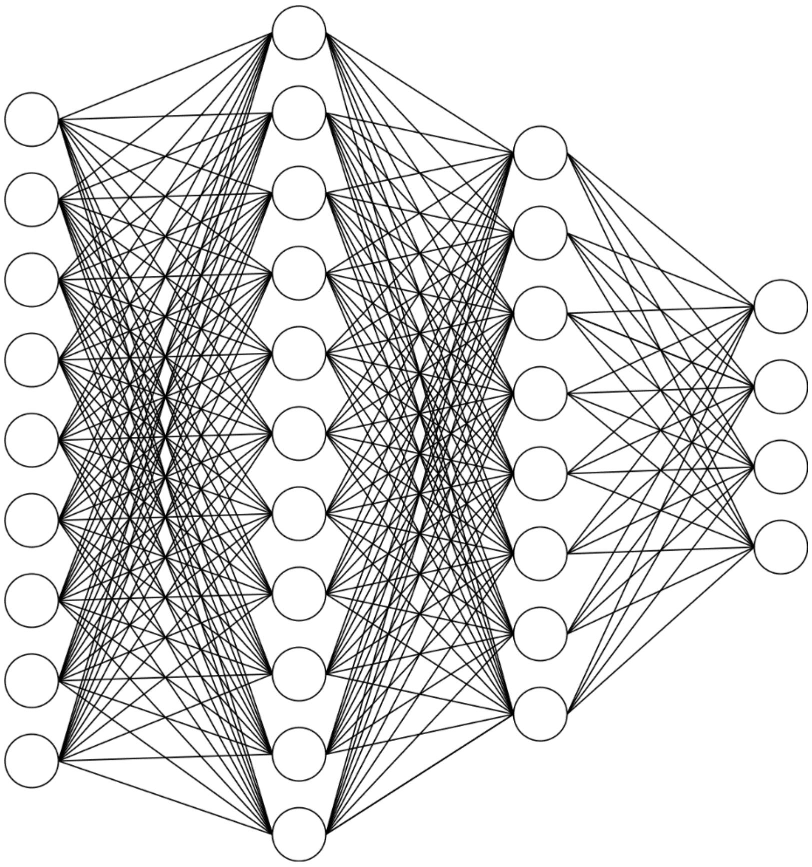
ZÜRICH, 2020



# NEURAL NETWORKS

A THING THAT SOMEHOW WORKS

FILIP SOKIRKO, 6E

STEFAN GRIEDER, ALEXANDER IRION

Kantonsschule Hohe Promenade

## TABLE OF CONTENTS

# FOREWORD

I thought neural networks were pretty cool, so I decided to make my project about them.

The original intent was to simply understand and explain how they work. At some point, I thought it would also be a good idea to make one. Now, it’s not exactly difficult to build a neural network with modern libraries, but since I wanted to get a good understanding of their inner workings, I made one from scratch in C++. This turned into a more difficult, but also very interesting task.

In the following chapters, you can expect an explanation of how they are built and how they learn, but none of the actual code I made. Here’s a link instead: https://github.com/fiso64/matura-project-nn

# INTRODUCTION

That’s a 1, you think.

But how do you know?
It’s not the only way those lines we call “ones” can look.
In fact, ones have been observed to come in all shapes and sizes; small, big, fat, thin, straight, bent.
Still, we recognize all of them as the same thing in a heartbeat, in a process that seems so deceptively simple. Try to emulate this behavior in a program, and you quickly realize how difficult of a task it really is: How do you tell a computer that a 1 can look like this, but sometimes, “a bit different”?

In a world where every case is a special one, a simple algorithm won’t do.

To recognize numbers, each one of us uses a brain: A powerful supercomputer with millions of trainable parameters, all making those daily tasks quick and seamless. What exactly goes on in this machine is still not exactly understood, but that hasn’t stopped us from finding inspiration in it.
Indeed, the artificial neurons we created, and by extension the artificial neural networks, were (originally) inspired by the ones found in our brains. They are an intuitive solution to the problem at hand: Instead of developing a complex algorithm to handle every special case, we let a neural network learn those on its own.
And they do work pretty well! While the hype surrounding AI can be tiring at this point, the results do speak themselves: Modern programming libraries make it extremely easy to make a computer recognize numbers (such as this 1) with high precision and speed[1]. In fact, the process of building such a program has become so simple that it is now referred to as the ‘Hello World’ of neural networks. And yet, we are doing something that would be unthinkable not too long ago; we are literally teaching a computer to recognize numbers.

You’ve most likely heard of self-driving cars. While digit recognition is something any 5-year-old could do, driving a car is a task even adults fail to do properly more than 1.3 million times a year[2]. Now, neural networks aren’t quite there yet to fully replace humans on the roads, but they’re not that far off either. They are likely to soon replace us in this *reasonably complex* task[3].
Other nets are being trained to write code based on a simple description of what you want to do[4]. But why stop there? We already have networks designed to compose music, paint, and write[5]–[7].
I mean, they’re not really good at it, but even artists might not be entirely safe in the long run.

And did I mention that an AI recently made a huge leap in the decade-old protein folding problem[8]?
I suppose it’s only downhill for us from now on.

Whether we cause an AI takeover or simply some abnormally high unemployment rates, neural networks sound exciting either way. It’s clear – the potential is immense.
But how do they work?

# 2

# THE ARTIFICIAL NEURON

Let’s start with the elementary unit of a neural network: The artificial neuron.

The exact mathematical definition of the artificial neuron might seem like it’s coming out of nowhere at first, but it’s worth keeping in mind that they are inspired by biological neurons*.
Let’s first look at a simplification of how our neurons work in order to get some intuition.

In your brain, you will find millions of excitable cells called neurons.
Neurons “communicate” with each other with signals travelling along connective pathways. At the end of each pathway, there’s a “valve”, the synapse, whose strength will regulate how much of the signal gets to pass to the connected neuron.
Now, if all of the signals that reach the neuron are collectively strong enough, the neuron enters an exited state. It “fires”, giving off a signal, which will travel to other neurons, possibly exciting those too.

The neuron is rather simple on its own, but there are just so many of them that we can get some complex behavior with the huge chain reactions they create.

We usually don’t get good at a task immediately. To improve, our brains constantly adjust themselves, they turn millions of knobs to get the desired behavior.
Of course, we can’t exactly change what we perceive. The adjustable parameters in our case are the synapses, which can be strengthened or weakened depending on the external response[9] [10].

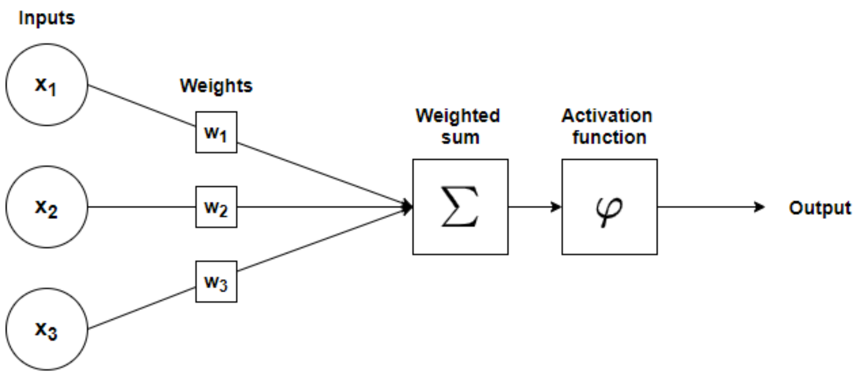
In the context of artificial neural networks, the strength of the synapse is known as the weight. The weight can be thought of as a representation of a neuron’s importance to another neuron: If the weight is very low (that is, if the synapse is very weak), the two neurons will barely affect each other, as a signal from one neuron would hardly reach the other.
The electrical signals biological neurons send simply become one number (0 when idle, 1 when excited), which, when multiplied by the weight, becomes the possibly weaker signal that finally arrives after travelling through the synapse.

Now, I explained that if all of those arriving signals are “collectively strong enough”, the neuron would fire/output 1. This process can be interpreted as two separate functions:

The first is combining those inputs – taking the sum. Each summand is a product of the input and its respective weight, so this sum is commonly referred to as the weighted sum.
Secondly, to determine if the sum is large enough for the neuron to fire, the weighted sum is passed through the so-called activation function.

So, depending on the size of the weighted sum, the activation function will output either a 0, or a 1.

* While initially conceived as a model of the biological neuron, modern artificial neurons are rather different. Programmers and engineers, having apparently realized that maths is superior to nature, seemingly just gave up trying to emulate the brain.



This is a diagram depicting an artificial neuron with 3 inputs.

The output, the weights $w_1$ through $w_3$, as well as the inputs $x_1$ through $x_3$ are all just numbers. Importantly, those numbers can be negative too, even though it wouldn't make much sense from the biological point of view. For a single neuron such as this one, the inputs are usually the real-world data we want it to work with. As an example, the inputs could be the values of each pixel in an image.

As explained above, each input is now multiplied by its weight (representing the signal's strength after the synapse), and summed with the other products in the weighted sum, which is then passed through the activation function, giving us the output of the neuron:

$$\text{output} = \varphi(x_1 w_1 + x_2 w_2 + x_3 w_3)$$

For a neuron with inputs $x_1$ through $x_n$, their respective weights $w_1$ through $w_n$, and an activation function $\varphi$, the output is therefore given by:

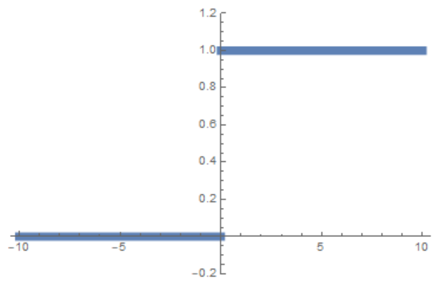
$$\text{output} = \varphi\left(\sum_{k=1}^{n} x_k w_k\right)$$

So, the neuron is basically a composition of functions taking two types of variables: The inputs x, and the weights w[*]. It's helpful to think of the weights as part of the neuron. Since they're the adjustable parameters (at least during training), the neuron can be thought of as a function with a "memory" in form of weights, which will affect its output.

Speaking of which, the output is obtained by passing the weighted sum through the activation function. But what does this function $\varphi$ do, exactly?

First off, the activation function can be seen as an adjustment to a common scale. Whether the weighted sum is tiny or large, the activation function will always just output either a 1 or a 0. But more importantly, the activation function represents a threshold: If the weighted sum is large enough, the neuron will activate/fire/output 1.

[*] However, there will never be a situation where both the weights and inputs are variables at the same time. Whenever we calculate the output of the neuron, the weights are constant. During training, the inputs are constant.



With this definition, the activation function can be simply shown as a binary step.

As you can see, it's both an adjustment to a common scale, as well as a threshold. Whether the weighted sum is -1 or -100, the output will be 0 regardless. But if the sum is above the threshold (here: 0), the neuron will suddenly output 1.

The perceptron*, one of the first types of artificial neurons, uses the binary step activation function (a version of it). The perceptron was invented in 1958 for the purpose of image recognition[11]. Just like the neuron on the previous page, it passes the weighted sum through the activation function. We can now give the full** equation for the perceptron's output:

$$\text{output} = \begin{cases} 0, & \left(\sum_{k=1}^{m} x_k w_k\right) \leq 0 \\ 1, & \left(\sum_{k=1}^{m} x_k w_k\right) > 0 \end{cases}$$

We can simplify the equation by thinking of the inputs and weights not as individual numbers, but rather as 2 vectors. In this case, the weighted sum simply becomes the dot product of the input and weight vectors, and the output is given by

$$\text{output} = \begin{cases} 0, & x \cdot w \leq 0 \\ 1, & x \cdot w > 0 \end{cases}$$

This equation conveniently describes the output of a perceptron. From a set of inputs, it will compute its prediction to a yes or no question.

This prediction, by the way, is likely to be false with random weights. It's clear, a perceptron won't just start correctly classifying images out of the box. To make it do things, its weights need to be adjusted so as to make the outputs as accurate as possible; it needs to be trained.

However, because the perceptron is mostly outdated, there's no use explaining how exactly the weights are adjusted during training (or learning).

* "Perceptron" usually refers not only to this neuron function, but also to the specific training algorithm.

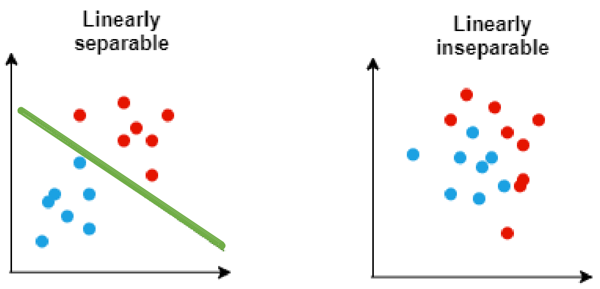
** The full equation has an adjustable "bias" term, which gets added to the weighted sum. The bias is the position of the threshold on the x axis. While important to ensure that the perceptron works properly, I left it out. (Also, larger neural networks are usually able to "simulate" the bias term anyways.)

What makes it outdated?

Well, have you noticed that the perceptron is rather... simple?
Passing the weighted sum through the binary step is pretty much all that happens when a prediction is computed. It's perhaps no big surprise that it is a somewhat limited classifier.

More specifically, the perceptron is a *linear* classifier, meaning that its output is only based on the result of the multiplication of each input followed by summation (a linear combination), which is exactly what the weighted sum is.

This, in turn, means that it can only classify data which is linearly separable[12]. That is, if you imagine the data as a set of points on a plane, it is linearly separable only if there exists a line which separates all points belonging to different classes:



Basically, a perceptron can only classify data if we can draw a straight line to tell true from false. One can actually visualize the perceptron's learning process as moving and rotating this line in order to best separate the classes, in order to best "fit" the data.

If we want to classify more than just two classes, we simply connect multiple perceptrons to the inputs, each one with its own weights. This is analogous to adding more lines to divide the data, which would help us classify 3 or more types of "things". But no matter how many perceptrons we add, we will never be able to properly classify linearly inseparable data.

Soon enough, linear separability proved itself to be a big limitation, since much of the data we get from the real world has a more complex pattern*. The excitement surrounding the perceptron died down quickly, as the algorithm was quickly shown to be a poor classifier.
What's more, a number of false promises all lead to disappointment, and the "failure" of the perceptron even caused the research in this field to stagnate for a while[11].

* Although it is sometimes possible to transform linearly inseparable data in such a way that it becomes separable. Thus, the data needs to be linearly separable, able.

# 3

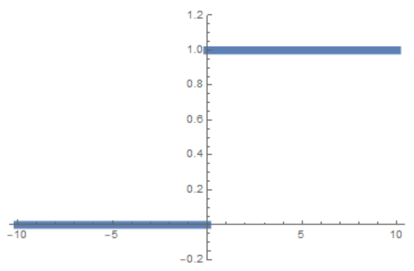
# THE NEURAL NETWORK

So, what if our data is linearly inseparable? What if you need a curvy line to properly classify the inputs, as is the case with most of the real-world data?

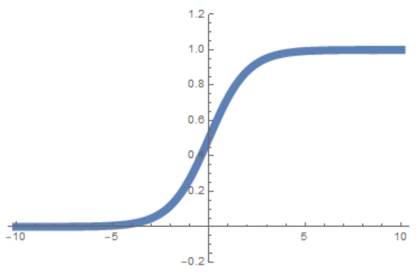
Behold, the solution: Stacking the neurons.

Of course, it's not *that* simple. Some adjustments have to first be made in order for the neuron to work in a network. One of those adjustments is replacing the old, simple, and plain boring binary step with a continuous activation function. Our new function should still have similar properties to the binary step (it should still resemble a threshold).
One example is the sigmoid function. Here it is, side-by-side with the binary step:

**Binary step**



**Sigmoid function: f(x) =** $\frac{1}{1+e^{-x}}$



This is just one of many of the more modern activation functions. As you can see, it looks like a smoothed-out version of the binary step, so it would probably produce similar results if we plugged it into our neuron. However, aside from looking a bit nicer, the new activation function now returns anything from 0 to 1, as opposed to the binary step, which could only return 2 different values.

So why replace our perfectly-fine binary step?

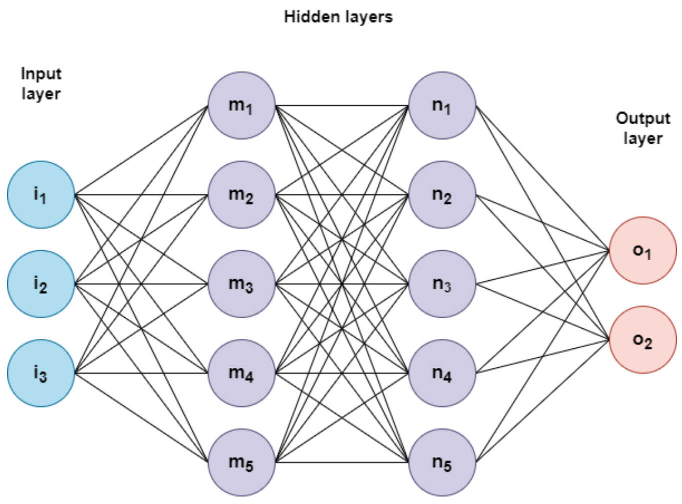
A good reason for the activation function to be continuous is because it allows us to tell how wrong our predictions really are: If a perceptron outputs a 1 where it should be a 0, we have no way of telling how far the weighted sum is from the threshold; no matter if the sum is 0.1 or 100, the binary step will output 1. But then, if you're very close to the threshold and change one weight only slightly, you suddenly see a big jump in the output. This makes it much more difficult to properly adjust the weights. It would be much better to instead have a small change in the weights result in a small change in the output.
In other words, it would be better if the activation function was continuous. Interestingly, this is one of the big differences between today's artificial neurons and biological ones: The ones in our brains really do seem to work like a binary step, rather than producing outputs on a continuous curve[10].

Actually, the fact that it's continuous now is what will finally allow us to train a neural network efficiently. Much of the reason as to why that is will be explained later, but first, let's look at how a neural network is built.

Neural networks consist of three types of layers: The input layer with the inputs, the output layer with the output neurons, and any number of hidden layers between those two.
So by this definition, the perceptron would technically have two layers: The inputs, and the outputs. However, as the input layer doesn't actually consist of neurons (just numbers), a perceptron counts as a single-layer classifier.
Generally, if a network has more than one hidden layer, it is referred to as a deep neural network (DNN)[13].

Each neuron in each layer produces an output, which is then used as an input in the next layer. So if a neuron is a composition of two functions, a neural network is a composition of compositions of functions.

The following diagram depicts a deep neural network with 2 hidden layers m and n:



Each line here signifies that the output of the neuron is used as an input for the next neuron, and stands for a specific weight. In such a simple network, we already have 50 different weights to adjust, though typical networks can consist of millions.
Still, the basic output equation for each neuron remains the same, that is, it's still about passing the weighted sum through an activation function.

Take the neuron $n_1$, for instance. Its output is given by the same formula:

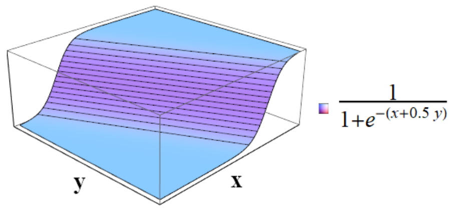
$$n1_{out} = \varphi(m_{out} \cdot w_{n1})$$

Where $\varphi$ is the activation function, $w_{n1}$ is the weight vector of $n_1$, and $m_{out}$ is the vector of outputs from the previous layer m. The outputs in $m_{out}$ are given by a similar equation too. And so if we wanted to write out the entire equation for the outputs of the last neurons $o_1$ and $o_2$ in terms of the first 3 inputs, the result would be a much longer composition of functions consisting of several sums, multiplications, and activation functions. Passing some inputs through this function and obtaining the outputs is known as forward propagation.

So, what makes a neural network so much better than a single-layer classifier (like a perceptron)?
Why can a network solve linearly inseparable problems, unlike linear classifiers?

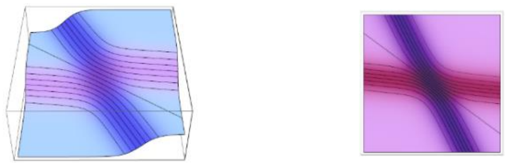
The formal proof is known as the Universal Approximation Theorem[14]–[16], which goes beyond the scope of this script. UAT states that, for any continuous function, there exists a network which can approximate it to an arbitrary precision.
I will show a couple of visual examples in order to provide some intuition.

First, consider a simple case: A single neuron with two inputs and sigmoid as activation function. The neuron has two weights which can be adjusted in order to approximate a function. This is a graph of the neuron's outputs depending on the inputs x and y (with weights 1 and 0.5):



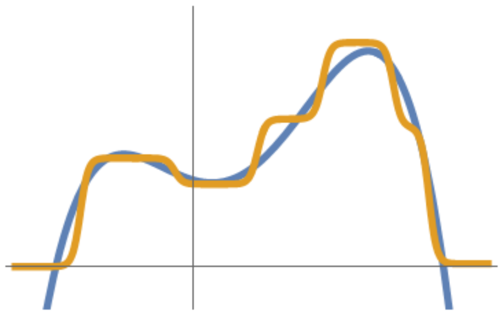
This plot simply looks like the sigmoid curve. We can squeeze, stretch, or move it however we like by changing the parameters, but the fundamental sigmoid shape will remain. And so will this linear "decision boundary": Viewed from above, it still looks like a straight line. Therefore, despite us using a non-linear activation function, the neuron still classifies linearly (the decision whether it should "fire" is based on a line), and is incapable of classifying linearly inseparable data.

But now, with a neural network, we obtain weighted sums of those outputs. The weighted sum of a neuron is now a linear combination of the previous weighted sums, each passed through the sigmoid function. A linear combination of sigmoid functions allows us to build something like this:



The "decision boundary" is no longer linear.
Of course, the fact that we can model some non-linear function would not necessarily allow us to approximate any.

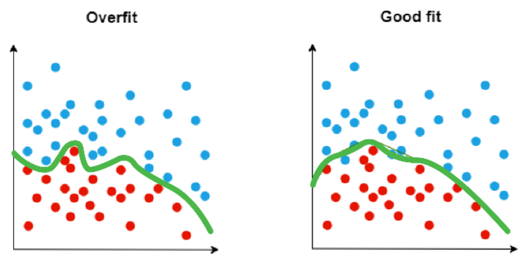
You might know that functions can be approximated with a sufficiently large weighted sum of sine waves (Fourier series). Sines form a basis, which can be used in a linear combination to construct the desired function. However, sine is not the only function that works as a basis, the activation functions we use do too. And in a neural network, this is pretty much exactly what happens: With a big combination of activation functions, we are building the function we're trying to approximate.
For example, the following is a rough approximation with a weighted sum of 7 sigmoid functions. You should be able to see each separate sigmoidal shape:



This is all well and good. We now know that we could, theoretically, build a network that classifies almost anything. But how exactly?

The Universal Approximation Theorem does not help here. While the structure of the input and output layers can, as you will later see, be easily inferred from the task at hand, it's a different story with the hidden layers. While there are a couple of general ideas that work well when it comes to finding the right number of hidden layers and their sizes, determining the specifics often involves some trial and error[17].

Add few hidden neurons, and so our network might have trouble learning with the limited number of weights.
Add too many, and it might lead to the network becoming a memory bank, basically causing it to memorize each element, rather than to learn to distinguish them.
The intent is, of course, to make it classify new, previously unseen data. With too many weights, the network is likely to even learn the noise and the outliers in our data, despite it not being representative of the general pattern.
In statistics, this act of overly adjusting to data is called "overfitting"[18].
If we again think of the learning process as the network trying to find a line which separates the classes as well as possible, one can think of an overfit as the line being "too good":



The issue here is that, although the first model perfectly classifies the given data, it will probably fail on a bigger dataset with different noise. The second model has worse results for this specific case, but it's possible that this line represents a general pattern, and so it might fare way better on average.
Overfitting is very likely to happen if the network has too many weights. The opposite, underfitting, often happens if the network doesn't have enough weights, where the model doesn't represent the pattern accurately enough.

As you can see, there is a lot to consider when designing the hidden layers. It's worth noting that while one hidden layer is theoretically enough to approximate pretty much anything (by the Universal Approximation Theorem), deep nets often perform much better. The reason is, as is the case with quite a couple of things concerning neural networks, not exactly understood*.
A commonly held belief is that deep nets are able to look at the more abstract features of the data with the help of deeper layers, rather than just at the immediate inputs[19].

But for now, let's assume we've found some sweet spot. How do we train this network?

* And why do networks work this well at all? Believe it or not, there is no shortage of universal approximators, neural networks are just one kind of them. What the Universal Approximation Theorem doesn't tell us is why neural nets are able to converge this quickly. Again, this is still a bit of a mystery.

# THE LEARNING ALGORITHM

If we want to train something, be it a perceptron or a modern neural network, we are going to need data. Our dataset, consisting of various images, texts, sounds, or whatever you want to classify, should contain a corresponding right answer (label) for each sample*.

To then measure the classifier's performance on the dataset, we define the so-called loss (also known as cost, or error) function. The loss function will compare the outputs from our classifier with the label and tell us how large the difference is. If the loss is large, that means the classifier is not doing a very good job, and we need to change the weights in order to decrease the loss as much as possible.
So, the act of training a classifier can be seen as trying to minimize the loss function.

We should define one. A simple loss function is the mean squared error (MSE). If a network produces outputs $y_1$ through $y_n$ for a particular train sample with a label vector $\hat{y}$, MSE is given by the average of the squared differences between the outputs and label elements:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Obviously, differences should all be positive, but why not take the absolute value (the mean *absolute* error) rather than squaring them?
With MSE, larger errors get punished more due to the square**. It's essentially like saying "this is really, REALLY bad", causing our network to frantically change its weights in shame.

Our job is now to minimize this loss function. When I first saw this, I was tempted to just do it the way we did it at school – find the derivative of the loss with respect to the weights and solve for 0. With the extrema, I would just check which one has the smallest value in order to find the absolute minimum, which would yield the weights for which the loss is minimal.

Sadly, this doesn't really work here. Because we're using more than one weight, the analytical solution would involve finding the partial derivatives and solving the system of equations for 0. However, a neural network has *a lot* of weights. The loss function therefore has *a lot* of variables, and so we would have to minimize a *very* multi-dimensional function. It quickly became apparent that the analytical approach of solving a huge system of equations is rather slow. And what's more, since the equations we would obtain are non-linear (due to the non-linearity of our classifier) there's a good chance we wouldn't be able to solve them at all!

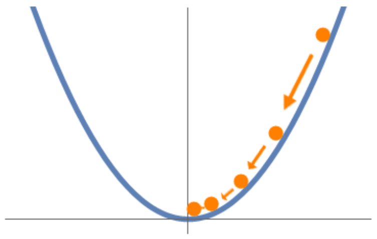
Though not without their own pitfalls, numerical approaches are often the only option we have[22].
One such approach, the one you'll find in neural networks, is called gradient descent.

---

* This is not a requirement. Neural networks can sometimes be trained without labels (unsupervised learning).
** MSE is simple and great for regression, but not used for classification problems[20] [21]. Also, as a consequence of punishing larger errors more, MSE often weights the outliers way too heavily (which leads to a bad fit).

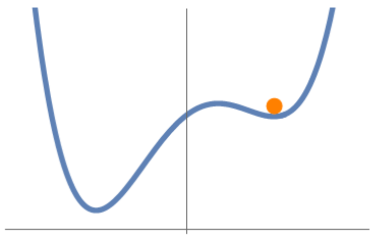
How does it work?

It's best explained if you picture a ball on a slope. Naturally, it will start rolling down and its altitude will decrease. Once the slope is 0 (or very close to it), the ball will stop, and so it has "found" a minimum. This is basically how gradient descent works: Instead of trying to find a solution right away, we look at the problem locally, and optimize the answer step-by-step.



To put it into more mathematical terms, gradient descent looks at the derivative of the function with respect to a weight. If the derivative at the current position is positive, we need to decrease the weight/x coordinate, as the function increases with increasing x. For a negative derivative, x needs to be increased so as to decrease loss. The step direction is therefore determined by the negative of the derivative at the current position.

To avoid overshooting the minimum, the size of each step depends on the size of the derivative; the smaller it is, the smaller the step. So, if the derivative is very small, the step becomes really small too, and the ball "stops rolling". We have found a minimum.



Note that it might not be the absolute minimum. We could land in a pit, a local minimum.

Here, the derivative is tiny, so the ball won't bulge. Even if it did, it wouldn't go far before returning back to the pit. Moreover, we can't even check if we're in a local minimum since an analytical solution is usually impossible.

There are a couple of ways to avoid this. For instance, we can start the algorithm at many different locations and then compare where it ends up (throwing multiple balls). Another way, which is commonly used in neural networks, is to modify the algorithm by adding a "momentum" to the ball, making it even closer to a physics simulation. With momentum, the step size also depends on the previous step size multiplied by some constant. So if the ball reaches this local minimum, and if it has a large enough momentum, it will just go over the curb and land in the global minimum.
However, recent findings suggest that local minima aren't such a big problem after all, in practice[23]–[25]. It turns out that the ones we land in mostly produce a similar loss to that of the global minimum*. Regardless, methods like momentum are still widely used to avoid landing in undesired places such as bad local minima and saddle points**.

Let's consider a ball rolling down a hill in 3 dimensions instead of just 2. At any point, the ball's position is given by two parameters (weights), the third coordinate is given by the loss. We want to know where to move in order to decrease loss as fast as possible; we need a 2-dimensional vector. Again, the first step to finding it is to calculate the (now partial) derivatives. A vector of those two derivatives points into the direction of steepest ascent, and is known as the gradient vector.

* And if they don't, you're probably overfitting with the global minimum anyways[25].
** Which, by the way, are more common than local minima in higher dimensions[26]. They're a bigger problem too, as in case of a saddle point we could obtain a very large loss, far from that of local minima.

Since we're usually working with many more weights than just 2, let's define the gradient vector for a loss function C with respect to weights $w_1$ through $w_n$. The gradient vector is usually denoted with ∇:

$$\nabla C = \left(\frac{\partial C}{\partial w_1}, \frac{\partial C}{\partial w_2}, \ldots, \frac{\partial C}{\partial w_n}\right)$$

And again, this simply represents the vector of steepest ascent for n dimensions[*].

Now let's actually update the weights with the help of the gradient vector. For the weight vector w of an arbitrary neuron, the updated weight vector w' is obtained by doing a step into the negative direction of the gradient vector (we want steepest descent of loss, not ascent):

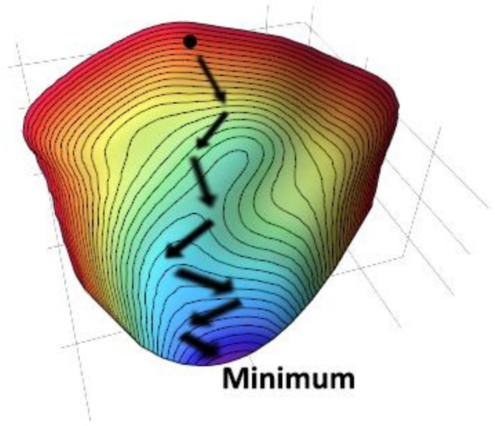
$$w' = w - \eta \nabla C$$

Where $\eta$ is a small number known as learning rate, which also determines the size of the step.
It's important to find a good value for the rate: If it's too big, the weights will change too quickly, and gradient descent will always overshoot the minimum. If it's too small, it will take a very long time to actually descend to the minimum.

And that's pretty much how we update the weights with gradient descent!

However, the equation above only describes the weight update of one single neuron, with the derivatives calculated from one single train sample. Since we want to decrease the "full" loss across the entire dataset, we would need to change the weights by the average gradient vector across all train samples.
But finding the derivatives for all samples, of which there can be hundreds of thousands, is a computationally expensive process. Instead, we divide the data into smaller subsets called mini-batches, for which the average gradient is computed.

This method is known as stochastic gradient descent, and results in a more chaotic, but also much faster descent to the minimum.



[*] Technically, this row vector needs to be transposed in order to obtain the column gradient vector.

# BACKPROPAGATION

The loss function is minimized by looking at the derivatives at each step.

But up until now, we haven't seen how one would calculate them. Finding derivatives and doing so efficiently is the job of backpropagation, an algorithm that was rediscovered several times and finally implemented with neural networks in the 80s[27]. One should keep in mind that there really is no complicated math behind the algorithm; the only danger here is getting confused by the numbering and notation.

Because the entire network can be seen as a composition of functions, the derivative of the loss with respect to a weight is given by the chain rule. To recapitulate, let us first look at the types of functions a neural network consists of.

First, each neuron multiplies its inputs by its weights and sums the products. The resulting weighted sums will be denoted with the letter s. The weighted sum $s_k^L$ of the kth neuron on layer L is defined as

$$s_k^L = \sum_{i=1}^{n} y_i^{L-1} w_i^L$$

where $y^{L-1}$ are the outputs of the previous layer, and $w^L$ are this neuron's weights. The weighted sum is differentiable of course; the derivative with respect to any weight is simply given by the output it is multiplied by.

To obtain the output of a neuron, its weighted sum is passed through an activation function. Since we're using the sigmoid function, the output $y_k^L$ is given by

$$y_k^L = \varphi(s_k^L) = \frac{1}{1 + e^{-s_k^L}}$$

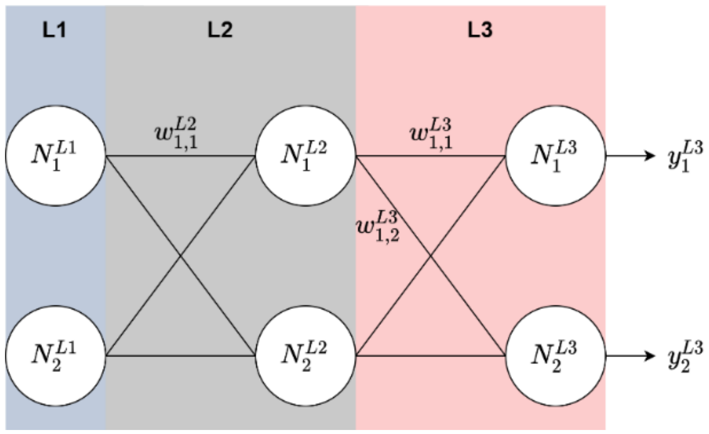
The output is easily differentiable with respect to the weighted sum.

Finally, the outputs of the output layer are compared with a label vector $\hat{y}$ in a loss function. We are using MSE, which is given by the average squared difference. MSE is often divided by 2 in order to obtain a cleaner derivative:

$$C = \frac{1}{2n} \sum_{i=1}^{n} (y_i^L - \hat{y}_i)^2$$

Again, this has a simple derivative with respect to the outputs $y^L$.

Since any function our network consists of is differentiable, we can apply the chain rule.



Consider a fully connected neural network with 6 neurons.

During forward propagation, some inputs are fed through the network, and two outputs are obtained. These are then compared with two labels, and the loss is finally given by MSE. The derivative of the loss with respect to the weight $w_{1,1}^{L3}$ is given by the chain rule:

$$\frac{\partial C}{\partial w_{1,1}^{L3}} = \frac{\partial C}{\partial y_1^{L3}} \cdot \frac{\partial y_1^{L3}}{\partial s_1^{L3}} \cdot \frac{\partial s_1^{L3}}{\partial w_{1,1}^{L3}}$$

This notation might look confusing, but here it is simply a matter of using the chain rule twice. At the start of layer L3, the loss is obtained by using three functions. Similarly, to obtain the derivative, we have to go "backwards" for each of those 3 functions. The terms in the above equation are (left to right): The derivative of the loss function, the derivative of the activation function, and the derivative of the weighted sum with respect to the weight $w_{1,1}^{L3}$, which is equal to the output of the first neuron in L2:

$$\frac{\partial C}{\partial w_{1,1}^{L3}} = \frac{1}{2}\left(y_1^{L3} - \hat{y}_1\right) \cdot \frac{e^{s_1^{L3}}}{\left(1 + e^{s_1^{L3}}\right)^2} \cdot y_1^{L2}$$

We have found the derivative for this weight. This also gives another reason why our activation function has to be differentiable.

However, the above only applies to the weights of L3.
The derivative with respect to a weight in the previous layer L2 will look slightly more complex, as a change in a weight there would affect the loss in both output neurons, rather than just one. While finding those derivatives still merely involves the chain rule, this kind of notation becomes confusing.

Understanding backpropagation becomes much easier if we apply some linear algebra. The good thing here is that we will not need to bother with the confusing notation, as we would not be talking about singular weights, but weight matrices. Instead of writing the output for a single neuron, let us define the output of an entire layer in vector form.

The output of layer L3 is obtained by multiplying its weight matrix by the output vector of L2, and by then passing the result through the activation function:

$$\begin{bmatrix} y_1^{L3} \\ y_2^{L3} \end{bmatrix} = \varphi\left(\begin{bmatrix} w_{1,1}^{L3} & w_{1,2}^{L3} \\ w_{2,1}^{L3} & w_{2,2}^{L3} \end{bmatrix}\begin{bmatrix} y_1^{L2} \\ y_2^{L2} \end{bmatrix}\right)$$

The output of any layer can be defined recursively:

$$y^{k+1} = \varphi\left(W^{k+1} \cdot y^k\right)$$

Where $W^{k+1}$ is the weight matrix of layer k+1.
If L is the output layer and x is a vector of inputs, the output of a network can therefore be written as a composition of functions:

$$y^L = \varphi\left(W^L \varphi\left(W^{L-1} \dots \varphi\left(W^2 \varphi(W^1 \cdot x)\right)\right)\right)$$

Our loss function C then produces a loss based on the output of the network and a label vector $\hat{y}$:

$$C\left(\varphi\left(W^L \varphi\left(W^{L-1} \dots \varphi\left(W^2 \varphi(W^1 \cdot x)\right)\right)\right),\ \hat{y}\right)$$

Assume we want to know how to change the weight matrix of the last layer, $W^L$. To do that, we need to find the gradient $\nabla_{W^L} C$, which is given by the chain rule[*]:

$$\nabla_{W^L} C = \frac{\partial C}{\partial y^L} \cdot \frac{\partial y^L}{\partial s^L} \cdot \frac{\partial s^L}{\partial W^L}$$

To get the gradient for $W^{L-1}$, we only take the first two terms (the gradient for the weighted sum $\nabla_{s^L} C$) and multiply them by 3 more derivatives, thereby going one layer deeper:

$$\nabla_{W^{L-1}} C = \frac{\partial C}{\partial y^L} \cdot \frac{\partial y^L}{\partial s^L} \cdot \frac{\partial s^L}{\partial y^{L-1}} \cdot \frac{\partial y^{L-1}}{\partial s^{L-1}} \cdot \frac{\partial s^{L-1}}{\partial W^{L-1}}$$

$$= \frac{\partial C}{\partial y^L} \cdot \varphi' \cdot W^L \cdot \varphi' \cdot y^{L-2}$$

The equations above show that at each step, the gradient for the weight matrix is always given by the current layer's gradient for the weighted sum $\nabla_s C$ (the first terms), multiplied the outputs of the previous layers (the last term).
A recursive definition for the gradient of the weight matrix can be given:

$$\nabla_{W^{k-1}} C = \nabla_{s^{k-1}} C \cdot y^{k-2}$$

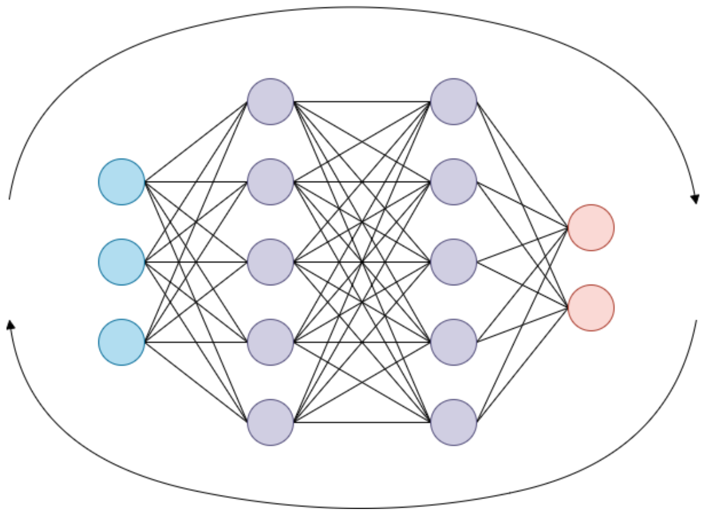
$$= \nabla_{s^k} C \cdot W^k \cdot \varphi' \cdot y^{k-2}$$

[*] Again, the gradient should be a column vector, so we technically need to transpose the RHS. Because $(AB)^T = B^T A^T$, we would need to multiply from right to left.

And that’s about it! As you can see, backpropagation is defined recursively too, just “in the other direction”.
Backpropagation bears a resemblance to forward propagation: Similarly to how we calculate the outputs for each successive layer, we also calculate the derivatives for the weights recursively. Except, here we start from the back with the loss; we propagate backwards for each function we used.

The fact that this process is defined recursively makes the computation much faster. In a way, that’s what backpropagation is really about: Since every function in the network is differentiable, it really comes to no surprise that we can find the derivatives. It’s also obvious that the chain rule is required.

The key, however, is that all of this can be done *efficiently*, and that’s one reason that allows neural networks to work this well.

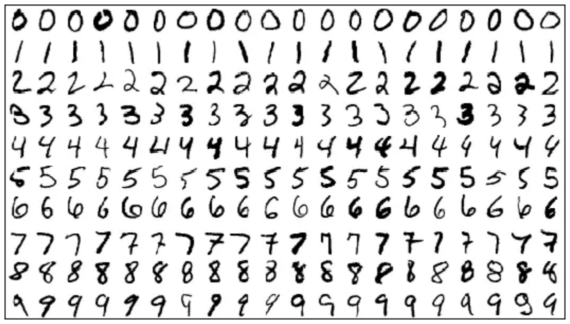


# 6

# IMPLEMENTATION: CLASSIFYING DIGITS

The following chapter will describe the construction of a basic neural network for digit classification built in C++. I will demonstrate the basic concepts described in the previous chapters with the help of the classic example: The MNIST database for handwritten digits.
MNIST contains 70000 black and white 28x28 images, of which 10000 are in a separate file intended for testing the classifier.

You've already seen images from MNIST throughout this script. Here is more:



The problem of hand-written digit classification has, of course, been already solved a while ago. Even in 1998, back when MNIST was released, other machine learning approaches such as support-vector machines were quite successful (and better than neural nets at the time)[28]. Lots of progress has been made since then, and it's now easy to make a well-performing neural network. Some are even able to reach an error rate lower than 0.2% on the test set of 10000 images[29].
Now, because I wrote mine from scratch, it won't perform well, nor will it beat classifiers from 1998 (which is almost embarrassingly bad). Still, it does work for the purpose of demonstration, if nothing else.

Let's see how a network for hand-written digit recognition should approximately look like.

Each image is in grayscale and made of 28x28 pixels, making it a vector of 784 values, all ranging from 0 to 255. The net therefore needs 784 input neurons.
Because the task is to determine which digit an image depicts, and since we use the decimal system, the network needs 10 output neurons, each one returning the "probability" of the image depicting a specific digit. After we have passed an image through the net, we obtain 10 output numbers, the highest signifying the digit a specific image most likely depicts.

The structure of the input and output layers is clear. As already mentioned, the structure of the hidden layers should be considered more carefully, at least in theory.
Interestingly, overfitting really wasn't an issue during training, even when I deliberately tried to do so with very large networks. Overall, It has been observed that many deep neural networks are surprisingly resistant to overfitting, which is actually contradictory to our current understanding[30].

Let’s create a neural network according to the previously established requirements. I mainly used a neural network with 3 hidden layers, each with 16 neurons, but other numbers work just fine too.
The following code defines a net with 784 input neurons, 3 hidden layers and 10 output neurons, along with the loss function and each layer’s activation function:

```
Network net({
	new Linear(784,16, func::act::reLU()),
	new Linear(16, 16, func::act::reLU()),
	new Linear(16, 16, func::act::reLU()),
	new Linear(16, 16, func::act::reLU()),
	new Linear(16, 10, func::act::sigmoid()) },
	new func::loss::MSE()); //use mean squared error loss function
```

The last layer uses the previously shown sigmoid activation function. All other layers use the very popular ReLU, which stands for “rectified linear unit”, and is simply given by x when x > 0, and 0 otherwise. ReLU technically fails the requirement of being differentiable (at x = 0), but we can think of it as having infinitely many derivatives there, all ranging from 0 to 1.

One reason ReLU got this popular is because of its simplicity. It is just much faster to compute than sigmoids, since it only consists of one if statement.

But more importantly, ReLU is not affected by a problem known as vanishing gradient[31].
As shown in the previous chapter, the derivatives are calculated by using the chain rule. If we use the sigmoid activation function, whose derivative is very small at both ends, the weights in the earlier layers are increasingly unlikely to be changed due to the gradient being repeatedly multiplied by the tiny derivatives while propagating backwards. This is known as vanishing gradient, which clearly isn’t a problem for ReLU, whose derivative is always either 1 or 0.

There isn’t that much happening during the creation of a network, but one thing is worth considering: How should the weights be initialized?
What one definitely shouldn’t do is just set all weights to the same value. This would break gradient descent: Because the weight gradient also depends on the weights themselves, and since they’re all the same, the gradient will also be the same for every element. So while the weights would change, all of them would also remain equal to one another, making the network essentially useless.
The next idea is initializing all weights with some random distribution. This is much better; at least the weights could then be changed independently from each other.
However, if we’re not careful, we could again introduce the problem of vanishing gradients (or the opposite, exploding gradients). Because the weight gradient depends on the weights, we shouldn’t choose weights that are too large or too small, or else gradient descent might update them too quickly/slowly.
We need to find a sweet spot for the size of the weights. I use a method known as He-initialization[32], which attempts to do just that by creating a distribution that depends on the size of the weight matrix.

Once the weights have been initialized and the network is ready, we should also create an optimizer to train the net:

```
optim::SGD optimizer(net.layers, 0.1, 0.001);
```

This creates a stochastic gradient descent optimizer with a learn rate of 0.1, which will update the weights of each layer in the network.
Additionally, I defined a decay rate for the learn rate. Decreasing the learn rate after a while helps to avoid overshooting the minimum when close to it, and therefore to further minimize the loss.

We can now begin training the network.
Usually, one would go multiple times through the entire dataset during training. The number of times we have passed through every element in the dataset is referred to as epoch.

Each step in an epoch now essentially consists of:

1. Getting a new batch from the dataset:

```
data::Batch batch = trainLoader.next();
```

   This line simply obtains a batch from the previously shuffled dataset. This batch contains 64 images (the size has been set previously in the code), as well as their respective labels.

2. Propagating forward on each image in the batch:

```
net.forward(*batch[i].input);
```

   During forward propagation, the function forward() is called from each layer, which computes its output. Each layer stores a reference to the output of the previous layer. The output is then simply given by multiplying the layer's weight matrix by the outputs of the previous layer and passing this through the activation function. Although forward() only needs the outputs of each layer, backward() will also require the weighted sums. Because of this, they are also stored in each layer when forward() executes.

3. Propagating backward on each label in the batch:

```
net.backward(*batch[i].label);
```

   During backward propagation, the derivative of the loss is first calculated by comparing the net's outputs obtained in forward() with the label of the image. Then, each layer's own backward() is called, which computes the derivatives of the loss function with respect to the layer's weights. The weighted sums and outputs obtained during forward() are now used again.
   Because the step of SGD is given by the average of all weight gradients across the batch, the computed weight gradient for this image now gets added to a weight gradient sum.

4. Updating the weights with SGD:

```
optimizer.step();
```

   The weights of each layer are updated by subtracting the average weight gradient times the learn rate.

5. Resetting each layer's accumulated sum of weight gradients to 0:

```
optimizer.zeroGrad();
```

Those steps will be repeated for each batch until the end of the dataset is reached. After that, the entire process can be started again in the second epoch in order to further improve the results.

Let’s test the trained network. For each of the 10000 images in the test set, we propagate forward and compare the net’s prediction with the label:

```
//get all 10000 images from the test set
data::Batch batch = testLoader.all();

int numRight = 0;

for (auto element : batch) //for each image & label in the batch
{
       //propagate forward to get the output
       net.forward(*element.input);

       //compare the net’s predicted digit with the label
       if (indexOfMax(*net.output) == indexOfMax(*element.label))
              numRight++;
}

float errorRate = 100 - (numRight * 100.0) / batch.size();
print("Error rate: ", errorRate);
```

With the previously described setup, an error rate of around 10% will usually be obtained after 1 epoch. After that, the improvement slows down significantly. It takes more than 5 epochs to get the error rate down to 5%, and honestly, I haven’t gotten very far past that.
The lowest error rate of 2.7 was achieved with a much larger neural network after only 4 (very slow) epochs, landing us in the ballpark of simple nets from 1998. Still, some of them performed quite a bit better. Why is that?

One thing we didn’t do is apply any preprocessing to the images before training. For example, we could try “deskewing” them, which is essentially a method of straightening a crooked image. This would make the net’s job slightly easier, as it wouldn’t have to learn that a skewed 5 is the same thing as a straight 5. The more general reason has to do with the fact that the sigmoid function and MSE aren’t really used for classification purposes. Modern approaches make use of functions like softmax and cross-entropy, which work a lot better[21] [33]. Cross-entropy in particular is an interesting function originating from information theory, but only works properly with an activation function that outputs probabilities*. However, backpropagation becomes more complicated for those kinds of functions, which is why I haven’t implemented them (yet).

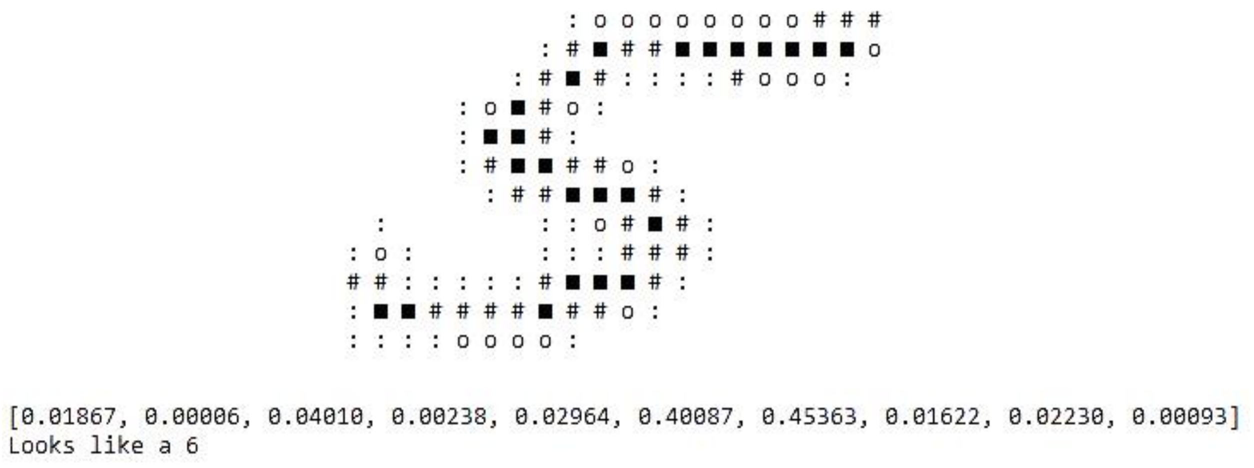
Of course, there are many other ways to make a better classifier, not to speak of the fact that we weren’t even using the type of networks suited for image recognition. There are always ways to improve, but that’s it for now!

* While the outputs of the sigmoid function range from 0 to 1, they can NOT be interpreted as probabilities, as they don’t necessarily sum to 1. This is the case for softmax (which is actually a generalization of sigmoid to multiple dimensions). However, the normalization that happens in softmax makes for a more complicated derivative.

# AFTERWORD

Exploring neural networks was quite fun.
While I obviously didn’t learn about everything in-depth, it was already noticeable how much we simply don’t understand, which I guess is one reason they’re so fascinating.
You MAY have noticed how my writing isn’t particularly formal. I was a bit relu-ctant to write anything at first, so this made my job slightly easier. I hope this information about neural nets propagated well, even though it’s a bit of a loss cause when it comes to “proper” writing.

*Figure 1: My net trying its best*



```
                : o o o o o o o o # # #
              : # ■ # # ■ ■ ■ ■ ■ ■ ■ o
            : # ■ # : : : : # o o o :
        : o ■ # o :
        : ■ ■ # :
        : # ■ ■ # # o :
          : # # ■ ■ ■ # :
  :           : : o # ■ # :
: o :         : : : # # # :
# # : : : : : # ■ ■ ■ # :
: ■ ■ # # # # ■ # # o :
: : : : o o o o :

[0.01867, 0.00006, 0.04010, 0.00238, 0.02964, 0.40087, 0.45363, 0.01622, 0.02230, 0.00093]
Looks like a 6
```

# REFERENCES

[1] "Training a Classifier — PyTorch Tutorials 1.7.1 documentation." https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html (accessed Dec. 11, 2020).

[2] CDC, "Road Traffic Injuries and Deaths—A Global Problem," *Centers for Disease Control and Prevention*, Nov. 25, 2020. https://www.cdc.gov/injury/features/global-road-safety/index.html (accessed Dec. 11, 2020).

[3] "Self-driving car," *Wikipedia*. Dec. 06, 2020, Accessed: Dec. 08, 2020. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Self-driving_car&oldid=992584090.

[4] F. Bussler, "Will GPT-3 Kill Coding?," *Medium*, Jul. 21, 2020. https://towardsdatascience.com/will-gpt-3-kill-coding-630e4518c04d (accessed Dec. 08, 2020).

[5] "MuseNet," *OpenAI*, Apr. 25, 2019. https://openai.com/blog/musenet/ (accessed Dec. 08, 2020).

[6] I. Krementsov, "Artistic GANs," *ISAAC KREMENTSOV NEXUS*. http://isaackrementsovnexus2.weebly.com/ (accessed Dec. 09, 2020).

[7] GPT-3, "A robot wrote this entire article. Are you scared yet, human? | GPT-3," *the Guardian*, Sep. 08, 2020. http://www.theguardian.com/commentisfree/2020/sep/08/robot-wrote-this-article-gpt-3 (accessed Dec. 08, 2020).

[8] V. Gulati, "A Breakthrough in Protein Folding Unfolds," *Medium*, Dec. 02, 2020. https://vgul.medium.com/a-breakthrough-in-protein-folding-unfolds-c6c128328d8 (accessed Dec. 08, 2020).

[9] "Neuron," *Wikipedia*. Dec. 06, 2020, Accessed: Dec. 11, 2020. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Neuron&oldid=992633407.

[10] M. Roos, "Deep Learning versus Biological Neurons: floating-point numbers, spikes, and neurotransmitters," *Medium*, Aug. 16, 2019. https://towardsdatascience.com/deep-learning-versus-biological-neurons-floating-point-numbers-spikes-and-neurotransmitters-6eebfa3390e9 (accessed Dec. 11, 2020).

[11] "Perceptron," *Wikipedia*. Dec. 02, 2020, Accessed: Dec. 08, 2020. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Perceptron&oldid=992000346.

[12] "Linear separability," *Wikipedia*. Nov. 20, 2020, Accessed: Dec. 11, 2020. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Linear_separability&oldid=989728103.

[13] "Artificial neural network," *Wikipedia*. Dec. 03, 2020, Accessed: Dec. 08, 2020. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Artificial_neural_network&oldid=992177465.

[14] "Universal approximation theorem," *Wikipedia*. Dec. 09, 2020, Accessed: Dec. 11, 2020. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Universal_approximation_theorem&oldid=993160017.

[15] A. Kratsios, *Universal Approximation Theorems*. 2019.

[16] Z. Wang, A. Albarghouthi, and S. Jha, "Abstract Universal Approximation for Neural Networks," *arXiv:2007.06093 [cs, stat]*, Jul. 2020, Accessed: Dec. 12, 2020. [Online]. Available: http://arxiv.org/abs/2007.06093.

[17] J. Brownlee, "How to Configure the Number of Layers and Nodes in a Neural Network," *Machine Learning Mastery*, Jul. 26, 2018. https://machinelearningmastery.com/how-to-configure-the-number-of-layers-and-nodes-in-a-neural-network/ (accessed Dec. 08, 2020).

[18] "Overfitting," *Wikipedia*. Dec. 02, 2020, Accessed: Dec. 08, 2020. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Overfitting&oldid=991840184.

[19] "machine learning - Why are neural networks becoming deeper, but not wider?," *Cross Validated*. https://stats.stackexchange.com/questions/222883/why-are-neural-networks-becoming-deeper-but-not-wider (accessed Dec. 15, 2020).

[20] R. Khan, "Why using Mean Squared Error(MSE) cost function for Binary Classification is a bad idea?," *Medium*, Dec. 08, 2020. https://towardsdatascience.com/why-using-mean-squared-error-mse-cost-function-for-binary-classification-is-a-bad-idea-933089e90df7 (accessed Dec. 11, 2020).

[21] jamesdmccaffrey, "Why You Should Use Cross-Entropy Error Instead Of Classification Error Or Mean Squared Error For Neural Network Classifier Training," *James D. McCaffrey*, Nov. 05, 2013. https://jamesmccaffrey.wordpress.com/2013/11/05/why-you-should-use-cross-entropy-error-instead-of-classification-error-or-mean-squared-error-for-neural-network-classifier-training/ (accessed Dec. 12, 2020).

[22] M. A. Nielsen, "Neural Networks and Deep Learning," 2015, Accessed: Dec. 08, 2020. [Online]. Available: http://neuralnetworksanddeeplearning.com.

[23] K. Kawaguchi and Y. Bengio, "Depth with nonlinearity creates no bad local minima in ResNets," *Neural Networks*, vol. 118, pp. 167–174, Oct. 2019, doi: 10.1016/j.neunet.2019.06.009.

[24] L. Bottou and T. B. Laboratories, "Stochastic Gradient Learning in Neural Networks," p. 12.

[25] A. Choromanska, M. Henaff, M. Mathieu, G. B. Arous, and Y. LeCun, "The Loss Surfaces of Multilayer Networks," *arXiv:1412.0233 [cs]*, Jan. 2015, Accessed: Dec. 11, 2020. [Online]. Available: http://arxiv.org/abs/1412.0233.

[26] Y. N. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, and Y. Bengio, "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization," p. 9.

[27] "Backpropagation," *Wikipedia*. Dec. 02, 2020, Accessed: Dec. 08, 2020. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Backpropagation&oldid=991849356.

[28] "MNIST handwritten digit database, Yann LeCun, Corinna Cortes and Chris Burges." http://yann.lecun.com/exdb/mnist/ (accessed Dec. 12, 2020).

[29] K. Kowsari, M. Heidarysafa, D. E. Brown, K. J. Meimandi, and L. E. Barnes, "RMDL: Random Multimodel Deep Learning for Classification," *Proceedings of the 2nd International Conference on Information System and Data Mining - ICISDM '18*, pp. 19–28, 2018, doi: 10.1145/3206098.3206111.

[30] L. Wu and Z. Zhu, "Towards Understanding Generalization of Deep Learning: Perspective of Loss Landscapes," p. 11.

[31] "Vanishing gradient problem," *Wikipedia*. Dec. 13, 2020, Accessed: Dec. 14, 2020. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Vanishing_gradient_problem&oldid=993905886.

[32] S. K. Kumar, "On weight initialization in deep neural networks," *arXiv:1704.08863 [cs]*, May 2017, Accessed: Dec. 08, 2020. [Online]. Available: http://arxiv.org/abs/1704.08863.

[33] "Cross-Entropy Loss Function. A loss function used in most… | by Kiprono Elijah Koech | Towards Data Science." https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e (accessed Dec. 08, 2020).