Chapter 2
Deep Learning and Optimization

Deep learning is currently one of the hottest fields not only in machine learning but in the whole of science. Since the mid-2000s, deep learning models have been revolutionizing artificial intelligence, significantly advancing state of the art across all fields of machine learning: computer vision, natural language processing, speech and sound processing, generative models, and much more. This book concentrates on synthetic data applications; we cannot hope to paint a comprehensive picture of the entire field and refer the reader to other books for a more detailed overview of deep learning [153, 289, 630, 631]. Nevertheless, in this chapter, we begin with an introduction to deep neural networks, describing the main ideas in the field. We especially concentrate on approaches to optimization in deep learning, starting from regular gradient descent and working our way towards adaptive gradient descent variations and state-of-the-art ideas.

2.1 The Deep Learning Revolution

In 2006–2007, machine learning underwent a true revolution that began a new, third "hype wave" for artificial neural networks (ANNs) in particular and artificial intelligence (AI) in general. Interestingly, one can say that artificial neural networks were directly responsible for all three AI "hype waves" in history[1]:

- in the 1950s and early 1960s, Frank Rosenblatt's *Perceptron* [735, 736], which in essence is a very simple ANN, became one of the first machine learning formalisms to be actually implemented in practice and featured in *The New York*

[1]For a very comprehensive account of the early history of ANNs and deep learning, I recommend a survey by one of the fathers of deep learning, Prof. Jürgen Schmidhuber [767]; it begins with Newton and Leibniz, whose results, as we will see, are still very relevant for ANN training today.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2021

S. I. Nikolenko, *Synthetic Data for Deep Learning*, Springer Optimization and Its Applications 174, https://doi.org/10.1007/978-3-030-75178-4_2

Times, which led to the first big surge in AI research; note that the first mathematical formalizations of neural networks appeared in the 1940s [589], well before "artificial intelligence" became a meaningful field of computer science with the foundational works of Alan Turing [878] and the Dartmouth workshop [587, 611] (see Section 2.3);

- although gradient descent is a very old idea, known and used at least since the early XIX century, only by the 1980s, it became evident that backpropagation, i.e., computing the gradient with respect to trainable weights in the network via its computational graph, can be used to apply gradient descent to deep neural networks with virtually arbitrary acyclic architectures; this idea became common in the research community in the early 1980s [924], and the famous *Nature* paper by Rumelhart et al. [742] marked the beginning of the second coming of neural networks into the popular psyche and business applications.

Both of these hype waves proved to be premature, and neither in the 1960s nor in the 1990s could neural networks live up to the hopes of researchers and investors. Interestingly, by now we understand that this deficiency was as much technological as it was mathematical: neural architectures from the late 1980s or early 1990s could perform very well if they had access to modern computational resources and, even more importantly, modern datasets. But at the moment, the big promises were definitely unfounded; let me tell just one story about it.

One of the main reasons the first hype wave came to a halt was the failure of a large project in no less than machine translation! It was the height of the Cold War, and the US government decided it would be a good idea to develop an automatic translation machine from Russian to English, at least for formal documents. They were excited by the Georgetown–IBM experiment, an early demonstration of a very limited machine translation system in 1954 [381]. The demonstration was a resounding success, and researchers of the day were sure that large-scale machine translation was just around the corner.

Naturally, this vision did not come to reality, and twelve years later, in 1966, the ALPAC (Automatic Language Processing Advisory Committee) published a famous report that had to admit that machine translation was out of reach at the moment and stressed that a lot more research in computational linguistics was needed [620]. This led to a general disillusionment with AI on the side of the funding bodies in the U.S., and when grants stop coming in, researchers usually have to switch to other topics, so the first "AI winter" followed. This is a great illustration of just how optimistic early AI was: naturally, researchers did not expect perfection and would be satisfied with the state of, say, modern Google Translate, but by now we realize how long and hard a road it has been to what Google Translate can do today.

However, in the mid-2000s, deep neural networks started working in earnest. The original approaches to training deep neural networks that proved to work around that time were based on unsupervised pretraining [226]: Prof. Hinton's group achieved the first big successes in deep learning with deep belief networks (DBN), a method where layers of deep architectures are pretrained with the restricted Boltzmann machines, and gradient descent comes only at the very end [344, 347], while in Prof. Ben-

gio's group, similar results on unsupervised pretraining were achieved by stacking autoencoders pretrained on the results of each other [62, 895]. Later, results on activation functions such as ReLU [281], new regularization methods for deep neural networks [387, 816], and better initialization of neural network weights [280] made unsupervised pretraining unnecessary for problems where large labeled datasets are available. These results have changed ANNs from "the second best way" into the method of choice revolutionizing one field of machine learning after another.

The first practical field where deep learning significantly improved state of the art in real-world applications was speech recognition, where breakthrough results were obtained by DBNs used to extract features from raw sound [346]. It was followed closely by computer vision, which we discuss in detail in Chapter 3, and later natural language processing (NLP). In NLP, the key contribution proved to be *word embeddings*, low-dimensional vectors that capture some of the semantic and syntactic properties of words and at the same time make the input dimensions for deep neural networks much more manageable [79, 600, 666]. These word embeddings were processed initially mostly by recurrent networks, but over recent years, the field has been overtaken by architectures based on self-attention: Transformers, BERT-based models, and the GPT family [94, 179, 192, 697, 698, 891]. We will touch upon natural language processing in Section 8.4, although synthetic data is not as relevant for NLP as it is for computer vision.

We have discussed in Section 1.1 that the data problem may become a limiting factor for further progress in some fields of deep learning, and definitely has already become such a factor for some fields of application. However, at present, deep neural networks define state of the art in most fields of machine learning, and progress does not appear to stop. In this chapter, we will discuss some of the basics of deep learning, and the next chapter will put a special emphasis on convolutional neural networks (Section 3.1 and further) because they are the main tools of deep learning in computer vision, and synthetic data is especially important in that field. But let me begin with a more general introduction, explaining how machine learning problems lead to optimization problems, how neural networks represent machine learning models, and how these optimization problems can be solved.

There is one important disclaimer before we proceed. I am writing this in 2020, and deep learning is constantly evolving. While the basic stuff such as the Bayes rule and neural networks as computational graphs will always be with us, it is very hard to say if the current state of the art in almost anything related to machine learning will remain the state of the art for long. Case in point: in the first draft of this book, I wrote that activation functions for individual units are more or less done. ReLU and its leaky variations work well in most problems, you can also try *Swish* found by automated search (pretty exhaustive, actually), and that's it, the future most probably shouldn't surprise us here. After all, these are just unary functions, and the *Swish* paper explicitly says that simpler activation functions outperform more complicated ones [702]. But in September 2020... well, let's not spoil it, see the end of Section 2.3.

That is why throughout this chapter and the next one, I am trying to mostly concentrate on the ideas and motivations behind neural architectures. I am definitely *not* trying to recommend any given architecture because most probably, when you

are reading this, the recommendations have already changed. When I say "current state of the art", it's just that the snapshot of ideas that I have attempted to make as up to date as I could, and some of which may have become obsolete by the time you are reading this. The time for comprehensive surveys of deep learning has not yet come. So I am glad that this book is about synthetic data, and all I need from these two chapters is a brief introduction.

2.2 A (Very) Brief Introduction to Machine Learning

Before proceeding to neural networks, let me briefly put them into a more general context of machine learning problems. I usually begin my courses in machine learning by telling students that machine learning is a field of applied probability theory. Indeed, most of machine learning problems can be mathematically formulated as an application of the *Bayes rule*:

$$p\left(\theta \mid D\right) = \frac{p(\theta)p\left(D \mid \theta\right)}{p(D)},$$

where D denotes the data and θ denotes the model parameters. The distributions in the Bayes rule have the following meaning and intuition behind them in machine learning:

- $p\left(D \mid \theta\right)$ is the *likelihood*, i.e., the model itself; the likelihood captures our assumptions about how data is generated in a probability distribution;
- $p(\theta)$ is the *prior* probability, i.e., the distribution of our beliefs about the model parameters *a priori*, before we get any data;
- $p\left(\theta \mid D\right)$ is the *posterior* probability, i.e., the distribution of our beliefs about the model parameters *a posteriori*, after we take available data into account;
- $p(D) = \int p\left(D \mid \theta\right) p(\theta) \mathrm{d}\theta$ is the *evidence* or *marginal probability* of the data averaged over all possible values of θ according to the likelihood.

This simple formula gives rise to the mathematical formulations of most machine learning problems. The first problem, common in classical statistics as well, is to find the *maximum likelihood hypothesis*

$$\theta_{ML} = \arg\max_{\theta} p\left(D \mid \theta\right).$$

The second problem is to multiply the likelihood by the prior, getting the posterior

$$p\left(\theta \mid D\right) \propto p\left(D \mid \theta\right) p(\theta),$$

and then find the *maximum a posteriori* hypothesis:

$$\theta_{MAP} = \arg\max_{\theta} p(\theta \mid D) = \arg\max_{\theta} p\left(D \mid \theta\right) p(\theta).$$

These two problems usually have similar structure when considered as optimization problems (we will see that shortly), and most practical machine learning is being done by maximizing either the likelihood or the posterior.

The final and usually the hardest problem is to find the *predictive distribution* for the next data point:

$$p(x \mid D) = \int p(x, \theta \mid D)\, \mathrm{d}\theta = \int p(x \mid \theta)\, p(\theta \mid D)\, \mathrm{d}\theta.$$

For at least moderately complex model likelihoods, this usually leads to intractable integrals and the need to develop approximate methods. Sometimes, it is this third problem which is called *Bayesian inference*, although the term is applicable as soon as a prior appears.

This mathematical essence can be applied to a wide variety of problems of different nature. With respect to their setting, machine learning problems are usually roughly classified into (Figure 2.1 provides an illustration):

- *supervised learning* problems, where data is given in the form of pairs $D = \{(\mathbf{x}_n, \mathbf{y}_n)\}_{n=1}^N$, with $\mathbf{x}_n$ being the nth data point (input of the model) and $\mathbf{y}_n$ being the target variable:
  - in *classification* problems, the target variable $\mathbf{y}$ is categorical, discrete, that is, we need to place $\mathbf{x}$ into one of a discrete set of classes;
  - in *regression* problems, the target variable $\mathbf{y}$ is continuous, that is, we need to predict values of $\mathbf{y}$ given values of $\mathbf{x}$ with as low error as possible;
- *unsupervised learning* problems that are all about learning a distribution of data points; in particular,

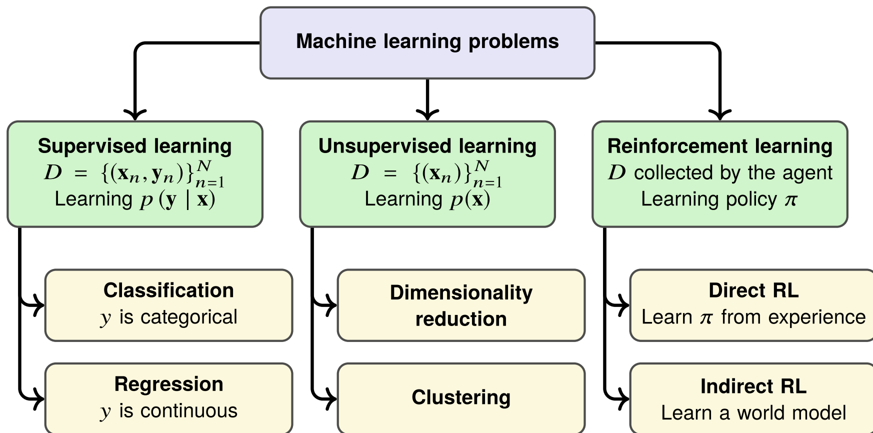

Fig. 2.1 A general taxonomy of machine learning problems.

  - *dimensionality reduction* techniques aim to learn a low-dimensional representation that still captures important information about a high-dimensional dataset;
  - *clustering* does basically the same but reduces not to a continuous space but to a discrete set of clusters, assigning each $\mathbf{x}$ from the input dataset with a cluster label; there is a parallel here with the classification/regression distinction in supervised learning;

- *reinforcement learning* problems where the data usually does not exist before learning begins, and an agent is supposed to collect its own dataset by interacting with the environment;
  - agents in *direct reinforcement learning* learn their behaviour policy π directly, either by learning a value function for various states and actions or by parameterizing π and learning it directly via policy gradient;
  - agents in *indirect reinforcement learning* use their experience to build a model of the environment, thus allowing for planning.

There are, of course, intermediate cases and fusions of these problems, the most important being probably *semi-supervised learning*, where a (usually small) part of the dataset is labeled and the other (usually far larger) part is not.

In this book, we will mostly consider supervised learning problems. For example, in computer vision, an image classification problem might be formalized with $\mathbf{x}_n$ being the image (a three-dimensional array of pixels, where the third dimension is the color) and $\mathbf{y}_n$ being a one-hot representation of target classes, i.e., $\mathbf{y}_n = (0 \ldots 0\ 1\ 0 \ldots 0)$, where 1 marks the position of the correct answer.

For a simple but already nontrivial example, consider the Bernoulli trials, the distribution of tossing a (not necessarily fair) coin. There is only one parameter here, let's say θ is the probability of the coin landing heads up. The data D is in this case a sequence of outcomes, heads or tails, and if D contains n heads and m tails, the likelihood is

$$p(D \mid \theta) = \theta^n (1 - \theta)^m .$$

The maximum likelihood hypothesis is, obviously,

$$\theta_{ML} = \arg\max_{\theta} \theta^n (1 - \theta)^m = \frac{n}{n + m},$$

but in real life, this single number is clearly insufficient. If you take a random coin from your purse, toss it once, and observe heads, your dataset will have $n = 1$ and $m = 0$, and the maximum likelihood hypothesis will be $\theta_{ML} = 1$, but you will hardly expect that this coin will now *always* land heads up. The problem is that you already have a prior distribution for the coin, and while the maximum likelihood hypothesis is perfectly fine in the limit, as the number of experiments approaches infinity, for smaller samples, the prior will definitely play an important role.

Suppose that the prior is uniform, $p(\theta) = 1$ for $\theta \in [0, 1]$ and 0 otherwise. Note that this is not quite what you think about a coin taken from your purse, you would

rather expect a bell-shaped distribution centered at $\frac{1}{2}$. This prior is more suitable for a new phenomenon about which nothing is known *a priori* except that it has two outcomes. But even for that prior, the conclusion will change. First, the posterior distribution is now

$$p(\theta \mid D) = \frac{p(\theta)p(D \mid \theta)}{p(D)} = \begin{cases} \frac{1}{p(D)}\theta^n (1-\theta)^m, & \text{for } \theta \in [0,1], \\ 0 & \text{otherwise,} \end{cases}$$

where the normalizing constant can be computed as

$$p(D) = \int p(\theta)p(D \mid \theta)\mathrm{d}\theta = \int_0^1 \theta^n (1-\theta)^m \mathrm{d}\theta =$$
$$= \frac{\Gamma(n+1)\Gamma(m+1)}{\Gamma(n+m+2)} = \frac{n!m!}{(n+m+1)!}.$$

Since the prior is uniform, the posterior distribution is still maximized at the exact same point:

$$\theta_{MAP} = \theta_{ML} = \frac{n}{n+m}.$$

This situation is illustrated in Figure 2.2a that shows the prior distribution, likelihood, and posterior distribution for the parameter θ with uniform prior and the dataset consisting of two heads. The posterior distribution, of course, has the same maximum as the likelihood, at $\theta_{MAP} = \theta_{ML} = 1$.

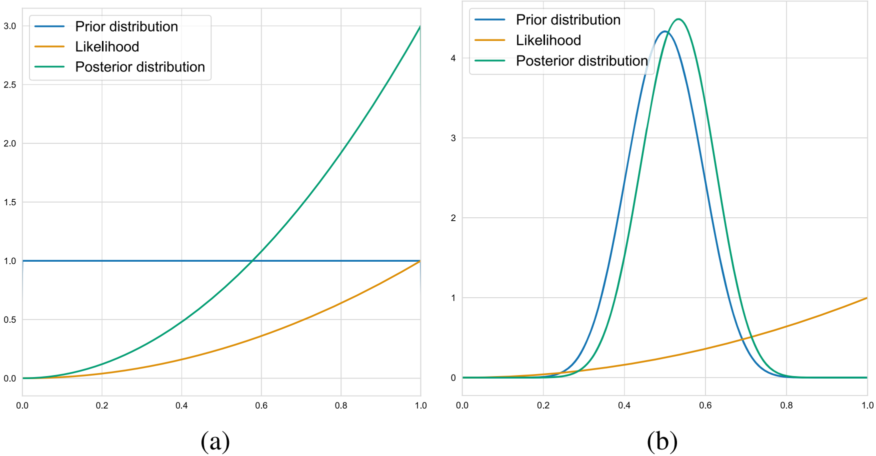

Fig. 2.2 Distributions related to the Bernoulli trials: (a) uniform prior, two heads in the dataset; (b) prior Beta(15, 15), two heads in the dataset.

But the predictive distribution will tell a different story because the posterior is maximized at its right border, and the predictions should integrate over the entire posterior. Let us find the probability of this coin landing heads on the next toss:

$$p(\text{heads}|D) = \int_0^1 p(\text{heads}|\theta)p(\theta|D)\mathrm{d}\theta = \int_0^1 \frac{\theta^{n+1}(1-\theta)^m}{p(D)}d\theta =$$
$$= \frac{(n+1)!m!}{(n+m+2)!} \cdot \frac{(n+m+1)!}{n!m!} = \frac{n+1}{n+m+2}.$$

In this formula, we have derived what is known as *Laplace's rule of succession*, which shows how to apply Bayesian smoothing to the Bernoulli trials.

Note that in reality, if you take a random coin out of your pocket, the uniform prior would be a poor model for your beliefs about this coin. It would probably be more like the prior shown in Fig. 2.2b, where we show the exact same dataset processed with a non-uniform, informative prior $p(\theta) = \text{Beta}(\theta; 15, 15)$. The beta distribution

$$\text{Beta}(\theta; \alpha, \beta) \propto \theta^{\alpha-1} (1-\theta)^{\beta-1}$$

is the *conjugate prior* distribution for the Bernoulli trials, which means that after multiplying a beta distribution by the likelihood of the Bernoulli trials, we again get a beta distribution in the posterior:

$$\text{Beta}(\theta; \alpha, \beta) \times \theta^n (1-\theta)^m \propto \text{Beta}(\theta; \alpha+n, \beta+m).$$

For instance, in Fig. 2.2b, the prior is $\text{Beta}(\theta; 15, 15)$, and the posterior, after multiplying by θ^2 and renormalizing, becomes $\text{Beta}(\theta; 17, 15)$.

In machine learning, one assumption that is virtually always made is that different data points are produced independently given the model parameters, that is,

$$p(D \mid \theta) = \prod_{n=1}^{N} p(d_n \mid \theta)$$

for a dataset $D = \{d_n\}_{n=1}^N$. Therefore, it is virtually always a good idea to take logarithms before optimizing, getting the log-likelihood

$$\log p(D \mid \theta) = \sum_{n=1}^{N} \log p(d_n \mid \theta)$$

and the log-posterior (note that proportionality becomes an additive constant after taking the logarithm)

$$\log p(\theta \mid D) = \log p(\theta) + \sum_{n=1}^{N} \log p(d_n \mid \theta) + \text{Const},$$

which are usually the actual functions being optimized. Therefore, in complex machine learning models priors usually come in the form of *regularizers*, additive terms that impose penalties on unlikely values of θ.

For another relatively simple example, let us consider *linear regression*, a supervised learning problem of fitting a linear model to data, that is, finding a vector of weights $\mathbf{w}$ such that $y \sim \mathbf{w}^\top \mathbf{x}$ for a dataset of pairs $D = (X, \mathbf{y}) = \{(\mathbf{x}_n, y_n)\}_{n=1}^N$. The first step here is to define the likelihood, that is, represent

$$y = \mathbf{w}^\top \mathbf{x} + \epsilon$$

for some random variable (noise) ϵ and define the distribution for ϵ. The natural choice is to take the normal distribution centered at zero, $\epsilon \sim \mathcal{N}(0, \sigma^2)$, getting the likelihood as

$$p(\mathbf{y} \mid \mathbf{w}, X) = \prod_{n=1}^{N} p(y_n \mid \mathbf{w}, \mathbf{x}_n) = \prod_{n=1}^{N} \mathcal{N}(y_n \mid \mathbf{w}^\top \mathbf{x}_n, \sigma^2).$$

Taking the logarithm of this expression, we arrive at the least squares optimization problem:

$$\log p(\mathbf{y} \mid \mathbf{w}, X) = \sum_{n=1}^{N} \log \mathcal{N}(y_n \mid \mathbf{w}^\top \mathbf{x}_n, \sigma^2) =$$

$$= -\frac{N}{2} \log \left(2\pi\sigma^2\right) - \frac{1}{2\sigma^2} \sum_{n=1}^{N} \left(y_n - \mathbf{w}^\top \mathbf{x}_n\right),$$

so maximizing $\log p(\mathbf{y} \mid \mathbf{w}, X)$ is the same as minimizing $\sum_{n=1}^{N} \left(y_n - \mathbf{w}^\top \mathbf{x}_n\right)$, and the exact value of σ^2 turns out not to be needed for finding the maximum likelihood hypothesis.

Linear regression is illustrated in Figure 2.3, with the simplest one-dimensional linear regression shown in Fig. 2.3a. However, even if the data is one-dimensional, the regression does not have to be: if we suspect a more complex dependency than linear, we can express it by extracting *features* from the input before running linear regression.

In this example, the data is generated from a single period of a sinusoid function, so it stands to reason that it should be interpolated well by a cubic polynomial. Figure 2.3b shows the resulting approximation, obtained by training the model

$$y = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + \epsilon,$$

which is equivalent to $y = \mathbf{w}^\top \mathbf{x} + \epsilon$ for $\mathbf{x} = \left(1\ x\ x^2\ x^3\right)^\top$, i.e., equivalent to manually extracting polynomial features from x before feeding it to linear regression. In this way, linear regression can be used to approximate much more complex depen-

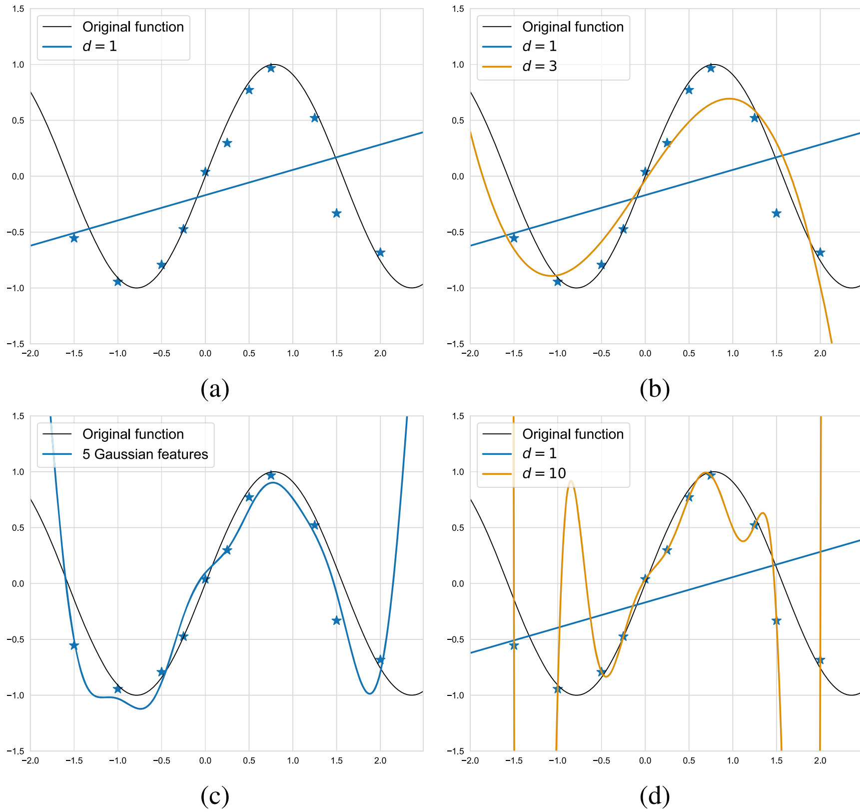

Fig. 2.3 Linear regression: (a) one-dimensional linear regression; (b) linear regression with polynomial features; (c) linear regression with Gaussian features; (d) overfitting in linear regression.

dencies. For example, Figure 2.3c shows the same dataset approximated with five Gaussian features, i.e., features of the form

$$\phi(x; \mu, s) = e^{-\frac{1}{2s}(x-\mu)^2}.$$

In fact, most neural networks that solve regression problems have a linear regression as their final layer, while neural networks for classification problems use a softmax layer, i.e., the logistic regression model. The difference and the main benefit that neural networks are providing is that the features for these simple models implemented at final layers are also learned automatically from data.

With this additional feature extraction, even linear regression can show signs of overfitting, for instance, if the features (components of the vector $\mathbf{x}$) are too heavily correlated with each other. The ultimate case of overfitting in linear regression is shown in Fig. 2.3d: if we fit a polynomial of degree $N-1$ to N points, it will

obviously be simply interpolating all of these points, getting a perfect zero error on the training set but providing quite useless predictions, as Fig. 2.3d clearly illustrates.

In this case, we might want to restrict the desirable values of $\mathbf{w}$, for instance say that the values of $\mathbf{w}$ should be "small". This statement, which I have quite intentionally made very vague, can be formalized via choosing a suitable prior distribution. For instance, we could set a normal distribution centered at zero as prior. This time, it's a multi-dimensional normal distribution, and let's say that we do not have preferences with respect to individual features so we assume the prior is round:

$$p(\mathbf{w}) = \mathcal{N}(\mathbf{w} \mid \mathbf{0}, \sigma_0^2 \mathbf{I}).$$

Then we get the following posterior distribution:

$$\log p\left(\mathbf{w} \mid \mathbf{y}, X\right) = \sum_{n=1}^{N} \log \mathcal{N}(y_n \mid \mathbf{w}^\top \mathbf{x}_n, \sigma^2) + \log \mathcal{N}(\mathbf{w} \mid \mathbf{0}, \sigma_0^2 \mathbf{I}) =$$

$$= -\frac{N}{2} \log\left(2\pi\sigma^2\right) - \frac{1}{2\sigma^2} \sum_{n=1}^{N} \left(y_n - \mathbf{w}^\top \mathbf{x}_n\right) - \frac{d}{2} \log\left(2\pi\sigma^2\right) - \frac{1}{2\sigma_0^2} \mathbf{w}^\top \mathbf{w}.$$

The maximization problem for this posterior distribution is now equivalent to minimizing

$$\sum_{n=1}^{N} \left(y_n - \mathbf{w}^\top \mathbf{x}_n\right) + \frac{\lambda}{2} \mathbf{w}^\top \mathbf{w}, \quad \text{where} \quad \lambda = \frac{\sigma^2}{\sigma_0^2}.$$

This is known as *ridge regression*. More generally speaking, regularization with a Gaussian prior centered around zero is known as L_2-regularization because as we have just seen, it amounts to adding the L_2-norm of the vector of weights to the objective function.

We will not spend much more time on Bayesian analysis in this book, but note one thing: machine learning problems are motivated by probabilistic assumptions and the Bayes rule, but from the algorithmic and practical standpoint, they are usually *optimization problems*. Finding the maximum likelihood hypothesis amounts to maximizing $p\left(D \mid \theta\right)$, and finding the maximum a posteriori hypothesis means to maximize $p(\theta \mid D)$; usually, the main computational challenge in machine learning lies either in these maximization procedures or in finding suitable methods to approximate the integral in the predictive distribution.

Therefore, once probabilistic assumptions are made and formulas such as the ones shown above are worked out, algorithmically machine learning problems usually look like an objective function depending on the data points and model parameters that need to be optimized with respect to model parameters. In simple cases, such as the Bernoulli trials or linear regression, these optimization problems can be worked out exactly. However, as soon as the models become more complicated, optimization problems become much harder and almost always nonconvex.

This means that for complex optimization problems, such as the ones represented by neural networks, virtually the only available way to solve them is to use first-order optimization methods based on gradient descent. Over the next sections, we will consider how neural networks define such optimization problems and what methods are currently available to solve them.

2.3 Introduction to Deep Learning

Before delving into state-of-the-art first-order optimization methods, let us begin with a brief introduction to neural networks in general. As a mathematical model, neural networks actually predate the advent of artificial intelligence in general: the famous paper by Warren McCulloch and Walter Pitts was written in 1943 [589], and AI as a field of science is generally assumed to be born in the works of Alan Turing, especially his 1950 essay *Computing Machinery and Intelligence* where he introduced the Turing test [877, 878]. What is even more interesting, the original work by McCulloch and Pitts already contained a very modern model of a single artificial neuron (perceptron), namely the linear threshold unit, which for inputs $\mathbf{x}$, weights $\mathbf{w}$, and threshold a outputs

$$y = \begin{cases} 1, & \text{if } \mathbf{w}^\top\mathbf{x} \geq a, \\ 0, & \text{if } \mathbf{w}^\top\mathbf{x} < a. \end{cases}$$

This is exactly how units in today's neural networks are structured, a linear combination of inputs followed by a nonlinearity:

$$y = h(\mathbf{w}^\top\mathbf{x}).$$

The *activation function* h is usually different today, and we will survey modern activation functions in a page or two.

The linear threshold unit was one of the first machine learning models actually implemented in software (more like hardware in those times): in 1958, the *Perceptron* device developed by Frank Roseblatt [735] was able to learn the weights from a dataset of examples and actually could receive a 20×20 image as input. The Perceptron was also an important factor in the first hype wave of artificial intelligence. For instance, a *New York Times* article (hardly an unreliable tabloid) devoted to the machine said the following: "The embryo of an electronic computer... learned to differentiate between right and left after fifty attempts in the Navy's demonstration... The service said that it would use this principle to build the first of its Perceptron thinking machines that will be able to read and write. It is expected to be finished in about a year at a cost of $100,000" [858]. Naturally, nothing like that happened, but artificial neural networks were born.

The main underlying idea of the deep neural network is *connectionism*, an approach in cognitive science and neurobiology that posits the emergence of complex behaviour and intelligence in very large networks of simple computational elements [51, 52]. As a movement in both philosophy and computer science, connectionism rose to prominence in the 1980s, together with the second AI "hype wave" caused by deep neural networks. Today, deep learning provides plenty of evidence that complex networks of simple units can perform well in the most complex tasks of artificial intelligence, even if we still do not understand the human brain fully and perhaps strong human-level AI cannot be achieved by simple stacking of layers (to be honest, we don't really know).

An *artificial neural network* is defined by its computational graph. The computational graph is a directed acyclic graph $G = (V, E)$ whose nodes correspond to elementary functions and edges incoming into vertices correspond to their arguments. The source vertices (vertices of indegree zero) represent input variables, and all other vertices represent functions of these variables obtained as compositions of the functions shown in the nodes (for brevity and clarity, I will not give the obvious formal recursive definitions). In the case of neural networks for machine learning, a computational graph usually contains a single sink vertex (vertex of outdegree zero) and is said to compute the function that corresponds to this sink vertex.

Figure 2.4 shows a sample computational graph composed of simple arithmetic functions. The graph shows variables and elementary functions inside the corresponding nodes and shows the results of a node as functions of input variables along its outgoing edge; the variables are artificially divided into "inputs" x and "weights"

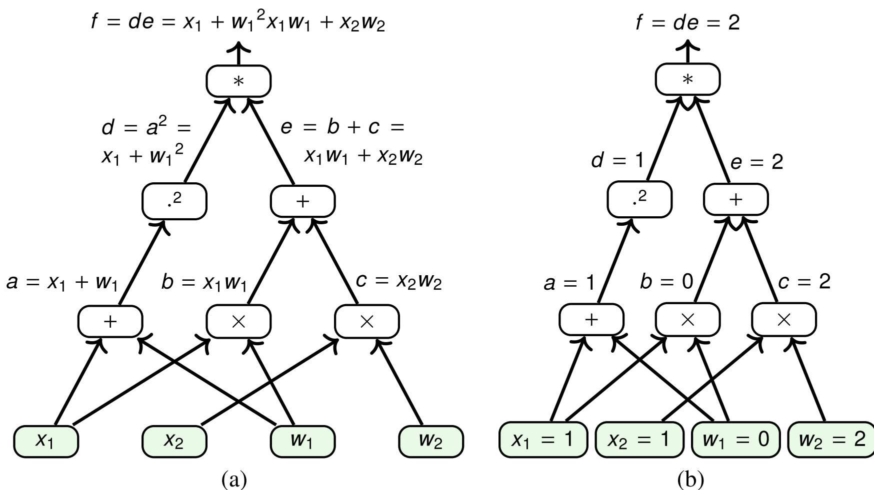

Fig. 2.4 A sample computational graph: (a) function definitions; (b) sample computation for $x_1 = x_2 = 1, w_1 = 0, w_2 = 2$.

w for illustrative purposes. In this example, the top vertex of the graph computes the function

$$f = (x_1 + w_1)^2(x_1w_1 + x_2w_2).$$

The main idea of using computational graphs is to be able to solve optimization problems with the functions computed by these graphs as objectives. To apply a first-order optimization method such as gradient descent to a function $f(\mathbf{w})$ with respect to its inputs $\mathbf{w}$, we need to be able to do two things:

(1) compute the function $f(\mathbf{w})$ at every point $\mathbf{w}$;
(2) take the gradient $\nabla_{\mathbf{w}} f$ of the objective function with respect to the optimization variables.

The computational graph provides an obvious algorithm for the first task: if we know how to compute each elementary function, we simply traverse the graph from sources (variables) to the sink, computing intermediate results and finally getting the value of f. For example, let us set $x_1 = x_2 = 1$, $w_1 = 0$, $w_2 = 2$; traversing the graph in Fig. 2.4 yields the values shown in Fig. 2.4b:

$$\begin{array}{lll} a = x_1 + w_1 = 1, & b = x_1w_1 = 0, & c = x_2w_2 = 2, \\ d = a^2 = 1, & e = b + c = 2, & f = de = 2. \end{array}$$

As for the second task, there are two possible ways to take the gradients given a computational graph. Suppose that in Fig. 2.4, we want to compute $\nabla_{\mathbf{w}} f$ for $x_1 = x_2 = 1$, $w_1 = 0$, $w_2 = 2$. The first approach, *forward propagation*, is to compute the partial derivatives along with the function values. In this way, we can compute the partial derivatives of each node in the graph with respect to the same variable; Fig. 2.5 shows the results for derivatives with respect to w_1:

$$\begin{array}{lll} \frac{\partial a}{\partial w_1} = \frac{\partial w_1}{\partial w_1} = 1, & \frac{\partial b}{\partial w_1} = x_1\frac{\partial w_1}{\partial w_1} = 0, & \frac{\partial c}{\partial w_1} = 0, \\ \frac{\partial d}{\partial w_1} = 2a\frac{\partial a}{\partial w_1} = 2, & \frac{\partial e}{\partial w_1} = \frac{\partial b}{\partial w_1} + \frac{\partial c}{\partial w_1} = 1, & \frac{\partial f}{\partial w_1} = d\frac{\partial e}{\partial w_1} + e\frac{\partial d}{\partial w_1} = 1 + 4 = 5. \end{array}$$

This approach, however, does not scale; it only yields the derivative $\frac{\partial f}{\partial w_1}$, and in order to compute $\frac{\partial f}{\partial w_2}$, we would have to go through the whole graph again! Since in deep learning, the problem is usually to compute the gradient $\nabla_{\mathbf{w}} f$ with respect to a vector of weights $\mathbf{w}$ that could have thousands or even millions of components, either running the algorithm $|\mathbf{w}|$ times or spending the memory equal to $|\mathbf{w}|$ on every computational node is entirely impractical.

That is why in deep learning, the main tool for taking the gradients is the reverse procedure, *backpropagation*. The main advantage is that this time we obtain both derivatives, $\frac{\partial f}{\partial w_1}$ and $\frac{\partial f}{\partial w_2}$, after only a single backwards pass through the graph. Again, the main tool in this computation is simply the chain rule. Given a graph node $v = h(x_1, \ldots, x_k)$ that has children $g_1, \ldots, g_l$ in the computational graph, the backpropagation algorithm computes

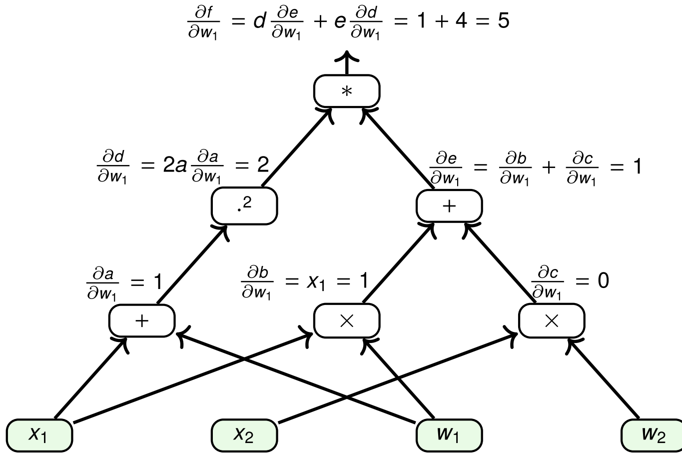

Fig. 2.5 Gradient computation on the graph from Fig. 2.4 for $x_1 = x_2 = 1$, $w_1 = 0$, $w_2 = 2$: forward propagation.

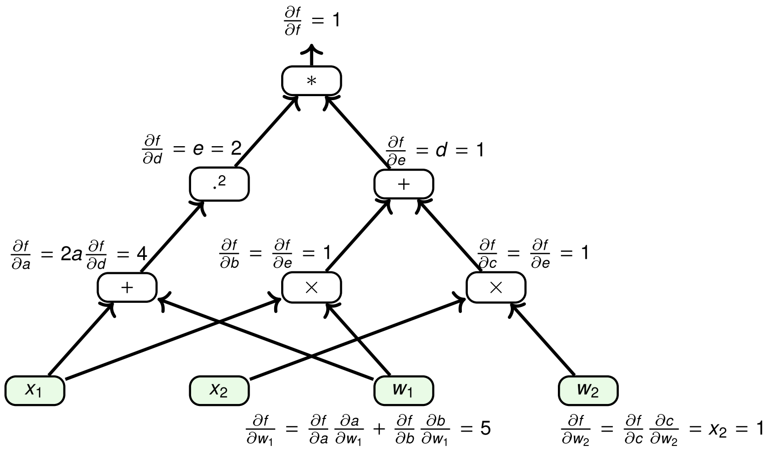

Fig. 2.6 Gradient computation on the graph from Fig. 2.4 for $x_1 = x_2 = 1$, $w_1 = 0$, $w_2 = 2$: backpropagation.

$$\frac{\partial f}{\partial v} = \frac{\partial f}{\partial g_1}\frac{\partial g_1}{\partial v} + \ldots + \frac{\partial f}{\partial g_l}\frac{\partial g_l}{\partial v},$$

where the values $\frac{\partial f}{\partial g_j}\frac{\partial g_j}{\partial v}$ have been obtained in the previous steps of the algorithm and received by the node v from its children, and sends to each of the parents x_i of the node v the value $\frac{\partial f}{\partial v}\frac{\partial v}{\partial x_i}$. The base of the induction here is the sink node, $\frac{\partial f}{\partial f} = 1$, rather than source nodes as before. In the example shown in Figure 2.4, we get the derivatives shown in Figure 2.6:

$$\frac{\partial f}{\partial f} = 1, \qquad \frac{\partial f}{\partial d} = e = 2, \qquad \frac{\partial f}{\partial e} = d = 1,$$
$$\frac{\partial f}{\partial a} = 2a\frac{\partial f}{\partial d} = 4, \qquad \frac{\partial f}{\partial b} = \frac{\partial f}{\partial e} = 1, \qquad \frac{\partial f}{\partial c} = \frac{\partial f}{\partial e} = 1,$$
$$\frac{\partial f}{\partial w_1} = \frac{\partial f}{\partial a}\frac{\partial a}{\partial w_1} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial w_1} = 5, \qquad \frac{\partial f}{\partial w_2} = \frac{\partial f}{\partial c}\frac{\partial c}{\partial w_2} = x_2 = 1.$$

A real-life neural network is almost always organized into *layers*. This means that the computational graph of a neural network has subsets of nodes that are incomparable in topological order and hence can be computed in parallel. These nodes usually also have the same inputs and activation functions (or at least the same structure of inputs, like convolutional neural networks that we will consider in Section 3.1), which means that operations on entire layers can be represented as matrix multiplications and componentwise applications of the same functions to vectors.

This structure enables the use of graphics processing units (GPUs) that are specifically designed as highly parallel architectures to handle matrix operations and componentwise operations on vectors, giving speedups of up to 10-50x for training compared to CPU-based implementations. The idea of using GPUs for training neural networks dates back at least to 2004 [639], and convolutional networks were put on GPUs already in 2006 [127]. This idea was quickly accepted across the board and became a major factor in the deep learning revolution: for many applications, this 10-50x speedup was exactly what was needed to bring neural networks into the realm of realistic solutions.

Therefore, one of the first and most natural neural network architectures is the *fully connected* (or *densely connected*) network: a sequence of layers such that a neuron at layer l receives as input activations from *all* neurons at layer $l - 1$ and sends its output to *all* neurons at layer $l + 1$, i.e., each two neighboring layers form a complete bipartite graph of connections.

Fully connected networks are still relevant in some applications, and many architectures include fully connected layers. However, they are usually ill-suited for unstructured data such as images or sound because they scale badly with the number of inputs, leading to a huge number of weights that will almost inevitably overfit. For instance, the first layer of a fully connected network that has 200 neurons and receives a 1024×1024 image as input will have more than 200 million weights! No amount of L_2 regularization is going to fix that, and we will see how to avoid such overkill in Section 3.1. On the other hand, once a network of a different structure has already extracted a few hundred or a couple thousand features from this image, it does make sense to have a fully connected layer or two at the end to allow the features to interact with each other freely, so dense connections definitely still have a place in modern architectures.

Figure 2.7 presents a specific example of a three-layered network together with the backpropagation algorithm worked out for this specific case. On the left, it shows the architecture: input $\mathbf{x}$ goes through two hidden layers with weight matrices $W^{(1)}$ and $W^{(2)}$ (let us skip the bias vectors in this example in order not to clutter notation even further). Each layer also has a nonlinear activation function h, so its outputs are $\mathbf{z}^{(1)} = h\left(W^{(1)}\mathbf{x}\right)$ and $\mathbf{z}^{(2)} = h\left(W^{(2)}\mathbf{z}^{(1)}\right)$. After that, the network has a scalar output $y = h\left({\mathbf{w}^{(3)}}^\top \mathbf{z}^{(2)}\right)$, again computed with activation function h and weight vector

$\mathbf{w}^{(3)}$ from $\mathbf{z}^{(2)}$, and then the objective function f is a function of the scalar y. The formulas in the middle column show the forward propagation part, i.e., computation along the graph, and formulas on the right show the backpropagation algorithm that begins with $\frac{\partial f}{\partial y}$ and progresses in the opposite direction. Dashed lines on the figure divide the architecture into computational layers, and the computations are grouped inside the dashed lines.

For example, on the second hidden layer, we have the weight matrix $W^{(2)}$, input $\mathbf{z}^{(1)}$ from the layer below, output $\mathbf{z}^{(2)} = h\left(W^{(2)}\mathbf{z}^{(2)}\right)$, and during backpropagation, we also have the gradient $\nabla_{\mathbf{z}^{(2)}} f$ coming from the layer above as part of the induction hypothesis. In backpropagation, this dense layer needs to do two things:

- compute the gradient with respect to its own matrix of weights $W^{(2)}$ so that they can be updated; this is achieved as

$$\nabla_{W^{(2)}} f = h'\left(W^{(2)}\mathbf{z}^{(2)}\right)\left(\nabla_{\mathbf{z}^{(2)}} f\right) {\mathbf{z}^{(1)}}^{\top};$$

  note how the dimensions match: $W^{(2)}$ is a 3×4 matrix in this example, $\nabla_{\mathbf{z}^{(2)}} f$ is a 3×1 vector, and $\mathbf{z}^{(1)}$ is a 4×1 vector;
- compute the gradient with respect to its input $\mathbf{z}^{(1)}$ and send it down to the layer below:

$$\nabla_{\mathbf{z}^{(1)}} f = h'\left(W^{(2)}\mathbf{z}^{(2)}\right) {W^{(2)}}^{\top} \nabla_{\mathbf{z}^{(2)}} f.$$

Figure 2.7 shows the rest of the computations, introducing intermediate vectors $\mathbf{o}$ for brevity. As we can see, this algorithm is entirely expressed in the form of matrix operations and componentwise applications of functions, and thus it lends itself easily to GPU-based parallelization.

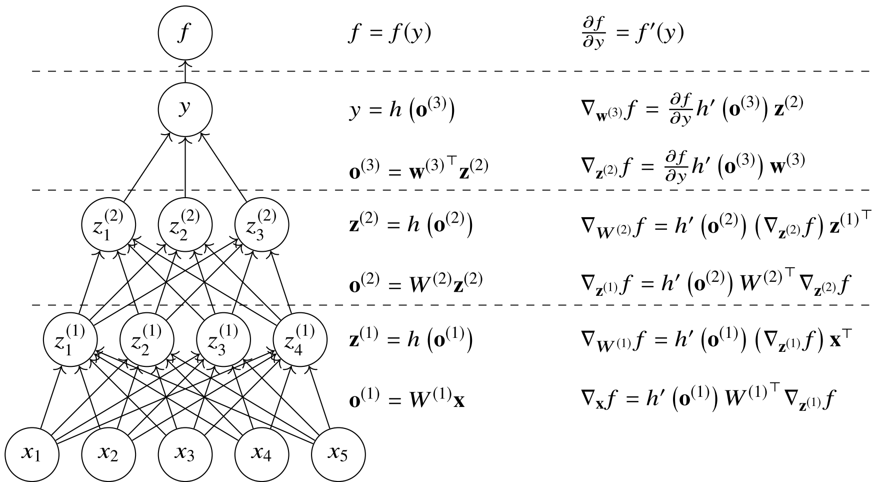

Fig. 2.7 A three-layered fully connected architecture with computations for backpropagation.

The only thing left for this section is to talk a bit more about activation functions. The original threshold activation, suggested by McCulloch and Pitts and implemented by Rosenblatt, is almost never used now: if nothing else, thresholds are hard to optimize by gradient descent because the derivative of a threshold function is everywhere zero or nonexistent. Throughout neural network history, the most popular activation functions had been sigmoids, usually either the logistic sigmoid

$$\sigma(a) = \frac{1}{1 + e^{-a}}$$

or the hyperbolic tangent

$$\tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}.$$

Several classical activation functions, from threshold to ReLU, are shown in Fig. 2.8.

Research of the last decade, however, shows that one can get a much better family of activation functions (for internal layers of deep networks—you still can't get around a softmax at the end of a classification problem, of course) if one does not restrict it by a horizontal asymptote at least on one side. The most popular activation function in modern artificial networks is the *rectified linear unit* (ReLU)

$$\text{ReLU}(x) = \begin{cases} 0, & \text{if } x < 0, \\ x, & \text{if } x \geq 0 \end{cases}$$

and its variations that do not have a hard zero for negative inputs but rather a slower growing function, for example, the *leaky ReLU* [570]

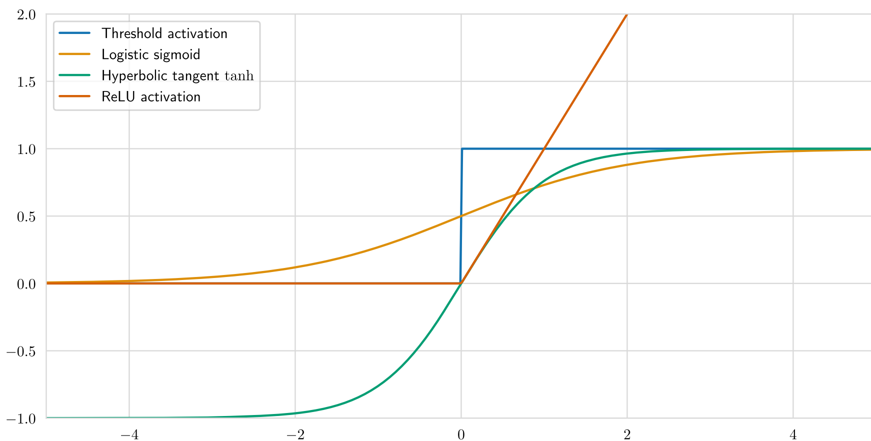

Fig. 2.8 A comparison of activation functions: classical activation functions.

$$\text{LReLU}(x) = \begin{cases} ax, & \text{if } x < 0, \\ x, & \text{if } x > 0 \end{cases}$$

or the *exponential linear unit* [161]

$$\text{ELU}(x) = \begin{cases} \alpha \left(e^x - 1\right), & x < 0, \\ x, & x \geq 0. \end{cases}$$

There is also a smooth variant of ReLU, known as the *softplus* function or [282]:

$$\text{softplus}(x) = \ln\left(1 + e^x\right).$$

You can see a comparison of ReLU variations in Figure 2.9. In any case, all activation functions used in modern neural networks must be differentiable so that gradient descent can happen; it's okay to have kinks on a subset of measure zero, like ReLU does, since one can always just set the derivative to zero at that point.

It is always tempting to try and get a better result just by switching the activation function, but most often it fails: it is usually the last optimization I would advise to actively try. However, if you try many different activation functions systematically and with a wide range of models, it might be possible to improve upon standard approaches.

In 2017, *Google Brain* researchers Ramachandran et al. [702] did exactly this: they constructed a search space of possible activation functions (basically a recursive computational graph with a list of possible unary and binary functions to insert there), used the ideas of *neural architecture search* [1031] to formulate the search for activation functions as a reinforcement learning problem, and made good use of *Google*'s huge computational power to search this space as exhaustively as they

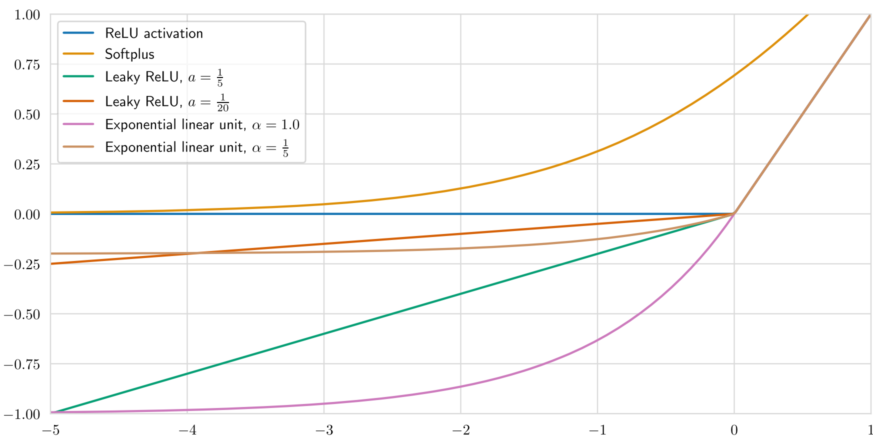

Fig. 2.9 A comparison of activation functions: ReLU variations.

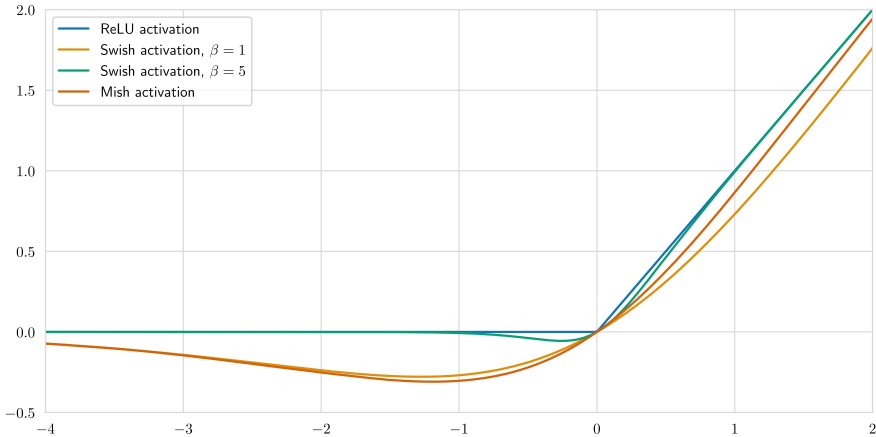

Fig. 2.10 A comparison of activation functions: *Swish* and *Mish*.

could. The results were interesting: in particular, Ramachandran et al. found that complicated activation functions consistently underperform simpler ones. As the most promising resulting function, they highlighted the so-called *Swish* activation:

$$\mathrm{Swish}(x) = x\sigma(\beta x) = \frac{x}{1 + e^{-\beta x}}.$$

Depending on the parameter β, *Swish* scales the range from perfectly linear (when $\beta = 0$) to ReLU (when $\beta \to \infty$). Figure 2.10 shows *Swish* activation and other variations; the most interesting feature of *Swish* is probably the fact that it is not monotone and has a minimum in the negative part of the spectrum.

In 2019, Misra [604] suggested the *Mish* activation, a variation of *Swish*:

$$\mathrm{Mish}(x) = x \tanh(\mathrm{softplus}(x)) = x \tanh\left(\ln\left(1 + e^x\right)\right).$$

Both *Swish* and *Mish* activations have been tested in many applications, including convolutional architectures for computer vision, and they now define state of the art, although good old ReLUs are far from completely replaced.

The text above was written in the summer of 2020. But, of course, this was not the end of the story. In September 2020, Ma et al. [567] presented a new look on *Swish* and ReLUs. They generalized them both by using a smooth approximation of the maximum function:

$$S_\beta(x_1, \ldots, x_n) = \frac{\sum_{i=1}^n x_i e^{\beta x_i}}{\sum_{i=1}^n e^{\beta x_i}}.$$

Let us substitute two functions in place of the arguments: for the hard maximum $\max(g(x), h(x))$, we get its smooth counterpart

$$S_\beta\left(g(x), h(x)\right) = g(x)\frac{e^{\beta g(x)}}{e^{\beta g(x)} + e^{\beta h(x)}} + h(x)\frac{e^{\beta h(x)}}{e^{\beta g(x)} + e^{\beta h(x)}} =$$
$$= g(x)\sigma\left(\beta\left(g(x) - h(x)\right)\right) + h(x)\sigma\left(\beta\left(h(x) - g(x)\right)\right) =$$
$$= \left(g(x) - h(x)\right)\sigma\left(\beta\left(g(x) - h(x)\right)\right) + h(x).$$

Ma et al. call this the *ActivateOrNot* (ACON) activation function. They note that

- for $g(x) = x$ and $h(x) = 0$, the hard maximum is $\mathrm{ReLU}(x) = \max(x, 0)$, and the smooth counterpart is

$$f_{ACON-A}\left(x, 0\right) = S_\beta(x, 0) = x\sigma\left(\beta x\right),$$

  that is, precisely the *Swish* activation function;
- for $g(x) = x$ and $h(x) = ax$ with some $a < 1$, the hard maximum is $\mathrm{LReLU}(x) = \max(x, px)$, and the smooth counterpart is

$$f_{ACON-B} = S_\beta(x, ax) = (1 - a)\, x\sigma\left(\beta(1 - a)x\right) + ax;$$

- both of these functions can be straightforwardly generalized to

$$f_{ACON-C} = S_\beta(a_1 x, a_2 x) = (a_1 - a_2)\, x\sigma\left(\beta(a_1 - a_2)x\right) + a_2 x;$$

  in this case, a_1 and a_2 can become learnable parameters, and their intuitive meaning is that they serve as the limits of ACON-C's derivative:

$$\lim_{x\to\infty}\frac{\mathrm{d}f_{ACON-C}}{\mathrm{d}x} = a_1, \qquad \lim_{x\to-\infty}\frac{\mathrm{d}f_{ACON-C}}{\mathrm{d}x} = a_2.$$

Figure 2.11 shows the ACON-C function for different values of a_1, a_2, and β, starting from exactly the *Swish* function and showing the possible variety.

All this has only just happened, and so far it is hard to say whether this idea is going to catch on across many neural architectures or die down quietly. But this is a great example of how hard it is to write about deep learning; we will see such examples in later sections as well.

Let us summarize. We have seen how neural networks are structured as computational graphs composed of simple functions, and how this structure allows us to develop efficient algorithms for computing the gradient of an objective function represented as such a graph with respect to any subset of its variables, in case of neural networks usually with respect to the weights. However, being able to take the gradient is only the first step towards an efficient optimization algorithm. In the next section, we briefly discuss the main first-order optimization algorithms currently used in deep learning.

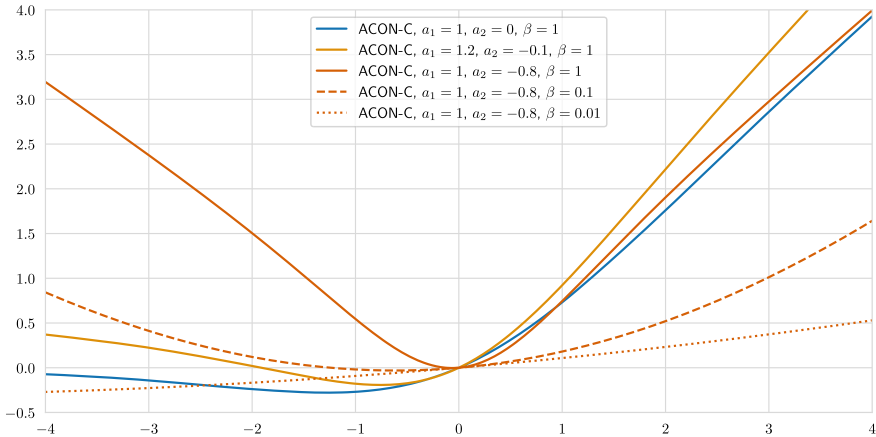

Fig. 2.11 A comparison of activation functions: the ACON-C function with different parameters.

2.4 First-Order Optimization in Deep Learning

In this section, we continue our introduction to deep learning, considering it from the optimization point of view. We have already seen how to compute the gradients, and here we will discuss how to use these gradients to find the local minima of given functions. Throughout this section, we assume that we are given a loss function $f(\mathbf{w}, d)$, where d is a data point and $\mathbf{w}$ is the vector of weights, and the optimization problem in question is to minimize the total loss function over a given dataset D with respect to $\mathbf{w}$:

$$F(\mathbf{w}) = \sum_{d \in D} f(\mathbf{w}, d) \rightarrow_{\mathbf{w}} \min .$$

Algorithm 1 shows the regular "vanilla" gradient descent (GD): at every step, compute the gradient at the current point and move in the opposite direction. In regular GD, a lot depends on the learning rate α. It makes sense that the learning rate should decrease with time, and the first idea would be to choose a fixed schedule for varying α:

- either with linear decay:

$$\alpha = \alpha_0 \left(1 - \frac{t}{T}\right)$$

- or with exponential decay:

$$\alpha = \alpha_0 e^{-\frac{t}{T}} .$$

In both cases, T is called the *temperature* (it does play a role similar to the temperature in statistical mechanics), and the larger it is, the slower the learning rate decays with time.

Algorithm 1: Gradient descent

Initialize $\mathbf{w}_0$, $k := 1$;
repeat
  $\mathbf{w}_{k+1} := \mathbf{w}_k - \alpha \sum_{d \in D} \nabla_{\mathbf{w}} F(\mathbf{w}_k, d)$;
  $k := k + 1$;
until *a stopping condition is met*;

But these are, of course, just the very first ideas that can be much improved. Optimization theory has a whole field of research devoted to gradient descent and how to find the optimal value of α on any given step. We refer to, e.g., books and surveys [84, 96, 625, 633] for a detailed treatment of this and give only a brief overview of the main ideas.

In particular, basic optimization theory known since the 1960s leads to the so-called *Wolfe conditions* and *Armijo rule*. If we are minimizing $f(\mathbf{w})$, and on step k we have already found the direction $\mathbf{p}_k$ to which we need to move—for instance, in gradient descent, we have $\mathbf{p}_k = \nabla_{\mathbf{w}} f(\mathbf{w}_k)$—the problem becomes

$$\min_{\alpha} f(\mathbf{w}_k + \alpha \mathbf{p}_k),$$

a one-dimensional optimization problem.

Studying this problem, researchers have found that

- for $\phi_k(\alpha) = f(\mathbf{w}_k + \alpha \mathbf{p}_k)$, we have $\phi_k'(\alpha) = \nabla f(\mathbf{w}_k + \alpha \mathbf{p}_k)^\top \mathbf{p}_k$, and if $\mathbf{p}_k$ is the direction of descent, then $\phi_k'(0) < 0$;
- the step size α must satisfy the Armijo rule:

$$\phi_k(\alpha) \leq \phi_k(0) + c_1 \alpha \phi_k'(0) \text{ for some } c_1 \in (0, \frac{1}{2});$$

- or even stronger Wolfe conditions, which mean the Armijo rule and, in addition,

$$|\phi_k'(\alpha)| \leq c_2 |\phi_k'(0)|,$$

  i.e., we aim to reduce the projection of the gradient.

The optimization process now should stop according to a stopping condition with respect to the L_2-norm of the gradient, i.e., when $\|\nabla_{\mathbf{w}} f(\mathbf{w}_k)\|^2 \leq \epsilon$ or $\|\nabla_{\mathbf{w}} f(\mathbf{w}_k)\|^2 \leq \epsilon \|\nabla_{\mathbf{w}} f(\mathbf{w}_0)\|^2$.

However, first-order methods such as gradient descent begin to suffer if the scale of different variables is different. A classical example of such behaviour is shown in Figure 2.12, where the three plots show gradient descent optimization for three very simple quadratic functions, all in the form of $x^2 + \rho y^2$ for different values of ρ.

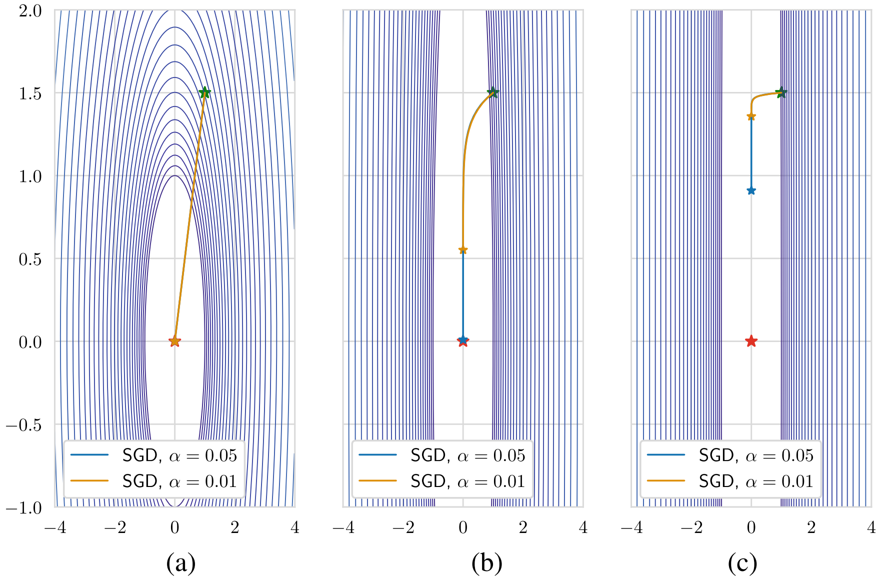

Fig. 2.12 Sample gradient descent optimization with different scale of variables: (a) $f(x, y) = x^2 + y^2$; (b) $f(x, y) = x^2 + \frac{1}{10}y^2$; (c) $f(x, y) = x^2 + \frac{1}{100}y^2$.

The functions are perfectly convex, and SGD (in fact, full GD in this case) should have no trouble at all in finding the optimum. And indeed it doesn't, but as the scale of variables becomes too different, gradient descent slows down to a crawl when it comes to the vertical axis; the plots show how the same learning rates work great for comparable x and y but slow down significantly as they become too different.

Algorithm 2: Stochastic gradient descent with mini-batches

Initialize $\mathbf{w}_0$, $k := 0$;
repeat
  $D_k := \text{Sample}(D)$;
  $\mathbf{w}_{k+1} := \mathbf{w}_k - \alpha \sum_{d \in D_k} \nabla_{\mathbf{w}} f(\mathbf{w}_k, d)$;
  $k := k + 1$;
until *a stopping condition is met*;

Cases like this are very common in deep learning: for instance, weights from different layers of a deep network certainly might have different scales. Therefore, for machine learning problems, it is much better to use *adaptive* gradient descent algorithms that set the scale for different variables adaptively, depending on the optimization landscape. Naturally, the best way to do that would be to pass to *second-order methods*. Theoretically, we could apply Newton's method here:

$$\mathbf{g}_k = \nabla_{\mathbf{w}} f(\mathbf{w}_k), \ \ H_k = \nabla^2_{\mathbf{w}} f(\mathbf{w}_k),$$

and we get that

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha_k H_k^{-1}\mathbf{g}_k.$$

The Armijo rule is applicable here as well: we should choose α_k such that

$$f(\mathbf{w}_{k+1}) \leq f(\mathbf{w}_k) - c_1\alpha_k g_k^\top H_k^{-1}\mathbf{g}_k, \quad \text{where } c_1 \approx 10^{-4}.$$

Using Newton's method to train deep neural networks would be great! Unfortunately, real-life neural networks have a lot of parameters, on the order of thousands or even millions. It is completely impractical to compute and support a Hessian matrix in this case, and even less practical to invert it—note that second-order methods make use of H_k^{-1}.

There exist a wide variety of *quasi-Newton methods* that do not compute the Hessian explicitly but rather approximate it via the values of the gradient. The most famous of them is probably the BFGS algorithm, named after Charles George Broyden, Roger Fletcher, Donald Goldfarb, and David Shanno [237]. The idea is not to compute the Hessian but keep a low-rank approximation and update it via the current value of the gradient. It's a great algorithm, it has versions with bounded memory, and it would also work great to rescale the gradients...

...But a huge Hessian is just the beginning of our troubles. What's more, in the case of machine learning, we cannot really afford gradient descent either! The problem is that the loss function is defined as a sum over the input dataset $\sum_{d\in D} f(\mathbf{w}, d)$, and in reality, it is usually infeasible to go over the entire dataset to make only a single update to the neural network weights. This means that we cannot use the BFGS algorithm and other quasi-Newton methods because we don't have the value of the gradient either.

Therefore, in deep learning, one usually implements *stochastic gradient descent* (SGD), shown in its general form in Algorithm 2: the difference is that on every step, the gradient is computed not over the entire dataset D but over a subsample of the data D_k.

How do we understand stochastic gradient descent formally and how does it fit into optimization theory? Usually, the problem we are trying to solve can be framed as a *stochastic optimization* problem:

$$F(\mathbf{w}) = \mathbb{E}_{q(\mathbf{y})}\left[f(\mathbf{w}, \mathbf{y})\right] \to \min_{\mathbf{w}},$$

where $q(\mathbf{y})$ is some known distribution. The basic example here is the minimization of empirical risk:

$$F(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N} f_i(\mathbf{w}) = \mathbb{E}_{i\sim \mathrm{U}(1,\ldots,N)}\left[f_i(\mathbf{w})\right] \to \min_{\mathbf{w}}.$$

Another important example is provided by minimizing the variational lower bound, but this goes beyond the scope of this section.

This formalization makes it clear what mini-batches are from the formal point of view. Averaging over a mini-batch can be thought of simply as an empirical estimate of the stochastic optimization objective function, computed on a subsample:

$$\hat{F}(\mathbf{w}) = \frac{1}{m}\sum_{i=1}^{m} f(\mathbf{w}, \mathbf{y}_i), \quad \hat{\mathbf{g}}(\mathbf{w}) = \frac{1}{m}\sum_{i=1}^{m} \nabla_{\mathbf{w}} f(\mathbf{w}, \mathbf{y}_i),$$

where m denotes the mini-batch size. Basic mathematical statistics tells us that these estimates have a lot of desirable properties: they are unbiased, they always converge to the true value of the expectation (albeit convergence might be slow), and they are easy to compute.

In general, stochastic gradient descent is motivated by these ideas and can be thought of as basically a Monte Carlo variation of gradient descent. Unfortunately, it still does not mean that we can plug these Monte Carlo estimates into a quasi-Newton method such as BFGS: the variance is huge, the gradient on a single mini-batch usually has little in common with the true gradient, and BFGS would not work with these estimates. It is a very interesting open problem to devise stochastic versions of quasi-Newton methods, but it appears to be a very hard problem.

But SGD has several obvious problems even in the first-order case:

- it never goes in the exactly correct direction;
- moreover, SGD does not even have zero updates at the exact point where $F(\mathbf{w})$ is minimized, i.e., even if we get lucky and reach the minimum, we won't recognize it, and SGD with constant step size will never converge;
- since we know neither $F(\mathbf{w})$ nor $\nabla F(\mathbf{w})$ (only their Monte Carlo estimates with huge variances), we cannot use the Armijo rule and Wolfe conditions to find the optimal step size.

There is a standard analysis that can be applied to SGD; let us look at a single iteration of SGD for some objective function

$$F(\mathbf{w}) = \mathbb{E}_{q(\mathbf{y})}\left[f(\mathbf{w}, \mathbf{y})\right] \to_{\mathbf{w}} \min.$$

In what follows, we denote by $\mathbf{g}_k$ the gradient of F at point $\mathbf{w}_k$, so that

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha_k \hat{\mathbf{g}}_k, \quad \mathbb{E}\left[\hat{\mathbf{g}}_k\right] = \mathbf{g}_k = \nabla F(\mathbf{w}_k).$$

Let us try to estimate the residue of the point on iteration k; denoting by $\mathbf{w}_{\mathrm{opt}}$ the true optimum, we get

$$\|\mathbf{w}_{k+1} - \mathbf{w}_{\mathrm{opt}}\|^2 = \|\mathbf{w}_k - \alpha_k \hat{\mathbf{g}}_k - \mathbf{w}_{\mathrm{opt}}\|^2 = \\ = \|\mathbf{w}_k - \mathbf{w}_{\mathrm{opt}}\|^2 - 2\alpha_k \hat{\mathbf{g}}_k^\top (\mathbf{w}_k - \mathbf{w}_{\mathrm{opt}}) + \alpha_k^2 \|\hat{\mathbf{g}}_k\|^2.$$

Taking the expectation with respect to $q(\mathbf{y})$ on iteration k, we get

$$\mathbb{E}\left[\|\mathbf{w}_{k+1}-\mathbf{w}_{\text{opt}}\|^2\right]=\|\mathbf{w}_k-\mathbf{w}_{\text{opt}}\|^2-2\alpha_k\mathbf{g}_k^\top(\mathbf{w}_k-\mathbf{w}_{\text{opt}})+\alpha_k^2\mathbb{E}\left[\|\hat{\mathbf{g}}_k\|^2\right].$$

And now comes the common step in optimization theory where we make assumptions that are far too strong. In further analysis, let us assume that F is convex; this is, of course, not true in real deep neural networks, but it turns out that the resulting analysis is indeed relevant to what happens in practice, so let's run with it for now. In particular, we can often assume that even in nonconvex optimization, once we get into a neighborhood of a local optimum, the function can be considered to be convex. Specifically, we will use the fact that

$$F(\mathbf{w}_{\text{opt}})\geq F(\mathbf{w}_k)+\mathbf{g}_k^\top(\mathbf{w}_k-\mathbf{w}_{\text{opt}}).$$

Now let us combine this with the above formula for $\mathbb{E}\left[\|\mathbf{w}_{k+1}-\mathbf{w}_{\text{opt}}\|^2\right]$:

$$\begin{aligned}\alpha_k(F(\mathbf{w}_k)-F(\mathbf{w}_{\text{opt}}))\leq\alpha_k\mathbf{g}_k^\top(\mathbf{w}_k-\mathbf{w}_{\text{opt}})=&\\ =\frac{1}{2}\|\mathbf{w}_k-\mathbf{w}_{\text{opt}}\|^2+\frac{1}{2}\alpha_k^2\mathbb{E}\left[\|\hat{\mathbf{g}}_k\|^2\right]-\frac{1}{2}\mathbb{E}\left[\|\mathbf{w}_{k+1}-\mathbf{w}_{\text{opt}}\|^2\right].&\end{aligned}$$

Next, we take the expectation of the left-hand side and sum it up:

$$\begin{aligned}&\sum_{i=0}^{k}\alpha_i(\mathbb{E}\left[F(\mathbf{w}_i)\right]-F(\mathbf{w}_{\text{opt}}))\leq\\ &\leq\frac{1}{2}\|\mathbf{w}_0-\mathbf{w}_{\text{opt}}\|^2+\frac{1}{2}\sum_{i=0}^{k}\alpha_i^2\mathbb{E}\left[\|\hat{\mathbf{g}}_i\|^2\right]-\frac{1}{2}\mathbb{E}\left[\|\mathbf{w}_{k+1}-\mathbf{w}_{\text{opt}}\|^2\right]\leq\\ &\leq\frac{1}{2}\|\mathbf{w}_0-\mathbf{w}_{\text{opt}}\|^2+\frac{1}{2}\sum_{i=0}^{k}\alpha_i^2\mathbb{E}\left[\|\hat{\mathbf{g}}_i\|^2\right].\end{aligned}$$

We have obtained a sum of values of the function in different points with weights α_i. Let us now use the convexity assumption:

$$\begin{aligned}&\mathbb{E}\left[F\left(\frac{\sum_i\alpha_i\mathbf{w}_i}{\sum_i\alpha_i}\right)-F(\mathbf{w}_{\text{opt}})\right]\leq\\ &\leq\frac{\sum_i\alpha_i(\mathbb{E}\left[F(\mathbf{w}_i)\right]-F(\mathbf{w}_{\text{opt}})}{\sum_i\alpha_i}\leq\frac{\frac{1}{2}\|\mathbf{w}_0-\mathbf{w}_{\text{opt}}\|^2+\frac{1}{2}\sum_{i=0}^{k}\alpha_i^2\mathbb{E}\left[\|\hat{\mathbf{g}}_i\|^2\right]}{\sum_i\alpha_i}.\end{aligned}$$

Thus, we have obtained a bound on the residue for some intermediate value in a linear combination of $\mathbf{w}_i$; this is also a common situation in optimization theory, and again, in practice, it usually turns out that there is no difference between the mean and the last point, or the last point $\mathbf{w}_K$ is even better.

In other words, we have found that if the initial residue is bounded by R, i.e., $\|\mathbf{w}_0 - \mathbf{w}_{\mathrm{opt}}\| \le R$, and if the variance of the stochastic gradient is bounded by G, i.e., $\mathbb{E}\left[\|\hat{\mathbf{g}}_k\|^2\right] \le G^2$, then

$$\mathbb{E}\left[F(\hat{\mathbf{w}}_k) - F(\mathbf{w}_{\mathrm{opt}})\right] \le \frac{R^2 + G^2 \sum_{i=0}^{k} \alpha_i^2}{2\sum_{i=0}^{k} \alpha_i}.$$

This is the main formula in the theoretical analysis of stochastic gradient descent. In particular, for a constant step size $\alpha_i = h$, we get that

$$\mathbb{E}\left[F(\hat{\mathbf{w}}_k) - F(\mathbf{w}_{\mathrm{opt}})\right] \le \frac{R^2}{2h(k+1)} + \frac{G^2 h}{2} \to_{k\to\infty} \frac{G^2 h}{2}.$$

Let us summarize the behaviour of SGD that follows from the above analysis:

- SGD comes to an "uncertainty region" of radius $\frac{1}{2}G^2 h$, and this radius is proportional to the step size;
- this means that the faster we walk, the faster we reach the uncertainty region, but the larger this uncertainty region will be; in other words, it makes sense to reduce the step size as optimization progresses;
- SGD converges quite slowly: it is known that the full gradient for convex functions converges at rate $O(1/k)$, and SGD has a convergence rate of only $O(1/\sqrt{k})$;
- on the other hand, the rate of convergence for SGD is also $O(1/k)$ when it is far from the uncertainty region, it slows down only when we have reached it;
- but all of this still depends on G, which in practice we cannot really estimate reliably, and this is also an important point for applications of Bayesian analysis in deep learning.

Algorithm 3: Stochastic gradient descent with momentum

Initialize $\mathbf{w}_0$, $\mathbf{u}_0 := 0$, $k := 0$;
repeat
  $D_k :=$ Sample(D);
  $\mathbf{u}_{k+1} := \gamma \mathbf{u}_k + \alpha \sum_{d \in D_k} \nabla_{\mathbf{w}} f(\mathbf{w}_k, d)$;
  $\mathbf{w}_{k+1} := \mathbf{w}_k - \mathbf{u}_{k+1}$;
  $k := k + 1$;
until *a stopping condition is met*;

All of the above means that we need some further improvements: plain vanilla SGD may be not the best way, there is no clear answer as to how to change the learning rate with time, and the problem of rescaling the gradients in an adaptive way still remains open and important.

Fortunately, there are plenty of improvements that do exactly that. We again refer to [84, 96, 625, 633] for classical optimization techniques and proceed to the approaches that have proven particularly fruitful for optimization in deep learning.

Algorithm 4: Nesterov accelerated gradient

Initialize $\mathbf{w}_0$, $\mathbf{u}_0 := 0$, $k := 0$;
repeat
  $D_k := \text{Sample}(D)$;
  $\mathbf{u}_{k+1} := \gamma \mathbf{u}_k + \alpha \sum_{d \in D_k} \nabla_{\mathbf{w}} f(\mathbf{w}_k - \gamma \mathbf{u}_k, d)$;
  $\mathbf{w}_{k+1} := \mathbf{w}_k - \mathbf{u}_{k+1}$;
  $k := k + 1$;
until *a stopping condition is met*;

2.5 Adaptive Gradient Descent Algorithms

As we have seen in the previous section, basic gradient descent is infeasible in the case of deep neural networks, and stochastic gradient descent needs one to be careful about the choice of the learning rate and, most probably, requires some rescaling along different directions as well. Here, we discuss various ideas in first-order optimization, some classical and some very recent, that have proven to work well in deep learning.

The first idea is the *momentum method*: let us think of the current value of $\mathbf{w}$ as a material point going down the landscape of the function F that we are minimizing, and let us say that, as in real Newtonian physics, this material point carries a part of its momentum from one time moment to the next. In discrete time, it means that in step k we are preserving a part of the previous update $\mathbf{u}_{k-1}$, as shown in Algorithm 3. In real situations, the momentum decay parameter γ is usually close to 1, e.g., $\gamma = 0.9$ or even $\gamma = 0.999$. The momentum method has been a staple of deep learning since at least the mid-1980s; it was proposed for neural networks in the famous *Nature* paper by Rumelhart, Hinton, and Williams [742].

The momentum method is often combined with another important heuristic, *implicit updates*. In regular SGD, using implicit updates means that the gradient is computed not at the point $\mathbf{w}_k$ but at the next point $\mathbf{w}_{k+1}$:

$$\mathbf{w}_{k+1} := \mathbf{w}_k - \alpha \sum_{d \in D_k} \nabla_{\mathbf{w}} f(\mathbf{w}_{k+1}, d).$$

This makes it an implicit equation rather than explicit and can be thought of as the stochastic form of the proximal gradient method. In classical optimization theory, using implicit updates often helps with numerical stability.

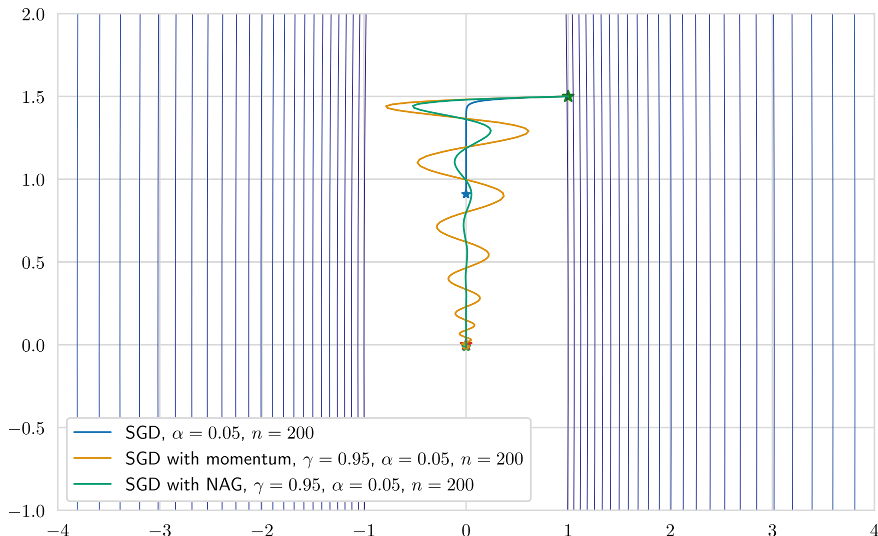

Fig. 2.13 Gradient descent optimization for $F(x, y) = x^2 + \frac{1}{100}y^2$: the effect of momentum.

Applied to the computation of momentum, implicit updates yield the *Nesterov accelerated gradient* (NAG) method, named after its inventor Yurii Nesterov [624]. In this approach, instead of computing the gradient at the point $\mathbf{w}_k$, we first apply the momentum update (after all, we already know that we will need to move in that direction) and then compute the gradient at the point $\mathbf{w}_k + \gamma\mathbf{u}_k$. In this way, the updates are still explicit but numerical stability is much improved, and Nesterov's result was that this version of gradient descent converges faster than the usual version. We show the Nesterov accelerated gradient in Algorithm 4.

Figure 2.13 shows how momentum-based methods solve the problem that we saw in Fig. 2.12. In Fig. 2.13, we consider the same problematic function $F(x, y) = x^2 + \frac{1}{100}y^2$ and show the same number of iterations for every method. Now both regular momentum and Nesterov accelerated gradient converge much faster and in fact have enough time to converge while regular SGD is still crawling towards the optimum. Note how the Nesterov accelerated gradient is more stable and does not oscillate as much as basic momentum-based SGD: this is exactly the stabilization effect of the "lookahead" in computing the gradients.

To move further, note that so far, the learning rate was the same along all directions in the vector $\mathbf{w}$, and we either set a global learning rate α with some schedule of decreasing or chose it with the Armijo rule along the exact chosen direction. Modern adaptive variations of SGD use the following heuristic: let us move faster along the components of $\mathbf{w}$ that change F slowly and move slower when we get to a region of rapid changes in F (which usually means that we are in the vicinity of a local extremum).

The first approach along these lines was *Adagrad* proposed in 2011 [209]. The idea was to keep track of the total accumulated gradient values in the form of their sum of squares; this is vectorized in the form of a diagonal matrix G_k whose diagonal elements contain sums of partial derivatives accumulated up to this point:

Algorithm 5: Adagrad

Initialize $\mathbf{w}_0$, $G_0 := 0$, $k := 0$;
repeat
  $D_k := \text{Sample}(D)$;
  $\mathbf{g}_k := \sum_{d \in D_k} \nabla_{\mathbf{w}} f(\mathbf{w}_k, d)$;
  $G_{k+1} := G_k + \text{diag}(\mathbf{g}_k)$;
  $\mathbf{w}_{k+1} := \mathbf{w}_k - \frac{\alpha}{\sqrt{G_{k+1} + \epsilon}} \mathbf{g}_{k+1}$;
  $k := k + 1$;
until *a stopping condition is met*;

$$G_{k,ii} = \sum_{l=1}^{k} \frac{\partial F_l}{\partial \mathbf{w}_i}, \quad \text{where } F_l(\mathbf{w}) = \sum_{d \in D_l} f(\mathbf{w}, d).$$

Adagrad is summarized in Algorithm 5. The learning rate now becomes adaptive: when the gradients along some direction i become large, the sum of their squares $G_{k,ii}$ also becomes large, and gradient descent slows down along this direction. Thus, one does not have to manually tune the learning rate anymore; in most implementations, the initial learning rate is set to $\alpha = 0.01$ or some other similar constant and left with no change.

However, the main problem of *Adagrad* is obvious as well: while it can slow descent down, it can never let it pick the pace back up. Thus, if the slope of the function F becomes steep in a given direction but then flattens out again, *Adagrad* will keep going very slowly along this direction. The fix for this problem is quite straightforward: instead of a sum of squares of the gradients, let's use an exponential moving average.

Algorithm 6: RMSprop

Initialize $\mathbf{w}_0$, $G_0 := 0$, $k := 0$;
repeat
  $D_k := \text{Sample}(D)$;
  $\mathbf{g}_k := \sum_{d \in D_k} \nabla_{\mathbf{w}} f(\mathbf{w}_k, d)$;
  $G_{k+1} := \gamma G_k + (1 - \gamma)\, \text{diag}(\mathbf{g}_k)$;
  $\mathbf{w}_{k+1} := \mathbf{w}_k - \frac{\alpha}{\sqrt{G_{k+1} + \epsilon}} \mathbf{g}_{k+1}$;
  $k := k + 1$;
until *a stopping condition is met*;

The first attempt at this is the *RMSprop* algorithm, proposed by Geoffrey Hinton in his *Coursera* class but, as far as I know, never officially published. It replaces the sum of squares of gradients $G_{k+1} := G_k + \text{diag}\,(\mathbf{g}_k)$ with a formula that computes the exponential moving average:

$$G_{k+1} := \gamma G_k + (1-\gamma)\,\text{diag}\,(\mathbf{g}_k)\,;$$

Hinton suggested to use $\gamma = 0.9$. We show *RMSprop* in Algorithm 6.

But there is one more, slightly less obvious problem. If you look at the final update rule in *RMSprop*, or actually at the update rule in any of the stochastic gradient descent variations we have considered so far, you can notice that the measurement units in the updates don't match! For instance, in the vanilla SGD, we update

$$\mathbf{w}_{k+1} := \mathbf{w}_k - \alpha \nabla_{\mathbf{w}} F_k(\mathbf{w}_k, d),$$

which means that we are subtracting from $\mathbf{w}$ the partial derivatives of F_k with respect to $\mathbf{w}$. In other words, if $\mathbf{w}$ is measured in, say, seconds and $f(\mathbf{w}, d)$ is measured in meters, we are subtracting meters per second from seconds, hardly a justified operation from the physical point of view! In mathematical terms, this means that the scale of these vectors may differ drastically, leading to mismatches and poor convergence.

Adagrad and *RMSprop* change the units but the problem remains: we are now dividing the gradient update by a square root of the sum of squared gradients, so instead of meters per second we are now subtracting a dimensionless value—hardly a big improvement. Note that in second-order methods, this problem does not arise; in Newton's method, the update rule is $\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha_k H_k^{-1}\mathbf{g}_k$; in the example above, we would get

$$\text{seconds} := \text{seconds} - \alpha \left(\frac{\text{meters}}{\text{second}^2}\right)^{-1} \frac{\text{meters}}{\text{second}},$$

and now the measurement units match nicely.

Algorithm 7: Adadelta

Initialize $\mathbf{w}_0$, $G_0 := 0$, $H_0 := 0$, $\mathbf{u}_0 := 0$, $k := 0$;
repeat
- $D_k :=$ Sample(D);
- $\mathbf{g}_k := \sum_{d\in D_k} \nabla_{\mathbf{w}} f(\mathbf{w}_k, d)$;
- $H_{k+1} := \rho H_k + (1-\rho)\,\text{diag}\,(\mathbf{u}_k)$;
- $G_{k+1} := \gamma G_k + (1-\gamma)\,\text{diag}\,(\mathbf{g}_k)$;
- $\mathbf{u}_{k+1} := \frac{\sqrt{H_{k+1}+\epsilon}}{\sqrt{G_{k+1}+\epsilon}}\mathbf{g}_k$;
- $\mathbf{w}_{k+1} := \mathbf{w}_k - \mathbf{u}_{k+1}$;
- $k := k+1$;

until *a stopping condition is met*;

Algorithm 8: Adam

Initialize $\mathbf{w}_0$, $G_0 := 0$, $\mathbf{m}_0 := 0$, $\mathbf{v}_0 := 0$, $\mathbf{u}_0 := 0$, $k := 1$;
repeat
- $D_k := \text{Sample}(D)$;
- $\mathbf{g}_k := \sum_{d \in D_k} \nabla_{\mathbf{w}} f(\mathbf{w}_k, d)$;
- $\mathbf{m}_k := \beta_1 \mathbf{m}_{k-1} + (1 - \beta_1)\mathbf{g}_k$;
- $\mathbf{v}_k := \beta_2 \mathbf{v}_{k-1} + (1 - \beta_2)\mathbf{g}_k^2$;
- $\hat{\mathbf{m}}_k := \frac{\mathbf{m}_k}{1-\beta_1^k}$, $\hat{\mathbf{v}}_k := \frac{\mathbf{v}_k}{1-\beta_2^k}$;
- $\mathbf{u}_k := \frac{\alpha}{\sqrt{\hat{\mathbf{v}}_k + \epsilon}} \hat{\mathbf{m}}_k$;
- $\mathbf{w}_{k+1} := \mathbf{w}_k - \mathbf{u}_k$;
- $k := k + 1$;

until *a stopping condition is met*;

To fix this problem without resorting to second-order methods, the authors of *Adadelta* [989] propose to add another exponential moving average, this time of the weight updates themselves, adding it to the numerator and thus arriving at the correct measurement units for the update. In other words, in *Adadelta*, we are rescaling the update with respect to the values of the weights, keeping track of the average weights:

$$\mathbf{w}_{k+1} := \mathbf{w}_k - \mathbf{u}_{k+1} = \mathbf{w}_k - \frac{\sqrt{H_{k+1} + \epsilon}}{\sqrt{G_{k+1} + \epsilon}} \mathbf{g}_k,$$

where

$$H_{k+1} = \rho H_k + (1 - \rho)\,\text{diag}\left(\mathbf{u}_k\right),$$

that is, H_k accumulates the weight updates from previous steps. In this way, the updates are properly rescaled, and the measurement units are restored to their proper values.

But that's not the end of the line. The next algorithm, *Adam* (Adaptive Moment Estimation) [454], in many applications remains the algorithm of choice in deep learning up to this day. It is very similar to *Adadelta* and *RMSprop*, but *Adam* also stores an average of the past gradients (that is, an exponential moving average as usual), which acts as a kind of momentum for its updates.

Formally, this means that *Adam* has two parameters for the decay of updates, β_1 and β_2, keeps two exponential moving averages, $\mathbf{m}_k$ for gradients and $\mathbf{v}_k$ for their squares, and computes the update similar to *RMSProp* but with momentum-based $\mathbf{m}_k$ instead of just $\mathbf{g}_k$. Another feature is that since $\mathbf{m}_k$ and $\mathbf{v}_k$ are initialized by zeros, they are biased towards zero, and the authors correct for this bias by dividing over $(1 - \beta_i^k)$:

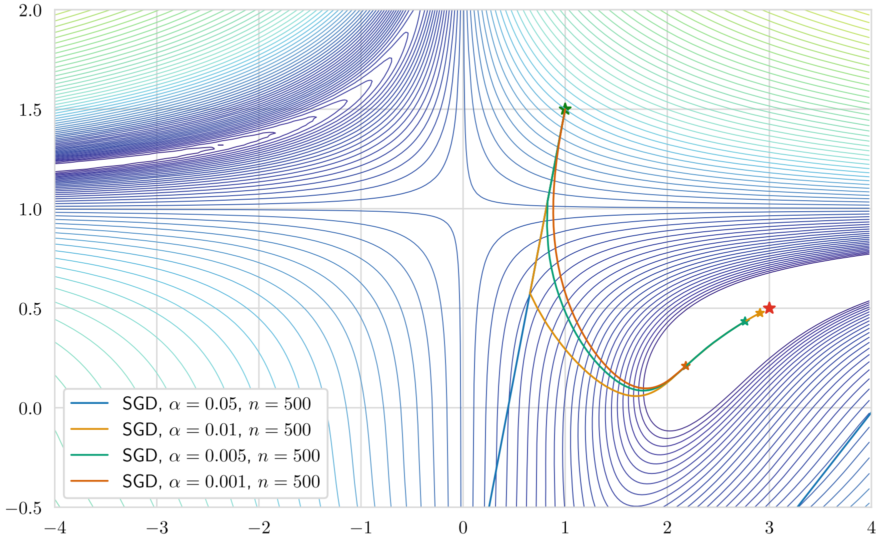

Fig. 2.14 Gradient descent with different learning rates for the Beale function.

$$\mathbf{m}_k = \beta_1 \mathbf{m}_{k-1} + (1 - \beta_1)\mathbf{g}_k,$$
$$\mathbf{v}_k = \beta_2 \mathbf{v}_{k-1} + (1 - \beta_2)\mathbf{g}_k^2.$$
$$\mathbf{u}_k = \frac{\alpha}{\sqrt{\frac{\mathbf{v}_k}{1-\beta_2^k}} + \epsilon} \frac{\mathbf{m}_k}{1 - \beta_1^k}.$$

We give a full description in Algorithm 8.

When *Adam* appeared, it quickly took the field of deep learning by storm. One of its best selling features was that it needed basically no tuning of the hyperparameters: the authors, Diederik Kingma and Jimmy Ba, recommended $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$, and these values work just fine for the vast majority of practical cases. Basically, by now *Adam* is the default method of training deep neural networks, and practitioners turn to something else only if *Adam* fails for some reason.

Before proceeding to a brief overview of other approaches and recent news, let me give an example of all these algorithms in action. For this example, I have chosen a standard function that is very common in examples like this; this is the Beale function

$$F(x, y) = (1.5 - x + xy)^2 + \left(2.25 - x + xy^2\right)^2 + \left(2.625 - x + xy^3\right)^2.$$

It is a simple and continuous but nonconvex function that has an interesting optimization landscape. All optimization algorithms were run starting from the same point $(1, \frac{3}{2})$, and the Beale function has a single global minimum at $(3, \frac{1}{2})$, which is our main goal.

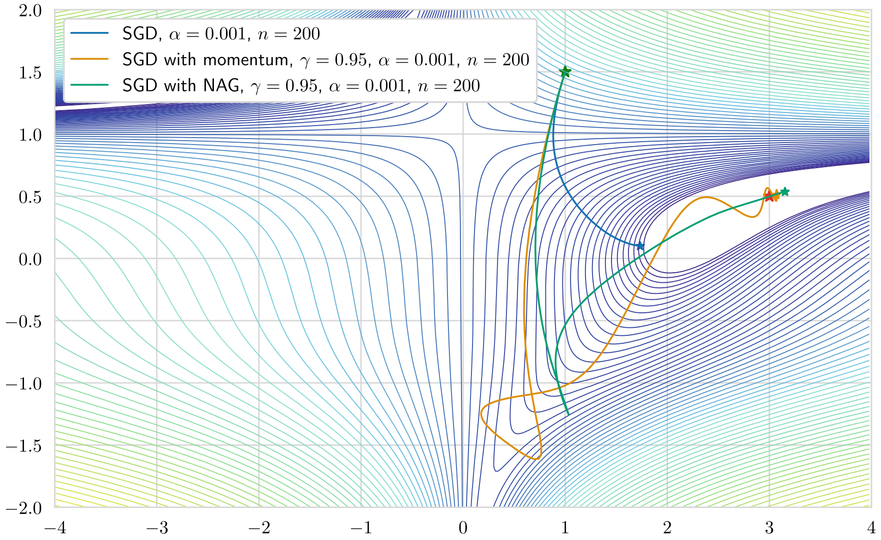

Fig. 2.15 Momentum-based methods for the Beale function.

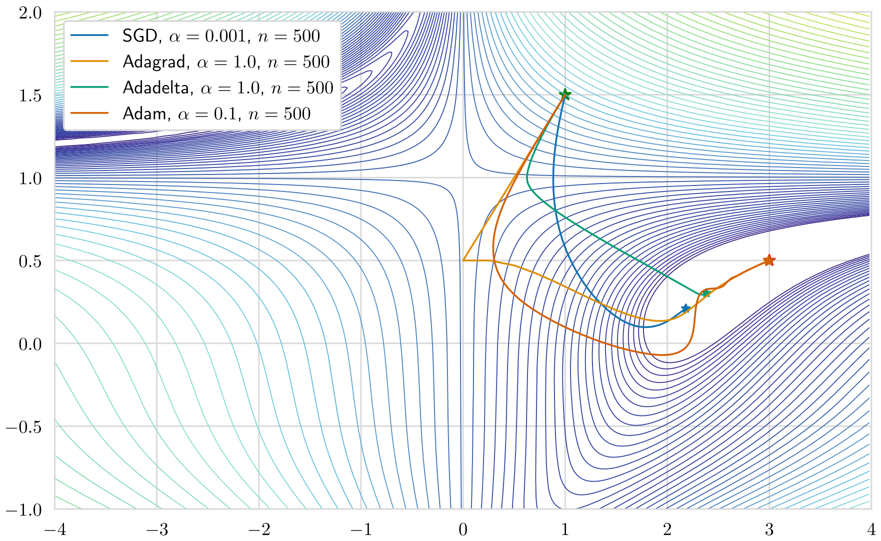

Fig. 2.16 Adaptive gradient descent methods for the Beale function.

Experimental results are shown in Figures 2.14, 2.15, and 2.16. Figure 2.14 shows that even in complex optimization landscapes, one usually can find a learning rate for the basic SGD that would work well. The problem is that this value is far from obvious: for instance, in this case, we see that overshooting the best learning rate (which appears to be around 0.01) even a little can lead to divergence or other undesirable behaviour: note how the plot with $\alpha = 0.05$ quickly gets out of hand. Figure 2.15 shows how momentum-based methods work: we see that for the same initial learning rate and the same number of iterations, SGD with momentum and SGD with Nesterov accelerated gradients find the optimum much faster. We also see the stabilization effect of NAG again: SGD with Nesterov momentum overshoots and oscillates much less than SGD with regular momentum. Finally, Fig. 2.16 shows the behaviour of adaptive gradient descent algorithms; in this specific example, *Adagrad* and *Adam* appear to work best although this two-dimensional example does not really let the adaptive approaches shine.

There have been attempts to explain what is going on with *Adam* and why it is so good. In particular, Heusel et al. in an influential paper [340] showed that stochastic optimization with *Adam* can be described as the dynamics of a heavy ball with friction (HBF), that is, *Adam* follows the differential equation for an HBF in Newtonian mechanics. Averaging over past gradients helps the “ball” (current value of $\mathbf{w}$) get out of small regions with local minima and prefer large “valleys” in the objective function landscape. Heusel et al. use this property to help a GAN generator avoid mode collapse (we will talk much more about GANs in Chapter 4), but the remark is fully general and applies to *Adam* optimization in any context.

There have also been critiques of *Adam* and similar approaches. In another influential paper, Wilson et al. [927] demonstrate that

- when the optimization problem has a lot of local minima, different adaptive algorithms can converge to different minima even from the same starting point;
- in particular, adaptive methods can overfit, i.e., find non-generalizing local solutions;
- and all of this happens not only in the theoretical worst case, which would be natural and fine with us, but in practical examples.

Wilson et al. conclude that adaptive gradient descent methods are not really advantageous over standard SGD and advise to use SGD with proper step size tuning over *Adam* and other algorithms.

Therefore, researchers have continued to search for the holy grail of a fast and universal adaptive gradient descent algorithm. Since *Adam* was presented in 2014, there have been a lot of attempts to improve adaptive gradient descent algorithms further. Since this is not the main subject of the book and since *Adam* still remains mostly the default, I will not give a comprehensive survey of these attempts but will only mention in passing a few of the most interesting ideas.

First of all, *Adam* itself has received several modifications:

- the original *Adam* paper [454] proposed *Adamax*, a modification based on the L_∞-norm instead of L_2-norm for scaling the gradients; in this variation, $\mathbf{m}_k$ is computed as above, and instead of $\mathbf{v}_k$ the scaling is done with

$$\mathbf{v}_k^\infty = \max\left(\beta_a \mathbf{v}_{k-1}, |\mathbf{g}_k|\right),$$

and $\mathbf{v}_k^\infty$ is used instead of $\hat{\mathbf{v}}_k$ in Algorithm 8 (initialization bias does not occur in this case);
- *AMSGrad* [708] is a very similar idea: the authors present an example of a simple problem where the original *Adam* does not converge and fix this by normalizing the running average of the gradient with a maximum of all $\mathbf{v}_t$ up to this point instead of the exponential moving average $\mathbf{v}_t$; in Algorithm 8, it means that we let

$$\mathbf{u}_k := \frac{\alpha}{\sqrt{\mathbf{v}'_k} + \epsilon}\hat{\mathbf{m}}_k$$

for $\mathbf{v}'_k = \max\left(\mathbf{v}'_{k-1}, \mathbf{v}_k\right)$, where max is understood componentwise and $\mathbf{v}_k$ is defined exactly as in Algorithm 8;
- *Nadam* [204] is the modification of *Adam* that uses the Nesterov momentum instead of regular momentum for $\mathbf{m}_k$; expanding one step back, the *Adam* update rule can be written as

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \frac{\alpha}{\sqrt{\hat{\mathbf{v}}_k} + \epsilon}\left(\frac{\beta_1 \mathbf{m}_{k-1}}{1-\beta_1^k} + \frac{(1-\beta_1)\mathbf{g}_k}{1-\beta_1^k}\right) \approx$$
$$\approx \mathbf{w}_k - \frac{\alpha}{\sqrt{\hat{\mathbf{v}}_k} + \epsilon}\left(\beta_1 \hat{\mathbf{m}}_{k-1} + \frac{(1-\beta_1)\mathbf{g}_k}{1-\beta_1^k}\right)$$

(approximate becase we do not distinguish between $1-\beta_1^k$ and $1-\beta_1^{k-1}$ in the denominator), and now we can replace the bias-corrected estimate $\hat{\mathbf{m}}_{k-1}$ with the current estimate $\hat{\mathbf{m}}_k$, thus changing regular momentum into Nesterov's version:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \frac{\alpha}{\sqrt{\hat{\mathbf{v}}_k} + \epsilon}\left(\beta_1 \hat{\mathbf{m}}_k + \frac{(1-\beta_1)\mathbf{g}_k}{1-\beta_1^k}\right);$$

- *QHAdam* (quasi-hyperbolic Adam) [565] replaces both momentum estimators in *Adam*, $\mathbf{v}_k$ and $\mathbf{m}_k$, with their quasi-hyperbolic versions, i.e., with weighted averages between plain SGD and momentum-based *Adam* updates; the update rule in *QHAdam* looks like

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha\frac{(1-\nu_1)\,\mathbf{g}_k + \nu_1 \hat{\mathbf{m}}_k}{(1-\nu_2)\,\mathbf{g}_k^2 + \nu_2 \hat{\mathbf{v}}_k},$$

where $\hat{\mathbf{m}}_k$ and $\hat{\mathbf{v}}_k$ are defined as in Algorithm 8 and ν_1, ν_2 are new constants;

- the critique of Wilson et al., combined with much faster convergence of *Adam* during the initial stages of the optimization process, led to the idea of switching from *Adam* to SGD at some strategic point during the training process [447].

AdamW [556, 557] is probably one of the most interesting *Adam* variations. It goes back to the 1980s, to the original L_2 regularization method for neural networks which was *weight decay* [322]:

$$\mathbf{w}_{k+1} = (1-\beta)\mathbf{w}_k - \alpha \nabla_{\mathbf{x}_k} F_k,$$

where β is the weight decay rate and α is the learning rate. In this formulation, the weights are brought closer to zero. Naturally, it was immediately noted (right in the original paper [322]) that this approach is completely equivalent to changing the objective function F_k:

$$F_k^{\text{reg}}(\mathbf{w}_k) = F_k(\mathbf{w}_k) + \frac{\beta}{2}\|\mathbf{w}_k\|_2^2$$

or even directly changing the gradient:

$$\nabla_{\mathbf{w}_k} F_k^{\text{reg}} = \nabla_{\mathbf{w}_k} F_k + \beta \mathbf{w}_k.$$

But while this equivalence holds for plain vanilla SGD, it does not hold for adaptive variations of gradient descent! The idea of *AdamW* is to go back to the original weight decay and "fix" *Adam* so that the equivalence is restored. The authors show how to do that without losing efficiency, by changing *Adam* updates only very slightly. The only change compared to Algorithm 8 is that now the update $\mathbf{u}_k$ is given by

$$\mathbf{u}_k := \frac{\alpha \hat{\mathbf{m}}_k}{\sqrt{\hat{\mathbf{v}}_k} + \epsilon} + \lambda \mathbf{w}_k$$

instead of adding $\lambda \mathbf{w}_k$ to $\mathbf{g}_k$ as it would happen if *Adam* was straightforwardly applied to a regularized objective function.

It has been shown in [556, 557] that *AdamW* has several important beneficial properties. Apart from improved generalization (in some experiments), it is better than the original *Adam* in decoupling the hyperparameters. This means that the best values of hyperparameters such as initial learning rate α and regularization parameter λ do not depend on each other and thus can be found with independent trials, which makes hyperparameter tuning much easier.

Despite this wide variety of first-order optimization algorithms and their variations, the last word has not yet been said in optimization for deep learning. As often as now, new ideas that at first glance might revolutionize the field fade into obscurity after the original experiments are not confirmed in wider research and engineering practice. One good example of such an idea is *super-convergence* [806], the idea that it is beneficial to change the learning rate with a cyclic schedule, increasing it back to large values from time to time in order to provide additional regularization and

improve generalization power. The original experiments were extremely promising, and the idea of curriculum learning has a long and successful history in deep learning [63] (we will actually return to this idea in a different context, in particular, in Section 6.4). But the "super" in "super-convergence" has not really proven to be true across a wide variety of situations. The idea of increasing the learning rate back has been added to the toolbox of deep learning practicioners, but cyclic learning rates have not become the staple of deep learning.

2.6 Conclusion

To sum up, in this section, we have seen the main ideas that have driven the first-order optimization as applied to deep neural networks over recent years. There has been a lot of progress in adaptive gradient methods: apart from classical momentum-based approaches, we have discussed the recently developed optimization methods that adapt their learning rates differently to different weights. By now, researchers working in applied deep learning mostly treat the optimization question as tentatively solved, using *Adam* or some later variation of it such as *AdamW* by default and falling back to SGD if *Adam* proves to get stuck in local minima too much. However, new variations of first-order adaptive optimization methods continue to appear, and related research keeps going strong.

Second-order methods or their approximations such as quasi-Newton optimization methods remain out of reach. It appears that it would be very hard indeed to develop their variations suitable for stochastic gradient descent with the huge variances inherent in optimizing large datasets by mini-batches. But there are no negative results that I know in this direction either, so who knows, maybe the next big thing in deep learning will be a breakthrough in applying second-order optimization methods or their approximations to neural networks.

I would like to conclude by noting some other interesting directions of study that so far have not quite led to new optimization algorithms but may well do so. In the latest years, researchers have begun to look at deep neural networks and functions expressed by them as objects of research rather than just tools for approximation or optimization. In particular, there have been interesting and enlightening recent studies of the optimization landscape in deep neural networks. I'd like to highlight a few works:

- Li et al. [514] study how the learning rate influences generalization and establishes a connection between the learning rates used in training and the curriculum of which patterns the model "learns" first;
- Huang et al. [370] show that a real-life neural network's optimization landscape has plenty of bad minima that have near-perfect training set accuracy but very bad generalization (test set accuracy); the authors describe this landscape as a "minefield" but find that SGD somehow "miraculously" avoids the bad minima and finds a local minimum with good generalization properties;

- Keskar et al. [446] and He et al. [326] study the landscape of the loss functions commonly used in deep learning, find that "flat" local minima have better generalization properties than "sharp" ones, and discuss which training modes are more likely to fall into flat or sharp local minima;
- Chen et al. [132] put some of this theory into practice by showing how to *deform* the optimization landscape in order to help the optimizer fall into flat minima;
- Izmailov et al. [391] note that better (wider) local optima can be achieved by a very simple trick of *stochastic weight averaging*, where the weights at several points along the SGD trajectory are combined together; this is a simplification of the previously developed *fast geometric ensembling* trick [263];
- Nakkiran et al. [618] discuss the *double descent* phenomenon, where performance gets worse before getting better as the model size increases; double descent had been known for some time, but Nakkiran et al. introduce *effective model complexity*, a numerical measure of a training procedure that might explain double descent;
- Wilson and Izmailov [928, 929] considers the same effects from the Bayesian standpoint, explaining some mysteries of deep learning from the point of view of Bayesian inference and studying the properties of the prior over functions that are implied by regularization used in deep learning.

These are just a few examples, there are many more. This line of research appears to be very promising and actually looks like it is still early in development. Therefore, I expect exciting new developments in the nearest future in this direction.

In the next chapter, we proceed from general remarks about deep learning and optimization to specific neural architectures. We will consider in more detail the field of machine learning where synthetic data is used most widely and with the best results: computer vision.