Understanding the Math behind Neural Networks

Written Assignment in Algorithmic Gems for AI, Games and Networks

by

Valentin Teutschbein

Tutor

Nadym Mallek

Chair of Algorithm Engineering

Hasso Plattner Institute at University of Potsdam

April 11, 2024

1 Table of Contents

Contents

2 Introduction

Artificial intelligence (AI) is gaining steady importance in the field of computer science, with deep learning methods becoming particularly popular. In this presentation, you will gain an understanding of a key component of deep learning: artificial neural networks and how to build them. Let's start by introducing the main ideas behind these mathematical concepts.

Deep learning is a subset of machine learning that focuses on solving prediction and classification problems through a hierarchical learning approach. Various methods are employed in deep learning, and one popular approach is the use of artificial neural networks, which I will refer to simply as neural networks in this presentation. These networks draw inspiration from the workings of the human brain, aiming to replicate its layout and functionality. Neural networks map input data to output using an algorithm trained on different samples. This enables them to model unknown data functions based on examples.

This presentation will specifically concentrate on feedforward neural networks, exploring how these networks utilize mathematical principles to solve various classes of problems.

2.1 Overview of a Network

In the picture below a general structure of a neural network is shown:

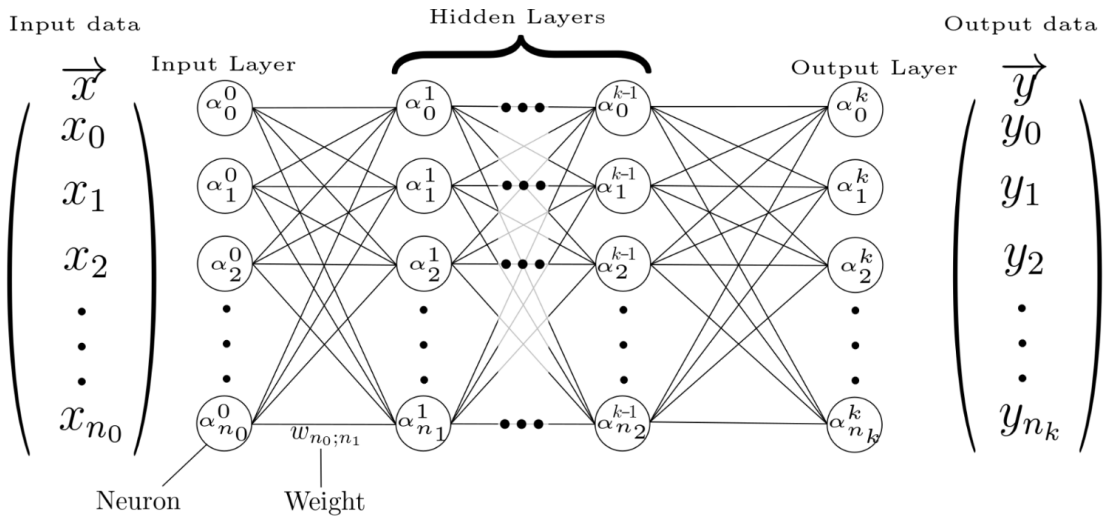


Figure 1: Concept of a neural network with k layers

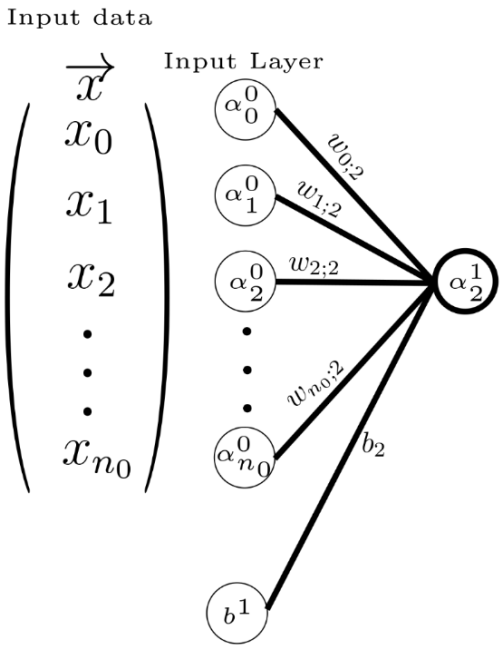
As described in the introduction, neural networks map input data $\vec{x}$ to an output $\vec{y}$. This mapping is achieved using so-called neurons, which are essentially values. In neural networks, we refer to a neuron's value as its activation, denoted by α. As shown in the figure, these activations are organized into layers. I've used superscripts to index the layers in this figure. There are three types of layers: one input layer, an arbitrary number of hidden layers, and one output layer. A neural network is represented as a layered graph, meaning that

only neighboring layer neurons are connected to each other.

In reality, there are various structures of neural networks, some of which may not be layered graphs. However, for today's discussion, we will focus on some basic network layouts. The connections between neurons are referred to as weights, denoted by w. For simplicity, this figure does not show biases b, which are additional values assigned to each neuron. Notably, biases do not depend on the neuron's input data or the values of previous layered neurons.

3 Forward Propagation

Now that we understand that neural networks model a function for classification or prediction purposes, the question arises: How do they calculate an output with given input data? This calculation is accomplished through the forward propagation algorithm. To comprehend how this algorithm works, let's focus on the previously introduced network layout and see how a single activation value is calculated. For example, let's take a closer look at the activation α_2^1 using the figure below:



We start by inserting the input vector $\vec{x}$ into our input layer. This means that the first activation of the input layer, α_0^0, is set to the first value of $\vec{x}$, α_1^0 is set to the second element of $\vec{x}$, and so on. Since our target activation, α_2^1, is in the next layer, we can directly calculate it from the input layer activations. We sum up the product of each input layer neuron times the weight that connects this input activation with our α_2^1, and finally, we add the bias connected to α_2^1. Now we input this value into an activation function σ. For now, we don't need to understand what this function does; we will discover that later on. So, in summary, we have $\alpha_2^1 = \sigma(\alpha_0^0 \cdot w_{0;2} + \alpha_1^0 \cdot w_{1;2} + ... + \alpha_{n_0}^0 \cdot w_{n_0;2} + b_2)$ as our activation value for a given input $\vec{x}$.

Task: Implement forward propagation for the given code snippet below. Use Python code or pseudocode. If you are unsure about a behavior, make assumptions.

```
class Layer:

    # in_size: input size
    # out_size: number of neurons/outputs

    def __init__(self, in_size, out_size):
        # Weights matrix
        self.weights = np.random.rand(in_size, out_size)
        # Bias array
        self.biases = np.random.rand(out_size)

    def activation(self, data):
        return σ(data)

    # returns activations of this layer
    # data: activations of previous layer
    def forward(self, data):
        # Your implementation:
```

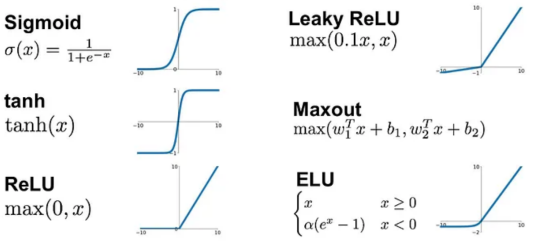
The straightforward solution for this task would be to build a nested for loop over each previous activation and the weights matrix and calculate the linear dependency, which then becomes the input of our σ activation function. However, if we imagine our input as a vector, the weights as a matrix, and our

biases as a vector, we can further simplify this procedure as the dot product $\sigma(W_i \cdot \vec{\alpha_{i-1}} + \vec{b_i})$ with W_i being the weights matrix of the i-th layer, $\vec{\alpha_{i-1}}$ being the activations vector of the previous layer, and $\vec{b_i}$ being the i-th layer's bias vector. Now we can calculate each activation of the network, enabling us to calculate the network output $\vec{y}$ using the same method layer by layer, starting with the input layer.

3.1 Activation Function

In this section, we briefly aim to understand why we need the previously mentioned activation functions, denoted as σ. We have already learned that a neural network calculates an output vector $\vec{y}$ from a given input $\vec{x}$ by summing up the product of weights and activation values. In the first layer, these activation values are just our input values. Currently, our network calculates a linear combination of input values and parameters. However, in reality, if we want to classify data or make predictions, the relationships may not be linear at all. Therefore, we need to introduce non-linearity to our network, and this is where the activation function comes into play.

As shown in the image below, there are multiple activation functions, each with different characteristics for various problem domains. The sigmoid activation is often used to introduce non-linearity to the network, and also ReLU has proven to be very useful in practice. The figure provides an overview of some activation functions:



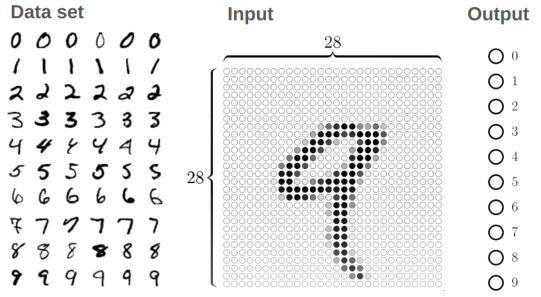
Later, when we delve into learning how networks learn, we will discuss gradients and how to compute them. Two popular problems that might occur when calculating gradients are the vanishing gradient problem and the exploding gradient problem, caused by multiplying a lot of large values or a lot of small values. The right choice of activation functions can help avoid these problems. However, it's important to note that these issues are beyond the scope of this presentation.

4 Idea of Layers

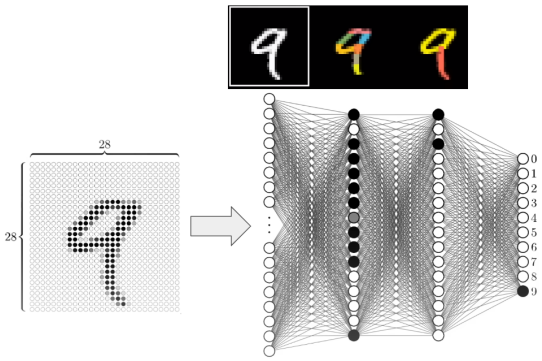
Right now, we have incorporated multiple layers of activations in our network. But why should we introduce this extra complexity, and what does each layer contribute? To grasp the concept of multiple layers, consider the following example: imagine we have a dataset of images of handwritten digits. Since an image is essentially a matrix of color values, let's simplify it by thinking of each pixel as a value between 0 and 1 (representing the brightness in grayscale). We can loop through our matrix of pixel values to get an input vector for our network.

Since this problem is a classification task, we want the network to activate the corresponding neuron of our 10 output neurons based on the input image. For instance, if we interpret our output as a classification from 0 to 9, we want the 10th neuron to be fully acti-

vated (having a high value), while all others should remain inactive if we input an image of a 9. The figure below illustrates how we can use an image from our dataset as a network input and what our output layer should look like[1].



For our human brain, this task is relatively simple. But how can we make a computer classify digits from a matrix filled with brightness values? Let's break down this problem into simpler components: first, identify certain areas of the image with low brightness, a task similar to edge detection, which was solved a long time ago. Now that we know certain areas are dark and others are bright, let's group those areas together. We can check which connected areas are activated and which ones are not. For example, if the image shows a 9, we expect certain area components to be activated, like the circle on top, while others, like the bottom-left corner, should remain inactive. By dividing our problem into multiple levels of abstraction, we can easily solve the digit recognition problem. Neural networks achieve these levels of abstraction with multiple layers. In our example, a neural network in the first layer could find small connected pixel groups that are highly activated, and in the next layer, it could find groups of those areas, and so on, until it determines which parts of the image are activated and which digit best fits those activated areas. The different abstraction levels are visualized in the picture below[2]:



This way, we can easily understand how neural networks use their different layers. However, in reality, this process can be quite different. Often, even in well-trained neural networks, the layer structure may not seem intuitive to a human, yet the network can still perform exceptionally well.

5 Making the Network Learn

Until now, we've explored how a neural network calculates an output vector from given input data using multiple layers with randomly initialized parameters. However, why should this output help us solve a prediction or classification problem?

To make the network solve our problems, we need to teach it to learn the desired output for a given input. During the training phase, we not only pass the input data to the network but also provide the expected output, allowing the network to adjust its parameters. This type of learning is called 'supervised learning.'

Now, we need a way to measure how good the network output $\vec{y}$ is for a given input vector $\vec{x}$ and expected output $\vec{y'}$. We want to quantify the difference be-

[1]Source: `https://www.youtube.com/watch?v=aircAruvnKk&ab_channel=3Blue1Brown`
[2]Source: `https://www.youtube.com/watch?v=aircAruvnKk&ab_channel=3Blue1Brown`

tween $\vec{y}$ and $\vec{y'}$. In Deep Learning, we call this measure the cost or loss. One common way to define a cost function C is by adding up the squared differences of corresponding vector entries, as shown in the figure below:

$$\begin{pmatrix} \vec{y} \\ y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n_k} \end{pmatrix} \quad \begin{pmatrix} \vec{y'} \\ y'_0 \\ y'_1 \\ y'_2 \\ y'_3 \\ \vdots \\ y'_{n_k} \end{pmatrix} \quad \begin{aligned} C(\vec{y}) = &(y_0 - y'_0)^2 + \\ &(y_1 - y'_1)^2 + \\ &(y_2 - y'_2)^2 + \\ &(y_3 - y'_3)^2 + \\ &\vdots \\ &(y_{n_k} - y'_{n_k})^2 \end{aligned}$$

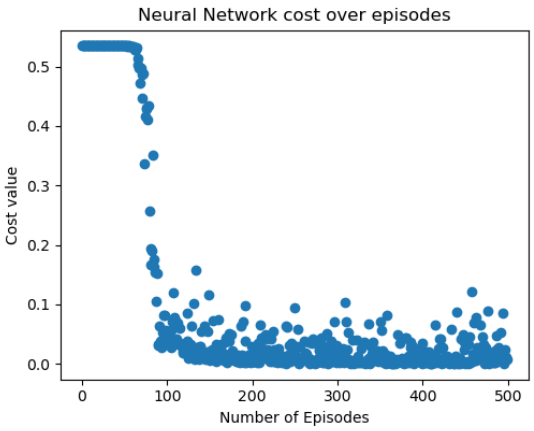
This formulation results in a high cost value C if the network output $\vec{y}$ is significantly different from the expected output $\vec{y'}$. Making the network learn is equivalent to changing the value of C.

Task: What adjustments do we need to make to C for a given sample? And how do we want to get our C?

As we know, the cost value depends on the network's output and the target output for a given sample. From forward propagation, we know that our parameters, the weights and biases, determine the network output. Since we can't influence the network output directly, we need to make adjustments to our weights and biases. Our goal is to minimize the cost.

Task: How can we calculate adjustments to our parameters to make our network better at solving its task?

A naive solution would be to randomly guess weights and biases for all input samples, check if the average cost is smaller than the previous cost, and repeat this for a fixed number of times (episodes). In my implementation of a neural network trained with randomly guessing the parameters [3], I visualized this using an array with 1000 randomly generated sample values between 0 and 1. I used the function $f : [-1, 1] \to \mathbb{R}, x \mapsto x^2$ to compute the target values. The cost values for an episode after the calculation for all samples in 500 episodes are shown in the figure below:



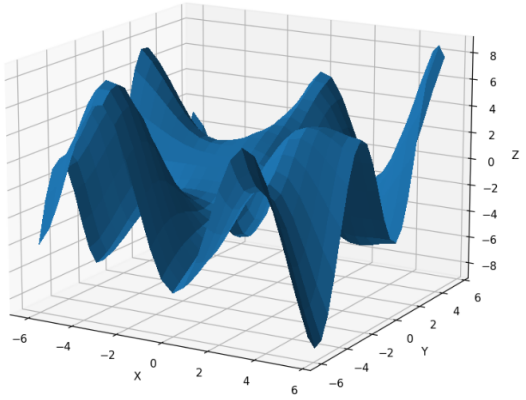
To verify the network's output, I entered the test cases $x_0 = 0.1, x_1 = 0.5, x_2 = -1$, and $x_3 = 0.01$, and I obtained the corresponding outputs $y_0 \approx 0.03, y_1 \approx 0.226, y_2 \approx 0.019$, and $y_3 \approx 0.025$, which are close to the expected results of x^2. However, intuitively, randomly guessing parameters doesn't seem to be a very precise or efficient optimization method. In the plot above, we can also observe that the cost values don't always improve, and even though the cost seems to converge towards 0 in general, there are still high cost values in some of the later episodes. In the following sections, we will cover how to actually calculate better parameters for our network.

[3] Code: `https://github.com/valteu/basic-neural-network` - run main.py

6 Cost Function

6.1 Minimizing a Function

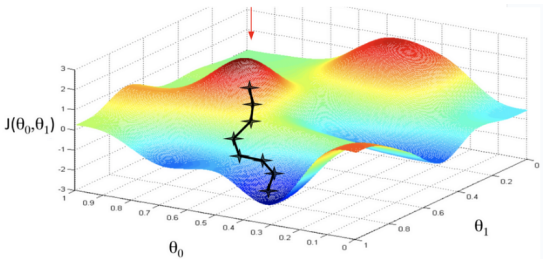
In the previous section, we learned that optimizing our cost function will train our neural network. Optimizing the cost function simply means choosing the right parameters. Since we want our cost to be minimal, we must find a minimum in our function. The figure below illustrates an example of a cost function with two parameters:



We all know from school that finding a minimum involves taking the derivative of our function, setting it equal to 0, and calculating the parameter. With the second derivative, we can confirm if our extreme point is indeed a minimum, and we are done. The problem with this approach is that although it works well for low-dimensional functions, computing the derivative set to 0 can be very costly for multidimensional functions. This is because for each parameter we add to our network, we add one dimension to C, and since each neuron has a bias as a parameter and one weight for each connection to the previous neurons, we can easily end up with millions of parameters and, therefore, millions of dimensions for our cost function. But how can we find a minimum in our multidimensional cost function? The next section will cover a very popular algorithm for solving exactly this problem.

6.2 Gradient Descent

In this section, we will learn how to find a local minimum in a multidimensional function using the gradient descent algorithm. The idea of the algorithm is that instead of computing the minimum directly, which is computationally very expensive, we calculate the 'direction' (meaning magnitude and sign), our parameters need to be changed, and then adjust our parameters accordingly to get a 'step closer' to the minimum, repeating the process. Let's give you an intuition about how this works using an anecdote: Imagine you are hiking on a mountain without paths and want to find the valley. But because it is very foggy, you have no idea of the direction to the valley. The gradient descent algorithm suggests looking at the slope you are standing on and taking a step downhill. Repeat this process until you reach a valley. With this algorithm, you are not guaranteed to find the right valley, but at least you found a local valley, as shown in the figure below:



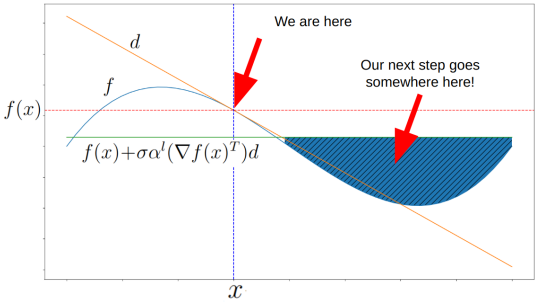
Now we need to find two important parameters: the direction and the step size. To get the direction, we can compute the gradient vector of the cost func-

tion ∇C. This vector contains all partial derivatives of the cost function with respect to the corresponding parameter. In the section on backpropagation, we will learn how to compute this vector. This vector behaves similarly to the derivative in one dimension. Since we want to find the minimum of our cost function, we need to go a step 'downhill,' meaning we need to change our parameters in the negative gradient direction.

Now we only need to find the correct step size for the parameter change. It turns out the magnitude of the gradient descent also corresponds to the 'steepness' of the current 'slope', so we could just use the magnitude of our gradient vector. Unfortunately, this could cause problems like overshooting, which we will discuss later on. For now, it is important to know that choosing a good step size matters and can be achieved using step size approximations. In this presentation, I want to introduce the **Armijo step size condition**: Let f be a continuous function $\mathbb{R}^n \to \mathbb{R}$, let $x \in \mathbb{R}^n$ be a vector, and $d = -\nabla f(x)$ be the negative gradient vector. To find an appropriate step from $f(x)$ to minimize the function, we use the Armijo step size condition, which chooses a step size $t = \alpha^l$ as the largest value with $l \in \mathbb{N}$ such that:

$$f(x + \alpha^l d) \leq f(x) + \sigma\alpha^l(\nabla f(x)^T)d$$

for two constants $\sigma \in]0,1[, \alpha \in]0,1[$. Since this condition looks pretty abstract and might be hard to understand initially, let's break it down step by step. Imagine we have our function f as described and are currently at the point x from where we want to take 'a step down.' With d, we already know the direction we want to go. So the only thing left is to find an appropriate step size t. In the following plot, you can see what the condition $f(x + \alpha^l d) \leq f(x) + \sigma\alpha^l(\nabla f(x)^T)d$ means:



So, we choose our next step to be so long that the new function value will be smaller or equal to some value $f(x)+\sigma\alpha^l(\nabla f(x)^T)d$ (green line) and in the direction d. Loosely speaking, this can be translated to 'if you want to take a large step, you should improve by a lot', because the $f(x) + \sigma\alpha^l(\nabla f(x)^T)d$ value decreases with larger step sizes.

Now we have a direction and a way of finding a good step size. Using gradient descent, we can finally start training our neural network. But before we do this, we need to address any doubts about whether gradient descent really finds a local minimum!

6.2.1 Proof of Gradient Descent Convergence

To prove the convergence of gradient descent, we at first define our algorithm for calculating it[4]:

Algorithm

Let $f \in C(\mathbb{R}^n, \mathbb{R})$

1.1 Choose start vector: $x^0 \in \mathbb{R}^n, \sigma \in]0,1[, \alpha \in]0,1[$

[4]Source of proof and algorithm: `https://www.math.uni-hamburg.de/home/oberle/skripte/optimierung/optim.pdf`, pages 31 and 32

1.2 If $||\nabla f(x^k)|| = 0$ STOP

(i) choose direction $d^k = -\nabla f(x^k)$

(ii) choose step size t_k as the largest value $t_k = \alpha^l$, $l = 0, 1, 2, \ldots$ with

$$f(x^k + \alpha^l d^k) \leq f(x^k) + \sigma\alpha^l(\nabla f(x^k)^T)d^k$$

(iii) $x^{k+1} := x^k + t_k d^k, k := k + 1$ goto (1.2)

This Algorithm really is just a formal way of describing what we just did in the previous sections, with the added parameter k for the current iteration.
Now we want to prove that this algorithm defines a sequence x^k such that for the limit x^* of f the gradient of the function $\nabla f(x^*) = 0$, which means that our algorithm finds a local minimum of f.

Proof

This proof is a contraciction proof with the Assumption, that there exists a subsequence (x^{k_j}) of (x^k) such that $x^{k_j} \to x^*$ as $j \to \infty$, but $\nabla f(x^*) \neq 0$.
Since the sequence $(f(x^k))$ monotonically decreases by construction, the continuity of f implies $f(x^k) \to f(x^*)$. Thus, according to the Algorithm,

$$\begin{aligned} f(x^k + \alpha^l d^k) &\leq f(x^k) + \sigma\alpha^l(\nabla f(x^k))^T d^k \text{ since } f(x^{k+1}) = f(x^k + \alpha^l d^k) \\ -\sigma\alpha^l(\nabla f(x^k))^T d^k &\leq f(x^k) - f(x^{k+1}) \text{ since } d^k = -\nabla f(x^k) \\ \sigma t_k \|\nabla f(x^k)\|^2 &\leq f(x^k) - f(x^{k+1}) \end{aligned}$$

So $t_{k_j} \to 0$ as $j \to \infty$
So for sufficiently large j and $k = k_j$, it can be assumed that $t_k < 1$. Till here, we basically proved, that our step size will eventually converge towards 0.

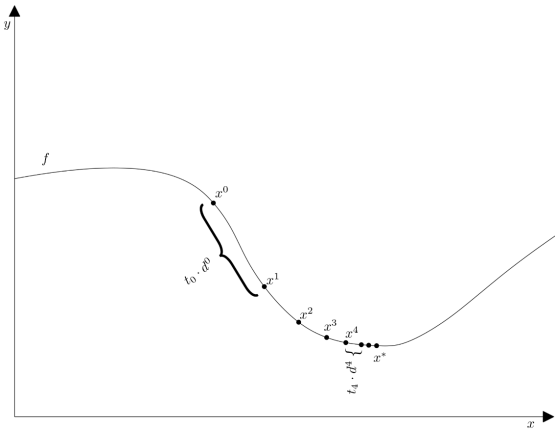


Figure 2: Thus our converging sequence can be imagined as shown in the image

Since this is a contradiction proof, we just need to break the rules of our known math applying just those rules to know that the assumption must have been false. In the next steps, we will do exactly this using a too large step size. That makes the next steps of the proof pretty unintuitive, but they allow us to prove what we want to achieve quite easily.
Using the Armijo step size control, we have

$$f(x^k + \alpha^{l_k - 1} d^k) > f(x^k) + \sigma\alpha^{l_k - 1}\nabla f(x^k)^T d^k,$$

or equivalently,

$$\frac{f(x^k + \alpha^{l_k-1}d^k) - f(x^k)}{\alpha^{l_k-1} \cdot d^k} > \sigma \nabla f(x^k)^T.$$

Now, applying the mean value theorem with an intermediate point $z^k = x^k + \Theta_k \alpha^{l_k-1} d^k$, $\Theta_k \in (0, 1)$.
We are using the mean value theorem, which states that for a differentiable function on $[a, b]$ there exists a c such that: $f'(c) = \frac{f(b)-f(a)}{b-a}$. In our case we have $a = x^k + \alpha^{l_k-1}d^k$ and $b = x^k$, so our $c = z^k$ we created must exist. Now, when applying the theorem and multiplying both sides with d^k, we get:

$$\nabla f(z^k)^T d^k > \sigma \nabla f(x^k)^T d^k.$$

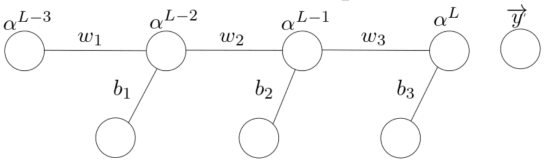
For $j \to \infty$, it follows that $\alpha^{l_k-1} = t_k/\alpha \to 0$ and $d^k \to -\nabla f(x^*) \neq 0$. Thus, $z^k \to x^*$, and also in the limit:

$$-\|\nabla f(x^*)\|^2 \geq -\sigma \|\nabla f(x^*)\|^2,$$

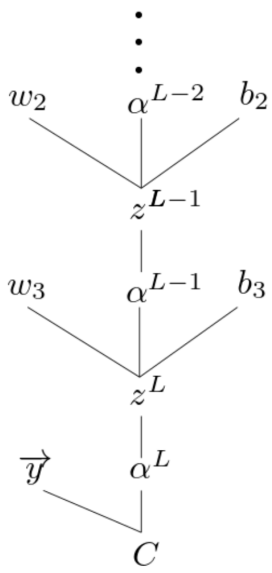
which contradicts the assumption $0 < \sigma < 1$.
Now we have successfully proved, that using our gradient descent algorithm, we can find a local minimum for each continous function $f : \mathbb{R}^n \to \mathbb{R}$ with $n \in \mathbb{N}$.

6.3 Backpropagation

To this point, we proved that with gradient descent and by choosing a good step size, we can minimize the cost function of the neural network and thus optimize the network parameters. But how can we actually compute our gradient vectors? The algorithm doing that is called backpropagation. To understand how it works, let's consider a very simple neural network like the one shown in the picture below:



The figure shows a simple Neural Network with the activations α, weights w, biases b and target network output $\vec{y'}$. It is important to know, that doing backpropagation means to first input a sample and then do forward propagation. Otherwise, we would not have a network output and no cost to minimize. As a consequence of that, all parameters and activations are just numbers right now.
As we learned earlier, the cost of the network for a given sample C is calculated as $(\alpha^L - \vec{y'})$. So how can we compute the gradient vector of the cost ∇C? As we know, this vector contains all partial derivatives of the parameters with respect to the cost function. In fact from forward propagation we know, that our α^L depends on the weighted (w_3) previous activation (α^{L-1}) plus a bias (b_3) passed into some activation function σ, and it is calculated using $\sigma(w_3\alpha^{L-1} + b_3)$. For the sake simplicity, let's call $z^L = w_3\alpha^{L-1} + b_3$ the pre-activated value. We can draw those dependencies using a tree-like structure, as you can see in the next picture:



To compute the influence the parameter w_3 has on the cost function, we need to calculate $\frac{\partial C}{\partial w_3}$. With a look at the dependency tree, we can see that this term depends on the influence that w_3 has to z^L, which itself depends on z^l's influence on α^L which finally depends on α^L's influence on C. As we know, the influence of a parameter to a function can be described as the partial derivative of this parameter to the function, so we get: $\frac{\partial C}{\partial w_3} = \frac{\partial z^L}{\partial w_3} \cdot \frac{\partial \alpha^L}{\partial z^L} \cdot \frac{\partial C}{\partial \alpha^L}$. Now let's compute those terms:
$\frac{\partial z^L}{\partial w_3} = \alpha^{L-1}$,
$\frac{\partial \alpha^L}{\partial z^L} = \sigma^(z^L)$
$\frac{\partial C}{\partial \alpha^L} = 2(\alpha^{L-1})$
Using simple rules of derivative calculations. So we get: $\frac{\partial C}{\partial w_3} = \alpha^{L-1} \cdot \sigma^(z^L) \cdot 2(\alpha^{L-1})$

Task:
Now how to calculate $\frac{\partial C}{\partial b_3}$

Computing the influence b_3 has to the cost function $\frac{\partial C}{\partial b_3}$ is very similar to $\frac{\partial C}{\partial w_3}$, you just have to calculate $\frac{\partial z^L}{\partial b_3}$ instead of $\frac{\partial z^L}{\partial w_3}$, and $\frac{\partial z^L}{\partial b_3} = 1$.
So we get: $\frac{\partial C}{\partial b_3} = 1 \cdot \sigma(z^L) \cdot 2(\alpha^{L-1})$. The rule of 'splitting' the partial derivatives we used is called the chain rule, and using it, we can now easily compute the influences of each parameter of the last layer. Remember: This parameter is just one part of the gradient vector. So in reality, we need to calculate those influences for all parameters. This can easily be done recursively, starting at the output layer of the network until we reach the input layer. That is why the algorithm is called backpropagation: It calculates the gradient vector from the back to the front of the network.
So all in all in this section, we learned how to calculate the gradient vector of our network for a given training example using the backpropagation algorithm.

7 Training the Network

After calculating the gradient vectors of our network for one sample, we now want to actually train our network. Before we do this, I want to clarify the meaning of samples, batches, and episodes:

The neural network uses training data, which we call samples. In our previous example of recognizing handwritten digits, a sample would be a single image of a handwritten digit. A batch, on the other hand, is a (most of the time, randomly chosen) subset of the samples used to train the network, not all samples at the same time. An episode, on the other hand, includes the training of the network with all samples. This is usually done multiple times, so we have multiple episodes. In this presentation, we just covered the gradient descent optimization algorithm, which does not use batches. Still, there are many popular algorithms that utilize batches, so it is important to know what batches are. Now we will finally learn how to train a neural network.

7.1 Learning Algorithm

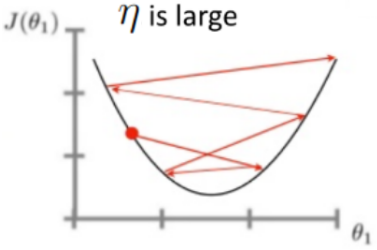
We now know how to calculate the network output for a given sample and how to compute the desired changes to the parameters for this sample. Now, the basic idea of training the network is to calculate this gradient vector for each sample we have and then update all the parameters by subtracting the average of the corresponding gradient values. Essentially, we find the average of all desired changes to the parameters for all samples. After that is done, we repeat the procedure until our network is trained well enough. The number of episodes can be hard-coded or related to the cost value. So, the learning algorithm can generally be described as follows:

```
∇ C_all = 0⃗
FOR EACH EPISODE:
    FOR EACH S ∈ SAMPLES:
        NETWORK.FORWARD(S)
        ∇ C_S = NETWORK.BACKWARD()
        ∇ C_all += C_S
    NETWORK.UPDATE_PARAMETERS(η, ∇ C_all)
```

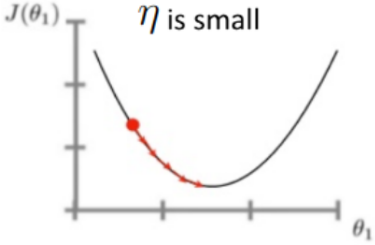
Thereby, the NETWORK.UPDATE_PARAMETERS function subtracts from each parameter its corresponding gradient vector value divided by the amount of samples (so we have the average desired change direction) times η. But why do we need the parameter η? This will be covered in the next section.

7.2 Learning Rate

As we know from the gradient descent algorithm, updating our parameters just using the gradient vector can lead our optimization to overshoot, as illustrated in the figure below[5].



That is why we used a step size determined by the Armijo condition to prevent our optimization from overshooting. However, besides overshooting, there is also the opposite problem: If η is too small, our optimization will be very slow because we will always improve only by a small amount, as shown below[6]:



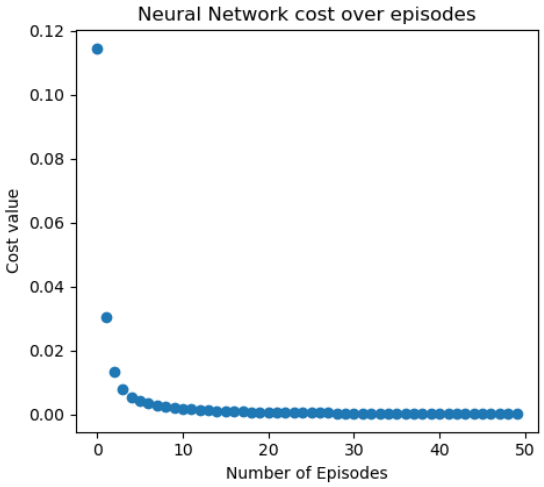
There also exist conditions preventing our step size from being too small, which will not be covered in this presentation. In neural networks, we call this step size the learning rate since it determines how much our network should learn each episode. You should just keep in mind that choosing a good learning rate is key to successfully training a neural network.

[5]Source: https://www.kdnuggets.com/2020/05/5-concepts-gradient-descent-cost-function.html

[6]Source: Same as above

7.3 Demonstration of training a neural network

Now, with the knowledge we gained about optimizing a neural network using gradient descent, it is time to implement and test it[7]. Within only 50 episodes, our cost decreased significantly to just ≈ 0.0003, as shown on the plot below:



The test samples $x_0 = 0.1, x_1 = 0.5, x_2 = -1$, and $x_3 = 0.01$ produced pretty good results: $y_0 \approx 0.03, y_1 \approx 0.23, y_2 \approx 0.03$, and $y_3 \approx 0.03$. These results are quite similar to the results we got when randomizing our weights and biases. However, we achieved this in just 50 episodes. If we were to determine our learning rate using the Armijo condition, these outputs would probably look much better after more episodes. So yes, the neural network works! But what did we do? We just implemented $f(x) = x^2$ in a very complicated way with an error and a long runtime.

Task:
So why did we do all of this?

The answer lies in the flexibility of our network. If we don't know how our samples were generated or measured, we can still construct a function that guesses them: Our neural network. We can also use our network for classification problems. So if we already know how to calculate an output value, we should, for sure, not use our neural network to do this. But if we don't, we can guess this function pretty well depending on the available data using our network.

7.4 Further Thoughts

In the gradient descent algorithm and its convergence proof, I mentioned and proved that this algorithm guarantees us to find a local minimum.

Task:
But why would finding a local minimum be enough? Couldn't this have an arbitrarily high cost?

To answer this question, we should remember the complexity of the cost function. Even though in this presentation we graphed this function with a 1 or 2 dimensional input mapping to one different dimension, in reality, our network has one dimension per parameter. So we can easily get a cost function mapping a $1,000,000,000$-th or higher-dimensional input to our cost value. But what does that change? Well, the intuition this gives is that since a minimum is just a place where every input dimension cost function has a minimum, adding more dimensions makes it easier to have at least one dimension without a minimum at this point, which allows us to optimize alongside this dimension. In reality, we are still not guaranteed to find a global minimum; it get's just more and more likely to not 'get stuck' in a local one.
If you remember the idea of batch optimizing, this can also be a way of 'es-

[7] Code: `https://github.com/valteu/basic-neural-network` - comment the line 'n.train_random' and uncomment 'n.train'; run main.py

caping' a local minimum since, with optimizing only for some random samples each time, we get randomness and don't always follow the straightest path 'downhill' of our cost function.
Besides the optimization of neural networks, there are a lot of design choices to do when building a neural network, for example, the number of layers.

Task:
Now the question arises whether more layers are generally mathematically better without considering the extra training time it would cause?

Solving this question formally is quite hard, so I will give you just an intuition: Imagine a small neural network that is trained pretty well. The probability that such a well-trained network is already a subset of a large neural network obviously increases with a larger network size. So in this case, we just need to deactivate all the other neurons, and thus within just a few episodes, we can solve the given problem easily. So more layers will mathematically be pretty probable to solve our problem within fewer episodes. In reality, of course, the time to train each episode and the storage space of a neural network increase with the network size, so there exists a trade-off.

8 Conclusion

In this comprehensive exploration of neural networks, we delved into the fundamental concepts that form the backbone of these powerful machine learning models. From the basic building blocks like neurons and layers to the intricate details of activation functions, backpropagation, and gradient descent, we have navigated through the intricate terrain of neural network architecture and training.

Understanding the importance of non-linearity introduced by activation functions, the role of multiple layers in abstraction, and the nuances of training through the supervised learning paradigm provided insights into the inner workings of neural networks. We explored the significance of the cost function, its minimization through gradient descent, and the crucial role played by the learning rate in determining the convergence of the optimization process.

As we ventured into the practical implementation and training of a neural network, we witnessed the interplay of concepts like batches, episodes, and the iterative adjustment of parameters to optimize for a given task. The demonstration showcased the network's ability to learn and adapt, highlighting the flexibility that makes neural networks suitable for a wide array of problems.

In the final reflections, we pondered the implications of finding local minima in high-dimensional spaces and considered the trade-offs associated with the design choices in neural network architecture. The journey through this intricate landscape aimed to equip you with a foundational understanding of neural networks, empowering you to explore further and harness the capabilities of this transformative field in machine learning.