Coin recognition using convolutional neural networks

1 Introduction

Coin recognition systems are widely used by vending machines, banks, supermarkets, etc. Most of them use mechanical or electromagnetic based systems. These systems are very costly and require specific equipment. If we would want to buy a coin counting machine for CHF, the prices start at 250.- [1]. And if one wants to count another currency one has to buy another machine. With the advances in computing, image processing-based systems have become more and more appealing as a potentially better alternative. Using computer vision has the potential to offer a cheaper and more portable alternative for coin recognition. Requiring only a camera and some computing power on the hardware side allows for a very democratic use of the system. Its potential integration into mobile apps would enable a new range of possible applications.

The field of coin detection systems using computer vision is not new. But many solutions, although they reach great results, require some kind of object next to the coins [link]. And, in the case where they don't, they often have a long runtime (up to a minute) [link]. It is worth noting that the task of recognizing a perfect image (no rotation, very good cropping, no angle) of a single coin has been successfully achieved by the mobile application *Coinoscop* or the website *Coinscatalog*, which identify foreign coins.

Change counting systems can also benefit everyday people. In our time of instant payment methods, counting change has become a tedious task. In addition, who has never been stressed at the grocery store line in front of the cashier because of it. You get your wallet out and realize you only have small coins, so you start counting them one by one. Finally, you end up giving the cashier a handful of coins that takes him/her just as much time to sort out, while having caused yourself unnecessary stress and having annoyed the whole lane behind you. This could be solved by a mobile app that tells you the total amount of change you have and would highlight through the smartphone camera which coins you should give the cashier. Although it's strict performing conditions, invariant background, right angle with the coins and more or less fixed distance to the coins, the proposed method can be seen as a small step towards a solution that could go beyond the limitations that I have just stated.

Coin localization and classification is an important task of computer vision as it is used in various autonomous machines; automatic fare collection machines, self-checkout machines, public transport ticket machines, arcade gaming machines, car park ticket machines, and vending machines. Coin recognition is also a challenging task because of the wide array of patterns that it must span. A good algorithm should be robust to changes in:

- rotation
- translation
- background
- lighting
- distance to coins (coin size)

In this paper, I am going to use Convolutional neural networks to detect coins in an image.

Convolutional neural networks, also known as ConvNets or CNNs, are a specific type of artificial neural network. They are primarily used for analyzing images. Yann Lecun developed the first convolutional neural networks called LeNet [2] in 1988 but because of the limited computing power at the time, it is only since 2008 that it has been the most popular method in image classification competitions. Unlike other computer vision algorithms, CNNs don't require humans to extract the important features to look for. Instead, they will learn through data which feature and combinations of features are important. Their main strength, compared to conventional neural networks, is that they take into account the spatial information of the image. Where a normal neural network would flatten an image into a single vector of pixels, the convolutional neural network will keep and process the image as a matrix. Convolutional neural networks excel at analyzing visual imagery as they are now beating humans on the ImageNet large scale visual recognition challenge [3]. Convolutional neural networks learn filters, which extract the important features, that in traditional algorithms [4], would have been hand-engineered. They therefore almost don't need any pre-processing. This makes them perfect for image classification and therefore good candidates to solve the coin recognition task. So, my task at hand in this paper is to develop an efficient algorithm that accurately localizes and recognizes coins with the help of convolutional neural networks as it is a consequent interest of mine. Another option could have been to use hand-engineered features in order to recognize the coins (but that would have been tedious and less interesting).

On a more personal note, this project interests me for several reasons. I have always been fascinated by the fact that we can teach a computer to "see". Most people think that I am overestimating the potential of machine learning techniques, although I still think to be underestimating them. I believe that automating low-level tasks, such as coin counting, are stepping stones to be able, in the future, to get to more complex tasks. By enabling the automation of tedious and boring jobs, machine learning is powering a transition to more interesting professions. Furthermore, I am extremely enthusiastic at the prospect of applying my programming knowledge into the most complex and unknown project I have ever faced, thereby improving my understanding of the subject and my ability to handle big projects.

But I also hope that the proposed method could be used in mobile applications for innovative use cases. Or in the worst-case scenario, to give information on what NOT to do when developing such algorithms. As the full code will be posted on GitHub, any app creator could integrate it into his or her app as a tool to detect coins. Moreover, I aim at conveying a good understanding of what convolutional neural networks are and give a first impression of what is possible (and maybe not possible) with such methods.

One important observation is necessary when searching for interesting approaches to this problem. The objects we want to detect all have one distinguishing feature in common: they are circles. In other words, the objects we want to classify all belong to the subset of circles and in this subset, we then have our different coins. However, finding circles in an image can be efficiently done by an algorithm that does not use neural networks (see list in [4]) such as Hough Circles [5]. This key point enables us to divide the problem into two parts:

1. Localize the coins in the image by drawing a bounding box around them. Then extract individual images out of each bounding box and pass them to step 2.
    ○ Input example of this step (figure 1): xxx

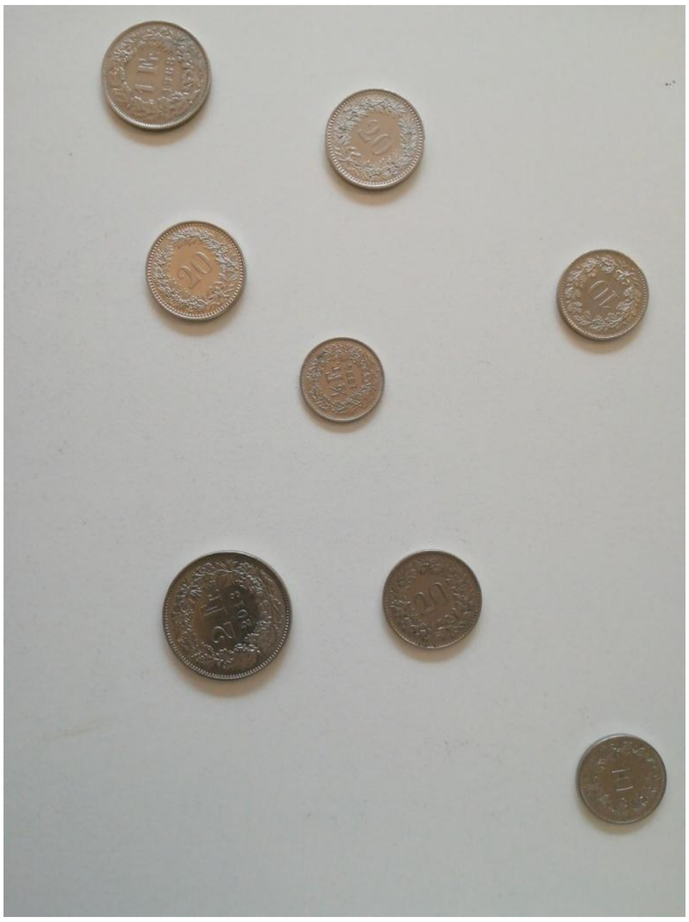


Figure 1: an example image that could be taken as input by the first algorithm

- Output example of this step (figure 2):

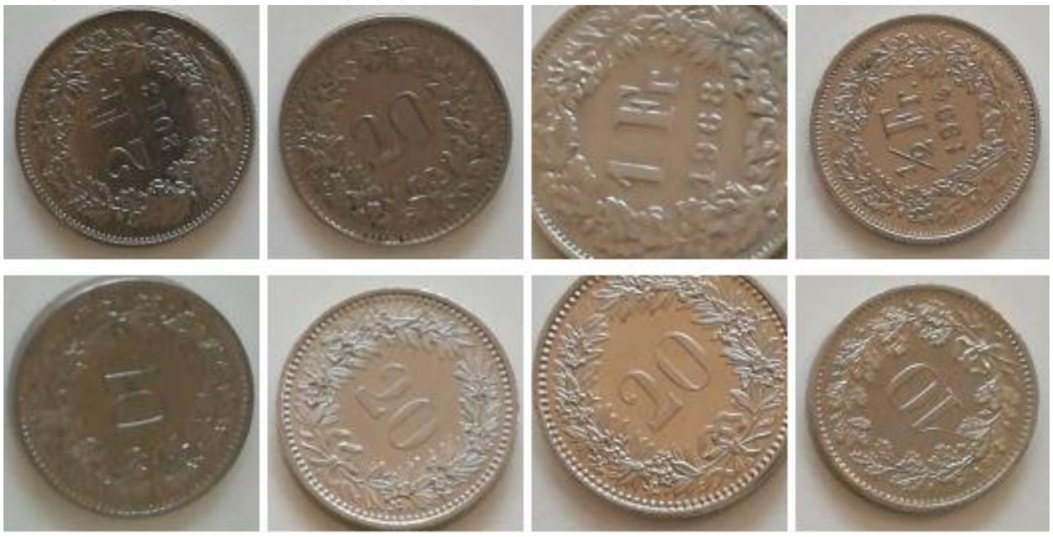


Figure 2: a set of individual images that represent all the coins in the original image. This set of images is the output of the first algorithm.

2. Take the images of step 1 and classify them in the appropriate category (2fr, 1fr, etc). I have chosen not to include 5fr technical reasons. The output of this step is, therefore, a 7-dimensional vector, each dimension representing the amount of a particular category found in the original image (2fr, 1fr, 50rp, 20rp, 10rp, back of fr, back of rp).

This method of proceeding should drastically reduce the computing power necessary to run my algorithm. This makes sense because the convolutional neural network used for classifying coins is sophisticated and is computationally expensive. It is searching for complex features and combinations of features in order to distinguish between coins. Therefore, it wouldn't make sense to apply all this processing to an entire image that contains mostly background (see image above). Dividing the problem into two distinct sub-problems enables us to use a more suited and adequate technique for each of them specifically. The first algorithm will have to detect circles on a large image efficiently, while the second must be trained to distinguish between types of coins on a smaller image.

One classical method to perform the first step, which consists of localizing all objects in the image which are circular, is to use the Hough transform. This technique can be implemented with a python library called OpenCV. The output of this step is a set of images that will undergo separate classification.

For the second part, one apparent problem should be addressed: the limited amount of data. Neural networks require a lot of training examples and generally perform better the more of it they receive, especially in computer vision tasks. The amount of data that we could get with an existing dataset like the Swiss Coins dataset [6] (~40 images per face) would not be enough to train a robust and accurate neural network. Two possible techniques that we could use to address this problem are data augmentation and acquiring more data. Data augmentation for those who might not know, is a strategy which increases the diversity and size of a dataset by applying certain transformations to the current dataset.

With regard to the network architecture, I will be using a modified version of one of the well-known architectures that follow:

- LeNet-5 was proposed by Yann LeCun, Leon Bottou, Yosuha Bengio, and Patrick Haffner in 1999 for optical character recognition [7]. It is an architecture that was designed to recognize handwritten digits and takes low resolution images(32x32) as input. It is able to work with patterns of medium to low complexity.

- AlexNet, this architecture won the ImageNet competition in 2012. It was published by Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton in 2012 [8]. It is an architecture that was designed for the ImageNet competition and takes medium resolution images(234x234) as inputs. It is able to work with patterns of high to medium complexity (cats, ships, cars, etc.).

- VGGNet, this architecture took second place in the ImageNet competition in 2014 and was developed by Karen Simonyan, Andrew Zisserman [9]. It is an architecture that was designed for the ImageNet competition and takes medium resolution images(224x224) as inputs. But It can manage more complexity and is more performant than the AlexNet.

I will be coding these neural network architectures in the programming framework TensorFlow and will be modifying the last layer in order to fit the task at hand (by having 7 outputs nodes instead of 1000 for example). I will use TensorFlow to build the convolutional neural network. TensorFlow is an open source library released by Google in 2015 to build and design machine learning models.

As in any machine learning project, my path will include a lot of trial and error. Therefore, I am sure that once finished I will laugh at how wrong I was while writing these above lines. Nonetheless, everyone has to start somewhere.

2 Neural Networks

In this section, I will give a high-level understanding of the neural network used. In order to understand the convolutional neural network used in the proposed method, some basics about neural networks, in general, must be outlined. Therefore, I will first introduce a simple neural network through the well-known handwritten digit recognition task. This will provide a stepping stone to then be able to more easily understand what convolutional neural networks are.

2.1 Simple Neural Network

This section is based on information from the following sources [10][11][12]

2.1.1 Dataset

As said earlier neural networks need data and often lots of it. In order to classify handwritten digits, we will use the MNIST dataset [13]. The MNIST dataset is a collection of images of handwritten numbers. It consists of 70'000 images with a resolution of 28 by 28 pixels. Some examples are shown below

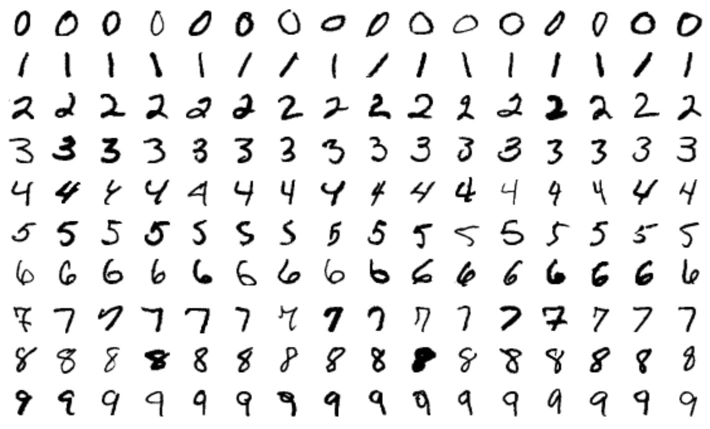


Figure 3: Examples of the MNIST dataset

2.1.2 The neural network architecture

Each image in our dataset is composed of 28 by 28 pixels. Each pixel has a value ranging from 0(black) to 1(white).

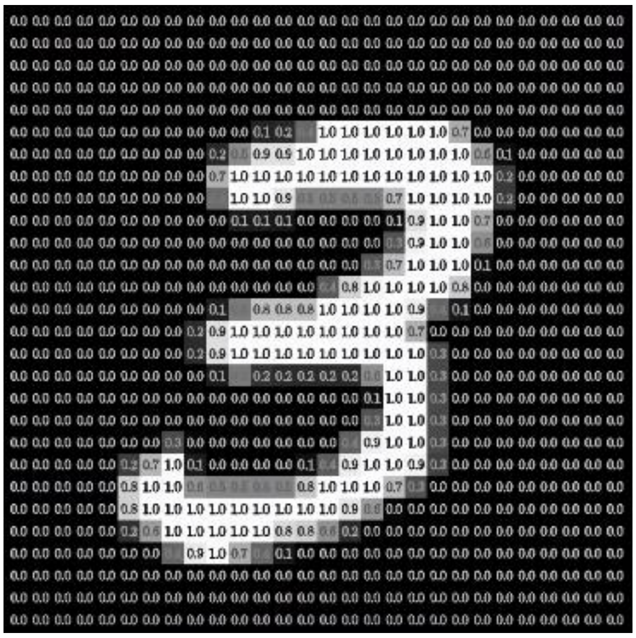


Figure 4: Matrix representation of a MNIST examples

First, we flatten our image into a 784-pixel vector. This vector will be the input of our neural network. In this example we will use the simplest neural network architecture in order to later be able to understand convolutional neural networks:

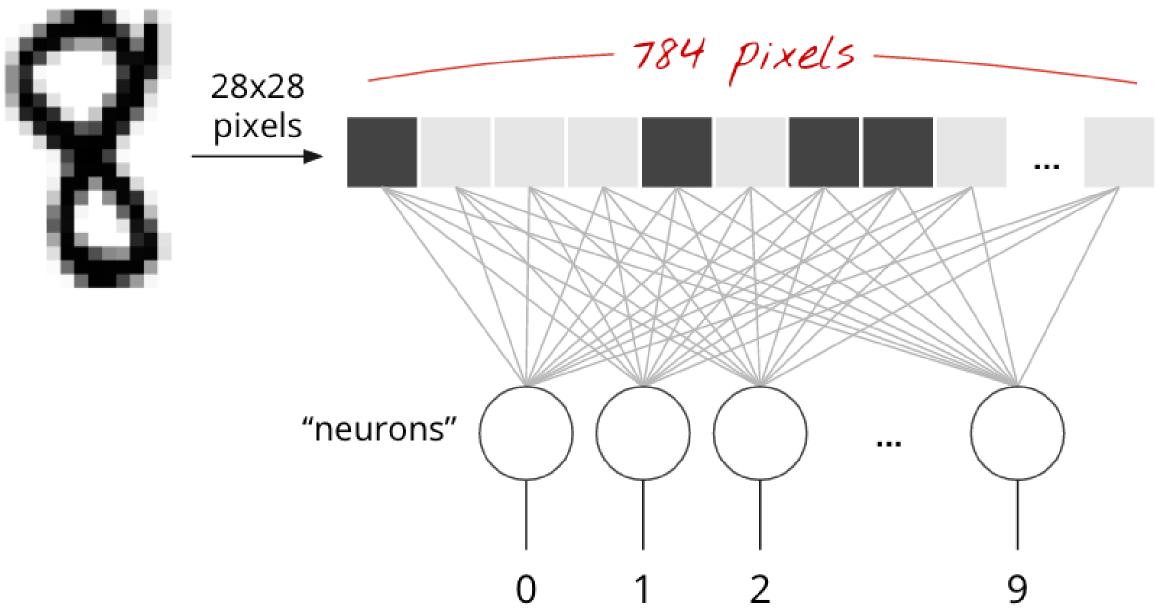


Figure 5: The neural network architecture that we will use to classify handwritten digits

In the above image, we can see that each "neuron" is connected to every single input pixel. There are ten neurons because we are classifying digits, from 0 to 9. The next logical questions are: What are these "connections"? What do these neurons actually do?

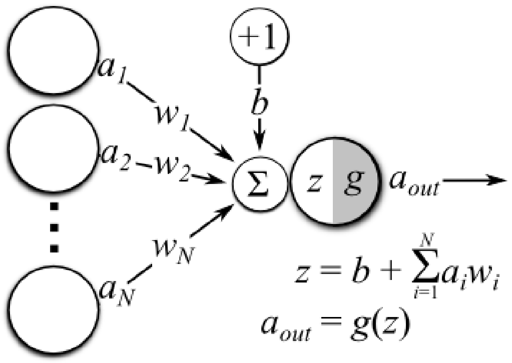


Figure 6: How a single neuron (also called a perceptron) work

Neurons are the building block of neural networks. In the image above a_1 to a_n are also neurons, but for the sake of explanation we will focus on the one on the right. Every neuron acts the same way:

- It does a weighted (w_n in the image) sum of all of its input values (a_n) in the image
- Then adds a constant, that is called a bias(b)
- And finally applies an activation function ($g(z)$) to the result (a_{out} in the image)

The output is called the activation of the neuron. The activation function is nonlinear so that it introduces non-linearity into the network. If we were to use a linear activation function, since the network is only composing linear function with a linear function, the output would always be linear. But, as we all know, the world is not always linear. The function that we will use in this example will be the sigmoid activation function. The output of this function is always between 0 and 1, which can be interpreted as probabilities.

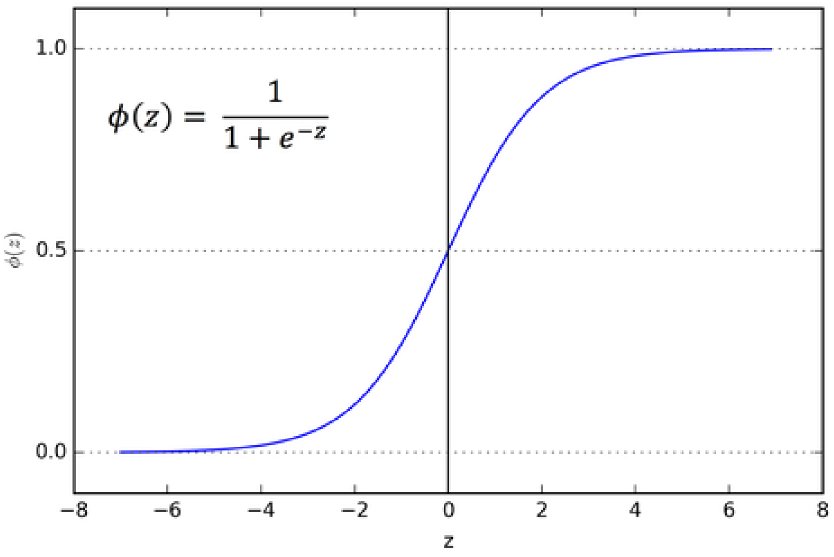


Figure 7: An example of activation function, the sigmoid function

To recapitulate:

- Flatten the image into a one-dimensional vector.
- Each neuron does a weighted sum of all of its inputs, adds a bias and then feeds the result into a nonlinear activation function.

- If we will have picked the right weights and biases, we can hope to see the correct neurons have a strong output, meaning an activation close to 1.

The output of each neuron can be interpreted as the probability that the network gives to the input image being this particular class. Our hope is that once we pass in an image of an 8, the 8th neuron will have a bright output. At the beginning, when we randomly initialize the weights, we will get random outputs. Here is where the training phase of building a neural network model comes in.

2.1.3 Training

As we understood in the previous section, the goal of training is to find the right weights and biases in order for the network to make correct predictions. If we see the network as a function, all the weights and biases of the network are parameters that can be tweaked in order for the function (our network) to output a certain result when a specific image is given. Knowing that we have 10 neurons that have 784 weights and 1 bias each, this adds up to a total of 7850 parameters.

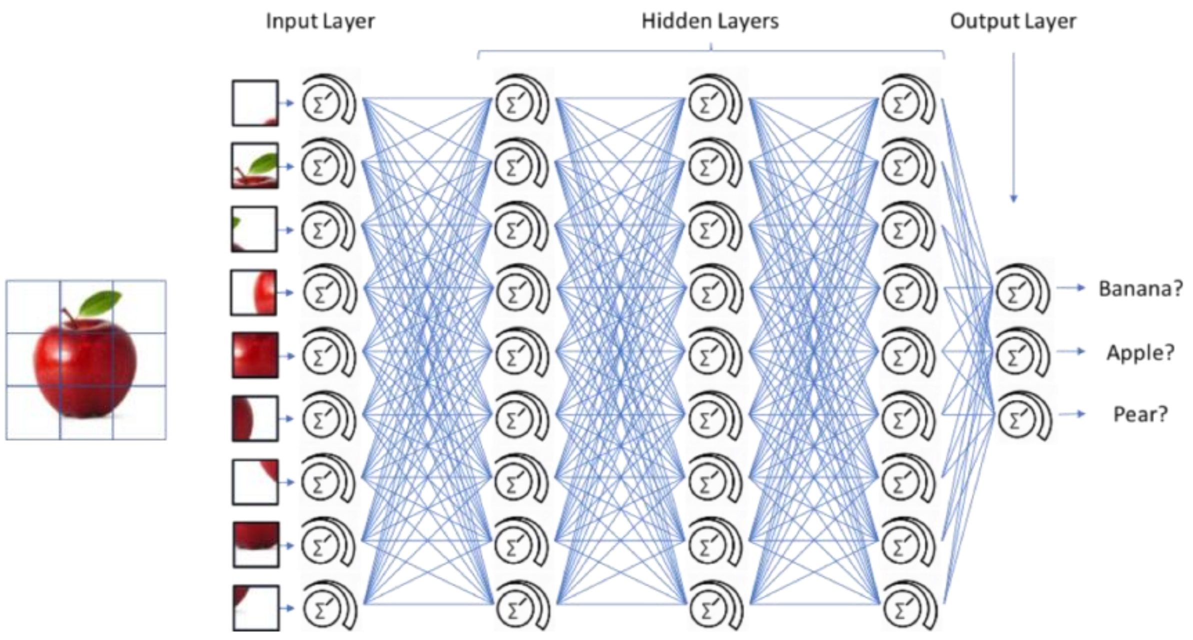


Figure 8: Analogy of a neuron being like a knob or dial

As an analogy, we can see these parameters as knobs and dials that we can tweak and turn to make this network behave in the way we want it to. In our case, we want it to take a 784-element vector and light up the right neuron. This architecture has very few parameters compared to other neural networks. For instance, in the architecture used in the proposed method, there are 60’000 parameters. Even 60’000 is not much compared to the state-of-the-art architectures with their 100’000’000 parameters.

We are now able to imagine how neural networks are capable to learn these extremely complex mappings between their input and output. All these parameters enable them to find the right series of computations that transforms an input (a vector in our case) into the appropriate activations of the output neurons. In a sense, neural networks search for algorithms that would take humans far too long to manually code into a computer. This exploration is not guided by the fingers of a programmer, but by raw data.

This insight is the one Alan Turing had in 1950, as Lê Nguyên Hoang explains it in *la formule du savoir* [14]:
"However, as early as 1950, Alan Turing predicted the future success of machine learning, even specifying that its emergence would take place at the end of the 20th century! Better still, Turing had pointed out the precise reason why machine learning would surpass man-made programs for many tasks: only machine learning is capable of exploring the space of algorithms whose length exceeds the gigabyte."

In order for the network to update the parameters it first has to know how wrong its prediction is, how far it is from the truth. That is where the cost function comes in. This function can be understood as computing the distance between the prediction and the actual value.

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \mathrm{f}(x^{(i)};\theta)\right)^2$$

Figure 9: Mean squared error loss

The equation above computes the cost for n examples. It takes a batch of images, gets the network's predictions and then computes the squared distance for every image in a batch of images. Finally, it takes the mean of all those squared distances. This function is called the mean squared error loss[1].

Now that we have the cost function, we can rephrase the goal of training to: finding the right weights and biases that minimize the cost function. This is also known as cost optimization.

[1] "cost" and "loss" can be used interchangeably

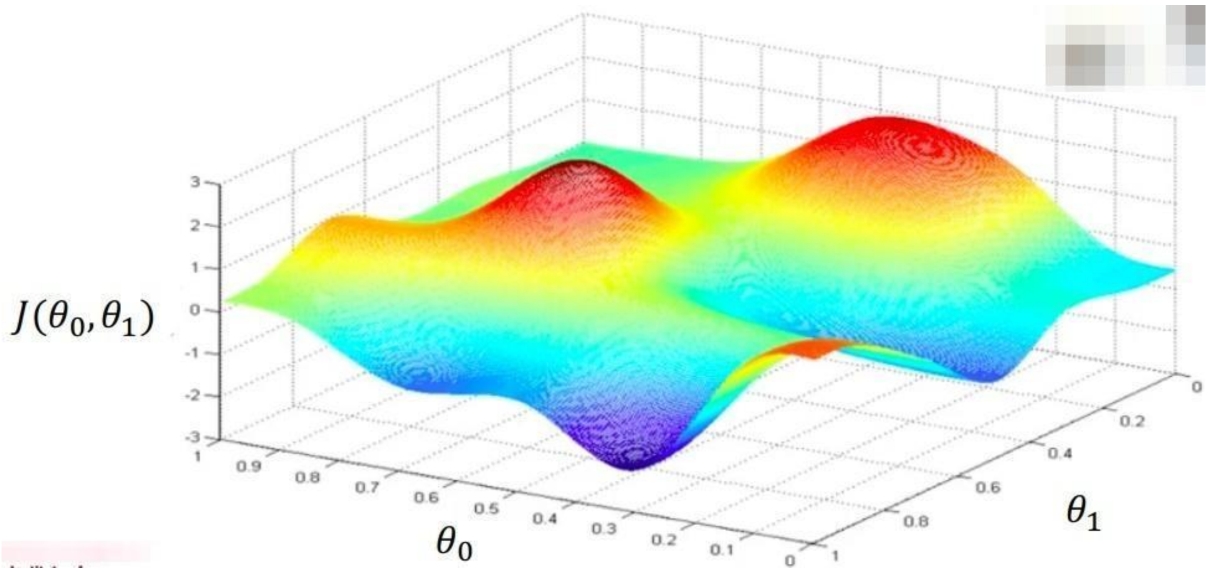


Figure 10: Graph of the cost function

As the network is only composed of weights and biases, the cost function is a function of the network's weights and bias. Now imagine we had only two weights θ_0 and θ_1. These are represented in the above graph on the x and y-axis. On the z-axis, we have the cost function of these two weights. We can visualize training as finding the local minimum of this terrain representing the cost function. So how can we do this? Well, it comes down to a calculus exercise.

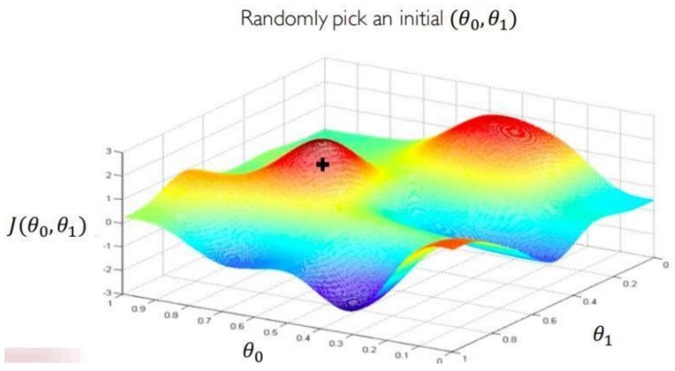


Figure 11: Randomly initialize the weights

First, we randomly initialize the weights and biases. On the graph, it is represented by a random point

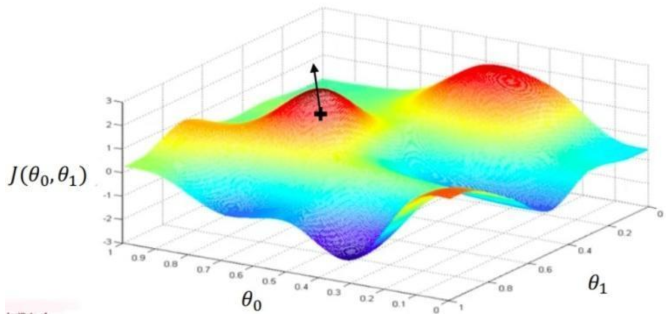


Figure 12: Compute the slope

We compute the slope of the cost function, which tells us the direction of steepest ascent.

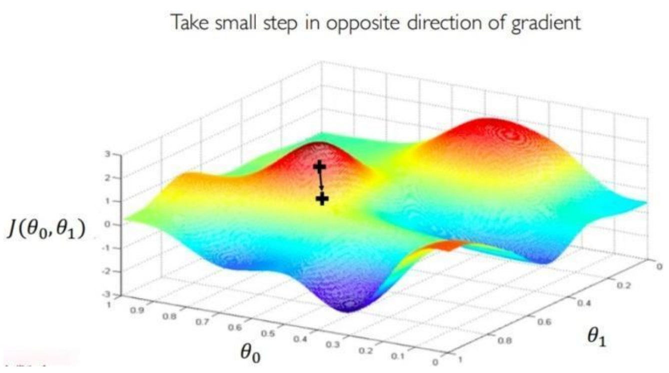


Figure 13: Step in the direction of where it goes down

Then take a small step downhill, which is the direction to step in that decreases the function most quickly.

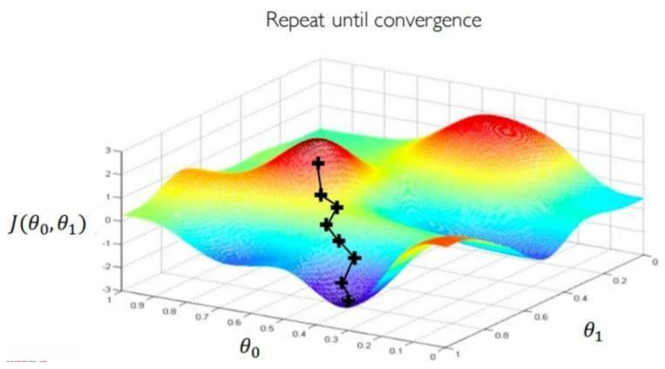


Figure 14: Repeat until convergence.

Finally, we repeat until the slope is close to null.

Coming back to classifying handwritten digits, it is the same as this but with 7850 rather than 2 weights and biases. The steps taken in the images above represent the update of the weights and biases. The length of these steps is called the learning rate and it is a hyperparameter that can be tuned to converge to a local minimum faster.

So, to recap training.

- We have the forward propagation step; we propagate a batch of images through the network. This means that we will be updating the weights and biases one batch at the time, not after every training example.
- We compute the current cost averaged over the batch, in our case we use the mean squared error loss.
- Do the backpropagation step. We compute the slope of that cost function which tells us for each weight and bias the nudge that will cause the most rapid decrease of the total cost. After that, update the weights and biases accordingly. The algorithm described here to find the local minimum of the cost function is called gradient descent.
- Repeat until convergence.

Remember, by minimizing the cost function we are minimizing the difference between the predicted value of our network and the true value. So, we will go from what seems to be random activations of the output neurons to only the right neuron having a strong output, i.e. an activation close to 1.

In order to implement this practice, we will use python with the Tensorflow library. The functions that Tensorflow provides will be used to do all the steps I mentioned above, building the architecture and then training it. All the computations that are made by Tensorflow, from forward propagation to computing the slope, are done by using and computing tensors. Tensors are “a generalization of vectors and matrices to potentially higher dimensions” [link] and that is why the library is called Tensorflow.

If we let it train for a dozen of minutes on a CPU, we get an accuracy of 98.22% [15][2]. This accuracy is already very good, but we can do better. The handwritten digits recognition task is an image recognition task where the spatial information of the pixels is important. In the field of image recognition, another type of neural network architecture has proven to be very effective. They dominate the ImageNet competition [16] since 2012. This type of neural network architecture is called a convolutional neural network, and it is the one I will be talking about in the next section.

2.1.4 Deeper networks

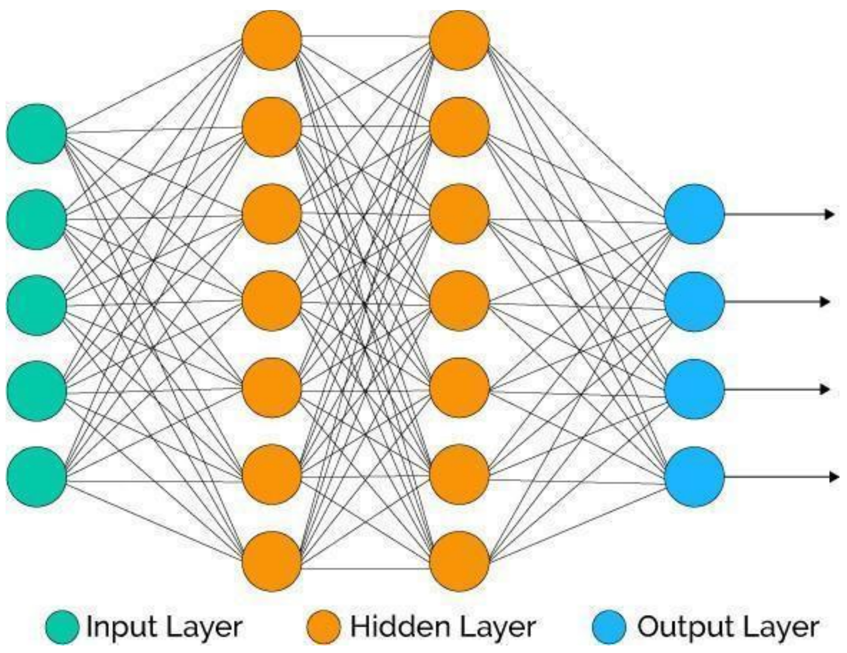


Figure 15: Example of a deeper network

Our architecture to classify handwritten digits only had one input layer and one output layer, but we can easily add layers in between as shown in the image above. These additional layers between the input and the output layer are called *hidden layers*. These layers of neurons act the exact same way as the output neurons in our architecture. But the second hidden layer and the output layer now take the activation of the neurons of the previous layer as input.

[2] The code can be found on my GitHub repository

Adding hidden layers enables the network to learn more complex mappings between the input layer and the output layer. The point of diminishing returns when adding hidden layers, or adding neurons in general, is when the network starts simply memorizing, in its parameters, all the examples of the dataset. This is called overfitting, and this dampens his ability to generalize when faced with new examples that it has never seen before. For example, if we were to add 10 hidden layers of 50 neurons each in our handwritten digit recognition architecture, while keeping a dataset of only 70’000 images, the network would after a certain training time effectively store all the training examples in its weights and biases. When faced with a novel image it would not perform well because it is only able to recognize the ones in that it has stored. At the opposite end of the spectrum, if we were to put too little hidden layers in a more complex task, the network would not be able to “capture” the necessary complexity of the task. It would be like trying to approximate a 5th degree polynomial with a first-degree polynomial.

2.2 Convolutional neural networks

This section is based on information from the following sources [17]. Convolutional neural networks are very similar to simple neural networks (also called multilayer perceptron) that we discovered in the previous section. The training and the cost function work exactly the same way. The first noticeable difference that can be seen is that the input is not flattened into a one-dimensional vector, instead, we keep our image as a matrix. The main difference is that instead of flattening the image into a one-dimensional vector, the convolutional neural network keeps the image as a matrix. It does achieve to do a weighted sum of its inputs, just as a simple neural network, by using convolutions.

2.2.1 Convolutional layers

A convolution is the application of a filter to an input that results in an activation. This filter, a matrix slides across the input which results in a map of activations called a feature map. This map indicates the locations and strength of a detected feature in an input image. Different filters detect different features in an image. For example, the filter below detects horizontal edges that have white on their right (detected by the ones), gray in the middle (detected by the zeros) and black on their right (detected by the minus ones). These filters are also “learned” and it just so happens that they turn out to be edge-detection filters and such after training. As shown in the image below, at each step we do a element-wise product between a 3x3 window of the original image and the our filter and store the result in our feature map.

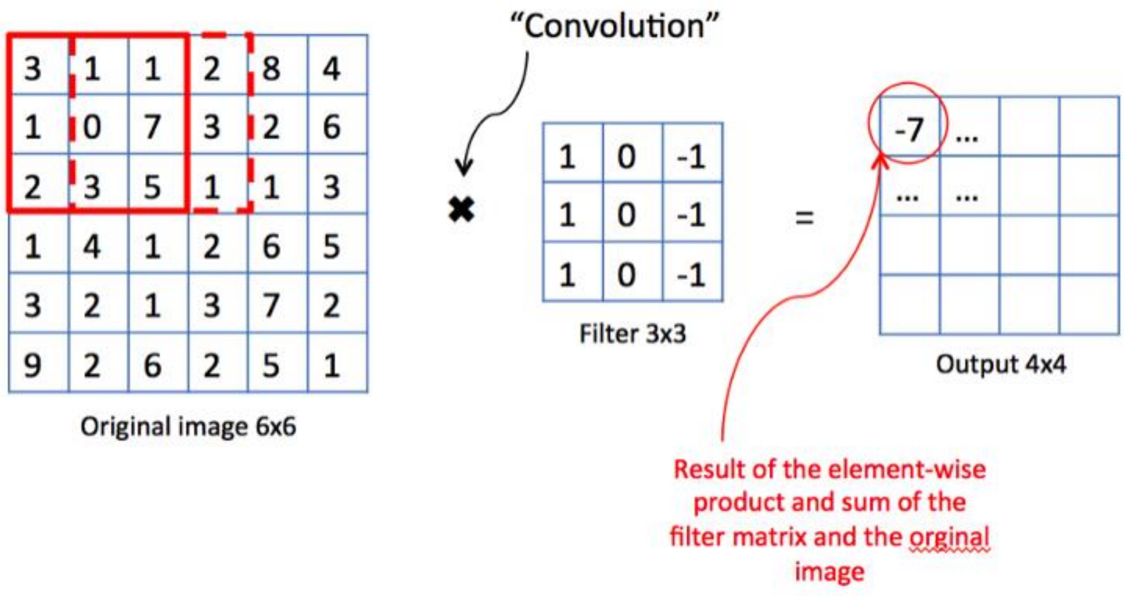
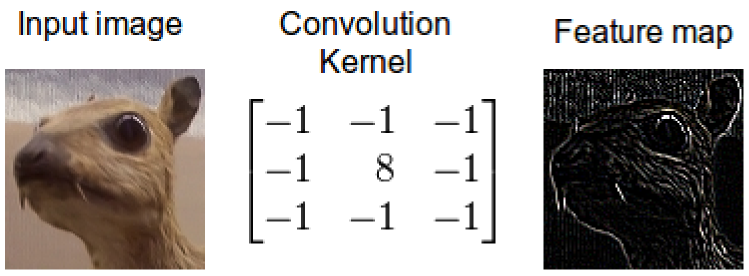


Figure 16: How to convolve an image

In the above image, the -7 is obtained by doing a element-wise product between the filter and the first red square and then doing the sum of all the values of that result. The size of the matrix is a choice that the designer of the architecture has to make. A convolution is often interpreted as a filter, where the filter searches for information of a certain kind and discard other information. It detects and highlights a specific feature in the input image. For example, in the image below, we have the input image that is convolved with the filter (also known as the kernel) that highlights edges. The output is the feature map on the right, which can also be interpreted as an image.



$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Figure 17: Example of a convolution

In the first convolutional layer, the filter may pick up on edges and such low-level features. The middle layers will combine these low-level features to detect mid-level features such as curves and corners. Finally, in the last convolution layers, filters detect combinations of the previous mid-level feature to finally detect high-level features such wheels, legs, and faces. In general, each layer performs a combination of the filters of the previous layer in order to be able to detect a higher-level feature as we can see in the image below. For example, if we had in the first layer of a CNN a filter that detects horizontal lines and one that detects vertical lines, a filter of the next layer could do a combination of the two in order to detect corners.

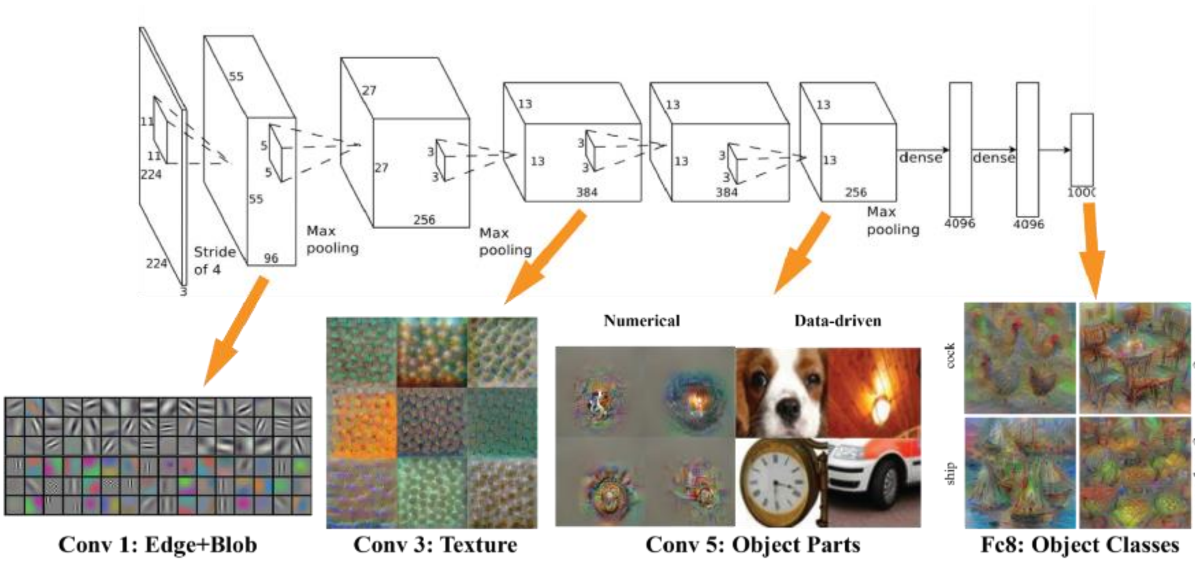


Figure 18: Visualization of what each layer of a CNN learns

In the image above, you can see a schema of a convolutional neural network. The images at the bottom are representations of what the filters detect at that particular layer. For example, the filters of the 3rd layer seem to detect texture, whereas the filters of the first layer seem to detect simpler patterns such as edges and corners. The 5th layer also shows, on the 4 images on the right, example images of what would be strongly detected by the 5th layer. Convolutional neural networks have weights, just as the multilayer perceptron that we used to recognize handwritten digits. Filters of a CNN are matrices of weights and therefore can be learned during training. For example, in the architecture of LeNet-5 that we will use in the proposed method, the filters are 2x2 matrices. The first convolutional layer is composed of 6 filters and the second layer is composed of 16 filters. Between the two, there is a pooling layer and this will be the subject of the next section.

2.2.2 Pooling layers

Pooling layers are often placed between convolutional layers. They down sample the feature maps of the previous layer. They output a low-resolution map which, for each area in the image, tells us the most present feature in that subset of the image. This makes the resulting smaller feature maps more translation invariant, meaning robust to changes in the position of the feature within the image. Unlike convolutional layers, no parameters of the pooling layers will be learned. They are all part of the architecture of the CNN.

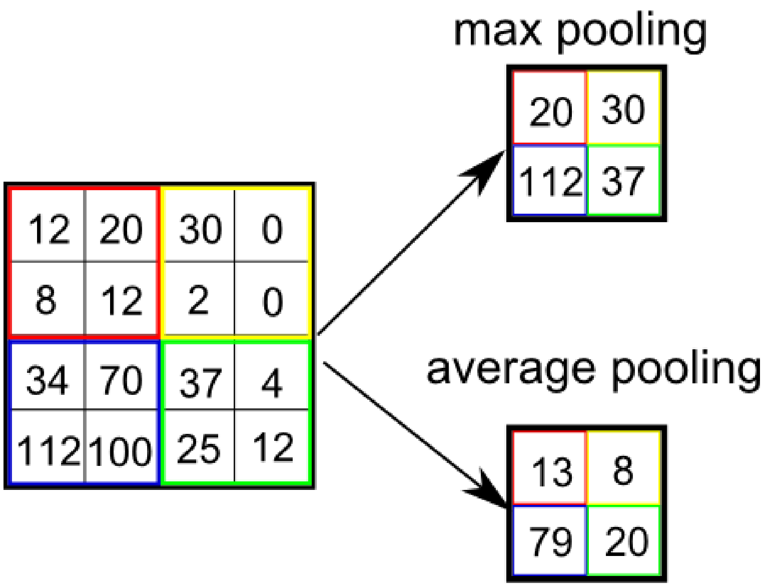


Figure 19: Various pooling methods

There are two main pooling methods average pooling and max pooling. Average pooling summarizes the average presence of a feature in a subset of the image. As shown in the image above, it calculates the average value for each subset of the feature map. Average pooling layers will be used in the proposed method. Max pooling summarizes the strongest feature, it keeps the maximum value for each subset of the feature map.

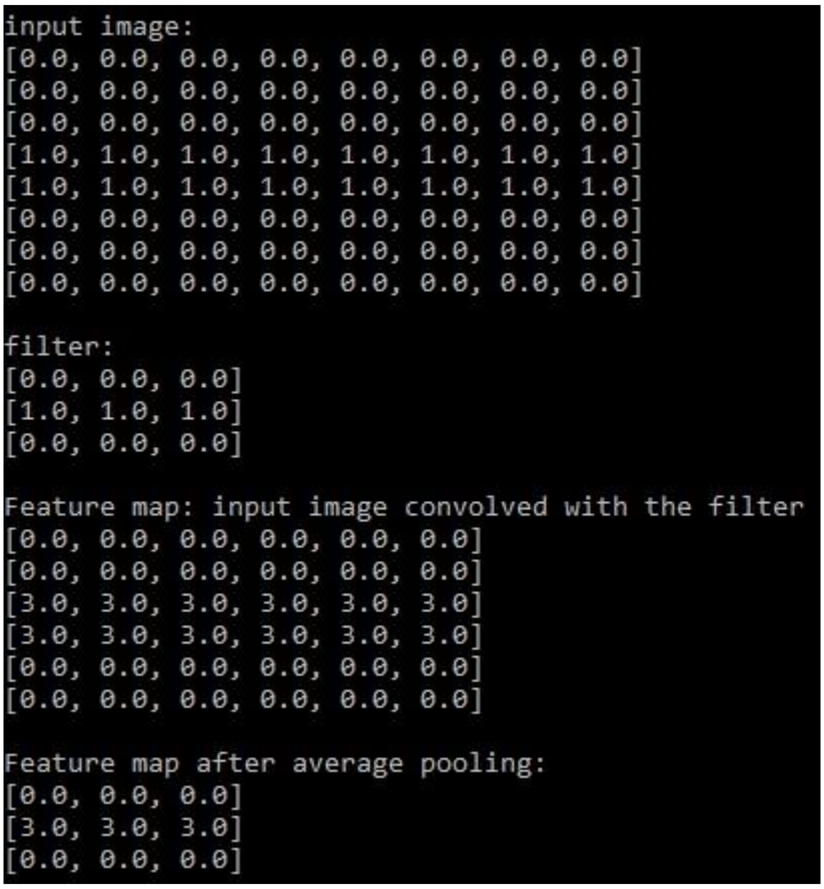
2.2.3 Multilayer perceptron

A convolutional neural network often ends with a multilayer perceptron, just as the one we used to classify handwritten digits. After the last pooling layer, we flatten our volume (composed of all the feature maps stacked on top of each other) into a one-dimensional vector in order to pass it to the multilayer perceptron. Just like the one used in our handwritten digit recognition task, the activations of an output neuron represent the probability that the network gives to the input image being this particular class.

The multilayer perceptron looks at the output of the previous average pooling layer (which represents the activation maps of high-level features) and determines which features correlate the most to a particular class. For example, if one of the output neurons classifies whether or not the image has a bird, this neuron will have a strong connection to the feature map that detects beaks, wings, etc.

2.2.4 Convolution and pooling example

In the image below, we have an 8x8 image that is convolved with a horizontal line detector. This results in a feature map where the horizontal lines have strong values. Then, just as in a convolutional neural network we do an average pooling on this feature map. Each value in the output of the average pooling is the average of a 2x2 window of the input feature map. The convolution has successfully highlighted the horizontal lines and then the average pooling has downsampled the feature map by summarizing the presence of a horizontal line in the input image. Thanks to this program we have detected and summarized the presence of a horizontal line in our input image.



```
input image:
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

filter:
[0.0, 0.0, 0.0]
[1.0, 1.0, 1.0]
[0.0, 0.0, 0.0]

Feature map: input image convolved with the filter
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[3.0, 3.0, 3.0, 3.0, 3.0, 3.0]
[3.0, 3.0, 3.0, 3.0, 3.0, 3.0]
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

Feature map after average pooling:
[0.0, 0.0, 0.0]
[3.0, 3.0, 3.0]
[0.0, 0.0, 0.0]
```

Figure 20: Convolution layer and pooling layer example

3 Implementation

I have programmed all of this project in python. Python is the language in which I studied machine learning. I had chosen it as python is the fastest growing language in the machine learning community and it has lots of great libraries such as TensorFlow (machine learning), NumPy (math), OpenCV (image processing), etc...

As said in the introduction, the coin recognition task will be divided into two tasks:

1. Localizing the coins in the image by drawing a bounding circle around them. Then make individual images out of each bounding circle and pass them to step 2.
2. Taking the images of step 1 and classifying them in the appropriate category (2fr, 1fr, etc.).

For the first task, localizing coins in an image, I have chosen to use the circle Hough Transform as it is the popular algorithm for circle segmentation. I have used the OpenCV implementation of the circle Hough Transform.

For the coding and training of the CNN, I have chosen TensorFlow because of its many advantages. TensorFlow is an open source library released by Google in 2015 to build and design machine learning models. Google uses it in whenever they need a machine learning model (which means in every product...). Tensorflow was built to be used at a large scale and is therefore very efficient. It is able to run on multiple CPUs, the central processing units which perform most of processing in a computer, and GPUs, the graphics processing units which are optimized to work with matrices. It is even able to run on mobile operating systems. Tensorflow models, once trained, can be much more easily used in a web-site or application than if the models had been coded in another library such as PyTorch or scikit-learn.

3.1 Circle localization algorithm

The function “cropping_algo” in the program has the task to take the raw input image and find the x and y coordinate of each center as well as the radius of each circle. This function will output these pieces of information (x, y, radius) as well as their corresponding cropped images. This matrix will serve as the input of our neural network.

In order to find the centers and the radiuses, the “cropping_algo” starts by resizing (while keeping the initial proportions) the image so that the biggest side is 500 pixels long. The Houghcircles function of OpenCV does not require a high-resolution image in order to find the bounding circles which also reduces the computation time needed.

The image is now ready to be applied the circle Hough Transform. The circle Hough Transform in OpenCV requires 5 parameters. These have been set by taking a random image of my dataset and simply trying which combination of parameters yield the best results. In a nutshell the parameters are, what is the minimal distance between the centers? How tolerant do you want the edge detector to be? Threshold for center detection? What is the minimal and the maximal radius you want to be able to detect?

Before applying the circle Hough Transform, the OpenCV function applies the Canny edge detector. The Canny edge detector, as his name suggests, transforms the image into an image only composed of the edges. This preprocessing step is required before applying the circle Hough Transform. In the next two sections I will attempt to convey an insight on how the circle Hough Transform works after having given an overview of what the Canny edge detector does.

3.1.1 Canny edge detector

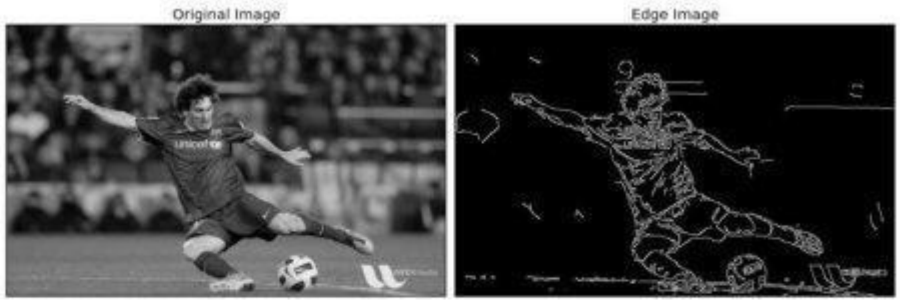


Figure 21: Canny edge detector example

The Canny edge detector uses kernels convolutions(filters) as we saw in the convolutional neural network section. First, it uses a Sobel operator which outputs an image where each pixel tells us the intensity of the edge at this point. After that, the Canny edge detector keeps only the dominant edges through a process called hysteresis thresholding, which favors the edges that are connected to strong edges. For more information the YouTube channel “computerphile” has done a beautiful job at explaining the Sobel operator [18] and the Canny edge detector [19].

3.1.2 Circle Hough Transform

This section is inspired by the explanation given in the following source [20]. The circle Hough Transform takes as input the output of the Canny edge detector, i.e. a binary image (edge/not edge) only composed of edges. In order to find circles, the algorithm needs, as we will see next, to know what radius it should be looking for. Unfortunately, we often do not know the exact radius. Therefore, the circle Hough Transform will simply sweep between the minimum radius and the maximum radius and run the algorithm described below. Now, how does it identify circles knowing their radius?

The intuition for the algorithm is quite intellectually pleasing and I do not believe it requires any mathematics to be understood. We start with an image (this image is very simple for explanation purposes):

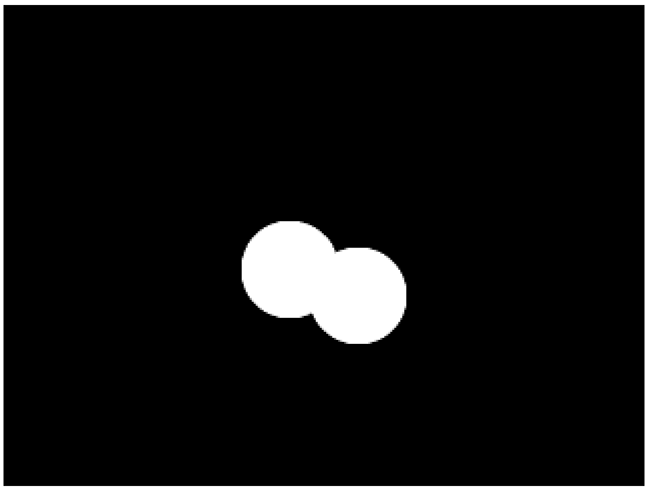


Figure 22: Simple image containing circles

At which we apply the Canny edge detector that detects the edges and generates a binary image:

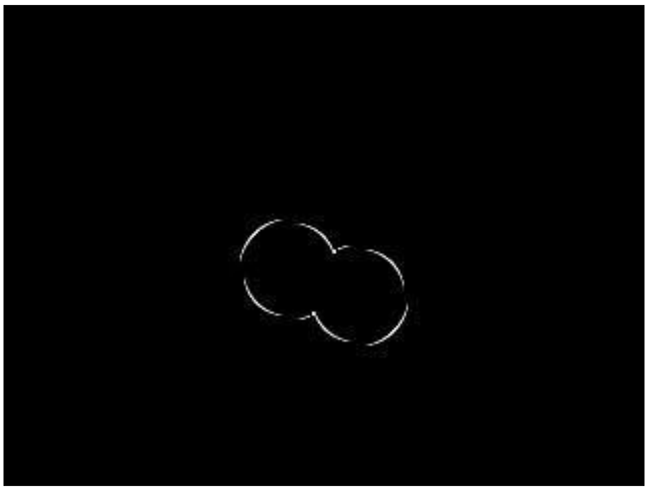


Figure 23: figure 25 at which we applied the Canny edge detector

Then imagine that every “edge” pixel (every white pixel in the image above), draws a circle of the specified radius around him. As this pixel thinks it is on the edge of a circle, it votes that the center must be somewhere on this circle it drew. Once all the pixels in the edges have done the same procedure, the cells with the greater number of votes are the centers. In the below image, three random points were chosen. Circles of specified radius are drawn around them (the red, blue and green circles). And then votes are cast at the pixels of these circles:

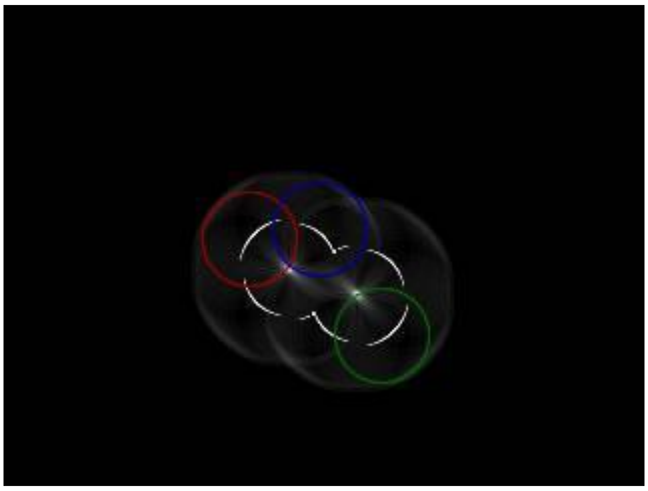


Figure 23: Image showing three random points and their votes

The program would then run the same algorithm with other radiuses within the specified intervals that the radius can take. Once this will be done, we will have found the circles of any radius within our set interval.

3.1.3 Results

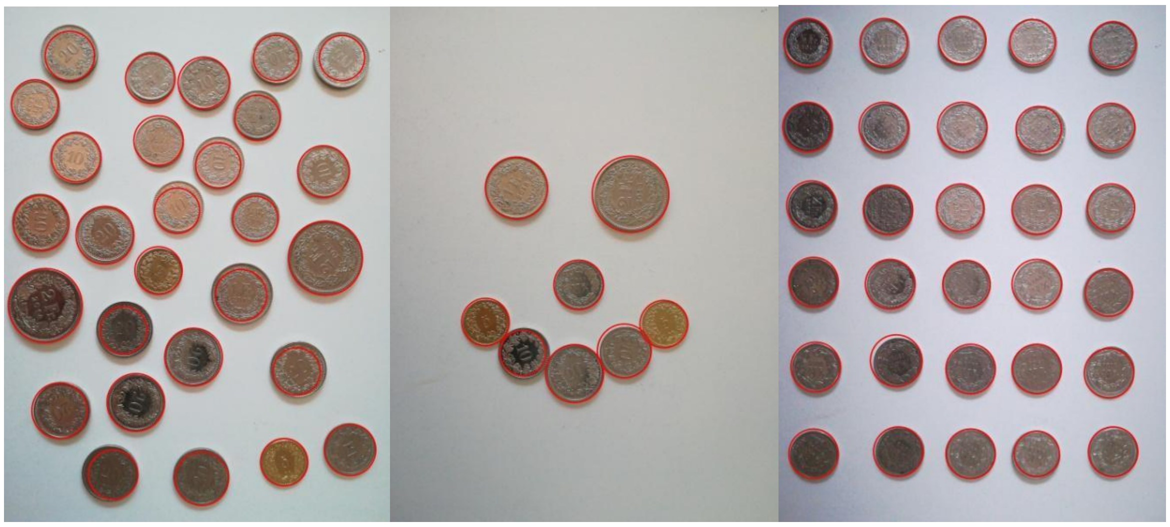


Figure 24: Results of the “cropping_algo” function

After a lot of fine-tuning to find the right combination of parameters that give good results for every type of coin, we can see in the images above that the results are quite good. The algorithm is able to localize all the coins. If we take a closer look at each image, we can see that some of the bounding circles drawn are overestimating or underestimating the size of the coins. This then results in a more zoomed in (or zoomed out) image of the coin when it is fed to the neural network:

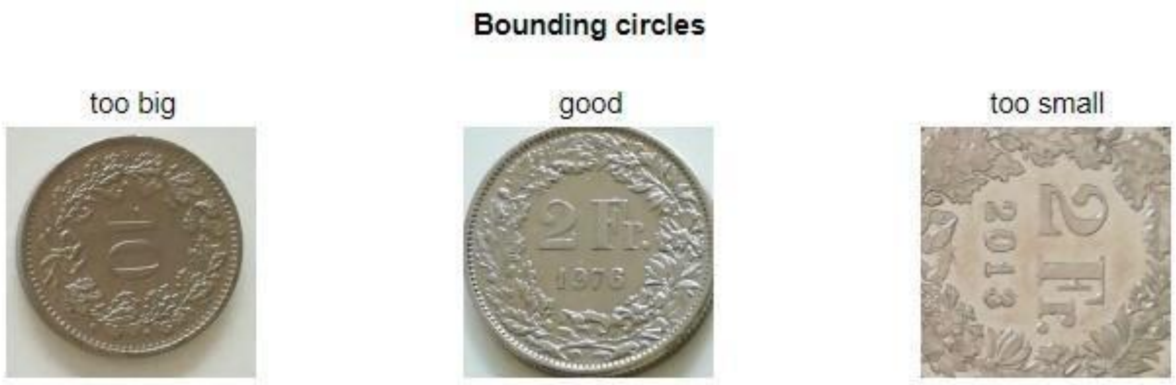


Figure 25: Visualization of overestimations and underestimations of the bounding circles by the “cropping_algo” function

When watching numerous examples, I noticed that the bounding circles of the small coins tend to be too big and the bounding circles of the big coins tend to be too small. This is not a major issue as it only means that it will be the neural network’s job to take this into account and come up with a way to be robust to changes in coin size.

With regard to the runtime, the “cropping_algo” function takes an average of one-tenth of a second to process a 2976x3968 image on a CPU Intel Core i5 and is insignificantly dependent on the number of coins in the input image. The runtime varies of +/- 0.02 seconds depending on the number of coins in the image. Surprisingly, the “cv2.HoughCircles()” function that actually finds the circles only takes 0.003 seconds to run!

What takes so much time then? What takes the most time (0.09 seconds) is loading and cropping the image. If we were to code it in C++, which is well known for its processing speed, we could expect a drastic decrease in the runtime.

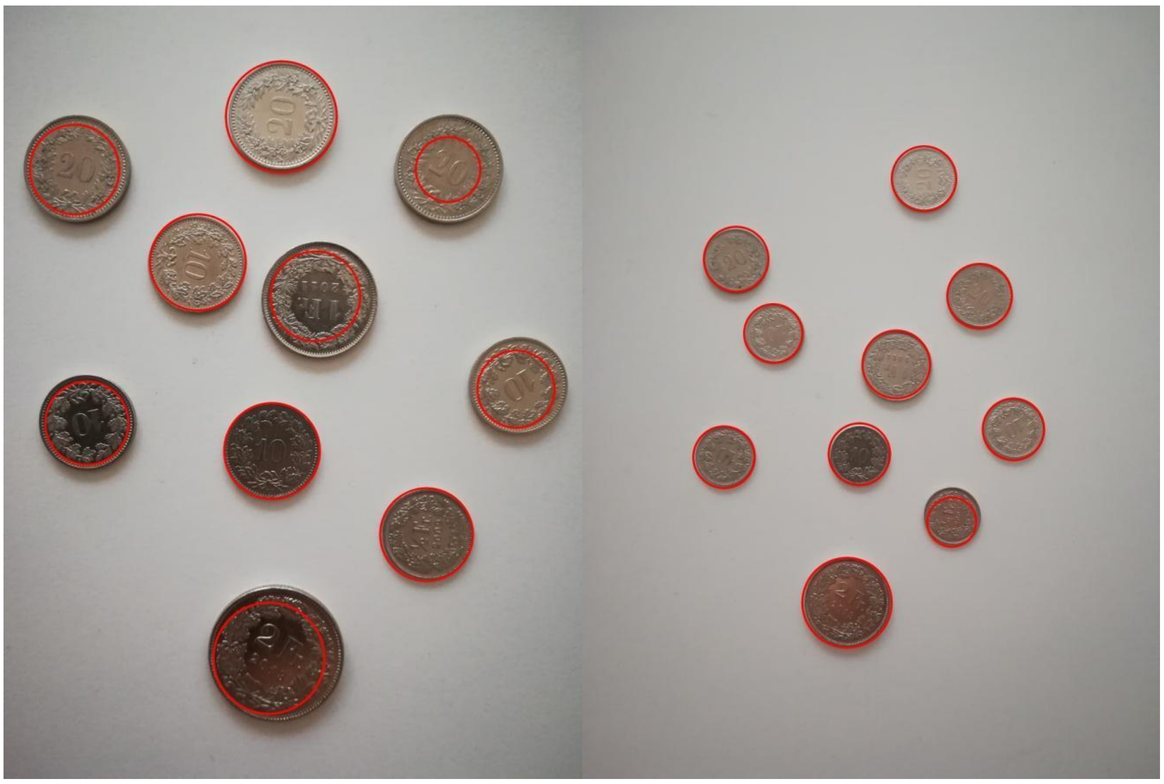


Figure 26: Results of the “cropping_algo” function when changing the distance to the coins

It is worth noting that our Houghcircles implementation is robust to changes in the distance to coins. The two images above show the results of the algorithm on a picture taken closer than normal and right on a picture taken further away than normal. As we can see, the algorithm is more robust to increases in the distance to coins than to decreases in the distance to coins. A hypothesis of why this could be the case is that the parameters setting of the circle detection algorithm is less suited to decreases in distance to the coins. The fact that by changing the parameters setting, I was able to have better results, as shown in the image below, supports this hypothesis.

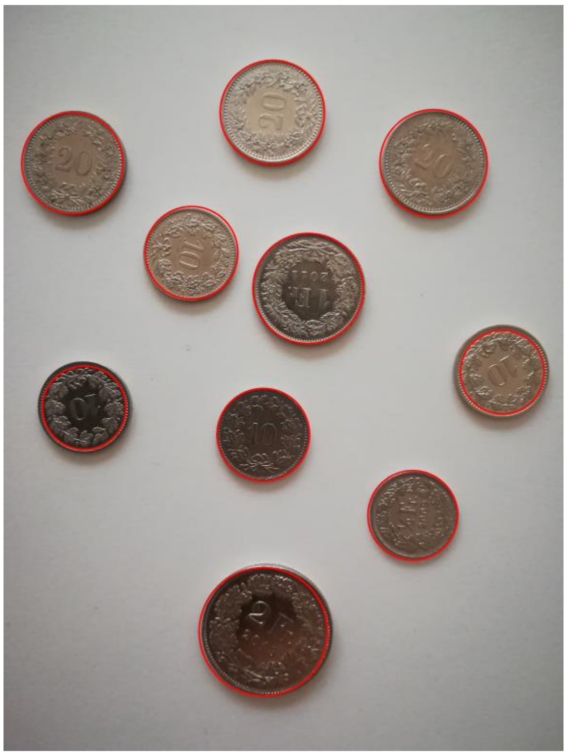


Figure 27: Results on a smaller distance to coin with changed parameters.

3.2 Coin classification

3.2.1 Data Collection

As told previously neural networks need data, and preferably lots of it. As I was looking for sophisticated ways to increase by 10x-40x an existing dataset, it was pointed out to me that It would be simpler to just take more images. I thought about it and came up with a method to gather a great number of images of coins using the cropping algorithm.

But first I must disclaim certain choices. I chose to bound the robustness of my overall method to changes in light, rotation and translation. Therefore, this method is not engineered to be robust to changes in distance to coins and background. Although as we will see, the dataset of the CNN will provide small changes in squale of the coins because of the fact that the cropping algorithm is not perfect in recognizing the exact radius of the coins.

The method used, helped gather 45’628 images of coins. Meaning approximately 5’700 images per category. And the method goes as follows:

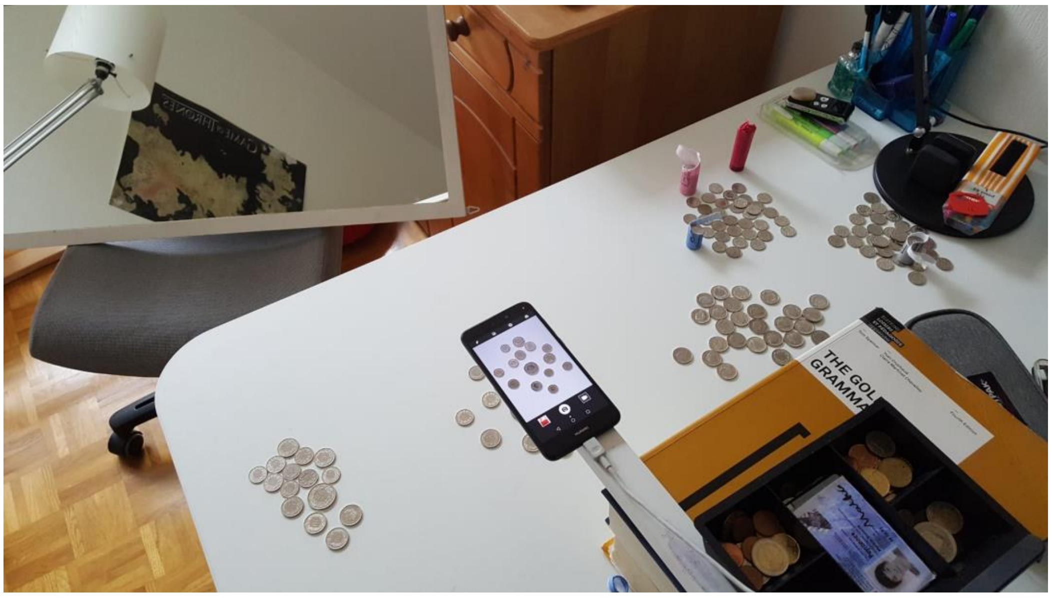


Figure 28: Setup used for data collection

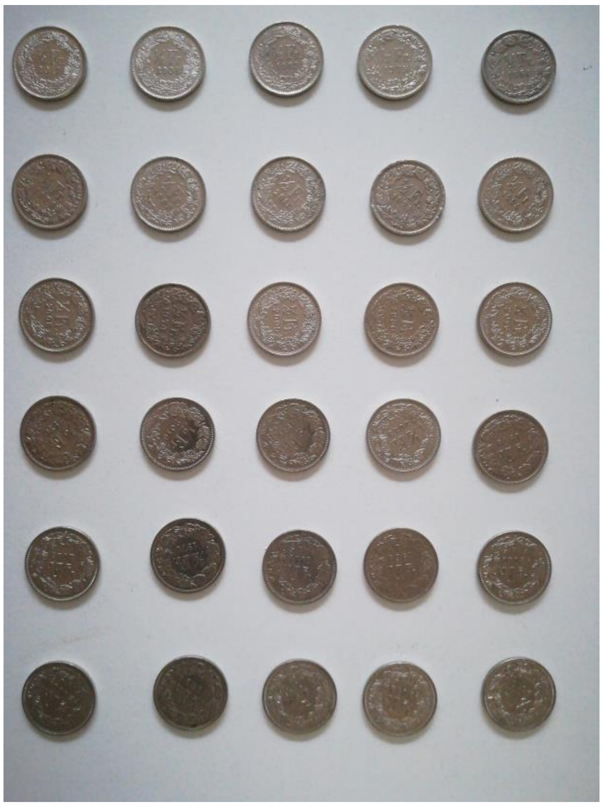


Figure 29: Example of image taken, before applying the cropping algorithm

First, I built the above setup. The coins (all of the same category) are arranged in order to form a 5x6 grid. This disposition will satisfy the need for translation. Furthermore, each coin is positioned with a different angle. This will satisfy the need for rotation. A camera (here, a cell-phone) must be placed above the coins.
Then the idea is to take a burst of photo (one every second), while continually changing the lighting. This is done with the help of mirrors and movable lighting. When 200 photos are taken, we can know repeat the steps with the seven other categories.

Once we gathered these images, we can know apply the cropping algorithm to each of them. It will segment 30 images of coins out of every image, leaving us with the finished dataset. The fact that we are using the cropping algorithm to generate our dataset is an advantage, as it makes the dataset closer to what it will face during test time. And its imperfect nature, means that the CNN will have to be robust to small changes in coins size, as we can see in the image figure 29. These changes in the scale of the coins are features that the CNN will have to learn to deal with.

We know have a dataset of 45’628 images showing coins with changes in lighting, rotation and translation. The next step is choosing the right CNN architecture to hopefully be able to, ounces trained on our newly acquired dataset, successfully classify novel coin images.

3.2.2 Convolutional neural network architecture

While trying to choose which neural network architecture, of the ones proposed in the introduction, would be the most appropriate for the task at hand, I looked at the problems each of these architectures had been designed for. The AlexNet and the VGGNet were designed to perform at the ImageNet competition. Therefore, they were engineered to classifying between 21 thousand classes of complex objects (camping tents, container ships, Madagascar cats), whereas my task require only to distinguish between 8 classes of medium complex objects (patterns on coins).

On the other hand, LeNet-5 architecture was designed to classify between 10 classes of medium-low complex objects (handwritten digits) and takes as input 32x32 images.

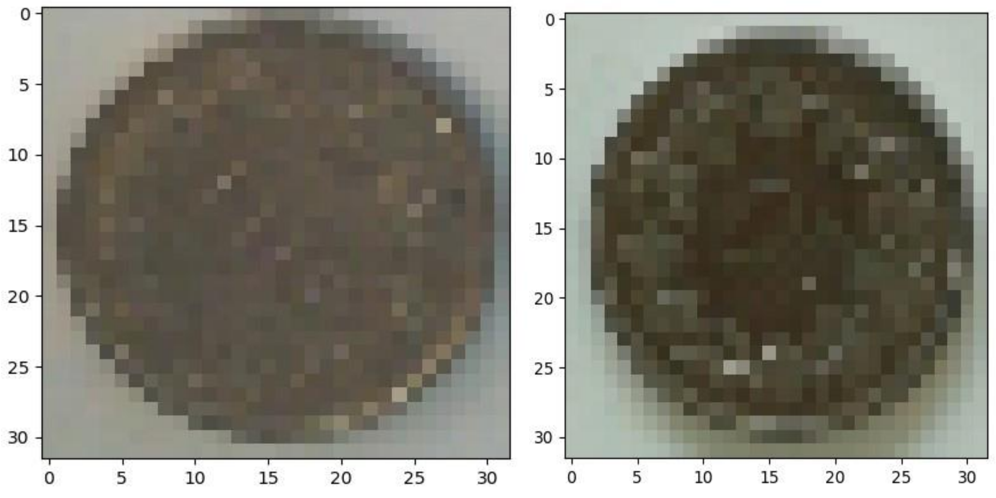


Figure 30: 32x32 images of coins.

As shown in the image above, 32x32 images of coins do not seem enough to be able to identify them. Therefore, to play on the safe side, I choose to modify slightly the LeNet-5 architecture. First, I choose 8 neurons for the output layer instead of 10 because we have 8 different coins (2fr, 1fr, 50rp, 20rp, 10rp, back of fr, back of rp).
Secondly, I have chosen to give 64x64 images as inputs. As we can see in the image below, contrary of 32x32 images, 64x64 images are identifiable.

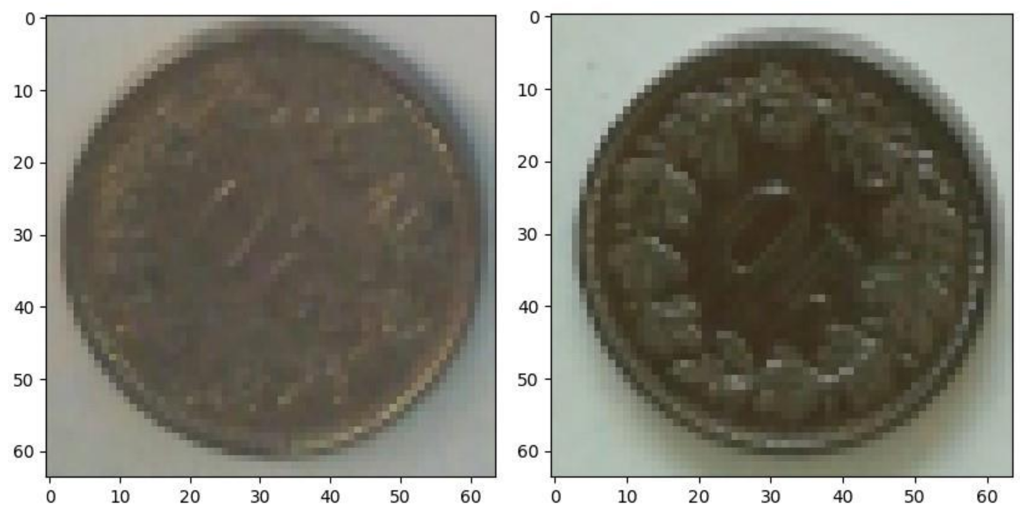


Figure 31: 64x64 images of coins.

In order for the network to accept 64x64 images, the 120 neurons of the first fully connected layer will be linked to 2704 neurons instead of 400.

4.2.3 Training

First, I resize the images into 64 by 64 images, for the reason's explained in the previous section. At this stage, I have a dataset of 45'628 images. Standard practice in machine learning is to then split the dataset into a training set and a test set. The training set, as the name suggests, is the set of images that will be used to train the neural network. The neural network will try to have the best accuracy on this particular set of images. Then the test set, a set of images that is left on the side, will not be used during training. Instead, once the neural network has been trained on the training set, we will test its performance on this test set that it has never seen. The network's accuracy on the training set is crucial information, as it tells us how well the network is able to generalize.

If the training accuracy is significantly higher than the test accuracy, this means that the neural network is overfitting the training set. Meaning, as explained in section 2.1.4, the model has, metaphorically speaking, memorized the background noise of the training set instead of only picking up the signal. But if the test accuracy is close to the training accuracy, the model has successfully extracted the signal out of the background noise. The training set has often the most examples with 80-99% of the total dataset depending on the size of the the total dataset. I have chosen 90% as my dataset is relatively small.

With regards to the hyperparameters, I started by rough guesses and then, from there, started to tweak them aiming at a good test accuracy. This took a while as training often takes between 10 and 16 hours. Through this process I started with a test accuracy that looked like the image below.

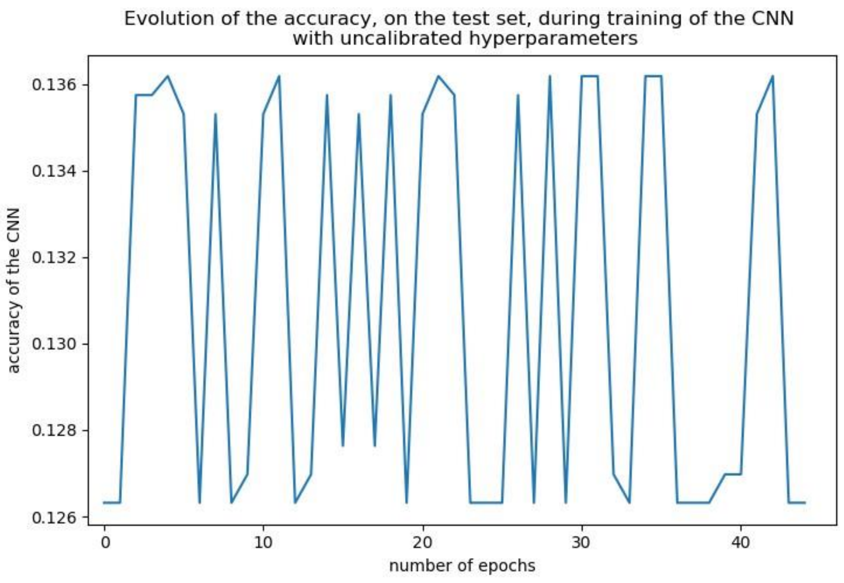


Figure 32: Evolution of the accuracy, on the test set, during training of the CNN with uncalibrated hyperparameters

Epochs, are the number of times that the network went through the whole dataset. As we can see in the above image, the loss optimization algorithm does not converge. Instead it is moving erratically on the optimization landscape because of it's too large learning rate. But with some hyperparameter tuning, I ended up with a test accuracy that looks like the image below.

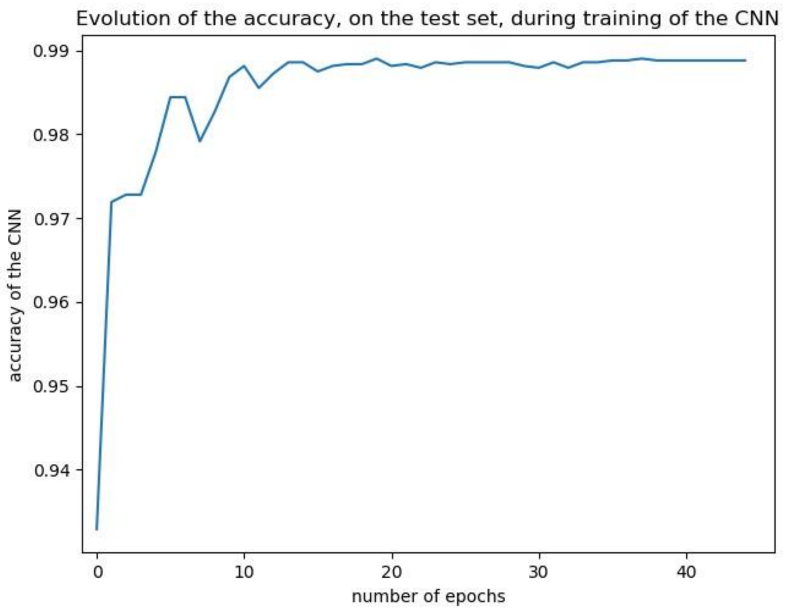


Figure 33: Evolution of the accuracy, on the test set, during training of the CNN

This is remarkably better, the loss optimization algorithm seems to have converged to a local minimum. It has reached an accuracy of 99.1% on the training set and an accuracy of 98.5% on the test set. This difference between the training and test accuracy is small enough to not be concerned about overfitting. We can see that the graph above resembles a logarithmic function. There is a lot of variation at the beginning and then gradually tends towards an asymptotic limit. We could interpret this graph by saying that the loss optimization algorithm

finds a local minimum at approximately the 10th epoch and then hovers around it, getting closer and closer to it towards the end.
If we know look at the evolution of the loss during training.

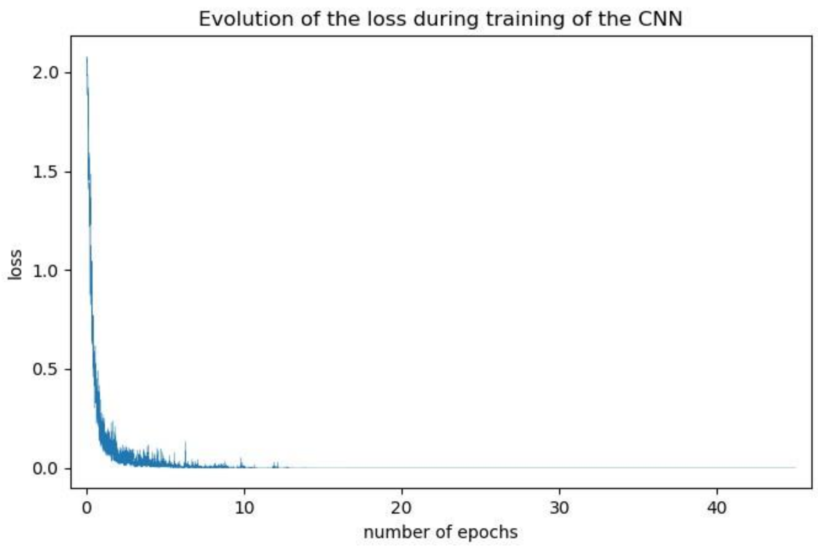


Figure 34: Evolution of the loss during training of the CNN

As in the graph of the accuracy, we see that the largest amount of learning is done in the first ~5 epochs and that the graph resembles a exponential function. We see that the loss seems to reach an asymptotic limit towards the end of training, most probably meaning that it has converged into a local minimum.

3.2.3 Results

Lastly, I programmed a central file to access the “cropping_algo” and the convolutional neural network in order to combine the two parts into a functional whole. The image below is an example of output that this program can produce.

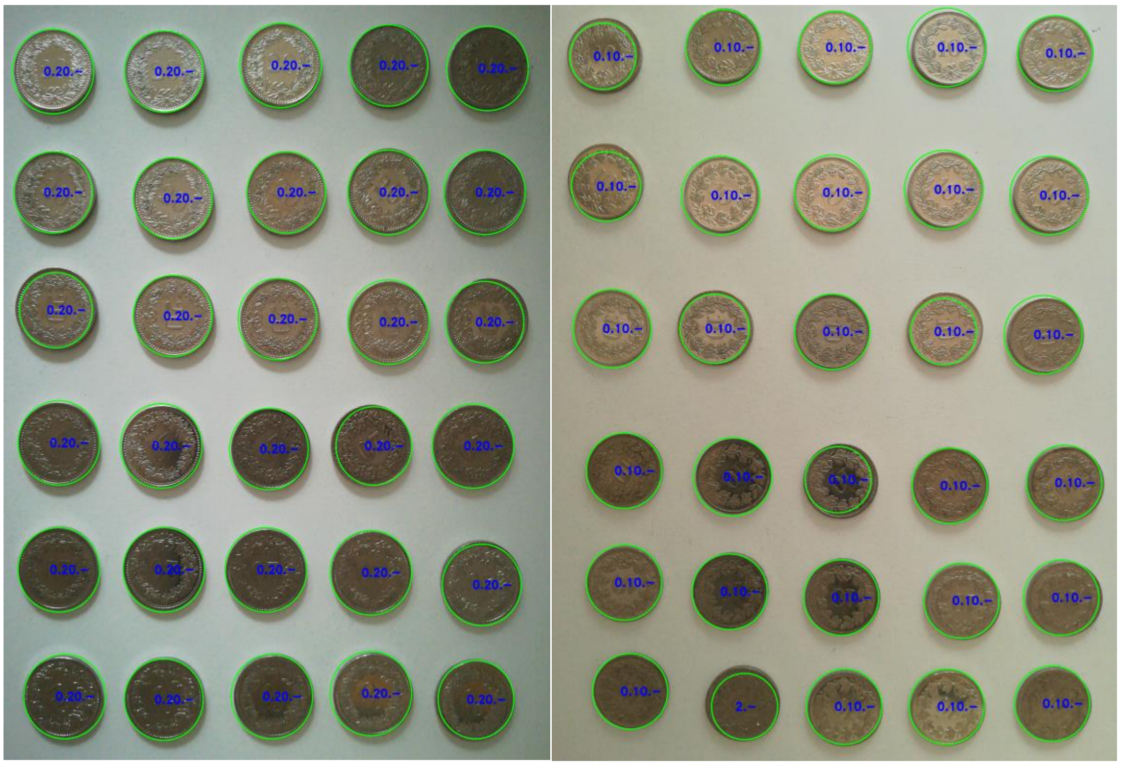


Figure 35: output example of the final program.

Although the 98.5% accuracy of the CNN on the test set could make us think that this accuracy would transfer to any images of coins, this is surprisingly not the case. After using the program on numerous images, such as the one shown in figure 37, we come to roughly evaluate the accuracy of the program's coin classifying ability to be around 70-80%.

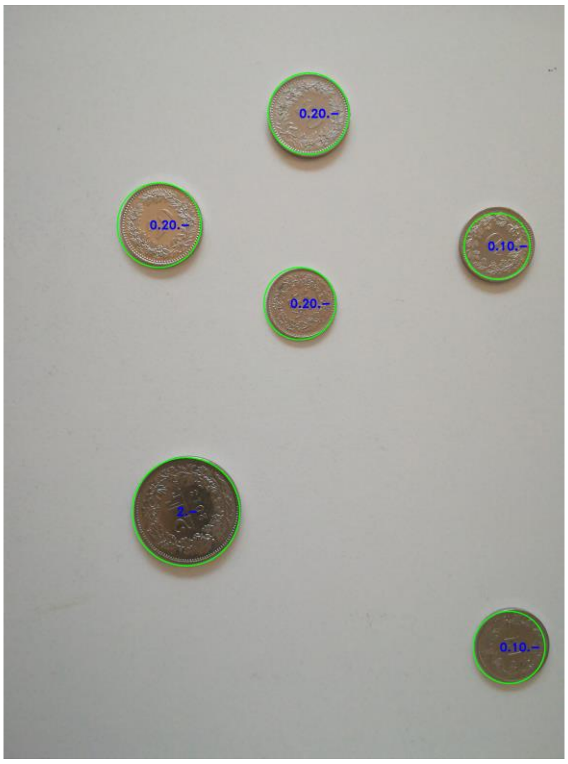


Figure 36: final program applied to a completely new image.

One hypothesis that would explain the difference between the accuracy on the test set and the accuracy with new images, is that the images of the test set are more similar to the training set than the new images in two ways. First the lighting, present in the new images, that would be commonly used is not the average lighting in my training dataset. Secondly although the coins in the training dataset are the exact same coins of the testing dataset, the ones used in the new images are novel coins that it has never seen. The CNN would have been able to generalize somewhat to coins which haven't been used in the training and testing dataset, but not perfectly.

The CNN has accomplished the task it was engineered to do perfectly. Meaning it has achieved a high testing accuracy while keeping a very narrow gap with the training accuracy. By my fault, the training dataset did not span the full diversity that the neural network would have needed in order to achieve a higher accuracy on new images. It has nonetheless a good accuracy on coins that it has never seen before. I am personally still amazed that it has learned to classify matrices of pixels.

If we try to picture what each filter detects by scaling the value of the weights from the range that they have to the range of colors (0 to 255) we get the images below.

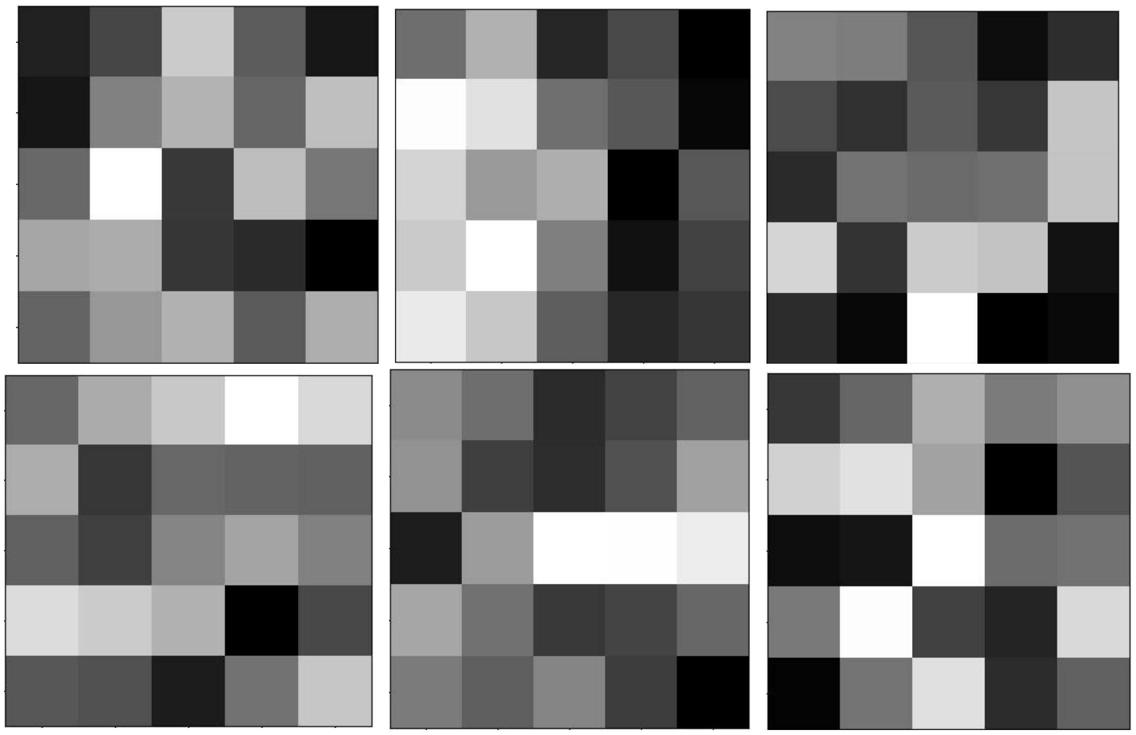


Figure 37: Representation of the learned filters of the first convolutional layer.

When we look at the six filters above, it is hard to understand what they are detecting as they are 5x5 filters. The interpretation of the filter get even more hard when we go to the second convolutional layer. In the second layer we have 16 filters, but they are now 5x5x6 volumes. This makes all attempt to interpret unthinkable.

4 Conclusion

To conclude, it can be said that the convolutional neural network has perfectly carried out the task it was engineered to do. But, of my ignorance, the training data was too dissimilar (lighting and coins) to the actual data the program would be used on. This leads to a 15% dope in accuracy between the testing dataset and novel coin images. I have therefor learned that, if we want a good accuracy on novel images, the CNN needs a training set which is perfectly representative of the data it will be used on. This is a very interesting observation as it is somewhat counter-intuitive. This begs the following question: What techniques and methods do machine learning teams use to make their dataset as close to reality as possible?

If we take the example of the "Vision For Robotics Lab" team at ETH which develop vision-based perception for robots, with a focus on small Unmanned Aerial Vehicles (UAVs). As Marco Karrer a PhD student part of this team explains, "getting the training data that your network needs is the hardest part essentially". Quality data is often the bottleneck of projects who heavily rely on neural networks. Finding training data which is close enough to the data the neural network will be used on is a real challenge. In every of their project, once they cannot acquire more/better data, they systematically use data augmentation. As Marco pointed out "[data augmentation] is a way to augment your existing training data so that your network is better able to generalize". At the start of one of their projects, he told me, they tried to train their CNN on the dataset that they had gathered. It performed very poorly. They figured out that they needed "more training data from a larger variety of perspectives". To do so they used specific transformation which, when applied to images of their existing dataset, produced the same image but as if we had taken it from another angle. This artificially augmented the variety of perspectives of the dataset. Thanks to these transformations, as Marco put it, "when we saw, okay, for this specific case we don't have enough data, we could just render more data to cover these cases". This enabled them to make their previous dataset closer to the data that their drones would encounter while in the air, which in turn produce far better results.

Coming back to my project, future work could consist of using data augmentation on the training data so that it could be made more similar to the actual data we want it to perform on. This could hopefully lead to a better accuracy on novel images and thereby free us of the hazard which is coin counting.

5 References

1. Safescan to count CHF: https://www.digitec.ch/en/s1/product/safescan-1250-fuer-chf-coin-counter-money-counters-5713830
2. Yann LeCun, Patrick Haffner, Léon Bottou, Yoshua Bengio, 1998, Gradient-Based Learning Applied to Document Recognition, Proceedings of the IEEE, 86. 2278 - 2324
3. R. Colin Johnson, 2015, "Microsoft, Google Beat Humans at Image Recognition", EETimes, https://www.eetimes.com/document.asp?doc_id=1325712#
4. Outline of object recognition: https://en.wikipedia.org/wiki/Outline_of_object_recognition

5. Yip, Raymond KK, Peter KS Tam, and Dennis NK Leung, 1992, ”Modification of Hough transform for circles and ellipses detection using a 2- dimensional array.” Pattern Recognition 25.9: 1007-1022. https://www.sciencedirect.com/science/article/abs/pii/003132039290064P
6. David Stark, Swiss Coins dataset, https://www.kaggle.com/ai-first/swisscoins [consulted on April 18, 2019]
7. Y. LeCun, L. Bottou, Y. Bengio and P. Haffner, November 1998, “Gradient-Based Learning Applied to Document Recognition”, Proceedings of the IEEE, 86(11):2278-2324
8. Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton, ImageNet Classification with Deep Convolutional Neural Networks, 2012
9. Karen Simonyan, Andrew Zisserman, Very Deep Convolutional Networks for Large-Scale Image Recognition, 2014
10. Serie of videos on Neural networks by 3Blue1Brown: https://www.3blue1brown.com/neural-networks
11. Tensorflow and deep learning, by Martin Görner: https://youtu.be/qyvlt7kiQoI
12. Dataset of paper [7]: http://yann.lecun.com/exdb/mnist/
13. MIT 6.S191 (2018): Introduction to Deep Learning, Alexander Amini, https://youtu.be/JN6H4rQvwgY?t=1568
14. Lê Nguyên Hoang, 2018, “La formule du savoir: une philosophie unifiée du savoir fondée sur le théorème de Bayes”, EDP Sciences, p.238
15. Github repository, “Handwritten_Digit_Recognition_NN”, Mathis Peyronne, https://github.com/MathisPeyronne/Handwritten_Digit_Recognition_NN
16. Survey of neural networks in autonomous driving - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/Winner-results-of-the-ImageNet-large-scale-visual-recognition-challenge-LSVRC-of-the_fig7_324476862
17. Lectures by Andrew Ng on convolutional neural networks: https://www.youtube.com/playlist?list=PLkDaE6sCZn6Gl29AoE31iwdVwSG-KnDzF
18. Computerphile, Finding the Edges(Sobel Operator), https://www.youtube.com/watch?v=uihBwtPIBxM
19. Computerphile, Canny Edge Detector, https://www.youtube.com/watch?v=sRFM5IEqR2w
20. AI Shack, circle hough transform, http://aishack.in/tutorials/circle-hough-transform/

6 Figures

All consulted on June 30, 2019

1. Mathis Peyronne
2. Mathis Peyronne
3. https://en.wikipedia.org/wiki/File:MnistExamples.png
4. Screenshot of: https://youtu.be/aircAruvnKk?t=69
5. Screenshot of: https://youtu.be/vq2nnJ4g6N0?t=225
6. https://theclevermachine.files.wordpress.com/2014/09/perceptron2.png
7. https://cdn-images-1.medium.com/max/800/1*Xu7B5y9gp0iL5ooBj7LtWw.png
8. https://www.naturalsemi.com/wp-content/uploads/2018/02/techtalk_feb-870x459.png

9. Screenshot of: https://youtu.be/JN6H4rQvwgY?t=1450
10. Screenshot of: https://youtu.be/JN6H4rQvwgY?t=1618
11. Screenshot of: https://youtu.be/JN6H4rQvwgY?t=1663
12. Screenshot of: https://youtu.be/JN6H4rQvwgY?t=1669
13. Screenshot of: https://youtu.be/JN6H4rQvwgY?t=1690
14. Screenshot of: https://youtu.be/JN6H4rQvwgY?t=1702
15. https://miro.medium.com/max/1400/1*DW0Ccmj1hZ0OvSXi7Kz5MQ.jpeg
16. https://miro.medium.com/max/770/1*7S266Kq-UCExS25iX_I_AQ.png
17. https://developer.nvidia.com/sites/default/files/pictures/2018/convolution-1.png
18. http://vision03.csail.mit.edu/cnn_art/index.html
19. https://mblogthumb-phinf.pstatic.net/20160909_264/cjh226_1473417671380l5LvW_PNG/20150126055504.png?type=w2
20. Mathis Peyronne
21. https://docs.opencv.org/3.1.0/canny1.jpg
22. http://aishack.in/static/img/tut/circles.gif
23. http://aishack.in/static/img/tut/circle_edges.jpg
24. http://aishack.in/static/img/tut/circlehough_explanation.jpg
25. Mathis Peyronne
26. Mathis Peyronne
27. Mathis Peyronne
28. Mathis Peyronne
29. Mathis Peyronne
30. Mathis Peyronne
31. Mathis Peyronne
32. Mathis Peyronne
33. Mathis Peyronne
34. Mathis Peyronne
35. Mathis Peyronne
36. Mathis Peyronne
37. Mathis Peyronne