbf{Q}^T\mathbf{z}^{(t-1)} + \mathbf{b}), \tag{5.63}$$

where $\mathbf{R}$ and $\tilde{\mathbf{R}}$ are upper triangular $M \times M$ matrices, and

$$\mathbf{Q} = (\mathbf{q}_1 \ldots \mathbf{q}_M)$$

with the columns $\mathbf{q}_m \in \mathbb{R}^D$ forming an orthonormal set of vectors. By Theorem 5.3, the determinant of the Jacobian $\mathbf{J}$ of this transformation reduces to:

$$\begin{aligned}\det(\mathbf{J}) &= \det\left(\mathbf{I}_M + \text{diag}\left(h'(\tilde{\mathbf{R}}\mathbf{Q}^T\mathbf{z}^{(t-1)} + \mathbf{b})\right)\tilde{\mathbf{R}}\mathbf{Q}^T\mathbf{QR}\right) \\ &= \det\left(\mathbf{I}_M + \text{diag}\left(h'(\tilde{\mathbf{R}}\mathbf{Q}^T\mathbf{z}^{(t-1)} + \mathbf{b})\right)\tilde{\mathbf{R}}\mathbf{R}\right),\end{aligned} \tag{5.64}$$

which can be computed in $O(M)$, since $\tilde{\mathbf{R}}\mathbf{R}$ is also upper triangular. The following theorem gives a sufficient condition for this transformation to be invertible.

Theorem [16]

**Theorem 5.4** *Let* $\mathbf{R}$ *and* $\tilde{\mathbf{R}}$ *be upper triangular matrices. Let* $h : \mathbb{R} \longrightarrow \mathbb{R}$ *be a smooth function with bounded, positive derivative. Then, if the diagonal entries of* $\mathbf{R}$ *and* $\tilde{\mathbf{R}}$ *satisfy* $r_{ii}\tilde{r}_{ii} > -1/\|h'\|_\infty$ *and* $\tilde{\mathbf{R}}$ *is invertible, the transformation given by* (5.63) *is invertible.*

The proof of this theorem can be found in [16].

Preserving Orthogonality of $\mathbf{Q}$

Orthogonality is a convenient property, mathematically, but hard to achieve in practice. In this section, we consider three different flows based on the theorem above and various ways to preserve the orthogonality of $\mathbf{Q}$. The first two use explicit differentiable constructions of orthogonal matrices, while the third variant assumes a specific fixed permutation matrix as the orthogonal matrix.

**Orthogonal Sylvester flows.** First, we consider a Sylvester flow using matrices with $M$ orthogonal columns (O-SNF). In this flow, we can choose $M < D$ and thus introduce a flexible bottleneck. Similar to [58], we ensure orthogonality of $\mathbf{Q}$ by applying the following differentiable iterative procedure proposed by [59, 60]:

$$\mathbf{Q}^{(k+1)} = \mathbf{Q}^{(k)} \left( \mathbf{I} + \frac{1}{2} \left( \mathbf{I} - \mathbf{Q}^{(k)\top} \mathbf{Q}^{(k)} \right) \right). \tag{5.65}$$

with a sufficient condition for convergence given by $\|\mathbf{Q}^{(0)\top}\mathbf{Q}^{(0)} - \mathbf{I}\|_2 < 1$. Here, the 2-norm of a matrix $\mathbf{X}$ refers to $\|\mathbf{X}\|_2 = \lambda_{\max}(\mathbf{X})$, with $\lambda_{\max}(\mathbf{X})$ representing the largest singular value of $\mathbf{X}$. In our experimental evaluations, we ran the iterative procedure until $\|\mathbf{Q}^{(k)\top}\mathbf{Q}^{(k)} - \mathbf{I}\|_F \leq \epsilon$, with $\|\mathbf{X}\|_F$ the Frobenius norm, and $\epsilon$ a small convergence threshold. We observed that running this procedure up to 30 steps was sufficient to ensure convergence with respect to this threshold. To minimize the computational overhead introduced by orthogonalization, we perform this orthogonalization in parallel for all flows.

Since this orthogonalization procedure is differentiable, it allows for the calculation of gradients with respect to $\mathbf{Q}^{(0)}$ by backpropagation, allowing for any standard optimization scheme such as stochastic gradient descent to be used for updating the flow parameters.

**Householder Sylvester flows.** Second, we study Householder Sylvester flows (H-SNF) where the orthogonal matrices are constructed by products of Householder reflections. Householder transformations are reflections about hyperplanes. Let $\mathbf{v} \in \mathbb{R}^D$, then the reflection about the hyperplane orthogonal to $\mathbf{v}$ is given by Eq. 5.58.

It is worth noting that performing a single Householder transformation is very cheap to compute, as it only requires $D$ parameters. Chaining together several Householder transformations results in more general orthogonal matrices, and Theorem 5.2

shows that any $M \times M$ orthogonal matrix can be written as the product of $M - 1$ Householder transformations. In our Householder Sylvester flow, the number of Householder transformations $H$ is a hyperparameter that trades off the number of parameters and the generality of the orthogonal transformation. Note that the use of Householder transformations forces us to use $M = D$, since Householder transformation results in square matrices.

**Triangular Sylvester flows.** Third, we consider a triangular Sylvester flow (T-SNF), in which all orthogonal matrices $\mathbf{Q}$ alternate per transformation between the identity matrix and the permutation matrix corresponding to reversing the order of $\mathbf{z}$. This is equivalent to alternating between lower and upper triangular $\tilde{\mathbf{R}}$ and $\mathbf{R}$ for each flow.

### Amortizing Flow Parameters

When using normalizing flows in an amortized inference setting, the parameters of the base distribution as well as the flow parameters can be functions of the data point $\mathbf{x}$ [19]. Figure 5.15 (left) shows a diagram of one SNF step and the amortization procedure. The inference network takes datapoints $\mathbf{x}$ as input and provides as an output the mean and variance of $\mathbf{z}^{(0)}$ such that $\mathbf{z}^{(0)} \sim \mathcal{N}(\mathbf{z}|\mu^0, \sigma^0)$. Several SNF transformations are then applied to $\mathbf{z}^{(0)} \rightarrow \mathbf{z}^{(1)} \rightarrow \ldots \mathbf{z}^{(T)}$, producing a flexible posterior distribution for $\mathbf{z}^{(T)}$. All of the flow parameters ($\mathbf{R}$, $\tilde{\mathbf{R}}$, and $\mathbf{Q}$ for each transformation) are produced as an output by the inference network and are thus fully amortized.

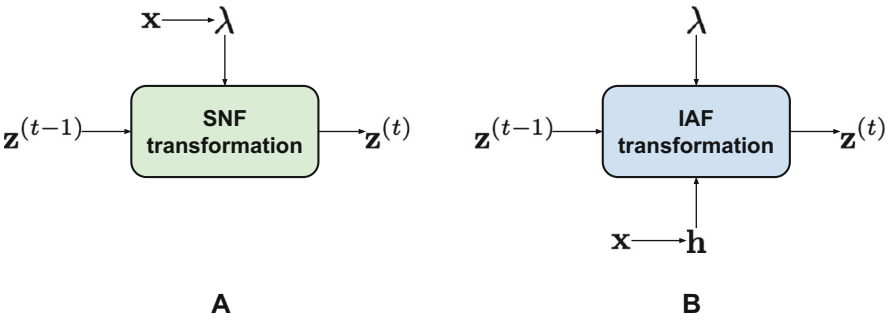



**Fig. 5.15** Different amortization strategies for Sylvester normalizing flows and inverse autoregressive flows (**a**) Our inference network produces amortized flow parameters. This strategy is also employed by planar flows. (**b**) Inverse autoregressive flow [18] introduces a measure of $\mathbf{x}$ dependence through a context variable $\mathbf{h}(\mathbf{x})$. This context acts as an additional input for each transformation. The flow parameters themselves are independent of $\mathbf{x}$.

#### 5.4.2.3 Hyperspherical Latent Space

Motivation

In the VAE framework, choosing Gaussian priors and Gaussian posteriors from the mathematical convenience leads to Euclidean latent space. However, such a choice could be limiting for the following reasons:

- In low dimensions, the standard Gaussian probability presents a concentrated probability mass around the mean, encouraging points to cluster in the center. However, this is particularly problematic when the data is divided into multiple clusters. Then, a better-suited prior would be *uniform*. Such a uniform prior, however, is not well defined on the hyperplane.
- It is a well-known phenomenon that the standard Gaussian distribution in high dimensions tends to resemble a uniform distribution on the surface of a hypersphere, with the vast majority of its mass concentrated on the hyperspherical shell (the so-called *soap bubble effect*). A natural question is whether it would be better to use a distribution defined on the hypersphere.

A distribution that would allow solving both problems at once is the von-Mises-Fisher distribution. It was advocated in [33] to use this distribution in the context of VAEs.

von-Mises-Fisher Distribution

The *von Mises-Fisher* (vMF) distribution is often described as the normal Gaussian distribution on a hypersphere. Analogous to a Gaussian, it is parameterized by $\mu \in \mathbb{R}^m$ indicating the mean direction, and $\kappa \in \mathbb{R}_{\geq 0}$ the concentration around $\mu$. For the special case of $\kappa = 0$, the vMF represents a uniform distribution. The probability density function of the vMF distribution for a random unit vector $\mathbf{z} \in \mathbb{R}^m$ (or $\mathbf{z} \in \mathcal{S}^{m-1}$) is then defined as

$$q(\mathbf{z}|\mu, \kappa) = C_m(\kappa) \exp(\kappa \mu^T \mathbf{z}) \tag{5.66}$$

$$C_m(\kappa) = \frac{\kappa^{m/2-1}}{(2\pi)^{m/2} \mathcal{I}_{m/2-1}(\kappa)}, \tag{5.67}$$

where $||\mu||^2 = 1$, $C_m(\kappa)$ is the normalizing constant and $\mathcal{I}_v$ denotes the modified Bessel function of the first kind at order $v$.

Interestingly, since we define a distribution over a hypersphere, it is possible to formulate a uniform prior over the hypersphere. Then it turns out that if we take the vMF distribution as the variational posterior, it is possible to calculate the Kullback-Leiber divergence between the vMF distribution and the uniform defined over $\mathcal{S}^{m-1}$ analytically [33]:

$$KL[\text{vMF}(\mu, \kappa)||\text{Unif}(\mathcal{S}^{m-1})] = \kappa + \log C_m(\kappa) - \log\left(\frac{2(\pi^{m/2})}{\Gamma(m/2)}\right)^{-1}. \tag{5.68}$$

To sample from the vMF, one can follow the procedure of [61]. Importantly, the reparameterization cannot be easily formulated for the vMF distribution. Fortunately, [62] allows extending the reparameterization trick to the wide class of distributions that can be simulated using rejection sampling. Davidson et al. [33] presents how to formulate the acceptance-rejection sampling reparameterization procedure. Being equipped with the sampling procedure and the reparameterization trick, and having an analytical form of the Kullback-Leibler divergence, we have everything to be able to build a hyperspherical VAE. However, please note that all these procedures are less trivial than the ones for Gaussians. Therefore, a curious reader is referred to [33] for further details.

## 5.5 Hierarchical Latent Variable Models

### 5.5.1 Introduction

The main goal of AI is to formulate and implement systems that can interact with an environment, process, store, and transmit information. In other words, we wish an AI system *understands* the world around it by identifying and disentangling hidden factors in the observed low sensory data [63]. If we think about the problem of building such a system, we can formulate it as learning a probabilistic model, i.e., a joint distribution over observed data, $\mathbf{x}$, and hidden factors, $\mathbf{z}$, namely, $p(\mathbf{x}, \mathbf{z})$. Then learning a *useful representation* is equivalent to finding a posterior distribution over the hidden factors, $p(\mathbf{z}|\mathbf{x})$. However, it is rather unclear what we really mean by *useful* in this context. In a beautiful blog post [64], Ferenc Huszar outlines why learning a latent variable model by maximizing the likelihood function is not necessarily useful from the representation learning perspective. Here, we will use it as a good starting point for a discussion of why applying hierarchical latent variable models could be beneficial.

Let us start by defining the setup. We assume the empirical distribution $p_{data}(\mathbf{x})$ and a latent variable model $p_\theta(\mathbf{x}, \mathbf{z})$. The way we parameterize the latent variable model is not constrained in any manner; however, we assume that the distribution is parameterized using deep neural networks (DNNs). This is important for two reasons:

1. DNNs are nonlinear transformations, and as such, they are flexible and allow parameterizing a wide range of distributions.
2. We must remember that DNNs **will not** solve all problems for us! In the end, we need to think about the model as a whole, not only about the parameterization. What I mean by that is the distribution we choose and how random variables interact, etc. DNNs are definitely helpful, but there are many potential pitfalls (we will discuss some of them later on) that even the largest and coolest DNN is unable to take care of.

It is worth remembering that the joint distribution could be factorized in two ways, namely:

$$p_\theta(\mathbf{x}, \mathbf{z}) = p_\theta(\mathbf{x}|\mathbf{z})p_\theta(\mathbf{z}) \tag{5.69}$$

$$= p_\theta(\mathbf{z}|\mathbf{x})p_\theta(\mathbf{x}). \tag{5.70}$$

Moreover, the training problem of learning $\theta$ could be defined as an unconstrained optimization problem with the following training objective:

$$KL[p_{data}(\mathbf{x})||p_\theta(\mathbf{x})] = -\mathbb{H}[p_{data}(\mathbf{x})] + \mathbb{CE}[p_{data}(\mathbf{x})||p_\theta(\mathbf{x})] \tag{5.71}$$

$$= const + \mathbb{CE}[p_{data}(\mathbf{x})||p_\theta(\mathbf{x})], \tag{5.72}$$

where $p_\theta(\mathbf{x}) = \int p_\theta(\mathbf{x}, \mathbf{z})\ \mathrm{d}\mathbf{z}$, $\mathbb{H}[\cdot]$ denotes the entropy and $\mathbb{CE}[\cdot||\cdot]$ is the cross entropy. Notice that the entropy of the empirical distribution is simply a constant, since it does not contain $\theta$. The cross-entropy could be further rewritten as follows:

$$\mathbb{CE}[p_{data}(\mathbf{x})||p_\theta(\mathbf{x})] = -\int p_{data}(\mathbf{x}) \ln p_\theta(\mathbf{x})\ \mathrm{d}\mathbf{x} \tag{5.73}$$

$$= -\frac{1}{N}\sum_{n=1}^{N} \ln p_\theta(\mathbf{x}_n). \tag{5.74}$$

Eventually, we obtained the objective function we use all the time, namely, the negative log-likelihood function.

If we think of *usefulness* of a representation (i.e., hidden factors) $\mathbf{z}$, we intuitively think of some kind of information that is shared between $\mathbf{z}$ and $\mathbf{x}$. However, the unconstrained training problem we consider, i.e., the minimization of the negative log-likelihood function, does not necessarily say **anything** about the latent representation. In the end, we optimize the **marginal** over observable variables, because we do not have access to values of latent variables. Even more, typically we do not know what these hidden factors are or should be! As a result, our latent variable model can learn to. . . disregard the latent variables completely. Let us look into this problem in more detail.

### A Potential Problem with Latent Variable Models

Following the discussion presented in [64], we can visualize two scenarios that are pretty common in deep generative modeling with latent variables models. Before delving into that, it is beneficial to explain the general picture. We are interested in analyzing a class of latent variable models with respect to *usefulness* of latents and the value of the objective function $KL[p_{data}(\mathbf{x})||p_\theta(\mathbf{x})]$. In Fig. 5.16, we depict a case when all models are possible, namely, a search space where models are evaluated according to the training objective (x-axis) and *usefulness* (y-axis). The ideal model is the one in the top left corner that maximizes both criteria. However, it is possible to find a model that completely disregards the latents (the bottom left

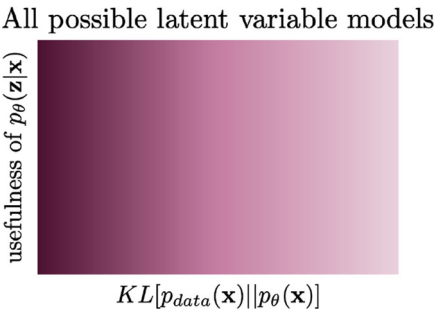



**Fig. 5.16** A schematic diagram representing a dependency between *usefulness* and the objective function for all possible latent variable models. The darker the color, the better the objective function value. (Reproduced based on [64]).

corner) while maximizing the fit to data. We already can see that there is a potentially huge problem! Running a (numerical) optimization procedure could give infinitely many models that are equally good with respect to $KL[p_{data}(\mathbf{x})||p_\theta(\mathbf{x})]$ but with completely different posteriors over latents! That puts in question the applicability of the latent variable models. However, in practice, we see that learned latent variables are useful (or, in other words, they contain information about observables). So how is it possible?

As pointed out by [64], the reason for that is the inductive bias of the chosen class of models. By picking a very specific class of DNNs, we implicitly constrain the search space. First, the left-most models in Fig. 5.16 are typically unattainable. However, using some kind of bottlenecks in our class of models potentially leads to a situation that latents must contain some information about observables. As a result, they become *useful*. An example of such a situation is depicted in Fig. 5.17. After running a training algorithm, we can end in one of the two "spikes" where the training objective is the highest and the *usefulness* is nonzero. Still, we can achieve the same performing models at two different levels of the *usefulness*, but at least the information flows from $\mathbf{x}$ to $\mathbf{z}$. Obviously, the considered scenario is purely hypothetical, but it shows that the inductive bias of a model can greatly help to learn representations without being specified by the objective function. Please keep this thought in mind, because it will play a crucial role later on!

The next situation is more tricky. Let us assume that we have a constrained class of models; however, the conditional likelihood $p(\mathbf{x}|\mathbf{z})$ is parameterized by a flexible, enormous DNN. A potential danger here is that this model could learn to completely disregard $\mathbf{z}$, treating it as a noise. As a result, $p(\mathbf{x}|\mathbf{z})$ becomes an unconditional distribution that mimics $p_{data}(\mathbf{x})$ almost perfectly. At first glance, this scenario sounds unrealistic, but it is a well-known phenomenon in the field. For instance, [10] conducted a thorough experiment with variational auto-encoders, and taking a PixelCNN++-based decoder resulted in a VAE that was unable to reconstruct images. Their conclusion was exactly the same, namely, taking a class of models with too flexible $p(\mathbf{x}|\mathbf{z})$ could lead to the model in the bottom left corner in Fig. 5.18.

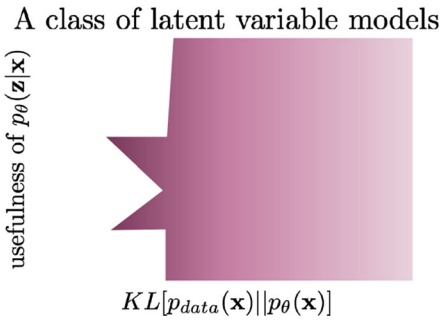



**Fig. 5.17** A schematic diagram representing a dependency between *usefulness* and the objective function for a constrained class of models. The darker the color, the better the objective function value. (Reproduced based on [64]).

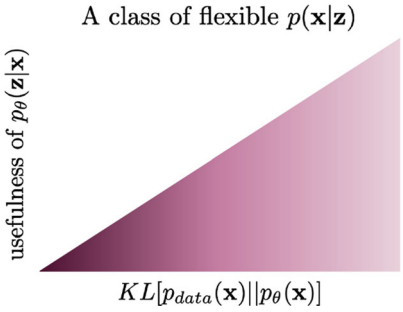



**Fig. 5.18** A schematic diagram representing a dependency between *usefulness* and the objective function for a class of models with flexible $p(\mathbf{x}|\mathbf{z})$. The darker the color, the better the objective function value. (Reproduced based on [64]).

### How to Define a *Proper* Class of Models?

Alright, you are probably a bit confused about what we have discussed so far. The general picture is rather pessimistic, because it seems that picking a proper class of models, i.e., a class of models that allow achieving *useful* latent representations, is a nontrivial task. Moreover, the whole story sounds like walking in the dark, trying out various DNNs architectures, and hoping that we obtain a meaningful representation.

Fortunately, the problem is not as horrible as it looks at first glance. Some ideas formulate a constrained optimization problem [12, 65] or add an auxiliary regularizer [66, 67] to (implicitly) define *usefulness* of the latents. Here, we will discuss one of the possible approaches that utilize hierarchical architectures. However, it is worth remembering that the issue of learning *useful* representations remains an open question and is a vivid research direction.

Hierarchical models have a long history in deep generative modeling and deep learning and were advocated by many prominent researchers, e.g., [68–70]. The main hypothesis is that the concepts describing the world around us could be organized hierarchically. In light of our discussion, if a latent variable model takes a hierar-

chical structure, it may introduce an inductive bias, constrain the class of models, and, eventually, force information flow between latents and observables. At least in theory. Shortly, we will see that we must be very careful with formulating stochastic dependencies in the hierarchy. In the next sections, we will focus on latent variable models with variational inference, i.e., hierarchical variational auto-encoders.

**A side note**

One may be tempted to associate hierarchical modeling with Bayesian hierarchical modeling. These two terms are not necessarily equivalent. Bayesian hierarchical modeling is about treating *(hyper)parameters* as random variables and formulating distributions over (hyper)parameters [71]. Here, we do not take advantage of Bayesian modeling and consider a hierarchy among latent variables, not parameters.

## 5.5.2 Hierarchical VAEs

### 5.5.2.1 Two-Level VAEs

Let us start with a VAE with two latent variables: $\mathbf{z}_1$ and $\mathbf{z}_2$. The joint distribution could be factorized as follows:

$$p(\mathbf{x}, \mathbf{z}_1, \mathbf{z}_2) = p(\mathbf{x}|\mathbf{z}_1)p(\mathbf{z}_1|\mathbf{z}_2)p(\mathbf{z}_2). \tag{5.75}$$

This model defines a straightforward generative process: First sample $\mathbf{z}_2$, then sample $\mathbf{z}_1$ given $\mathbf{z}_2$, and eventually sample $\mathbf{x}$ given $\mathbf{z}_1$.

Since we know already that even for a single latent variable calculating posteriors over latents is intractable (except the linear Gaussian case, it is worth remembering that!), we can utilize the variational inference with a family of variational posteriors $Q(\mathbf{z}_1, \mathbf{z}_2|\mathbf{x})$. Now, the main part is how to define the variational posteriors. A rather natural approach would be to reverse the dependencies and factorize the posterior in the following fashion:

$$Q(\mathbf{z}_1, \mathbf{z}_2|\mathbf{x}) = q(\mathbf{z}_1|\mathbf{x})q(\mathbf{z}_2|\mathbf{z}_1, \mathbf{x}), \tag{5.76}$$

or even we can simplify it as follows (dropping the dependency on $\mathbf{x}$ for the second latent variable):

$$Q(\mathbf{z}_1, \mathbf{z}_2|\mathbf{x}) = q(\mathbf{z}_1|\mathbf{x})q(\mathbf{z}_2|\mathbf{z}_1). \tag{5.77}$$

If we take the continuous latents, we can use the Gaussian distributions:

$$p(\mathbf{z}_1|\mathbf{z}_2) = \mathcal{N}(\mathbf{z}_1|\mu(\mathbf{z}_2), \sigma^2(\mathbf{z}_2)) \tag{5.78}$$

$$p(\mathbf{z}_2) = \mathcal{N}(\mathbf{z}_2|0, 1) \tag{5.79}$$

$$q(\mathbf{z}_1|\mathbf{x}) = \mathcal{N}(\mathbf{z}_1|\mu(\mathbf{x}), \sigma^2(\mathbf{x})) \tag{5.80}$$

$$q(\mathbf{z}_2|\mathbf{z}_1) = \mathcal{N}(\mathbf{z}_2|\mu(\mathbf{z}_1), \sigma^2(\mathbf{z}_1)), \tag{5.81}$$

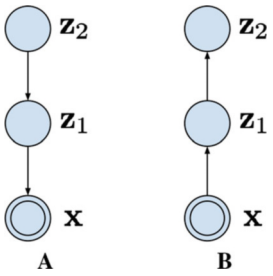



**Fig. 5.19** An example of a two-level VAE. (**a**) The generative part. (**b**) The variational part.

where $\mu_i(\mathbf{v})$ means that a mean parameter is parameterized by a neural network that takes a random variable $\mathbf{v}$ as input, analogously we parameterize variances (i.e., diagonal covariance matrices). As we can see, this is a straightforward extension of a VAE we discussed before.

The two-level VAE is depicted in Fig. 5.19. Notice how the stochastic dependencies are defined; namely, there is always a dependency on a single random variable.

### A Potential Pitfall

Alright, so are we done? Do we have a better class of VAEs? Unfortunately, the answer is **no**. We noticed that this two-level version of a VAE is a rather straightforward extension of a one-level VAE. Thus, our discussion about potential problems with latent variable models holds true. We get even get extra insight if we look into the ELBO for the two-level VAE (if you do not remember how to derive the ELBO, please go back to the previous sections on VAEs first):

$$ELBO(\mathbf{x}) = \mathbb{E}_{Q(\mathbf{z}_1,\mathbf{z}_2|\mathbf{x})}\Big[\ln p(\mathbf{x}|\mathbf{z}_1) - \ln \frac{q(\mathbf{z}_1|\mathbf{x})}{p(\mathbf{z}_1|\mathbf{z}_2)} - KL[q(\mathbf{z}_2|\mathbf{z}_1)||p(\mathbf{z}_2)]\Big]. \quad (5.82)$$

To shed some light on the ELBO for the two-level VAE, we notice the following:

1. All conditions $(\mathbf{z}_1, \mathbf{z}_2, \mathbf{x})$ are either samples from $Q(\mathbf{z}_1, \mathbf{z}_2|\mathbf{x})$ or $p_{data}(\mathbf{x})$.
2. We obtain the Kullback-Leibler divergence term for the last layer, i.e., $\mathbf{z}_2$. For the term the middle, we cannot obtain the Kullback-Leibler, because we need to sample $\mathbf{z}_1$ from $q(\mathbf{z}_1|\mathbf{x})$ first, and then we can get $\mathbf{z}_2$ from $q(\mathbf{z}_2|\mathbf{z}_1)$. As a result, we cannot "swap" distributions to obtain the Kullback-Leibler term. You are encouraged to derive the ELBO step by step; it is a great exercise to get familiar with the variational inference.
3. It is worth remembering that the Kullback-Leibler divergence is always nonnegative.

Theoretically, everything should work perfectly fine, but there are a couple of potential problems. First, we initialize all DNNs that parameterize the distributions randomly. As a result, all Gaussians are basically standard Gaussians. Second, if

the decoder is powerful and flexible, there is a huge danger that the model will try to take advantage of the optimum for the last KL-term, $KL[q(\mathbf{z}_2|\mathbf{z}_1)||p(\mathbf{z}_2)]\Big|$, that is $q(\mathbf{z}_2|\mathbf{z}_1) \approx p(\mathbf{z}_2) \approx \mathcal{N}(0, 1)$. Then, since $q(\mathbf{z}_2|\mathbf{z}_1) \approx \mathcal{N}(0, 1)$, the second layer is not used at all (it is a Gaussian noise) and we get back to the same issues as in the one-level VAE architecture. It turns out that learning the two-level VAE is even more problematic than a VAE with a single latents, because even for a relatively simple decoder, the second latent variables $\mathbf{z}_2$ are mostly unused [15, 72]. This effect is called the *posterior collapse*.

#### 5.5.2.2 Top-Down VAEs

A takeaway from our considerations in the two-level VAE is that adding an extra level does not necessarily provide anything compared to the one-level VAE. However, so far, we have considered only one class of variational posteriors, namely:

$$Q(\mathbf{z}_1, \mathbf{z}_2|\mathbf{x}) = q(\mathbf{z}_1|\mathbf{x})q(\mathbf{z}_2|\mathbf{z}_1). \tag{5.83}$$

A natural question is whether we can do better. You can already guess the answer, but before shouting it out loud, let us think for a second. In the generative part, we have *top-down* dependencies, going from the highest level of abstraction (latents) down to the observable variables. Let us repeat it here again:

$$p(\mathbf{x}, \mathbf{z}_1, \mathbf{z}_2) = p(\mathbf{x}|\mathbf{z}_1)p(\mathbf{z}_1|\mathbf{z}_2)p(\mathbf{z}_2). \tag{5.84}$$

Perhaps, we can mirror such dependencies in the variational posteriors as well. Then we get the following:

$$Q(\mathbf{z}_1, \mathbf{z}_2|\mathbf{x}) = q(\mathbf{z}_1|\mathbf{z}_2, \mathbf{x})q(\mathbf{z}_2|\mathbf{x}). \tag{5.85}$$

Do you see any resemblance? Yes, the variational posteriors have the extra $\mathbf{x}$, but the dependencies are pointing in the same direction. Why this could be beneficial? Because now we could have a shared *top-down* path that would make the variational posteriors and the generative part tightly connected through a shared parameterization. That could be a very useful inductive bias!

This idea was originally proposed in ResNet VAEs [18] and Ladder VAEs [73], and it was further developed in BIVA [45], NVAE [46], and the very deep VAE [47]. These approaches differ in their implementations and parameterizations used (i.e., architectures of DNNs); however, they all could be categorized as instantiations of top-down VAEs. The main idea, as mentioned before, is to share the top-down path between the variational posteriors and the generative distributions and use a *side*, a deterministic path going from $\mathbf{x}$ to the last latents. Alright, let us write this idea down.

First, we have the top-down path that defines $p(\mathbf{x}|\mathbf{z}_1)$, $p(\mathbf{z}_1|\mathbf{z}_2)$, and $p(\mathbf{z}_2)$. Thus, we need a DNN that outputs $\mu_1$ and $\sigma_1^2$ for given $\mathbf{z}_2$, and another DNN that outputs

the parameters of $p(\mathbf{x}|\mathbf{z}_1)$ for given $\mathbf{z}_1$. Since $p(\mathbf{z}_2)$ is an unconditional distribution (e.g., the standard Gaussian), we do not need a separate DNN here.

Second, we have a side, deterministic path that gives two deterministic variables: $\mathbf{r}_1 = f_1(\mathbf{x})$ and $\mathbf{r}_2 = f_2(\mathbf{r}_1)$. Both transformations, $f_1$ and $f_2$, are DNNs. Then, we can use additional DNNs that return some modifications of the means and the variances, namely, $\Delta\mu_1, \Delta\sigma_1^2$, and $\Delta\mu_2, \Delta\sigma_2^2$. These modifications could be defined in many ways. Here we follow the way it is done in NVAE [46], namely, the modifications are relative location and scales of the values given in the top-down path. If you do not fully follow this idea, it should be clear once we define the variational posteriors.

Finally, we can define the whole procedure. We define various neural networks by specifying different indices. For sampling, we use the top-down path:

**Top-down path**

1. $\mathbf{z}_2 \sim \mathcal{N}(0, 1)$
2. $[\mu_1, \sigma_1^2] = NN_1(\mathbf{z}_2)$
3. $\mathbf{z}_1 \sim \mathcal{N}(\mu_1, \sigma_1^2)$
4. $\vartheta = NN_x(\mathbf{z}_1)$
5. $\mathbf{x} \sim p_\vartheta(\mathbf{x}|\mathbf{z}_1)$

Now (please focus!) we calculate samples from the variational posteriors as follows:

**Bottom-up path**

1. (*Bottom-up deterministic path*) $\mathbf{r}_1 = f_1(\mathbf{x})$ and $\mathbf{r}_2 = f_2(\mathbf{r}_1)$
2. $[\Delta\mu_1, \Delta\sigma_1^2] = NN_{\Delta 1}(r_1)$
3. $[\Delta\mu_2, \Delta\sigma_2^2] = NN_{\Delta 2}(r_2)$
4. $\mathbf{z}_2 \sim \mathcal{N}(0 + \Delta\mu_2, 1 \cdot \Delta\sigma_2^2)$
5. $[\mu_1, \sigma_1^2] = NN_1(\mathbf{z}_2)$
6. $\mathbf{z}_1 \sim \mathcal{N}(\mu_1 + \Delta\mu_1, \sigma_1^2 \cdot \Delta\sigma_1^2)$
7. $\vartheta = NN_x(\mathbf{z}_1)$
8. $\mathbf{x} \sim p_\vartheta(\mathbf{x}|\mathbf{z}_1)$

These operations are schematically presented in Fig. 5.20.

Please note that the deterministic bottom-up path modifies the parameters of the top-down path. As advocated by [46], this idea is especially useful because "when the prior moves, the approximate posterior moves accordingly, if not changed." Moreover, as noted in [46], the Kullback-Leibler between two Gaussians simplifies as follows (we remove some additional dependencies for clarity):

$$KL\left(q\left(z_i \mid \boldsymbol{x}\right) \| p\left(z_i\right)\right) = \frac{1}{2}\left(\frac{\Delta\mu_i^2}{\sigma_i^2} + \Delta\sigma_i^2 - \log\Delta\sigma_i^2 - 1\right)$$

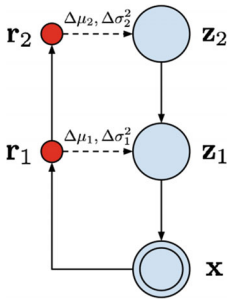



**Fig. 5.20** An example of the top-down VAE. Red nodes denote the deterministic path, and blue nodes depict random variables.

Eventually, we implicitly force a close connection between the variational posteriors and the generative part. This inductive bias helps to encode information about the observables in the latents. Moreover, there is no need to use overly flexible decoders, since the latents take care of distilling the essence from data. I know, it is still a bit hand-wavy, since we do not define the magical *usefulness*, but I hope you get the picture. The top-down VAEs entangle the variational posteriors and the generative path, and, as a result, the Kullback-Leibler terms will not collapse (i.e., they will be greater than zero). Empirical studies strongly back up this hypothesis [45–47, 73].

#### 5.5.2.3 Code

Let us delve into an implementation of a top-down VAE. We stick to the two-level VAE to match the description provided above. We will use precisely the same steps as in the procedures used above. For clarity, we will use a single class to the code as similar to the mathematical expressions above as possible. We use the reparameterization trick for sampling. There is one difference between the math and the code, namely, in the code we use $\log \Delta\sigma$ instead of $\Delta sigma$. Then, we use $\log\sigma + \log\Delta\sigma$ instead of $\sigma \cdot \Delta\sigma$ because $e^{\log a + \log b} = e^{\log a} \cdot e^{\log b} = a \cdot b$.

```
class HierarchicalVAE(nn.Module):
    def __init__(self, nn_r_1, nn_r_2, nn_delta_1, nn_delta_2,
    nn_z_1, nn_x, num_vals=256, D=64, L=16, likelihood_type='
    categorical'):
        super(HierarchicalVAE, self).__init__()

        print('Hierachical VAE by JT.')

        # bottom-up path
        self.nn_r_1 = nn_r_1
        self.nn_r_2 = nn_r_2

        self.nn_delta_1 = nn_delta_1
        self.nn_delta_2 = nn_delta_2

```

```
        # top-down path
        self.nn_z_1 = nn_z_1
        self.nn_x = nn_x


        # other params
        self.D = D # dim of inputs

        self.L = L # dim of the second latent layer

        self.num_vals = num_vals # num of values per pixel

        self.likelihood_type = likelihood_type # the conditional likelihood type (categorical/bernoulli)

    # If you don't remember the reparameterization trick, please go back to the section on VAEs.
    def reparameterization(self, mu, log_var):
        std = torch.exp(0.5*log_var)
        eps = torch.randn_like(std)
        return mu + std * eps

    def forward(self, x, reduction='avg'):
        #=====
        # First, we need to calculate the bottom-up deterministic path.
        # Here we use a small trick to keep the delta of variance constrained, namely, we apply the hard-tanh nonlinearity.

        # bottom-up
        # step 1
        r_1 = self.nn_r_1(x)
        r_2 = self.nn_r_2(r_1)

        #step 2
        delta_1 = self.nn_delta_1(r_1)
        delta_mu_1, delta_log_var_1 = torch.chunk(delta_1, 2, dim=1)
        delta_log_var_1 = F.hardtanh(delta_log_var_1, -7., 2.)

        # step 3
        delta_2 = self.nn_delta_2(r_2)
        delta_mu_2, delta_log_var_2 = torch.chunk(delta_2, 2, dim=1)
        delta_log_var_2 = F.hardtanh(delta_log_var_2, -7., 2.)


        # Next, we can do the top-down path.

        # top-down
        # step 4
        z_2 = self.reparameterization(delta_mu_2, delta_log_var_2)

```

```
        # step 5
        h_1 = self.nn_z_1(z_2)
        mu_1, log_var_1 = torch.chunk(h_1, 2, dim=1)

        # step 6
        z_1 = self.reparameterization(mu_1 + delta_mu_1,
    log_var_1 + delta_log_var_1)

        # step 7
        h_d = self.nn_x(z_1)

        if self.likelihood_type == 'categorical':
            b = h_d.shape[0]
            d = h_d.shape[1]//self.num_vals
            h_d = h_d.view(b, d, self.num_vals)
            mu_d = torch.softmax(h_d, 2)

        elif self.likelihood_type == 'bernoulli':
            mu_d = torch.sigmoid(h_d)

        #=====ELBO
        # RE
        if self.likelihood_type == 'categorical':
            RE = log_categorical(x, mu_d, num_classes=self.
    num_vals, reduction='sum', dim=-1).sum(-1)

        elif self.likelihood_type == 'bernoulli':
            RE = log_bernoulli(x, mu_d, reduction='sum', dim=-1)

        # KL
        # For the Kullback-Leibler part, we need to calculate two
     divergences:
        # 1) KL[q(z_2|z) || p(z_2)] where p(z_2) = N(0,1)
        # 2) KL[q(z_1|z_2, x) || p(z_1|z_2)]
        # Note: We use the analytical of the KL between two
    Gaussians here. If you use a different distribution,
        # please pay attention! You would need to use a different
     expression here.
        KL_z_2 = 0.5 * (delta_mu_2**2 + torch.exp(delta_log_var_2
    ) - delta_log_var_2 - 1).sum(-1)
        KL_z_1 = 0.5 * (delta_mu_1**2 / torch.exp(log_var_1) +
    torch.exp(delta_log_var_1) -\
                        delta_log_var_1 - 1).sum(-1)

        KL = KL_z_1 + KL_z_2

        # Final ELBO
        if reduction == 'sum':
            loss = -(RE - KL).sum()
        else:
            loss = -(RE - KL).mean()

        return loss

```

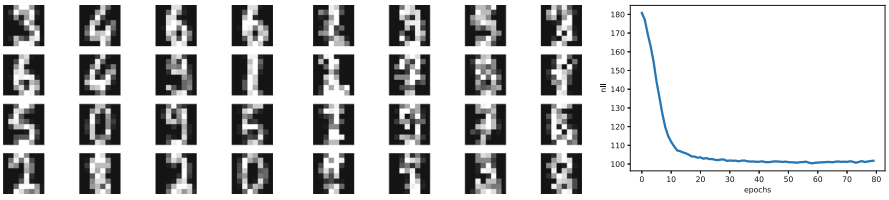
```
    # Sampling is the top-down path but without calculating delta
     mean and delta variance.
    def sample(self, batch_size=64):
        # step 1
        z_2 = torch.randn(batch_size, self.L)
        # step 2
        h_1 = self.nn_z_1(z_2)
        mu_1, log_var_1 = torch.chunk(h_1, 2, dim=1)
        # step 3
        z_1 = self.reparameterization(mu_1, log_var_1)

        # step 4
        h_d = self.nn_x(z_1)

        if self.likelihood_type == 'categorical':
            b = batch_size
            d = h_d.shape[1]//self.num_vals
            h_d = h_d.view(b, d, self.num_vals)
            mu_d = torch.softmax(h_d, 2)
            # step 5
            p = mu_d.view(-1, self.num_vals)
            x_new = torch.multinomial(p, num_samples=1).view(b,d)

        elif self.likelihood_type == 'bernoulli':
            mu_d = torch.sigmoid(h_d)
            # step 5
            x_new = torch.bernoulli(mu_d)
        return x_new
```

**Listing 5.12** A top-down VAE class.

That's it! Now we are ready to run the full code. After training our top-down VAE, we should obtain results like in Fig. 5.21.

#### 5.5.2.4 Further Reading

What we have discussed here is just touching upon the topic. Hierarchical models in probabilistic modeling seem to be an important research direction and modeling




A B C

**Fig. 5.21** An example of outcomes after the training of a top-down VAE: (**a**) Randomly selected real images. (**b**) Unconditional generations from the top-down VAE. (**c**) The validation curve during training.

paradigm. Moreover, the technical details are also crucial for achieving state-of-the-art performance. I strongly suggest reading about NVAE [46], ResNet VAE [18], Ladder VAE [73], BIVA [45], and very deep VAEs [47] and compare various tricks and parameterizations used therein. These models share the same idea, but implementations vary significantly.

The research on hierarchical generative modeling is very up to date and develops very quickly. As a result, this is nearly impossible to mention even a fraction of interesting papers. I will mention only a few worth noticing papers:

- Pervez and Gavves [74] provides an insightful analysis of a potential problem with hierarchical VAEs; namely, the KL divergence term is closely related to the harmonics of the parameterizing function. In other words, using DNNs results in high-frequency components of the KL term, and, eventually, it leads to the posterior collapse. The authors propose to smooth the VAE by applying Ornstein-Uhlenbeck (OU) Semigroup. I refer to the original paper for details.
- Wu et al. [75] proposes greedy layer-wise learning of a hierarchical VAE. The authors used this idea in the context of video prediction, so their approach could be also motivated by computational constrained. However, the idea of greedy layer-wise training has been extensively utilized in the past [68–70].
- Gatopoulos and Tomczak [25] discusses incorporating predefined transformations like downscaling into the model. The idea is to learn a reversed transformation to, e.g., downscaling in a stochastic manner. The resulting VAE has a set of auxiliary variables (e.g., downscaled versions of observables) a set of latent variables that encode missing information in the auxiliary variables. The hypothesis in such an approach is that learning a distribution over smaller or already processed observable variables is easier, and, thus, we can decompose the problem into multiple problems of learning simpler distributions. A diagram for this approach is presented in Fig. 5.22.

The beauty of the latent-variable modeling paradigm is that we can play with stochastic relationships among objects and, eventually, formulate a *useful* representation of data. As we will see in the next chapters, there are other interesting classes of models that take advantage of diffusion models and energy functions.

### 5.5.3 Diffusion-Based Deep Generative Models

#### 5.5.3.1 Introduction

In Sect. 5.5, we discussed the issue of learning *useful* representations in latent variables models, taking a closer look at hierarchical variational auto-encoders. We hypothesize that we can obtain *useful* data representations by applying a hierarchical latent variable model. Moreover, highlighted a real problem in hierarchical VAEs of the variational posterior collapsing to the prior, resulting in learning meaningless representation. In other words, it seems that architecture with bottom-up variational posteriors (i.e., stochastic dependencies going from observables to the last latents)

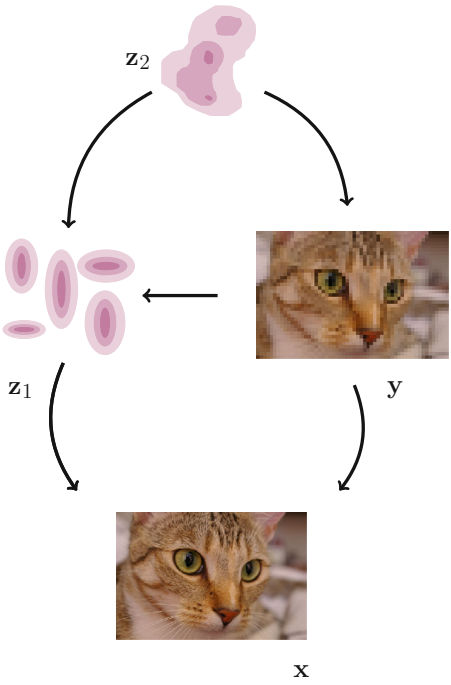



**Fig. 5.22** A two-level VAE with an auxiliary set of deterministic variables $\mathbf{y}$ (e.g., downscaled images).

and top-down generative distributions seems to be a mediocre inductive bias and is rather troublesome to train. A potential solution is top-down VAEs. However, is there nothing we can do about the *vanilla* structure? As you may imagine, nothing is lost, and some approaches take advantage of the bottom-up and the top-down structures. Here, we will look into the *diffusion-based deep generative models* (DDGM) (a.k.a. *deep diffusion probabilistic models*) [76, 77].

DDGM could be briefly explained as hierarchical VAEs with the bottom-up path (i.e., the variational posteriors) defined by a diffusion process (e.g., a Gaussian diffusion) and the top-down path parameterized by DNNs (a reversed diffusion). Interestingly, the bottom-up path could be **fixed**; namely, it necessarily does not have any learnable parameters. An example of applying a Gaussian diffusion is presented in Fig. 5.23. Since the variational posteriors are fixed, we can think of them as adding Gaussian noise at each layer. Then, the final layer resembles Gaussian noise (see $\mathbf{z}_5$ in Fig. 5.23). If we recall the discussion about a potential issue of posterior collapse in hierarchical VAEs, this should not be a problem anymore. Why? Because we should get a standard Gaussian distribution in the last layer **by design**. Pretty neat, isn't it?

DDGMs have become extremely popular these days. They are appealing for at least two reasons:

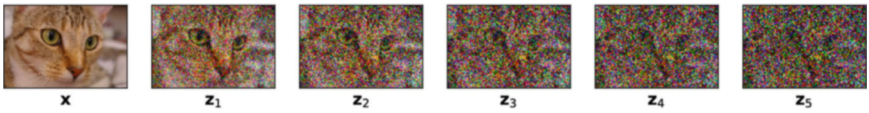



**Fig. 5.23** An example of applying a Gaussian diffusion to an image of a cat, $\mathbf{x}$.

1. They give amazing results for image synthesis [76, 78, 79], audio synthesis [80], and promising results for text synthesis [81, 82] while being relatively simple to implement.
2. They are closely related to stochastic differential equations, and, thus, their theoretical properties seem to be especially of great interest [83–85].

There are two potential drawbacks though, namely:

1. DDGMs are unable (for now at least) to learn a representation.
2. Similarly to flow-based models, the dimensionality of input is kept across the whole model (i.e., there is no bottleneck on the way).

#### 5.5.3.2 Model Formulation

Originally, deep diffusion probabilistic models were proposed in [77], and they took inspiration from nonequilibrium statistical physics. The main idea is to iteratively destroy the structure in data through a forward diffusion process and, afterward, to learn a reverse diffusion process to restore the structure in data. In a follow-up paper [76], recent developments in deep learning were used to train a powerful and flexible diffusion-based deep generative model that achieved SOTA results in the task of image synthesis. Here, we will abuse the original notation to make a clear connection between hierarchical latent variable models and DDGMs. As previously, we are interested in finding a distribution over data, $p_\theta(\mathbf{x})$; however, we assume an additional set of latent variables $\mathbf{z}_{1:T} = [\mathbf{z}_1, \ldots, \mathbf{z}_T]$. The marginal likelihood is defined by integrating out all latents:

$$p_\theta(\mathbf{x}) = \int p_\theta(\mathbf{x}, \mathbf{z}_{1:T}) \, \mathrm{d}\mathbf{z}_{1:T}. \tag{5.86}$$

The joint distribution is modeled as a first-order Markov chain with Gaussian transitions, namely:

$$p_\theta(\mathbf{x}, \mathbf{z}_{1:T}) = p_\theta(\mathbf{x}|\mathbf{z}_1) \left( \prod_{i=1}^{T-1} p_\theta(\mathbf{z}_i|\mathbf{z}_{i+1}) \right) p_\theta(\mathbf{z}_T), \tag{5.87}$$

where $\mathbf{x} \in \mathbb{R}^D$ and $\mathbf{z}_i \in \mathbb{R}^D$ for $i = 1, \ldots, T$. Please note that the latents have the same dimensionality as the observables. This is the same situation as in the case of flow-based models. We parameterize all distributions using DNNs.

So far, we have not introduced anything new! This is again a hierarchical latent variable model. As in the case of hierarchical VAEs, we can introduce a family of variational posteriors as follows:

$$Q_\phi(\mathbf{z}_{1:T}|\mathbf{x}) = q_\phi(\mathbf{z}_1|\mathbf{x})\left(\prod_{i=2}^{T} q_\phi(\mathbf{z}_i|\mathbf{z}_{i-1})\right). \tag{5.88}$$

The key point is how we define these distributions. Before, we used normal distributions parameterized by DNNs, but now we formulate them as the following Gaussian diffusion process [77]:

$$q_\phi(\mathbf{z}_i|\mathbf{z}_{i-1}) = \mathcal{N}(\mathbf{z}_i|\sqrt{1-\beta_i}\mathbf{z}_{i-1}, \beta_i\mathbf{I}), \tag{5.89}$$

where $\mathbf{z}_0 = \mathbf{x}$. Notice that a single step of the difussion, $q_\phi(\mathbf{z}_i|\mathbf{z}_{i-1})$, works in a relatively easy way. Namely, it takes the previously generated object $\mathbf{z}_{i-1}$, scales it by $\sqrt{1-\beta_i}$, and then adds noise with variance $\beta_i$. To be even more explicit, we can write it using the reparameterization trick:

$$\mathbf{z}_i = \sqrt{1-\beta_i}\mathbf{z}_{i-1} + \sqrt{\beta_i} \odot \epsilon, \tag{5.90}$$

where $\epsilon \sim \mathcal{N}(0, \mathbf{I})$. In principle, $\beta_i$ could be learned by backpropagation; however, as noted by [76, 77], it could be fixed. For instance, [76] suggests to change it linearly from $\beta_1 = 10^{-4}$ to $\beta_T = 0.02$.

Since we realized that the difference between a DDGM and a hierarchical VAE lies in the definition of the variational posteriors and the dimensionality of the latents, but the whole construction is basically the same, we can predict what is the learning objective. Do you remember? Yes, it is ELBO! We can derive the ELBO as follows:

$$\begin{aligned}
\ln p_\theta(\mathbf{x}) = &\ln \int Q_\phi(\mathbf{z}_{1:T}|\mathbf{x}) \frac{p_\theta(\mathbf{x}, \mathbf{z}_{1:T})}{Q_\phi(\mathbf{z}_{1:T}|\mathbf{x})} \, \mathrm{d}\mathbf{z}_{1:T} \\
\geq &\ \mathbb{E}_{Q_\phi(\mathbf{z}_{1:T}|\mathbf{x})}\left[\ln p_\theta(\mathbf{x}|\mathbf{z}_1) + \sum_{i=1}^{T-1} \ln p_\theta(\mathbf{z}_i|\mathbf{z}_{i+1}) + \ln p_\theta(\mathbf{z}_T) + \right. \\
&\left. - \sum_{i=2}^{T} \ln q_\phi(\mathbf{z}_i|\mathbf{z}_{i-1}) - \ln q_\phi(\mathbf{z}_1|\mathbf{x})\right] \\
= &\ \mathbb{E}_{Q_\phi(\mathbf{z}_{1:T}|\mathbf{x})}\left[\ln p_\theta(\mathbf{x}|\mathbf{z}_1) + \ln p_\theta(\mathbf{z}_1|\mathbf{z}_2) + \sum_{i=2}^{T-1} \ln p_\theta(\mathbf{z}_i|\mathbf{z}_{i+1}) + \ln p_\theta(\mathbf{z}_T) + \right. \\
&\left. - \sum_{i=2}^{T-1} \ln q_\phi(\mathbf{z}_i|\mathbf{z}_{i-1}) - \ln q_\phi(\mathbf{z}_T|\mathbf{z}_{T-1}) - \ln q_\phi(\mathbf{z}_1|\mathbf{x})\right] \\
= &\ \mathbb{E}_{Q_\phi(\mathbf{z}_{1:T}|\mathbf{x})}\left[\ln p_\theta(\mathbf{x}|\mathbf{z}_1) + \sum_{i=2}^{T-1} \left(\ln p_\theta(\mathbf{z}_i|\mathbf{z}_{i+1}) - \ln q_\phi(\mathbf{z}_i|\mathbf{z}_{i-1})\right) + \right. \\
&\left. + \ln p_\theta(\mathbf{z}_T) - \ln q_\phi(\mathbf{z}_T|\mathbf{z}_{T-1}) + \right.
\end{aligned}$$

$$+ \ln p_\theta(\mathbf{z}_1|\mathbf{z}_2) - \ln q_\phi(\mathbf{z}_1|\mathbf{x})\Big] \tag{5.91}$$
$$\stackrel{df}{=} \mathcal{L}(\mathbf{x};\theta,\phi).$$

We can rewrite the ELBO in terms of Kullback-Leibler divergences (note that we use the expected value with respect to $Q_\phi(\mathbf{z}_{-i}|\mathbf{x})$ to highlight that a proper variational posterior is used for the definition of the Kullback-Lebler divergence):

$$\begin{aligned}\mathcal{L}(\mathbf{x};\theta,\phi) = \; & \mathbb{E}_{Q_\phi(\mathbf{z}_{1:T}|\mathbf{x})}\left[\ln p_\theta(\mathbf{x}|\mathbf{z}_1)\right] + \\ & - \sum_{i=2}^{T-1} \mathbb{E}_{Q_\phi(\mathbf{z}_{-i}|\mathbf{x})}\left[KL\left[q_\phi(\mathbf{z}_i|\mathbf{z}_{i-1})||p_\theta(\mathbf{z}_i|\mathbf{z}_{i+1})\right]\right] + \\ & - \mathbb{E}_{Q_\phi(\mathbf{z}_{-T}|\mathbf{x})}\left[KL\left[q_\phi(\mathbf{z}_T|\mathbf{z}_{T-1})||p_\theta(\mathbf{z}_T)\right]\right] + \\ & - \mathbb{E}_{Q_\phi(\mathbf{z}_{-1}|\mathbf{x})}\left[KL\left[q_\phi(\mathbf{z}_1|\mathbf{x})||p_\theta(\mathbf{z}_1|\mathbf{z}_2)\right]\right].\end{aligned} \tag{5.92}$$

**Example**

Let us take $T = 5$. This is not much (e.g., [76] uses $T = 1000$) but it is easier to explain the idea with a very specific model. Moreover, let us use a fixed $\beta_t \equiv \beta$. Then we have the following DDGM:

$$p_\theta(\mathbf{x}, \mathbf{z}_{1:5}) = p_\theta(\mathbf{x}|\mathbf{z}_1)p_\theta(\mathbf{z}_1|\mathbf{z}_2)p_\theta(\mathbf{z}_2|\mathbf{z}_3)p_\theta(\mathbf{z}_3|\mathbf{z}_4)p_\theta(\mathbf{z}_4|\mathbf{z}_5)p_\theta(\mathbf{z}_5), \tag{5.93}$$

and the variational posteriors:

$$Q_\phi(\mathbf{z}_{1:5}|\mathbf{x}) = q_\phi(\mathbf{z}_1|\mathbf{x})q_\phi(\mathbf{z}_2|\mathbf{z}_1)q_\phi(\mathbf{z}_3|\mathbf{z}_2)q_\phi(\mathbf{z}_4|\mathbf{z}_3)q_\phi(\mathbf{z}_5|\mathbf{z}_4). \tag{5.94}$$

In the considered case, the ELBO takes the following form:

$$\begin{aligned}\mathcal{L}(\mathbf{x};\theta,\phi) = & \mathbb{E}_{Q_\phi(\mathbf{z}_{1:5}|\mathbf{x})}\left[\ln p_\theta(\mathbf{x}|\mathbf{z}_1)\right] + \\ & - \sum_{i=2}^{4} \mathbb{E}_{Q_\phi(\mathbf{z}_{-i}|\mathbf{x})}\left[KL\left[q_\phi(\mathbf{z}_i|\mathbf{z}_{i-1})||p_\theta(\mathbf{z}_i|\mathbf{z}_{i+1})\right]\right] + \\ & - \mathbb{E}_{Q_\phi(\mathbf{z}_{-i}|\mathbf{x})}\left[KL\left[q_\phi(\mathbf{z}_5|\mathbf{z}_4)||p_\theta(\mathbf{z}_5)\right]\right] + \\ & - \mathbb{E}_{Q_\phi(\mathbf{z}_{-i}|\mathbf{x})}\left[KL\left[q_\phi(\mathbf{z}_1|\mathbf{x})||p_\theta(\mathbf{z}_1|\mathbf{z}_2)\right]\right],\end{aligned} \tag{5.95}$$

where

$$p_\theta(\mathbf{z}_5) = \mathcal{N}(\mathbf{z}_5|0,\mathbf{I}). \tag{5.96}$$

The last interesting question is how to model inputs and, eventually, what distribution we should use to model $p(\mathbf{x}|\mathbf{z}_1)$. So far, we used the categorical distribution because pixels were integer-valued. However, for the DDGM,

we assume they are continuous and we will use a simple trick. We normalize our inputs to values between $-1$ and 1 and apply the Gaussian distribution with the unit variance and the mean being constrained to $[-1, 1]$ using the tanh nonlinearity:

$$p(\mathbf{x}|\mathbf{z}_1) = \mathcal{N}(\mathbf{x}|\tanh(NN(\mathbf{z}_1)), \mathbf{I}), \tag{5.97}$$

where $NN(\mathbf{z}_1)$ is a neural network. As a result, since the variance is one, $\ln p(\mathbf{x}|\mathbf{z}_1) = -MSE(\mathbf{x}, \tanh(NN(\mathbf{z}_1))) + const$, so it is equivalent to the (negative) *Mean Squared Error*! I know, it is not a perfect way to do but it is simple and it works.

That's it! As you can see, there is no special magic here, and we are ready to implement our DDGM. In fact, we can use the code of a hierarchical VAE and modify it accordingly. What is convenient about the DDGM is that the forward diffusion (i.e., the variational posteriors) is fixed and we need to sample from them, and only the reverse diffusion requires applying DDNs. But without any further mumbling, let us dive into the code!

#### 5.5.3.3 Code

At this point, you might think that it is pretty complicated and a lot of math is involved here. However, if you followed our previous discussions on VAEs, it should be rather clear what we need to do here.

```
class DDGM(nn.Module):
    def __init__(self, p_dnns, decoder_net, beta, T, D):
        super(DDGM, self).__init__()

        print('DDGM by JT.')

        self.p_dnns = p_dnns  # a list of sequentials; a single
    Sequential defines a DNN to parameterize a distribution p(z_i
     | z_i+1)

        self.decoder_net = decoder_net # the last DNN for p(x|z1)

        # other params
        self.D = D # the dimensionality of the inputs (necessary
    for sampling!)

        self.T = T # the number of steps

        self.beta = torch.FloatTensor([beta]) # the fixed
    variance of diffusion

    # The reparameterization trick for the Gaussian distribution
    @staticmethod
```

```
    def reparameterization(mu, log_var):
        std = torch.exp(0.5*log_var)
        eps = torch.randn_like(std)
        return mu + std * eps

    # The reparameterization trick for the Gaussian forward diffusion
    def reparameterization_gaussian_diffusion(self, x, i):
        return torch.sqrt(1. - self.beta) * x + torch.sqrt(self.beta) * torch.randn_like(x)

    def forward(self, x, reduction='avg'):
        # =====
        # Forward Diffussion
        # Please note that we just "wander" around in the space using Gaussian random walk.
        # We save all z's in a list
        zs = [self.reparameterization_gaussian_diffusion(x, 0)]

        for i in range(1, self.T):
            zs.append(self.reparameterization_gaussian_diffusion(zs[-1], i))

        # =====
        # Backward Diffusion
        # We start with the last z and proceed to x.
        # At each step, we calculate means and variances.
        mus = []
        log_vars = []

        for i in range(len(self.p_dnns) - 1, -1, -1):
            h = self.p_dnns[i](zs[i+1])
            mu_i, log_var_i = torch.chunk(h, 2, dim=1)
            mus.append(mu_i)
            log_vars.append(log_var_i)

        # The last step: outputting the means for x.
        # NOTE: We assume the last distribution is Normal(x | tanh(NN(z_1)), 1)!
        mu_x = self.decoder_net(zs[0])

        # =====ELBO
        # RE
        # This is equivalent to - MSE(x, mu_x) + const
        RE = log_standard_normal(x - mu_x).sum(-1)

        # KL: We need to go through all the levels of latents
        KL = (log_normal_diag(zs[-1], torch.sqrt(1. - self.beta) * zs[-1], torch.log(self.beta)) - log_standard_normal(zs[-1])).sum(-1)

        for i in range(len(mus)):
```

```
            KL_i = (log_normal_diag(zs[i], torch.sqrt(1. - self.
    beta) * zs[i], torch.log(self.beta)) - log_normal_diag(zs[i],
     mus[i], log_vars[i])).sum(-1)

            KL = KL + KL_i

        # Final ELBO
        if reduction == 'sum':
            loss = -(RE - KL).sum()
        else:
            loss = -(RE - KL).mean()

        return loss

    # Sampling is the reverse diffusion with sampling at each
    step.
    def sample(self, batch_size=64):
        z = torch.randn([batch_size, self.D])
        for i in range(len(self.p_dnns) - 1, -1, -1):
            h = self.p_dnns[i](z)
            mu_i, log_var_i = torch.chunk(h, 2, dim=1)
            z = self.reparameterization(torch.tanh(mu_i),
    log_var_i)

        mu_x = self.decoder_net(z)

        return mu_x

    # For sanity check, we also can sample from the forward
    diffusion.
    # The result should resemble a white noise.
    def sample_diffusion(self, x):
        zs = [self.reparameterization_gaussian_diffusion(x, 0)]

        for i in range(1, self.T):
            zs.append(self.reparameterization_gaussian_diffusion(
    zs[-1], i))

        return zs[-1]
```

**Listing 5.13** A DDGM class.

That's it! Now we are ready to run the full code. After training our DDGM, we should obtain results like in Fig. 5.24.

#### 5.5.3.4 Discussion

Extensions

Currently, DDGMs are very popular deep generative models. What we present here is very close to the original formulation of the DDGMs [77]. However, [76] introduced many interesting insights and improvements on the original idea, such as:

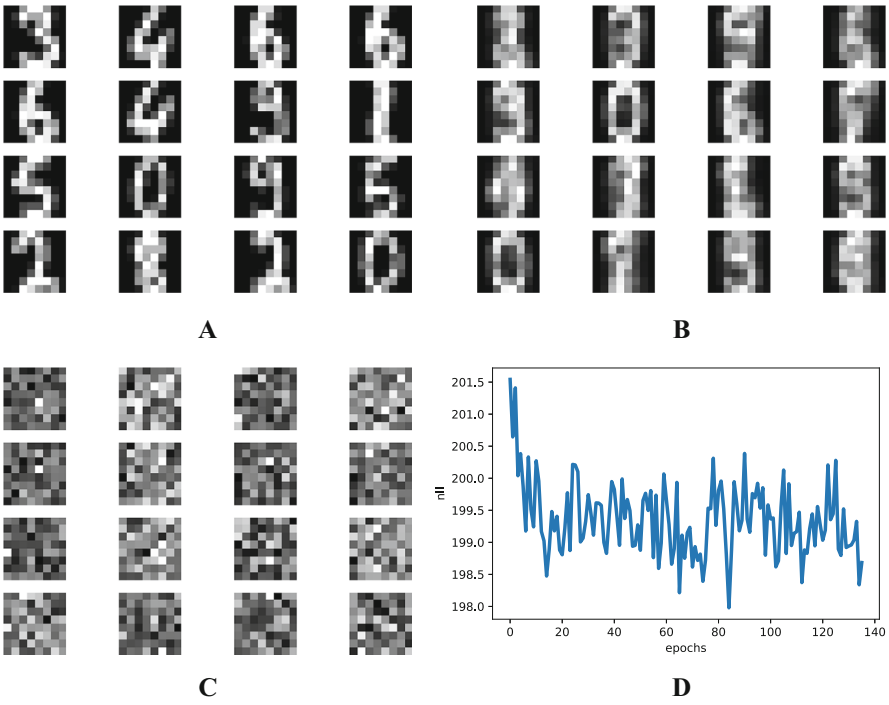



**Fig. 5.24** Outcomes after the training: (**a**) Randomly selected real images. (**b**) Unconditional generations from the DDGM. (**c**) A visualization of the last stochastic level after applying the forward diffusion. As expected, the resulting images resemble pure noise. (**d**) An example of a validation curve for the ELBO.

- Since the forward diffusion consists of Gaussian distributions and linear transformations of means, it is possible to analytically marginalize out intermediate steps that yields:

$$q(\mathbf{z}_t|\mathbf{x}) = \mathcal{N}(\mathbf{z}_t|\sqrt{\bar{\alpha}_t}\mathbf{x}, (1-\bar{\alpha}_t)\mathbf{I}), \tag{5.98}$$

  where $\alpha_t = 1 - \beta_t$ and $\bar{\alpha}_t = \prod_{s=1}^{t} \alpha_s$. This is an extremely interesting result, because we can sample $\mathbf{z}_t$ without sampling all intermediate steps!
- As a follow-up, we can calculate also the following distribution:

$$q(\mathbf{z}_{t-1}|\mathbf{z}_t, \mathbf{x}) = \mathcal{N}\left(\mathbf{z}_{t-1}|\tilde{\boldsymbol{\mu}}_t(\mathbf{z}_t, \mathbf{x}), \tilde{\beta}_t\mathbf{I}\right), \tag{5.99}$$

  where:

$$\tilde{\boldsymbol{\mu}}_t(\mathbf{z}_t, \mathbf{x}) = \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t}\mathbf{x} + \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\mathbf{z}_t \tag{5.100}$$

  and

$$\tilde{\beta}_t = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t. \tag{5.101}$$

Then, we can rewrite the ELBO as follows:

$$\mathcal{L}(\mathbf{x};\theta,\phi) = \mathbb{E}_Q\Bigg[\underbrace{KL\left[q(\mathbf{z}_T|\mathbf{x})\|p(\mathbf{z}_T)\right]}_{L_T} + \\ + \sum_{t>1}\underbrace{KL\left[q(\mathbf{z}_{t-1}|\mathbf{z}_t,\mathbf{x})\|p_\theta(\mathbf{z}_{t-1}|\mathbf{z}_t)\right]}_{L_{t-1}} + \\ \underbrace{-\log p_\theta\left(\mathbf{x}\mid\mathbf{z}_1\right)}_{L_0}\Bigg]. \tag{5.102}$$

Now, instead of differentiating all components of the objective, we can randomly pick $L_t$ and treat it as the objective. Such an approach has a clear advantage: It does not require keeping all gradients in the memory! Instead, we update only one layer at a time. Since our training is stochastic anyway (remember that we typically use stochastic gradient descent), we can introduce this extra stochasticity during training. And the benefit is enormous, because we can train extremely deep models, even with 1000 layers as in [76].

- If you play a little with the code here, you may notice that training the reverse diffusion is pretty problematic. Why? Because by adding extra layers of latents, we add additional KL-terms. In the case of far-from-perfect models, each KL-term will be strictly greater than 0, and, thus, we will increase the ELBO with each additional step of stochasticity. Therefore, it is so important to be smart about formulating reverse diffusion. Ho et al. [76] again provides very interesting insight! We skip here the full reasoning, but it turns out that to make the model $p_\theta\left(\mathbf{z}_{t-1}\mid\mathbf{z}_t\right) = \mathcal{N}\left(\mathbf{z}_{t-1}|\mu_\theta\left(\mathbf{z}_t\right),\sigma_t^2\mathbf{I}\right)$ more powerful, $\mu_\theta\left(\mathbf{z}_t\right)$ should be as close as possible to $\tilde{\boldsymbol{\mu}}_t\left(\mathbf{z}_t,\mathbf{x}\right)$. Following the derivation in [76], we get:

$$\mu_\theta\left(\mathbf{z}_t\right) = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{z}_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(\mathbf{z}_t)\right),$$

where $\epsilon_\theta(\mathbf{z}_t)$ is parameterized by a DNN and it aims to estimate the noise from $\mathbf{z}_t$.
- Even further, each $L_t$ could be simplified to:

$$L_{t,\text{simple}} = \mathbb{E}_{t,\mathbf{x}_0,\boldsymbol{\epsilon}}\left[\left\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta\left(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t\right)\right\|^2\right].$$

Ho et al. [76] provides empirical results that such an objective could be beneficial for training and the final quality of synthesized images.

There were many follow-ups on [76], but we mention only a few of them here:

- *Improving DDGMs*: Nichol and Dhariwal [86] introduces further tricks on improving training stability and performance of DDGMs by learning the covariance matrices in the reverse diffusion, proposing a different noise schedule, among

others. Interestingly, the authors of [78] propose to extend the observables (i.e., pixels) and latents with Fourier features as additional channels. The rationale behind this is that the high-frequency features allow neural networks to cope better with noise. Moreover, they introduce a new noise scheduling and an application of certain functions to improve the numerical stability of the forward diffusion process.

- *Sampling speed-up*: Kong and Ping [87] and Watson et al. [88] focus on speeding up the sampling process.
- *Superresolution*: Saharia et al. [79] uses DDGMs for the task of superresolution.
- *Connection to score-based generative models*: It turns out that score-based generative models [89] are closely related to DDGMs as indicated by [89, 90]. This perspective gives a neat connection between DDGMs and stochastic differential equations [83, 85].
- *Variational perspective on DDGMs*: There is a nice variational perspective on DDGMs [78, 83] that gives an intuitive interpretation of DDGMs and allows achieving astonishing results on the image synthesis task. It is worth studying [78] further, because there are a lot of interesting improvements presented therein.
- *Denoising-generative perspective*: Diffusion-based models could be seen as denoising auto-encoders with a hierarchical prior on top. This division into a denoising part and a generative part was proposed in [91].
- *Discrete DDGMs*: So far, DDGMs are mainly focused on continuous spaces. Austin et al. [81] and Hoogeboom et al. [82] propose DDGMs on discrete spaces.
- *DDGMs for audio*: Kong et al. [80] proposes to use DDGMs for audio synthesis.
- *DDGMs as priors in VAEs*: Vahdat et al. [92], Wehenkel and Louppe [93] propose to use DDGMs as flexible priors in VAEs.

### DDGMs vs. VAEs vs. Flows

In the end, it is worth making a comparison of DDGMs with VAEs and flow-based models. In Table 5.1, we provide a comparison based on rather arbitrary criteria:

- whether the training procedure is stable or not;
- whether the likelihood could be calculated;
- whether a reconstruction is possible;
- whether a model is invertible;
- whether the latent representation could be lower-dimensional than the input space (i.e., a bottleneck in a model).

**Table 5.1** A comparison among DDGMs, VAEs, and flows.

| Model | Training | Likelihood | Reconstruction | Invertible | Bottleneck (latents) |
|---|---|---|---|---|---|
| DDGMs | Stable | Approximate | Difficult | No | No |
| VAEs | Stable | Approximate | Easy | No | Possible |
| Flows | Stable | Exact | Easy | Yes | No |

The three models share a lot of similarities. Overall, training is rather stable even though numerical issues could arise in all models. Hierarchical VAEs could be seen as a generalization of DDGMs. There is an open question of whether it is indeed more beneficial to use fixed variational posteriors by sacrificing the possibility of having a bottleneck. There is also a connection between flows and DDGMs. Both classes of models aim to go from data to noise. Flows do that by applying invertible transformations, while DDGMs accomplish that by a diffusion process. In flow-based models, we know the inverse, but we pay the price of calculating the Jacobian determinant, while DDGMs require flexible parameterizations of the reverse diffusion, but there are no extra strings attached. Looking into connections among these models is definitely an interesting research line.

#### 5.5.3.5 Further Discussion

Diffusion-based models have attracted a lot of attention due to their recent successes in (conditional) image synthesis [94, 95]. Since their original introduction [77], there has been a lot of effort into improving their performance by utilizing powerful parameterizations and tweaking with training details, e.g., [76, 78, 86].

Interestingly, diffusion-based models could be seen as hierarchical latent variable models with $T$ levels of stochastic variables, each $\mathbf{z}_t$ belongs to $\mathcal{X}$, trained with variational inference. The family of variational posteriors in these models is peculiar, since it is defined by the Gaussian diffusion, namely:

$$q_\phi(\mathbf{z}_{1:T}|\mathbf{x}) = \left[\prod_{t=1}^{T-1} q_\phi(\mathbf{z}_t|\mathbf{z}_{t-1})\right] q_\phi(\mathbf{z}_T|\mathbf{z}_{T-1}), \tag{5.103}$$

<p>where $q_\phi(\mathbf{z}_t|\mathbf{z}_{t-1}) = \mathcal{N}(\mathbf{z}_t|\sqrt{1-\beta_t}\mathbf{z}_{t-1}, \beta_t\mathbf{I})$ and we define $\mathbf{z}_0 \equiv \mathbf{x}$. In the context of diffusion-based models, variational posteriors form the *forward diffusion*. The goal of the forward diffusion is to consecutively add (Gaussian) noise such that the last $\mathbf{z}_T$ follows the standard Gaussian distribution.

The *generative* part (a.k.a., the *backward diffusion*) is defined as follows:

$$p_\lambda(\mathbf{z}_{1:T}) = \left[\prod_{t=1}^{T-1} p_\lambda(\mathbf{z}_t|\mathbf{z}_{t+1})\right] p_\lambda(\mathbf{z}_T), \tag{5.104}$$

where all distributions are parameterized by neural networks (or a single, shared neural network [76]). Contrary to forward diffusion, backward diffusion aims at removing noise in such a way that, eventually, it ends up with an object (e.g., an image).

Since diffusion-based models are latent variable models with variational posteriors, we can derive the ELBO by using the ELBO for a single-level latent variable model as a starting point, that is:

$$\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}(\mathbf{x})}\left[\ln p_{\text{model}}(\mathbf{x})\right] \geq \underbrace{\mathbb{E}_{\mathbf{x},\mathbf{z}_1\sim q_\phi(\mathbf{x},\mathbf{z}_1)}\left[\ln p_\theta(\mathbf{x}|\mathbf{z}_1)\right]}_{\stackrel{df}{=}\text{RE}(\phi,\theta)} +$$

$$- D_{\text{KL}}\left[q_\phi(\mathbf{z}_{1:T})\|p_\lambda(\mathbf{z}_{1:T})\right] - \mathbb{I}_{q_\phi(\mathbf{x},\mathbf{z}_{1:T})}\left[\mathbf{x};\mathbf{z}_{1:T}\right] \tag{5.105}$$

$$\geq \text{RE}(\phi,\theta) - \mathbb{E}_{\mathbf{x},\mathbf{z}_{1:T}\sim q_\phi(\mathbf{z}_{1:T},\mathbf{x})}\left[\sum_{t=1}^{T-1}\ln\frac{q_\phi(\mathbf{z}_t|\mathbf{z}_{t-1})}{p_\lambda(\mathbf{z}_t|\mathbf{z}_{t+1})} + \ln\frac{q_\phi(\mathbf{z}_T|\mathbf{z}_{T-1})}{p_\lambda(\mathbf{z}_T)}\right] +$$

$$- \sum_{t=1}^{T}\mathbb{I}_{q_\phi(\mathbf{x},\mathbf{z}_{1:T})}[\mathbf{z}_t;\mathbf{x}|\mathbf{z}_{1:t-1}] \tag{5.106}$$

$$= \text{RE}(\phi,\theta) - \mathbb{E}_{\mathbf{x},\mathbf{z}_{1:T}\sim q_\phi(\mathbf{z}_{1:T},\mathbf{x})}\left[\sum_{t=1}^{T-1}\ln\frac{q_\phi(\mathbf{z}_t|\mathbf{z}_{t-1})}{p_\lambda(\mathbf{z}_t|\mathbf{z}_{t+1})} + \ln\frac{q_\phi(\mathbf{z}_T|\mathbf{z}_{T-1})}{p_\lambda(\mathbf{z}_T)}\right] +$$

$$- \sum_{t=2}^{T}\mathbb{I}_{q_\phi(\mathbf{x},\mathbf{z}_{1:T})}\left[\mathbf{x};\mathbf{z}_t|\mathbf{z}_{1:t-1}\right] - \mathbb{I}_{q_\phi(\mathbf{x},\mathbf{z}_1)}\left[\mathbf{x};\mathbf{z}_1\right] \tag{5.107}$$

$$= \text{RE}(\phi,\theta) - \mathbb{E}_{\mathbf{x},\mathbf{z}_{1:T}\sim q_\phi(\mathbf{z}_{1:T},\mathbf{x})}\left[\sum_{t=1}^{T-1}\ln\frac{q_\phi(\mathbf{z}_t|\mathbf{z}_{t-1})}{p_\lambda(\mathbf{z}_t|\mathbf{z}_{t+1})} + \ln\frac{q_\phi(\mathbf{z}_T|\mathbf{z}_{T-1})}{p_\lambda(\mathbf{z}_T)}\right] +$$

$$- \sum_{t=2}^{T}\mathbb{I}_{q_\phi(\mathbf{x},\mathbf{z}_{1:T})}\left[\mathbf{x};\mathbf{z}_t|\mathbf{z}_{t-1}\right] - \mathbb{I}_{q_\phi(\mathbf{x},\mathbf{z}_1)}\left[\mathbf{x};\mathbf{z}_1\right]. \tag{5.108}$$

In the equations above, we used the following two facts:

1. The convexity of the Kullback-Leibler divergence:

$$D_{\text{KL}}\left[\sum_i \pi_i q_i(\mathbf{z})||\sum_i \pi_i p_i(\mathbf{z})\right] \leq \sum_i \pi_i D_{\text{KL}}\left[q_i(\mathbf{z})||p_i(\mathbf{z})\right],$$

   where $\sum_i \pi_i = 1$.
2. The chain rule for the mutual information: $\mathbb{I}[\mathbf{z}_{1:T};\mathbf{x}] = \sum_{t=1}^{T}\mathbb{I}[\mathbf{z}_t;\mathbf{x}|\mathbf{z}_{1:t-1}]$.
3. Since $q_\phi(\mathbf{z}_t|\mathbf{z}_{1:t-1}) = q_\phi(\mathbf{z}_t|\mathbf{z}_{t-1})$, we get

$$\mathbb{I}_{q_\phi(\mathbf{x},\mathbf{z}_{1:T})}\left[\mathbf{x};\mathbf{z}_t|\mathbf{z}_{1:t-1}\right] = \mathbb{I}_{q_\phi(\mathbf{x},\mathbf{z}_{1:T})}\left[\mathbf{x};\mathbf{z}_t|\mathbf{z}_{t-1}\right].$$

**Mutual information terms are constants!** Let us quickly remind our findings after analyzing the ELBO for a general class of latent variable models. The main puzzling outcome is the negative mutual information that pushes latent variables and observable variables to be stochastically independent. However, the situation for diffusion-based models is completely different, because the family of variational posteriors is fixed (non-trainable). The consequences are twofold. First, the negative mutual information terms, $\sum_{t=2}^{T}\mathbb{I}_{q_\phi(\mathbf{x},\mathbf{z}_{1:T})}\left[\mathbf{x};\mathbf{z}_t|\mathbf{z}_{t-1}\right]$ and $\mathbb{I}_{q_\phi(\mathbf{x},\mathbf{z}_1)}\left[\mathbf{x};\mathbf{z}_1\right]$, are constants for given $\mathbf{x}$ in the view of optimization, since the forward diffusion is not adaptive (it is *just* about adding a fixed amount of Gaussian noise). Second,

$\mathbb{I}_{q_\phi(\mathbf{x},\mathbf{z}_1)}[\mathbf{x};\mathbf{z}_1]$ is a nonzero, constant component. Interestingly, all other mutual information terms are actually zero! To see that, we must remember that the assumed stochastic dependencies in the variational posteriors are the following:

$$\mathbf{x} \rightarrow \mathbf{z}_1 \rightarrow \ldots \rightarrow \mathbf{z}_{t-1} \rightarrow \mathbf{z}_t \rightarrow \ldots \rightarrow \mathbf{z}_T, \tag{5.109}$$

thus, once $\mathbf{z}_{t-1}$ is given, we get $q(\mathbf{z}_t|\mathbf{z}_{1:t-1}, \mathbf{x}) = q(\mathbf{z}_t|\mathbf{z}_{t-1})$. Then, a single mutual information term is the following:

$$\begin{aligned}
\mathbb{I}_{q_\phi(\mathbf{x},\mathbf{z}_{1:T})}[\mathbf{x};\mathbf{z}_t|\mathbf{z}_{t-1}] &= D_{\mathrm{KL}}\left[q_\phi(\mathbf{x},\mathbf{z}_t|\mathbf{z}_{t-1})\|q_\phi(\mathbf{z}_t|\mathbf{z}_{t-1})\,q_\phi(\mathbf{x}|\mathbf{z}_{t-1})\right] && (5.110)\\
&= D_{\mathrm{KL}}\left[q_\phi(\mathbf{z}_t|\mathbf{x},\mathbf{z}_{t-1})\,q_\phi(\mathbf{x}|\mathbf{z}_{t-1})\|q_\phi(\mathbf{z}_t|\mathbf{z}_{t-1})\,q_\phi(\mathbf{x}|\mathbf{z}_{t-1})\right] && (5.111)\\
&= D_{\mathrm{KL}}\left[q_\phi(\mathbf{z}_t|\mathbf{z}_{t-1})\,q_\phi(\mathbf{x}|\mathbf{z}_{t-1})\|q_\phi(\mathbf{z}_t|\mathbf{z}_{t-1})\,q_\phi(\mathbf{x}|\mathbf{z}_{t-1})\right] && (5.112)\\
&= 0, && (5.113)
\end{aligned}$$

where we used the property of the stochastic structure, i.e., $q(\mathbf{z}_t|\mathbf{x},\mathbf{z}_{t-1}) = q(\mathbf{z}_t|\mathbf{z}_{t-1})$. We will look into this result in the next post as well.

Now, we can look into our hypothesis about latent variable models, namely:

**Hypothesis**

To successfully learn a latent variable model with variational inference, one needs to define such a family of variational distributions for which the mutual information between latents and observables is not minimized.

In the case of diffusion-based models, we achieve exactly that! By formulating variational posteriors in a very specific manner, i.e., the Gaussian diffusion, the mutual information terms are not optimized at all. One may immediately say that it is not the only component that matters here and makes these models so successful; it is true. However, by not minimizing the mutual information terms, we force the optimization procedure to further minimize the reconstruction error and minimize the Kullback-Leibler terms. This is precisely what we want to achieve! On one hand, the reconstructions should be as good as possible; on the other hand, each Kullback-Leibler term should be as small as possible (in the case of diffusion-based models, this corresponds to removing Gaussian noise).

The next question is whether fulfilling the hypothesis alone is enough to learn a successful latent variable model. Obviously, it is not true either because, trivially, we need to optimize the other terms. Thus, another important topic is the parameterization of the backward diffusion. The fashion we parameterize the model will result in a better or worse performance in the end. However, from the optimization perspective though, the objective function is extremely important, and, as mentioned above, now the optimizer focuses entirely on the reconstruction error and the Kullback-Leibler term. This is the strength of the diffusion-based models!

It seems that training a latent variable model by applying variational inference could be challenging. Moreover, it is rather apparent that treating latent variables as a data representation and using variational inference for this purpose makes little sense. The mutual information in the ELBO clearly indicates that the goal is the very opposite: Make the stochastic dependency between $\mathbf{x}$ and $\mathbf{z}$ as small as possible! I think we can all agree that this statement has clearly nothing in common with representation learning.

**Self-supervised perspective** There is also another interesting perspective that is very specific to diffusion-based model. Since it is possible to analytically calculate $q_\phi(\mathbf{z}_t|\mathbf{x})$ and $q_\phi(\mathbf{z}_t|\mathbf{z}_{t+1}, \mathbf{x})$ due to Gaussianity and linearity (see [76] for details), we can express the ELBO as follows:

$$\begin{aligned}\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}(\mathbf{x})}\left[\ln p_{\text{model}}(\mathbf{x})\right] \geq \text{RE}(\phi,\theta)+ \\ -\sum_{t=1}^{T-1}\mathbb{E}_{q(\mathbf{z}_t,\mathbf{x})}\left[D_{\text{KL}}\left[q_\phi(\mathbf{z}_t|\mathbf{z}_{t+1},\mathbf{x})\|p_\lambda(\mathbf{z}_t|\mathbf{z}_{t+1})\right]\right]+ \\ -\mathbb{E}_{q(\mathbf{z}_T,\mathbf{x})}\left[D_{\text{KL}}\left[q_\phi(\mathbf{z}_T|\mathbf{x})\|p_\lambda(\mathbf{z}_T)\right]\right]. \qquad (5.114)\end{aligned}$$

Now, it is possible to easily sample $\mathbf{z}_t$, because we know the analytical form of $q_\phi(\mathbf{z}_t|\mathbf{x})$, which is again Gaussian. Moreover, we can express $q_\phi(\mathbf{z}_t|\mathbf{z}_{t+1}, \mathbf{x})$ in a closed form, so every $D_{\text{KL}}$ term could be calculated analytically as well (due to Gaussianity). As a result, as mentioned by [76], it is possible to sample $t$ and optimize the ELBO stochastically, without the necessity of optimizing all $T$ steps. Why this is useful? Because it could be seen as self-supervised learning! For given $\mathbf{x}$, we can obtain its noisy version $\mathbf{z}_t$, and then we learn how to remove this noise by optimizing $D_{\text{KL}}$ at the $(t-1)$-th level. In other words, we take the original image, select how much noise we want to add (this is equivalent to selecting $t$), and then learn a model to remove a bit of this noise (i.e., going from $\mathbf{z}_{t+1}$ to $\mathbf{z}_t$). In this setup, noisy pixels at the $t$-th level could be seen as labels for the *more* noisy pixels at the $(t+1)$-th level.

The self-supervised learning perspective is not necessarily connected with the discussed hypothesis, and it is rather a *post factum* observation. Besides, there are many facets of self-supervised learning; hence, it is arguable whether that is the main reason why diffusion-based models work so well. Nevertheless, the choice of the family of variational distributions that results in the diffusion-based model is definitely important if not essential even.

## References

1. Christopher M Bishop. *Pattern recognition and machine learning*. Springer, 2006.
2. Michael E Tipping and Christopher M Bishop. Probabilistic principal component analysis. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 61(3):611–622, 1999.

3. Christophe Andrieu, Nando De Freitas, Arnaud Doucet, and Michael I Jordan. An introduction to mcmc for machine learning. *Machine learning*, 50(1):5–43, 2003.
4. Michael I Jordan, Zoubin Ghahramani, Tommi S Jaakkola, and Lawrence K Saul. An introduction to variational methods for graphical models. *Machine learning*, 37(2):183–233, 1999.
5. Yoon Kim, Sam Wiseman, Andrew Miller, David Sontag, and Alexander Rush. Semi-amortized variational autoencoders. In *International Conference on Machine Learning*, pages 2678–2687. PMLR, 2018.
6. Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.
7. Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic backpropagation and approximate inference in deep generative models. In *International conference on machine learning*, pages 1278–1286. PMLR, 2014.
8. Luc Devroye. Random variate generation in one line of code. In *Proceedings Winter Simulation Conference*, pages 265–272. IEEE, 1996.
9. Diederik Kingma and Max Welling. Efficient gradient-based inference through transformations between bayes nets and neural nets. In *International Conference on Machine Learning*, pages 1782–1790. PMLR, 2014.
10. Alexander Alemi, Ben Poole, Ian Fischer, Joshua Dillon, Rif A Saurous, and Kevin Murphy. Fixing a broken elbo. In *International Conference on Machine Learning*, pages 159–168. PMLR, 2018.
11. Samuel Bowman, Luke Vilnis, Oriol Vinyals, Andrew Dai, Rafal Jozefowicz, and Samy Bengio. Generating sentences from a continuous space. In *Proceedings of The 20th SIGNLL Conference on Computational Natural Language Learning*, pages 10–21, 2016.
12. Danilo Jimenez Rezende and Fabio Viola. Taming vaes. *arXiv preprint arXiv:1810.00597*, 2018.
13. Eric Nalisnick, Akihiro Matsukawa, Yee Whye Teh, Dilan Gorur, and Balaji Lakshminarayanan. Do deep generative models know what they don't know? In *International Conference on Learning Representations*, 2018.
14. Charline Le Lan and Laurent Dinh. Perfect density models cannot guarantee anomaly detection. *arXiv preprint arXiv:2012.03808*, 2020.
15. Yuri Burda, Roger Grosse, and Ruslan Salakhutdinov. Importance weighted autoencoders. *arXiv preprint arXiv:1509.00519*, 2015.
16. Rianne Van Den Berg, Leonard Hasenclever, Jakub M Tomczak, and Max Welling. Sylvester normalizing flows for variational inference. In *34th Conference on Uncertainty in Artificial Intelligence 2018, UAI 2018*, pages 393–402. Association For Uncertainty in Artificial Intelligence (AUAI), 2018.
17. Emiel Hoogeboom, Victor Garcia Satorras, Jakub M Tomczak, and Max Welling. The convolution exponential and generalized sylvester flows. *arXiv preprint arXiv:2006.01910*, 2020.
18. Durk P Kingma, Tim Salimans, Rafal Jozefowicz, Xi Chen, Ilya Sutskever, and Max Welling. Improved variational inference with inverse autoregressive flow. *Advances in Neural Information Processing Systems*, 29:4743–4751, 2016.

19. Danilo Rezende and Shakir Mohamed. Variational inference with normalizing flows. In *International Conference on Machine Learning*, pages 1530–1538. PMLR, 2015.
20. Jakub M Tomczak and Max Welling. Improving variational auto-encoders using householder flow. *arXiv preprint arXiv:1611.09630*, 2016.
21. Jakub M Tomczak and Max Welling. Improving variational auto-encoders using convex combination linear inverse autoregressive flow. *arXiv preprint arXiv:1706.02326*, 2017.
22. Ishaan Gulrajani, Kundan Kumar, Faruk Ahmed, Adrien Ali Taiga, Francesco Visin, David Vazquez, and Aaron Courville. PixelVAE: A latent variable model for natural images. *arXiv preprint arXiv:1611.05013*, 2016.
23. Jakub Tomczak and Max Welling. VAE with a VampPrior. In *International Conference on Artificial Intelligence and Statistics*, pages 1214–1223. PMLR, 2018.
24. Xi Chen, Diederik P Kingma, Tim Salimans, Yan Duan, Prafulla Dhariwal, John Schulman, Ilya Sutskever, and Pieter Abbeel. Variational lossy autoencoder. *arXiv preprint arXiv:1611.02731*, 2016.
25. Ioannis Gatopoulos and Jakub M Tomczak. Self-supervised variational auto-encoders. *Entropy*, 23(6):747, 2021.
26. Amirhossein Habibian, Ties van Rozendaal, Jakub M Tomczak, and Taco S Cohen. Video compression with rate-distortion autoencoders. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 7033–7042, 2019.
27. Matthias Bauer and Andriy Mnih. Resampled priors for variational autoencoders. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 66–75. PMLR, 2019.
28. Diederik P Kingma, Danilo J Rezende, Shakir Mohamed, and Max Welling. Semi-supervised learning with deep generative models. In *Proceedings of the 27th International Conference on Neural Information Processing Systems*, pages 3581–3589, 2014.
29. Christos Louizos, Kevin Swersky, Yujia Li, Max Welling, and Richard Zemel. The variational fair autoencoder. *arXiv preprint arXiv:1511.00830*, 2015.
30. Maximilian Ilse, Jakub M Tomczak, Christos Louizos, and Max Welling. DIVA: Domain invariant variational autoencoders. In *Medical Imaging with Deep Learning*, pages 322–348. PMLR, 2020.
31. Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. Weight uncertainty in neural network. In *International Conference on Machine Learning*, pages 1613–1622. PMLR, 2015.
32. Wengong Jin, Regina Barzilay, and Tommi Jaakkola. Junction tree variational autoencoder for molecular graph generation. In *International Conference on Machine Learning*, pages 2323–2332. PMLR, 2018.
33. Tim R Davidson, Luca Falorsi, Nicola De Cao, Thomas Kipf, and Jakub M Tomczak. Hyperspherical variational auto-encoders. In *34th Conference on Uncertainty in Artificial Intelligence 2018, UAI 2018*, pages 856–865. Association For Uncertainty in Artificial Intelligence (AUAI), 2018.

34. Tim R Davidson, Jakub M Tomczak, and Efstratios Gavves. Increasing expressivity of a hyperspherical vae. *arXiv preprint arXiv:1910.02912*, 2019.
35. Emile Mathieu, Charline Le Lan, Chris J Maddison, Ryota Tomioka, and Yee Whye Teh. Continuous hierarchical representations with poincaré variational auto-encoders. *Advances in neural information processing systems*, 32, 2019.
36. Sharvaree Vadgama, Jakub M Tomczak, and Erik Bekkers. Continuous Kendall shape variational autoencoders. In *International Conference on Geometric Science of Information*, pages 73–81. Springer, 2023.
37. Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*, 2016.
38. C Maddison, A Mnih, and Y Teh. The concrete distribution: A continuous relaxation of discrete random variables. In *Proceedings of the international conference on learning Representations*. International Conference on Learning Representations, 2017.
39. Emile van Krieken, Jakub M Tomczak, and Annette ten Teije. Storchastic: A framework for general stochastic automatic differentiation. *Advances in Neural Information Processing Systems*, 2021.
40. Junxian He, Daniel Spokoyny, Graham Neubig, and Taylor Berg-Kirkpatrick. Lagging inference networks and posterior collapse in variational autoencoders. *arXiv preprint arXiv:1901.05534*, 2019.
41. Adji B Dieng, Yoon Kim, Alexander M Rush, and David M Blei. Avoiding latent variable collapse with generative skip models. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 2397–2405. PMLR, 2019.
42. Adji B Dieng, Dustin Tran, Rajesh Ranganath, John Paisley, and David M Blei. Variational Inference via $\chi$-Upper Bound Minimization. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 2729–2738, 2017.
43. Irina Higgins, Loic Matthey, Arka Pal, Christopher Burgess, Xavier Glorot, Matthew Botvinick, Shakir Mohamed, and Alexander Lerchner. beta-VAE: Learning basic visual concepts with a constrained variational framework. *ICLR*, 2016.
44. Partha Ghosh, Mehdi SM Sajjadi, Antonio Vergari, Michael Black, and Bernhard Scholkopf. From variational to deterministic autoencoders. In *International Conference on Learning Representations*, 2019.
45. Lars Maaløe, Marco Fraccaro, Valentin Liévin, and Ole Winther. BIVA: A Very Deep Hierarchy of Latent Variables for Generative Modeling. In *NeurIPS*, 2019.
46. Arash Vahdat and Jan Kautz. NVAE: A deep hierarchical variational autoencoder. *arXiv preprint arXiv:2007.03898*, 2020.
47. Rewon Child. Very deep vaes generalize autoregressive models and can outperform them on images. *arXiv preprint arXiv:2011.10650*, 2020.
48. Alireza Makhzani, Jonathon Shlens, Navdeep Jaitly, Ian Goodfellow, and Brendan Frey. Adversarial autoencoders. *arXiv preprint arXiv:1511.05644*, 2015.
49. Matthew D Hoffman and Matthew J Johnson. ELBO surgery: Yet another way to carve up the variational evidence lower bound. In *Workshop in Advances in Approximate Bayesian Inference, NIPS*, volume 1, page 2, 2016.

50. Anna Kuzina, Max Welling, and Jakub Tomczak. Alleviating adversarial attacks on variational autoencoders with mcmc. *Advances in Neural Information Processing Systems*, 35:8811–8823, 2022.
51. Ricky TQ Chen, Xuechen Li, Roger Grosse, and David Duvenaud. Isolating sources of disentanglement in vaes. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 2615–2625, 2018.
52. Frantzeska Lavda, Magda Gregorová, and Alexandros Kalousis. Data-dependent conditional priors for unsupervised learning of multimodal data. *Entropy*, 22(8):888, 2020.
53. Shuyu Lin and Ronald Clark. Ladder: Latent data distribution modelling with a generative prior. *arXiv preprint arXiv:2009.00088*, 2020.
54. Christopher M Bishop, Markus Svensén, and Christopher KI Williams. GTM: The generative topographic mapping. *Neural computation*, 10(1):215–234, 1998.
55. Christian H Bischof and Xiaobai Sun. On orthogonal block elimination. *Preprint MCS-P450–0794, Mathematics and Computer Science Division, Argonne National Laboratory*, page 4, 1994.
56. Xiaobai Sun and Christian Bischof. A basis-kernel representation of orthogonal matrices. *SIAM journal on matrix analysis and applications*, 16(4):1184–1196, 1995.
57. Alston S Householder. Unitary triangularization of a nonsymmetric matrix. *Journal of the ACM (JACM)*, 5(4):339–342, 1958.
58. Leonard Hasenclever, Jakub Tomczak, Rianne van den Berg, and Max Welling. Variational inference with orthogonal normalizing flows. 2017.
59. Åke Björck and Clazett Bowie. An iterative algorithm for computing the best estimate of an orthogonal matrix. *SIAM Journal on Numerical Analysis*, 8(2):358–364, 1971.
60. Zdislav Kovarik. Some iterative methods for improving orthonormality. *SIAM Journal on Numerical Analysis*, 7(3):386–389, 1970.
61. Gary Ulrich. Computer generation of distributions on the m-sphere. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 33(2):158–163, 1984.
62. Christian Naesseth, Francisco Ruiz, Scott Linderman, and David Blei. Reparameterization gradients through acceptance-rejection sampling algorithms. In *Artificial Intelligence and Statistics*, pages 489–498. PMLR, 2017.
63. Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.
64. Ferenc Huszár. Is maximum likelihood useful for representation learning? https://www.inference.vc/maximum-likelihood-for-representation-learning-2/, 2017. [Online; accessed 16-September-2021].
65. Mary Phuong, Max Welling, Nate Kushman, Ryota Tomioka, and Sebastian Nowozin. The mutual autoencoder: Controlling information in latent code representations. https://openreview.net/pdf?id=HkbmWqxCZ, 2018. [Online; accessed 16-September-2021].

66. Samarth Sinha and Adji B Dieng. Consistency regularization for variational auto-encoders. *arXiv preprint arXiv:2105.14859*, 2021.
67. Jakub M Tomczak. Learning informative features from restricted boltzmann machines. *Neural Processing Letters*, 44(3):735–750, 2016.
68. Yoshua Bengio. *Learning deep architectures for AI*. Now Publishers Inc, 2009.
69. Ruslan Salakhutdinov. Learning deep generative models. *Annual Review of Statistics and Its Application*, 2:361–385, 2015.
70. Ruslan Salakhutdinov and Geoffrey Hinton. Deep boltzmann machines. In *Artificial intelligence and statistics*, pages 448–455. PMLR, 2009.
71. Andrew Gelman, John B Carlin, Hal S Stern, and Donald B Rubin. *Bayesian data analysis*. Chapman and Hall/CRC, 1995.
72. Lars Maaløe, Marco Fraccaro, and Ole Winther. Semi-supervised generation with cluster-aware generative models. *arXiv preprint arXiv:1704.00637*, 2017.
73. Casper Kaae Sønderby, Tapani Raiko, Lars Maaløe, Søren Kaae Sønderby, and Ole Winther. Ladder variational autoencoders. *Advances in Neural Information Processing Systems*, 29:3738–3746, 2016.
74. Adeel Pervez and Efstratios Gavves. Spectral smoothing unveils phase transitions in hierarchical variational autoencoders. In *International Conference on Machine Learning*, pages 8536–8545. PMLR, 2021.
75. Bohan Wu, Suraj Nair, Roberto Martin-Martin, Li Fei-Fei, and Chelsea Finn. Greedy hierarchical variational autoencoders for large-scale video prediction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2318–2328, 2021.
76. Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *Advances in Neural Information Processing Systems*, 33:6840–6851, 2020.
77. Jascha Sohl-Dickstein, Eric Weiss, Niru Maheswaranathan, and Surya Ganguli. Deep unsupervised learning using nonequilibrium thermodynamics. In *International Conference on Machine Learning*, pages 2256–2265. PMLR, 2015.
78. Diederik P Kingma, Tim Salimans, Ben Poole, and Jonathan Ho. Variational diffusion models. *arXiv preprint arXiv:2107.00630*, 2021.
79. Chitwan Saharia, Jonathan Ho, William Chan, Tim Salimans, David J Fleet, and Mohammad Norouzi. Image super-resolution via iterative refinement. *arXiv preprint arXiv:2104.07636*, 2021.
80. Zhifeng Kong, Wei Ping, Jiaji Huang, Kexin Zhao, and Bryan Catanzaro. DiffWave: A versatile diffusion model for audio synthesis. In *International Conference on Learning Representations*, 2020.
81. Jacob Austin, Daniel Johnson, Jonathan Ho, Danny Tarlow, and Rianne van den Berg. Structured denoising diffusion models in discrete state-spaces. *arXiv preprint arXiv:2107.03006*, 2021.
82. Emiel Hoogeboom, Didrik Nielsen, Priyank Jaini, Patrick Forré, and Max Welling. Argmax flows and multinomial diffusion: Towards non-autoregressive language models. *arXiv preprint arXiv:2102.05379*, 2021.

83. Chin-Wei Huang, Jae Hyun Lim, and Aaron Courville. A variational perspective on diffusion-based generative models and score matching. *arXiv preprint arXiv:2106.02808*, 2021.
84. Yang Song, Jascha Sohl-Dickstein, Diederik P Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. Score-based generative modeling through stochastic differential equations. In *International Conference on Learning Representations*, 2020.
85. Belinda Tzen and Maxim Raginsky. Neural stochastic differential equations: Deep latent gaussian models in the diffusion limit. *arXiv preprint arXiv:1905.09883*, 2019.
86. Alex Nichol and Prafulla Dhariwal. Improved denoising diffusion probabilistic models. *arXiv preprint arXiv:2102.09672*, 2021.
87. Zhifeng Kong and Wei Ping. On fast sampling of diffusion probabilistic models. *arXiv preprint arXiv:2106.00132*, 2021.
88. Daniel Watson, Jonathan Ho, Mohammad Norouzi, and William Chan. Learning to efficiently sample from diffusion probabilistic models. *arXiv preprint arXiv:2106.03802*, 2021.
89. Yang Song and Stefano Ermon. Generative modeling by estimating gradients of the data distribution. *arXiv preprint arXiv:1907.05600*, 2019.
90. Yang Song and Diederik P Kingma. How to train your energy-based models. *arXiv preprint arXiv:2101.03288*, 2021.
91. Kamil Deja, Anna Kuzina, Tomasz Trzcinski, and Jakub Tomczak. On analyzing generative and denoising capabilities of diffusion-based deep generative models. *Advances in Neural Information Processing Systems*, 35:26218–26229, 2022.
92. Arash Vahdat, Karsten Kreis, and Jan Kautz. Score-based generative modeling in latent space. *arXiv preprint arXiv:2106.05931*, 2021.
93. Antoine Wehenkel and Gilles Louppe. Diffusion priors in variational autoencoders. In *ICML Workshop on Invertible Neural Networks, Normalizing Flows, and Explicit Likelihood Models*, 2021.
94. Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. Hierarchical text-conditional image generation with clip latents. *arXiv preprint arXiv:2204.06125*, 1(2):3, 2022.
95. Chitwan Saharia, William Chan, Saurabh Saxena, Lala Li, Jay Whang, Emily L Denton, Kamyar Ghasemipour, Raphael Gontijo Lopes, Burcu Karagol Ayan, Tim Salimans, et al. Photorealistic text-to-image diffusion models with deep language understanding. *Advances in neural information processing systems*, 35:36479–36494, 2022.

# Chapter 6
# Hybrid Modeling


## 6.1 Introduction

In Chap. 1, I tried to convince you that learning the conditional distribution $p(y|\mathbf{x})$ is not enough and, instead, we should focus on the joint distribution $p(\mathbf{x}, y)$ factorized as follows:

$$p(\mathbf{x}, y) = p(y|\mathbf{x})p(\mathbf{x}). \tag{6.1}$$

Why? Let me remind you of my reasoning. The conditional $p(y|\mathbf{x})$ does not allow us to say anything about $\mathbf{x}$, but, instead, it will do its best to provide a decision. As a result, I can provide an object that has never been observed so far, and $p(y|\mathbf{x})$ could still be pretty certain about its decision (i.e., assigning high probability to one class). On the other hand, once we have trained $p(\mathbf{x})$, we should be able to, at least in theory, access the probability of the given object. And, eventually, determine whether our decision is reliable or not.

In the previous chapters, we completely focused on answering the question of how to learn $p(\mathbf{x})$ alone. Since we had in mind the necessity of using it for evaluating the probability, we discussed only the likelihood-based models, namely, the autoregressive models (ARMs), the flow-based models (flows), and the variational autoencoders (VAEs). Now, the naturally arising question is how to use a deep generative model together with a classifier (or a regressor). Let us focus on a classification task for simplicity and think of possible approaches.

### 6.1.1 Approach 1: Let's Be Naive!

Let us start with some easy, naive almost approaches. In the most straightforward way, we can train $p(y|\mathbf{x})$ and $p(\mathbf{x})$ separately. And that is it, we have a classifier and a marginal distribution over objects. This approach is schematically presented in Fig. 6.1 where we use different colors (purple and blue) to highlight that we use two different neural networks to parameterize the two distributions.



J. M. Tomczak, *Deep Generative Modeling*,
https://doi.org/10.1007/978-3-031-64087-2_6

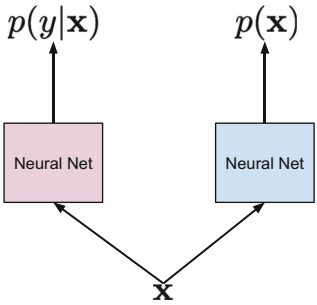



**Fig. 6.1** A naive approach to learning the joint distribution by considering both distributions separately.

Taking the logarithm of the joint distribution yields:

$$\ln p(\mathbf{x}, y) = \ln p_\alpha(y|\mathbf{x}) + \ln p_\beta(\mathbf{x}), \tag{6.2}$$

where $\alpha$ and $\beta$ denote parameterizations of both distributions (i.e., neural networks). Once we start training and calculate gradients with respect to $\alpha$ and $\beta$, we clearly see that we get:

$$\nabla_\alpha \ln p(\mathbf{x}, y) = \nabla_\alpha \ln p_\alpha(y|\mathbf{x}) + \underbrace{\nabla_\alpha \ln p_\beta(\mathbf{x})}_{=0}, \tag{6.3}$$

because $\ln p_\beta(\mathbf{x})$ is not dependent on $\alpha$, and

$$\nabla_\beta \ln p(\mathbf{x}, y) = \underbrace{\nabla_\beta \ln p_\alpha(y|\mathbf{x})}_{=0} + \nabla_\beta \ln p_\beta(\mathbf{x}), \tag{6.4}$$

because $\ln p_\alpha(y|\mathbf{x})$ does not depend on $\beta$.

In other words, we can simply first train $p_\alpha(y|\mathbf{x})$ using all data with labels and then train $p_\beta(\mathbf{x})$ using all available data. So what is a potential pitfall with this approach? Intuitively, we can say that there is no guarantee that both distributions treat $\mathbf{x}$ in the same manner and, thus, could introduce some errors. Moreover, due to the stochasticity during training, there is no information flow between random variables $\mathbf{x}$ and $y$, and as a result, the neural networks seek for own (local) minima. To use a metaphor, they are like two wings of a bird that move in total separation, completely asynchronously. Moreover, training both models separately is also inefficient. We must use two different neural networks, with no weight sharing. Since training is stochastic, we really could worry about potential bad local optima, and our worries are even doubled now.

Would such an approach fail? Well, there is no simple answer to this question. Probably, it could work pretty well even, but it might lead to models far from optimal ones. Either way, who does like being unclear about training models? At least not me.

### 6.1.2 Approach 2: Shared Parameterization!

Alright, so since I whine about sharing the parameterization, it is obvious that the second approach uses (drums here) a shared parameterization! To be more precise, a partially shared parameterization assumes that there is a neural network that processes $\mathbf{x}$ and then its output is fed to two neural networks: one for the classifier and one for the marginal distribution over $\mathbf{x}$s. An example of this approach is depicted in Fig. 6.2 (the shared neural network is shown in purple).

Now, taking the logarithm of the joint distribution gives:

$$\ln p(\mathbf{x}, y) = \ln p_{\alpha,\gamma}(y|\mathbf{x}) + \ln p_{\beta,\gamma}(\mathbf{x}), \tag{6.5}$$

where it is worth highlighting, that both distributions partially share the parameterization $\gamma$ (i.e., the purple neural network in Fig. 6.2). As a result, during training, there is a piece of obvious information sharing between $\mathbf{x}$ and $y$! Intuitively, both distributions operate on a processed $\mathbf{x}$ in the same manner, and then this representation is specialized to give probabilities for classes and objects.

Again, one might ask what is this all fuzz about?! Well, first of all, now, the two distributions are tightly connected. Like in the metaphor of a bird used before, both wings can move together, in a synchronized fashion. Second, from the optimization perspective, the gradients flow through the $\gamma$ network, and, thus, it contains information about both $\mathbf{x}$ and $y$. This may greatly help in finding a better solution.

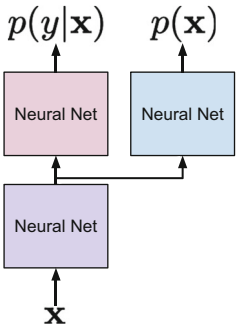



**Fig. 6.2** An approach to learning the joint distribution by using a partially shared parameterization.

## 6.2 Hybrid Modeling

At first glance, there is nothing wrong with learning using the training objective expressed as:

$$\ln p(\mathbf{x}, y) = \ln p_{\alpha,\gamma}(y|\mathbf{x}) + \ln p_{\beta,\gamma}(\mathbf{x}). \tag{6.6}$$

However, let us think about the dimensionalities of $y$ and $\mathbf{x}$. For instance, if $y$ is binary, then we have one single bit representing a class label. For a binary vector of $\mathbf{x}$, we have $D$ bits. Hence, there is a clear discrepancy in scales! Let us take a look at the gradient with respect to $\gamma$ first, namely:

$$\nabla_\gamma \ln p(\mathbf{x}, y) = \nabla_\gamma \ln p_{\alpha,\gamma}(y|\mathbf{x}) + \nabla_\gamma \ln p_{\beta,\gamma}(\mathbf{x}). \tag{6.7}$$

If we think about it, during training, the $\gamma$ network obtains a much stronger signal from $\ln p_{\beta,\gamma}(\mathbf{x})$. Following our example of binary variables, let us assume that our neural nets return all probabilities equal 0.5, so for the independent Bernoulli variables, we get:

$$\begin{aligned} \ln Bern(y|0.5) &= y \ln 0.5 + (1 - y) \ln 0.5 \\ &= -\ln 2. \end{aligned}$$

where we use the property of the logarithm ($\ln 0.5 = \ln 2^{-1} = -\ln 2$) and it does not matter what is the value of $y$ because the neural network returns 0.5 for $y = 0$ and $y = 1$. Similarly, for $\mathbf{x}$, we get:

$$\begin{aligned} \ln \prod_{d=1}^{D} Bern(x_d|0.5) &= \sum_{d=1}^{D} \ln Bern(x_d|0.5) \\ &= -D \ln 2. \end{aligned}$$

Therefore, we see that the $\ln p_{\beta,\gamma}(\mathbf{x})$ part is $D$-times stronger than the $\ln p_{\alpha,\gamma}(y|\mathbf{x})$ part! How does it influence the final gradients during training? Try to visualize a bar of height $\ln 2$ and the other that is $D$-times higher. Now, imagine these bars "flow" through $\gamma$. Do you see it? Yes, the $\gamma$ neural network will obtain more information from the marginal distribution, and this information could cripple the classification part. In other words, our final model will be always *biased toward the marginal part*. Can we do something about it? Fortunately, yes!

In [1], it was proposed to consider the convex combination of $\ln p(y|\mathbf{x})$ and $\ln p(\mathbf{x})$ as the objective function, namely:

$$\mathcal{L}(\mathbf{x}, y; \lambda) = (1 - \lambda) \ln p(y|\mathbf{x}) + \lambda \ln p(\mathbf{x}), \tag{6.8}$$

where $\lambda \in [0, 1]$. Unfortunately, this weighting scheme is not derived from a well-defined distribution, and it breaks the elegance of the likelihood-based approach. However, if you do not mind being inelegant, then this approach should work well!

A different approach is proposed in [2] where only $\ln p(\mathbf{x})$ is weighted:

$$\ell(\mathbf{x}, y; \lambda) = \ln p(y|\mathbf{x}) + \lambda \ln p(\mathbf{x}), \tag{6.9}$$

where $\lambda \geq 0$. This kind of weighting was proposed in various forms before (e.g., see [3, 4]). Still, the fudge factor $\lambda$ is not derived from a probabilistic perspective. However, [2] argues that we can interpret $\lambda$ as a way of encouraging robustness to input variations. They also mention that scaling $\ln p(\mathbf{x})$ can be seen as a Jacobian-based regularization penalty. It is still not a valid distribution (because it is equivalent to $p(\mathbf{x})^{\lambda}$), but at least we can provide some interpretations.

In [2], the hybrid modeling idea has been pursued with $p(\mathbf{x})$ being modeled by flows (in the paper, they used GLOW [5]), and then, the resulting latents $\mathbf{z}$ were used as the input to the classifier. In other words, a flow-based model is used for $p(\mathbf{x})$, and the invertible neural network (e.g., consisting of coupling layers) is shared with the classifier. Then, the final layers on top of the invertible neural network are used to make a decision $y$. The objective function is $\ell(\mathbf{x}, y; \lambda)$ as defined in Eq. 6.9. The approach is schematically presented in Fig. 6.3.

There are a couple of interesting properties of this approach. First, we can use the invertible neural network for both generative and discriminative parts of the model. Hence, the flow-based model is well-informed about the label. Second, the weighting $\lambda$ allows controlling whether the model is *more* discriminative or *more* generative. Third, we can use **any** flow-based model! GLOW was used in [2]; however, [6] used residual flows and [7] applied invertible DenseNets. Fourth, as presented by [2], we can use **any** classifier (or regressor), e.g., Bayesian classifiers.

A potential drawback of this approach lies in the necessity of determining $\lambda$. This is an extra hyperparameter that requires tuning. Moreover, as noticed in previous papers [2, 6, 7], the value of $\lambda$ drastically changes the performance of the model from discriminative to generative. That is an open question of how to deal with that.

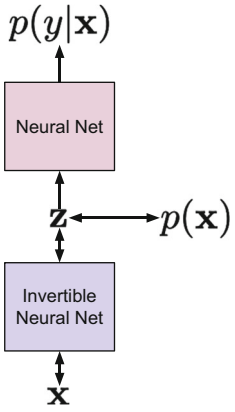



**Fig. 6.3** Hybrid modeling using invertible neural networks and flow-based models.

## 6.3 Let's Implement It!

Now, it is time to be more specific and formulate a hybrid model. Let us start with the classifier and consider a fully connected neural network to model the conditional distribution $p(y|\mathbf{x})$, namely:

$$\mathbf{z} \rightarrow \text{Linear}(D, M) \rightarrow \text{ReLU} \rightarrow \text{Linear}(M, M) \rightarrow \text{ReLU} \rightarrow \\ \rightarrow \text{Linear}(M, K) \rightarrow \text{Softmax}$$

where $D$ is the dimensionality of $\mathbf{x}$ and $K$ is the number of classes. The softmax gives us probabilities for each class. Remember that $\mathbf{z} = f^{-1}(\mathbf{x})$, where $f$ is an invertible neural network.

In our example, we use the classifier, so we should take the categorical distribution for the conditional $p(y|\mathbf{x})$:

$$p(y|\mathbf{x}) = \prod_{k=1}^{K} \theta_k(\mathbf{x})^{[y=k]}, \tag{6.10}$$

where $\theta_k(\mathbf{x})$ is the softmax value for the $k$-th class and $[y = k]$ is the Iverson bracket (i.e., $[y = k] = 1$ if $y$ equals $k$ and 0 otherwise).

Next, we focus on modeling $p(\mathbf{x})$. We can use any marginal model, e.g., we can apply flows and the change of variable formula, namely:

$$p(\mathbf{x}) = \pi\left(\mathbf{z} = f^{-1}(\mathbf{x})\right) \ |\mathbf{J}_f(\mathbf{x})|^{-1}, \tag{6.11}$$

where $\mathbf{J}_f(\mathbf{x})$ denotes the Jacobian of the transformation (i.e., neural network) $f$ evaluated at $\mathbf{x}$. In the case of the flow, we typically use $\pi(\mathbf{z}) = \mathcal{N}(\mathbf{z}|0, 1)$, i.e., the standard Gaussian distribution.

Plugging these all distributions to the objective of the hybrid modeling $\ell(\mathbf{x}, y; \lambda)$, we get:

$$\ell(\mathbf{x}, y; \lambda) = \sum_{k=1}^{K} [y = k] \ln \theta_{k,g,f}(\mathbf{x}) + \lambda \, \mathcal{N}(\mathbf{z} = f^{-1}(\mathbf{x})|0, 1) - \ln |\mathbf{J}_f(\mathbf{x})|. \tag{6.12}$$

where we additionally highlight that $\theta_{k,g,f}$ is parameterized by two neural networks: $f$ from the flow and $g$ for the final classification.

Now, if we would like to follow [2], we could pick **coupling layers** as the components of $f$, and eventually, we would model $p(\mathbf{x})$ using RealNVP or GLOW, for instance. However, we want to be fancier, and we will utilize integer discrete flows (IDFs) [8, 9]. Why? Because we simply can and also IDFs do not require calculating the Jacobian. Besides, we can practice a bit of formulating various hybrid models.

Let us quickly recall IDFs. First, they operate on $\mathbb{Z}^D$, i.e., integers. Second, we need to pick an appropriate $\pi(\mathbf{z})$ that in this case could be the **discretized logistic** (DL), $\text{DL}(z|\mu, \nu)$ with mean $\mu$ and scale $\nu$. Since the change of variable formula for

discrete random variables does not require calculating the Jacobian (remember no change of volume here!), we can rewrite the hybrid modeling objective as follows:

$$\ell(\mathbf{x}, y; \lambda) = \sum_{k=1}^{K} [y = k] \ln \theta_{k,g,f}(\mathbf{x}) + \lambda \, \mathrm{DL}(\mathbf{z} = f^{-1}(\mathbf{x}) | \mu, \nu). \tag{6.13}$$

That's it! Congratulations, if you have followed all these steps, you have arrived at a new hybrid model that uses IDFs to model the distribution of $\mathbf{x}$. Notice that the classifier takes integers as inputs.

## 6.4 Code

We have all the components to implement our own hybrid integer discrete flow (HybridIDF)! Below, there is a code with a lot of comments that should help to understand every single line of it.

```
class HybridIDF(nn.Module):
    def __init__(self, netts, classnet, num_flows, alpha=1., D=2):
        super(HybridIDF, self).__init__()

        print('HybridIDF by JT.')

        # Here we use the two options discussed previously: a coupling layer or a generalized invertible transformation
        # These formulate the transformation f.
        # NOTE: Please pay attention to a new variable here, namely, beta. This is the rezero trick used in (van den Berg et~al., 2020).
        if len(netts) == 1:
            self.t = torch.nn.ModuleList([netts[0]() for _ in range(num_flows)])
            self.idf_git = 1
            self.beta = nn.Parameter(torch.zeros(len(self.t)))

        elif len(netts) == 4:
            self.t_a = torch.nn.ModuleList([netts[0]() for _ in range(num_flows)])
            self.t_b = torch.nn.ModuleList([netts[1]() for _ in range(num_flows)])
            self.t_c = torch.nn.ModuleList([netts[2]() for _ in range(num_flows)])
            self.t_d = torch.nn.ModuleList([netts[3]() for _ in range(num_flows)])
            self.idf_git = 4
            self.beta = nn.Parameter(torch.zeros(len(self.t_a)))

        else:
```

```
            raise ValueError('You can provide either 1 or 4 translation nets.')

        # This contains extra layers for classification on top of z.
        self.classnet = classnet

        # The number of flows (i.e., f's).
        self.num_flows = num_flows

        # The rounding operator.
        self.round = RoundStraightThrough.apply

        # The mean and log-scale for the base distribution pi.
        self.mean = nn.Parameter(torch.zeros(1, D))
        self.logscale = nn.Parameter(torch.ones(1, D))

        # The dimensionality of the input.
        self.D = D

        # Since using "lambda" is confusing for Python, we will use alpha in the code for lambda in previous equations (not confusing at all, right?!)
        self.alpha = alpha

        # We use the built-in PyTorch loss function. It is for educational purposes! Otherwise, we could use the log-categorical.
        self.nll = nn.NLLLoss(reduction='none') #it requires log-softmax as input!!

    # The coupling layer as introduced before.
    # NOTE: We use the rezero trick!
    def coupling(self, x, index, forward=True):

        if self.idf_git == 1:
            (xa, xb) = torch.chunk(x, 2, 1)

            if forward:
                yb = xb + self.beta[index] * self.round(self.t[index](xa))
            else:
                yb = xb - self.beta[index] * self.round(self.t[index](xa))

            return torch.cat((xa, yb), 1)

        elif self.idf_git == 4:
            (xa, xb, xc, xd) = torch.chunk(x, 4, 1)

            if forward:
                ya = xa + self.beta[index] * self.round(self.t_a[index](torch.cat((xb, xc, xd), 1)))
```

```
67                 yb = xb + self.beta[index] * self.round(self.t_b[
    index](torch.cat((ya, xc, xd), 1)))
68                 yc = xc + self.beta[index] * self.round(self.t_c[
    index](torch.cat((ya, yb, xd), 1)))
69                 yd = xd + self.beta[index] * self.round(self.t_d[
    index](torch.cat((ya, yb, yc), 1)))
70             else:
71                 yd = xd - self.beta[index] * self.round(self.t_d[
    index](torch.cat((xa, xb, xc), 1)))
72                 yc = xc - self.beta[index] * self.round(self.t_c[
    index](torch.cat((xa, xb, yd), 1)))
73                 yb = xb - self.beta[index] * self.round(self.t_b[
    index](torch.cat((xa, yc, yd), 1)))
74                 ya = xa - self.beta[index] * self.round(self.t_a[
    index](torch.cat((yb, yc, yd), 1)))
75
76             return torch.cat((ya, yb, yc, yd), 1)
77
78     # The permutation layer.
79     def permute(self, x):
80         return x.flip(1)
81
82     # The flow transformation: forward pass...
83     def f(self, x):
84         z = x
85         for i in range(self.num_flows):
86             z = self.coupling(z, i, forward=True)
87             z = self.permute(z)
88
89         return z
90     # ... and the inverse pass.
91     def f_inv(self, z):
92         x = z
93         for i in reversed(range(self.num_flows)):
94             x = self.permute(x)
95             x = self.coupling(x, i, forward=False)
96
97         return x
98
99     # A new function: This is used for classification. First, we
    predict probabilities, and then pick the most probable value.
100    def classify(self, x):
101        z = self.f(x)
102        y_pred = self.classnet(z) #output: probabilities (i.e.,
    softmax)
103        return torch.argmax(y_pred, dim=1)
104
105    # An auxiliary function: We use it for calculating the
    classification loss, namely, the negative log-likelihood for
    p(y|x).
106    # NOTE: We first apply the invertible transformation f.
107    def class_loss(self, x, y):
108        z = self.f(x)
```

```
        y_pred = self.classnet(z) #output: probabilities (i.e., softmax)
        return self.nll(torch.log(y_pred), y)

    def sample(self, batchSize):
        # sample z:
        z = self.prior_sample(batchSize=batchSize, D=self.D)
        # x = f^-1(z)
        x = self.f_inv(z)
        return x.view(batchSize, 1, self.D)

    # The log-probability of the base distribution (a.k.a. prior).
    def log_prior(self, x):
        log_p = log_integer_probability(x, self.mean, self.logscale)
        return log_p.sum(1)

    # Sampling from the base distribution.
    def prior_sample(self, batchSize, D=2):
        # Sample from logistic
        y = torch.rand(batchSize, self.D)
        x = torch.exp(self.logscale) * torch.log(y / (1. - y)) + self.mean
        # And then round it to an integer.
        return torch.round(x)

    # The forward pass: Now, we use the hybrid model objective!
    def forward(self, x, y, reduction='avg'):
        z = self.f(x)
        y_pred = self.classnet(z) #output: probabilities (i.e., softmax)

        idf_loss = -self.log_prior(z)
        class_loss = self.nll(torch.log(y_pred), y) #remember to use logarithm on top of softmax!

        if reduction == 'sum':
            return (class_loss + self.alpha * idf_loss).sum()
        else:
            return (class_loss + self.alpha * idf_loss).mean()
```

**Listing 6.1** A HybridIDF class.

```
# The number of invertible transformations
num_flows = 2

# Here, we present only for the option 1 IDF.
nett = lambda:nn.Sequential(nn.Linear(D // 2, M), nn.LeakyReLU(),
                            nn.Linear(M, M), nn.LeakyReLU(),
                            nn.Linear(M, D // 2))
netts = [nett]

# And a three-layered classifier.
classnet = nn.Sequential(nn.Linear(D, M), nn.LeakyReLU(),
                            nn.Linear(M, M), nn.LeakyReLU(),
                            nn.Linear(M, K),
                            nn.Softmax(dim=1))

# Init HybridIDF
model = HybridIDF(netts, classnet, num_flows, D=D, alpha=alpha)
```

**Listing 6.2** Examples of neural networks.

And we are done; this is all we need to have! After running the code and training the HybridIDFs, we should obtain results similar to those in Fig. 6.4.

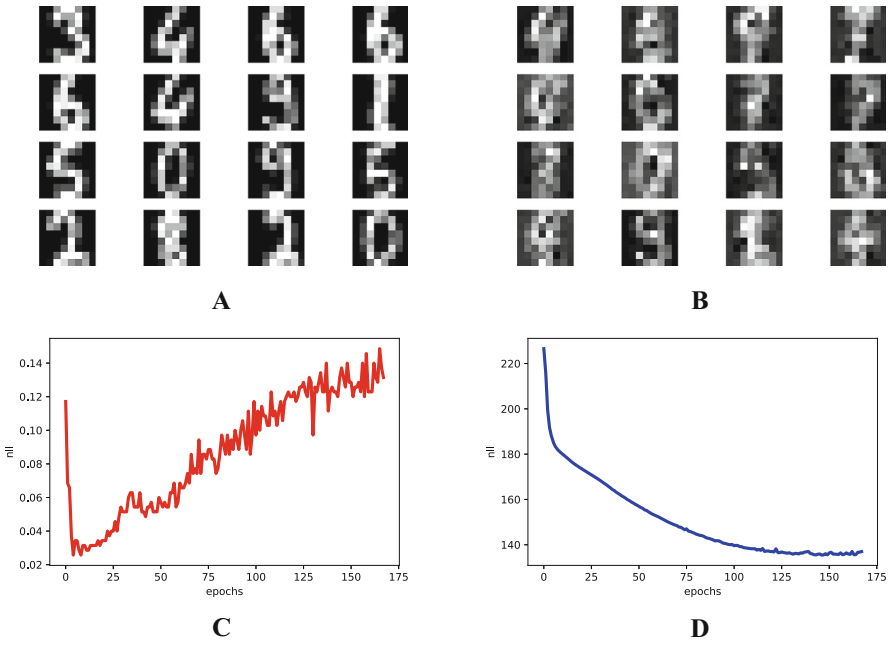



**Fig. 6.4** An example of outcomes after the training: (**a**) Randomly selected real images. (**b**) Unconditional generations from the HybridIDF. (**c**) An example of a validation curve for the classification error. (**d**) An example of a validation curve for the negative log-likelihood, i.e., $-\ln p(\mathbf{x})$.

## 6.5 What's Next?

**Hybrid VAE** The hybrid modeling idea goes beyond using flows for $p(\mathbf{x})$. Instead, e.g., we can pick VAE, and then, after applying the variational inference, we get a lower bound to the hybrid modeling objective:

$$\tilde{\ell}(\mathbf{x}, y; \lambda) = \ln p(y|\mathbf{x}) + \lambda \, \mathbb{E}_{\mathbf{z}\sim q(\mathbf{z}|\mathbf{x})} \left[\ln p(\mathbf{x}|\mathbf{z}) + \ln p(\mathbf{z}) - \ln q(\mathbf{z}|\mathbf{x})\right]. \tag{6.14}$$

where $p(y|\mathbf{x})$ uses the encoder inside, i.e., $q(\mathbf{z}|\mathbf{x})$.

**Semi-supervised hybrid learning** The hybrid modeling perspective is perfectly suited to the semi-supervised scenario. For the labeled data, we can use the objective $\ell(\mathbf{x}, y; \lambda) = \ln p(y|\mathbf{x}) + \lambda \ln p(\mathbf{x})$. However, for the unlabeled data, we can simply consider only the $\ln p(\mathbf{x})$ part. Such an approach was used by, for example, [10] for VAEs.

A very interesting perspective on learning semi-supervised VAE was presented in [11]. The authors end up with an objective that resembles the hybrid modeling objective but without the cumbersome $\lambda$!

**The factor** $\lambda$ As mentioned before, the fudge factor $\lambda$ could be troublesome. First, it does not follow a proper probability distribution. Second, it must be tuned which is always extra trouble. But, as mentioned before, [11] showed that we can get rid of $\lambda$!

**New parameterizations** An interesting open research direction is whether we can get rid of $\lambda$ by using a different learning algorithm and/or other parameterizations (e.g., some peculiar neural networks). It turns out that it is possible. One solution lies in a proper connection of a UNet parameterizing a diffusion model and attaching a predictive neural net to the encoder part of the UNet [12]. This paper clearly shows that it is perfectly fine to train a model with the joint likelihood function as the objective.

**Is this a good factorization?** I am almost sure that some of you wonder whether this factorization of the joint, i.e., $p(\mathbf{x}, y) = p(y|\mathbf{x})\, p(\mathbf{x})$, is indeed better than $p(\mathbf{x}, y) = p(\mathbf{x}|y)\, p(y)$. If I were to sample $\mathbf{x}$ from a specific class $y$, then the latter is better. However, if you go back to Chap. 1, you will notice that I do not care about *generating*. I prefer to have a good model that will assign proper probabilities to the world. That is why I prefer $p(\mathbf{x}, y) = p(y|\mathbf{x})\, p(\mathbf{x})$.

## References

1. Guillaume Bouchard and Bill Triggs. The tradeoff between generative and discriminative classifiers. In *16th IASC International Symposium on Computational Statistics (COMPSTAT'04)*, pages 721–728, 2004.
2. Eric Nalisnick, Akihiro Matsukawa, Yee Whye Teh, Dilan Gorur, and Balaji Lakshminarayanan. Hybrid models with deep and invertible features. In *International Conference on Machine Learning*, pages 4723–4732. PMLR, 2019.

3. Diederik P Kingma, Danilo J Rezende, Shakir Mohamed, and Max Welling. Semi-supervised learning with deep generative models. In *Proceedings of the 27th International Conference on Neural Information Processing Systems*, pages 3581–3589, 2014.
4. Sergey Tulyakov, Andrew Fitzgibbon, and Sebastian Nowozin. Hybrid VAE: Improving deep generative models using partial observations. *arXiv preprint arXiv:1711.11566*, 2017.
5. Diederik P Kingma and Prafulla Dhariwal. Glow: generative flow with invertible 1× 1 convolutions. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 10236–10245, 2018.
6. Ricky TQ Chen, Jens Behrmann, David Duvenaud, and Jörn-Henrik Jacobsen. Residual flows for invertible generative modeling. *arXiv preprint arXiv:1906.02735*, 2019.
7. Yura Perugachi-Diaz, Jakub M Tomczak, and Sandjai Bhulai. Invertible densenets with concatenated lipswish. *Advances in Neural Information Processing Systems*, 2021.
8. Rianne van den Berg, Alexey A Gritsenko, Mostafa Dehghani, Casper Kaae Sønderby, and Tim Salimans. Idf++: Analyzing and improving integer discrete flows for lossless compression. *arXiv e-prints*, pages arXiv–2006, 2020.
9. Emiel Hoogeboom, Jorn WT Peters, Rianne van den Berg, and Max Welling. Integer discrete flows and lossless compression. *arXiv preprint arXiv:1905.07376*, 2019.
10. Maximilian Ilse, Jakub M Tomczak, Christos Louizos, and Max Welling. DIVA: Domain invariant variational autoencoders. In *Medical Imaging with Deep Learning*, pages 322–348. PMLR, 2020.
11. Tom Joy, Sebastian M Schmon, Philip HS Torr, N Siddharth, and Tom Rainforth. Rethinking semi-supervised learning in VAEs. *arXiv preprint arXiv:2006.10102*, 2020.
12. Kamil Deja, Tomasz Trzciński, and Jakub M Tomczak. Learning data representations with joint diffusion models. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 543–559. Springer, 2023.

# Chapter 7
# Energy-Based Models


## 7.1 Introduction

So far, we have discussed various deep generative models for modeling the marginal distribution over observable variables (e.g., images), $p(\mathbf{x})$, such as autoregressive models (ARMs), flow-based models (flows, for short), variational autoencoders (VAEs), and hierarchical models like hierarchical VAEs and diffusion-based deep generative models (DDGMs). However, from the very beginning, we advocate for using deep generative modeling in the context of finding the joint distribution over observables and decision variables that is factorized as $p(\mathbf{x}, y) = p(y|\mathbf{x})p(\mathbf{x})$. After taking the logarithm of the joint, we obtain two additive components: $\ln p(\mathbf{x}, y) = \ln p(y|\mathbf{x}) + \ln p(\mathbf{x})$. We outlined how such a joint model could be formulated and trained in the hybrid modeling setting (see Chap. 6). The drawback of hybrid modeling though is the necessity of weighting both distributions, i.e., $\ell(\mathbf{x}, y\lambda) = \ln p(y|\mathbf{x}) + \lambda \ln p(\mathbf{x})$, and for $\lambda \neq 1$, this objective does not correspond to the log-likelihood of the joint distribution. The question is whether it is possible to formulate a model to learn with $\lambda = 1$. Here, we are going to discuss a potential solution to this problem using probabilistic **energy-based models** (EBMs) [1].

The history of EBMs is long and dates back to the 1980s of the previous century when models dubbed **Boltzmann machines** were proposed [2, 3]. Interestingly, the idea behind Boltzmann machines is taken from statistical physics and was formulated by cognitive scientists. A nice mix-up, isn't it? In a nutshell, instead of proposing a specific distribution like Gaussian or Bernoulli, we can define an **energy function**, $E(\mathbf{x})$, that assigns a value (*energy*) to a given state. There are no restrictions on the energy function, so you can already think of parameterizing it with neural networks. Then, the probability distribution could be obtained by transforming the energy to the unnormalized probability $e^{-E(\mathbf{x})}$ and normalizing it by $Z = \sum_{\mathbf{x}} e^{-E(\mathbf{x})}$ (a.k.a. a *partition function*), which yields the Boltzmann (also called Gibbs) distribution:

$$p(\mathbf{x}) = \frac{e^{-E(\mathbf{x})}}{Z}. \tag{7.1}$$



J. M. Tomczak, *Deep Generative Modeling*,
https://doi.org/10.1007/978-3-031-64087-2_7

If we consider continuous random variables, then the sum sign should be replaced by the integral. In physics, the energy is scaled by an inverse of temperature [4]; however, we skip it to keep the notation uncluttered. Understanding how the Boltzmann distribution works is relatively simple. Imagine a grid 5 by 5. Then, assign some value (energy) to each of the 25 points where a larger value means that a point has higher energy. Exponentiating the energy ensures that we do not obtain negative values. To calculate the probability for each point, we need to divide all exponentiated energies by their sum, in the same way how we do it for calculating softmax. In the case of continuous random variables, we must normalize by calculating an integral (i.e., a sum over all infinitesimal regions). For instance, the Gaussian distribution could be also expressed as the Boltzmann distribution with an analytically tractable partition function and the energy function of the following form:

$$E(x; \mu, \sigma^2) = \frac{1}{2\sigma^2}(x - \mu)^2, \tag{7.2}$$

which yields:

$$p(x) = \frac{e^{-E(x)}}{\int e^{-E(x)} \mathrm{d}x} \tag{7.3}$$

$$= \frac{e^{\frac{1}{2\sigma^2}(x-\mu)^2}}{\sqrt{2\pi\sigma^2}}. \tag{7.4}$$

In practice, most energy functions do not result in a nicely computable partition function. And, typically, the partition function is the key element that is problematic in learning energy-based models. The second problem is that, in general, it is hard to sample from such models. Why? Well, we know the probability for each point, but there is no generative process like in ARMs, flows, or VAEs. It is unclear how to start and what is the graphical model for an EBM. We can think of the EBM as a box that for a given $\mathbf{x}$ can tell us the (unnormalized) probability of that point. Notice that the energy function does not distinguish variables in any way; it does not care about any structure in $\mathbf{x}$. It says give me $\mathbf{x}$ and I will return the value. That's it! In other words, the energy function defines mountains and valleys over the space of random variables.

A curious reader (yes, I am referring to you!) may ask why we want to bother with EBMs. Previously discussed models are at least tractable and comprehensible in the sense that some stochastic dependencies are defined. Now, we suddenly invert the logic and say that we do not care about modeling the structure and, instead, we want to model an energy function that returns unnormalized probabilities. Is it beneficial? Yes, for at least three reasons: First, in principle, the energy function is unconstrained; it could be any function! Yes, you have probably guessed already; it could be a neural network! Second, notice that the energy function could be multimodal without being defined as such (i.e., opposing a mixture distribution). Third, there is no difference if we define it over discrete or continuous variables. I hope you see now that EBMs

have a lot of advantages! They possess a lot of deficiencies too, but, hey, let us stick to the positive aspects before we start, ok?

## 7.2 Model Formulation

As mentioned earlier, we formulate an energy function with some parameters $\theta$ over observable and decision random variables, $E(\mathbf{x}, y; \theta)$, that assigns a value (*energy*) to a pair $(\mathbf{x}, y)$ where $\mathbf{x} \in \mathbb{R}^D$ and $y \in \{0, 1, \ldots, K-1\}$. Let $E(\mathbf{x}, y; \theta)$ be parameterized by a neural network $NN_\theta(\mathbf{x})$ that returns $K$ values: $NN_\theta : \mathbb{R}^D \rightarrow \mathbb{R}^K$. In other words, we can define the energy as follows:

$$E(\mathbf{x}, y; \theta) = -NN_\theta(\mathbf{x})[y] \tag{7.5}$$

where we indicate by $[y]$ the specific output of the neural net $NN_\theta(\mathbf{x})$. Then, the joint probability distribution is defined as the Boltzmann distribution:

$$p_\theta(\mathbf{x}, y) = \frac{\exp\{NN_\theta(\mathbf{x})[y]\}}{\sum_{\mathbf{x},y} \exp\{NN_\theta(\mathbf{x})[y]\}} \tag{7.6}$$

$$= \frac{\exp\{NN_\theta(\mathbf{x})[y]\}}{Z_\theta}, \tag{7.7}$$

where we define the partition function as $Z_\theta = \sum_{\mathbf{x},y} \exp\{NN_\theta(\mathbf{x})[y]\}$.

Since we have the joint distribution, we can calculate the marginal distribution and the conditional distribution. First, let us take a look at the marginal $p(\mathbf{x})$. Applying the sum rule to the joint distribution yields:

$$p_\theta(\mathbf{x}) = \sum_y p_\theta(\mathbf{x}, y) \tag{7.8}$$

$$= \frac{\sum_y \exp\{NN_\theta(\mathbf{x})[y]\}}{\sum_{\mathbf{x},y} \exp\{NN_\theta(\mathbf{x})\}[y]} \tag{7.9}$$

$$= \frac{\sum_y \exp\{NN_\theta(\mathbf{x})[y]\}}{Z_\theta}. \tag{7.10}$$

Let us notice that we can express this distribution differently. First, we can rewrite the numerator in the following manner:

$$\sum_y \exp\{NN_\theta(\mathbf{x})[y]\} = \exp\left\{\log\left(\sum_y \exp\{NN_\theta(\mathbf{x})[y]\}\right)\right\} \tag{7.11}$$

$$= \exp\left\{\text{LogSumExp}_y\{NN_\theta(\mathbf{x})[y]\}\right\} \tag{7.12}$$

where we define $\text{LogSumExp}_y\{f(y)\} = \ln \sum_y \exp\{f(y)\}$. In other words, we can say that the energy function of the marginal distribution is expressed as

$-\text{LogSumExp}_y\{NN_\theta(\mathbf{x})[y]\}$. Then, the marginal distribution could be defined as follows:

$$p_\theta(\mathbf{x}) = \frac{\exp\left\{\text{LogSumExp}_y\{NN_\theta(\mathbf{x})[y]\}\right\}}{Z_\theta}. \tag{7.13}$$

Now, we can calculate the conditional distribution $p_\theta(y|\mathbf{x})$. We know that $p_\theta(\mathbf{x}, y) = p_\theta(y|\mathbf{x})\, p_\theta(\mathbf{x})$; thus:

$$p_\theta(y|\mathbf{x}) = \frac{p_\theta(\mathbf{x}, y)}{p_\theta(\mathbf{x})} \tag{7.14}$$

$$= \frac{\frac{\exp\{NN_\theta(\mathbf{x})[y]\}}{Z_\theta}}{\frac{\sum_y \exp\{NN_\theta(\mathbf{x})[y]\}}{Z_\theta}} \tag{7.15}$$

$$= \frac{\exp\{NN_\theta(\mathbf{x})[y]\}}{\sum_y \exp\{NN_\theta(\mathbf{x})[y]\}}. \tag{7.16}$$

The last line should resemble something; do you see it? Yes, you are right; it is the **softmax** function! We have shown that the energy-based model could be used either as a classifier or as a marginal distribution. And it is enough to define a single neural network for that! Isn't it beautiful? The same observations were made in [5] that any classifier could be seen as an energy-based model.

Interestingly, the logarithm of the joint distribution is the following:

$$\ln p_\theta(\mathbf{x}, y) = \ln p_\theta(y|\mathbf{x}) + \ln p_\theta(\mathbf{x}) \tag{7.17}$$

$$= \ln \frac{\exp\{NN_\theta(\mathbf{x})[y]\}}{\sum_y \exp\{NN_\theta(\mathbf{x})[y]\}} + \ln \frac{\sum_y \exp\{NN_\theta(\mathbf{x})[y]\}}{Z_\theta} \tag{7.18}$$

$$= \ln \text{softmax}\{NN_\theta(\mathbf{x})[y]\} + \left(\text{LogSumExp}_y\{NN_\theta(\mathbf{x})[y]\} - \ln Z_\theta\right), \tag{7.19}$$

where we define $\text{LogSumExp}_y\{f(y)\} = \ln \sum_y \exp\{f(y)\}$. We clearly see that the model requires a shared neural network that is used for calculating both distributions. To obtain a specific distribution, we pick the final activation function. The model is schematically presented in Fig. 7.1.

## 7.3 Training

We have a single neural network to train, and the training objective is the logarithm of the joint distribution. Since the training objective is a sum of the logarithm of the conditional $p_\theta(y|\mathbf{x})$ and the logarithm the marginal $p_\theta(\mathbf{x})$, calculating the gradient with respect to the parameters $\theta$ requires taking the gradient of each of the component

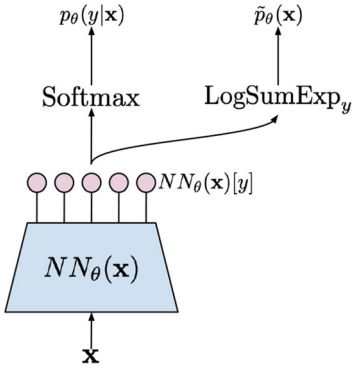



**Fig. 7.1** A schematic representation of an EBM. We denote the output of $\text{LogSumExp}_y$ by $\tilde{p}_\theta(\mathbf{x})$ to highlight that it is the unnormalized distribution since calculating the partition function is troublesome.

separately. We know that there is no problem with learning a classifier so let us take a closer look at the second component, namely:

$$\nabla_\theta \ln p_\theta(\mathbf{x}) = \nabla_\theta \text{LogSumExp}_y\{NN_\theta(\mathbf{x})[y]\} - \nabla_\theta \ln Z_\theta \tag{7.20}$$

$$= \nabla_\theta \text{LogSumExp}_y\{NN_\theta(\mathbf{x})[y]\} + - \nabla_\theta \ln \sum_{\mathbf{x}} \exp\left\{\text{LogSumExp}_y\{NN_\theta(\mathbf{x})[y]\}\right\} \tag{7.21}$$

$$= \nabla_\theta \text{LogSumExp}_y\{NN_\theta(\mathbf{x})[y]\} + - \sum_{\mathbf{x}'} \frac{\exp\left\{\text{LogSumExp}_y\{NN_\theta(\mathbf{x}')[y]\}\right\}}{\sum_{\mathbf{x}'',y''} \exp\{NN_\theta(\mathbf{x}'')[y'']\}} \nabla_\theta \text{LogSumExp}_y\{NN_\theta(\mathbf{x}')[y]\} \tag{7.22}$$

$$= \nabla_\theta \text{LogSumExp}_y\{NN_\theta(\mathbf{x})[y]\} + - \mathbb{E}_{\mathbf{x}'\sim p_\theta(\mathbf{x})}\left[\nabla_\theta \text{LogSumExp}_y\{NN_\theta(\mathbf{x}')[y]\}\right]. \tag{7.23}$$

We can decipher what has just happened here. The gradient of the first part, $\nabla_\theta \text{LogSumExp}_y\{NN_\theta(\mathbf{x})[y]\}$, is calculated for a given data point $\mathbf{x}$. The log-sum-exp function is differentiable, so we can apply autograd tools. However, the second part, $\mathbb{E}_{\mathbf{x}'\sim p_\theta(\mathbf{x})}\left[\nabla_\theta \text{LogSumExp}_y\{NN_\theta(\mathbf{x}')[y]\}\right]$, is a totally different story for two reasons:

- First, the gradient of the logarithm of the partition function turns into the expected value over $\mathbf{x}$ distributed according to the model! That is really a problem because the expected value cannot be analytically calculated and sampling from the marginal distribution $p_\theta(\mathbf{x})$ is nontrivial.

- Second, we need to calculate the expected value of the log-sum-exp of $NN_\theta(\mathbf{x})$. That is good news because we can do it using automatic differentiation tools.

Thus, the only problem lies in the expected value. Typically, it is approximated by Monte Carlo samples; however, it is not clear how to sample effectively and efficiently from an EBM. Grathwohl et al. [5] proposes to use the Langevin dynamics [6] that is an MCMC method. The Langevin dynamics in our case starts with a randomly initialized $\mathbf{x}_0$ and then uses the information about the landscape of the energy function (i.e., the gradient) to seek for new $\mathbf{x}$, that is:

$$\mathbf{x}_{t+1} = \mathbf{x}_t + \alpha \nabla_{\mathbf{x}_t} \text{LogSumExp}_y \{NN_\theta(\mathbf{x})[y]\} + \sigma \cdot \epsilon, \tag{7.24}$$

where $\alpha > 0$, $\sigma > 0$, and $\epsilon \sim \mathcal{N}(0, I)$. The Langevin dynamics could be seen as the stochastic gradient descent in the observable space with a small Gaussian noise added at each step. Once we apply this procedure for $\eta$ steps, we can approximate the gradient as follows:

$$\nabla_\theta \ln p_\theta(\mathbf{x}) \approx \nabla_\theta \text{LogSumExp}_y \{NN_\theta(\mathbf{x})[y]\} - \nabla_\theta \text{LogSumExp}_y \{NN_\theta(\mathbf{x}_\eta)[y]\}, \tag{7.25}$$

where $\mathbf{x}_\eta$ denotes the last step of the Langevin dynamics procedure.

We are ready to put it all together! Please remember that our training objective is the following:

$$\ln p_\theta(\mathbf{x}, y) = \ln \text{softmax}\{NN_\theta(\mathbf{x})[y]\} + \Big(\text{LogSumExp}_y \{NN_\theta(\mathbf{x})[y]\} - \ln Z_\theta\Big), \tag{7.26}$$

where the first part is for learning a classifier and the second part is for learning a generator (so to speak). As a result, we can say we have a sum of two objectives for a fully shared model. The gradient with respect to the parameters is the following:

$$\begin{aligned} \nabla_\theta \ln p_\theta(\mathbf{x}, y) =& \nabla_\theta \ln \text{softmax}\{NN_\theta(\mathbf{x})[y]\} + \\ &+ \nabla_\theta \text{LogSumExp}_y \{NN_\theta(\mathbf{x})[y]\} + \\ &- \mathbb{E}_{\mathbf{x}' \sim p_\theta(\mathbf{x})} \left[\nabla_\theta \text{LogSumExp}_y \{NN_\theta(\mathbf{x}')[y]\}\right]. \end{aligned} \tag{7.27}$$

The last two components come from calculating the gradient of the marginal distribution. Remember that the problematic part is only the last component! We will approximate this part using the Langevin dynamics (i.e., a sampling procedure) and a single sample. The final training procedure is the following:

**Training of EMBs**

1. Sample $\mathbf{x}_n$ and $y_n$ from a dataset.
2. Calculate $NN_\theta(\mathbf{x}_n)[y]$.
3. Initialize $\mathbf{x}_0$ using, for example, a uniform distribution.
4. Run the Langevin dynamics for $\eta$ steps:

$$\mathbf{x}_{t+1} = \mathbf{x}_t + \alpha \nabla_{\mathbf{x}_t} \text{LogSumExp}_y \{NN_\theta(\mathbf{x})[y]\} + \sigma \cdot \epsilon. \tag{7.28}$$

5. Calculate the objective:

$$L_{clf}(\theta) = \sum_y \mathbf{1}[y = y_n]\, \theta_y \ln \{NN_\theta(\mathbf{x}_n)[y]\} \tag{7.29}$$

$$L_{gen}(\theta) = \text{LogSumExp}_y \{NN_\theta(\mathbf{x})[y]\} - \text{LogSumExp}_y \{NN_\theta(\mathbf{x}_\eta)[y]\} \tag{7.30}$$

$$L(\theta) = L_{clf}(\theta) + L_{gen}(\theta). \tag{7.31}$$

6. Apply the autograd tool to calculate gradients $\nabla_\theta L(\theta)$, and update the neural network.

Notice that $L_{clf}(\theta)$ is nothing else than the cross-entropy loss and $L_{gen}(\theta)$ is a (crude) approximation to the log-marginal distribution over $\mathbf{x}$s.

## 7.4 Code

What do we need to code then? First, we must specify the neural network that defines the energy function (let us call it the *energy net*). Classifying using the energy net is rather straightforward. The main problematic part is a sampling from the model using the Langevin dynamics. Fortunately, the autograd tools allow us to easily access the gradient with respect to $\mathbf{x}$! In fact, it is a single line in the code below. Then we require writing a loop to run the Langevin dynamics for $\eta$ iterations with the steps size $\alpha$ and the noise level equal $\sigma$. In the code, we assume the data are normalized and scaled to $[-1, 1]$ similar to [5].

```
class EBM(nn.Module):
    def __init__(self, energy_net, alpha, sigma, ld_steps, D):
        super(EBM, self).__init__()

        print('EBM by JT.')

        # the neural net used by the EBM
        self.energy_net = energy_net

        # the loss for classification
        self.nll = nn.NLLLoss(reduction='none')  # it requires
    log-softmax as input!!
```

```
12
13          # hyperparams
14          self.D = D
15
16          self.sigma = sigma
17
18          self.alpha = torch.FloatTensor([alpha])
19
20          self.ld_steps = ld_steps
21
22      def classify(self, x):
23          f_xy = self.energy_net(x)
24          y_pred = torch.softmax(f_xy, 1)
25          return torch.argmax(y_pred, dim=1)
26
27      def class_loss(self, f_xy, y):
28          # - calculate logits (for classification)
29          y_pred = torch.softmax(f_xy, 1)
30
31          return self.nll(torch.log(y_pred), y)
32
33      def gen_loss(self, x, f_xy):
34          # - sample using Langevin dynamics
35          x_sample = self.sample(x=None, batch_size=x.shape[0])
36
37          # - calculate f(x_sample)[y]
38          f_x_sample_y = self.energy_net(x_sample)
39
40          return -(torch.logsumexp(f_xy, 1) - torch.logsumexp(
    f_x_sample_y, 1))
41
42      def forward(self, x, y, reduction='avg'):
43          # =====
44          # forward pass through the network
45          # - calculate f(x)[y]
46          f_xy = self.energy_net(x)
47
48          # =====
49          # discriminative part
50          # - calculate the discriminative loss: the cross-entropy
51          L_clf = self.class_loss(f_xy, y)
52
53          # =====
54          # generative part
55          # - calculate the generative loss: E(x) - E(x_sample)
56          L_gen = self.gen_loss(x, f_xy)
57
58          # =====
59          # Final objective
60          if reduction == 'sum':
61              loss = (L_clf + L_gen).sum()
62          else:
63              loss = (L_clf + L_gen).mean()
64
```

```
        return loss

    def energy_gradient(self, x):
        self.energy_net.eval()

        # copy original data that doesn't require grads!
        x_i = torch.FloatTensor(x.data)
        x_i.requires_grad = True  # WE MUST ADD IT, otherwise
    autograd won't work

        # calculate the gradient
        x_i_grad = torch.autograd.grad(torch.logsumexp(self.
    energy_net(x_i), 1).sum(), [x_i], retain_graph=True)[0]

        self.energy_net.train()

        return x_i_grad

    def langevin_dynamics_step(self, x_old, alpha):
        # Calculate gradient wrt x_old
        grad_energy = self.energy_gradient(x_old)
        # Sample eta ~ Normal(0, alpha)
        epsilon = torch.randn_like(grad_energy) * self.sigma

        # New sample
        x_new = x_old + alpha * grad_energy + epsilon

        return x_new

    def sample(self, batch_size=64, x=None):
        # - 1) Sample from uniform
        x_sample = 2. * torch.rand([batch_size, self.D]) - 1.

        # - 2) run Langevin Dynamics
        for i in range(self.ld_steps):
            x_sample = self.langevin_dynamics_step(x_sample,
    alpha=self.alpha)

        return x_sample
```

**Listing 7.1** An EBM class.

And we are done; this is all we need to have! After running the code and training the EBM, we should obtain results similar to those in Fig. 7.2.

## 7.5 Restricted Boltzmann Machines

The idea of defining a model through the energy function is the foundation of a broad family of Boltzmann machines (BMs) [2, 7]. The Boltzmann machines define an energy function as follows:

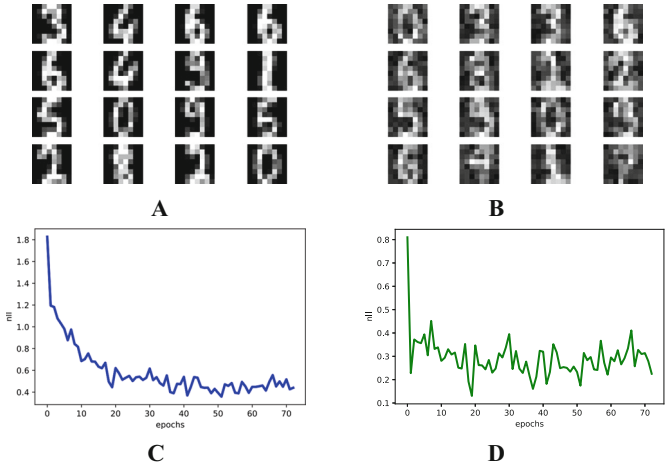



**Fig. 7.2** An example of outcomes after the training: (**a**) Randomly selected real images. (**b**) Unconditional generations from the EBM after applying $\eta = 20$ steps of the Langevin dynamics. (**c**) An example of a validation curve of the objective ($L_{clf} + L_{gen}$). (**d**) An example of a validation curve of the generative objective $L_{gen}$.

$$E(\mathbf{x}; \theta) = -\left(\mathbf{x}^\top \mathbf{W}\mathbf{x} + \mathbf{b}^\top \mathbf{x}\right), \tag{7.32}$$

where $\theta = \{\mathbf{W}, \mathbf{b}\}$ and $\mathbf{W}$ is the weight matrix and $\mathbf{b}$ is the bias vector (bias weights), which is the same as that in Hopfield networks and Ising models. The problem with BM is that they are hard to train (due to the partition function). However, we can alleviate the problem by introducing latent variables and restricting connections among observables.

### 7.5.1 Restricting BMs

Let us consider a BM that consists of binary observable variables $\mathbf{x} \in \{0, 1\}^D$ and binary latent (hidden) variables $\mathbf{z} \in \{0, 1\}^M$. The relationships among variables are specified through the following *energy function*:

$$E(\mathbf{x}, \mathbf{z}; \theta) = -\mathbf{x}^\top \mathbf{W}\mathbf{z} - \mathbf{b}^\top \mathbf{x} - \mathbf{c}^\top \mathbf{z}, \tag{7.33}$$

where $\theta = \{\mathbf{W}, \mathbf{b}, \mathbf{c}\}$ is a set of parameters and $\mathbf{W} \in \mathbb{R}^{D \times M}$, $\mathbf{b} \in \mathbb{R}^D$, and $\mathbf{c} \in \mathbb{R}^M$ are, respectively, weights, observable biases, and hidden biases. For the energy function in Eq. 7.33, RBM is defined by the *Gibbs distribution*:

$$p(\mathbf{x}, \mathbf{z}|\theta) = \frac{1}{Z_\theta} \exp\big(-E(\mathbf{x}, \mathbf{z}; \theta)\big), \tag{7.34}$$

where

$$Z_\theta = \sum_{\mathbf{x}} \sum_{\mathbf{z}} \exp\big(-E(\mathbf{x}, \mathbf{z}; \theta)\big) \tag{7.35}$$

is the *partition function*. The marginal probability over observables (the likelihood of observation) is:

$$p(\mathbf{x}|\theta) = \frac{1}{Z_\theta} \exp\big(-F(\mathbf{x}; \theta)\big), \tag{7.36}$$

where $F(\cdot)$ is the *free energy*:[1]

$$F(\mathbf{x}; \theta) = -\mathbf{b}^\top \mathbf{x} - \sum_j \log\Big(1 + \exp\big(\mathbf{c}_j + (\mathbf{W}_{\cdot j})^\top \mathbf{x}\big)\Big). \tag{7.37}$$

The presented model is called a restricted Boltzmann machine (RBM). It possesses the very useful property that the conditional distribution over the hidden variables factorizes given the observable variables and vice versa, which yields the following:

$$p(\mathbf{z}_m = 1|\mathbf{x}, \theta) = \text{sigm}\big(\mathbf{c}_m + (\mathbf{W}_{\cdot m})^\top \mathbf{x}\big), \tag{7.38}$$

$$p(\mathbf{x}_d = 1|\mathbf{z}, \theta) = \text{sigm}(\mathbf{b}_d + \mathbf{W}_{d\cdot} \mathbf{z}). \tag{7.39}$$

### 7.5.2 Learning RBMs

For given data $\mathcal{D} = \{\mathbf{x}_n\}_{n=1}^N$, we can train an RBM using the maximum likelihood approach that seeks the maximum of the log-likelihood function:

$$\ell(\theta) = \frac{1}{N} \sum_{\mathbf{x}_n \in \mathcal{D}} \log p(\mathbf{x}_n|\theta). \tag{7.40}$$

The gradient of the learning objective $\ell(\theta)$ with respect to $\theta$ takes the following form:

$$\nabla_\theta \ell(\theta) = -\frac{1}{N} \sum_{n=1}^N \Big(\nabla_\theta F(\mathbf{x}_n; \theta) - \sum_{\hat{\mathbf{x}}} p(\hat{\mathbf{x}}|\theta) \nabla_\theta F(\hat{\mathbf{x}}; \theta)\Big). \tag{7.41}$$

In general, the gradient in Eq. 7.41 cannot be computed analytically because the second term requires summing over all configurations of observables. One way to sidestep this issue is the standard stochastic approximation of replacing the expectation under $p(\mathbf{x}|\theta)$ by a sum over $S$ samples $\{\hat{\mathbf{x}}_1, \ldots, \hat{\mathbf{x}}_S\}$ drawn according to $p(\mathbf{x}|\theta)$ [8]:

$$\nabla_\theta \ell(\theta) \approx -\Big(\frac{1}{N} \sum_{n=1}^N \nabla_\theta F(\mathbf{x}_n; \theta) - \frac{1}{S} \sum_{s=1}^S \nabla_\theta F(\hat{\mathbf{x}}_s; \theta)\Big). \tag{7.42}$$

[1] We use the following notation: for given matrix $\mathbf{A}$, $\mathbf{A}_{ij}$ is its element at location $(i, j)$, $\mathbf{A}_{\cdot j}$ denotes its $j$th column , and $\mathbf{A}_{i\cdot}$ denotes its $i$th row, and for given vector $\mathbf{a}$, $\mathbf{a}_i$ is its $i$th element.

A different approach, *contrastive divergence*, approximates the expectation under $p(\mathbf{x}|\theta)$ in Eq. 7.41 by a sum over samples $\bar{\mathbf{x}}_n$ drawn from a distribution obtained by applying $K$ steps of the block Gibbs sampling procedure:

$$\nabla_\theta \ell(\theta) \approx -\frac{1}{N} \sum_{n=1}^{N} \Big( \nabla_\theta F(\mathbf{x}_n; \theta) - \nabla_\theta F(\bar{\mathbf{x}}_n; \theta) \Big). \tag{7.43}$$

**A Side Note**

The original CD [9] used $K$ steps of the Gibbs chain, starting from each data point $\mathbf{x}_n$ to obtain a sample $\bar{\mathbf{x}}_n$, and is restarted after every parameter update. An alternative approach, *Persistent Contrastive Divergence* (PCD) does not restart the chain after each update; this typically results in a slower convergence rate but eventually better performance [10].

### 7.5.3 Defining Higher-Order Relationships Through the Energy Function

The energy function is an interesting concept because it allows the modeling of higher-order dependencies among variables. For instance, the binary RBM could be extended to third-order multiplicative interactions by introducing two kinds of hidden variables, i.e., subspace units and gate units. The subspace units are hidden variables that reflect variations of a feature, and, thus, they are more robust to invariances. The gate units are responsible for activating the subspace units, and they can be seen as pooling features composed of the subspace features.

Let us consider the following random variables: $\mathbf{x} \in \{0,1\}^D, \mathbf{h} \in \{0,1\}^M, \mathbf{S} \in \{0,1\}^{M \times K}$. We are interested in the situation where there are three variables connected, namely, one observable $x_i$ and two types of hidden binary units, a gate unit $h_j$ and a subspace unit $s_{jk}$. Each gate unit is associated with a group of subspace hidden units. The energy function of a joint configuration is then defined as follows:[2]

$$E(\mathbf{x}, \mathbf{h}, \mathbf{S}; \boldsymbol{\theta}) = -\sum_{i=1}^{D} \sum_{j=1}^{M} \sum_{k=1}^{K} W_{ijk} x_i h_j s_{jk} - \sum_{i=1}^{D} b_i x_i - \sum_{j=1}^{M} c_j h_j - \sum_{j=1}^{M} h_j \sum_{k=1}^{K} D_{jk} s_{jk}, \tag{7.44}$$

where the parameters are $\boldsymbol{\theta} = \{\mathbf{W}, \mathbf{b}, \mathbf{c}, \mathbf{D}\}$, where $\mathbf{W} \in \mathbb{R}^{D \times M \times K}$, $\mathbf{b} \in \mathbb{R}^D$, $\mathbf{c} \in \mathbb{R}^M$, and $\mathbf{D} \in \mathbb{R}^{M \times K}$.

The Gibbs distribution with the energy function in (7.44) is called *subspace restricted Boltzmann machine* (subspaceRBM) [11]. For the subspaceRBM, the following conditional dependencies hold true:[3]

[2] Unlike in other cases, we use sums instead of matrix products because now we have third-order multiplications that would complicate the notation.

[3] $\text{softplus}(a) = \log\big(1 + \exp(a)\big)$

$$p(x_i = 1|\mathbf{h}, \mathbf{S}) = \text{sigm}\Big(\sum_j \sum_k W_{ijk} h_j s_{jk} + b_i\Big), \tag{7.45}$$

$$p(s_{jk} = 1|\mathbf{x}, h_j) = \text{sigm}\Big(\sum_i W_{ijk} x_i h_j + h_j D_{jk}\Big), \tag{7.46}$$

$$p(h_j = 1|\mathbf{x}) = \text{sigm}\Big(-K\log 2 + c_j + \sum_{k=1}^{K} \text{softplus}\Big(\sum_i W_{ijk} x_i + D_{jk}\Big)\Big), \tag{7.47}$$

which can be straightforwardly used in formulating a contrastive divergence-like learning algorithm. Notice that in (7.47) a term $-K\log 2$ imposes a natural penalty of the hidden unit activation which is linear to the number of subspace hidden variables. Therefore, the gate unit is inactive unless the sum of softplus of total input exceeds the penalty term and the bias term.

The example of the subspaceRBM shows that the energy function is handy and allows the modeling of various stochastic relations. The subspaceRBM was used to model invariance features, but other modifications of the energy function in RBMs could be formulated to allow training spatial transformations [12] or spike-and-slab features [13].

## 7.6 Final Remarks

The paper of [5] is definitely a milestone in the EBM literature because it shows that we can use *any* energy function parameterized by a neural network. However, to get to that point, there was a lot of work on the energy-based models.

**Restricted Boltzmann Machines** RBMs possess a couple of useful traits: First, the bipartite structure helps training that could be further used in formulating an efficient training procedure for RBMs called contrastive divergence [9] that takes advantage of block Gibbs sampling. As mentioned earlier, a chain is initialized either at a random point or a sample of latents, and then, conditionally, the other set of variables are trained. Similar to the ping-pong game, we sample some variables given the others until convergence or until we decide to stop. Second, the distribution over latent variables could be calculated analytically. Moreover, it could be seen as being parameterized by logistic regression. That is an interesting fact that the sigmoid function arises naturally from the definition of the energy function! Third, the restrictions among connections show that we can still build powerful models that are (partially) analytically tractable. This opened a new research direction that aimed to formulate models with more sophisticated structures like spike-and-slab RBMs [13] and higher-order RBMs [11, 12] or RBMs for categorical observables [14] or real-valued observables [15]. Moreover, RBMs could be also modified to handle temporal data [16] that could be applied to, for example, human motion tracking [17]. The precursor of the idea presented in [5] was the work on classification RBMs [18, 19]. The training of RBMs is based on the MCMC techniques, for example, the contrastive divergence algorithm. However, RBMs could be trained to achieve

specific features by regularization [20] or other learning algorithms like the perturb-and-MAP method [21, 22], minimum probability flow [23], or other algorithms [8, 24].

**Deep Boltzmann Machines** A natural extension of BMs is a class of models with a deep architecture or hierarchical BMs. As indicated by many, the idea of hierarchical models plays a crucial role in AI [25]; therefore, there are many extensions of BMs with hierarchical (deep) architectures [15, 26, 27].

Training of deep BMs is even more challenging due to the complexity of the partition function [28]. One of the main approaches to the training of deep BMs relies on treating each pair of consecutive layers as an RBM and training them layer by layer where the layer at the lower layer is treated as observed [27]. This procedure was successfully applied in the seminal paper on unsupervised pre-training of neural nets [29].

**Approximating the Partition Function** The crucial quantity in the EBMs is the partition function because it allows calculating the Boltzmann distribution. Unfortunately, summing over all values of random variables is computationally infeasible. However, we can use one of the available approximation techniques:

- *Variational methods*: There are a few variational methods that lower-bound the log-partition function using the Bethe approximation [30, 31] or upper-bound the log-partition function using a tree-reweighted sum-product algorithm [32]. *Perturb-and-MAP methods*: An alternative approach is to relate the partition function to the max-statistics of random variables and apply the perturb-and-MAP method [33].
- *Stochastic approximations*: A different approach, probably the most straightforward, is to utilize a sampling procedure. One widely used technique is the annealed importance sampling [28].

Some of the approximations are useful for specific BMs, e.g., BMs with binary variables and BMs with a specific structure. In general, however, approximating the partition function remains an open question and is the main roadblock to using EMBs in practice and on a large scale.

## 7.7 Are EBMs the Future?

There is definitely a lot of potential in EBMs for at least two reasons:

1. They do not require using any fudge factor to balance the classification loss and the generative loss like in the hybrid modeling approach.
2. The results obtained by [5] clearly indicate that EBMs can achieve the SOTA classification error, synthesize images of high fidelity, and be of great use for the out-of-distribution selection.

However, there is one main problem that has not been yet solved: The calculation of $p(\mathbf{x})$. As I claim all the time, the deep generative modeling paradigm is useful not

only because we can synthesize nice-looking images but also because a Generative AI system can assess the uncertainty of the surrounding environment and share this information with us or other AI systems. Since calculating the marginal distribution in EBMs is troublesome, it is doubtful we can use these models in many applications. However, it is an extremely interesting research direction, and figuring out how to efficiently calculate the partition function and how to efficiently sample from the model is crucial for training powerful EBMs.

## References


1. Yann LeCun, Sumit Chopra, Raia Hadsell, M Ranzato, and F Huang. A tutorial on energy-based learning. *Predicting structured data*, 1(0), 2006.
2. David H Ackley, Geoffrey E Hinton, and Terrence J Sejnowski. A learning algorithm for Boltzmann machines. *Cognitive science*, 9(1):147–169, 1985.
3. Paul Smolensky. Information processing in dynamical systems: Foundations of harmony theory. Technical report, Colorado Univ at Boulder Dept of Computer Science, 1986.
4. Edwin T Jaynes. *Probability theory: The logic of science*. Cambridge university press, 2003.
5. Will Grathwohl, Kuan-Chieh Wang, Joern-Henrik Jacobsen, David Duvenaud, Mohammad Norouzi, and Kevin Swersky. Your classifier is secretly an energy-based model and you should treat it like one. In *International Conference on Learning Representations*, 2019.
6. Max Welling and Yee W Teh. Bayesian learning via stochastic gradient Langevin dynamics. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 681–688. Citeseer, 2011.
7. Geoffrey E Hinton, Terrence J Sejnowski, et al. Learning and relearning in Boltzmann machines. *Parallel distributed processing: Explorations in the microstructure of cognition*, 1(282–317):2, 1986.
8. Benjamin Marlin, Kevin Swersky, Bo Chen, and Nando Freitas. Inductive principles for restricted Boltzmann machine learning. In *Proceedings of the thirteenth International Conference on Artificial Intelligence and Statistics*, pages 509–516, 2010.
9. Geoffrey E Hinton. Training products of experts by minimizing contrastive divergence. *Neural computation*, 14(8):1771–1800, 2002.
10. Tijmen Tieleman. Training restricted Boltzmann machines using approximations to the likelihood gradient. In *ICML*, pages 1064–1071, 2008.
11. Jakub M Tomczak and Adam Gonczarek. Learning invariant features using subspace restricted Boltzmann machine. *Neural Processing Letters*, 45(1):173–182, 2017.
12. Roland Memisevic and Geoffrey E Hinton. Learning to represent spatial transformations with factored higher-order Boltzmann machines. *Neural computation*, 22(6):1473–1492, 2010.

13. Aaron Courville, James Bergstra, and Yoshua Bengio. A spike and slab restricted Boltzmann machine. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 233–241. JMLR Workshop and Conference Proceedings, 2011.
14. Ruslan Salakhutdinov, Andriy Mnih, and Geoffrey Hinton. Restricted Boltzmann machines for collaborative filtering. In *Proceedings of the 24th international conference on Machine learning*, pages 791–798, 2007.
15. Kyung Hyun Cho, Tapani Raiko, and Alexander Ilin. Gaussian-Bernoulli deep Boltzmann machine. In *The 2013 International Joint Conference on Neural Networks (IJCNN)*, pages 1–7. IEEE, 2013.
16. Ilya Sutskever, Geoffrey E Hinton, and Graham W Taylor. The recurrent temporal restricted Boltzmann machine. In *Advances in Neural Information Processing Systems*, pages 1601–1608, 2009.
17. Graham W Taylor, Leonid Sigal, David J Fleet, and Geoffrey E Hinton. Dynamical binary latent variable models for 3d human pose tracking. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 631–638. IEEE, 2010.
18. Hugo Larochelle and Yoshua Bengio. Classification using discriminative restricted Boltzmann machines. In *Proceedings of the 25th international conference on Machine learning*, pages 536–543, 2008.
19. Hugo Larochelle, Michael Mandel, Razvan Pascanu, and Yoshua Bengio. Learning algorithms for the classification restricted Boltzmann machine. *The Journal of Machine Learning Research*, 13(1):643–669, 2012.
20. Jakub M Tomczak. Learning informative features from restricted Boltzmann machines. *Neural Processing Letters*, 44(3):735–750, 2016.
21. Jakub M Tomczak. On some properties of the low-dimensional Gumbel perturbations in the Perturb-and-MAP model. *Statistics & Probability Letters*, 115:8–15, 2016.
22. Jakub M Tomczak, Szymon Zaręba, Siamak Ravanbakhsh, and Russell Greiner. Low-dimensional perturb-and-map approach for learning restricted Boltzmann machines. *Neural Processing Letters*, 50(2):1401–1419, 2019.
23. Jascha Sohl-Dickstein, Peter B Battaglino, and Michael R DeWeese. New method for parameter estimation in probabilistic models: Minimum Probability Flow. *Physical review letters*, 107(22):220601, 2011.
24. Yang Song and Diederik P Kingma. How to train your energy-based models. *arXiv preprint arXiv:2101.03288*, 2021.
25. Yoshua Bengio. *Learning deep architectures for AI*. Now Publishers Inc, 2009.
26. Honglak Lee, Roger Grosse, Rajesh Ranganath, and Andrew Y Ng. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *Proceedings of the 26th annual international conference on machine learning*, pages 609–616, 2009.
27. Ruslan Salakhutdinov. Learning deep generative models. *Annual Review of Statistics and Its Application*, 2:361–385, 2015.

28. Ruslan Salakhutdinov and Iain Murray. On the quantitative analysis of deep belief networks. In *Proceedings of the 25th international conference on Machine learning*, pages 872–879, 2008.
29. Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.
30. Max Welling and Yee Whye Teh. Approximate inference in Boltzmann machines. *Artificial Intelligence*, 143(1):19–50, 2003.
31. Jonathan S Yedidia, William T Freeman, and Yair Weiss. Constructing free-energy approximations and generalized belief propagation algorithms. *IEEE Transactions on information theory*, 51(7):2282–2312, 2005.
32. Martin J Wainwright, Tommi S Jaakkola, and Alan S Willsky. A new class of upper bounds on the log partition function. *IEEE Transactions on Information Theory*, 51(7):2313–2335, 2005.
33. Tamir Hazan and Tommi Jaakkola. On the partition function and random maximum a-posteriori perturbations. In *Proceedings of the 29th International Conference on International Conference on Machine Learning*, pages 1667–1674, 2012.

# Chapter 8
# Generative Adversarial Networks


## 8.1 Introduction

Once we discussed latent variable models, we claimed that they naturally define a generative process by first sampling latents $\mathbf{z} \sim p(\mathbf{z})$ and then generating observables $\mathbf{x} \sim p_\theta(\mathbf{x}|\mathbf{z})$. That is nice! However, the problem appears when we start thinking about training. To be more precise, the training objective is an issue. Why? Well, the probability theory tells us to *get rid of* all unobserved random variables by marginalizing them out. In the case of latent variable models, this is equivalent to calculating the (marginal) log-likelihood function in the following form:

$$\log p_\theta(\mathbf{x}) = \log \int p_\theta(\mathbf{x}|\mathbf{z})\, p(\mathbf{z})\, \mathrm{d}\mathbf{z}. \tag{8.1}$$

As we mentioned already in the section about VAEs (see Sect. 5.3), the problematic part is calculating the integral because it is not analytically tractable unless all distributions are Gaussian and the dependency between $\mathbf{x}$ and $\mathbf{z}$ is linear. However, let us forget for a moment about all these issues and take a look at what we can do here. First, we can approximate the integral using Monte Carlo samples from the prior $p(\mathbf{z})$ that yields:

$$\log p_\theta(\mathbf{x}) = \log \int p_\theta(\mathbf{x}|\mathbf{z})\, p(\mathbf{z})\, \mathrm{d}\mathbf{z} \tag{8.2}$$

$$\approx \log \frac{1}{S} \sum_{s=1}^{S} p_\theta(\mathbf{x}|\mathbf{z}_s) \tag{8.3}$$

$$= \log \sum_{s=1}^{S} \exp\left(\log p_\theta(\mathbf{x}|\mathbf{z}_s)\right) - \log S \tag{8.4}$$

$$= \mathrm{LogSumExp}_s\left\{p_\theta(\mathbf{x}|\mathbf{z}_s)\right\} - \log S, \tag{8.5}$$

where $\mathrm{LogSumExp}_s\left\{f(s)\right\} = \log \sum_{s=1}^{S} \exp\left(f(s)\right)$ is the log-sum-exp function.



J. M. Tomczak, *Deep Generative Modeling*,
https://doi.org/10.1007/978-3-031-64087-2_8

Assuming for a second that this is a good (i.e., a tight) approximation, we turn the problem of calculating the integral into a problem of sampling from the prior. For simplicity, we can assume a prior that is relatively easy to be sampled from, e.g., the standard Gaussian, $p(\mathbf{z}) = \mathcal{N}(\mathbf{z}|0, \mathbf{I})$. In other words, we need to model $p_\theta(\mathbf{x}|\mathbf{z})$ only, i.e., pick a parameterization for it. Guess what, we will use a neural network again! If we model images, then we can use the categorical distribution for the conditional likelihood, and then a neural network parameterizes the probabilities. Or if we use a Gaussian distribution like in the case of energy-based models or diffusion-based deep generative models, then $p_\theta(\mathbf{x}|\mathbf{z})$ could be Gaussian as well, and the neural network outputs the variance and/or the mean. Since the log-sum-exp function is differentiable (and the application of the log-sum-exp trick makes it even numerically stable), there is no problem learning this model end to end! This approach is a precursor of many deep generative models and was dubbed *density networks* [1]; see Fig. 8.1 for a schematic representation of density networks.

Density networks are important and, unfortunately, underappreciated deep generative models. It is worth knowing them at least for three reasons: First, understanding how they work helps a lot in comprehending other latent variable models and how to improve them. Second, they serve as a great starting point for understanding the difference between *prescribed models* and *implicit models*. Third, they allow us to formulate a nonlinear latent variable model and train it using backpropagation (or a gradient descent, in general).

Alright, so now you may have some questions because we made a few assumptions on the way that might have been pretty confusing. The main assumptions made here are the following:

- We need to specify the prior distribution $p(\mathbf{z})$, e.g., the standard Gaussian.
- We need to specify the form of the conditional likelihood $p(\mathbf{x}|\mathbf{z})$. Typically, people use the Gaussian distribution or a mixture of Gaussians. Hence, density nets are the *prescribed* models because we need to analytically formulate all distributions in advance.

As a result, we get the following:

- The objective function is the (approximated) log-likelihood function.
- We can optimize the objective using gradient-based optimization methods and the autograd tools.
- We can parameterize the conditional likelihood using deep neural networks.

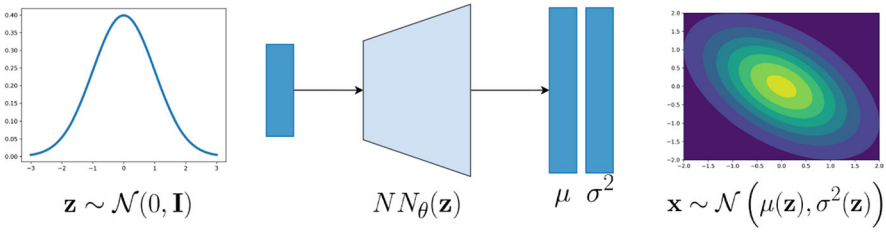



**Fig. 8.1** A schematic representation of a density net.

However, we pay a great price for all the goodies coming from the formulation of the density networks:

- There is no analytical solution (except the case equivalent to the probabilistic PCA).
- We get an approximation of the log-likelihood function.
- We need a lot of samples from the prior to get a reliable approximation of the log-likelihood function.
- It suffers from the curse of dimensionality.

As you can see, the issue with dimensionality is especially limiting. What can we do with a model if it cannot be efficient for higher-dimensional problems? All interesting applications like the image or audio analysis/synthesis are gone! So what can we do then? One possible direction is to stick to the prescribed models and apply variational inference (see Sect. 5.3). However, the other direction is to abandon the likelihood-based approach. I know it sounds ridiculous, but it is possible and, *unfortunately*, works pretty well in practice.

## 8.2 Implicit Modeling with Generative Adversarial Networks (GANs)

### 8.2.1 Getting Rid of Kullback-Leibler

Let us think again about what density networks tell us. First of all, they define a nice generative process: first, sample latents and then generate observables. Clear! Then, for training, they use the (marginal) log-likelihood function. In other words, the log-likelihood function assesses the difference between a training datum and a generated object. To be even more precise, we first pick the specific probability distribution for the conditional likelihood $p_\theta(\mathbf{x}|\mathbf{z})$ that defines how to calculate the difference between the training point and the generated observables.

One may ask here whether there is a different fashion of calculating the *difference* between real data and generated objects. If we recall our considerations about hierarchical VAEs (see Sect. 5.5.2), learning of the likelihood-based models is equivalent to optimizing the Kullback-Leibler (KL) divergence between the empirical distribution and the model, $KL\left[p_{data}(\mathbf{x})||p_\theta(\mathbf{x})\right]$. The KL-based approach requires a well-behaved distribution because of the logarithms. Moreover, we can think of it as a *local* way of comparing the empirical distribution (i.e., given data) and the generated data (i.e., data generated by our prescribed model). By *local*, we mean considering one point at a time and then summing all individual errors instead of comparing samples (i.e., collections of individuals) that we can refer to as a *global* comparison. However, we do not need to stick to the KL divergence! Instead, we can use other metrics that look at a set of points (i.e., distributions represented by a set of points) like integral probability metrics [2] (e.g., the maximum mean discrepancy (MMD) [3]) or use other divergences [4].

Still, all of the mentioned metrics rely on defining explicitly how we measure the error. The question is whether we can parameterize our loss function and learn it alongside our model. Since we talk all the time about neural networks, can we go even further and utilize a neural network to calculate differences?

### 8.2.2 Getting Rid of Prescribed Distributions

Alright, we agreed on the fact that the KL divergence is only one of many possible loss functions. Moreover, we asked ourselves whether we could use a learnable loss function. However, there is also one question floating in the air, namely, do we need to use the prescribed models in the first place? The reason is the following: Since we know that density networks take noise and turn it into distribution in the observable space, do we really need to output a full distribution? What if we return a single point? In other words, what if we define the conditional likelihood as Dirac's delta:

$$p_\theta(\mathbf{x}|\mathbf{z}) = \delta\left(\mathbf{x} - NN_\theta(\mathbf{z})\right). \tag{8.6}$$

This is equivalent to saying that instead of a Gaussian (i.e., a mean and a variance), $NN_\theta(\mathbf{z})$ outputs the mean only. Interestingly, if we consider the marginal distribution over $\mathbf{x}$s, we get a nicely behaved distribution. To see that, let us first calculate the marginal distribution:

$$p_\theta(\mathbf{x}) = \int \delta\left(\mathbf{x} - NN_\theta(\mathbf{z})\right)\ p(\mathbf{z})\ \mathrm{d}\mathbf{z}. \tag{8.7}$$

Then, let us understand what is going on! The marginal distribution is an infinite mixture of delta peaks. In other words, we take a single $\mathbf{z}$ and plot a peak (or a point in 2D, as it is easier to imagine) in the observable space. We proceed to infinity, and once we do that, the observable space will be covered by more and more points, and some regions will be *denser* than the others. This kind of modeling of a distribution is also known as *implicit modeling*.

So where is the problem then? Well, the problem in the prescribed modeling setting is that the term $\log \delta\left(\mathbf{x} - NN_\theta(\mathbf{z})\right)$ is ill-defined and cannot be used in many probability measures, including the KL-term because we cannot calculate the loss function. Therefore, we can ask ourselves whether we can define our own loss function, perhaps? And, even more, parameterize it with neural networks! You must admit it sounds appealing! So how to accomplish that?

### 8.2.3 Adversarial Loss

Let us start with the following story: There is a con artist (a fraud) and a friend of the fraud (an expert) who knows little about art. Moreover, there is a real artist who

has passed away (e.g., Pablo Picasso). The fraud tries to mimic the style of Pablo Picasso as well as possible. The friend expert browses for paintings of Picasso and compares them to the paintings provided by the fraud. Hence, the fraud tries to fool his friend, while the expert tries to distinguish real paintings of Picasso from fakes. Over time, the fraud becomes better and better, and the expert also learns how to decide whether a given painting is a fake. Eventually and unfortunately to the world of art, the work of the fraud may become indistinguishable from Picasso, and the expert may be completely uncertain about the paintings and whether they are fakes.

Now, let us formalize this wicked game. We call the expert a *discriminator* that takes an object $\mathbf{x}$ and returns a probability whether it is *real* (i.e., coming from the empirical distribution), $D_\alpha : \mathcal{X} \to [0, 1]$. We refer to the fraud as a *generator* that takes noise and turns it into an object $\mathbf{x}$, $G_\beta : \mathcal{Z} \to \mathcal{X}$. All $\mathbf{x}$s coming from the empirical distribution $p_{data}(\mathbf{x})$ are called *real*, and all $\mathbf{x}$s generated by $G_\beta(\mathbf{z})$ are dubbed *fake*. Then, we construct the objective function as follows:

- We have two sources of data: $\mathbf{x} \sim p_\theta(\mathbf{x}) = \int G_\beta(\mathbf{z})\, p(\mathbf{z})\, \mathrm{d}\mathbf{z}$ and $\mathbf{x} \sim p_{data}(\mathbf{x})$.
- The discriminator solves the classification task by assigning 0 to all fake datapoints and 1 to all real datapoints.
- Since the discriminator can be seen as a classifier, we can use the binary cross-entropy loss function in the following form:

$$\ell(\alpha, \beta) = \mathbb{E}_{\mathbf{x} \sim p_{real}} \left[\log D_\alpha(\mathbf{x})\right] + \mathbb{E}_{\mathbf{z} \sim p(\mathbf{z})} \left[\log \left(1 - D_\alpha \left(G_\beta(\mathbf{z})\right)\right)\right]. \tag{8.8}$$

  The left part corresponds to the real data source, and the right part contains the fake data source.
- We try to maximize $\ell(\alpha, \beta)$ with respect to $\alpha$ (i.e., the discriminator). In plain words, we want the discriminator to be as good as possible.
- The generator tries to fool the discriminator; thus, it tries to minimize $\ell(\alpha, \beta)$ with respect to $\beta$ (i.e., the generator).

  Eventually, we face the following learning objective:

$$\min_\beta \max_\alpha \mathbb{E}_{\mathbf{x} \sim p_{real}} \left[\log D_\alpha(\mathbf{x})\right] + \mathbb{E}_{\mathbf{z} \sim p(\mathbf{z})} \left[\log \left(1 - D_\alpha \left(G_\beta(\mathbf{z})\right)\right)\right]. \tag{8.9}$$

We refer to $\ell(\alpha, \beta)$ as the *adversarial loss* since there are two actors trying to achieve two opposite goals.

### 8.2.4 GANs

Let us put everything together:

- We have a generator that turns noise into fake data.
- We have a discriminator that classifies given input as either fake or real.
- We parameterize the generator and the discriminator using deep neural networks.

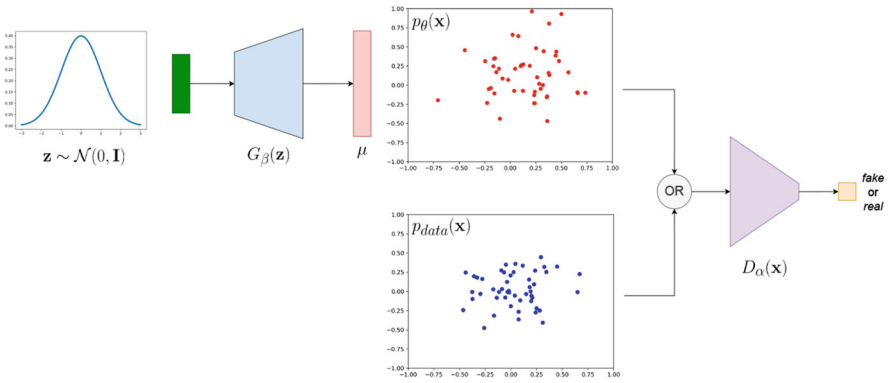



**Fig. 8.2** A schematic representation of GANs. Please note the part of the generator and its resemblance to density networks.

- We learn the neural networks using the adversarial loss (i.e., we optimize the min-max problem).

The resulting class of models is called Generative Adversarial Networks (GANs) [5]. In Fig. 8.2, we present the idea of GANs and how they are connected to density networks. Notice that the generator part constitutes an implicit distribution, i.e., a distribution from an unknown family of distributions, and its analytical form is unknown as well; however, we can sample from it.

## 8.3 Implementing GANs

Believe it or not, we have all the components to implement GANs. Let us look into all of them step by step. In fact, the easiest way to understand them is to implement them.

### 8.3.1 Generator

The first part is the *generator*, $G_\beta(\mathbf{z})$, which is simply a deep neural network. The code for a class of the generator is presented below. Notice that we distinguish between a function for generating, namely, transforming $\mathbf{z}$ to $\mathbf{x}$, and sampling that first samples $\mathbf{z} \sim \mathcal{N}(0, \mathbf{I})$ and then calls `generate`:

```
class Generator(nn.Module):
    def __init__(self, generator_net, z_size):
        super(Generator, self).__init__()

        # We need to init the generator neural net.
        self.generator_net = generator_net
        # We also need to know the size of the latents.
        self.z_size = z_size

    def generate(self, z):
        # Generating for given z is equivalent to applying the
    neural net.
        return self.generator_net(z)

    def sample(self, batch_size=16):
        # For sampling, we need to sample first latents.
        z = torch.randn(batch_size, self.z_size)
        return self.generate(z)

    def forward(self, z=None):
        if z is None:
            return self.sample()
        else:
            return self.generate(z)
```

**Listing 8.1** A generator class.

### 8.3.2 Discriminator

The second component is the *discriminator*. Here, the code is even simpler because it consists of a single neural network. The code for a class of the discriminator is provided below:

```
class Discriminator(nn.Module):
    def __init__(self, discriminator_net):
        super(Discriminator, self).__init__()
        # We need to init the discriminator neural net.
        self.discriminator_net = discriminator_net

    def forward(self, x):
        # The forward pass is just about applying the neural net.
        return self.discriminator_net(x)
```

**Listing 8.2** A discriminator class.

### 8.3.3 GAN

Now, we are ready to combine these two components. In our implementation, a GAN outputs the adversarial loss either for the generator or for the discriminator. Maybe

the code below is overkill; however, it is better to write a few more lines and properly understand what is going on than apply some unclear tricks:

```
class GAN(nn.Module):
    def __init__(self, generator, discriminator, EPS=1.e-5):
        super(GAN, self).__init__()

        print('GAN by JT.')

        # To put everything together, we need the generator and
        # the discriminator. NOTE: Both are instances of classes!
        self.generator = generator
        self.discriminator = discriminator

        # For numerical issues, we introduce a small epsilon.
        self.EPS = EPS

    def forward(self, x_real, reduction='avg', mode='
    discriminator'):
        # The forward pass calculates the adversarial loss.
        # More specifically, either its part for the generator or
        #  the part for the discriminator.
        if mode == 'generator':
            # For the generator, we first sample FAKE data.
            x_fake_gen = self.generator.sample(x_real.shape[0])

            # Then, we calculate the outputs of the discriminator
     for the FAKE data.
            # NOTE: We clamp here for the numerical stability
    later on.
            d_fake = torch.clamp(self.discriminator(x_fake_gen),
    self.EPS, 1. - self.EPS)

            # The loss for the generator is log(1 - D(G(z))).
            loss = torch.log(1. - d_fake)

        elif mode == 'discriminator':
            # For the discriminator, we first sample FAKE data.
            x_fake_gen = self.generator.sample(x_real.shape[0])

            # Then, we calculate the outputs of the discriminator
     for the FAKE data.
            # NOTE: We clamp for the numerical stability later on
    .
            d_fake = torch.clamp(self.discriminator(x_fake_gen),
    self.EPS, 1. - self.EPS)

            # Moreover, we calculate the outputs of the
    discriminator for the REAL data.
            # NOTE: We clamp for... the numerical stability (
    again).
            d_real = torch.clamp(self.discriminator(x_real), self
    .EPS, 1. - self.EPS)

```

```
            # The final loss for the discriminator is log(1 - D(G(z))) + log D(x).
            # NOTE: We take the minus sign because we MAXIMIZE the adversarial loss wrt
            # discriminator, so we MINIMIZE the negative adversarial loss wrt discriminator.
            loss = -(torch.log(d_real) + torch.log(1. - d_fake))

        if reduction == 'sum':
            return loss.sum()
        else:
            return loss.mean()

    def sample(self, batch_size=64):
        return self.generator.sample(batch_size=batch_size)
```

**Listing 8.3** A GAN class.

Examples of architectures for a generator and a discriminator are presented in the code below:

```
# First, we initialize the generator and the discriminator
# -generator
generator_net = nn.Sequential(nn.Linear(L, M), nn.ReLU(),
                              nn.Linear(M, D), nn.Tanh())

generator = Generator(generator_net, z_size=L)

# -discriminator
discriminator_net = nn.Sequential(nn.Linear(D, M), nn.ReLU(),
                                  nn.Linear(M, 1), nn.Sigmoid())

discriminator = Discriminator(discriminator_net)

# Eventually, we initialize the full model
model = GAN(generator=generator, discriminator=discriminator)
```

**Listing 8.4** Examples of architectures.

### 8.3.4 Training

One might think that the training procedure for GANs is more complicated than for any of the likelihood-based models. However, this is not the case. The only difference is that we need **two optimizers** instead of one. An example of a code with a training loop is presented below:

```
# We use two optimizers:
# optimizer_dis - an optimizer that takes the parameters of the discriminator
# optimizer_gen - an optimizer that takes the parameters of the generator
for indx_batch, batch in enumerate(training_loader):

    # -Discriminator
    # Notice that we call our model with the 'discriminator' mode.
    loss_dis = model.forward(batch, mode='discriminator')

    optimizer_dis.zero_grad()
    optimizer_gen.zero_grad()
    loss_dis.backward(retain_graph=True)
    optimizer_dis.step()

    # -Generator
    # Notice that we call our model with the 'generator' mode.
    loss_gen = model.forward(batch, mode='generator')

    optimizer_dis.zero_grad()
    optimizer_gen.zero_grad()
    loss_gen.backward(retain_graph=True)
    optimizer_gen.step()
```

**Listing 8.5** A training loop.

### 8.3.5 Results and Comments

In the experiments, we normalized images and scaled them to $[-1, 1]$ as we did for EBMs. After running it, you can expect similar results to those in Fig. 8.3.

In the previous chapters, we did not comment on the results. However, we make an exception here. Please note, my curious reader, that now we do not have a nicely converging objective. On the contrary, the adversarial loss or its generating part is jumping all over the place. That is a known fact following the min-max optimization problem. Moreover, the loss is learnable now, so it is troublesome to say where the optimal solution is since we update the loss function as well.

Another important piece of information is that training GANs is indeed a pain. First, it is hard to decipher and properly understand the values of adversarial loss. Second, learning is rather slow and requires many iterations (by many I mean hundreds if not thousands). If you look into generations in the first few epochs (e.g., see Fig. 8.4), you may be discouraged because a model may seem to overfit. That is the problem; we must be really patient to see whether we are on the right track. Moreover, you may also need to pay special attention to hyperparameters, e.g., learning rates. It requires a bit of experience or simply time to play around with learning rate values in your problem.

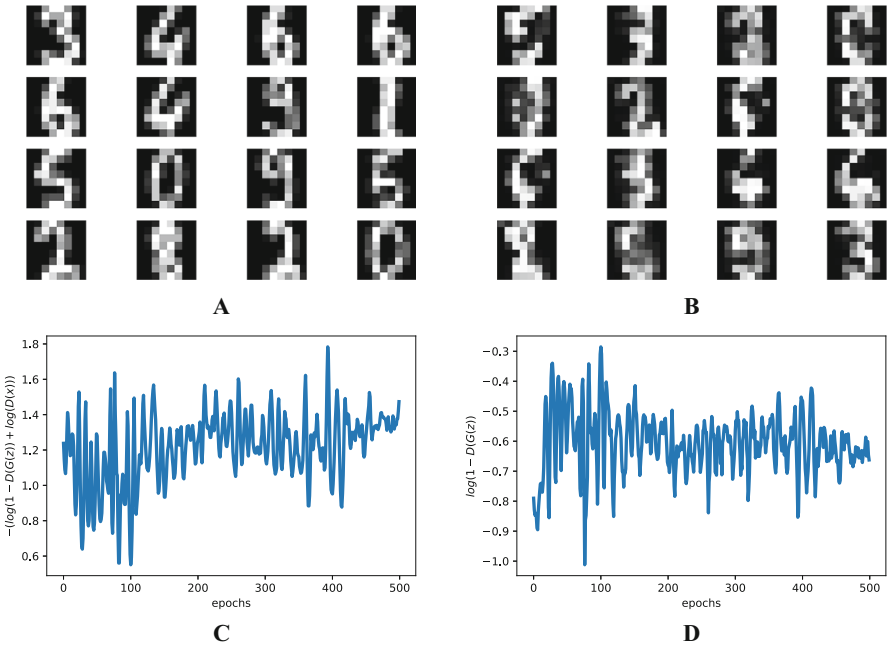



**Fig. 8.3** Examples of results after training GANs: (**a**) Real images. (**b**) Fake images. (**c**) The validation curve for the discriminator. (**d**) The validation curve for the generator.


**Fig. 8.4** Generated images after (**a**) 10 epochs of training and (**b**) 50 epochs of training.

Once you get through learning GANs, the reward is truly amazing. In the presented problem, with extremely simple neural nets, we can synthesize digits of high quality. That's the biggest advantage of GANs!

## 8.4 There Are Many GANs Out There!

Since the publication of the seminal paper on GANs [5] (however, the idea of the adversarial problem could be traced back to [6]), there was a flood of GAN-based

ideas and papers. I would not even dare to mention a small fraction of them. The field of implicit modeling with GANs is growing constantly. I will try to point to a few important papers:

- *Conditional GANs*: An important extension of GANs is allowing them to generate data conditionally [7].
- *GANs with encoders*: An interesting question is whether we can extend conditional GANs to a framework with encoders. It turns out that it is possible; see BiGAN [8] and ALI [9] for details.
- *StyleGAN* and *CycleGAN*: The flexibility of GANs could be utilized in formulating specialized image synthesizers. For instance, StyleGAN is formulated in such a way to transfer style between images [10], while CycleGAN tries to "translate" one image into another, e.g., a horse into a zebra [11].
- *Wasserstein GANs*: In [12], it was claimed that the adversarial loss could be formulated differently using the Wasserstein distance (a.k.a. the earth-mover distance), that is:

$$\ell_W(\alpha, \beta) = \mathbb{E}_{\mathbf{x}\sim p_{real}}\left[D_\alpha(\mathbf{x})\right] - \mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}\left[D_\alpha\left(G_\beta(\mathbf{z})\right)\right], \tag{8.10}$$

  where $D_\alpha(\cdot)$ must be a 1-Lipschitz function. The simpler way to achieve that is by clipping the weight of the discriminator to some small value $c$. Alternatively, *spectral normalization* could be applied [13] by using the power iteration method. Overall, constraining the discriminator to be a 1-Lipshitz function stabilizes training; however, it is still hard to comprehend the learning process.
- *f-GANs*: The Wasserstein GAN indicated that we can look elsewhere for alternative formulations of the adversarial loss. In [14], it is advocated to use f-divergences for that.
- *Generative moment matching networks* [15, 16]: As mentioned earlier, we could use other metrics instead of the likelihood function. We can fix the discriminator and define it as the maximum mean discrepancy with a given kernel function. The resulting problem is simpler because we do not train the discriminator; thus, we get rid of the cumbersome min-max optimization. However, the final quality of synthesized images is typically poorer.
- *Density difference vs. density ratio*: An interesting perspective is presented in [17, 18] where we can see various GANs either as a difference of densities or a ratio of densities. I refer to the original papers for further details.
- *Hierarchical implicit models*: The idea of defining implicit models could be extended to hierarchical models [19].
- *GANs and EBMs*: If you recall the EBMs, you may notice that there is a clear connection between the adversarial loss and the logarithm of the Boltzmann distribution. In [20, 21], it was noticed that introducing a variational distribution over observables, $q(\mathbf{x})$, leads to the following objective:

$$\mathcal{J}(\mathbf{x}) = \mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}\left[E(\mathbf{x})\right] - \mathbb{E}_{\mathbf{x}\sim q(\mathbf{x})}\left[E(\mathbf{x})\right] + \mathbb{H}\left[q(\mathbf{x})\right], \tag{8.11}$$

where $E(\cdot)$ is the energy function and $\mathbb{H}[\cdot]$ is the entropy. The problem again boils down to the min-max optimization problem, namely, minimizing with respect to the energy function and maximizing with respect to the variational distribution. The second difference between the adversarial loss and the variational lower bound here is the entropy term that is typically intractable.

- *What GAN to use?*: That is the question! Interestingly, it seems that training GANs greatly depends on the initialization and the neural nets rather than the adversarial loss or other tricks. You can read more about it in [22].
- *Training instabilities*: The main problem of GANs is unstable learning and a phenomenon called *mode collapse*, namely, a GAN samples beautiful images but only from some regions of the observable space. This problem has been studied for a long time by many (e.g., [23–25]); however, it still remains an open question.
- *Prescribed GANs*: Interestingly, it is possible to still calculate the likelihood of a GAN! See [26] for more details.
- *Regularized GANs*: There are many ideas to regularize GANs to achieve specific goals. For instance, InfoGAN aims to learn disentangled representations by introducing the mutual information-based regularizer [27].

Each of these ideas constitutes a separate research direction followed by thousands of researchers. If you are interested in pursuing any of these, I suggest picking one of the papers mentioned here and starting digging!

## References


1. David JC MacKay and Mark N Gibbs. Density networks. *Statistics and neural networks: advances at the interface*, pages 129–145, 1999.
2. Bharath K Sriperumbudur, Kenji Fukumizu, Arthur Gretton, Bernhard Schölkopf, and Gert RG Lanckriet. On integral probability metrics, $\phi$-divergences and binary classification. *arXiv preprint arXiv:0901.2698*, 2009.
3. Arthur Gretton, Karsten Borgwardt, Malte Rasch, Bernhard Schölkopf, and Alex Smola. A kernel method for the two-sample-problem. *Advances in Neural Information Processing Systems*, 19:513–520, 2006.
4. Tim Van Erven and Peter Harremos. Rényi divergence and Kullback-Leibler divergence. *IEEE Transactions on Information Theory*, 60(7):3797–3820, 2014.
5. Ian J Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *arXiv preprint arXiv:1406.2661*, 2014.
6. Jürgen Schmidhuber. Making the world differentiable: On using fully recurrent self-supervised neural networks for dynamic reinforcement learning and planning in non-stationary environments. *Institut für Informatik, Technische Universität München. Technical Report FKI-126*, 90, 1990.
7. Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784*, 2014.

8. Jeff Donahue, Philipp Krähenbühl, and Trevor Darrell. Adversarial feature learning. *arXiv preprint arXiv:1605.09782*, 2016.
9. Vincent Dumoulin, Ishmael Belghazi, Ben Poole, Olivier Mastropietro, Alex Lamb, Martin Arjovsky, and Aaron Courville. Adversarially learned inference. *arXiv preprint arXiv:1606.00704*, 2016.
10. Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4401–4410, 2019.
11. Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Proceedings of the IEEE international conference on computer vision*, pages 2223–2232, 2017.
12. Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein generative adversarial networks. In *International conference on machine learning*, pages 214–223. PMLR, 2017.
13. Takeru Miyato, Toshiki Kataoka, Masanori Koyama, and Yuichi Yoshida. Spectral normalization for generative adversarial networks. *arXiv preprint arXiv:1802.05957*, 2018.
14. Sebastian Nowozin, Botond Cseke, and Ryota Tomioka. f-GAN: Training generative neural samplers using variational divergence minimization. In *Advances in Neural Information Processing Systems*, pages 271–279, 2016.
15. Gintare Karolina Dziugaite, Daniel M Roy, and Zoubin Ghahramani. Training generative neural networks via maximum mean discrepancy optimization. In *Proceedings of the Thirty-First Conference on Uncertainty in Artificial Intelligence*, pages 258–267, 2015.
16. Yujia Li, Kevin Swersky, and Rich Zemel. Generative moment matching networks. In *International Conference on Machine Learning*, pages 1718–1727. PMLR, 2015.
17. Ferenc Huszár. Variational inference using implicit distributions. *arXiv preprint arXiv:1702.08235*, 2017.
18. Shakir Mohamed and Balaji Lakshminarayanan. Learning in implicit generative models. *arXiv preprint arXiv:1610.03483*, 2016.
19. Dustin Tran, Rajesh Ranganath, and David M Blei. Hierarchical implicit models and likelihood-free variational inference. *Advances in Neural Information Processing Systems*, 2017:5524–5534, 2017.
20. Taesup Kim and Yoshua Bengio. Deep directed generative models with energy-based probability estimation. *arXiv preprint arXiv:1606.03439*, 2016.
21. Shuangfei Zhai, Yu Cheng, Rogerio Feris, and Zhongfei Zhang. Generative adversarial networks as variational training of energy based models. *arXiv preprint arXiv:1611.01799*, 2016.
22. Mario Lucic, Karol Kurach, Marcin Michalski, Sylvain Gelly, and Olivier Bousquet. Are gans created equal? a large-scale study. *Advances in Neural Information Processing Systems*, 31, 2018.

23. Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans. *Advances in neural information processing systems*, 29:2234–2242, 2016.
24. Lars Mescheder, Andreas Geiger, and Sebastian Nowozin. Which training methods for gans do actually converge? In *International Conference on Machine Learning*, pages 3481–3490. PMLR, 2018.
25. Augustus Odena, Christopher Olah, and Jonathon Shlens. Conditional image synthesis with auxiliary classifier gans. In *International conference on machine learning*, pages 2642–2651. PMLR, 2017.
26. Adji B Dieng, Francisco JR Ruiz, David M Blei, and Michalis K Titsias. Prescribed generative adversarial networks. *arXiv preprint arXiv:1910.04302*, 2019.
27. Xi Chen, Yan Duan, Rein Houthooft, John Schulman, Ilya Sutskever, and Pieter Abbeel. Infogan: Interpretable representation learning by information maximizing generative adversarial nets. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, pages 2180–2188, 2016.

# Chapter 9
# Score-Based Generative Models


## 9.1 Introduction

Let us start with the following quote from the abstract of [1]:

*Creating noise from data is easy; creating data from noise is generative modeling.*

I must say that it is hard to come up with a shorter definition of concurrent generative modeling. Once we look at various classes of models, we immediately notice that this is exactly what we try to do: generate data from noise! Don't believe me? Ok, we should have a look at how various classes of generative models work:

- GANs: Sample noise $\mathbf{z}$ from a known $p(\mathbf{z})$ and use a generator $G(\mathbf{z})$ to get data.
- VAEs: Sample noise $\mathbf{z}$ from a prior $p(\mathbf{z})$ and use a decoder $p(\mathbf{x}|\mathbf{z})$ to sample data.
- Normalizing flows: Sample noise $\mathbf{z}$ from a base distribution $p(\mathbf{z})$, and use an invertible transformation $f$ to get data, $\mathbf{x} = f^{-1}(\mathbf{z})$.

I hope you see the pattern, my curious reader. In general, we can say that we look for a transformation $\psi$ that maps noise $\mathbf{z}$ to data $\mathbf{x}$, namely, $\mathbf{x} = \psi(\mathbf{z})$. An example is presented in Fig. 9.1.

In GANs, VAEs, and normalizing flows, the transformation is done in a "single step" by some neural net, namely, noise is mapped to data through a generator, a decoder, or an inversion of $f$, respectively. However, this does not need to be accomplished in this manner. In fact, we could easily think of an iterative process. And we know such a class of models already: diffusion-based models! The forward diffusion adds Gaussian noise to a datapoint in the consecutive steps until the datapoint becomes pure Gaussian noise, i.e., $\mathbf{x}_0 \sim p_{data}(\mathbf{x})$ and $\mathbf{x}_1 \sim \mathcal{N}(0, \mathbf{I})$ (a notation reminder: a datapoint $\mathbf{x} \equiv \mathbf{x}_0$ and noise $\mathbf{z} \equiv \mathbf{x}_1$). Then, the generative part (a.k.a. the backward diffusion) starts with $\mathbf{x}_1$ (i.e., pure Gaussian noise) and removes a small amount of noise step by step until data $\mathbf{x}_0$ is achieved. In the first implementations of diffusion models [2, 3], a finite number of steps were used, $T < +\infty$; thus, diffusion-based models could be seen as hierarchical VAEs with fixed encoders; see Chap. 5 and [4].



J. M. Tomczak, *Deep Generative Modeling*,
https://doi.org/10.1007/978-3-031-64087-2_9

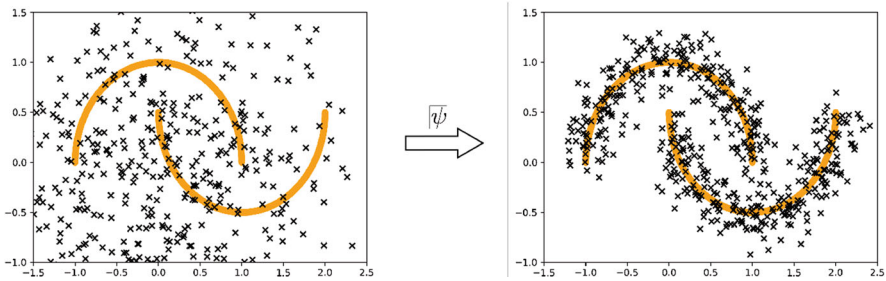



**Fig. 9.1** A schematic representation of generative modeling seen as mapping noise (left: black crosses) to data (right: black crosses around orange moons) using some transformation $\psi$.

In practice, the success of diffusion-based models comes from their depth, among other aspects. In other words, we can learn extremely deep hierarchical models (e.g., $T = 1000$) because the generation process is defined as removing small bits of noise for two consecutive variables. During training, a shared network is trained to predict this tiny-teeny amount of noise in a stochastic manner by sampling $t \in [0, 1]$ and then optimizing the following loss:

$$\mathcal{L}_t(\mathbf{x}_t) = \mathbb{E}_{\mathcal{N}(\epsilon|0,\mathbf{I})}\left[\gamma_t \|\epsilon - \epsilon_\theta(\mathbf{x}_t, t)\|^2\right], \tag{9.1}$$

where $\epsilon_t$ is a sample from the standard Gaussian, $\gamma_t$ is a *variance* following from the noise schedule, $\epsilon_\theta(\cdot)$ is a shared neural net across all $t$s that aims at predicting $\epsilon$ given $\mathbf{x}_t$ and $t$, and $\mathbf{x}_t$ could be expressed using given data $\mathbf{x}_0$ and $\epsilon$. For simplicity, $\gamma_t$ is set to 1 which results in the so-called simplified objective [3], but it could be also learned [5].

I know you, my curious reader, and I see your question coming. Since diffusion models work so well because the number of steps $T$ is large, why we should not go wild and take infinitely many steps, $T = +\infty$. It is a reasonable way of thinking; nonetheless, it entails some issues. First, we would need to deal with continuous $t \in [0, 1]$. However, we actually know that we can use differential equations to take care of this issue. But then a second issue arises: How to define generative models through differential equations? Ideally, we would like to obtain a method that allows us to train such models in a simple manner with a loss similar to (9.1). It turns out that it is possible! But first, we will look into a different way of changing noise to data, a method called **score matching**. After that, we will be equipped with a learning strategy for dealing with continuous time (i.e., infinitely deep generative models). Eventually, we will discuss an alternative to score matching called **flow matching**.

## 9.2 Score Matching

### 9.2.1 Modeling and the Objective

Keep in mind that our goal is to propose a model that turns noise into data in an iterative manner. Here is our first attempt. Imagine the following situation:

- We have access to the real distribution $p_{real}(\mathbf{x})$, where $\mathbf{x} \in \mathcal{X}$.
- We take any point in the data space at random (e.g., we sample from a uniform distribution over $\mathcal{X}$), $\hat{\mathbf{x}} \in \mathcal{X}$. There is a very high chance that $p_{real}(\hat{\mathbf{x}})$ is close to 0, $p_{real}(\hat{\mathbf{x}}) \approx 0$.
- The **question** is then this: How to turn $\hat{\mathbf{x}}$ into a legit datapoint (i.e., an object that could be observed in real life)?

Actually, there is a good (i.e., relatively fast and theoretically well-grounded) method called **stochastic gradient Langevin dynamics** (SGLD) a.k.a. Langevin dynamics [6] that allows sampling from $p_{real}(\mathbf{x})$. If we start in $\mathbf{x}_0 \equiv \hat{\mathbf{x}}$, then by following this procedure:

$$\mathbf{x}_{t+\Delta} = \mathbf{x}_t + \alpha \nabla_{\mathbf{x}_t} \ln p_{real}(\mathbf{x}) + \eta \cdot \epsilon, \tag{9.2}$$

where $\Delta$ is an increment, $\alpha > 0$, $\eta > 0$, and $\epsilon \sim \mathcal{N}(0, \mathbf{I})$, we eventually end up in a point $\mathbf{x}_0$ for which $p_{real}(\mathbf{x}_0) > 0$ and which is somewhere around a mode.

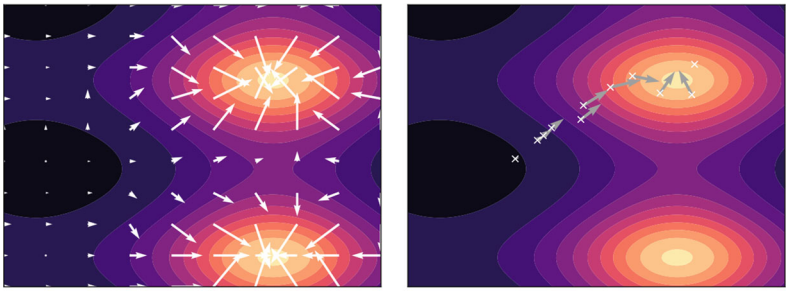
That is great! However, there is one big issue, namely, we do not know the real distribution $p_{real}(\mathbf{x})$. We are back to square one, and we can use GANs, VAEs, normalizing flows, etc. for finding a model $p_\theta(\mathbf{x})$. Or do we? Let us have another look at (9.2). We do not require to know $p_{real}(\mathbf{x})$ but the gradient of the logarithm of the real distribution, $\nabla_{\mathbf{x}} \ln p_{real}(\mathbf{x})$. This quantity is known as the **score function**, $s(\mathbf{x}) \stackrel{df}{=} \nabla_{\mathbf{x}} \ln p_{real}(\mathbf{x})$. Having $s(\mathbf{x})$ allows us to run SGLD to find new points that, eventually, follow the real distribution. An example of the score function for a multimodal distribution is presented in Fig. 9.2. The score function is represented as vectors over a grid (Fig. 9.2a). As we can see, the score function indicates moving toward the modes, and, thus, it steers Langevin dynamics (Fig. 9.2b).

As a result, instead of fitting a model to the data distribution, ***we can fit a model to the score function***. Keep in mind that the score function is a gradient; thus, it returns an object of the same shape as $\mathbf{x}$ that could be represented as a vector. Therefore, we can find the best model of the score function by optimizing the following regression problem [7, 8]:

$$\mathcal{J}(\theta) = \frac{1}{2} \int \| s_\theta(\mathbf{x}) - \nabla_{\mathbf{x}} \ln p_{data}(\mathbf{x}) \|^2 \, p_{data}(\mathbf{x}) \, \mathrm{d}\mathbf{x}. \tag{9.3}$$

This integral is not anything special because it is the mean squared error loss, a typical loss used for regression.

The arising problem with (9.3) is that it is not differentiable since $p_{data}(\mathbf{x}) = \frac{1}{N} \sum_{n=1}^{N} \delta(\mathbf{x} - \mathbf{x}_n)$, where $\delta(\cdot)$ is Dirac's delta. Since Dirac's delta is a non-



A B

**Fig. 9.2** (**a**) An illustration of the score function $s(\mathbf{x})$ plotted as vectors (white arrows) on a regular grid for a multimodal distribution (dark colors correspond to low probability; bright colors depict high probability). (**b**) A trajectory after applying the SGLD (consecutive points are represented by white crosses, and their scores are denoted by gray arrows).

differentiable function of $\mathbf{x}$, we cannot solve (9.3) using autograd. There is a solution, though, by adding some small Gaussian noise with variance $\sigma^2$ to data, i.e., $\tilde{\mathbf{x}}_n = \mathbf{x}_n + \sigma \cdot \epsilon$, where $\epsilon \sim \mathcal{N}(0, \mathbf{I})$. The resulting distribution is Gaussian, $\mathcal{N}(\tilde{\mathbf{x}}_n|\mathbf{x}_n, \sigma^2)$. In other words, we can turn $p_{data}(\mathbf{x})$ into a mixture of Gaussians with data as means and some small variance $\sigma^2$, namely:

$$q_{data}(\tilde{\mathbf{x}}_n) = \frac{1}{N}\sum_{n=1}^{N} \mathcal{N}(\tilde{\mathbf{x}}_n|\mathbf{x}_n, \sigma^2). \tag{9.4}$$

Eventually, instead of using the non-differentiable objective in (9.3), we can formulate a differentiable objective by replacing $p_{data}(\mathbf{x}_n)$ with $q_{data}(\tilde{\mathbf{x}}_n)$ in (9.3) [7–9]:

$$\mathcal{L}(\theta) = \frac{1}{2N}\sum_{n=1}^{N} \int \|s_\theta(\tilde{\mathbf{x}}) - \nabla_{\tilde{\mathbf{x}}} \ln \mathcal{N}(\tilde{\mathbf{x}}|\mathbf{x}_n, \sigma^2)\|^2 \, \mathcal{N}(\tilde{\mathbf{x}}_n|\mathbf{x}_n, \sigma^2) \, \mathrm{d}\tilde{\mathbf{x}}. \tag{9.5}$$

One may say that there is still a problem because we have a gradient to calculate. However, we know the closed form of the score function for the Gaussian distribution:

$$\begin{aligned}
\nabla_{\tilde{\mathbf{x}}} \ln \mathcal{N}(\tilde{\mathbf{x}}_n|\mathbf{x}_n, \sigma^2) &= \nabla_{\tilde{\mathbf{x}}} \left( -\ln(2\pi\sigma^D) - \frac{1}{2\sigma^2}(\tilde{\mathbf{x}}_n - \mathbf{x}_n)^2 \right) && (9.6)\\
&= -\nabla_{\tilde{\mathbf{x}}} \frac{1}{2\sigma^2}(\tilde{\mathbf{x}}_n - \mathbf{x}_n)^2 && (9.7)\\
&= -\frac{1}{\sigma^2}(\tilde{\mathbf{x}}_n - \mathbf{x}_n) && (9.8)\\
&= -\frac{1}{\sigma^2}(\mathbf{x}_n + \sigma \cdot \epsilon - \mathbf{x}_n) && (9.9)\\
&= -\frac{1}{\sigma}\epsilon, && (9.10)
\end{aligned}$$

where in the fourth equation we used the definition of $\tilde{\mathbf{x}}$, $\tilde{\mathbf{x}}_n = \mathbf{x}_n + \sigma \cdot \epsilon$. Thus, the score is the following:

$$\nabla_{\tilde{\mathbf{x}}} \ln \mathcal{N}(\tilde{\mathbf{x}}_n | \mathbf{x}_n, \sigma^2) = -\frac{1}{\sigma}\epsilon, \tag{5}$$

and by plugging (5) in (9.5), we get the differentiable objective in the final form:

$$\mathcal{L}(\theta) = \frac{1}{2N}\sum_{n=1}^{N}\int \|s_\theta(\tilde{\mathbf{x}}) + \frac{1}{\sigma}\epsilon\|^2 \, \mathcal{N}(\epsilon|0, \sigma^2\mathbf{I}) \, \mathrm{d}\epsilon \tag{9.11}$$

$$= \frac{1}{2N}\sum_{n=1}^{N}\mathbb{E}_{\mathcal{N}(\epsilon|0,\mathbf{I})}\left[\|s_\theta(\tilde{\mathbf{x}}) + \frac{1}{\sigma}\epsilon\|^2\right] \tag{9.12}$$

$$= \frac{1}{N}\sum_{n=1}^{N}\mathbb{E}_{\mathcal{N}(\epsilon|0,\mathbf{I})}\left[\frac{1}{2\sigma}\|\epsilon - \tilde{s}_\theta(\tilde{\mathbf{x}})\|^2\right] \tag{9.13}$$

where in the last line we modified the score model to be of the following form, $\tilde{s}_\theta(\mathbf{x}) = -\sigma s_\theta(\tilde{\mathbf{x}})$, and we used the property of $\sigma$ being positive (remember, it is the standard deviation that is always positive!). Of course, while implementing, we can parameterize $\tilde{s}_\theta(\tilde{\mathbf{x}})$ using a neural network directly; we do not have to parameterize $s_\theta(\mathbf{x})$ first and then rescale it; there is no need for that. We just need to be aware of all the steps here. Learning the score model $\tilde{s}_\theta(\tilde{\mathbf{x}})$ by optimizing the objective in (9.13) is called (denoising) **score matching** [7, 8, 10].

Score matching allows us to learn a generative model in a completely different manner. Instead of **matching distributions**, we can perform **matching scores** by solving a regression problem. This is a new perspective on training generative models, but keep in mind that, eventually, this approach is used to generate data from noise by applying Langevin dynamics. Below, we gather practical information about score matching.

### 9.2.2 Training

We can learn the score model by repeating the following steps:

**Training for SM**

1. Pick a datapoint $\mathbf{x}$.
2. Sample $\epsilon$ from $\mathcal{N}(\epsilon|0, \mathbf{I})$.
3. Calculate the noisy version of data, $\tilde{\mathbf{x}} = \mathbf{x} + \sigma \cdot \epsilon$.
4. Calculate the score, $\tilde{s}_\theta(\tilde{\mathbf{x}})$.
5. Calculate gradient with respect to $\theta$ of the score matching objective, i.e., $\Delta\theta = \nabla_\theta \frac{1}{2\sigma}\|\epsilon - \tilde{s}_\theta(\tilde{\mathbf{x}})\|^2$. We accomplish that by applying automatic differentiation.
6. Update the score model: $\theta := \theta - \Delta\theta$. Go to 1 until STOP.

In practice, we use mini-batches, but the whole procedure remains the same.

### 9.2.3 Sampling (Generation)

Once we learn the score model, we can use it for sampling using Langevin dynamics. The generative procedure is the following:

**Sampling with Langevin Dynamics for SM**

1. Sample a point $\mathbf{x}_0$ at random (e.g., using a uniform distribution).
2. Run Langevin dynamics for $T$ steps, where $\Delta = \frac{1}{T}$:

$$\mathbf{x}_{t+\Delta} = \mathbf{x}_t + \alpha\, \tilde{s}_\theta(\mathbf{x}_t) + \eta \cdot \epsilon, \tag{9.14}$$

3. Return the final point $\mathbf{x}_1$.

The final point should follow the data distribution. Note that to keep the formulation of Langevin dynamics unchanged, we used $s_\theta(\mathbf{x}_t)$ instead $\tilde{s}_\theta(\mathbf{x}_t)$. There is no problem though because we know the relation $\tilde{s}_\theta(\mathbf{x}_t) = -\sigma s_\theta(\mathbf{x}_t)$; thus, $s_\theta(\mathbf{x}_t) = -\frac{1}{\sigma}\tilde{s}_\theta(\mathbf{x}_t)$.

### 9.2.4 Score Matching and Diffusion-Based Models

I know you, my curious reader, and I can bet you have noticed something. If we look at the objective for diffusion-based models in (9.3) and the score models in (9.13), we can clearly see they are almost identical (especially, if we set $\gamma_t = \frac{1}{\sigma}$)! In fact, we can turn score matching into an (almost) diffusion-based approach. For this, we need three things:

1. We introduce a schedule for variances $\sigma_t^2$, for

$$t \in \mathcal{T} = \{t : t \text{ goes from 0 to 1 with the increment } \Delta = \frac{1}{T}\}.$$

2. We modify the score model to be "aware" of this schedule, namely, $\tilde{s}_\theta(\mathbf{x}, \sigma_t^2)$.
3. We modify the objective to be "aware" of this schedule:

$$\mathcal{L}(\theta) = \frac{1}{2N}\sum_{n=1}^{N}\sum_{t\in\mathcal{T}} \lambda_t \mathbb{E}_{\mathcal{N}(\epsilon|0,\mathbf{I})}\left[\frac{1}{\sigma}\|\epsilon - \tilde{s}_\theta(\tilde{\mathbf{x}}, \sigma_t^2)\|^2\right], \tag{8}$$

where $\lambda_t$ are weighting coefficients, e.g., $\lambda_t = \sigma_t^2$.

Sampling for such a model requires some changes. For instance, annealed Langevin dynamics could be used; see Algorithm 1 in [10]. Eventually, we end

up with a sampling procedure that is very similar to diffusion-based models. The differences lie in the sampling scheme that runs multiple steps of Langevin dynamics per each $\sigma_t^2$, while diffusion-based models do a single step per one "denoising."

### 9.2.5 Coding Score Matching

We outlined how to implement training and sampling (generation) using score matching. The code is pretty straightforward. The only *trick* is to transform data to be in $[-1, 1]$. In the code below, we make only one difference compared to our analysis above, namely, we use $\frac{1}{2\sigma}\|\epsilon + \tilde{s}_\theta(\tilde{\mathbf{x}})\|^2$ such that Langevin dynamics is in its standard form: $\mathbf{x}_{t+\Delta} = \mathbf{x}_t + \alpha\ \tilde{s}_\theta(\mathbf{x}_t) + \eta \cdot \epsilon$.

```
class ScoreMatching(nn.Module):
    def __init__(self, snet, alpha, sigma, eta, D, T):
        super(ScoreMatching, self).__init__()

        print('Score Matching by JT.')

        self.snet = snet

        # other hyperparams
        self.D = D

        self.sigma = sigma

        self.T = T

        self.alpha = alpha

        self.eta = eta

    def sample_base(self, x_1):
        # Uniform over [-1, 1]**D
        return 2. * torch.rand_like(x_1) - 1.

    def langevin_dynamics(self, x):
        for t in range(self.T):
            x = x + self.alpha * self.snet(x) + self.eta * torch.
    randn_like(x)
        return x

    def forward(self, x, reduction='mean'):
        # =====Score Matching
        # sample noise
        epsilon = torch.randn_like(x)

        # =====
        # calculate the noisy data
        tilde_x = x + self.sigma * epsilon

```

```
        # =====
        # calculate the score model
        s = self.snet(tilde_x)

        # =====LOSS: the Score Matching Loss
        SM_loss = (1. / (2. * self.sigma)) * ((s + epsilon)**2.).
    sum(-1) # in order to keep the Langevin dynamics unchanged,
    we do not use \tilde{s} = -sigma * s but we use \tilde{s} =
    sigma * s

        # Final LOSS
        if reduction == 'sum':
            loss = SM_loss.sum()
        else:
            loss = SM_loss.mean()

        return loss

    def sample(self,  batch_size=64):
        # sample x_0
        x = self.sample_base(torch.empty(batch_size, self.D))

        # run langevin dynamics
        x = self.langevin_dynamics(x)

        x = torch.tanh(x)
        return x
```

**Listing 9.1** A class for score matching.

After running the code with an MLP-based scoring model and the following values of the hyperparameters $\alpha = 0.1$, $\sigma = 0.1$, $\eta = 0.05$, and $T = 100$, we can expect results like in Fig. 9.3.

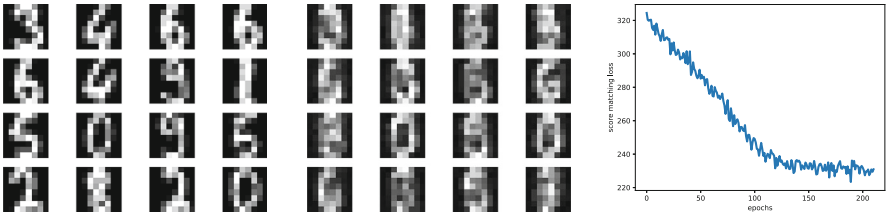



**Fig. 9.3** Results for score matching: (**a**) A sample of real images. (**b**) A sample of generated images. (**c**) An example of the score matching loss calculated on the validation set.

### 9.2.6 What Can We Do with Score Matching?

**Improving score matching.** As we can see, we can get some (quite reasonable) generations using an extremely simplistic code. However, there are some issues with score matching as indicated in the original paper [10]:

- First and foremost, low-density regions (i.e., regions with little to no data) can cause difficulties in score estimation. Take a look at Fig. 9.2a again, and look closer at the real score in the dark areas. In these regions, it is very unlikely to see any datapoint. As a result, we can very badly estimate the score function.
- The second issue follows from the first one, namely, for a badly trained score model, Langevin dynamics could result in bad samples. To some degree, we can see that in our example in Fig. 9.3b.
- Third, if modes of real data distribution are far away, Langevin dynamics would require multiple steps to reach them. This problem is known as *slow mixing*.

To overcome these issues, we can use the following methods:

- Instead of using a single variance, we can choose a sequence of variances and the annealed Langevin dynamics method, as discussed earlier. For details, we recommend [10] and a follow-up with other useful tricks [11].
- A better score model, especially for high-dimensional cases, could be trained by following *sliced score matching* [12]. The idea is similar to sliced Wasserstein distance [13] in which the score function and the score model are projected onto some random direction. Then, the new sliced score matching objective is easier to calculate (it requires only vector multiplication). This new objective is closely related to Hutchinson's trace estimator.

**Other extensions.** Score matching is a general framework that could be further extended. Here are some examples:

- Score matching is a suitable approach for energy-based models. This idea was used for learning Boltzmann machines [14, 15] and other energy-based models [16].
- The idea of matching scores of distributions over random variables could be utilized in learning distributions over discrete random variables [17]. For this, we need to use a different definition of scores, appropriate for probability mass functions.
- Score matching could be used beyond images. In [18], the authors proposed a method for modeling point clouds. Their approach utilized a specific kind of score model that is feasible for processing a cloud of points.

## 9.3 Generative Models as Stochastic/Ordinary Differential Equations

### 9.3.1 A Reminder on Diffusion-Based Models

Let us think about diffusion-based models again. We introduced them as hierarchical VAEs with very specific variational posteriors that were constructed as a linear transformation of previous latents and some added Gaussian noise. They could be also seen as a dynamical system with discretized time. To see that, let us say we transform a datapoint, $\mathbf{x}_0$, $\mathbf{x}_0 \sim p_0(\mathbf{x}) \equiv p_{data}(\mathbf{x})$ to noise, $\mathbf{x}_1$, sampled from a known distribution $\pi$, $\mathbf{x}_1 \sim p_1(\mathbf{x}) \equiv \pi(\mathbf{x})$, the "time" is denoted by $t \in [0, 1]$, and there are $T$ steps between $\mathbf{x}_0$ and $\mathbf{x}_1$, namely, a step size is $\Delta = \frac{1}{T}$. Then we can represent the *forward diffusion* as follows:

$$\mathbf{x}_{t+\Delta} = \sqrt{1-\beta_t}\,\mathbf{x}_t + \sqrt{\beta_t}\,\epsilon_t, \tag{9.15}$$

where $\epsilon_t \sim \mathcal{N}(0, \mathbf{I})$. As you can notice, my curious reader, this is a dynamical system with discretized time. Please keep it in mind; we will come back to that later.

What is an interesting trait of diffusion-based models is that calculating $\mathbf{x}_t$ directly from data $\mathbf{x}_0$ is possible because of the linearity and Gaussianity, namely:

$$\mathbf{x}_t = \sqrt{\alpha_t}\,\mathbf{x}_0 + \sqrt{1-\alpha_t}\,\epsilon_t, \tag{9.16}$$

where $\alpha_t = \prod_{\tau=1}^{t}(1-\beta_\tau)$. Further, we can calculate the datapoint back in the following manner:

$$\mathbf{x}_0 = \frac{1}{\sqrt{\alpha_t}}\,\mathbf{x}_t - \frac{\sqrt{1-\alpha_t}}{\sqrt{\alpha_t}}\,\epsilon_t. \tag{9.17}$$

However, unlike in the forward diffusion in which we sample $\epsilon_t$, when we reverse $\mathbf{x}_t$ to $\mathbf{x}_0$, we do not have access to the noise $\epsilon_t$. The standard situation is the following: We proceed with the forward diffusion until $\mathbf{x}_t$ and disregard all previous $\mathbf{x}$s and $\epsilon$s. However, not everything is lost! After all, we can introduce a neural network $\epsilon_\theta(\mathbf{x}_t, t)$ that aims to predict the noise.

How to learn this neural network? In the seminal paper [3], the loss is a sum of the following losses:

$$\mathcal{L}_t(\theta) = \|\epsilon_\theta(\mathbf{x}_t, t) - \epsilon_t\|^2, \tag{9.18}$$

which was shown to be equivalent to the ELBO. But wait a second, do you see what I see, my curious reader? This loss is equivalent to another loss! Yes, it is score matching as presented in the previous section. There exists the following correspondence between the score model and the noise model (for Gaussian denoising distribution with the standard deviation $\sigma$):

$$s_\theta(\mathbf{x}, t) = -\frac{\epsilon_\theta(\mathbf{x}_t, t)}{\sigma}. \tag{9.19}$$

Let us sum these considerations up:

- The forward diffusion defines a discrete-time dynamical system.
- The loss function for diffusion-based models is (almost) identical to the score matching loss.
- In fact, diffusion-based models correspond to score models.
- Diffusion-based models are very similar to score models trained with a schedule for $\sigma$ because both models are iterative and both models consider more noise at each step.

These similarities indicate that there may be an underlying framework that could generalize score models and diffusion-based models.

### 9.3.2 In the Pursuit of a General Framework

Before we see an answer to this question about a general framework for score models and diffusion-based models, let us go back to the remark we made in the very beginning: "Diffusion-based models could be also seen as a dynamical system with discretized time" because the forward diffusion is the following (we repeat it, but it never hurts to repeat something!):

$$\mathbf{x}_{t+\Delta} = \sqrt{1-\beta_t}\,\mathbf{x}_t + \sqrt{\beta_t}\,\epsilon_t\,. \tag{9.20}$$

We can be honest with each other, my curious reader, not everyone learned dynamical systems during their studies or remembers them. Before I worked on this section, my memory of ordinary differential equations (ODEs) and stochastic differential equations (SDEs) was very (very!) rusty. Therefore, let us delve into the world of differential equations very briefly! For more curious readers (yes, yes, I mean you!), I highly recommend the following book: [19]. It is a great balance between basic and advanced topics.

Just a glimpse into the future: It turns out that SDEs and ODEs have a lot to offer for generative modeling. Hence, please stay with me, and we will discover a new, beautiful world of a new class of generative models!

#### 9.3.2.1 ODEs and Numerical Methods

We start our discussion with ODEs. In general, we can define them as follows:

$$\frac{\mathrm{d}\mathbf{x}_t}{\mathrm{d}t} = f(\mathbf{x}_t, t), \tag{9.21}$$

with some initial conditions $\mathbf{x}_0$. Sometimes, $f(\mathbf{x}_t, t)$ is referred to as a *vector field*.

We can solve this ODE by running one of the numerical methods that aims to discretize time in a specific way. For instance, Euler's method carries it out in the

following way (starting from $t = 0$ and proceeding to $t = 1$ with a step $\Delta$):

$$\mathbf{x}_{t+\Delta} - \mathbf{x}_t = f(\mathbf{x}_t, t) \cdot \Delta \tag{9.22}$$

$$\mathbf{x}_{t+\Delta} = \mathbf{x}_t + f(\mathbf{x}_t, t) \cdot \Delta. \tag{9.23}$$

Sometimes, it is necessary to run from $t = 1$ to $t = 0$, and then we can apply backward Euler's method:

$$\mathbf{x}_t = \mathbf{x}_{t+\Delta} - f(\mathbf{x}_{t+\Delta}, t + \Delta) \cdot \Delta. \tag{9.24}$$

I know you, my brilliant reader, you see a connection, don't you? If we take $\mathbf{x}_0$ to be our data and $\mathbf{x}_1$ to be noise, then *if* we knew $f(\mathbf{x}_t, t)$, we could run backward Euler's method to get a generative model! Please keep this thought in mind for now. But yes, you are right!

#### 9.3.2.2 SDEs and Probability Flow ODEs

Now, we will look into SDEs. In general, we can think of SDEs as ODEs whose trajectories are random and are distributed according to some probabilities at each $t$, $p_t(\mathbf{x})$. SDEs could be defined as follows:

$$\mathrm{d}\mathbf{x}_t = f(\mathbf{x}_t, t)\mathrm{d}t + g(t)\mathrm{d}\mathbf{v}_t, \tag{9.25}$$

where $\mathbf{v}$ is a standard Wiener process, $f(\cdot, t)$ in this context is referred to as *drift*, and $g(\cdot)$ is a scalar function called *diffusion*. The drift component is deterministic (take a look at the formulation of ODEs above), but the diffusion element is stochastic due to the standard Wiener process (for more about Wiener processes, see [19]; here, we do not need to know more than that they behave like Gaussians, e.g., the difference of its increments is normally distributed, $\mathbf{v}_{t+\Delta} - \mathbf{v}_t \sim \mathcal{N}(0, \Delta)$). A side note: The forms of drift and diffusion are assumed to be **known**. We will give an example later; for now, just remember that they are given, my curious reader.

An important property of this SDE is the existence of a corresponding ordinary differential equation whose solutions follow the same distribution! If we start with a datapoint $\mathbf{x}_0$, we can get noise $\mathbf{x}_1 \sim p_1(\mathbf{x})$ by solving the following probability flow (PF) ODE [1]:

$$\frac{\mathrm{d}\mathbf{x}_t}{\mathrm{d}t} = \left( f(\mathbf{x}_t, t) - \frac{1}{2} g^2(t) \nabla_{\mathbf{x}_t} \ln p_t(\mathbf{x}_t) \right). \tag{9.26}$$

Let us take another look at that and see what we got:

- First of all, we do not have the Wiener process anymore. As a result, we deal with an ODE instead of an SDE.
- Second, the drift component and the diffusion components are still here, but the diffusion is multiplied by $-\frac{1}{2}$ and squared.
- Third, there is the score function! If you want to see a derivation, please check [1]; here, we take it for granted as true. However, it looks reasonable. SDEs have

solutions that are distributed according to $p_t(\mathbf{x})$; hence, it is not surprising that the score function pops up here. After all, the score function indicates how a trajectory should look according to $p_t(\mathbf{x})$.

#### 9.3.2.3 PF-ODEs as Score-Based Generative Models

But wait a second! What do we have here? If we assume for a second that the score function is known, we can use this PF-ODE as a generative model by applying backward Euler's method starting from $\mathbf{x}_1 \sim \pi(\mathbf{x})$:

$$\mathbf{x}_t = \mathbf{x}_{t+\Delta} - \left( f(\mathbf{x}_{t+\Delta}, t+\Delta) - \frac{1}{2} g^2(t+\Delta) \nabla_{\mathbf{x}_{t+\Delta}} \ln p_t(\mathbf{x}_{t+\Delta}) \right) \cdot \Delta. \tag{9.27}$$

Perfect! Now, the problem is the score function, but we know already how to deal with that; we can use denoising score matching (i.e., score matching with the noisy empirical distribution being a mixture of Gaussians centered at datapoints) for learning it. The difference to denoising score matching is that we need to take time $t$ into account:

$$\mathcal{L}(\theta) = \int_0^1 \mathcal{L}_t(\theta) \mathrm{d}t. \tag{9.28}$$

How to define $\mathcal{L}_t(\theta)$? The score matching idea tells us that we should consider $\lambda_t \| s_\theta(\mathbf{x}_t, t) - \nabla_{\mathbf{x}_t} \ln p_t(\mathbf{x}_t) \|^2$, but since we cannot calculate $p_t(\mathbf{x}_t)$, we should use something else. Instead, we can define a distribution $p_{0t}(\mathbf{x}_t | \mathbf{x}_0)$ that allows sampling noisy versions of our original datapoints $\mathbf{x}_0$s. Putting it all together yields:

$$\mathcal{L}_t(\theta) = \frac{1}{2} \mathbb{E}_{\mathbf{x}_0 \sim p_{data}(\mathbf{x})} \mathbb{E}_{\mathbf{x}_t \sim p_{0t}(\mathbf{x}_t | \mathbf{x}_0)} \left[ \lambda_t \| s_\theta(\mathbf{x}_t, t) - \nabla_{\mathbf{x}_t} \ln p_{0t}(\mathbf{x}_t | \mathbf{x}_0) \|^2 \right]. \tag{9.29}$$

Importantly, if we take $p_{0t}(\mathbf{x}_t | \mathbf{x}_0)$ to be Gaussian, then we could calculate the score function analytically. We will look into an example soon. Moreover, to calculate $\mathcal{L}_t(\theta)$, we can use a single sample (i.e., the Monte Carlo estimate).

After finding $s_\theta(\mathbf{x}_t, t)$, we can sample data by running backward Euler's method as follows:

$$\mathbf{x}_t = \mathbf{x}_{t+\Delta} - \left( f(\mathbf{x}_{t+\Delta}, t+\Delta) - \frac{1}{2} g^2(t+\Delta) s_\theta(\mathbf{x}_{t+\Delta}, t+\Delta) \right) \cdot \Delta. \tag{9.30}$$

Please keep in mind that drift and diffusion are assumed to be known. Additionally, we stick to (backward) Euler's method, but you can pick another ODE solver, go ahead, and be wild! But here, we want to be clear and as simple as possible.

In Fig. 9.4, we present an example of using backward Euler's method for sampling. For some PF-ODE and a given score function, we can obtain samples from a multimodal distribution. As one may imagine, the ODE solver "goes" toward modes. In this simple example, we can notice that defining PF-ODEs is a powerful genera-

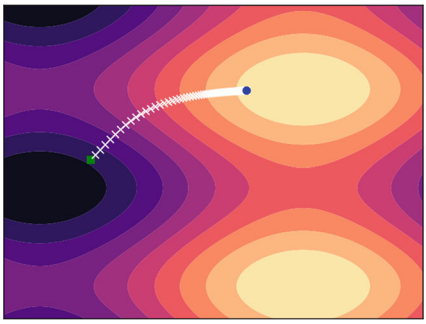


**Fig. 9.4** An example of running backward Euler's method for a multimodal distribution. Here, we used $f(\mathbf{x}, t) = 0$, $g(t) = 9^t$, $T = 100$, and the score function (not model!) was calculated using autograd. The green square denotes $\mathbf{x}_1 \sim \pi$, and the blue circle is a sample $\mathbf{x}_0$.

tive tool! Once the score function is properly approximated, we can sample from the original distribution in a straightforward manner.

### 9.3.3 An Example of Score-Based Generative Models: Variance Exploding PF-ODE

#### 9.3.3.1 Model Formulation

To define our own score-based generative model (SBGM), we need the following element:

- The drift $f(\mathbf{x}, t)$
- The diffusion $g(t)$
- The form of $p_{0t}(\mathbf{x}_t|\mathbf{x}_0)$

In [1] and [20], we can find three examples of SGBMs, namely, variance exploding (VE) SDE, variance preserving SDE, and sub-VP SDE. Here, we focus on the VE SDE that assumes the following choices of the drift and the diffusion:

- $f(\mathbf{x}, t) = 0$,
- $g(t) = \sigma^t$, where $\sigma > 0$ is a hyperparameter; note that we take $\sigma$ to the power of time $t \in [0, 1]$.

Then, according to our discussion above, plugging in our choices for $f(\mathbf{x}, t)$ and $g(t)$ in the general form of the PF-ODE yields:

$$\frac{\mathrm{d}\mathbf{x}_t}{\mathrm{d}t} = -\frac{1}{2}\sigma^{2t}\nabla_{\mathbf{x}_t} \ln p_t(\mathbf{x}_t). \tag{9.31}$$

Now, to learn the score model, we need to define the conditional distribution for obtaining a noisy version of $\mathbf{x}_0$. Fortunately, the theory of SDEs (e.g., see Chapter 5

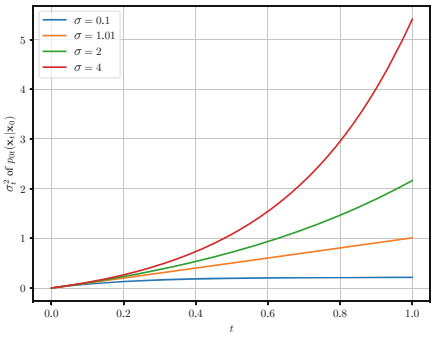



**Fig. 9.5** The dependency of the standard deviation of $p_{0t}(\mathbf{x}_t|\mathbf{x}_0)$ on $t$ for different choices of $\sigma$.

of [19]) tells us how to calculate $p_{0t}(\mathbf{x}_t|\mathbf{x}_0)$! Specific formulas are presented in the Appendix of [1]. Here, we provide the final solution:

$$p_{0t}(\mathbf{x}_t|\mathbf{x}_0) = \mathcal{N}\left(\mathbf{x}_t|\mathbf{x}_0, \frac{1}{2\ln\sigma}(\sigma^{2t}-1)\mathbf{I}\right); \tag{9.32}$$

thus, the variance function over time is the following:

$$\sigma_t^2 = \frac{1}{2\ln\sigma}(\sigma^{2t}-1). \tag{9.33}$$

Eventually, the final distribution, $p_{01}(\mathbf{x})$, gets approximately close to the following Gaussian (for sufficiently large $\sigma$):

$$p_{01}(\mathbf{x}) = \int p_0(\mathbf{x}_0) * \mathcal{N}\left(\mathbf{x}|\mathbf{x}_0, \frac{1}{2\ln\sigma}(\sigma^2-1)\mathbf{I}\right) \tag{9.34}$$

$$\approx \mathcal{N}\left(\mathbf{x}|0, \frac{1}{2\ln\sigma}(\sigma^2-1)\mathbf{I}\right). \tag{9.35}$$

We will use $p_{01}(\mathbf{x})$ to sample noise $\mathbf{x}_1$ and then for reverting it to data $\mathbf{x}_0$.

**A Side Note**

To better understand how various values of $\sigma$ influence $\sigma_t$, we can have a look at curves in Fig. 9.5. For obvious reasons, we cannot pick $\sigma = 1$ (why, you ask? well, $ln(1) = 0$, so we get into an issue of dividing by 0), but for $\sigma = 1.01$, we get $p_1(\mathbf{x})$ close to the standard Gaussian.

#### 9.3.3.2 The Choice of $\lambda_t$

The last remark, before we move to the training procedure, is about the choice of $\lambda_t$ in the definition of $\mathcal{L}_t(\theta)$. So far, I simply omitted that but I had a good reason. Ho et al. [3] simply set $\lambda_t \equiv 1$. Done! Really though? Well, as you can imagine, but smart reader, that is not so easy. Song and Kingma [21] showed that it is actually beneficial to set $\lambda_t = \sigma_t^2$ in the case of VE PF-ODE. Then, we can even use the sum over $\mathcal{L}_t(\theta)$ as a proxy to the log-likelihood function. We will take advantage of that for early stopping in our training procedure.

#### 9.3.3.3 Training

We present a training procedure based on the chosen example of the VE SBGM. As we outlined earlier in the case of the score matching method, the procedure is relatively easy and straightforward. It consists of the following steps:

**Training Procedure for VE SBGM**

1. Pick a datapoint $\mathbf{x}_0$.
2. Sample $\mathbf{x}_1 \sim \pi(\mathbf{x}) = \mathcal{N}(\mathbf{x}|0, \mathbf{I})$.
3. Sample $t \sim \text{Uniform}(0, 1)$.
4. Calculate $\mathbf{x}_t = \mathbf{x}_0 + \sqrt{\frac{1}{2 \ln \sigma}(\sigma^{2t} - 1)} \cdot \mathbf{x}_1$. This is a sample from $p_{0t}(\mathbf{x}_t|\mathbf{x}_0)$.
5. Evaluate the score model at $\mathbf{x}_t$ and $t$, $s_\theta(\mathbf{x}_t, t)$.
6. Calculate the score matching loss for a single sample, $\mathcal{L}_t(\theta) = \sigma_t^2 \|\mathbf{x}_1 - \sigma_t s_\theta(\mathbf{x}_t, t)\|^2$.
7. Update $\theta$ using a gradient-based method with $\nabla_\theta \mathcal{L}_t(\theta)$.

We repeat these seven steps for available training data until some stop criterion is met. Obviously, in practice, we use mini-batches instead of single datapoints.

In this training procedure, we use $-\sigma_t s_\theta(\mathbf{x}_t, t)$ on purpose because $-\sigma_t s_\theta(\mathbf{x}_t, t) = \epsilon_\theta(\mathbf{x}_t, t)$ and then the criterion $\sigma_t^2 \|\mathbf{x}_1 - \epsilon_\theta(\mathbf{x}_t, t)\|^2$ corresponds to diffusion-based models [4, 5]. Now, you see why we pushed for seeing diffusion-based models as dynamical systems!

#### 9.3.3.4 Sampling

After training the score model, we can finally generate! For that, we need to run backward Euler's method (or other ODE solvers, please remember that) that takes the following form for the VE PF-ODE:

$$\mathbf{x}_t = \mathbf{x}_{t+\Delta} + \left(\frac{1}{2}\sigma^{2(t+\Delta)}\left\{-\frac{1}{\sigma^{t+\Delta}}s_\theta(\mathbf{x}_{t+\Delta}, t+\Delta)\right\}\right)\cdot\Delta \tag{9.36}$$

$$= \mathbf{x}_{t+\Delta} - \left(\frac{1}{2}\sigma^{t+\Delta}s_\theta(\mathbf{x}_{t+\Delta}, t+\Delta)\right)\cdot\Delta \tag{9.37}$$

starting from $\mathbf{x}_1 \sim p_{01}(\mathbf{x}) = \mathcal{N}\left(\mathbf{x}|0, \frac{1}{2\ln\sigma}(\sigma^2-1)\mathbf{I}\right)$. Note that in the first line, we have the plus sign because the diffusion for the VE PF-ODE is $-\frac{1}{2}\sigma^{2t}$; therefore, the minus sign in backward Euler's method turns to plus. Maybe this is very obvious to you, my reader, but I always mess around with pluses and minuses, so I prefer to be very precise here.

### 9.3.4 Finally Some Code!

We have everything ready for implementing our VE SBGM! Similar to score matching, we transform data to be in $[-1, 1]$. As you will notice, this code shares many similarities to the code of score matching. But this is to be expected since they are conceptually very similar:

```
class SBGM(nn.Module):
    def __init__(self, snet, sigma, D, T):
        super(SBGM, self).__init__()

        print("SBGM by JT.")

        # sigma parameter
        self.sigma = torch.Tensor([sigma])

        # define the base distribution (multivariate Gaussian
    with the diagonal covariance)
        var = (1./(2.* torch.log(self.sigma))) * (self.sigma**2 -
     1.)
        self.base = torch.distributions.multivariate_normal.
    MultivariateNormal(torch.zeros(D), var * torch.eye(D))

        # score model
        self.snet = snet

        # time embedding (a single linear layer)
        self.time_embedding = nn.Sequential(nn.Linear(1, D), nn.
    Tanh())

        # other hyperparams
        self.D = D

        self.T = T

        self.EPS = 1.e-5

```

```
    def sigma_fun(self, t):
        # the sigma function (dependent on t), it is the std of the distribution
        return torch.sqrt((1./(2. * torch.log(self.sigma))) * (self.sigma**(2.*t) - 1.))

    def log_p_base(self, x):
        # the log-probability of the base distribution, p_1(x)
        log_p = self.base.log_prob(x)
        return log_p

    def sample_base(self, x_0):
        # sampling from the base distribution
        return self.base.rsample(sample_shape=torch.Size([x_0.shape[0]]))

    def sample_p_t(self, x_0, x_1, t):
        # sampling from p_0t(x_t|x_0)
        # x_0 ~ data, x_1 ~ noise
        x = x_0 + self.sigma_fun(t) * x_1

        return x

    def lambda_t(self, t):
        # the loss weighting
        return self.sigma_fun(t)**2

    def diffusion_coeff(self, t):
        # the diffusion coefficient in the SDE
        return self.sigma**t

    def forward(self, x_0, reduction='mean'):
        # =====
        # x_1 ~ the base distribution
        x_1 = torch.randn_like(x_0)
        # t ~ Uniform(0, 1)
        t = torch.rand(size=(x_0.shape[0], 1))  * (1. - self.EPS) + self.EPS

        # =====
        # sample from p_0t(x|x_0)
        x_t = self.sample_p_t(x_0, x_1, t)

        # =====
        # invert noise
        # NOTE: here we use the correspondence eps_theta(x,t) = -sigma*t score_theta(x,t)
        t_embd = self.time_embedding(t)
        x_pred = -self.sigma_fun(t) * self.snet(x_t + t_embd)

        # =====LOSS: Score Matching
        # NOTE: since x_pred is the predicted noise, and x_1 is noise, this corresponds to Noise Matching
```

```
        #       (i.e., the loss used in diffusion-based models by Ho et al.)
        SM_loss = 0.5 * self.lambda_t(t) * torch.pow(x_pred + x_1, 2).mean(-1)

        if reduction == 'sum':
            loss = SM_loss.sum()
        else:
            loss = SM_loss.mean()

        return loss

    def sample(self, batch_size=64):
        # 1) sample x_0 ~ Normal(0,1/(2log sigma) * (sigma**2 - 1))
        x_t = self.sample_base(torch.empty(batch_size, self.D))

        # Apply Euler's method
        # NOTE: x_0 - data, x_1 - noise
        #       Therefore, we must use BACKWARD Euler's method! This results in the minus sign!
        ts = torch.linspace(1., self.EPS, self.T)
        delta_t = ts[0] - ts[1]

        for t in ts[1:]:
            tt = torch.Tensor([t])
            u = 0.5 * self.diffusion_coeff(tt) * self.snet(x_t + self.time_embedding(tt))
            x_t = x_t - delta_t * u

        x_t = torch.tanh(x_t)
        return x_t

    def log_prob_proxy(self, x_0, reduction="mean"):
        # Calculate the proxy of the log-likelihood (see (Song et al., 2021))
        # NOTE: Here, we use a single sample per time step (this is done only for simplicity and speed);
        # To get a better estimate, we should sample more noise
        ts = torch.linspace(self.EPS, 1., self.T)

        for t in ts:
            # Sample noise
            x_1 = torch.randn_like(x_0)
            # Sample from p_0t(x_t|x_0)
            x_t = self.sample_p_t(x_0, x_1, t)
            # Predict noise
            t_embd = self.time_embedding(torch.Tensor([t]))
            x_pred = -self.snet(x_t + t_embd) * self.sigma_fun(t)
            # loss (proxy)
            if t == self.EPS:
                proxy = 0.5 * self.lambda_t(t) * torch.pow(x_pred + x_1, 2).mean(-1)
            else:
```

```
                proxy = proxy + 0.5 * self.lambda_t(t) * torch.
    pow(x_pred + x_1, 2).mean(-1)

        if reduction == "mean":
            return proxy.mean()
        elif reduction == "sum":
            return proxy.sum()
```

**Listing 9.2** A class for the VE PF-ODE model.

After running the code with an MLP-based scoring model and the following values of the hyperparameters $\sigma = 1.01$ and $T = 20$, we can expect results like those in Fig. 9.6.

I rarely comment on the results, especially since the dataset I use is oversimplistic (but on purpose so that you can run the code quickly!). However, I want to point out two things:

- The VE SBGM is not a very easy model to work with because it is pretty noisy. Inspecting the generated images gives a good feeling of potential issues. We run the ODE solver for 20 steps. However, SBGMs require typically hundreds or even thousands of steps.
- Take a look at the validation value of the proxy. It is a known fact that the convergence speed of SBGMs is rather slow.

### 9.3.5 There Is a Fantastic World of Score-Based Generative Models Out There!

**Other SBGMs** There are other classes of SBGMs besides variance exploding, namely [1]:

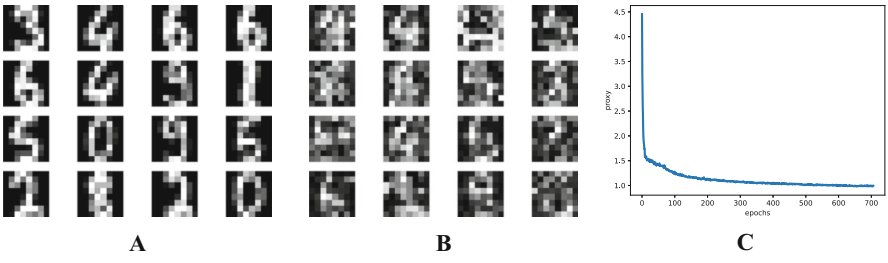



**Fig. 9.6** Results for the VE PF-ODE model: (**a**) A sample of real images. (**b**) A sample of generated images. (**c**) An example of the score matching loss calculated on the validation set.

- Variance preserving (VP): The drift is $f(\mathbf{x}, t) = -\frac{1}{2}\beta_t \mathbf{x}$, the diffusion is $g(t) = \sqrt{\beta_t}$, and the loss weighting is $\lambda_t = 1 - \exp\{-\int_0^t \beta_s \mathrm{d}s\}$, where $\beta_t$ is some function of time $t$.
- Sub-VP: The drift is $f(\mathbf{x}, t) = -\frac{1}{2}\beta_t \mathbf{x}$, the diffusion is $g(t) = \sqrt{\beta_t \left(1 - \exp\{-2\int_0^t \beta_s \mathrm{d}s\}\right)}$, and the loss weighting is $\lambda_t = \left(1 - \exp\{-\int_0^t \beta_s \mathrm{d}s\}\right)^2$, where $\beta_t$ is some function of time $t$.

There exist various versions of these models, especially there are different ways of defining $\lambda_t$ and other functions dependent on $t$ like $\sigma_t$ in VE and $\beta_t$ in VP. See [5] for an overview.

**Better solvers** As briefly mentioned earlier, one drawback of SBGMs is a large number of steps during sampling (i.e., the number of steps of an ODE solver). [22] presented specialized ODE solvers that could achieve great performance within $T = 10$ that was further improved to $T = 5$ in [23]! In general, a better-suited solver could be used to obtain better results, e.g., by using Heun's method [24].

**Other improvements** There are many ideas within the domain of score-based models! Here, I will name only a few:

- Using SBGMs in the latent space [25].
- In fact, it is possible to calculate the log-likelihood function for SBGMs in a similar manner to neural ODE [26]. Song et al. [20] showed that the log-likelihood function could be upper-bounded by some modification of the score matching loss.
- Using various tricks to improve the log-likelihood estimation like dequantization and importance weighting [27].
- In [28], a new class of models was proposed dubbed *consistency models*. The idea is to learn a model that could match noise to data in a single step.
- An extension of SBGMs to Riemannian manifolds was proposed by [29].

**There is a lot of work done!** There are many, many papers on SBGMs being published as we speak. Check out this webpage for an up-to-date overview of SBGMs:

https://scorebasedgenerativemodeling.github.io/

## 9.4 Flow Matching

### 9.4.1 A Different Perspective on Generative Models with ODEs: Continuous Normalizing Flows (CNFs)

#### 9.4.1.1 About ODEs, Again

Previously, we discussed how generative models could be defined through stochastic differential equations (SDEs) or, equivalently, corresponding probability flow ordinary differential equations (PF-ODEs). We showed that by solving SDEs/ODEs using a numerical solver like backward Euler's method, we obtain an iterative generative procedure of turning noise into data. However, there is a question of whether we need to first formulate an SDE and its PF-ODE equivalent or maybe we can take *any* ODE to define a generative model. I know you, my curious reader, and I feel you! We use SDEs because we know they are trained by score matching. If we take any ODE, how could we learn such generative models?

Let us be more concrete here. We recall the definition of an ODE:

$$\frac{\mathrm{d}\mathbf{x}_t}{\mathrm{d}t} = v(\mathbf{x}_t, t), \tag{9.38}$$

where the *vector field*, $v(\mathbf{x}_t, t)$, defines the dynamics. Parameterizing the vector field with a neural network with weights $\theta$, $v_\theta(\mathbf{x}_t, t)$, leads to a so-called **neural ODE** [26]. If we denote by $\mathbf{x}_0$ the initial condition for this neural ODE, e.g., noise, then by solving it, i.e., integrating over *time* $t$, we get the output (e.g., data):

$$\mathbf{x}_1 = \int_0^1 v_\theta(\mathbf{x}_t, t)\mathrm{d}t. \tag{9.39}$$

So far so good, but there is (almost) nothing new compared to score-based generative models where we match scores instead of distributions (i.e., an empirical distribution to a model). Could we formulate a likelihood-based training? The short answer is: Yes. Is it easy? Again, the short answer is *no*. But let us look into both answers more in detail.

#### 9.4.1.2 From the Continuity Equation (Conservation of Mass) to the Instantaneous Change of Variables

Again, sampling from an ODE, namely, integrating from $t = 0$ to $t = 1$ is not difficult once we have a model of the vector field. We can use Euler's method for that (or any other numerical solver). However, obtaining the model is problematic if we prefer

fitting a data distribution to a distribution induced by $v_\theta(x, t)$. After all, starting with a known distribution $\mathbf{x}_0 \sim \pi(\mathbf{x})$ like standard Gaussian and then solving the ODE yield another distribution! We can express this induced distribution analytically using the **continuity equation**. Here comes some math (and even physics!), so buckle up and let us dive in.

Imagine for a second that probability is a mass (I always think of clay, but it could be water if you prefer), something we can touch with our fingers. Now, let us visualize a pipe of the same cross-section volume across its length in which our mass (e.g., water) flows. At each moment of time, we have some *flux* of this mass, $f_t$, i.e., our (probability) mass is moved according to the vector field (or velocity), $f_t(\mathbf{x}_t) = p_t(\mathbf{x}_t)v(\mathbf{x}_t, t)$. Since we talk about probability mass (or water flowing through the pipe of the same volume of cross sections everywhere), the mass is conserved, i.e., no new mass (water) (dis)appears (no leaking or pouring in). Mathematically, it means that the change of the mass $\frac{\partial p_t(\mathbf{x}_t)}{\partial t}$ plus the change of the flux volume in all directions (a.k.a. the divergence of the flux) is constant (i.e., the mass is conserved):

$$\frac{\partial p_t(\mathbf{x}_t)}{\partial t} + \operatorname{div}\left(p_t(\mathbf{x}_t)v(\mathbf{x}_t, t)\right) = 0, \tag{9.40}$$

where $\operatorname{div}(\cdot)$ is the divergence defined as follows: $\operatorname{div}(V(x_1, \ldots, x_D)) = \sum_{d=1}^{D} \frac{\partial V_d(x_1,\ldots,x_D)}{\partial x_d}$, i.e., the sum of first derivatives of $V$ over all variables separately.

**A Side Note**

The trace of the Jacobian matrix is the divergence of the vector field! For a two-dimensional space and a vector field $V(x_1, x_2)$, $\operatorname{div}(V(x_1, x_2)) = \frac{\partial V_1(x_1,x_2)}{\partial x_1} + \frac{\partial V_2(x_1,x_2)}{\partial x_2} = \operatorname{Tr}\left(\frac{\partial V_i(x_1,x_2)}{\partial x_i}\right)$, where $\frac{\partial V_i(x_1,x_2)}{\partial x_i} = \begin{bmatrix} \frac{\partial V_1}{\partial x_1} & \frac{\partial V_1}{\partial x_2} \\ \frac{\partial V_2}{\partial x_1} & \frac{\partial V_2}{\partial x_2} \end{bmatrix}$ is the Jacobian matrix and $\operatorname{Tr}(\mathbf{A}) = \sum_i \mathbf{A}_{ii}$ is the trace of a matrix $\mathbf{A}$.

It turns out that applying identities of vector calculus and the properties of the divergence allows us to write the continuity equation using the logarithm of the probability distribution (a.k.a. **the instantaneous change of variables** [26]):

$$\frac{\mathrm{d} \ln p(\mathbf{x}_t)}{\mathrm{d}t} + \operatorname{Tr}\left(\frac{\partial v(\mathbf{x}_t, t)}{\partial \mathbf{x}_t}\right) = 0. \tag{9.41}$$

**A Side Note**

I know you, my curious reader, you dislike being told about something without proof. It is a side note, so if you do not have time for it, just skip it. But I find this pretty cool and it will be also helpful in our further discussion. Ok, so what do we need? First, a general property, $\frac{\partial \ln p_t}{\partial t} = \frac{1}{p_t}\frac{\partial p_t}{\partial t}$, or $p_t \frac{\partial \ln p_t}{\partial t} = \frac{\partial p_t}{\partial t}$. Second, we have $\text{div}(p_t v) = \langle \nabla_{\mathbf{x}} p_t, v\rangle + p_t \text{div}(v)$; hence, $\frac{\text{div}(p_t v)}{p_t} = \langle \nabla_{\mathbf{x}} \ln p_t, v\rangle + \text{div}(v)$. Third, we saw already that the divergence of the vector field equals its trace. Now, we can plug these three facts into the continuity equation:

$$\frac{\partial p_t(\mathbf{x}_t)}{\partial t} + \text{div}\left(p_t(\mathbf{x}_t) v(\mathbf{x}_t, t)\right) = 0 \tag{9.42}$$

$$\frac{1}{p_t(\mathbf{x}_t)}\frac{\partial p_t(\mathbf{x}_t)}{\partial t} + \frac{1}{p_t(\mathbf{x}_t)}\text{div}\left(p_t(\mathbf{x}_t) v(\mathbf{x}_t, t)\right) = 0 \tag{9.43}$$

$$\frac{\partial \ln p_t(\mathbf{x}_t)}{\partial t} + \langle \nabla_{\mathbf{x}} \ln p_t, v\rangle + \text{div}\left(v(\mathbf{x}_t, t)\right) = 0 \tag{9.44}$$

$$\langle \nabla_{\mathbf{x}} \ln p_t, v\rangle = -\frac{\partial \ln p_t(\mathbf{x}_t)}{\partial t} - \text{div}\left(v(\mathbf{x}_t, t)\right) \tag{9.45}$$

Next, calculating the total derivative of $\ln p_t(\mathbf{x}_t)$ and plugging in the above equation for $\langle \nabla_{\mathbf{x}} \ln p_t, v\rangle$, we get the following [30]:

$$\frac{\text{d} \ln p(\mathbf{x}_t)}{\text{d}t} = \langle \frac{\partial \ln p(\mathbf{x}_t)}{\partial t}, \frac{\partial t}{\partial t}\rangle + \langle \nabla_{\mathbf{x}} \ln p(\mathbf{x}_t), \frac{\partial \mathbf{x}_t}{\partial t}\rangle \tag{9.46}$$

$$= \frac{\partial \ln p(\mathbf{x}_t)}{\partial t} + \langle \nabla_{\mathbf{x}} \ln p(\mathbf{x}_t), \frac{\partial \mathbf{x}_t}{\partial t}\rangle \tag{9.47}$$

$$= \frac{\partial \ln p(\mathbf{x}_t)}{\partial t} + \langle \nabla_{\mathbf{x}} \ln p(\mathbf{x}_t), v(\mathbf{x}_t, t)\rangle \tag{9.48}$$

$$= \frac{\partial \ln p(\mathbf{x}_t)}{\partial t} - \frac{\partial \ln p_t(\mathbf{x}_t)}{\partial t} - \text{div}\left(v(\mathbf{x}_t, t)\right) \tag{9.49}$$

$$= -\text{div}\left(v(\mathbf{x}_t, t)\right) \tag{9.50}$$

$$= -\text{Tr}\left(\frac{\partial v(\mathbf{x}_t, t)}{\partial \mathbf{x}_t}\right) \tag{9.51}$$

Then, by integrating across time, we can compute the total change in log-density as follows:

$$\int_0^1 \left( \frac{\mathrm{d} \ln p(\mathbf{x}_t)}{\mathrm{d}t} + \mathrm{Tr}\left( \frac{\partial v(\mathbf{x}_t, t)}{\partial \mathbf{x}_t} \right) \right) \mathrm{d}t = 0 \tag{9.52}$$

$$\ln p(\mathbf{x}_1) - \ln \pi(\mathbf{x}_0) + \int_0^1 \mathrm{Tr}\left( \frac{\partial v(\mathbf{x}_t, t)}{\partial \mathbf{x}_t} \right) \mathrm{d}t = 0 \tag{9.53}$$

$$\ln p(\mathbf{x}_1) = \ln \pi(\mathbf{x}_0) - \int_0^1 \mathrm{Tr}\left( \frac{\partial v(\mathbf{x}_t, t)}{\partial \mathbf{x}_t} \right) \mathrm{d}t. \tag{9.54}$$

Why do we bother to calculate everything as log-probabilities? Because the last line is a continuous version of the change of variables used for normalizing flows! Here, we have the integral over time of the trace of the Jacobian matrix instead of the sum of the log-determinants of the Jacobian matrix. Therefore, training neural ODEs is similar to training normalizing flows but with continuous time. As a result, neural ODEs in this context are referred to as **continuous normalizing flows** (CNFs).

#### 9.4.1.3 Calculating the Log-Likelihood for CNFs

However, unlike in discrete time normalizing flows, we do not require *invertibility* of $v$; thus, for given datapoint $\mathbf{x}_1$, typically, we cannot simply invert the transformation to obtain $\mathbf{x}_0$. However, under pretty mild conditions (namely, $v$ and its first derivative are Lipschitz continuous, e.g., for a neural net with Lipschitz continuous activation functions like SELU or SiLU, among others), we can uniquely solve the following problem [31]:

$$\begin{bmatrix} \mathbf{x}_0 \\ \ln p_1(\mathbf{x}_1) - \ln \pi(\mathbf{x}_0) \end{bmatrix} = \int_1^0 \begin{bmatrix} v_\theta(\mathbf{x}_t, t) \\ -\mathrm{Tr}\left( \frac{\partial v_\theta(\mathbf{x}_t, t)}{\partial \mathbf{x}_t} \right) \end{bmatrix} \mathrm{d}t, \tag{9.55}$$

with the following initial conditions:

$$\begin{bmatrix} \mathbf{x}_1 \\ \ln p_1(\mathbf{x}_{data}) - \ln p_1(\mathbf{x}_1) \end{bmatrix} = \begin{bmatrix} \mathbf{x}_{data} \\ 0, \end{bmatrix} \tag{9.56}$$

in which $\mathbf{x}_1$ is a datapoint $\mathbf{x}_{data}$ and the difference in log-probability is zero. Note that we solve the problem in the reverse order, namely, from data $\mathbf{x}_1$ to noise $\mathbf{x}_0$.

**Calculating the Log-Likelihood Function**

To sum up, we need to carry out the following steps:

1. Take a datapoint $\mathbf{x}_1 = \mathbf{x}_{data}$.
2. Solve the problem in (9.55) by applying a solver to find $\mathbf{x}_0$ and keeping track of traces over time.
3. Calculate the log-likelihood by adding $\ln \pi(\mathbf{x}_0)$ to the sum of negative traces $-\int_0^1 \mathrm{Tr}\left(\frac{\partial v_\theta(\mathbf{x}_t, t)}{\partial \mathbf{x}_t}\right) \mathrm{d}t$.

Now, we can backpropagate through a solver and *viola*! Or really? What about the complexity of this whole procedure? At first glance, it seems very expensive.

#### 9.4.1.4 Hutchinson's Trace Estimator

If you remember correctly, my curious reader, the problem with normalizing flows was about calculating the log-determinant of the Jacobian matrix of size $D \times D$, which in general case costs $O(D^3)$. Computing the trace requires $O(D^2)$ since we need the sum of the diagonal, but each entry in the diagonal requires a separate forward propagation, thus, the quadratic complexity. It is better than in the discrete case, but there is another caveat: We need to backpropagate through a numerical solver! No free lunch, I am afraid. Chen et al. [26] proposed to use the *adjoint sensitivity method*, which could be seen as a version of backpropagation with continuous time. Then, the neural ODE is trained by maximizing the log-likelihood $\ln p(\mathbf{x}_1)$. However, it requires running the numerical method to solve the ODE and then backpropagating through it for each new datapoint. This is a very costly operation!

As a result, we need to look for improvements to cut costs everywhere we can. One trick we can apply is about calculating the trace. By utilizing **Hutchinson's trace estimator** [31], the quadratic complexity is decreased to $O(D)$, and it is relatively easy to calculate for any square matrix $\mathbf{A}$, namely:

$$\mathrm{Tr}(\mathbf{A}) = \mathbb{E}_\epsilon \left[\epsilon^\top \mathbf{A} \epsilon\right], \tag{9.57}$$

where $\epsilon$ follows a distribution with zero mean and unit variance, e.g., $\epsilon \sim \mathcal{N}(0, \mathbf{I})$. For a specific $\epsilon$, the product of $\mathbf{A}\epsilon$ could be calculated in a single forward pass and it is "backpropagatable"; therefore, we can estimate the trace by taking $M$ Monte Carlo samples:

$$\mathrm{Tr}(\mathbf{A}) \approx \frac{1}{M} \sum_{m=1}^{M} \epsilon_m^\top \mathbf{A} \epsilon_m. \tag{9.58}$$

In practice, we take $M = 1$, namely, a single sample of $\epsilon$ for every newly coming datapoint. This is a noisy estimate, obviously; however, it is unbiased. As a result, during training with a stochastic gradient-based method, it does not matter too much.

Eventually, we obtain a procedure that is $O(D)$ plus the cost of running the adjoint sensitivity method (a specific numerical solver). Overall, not bad, but far from fantastic. We do not even provide a code here, because scaling up CNFs is a known problem. Is there any alternative then? Can we do better? Of course, my curious reader, of course we can!

## 9.4.2 Going with the Flow: Flow Matching

### 9.4.2.1 The Idea

Let us take another look at the ODE we introduced earlier:

$$\frac{\mathrm{d}\mathbf{x}_t}{\mathrm{d}t} = v(\mathbf{x}_t, t). \tag{9.59}$$

Additionally to this ODE, we assume a *known* distribution $q_0(\mathbf{x})$ (e.g., the standard Gaussian) and a data distribution $q_1(\mathbf{x})$.

**A Side Note**

To stay consistent with the flow matching literature, we now go from noise ($t = 0$) to data ($t = 1$), which defines the forward dynamics and, thus, the generative process—unlike the diffusion models (and score-based models), where the time goes in the other direction, i.e., from data to noise.

We know from our discussion on CNFs above that the distribution defined at any moment $t$ is characterized by the continuity equation. And, moreover, by applying the instantaneous change of variables, we can find a solution, i.e., a probability distribution. However, all of this sounds quite complicated, and as discussed earlier, it results in pretty computationally heavy training. What can we do then? Or a different question is what we could do if we *knew* the vector field $v(\mathbf{x}_t, t)$ and distributions $p_t(\mathbf{x})$. How could we train our model then? And what would be our model? Do you remember the score matching approach? Take a look at the denoising score matching loss again. What if we apply a similar approach here, namely, instead of looking for a distribution, we find a model of the vector field $v_\theta(\mathbf{x}_t, t)$. Similar to score matching, we solve the regression problem in the following form:

$$\ell_{FM}(\theta) = \mathbb{E}_{t\sim U(0,1),\mathbf{x}_t\sim p_t(\mathbf{x})}\left[\|v_\theta(\mathbf{x}_t, t) - v(\mathbf{x}_t, t)\|^2\right], \tag{9.60}$$

instead of looking for a distribution like in CNFs. In plain words, for any time $t$ sampled uniformly at random, we sample $\mathbf{x}_t$ from the distribution $p_t(\mathbf{x})$ (we assume we *know* it!) and aim at minimizing the difference between the model $v_\theta(\mathbf{x}_t, t)$ and the *real* vector field $v(\mathbf{x}_t, t)$ (we assume we *know* it!). We refer to this objective as **flow matching** (FM).

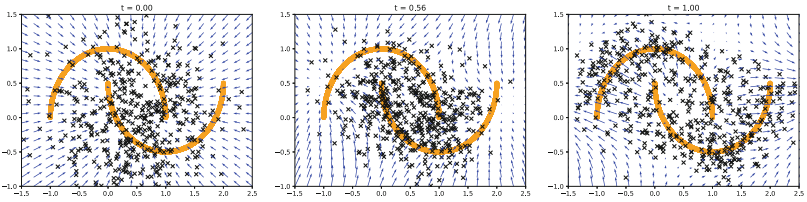



**Fig. 9.7** An example of how a model of the vector field (blue arrows) changes over time around datapoints (orange two moons).

Why would this work? For a very simple reason, if $\ell_{FM}(\theta) = 0$, i.e., our model perfectly imitates the real vector field, we can transform any noise distribution to the data distribution! Why? Because the vector field *pushes* points toward the data distribution over time. Take a look at Fig. 9.7 where blue arrows (the vector field) indicate how points should *evolve* over time from a noise distribution $q_0(\mathbf{x})$ (e.g., the standard Gaussian) to the data distribution $q_1(\mathbf{x})$ (orange half-moons in Fig. 9.7).

The second question is why this is so great. The answer is (again) simple: This is the regression problem, the mean squared error loss! Nothing complicated, nothing tricky, a well-behaved convex loss. One can run autograd and use any deep learning library to implement that. Fantastic!

But here comes the "but": We do not know $p_t(\mathbf{x})$ and $v(\mathbf{x}_t, t)$. I know you, my curious reader, you kept it in your head throughout the whole discussion so far. You are polite, so you waited patiently. I would already say "bollocks," the assumption that we know both the vector field and the distributions $p_t(\mathbf{x})$ are simply bollocks. What we can do about it then, you ask? In the following, we will show how to deal with that!

#### 9.4.2.2 Conditional Flow Matching

First, let us consider a modified problem in which we introduce additional variables $\mathbf{z}$ sampled from a given distribution $q(\mathbf{z})$. The *conditional* ODE takes the following form:

$$\frac{\mathrm{d}\mathbf{x}_t}{\mathrm{d}t} = v(\mathbf{x}_t, t; \mathbf{z}). \tag{9.61}$$

For now, please think of this problem as a proxy for the unconditional ODE introduced before. In general, it is typically easier to work with conditional problems as long as the conditioning information is relevant. Regarding $\mathbf{z}$, we can think of it as extra information like data $\mathbf{x}_1$ or anything else like a class label, a piece of text, an audio signal, or an additional image. Then, since we have to also sample $\mathbf{z}$s from some distribution $q(\mathbf{z})$, the **conditional flow matching** (CFM) loss can be defined as follows:

$$\ell_{CFM}(\theta) = \mathbb{E}_{t\sim U(0,1),\mathbf{x}_t\sim p_t(\mathbf{x}|\mathbf{z}),\mathbf{z}\sim q(\mathbf{z})}\left[\|v_\theta(\mathbf{x}_t, t) - v(\mathbf{x}_t, t; \mathbf{z})\|^2\right], \tag{9.62}$$

where we still use an unconditional model of the conditional vector field.

In the CFM loss, we need to define the conditional distribution at every $t$, $p_t(\mathbf{x}|\mathbf{z})$, and the *real* vector field is conditioned on $\mathbf{z}$, $v(\mathbf{x}_t, t; \mathbf{z})$. At first, the CFM problem does not seem to be very useful regarding the FM problem. Why would adding conditioning help to learn a model that should work for the unconditional case? As proved in [32, 33], both losses are equal up to a constant independent of $\theta$, and, thus, their gradients are equal!

Theorem [32]

***Theorem 1*** If $p_t(\mathbf{x}) > 0$ for all $\mathbf{x} \in \mathbb{R}^D$ amd $t \in [0, 1]$, then, up to a constant independent of $\theta$, $\ell_{FM}$ and $\ell_{CFM}$ are equal, and hence $\nabla_\theta \ell_{FM}(\theta) = \nabla_\theta \ell_{CFM}(\theta)$. □

This is a marvelous result! It means that the model $v_\theta(\mathbf{x}, t)$ trained with the conditional version of the loss, but which is unconditional, coincides with the solution of the unconditional flow matching problem. Fantastic! One problem is gone; we can use CFM instead of FM! Alright, but now we have another problem, namely, what this conditioning $\mathbf{z}$ should be and what is its distribution $q(\mathbf{z})$. Fortunately, there are multiple options (see, for example, [33]); here, we focus on two of those:

1. In [32] CNF, $\mathbf{z}$ is a datapoint $\mathbf{x}_1$, and thus, $q(\mathbf{z}) = q_1(\mathbf{z})$; in other words, $q(\mathbf{z})$ is the data distribution.
2. In [33] CNF (a.k.a. independent CFM (iCFM)), $\mathbf{z}$ is a pair of noise and data, $\mathbf{z} = (\mathbf{x}_0, \mathbf{x}_1)$, sampled independently from each other, i.e., $q_0(\mathbf{z}) = q(\mathbf{x}_0)\, q_1(\mathbf{x}_1)$.

An extension of iCFM is sampling $\mathbf{z} = (\mathbf{x}_0, \mathbf{x}_1)$ by solving the optimal transport problem. Interested readers (yes, you!) should check out [33] for further details.

#### 9.4.2.3 Conditional Probability Paths

Now, we know that we can consider the conditional flow matching problem as the perfect proxy for the unconditional flow matching problem. However, the last piece of the puzzle is how to obtain conditional distributions $p_t(\mathbf{x}|\mathbf{z})$ a.k.a. (conditional) probability paths. We know from our discussions on CNFs that the continuity equation allows us to calculate the probability path. But for that, we need to know the vector field. It is a classical example of circular reasoning. How to avoid it?

My curious reader, so far you have learned a lot about AI, and you probably can guess already what is the simplest approach we can take here. Do I hear properly? Do you whisper Gaussians? Yes, indeed! Let us consider the form of $p_t(\mathbf{x}|\mathbf{z})$ first, and then, from that, we will derive the vector field $v(\mathbf{x}, t; \mathbf{z})$. This is probably a limiting factor of the whole approach, but dang, we want to calculate something, and is there a better candidate than Gaussian? Let us consider the conditional probability path of the following form:

$$p_t(\mathbf{x}|\mathbf{z}) = \mathcal{N}(\mathbf{x}|\mu(\mathbf{z}, t), \sigma^2(\mathbf{z}, t)\mathbf{I}), \tag{9.63}$$

which is a Gaussian distribution with the mean function $\mu(\mathbf{z}, t)$ and a diagonal covariance matrix with the standard deviation function $\sigma(\mathbf{z}, t)$. In general, there is no unique ODE that generates these distributions. However, the following theorem shows that there is a unique vector field that leads to those!

Theorem [32]

***Theorem 2*** The unique vector field with initial conditions $p_0(\mathbf{x}) = \mathcal{N}(\mu_0, \sigma_0^2\mathbf{I})$ that generate $p_t(\mathbf{x}|\mathbf{z}) = \mathcal{N}(\mathbf{x}|\mu(\mathbf{z}, t), \sigma^2(\mathbf{z}, t)\mathbf{I})$ has the following form:

$$v(\mathbf{x}, t; \mathbf{z}) = \frac{\sigma^{'}(\mathbf{z}, t)}{\sigma(\mathbf{z}, t)}(\mathbf{x} - \mu(\mathbf{z}, t)) + \mu^{'}(\mathbf{z}, t), \tag{9.64}$$

where $\sigma^{'}(\mathbf{z}, t)$ and $\mu^{'}(\mathbf{z}, t)$ denote the time derivates of $\sigma(\mathbf{z}, t)$ and $\mu(\mathbf{z}, t)$, respectively.

Ok, we should stop and think of it for a while, maybe read it again. You got it? Yes, this theorem gives us an incredible result! If we consider a class of conditional probability paths in the form of Gaussians, we can **analytically** calculate the conditional vector field as long as the means and the standard deviations are differentiable. In other words, we have a general prescription for flow matching! First, we define means and standard deviations as functions of $\mathbf{z}$ and time $t$. Second, for given forms of the means and the standard deviations, we calculate the conditional vector field. Third, we optimize the conditional flow matching loss for a given vector field model. That is all, as simple as it is. To get a better understanding of how it works, we look into two specific forms of conditional flow matching, namely:

- **Lipman et al. CFM** (we refer to it as `fm` later on in the code): We take $\mathbf{z} \equiv \mathbf{x}_1$ to be data sampled from the data distribution, $\mathbf{x}_1 \sim q_1(\mathbf{x})$. Then, we define the mean and the standard deviation functions as follows:

$$\mu(\mathbf{z}, t) = t\mathbf{x}_1, \tag{9.65}$$

$$\sigma(\mathbf{z}, t) = t\sigma_{const} - t + 1, \tag{9.66}$$

  where $\sigma_{const} > 0$ is a smoothing constant. As a result, we obtain the following conditional probability path and the conditional vector field:

$$p_t(\mathbf{x}|\mathbf{z}) = \mathcal{N}\left(\mathbf{x}|t\mathbf{x}_1, (t\sigma_{const} - t + 1)^2\mathbf{I}\right), \tag{9.67}$$

$$v(\mathbf{x}, t; \mathbf{z}) = \frac{\mathbf{x}_1 - (1 - \sigma_{const})\mathbf{x}}{1 - (1 - \sigma_{const})t}. \tag{9.68}$$

  To obtain the analytical form of $v(\mathbf{x}, t; \mathbf{z})$, we need to apply the theorem presented above. It turns out that we get a probability path from the standard Gaussian distribution, $p_0(\mathbf{x}) = \mathcal{N}(\mathbf{x}|0, \mathbf{I})$, to a Gaussian distribution centered at a datapoint with standard deviation $\sigma_{const}$, $p_1(\mathbf{x}) = \mathcal{N}(\mathbf{x}|\mathbf{x}_1, \sigma_{const}^2\mathbf{I})$ [32].

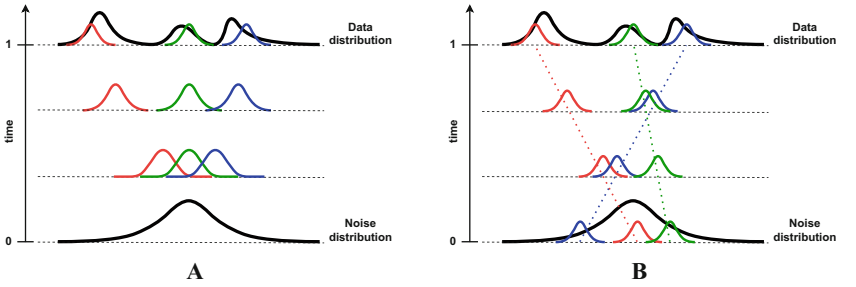



**Fig. 9.8** A schematic visualization of how flow matching approaches work: (**a**) An example of Lipman et al. CNF. (**b**) An example of Tong et al. iCNF. The dotted line indicates the interpolation between noise and data.

- **Tong et al. iCFM** We consider $\mathbf{z} \equiv (\mathbf{x}_0, \mathbf{x}_1)$ and $q(\mathbf{z}) = q_0(\mathbf{x}_0)q_1(\mathbf{x}_1)$. Next, we choose the following means and standard deviations:

$$\mu(\mathbf{z}, t) = t\mathbf{x}_1 + (1-t)\mathbf{x}_0, \tag{9.69}$$

$$\sigma(\mathbf{z}, t) = \sigma_{const}, \tag{9.70}$$

  where $\sigma_{const} > 0$ is a smoothing constant. The mean function is an *interpolation* between noise and data since $t \in [0, 1]$. The resulting conditional probability path and the conditional vector fields are the following:

$$p_t(\mathbf{x}|\mathbf{z}) = \mathcal{N}\left(\mathbf{x}|t\mathbf{x}_1 + (1-t)\mathbf{x}_0, \sigma^2_{const}\mathbf{I}\right), \tag{9.71}$$

$$v(\mathbf{x}, t; \mathbf{z}) = \mathbf{x}_1 - \mathbf{x}_0. \tag{9.72}$$

Interestingly, the vector field results in a difference between a datapoint and a sampled noise. This comes from the fact that we assume a fixed standard deviation in the probability path and, after applying Theorem 2, we end up with the derivate of the mean function only. Tong et al. [33] showed (see Proposition 3.3 therein) that the boundary distributions are $p_0(\mathbf{x}) = q_0(\mathbf{x}) * \mathcal{N}(\mathbf{x}|0, \sigma^2_{const}\mathbf{I})$ and $p_1(\mathbf{x}) = q_1(\mathbf{x}) * \mathcal{N}(\mathbf{x}|0, \sigma^2_{const}\mathbf{I})$, where $*$ denotes the convolution operator. For instance, if we take $q_0(\mathbf{x}) = \mathcal{N}(\mathbf{x}|0, \mathbf{I})$, then $p_0(\mathbf{x}) = \mathcal{N}(\mathbf{x}|0, (\sigma^2_{const} + 1)\mathbf{I})$. However, $q_0(\mathbf{x})$ could be any distribution, not only Gaussian. For $q_1(\mathbf{x})$ being the data distribution, the other boundary distribution is $p_1(\mathbf{x}) = \mathcal{N}(\mathbf{x}|\mathbf{x}_1, \sigma^2_{const}\mathbf{I})$ that is the same as in the case of the Lipman et al. CFM.

The differences between Lipman et al. CFM and Tong et al. iCFM are rather subtle. However, these subtle differences lead to different behavior of probability paths. In Fig. 9.8, examples of these two CFMs are presented. Lipman at al. CFM starts with the standard Gaussian that evolves over time to a small Gaussian (i.e., standard deviation decreases) in the data space (see Fig. 9.8a). Tong et al. CFM, on the other hand, defines a small Gaussian that is *moved* over time to the data space (see Fig. 9.8b).

#### 9.4.2.4 Training

We have all the components necessary to define the training of vector field models. The general algorithm is defined as follows:

**Training of FM**

1. Sample $t \sim \text{Uniform}(0, 1)$.
2. Sample $\mathbf{z} \sim q(\mathbf{z})$.
3. Calculate $\mu_t(\mathbf{z})$ and $\sigma_t(\mathbf{z})$.
4. Sample $\mathbf{x}_t \sim \mathcal{N}(\mathbf{x}|\mu(\mathbf{z}, t), \sigma(\mathbf{z}, t))$.
5. Calculate the vector field $v(\mathbf{x}_t, t; \mathbf{z})$.
6. Calculate loss $\ell_{CFM}(\theta) = \|v_\theta(\mathbf{x}_t, t) - v(\mathbf{x}_t, t; \mathbf{z})\|^2$.
7. Update parameters: $\theta \leftarrow \text{Update}(\theta, \nabla_\theta \ell_{CFM}(\theta))$.

Depending on the choice of the mean function and the standard deviation function, we get a specific form of the vector field. Eventually, we obtain particular formulas in steps 3 and 5, while all other steps remain the same.

#### 9.4.2.5 Sampling from FM

Let me remind you, my curious reader, that we learn a model of the vector field. Once we have it, sampling is straightforward, namely:

**Sampling**

1. Sample $\mathbf{x} \sim q_0(\mathbf{x})$.
2. Run forward Euler method until $t = 1$ and with a step size $\Delta$:

$$\mathbf{x}_{t+\Delta} = \mathbf{x}_t + v_\theta(\mathbf{x}_t, t)\,\Delta.$$

Please note that in the case of flow matching, unlike in score-based generative models, we assume that time flows from $t = 0$ (i.e., noise) to $t = 1$ (i.e., data). As a result, once the model of the vector field is trained, we run the forward Euler method, not the backward Euler method like in score-based generative models.

#### 9.4.2.6 Calculating the Log-Likelihood Function

In the discussion on CNFs, we showed that calculating the (log-)likelihood function is possible. Since flow matching does not require the log-likelihood function during training, we did not care about it so far. However, especially for evaluation but also in many practical applications, being able to calculate the value of the log-likelihood function is beneficial. The full derivation of the log-likelihood function for flow

matching is presented in Appendix C of [32]. The main idea is similar to CNFs, and to make calculations practical, we use Hutchinson's trace estimator, which yields:

$$\ln p_1(\hat{\mathbf{x}}_1) \approx \ln p_0(\hat{\mathbf{x}}_0) - f_0, \tag{9.73}$$

where $\hat{\mathbf{x}}_1$ is a datapoint, $\hat{\mathbf{x}}_0$ is the noise that corresponds to $\hat{\mathbf{x}}_1$, and $f_0$ is an approximation of the trace of the vector field Jacobian. Note that we have only $\hat{\mathbf{x}}_1$, but we can obtain $\hat{\mathbf{x}}_0$ and $f_0$ by running the following procedure [32]:

**Calculating the Log-Likelihood for FM**

1. For given datapoint $\hat{\mathbf{x}}_1$, set the initial conditions:
$$\begin{bmatrix} \phi_1 \\ f_1 \end{bmatrix} = \begin{bmatrix} \hat{\mathbf{x}}_1 \\ 0 \end{bmatrix}.$$
2. Define the following ODE:
$$\frac{\mathrm{d}}{\mathrm{d}s} \begin{bmatrix} \phi_{1-s} \\ f_{1-s} \end{bmatrix} = \begin{bmatrix} -v_\theta(\phi_{1-s}, 1-s) \\ \epsilon^\top \nabla_\phi v_\theta(\phi_{1-s}, 1-s)\epsilon \end{bmatrix}$$
where $\epsilon \sim \mathcal{N}(0, \mathbf{I})$.
*Note: We calculate $\nabla_\phi v_\theta(\phi_{1-s}, 1-s)$ by using autograd.*
3. Solve the ODE in step 2 by running the backward Euler method.
*Note: Here, we need to go from $t = 1$ to $t = 0$, thus, the backward Euler.*
4. Output the result:
$$\begin{bmatrix} \phi_0 \\ f_0 \end{bmatrix} = \begin{bmatrix} \hat{\mathbf{x}}_0 \\ \hat{c} \end{bmatrix}.$$

The outputs of this procedure are then plugged into the equation (9.73) that yields:

$$\ln p_1(\hat{\mathbf{x}}_1) \approx \ln p_0(\hat{\mathbf{x}}_0) - \hat{c}. \tag{9.74}$$

Please remember, my curious reader, that this is an approximation due to the trace estimator. However, the estimator is unbiased, and the variance can be reduced by running this procedure $M$ times for a given datapoint and then averaging.

### 9.4.3 Calculating the Log-Likelihood Function

We outlined how to implement training and sampling (generation) using flow matching. The code is pretty straightforward and is outlined in the following code snippet. We picked the standard Gaussian noise distribution in our implementation, but please be aware that any other distribution would work:

```
class FlowMatching(nn.Module):
    def __init__(self, vnet, sigma, D, T, stochastic_euler=False,
     prob_path="icfm"):
        super(FlowMatching, self).__init__()

        print('Flow Matching by JT.')

        self.vnet = vnet
        self.time_embedding = nn.Sequential(nn.Linear(1, D), nn.
    Tanh())

        # other params
        self.D = D
        self.T = T
        self.sigma = sigma
        self.stochastic_euler = stochastic_euler

        assert prob_path in ["icfm", "fm"], f"Error: The
    probability path could be either Independent CFM (icfm) or
    Lipman's Flow Matching (fm) but {prob_path} was provided."
        self.prob_path = prob_path

        self.PI = torch.from_numpy(np.asarray(np.pi))

    def log_p_base(self, x, reduction='sum', dim=1):
        log_p = -0.5 * torch.log(2. * self.PI) - 0.5 * x**2.
        if reduction == 'mean':
            return torch.mean(log_p, dim)
        elif reduction == 'sum':
            return torch.sum(log_p, dim)
        else:
            return log_p

    def sample_base(self, x_1):
        # Gaussian base distribution
        if self.prob_path == "icfm":
            return torch.randn_like(x_1)
        elif self.prob_path == "fm":
            return torch.randn_like(x_1)
        else:
            return None

    def sample_p_t(self, x_0, x_1, t):
        if self.prob_path == "icfm":
            mu_t = (1. - t) * x_0 + t * x_1
            sigma_t = self.sigma
        elif self.prob_path == "fm":
            mu_t = t * x_1
            sigma_t = t * self.sigma - t + 1.

        x = mu_t + sigma_t * torch.randn_like(x_1)

        return x

```

```
    def conditional_vector_field(self, x, x_0, x_1, t):
        if self.prob_path == "icfm":
            u_t = x_1 - x_0
        elif self.prob_path == "fm":
            u_t = (x_1 - (1. - self.sigma) * x) / (1. - (1. - self.sigma) * t)

        return u_t

    def forward(self, x_1, reduction='mean'):
        # =====Flow Matching
        # =====
        # z ~ q(z), e.g., q(z) = q(x_0) q(x_1), q(x_0) = base, q(x_1) = empirical
        # t ~ Uniform(0, 1)
        x_0 = self.sample_base(x_1)  # sample from the base distribution (e.g., Normal(0,I))
        t = torch.rand(size=(x_1.shape[0], 1))

        # =====
        # sample from p(x|z)
        x = self.sample_p_t(x_0, x_1, t)  # sample independent rv

        # =====
        # invert interpolation, i.e., calculate vector field v(x, t)
        t_embd = self.time_embedding(t)
        v = self.vnet(x + t_embd)

        # =====
        # conditional vector field
        u_t = self.conditional_vector_field(x, x_0, x_1, t)

        # =====LOSS: Flow Matching
        FM_loss = torch.pow(v - u_t, 2).mean(-1)

        # Final LOSS
        if reduction == 'sum':
            loss = FM_loss.sum()
        else:
            loss = FM_loss.mean()

        return loss

    def sample(self,  batch_size=64):
        # Euler method
        # sample x_0 first
        x_t = self.sample_base(torch.empty(batch_size, self.D))

        # then go step-by-step to x_1 (data)
        ts = torch.linspace(0., 1., self.T)
        delta_t = ts[1] - ts[0]

        for t in ts[1:]:
```

```
            t_embedding = self.time_embedding(torch.Tensor([t]))
            x_t = x_t + self.vnet(x_t + t_embedding) * delta_t
            # Stochastic Euler method
            if self.stochastic_euler:
                x_t = x_t + torch.randn_like(x_t) * delta_t

        x_final = torch.tanh(x_t)
        return x_final

    def log_prob(self, x_1, reduction='mean'):
        # backward Euler (see Appendix C in Lipman's paper)
        ts = torch.linspace(1., 0., self.T)
        delta_t = ts[1] - ts[0]

        for t in ts:
            if t == 1.:
                x_t = x_1 * 1.
                f_t = 0.
            else:

                t_embedding = self.time_embedding(torch.Tensor([t
])))
                x_t =x_t - self.vnet(x_t + t_embedding) * delta_t

                # Calculate f_t
                # approximate the divergence using the Hutchinson
 trace estimator and the autograd
                self.vnet.eval()  # set the vector field net to
evaluation

                x = torch.FloatTensor(x_t.data)  # copy the
original data (it doesn't require grads!)
                x.requires_grad = True
                e = torch.randn_like(x)  # epsilon ~ Normal(0, I)
                e_grad = torch.autograd.grad(self.vnet(x).sum(),
x, create_graph=True)[0]
                e_grad_e = e_grad * e
                f_t = e_grad_e.view(x.shape[0], -1).sum(dim=1)

                self.vnet.eval()  # set the vector field net to
train again

        log_p_1 = self.log_p_base(x_t, reduction='sum') - f_t

        if reduction == "mean":
            return log_p_1.mean()
        elif reduction == "sum":
            return log_p_1.sum()
```

**Listing 9.3** A class for conditional flow matching.

Before we present the results of generated images, let us see how flow matching behaves on the two-moon dataset in Fig. 9.9. We start with data generated according to the standard Gaussian, and then the vector field model pushes the points toward two orange half-moons.

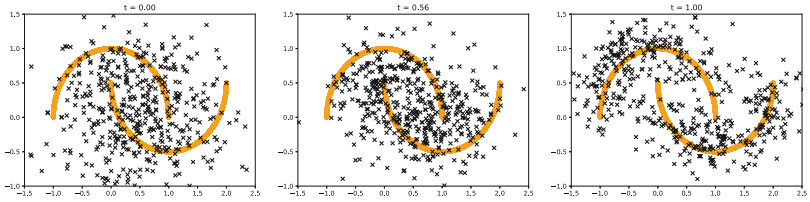



**Fig. 9.9** An example of sampling with a vector field model trained with flow matching: Sampled points (black crosses) evolve to data (orange half-moons).

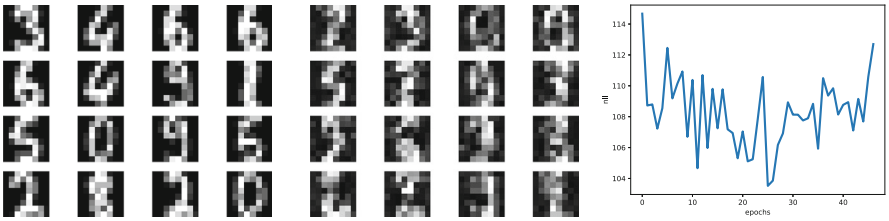



**Fig. 9.10** Some results: (**a**) A sample of real images. (**b**) A sample of generated images. (**c**) An example of the score matching loss calculated on the validation set.

After running the code with an MLP-based vector field model and the following values of the hyperparameters, $\sigma_{const} = 0.1$ and $T = 100$, we can expect results like in Fig. 9.10. Note that we report the negative log-likelihood estimated according to the procedure presented before.

### 9.4.4 What Is the Future of Flow Matching?

Flow matching is a framework first outlined in [32], published in the fall of 2022. Since then, there have been multiple extensions proposed. Here, I will point out a few but I highly recommend searching for new development, my curious reader, because I find this framework especially interesting! Alright, let me attract your attention to the following papers:

- Similar to latent diffusion, in [34], flow matching is used as a "prior" in an auto-encoder setting. First, the auto-encoder is trained, and then the vector field model is trained in the latent space. Afterward, a sample from the FM model is decoded back to the data space.
- The idea of interpolations in CFM was further extended to *stochastic interpolants* proposed by [35, 36]. I highly recommend looking these papers up since they provide many interesting extensions, both theoretical and practical.

- As I mentioned earlier, one can propose a better distribution $q(\mathbf{z})$ by using optimal transport (OT). This results in OT-CFM [33] and Schrodinger bridge CFM [29, 37].
- Action matching, a closely related approach to CFM, proposed by [38], allows learning an underlying mechanism of moving points in time without modeling the distributions at each step.
- Here, we considered interpolations in Euclidean spaces. Chen and Lipman [39] proposed an extension of CFMs to general Riemannian manifolds.
- To take advantage of symmetries in data, [40] modified the cost function in OT-CFM to account for those. Additionally, they used equivariant graph neural networks to formulate equivariant flow matching.
- Here, we discussed the case of continuous random variables. However, in practice, we often deal with discrete data, e.g., molecules, proteins, and pixel values. Campbell et al. [41] proposed discrete flow models that could be seen as a version of CFM for handling discrete data.
- CFM was also proposed as a method for simulation-based inference [42], i.e., a problem in which one has access to a simulator but the likelihood function is unknown or intractable.
- There are very close connections between score-based generative models and flow matching. I highly recommend looking into [33] and [5] (the appendix therein is simply marvelous!) for further details.

There are multiple amazing papers that I missed, but I did not do that on purpose. It is simply very difficult to follow all of them. This statement is also a testament that flow matching is becoming the mainstream research direction in Generative AI. In my opinion, it is a fascinating framework that could bring multiple breakthroughs!

## References

1. Yang Song, Jascha Sohl-Dickstein, Diederik P Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. Score-based generative modeling through stochastic differential equations. In *International Conference on Learning Representations*, 2020.
2. Jascha Sohl-Dickstein, Eric Weiss, Niru Maheswaranathan, and Surya Ganguli. Deep unsupervised learning using nonequilibrium thermodynamics. In *International Conference on Machine Learning*, pages 2256–2265. PMLR, 2015.
3. Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *Advances in Neural Information Processing Systems*, 33:6840–6851, 2020.
4. Diederik P Kingma, Tim Salimans, Ben Poole, and Jonathan Ho. Variational diffusion models. *arXiv preprint arXiv:2107.00630*, 2021.

5. Diederik Kingma and Ruiqi Gao. Understanding diffusion objectives as the elbo with simple data augmentation. *Advances in Neural Information Processing Systems*, 36, 2024.
6. Max Welling and Yee W Teh. Bayesian learning via stochastic gradient langevin dynamics. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 681–688. Citeseer, 2011.
7. Aapo Hyvärinen and Peter Dayan. Estimation of non-normalized statistical models by score matching. *Journal of Machine Learning Research*, 6(4), 2005.
8. Aapo Hyvärinen. Some extensions of score matching. *Computational statistics & data analysis*, 51(5):2499–2512, 2007.
9. Pascal Vincent. A connection between score matching and denoising autoencoders. *Neural computation*, 23(7):1661–1674, 2011.
10. Yang Song and Stefano Ermon. Generative modeling by estimating gradients of the data distribution. *arXiv preprint arXiv:1907.05600*, 2019.
11. Yang Song and Stefano Ermon. Improved techniques for training score-based generative models. *Advances in neural information processing systems*, 33:12438–12448, 2020.
12. Yang Song, Sahaj Garg, Jiaxin Shi, and Stefano Ermon. Sliced score matching: A scalable approach to density and score estimation. In *Uncertainty in Artificial Intelligence*, pages 574–584. PMLR, 2020.
13. Julien Rabin, Gabriel Peyré, Julie Delon, and Marc Bernot. Wasserstein barycenter and its application to texture mixing. In *Scale Space and Variational Methods in Computer Vision: Third International Conference, SSVM 2011, Ein-Gedi, Israel, May 29–June 2, 2011, Revised Selected Papers 3*, pages 435–446. Springer, 2012.
14. Benjamin Marlin, Kevin Swersky, Bo Chen, and Nando Freitas. Inductive principles for restricted boltzmann machine learning. In *Proceedings of the thirteenth International Conference on Artificial Intelligence and Statistics*, pages 509–516, 2010.
15. Kevin Swersky, Marc'Aurelio Ranzato, David Buchman, Nando D Freitas, and Benjamin M Marlin. On autoencoders and score matching for energy based models. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 1201–1208, 2011.
16. Zengyi Li, Yubei Chen, and Friedrich T Sommer. Learning energy-based models in high-dimensional spaces with multiscale denoising-score matching. *Entropy*, 25(10):1367, 2023.
17. Chenlin Meng, Kristy Choi, Jiaming Song, and Stefano Ermon. Concrete score matching: Generalized score matching for discrete data. *Advances in Neural Information Processing Systems*, 35:34532–34545, 2022.
18. Shitong Luo and Wei Hu. Score-based point cloud denoising. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4583–4592, 2021.
19. Simo Särkkä and Arno Solin. *Applied stochastic differential equations*, volume 10. Cambridge University Press, 2019.

20. Yang Song, Conor Durkan, Iain Murray, and Stefano Ermon. Maximum likelihood training of score-based diffusion models. *Advances in neural information processing systems*, 34:1415–1428, 2021.
21. Yang Song and Diederik P Kingma. How to train your energy-based models. *arXiv preprint arXiv:2101.03288*, 2021.
22. Cheng Lu, Yuhao Zhou, Fan Bao, Jianfei Chen, Chongxuan Li, and Jun Zhu. Dpm-solver: A fast ode solver for diffusion probabilistic model sampling in around 10 steps. *Advances in Neural Information Processing Systems*, 35:5775–5787, 2022.
23. Zhenyu Zhou, Defang Chen, Can Wang, and Chun Chen. Fast ode-based sampling for diffusion models in around 5 steps. *arXiv preprint arXiv:2312.00094*, 2023.
24. Tero Karras, Miika Aittala, Timo Aila, and Samuli Laine. Elucidating the design space of diffusion-based generative models. *Advances in Neural Information Processing Systems*, 35:26565–26577, 2022.
25. Arash Vahdat, Karsten Kreis, and Jan Kautz. Score-based generative modeling in latent space. *arXiv preprint arXiv:2106.05931*, 2021.
26. Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. *Advances in neural information processing systems*, 31, 2018.
27. Kaiwen Zheng, Cheng Lu, Jianfei Chen, and Jun Zhu. Improved techniques for maximum likelihood estimation for diffusion odes. In *International Conference on Machine Learning*, pages 42363–42389. PMLR, 2023.
28. Yang Song, Prafulla Dhariwal, Mark Chen, and Ilya Sutskever. Consistency models. *arXiv preprint arXiv:2303.01469*, 2023.
29. Valentin De Bortoli, James Thornton, Jeremy Heng, and Arnaud Doucet. Diffusion schrödinger bridge with applications to score-based generative modeling. *Advances in Neural Information Processing Systems*, 34:17695–17709, 2021.
30. Heli Ben-Hamu, Samuel Cohen, Joey Bose, Brandon Amos, Aditya Grover, Maximilian Nickel, Ricky TQ Chen, and Yaron Lipman. Matching normalizing flows and probability paths on manifolds. *arXiv preprint arXiv:2207.04711*, 2022.
31. Will Grathwohl, Ricky TQ Chen, Jesse Bettencourt, Ilya Sutskever, and David Duvenaud. Ffjord: Free-form continuous dynamics for scalable reversible generative models. *arXiv preprint arXiv:1810.01367*, 2018.
32. Yaron Lipman, Ricky TQ Chen, Heli Ben-Hamu, Maximilian Nickel, and Matt Le. Flow matching for generative modeling. *arXiv preprint arXiv:2210.02747*, 2022.
33. Alexander Tong, Nikolay Malkin, Guillaume Huguet, Yanlei Zhang, Jarrid Rector-Brooks, Kilian Fatras, Guy Wolf, and Yoshua Bengio. Improving and generalizing flow-based generative models with minibatch optimal transport. *arXiv preprint arXiv:2302.00482*, 2023.
34. Quan Dao, Hao Phung, Binh Nguyen, and Anh Tran. Flow matching in latent space. *arXiv preprint arXiv:2307.08698*, 2023.

35. Michael S Albergo, Nicholas M Boffi, and Eric Vanden-Eijnden. Stochastic interpolants: A unifying framework for flows and diffusions. *arXiv preprint arXiv:2303.08797*, 2023.
36. Michael S Albergo, Mark Goldstein, Nicholas M Boffi, Rajesh Ranganath, and Eric Vanden-Eijnden. Stochastic interpolants with data-dependent couplings. *arXiv preprint arXiv:2310.03725*, 2023.
37. Francisco Vargas, Pierre Thodoroff, Austen Lamacraft, and Neil Lawrence. Solving schrödinger bridges via maximum likelihood. *Entropy*, 23(9):1134, 2021.
38. Kirill Neklyudov, Daniel Severo, and Alireza Makhzani. Action matching: A variational method for learning stochastic dynamics from samples, 2022. *ICML*, 6662.
39. Ricky TQ Chen and Yaron Lipman. Riemannian flow matching on general geometries. *arXiv preprint arXiv:2302.03660*, 2023.
40. Leon Klein, Andreas Krämer, and Frank Noé. Equivariant flow matching. *Advances in Neural Information Processing Systems*, 36, 2024.
41. Andrew Campbell, Jason Yim, Regina Barzilay, Tom Rainforth, and Tommi Jaakkola. Generative flows on discrete state-spaces: Enabling multimodal flows with applications to protein co-design. *arXiv preprint arXiv:2402.04997*, 2024.
42. Jonas Wildberger, Maximilian Dax, Simon Buchholz, Stephen Green, Jakob H Macke, and Bernhard Schölkopf. Flow matching for scalable simulation-based inference. *Advances in Neural Information Processing Systems*, 36, 2024.

# Chapter 10
# Deep Generative Modeling for Neural Compression


## 10.1 Introduction

In December 2020, Facebook reported having around 1.8 billion daily active users and around 2.8 billion monthly active users [1]. Assuming that users uploaded, on average, a single photo each day, the resulting volume of data would give a very rough (let me stress it, **a very rough**) estimate of around 3000 TB of new images per day. This single case of Facebook alone already shows us the potential great costs associated with storing and transmitting data. In the digital era, we can simply say this: efficient and effective manner of handling data (i.e., **faster** and **smaller**) means more money in the pocket.

The most straightforward way of dealing with these issues (i.e., smaller and faster) is based on applying compression, and, in particular, image compression algorithms (codecs) that allow us to decrease the size of an image. Instead of changing infrastructure, we can efficiently and effectively store and transmit images by making them simply smaller! Let us be honest, the more we compress an image, the more and faster we can send it and the less disk memory we need!

If we think of image compression, probably the first association is JPEG or PNG, standards used on a daily basis by everyone. I will not go into details of these standards (e.g., see [2, 3] for an introduction), but what it is important to know is that they use some predefined math like discrete cosine transform. The main advantage of standard codecs like JPEG is that they are interpretable, i.e., all steps are hand-designed, and their behavior can be predicted. However, this comes at the cost of insufficient flexibility which could drastically decrease their performance. So how we can increase the flexibility of transformations? Any idea? Anyone? Do I hear deep learning [4, 5]? Indeed! Many of today's image compression algorithms are enhanced by neural networks.

The emerging field of compression algorithms using neural networks is called **neural compression**. Neural compression becomes a leading trend in developing new codecs where neural networks replace parts of the standard codecs [6] or neural-based codecs are trained [7] together with quantization [8] and entropy coding [9–12].



J. M. Tomczak, *Deep Generative Modeling*,
https://doi.org/10.1007/978-3-031-64087-2_10

We will discuss the general compression scheme in detail in the next subsection, but here it is important to understand why deep generative modeling is important in the context of neural compression. The answer was given a long time ago by Claude Shannon who showed in [13] that (informally):

```
The length of a message representing source data is
    proportional to the entropy of this data.
```

We do not know the entropy of data because we do not know the probability distribution of data, $p(\mathbf{x})$, but we can estimate it using one of the deep generative models we have discussed so far! Because of that, recently, there has been an increasing interest in using deep generative modeling to improve neural compression. We can use deep generative models for modeling probability distribution for entropy coders [9–12] but also to significantly increase the final reconstruction and compression quality by incorporating new schemes for inference [14] and reconstruction [15].

## 10.2 General Compression Scheme

Before we jump into neural compression, it is beneficial (and educational, don't be afraid of refreshing the basics!) to remind ourselves what is image (or data, in general) compression. We can distinguish two image compression approaches [16], namely:

- *Lossless compression*: a method that preserves all information and reconstructions are error-free
- *Lossy compression*: information is not preserved completely by a compression method

A general recipe for designing a compression algorithm relies on devising a uniquely decodable code whose expected length is as close as possible to the entropy of the data [13]. The general compression system consists of two components [16, 17]: an **encoder** and a **decoder**. Please do not think of it as a deterministic VAE because it is not the same. There are some similarities, but in the compression task, we are really interested in sending the **bitstream**, while in VAEs we typically do not care at all. We play with floats and say about codes just to make it more intuitive, but it requires a few extra steps to turn it into a "real" compression scheme. We are going to explain these extra in the next sections. Alright, let's start!

### 10.2.1 Encoder

In the encoder, the goal is to transform an image into a discrete signal. It is important to understand that the signal does not necessarily need to be binary. The transformation we use could be invertible; however, it is not a requirement. If the transformation

is invertible, then we can use its inverse in the decoder, and, in principle, we can talk about lossless compression. Why? Take a look at the flow-based models where we discussed invertibility. However, if the transformation is not invertible, then some information is lost, and we land in the group of lossy compression methods. Next, after applying the transformation to the input image, the discrete signal is encoded into a bitstream in a lossless manner. In other words, discrete symbols are mapped to binary variables (bits). Typically, entropy coders utilize information about the probability of symbol occurrence, e.g., Huffman coders or arithmetic coders [16]. This is important to understand that for many entropy coders we need to know $p(\mathbf{x})$, and here we can use deep generative models.

### 10.2.2 Decoder

Once the message (i.e., the bits) is sent and received, the bitstream is decoded to the discrete signal by the entropy decoder. The entropy decoder is the inversion of the entropy encoder. Entropy coding methods allow us to receive the original symbols from the bitstream. Eventually, an inverse transformation (not necessarily the inversion of the encoder transformation) is applied to reconstruct the original image.

### 10.2.3 The Full Scheme

Please see Fig. 10.1 for a general scheme of a compression system (a codec). The standard codecs mainly utilize multi-scale image decomposition like wavelet representation [3, 18] that are further quantized. Considering a specific discrete transformation (e.g., discrete cosine transform (DCT)) results in a specific codec (e.g., JPEG [2]).

### 10.2.4 The Objective

The final performance of a codec is evaluated in terms of reconstruction error and compression ratio. The reconstruction error is called **distortion** measure and is calculated as a difference between the original image and the reconstructed image using the mean square error ($MSE$) (typically, the peak signal-to-noise ratio ($PSNR$) expressed as $10 \log_{10} \frac{255^2}{MSE}$ is reported) or perceptual metrics like the multi-scale structure similarity index ($MS\text{-}SSIM$) [19]. The compression ratio, called **rate**, is usually expressed by the **bits per pixel** (**bpp**), i.e., the total size in bits of the encoder output divided by the total size in pixels of the encoder input [17]. Typically,

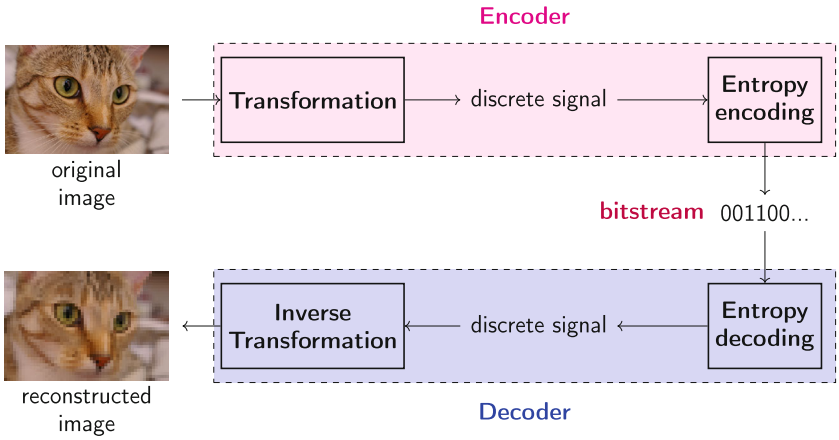



**Fig. 10.1** A general image compression system (a codec).

the performance of codecs is compared by inspecting the rate-distortion plane (i.e., plotting curves on a plane with the rate on the x-axis and the distortion on the y-axis).

Formally, we assume an autoencoder architecture (see Fig. 10.1 again) with an encoding transformation, $f_e : \mathcal{X} \to \mathcal{Y}$, that takes an input $\mathbf{x}$ and returns a discrete signal $\mathbf{y}$ (a code). After sending the message, a reconstruction $\hat{\mathbf{x}}$ is given by a decoder, $f_d : \mathcal{Y} \to \mathcal{X}$. Moreover, there is an (adaptive) entropy coding model that learns the distribution $p(\mathbf{y})$ and is further used to turn the discrete signal $\mathbf{y}$ into a bitstream by an entropy coder (e.g., Huffman coding, arithmetic coding). If a compression method has any adaptive (hyper)parameters, it could be learned by optimizing the following objective function:

$$\mathcal{L}(\mathbf{x}) = d\left(\mathbf{x}, \hat{\mathbf{x}}\right) + \beta r\left(\mathbf{y}\right), \tag{10.1}$$

where $d(\cdot, \cdot)$ is the **distortion** measure (e.g., PSNR, MS-SSIM), $r(\cdot)$ is the **rate** measure (e.g., $r\left(\mathbf{y}\right) = -\ln p(\mathbf{y})$), and $\beta > 0$ is a weighting factor controlling the balance between rate and distortion. Notice that distortion requires both the encoder and the decoder and rate requires the encoder and the entropy model.

## 10.3 A Short Detour: JPEG

We have discussed all necessary concepts of image compression, and now we can delve into neural compression. However, before we do that, the first question we face is whether we can get any benefit from using neural networks for compression and where and how we can use them in this context. As mentioned already, standard codecs utilize a series of predefined transformations and mathematical operations. But how does it work?

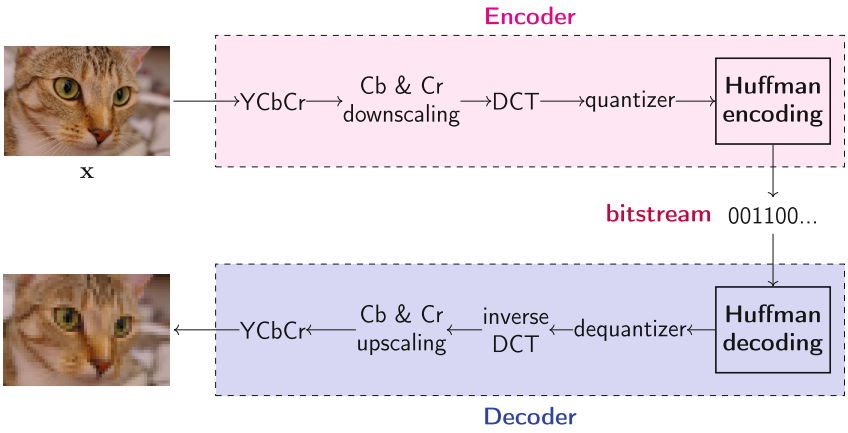



**Fig. 10.2** A JPEG compression system.

Let us quickly discuss one of the most commonly used codecs: JPEG. In the JPEG codec, an RGB image is first linearly transformed to the YCbCr format:

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix} + \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.168736 & -0.331264 & 0.5 \\ 0.5 & -0.48688 & -0.081312 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{10.2}$$

Then, Cb and Cr channels are downscaled, typically two or three times (the first compression stage). After that, each channel is split into, e.g., $8 \times 8$ blocks, and fed to the discrete cosine transform (DCT) that is eventually quantized (the second compression stage). After all, the Huffman coding could be used. To decode the signal, the inverse DCT is used, the Cb and Cr channels are upscaled, and the RGB representation is recovered. The whole system is presented in Fig. 10.2. As you can notice, each step is easy to follow, and if you know how DCT works, the whole procedure is a white box. There are some hyperparameters, but, again, they have a very clear interpretation (e.g., how many times Cb and Cr channels are downscaled, the size of blocks).

## 10.4 Neural Compression: Components

Alright, we know now how a standard codec works. However, one of the problems with standard codecs is that they are not necessarily flexible. One may ask whether DCT is the optimal transformation for all images. The answer is, with high probability, no. If we are willing to give up the nicely designed white box, we can turn it into a black box by replacing all mathematical operations with neural networks. The potential gain is increased flexibility and potentially better performance (both in terms of distortion and rate).

We need to remember though that learning neural networks requires the **differentiability** of the whole approach. However, we require a discrete output of the neural network that could break the backpropagation! For this purpose, we must formulate a **differentiable quantization procedure**. Additionally, to obtain a powerful model, we need an adaptive entropy coding model. This is an important component of the neural compression pipeline because it not only optimizes the compression ratio (i.e., the rate) but also helps to learn a useful codebook. Next, we will discuss both components in detail.

### 10.4.1 Encoders and Decoders

In neural compression, unlike in VAEs, the encoder and the decoder consist of neural networks with no additional functions. As a result, we focus on architectures rather than how to parameterize a distribution, for instance. The output of the encoder is a continuous code (floats), and the output of the decoder is a reconstruction of an image. Below, we present PyTorch classes for an encoder and a decoder with examples of neural networks:

```
# The encoder is simply a neural network that takes an image and outputs a corresponding code.
class Encoder(nn.Module):
    def __init__(self, encoder_net):
        super(Encoder, self).__init__()

        self.encoder = encoder_net

    def encode(self, x):
        h_e = self.encoder(x)
        return h_e

    def forward(self, x):
        return self.encode(x)

# The decoder is simply a neural network that takes a quantized code and returns an image.
class Decoder(nn.Module):
    def __init__(self, decoder_net):
        super(Decoder, self).__init__()

        self.decoder = decoder_net

    def decode(self, z):
        h_d = self.decoder(z)
        return h_d

    def forward(self, z, x=None):
        x_rec = self.decode(z)
        return x_rec
```

**Listing 10.1** Classes for an encoder and a decoder.

```
# ENCODER
e_net = nn.Sequential(
            nn.Linear(D, M*2), nn.BatchNorm1d(M*2), nn.ReLU(),
            nn.Linear(M*2, M), nn.BatchNorm1d(M), nn.ReLU(),
            nn.Linear(M, M//2), nn.BatchNorm1d(M//2), nn.ReLU(),
            nn.Linear(M//2, C))

encoder = Encoder(encoder_net=e_net)

# DECODER
d_net = nn.Sequential(
            nn.Linear(C, M//2), nn.BatchNorm1d(M//2), nn.ReLU(),
            nn.Linear(M//2, M), nn.BatchNorm1d(M), nn.ReLU(),
            nn.Linear(M, M*2), nn.BatchNorm1d(M*2), nn.ReLU(),
            nn.Linear(M*2, D))

decoder = Decoder(decoder_net=d_net)
```

**Listing 10.2** Examples of neural networks for the encoder and the decoder.

### 10.4.2 Differentiable Quantization

The problem with utilizing neural networks in the compression context is that we must ensure training by backpropagation that is equivalent to using only differentiable operations. Unfortunately, working with discrete outputs of a neural network breaks differentiability and requires applying approximations of gradients (e.g., the straight-through estimator). However, we can use quantization of codes $\mathbf{y}$ and make it differentiable with relatively simple tricks.

We assume that the encoder gives us a code $\mathbf{y} \in \mathbb{R}^M$. Moreover, we assume that there is a codebook $\mathbf{c} \in \mathbb{R}^K$. We can think of the codebook as a vector of additional parameters (yes, parameters, we can also learn it!). Now, the whole idea relies on quantizing $\mathbf{y}$ to values in the codebook $\mathbf{c}$. Easy right? Well, it is easy but it still tells us nothing. Quantizing in this context means that we will take every element in $\mathbf{y}$ and find the closest value in the codebook and replace it with this codebook value. We can implement it using matrix calculus in the following manner. First, we repeat $\mathbf{y}$ $K$-times, and we repeat $\mathbf{c}$ $M$-times that gives us two matrices: $\mathbf{Y} \in \mathbb{R}^{M \times K}$ and $\mathbf{C} \in \mathbb{R}^{M \times K}$. Now, we can calculate a similarity matrix, for instance, $\mathbf{S} = \exp\{-\sqrt{(\mathbf{Y} - \mathbf{C})^2}\} \in \mathbb{R}^{M \times K}$. The matrix $\mathbf{S}$ has the highest value where the $m$-th value of $\mathbf{y}$, $\mathbf{y}_m$, is closest to the $k$-th vale of $\mathbf{c}$, $\mathbf{c}_k$. So far so good, all operations are differentiable. However, there is no quantization here (i.e., values are not discrete). Since we have the similarity matrix $\mathbf{S} \in \mathbb{R}^{M \times K}$ and we can apply the softmax nonlinearity with temperature to the second dimension of $\mathbf{S}$, namely, $\hat{\mathbf{S}} = \text{softmax}_2(\tau \cdot \mathbf{S})$ (here, the subscript denotes that we calculate the softmax wrt the second dimension) where $\tau \gg 1$ (e.g., $\tau = 10^7$). Since we apply the softmax to the similarity matrix multiplied by a very large number, the resulting matrix, $\hat{\mathbf{S}}$,

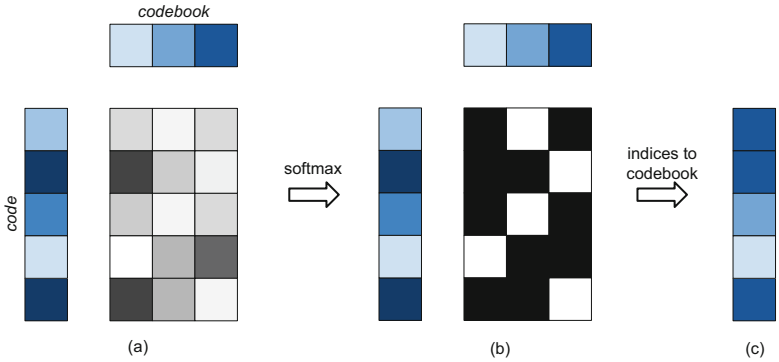



**Fig. 10.3** An example of the quantization of codes. (**a**) Distances. (**b**) Indices. (**c**) Quantized code.

will still consist of floats, but numerically these values will be 0s and a single 1. An example of this kind of quantization is presented in Fig. 10.3.

Importantly, the softmax nonlinearity is differentiable, and, eventually, the whole procedure is differentiable. In the end, we can calculate quantized codes by multiplying the codebook with the 0–1 similarity matrix, namely,

$$\hat{\mathbf{y}} = \hat{\mathbf{S}}\mathbf{c}. \tag{10.3}$$

The resulting code, $\hat{\mathbf{y}}$, consists of values from the codebook only.

We can ask ourselves whether indeed we gain anything because values of $\hat{\mathbf{y}}$ are still floats. So, in other words, where is the discrete signal we want to turn into the bitstream? We can answer it in two ways: First, there are only $K$ possible values in the codebook. Hence, the values are discrete but represented by a finite number of floats. Second, the real magic happens when we calculate the matrix $\hat{\mathbf{S}}$. Notice that this matrix is indeed discrete! In each row, there is a single position with 1 and 0s elsewhere. As a result, either we look at it from the codebook perspective or the similarity matrix perspective, we should be convinced that indeed we can turn any real-valued vector into a vector consisting of real values but from a finite set. And, most importantly, this whole procedure of quantizing allows us to apply the backpropagation algorithm! This quantization approach (or very similar) was used in many neural compression methods, e.g., [8], [11], or [10]. We can also use other differential quantization techniques, e.g., vector quantization [16]. However, we prefer to stick to a simple codebook that turns out to be pretty effective in practice:

```
class Quantizer(nn.Module):
    def __init__(self, input_dim, codebook_dim, temp=1.e7):
        super(Quantizer, self).__init__()
        #temperature for softmax
        self.temp = temp

        # dimensionality of the inputs and the codebook
        self.input_dim = input_dim
        self.codebook_dim = codebook_dim
```

```
10
11          # codebook layer (a codebook)
12          # initialize uniformly and a Parameter (learnable)
13          self.codebook = nn.Parameter(torch.FloatTensor(1, self.
    codebook_dim,).uniform_(-1/self.codebook_dim, 1/self.
    codebook_dim))
14
15      # A function for codebook indices (a one-hot representation)
        to values in the codebook.
16      def indices2codebook(self, indices_onehot):
17          return torch.matmul(indices_onehot, self.codebook.t()).
    squeeze()
18
19      # A function to change integers to a one-hot representation.
20      def indices_to_onehot(self, inputs_shape, indices):
21          indices_hard = torch.zeros(inputs_shape[0], inputs_shape
    [1], self.codebook_dim)
22          indices_hard.scatter_(2, indices, 1)
23
24      # The forward function:
25      # - First, distances are calculated between input values and
        codebook values.
26      # - Second, indices (soft - differentiable, hard - non-
        differentiable) between the encoded values and the codebook
        values are calculated.
27      # - Third, the quantizer returns indices and quantized code (
        the output of the encoder).
28      # - Fourth, the decoder maps the quantized code to the
        observable space (i.e., it decodes the code back).
29      def forward(self, inputs):
30          # inputs - a matrix of floats, B x M
31          inputs_shape = inputs.shape
32          # repeat inputs
33          inputs_repeat = inputs.unsqueeze(2).repeat(1, 1, self.
    codebook_dim)
34          # calculate distances between input values and the
        codebook values
35          distances = torch.exp(-torch.sqrt(torch.pow(inputs_repeat
     - self.codebook.unsqueeze(1), 2)))
36
37          # indices (hard, i.e., nondiff)
38          indices = torch.argmax(distances, dim=2).unsqueeze(2)
39          indices_hard = self.indices_to_onehot(inputs_shape=
    inputs_shape, indices=indices)
40
41          # indices (soft, i.e., diff)
42          indices_soft = torch.softmax(self.temp * distances, -1)
43
44          # quantized values: we use soft indices here because it
        allows backpropagation
45
46
47
48
```

```
        quantized = self.indices2codebook(indices_onehot=
    indices_soft)

        return (indices_soft, indices_hard, quantized)
```

**Listing 10.3** A quantizer class.

### 10.4.3 Adaptive Entropy Coding Model

The last piece in the whole puzzle is entropy coding. We rely on entropy coders like Huffman coding or arithmetic coding. Either way, these entropy coders require from us an estimate of the probability distribution over codes, $p(\mathbf{y})$. Once they have it, they can losslessly compress the discrete signal into a bitstream. In general, we can encode discrete symbols to bits separately (e.g., Huffman coding) or encode the whole discrete signal into a bit representation (e.g., arithmetic coding). In compression systems, arithmetic coding is preferable over Huffman coding because it is faster and more accurate (i.e., better compression) [16].

We will not review and explain in detail how arithmetic coding works. We refer to [16] (or any other book on data compression) for details. There are two facts we need to know and remember: First, if we provide probabilities of symbols, then arithmetic coding does not need to make an extra pass through the signal to estimate them. Second, there is an adaptive variant of arithmetic coding that allows modifying probabilities while compressing symbols sequentially (also known as *progressive coding*).

These two remarks are important for us because, as mentioned earlier, we can estimate $p(\mathbf{y})$ using a deep generative model. Once we learn the deep generative model, the arithmetic coding can use this distribution for the lossless compression of codes. Moreover, if we use a model that factorizes the distribution, e.g., an autoregressive model, then we can also utilize the idea of progressive coding.

In our example, we use the autoregressive model that takes quantized code and returns the probability of each value in the codebook (i.e., the indices). In other words, the autoregressive model outcomes probabilities over the codebook values. It is worth mentioning though that in our implementation we use the term "entropy coding" but we mean an entropy coding model. Moreover, it is worth mentioning that there are specialized distributions for compression purposes, e.g., the scale hyperprior [9], but here we are interested in deep generative modeling for neural compression:

```
class ARMEntropyCoding(nn.Module):
    def __init__(self, code_dim, codebook_dim, arm_net):
        super(ARMEntropyCoding, self).__init__()
        self.code_dim = code_dim
        self.codebook_dim = codebook_dim
        self.arm_net = arm_net # it takes B x 1 x code_dim and
    outputs B x codebook_dim x code_dim

```

```
    def f(self, x):
        h = self.arm_net(x.unsqueeze(1))
        h = h.permute(0, 2, 1)
        p = torch.softmax(h, 2)

        return p

    def sample(self, quantizer=None, B=10):
        x_new = torch.zeros((B, self.code_dim))

        for d in range(self.code_dim):
            p = self.f(x_new)
            indx_d = torch.multinomial(p[:, d, :], num_samples=1)
            codebook_value = quantizer.codebook[0, indx_d].
squeeze()
            x_new[:, d] = codebook_value

        return x_new

    def forward(self, z, x):
        p = self.f(x)
        return -torch.sum(z * torch.log(p), 2)
```

**Listing 10.4** An adaptive entropy coding model using an ARM class.

### 10.4.4 A Neural Compression System

We discussed all the components of a neural compression system and gave some very specific examples of how they could be implemented. There are many other propositions on how these could be formulated, ranging from elaborated neural network architectures for encoders and decoders to various quantization schemes and entropy coding models. Nevertheless, the presented neural compressor should give you a good idea of how neural networks could be utilized for image (or, generally, data) compression.

In Fig. 10.4, we represent a neural compression system where the transformations in Fig. 10.1 are replaced by neural networks together with a (differentiable) quantization procedure. We also highlight that floats are quantized to obtain a discrete signal. Altogether, comparing Figs. 10.1 and 10.4, we can notice that the whole pipeline is the same and the differences lie in how transformations are implemented.

The main difference, perhaps, is that a neural compressor could be trained end to end and specialized to given data. In fact, the objective of the neural compressor could be seen as a penalized reconstruction error of autoencoders. Let us assume we have given training data $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ and the corresponding empirical distribution $p_{data}(\mathbf{x})$. Moreover, we have an encoder network with weights $\phi$, $\mathbf{y} = f_{e,\phi}(\mathbf{x})$, a differentiable quantizer with a codebook $\mathbf{c}$, $\hat{\mathbf{y}} = Q(\mathbf{y}; \mathbf{x})$, a decoder network with weights $\theta$, $\hat{\mathbf{x}} = f_{d,\theta}(\hat{\mathbf{y}})$, and the entropy coding model with weights $\lambda$, $p_\lambda(\hat{\mathbf{y}})$. We can train the model in the end-to-end-fashion by minimizing the following objective ($\beta > 0$):

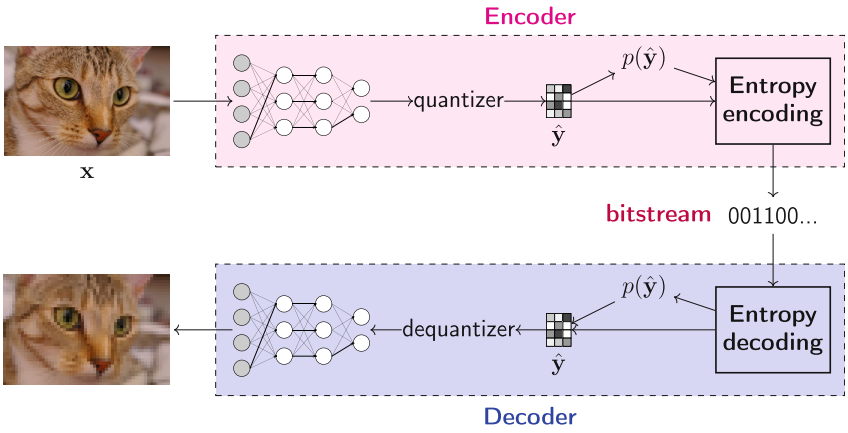



**Fig. 10.4** A neural compression system.

$$\begin{aligned}\mathcal{L}(\theta,\phi,\lambda,\mathbf{c}) =& \mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}\left[(\mathbf{x}-f_{d,\theta}(Q(f_{e,\phi}(\mathbf{x});\mathbf{c})))^2\right] + \\ &+ \beta\mathbb{E}_{\hat{\mathbf{y}}\sim p_{data}(\mathbf{x})\,\delta(Q(f_{e,\phi}(\mathbf{x});\mathbf{c})-\hat{\mathbf{y}})}\left[-\ln p_\lambda(\hat{\mathbf{y}})\right] \qquad (10.4)\\ =& \mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}\left[(\mathbf{x}-\hat{\mathbf{x}})^2\right] + \beta\mathbb{E}_{\hat{\mathbf{y}}\sim p_{data}(\mathbf{x})\,\delta\left(Q(f_{e,\phi}(\mathbf{x});\mathbf{c})-\hat{\mathbf{y}}\right)}\left[-\ln p_\lambda(\hat{\mathbf{y}})\right]. \qquad (10.5)\end{aligned}$$

If we look into the objective, we can immediately notice that the first component, $\mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}\left[(\mathbf{x}-\hat{\mathbf{x}})^2\right]$, is the mean squared error (MSE) loss. In other words, it is the reconstruction error. The second part, $\mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}\left[-\ln p_\lambda(\hat{\mathbf{y}})\right]$, is the cross-entropy between $q(\hat{\mathbf{y}}) = p_{data}(\mathbf{x})\,\delta\left(Q(f_{e,\phi}(\mathbf{x});\mathbf{c})-\hat{\mathbf{y}}\right)$ and $p_\lambda(\hat{\mathbf{y}})$, where $\delta(\cdot)$ denotes Dirac's delta. To see that, let us write it down step by step:

$$\begin{aligned}\mathbb{CE}\left[q(\hat{\mathbf{y}})||p_\lambda(\hat{\mathbf{y}})\right] &= -\sum_{\hat{\mathbf{y}}} q(\hat{\mathbf{y}})\ln p_\lambda(\hat{\mathbf{y}}) && (10.6)\\ &= -\sum_{\hat{\mathbf{y}}} p_{data}(\mathbf{x})\,\delta\left(Q(f_{e,\phi}(\mathbf{x});\mathbf{c})-\hat{\mathbf{y}}\right)\ln p_\lambda(\hat{\mathbf{y}}) && (10.7)\\ &= -\frac{1}{N}\sum_{n=1}^{N}\ln p_\lambda\left(Q(f_{e,\phi}(\mathbf{x}_n);\mathbf{c})\right). && (10.8)\end{aligned}$$

Eventually, we can write the training objective explicitly, replacing expectations with sums:

$$\mathcal{L}(\theta, \phi, \lambda, \mathbf{c}) = \underbrace{\frac{1}{N}\sum_{n=1}^{N}\left(\mathbf{x}_n - f_{d,\theta}\left(Q\left(f_{e,\phi}(\mathbf{x}_n); \mathbf{c}\right)\right)\right)^2}_{distortion} + \underbrace{\frac{\beta}{N}\sum_{n=1}^{N}\left[-\ln p_\lambda\left(Q\left(f_{e,\phi}(\mathbf{x}_n); \mathbf{c}\right)\right)\right]}_{rate}. \tag{10.9}$$

In the training objective, we have a sum of distortion and rate. Please note that during training, we do not need to use entropy coding at all. However, it is necessary if we want to use neural compression in practice:

**Bits per Pixel**

It is beneficial to discuss how the **bits per pixel** (bpp) is calculated. The definition of the bpp is the total size in bits of the encoder output divided by the total size in pixels of the encoder input. In our case, the encoder returns a code of size $M$, and each value is mapped to one of $K$ values (let us assume that $K = 2^\kappa$. As a result, we can represent the quantized code using indices, i.e., integers. Since we have $K$ possible integers, we can use $\kappa$ bits to represent each of them. As a result, the code is described by $\kappa \times M$ bits. In other words, we can use a uniform distribution with probability equal $1/(\kappa \times M)$ that gives the bpp equal $-\log_2(1/(\kappa \times M))/D$. However, we can improve this score by using entropy coding. As a result, we can use the rate value and divide it by the size of the image, $D$, to obtain the bpp, i.e., $-\log_2 p(\hat{\mathbf{y}})/D$.

```
class NeuralCompressor(nn.Module):
    def __init__(self, encoder, decoder, entropy_coding,
    quantizer, beta=1., detaching=False):
        super(NeuralCompressor, self).__init__()

        print('Neural Compressor by JT.')

        # we
        self.encoder = encoder
        self.decoder = decoder
        self.entropy_coding = entropy_coding
        self.quantizer = quantizer

        # beta determines how strongly we focus on compression
    against reconstruction quality
        self.beta = beta

        # We can detach inputs to the rate, then we learn rate
    and distortion separately
        self.detaching = detaching

    def forward(self, x, reduction='avg'):
```

```
        # encoding
        #-non-quantized values
        z = self.encoder(x)
        #-quantizing
        quantizer_out = self.quantizer(z)

        # decoding
        x_rec = self.decoder(quantizer_out[2])

        # Distortion (e.g., MSE)
        Distortion = torch.mean(torch.pow(x - x_rec, 2), 1)

        # Rate: we use the entropy coding here
        Rate = torch.mean(self.entropy_coding(quantizer_out[0],
    quantizer_out[2]), 1)

        # Objective
        objective = Distortion + self.beta * Rate

        if reduction == 'sum':
            return objective.sum(), Distortion.sum(), Rate.sum()
        else:
            return objective.mean(), Distortion.mean(), Rate.mean
    ()
```

**Listing 10.5** A neural compression class.

To conclude the description of the neural compressor, the whole compression procedure consists of the following steps, assuming that the model has been trained already:

**Compression Procedure**

1. Encode the input image, $\mathbf{y} = f_{e,\phi}(\mathbf{x})$.
2. Quantize the code, $\hat{\mathbf{y}} = Q(\hat{\mathbf{y}}; \mathbf{c})$.
3. Turn the quantized code $\hat{\mathbf{y}}$ into a bitstream using $p_\lambda(\hat{\mathbf{y}})$ and, e.g., arithmetic encoding.
4. Send the bits.
5. Decode bits into $\hat{\mathbf{y}}$ using $p_\lambda(\hat{\mathbf{y}})$ and, e.g., arithmetic decoding.
6. Decode $\hat{\mathbf{y}}$, $\hat{\mathbf{x}} = f_{d,\theta}(\hat{\mathbf{y}})$.

### 10.4.5 Example

After training the whole neural compression system with $\beta = 1$, you can get results as in Fig. 10.5.

Interestingly, since the entropy coder is also a deep generative model, we can sample from it. In Fig. 10.5c, there are four samples presented. They indicate that the model indeed can learn the data distribution of the quantized codes!

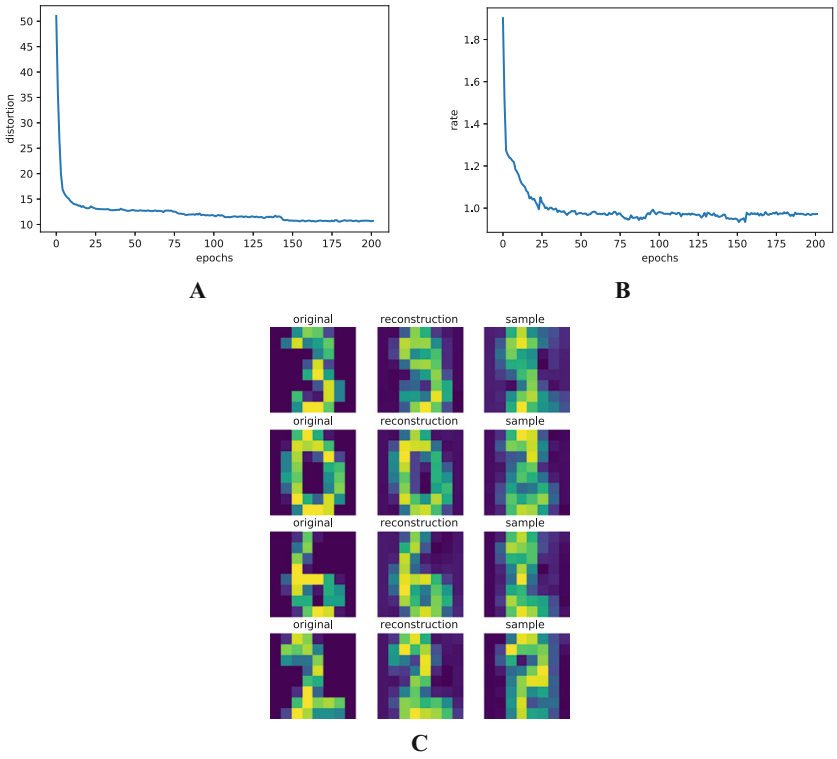



**Fig. 10.5** An example of outcomes after the training: (**a**) A distortion curve. (**b**) A rate curve. (**c**) Real images (left columns) and their reconstructions (middle column) and samples from the autoregressive entropy coder (right column).

## 10.5 What's Next?

Neural compression is a fascinating field. Utilizing neural networks for compression opens new possibilities for developing new coding schemes. Neural compression achieved comparable or (sometimes) better results than standard codecs on image compression [20]; however, there is still room for improvement in video compression or audio compression even though recent attempts are very promising [21, 22]. There are many interesting research directions I missed here. I highly recommend taking a look at a nice overview of neural compression methods [20]. Lastly, I want to highlight that here we used the deep autoregressive generative model; however, we can use other deep generative models (e.g., flows, VAEs).

## References


1. Facebook. Facebook reports fourth quarter and full year 2020 results, 2020.
2. Rashid Ansari, Christine Guillemot, and Nasir Memon. Jpeg and jpeg2000. In *The Essential Guide to Image Processing*, pages 421–461. Elsevier, 2009.
3. Zixiang Xiong and Kannan Ramchandran. Wavelet image compression. In *The Essential Guide to Image Processing*, pages 463–493. Elsevier, 2009.
4. Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.
5. Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.
6. Lionel Gueguen, Alex Sergeev, Ben Kadlec, Rosanne Liu, and Jason Yosinski. Faster neural networks straight from jpeg. *Advances in Neural Information Processing Systems*, 31:3933–3944, 2018.
7. Lucas Theis, Wenzhe Shi, Andrew Cunningham, and Ferenc Huszár. Lossy image compression with compressive autoencoders. *arXiv preprint arXiv:1703.00395*, 2017.
8. Eirikur Agustsson, Fabian Mentzer, Michael Tschannen, Lukas Cavigelli, Radu Timofte, Luca Benini, and Luc Van Gool. Soft-to-hard vector quantization for end-to-end learning compressible representations. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 1141–1151, 2017.
9. Johannes Ballé, David Minnen, Saurabh Singh, Sung Jin Hwang, and Nick Johnston. Variational image compression with a scale hyperprior. In *International Conference on Learning Representations*, 2018.
10. Amirhossein Habibian, Ties van Rozendaal, Jakub M Tomczak, and Taco S Cohen. Video compression with rate-distortion autoencoders. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 7033–7042, 2019.
11. Fabian Mentzer, Eirikur Agustsson, Michael Tschannen, Radu Timofte, and Luc Van Gool. Conditional probability models for deep image compression. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4394–4402, 2018.
12. David Minnen, Johannes Ballé, and George Toderici. Joint autoregressive and hierarchical priors for learned image compression. *arXiv preprint arXiv:1809.02736*, 2018.
13. Claude Elwood Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.
14. Yibo Yang, Robert Bamler, and Stephan Mandt. Improving inference for neural image compression. *Advances in Neural Information Processing Systems*, 33, 2020.
15. Fabian Mentzer, George D Toderici, Michael Tschannen, and Eirikur Agustsson. High-fidelity generative image compression. *Advances in Neural Information Processing Systems*, 33, 2020.

16. David Salomon. *Data compression: the complete reference*. Springer Science & Business Media, 2004.
17. LJ Karam. Lossless image compression. In Al Bovik, editor, *The Essential Guide to Image Processing*. Elsevier Academic Press, 2009.
18. Pierre Moulin. Multiscale image decompositions and wavelets. In *The essential guide to image processing*, pages 123–142. Elsevier, 2009.
19. Zhou Wang, Eero P Simoncelli, and Alan C Bovik. Multiscale structural similarity for image quality assessment. In *The Thirty-Seventh Asilomar Conference on Signals, Systems & Computers, 2003*, volume 2, pages 1398–1402. IEEE, 2003.
20. Johannes Ballé, Philip A Chou, David Minnen, Saurabh Singh, Nick Johnston, Eirikur Agustsson, Sung Jin Hwang, and George Toderici. Nonlinear transform coding. *IEEE Journal of Selected Topics in Signal Processing*, 15(2):339–353, 2020.
21. Adam Golinski, Reza Pourreza, Yang Yang, Guillaume Sautiere, and Taco S Cohen. Feedback recurrent autoencoder for video compression. In *Proceedings of the Asian Conference on Computer Vision*, 2020.
22. Yang Yang, Guillaume Sautière, J Jon Ryu, and Taco S Cohen. Feedback recurrent autoencoder. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3347–3351. IEEE, 2020.

# Chapter 11
# From Large Language Models to Generative AI Systems


## 11.1 Introduction

How is it possible, my curious reader, that we can share our thoughts? How can it be that we discuss generative modeling, probability theory, or other interesting concepts? How come? The answer is simple: language. We communicate because the human species developed a pretty distinctive trait that allows us to formulate sounds in a very complex manner to express our ideas and experiences. At some point in our history, some people realized that we forget, we lie, and we can shout as strongly as we can, but we will not be understood farther than a few hundred meters. The solution was a huge breakthrough: writing. This whole mumbling on my side here could be summarized using one word: text. We know how to write (and read), and we can use the word *text* to mean *language* or *natural language* to avoid any confusion with artificial languages like Python or formal language.

Communication is an essential component of our lives. Therefore, it is no surprise that processing text, or natural language in general, attracted a lot of attention from day one of AI. Since the 1950s of the twentieth century, people have been fascinated by the possibility of building communicating bots. Thanks to Alan Turing, we have even a famous test named after him which states that if a machine can *talk* to a human being without being recognized as a machine, then we can call it *intelligent*. The first chatbot was even introduced in the 1960s of the twentieth century. ELIZA, because it was its name, was able to match patterns and mimic an "intelligible" conversation. But it was way far from being *intelligent*, and, after all, it was unable to *generate* responses that would fool good Dr. Turing. It took us over 60 years (nothing in the history of human beings but eons in the history of AI) to get to the point where we can debate about intelligence, originality, novelty, and many other aspects that were exclusive to us, humans. And all these are possible due to generative modeling.

Obviously, we communicate using not only text but also images, audio, diagrams, and all sorts of *languages* (e.g., chemical reactions, mathematics). That is why, multimodality plays a crucial role in developing the next level of AI systems. In the following, we will talk about natural language processing (NLP) and how it was



J. M. Tomczak, *Deep Generative Modeling*,
https://doi.org/10.1007/978-3-031-64087-2_11

influenced by deep learning. The combination of deep learning and NLP gave rise to their baby called a Large Language Model (LLM). Eventually, we will outline how LLMs have changed our way of thinking about AI systems. As a result, we will talk about Generative AI Systems (GenAISys).

Oh and before we start, LLM is not equivalent to Generative AI. Generative AI is about all modalities, and LLMs are about processing some language. I am not a "terminology police", I am really flexible, but it is quite sad that *experts* nowadays repeat this nonsense. It is hurtful for the field and the industry. Alright, enough of being an old prick, and let us delve into LLMs and GenAISys!

## 11.2 Large Language Models

### 11.2.1 What Are Large Language Models (LLMs)?

#### 11.2.1.1 Natural Language Processing and Deep Learning

Natural language processing (NLP) is the field that focuses on building machines that can manipulate human language. NLP aims at developing methods and techniques for processing written language (i.e., text). The tasks of NLP (see Fig. 11.1) vary from text classification (e.g., sentiment analysis, spam detection), text correction (e.g., grammatical error correction), machine translation (e.g., translating using sequence-to-sequence models a.k.a. Seq2Seq), semantic analysis (e.g., topic modeling), text generation (e.g., chatbots), and text summarization, named entity recognition (NER) and information retrieval (IR) to question answering and chatbots. The main question though in NLP is about how to represent text such that it is useful for downstream tasks.

Classical NLP methods focused a lot on syntactic, i.e., grammar and sentence structures [1]. However, semantics is extremely important for proper communication and this remained a challenge for a long time. The first attempts to grasp the

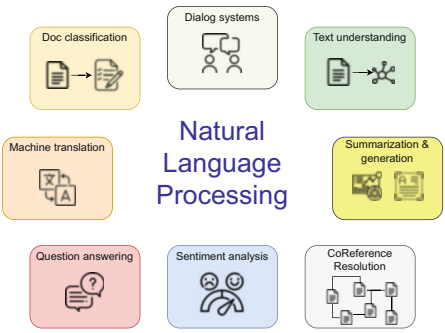



**Fig. 11.1** Selected most popular tasks in NLP.

meaning of a piece of text relied on formulating **tokenizers**, specialized modules that represented text in a machine-readable manner. The first tokenizers counted words (so-called **bag of words**) in a document or an n-gram (a sequence of n words), giving rise to a numerical representation of text. However, such a representation is dependent on the document length; therefore, it seems more appropriate to look at normalized quantities like term frequency (**tf**), namely, the number of occurrences of a word in a document divided by the number of words in a document, and inverse document frequency (**idf**) that corresponds to the importance of a term in the whole corpus, calculated as the logarithm of the number of documents in the corpus divided by the number of documents that include the term. Typically, the tf-idf representations are used. These representations provide a lot of information about documents and can be easily manipulated by machines. However, they are pretty limited due to chosen n-grams and heavily depend on the available corpus of documents. Simpler tokenizers replace characters (or n-grams of characters) with integers such that text could be treated as a sequence of numbers. These tokenizers do not provide any meaning of text but play a crucial role in contemporary NLP methods, as we will see shortly.

I think you feel what is coming, my curious reader. I can almost read your mind. What are you whispering? It would be amazing to have a method that can learn semantics from data. And, ideally, by applying neural networks? You are right! The big breakthrough in NLP came with the Word2Vec method [2] which allowed learning word embeddings from raw text by using neural networks for a given context. In some sense, this paper opened Pandora's box (but in a good sense) that led to the development of language models parameterized by neural networks. Since these neural networks consisted of millions of weights and later on (and nowadays) even billions of weights, many researchers and practitioners have started calling them Large Language Models to distinguish them from more classical language models.

#### 11.2.1.2 General Architectures of LLMs

Before we talk more about details like how to parameterize LLMs (i.e., what kind of neural networks to use) and how to learn them, let us briefly discuss their general architectures. No matter what, each LLM requires a tokenizer to turn text into numbers (e.g., integers) and an embedding that changes them eventually to real-valued vectors. Sometimes, both modules are treated as one (e.g., vectorizers in `scikit-learn`). A popular choice for a tokenizer these days is *byte pair encoding* which greedily merges commonly occurring sub-strings based on their frequency [3]. The embedding module serves only a single purpose, namely, to map a token represented as a one-hot vector to a real-valued vector of size $D$. Then, after processing embeddings using a neural network, the output must be de-tokenized to a string again.

In general, we can distinguish three types of LLMs:

1. **Encoders**: These LLMs take a piece of text (string) and return an encoding, i.e., a numerical representation of the input. Encoders can have access to the whole input at any point of processing, and they do not require any specific constraints.

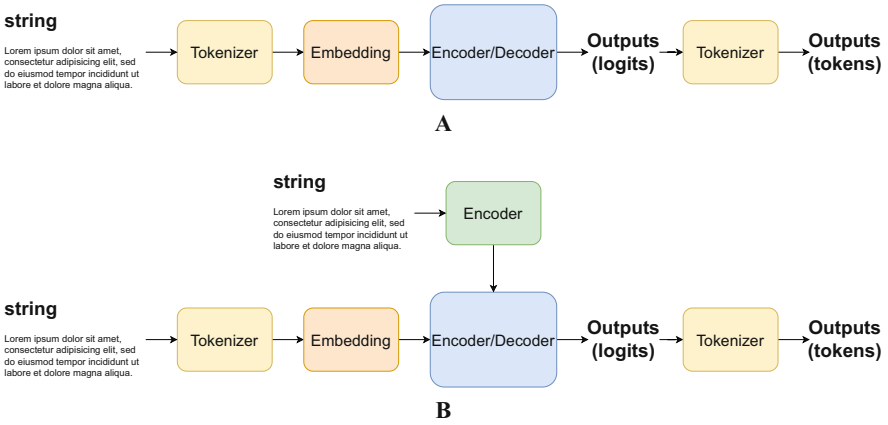



**Fig. 11.2** Diagrams for LLMs: (**a**) An unconditional LLM. (**b**) A conditional LLM.

They provide outputs in a single forward run both during training and at the inference time.

2. **Decoders**: This class of LLMs is pretty specific because they are used for *generating* new texts. They can be seen as autoregressive models, and, as such, neural networks used to parameterize them must be *causal*. For decoders, the sampling procedure is an iterative process; thus, it is typically pretty slow.
3. **Encoder-Decoders** and **Encoder-Encoders**: LLMs can be conditional, and then we need to combine an encoder processing conditioning and an encoder or a decoder that provides an encoding of input text or generates new text, respectively.

In Fig. 11.2a, you can see a schematic representation of an encoder or a decoder, and in Fig. 11.2b, there is an encoder-encoder(decoder) presented. Now, the question is how to parameterize LLMs.

#### 11.2.1.3 Parameterizations

The key component of any LLM is its parameterization. No matter if it is an encoder or a decoder, it is large because it uses hierarchical, (very) deep neural networks. But we cannot just use *any* neural network because we deal with text (i.e., sequences). As a result, a neural network must process vectors and, for decoders, output new text in the so-called causal manner (i.e., without looking "into the future").

If we go back to autoregressive models, we know that we can use recurrent neural networks (RNNs) and convolutional layers (CNNs) for processing text. RNNs seem to fit perfectly language modeling due to their intrinsic sequential structure. They were used by [4] to formulate one of the first RNN-based decoders and then RNN-based encoder-decoders [5]. As shown in [6], the combination of CNNs and RNNs indicated an improvement in terms of language modeling. However, this was no surprise because the first successful encoders were based entirely on convolutional layers [7] before they were used for decoders [8] and encoder-decoders [9]. However, RNN-based and CNN-based language models suffered from either forgetting or scaling issues.

The big breakthrough in LLMs has come with the introduction of **transformers** [10] and their utilization of (multihead) attention layers. In all fairness, transformers proposed a few important implementation tricks, such as multihead attention to learn multiple patterns in input tokens, layer normalization for preventing gradient blowing, and positional embeddings to ensure that tokens are treated as sequences and not as a set of tokens. Transformers turned out to be great at scaling up models, resulting in models with billions of weights. However, transformers require quadratic time and quadratic memory in the number of tokens (however, with FlashAttention [11], it becomes linear).

Recently, many people working on LLMs have focused on making them leaner (we can refer to those models as L3M: Lean Large Language Models). There are three main aspects to obtain L3M: (i) quantization of LLMs (i.e., fewer bits per weight that leads to less physical memory on disc), (ii) faster training, and (iii) faster inference.

**Quantization** is a long-standing problem in deep learning, and it poses new challenges for various architectures. For transformers, there have been a lot of successes like quantization-aware training with modified self-attention [12]. Recently, it was shown that using transformers with weights of linear layers taking values in $\{-1, 0, 1\}$ results in leaner methods (around $\times 3$ lower memory requirements and around $\times 2$ lower latency in ms) while achieving comparable performance as float16 models.

For faster training/inference, there were various methods proposed. For instance, FlashAttention focused on speeding up attention layers by accounting for I/O operations (reads and writes) between different levels of GPU memory [11]. A different approach rephrases the attention mechanism using a linear dot-product of kernel feature maps. The resulting *linear transformer* [13] makes further use of the associativity property of matrix products to reduce the quadratic complexity to linear complexity in the big-O notation of the length of the input sequence. However, a transformer-based language model is not all we need (and have)! Other classes of LLMs operate like RNNs at inference time and are fully parallelizable during training. Selective state space models (S3Ms) are built on top of state space models, i.e.,

**Table 11.1** A comparative analysis of various parameterizations of LLMs regarding time and memory complexity for both inference and training with the sequence length denoted by $N$. (Adapted from [17]).

| | Inference | | | Training | |
|---|---|---|---|---|---|
| Architecture | Time | Memory | Parallel | Time | Memory |
| RNNs | $O(1)$ | $O(1)$ | NO | $O(N)$ | $O(N)$ |
| Transformer | $O(N)$ | $O(N)$ | YES | $O(N^2)$ | $O(N)$ |
| Linear transformer | $O(1)$ | $O(1)$ | YES | $O(N)$ | $O(N)$ |
| S4M | $O(1)$ | $O(1)$ | YES | $O(N \log N)$ | $O(N)$ |
| RWKV/RetNet/Mamba | $O(1)$ | $O(1)$ | YES | $O(N)$ | $O(N)$ |

a linear dynamical system with hidden variables/states [14]. The adverb *selective* comes from the fact that S3Ms use an attention mechanism. One of the extensions of S3Ms, Mamba [15], outperformed some transformers. Other S3M-inspired models like Retentive Networks (RetNets) [16] or RWKV [17] successfully compete with the largest transformer-based models while maintaining linear training complexity and constant inference complexity. An interesting LLM, Jamba [18], is a combination of Mamba layers with transformer layers. Taking a hybrid approach poses a promising future direction.

Another important extension of current LLMs is based on the idea of mixture of experts (MoE) [19], a concept well-known in machine learning [20]. Instead of learning a single model, we train multiple LLMs (experts) that can specialize in certain topics. Then, for given input tokens, a router selects multiple experts, and the final outcome is calculated as a weighted average of the outcomes given by the experts. An example of an implementation of a MoE-LLM is mixtral of experts [21].

An overview of various parameterizations of LLMs is presented in Table 11.1.

### 11.2.2 Learning LLMs

As you can imagine, my curious reader, training of LLMs is probably something a bit more complicated than *just* LMs (or other models, especially the *small* ones). The answer is yes and no. *No* because, after all, they are models that either encode or generate, so nothing we have not dealt with. *Yes* because LLMs are *large* and we need *a lot* of data. Very often, LLMs are seen as **foundation models** (FMs) [22] that assume two phases of training:

**Training Procedure of FMs and LLMs**

1. **Pre-training**: This is the initial stage that aims at preparing an LLM for further tasks. A model is trained either by using the masked loss, $\ell_{\text{masked}}(\theta) = -\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}(\mathbf{x})}\mathbb{E}_M\left[\sum_{m\in M} \ln p_\theta(x_m \mid \mathbf{x}_{-m})\right]$, where $M$ indicates which tokens should be dropped from $\mathbf{x}$ and the goal is to reconstruct masked tokens, or by minimizing the negative log-likelihood $\ell_{\text{gen}}(\theta) = -\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}(\mathbf{x})}\left[\ln p_\theta(\mathbf{x})\right]$. The masked loss is used for encoders, while the negative log-likelihood is typically utilized by decoders. The pre-training stage involves a lot (and I really mean *a lot*!) of data because the goal of this stage is to train general patterns in data, e.g., grammar and co-occurrences of words, or a specific programming language.
2. **Fine-tuning**: A pre-trained model is further specialized on another dataset for a downstream task. For instance, an LLM can be trained on specific data, e.g., legal data to generate legal documents or a new programming language. However, the LLM can be also fine-tuned to carry out other tasks like text summarization, Q&A, text classification, sentiment analysis, etc. Depending on the task at hand, the LLM is optimized using either $\ell_{\text{masked}}(\theta)$ or $\ell_{\text{gen}}(\theta)$ or serves as a starting point for a new LLM for another task. In the former case, we typically optimize a different objective with an additional neural network, $\ell_{\text{pred}}(\theta, \phi) = -\mathbb{E}_{\mathbf{x},y\sim p_{\text{data}}(\mathbf{x},y)}\left[\ln p_{\theta,\phi}(\mathbf{x}, y)\right]$.

These two steps are quite general and fully depend on a given task at hand. For instance, the first generative pre-trained transformers (GPTs) [23] were pre-trained using the negative log-likelihood, or they were initialized by training with the masked loss like in bidirectional encoder representations from transformers (BERT) [24]. Eventually, GPTs were fine-tuned with the negative log-likelihood loss. It is also possible to combine various losses, e.g., $\ell_\alpha(\theta) = \ell(\theta) + \alpha\ell_{\text{masked}}(\theta)$, which could be seen as a penalized negative log-likelihood objective. This idea was utilized in pre-training LLMs for various problems at once [25] or, with $\alpha = 1$, for pre-training an LLM for molecules [26].

The problem with fine-tuning LLMs lies in the size of LLMs. Ideally, fine-tuning should be quick and cheap, but it is hard to achieve if we deal with models with billions of weights. A possible solution is a technique known as Low-Rank Adaptation (LoRA) [27]. The idea is relatively simple but crazily smart! Instead of updating a full-rank matrix of weights $\mathbf{W}$, which is kept fixed, we introduce a new set of learnable matrices during fine-tuning, $\mathbf{A}$ and $\mathbf{B}$, while keeping $\mathbf{W}$ frozen. As a result, the forward pass looks as follows:

$$\mathbf{h}_l = \mathbf{W}\mathbf{h}_{l-1} + \mathbf{B}\mathbf{A}\mathbf{h}_{l-1}. \tag{11.1}$$

**A Side Note**

LoRA typically updates only attention matrices since they pose the greatest bottleneck. However, there are various approaches to which components of transformers are updated using LoRA.

Where does the whole magic come from then? From sizes of the matrices! If $\mathbf{W}$ is a $D \times d$ matrix, then $\mathbf{B}$ is a $D \times k$ matrix and $\mathbf{A}$ is a $k \times d$ matrix, where $k \ll d$. As a result, we have a knob in the form of the dimensionality $k$ that controls the number of weights in the matrices $\mathbf{A}$ and $\mathbf{B}$, while the resulting matrix $\mathbf{BA}$ is of the same size as $\mathbf{W}$. As a result, after applying LoRA, only a small fraction of the original number of weights is updated during fine-tuning. Typically, enlarging an LLM with even fewer than 1% of new trainable weights is enough to achieve the same results as if the whole LLM is fine-tuned.

### 11.2.3 Famous LLMs

There is a plethora of LLMs, a few already mentioned here. I do not even dare to start pointing them out. In Table 11.2, I gather only (very subjectively!) selected LLMs.

### 11.2.4 Coding Up Our teenyGPT

Ufff... it was a lot of text, but what you can expect from a discussion about Large Language Models if not a lot of text? But without further ado, let us delve into some code! In the following, we will discuss our own GPT model we can call **teenyGPT**, an (extremely) tiny implementation of a decoder LLM. We focus on a small dataset of newspaper headlines and a simple tokenizer working at a character level. Our data consists of the batch dimension, the number of tokens, and the values, and our loss function is simply the negative log-likelihood:

```
class LossFun(nn.Module):
    def __init__(self,):
        super().__init__()

        self.loss = nn.NLLLoss(reduction='none')

    def forward(self, y_model, y_true, reduction='sum'):
        # y_model: B(atch) x T(okens) x V(alues)
        # y_true: B x T
        B, T, V = y_model.size()

        y_model = y_model.view(B * T, V)
        y_true = y_true.view(B * T,)

```

**Table 11.2** A list of selected LLMs with their corresponding types, sizes, and references/release dates.

| Name | Type | Size | Reference or release date |
|---|---|---|---|
| BERT | Encoder | 0.11B–0.34B | [24] |
| BioGPT | Decoder | 0.35B | [28] |
| Claude-3 | Decoder | 20B–~2000B | Mar 4, 2024 |
| Falcon | Decoder | 7B–180B | [29] |
| GPT-2 | Decoder | ~1.5B | [30] |
| GPT-3 | Decoder | ~175B | September, 2020 |
| GPT-4 | Decoder | ~1000B | March 14, 2023 |
| LLaMA-2 | Decoder | 7B–70B | [31] |
| LLaMA-3 | Decoder | 8B–70B | April 18, 2024 |
| Mamba | Decoder | 3B | [15] |
| Mistral | Decoder | 7B | September 27, 2023 |
| Mixtral | Decoder | 8×7B | [21] |
| PaLM | Decoder | 8B–540B | [32] |
| RoBERTa | Encoder | 0.125B–0.355B | [33] |
| RWKV-5 (Eagle) | Decoder | 0.4B–7B | [17] |
| RWKV-6 (Finch) | Decoder | 1.6B–3B | [17] |
| T5 | Encoder-Decoder | 0.06B–11B | [34] |
| XLNet | Decoder | 0.11B–0.34B | [35] |

```
        loss_matrix = self.loss(y_model, y_true) # B*T

        if reduction == 'sum':
            return torch.sum(loss_matrix)
        elif reduction == 'mean':
            loss_matrix = loss_matrix.view(B, T)
            return torch.mean(torch.sum(loss_matrix, 1))
        else:
            raise ValueError('Reduction could be either `sum` or
    `mean`.')
```

**Listing 11.1** A class for the loss function.

The essential component is the transformer block. We define them using the PyTorch implementation of multihead attention layers:

```
class TransformerBlock(nn.Module):
    def __init__(self, num_emb, num_neurons, num_heads=4):
        super().__init__()

        # hyperparams
```

```
        self.D = num_emb
        self.H = num_heads
        self.neurons = num_neurons

        # components
        self.msha = nn.MultiheadAttention(embed_dim=self.D,
    num_heads=self.H, batch_first=True)
        self.layer_norm1 = nn.LayerNorm(self.D)
        self.layer_norm2 = nn.LayerNorm(self.D)

        self.mlp = nn.Sequential(nn.Linear(self.D, self.neurons *
     self.D),
                                 nn.GELU(),
                                 nn.Linear(self.neurons * self.D,
    self.D))

    def forward(self, x, causal=True):
        # Multi-Head Self-Attention
        x_attn, _ = self.msha(x, x, x, is_causal=causal,
    attn_mask=torch.empty(1,1), need_weights=False)
        # LayerNorm
        x = self.layer_norm1(x_attn + x)
        # MLP
        x_mlp = self.mlp(x)
        # LayerNorm
        x = self.layer_norm2(x_mlp + x)

        return x
```

**Listing 11.2** A class for a transformer block.

Finally, we need to define our teenyGPT with a forward pass for the transformer and a sampling procedure. Additionally, we can define an auxiliary metric, top one reconstruction accuracy, which takes the most probable token and uses it to check whether it is the same as the input token:

```
class teenyGPT(nn.Module):
    def __init__(self, num_tokens, num_token_vals, num_emb,
    num_neurons, num_heads=2, dropout_prob=0.1, num_blocks=10,
    device='cpu'):
        super().__init__()

        # Remember, always credit the author, even if it's you ;)
        print('teenyGPT by JT.')

        # hyperparams
        self.device = device
        self.num_tokens = num_tokens
        self.num_token_vals = num_token_vals
        self.num_emb = num_emb
        self.num_blocks = num_blocks

        # embedding layer
        self.embedding = torch.nn.Embedding(num_token_vals,
    num_emb)
```

```
        # positional embedding
        self.positional_embedding = nn.Embedding(num_tokens, num_emb)

        # transformer blocks
        self.transformer_blocks = nn.ModuleList()
        for _ in range(num_blocks):
            self.transformer_blocks.append(TransformerBlock(num_emb=num_emb, num_neurons=num_neurons, num_heads=num_heads))

        # output layer (logits + softmax)
        self.logits = nn.Sequential(nn.Linear(num_emb, num_token_vals))

        # dropout layer
        self.dropout = nn.Dropout(dropout_prob)

        # loss function
        self.loss_fun = LossFun()

    def transformer_forward(self, x, causal=True, temperature=1.0):
        # x: B(atch) x T(okens)
        # embedding of tokens
        x = self.embedding(x) # B x T x D
        # embedding of positions
        pos = torch.arange(0, x.shape[1], dtype=torch.long).unsqueeze(0).to(self.device)
        pos_emb = self.positional_embedding(pos)
        # dropout of embedding of inputs
        x = self.dropout(x + pos_emb)

        # transformer blocks
        for i in range(self.num_blocks):
            x = self.transformer_blocks[i](x)

        # output logits
        out = self.logits(x)

        return F.log_softmax(out/temperature, 2)

    @torch.no_grad()
    def sample(self, batch_size=4, temperature=1.0):
        x_seq = np.asarray([[self.num_token_vals - 1] for i in range(batch_size)])

        # sample next tokens
        for i in range(self.num_tokens-1):
            xx = torch.tensor(x_seq, dtype=torch.long, device=self.device)
            # process x and calculate log_softmax
```

```
            x_log_probs = self.transformer_forward(xx,
    temperature=temperature)
            # sample i-th tokens
            x_i_sample = torch.multinomial(torch.exp(x_log_probs
    [:,i]), 1).to(self.device)
            # update the batch with new samples
            x_seq = np.concatenate((x_seq, x_i_sample.to('cpu').
    detach().numpy()), 1)

        return x_seq

    @torch.no_grad()
    def top1_rec(self, x, causal=True):
        x_prob = torch.exp(self.transformer_forward(x, causal=
    True))[:,:-1,:].contiguous()
        _, x_rec_max = torch.max(x_prob, dim=2)
        return torch.sum(torch.mean((x_rec_max.float() == x
    [:,1:].float().to(device)).float(), 1).float())

    def forward(self, x, causal=True, temperature=1.0, reduction=
    'mean'):
        # get log-probabilities
        log_prob = self.transformer_forward(x, causal=causal,
    temperature=temperature)

        return self.loss_fun(log_prob[:,:-1].contiguous(), x
    [:,1:].contiguous(), reduction=reduction)
```

**Listing 11.3** A class for a teenyGPT.

For 128 neurons in MLPs, 8 attention heads, 4 transformer blocks, and the embedding size equal 32, we get a model with about 1 million weights and the performance as in Fig. 11.3, Tables 11.3, and 11.4.

As you can notice, my curious reader, the teeny LLM trains successfully, even with only a small amount of data, and around 1 million of weights! The negative

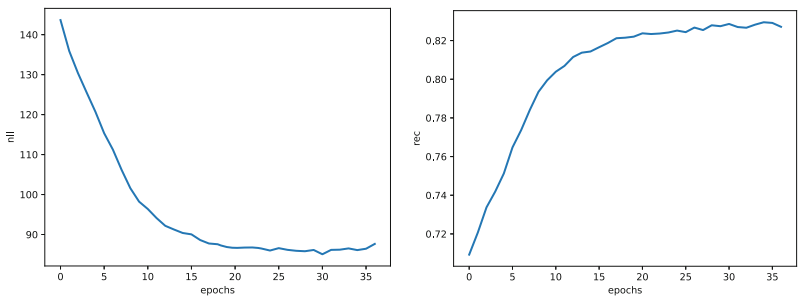



**Fig. 11.3** Examples of results: (**a**) The negative log-likelihood calculated on the validation set. (**b**) The top one reconstruction accuracy calculated on the validation set.

**Table 11.3** Examples of sampled headlines with temperature 0.1.

| No. | Generation |
|---|---|
| 1 | British announces star market star market contract coronavirus case |
| 2 | British announces show announces set series |
| 3 | Breaking say court coronavirus test positive |
| 4 | Coronavirus death trump court star reade say say announcement say |
| 5 | Coronavirus death test positive coronavirus case state state state state state state court |
| 6 | Coronavirus case start star reade country test positive |
| 7 | British announces set coronavirus test positive |
| 8 | Coronavirus case show death committee |
| 9 | Michigan pro recovered country coronavirus test positive |
| 10 | Coronavirus case star set state state state state state state state state allegation |

**Table 11.4** Examples of sampled headlines with temperature 1.0.

| No. | Generation |
|---|---|
| 1 | Waterment share brand fabst dailey available hystering |
| 2 | pm rate murder test personn hirr kannish reported homeless tested decision |
| 3 | Torotteney headlines sentencer everyone |
| 4 | Milamican combinize give test pair catch attack |
| 5 | ugc harry first bitcoin boat wuhan chip man end masking precain |
| 6 | True fansed andrap new full forecasts |
| 7 | Galaxy nigeria check police liverpool dam new deal threatens order |
| 8 | Senator addeedly discharge tonie |
| 9 | Grime everdille get improves reveals fan talk new revela false |
| 10 | UK coronavirus delivery due five aviversau despite return horror year |

log-likelihood drops nicely (Fig. 11.3a), while the reconstruction accuracy grows (Fig. 11.3b).

Comparing samples in Tables 11.3 and 11.4, we can clearly see that being too stochastic (i.e., taking too small temperature) results in nonsensical headlines. However, overall, even though the model does not work perfectly, it learned to combine characters in such a way they constitute words (most of the time), and some headlines make even sense.

### 11.2.5 Other (Selected) Topics on LLMs

There are so many topics in LLMs at the moment, that it is nearly impossible to cover them all. In the following, I stress out some developments that are important and require a separate write-up, but I leave these to others. Here are some highlights of the LLM community:

- **Prompt engineering**: Prompt engineering is a bit tricky business, and I mention it only for completeness of trendy topics in LLMs. However, my personal opinion is that prompt engineering is rather pseudo-engineering, a bunch of useful tricks for currently trained LLMs. From a practical point of view, it is useful to share

commonly used practices on how to prompt an LLM. Nevertheless, in the long run, prompt engineering as we know it right now could be completely useless. Either way, my general advice is this: Prompt wisely, be precise, be nice (even though you talk to a machine!), instruct an LLM, and take whatever an LLM gives you with a pinch of salt. After all, LLMs are models and they can be (often are!) wrong.

- **Instructing and human feedback**: A big breakthrough in LLMs came with instructing them and giving them human feedback [36]. The first model, a predecessor of ChatGPT, InstructGPT, was based on the idea of fine-tuning a GPT model using a dataset of rankings of model outputs provided by human evaluators. This approach was dubbed reinforcement learning from human feedback (RLHF), and during fine-tuning, first, a reward model is trained on the rankings, and then the reward model is used to assess generations according to the Proximal Policy Optimization (PPO) algorithm [37]. Later on, it was noticed that there is no need to train a separate reward model as a proxy to a human evaluator but it is possible to formulate the probability of preference of one generation over the other by using the LLM itself. This is the idea (and math, a really neat math) behind Direct Preference Optimization (DPO) [38].
- **In-context learning**: A great and fascinating capability of LLMs is so-called in-context learning [39]. The idea relies on providing a few examples in the prompt to an LLM together with a query so that the LLM can figure out the answer from a very limited amount of data. To strengthen the LLM to be capable of in-context learning, it is important to cultivate it during training (e.g., during fine-tuning).
- **Knowledge graphs + LLMs**: Knowledge graphs (KGs) present a way of structuring data and relationships among them. However, their main drawbacks are lack of generalizability (if facts are missing, KGs are unable to provide any sort of approximate solution) and lack of language understanding. An enhancement of KGs with LLMs or LLMs with KGs presents an interesting solution since LLMs are flexible but they lack proper grounding in facts [40].
- **Adversarial attacks**: Similar to images, it is also possible to formulate adversarial attacks for LLMs [41]. By proper prompting, it is possible to retrieve information that could be harmful to others but also to obtain original training data (e.g., personal data).
- **LLMs for programming**: There are multiple LLMs for other kinds of languages. For instance, programming languages are a perfect fit for training your own coding tools for enhancing programmers. Thanks to Generative AI, the phrase *no-code programming* has attracted a lot of attention these days. Some examples of important LLMs for coding are CodeBERT [42] and GitHub Copilot [43].

These topics constitute only the tip of the iceberg. The speed at which LLMs evolve is unprecedented. There are multiple extremely interesting new concepts, but, in my opinion, some directions are definitely overhyped. Nevertheless, Generative AI owes a lot to LLMs because they have changed the AI landscape and brought generative modeling to a completely new level.

## 11.3 Generative AI Systems

### 11.3.1 Introduction

LLMs opened doors to the next level of applications, allowing communication with machines using natural language. This is definitely a technological jump but also a great leap in the way we think about AI. The recent developments in GenAI, e.g., in the GPT family (GPT-4V: GPT-4 with Vision) or in the Google-based models (e.g., Gemini), brought multimodal learning in DL back and showed great promise in using LLMs as multimodal systems. Currently developed LLMs are capable of processing various data modalities (not only text but also images, audio, and video) and generating different data types! As a result, it becomes more appropriate to talk about GenAI Systems rather than LLMs. Text (or natural language) constitutes communication means, instead of predefined formal protocols and modality encoders that play the role of I/O interfaces for processing various data sources. However, we can go even further and can attach a database (or, in particular, a knowledge graph) or external specialized tools (e.g., a calculator, an app for finding a route from A to B) that communicate with the system through a module for information retrieval and storage. An example of such a system is a tool-augmented Seq2Seq model proposed by [44] that uses a fine-tuned T5 LLM as a backbone to deal with inputs and outputs to and from some external tools. A more specific use of an LLM with an external tool is a system for chemistry [45] that is based on external databases and tools for providing reliable answers and drawing molecules, among other features.

In the following, we will look into various systems using Generative AI modules that we will refer to as Generative AI Systems (GenAISys). It is interesting to discuss these systems at a high level since we are far from the full understanding of their building blocks (i.e., neural networks or even their simplest units). As the community, we have a good *feeling* that the strength of DL lies in its **compositionality** [46, 47, 59], but we still need to accept many claims to be hypotheses. Nevertheless, it should not stop us from discussing *compositionality* at a higher level, similar to control theory, or, more broadly, systems science and engineering [48]. Let us do that then!

### 11.3.2 GenAISys: A General Architecture

Talking about systems immediately brings to mind multiple modules, some components combined together to solve a set of goals. The discussion might be a bit blurry at times because the question is the following: Is a composition of two functions with a single line of code each a system already? Yes, because systems are composed of distinguished parts or, to put it differently, systems are complex things that can be assembled out of simpler parts. An interesting perspective on systems and compositionality provides category theory [49, 59] if you are interested in it, my curious reader. However, here we will continue in a more intuitive manner.

We already mentioned what GenAI Systems are built of, so let us make it more specific then. A GenAISys consists of the following parts:

- **Data encoders (DEs)**: All input (raw) data are processed using models that encode them into tensors. Typically, these encoders are pre-trained and kept non-trained (*frozen*). We can use any foundation model (FM) as an encoder, e.g., BERT for text encoding, ConvNeXT (without the predictive head) for image encoding, etc. Encoders could be composed of other modules (encoders) as well. For instance, speech could be transformed into text (e.g., using Whisper), and then text could be represented as a tensor (e.g., using BERT). Data encoders play a crucial role in formulating GenAISys, and their introduction was a catalyzer for the development of GenAI Systems as we know it now.
- **GenAI Model (GeM)**: The *central unit* of a GenAISys is a GenAI model (e.g., an LLM) that processes (encoded) input data, communicates with external *memory* (a database) and external tools, and generates an output. It is the core part of the whole architecture. It can have its own short-term memory (a cache), e.g., in the form of available input tokens, and built-in instructions for communicating with other components. Moreover, it can be equipped with other (sub)models carrying out other tasks, e.g., named entity recognition (NER) for detecting specific instructions in inputs. In the simplest form, it could be trained in such a way that specific instructions in natural language trigger specific actions like running a calculator. Overall, a GeM can be a complicated system by itself, comprising multiple modules including separate models.
- **Retrieval/Storage module (R/S)**: This module serves an extremely important function to assist GeM with retrieving facts (*long-term memory*) but also utilizing inputs processed by DEs. Additionally, this module can have its own models (e.g., BERT to embed a part of the input text of an internal instruction) and sets of instructions (e.g., routing algorithms).

A general scheme of GenAISys is presented in Fig. 11.4. Please keep in mind that this is a simplified architecture that highlights only the main components. Each block could consist of multiple submodules, models, and procedures (algorithms). It is also no surprise that the diagram looks like the general architecture of a computer. After all, such architectures are quite *natural*. However, unlike a computer, GenAISys consists of trainable components and, as a result, is extremely flexible in the sense of its functionality.

### 11.3.3 Training

Training of a GenAISys is nontrivial since all components are, in principle at least, trainable. One can imagine training GenAISys end to end; however, it is infeasible for currently used hardware. Nowadays, a widely applied solution is to take advantage of foundation models that are pre-trained separately to formulate DEs and an R/S. Eventually, only a GeM is trained. But, with the improvement of hardware and

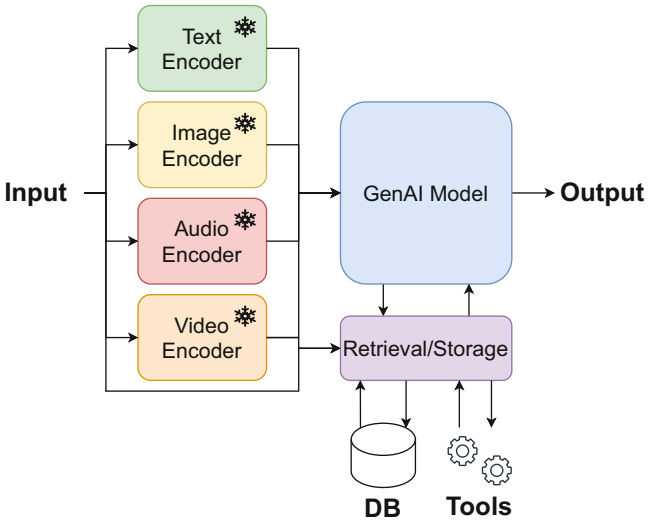



**Fig. 11.4** A general architecture of a Generative AI System with encoders for various modalities, a retrieval/storage module for accessing external tools and databases, and a central generative AI model producing new content (output). The snowflake icon represents that a module is "frozen" (i.e., already trained).

probably with new training schemes, GenAISys will be trained in a better way and, eventually, will get better through the utilization of multiple data modalities at the same time.

**A Side Note**

The general scheme in Fig. 11.4 resembles a map of the functions of the human brain. At this point, we are definitely close to applying a coordinate descent algorithm that optimizes the whole system one modality (or a block) at a time. This would keep the complexity at a relatively low level, no greater than what we have right now. However, the main issue is the potential **forgetting** of previously learned representations. We do not have fully satisfactory **continual learning** methods, but we are making progress [50, 51]. Once this problem is solved, we will get a real breakthrough in AI!

### 11.3.4 Examples of GenAISys

Following the general scheme for GenAISys in Fig. 11.4, we can indicate how currently used Generative AI approaches fit this scheme. We focus on Large Vision Models and a specific LLM-based solution for reliable text generation. There are

many more examples, but once you get familiar with the most successful ones, you will definitely be able to see this scheme in other models, my curious reader.

#### 11.3.4.1 RAGs

The main drawback of decoder-based LLMs is hallucinating when a prompt takes the model away from its "comfort zone" (i.e., from regions where training data lie in the representation space). This could be fixed to some degree with proper fine-tuning; however, LLMs tend to make up or skip some facts. Since their responses are typically very colorful with distinctive and unusual wording, human beings can miss some deficiencies and false statements. Therefore, sometimes, I refer to LLMs as *Lying* Language Models. In some applications, there is no place for fake news, or, in general, outcomes cannot be untrustworthy. For instance, in health-related situations, medicine, drug discovery, or manufacturing (e.g., diagnostics), there is simply no space for made-up facts. Therefore, even though generative LLMs are so hyped, they do not pose a possible solution due to their high risk of hallucinating.

A huge breakthrough, especially in real-life applications, came with retrieval-augmented generations (RAGs) [52]. The idea is based on utilizing two LLMs, an encoder-LLM and a decoder-LLM, and a database of texts. The encoder-LLM is used in two ways: (i) for embedding all texts in the database and (ii) for embedding an incoming query. For a new query, the closest documents are picked based on the distance between the embedding of the query and the embeddings of the documents in the database. Eventually, the closest documents, together with the query, are passed to the decoder-LLM to generate an outcome. Since the outcome is based on the decoder-LLM and real documents, there is a much lower chance of hallucinations. Moreover, with a bit of tweaking around, the RAG could rely heavily on facts provided during the retrieval stage.

The diagram for a RAG is presented in Fig. 11.5, and it corresponds very closely to the general scheme of GenAISys in Fig. 11.4 where the decoder-LLM is the GeM and the encoder-LLM is the DE, and it is also used as a part of the R/S module.

#### 11.3.4.2 Speech2Txt

A great example of a GenAISys for transforming speech to text is Whisper [53], an encoder-decoder transformer with a specific form of the encoder that first represents raw speech (audio) using a log-magnitude Mel spectrogram before being fed to an encoder-transformer for processing audio signal and then to a decoder-transformer for generating text. The model is an automatic speech recognition system with 39M weights (a tiny version) to even 1.55B weights (a large version). It was trained on 680,000 hours of multilingual and multitask supervised data collected from the web. This model achieved SOTA performance at the time of its release, and still it remains one of the top Speech2Txt models available. The tiny version could even deployed on edge devices. Whisper is a great example of how GenAISys can be formulated

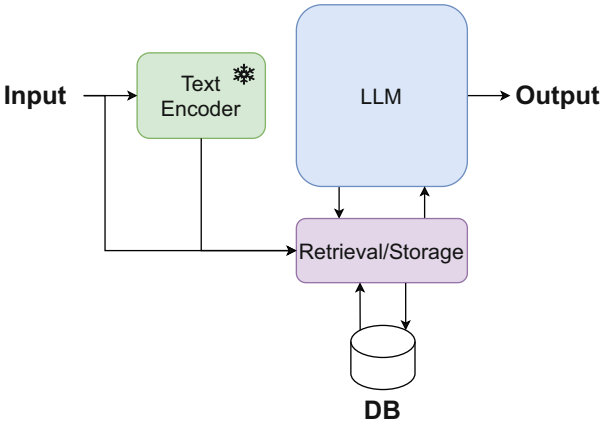



**Fig. 11.5** A schematic representation of a retrieval-augmented generation (RAG) architecture.

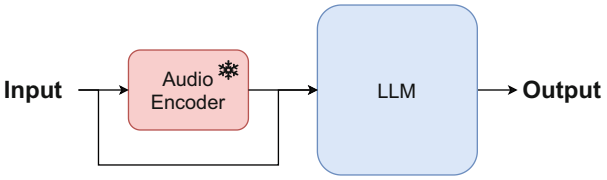



**Fig. 11.6** A schematic representation of a Speech2Txt architecture.

and how important it is to combine various components together for more advanced tasks like automatic speech recognition. The Speech2Txt diagram is presented in Fig. 11.6.

#### 11.3.4.3 Large Vision Models (LVMs)

Beside LLMs, Large Vision Models (LVMs) are perfect examples of GenAISys. There are many models that fall under the umbrella of Img2Img or Img2Txt, but the most popular LVMs these days are Txt2Img. Since the original paper on latent diffusion models [54], the resulting models like Stable Diffusion 2 and very recent Stable Diffusion 3, are widely used for generating images for a given prompt. Latent diffusion models (Stable Diffusion) or Dalle 2 [55] with a diffusion-based prior fit perfectly a scheme in Fig. 11.7a. Comparing these LVMs to a general GenAISys, a text encoder and an image encoder (for either training or reconstruction) are DEs, while a combination of a diffusion model and a decoder is a GeM. These models do not use any database or external tools; however, it is possible to use those to modify images somehow.

Another example of an LVM in the form of Txt2Img is ImaGen [56] which uses a T5-based text encoder and a diffusion model together with superresolution

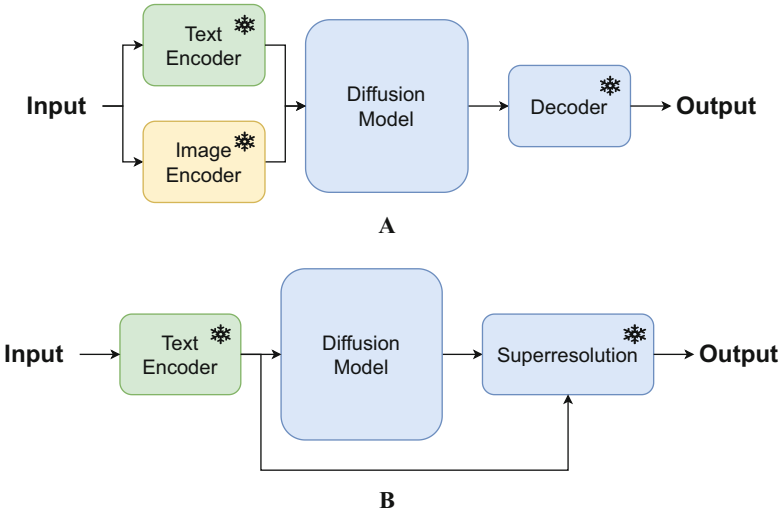



**Fig. 11.7** Examples of LVMs: (**a**) Stable Diffusion (i.e., latent diffusion) (**b**) ImaGen.

blocks. The corresponding architecture is presented in Fig. 11.7b (the superresolution module consists of multiple steps, going from 64×64 images to 1024×1024 images). Again, this is a complicated GenAISys even though it is composed of three blocks, but it has about 13B weights (11B weights for T5, about 2B weights for a UNet used in the diffusion model and the superresolution module) which is a large model in terms of the number of weights.

### 11.3.5 The Future of AI Is GenAISys

The idea of using LLMs as a backbone for operating systems and agents as applications has attracted a lot of attention [57]. Here, we consider general systems with various GenAI-based components, not only LLM-based compartments. Either way, moving toward GenAI-based (operating) systems seems like the future and the next step of cloud-based systems. Indeed, GenAISys can be deployed locally but also in a cloud server or as a hybrid (e.g., a GeM, a cache storage, and DEs are local but external tools and storage are in a cloud). The last option can be especially appealing for manufacturing since all real-life operations must be executed in real time, while data storage and other operations are carried out by external services (or agents).

Another idea that is pretty hyped these days is Agentic AI, i.e., the development of GenAISys-based agents operating autonomously. This idea is pushed by many Big Tech players. For instance, Microsoft proposed a framework for conversational LLM-based agents called AutoGen [58]. OpenAI sees their chatbots (including ChatGPT) as agents, and by equipping them with various tools and features, they could serve as **copilots** (i.e., assisting a human operator by proposing partial solutions) or **autopilots**

(i.e., assisting human operators by proposing complete solutions). The analogy here corresponds to controlling a plane, and a copilot helps to stabilize a flight, while an autopilot takes care of flying. In both cases, a human pilot can take over at any point.

My curious reader, I hope I managed to present you with a wonderful world of (deep) generative modeling and convince you that Generative AI Systems are inevitably next steps in the evolution of AI. I dare even say that they will not only assist in many jobs, ranging from office jobs, healthcare, and education to the industry like manufacturing but will also lead us to formulate the first *true* AI. There are many other aspects like embodied AI, but in my (not so humble) opinion, GenAISys is *a must* to create an artificial brain for embodied or not machine. But we will see what the distant future (in the AI world, it means the next 6 to 12 months) will bring. Either way, I am sure we live in extremely fascinating times, totally unprecedented.

## References


1. N Chomsky. Aspects of the theory of syntax. *Special technical report. Research laboratory of electronics. Massachusetts institute of technology*, (11), 1965.
2. Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In C.J. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26, 2013.
3. Philip Gage. A new algorithm for data compression. *The C Users Journal*, 12(2):23–38, 1994.
4. Tomás Mikolov, Martin Karafiát, Lukás Burget, Jan Cernocký, and Sanjeev Khudanpur. Recurrent neural network based language model. In Takao Kobayashi, Keikichi Hirose, and Satoshi Nakamura, editors, *INTERSPEECH 2010, 11th Annual Conference of the International Speech Communication Association, Makuhari, Chiba, Japan, September 26–30, 2010*, pages 1045–1048. ISCA, 2010.
5. Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.
6. Rafal Jozefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410*, 2016.
7. Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. A convolutional neural network for modelling sentences. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*, pages 212–217. Association for Computational Linguistics, 2014.
8. Yann N Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. In *International conference on machine learning*, pages 933–941. PMLR, 2017.

9. Nal Kalchbrenner, Lasse Espeholt, Karen Simonyan, Aaron van den Oord, Alex Graves, and Koray Kavukcuoglu. Neural machine translation in linear time. *arXiv preprint arXiv:1610.10099*, 2016.
10. Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.
11. Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.
12. Yelysei Bondarenko, Markus Nagel, and Tijmen Blankevoort. Quantizable transformers: Removing outliers by helping attention heads do nothing. *Advances in Neural Information Processing Systems*, 36, 2024.
13. Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are RNNs: Fast autoregressive transformers with linear attention. In *International conference on machine learning*, pages 5156–5165. PMLR, 2020.
14. Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces. *arXiv preprint arXiv:2111.00396*, 2021.
15. Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.
16. Yutao Sun, Li Dong, Shaohan Huang, Shuming Ma, Yuqing Xia, Jilong Xue, Jianyong Wang, and Furu Wei. Retentive network: A successor to transformer for large language models, 2023.
17. Bo Peng, Daniel Goldstein, Quentin Anthony, Alon Albalak, Eric Alcaide, Stella Biderman, Eugene Cheah, Xingjian Du, Teddy Ferdinan, Haowen Hou, Przemysław Kazienko, Kranthi Kiran GV, Jan Kocoń, Bartłomiej Koptyra, Satyapriya Krishna, Ronald McClelland Jr. au2, Niklas Muennighoff, Fares Obeid, Atsushi Saito, Guangyu Song, Haoqin Tu, Stanisław Woźniak, Ruichong Zhang, Bingchen Zhao, Qihang Zhao, Peng Zhou, Jian Zhu, and Rui-Jie Zhu. Eagle and finch: Rwkv with matrix-valued states and dynamic recurrence, 2024.
18. Opher Lieber, Barak Lenz, Hofit Bata, Gal Cohen, Jhonathan Osin, Itay Dalmedigos, Erez Safahi, Shaked Meirom, Yonatan Belinkov, Shai Shalev-Shwartz, Omri Abend, Raz Alon, Tomer Asida, Amir Bergman, Roman Glozman, Michael Gokhman, Avashalom Manevich, Nir Ratner, Noam Rozen, Erez Shwartz, Mor Zusman, and Yoav Shoham. Jamba: A Hybrid Transformer-Mamba Language Model, 2024.
19. William Fedus, Jeff Dean, and Barret Zoph. A review of sparse expert models in deep learning. *arXiv preprint arXiv:2209.01667*, 2022.
20. Robert A Jacobs, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87, 1991.
21. Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. Mixtral of experts. *arXiv preprint arXiv:2401.04088*, 2024.

22. Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.
23. Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018.
24. Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.
25. Li Dong, Nan Yang, Wenhui Wang, Furu Wei, Xiaodong Liu, Yu Wang, Jianfeng Gao, Ming Zhou, and Hsiao-Wuen Hon. Unified language model pre-training for natural language understanding and generation. *Advances in neural information processing systems*, 32, 2019.
26. Adam Izdebski, Ewelina Weglarz-Tomczak, Ewa Szczurek, and Jakub M Tomczak. De novo drug design with joint transformers. *arXiv preprint arXiv:2310.02066*, 2023.
27. Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.
28. Renqian Luo, Liai Sun, Yingce Xia, Tao Qin, Sheng Zhang, Hoifung Poon, and Tie-Yan Liu. BioGPT: generative pre-trained transformer for biomedical text generation and mining. *Briefings in Bioinformatics*, 23(6):bbac409, 09 2022.
29. Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Mérouane Debbah, Étienne Goffinet, Daniel Hesslow, Julien Launay, Quentin Malartic, et al. The falcon series of open language models. *arXiv preprint arXiv:2311.16867*, 2023.
30. Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.
31. Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023.

32. Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.
33. Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019.
34. Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.
35. Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.
36. Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 27730–27744, 2022.
37. John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.
38. Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.
39. Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. A survey on in-context learning. *arXiv preprint arXiv:2301.00234*, 2022.
40. Shirui Pan, Linhao Luo, Yufei Wang, Chen Chen, Jiapu Wang, and Xindong Wu. Unifying large language models and knowledge graphs: A roadmap. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–20, 2024.
41. Andy Zou, Zifan Wang, Nicholas Carlini, Milad Nasr, J. Zico Kolter, and Matt Fredrikson. Universal and transferable adversarial attacks on aligned language models, 2023.
42. Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

43. Github copilot – your ai pair programmer. https://copilot.github.com/. Accessed: 2024-05-01.
44. Aaron Parisi, Yao Zhao, and Noah Fiedel. Talm: Tool augmented language models. *arXiv preprint arXiv:2205.12255*, 2022.
45. Andres M Bran, Sam Cox, Oliver Schilter, Carlo Baldassari, Andrew White, and Philippe Schwaller. Augmenting large language models with chemistry tools. In *NeurIPS 2023 AI for Science Workshop*, 2023.
46. Leonard Bereska and Efstratios Gavves. Mechanistic interpretability for ai safety–a review. *arXiv preprint arXiv:2404.14082*, 2024.
47. Jerry Swan, Eric Nivel, Neel Kant, Jules Hedges, Timothy Atkinson, and Bas Steunebrink. *The Road to General Intelligence*. Springer Nature, 2022.
48. Zdzislaw Bubnicki et al. *Modern control theory*, volume 2005925392. Springer, 2005.
49. Brendan Fong and David I Spivak. Seven sketches in compositionality: An invitation to applied category theory. *arXiv preprint arXiv:1803.05316*, 2018.
50. Martin Mundt, Yongwon Hong, Iuliia Pliushch, and Visvanathan Ramesh. A wholistic view of continual learning with deep neural networks: Forgotten lessons and the bridge to active and open world learning. *Neural Networks*, 160:306–336, 2023.
51. Liyuan Wang, Xingxing Zhang, Hang Su, and Jun Zhu. A comprehensive survey of continual learning: Theory, method and application. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2024.
52. Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.
53. Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision. In *International Conference on Machine Learning*, pages 28492–28518. PMLR, 2023.
54. Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10684–10695, 2022.
55. Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. Hierarchical text-conditional image generation with clip latents. *arXiv preprint arXiv:2204.06125*, 1(2):3, 2022.
56. Chitwan Saharia, Jonathan Ho, William Chan, Tim Salimans, David J Fleet, and Mohammad Norouzi. Image super-resolution via iterative refinement. *arXiv preprint arXiv:2104.07636*, 2021.
57. Yingqiang Ge, Yujie Ren, Wenyue Hua, Shuyuan Xu, Juntao Tan, and Yongfeng Zhang. LLM as OS, agents as apps: Envisioning AIOS, agents and the AIOS-agent ecosystem. *arXiv e-prints*, pages arXiv–2312, 2023.

58. Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. Autogen: Enabling next-gen LLM applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*, 2023.
59. Bakirtzis Georgios, Fleming Cody H, and Vasilakopoulou Christina. Categorical semantics of cyber-physical systems theory. *ACM Transactions on Cyber-Physical Systems*, 5(3):1–32, 2021.

# Appendix A
# Useful Facts from Algebra and Calculus

## A.1 Norms and Inner Products

### *A.1.1 Norm Definition*

Norm is a function $\|\cdot\| : \mathbb{X} \to \mathbb{R}_+$ with the following properties:

1. $\|\mathbf{x}\| = 0 \Leftrightarrow \mathbf{x} = \mathbf{0}$,
2. $\|\alpha\mathbf{x}\| = |\alpha|\|\mathbf{x}\|$, where $\alpha \in \mathbb{R}$,
3. $\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$.

### *A.1.2 Inner Product Definition*

The inner product is a function $\langle\cdot,\cdot\rangle : \mathbb{X} \times \mathbb{X} \to \mathbb{R}$ with the following properties:

1. $\langle\mathbf{x}, \mathbf{x}\rangle \geq 0$,
2. $\langle\mathbf{x}, \mathbf{y}\rangle = \langle\mathbf{y}, \mathbf{x}\rangle$,
3. $\langle\alpha\mathbf{x}, \mathbf{y}\rangle = \alpha\langle\mathbf{x}, \mathbf{y}\rangle$,
4. $\langle\mathbf{x} + \mathbf{z}, \mathbf{y}\rangle = \langle\mathbf{x}, \mathbf{y}\rangle + \langle\mathbf{z}, \mathbf{y}\rangle$.

### *A.1.3 Chosen Properties of Norm and Inner Product*

- $\langle\mathbf{x}, \mathbf{x}\rangle = \|\mathbf{x}\|^2$,
- $|\langle\mathbf{x}, \mathbf{y}\rangle| \leq \|\mathbf{x}\|\|\mathbf{y}\|$ (for a vector in $\mathbb{R}^D$ $\langle\mathbf{x}, \mathbf{y}\rangle = \|\mathbf{x}\|\|\mathbf{y}\| \cos\theta$),
- $\|\mathbf{x} + \mathbf{y}\|^2 = \|\mathbf{x}\|^2 + 2\langle\mathbf{x}, \mathbf{y}\rangle + \|\mathbf{y}\|^2$.
- $\|\mathbf{x} - \mathbf{y}\|^2 = \|\mathbf{x}\|^2 - 2\langle\mathbf{x}, \mathbf{y}\rangle + \|\mathbf{y}\|^2$.



J. M. Tomczak, *Deep Generative Modeling*,
https://doi.org/10.1007/978-3-031-64087-2

## A.2 Matrix Calculus

### A.2.1 Liner Dependency

Let $\phi_m$ be a nonlinear transformation and $\mathbf{x} \in \mathbf{R}^M$. A linear product of these two vectors is:

$$\begin{aligned}\boldsymbol{\phi}(\mathbf{x})^{\mathrm{T}}\mathbf{w} &= w_0\phi_0(\mathbf{x}) + w_1\phi_1(\mathbf{x}) + \ldots + w_{M-1}\phi_{M-1}(\mathbf{x}) \\ &= \sum_{m=0}^{M-1} w_m\phi_m(\mathbf{x}),\end{aligned}$$

where $\mathbf{w} = (w_0\ w_1 \ldots w_{M-1})^{\mathrm{T}}$, $\boldsymbol{\phi}(\mathbf{x}) = (\phi_0(\mathbf{x})\ \phi_1(\mathbf{x}) \ldots \phi_{M-1}(\mathbf{x}))^{\mathrm{T}}$.

### A.2.2 Orthogonal and Orthonormal Vectors

Vectors $\mathbf{x}$ and $\mathbf{y}$ are orthogonal vectors if $\langle \mathbf{x}, \mathbf{y} \rangle = 0$. Additionally, if $\|\mathbf{x}\| = \|\mathbf{y}\| = 1$, these vectors are called orthonormal.

### A.2.3 Chosen Properties of Matrix Calculus

- $(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$.
- $(\mathbf{AB})^{\mathrm{T}} = \mathbf{B}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}$.
- Matrix $\mathbf{A}$ is positive definite $\Leftrightarrow$ for all vectors such that $\mathbf{x} \neq 0$ the following inequality holds true $\mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x} > 0$.
- $\nabla_{\mathbf{x}}\mathbf{x}^{\mathrm{T}}\mathbf{x} = 2\mathbf{x}$.
- $\nabla_{\mathbf{x}}\|\mathbf{W}^{\frac{1}{2}}(\mathbf{b} - \mathbf{A}\mathbf{x})\|_2^2 = -2\mathbf{A}^{\mathrm{T}}\mathbf{W}(\mathbf{b} - \mathbf{A}\mathbf{x})$, where $\mathbf{W}$ is a symmetric matrix.

For given vectors $\mathbf{x}$, $\mathbf{y}$ and a matrix $\mathbf{A}$, which is symmetric and positive definite, one gets:

- $\dfrac{\partial}{\partial \mathbf{y}}(\mathbf{x} - \mathbf{y})^{\mathrm{T}}\mathbf{A}(\mathbf{x} - \mathbf{y}) = -2\mathbf{A}(\mathbf{x} - \mathbf{y})$
- $\dfrac{\partial(\mathbf{x} - \mathbf{y})^{\mathrm{T}}\mathbf{A}^{-1}(\mathbf{x} - \mathbf{y})}{\partial \mathbf{A}} = -\mathbf{A}^{-1}(\mathbf{x} - \mathbf{y})(\mathbf{x} - \mathbf{y})^{\mathrm{T}}\mathbf{A}^{-1}$
- $\dfrac{\partial \ln \det(\mathbf{A})}{\partial \mathbf{A}} = \mathbf{A}^{-1}$

### *A.2.4 Special Cases of Invertible Matrices*

$$\mathbf{A}^{-1} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

$$\mathbf{A}^{-1} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & k \end{bmatrix}^{-1} = \frac{1}{\det \mathbf{A}} \begin{bmatrix} (ek - fh) & (ch - bk) & (bf - ce) \\ (fg - dk) & (ak - cg) & (cd - af) \\ (dh - eg) & (bg - ah) & (ae - bd) \end{bmatrix}$$

# Appendix B
# Useful Facts from Probability Theory and Statistics

## B.1 Commonly Used Probability Distributions

### *B.1.1 Bernoulli Distribution*

$\mathrm{B}(x|\theta) = \theta^x(1-\theta)^{1-x}, \quad \text{where } x \in \{0,1\} \text{ i } \theta \in [0,1]$
$\mathbb{E}[x] = \theta$
$\mathrm{Var}[x] = \theta(1-\theta)$

### *B.1.2 Categorical (Multinoulli) Distribution*

$\mathrm{M}(\mathbf{x}|\boldsymbol{\theta}) = \prod_{d=1}^{D} \theta_d^{x_d}, \quad \text{where } x_d \in \{0,1\} \text{ i } \theta_d \in [0,1] \text{ for all } d = 1,2,\ldots,D, \sum_{d=1}^{D} \theta_d = 1$
$\mathbb{E}[x_d] = \theta_d$
$\mathrm{Var}[x_d] = \theta_d(1-\theta_d)$

### *B.1.3 Normal Distribution*

$\mathcal{N}(x|\mu,\sigma^2) = \frac{1}{\sqrt{2\pi}\,\sigma} \exp\left\{-\frac{(x-\mu)^2}{2\sigma^2}\right\}$
$\mathbb{E}[x] = \mu$
$\mathrm{Var}[x] = \sigma^2$



J. M. Tomczak, *Deep Generative Modeling*,
https://doi.org/10.1007/978-3-031-64087-2

### B.1.4 Multivariate Normal Distribution

$$\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{D/2}} \frac{1}{|\boldsymbol{\Sigma}|^{1/2}} \exp\left\{ -\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu})\right\},$$

where $\mathbf{x}$ is $D$-dimensional vector, $\boldsymbol{\mu}$ is $D$-dimensional vector of means, and $\boldsymbol{\Sigma}$ is $D \times D$ covariance matrix
$\mathbb{E}[\mathbf{x}] = \boldsymbol{\mu}$
$\text{Cov}[\mathbf{x}] = \boldsymbol{\Sigma}$.

### B.1.5 Beta Distribution

$$\text{Beta}(x|a, b) = \frac{\Gamma(a + b)}{\Gamma(a)\Gamma(b)} x^{a-1}(1 - x)^{b-1},$$

where $x \in [0, 1]$ and $a > 0$ i $b > 0$, $\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} \mathrm{d}t$
$\mathbb{E}[x] = \frac{a}{a+b}$
$\text{Var}[x] = \frac{ab}{(a+b)^2(a+b+1)}$.

### B.1.6 Marginal Distribution

In the continuous case:

$$p(x) = \int p(x, y)\mathrm{d}y$$

and in the discrete case:

$$p(x) = \sum_y p(x, y)$$

### B.1.7 Conditional Distribution

$$p(y|x) = \frac{p(x, y)}{p(x)}$$

### B.1.8 Marginal Distribution and Conditional Distribution for Multivariate Normal Distribution

Assuming $\mathbf{x} \sim \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$, where

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}_a \\ \mathbf{x}_b \end{bmatrix}, \qquad \boldsymbol{\mu} = \begin{bmatrix} \boldsymbol{\mu}_a \\ \boldsymbol{\mu}_b \end{bmatrix}, \qquad \Sigma = \begin{bmatrix} \Sigma_a & \Sigma_c \\ \Sigma_c^T & \Sigma_b \end{bmatrix},$$

then we get the following dependencies:
$p(\mathbf{x}_a) = \mathcal{N}(\mathbf{x}_a|\boldsymbol{\mu}_a, \Sigma_a)$,
$p(\mathbf{x}_a|\mathbf{x}_b) = \mathcal{N}(\mathbf{x}_a|\hat{\boldsymbol{\mu}}_a, \hat{\Sigma}_a)$, where
$\hat{\boldsymbol{\mu}}_a = \boldsymbol{\mu}_a + \Sigma_c \Sigma_b^{-1}(\mathbf{x}_b - \boldsymbol{\mu}_b)$,
$\hat{\Sigma}_a = \Sigma_a - \Sigma_c \Sigma_b^{-1} \Sigma_c^T$.

### *B.1.9 Sum Rule*

$$p(x) = \sum_y p(x, y)$$

### *B.1.10 Product Rule*

$$\begin{aligned} p(x, y) &= p(x|y)p(y) \\ &= p(y|x)p(x) \end{aligned}$$

### *B.1.11 Bayes' Rule*

$$p(y|x) = \frac{p(x|y)p(y)}{p(x)}$$

## B.2 Statistics

### *B.2.1 Maximum Likelihood Estimator*

There are given $N$ independent examples of $\mathbf{x}$ from the identical distribution $p(\mathbf{x}|\theta)$, $\mathcal{D} = \{\mathbf{x}_1 \ldots \mathbf{x}_N\}$. The likelihood function is the following function:

$$p(\mathcal{D}|\theta) = \prod_{n=1}^{N} p(\mathbf{x}_n|\theta).$$

The logarithm of the likelihood function $p(\mathcal{D}|\theta)$ is given by the following expression:

$$\log p(\mathcal{D}|\theta) = \sum_{n=1}^{N} \log p(\mathbf{x}_n|\theta).$$

Maximum likelihood estimator of the parameters $\theta_{ML}$ minimizes the likelihood function:

$$p(\mathcal{D}|\theta_{ML}) = \max_{\theta} p(\mathcal{D}|\theta).$$

### *B.2.2 Maximum A Posteriori Estimator*

There are given $N$ independent examples of $\mathbf{x}$ from the identical distribution $p(\mathbf{x}|\theta)$, $\mathcal{D} = \{\mathbf{x}_1 \dots \mathbf{x}_N\}$. Maximum a posteriori (MAP) estimator of the parameters $\theta_{MAP}$ minimizes the a posteriori distribution:

$$p(\theta_{MAP}|\mathcal{D}) = \max_{\theta} p(\theta|\mathcal{D}).$$

### *B.2.3 Risk in Decision-Making*

Risk (expected loss) is defined as follows:

$$\mathcal{R}[\overline{y}] = \iint L(y, \overline{y}(\mathbf{x}))\, p(\mathbf{x}, y) \mathrm{d}\mathbf{x} \mathrm{d}y,$$

where $L(\cdot, \cdot)$ is the loss function.

# Index





J. M. Tomczak, *Deep Generative Modeling*,
https://doi.org/10.1007/978-3-031-64087-2