CHAPTER 1

Optimization and Neural Networks

This chapter contains a basic introduction to the most important concepts of optimization and explains how they are related to neural networks. This chapter doesn't go into detail, leaving longer discussions to the following chapters. But at the end of this chapter, you should have a basic understanding of the most important concepts and challenges related to neural networks. This chapter covers the problem of learning, constrained and unconstrained optimization problems, what optimization algorithms are, and the gradient descent algorithm and its variations (mini-batch and stochastic gradient descent).

A Basic Understanding of Neural Networks

It is very useful to have a basic understanding of what is a neural network (NN) and how it learns. For this introductory section, we consider only what is called supervised learning.[1] Suppose you have a dataset of M tuples (x_i, y_i) with $i = 1, \ldots, M$. The x_i, called input observations or simply inputs, can be anything from images to multidimensional arrays, one-dimensional arrays or even simple numbers. The y_i are outputs (also called target variables or sometimes labels) and can be multidimensional arrays, numbers (for example, the probability of the input observation x_i being of a specific class), or even images. In the most basic formulation, an NN is a mathematical function (sometimes called a *network function*) that takes some kind of input (typically multi-dimensional) called x_i and generate some output. The subscript i indicates one of the inputs from a

[1] Supervised learning (SL) is the machine learning task of learning a function that maps an input to an output based on example input-output pairs (Source: Wikipedia).

U. Michelucci, *Applied Deep Learning with TensorFlow 2*, https://doi.org/10.1007/978-1-4842-8020-1_1

dataset of observations that is at your disposal. The output generated by the network function is called $\hat{y}_i$. The network function normally depends on a certain number N of parameters, which we will indicate with θ_i. We can write this mathematically as

$$\hat{y}_i \equiv f(\boldsymbol{\theta}, x_i)$$

Where we have indicated the parameters in vector form: $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_N) \in R^N$. Figure 1-1 shows a diagram of this idea. The blob in the middle represents the network function f that maps the input x_i to the output. Naturally, the output will depend on the parameters. The idea behind learning is to change the parameters until $\hat{y}_i$ is as close to y_i as possible. There are two very important undefined concepts in the last sentence: first, what "close" means, and second, how you update the parameters in an intelligent way to make $\hat{y}_i$ and y_i "close." We answer those exact two questions in depth in this book.

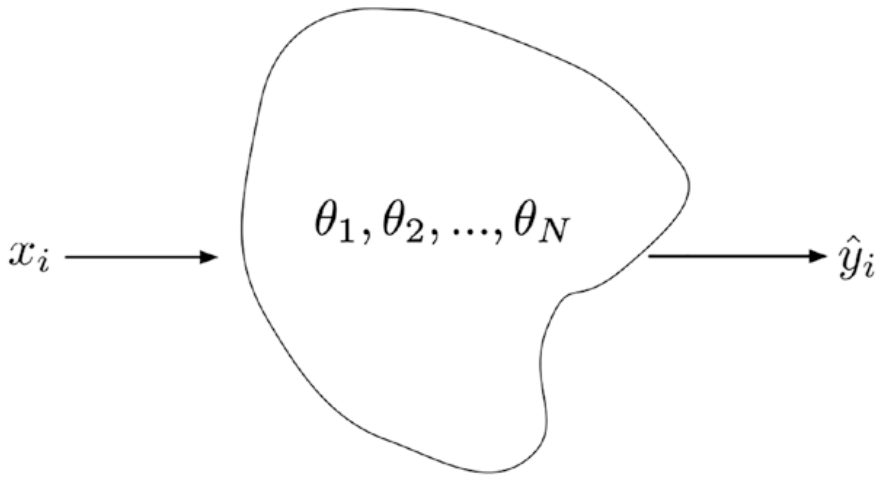

Figure 1-1. *A intuitive diagram of a neural network. x_i are the inputs (for i = 1, …, M), θ_i are the parameters (for i = 1, …, N), and $\hat{y}$ is the output of the network. The network function is depicted by the irregular shape in the middle*

To summarize, a neural network is nothing more than a mathematical function that depends on a set of parameters that are tuned, hopefully in some smart way, to make the network output as close as possible to some expected output. The concept of "close" is not defined here, but for the purposes of this section, a basic understanding is good enough. By the end of this book, you will have a more complete understanding of its meaning.

Note A neural network is nothing more than a mathematical function that depends on a set of parameters that are tuned, hopefully in some smart way, to make the network output as close as possible to some expected output.

The Problem of Learning

A First Definition of Learning

Let's now look at a more mathematical formulation of "learning" in the context of neural networks. For notation's simplicity, let's assume that each input is a mono-dimensional array, $x_i \in R^n$ with $i = 1, \ldots, M$. By the same token, we will assume that the output is a mono-dimensional array, $\hat{y}_i \in \mathbb{R}^k$, with k being some integer. We will assume we have a set of M input observations with the expected target variables $y_i \in R^k$. We also assume that we have a mathematical function $L(\hat{y},y) = L\left(f(\theta,\hat{y}),y\right)$, called a *loss function,* where we have used the vector notation $y = (y_1, \ldots, y_M)$, $\hat{y} = \left(\hat{y}_1,\ldots,\hat{y}_M\right)$ and $\theta = (\theta_1, \ldots, \theta_N)$. This function measures how "close" the expected (y) and predicted ($\hat{y}$) values are, given specific values of the parameters θ_i. We will not yet define how this function looks, as this is not relevant for this discussion. Let's summarize the notation we have defined so far:

- $x_i \in R^n$: Input observations (for this discussion, we assume that they are a mono-dimensional array of dimension $n \in N$). Examples could be age, weight, and height of a person, gray level values of pixels in an image, and so on.
- $y_i \in R^k$: Target variables (what we want the neural network to predict). Examples could be the class of an image, what movie to suggest to a specific viewer, the translated version of a sentence in a different language, and so on.
- $f(\theta, x_i)$: Network function. This function is built with neural networks and depends on the specific architecture used (feed-forward, convolutional, recurrent, etc.).

- $\theta = (\theta_1, \ldots, \theta_N)$: A set of real numbers, also called parameters or weights.
- $L(\hat{y}, y) = L(f(\theta, \hat{y}), y)$: The loss or cost function. This function is a measure of how "close" y and $\hat{y}$ are. Or, in other words, how good the neural network's predictions are.

Those are the fundamentals elements that we need in order to understand neural networks.

[Advanced Section] Assumption in the Formulation

If you already have some experience with neural networks, it is important to discuss one important assumption that we have silently made. Note that skipping this short section in a first reading of this book will not impact the understanding of the rest. If you don't understand the points discussed here, feel free to skip this part and come back to it later.

The most important assumption here can be found in how the loss function $L(\hat{y}, y)$ is written. In fact, as written, it is a function of all the M components of the two vectors $\hat{y}$ and y and this translates in *not* using a mini-batch during the training. The assumption here is that we will measure how good the network's predictions are by considering *all* the inputs and outputs simultaneously. This assumption will be lifted in the following sections and chapters and discussed at length. The experienced reader may notice that this will lead to advanced optimization techniques—stochastic gradient descent and the concept of mini-batch. Using all the M components of the two vectors $\hat{y}$ and y makes the learning process generally slower, although in some situations it's more stable.

A Definition of Learning for Neural Networks

With the notation defined previously, we can now formally define learning in the context of neural networks.

Definition Given a set of tuples (x_i, y_i) with $i = 1, \ldots M$, a mathematical function $f(\theta, \hat{y})$ (the network function), and a function (the loss function) $L(\hat{y}, y) = L(f(\theta, \hat{y}), y)$, the process of *learning* is equivalent to minimizing the loss function with respect to the parameters $\boldsymbol{\theta}$. Or in mathematical notation

$$\min_{\boldsymbol{\theta} \in \mathbb{R}^N} L(f(\theta, \hat{y}), y)$$

Note *Learning* is equivalent to minimizing the loss function with respect to the parameters θ, given a set of tuples (x_i, y_i) with $i = 1, \ldots M$.

The typical term for learning is *training* and that is the one we will use in this book. Basically, training a neural network is nothing more than minimizing a very complicated function that depends on a very large number of parameters (sometimes billions). This presents very difficult technical and mathematical challenges that we discuss at length in this book. But for now, this understanding is sufficient to start understanding how to tackle this problem.

The following sections discuss how to solve the problem of minimizing a function in general and explain the fundamentals theoretical concepts that are necessary to understand more advanced topics. Note that the problem of minimizing a function is called an *optimization problem*.

Constrained vs. Unconstrained Optimization

The problem of minimizing a function as described in the previous section is called an *unconstrained optimization problem*.

The problem of minimizing a function can be generalized by adding constraints to the problem. This can be formulated in the following way: we want to minimize a generic function $g(x)$ subject to a set of constraints

$$\begin{cases} c_i(x) = 0, \quad i = 1, \ldots, C_1 \quad \text{with } C_1 \in \mathbb{N} \\ q_i(x) \geq 0 \quad i = 1, \ldots, C_2 \quad \text{with } C_2 \in \mathbb{N} \end{cases}$$

Where $c_i(x)$ and $q_i(x)$ are constraint functions that define some equations and inequalities that need to be satisfied. In the context of neural networks, you may have the constraint that the output (suppose for a moment that $\hat{y}_i$ is simply a number) must lie in the interval $[0, 1]$. Or maybe that it must be always greater than zero or smaller than a certain value. Or another typical constraint that you will encounter is when you want the network to output only a finite number of outputs, for example, in a classification problem.

Let's consider an example. Suppose that we want our network output to be $\hat{y}_i \in [0,1]$. Our learning problem could be formulated as follows

$$\min_{\theta \in \mathbb{R}^N} L\left(f(\theta,x),y\right) \text{subject to } f(\theta,x_i) \in [0,1] \ i=1,\ldots,M$$

Or even more generally

$$\min_{\theta \in \mathbb{R}^N} L\left(f(\theta,x),y\right) \text{subject to } f(\theta,x) \in [0,1] \ \forall \theta, x$$

This is clearly a *constrained optimization* problem. When dealing with neural networks, this problem is typically reformulated by designing the neural network in such a way that the constraint is automatically satisfied, and learning is brought back to an unconstrained optimization problem.

[Advanced Section] Reducing a Constrained Problem to an Unconstrained Optimization Problem

You may be confused by the previous section and wonder how constraints can be integrated into the network architecture design. This happens typically in the output layer of the network. For example, in the examples discussed in the previous section, to ensure that $f(\theta,\hat{y}_i) \in [0,1] \ i=1,\ldots,M$, it is enough to use the sigmoid function $\sigma(s)$ as an activation function for the output neuron. This will guarantee that the network output will always be between 0 and 1 since the sigmoid function maps any real number to the open interval (0, 1). If the output of the neural network should always be 0 or greater, you could use the ReLU activation function for the output neuron.

Note When dealing with neural networks, constraints are typically built into the network architecture, thus reframing the original constrained optimization problem into an unconstrained one.

Building constraints into the network architecture is extremely useful and it typically makes the learning process much more efficient. Constraints often come from a deep knowledge of the data and the problem you are trying to solve. It pays off to find as many constraints as possible and to build them into the network architecture.

Another example of a constrained optimization problem is when you have a classification problem with k classes. Typically, you want your network to output k real numbers p_i with $i = 1, \ldots, k$, where each p_i could be interpreted as the probability of the input observation being in a specific class. If we want to interpret p_i as the probability, the following equation must be satisfied

$$\sum_{i=1}^{k} p_i = 1$$

This is realized by having k neurons in the output layer and using for them the *softmax* activation function. This step reframes the problem into an unconstrained optimization problem, since the previous equation will be satisfied by the network architecture. If you don't know what the softmax activation function is, don't fret. We discuss it in the following chapters. Keep this example in mind, as it is the key to any classification problem with neural networks.

Absolute and Local Minima of a Function

Many algorithms that minimize a function are, by design, only able to find what is called a "local" minimum, or in other words, a point x_0 at which the function to minimize is smaller than at all other points in any *close* vicinity of x_0. Mathematically speaking x_0 is a local minimum of f if the following is satisfied (in a one-dimensional case)

$$\exists \eta \in \mathbb{R} \text{ such that } f(x) \leq f(x_0) \ \forall x \in [x_0 - \eta, x_0 + \eta]$$

In principle, we want to find the *global minimum* or, in other words, the point for which the function value is the smallest between all possible points. In the case of neural networks, identifying if the minimum is a local or a global minimum is impossible, due to the network function complexity. This is one (albeit not the only one) of the reasons that training large neural networks is such a challenging numerical problem. In the next chapters, we discuss at length what factors[2] may make finding the global minimum easier or more challenging.

[2] Factors include things like weight initialization, optimizer algorithm, optimizer parameters (as the learning rate) and so on.

Optimization Algorithms

So far, we have discussed the idea that learning is nothing less than minimizing a specific function, but we have not touched the issue of how this "minimizing" happens. This is achieved by what is called an "optimization algorithm," whose goal it is to find the location of the (hopefully) absolute minimum. Practically speaking, all unconstrained minimization algorithms require the choice of a starting point, which you denote by x_0. In the example of neural networks, this initial point would be the initial values of the weights. Typically starting from x_0, the optimization algorithms will generate a sequence of iterates $\{x_k\}_{k=0}^{\infty}$ that will converge toward the global minimum.

In all practical applications, only a finite number of terms will be generated, since we cannot generate an infinite number of x_k of course. The sequence will stop when progress can no longer be made (the value of x_k will not change any more[3]) or a specific solution has been reached with sufficient accuracy. Usually, the rule to generate a new x_k will use information about the function f to be minimized and one or more previous values (often properly weighted) of the x_k. In general, there are two main strategies for optimization algorithms: line search and trust regions. Optimizers for neural networks use all a line search approach.

Line Search and Trust Region

In the *line search* approach, the algorithm chooses a direction p_k and searches along this direction for a new value x_{k+1} when trying to minimize a generic function $L(x)$. In general, this approach, once a direction p_k has been chosen, consists in solving

$$\min_{\alpha>0} L(x_k + \alpha p_k)$$

for each iteration. In other words, you would need to choose the optimal α along the direction p_k. In general, this cannot be solved exactly, thus in a practical application (as you will see later), this approach is used by choosing a fixed α, or by reducing it in a way that is easy to calculate (independently of L). α is what is known as the *learning rate* when you deal with neural networks and is one of the most important hyper-parameters[4] when training networks. After you decide on a value for α, the new x_{k+1} is determined with the equation

[3] Don't be annoyed by the basic formulation. We will discuss it in more detail later.

[4] A *hyper-parameter* is a parameter that does not change during training and is not related to the training data. Even if the learning rate will change according to some fixed strategy, is still called a hyper-parameter since it does not change due to the training data.

$$x_{k+1} = x_k + \alpha p_k$$

In the *trust region* approach, the information available on L is used to build a model function m_k (typically quadratic in nature) that approximates f in a sufficiently small region around x_k. Then this approximation is used to choose a new x_{k+1}. This book does not cover trust region approaches, but the interested reader can find a very complete introduction in *Numerical Optimization, 2nd edition* by J. Nocedal and S.J. Wright, published by Springer.

Steepest Descent

The most obvious, and the most used search direction for line search methods is the steepest direction $p_k = -\nabla L(x_k)$. After all, this is the direction along which the function f decreases more rapidly. To prove it, we can use Taylor expansion[5] for $L(x_k + \alpha p)$ and try to determine along which direction the function decreases the most rapidly. We will stop at the first order and write

$$L(x_k + \alpha p) \approx L(x_k) + \alpha p \cdot \nabla L(x_k)$$

assuming that α is small enough. Our question (along which direction does the function L decrease more rapidly?) can be formulated as solving

$$\min_p L(x_k + \alpha p) \quad \text{subject to} \quad \|p\| = 1$$

Where $\|p\| = 1$ is the norm of the vector p (or in other words $\|p\|^2 = (p_1^2 + \ldots + p_n^2)$). Using the Taylor expansion and noting that $f(x_k)$ is a constant, we simply must solve for

$$\min_p p \cdot \nabla L(x_k)$$

Always subject to $\|p\| = 1$. Now, indicating with θ the angle between the direction p and $\nabla L(x_k)$, we can write

$$p \cdot \nabla L(x_k) = \|p\| \|\nabla L(x_k)\| \cos\theta$$

[5] If you don't know what the Taylor expansion is, you can visit Wikipedia to get an idea at https://en.wikipedia.org/wiki/Taylor_series. This is a fundamental tool that is used in calculus for various application.

It's easy to see that this is minimized when $\cos\theta = -1$, or in other words, by choosing the search direction parallel to the gradient of the loss function but pointing in the opposite direction. In other words

$$p = -\frac{\nabla L(x_k)}{\|\nabla L(x_k)\|}$$

as we claimed at the beginning.

Note The *steepest descent* method is a line-search method that searches for a better approximation of the minimum along the direction, minus the gradient of the function for every step. This method is at the basis of the *gradient descent optimizer*.

There are of course other directions that may be used, but for neural networks, those can be neglected. Just to cite an example, possibly the most important is the Newton direction, which can be derived from the second-order Taylor expansion of $f(x_k + p)$, but it requires you to know the Hessian $\nabla^2 L(x)$.

The Gradient Descent Algorithm

The gradient descent (GD) optimizer finds x_{k+1} by using the gradient of the function L according to the formula

$$x_{k+1} = x_k - \alpha \nabla L(x_k).$$

Thus, the GD algorithm is simply a line-search algorithm that searches for better approximations along the steepest descent direction. We can create a simple one-dimensional example ($x \in R$) and try the algorithm. Let's suppose we want to minimize the function

$$L(x) = x^2$$

This has a clear minimum at $x = 0$ as this is a simple quadratic form. If you know how to find the minimum of a function with calculus, it's easy to see that

$$\frac{dL(x)}{dx} = 0$$

implies that $2x = 0 \rightarrow x = 0$. This is indeed a minimum since

$$\frac{d^2 L(x)}{dx^2} = 2 > 0$$

How does the GD algorithm work in this case? The algorithm will generate a sequence of x_k by using the formula $x_{k+1} = x_k - 2\alpha x_k$ (remember we are trying to minimize $f(x) = x^2$). We need of course to choose an initial value x_0 and a step α. For a first try, let's choose $x_0 = 1$ and $\alpha = 0.1$. The sequence can be seen in Table 1-1.

Table 1-1. *The Sequence x_k Generated for the Function $L(x) = x^2$ with the Parameters $x_0 = 1$ and $\alpha = 0.1$*

k	**x_k $x_0 = 1$**
0	1
1	0.8
2	0.64
3	0.512
…	…
40	0.00013
…	…
500	$3.5 \cdot 10^{-49}$

From Table 1-1 it should be evident how, albeit slowly, the GD algorithm converges toward the right answer, $x = 0$. That sounds good, right? What could go wrong? Not everything is so easy, and in the GD there is a marvelous hidden complexity. Let's rewrite the formula that is used to generate the sequence x_k:

$$x_{k+1} = x_k - 2\alpha x_k = x_k(1 - 2\alpha)$$

Consider for example the value $\alpha = 1$. In this case $x_{k+1} = -x_k$. It is easy to see that this generates an oscillating sequence that never converges. In fact, it is easy to calculate that $x_1 = -1$, $x_2 = 1$ and so on. An oscillating sequence will always be generated for all values of $1 - 2\alpha < 0$, or for $\alpha > \frac{1}{2}$. In Figure 1-2, you can see a plot of the sequence x_k for various values of the parameter α.

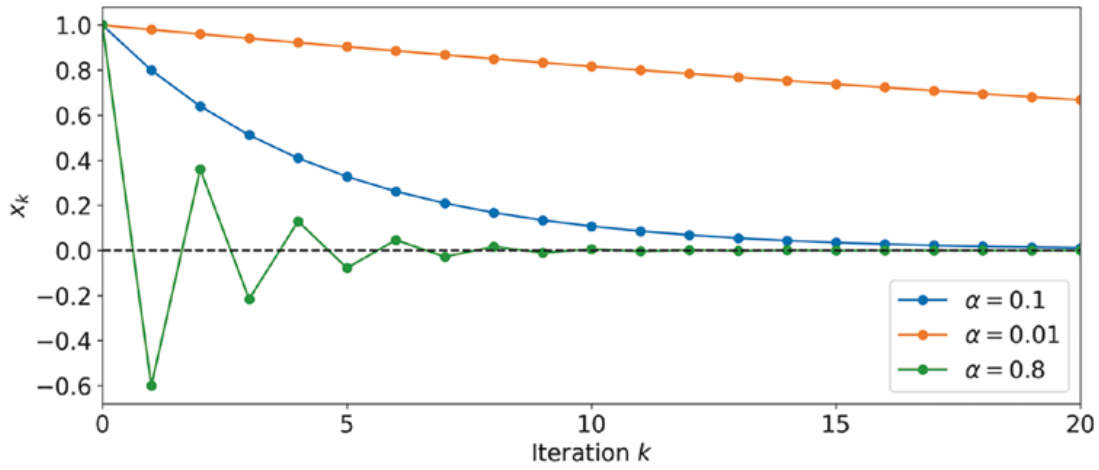

Figure 1-2. *The sequence x_k generated for the function $L(x) = x^2$ for various values of α*

From Figure 1-2, when α is small, the convergence is very small (orange line), and as we discussed, for a value $\alpha > \frac{1}{2}$ (green line), an oscillating sequence is generated. It is interesting to note how this oscillating sequence converges quite faster than the others. The value $\alpha = 1$ generates a sequence that does not converge. But what happens for $\alpha > 1$? This is a very interesting case, as it turns out that the sequence diverges (albeit oscillating from positive to negative values). In Figure 1-3, you can see the plot of the sequence for $\alpha = 1.01$.

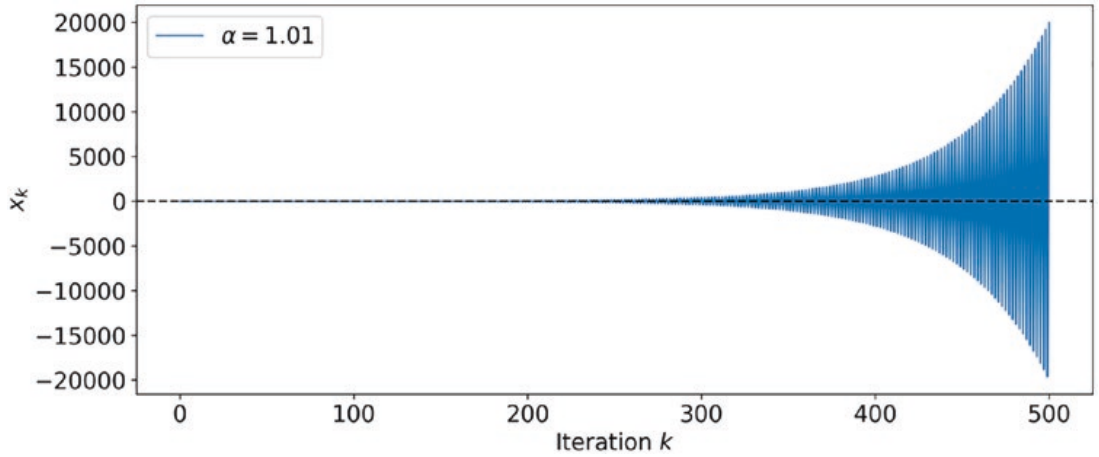

Figure 1-3. *The sequence x_k for the parameter $\alpha = 1.01$. The sequence oscillates from positive to negative values while diverging in absolute value*

You can clearly see how it diverges. Note that from a numerical point of view, it is easy to get NaN (if you are using Python) or errors. If you are trying neural networks and you get NaN for your loss function (for example), one possible reason may be a learning rate that is too big.

Note The learning rate is possibly one of the most important hyper-parameters that you have to decide on when training neural networks. Choose one that's too small, and the training will be very slow, but choose one that's too big, and the training will not converge! More advanced optimizers (Adam for example) try to compensate this shortcoming by effectively varying the learning rate[6] dynamically, but the initial value is still important.

Now I must admit this is a very trivial case. In fact, the formula for x_k can also be written as

$$x_k = x_0 \left(1-2\alpha\right)^k$$

Therefore, it's easy to see that this sequence converges for $|1 - 2\alpha| < 1$ and diverges for $|1 - 2\alpha| > 1$. For $|1 - 2\alpha| = 1$, it stays at 1 and if $1 - 2\alpha = -1$, it oscillates between 1 and −1. Still, it is instructive to see how important the role of the learning rate is when using the GD.

Choosing the Right Learning Rate

You may be wondering how to choose the right α at this point. This is a good question but unfortunately there is no real precise answer, and some clarifications are in order. In all practical cases, you will not use the plain GD algorithm. Consider that, for example, in TensorFlow 2.X, the GD is not even available out of the box, due to its inefficiency. But in general, to check if the (in some cases only the initial) learning rate is optimal, you can follow these steps, assuming you are trying to minimize a function $L(x)$:

[6] To be precise, as we discuss next chapters, the Adam optimizer does not change the learning rate but uses a different algorithm that is very similar to, but not the same as, the GD. It is said that Adam updates the learning rate dynamically.

1. Choose an initial learning rate. Typical values[7] are 10^{-2} or 10^{-3}.
2. Let your optimizer run for a certain number of iterations, saving the $L(x_k)$ each time.
3. Plot the sequence $L(x_k)$. This sequence should show a convergent behavior. From the plot, you can get an idea whether the learning is too small (slow convergence) or too big (divergence). For example, Figure 1-4 shows the sequence $L(x_k)$ for the example we discussed in the previous section. The figure tells us that using $\alpha = 0.01$ (orange line) is very slow. Trying larger values for α shows how convergence can be faster (blue and green). With $\alpha = 0.1$ after 12-13 iterations, you already have a good approximation of the minimum, while for $\alpha = 0.01$ you are still very far from a solution.

Note When training neural networks, always check the behavior of your loss function. This will give you important information about how the training process is going.

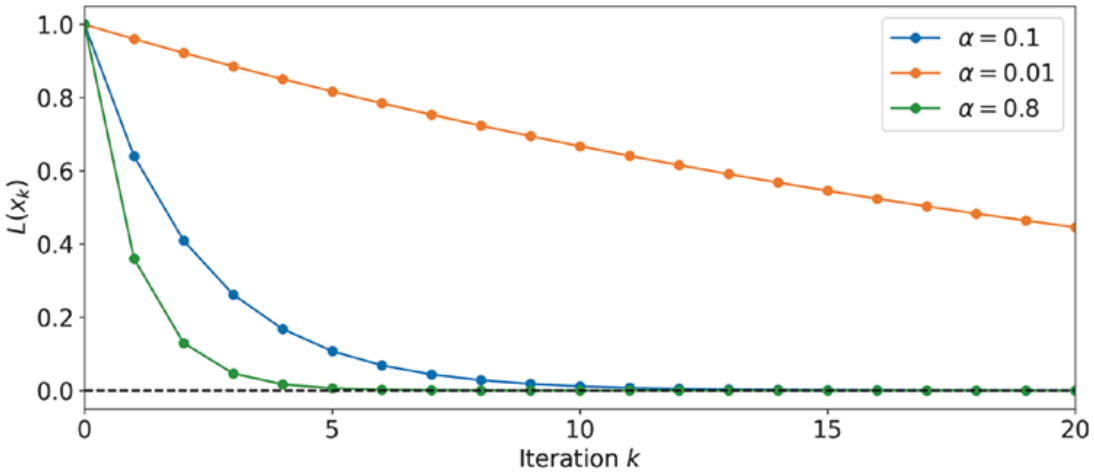

Figure 1-4. *The sequence $L(x_k)$ for the function $L(x) = x^2$ for various values of α*

[7] For example, the Adam optimizer in TensorFlow 2.X uses 0.001 as the standard learning rate, unless you specify otherwise.

This is why, when training neural networks, it is important to always check the behavior[8] of the loss function that you are trying to minimize. Never assume that your model is converging without checking the sequence $L(x_k)$.

Variations of GD

To understand variations of GD, the easiest way is to start with the loss function. As mentioned at the beginning of the chapter, in the "The Problem of Learning" section, the goal is to minimize the loss function $L\left(f\left(\theta,\hat{y}\right),y\right)$, where we have used the vector notation $y = (y_1, \ldots, y_M)$, $\hat{y} = \left(\hat{y}_1, \ldots, \hat{y}_M\right)$ and $\theta = (\theta_1, \ldots, \theta_N)$. In other words, we have at our disposal M input tuples that we can use. In the plain version of GD, the loss function is written as

$$L\left(f\left(\theta,\hat{y}\right),y\right) = \frac{1}{M}\sum_{i=1}^{M} l_i\left(f\left(\theta,\hat{y}_i\right),y_i\right)$$

Where l_i is the loss function evaluated over a single observation. For example, we could have a one-dimensional regression problem where our loss function is the mean square error (MSE). In this case we would have

$$l_i\left(f\left(\theta,\hat{y}_i\right),y_i\right) = \left|f\left(\theta,\hat{y}_i\right) - y_i\right|^2$$

And therefore

$$L\left(f\left(\theta,\hat{y}\right),y\right) = \frac{1}{M}\sum_{i=1}^{M}\left|f\left(\theta,\hat{y}_i\right) - y_i\right|^2$$

That is the classical formula for the MSE that you may have already seen. In plain GD, we would use this formula to evaluate the gradient that we need to minimize L. Using all M observations has pros and cons.

Advantages:

- Plain GD shows a stable convergence behavior

[8] Tools such as TensorBoard (from TensorFlow) have been built with exactly this problem in mind, to provide a real-time check about how the training is going.

Disadvantages:

- Usually, this algorithm is implemented in such a way that all the dataset must be in memory; therefore, it is computationally intensive.
- This algorithm is typically very slow with very big datasets.

Variations of the GD are based on the idea of considering only some of the observations in the sum in the previous equation instead of all M. The two most important variations are called *mini-batch GD* (MBGD, where you consider a small number of observations $m < M$) and *stochastic GD* (SGD, where you consider only one observation at a time). Let's look at both in detail, starting with the mini-batch GD.

Mini-Batch GD

To clarify the idea behind the method, we can write the loss function as

$$L_m\left(f\left(\theta,\hat{y}\right),y\right)=\frac{1}{M}\sum_{i=1}^{m}\left|f\left(\theta,\hat{y}_i\right)-y_i\right|^2$$

Where we have introduced $m \in \mathbb{N}$ with $m < M$, called *batch size*. L_m is defined by summing over m observations sampled from the initial dataset.

The mini-batch GD is implemented according to the following algorithm:

1. A mini-batch size m is chosen. Typical values are 32, 64, 128, or 256 (note that the mini-batch size m does not have to be a power of 2, and it can be any number, such as 137 or 17).
2. $N_b=\left\lfloor\frac{M}{m}\right\rfloor+1$ subsets of observations are created[9] by sampling each time m observation from the initial dataset S without repetition. We indicate them with $S_1,S_2,\ldots,S_{N_b}$. Note that in general if M is not a multiple of m the last batch, S_{N_b} may have a number of observations smaller than m.
3. The parameters θ are updated N_b times using the GD algorithm with the gradient of L_m evaluated over the observations in S_i for $i = 1, \ldots, N_b$.
4. Repeat Step 3 until the desired result is achieved (for example, the loss function does not vary much anymore).

[9] The symbol $\lfloor x \rfloor$ indicates the integer part of x.

When training neural networks, you may have heard the term "epoch" instead of iteration. An epoch is *finished* after all the data has been used in the previous algorithm. Let's look at an example. Suppose we have $M = 1000$ and we choose $m = 100$. The parameters θ will be updated using 100 input observations each time. After ten iterations ($\frac{M}{m}$) the network will have used all M observations for its training. At this point, it is said that one epoch is finished. One epoch in this example consists of ten times the parameters being updated (or ten iterations). Here are the advantages and disadvantages.

Advantages:

- The parameters update frequency is higher than with plain gradient descent but lower than SGD, therefore allowing for a more robust convergence than SGD
- This method is computationally much more efficient than plain gradient descent or Stochastic GD since fewer calculations (as in SGD) and resources (as in Plain GD) are needed.
- This variation is by far the fastest of the three and the most commonly used.

Disadvantages:

- The use of this variation introduces a new hyper-parameter that needs to be tuned: the batch size (the number of observations in the mini-batch).

Note An epoch is *finished* after all the input data has been used to update the parameters of the neural network. Remember that in one epoch, the parameters of the network may be updated many times.

Stochastic GD

SGD is a very common version of the GD, and it simply is the mini-batch version with $m = 1$. This involves updating the parameters of the network by using one observation at a time for the loss function. This also has advantages and disadvantages.

Advantages:

- The frequent updates provide an easy way to check how the model learning is going (you don't need to wait until all the dataset has been considered).
- In a few problems this algorithm may be faster than plain gradient descent.
- The model is intrinsically noisy and that may help the model avoid local minima when trying to find the absolute minimum of the cost function.

Disadvantages:

- On large datasets, this method is quite slow, as it's very computationally intensive due to the continuous updates.
- The fact that the algorithm is noisy can make it hard for the algorithm to settle on a minimum for the cost function, and the convergence may not be as stable as expected.

How to Choose the Right Mini-Batch Size

So, what is the right mini-batch size m? Typical values used by practitioners are in the order of 100 or less. For example, the TensorFlow standard value (if you don't specify otherwise) is 32. Why is this value so special? To understand why, you need to study the behavior of MBGD for various choices of m. To make it resemble real cases, consider the MNIST dataset. You may have already seen it. It is a dataset that contains 70,000 hand-written digits from 0 to 9. The images are gray-level 28x28 pixel images. We will build a classifier with a neural network with 16 neurons using the ReLU activation function and use the Adam optimizer. Note that if you don't know what I am talking about, you can skip those details. The following discussion can be followed even without understanding the details of how the network is designed.

Secondly, using Adam is only for practical reasons, as in TensorFlow the MBGD is not available out of the box. But the conclusions continue to be valid. We have trained the network for ten epochs on 60,000 training images and then measured the running time[10]

[10] I have run these tests on Google Colab, and at the time of this writing, that meant an Intel Xeon CPU @ 2.20GHz and a GPU Tesla T4 with 15 GB memory.

needed, the reached value of the loss function, and the accuracy at the end of the training. We used the following values for the mini-batch size m: 60000 (effectively using all the data, so no mini-batches), 20000, 5000, 500, 50, 10, and 1. Note that while for $m = 10$ the required time is 2.34 min, when using $m = 1$, 19.18 minutes are needed for ten epochs!

Figure 1-4 shows the results of this study.

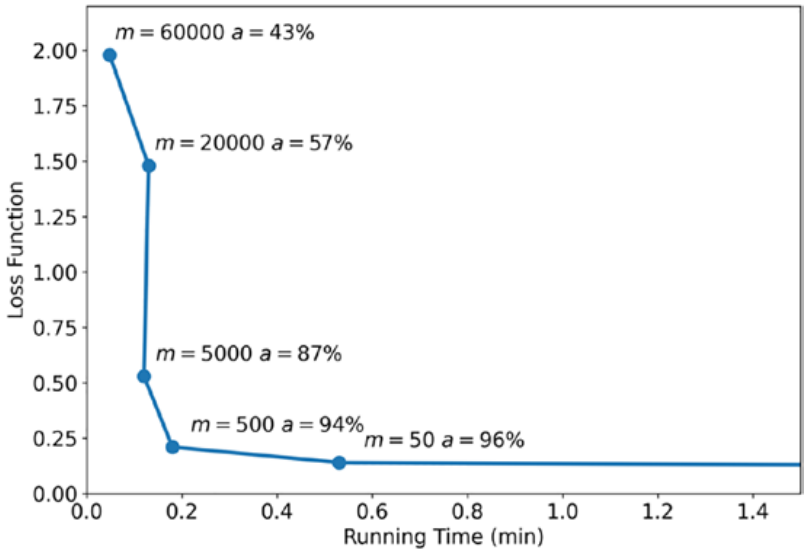

Figure 1-5. *A plot of the loss function reached after ten epochs on the MNIST dataset plotted vs. the running time needed. a indicates the accuracy reached*

Let's see what Figure 1-5 is telling us. When we use $m = 60000$, the running time needed for ten epochs is the lowest, but the accuracy is quite low. Decreasing m increases the accuracy quite rapidly, until we reach the "elbow." Between $m = 500$ and $m = 50$, the behavior changes. Decreasing m does not increase the accuracy much, but the running time becomes larger and larger. So, when we reach the elbow, a decrease in m is no longer advantageous. As you will notice, around the elbow, m is in the order of 100. Figure 1-5 shows why typical values for m are in the order of 100. Of course, the optimal value is dependent on the data and some testing is required, but in most cases a value around 100 is a very good starting point.

Note A good starting point for the mini-batch size is in the order of 100. The optimal value is dependent on the data you use and on the neural network architecture you train, and testing is required to find the optimal value.

[Advanced Section] SGD and Fractals

In the previous sections, we discussed how choosing the wrong learning rate can make the convergence slow or even diverge. But the discussion was for a one-dimensional case and thus was very simple. This section shows you how much complexity is hidden when using SGD. You'll see how specific ranges of the learning rate make the convergence chaotic (in the mathematical sense of the word). This is one of the many hidden gems that you can find when dealing with optimization problems. Let's consider a problem[11] in which the M inputs $x^{[i]}$ are bi-dimensional, in other words $x^{[i]} = \left(x_1^{[i]}, x_2^{[i]}\right) \in \mathbb{R}^2$. We call the target variables $y^{[i]}$. The optimization problem we are trying to solve involves minimizing this function

$$L = \frac{1}{2}\sum_{i=1}^{M}\left(f\left(x_1^{[i]}, x_2^{[i]}\right) - y^{[i]}\right)^2$$

with

$$f\left(x_1^{[i]}, x_2^{[i]}\right) = w_1 x_1^{[i]} + w_2\ x_2^{[i]}$$

This is a simple linear combination of the inputs. The problem is simple enough right? We minimize the MSE (Mean Square Error) and try to find the best parameters w_1 and w_2 that minimize L. Consider $M = 3$ inputs as well. In particular, to make it more concrete, consider the following input matrix[12]

$$X = \begin{pmatrix} 0 & 1 \\ 1 & \frac{1}{2} \\ 1 & -\frac{1}{2} \end{pmatrix}$$

We write our labels in matrix form as well

$$Y = \begin{pmatrix} 0 \\ 4 \\ 0 \end{pmatrix}$$

[11] This problem is an adaptation of the one described in *Rojas, R. (2013). Neural Networks: A Systematic Introduction.* Springer Science & Business Media.

[12] Note that all the inputs can be written in matrix form for simplicity.

Note that what we show you here is not dependent on the numerical values. You can reproduce the results with different values without a problem. Let's first find the minimum of L exactly (since in this easy case, we can do that). To do that, we need simply to derive L and solve the two equations

$$\frac{\partial L}{\partial w_1}=0; \qquad \frac{\partial L}{\partial w_2}=0$$

The calculations are boring but not overly complex. By solving these two equations, you will find that the minimum is at $x^*=\left(2,\frac{4}{3}\right)$.

Exercises

EXERCISE 1

Solve these two equations

$$\frac{\partial L}{\partial w_1}=0; \qquad \frac{\partial L}{\partial w_2}=0$$

And prove that L has its global minimum at $x^*=\left(2,\frac{4}{3}\right)$.

To implement an SGD optimizer, the following algorithm can be followed:

1. Choose a learning rate α.
2. Choose a random value between {1,2,3} and assign it to i.
3. Update the parameters w_1, w_2 by using $l_i=\frac{1}{2}\left(f\left(x_1^{[i]},x_2^{[i]}\right)-y^{[i]}\right)^2$; in other words, use l_i to calculate the derivatives to update the weights according to the gradient descent rule $w_j \rightarrow w_j - \alpha\partial l_i/\partial w_j$ for $j = 1, 2$. Each time, save the values w_1, w_2, for example in a Python list.
4. Repeat Steps 2 and 3 a certain number of times, N.

By following the previous algorithm, you can plot in the (w_1, w_2) space all the points you obtained and saved in Step 3. Those are the all the values that the two parameters w_1 and w_2 will assume during the optimization procedure. Figure 1-6 shows the result for $\alpha = 0.65$. The result is nothing short of amazing.

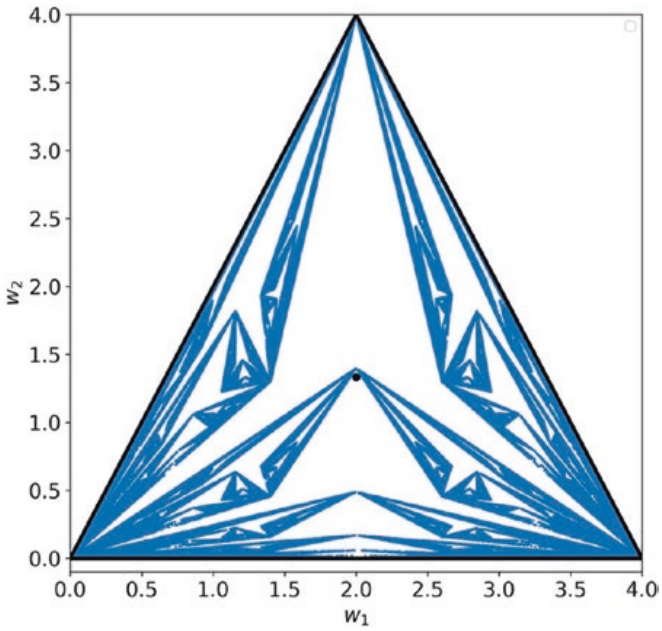

Figure 1-6. *Each blue point is a tuple of values (w_1, w_2) that are generated by using SGD as described in the section for a value of the learning rate $\alpha = 0.65$. The plot has been obtained using $4 \cdot 10^5$ iterations*

EXERCISE 2 (LEVEL: DIFFICULT)

Can you derive the equations of the lines delimiting the triangle from the input matrix X?

EXERCISE 3

Try to reproduce the image by implementing the SGD algorithm as described in this section from scratch. If you are stuck, you can find a complete implementation in the online version of the book.

It can be shown that what you see in Figure 1-6 is indeed a fractal. The mathematical proof is way beyond the scope of this book, but if you are interested, you can consult the book *Fractals Everywhere*, by M.F. Barnsley, published by Dover. One of the main property of fractals is that when you zoom in, you will find the same structure that you observe at a larger scale. To convince you that this is happening, Figure 1-7 shows you a

detail of Figure 1-6. In the zoomed area you can observe the same kind of structure that you see at a larger scale.

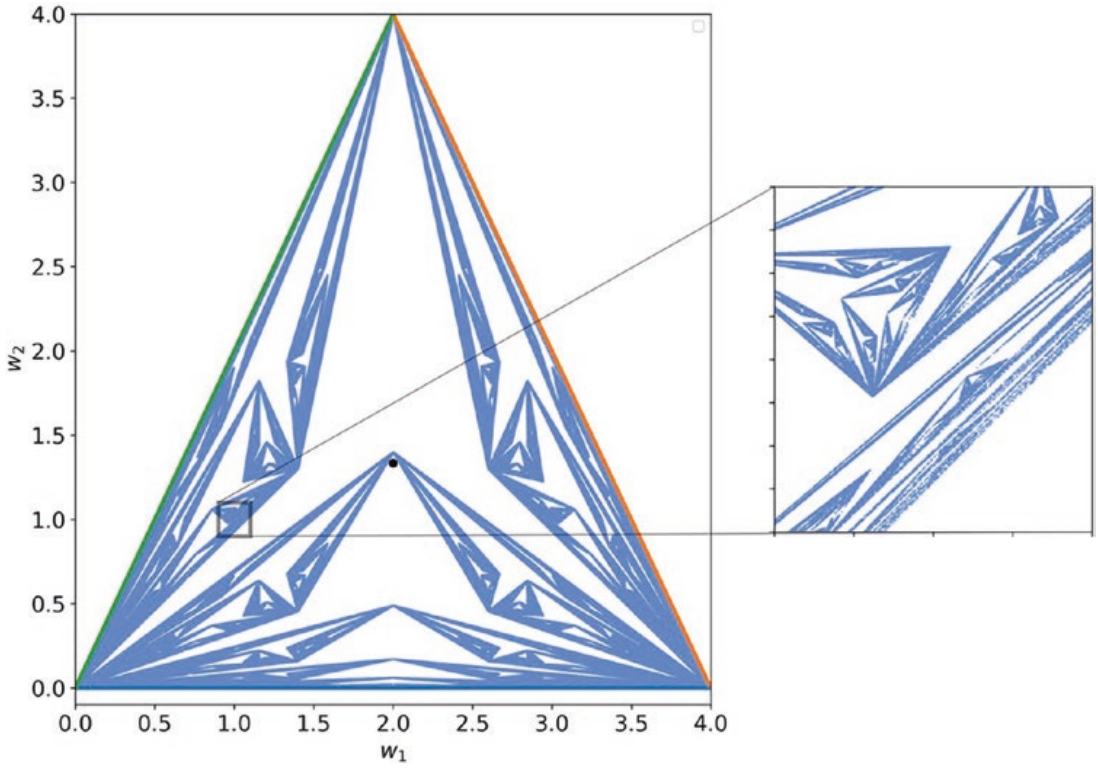

Figure 1-7. *A zoomed region that shows the fractal nature that the SGD algorithm can generate. This picture has been generated with a learning rate of $\alpha = 0.65$ and with 10^7 iterations. In the zoomed region, you can clearly see the same kind of structure that you observe at a larger scale on the left. The zoomed region is less sharp than the one on the left since only a fraction of the 10^7 points happen to be in the small regioned zoomed in*

The particular structure of the fractal depends on the learning rate. In Figure 1-8, you can see the fractal structure for different learning rates, from 0.65 to 1.0. It is fascinating to see how the structure changes, showing the great complexity that is hidden in using SGD.

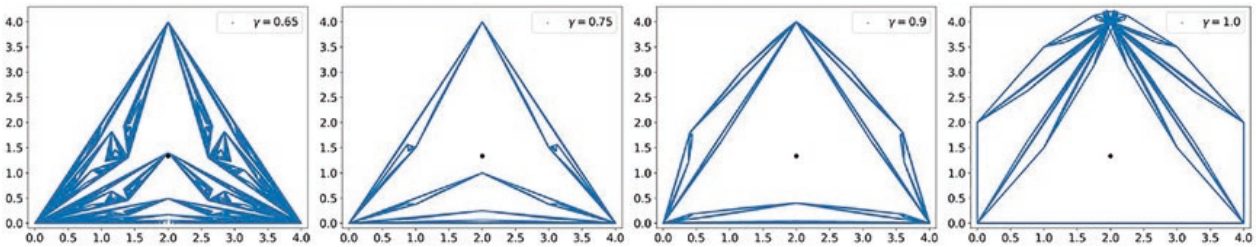

Figure 1-8. *Fractal shapes obtained by SGD for different learning rates. Note in the figure the learning rate is indicated with γ.*

When using smaller learning rates, at a certain point the fractal structure completely disappears quite suddenly, leaving an unstructured cloud of points, as you can see in Figure 1-9. The smaller the learning rate, the smaller the cloud of points.

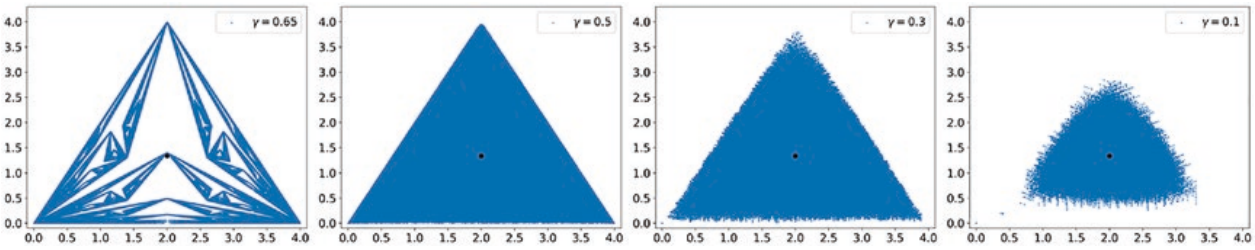

Figure 1-9. *By choosing smaller and smaller learning rates, the fractal structures completely disappear, leaving an unstructured cloud of points centered on the global minimum x^* of L.*

Finally, by choosing a very small learning rate, for example $\alpha = 5 \cdot 10^{-4}$, SGD delivers the behavior you would expect, meaning the algorithm converges and remains close to the expected minimum. You can see how the plot looks in Figure 1-10.

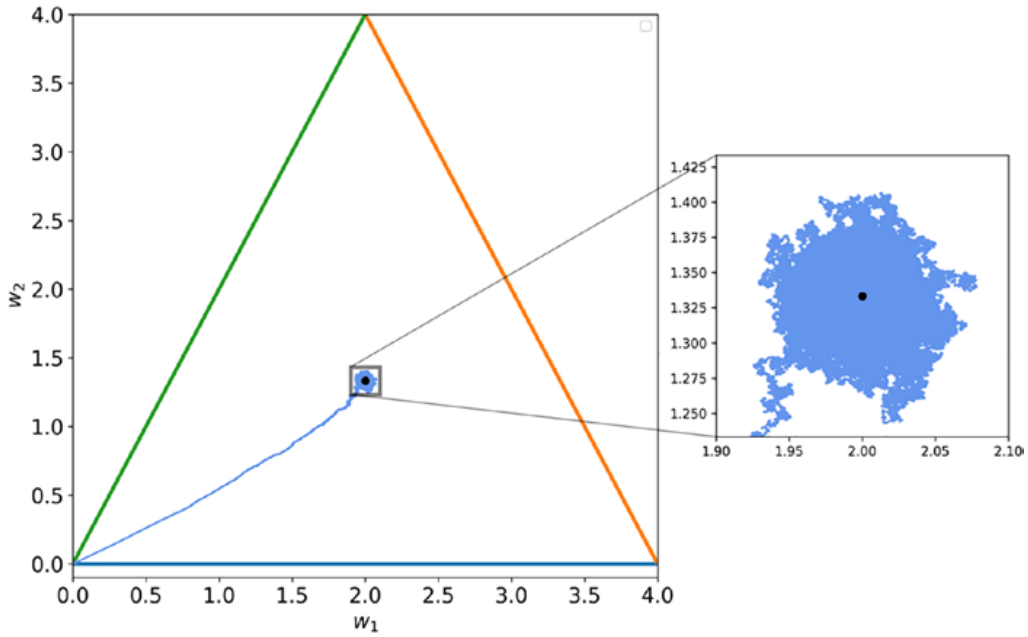

Figure 1-10. *By choosing a very small learning rate of $\alpha = 5 \cdot 10^{-4}$, SGD will move in the direction of the expected global minimum x^* and remain in its vicinity, as can be seen in the zoomed-in region. Still, SGD continues to deliver points that remain around x^*, but never converge to it. The smaller the learning rate, the smaller the cloud of points around x^**

EXERCISE 4 (LEVEL: DIFFICULT)

Prove that each of the updates done with SGD, as described in the previous section, moves the point in parameter space in the direction perpendicular to one of the three lines that describe the triangle in the previous figures. In other words, the direction between two subsequent updates of the parameters (w_1, w_2) and $\left(w_1 - \gamma \frac{\partial L}{\partial w_1}, w_2 - \frac{\partial L}{\partial w_2} \right)$ is perpendicular to one of the three lines that delimitate the triangle in the previous figures. Note that this is not an easy exercise, and you can find some tips in the online version of the book if you get stuck.

Note The gradient descent algorithm, especially in its stochastic version, has incredible hidden complexity, even for a trivial case, as the one described in the previous section. This is the reason that training neural networks can be so difficult and tricky, and why choosing the right learning rate and optimizer is so important.

Conclusion

You should now have all the ingredients to (at least at a basic level) understand what it means for neural networks to learn. We have not yet covered how to build a neural network, except for the fact that it is a very complicated function f of the inputs that depend on a large number of parameters. We go into a lot more detail about how neurons work, how non-linearity is introduced, and so much more.

Let's now summarize what we discussed in this chapter. To train neural networks, you need the following major ingredients:

- A neural network architecture, namely a way of getting from an input x to an answer $\hat{y}$ (remember the function f?) that can be tuned by changing a large number of parameters.
- A set of input observations x_k (possibly a large number of them) with the expected values we want to predict (we are dealing with supervised learning).

- An optimizer, or in other words, an algorithm that can find the best parameters of the network to get the outputs as close as possible to what you expect.

This chapter discussed these points in a basic way. It is important that you get the main idea behind what training neural networks means. The next chapters discuss each of the three points in more detail and include a lot of examples to make the discussion as clear as possible. We build on what's discussed here to bring you to a point where you can use these more advanced techniques for your projects.