Optimization and neural networks

This chapter contains a very easy introduction to the most important concepts about optimization and how they are related to the neural networks. In this chapter we will not go into many details, leaving longer discussions to the following chapters. But at the end of this chapter the student should have at least an intuitive understanding of the most important concepts and challenges of neural networks. Here I will discuss the problem of learning, constrained and unconstrained optimization problems, what optimization algorithms are and discuss at length the gradient descent algorithm and its variations (mini-batch and stochastic gradient descent).

# An intuitive understanding of neural networks

It is very useful to have an intuitive understanding of what a neural network (NN) is and how it learns. For this introductory section, we will consider only what is called supervised learning[[1]](#footnote-1). Let us suppose we have a dataset of tuples with . The , called input observations or simply inputs, can be anything from images to multidimensional arrays, to one-dimensional arrays or even simple numbers. The outputs (also called target variables or sometime labels) can be multidimensional arrays, numbers (for example the probability of the input observation of being of a specific class) or even images. In the most basic formulation, a NN is a mathematical function (sometime called *network function*) that takes some kind of input (typically multi-dimensional) (the subscript indicates that we have a dataset of input observations at disposal, and we are now considering only the one) and with it calculate some output . This function depends on a certain number of parameters, that we will indicate with . We can write this mathematically as

Where we have indicated the parameters in vector form . In Figure 1.1 you can see an intuitive diagram of the idea. The blob in the middle represents the network function that maps the input to the output. Naturally the output will depend on the parameters. The idea behind learning is to change the parameters until is as close as as possible. There are two very important undefined concepts in the last sentence: firstly what “close” means, and secondly how can we update the parameters in an intelligent way to make and “close”. We will answer those exact two questions in depth in this book.

Diagram

Description automatically generated

Figure 1.1: a diagram that gives an intuitive understanding of what is a neural network. are the inputs (for , are numbers (or parameters) (for , and is the output of the network. The network itself is the depicted intuitively as the irregular shape in the middle of the figure.

To summarize a neural network is nothing else 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 an intuitive understanding will be perfectly enough. At the end of this book the student will have a much more complete understanding of its meaning.

**Note**: a neural network is nothing else 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 us now gives a more mathematical formulation of “learning” in the context of neural networks. For notation simplicity let’s assume that each input is a mono-dimensional array with . In the same way we will assume that the output will be a mono-dimensional array   with some integer. We will assume to have a set of input observations with expected target variables . We also assume that we will have a mathematical function called Loss function, where we have used the vector notation ,  and . This function will be a measure of how “close” expected () and predicted () values are, given specific values of the parameters . We will not define yet how this function may be looking like as this is not relevant for this discussion yet. Let us summarize the notation we have defined so far.

* : input observations (for this discussion we will assume that they are a mono-dimensional array of dimension ). Examples could be age, weight and height of a person, gray level values of pixels in an image and so on.
* : target variables (what we would like the neural network to predict). Examples could be 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.
* : network function. This function will be built with neural networks and will depend on the specific architecture used (feed-forward, convolutional, recurrent, etc.).
* : a set of real numbers, also called parameters or weights.
* : loss or cost function. This function is a measure of how “close” and are. Or in other words how good the neural network’s prediction are.

Those are the fundamentals elements that we will need to understand the basics of learning with neural networks.

### *[Advanced Section]* Assumption in the formulation

For the student that already have some experience with neural networks it is important to discuss one important assumption that have silently been made. Note that skipping this short section in a first reading of this book will not impact the understanding of the rest. In case 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 is written. In fact, as written, it is a function of all the components of the two vectors and andthistranslatesin **not** using any 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 as stochastic gradient descent and the concept of mini-batch. Using all the components of the two vectors and makes the learning generally slower, although in some situations more stable.

## **A definition** of learning for neural networks

With the notation defined previously we can now give a formal definition of learning in the context of neural networks.

**Definition** Given a set of tuples ) with , a mathematical function (the network function) and a function (the Loss function) the problem of *learning*is equivalent to minimize the loss function with respect to the parameters . Or in mathematical notation

**Note:***learning*is equivalent to minimize the loss function with respect to the parameters given a set of tuples ) with .

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

In what follows we will discuss how to solve the problem of minimizing a function in general and discuss 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 what 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 subject to a set of constraints

Where and are constraint functions that define some equations and inequalities that needs to be satisfied. In the context of neural networks, you may have the constraint that the output (suppose for a moment that   is simply a number) must lie in the interval . Or maybe that it must be always greater than zero or smaller than a certain value. Or another typical constraint that we will encounter is when you want the network to output only a finite number of outputs, for example in a classification problem.

Let’s make an example. Let’s suppose that we want our network output to be . Our learning problem could be formulated as

Or even more generally

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 to an unconstrained optimization problem

The student may be confused by the previous section and wonder how constraints can be integrated in 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 it is enough to use the sigmoid function as 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 . If the output of the neural network should always be 0 or greater one could use the ReLU activation function for the output neuron.

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

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

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

This is realized by having neurons in the output layer and use 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 is the softmax activation function, don’t fret. We will discuss it in the following chapters but 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 at which the function to minimize is smaller than at all other points in any *close* vicinity of . Mathematically speaking is a local minimum of if the following is satisfied (in a one-dimensional case)

In principle we would like 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 found is a local or a global minimum is impossible, due to the network function complexity and this is one (albeit not the only one) of the reasons because training large neural networks is such a challenging numerical problem. In the next chapters we will discuss at length what factors[[2]](#footnote-2) may make finding the global minimum easier or more challenging.

## 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” looks like. This is achieved with what is called an “optimization algorithm”, whose goal is to find the location of the (hopefully) absolute minimum. Practically all unconstrained minimization algorithms require the choice of a starting point, that we will denote by . In the example of neural networks this initial point would be the initial values of the weights. Typically starting from , the optimization algorithms will generate a sequence of iterates that hopefully 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 of course. The sequence will stop when no progress can be made anymore (the value of will not change anymore too much[[3]](#footnote-3)) or a specific solution has been reached with sufficient accuracy. Usually, the rule to generate a new will use information about the function to be minimized and one or more previous values (often properly weighted) of the . 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 and searches along this direction for a new value when trying to minimize a generic function . In general, this approach, once a direction has been chosen consists in solving

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

In the *trust region* approach, the information available on is used to build a model function (typically quadratic in nature) that approximates in a sufficiently small region around . Then this approximation is used to choose a new . In this book we will 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 . After all, this is the direction along which the function decreases more rapidly. To prove it, we can use Taylor expansion[[5]](#footnote-5) for and try to determine along which direction the function decreases the most rapidly. We will stop at the first order and write

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

Where is the norm of the vector (or in other words ). Using the Taylor expansion and noting that is a constant we simply must solve for

Always subject to . Now, indicating with the angle between the direction and we can write

And is easy to see that this is minimized when , or in other words choosing

as we claimed at the beginning.

**Note** The *steepest descent* method is a line search method that search for a better approximation for the minimum along the direction of 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 (unless you are active in research in optimization algorithms, in that case you probably don’t need to read this book). Just to cite an example, possibly the most important is the Newton direction, that can be derived from the second order Taylor expansion of , but that requires to know the Hessian .

### The Gradient Descent Algorithm

The gradient descent (GD) optimizer finds by using the gradient of the function according to the formula

Thus, the GD algorithm is simply a line search algorithm that search for better approximations along the steepest descent direction. We can make a simple one-dimensional example () and try the algorithm. Let’s suppose we want to minimize the function

This has a clear minimum at as this is a simple quadratic form. If you know how to find the minimum of a function with calculus is easy to see that

implies that . This is indeed a minimum since

How the GD algorithm looks like in this case? The algorithm will generate a sequence of by using the formula (remember we are trying to minimize ). We need of course to choose an initial value and a step . For a first try, let us choose and The sequence can be seen in Table 1.

|  |  |
| --- | --- |
|  |  |
| 0 | 1 |
| 1 | 0.8 |
| 2 | 0.64 |
| 3 | 0.512 |
| … | … |
| 40 | 0.00013 |
| … | … |
| 500 | 3.5 |

Table 1: The sequence generated for the function with the parameters and .

From Table 1 is evident how, albeit slowly, the GD algorithm converges toward the right answer . 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 re-write the formula that in this case is used to find to generate the sequence :

Consider for example the value . In this case . It is easy to see that this generates an oscillating sequence that never converges. In fact, it is easy to calculate that , and so on. An oscillating sequence will always be generated for all values of , or for In Figure 1.2 you can see a plot of the sequence for various values of the parameter .

Chart, line chart

Description automatically generated

Figure 1.2: The sequence generated for the function for various values of .

From Figure 1.2 when is small, the convergence is very small (orange line), and as we discussed, for a value (green line) an oscillating sequence is generated. It is interesting to note how this oscillating sequence converges quite faster than the others. The value generates a sequence that does not converge but what happens for ? 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 .

Chart

Description automatically generated

Figure 1.3: The sequence for the parameter . 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 well be a learning rate that is too big.

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

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

And therefore, is easy to see that this sequence converges for and diverges for . For it stays at 1 and if it oscillates between and . Still, it is quite 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 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 initial) learning rate is the optimal you can follow the steps, assuming you are trying to minimize a function :

1. You choose an initial learning rate. Typical values[[7]](#footnote-7) are or .
2. You let your optimizer run for a certain number of iterations saving each time the
3. You plot the sequence . This sequence should show a convergent behavior. From the plot you can get an idea if the learning is too small (slow convergence) or too big (divergence). For example, Figure 1.4 shows the sequence for the example we discussed in the previous section. The Figure would tell me that using (orange line) is very slow. Trying larger values for makes clear how convergence can be faster (blue and green). With after 12-13 iterations you already have a good approximation of the minimum, while for you are still very far.

**Note** When training neural networks always check the behavior of your Loss Function. This will give you important information on how the training process is going.

Chart, line chart

Description automatically generated

Figure 1.4: The sequence for the function for various values of .

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

## Variations of GD

To understand variations of GD, the easiest way is to start with the loss function. As we mentioned at the beginning of the chapter, in the *problem of learning* section, our goal is to minimize the loss function where we have used the vector notation ,  and . In other words, we have at our disposal input tuples that we can use. In the plain version of GD, the loss function is written as

Where is the loss function evaluated over **one 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

And therefore

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 . Using all observations have pros and cons.

**Advantages**

* Plain GD shows a stable convergence behavior

**Downsides**

* Usually, this algorithm is implemented in such a way that all the dataset must be in memory, therefore it is computationally quite intensive
* This algorithm is typically very slow for 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 . The two most important variations are called *mini-batch GD* (MBGD, where you consider a small number of observations ) and *Stochastic GD* (SGD, where you consider only one observation at a time). Let us 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

Where we have introduced with called here *batch size.* is defined by summing over observations sampled from the initial dataset.

**Mini-batch GD** is implemented according to the following algorithm:

1. A mini-batch size is chosen. Typical values are 32, 64, 128 or 256 (note that the mini-batch size does not have to be a power of 2, and could be any number as 137 or 17);
2. subsets of observations are created[[9]](#footnote-9) by sampling each time observations from the initial dataset without repetition. We will indicate them with . Note that in general if is not a multiple of the last batch, may have a number of observations smaller than ;
3. The parameters are updated times using the GD algorithm with the gradient of evaluated over the observations in for ;
4. Repeat point 3 until the desired result is achieved (for example the loss function does not vary that much anymore)

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 us make an example. Suppose we have and we choose . The parameters will be updated using each time input observations. After 10 iterations () the network will have used all observations for its training. At this point it is said that one epoch is finished. One epoch in this example will consist of 10 parameters update (or 10 iterations). Here are advantages and disadvantages.

**Advantages**

* The model update frequency is higher than with plain gradient descent but lower than SGD (see below), therefore allowing for a more robust convergence than SGD
* This method is computationally much more efficient than plain gradient descent or Stochastic GD since less calculations (as in SGD) and resources (as in Plain GD) are needed
* This variation is by far (as we will see later) the fastest of the three and the most used

**Downsides**

* The use of this variation introduces a new hyperparameter that needs to be tuned: the batch size (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 also a very commonly used version of the GD, and it simply is the mini-batch version with . This means updating the parameters of the network by using one observation at a time for the loss function. This has of course also advantages and disadvantages.

**Advantages**

* The frequent updates allow an easy check on 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 to avoid local minima when trying to find the absolute minimum of the cost function

**Downsides**

* On large dataset this method is quite slow, since is 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 be not so stable as expected.

## How to choose the right mini-batch size

Now what is the right mini-batch size ? Typical values used by practitioners are of the order of 100 or less. For example, TensorFlow standard value (if you don’t specify otherwise) is 32. Why this value? What is so special? To understand we need to study the behavior of MBGD for various choices of . To make it resembling real cases, consider as dataset the MNIST one. You may have already seen it. It is a dataset that contains 70000 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 with the ReLu activation function and we will use the Adam optimizer. Firstly, if you don’t exactly what I am talking about you can simply skip those details. The discussion below 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 even available out of the box. But the conclusions continue to be valid. I have trained the network for 10 epochs (remember what an epoch is?) on 60000 training images and I have measured the running time[[10]](#footnote-10) needed, the reached value of the loss function and the accuracy at the end of the training. I have used the following values for the mini-batch size : 60000 (effectively using all the data, so no mini-batch), 20000, 5000, 500, 50, 10 and 1. Note that while for the required time is 2.34 min, when using 19.18 minutes are needed for 10 epochs!

In Figure 4 you can see the results of this study.

Chart

Description automatically generated with medium confidence

Figure 1.5: A plot of the loss function value reached after 10 epochs on the MNIST dataset plotted vs. the running time needed. indicates the accuracy reached.

Let us see what Figure 1.5 tells us. When we use , the running time needed for 10 epochs is the lowest, but the accuracy reached is quite low. Decreasing increase the accuracy quite rapidly until we reach the “elbow”. Between and the behavior changes. Decreasing does not increase the accuracy much, but the running time becomes larger and larger. So, when we reach the elbow decrease does not bring many advantages anymore. As you will notice, around the elbow is of the order of 100. Figure 1.5 is an explanation why typical values for are of 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 of 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

We have discussed in the previous sections how choosing the wrong learning rate can make the convergence slow or even diverge. But the discussion done was for a one-dimensional case and thus was very simple. In this section I want to show you how much complexity is hidden when using SGD. I want you to see how specific ranges of the learning rate make the convergence chaotic (in the mathematical sense of the word), thus showing one of the many hidden gems that you can find when dealing with optimization problems. Let us consider a problem[[11]](#footnote-11) in which our inputs are bi-dimensional, in other words . We call our target variables . The optimization problem we are trying to solve involve minimizing the function

with

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 and that minimize . Let simply even more the problem. Consider inputs. In particular, to make it more concrete, consider the following input matrix[[12]](#footnote-12)

We write our labels also in matrix form

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

Calculations are boring but not overly complex. By solving the two above mentioned equations you will find that the minimum is at .

Exercise

Solve the two equations

And prove that has its global minimum at .

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

1. Choose a learning rate ;
2. Choose a random value between from {1,2,3} and assign it to
3. Update the parameters by using ; in other words we use to calculate the derivatives to update the weights according to the Gradient Descent rule for ; each time save the values , for example in a python list;
4. Repeat points 2 and 3 a certain number of times .

By following the previous algorithm, we can plot in the space all the points we have obtained and saved in point 3 above. Those are the all the values that the two parameters and will assume during the optimization procedure. In Figure 1.6 you can see the result for . The result is nothing short of amazing.

Shape

Description automatically generated with low confidence

Figure 1.6: each blue point is a tuple of values that are generated by using SGD as described in the section for a value of the learning rate . The plot has been obtained with iterations.

EXERCISE

This is difficult question, but can you derive the equations of the lines delimiting the triangle from the input matrix ?

EXERCISE

Try to reproduce the image by implementing the SGD algorithm as described in this section from scratch. In case 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 5 is indeed a fractal. The mathematical proof is way beyond the scope of this book, but in case you are interested you can consult the beautiful book *Fractals Everywhere****,*** by M.F. Barnsley published by Dover. One of the main property of fractals, is that when you zoom in a detail, you will find the same structure that you observe at a larger scale. To convince you, at least intuitively, that this is what is happening Figure 6 shows you a detail of Figure 1.5. In the zoomed area you can observe the same kind of structure that you see at a larger scale.

A picture containing shape

Description automatically generated

Figure 1.6: a zoomed region that shows the fractal nature that the SGD algorithm can generate. This picture has been generated with a learning rate of and with 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 points happen to be in the small regioned zoomed in.

The particular structure of the fractal depends on the learning rate. In Figure 7 you can see the fractal structure for different learning rates, from 0.65 to 1.0. It is quite fascinating to see how the structure change, showing the great complexity that is hidden in using SGD.

A picture containing radar chart

Description automatically generated

Figure 7: fractal shapes obtained by SGD for different learning rates.

Now 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 8. The smaller the learning rate, the smaller the cloud of points.

## A picture containing icon Description automatically generated

Figure 8: Choosing smaller and smaller learning rates fractal structures completely disappear, leaving an unstructured cloud of points centered on the global minimum of .

Finally, by choosing a very small learning rate, for example , SGD delivers the behavior we would expect, meaning the algorithm converges and remain close to the expected minimum. You can see how the plot looks like in Figure 9.

A picture containing polygon

Description automatically generated

Figure 9: by choosing a very small learning rate SGD will move into the direction of the expected global minimum and remain in its vicinity, as it can be seen from the zoomed in region. Still, SGD continues to deliver points that remain around , but never converge to it. The smaller the learning rate, the smaller the cloud of points around .

EXERCISE (HARD)

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 and 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 in case you are stuck

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

## Conclusions

Now you should have all the ingredients to (at least on an intuitive level) understand what it means learning with neural networks. We have not yet spoken about how to build a neural network, except for the fact that it is a very complicated function of the inputs that depend on a large number of parameters. We will go into a lot of details on how neurons work, how non-linearity is introduced and so much more.

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

* A neural network architecture, namely a way of getting from an input an answer (remember the function we discussed?) that can be tuned by changing a large number of parameters;
* A set of input observations (possibly a large number of them) with the expected values we want to predict (remember 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.

In this chapter we discussed those points in an almost intuitive way. It is important that you get the main idea behind what training neural networks means. In the next chapters I will discuss each of the three points in a lot of details and I will give you lot of examples to make the discussion as clear as possible. I will build on what we discuss here to bring you to a point where you can use the more advanced techniques for your projects.

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). [↑](#footnote-ref-1)
2. Factors include things like weight initialization, optimizer algorithm, optimizer parameters (as the learning rate) and so on. [↑](#footnote-ref-2)
3. Don’t be annoyed by the intuitive formulation. We will discuss it better later on. [↑](#footnote-ref-3)
4. A Hyper-parameter is a parameter that does not change during the training and is not related to the training data. Even if the learning rate will change according to some fixed strategy, is still called an hyper parameter since it does not change due to the training data. [↑](#footnote-ref-4)
5. In case you don’t know what the Taylor expansion is, you can check 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 and that you should learn. [↑](#footnote-ref-5)
6. To be precise, as we will discuss next chapters, the Adam optimizer does not change the learning rate but uses a different algorithm that is very similar, but not the same, to the GD and therefore something it is said that Adam updates the learning rate dynamically. [↑](#footnote-ref-6)
7. For example, the Adam optimizer in TensorFlow 2.X uses 0.001 as the standard learning rate, unless you specify otherwise. [↑](#footnote-ref-7)
8. Tools as TensorBoard (from TensorFlow) have been built with exactly this problem in mind, to give a real time check on how the training is going. [↑](#footnote-ref-8)
9. The symbol indicates the integer part of . [↑](#footnote-ref-9)
10. I have run these tests on Google Colab, and at the time of writing that meant an Intel(R) Xeon(R) CPU @ 2.20GHz and a GPU Tesla T4 with 15 Gb Memory. [↑](#footnote-ref-10)
11. This problem is an adaptation of the one described in *Rojas, R. (2013). Neural networks: a systematic introduction. Springer Science & Business Media.* [↑](#footnote-ref-11)
12. Note that all the inputs can be written in matrix form for simplicity. [↑](#footnote-ref-12)