THE CREATION OF A NEURAL NETWORK IN PYTHON THAT IDENTIFIES HANDWRITTEN DIGITS

Michael Nielsen's book, chapter 1 adaptation

INDEX

INTRODUCTION

The vision system of the human eye is one of the great wonders of the world. Let's consider the following sequence of digits:

The vast majority of people are able to recognize these digits as 504192, which may seem trivial to us, but technically it is not so simple. In each hemisphere of our brains, we have a primary visual cortex with 140 million neurons, which in turn have tens of trillions of connections between them. We have a supercomputer in our heads, which has been modified and improved over hundreds of millions of years, and has adapted to the visual world.

That is why people are very good at giving meaning to what our eyes usually show us, and almost all the work is done unconsciously, which is why we do not really appreciate how complicated the problems our visual system solves are.

This difficulty becomes apparent when trying to write a program that recognizes handwritten digits. What seemed simple to us suddenly becomes extremely complicated, since simple intuitions like "a 9 has a circle on top and a vertical stick below" are not so "simple" to express within an algorithm. When trying to make such rules precise, exceptions and things to consider start to arise, making the task practically hopeless.

On the other hand, neural networks approach the problem from a different perspective. The idea behind these neural networks is to take a large number of handwritten digits (known as training examples) and develop a system that can learn from these training examples.
In other words, neural networks use the examples provided to them to deduce rules for recognizing these digits. Furthermore, if the amount of training examples is increased, the network can learn more about handwriting and therefore increase its percentage of accuracy.

The goal of this work is to write a Python program that implements a neural network that learns to recognize handwritten digits. It is not a very complex or lengthy program, but it ends up having a deduction rate of 96% (which could be increased to 99% with future improvements, although these will not be considered due to the amount of work and effort required for the relatively small gain).
These types of neural networks are already being used today in the world, such as in banks for processing checks and in post offices for recognizing postal addresses.

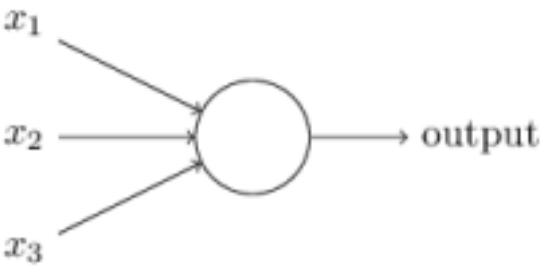
The reason for deciding to do it on handwritten digits instead of digital numbers is that it poses more of a challenge than digital numbers, but at the same time it is not extremely difficult to perform.
In fact, this application is a good way to develop more advanced learning techniques, such as deep learning, which will be discussed later.

To make this work explanation document not so short and limited, the most important key ideas that have been taken into consideration during the project will be developed, in addition to including an explanation of two important types of artificial neurons (perceptrons and sigmoid neurons), and about the stochastic gradient descent model.
That is why the document will be longer than expected, but in exchange, the reader will be able to understand at the end of the work what deep learning really is and why it is so important.

PERCEPTRONS, WHAT ARE THEY?

What is a neural network? To begin with, we will explain what an artificial neuron called a perceptron is, which was created by Frank Rosenblatt between 1950 and 1960. Nowadays, it is more common to use other types of artificial neurons, such as the sigmoid neuron, which we will explain later. However, in order to understand them, we must first understand perceptrons.

So, how do perceptrons work? A perceptron takes different inputs (x1, x2, ...) and produces a simple output:



Rosenblatt proposed a simple rule for calculating the output of perceptrons, using weights which are real numbers that express the importance of the inputs in the total output. The output of the neuron (0 or 1) is determined by the total sum of the weights, which is then compared to a certain value (threshold) to see if it is greater or less than that threshold. Expressed in algebraic terms:

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases}$$

And that's all the complexity of how perceptrons work.

Let's see an example to understand it more clearly. Imagine that a friend invites you to spend a weekend in the countryside, and you have to take into account several factors.

1-. Will the weather be good?

2-. Will more people come or will it just be you and your friend?

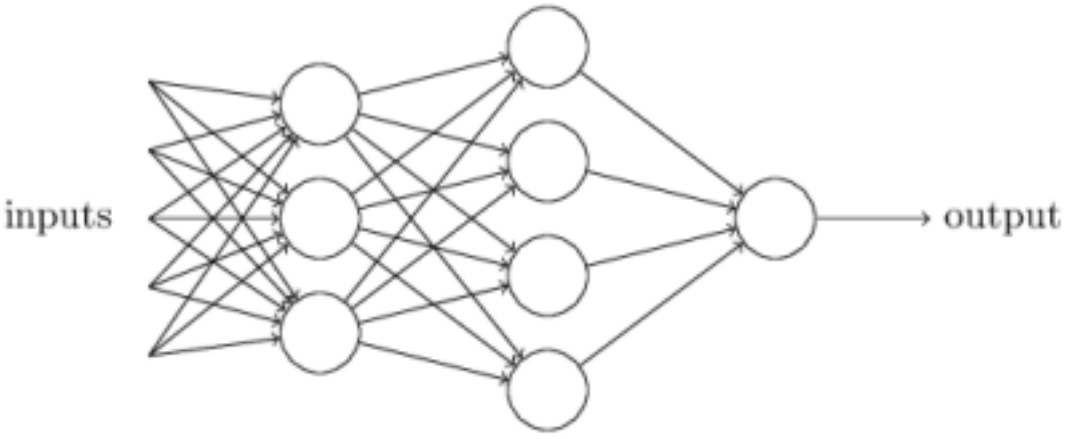
3-. Is it possible to get to the village by public transportation?

We can represent these values by x1, x2, and x3. For example, x1 = 1 if the weather will be good, or x1 = 0 if it will be bad. Similarly, x2 = 1 if you are going with more friends, or x2 = 0 if you are going alone. And likewise for x3 if there is public transportation available or not.

Now imagine that you know that in your friend's town you always have a good time even if not many of your friends are there, but bad weather ruins the plan completely because you wouldn't be able to do anything. This is when you can use perceptrons to make a decision.
For this, we could set the weight w1 = 6 for good weather, and w2 = 2 and w3 = 2. This means that the weather condition is very important for this plan, but going with more friends or the possibility of public transportation doesn't represent as much importance.
Finally, imagine that you choose the number 5 as the threshold. This will result in everything depending on the weather condition in this case. If x1 = 1, the plan will always go ahead, and if x1 = 0, the plan will not go ahead.

Apart from this result, we can also vary the weights and the threshold to achieve different results and ways of making decisions. For example, setting the threshold to 3, then if your friends are going and there is public transportation (x2 = 1 and x3 = 1), the plan will go ahead even if the weather is bad, which will generate another way of making decisions.
Likewise, the lower the threshold, the more you will want the plan to go ahead.

However, obviously perceptrons are not a complete model of human thinking and decision-making, but the example illustrates how a perceptron can give more or less weight to different values to make decisions, and therefore it can be inferred that a more complex perceptron network can lead to quite good and decent decisions.



In this network that we can observe here, the first column of perceptrons (the first layer) makes three simple decisions giving weight to the different inputs, but what do the perceptrons in the second column do?
Well, each of these perceptrons makes a decision based on the decision made in the previous layer, resulting in the ability to make decisions at a more complex and abstract level than the perceptrons in the first layer, and so on, the more layers, the more complexity and sophistication.

It has been mentioned before that perceptrons have a single output and in this network, there are different outputs per perceptron, but they are all the same output, it's just that for visual understanding, the output of one is used as input in several other perceptrons, it's clearer than drawing a single line that is then divided into several.

Let's simplify now the way we describe perceptrons, the way to calculate the output explained above:

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases}$$

This summation can be modified and expressed as a dot product of w and x.

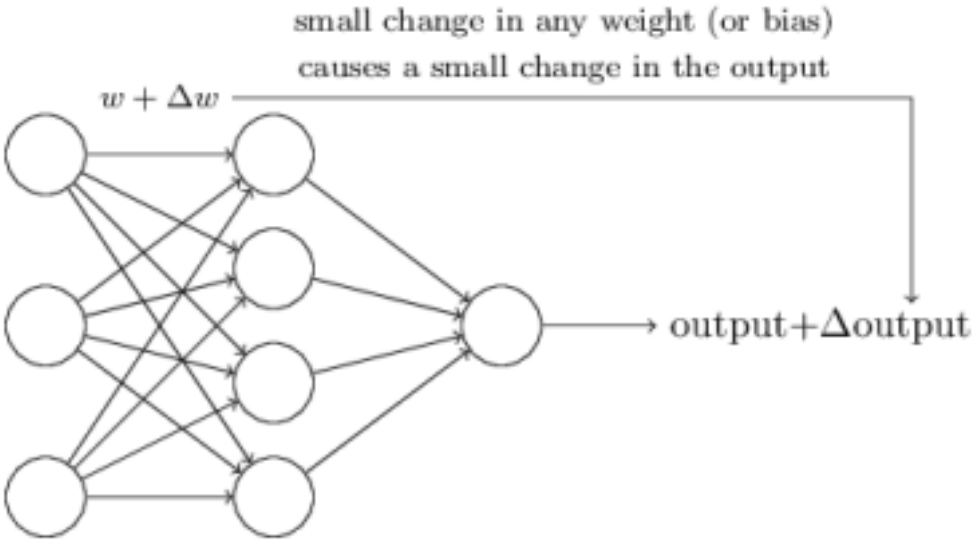
Furthermore, the threshold can be moved to the other side of the greater-than/less-than sign, becoming the perceptron's bias, resulting in the formula:

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

This bias can be taken as how easy it is for a perceptron to output 1, for example, for a perceptron with a very high bias, it is very easy to output 1, but if the bias is very negative, then outputting 1 is really difficult.
This introduction of biases is only a small change in perceptrons, but we will see how it affects in the future (from now on, bias will always be used instead of threshold).

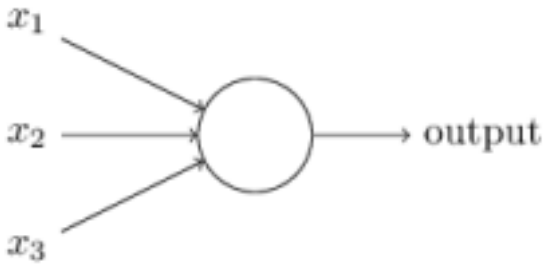
SIGMOID NEURONS, WHAT ARE THEY?

Let's imagine that we have a perceptron network that we want to use to solve a certain problem. For example, the inputs of the network can be the pixel data of a scanned image of a handwritten digit, and we want the network to learn with the weights and biases so that the output is the correct classification of that same handwritten digit.
So, what we're looking for is to make small changes in the weights to generate a small corresponding change in the output of the network, which will make learning of the network possible.



The problem is that this ability to produce small changes in the output by making small changes in the weights is not what happens with a network containing perceptrons. In this network, making such small changes would cause the output to change completely from 0 to 1 (the only values that the output can take with perceptrons).

The way we can solve this problem is by introducing a new type of neuron, sigmoid neurons. These neurons are similar to perceptrons, but modified so that they can undergo small changes in their weights and biases, and thus only produce small changes in the output. This is the crucial factor that will enable them to learn.



Sigmoid neurons can be represented similarly to perceptrons, as they take multiple inputs (x1, x2, ...) and produce a single output. However, in this case, the inputs can be ANY value between 0 and 1, not just 0 and 1 (for example, 0.638 would be a valid input).

Additionally, like perceptrons, sigmoid neurons have a weight for each input (w1, w2, ...) and a total bias b, and the output they produce is not 0 or 1, but $\sigma(w \cdot x+b)$, where σ is what we call the sigmoid function, defined as:

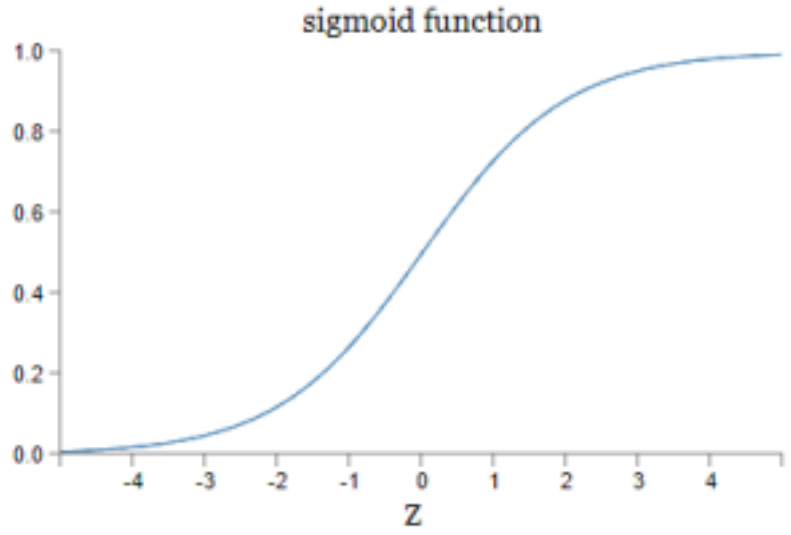
$$\sigma(z) \equiv \frac{1}{1+e^{-z}}.$$

And the total output of a sigmoid neuron with inputs x1, x2, ..., weights w1, w2, ..., and bias b is:

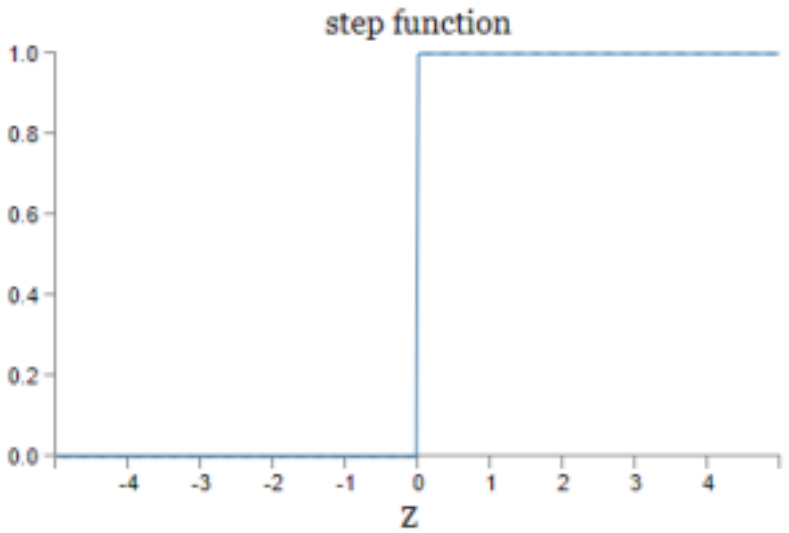
$$\frac{1}{1+\exp(-\sum_j w_j x_j - b)}.$$

At first glance, it may seem that sigmoid neurons are very different from perceptrons, and that their algebraic formula is opaque and difficult to understand if you are not familiar with it. However, the truth is that there are countless similarities between perceptrons and sigmoid neurons, and that the algebraic formula of sigmoid neurons is more of a technical detail than a barrier to understanding.

In truth, the formula for σ itself is not so important; what really matters is the shape of the function it traces:



Which is a "smoothed" version of the step function:

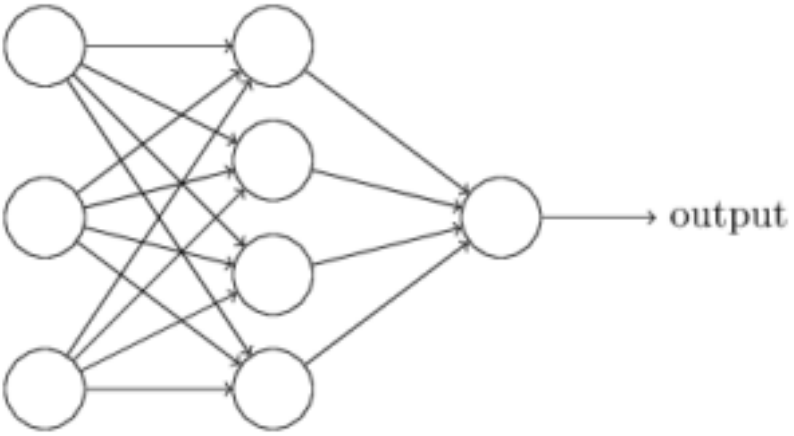


In fact, if σ were the step function, then the sigmoid neuron WOULD BE a perceptron, since the output would be 0 or 1 depending on whether $w \cdot x+b$ is positive or negative.
The result we get when using the current σ function is a "smoothed" perceptron, and this "smoothness" is the key factor, it is what makes small changes in the weights of each neuron and in the bias produce different changes in the output of the neuron.

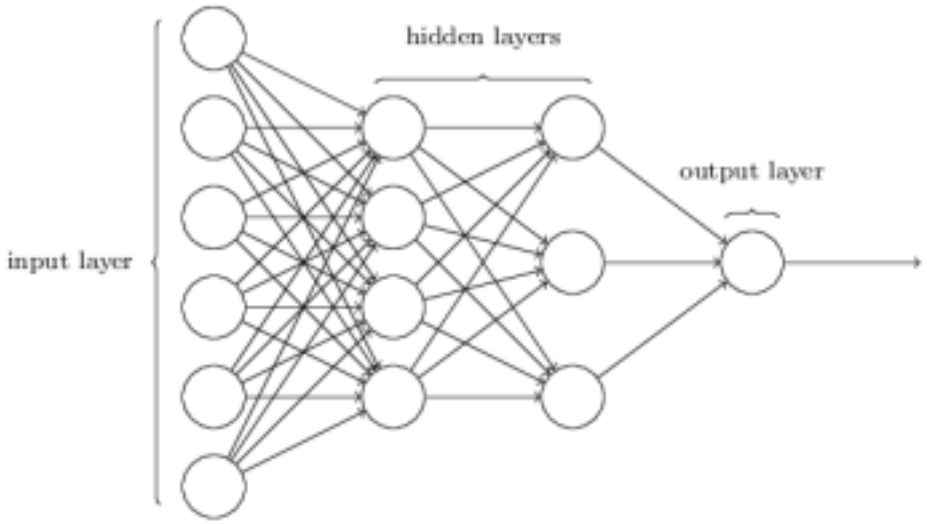
And how is the output of a sigmoid neuron interpreted? Obviously, there is a big difference between perceptrons and sigmoid neurons, and that is that the latter do not just produce a 0 or a 1, and can produce any real number between 0 and 1 (for example, 0.173, 0.689...), which is the key piece for cases where, for example, the average intensity of the pixels of an image introduced as a parameter in a neural network needs to be represented.

THE ARCHITECTURE OF NEURAL NETWORKS

In this section, we will explain, describe, and showcase a neural network that is responsible for classifying hand-written numbers. But before that, let's first clarify some terminology used in neural networks.
Let's imagine we have this network:



The neurons on the left, which receive the input, are called input neurons, and the neuron on the right, which produces the output, is called the output neuron. The intermediate layers are called hidden neurons and are neither input nor output neurons.
In this case, the network above only has one hidden layer, but the complexity of the neural network depends on what is required, for example, this one below has two hidden layers:



The way input and output neurons are designed is very simple. Let's imagine that we want to determine whether a certain image shows a handwritten "9". To do this, the way the neural network will be programmed is to encode the pixel intensities in each of the input neurons.
In this way, if the image is 64x64 pixels in size, then there will be 4,096 input neurons (64x64 = 4,096), and each will receive an intensity scaled approximately between 0 and 1 (thanks to being sigmoid neurons).
Similarly, the output neuron will be a simple neuron that, if its value is above 0.5, then the input image will be a "9", while if its value is below 0.5, then the input image is not a "9".

Now that the design of the input and output neurons is clear, let's move on to talking about the hidden neurons, whose design is a bit more complex. In fact, it is not possible to decide the behavior of hidden neurons with certain pre-established rules.
Instead, researchers and experts in neural networks have developed many heuristic designs for hidden neurons, so that anyone can achieve the behavior they want in their neural network.
These heuristics can also be used to help determine the exact number of hidden neurons that should be used in a particular neural network to avoid hindering the time required to train it.

So far, we have talked about neural networks in which the output of one neuron is the input of another in the next layer; this type of neural network is called feedforward networks, and it means that there are no loops in the network, information and data always move forward, never backward.

In fact, if there were a loop for some reason, we could find ourselves in situations where the input of the σ (sigmoid) functions depended on the output, which would be very difficult to make sense of, so in our case it is better not to allow any loops.

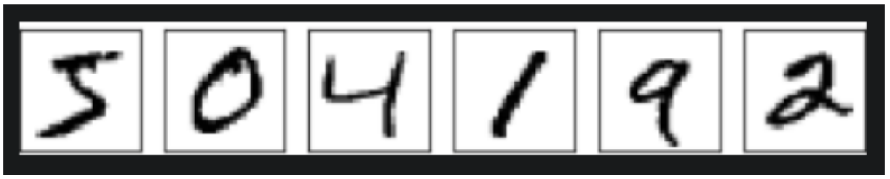
Still, it is important to show the reader that in other types of artificial models of neural networks, loops are possible and useful, and they are called recurrent neural networks. The idea behind these models is to have neurons that produce results for a certain amount of time and then stop and remain inactive, and in that amount of time they have been active producing results, they have been activating other neurons, which also activate for a short period of time, and so on, creating a wave of neurons turning on and off.
In these cases, loops do not generate any kind of problem, since the output of a neuron only affects its input at a later time, not at the same instant.

Recurrent neural networks have been less influential than feedforward networks, in part because the learning algorithms for recurrent networks are (so far) less powerful and capable of performing more complex tasks, but they are still very interesting and much more similar to the way our brain works than feedforward networks.
Furthermore, it is also possible that recurrent neural networks solve important problems that cannot be solved with feedforward networks or can only be solved with great difficulty, but the main objective of this work is the widely used feedforward networks.

GENERATION OF A SIMPLE NEURAL NETWORK

After defining neural networks, we now return to the topic of recognizing people's handwriting, which can be divided into two sub-problems.
First, we need to be able to divide an image containing multiple digits into separate images of individual digits. For example, we need to divide the image:

into six separate images:

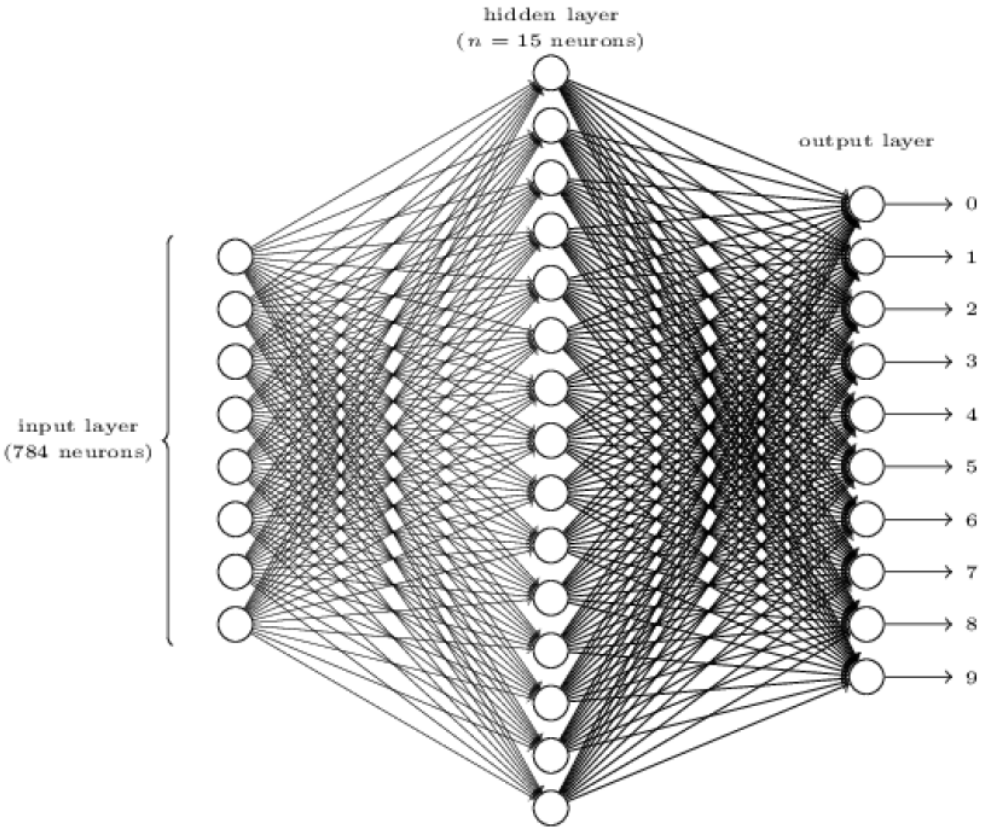


It is clear that this topic, segmenting different digits, is something that we as humans can easily solve, but it is very difficult for a program to properly crop an image.
Secondly, once the image has been divided, the program needs to be able to classify each digit individually. For example, it would be ideal if our program could see the following image:

and identify it as the number 5.

Let's focus on the second problem, classifying individual numbers, since the segmentation problem becomes trivial once you have a good way of classifying individual numbers, due to the many solutions we can apply to achieve it (such as randomly dividing and deciphering if a digit is found in the divided area, thus deciding whether the segmentation has been successful or not). So, instead of worrying about segmentation, let's focus on developing a neural network that can solve a more complicated problem: individual handwritten digit recognition.

To achieve individual digit recognition, we will use a neural network with three layers:



The input layer contains neurons encoding the value of the pixels of the image input at that moment, which in this case would be a 28x28 pixel image, so there would be 784 input neurons (28x28 = 784), each with a determined value between 0 and 1 based on the grayscale value of each pixel (0.0 represents white and 1.0 represents black).

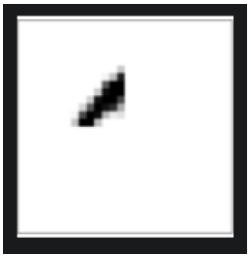
The second layer is the hidden layer, and the number of neurons in this layer is denoted by n, and different values of n can be experimented with (in the image, for example, it takes the value of n = 15).

Finally, the output layer contains 10 neurons, one for each number from 0 to 9, and depending on which neuron ends up with the highest result, the corresponding number to that neuron will be the number the neural network has identified with the input image.

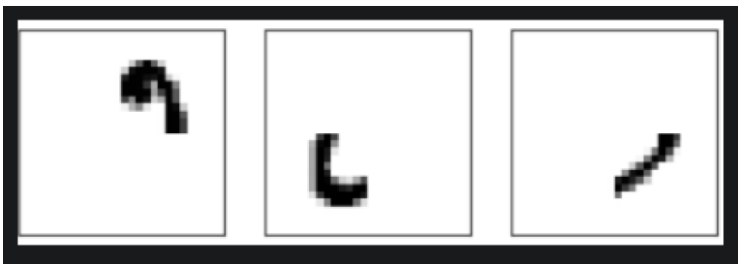
But the reader may ask, why 10 output neurons? If the goal is to identify the number (0, 1, 2,..., 9), then 4 output neurons could be used, treating each neuron as a binary value depending on whether the output is closer to 0 or 1.
So, why does our network have to use 10 output neurons? Wouldn't this be inefficient? Well, the justification is empirical: both ways can be tested and the final result is that the network with 10 output neurons

works much better than the one with 4. Is there any heuristic that tells us in advance why we should use 10 output neurons instead of 4?

To understand this, it will help to think about how the neural network works from its simplest base, let's say we have 10 output neurons this time.
So, first, let's focus on the first output neuron, which is responsible for deciding whether or not the digit is a 0. The way it does this is by weighing the values that have been passed to it from the hidden layer of neurons. And what are the neurons in that hidden layer doing? Well, let's say the first hidden neuron in the hidden layer detects if the image has a segment like this:



To do this, we can give a higher weight to the pixels that intersect with the black parts of the image, and give lower weights to the other input pixels.
Similarly, let's say that the second, third, and fourth hidden neurons detect if the following segments are present:



As you may have deduced, the four images together form a 0.

So if these mentioned four hidden neurons produce a positive result at the same time, we can conclude that the digit in this case is a 0.

Of course, this is not the ONLY proof to conclude that this image represents a 0; you can write a 0 in many other ways (moving it a little bit, slightly deforming the number), but it is safe to assume that in this case the entered number is a 0.

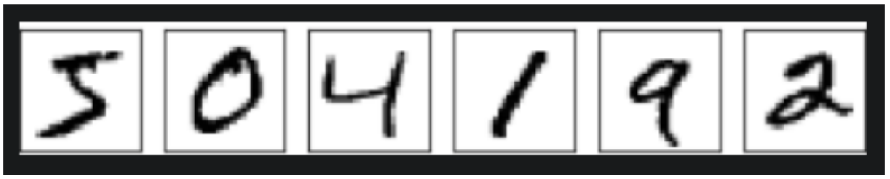
Assuming that the neural network works like this, a plausible explanation can be given as to why it is better to have 10 output neurons instead of 4, since if we had 4, the first neuron would be in charge of deciding which is the most significant bit of the digit, and there is no easy way to relate its most significant bit to the shape of the digit.

Now, with all that said, this is all just heuristic; nothing confirms that a three-layer neural network works the way it has been explained in this document, with hidden neurons detecting simple shapes and partial figures of a digit. Perhaps an intelligent learning algorithm can find a way to assign weights for a neural network with only 4 output neurons, but heuristically, the explained way to make the neural network work is quite correct, and a lot of time can be saved by designing a good architecture for the neural network.

LEARNING WITH *GRADIENT DESCENT*

Now that we have designed the neural network, how can we make it recognize digits?

The first thing we are going to need is a certain dataset for the neural network to learn from, and for this project we will use the MNIST dataset, which contains tens of thousands of scanned images of handwritten digits and their correct classification (mnist.pkl.gz).
The name MNIST comes from the fact that it is a dataset provided and collected by NIST, the National Institute of Standards and Technology of the United States, here are some sample images:



These are, indeed, the digits that have been shown throughout the document, and they will be used by our program for training.

This MNIST dataset is divided into two parts, the first containing 60,000 images that will be used to train the neural network, they are scanned images of digits written by 250 people, both adults and high school students, and are black and white images of 28 by 28 pixels.
The second part contains 10,000 images that will be used as test images, and will serve to evaluate how well the neural network has learned to recognize digits.
To make it more effective, the digits in this second part have been extracted from another 250 different people than those chosen to create the training images, which will give us more confidence that our system can even recognize digits from people whose writing it has not seen during training.

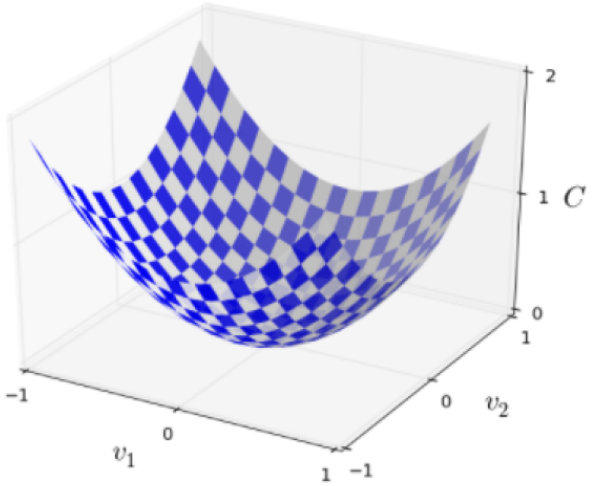
So now, what we want is an algorithm that allows us to decipher which weights and biases we need so that the output result of the neural network is correct for every input data that is entered.

To quantify how well this is going to be achieved, the following cost function is defined:

$$C(w,b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2$$

Here, w is the collection of all the weights in the network, b is the collection of all the biases, n is the total number of training data inputs, a is the vector of output neurons for a given input x (for example, if we input the digit 5 as x, then a should be equal to (0, 0, 0, 0, 0, 1, 0, 0, 0, 0)), and the summation is over all the input data.
We will call C the quadratic cost function, and by inspecting this formula, we can see that C(w, b) will never be negative because none of its components in the summation are negative.
In fact, C(w, b) will tend towards 0 when y(x) is approximately equal to the output a for all input data. This means that the better the weights and biases are adjusted, the closer to zero this function will be. Conversely, the larger the value of C, the worse the algorithm will be at identifying the weights and biases.
That is, the main objective will be to find a set of weights and biases whose value makes the cost function as close to zero as possible, and we do this with gradient descent.

But well, this was just out of curiosity on how we introduce gradient descent into our system. Let's now simplify the function so that it is easier to understand and there are not so many variables at play.
Okay, let's assume that our goal is to try to reduce a certain function, C(v) (this can work for any function of as many values as v = v1, v2,...) (note that w and b have been replaced by v, as we are no longer within the context of neural networks, but in a broader context).
So, to try to reduce C(v), it might help to imagine a two-variable function, v1 and v2:



And the goal we have is, as it may be obvious, to try to get C to the global minimum of the function.
Well, in this particular example maybe it's very simple, but often C will be a much more complicated function with many more variables (as is our case with the neural network), and it won't be as easy to find the minimum point.

One way to approach this problem could be to use algebra and calculus to find the minimum analytically. We can calculate derivatives and try to find the places where C is an extremum, and this can work if, fortunately, C is a function with few variables, but this is a problem when C has a very large number of variables, which is the case with neural networks (as is our case, since the result depends on billions of weights and biases calculated in an extremely complicated way), this option is not effective in our case.

Having ruled out calculus, let's think about the problem in a much more visual way.
Imagine that our function is like a kind of valley between two mountains. Logically, if you place a ball in that valley, it tends to go towards the lowest point of the valley, and it will gradually go down it. Can we use this idea to find the minimum of the function? Let's try it.

We will start by choosing a random point for this (imaginary) ball, and then simulate as if it were rolling down. To perform this simulation, we can calculate the derivatives (and perhaps second derivatives) of C, and these derivatives will tell us what we need to know about the shape of the valley and, therefore, how our ball should roll to reach the bottom.

Let's try to see what happens when we move the ball a certain increment in the direction of v1 and another certain increment in the direction of v2, the formula will change as follows:

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2$$

So, let's try to choose two increments that make C decrease (to make the ball roll downhill in the valley).
To decipher which choice we need to make for this to happen, it will help us to define the total increment of v as follows Δv ≡ (Δv1, Δv2)^T where T is the transpose of the operation, which will change row vectors into column vectors.

We will also define the gradient of C as the vector of partial derivatives, we will define this gradient vector with ∇C:

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2} \right)^T$$

Before we continue, I'll define what the symbol ∇ means so that it's clear. In fact, ∇C can be defined as a unique mathematical object (the vector defined in the above formula), but ∇ by itself is like a label that tells you, "hey, ∇C is a gradient vector."

Combining the definitions given so far, ΔC can be defined as:

$$\Delta C \approx \nabla C \cdot \Delta v$$

And this equation shows why ∇C is called a gradient vector: ∇C relates changes in v to changes in C, which is what we would expect a gradient to do. But the really interesting thing about this equation is that it allows us to vary Δv to make ΔC more and more negative.

Specifically, suppose we choose

$$\Delta v = -\eta \nabla C$$

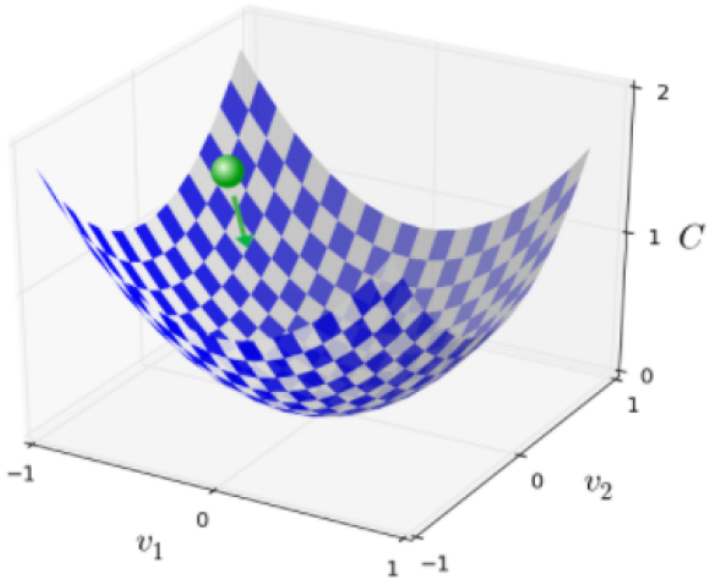
where η is a small positive value (known as the learning rate). Then, the above equation (ΔC ≈ ∇C · Δv) can be modified as

$$\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2$$

And in this way, since ||∇C||^2 will always be greater than or equal to zero, this ensures that ΔC will always be less than or equal to zero, i.e., C will always decrement, never increment if we vary v as defined above, depending on η.
This is exactly the property we were looking for! This function Δv = −η∇C will define the "motion law" of the ball in our gradient descent algorithm, so we'll calculate a value for Δv and then move the ball in the direction of v by that amount:

$$v \to v' = v - \eta \nabla C$$

In this way, we will update the position of the ball using this function every time, and by doing this continuously, we will be able to reduce C until (in theory) we reach the global minimum.



And how can we apply gradient descent to learn in a neural network? Just like in the previous explanation, by varying v, we can make the ball roll to the bottom of the valley. In the neural network, we need to make the weights and biases work in the same way to achieve the minimum of the previously mentioned quadratic cost function.

$$w_k \to w_k' = w_k - \eta \frac{\partial C}{\partial w_k}$$

$$b_l \to b_l' = b_l - \eta \frac{\partial C}{\partial b_l}.$$

To make all of this work correctly, as mentioned before, we need to choose a learning rate (η) that is small enough, because if it's not, we could end up with ΔC values that are greater than 0, which obviously wouldn't be good.

At the same time, we don't want η to be excessively small, because then the gradient descent would move very slowly and the algorithm would act very slowly.

That is, we need to ensure that η works well in the equation and is a good approximation, but it should not make the algorithm excessively slow. We will see later how to achieve this.

This form of gradient descent that we are going to apply has many advantages as we have seen throughout the document, but it has a major drawback: it turns out that computing the second partial derivatives of C can be very costly.
To see why it's very costly, let's consider this; suppose we want to compute all the partial derivatives,

$$\partial^2 C / \partial v_j \partial v_k$$

If there are a million variables vj, then we need to compute around a trillion (a million squared) second partial derivatives, which is computationally very very expensive.

To try to solve this problem, we will apply the so-called stochastic gradient descent, and thus accelerate the program's learning. The idea is to estimate the gradient of ∇C by calculating ∇Cx for a small sample of randomly selected points, and by taking the average of this small sample, we can obtain a great estimate of the true gradient ∇C, having thus greatly accelerated the gradient descent itself, and therefore the learning.

Stochastic gradient descent works by selecting a small number of m randomly selected training inputs, which we will label as X1, X2,..., Xm and refer to them as a mini-batch, and estimating the total cost gradient by computing the gradients of randomly selected values from the mini-batch at each moment.

To connect this with learning in neural networks, suppose wk and bl define the weights and biases in our neural network, then stochastic gradient descent works by selecting a small amount (mini-batch) of training inputs and starting to train with them,

$$w_k \to w_k' = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w_k}$$

$$b_l \to b_l' = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b_l},$$

Then we take another different mini-batch and train with those values as well, and so on until there are no training values left unprocessed, thus completing one training epoch and starting a new and improved one.

Stochastic gradient descent can be thought of as a political poll, it is much simpler to test and work on a small group of data/people (our mini-batch) than to apply gradient descent to the entire group of data/people.
For example, if we have a training dataset of n = 60,000 (like MNIST) and select a small group of m = 10, it means that we will have accelerated the gradient estimation by a factor of 6,000, which of course will not be 100% perfect, but it doesn't have to be, we only care about knowing in which direction C decreases, which means that getting the exact calculation of the gradient is not important.
In fact, in practice, stochastic gradient descent is widely used and a very powerful technique for teaching and learning in a neural network.

IMPLEMENTING THE NEURAL NET IN PYTHON TO CLASSIFY DIGITS

Let's now move on to the interesting part, what really concerns us and the objective of this work, which is to write a program in Python that learns to recognize handwritten digits using stochastic gradient descent and the MNIST data.

Firstly, the cornerstone of the program is the Network class, which will serve to represent the neural network we will be working with, here is the code for initializing a Network object:

```
class Network(object):

    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], sizes[1:])]
```

In this code snippet, sizes is a list containing the number of neurons in each layer of the neural network. For example, let's say we want to create a Network with 2 neurons in the first layer, 3 neurons in the second layer, and 1 neuron in the last layer, generating a neural network of three layers, it would be initialized as follows:

net = Network([2, 3, 1]).

The biases and weights of the Network object are randomly initialized upon initialization, using the random.randn function from the Numpy library (which needs to be imported for the project, along with random) to generate Gaussian distributions with a mean of 0 and a standard deviation of 1.
This random initialization provides a starting point for our stochastic gradient descent algorithm, and although there may be other ways to initialize these biases and weights, this serves our problem well.

It is also worth noting that the Network assumes that the first layer of neurons is the input layer and omits any biases for those neurons since biases are only used to calculate the results of future layers.

It is also worth noting that biases and weights are stored in lists of Numpy matrices, so for example, net.weights[1] is a Numpy matrix that stores the weights connecting the second and third layer of neurons.

To make it simpler (and not write net.weights[1]), we will denote the weight matrix as w, where W_{jk} is the weight that affects the connection between the kth neuron in the second layer and the jth neuron in the third layer.

Here, the reader may think that the letter indexing should be reversed, but using this order means that the activation vector of the third layer would be:

$$a' = \sigma(wa + b)$$

Let's explain this equation: a is the activation vector of the second layer of neurons, and to obtain a', we multiply a by the weight matrix w and the bias vector b, and then apply the sigmoid function to the entire set wa + b.
And with all this in mind, we can now introduce the sigmoid function into our Python program outside the Network class:

```
def sigmoid(z):
    return 1.0/(1.0+np.exp(-z))
```

After this, we need to add the feedforward method to the Network class, which, given a certain input a of the neural network, calculates and returns its result, applying the function seen above for a':

```
def feedforward(self, a):
    for b, w in zip(self.biases, self.weights):
        a = sigmoid(np.dot(w, a)+b)
    return a
```

Of course, the main thing we want our neural network to do is learn, and to do so we will add an SGD method that implements stochastic gradient descent:

```
def SGD(self, training_data, epochs, mini_batch_size, eta,
        test_data=None):

    training_data = list(training_data)
    n = len(training_data)

    if test_data:
        test_data = list(test_data)
        n_test = len(test_data)

    for j in range(epochs):
        random.shuffle(training_data)
        mini_batches = [
            training_data[k:k+mini_batch_size]
            for k in range(0, n, mini_batch_size)]
        for mini_batch in mini_batches:
            self.update_mini_batch(mini_batch, eta)
        if test_data:
            print("Epoch {} : {} / {}".format(j,self.evaluate(test_data),n_test))
        else:
            print("Epoch {} complete".format(j))
```

Let's first talk about the variables/arguments passed to the function. training_data is a list of tuples (x, y) that represents the input training data and expected results, epochs and mini_batch_size are what they seem, the number of times to train and the size of the groups to sample, and eta is the learning rate. If test_data (optional argument) is present, the program will evaluate the neural network after each training epoch, and print out the partial progress made in each one, allowing for progress and learning to be recorded, although it slows down the process a bit.

The code works as follows: in each epoch, the training data is randomly shuffled and divided into mini-batches of approximately the same size, providing an easy way to sample the training data.
Then, for each mini-batch, a single gradient descent step is applied through self.update_mini_batch(mini_batch, eta), which updates the weights and biases of the neural network according to a single iteration of the gradient descent, using only the training data in that mini-batch (the ball starts rolling down the mountain).

This update_mini_batch method is programmed as follows within our Network:

```
def update_mini_batch(self, mini_batch, eta):
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    for x, y in mini_batch:
        delta_nabla_b, delta_nabla_w = self.backprop(x, y)
        nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
        nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
    self.weights = [w-(eta/len(mini_batch))*nw
                    for w, nw in zip(self.weights, nabla_w)]
    self.biases = [b-(eta/len(mini_batch))*nb
                   for b, nb in zip(self.biases, nabla_b)]
```

Where most of the work and computation is done on the line delta_nabla_b, delta_nabla_w = self.backprop(x, y), which invokes something called the backpropagation algorithm, a fast and efficient way to compute the gradient of the cost function.
Thus, update_mini_batch simply works by computing these gradients for each training example within the mini_batch, and then updates the self.weights and self.biases appropriately and in accordance with what has been computed.

The code for self.backprop will not be developed in this work for practical reasons, and that is because it would make the document too lengthy (more than it already is) and the necessary mathematical calculations and expressions that need to be explained are more complex, so here is a link to a video (it is not necessary to watch the previous parts if this work has been understood so far) that explains it in a very didactic and simple way so that if anyone is interested in its functioning, they can satisfy that knowledge.
For those who do not want to delve further into the topic, let's assume that self.backprop behaves as described, and returns the appropriate gradient for the cost associated with the training example x.

Let's now take a look at the entire program, adding more functions to those already explained here.
Except for self.backprop, the program is fairly self-explanatory, and all the important weight and calculation work is done in self.SGD and self.update_mini_batch, which have already been seen and explained. In addition, self.backprop makes use of two extra functions, sigmoid_prime, which computes the derivative of the σ function, and self.cost_derivative, which returns the vector of partial derivatives for a given vector of activations (output_activations).

Now let's test the program we have developed, how well does it recognize handwritten digits?

Well, let's start by loading the MNIST data using a small program, mnist_loader.py (provided by MNIST to process their data). This can be done by running the following in the Python shell:

```
>>> import mnist_loader
>>> training_data, validation_data, test_data = \
...                 mnist_loader.load_data_wrapper()
```

After loading the MNIST data, we will use our Network.py file to create a Network object with 30 hidden neurons by writing the following in the Python shell:

```
>>> import Network
>>> net = Network.Network([784, 30, 10])
```

The 784 refers to each of the pixels in an MNIST image (as explained earlier in the work, 28x28), and the 10 refers to each of the output neurons, each representing a digit (0 to 9).
Once our Network object is created, we will apply stochastic gradient descent to learn from the MNIST data over a period of 30 epochs, with a mini-batch size of 10 and a learning rate of η = 3.0:

```
>>> net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
```

This will take some time to execute (due to the number of epochs, if you want to speed up the process, you can reduce the number of epochs required for learning and also reduce the number of hidden neurons).
Once our Network is trained, it will have no trouble analyzing and classifying each digit that is input to it, the slowness only occurs during the learning process, as is typical in any machine learning algorithm.

While the program is running during its different stages, it can be observed that in just the first epoch it already has around 9,000 out of 10,000 correct predictions, because that's how good this method is, and this value increases as the number of training epochs increases:

```
Epoch 0 : 9028 / 10000        Epoch 26 : 9458 / 10000
Epoch 1 : 9143 / 10000        Epoch 27 : 9469 / 10000
Epoch 2 : 9241 / 10000        Epoch 28 : 9450 / 10000
Epoch 3 : 9289 / 10000        Epoch 29 : 9485 / 10000
```

As a result, our Network achieves a success rate of 95% (it may vary depending on the execution due to the random initialization of weights and biases), but it is a very encouraging result for its first attempt.

Let's now see what happens if we vary the number of hidden neurons from 30 to 100, executing exactly the same code but in this case, we will create the Network object like this:

```
>>> net = Network.Network([784, 100, 10])
>>> net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
```

This will raise the results to around 96-97% (although it starts with lower values than the previous example, it will achieve greater accuracy with more hidden neurons).

Achieving these accuracies has not been by chance, and it's because we have had to do tests with training epochs, mini-batch size, and the learning rate η, which are known as hyper-parameters of our Network (to differentiate them from weights and biases, normal parameters that our neural network learns to give them value).
If these hyper-parameters are poorly chosen, it is not difficult to obtain bad results, as occurs, for example, if we choose a learning rate of η = 0.001:

```
>>> net = Network.Network([784, 30, 10])
>>> net.SGD(training_data, 30, 10, 0.001, test_data=test_data)
```

It can be seen that the obtained data is much less encouraging:

```
Epoch 0 : 931 / 10000        Epoch 27 : 2470 / 10000
Epoch 1 : 965 / 10000        Epoch 28 : 2520 / 10000
Epoch 2 : 982 / 10000        Epoch 29 : 2569 / 10000
```

Even so, some learning is perceived during the stages, suggesting that by increasing the learning rate to, for example, 0.01, we will obtain better results, and so on until we reach 3.0, which is the "ideal" learning rate for our project according to the tests carried out.
This means that, even if we poorly choose the hyper-parameters of our Network, we have some information that will help us improve them and reach the correct or most suitable ones for our case.

But although it may seem trivial, debugging a neural network is a challenge. For example, imagine that with 30 hidden neurons, we are trying to keep increasing the learning rate, and in this case, we have reached η = 100.0:

```
>>> net = Network.Network([784, 30, 10])
>>> net.SGD(training_data, 30, 10, 100.0, test_data=test_data)
```

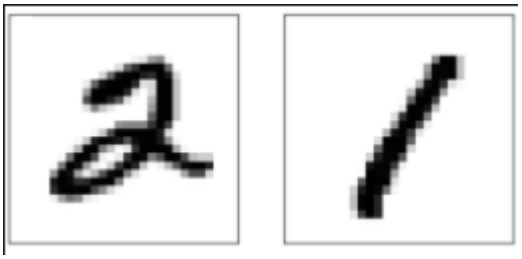
Here we will realize that we have gone too far, and the learning rate has a value that is too high:

```
Epoch 0 : 892 / 10000        Epoch 27 : 892 / 10000
Epoch 1 : 892 / 10000        Epoch 28 : 892 / 10000
Epoch 2 : 892 / 10000        Epoch 29 : 892 / 10000
```

The lesson to learn here is that debugging a neural network is not easy, and although it may seem like simple programming, there is a certain art and complexity behind it. If someone wants to obtain good results in their neural network or any machine learning program, they have to learn to debug their projects correctly.

As we have seen, our program and neural network obtain fairly decent results, but what does that mean? Decent compared to what?
It is informative to have a test program (not a neural network) to compare results and understand if our neural network is really working correctly or not.
The simplest thing is to have a program that randomly guesses the number, although that would be a very simple example, it's better than nothing, but let's go further.

Let's try to see how "dark" an image is, since for example, a 2 will always have a little more "darkness" than a 1, since there will be more pixels marked in black, which can be checked in these images:



Upon receiving each new image, the program calculates how "dark" it is, and then tries to guess and deduce which digit it is by the average of dark pixels in the image.

This is a very simple procedure, and it is easy to program, so the entire procedure will not be explained. We will simply add the program as mnist_average_darkness.py, and we will get a result of 2,225 out of 10,000, that is, an accuracy of 22.25%.

It is not very difficult to achieve accuracies between 20 and 50 percent, or even surpass 50 percent.
Let's try to do this by using one of the most well-known algorithms, the SVM algorithm or support vector machine (if you are not familiar with this algorithm, it is not necessary to know the details of it to understand this application). Instead of applying this algorithm, we will use scikit-learn, a Python library that provides a library for SVM known as LIBSVM.

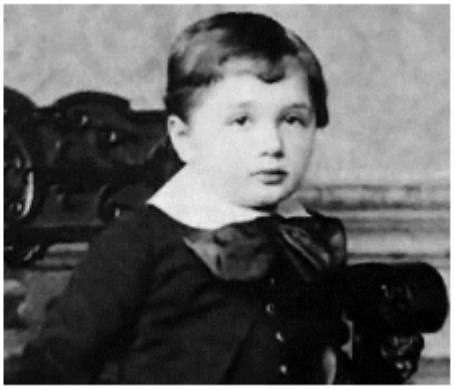
If we execute scikit-learn's classifier using default settings, we obtain an accuracy of 93-94 percent (the code can be found in mnist_svm.py), which is already a great improvement over our previous simple program.
In fact, this means that the SVM algorithm is performing almost as well as neural networks, and changing its parameters can make its performance reach 98.5% accuracy, which means that this algorithm, with well-chosen and adapted parameters, only fails one digit out of every 70, which is very good. But can neural networks do better?

The answer is yes, currently neural networks outperform any other way of classifying MNIST data, including the SVM algorithm, and the current record is the correct classification of 9,979 digits out of 10,000, which reaches a level very equivalent to that of humans, and may even be arguably better, since some images from MNIST can be difficult to identify even for humans, such as:

And to achieve this accuracy, only some improvements have been made to the program presented in this exercise, which demonstrates that a simple learning algorithm and good training data often outperform a very sophisticated algorithm.

EPILOGUE: TOWARDS "DEEP LEARNING"

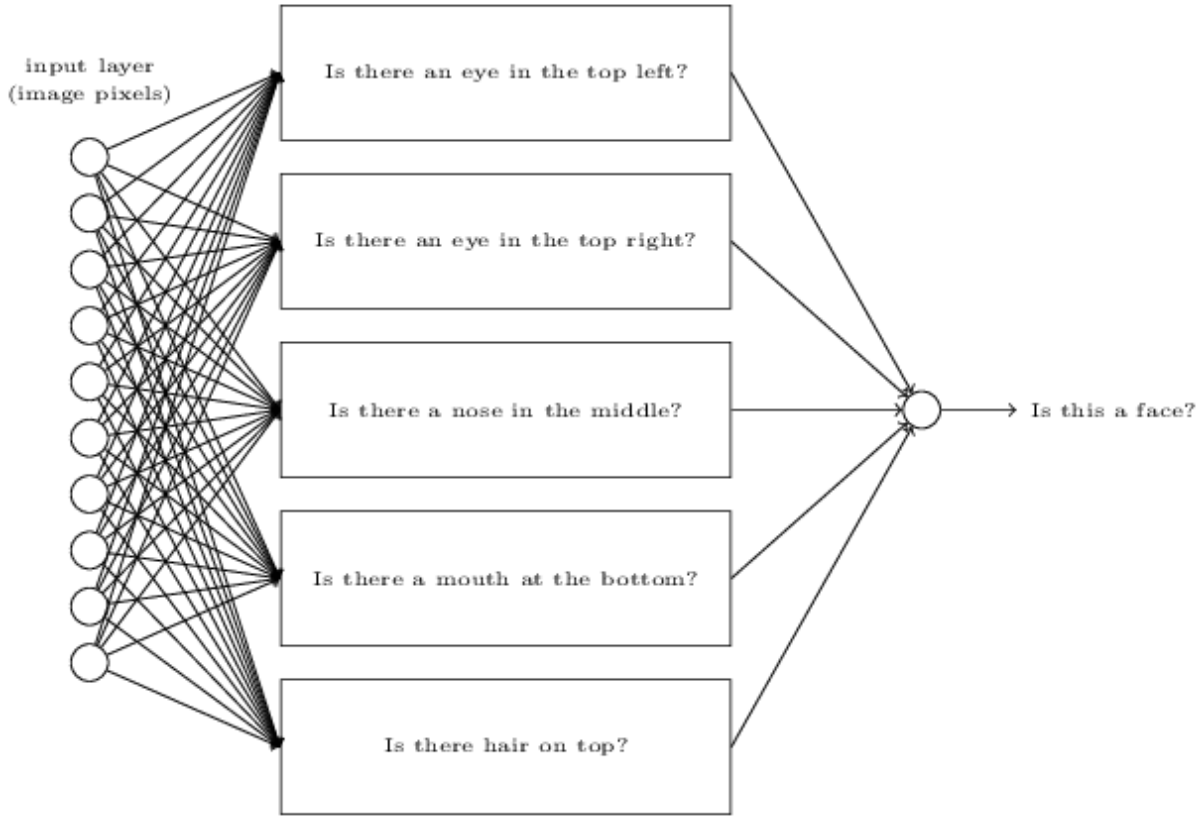
To conclude this work, let's go back to the interpretation given to artificial neurons at the beginning, as tools to weigh evidence and data. Let's imagine we want to determine whether an image shows a human face or not:


We can approach this problem in the same way we approached the problem of recognizing handwritten digits, using the pixels of the image as input data and having the network calculate and output whether it is indeed "a face" or "No, it is not a face".

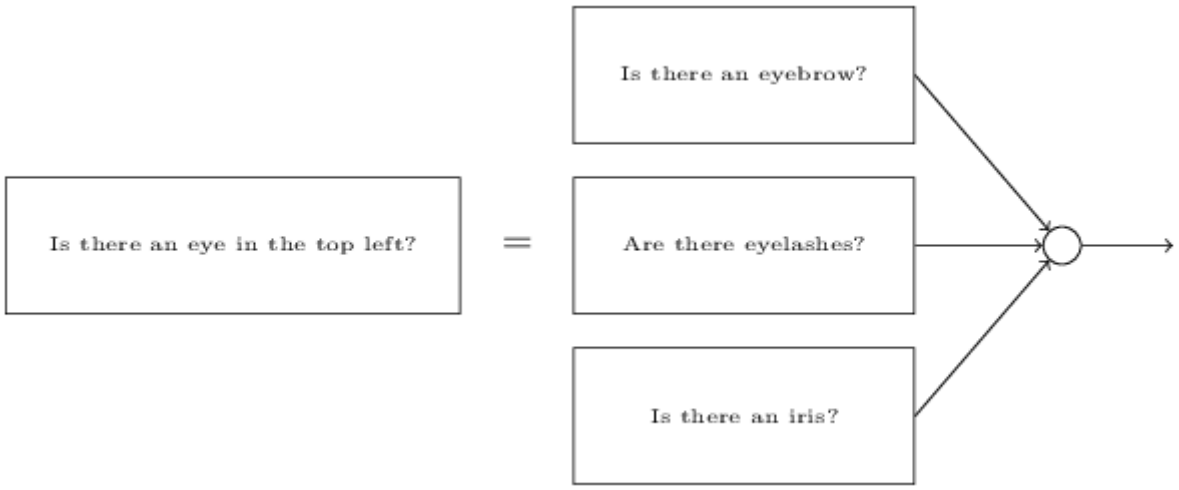
Suppose we do this, but without using a learning algorithm, and use the following heuristic: "Does the image have an eye in the upper left?" "And in the upper right?" "Is there a nose in the middle?" "Does it have a mouth in the central lower part of the image and hair in the central upper part?"...
If the answer to many of these questions is "yes," or "probably or partially yes," then the network will conclude that it is a face, and vice versa, if many answers are "no," then it will determine that it is not a human face that the image shows.

Of course, this is a somewhat "naive" heuristic and suffers from many deficiencies (the person may be bald and therefore have no hair, only part of the face may be shown, or the face may be at a certain angle...), but this heuristic suggests that if we can build neural networks to respond to each of the above questions, then we can also combine those neural networks to address the facial detection problem.
Next, I add a possible architecture, with rectangles for each of the sub neural networks (this is not a realistic approach to the problem, it will only help us better understand how neural networks work).



So let's imagine that we enter into the first neural subnetwork, "Is there an eye in the upper left part?", which can be broken down into other questions such as, for example, "Is there an eyebrow?", "Are there eyelashes?", "Is there an iris in the center?"..., and although these questions also depend on different situations and positions, we will keep the simple sentences instead of "Is there an eyebrow in the upper left part and is it also above the eyelashes and iris?" to show the problem more simply.

Now we know that we can divide this neural subnetwork into the following:



These questions can also be subdivided many more times into multiple layers, and the final result will be subnetworks whose question can be answered with simple individual pixels.
These questions will check, for example, the presence or absence of simple shapes at certain points in the image, questions that can be answered with simple neurons connected to the same pixels of the image.

The final result is networks that divide very complex problems (such as whether a face appears in the image or not) into very simple problems that can be answered at the level of individual pixels, and this simplification occurs through different layers, with the first ones answering very simple and specific questions about the image and the latest and most advanced ones that build a hierarchy of more complex and abstract concepts.
These neural networks with this type of structure with multiple layers (two or more layers of hidden neurons) are called deep neural networks.

And since 2006, a set of techniques has been developed that allow learning in deep neural networks, and this learning is based on stochastic gradient descent and backpropagation, although they also include other ideas.
These techniques together have allowed much larger and deeper-layered networks to be trained (now, neural networks with 5 to 10 hidden layers are normally trained), and it turns out that they work and learn different problems than simple neural networks that have, for example, one or two hidden layers.
The reason for this is, of course, the ability of deep neural networks to build a hierarchy of concepts, and comparing these networks with simpler networks is like comparing a programming language with the ability to call functions with a "naked" programming language without the ability to make those calls, and abstraction in neural networks works differently than it does in conventional programming, but it is equally important in both cases.