The mathematics of neural networks behind image recognition

Jarno Jäppinen

Master's thesis
June 2024

Supervisor:
Prof. Ion Petre
Reviewer:
Dos. Yury Nikulin

DEPARTMENT OF MATHEMATICS AND STATISTICS

UNIVERSITY OF TURKU
Department of Mathematics and Statistics

JARNO JÄPPINEN: The mathematics of neural networks behind image recognition
Master's thesis, 48 p.
Mathematics
June 2024

This thesis aims to present a special case of neural network for image recognition: the convolutional neural network. Before that, the neural network in general, its components, architecture and operation will be presented. The idea of a convolutional neural network is to reduce the mathematical computation by focusing more on extracting the information that is necessary for recognition. With the convolution neural network technique, networks can be made quite large and complex without the computational needs increasing proportionally. In this paper, we will introduce the mathematics needed to train a neural network. The computation is based on optimizing the weight coefficients of the network by minimizing the error function using a backpropagation algorithm.

The work also includes a simple fully connected neural network and a more complex convolutional neural network trained on two different datasets. The CIFAR-10 and Fashion-MNIST datasets have been used. The CIFAR-10 dataset consists of small coloured images which were more challenging to recognize with the models compared to the Fashion-MNIST dataset where the images are greyscale and slightly smaller. With the CIFAR-10 data, the convolutional neural network learned with a fairly good accuracy in recognition while the fully connected neural network did not achieve very good learning results. With the Fashion-MNIST dataset, both models, fully connected and convolutional, learned to recognize objects quite well.

I conclude that the learning of recognition is particularly influenced by the content of the material. If the images are clear and contain only one object, classification is easy, as it is for humans. But if there are many different things in the picture it is much harder for a human as well as a machine to tell what is in the picture. The amount of computation required is also greatly affected by the size of the images and the number of different classification categories.

Keywords: neural network, fully connected neural network, convolutional neural network, backpropagation.

Contents

1 Introduction

This topic has been chosen because artificial intelligence has recently taken on a very large role in humanity. In particular, autonomous transportation robots such as shopping delivery, cleaning robots and even taxis in some countries. The aim has been to investigate what this machine vision is based on and how a machine/program can be taught to autonomously recognize things it has learned using big data. The primary goal is to learn about the mathematics behind image recognition. The mathematical implementation is based on neural networks inspired by biology. In simplified terms, neural networks should be able to generate a function with X as the initial value and Y as the response, $Y = f(X)$, where X is the information about the pixels in the image and Y is the information about what is in the image.

The first aim is to go through the structure and architecture of artificial neural networks (ANN), the components that make up a neural network. This will be followed by an introduction to the mathematical functions used in the training of neural networks. Mainly the functions are threshold/activation functions and functions that measure the error in learning recognition. After the introduction of the functions, the forward computation (forward propagation) of the neural network is introduced where values are computed for the neurons in the network based on the data entering the network and its values. Once the calculations have been done up to the end of the network, i.e. the prediction result, we next explore the minimization of the prediction error using a backpropagation algorithm to update the weights of the neurons in the network and a few other parameters for a new round of forward computation.

We then present the structure and operation of a special case of a neural network, a convolutional neural network (CNN), which is particularly developed for image recognition. From the convolutional neural network, we present the different layers, followed by the computations in the network and the operation of the backpropagation algorithm in the convolutional neural network. After that, two different important/renowned convolutional neural networks LeNet-5 and AlexNet will be presented.

We will then briefly introduce the Keras and TensorFlow libraries, which are widely used in the code of these neural networks.

This is followed by two examples using the well-known datasets CIFAR-10 and Fashion-MNIST. For each of these datasets, a fully connected neural network and a convolutional neural network have been programmed and it has been examined how the models learn to recognize the dataset images.

This work is written in LaTeX-code on overleaf.com. The images in the thesis are made with LaTeX-code, drawio.com and Python code. The deepl.com translator and the Grammarly application have been used to assist with the English language.

2 Neural Networks

Section 2 is based mainly on the Section *2.6. Artificial Neural Networks* of the book [7] and the Section *11. Multilayer Perceptrons* of the book [5].

2.1 Architecture of the Neural Networks

2.1.1 Neuron

A neuron is the smallest unit in a neural network. Each node in a neural network graph is called a neuron. There are as many neurons in the input layer as the data we use needs. The output layer contains as many neurons as there are classes to be classified. Between these layers, there are several hidden layers, and in these layers, there are several neurons. The number of layers and neurons needed depends on the complexity of the task the network is supposed to learn. Unfortunately, there is no mathematical way to calculate how many layers or neurons are needed, but it has to be found by experimenting with different combinations.

2.1.2 Perceptron

Perceptron is the smallest version of neural networks. The perceptron has a single input layer and in that layer, a set of neurons with values $x_i \in \mathbb{R}, i = 1, \ldots, d$ and each neuron has its own weight $w_i \in \mathbb{R}$. There is also an extra neuron called *bias*, x_0, which always has a value of $+1$ and is connected to the weight value $w_0 \in \mathbb{R}$. Neurons can be outputs of other perceptrons or they can come from the environment, such as the original dataset. The values of each neuron are multiplied by the corresponding weights and then summed. Now the weighted sum of the inputs of the perceptron is:

$$z = \sum_{i=1}^{d} w_i x_i + w_0 = \boldsymbol{w}^T \boldsymbol{x} \tag{1}$$

where $x_i \in \mathbb{R}$ is one of the input values and $w_i \in \mathbb{R}$ is the associated weight and in the simplest case z is the perceptron output value. Figure 1 illustrates what a perceptron looks like and how it works. On the left are the incoming values $\boldsymbol{x} = [1, x_1, \ldots, x_d]^T$, which we multiply by the values $\boldsymbol{w} = [w_0, w_1, \ldots, w_d]^T$. Now we get a weighted sum z, which is equal to the dot product of the vectors $\boldsymbol{w}^T \boldsymbol{x}$. Often there is some *threshold function* or *activation function* that gives the final output of the perceptron, $f(z) = a$.

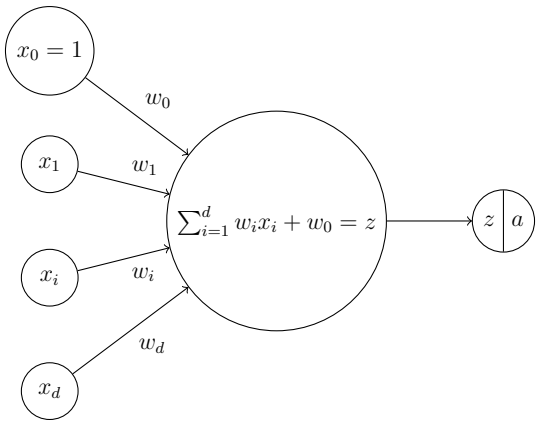


Figure 1: The perceptron model.

In order to learn, perceptron needs feedback on how well it responds. To do this, we need some kind of function that estimates the difference between the output of the perceptron and the actual label of the data. An error function can make such an estimate, and in this case, our main goal is to minimize the value of the error function. It would be better if the error function were differentiable because then we can minimize it using the *gradient descent* method. The minimization has to be done by updating the values of the weights w_i. This can be done by taking the *partial derivative* of the error function with respect to the variable w_i and multiplying it by the *learning factor* η:

$$\Delta w_i = \eta \frac{\partial E}{\partial w_i}. \tag{2}$$

Let us take an example where a perceptron solves a classification problem with two classes (see Section 2.3.2 for more information). In this case, we can use a logistic cost function:

$$E = -y \log(a) - (1 - y) \log(1 - a) \tag{3}$$

for cross-entropy as an error function that measures the goodness of the prediction. Here y is the actual value, so it takes the value one when the data point belongs to the class and zero when it does not. We can use the *chain rule* to compute the partial derivative of the error function with respect to the variable w_i:

$$\Delta w_i = \eta \frac{\partial E}{\partial w_i} = \eta \frac{\partial E}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_i}. \tag{4}$$

Let us first calculate three different partial derivatives. In this example we use the logistic function as the activation function:

$$a = f(z) = \frac{1}{1 + e^{-z}}. \tag{5}$$

We get the following results:

$$\begin{aligned} \frac{\partial E}{\partial a} &= -\frac{y}{a} + \frac{1 - y}{1 - a} = \frac{a - y}{a(1 - a)} \\ \frac{\partial a}{\partial z} &= a(1 - a) \\ \frac{\partial z}{\partial w_i} &= x_i. \end{aligned} \tag{6}$$

Now we can use these results to calculate the update rule for Δw_i:

$$\Delta w_i = \eta \cdot \frac{a - y}{a(1 - a)} \cdot a(1 - a) \cdot x_i = \eta(a - y)x_i. \tag{7}$$

In error minimization, we go in the opposite direction to the gradient, and the function finally takes the form:

$$\Delta w_i = \eta(y - a)x_i. \tag{8}$$

2.1.3 Architecture

Figure 2 illustrates what a feedforward neural network (**FNN**) looks like and how it works. The FNN takes in a data matrix $\boldsymbol{X}^{(in)}_{m\times n} \in \mathbb{R}^{m\times n}$ and modifies it to vector shape $\boldsymbol{x}^{(in)} \in \mathbb{R}^d$. Then each value of vector $\boldsymbol{x}^{(in)}$ is set to different neuron in the input layer of the FNN. In the input layer, there are $m \times n = d$ different neurons plus a bias neuron whose value is always 1. In the input layer the values $x_i^{(in)}, i = 1, \ldots, d$ and $a_i^{(in)}$ are the same. However, the data points may have been modified to correct form before that i.e $\boldsymbol{X}^{(in)}$ is already modified to suitable form from the original data.

The next layer is the first hidden layer and all layers between the input and output layers are hidden layers. In Figure 2 we have two hidden layers. The incoming value of the j:th neuron of the first hidden layer is:

$$z_j^{(h1)} = \sum_{i=1}^{d} w_{i,j}^{(h1)} a_i^{(in)} + w_{0,j}^{(h1)} \tag{9}$$

where $j = 1, \ldots, e$. Then we use selected activation function for every $z_j^{(h1)}$ to get a new activated outgoing value of the j:th neuron of the first hidden layer:

$$a_j^{(h1)} = f(z_j^{(h1)}) \tag{10}$$

where $f(\cdot)$ is the activation function. Now respectively we get the following and all the remaining possible hidden layers in the network. The same calculation method can be used up to the calculation of the output values of the output layer. It may often be necessary to use a different activation function than in the previous layers.

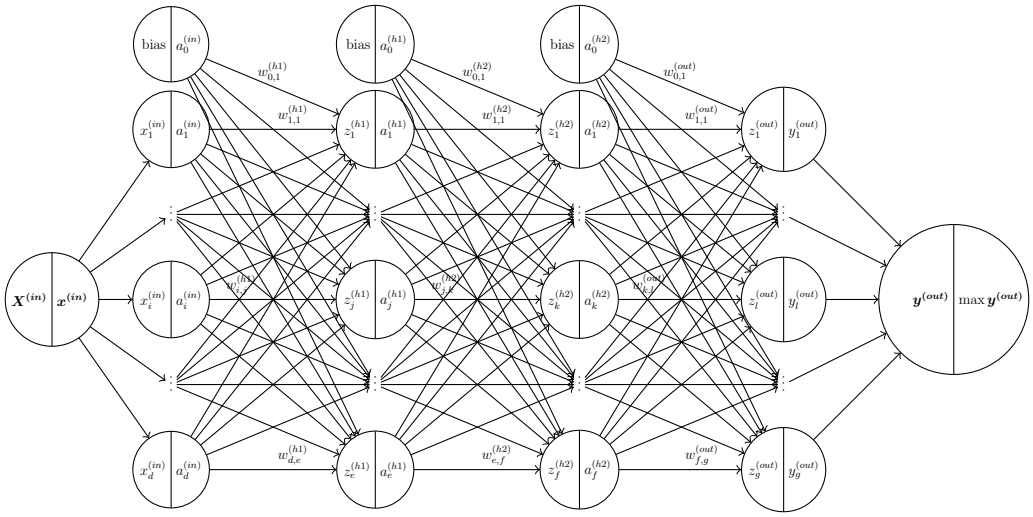


Figure 2: FNN Architecture. The first layer on the left is called the input layer and the last layer on the right is called the output layer. The layers between them are called hidden layers, and the more hidden layers there are, the deeper the network. Every neuron in each layer is connected to every neuron in the next layer except for the bias neuron in the next layer, there are no incoming connections to bias. Each connection has its weight and each neuron is activated with the activation function.

2.2 Activation Functions

Section 2.2 is based mainly on the Section *2.6.2 Activation Functions* of the book [7] and also on the Section *6.4 Activation function* of the book [13].

The activation function should be nonlinear and continuously differentiable. If there are only linear activation functions, then the model will be just linear. A neural network is mainly trained by using gradient descent method so that is why we need the differentiability property. There are other ways than gradient descent, and the differentiability property is not always needed. A couple of examples of these situations:

- Non-Differentiable Components: Some modern techniques incorporate non-differentiable components (e.g., reinforcement learning with discrete action spaces, certain regularization methods, and attention mechanisms). In such cases, techniques like REINFORCE, Gumbel-Softmax, or straight-through estimators are used to handle non-differentiability.

- Alternative Optimization Methods: Though gradient descent is predominant, there are alternative optimization methods being explored, such as evolutionary algorithms, particle swarm optimization, and more. These methods do not always require differentiability.

- Automatic Differentiation: Modern deep learning frameworks (e.g., TensorFlow, PyTorch) provide automatic differentiation, which simplifies the implementation of backpropagation even for complex models.

2.2.1 Sigmoid

Section 2.2.1 is based mainly on the Section *2.6.2.1 Sigmoid* of the book [7] and also on the Section *6.4.1 Sigmoid* of the book [13].

First, we present the sigmoid activation function, also called the logistic function. It is smooth and differentiable everywhere. In the past sigmoid function was a very popular activation function in feedforward neural networks. But there are two problems with that function. The first one is that the identity function is not approximated near zero which we can see from the fact that sigmoid(0) is not close to zero and also $\text{sigmoid}'(z)$ is not close to 1. The second one and the bigger problem is that the gradient of the sigmoid function is close to zero when the input value z is not close to the origin this is called as *vanishing gradients* problem.

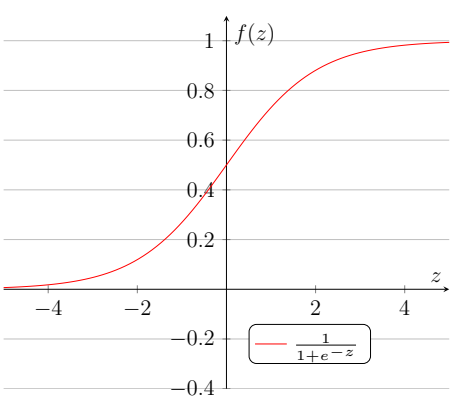
In the backpropagation algorithm, these close to zero values are going to be multiplied because each neuron has its local gradient, so neurons that get first own local gradient are also affecting those neurons that come after. The more layers in the network, the closer the local gradients are to zero in the layers at the beginning of the network. Cause of that the weight updates will be very small and finally the learning will stuck and stop so the sigmoid function is not used in deep networks where there are many hidden layers.

The sigmoid function takes a real number and returns a real number from the interval $(0, 1)$: $\text{sigmoid}(z) : \mathbb{R} \to [0, 1]$. The sigmoid function is the following function:

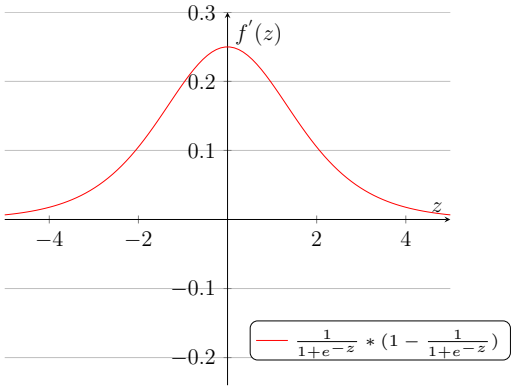
$$f(z) = \text{sigmoid}(z) = \frac{1}{1 + e^{-z}}. \tag{11}$$

Next, we show how the derivative of the sigmoid function with respect to z is obtained:

$$\begin{aligned} f'(z) &= \frac{(1 + e^{-z}) \cdot 0 - 1 \cdot (0 + e^{-z}) \cdot (-1)}{(1 + e^{-z})^2} \\ &= \frac{e^{-z}}{(1 + e^{-z})^2} = \frac{1 + e^{-z} - 1}{(1 + e^{-z})^2} \\ &= \frac{1 + e^{-z}}{(1 + e^{-z})^2} - \frac{1}{(1 + e^{-z})^2} \\ &= \frac{1}{1 + e^{-z}} - \frac{1}{(1 + e^{-z})^2} \\ &= \frac{1}{1 + e^{-z}}(1 - \frac{1}{1 + e^{-z}}) \\ &= \text{sigmoid}(z)(1 - \text{sigmoid}(z)) \end{aligned} \tag{12}$$



(a) Sigmoid-function.



(b) Derivative function of the sigmoid-function.

Figure 3: Sigmoid-function and its derivative function. (Based on Figure 2.39 of [7])

2.2.2 ReLU

In Section *3.5.1 ConvNet Architecture* of the book [7] it is stated that an activation function should be chosen from the family of ReLU (Rectified Linear Unit) function. So let us introduce those family members here. Section 2.2.2 is based mainly on the Section *2.6.2.4 Rectified Linear Unit* of the book [7] and also on the Section *6.4.2 ReLU* of the book [13].

The sigmoid function was typically used for activation before ReLU (rectified linear unit) function was proposed. ReLU was proposed as an activation function in the 2012 architecture called AlexNet [13], after which it grew quickly in popularity. Especially networks with many hidden layers, which are called *deep* neural networks, work better with ReLU- than sigmoid-function. With ReLU-function the gradient is not vanishing.

We define the ReLU function as follows:

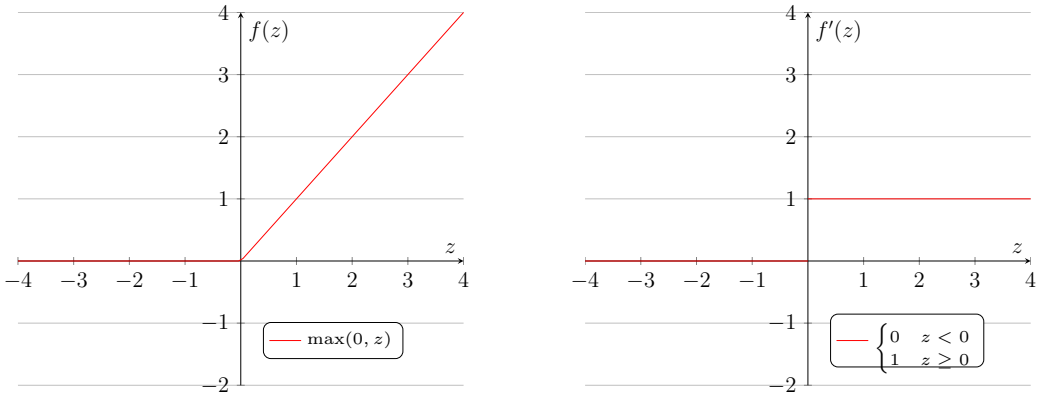
$$f(z) = \max(0, z) \tag{13}$$

and its derivative:

$$f'(z) = \begin{cases} 0 & z < 0 \\ 1 & z \geq 0. \end{cases} \tag{14}$$

From the graph of the ReLU-function 4a we easily see that if input is negative function gives 0 and positive input is output as such. From the graph 4b we can easily see that also the derivative of the ReLU is always zero in the region $\mathbb{R}^-$ and always one in the region $\mathbb{R}^+$. ReLU is very efficient in a computational way. Even though ReLU does not approximate the identity function near the origin but it is not saturated in $\mathbb{R}^+$ so it can produce a strong gradient always in that region and that is why it does not have a vanishing gradient problem and it is always a good choice to deep neural networks. However, not so much good as not something bad because there can be dead neurons because of ReLU activation. The dead neuron

is always getting zero from the activation because the adjustment of the weight of the neuron is like that wx is always negative. The good side of those neurons which are always zero is that those are, in a sense, excluded from the learning calculus and make the network more simpler and more efficient computationally. But the bad side of it is that missing some neurons which are always zero can affect to whole network's accuracy so it should be checked while training the network if there are any dead neurons.



(a) ReLU-function. (b) Derivative function of the ReLU-function.

Figure 4: ReLU-function and its derivative function. (Based on Figure 2.42 of [7])

2.2.3 LReLU

This Section 2.2.3 is based mainly on the Section *2.6.2.5 Leaky Rectified Linear Unit* of the book [7].

Second, we have LReLU (Leaky ReLU), which is defined as follows:

$$f(z) = \max(0, z) + b \min(0, z) \tag{15}$$

or equivalently:

$$f(z) = \begin{cases} z, & \text{if } z > 0 \\ bz, & \text{if } z \leq 0 \end{cases} \tag{16}$$

$$f'(z) = \begin{cases} 1, & \text{if } z > 0 \\ b, & \text{if } z \leq 0, \end{cases} \tag{17}$$

where b is a fixed and small value usually between $[0, 1]$, often $b = 0.01$. That constant b is the difference between the original ReLU and it is there for solving the problem of dead neurons. Because of that the gradient does not vanish when z gets negative values.

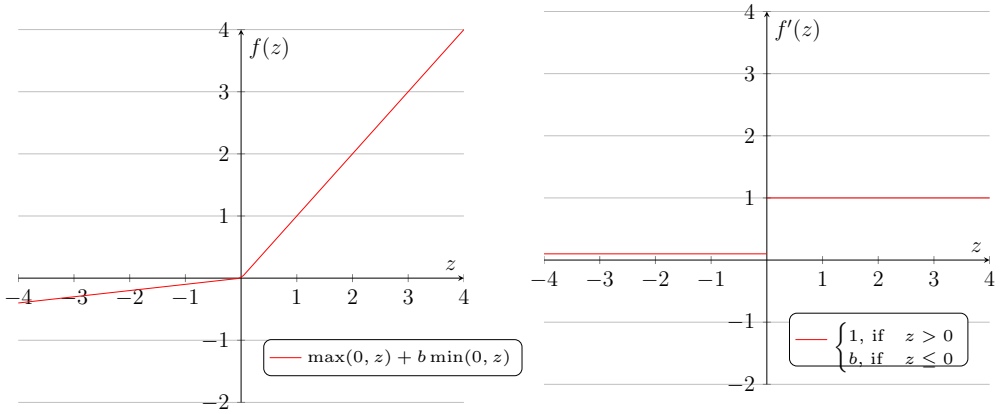
2.2.4 PReLU

This Section 2.2.4 is based mainly on the Section *2.6.2.6 Parameterized Rectified Linear Unit* of the book [7].

Third, we present PReLU (Parameterized ReLU), which is the same as LReLU, but now the b is not a fixed hyperparameter but a learning parameter, so its value may change while the network is learning. But it needs that we have to compute the gradient of the LReLU with respect to b:

$$\frac{\partial f(z)}{\partial b} = \begin{cases} 0, & \text{if } z > 0 \\ z, & \text{if } z \leq 0, \end{cases} \tag{18}$$

and the gradient of the error function with respect to b is also obtained with the backpropagation algorithm and updated as the other parameters in the network.



(a) Leaky ReLU / PReLU.

(b) Derivative function of the LReLU / PReLU-function.

Figure 5: LReLU / PReLU -function and its derivative function. For clarity of the graph, $b = 0.1$ instead of 0.01. (Based on Figure 2.43 of [7])

2.2.5 ELU

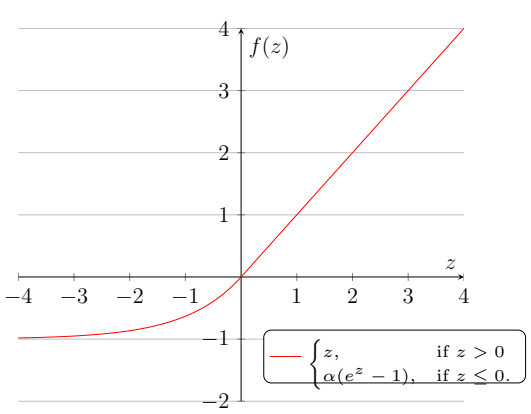
This Section 2.2.5 is based mainly on the Section *2.6.2.8 Exponential Linear Unit* of the book [7].

As the last of the ReLU family, we present ELU (Exponential Linear Unit):

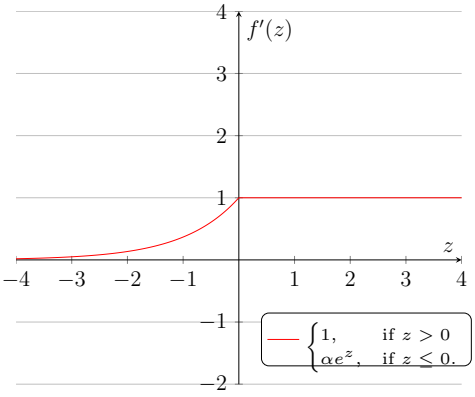
$$f(z) = \begin{cases} z, & \text{if } z > 0 \\ \alpha(e^z - 1), & \text{if } z \leq 0 \end{cases} \tag{19}$$

$$f'(z) = \begin{cases} 1, & \text{if } z > 0 \\ \alpha(e^z), & \text{if } z \leq 0. \end{cases} \tag{20}$$

Learning is usually sped up by ELU activation. Because of the non-linearity, its negative gradient is also smoother than the others and does not fall as steeply, which can be also seen in the graph (6b) of the gradient of the ELU-function.



(a) ELU.



(b) Derivative function of the ELU-function.

Figure 6: ELU-function and its derivative function ($\alpha = 1$). (Based on Figure 2.45 of [7])

2.2.6 Softmax

Section 2.2.6 is based mainly on the Sections *6.5.3 [0,1] Interval with Sum 1* and *7.4.2 Gradient of Cross-Entropy Function* of the book [13] and also on the Section *2.3.4 Multinomial Logistic Function* of the book [7].

For multi-class classification, we need some function that can convert the input real values to an interval of $[0, 1]$ and also ensure that the output values sum to one. This is because we need a probability value for each category in the output layer. For this requirement, there is a function called the softmax activation function or just the softmax. The softmax function takes a real number and returns a real number from the interval $(0, 1)$: $\text{softmax}(z) : \mathbb{R} \to [0, 1]$. The sum of the softmax outputs is equal to one $\sum_{i=1}^{g} \text{softmax}(z_i) = 1$. The softmax function is the following function:

$$f(z_i) = \text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{g} e^{z_j}}, \tag{21}$$

where g is the number of classification classes and

$$\sum_{j=1}^{g} e^{z_j} \tag{22}$$

is the normalization factor that makes the sum of the output values of the softmax function equal to one. It also relates all input values to each other, i.e. each input value gets its value from the interval of $[0, 1]$ depending on how large the value is. The values are then comparable, and in our case, they are given a probability value.

For the backpropagation, we also need the partial derivative of the softmax with respect to z. Since the softmax function has a sum function in the denominator with different values of z, we also need to take the partial derivative with respect to all of these values. There are two different cases, in one case z_i in the numerator is different from z_j in the partial derivative, and in the other case they are the same.

Mathematically speaking $\frac{\partial \text{softmax}(z_i)}{\partial z_j}$, when $i \neq j$ and when $i = j$. Let us first look at the case one, $i \neq j$:

$$\begin{aligned}\frac{\partial f(z_i)}{\partial z_j} &= \frac{\sum_{k=1}^{g} e^{z_k} \cdot 0 - e^{z_i} \cdot e^{z_j}}{(\sum_{k=1}^{g} e^{z_k})^2} = \frac{-e^{z_i} \cdot e^{z_j}}{(\sum_{k=1}^{g} e^{z_k})^2} \\ &= \frac{-e^{z_i}}{\sum_{k=1}^{g} e^{z_k}} \cdot \frac{e^{z_j}}{\sum_{k=1}^{g} e^{z_k}} = -f(z_i) \cdot f(z_j).\end{aligned} \tag{23}$$

So it is just a minus sign multiplication of two different values activated by the softmax function. Let us look at the second case, $i = j$:

$$\begin{aligned}\frac{\partial f(z_i)}{\partial z_j} &= \frac{\sum_{k=1}^{g} e^{z_k} \cdot e^{z_i} - e^{z_i} \cdot e^{z_j}}{(\sum_{k=1}^{g} e^{z_k})^2} = \frac{e^{z_i} \cdot (\sum_{k=1}^{g} e^{z_k} - e^{z_j})}{(\sum_{k=1}^{g} e^{z_k})^2} \\ &= \frac{e^{z_i}}{\sum_{k=1}^{g} e^{z_k}} \cdot \left(\frac{\sum_{k=1}^{g} e^{z_k} - e^{z_j}}{\sum_{k=1}^{g} e^{z_k}} \right) = \frac{e^{z_i}}{\sum_{k=1}^{g} e^{z_k}} \cdot \left(\frac{\sum_{k=1}^{g} e^{z_k}}{\sum_{k=1}^{g} e^{z_k}} - \frac{e^{z_j}}{\sum_{k=1}^{g} e^{z_k}} \right) \\ &= \frac{e^{z_i}}{\sum_{k=1}^{g} e^{z_k}} \cdot \left(1 - \frac{e^{z_j}}{\sum_{k=1}^{g} e^{z_k}} \right) = f(z_i) \cdot (1 - f(z_j)) = f(z_i) \cdot (1 - f(z_i)).\end{aligned} \tag{24}$$

And this is just multiplying the probability value of one neuron by the probability value of its negation. So both cases together:

$$\frac{\partial f(z_i)}{\partial z_j} = \begin{cases} -f(z_i) \cdot f(z_j), & \text{when } i \neq j \\ f(z_i) \cdot (1 - f(z_j)), & \text{when } i = j. \end{cases} \tag{25}$$

2.3 Error Functions

This Section 2.3 is based mainly on the lecture material of Ion Petre [10].

2.3.1 Regression

We use regression when we want to determine the relationship between two variables [15]. For example when the value of x increases how will the value of $y = f(x)$ change. We can develop a straight line to describe the correlation between those two values. When we have that regression line, which is the same as our predicting function, then we can use the mean squared error (MSE) to evaluate the error of our predicting function. So, the error function for regression is:

$$E(\boldsymbol{y}, \boldsymbol{r}) = \sum_{i=1}^{n} \frac{1}{2}(y_i - r_i)^2, \tag{26}$$

where y_i is the actual value and r_i is the calculated/predicted value. And the gradient of that with respect to variable r_i is:

$$\frac{\partial E(\boldsymbol{y}, \boldsymbol{r})}{\partial r_i} = \frac{\partial \frac{1}{2}(y_i - r_i)^2}{\partial r_i} = \frac{1}{2} \cdot 2 \cdot (y_i - r_i) \cdot (-1) = (r_i - y_i). \tag{27}$$

2.3.2 Binary Classification

Next, we present the binary classification. We use it when there are only two classes to classify. So, the error function for binary classification is:

$$E(\boldsymbol{y}, \boldsymbol{r}) = -\sum_{i=1}^{n} y_i \log(r_i) + (1 - y_i) \log(1 - r_i), \tag{28}$$

where the actual value $y_i = 1$, if data point $x \in C_1$ and $y_i = 0$, if $x \in C_0$. And $r_i \in [0, 1]$ is the calculated/predicted value. In that equation, there are two different log-functions of which only one is valid at a time. The value of this negative log-function increases the further the prediction is from the true value so the error function gets a bigger value depending on how incorrect the prediction has been. Figure 7 illustrates the graph of the log-function. When the input value is 1 the output is 0 and the further the value is from one toward zero the higher the output is.

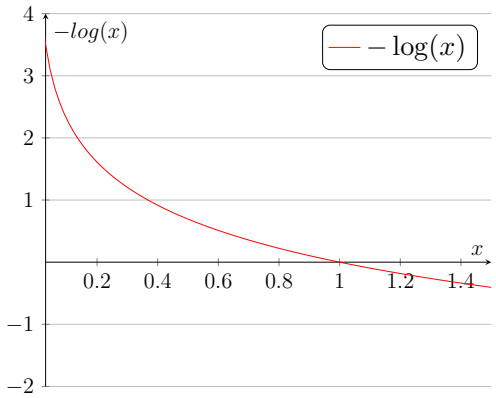


Figure 7: Negative log function. Returns zero when $x = 1$, and increases as x approaches zero.

And the gradient of that error function with respect to variable r_i is:

$$\begin{aligned}
\frac{\partial E(\boldsymbol{y}, \boldsymbol{r})}{\partial r_i} &= \frac{\partial(-y_i \log(r_i) - (1 - y_i) \log(1 - r_i))}{\partial r_i} = -y_i \cdot \frac{1}{r_i} - (1 - y_i) \cdot \frac{1}{1 - r_i} \cdot (-1) \\
&= \frac{-y_i}{r_i} + \frac{1 - y_i}{1 - r_i} = \frac{-y_i(1 - r_i)}{r_i(1 - r_i)} + \frac{r_i(1 - y_i)}{r_i(1 - r_i)} = \frac{-y_i + y_i r_i + r_i - y_i r_i}{r_i(1 - r_i)} \\
&= \frac{r_i - y_i}{r_i(1 - r_i)}.
\end{aligned} \tag{29}$$

2.3.3 Multiclass Classification

Last, we present the multiclass classification. We use it when there are more than two classes to classify. So, the error function for multiclass classification is:

$$E(\boldsymbol{y}, \boldsymbol{r}) = -\sum_{i=1}^{n} y_i \log(r_i), \tag{30}$$

where the actual value $y_i = 1$, if data point $x \in C_i$, otherwise it is 0. $r_i \in [0, 1]$ is the calculated/predicted value and

$$\sum_{i=1}^{n} r_i = 1. \tag{31}$$

Equation (30) is the same as the first part of the previous equation (28). In that sum, all but one of these terms will be zeros because $y_i = 1$ only when the index i is the same as the class where x belongs. When $y_i = 1$ then the magnitude of the error depends on how close also r_i is to one. And the magnitude of the error will increase according to the graph of Figure 7.

2.4 Forward Propagation

This Section 2.4 is based mainly on the lecture material of Ion Petre [10].

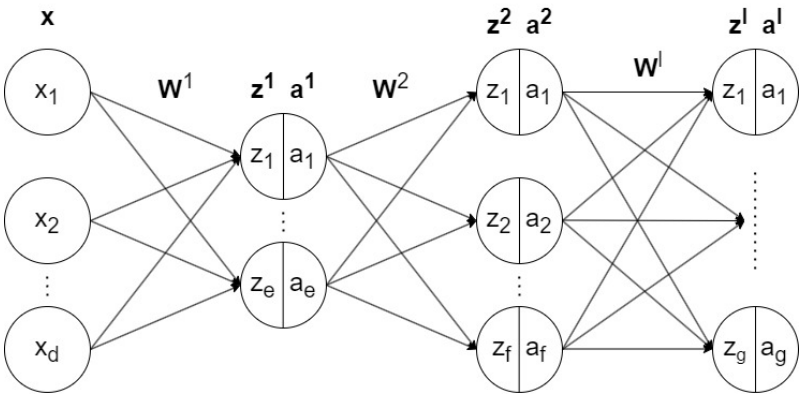


Figure 8: The beginning of a small fully connected neural network.

Figure 8 demonstrates the beginning of a small fully connected network. Let us denote the input data with $\boldsymbol{x} \in \mathbb{R}^{d \times 1}$ where d is the needed quantity of the neurons on the input layer. Let us denote the values of different weights that are connected from layer $l-1$ to layer l with $\boldsymbol{W}^l \in \mathbb{R}^{m \times n}$ where m and n are the quantity of neurons on layers l and $l-1$ respectively. Now we can start to calculate the values on the first hidden layer:

$$\boldsymbol{z}^1 = \boldsymbol{W}^1 \boldsymbol{x} \tag{32}$$

where $\boldsymbol{z}^1 \in \mathbb{R}^{e \times 1}$ represents the incoming values of the layer 1. Next, we are using the activation function to activate all the income values:

$$\boldsymbol{a}^1 = f^1(\boldsymbol{z}^1) = f^1(\boldsymbol{W}^1 \boldsymbol{x}) \tag{33}$$

where $\boldsymbol{a}^1 \in \mathbb{R}^{e \times 1}$ represents the activated values of the layer 1. Now the following layers will be calculated in the same way. Let us calculate the values of the second

layer. Incoming values:

$$\boldsymbol{z}^2 = \boldsymbol{W}^2\boldsymbol{a}^1 = \boldsymbol{W}^2 f^1(\boldsymbol{W}^1\boldsymbol{x}) \tag{34}$$

and activated outgoing values:

$$\boldsymbol{a}^2 = f^2(\boldsymbol{z}^2) = f^2(\boldsymbol{W}^2 f^1(\boldsymbol{W}^1\boldsymbol{x})). \tag{35}$$

Now we can generalize the calculation to any layer l, incoming values:

$$\boldsymbol{z}^l = \boldsymbol{W}^l\boldsymbol{a}^{l-1} = \boldsymbol{W}^l f^{l-1}(\boldsymbol{W}^{l-1} \dots f^1(\boldsymbol{W}^1\boldsymbol{x}) \dots) \tag{36}$$

and activated outgoing values:

$$\boldsymbol{a}^l = f^l(\boldsymbol{z}^l) = f^l(\boldsymbol{W}^l f^{l-1}(\boldsymbol{W}^{l-1} \dots f^1(\boldsymbol{W}^1\boldsymbol{x}) \dots)) \tag{37}$$

also, the output layer has the same calculation but the activation function often be different.

2.5 Parameters

2.5.1 Weights and Biases

Section 2.5.1 is based mainly on the Section *2.6.4 Initialization* of the book [7].

The artificial network has mainly two parameters, weights and biases. In the beginning, the values of those parameters are initialized as zero for the biases and nearly zero for the weights. Also, it is very important to initialize different values for every weight in the same layer. Otherwise, their update rule will be generated the same because they will have the same gradients.

2.6 Backpropagation

With the backpropagation algorithm, we are updating the weights of the neural network. The update of the weights is based on the minimization of the selected *loss function*. In backpropagation, we are traveling the network from the end to that connection line which weight we are going to update. Figure 9 illustrates the connections after the connection line where $w^1_{1,1}$ is the weight. All these connections depend on the weight $w^1_{1,1}$ so in backpropagation we have to go all these paths back to updated weight $w^1_{1,1}$. Next, we are going to see how that is done in practice.

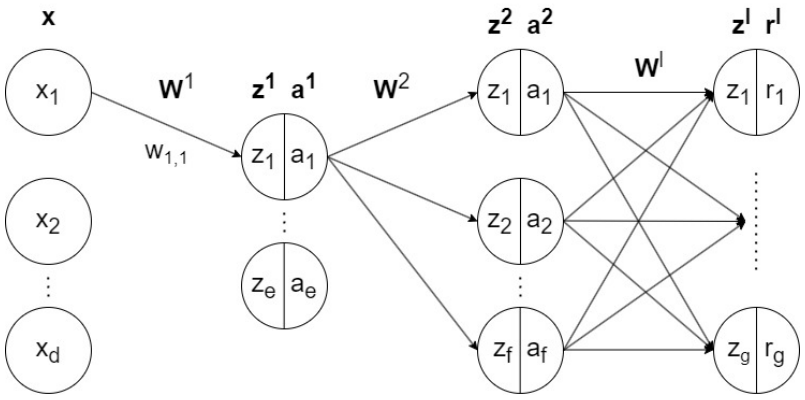


Figure 9: Weight $w^1_{1,1}$ update.

Let us choose to our loss function:

$$E(\boldsymbol{y}, \boldsymbol{r}) = -\sum_{i=1}^{g} y_i \log(r_i) \tag{38}$$

where $\boldsymbol{y}$ is the actual value where $y_i = 1$ if datapoint $x \in C_i$ and $y_i = 0$ if $x \notin C_i$. And $\boldsymbol{r}$, where $r_i = P(C_i|x)$, is the vector that includes different probabilities that our network gives for different classes. Now if we would like to update the weight value of the last outgoing layer it would happen in the following manner:

$$\Delta w^l_{j,i} = -\eta \frac{\partial E}{\partial w^l_{j,i}} = -\eta \frac{\partial E}{\partial r^l_j} \cdot \frac{\partial r^l_j}{\partial z^l_j} \cdot \frac{\partial z^l_j}{\partial w^l_{j,i}}, \tag{39}$$

where $w^l_{j,i}$, $0 \leq j \leq g$ and $0 \leq i \leq f$, is the weight value to the j:th neuron in layer l from the i:th neuron in layer $l-1$.

We can use the equation 39 to generalize that same calculation over the whole network. Let us make the following notation:

$$\frac{\partial E}{\partial r^l_j} \cdot \frac{\partial r^l_j}{\partial z^l_j} = \frac{\partial E}{\partial z^l_j} = \delta^l_j, \tag{40}$$

where δ^l_j is a local gradient for the neuron j in layer l [10]. Let us now use that same model also for other layers. If we now use Figure 9 to make it easier for us to understand and let us update the weight from the penultimate layer then we get:

$$\Delta w^{l-1}_{i,h} = -\eta \frac{\partial E}{\partial w^{l-1}_{i,h}} = -\eta \frac{\partial E}{\partial a^{l-1}_i} \cdot \frac{\partial a^{l-1}_i}{\partial z^{l-1}_i} \cdot \frac{\partial z^{l-1}_i}{\partial w^{l-1}_{i,h}}, \tag{41}$$

where

$$\frac{\partial E}{\partial a^{l-1}_i} = \sum_{j=1}^{g} \frac{\partial E}{\partial r^l_j} \cdot \frac{\partial r^l_j}{\partial z^l_j} \cdot \frac{\partial z^l_j}{\partial a^{l-1}_i} = \sum_{j=1}^{g} \frac{\partial E}{\partial z^l_j} \cdot \frac{\partial z^l_j}{\partial a^{l-1}_i} = \sum_{j=1}^{g} \delta^l_j \cdot w^l_{j,i}. \tag{42}$$

Now we easily see that the local gradient for neurons in hidden layers is:

$$\delta_i^l = \frac{\partial E}{\partial z_i^l} = \frac{\partial E}{\partial a_i^l} \cdot \frac{\partial a_i^l}{\partial z_i^l} = \sum_{j=1}^{N_{l+1}} \delta_j^{l+1} \cdot w_{j,i}^{l+1} \cdot \frac{\partial a_i^l}{\partial z_i^l}, \tag{43}$$

where N_{l+1} is the number of the neurons in the layer $l+1$. We see that the new gradient is the sum of earlier gradients and weights multiplication which is multiplied with the gradient of the activation function. Now we can rewrite the equation 39:

$$\Delta w_{j,i}^l = -\eta \frac{\partial E}{\partial w_{j,i}^l} = -\eta \frac{\partial E}{\partial r_j^l} \cdot \frac{\partial r_j^l}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial w_{j,i}^l} = -\eta \frac{\partial E}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial w_{j,i}^l} = -\eta \cdot \delta_j^l \cdot \frac{\partial z_j^l}{\partial w_{j,i}^l}, \tag{44}$$

now here we have a clear way to update the weights.

2.6.1 Adaptive Gradient Algorithm, AdaGrad

This 2.6.1 Section is based on the Section *4.3 Adagrad* of the article *An overview of gradient descent optimization algorithms* [16] and also to *AdaGrad* textbook of *the Cornell University* [3].

Let us denote for brevity, the gradient of the loss function E with respect to the parameter $w_{t,j,i}^l$ at time step t:

$$g_{t,j,i}^l = \frac{\partial E}{\partial w_{t,j,i}^l}, \tag{45}$$

now, we can denote the normal update rule as follows:

$$w_{t+1,j,i}^l = w_{t,j,i}^l - \eta \cdot g_{t,j,i}^l, \tag{46}$$

here the learning rate η has the same value in every time step. In Adagrad there will be also another factor so the learning factor is in a sense different in every time step. Let us present a diagonal matrix:

$$G_{t,i,i}^l = \sum_{\tau=1}^{t} g_{\tau,j,i}^l (g_{\tau,j,i}^l)^T. \tag{47}$$

On the diagonal of this matrix $\boldsymbol{G}_t^l \in \mathbb{R}^{f \times f}$ there is the sum of the squares of the gradients up to the time step t. In the final Adagrad update algorithm, we have the dividing factor:

$$\frac{1}{\sqrt{G_{t,ii}^l + \epsilon}} \tag{48}$$

where ϵ is mainly to prevent division by zero and without that square root the next algorithm works a lot worse [16].

$$w^l_{t+1,j,i} = w^l_{t,j,i} - \eta \cdot g^l_{t,j,i} \cdot \frac{1}{\sqrt{G^l_{t,ii} + \epsilon}} = w^l_{t,j,i} - \frac{\eta}{\sqrt{G^l_{t,ii} + \epsilon}} \cdot g^l_{t,j,i}. \tag{49}$$

The main benefit of this Adagrad algorithm is that the learning rate does not need to be adjusted manually. Adagrad's weakness is that the accumulated sum of gradients $\boldsymbol{G}_t$ in the denominator continually increases during training, leading to a decrease in the learning rate and eventually it becomes too small and no longer able to learn. This can lead to premature convergence and poor performance on large datasets.

2.6.2 Root Mean Square Propagation, RMSProp

This 2.6.2 Section is based on the Section *4.4 Adadelta* and *4.5 RMSProp* of the article *An overview of gradient descent optimization algorithms* [16].

RMSProp is a popular choice for training neural networks due to its efficiency and robustness. It is commonly used in deep learning frameworks and often performs well with default hyperparameters. However, the performance of RMSProp can be sensitive to the choice of hyperparameters, especially the learning rate and decay rate.

In RMSProp is used a running average $A[g^2]_t$ which is recursively defined by the running average of the preceding time step and gradient of this time step:

$$A[g^2]^l_{t,j,i} = \gamma A[g^2]^l_{t-1,j,i} + (1-\gamma)(g^2)^l_{t,j,i}, \tag{50}$$

here γ is suggested to be set to 0.9 by Geoff Hinton who proposed this method [16]. Now we can use that value for the update rule:

$$w^l_{t+1,j,i} = w^l_{t,j,i} - \eta \cdot g^l_{t,j,i} \cdot \frac{1}{\sqrt{A[g^2]^l_{t,j,i} + \epsilon}} = w^l_{t,j,i} - \frac{\eta}{\sqrt{A[g^2]^l_{t,j,i} + \epsilon}} \cdot g^l_{t,j,i}. \tag{51}$$

2.6.3 Adaptive Moment Estimation, Adam

This 2.6.3 Section is based on the Section *4.6 Adam* of the article *An overview of gradient descent optimization algorithms* [16], slideshow of *Nadav Cohen* [14] and also to *Adam* textbook of *the Cornell University* [2].

Adaptive Moment Estimation (Adam) is combining advantages of the AdaGrad and RMSProp [14]. Adam is storing both exponentially decaying averages of past gradients and past squared gradients, m_t and v_t respectively. Storing methods are quite the same as in the RMSProp:

$$\begin{aligned} m^l_{t,j,i} &= \beta_1 m^l_{t-1,j,i} + (1-\beta_1) g^l_{t,j,i} \\ v^l_{t,j,i} &= \beta_2 v^l_{t-1,j,i} + (1-\beta_2)(g^2)^l_{t,j,i}, \end{aligned} \tag{52}$$

here $m^l_{t,j,i}$ is estimate of the first moment and $v^l_{t,j,i}$ is estimate of the second moment. At the beginning $m^l_{t,j,i}$ and $v^l_{t,j,i}$ are initialized as zeros so those equations (52) are giving zeros especially when β_1 and β_2 are close to 1 [16]. The authors of Adam fixed this flaw by computing bias-corrected estimates for both. Next, we show how Cohen [14] mathematically argues the derivation of the bias-corrections:

$$m^l_{t,j,i} = \beta_1 m^l_{t-1,j,i} + (1-\beta_1) g^l_{t,j,i} = \sum_{\tau=1}^{t} (1-\beta_1)(\beta_1)^{t-\tau} \cdot g^l_{\tau,j,i} \tag{53}$$

and here

$$\sum_{\tau=1}^{t} (1-\beta_1)(\beta_1)^{t-\tau} = (1-(\beta_1)^t) \tag{54}$$

so we can divided by $(1-(\beta_1)^t)$ to achieve unbiased estimates, analogous arguments to both:

$$\begin{aligned} \hat{m}^l_{t,j,i} &= \frac{m^l_{t-1,j,i}}{1-\beta_1^t} \\ \hat{v}^l_{t,j,i} &= \frac{v^l_{t-1,j,i}}{1-\beta_2^t}. \end{aligned} \tag{55}$$

Now we have every elements to use the Adam update rule:

$$w^l_{t+1,j,i} = w^l_{t,j,i} - \frac{\eta}{\sqrt{\hat{v}^l_{t,j,i}} + \epsilon} \cdot \hat{m}^l_{t,j,i}, \tag{56}$$

here proposed values are 0.9 for β_1, 0.999 for β_2 and 10^{-8} for ϵ [16]. It is empirically shown by the authors that Adam works better in practice than other adaptive learning-method algorithms [16]. Still, like many adaptive methods, Adam can sometimes lead to overfitting on training data if regularization is not done properly.

3 Convolutional Neural Networks

This Section 3 is based mainly on the Section *3. Convolutional Neural Networks* of the book [7] and also on the lecture material of Ion Petre [10].

The main motivation for using convolution is that it requires far fewer parameters than fully connected networks. It makes learning so much easier from a mathematical point of view and also less data is needed for learning. Let us say that we have a colour image size of 32 pixels times 32 pixels, which is a rather small image size, still there would be $32 \times 32 \times 3 = 1024 \times 3 = 3072$ different data points which correspond to the number of neurons needed in our case. Now if we use the FNN architecture, where every neuron in each layer is connected to every neuron in the previous layer,

we get a huge number of connections. Each of these connections would have its own weight which would need to be updated by calculating new values for them. Let us go back to calculating the image size. Let us say that there are 4704 neurons in the first hidden layer. So that makes $3072 \times 4704 = 14,450,688$ different connections, all with different weights to update, and this is only in the first layer.

If we start thinking about how we could reduce that number of different weights, we could easily come up with the idea of reducing the number of neurons in the hidden layer. However, this is not a good idea because it can adversely affect the classification performance, which is why we want to keep the number of neurons in the first hidden layer high. If we rearrange the neurons in the first hidden layer, we can do this, for example, by making six different 28×28 neuron blocks. In an image, pixels that are close together tend to correlate more than pixels that are far apart. Now we could try to take, for example, a 5×5 pixel area from the original image and, by some operation, connect them to just one neuron in the first hidden layer. Then another area of the same size is taken, then a third, and so on. This will help us reduce the number of connections. This would be $(5 \times 5 \times 3) \times (28 \times 28 \times 6) = 352,800$ different connections, which is less than 3% of the original number of connections. This number can still be reduced more. In the previous example, we had different weights for each connection, but we could also use the same weights between neurons in the same block, which is called *weight sharing*. In this case, it can be thought that there is some kind of filter between the image and each block of the first hidden layer, as shown in Figure 10. Now the number of different weights is $(5 \times 5 \times 3) \times (6) = 450$ which is so far from the original $14,450,688$. This is the main idea of the convolution.

3.1 Architecture

This Section 3.1 is based mainly on the Section *3.5.1 ConvNet Architecture* of the book [7].

The architecture of convolutional neural networks often starts with a convolutional layer or even several convolutional layers, followed by a pooling layer. This usually continues for a few layers, followed by a couple of fully connected layers and finally, always an output layer comes last one. A couple of architectural rules are that there is only one classification layer, and that is the output layer. It does not make sense to use more than one layer for classification. Typically, filters of size 3×3, 5×5 and 7×7 are used in the convolution. typically, the activation function is only used immediately after the convolution layer, but if there is more than one convolution layer in a row, only one activation function is still used. It is also suggested to use ReLU or a similar activation function from the ReLU family. It is also important to note that the number of network parameters to be trained should not be in the millions if there is very little training data.

3.2 Types of Layers

3.2.1 Convolutional

This Section 3.2.1 is based mainly on the lecture material of Ion Petre [10].

In convolution, a filter or typically many different filters of a certain size (for example 5×5) is applied to the original image to pick up the numbers from that size range by some rule that is usually stored in the filter (coefficients e.g. -1,0,+1, etc...). Filters are designed to recognize certain features, e.g. contours. The filter outputs only one number which is stored in one pixel of the output matrix. The number of filters tells how many different matrices there are in the post-convolution layer. The different filters run their own rules to pick up the data from the same points and form a single number in their own matrix. Filters can also be used to reduce parameters when using weight sharing to update filter weights. Figure 10 shows how the convolution works. There are $5 \times 5 \times 3$ filters that first take the 5×5 region from the top left of each of the three input channels, filter it and store it in the output matrix. Next, the filters move to the right by as many steps as the stride is. In this case, one step. Once the filter has reached the top right corner of the input matrix, it returns to the left side and goes down as many steps as the stride is. Now the filters move again to the right and then downwards, thus filtering the entire input matrix into output matrices.

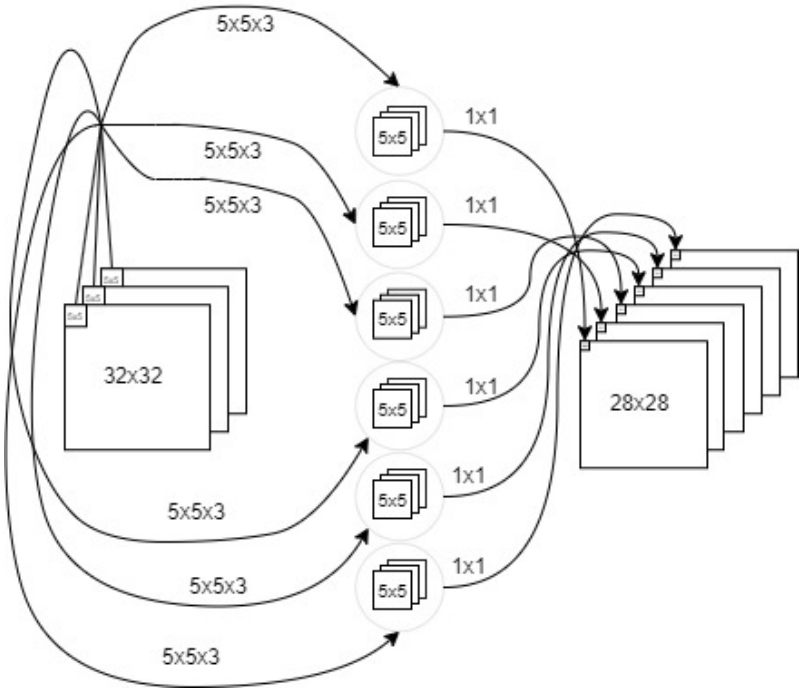


Figure 10: Illustrative figure of the image, filters and first hidden layer.

Let us first introduce the hyperparameters of convolutional layer l:

- The size of the filter: $f^{(l)} \times f^{(l)} \times c^{(l-1)}$
- The number of the filters: $n_t^{(l)}$
- Padding: $p^{(l)}$
- Stride: $s^{(l)}$

In general, the filters are square matrices and the number of channels $c^{(l-1)}$ is the same as in the input matrix of the convolution layer, i.e. the previous layer. The number of filters is decided by the designer of the network who tries to find the best match. Padding means how many rows and columns there are extra, typically they are just zeros. For example, if the padding is one, the matrix will have one extra row at the top and bottom and one extra column at the beginning and end of the matrix. Padding can be used to make neurons at the edge of the matrix more valuable. Especially in corners where neurons would otherwise have been detected with filters only once. Stride means how many steps the filters move at a time. It is the same to the right and down, but only in one direction at a time.

Second, we introduce the size of the input matrix and how to calculate the size of the output matrix from that in layer l:

- Input: $h^{(l-1)} \times w^{(l-1)} \times c^{(l-1)}$
- Output: $h^{(l)} \times w^{(l)} \times c^{(l)}$
  - $h^{(l)} = \left\lfloor \frac{h^{(l-1)} + 2p^{(l)} - f^{(l)}}{s^{(l)}} \right\rfloor + 1$
  - $w^{(l)} = \left\lfloor \frac{w^{(l-1)} + 2p^{(l)} - f^{(l)}}{s^{(l)}} \right\rfloor + 1$
  - $c^{(l)} = n_t^{(l)}$

The size of the output matrix is equal to the number of times the filter covers the input matrix. This is based on the fact that for each region where the filter is, it stores one value in the output matrix. So, here the height and width are calculated in the same way. First, add the padding times two to get the size of the input matrix, then subtract the size of the filter. To get a figure for how many times the filter covers the input matrix, divide the calculation by the stride value and if the result is not an integer, round down to the nearest integer and then we add one. The number of channels in the output matrix $c^{(l)}$ is the same as the number of filters $n_t^{(l)}$. This is based on the fact that each filter passes through an input matrix and stores the values in its own matrix. Therefore there are as many channels as there are filters, the channels in this case could also be thought of as just separate matrices.

Third, we introduce the trainable parameters of the convolutional layer:

- Each filter has $f^{(l)} \cdot f^{(l)} \cdot c^{(l-1)}$ parameters
- Each filter has one bias parameter
- So altogether there is $n_t^{(l)} \cdot (f^{(l)} \cdot f^{(l)} \cdot c^{(l-1)} + 1)$ parameters

Each filter has certain values that are used to calculate the network. These values are weighted differently and updated while the network is learning. Each filter therefore has as many parameters as the size of the filter is. Each filter also has its own bias parameter. Therefore, there are as many parameters as the size of the filters plus one and that times the number of filters.

3.2.2 Pooling

This Section 3.2.2 is based mainly on the lecture material of Ion Petre [10].

Like the convolution layer, the pooling layer has a filter that goes through the input matrices. The pooling layer has only one filter that passes through each channel of the input matrix. The filter stores one value in the output matrix for each step it takes from the input matrix.

Let us first introduce the hyperparameters of pooling layer l:

- The size of the filter: $f^{(l)} \times f^{(l)}$
- The number of the filters: 1
- Padding: $p^{(l)}$ (very often $p = 0$)
- Stride: $s^{(l)}$
- Type of pooling is MAX (maximum value) or AVG (average value)

Also in the pooling layer, the filter is a square matrix, for example 2×2, but with only one channel. The same filter goes through each channel, and no other filter is used at the same time, just this one. Padding means the same as in convolution, but in pooling it is very often just zero, so it is rarely used. Stride is also the same as in convolution, it means step length, how many steps the filter takes when moving. In pooling, the filter is not involved in the calculation, so the filter does not contain any numbers. There are two types of filters, a maximum filter and an average filter. From the area covered by the maximum filter is selected the maximum number and from the area covered by the average filter is calculated the average value.

Second, we present the size of the input matrix and how the size of the output matrix is calculated using it in layer l:

- Input: $h^{(l-1)} \times w^{(l-1)} \times c^{(l-1)}$
- Output: $h^{(l)} \times w^{(l)} \times c^{(l)}$
  - $h^{(l)} = \left\lfloor \frac{h^{(l-1)} - f^{(l)}}{s^{(l)}} \right\rfloor + 1$
  - $w^{(l)} = \left\lfloor \frac{w^{(l-1)} - f^{(l)}}{s^{(l)}} \right\rfloor + 1$
  - $c^{(l)} = c^{(l-1)}$

Heights and widths are calculated in the same way as for the convolution layer. The padding is omitted because it is almost always zero, but otherwise, the calculation is the same. The filter goes through each channel in the input and forms a new channel in the output, so the number of channels in the output is the same as in the input. Since the filter does not contain any numbers that would be included in the network calculation, it has no values to update. So the pooling layer has no parameters to learn.

3.2.3 Batch Normalization

This Section 3.2.3 is based on the Section *3.8.4 Batch Normalization* of the book [7].

Let us first introduce the hyperparameters of batch normalization layer l:

- Mean: μ
- Variance: σ^2
- Epsilon: ϵ

Here the mean and variance can be learned from the training set using the exponential moving average method. The epsilon is a very small constant which just ensures that we do not divide by zero.

The size of the input and output matrix of the batch normalization layer l is the same because only the values are modified, but everything else remains the same:

- Input: $h^{(l-1)} \times w^{(l-1)} \times c^{(l-1)}$
- Output: $h^{(l)} \times w^{(l)} \times c^{(l)}$
  - $h^{(l)} = h^{(l-1)}$
  - $w^{(l)} = w^{(l-1)}$
  - $c^{(l)} = c^{(l-1)}$

Third, we introduce the parameters of the batch normalization layer:

- Scale: γ
- Shift: β

Using these trainable parameters, we can rescale the values of the neurons and move them, for example, in a coordinate system or on a line.

When the convolutional network is trained all its distributions between different neurons are different and the distributions are also different in different layers. The distributions also change with training cycles. This is problematic for the convergence rate of the network, which is why it decreases. In 2015, Ioffe and Szegedy discovered a technique to solve this problem. It is based on the normalisation of mean and variance:

$$z = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}\gamma + \beta, \tag{57}$$

where this equation:

$$z = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} \tag{58}$$

normalizes the input values x using mean μ and variance σ^2. The output z is then more comparable with the other z values in the network. Then there are two parameters used to modify the output, γ is used for linear scaling and β is used for shifting. typically, the batch normalization layer is set before the activation function of the fully connected or convolution layer.

3.2.4 Dropout

This Section 3.2.4 is based on the Section *3.6.3.4 Dropout* of the book [7].

Let us first introduce the hyperparameters of the dropout layer:

- Dropout ratio: p
- Keeping ratio: $1 - p$

Each neuron in the dropout layer is assigned a random value between 0 and 1 according to a uniform distribution, let us call it the dropout number. Dropout ratio is defined by the network designer and it is usually set to 0.5. If a neuron receives a value that is less than the dropout rate, it is dropped from that training round. In the next layer, the weighted sums of neurons are divided by the keeping ratio so that the weighted sums are on the same scale as if each neuron were active in the network.

The size of the output matrix is the same as the input matrix because only some of the neurons drop out, so the information received by the output neuron may be obtained from a smaller number of neurons, but the number of neurons in the next layer does not change by the dropout.

- Input: $h^{(l-1)} \times w^{(l-1)} \times c^{(l-1)}$
- Output: $h^{(l)} \times w^{(l)} \times c^{(l)}$
  - $h^{(l)} = h^{(l-1)}$
  - $w^{(l)} = w^{(l-1)}$
  - $c^{(l)} = c^{(l-1)}$

In the dropout layer, there are no parameters to learn. The dropout technique is an attempt to avoid overfitting the network. Some of the neurons in the original network are dropped out, i.e. they are then inactive. A forward and backward calculation is then performed on the new smaller network, updating the parameters that are present in the network and then the new updated values are added to the original network and the same is repeated for different subsets of the network where each neuron in dropout layer has a new dropout number. Often dropout layers are after the fully connected layers.

3.3 Parameters

In convolutional neural networks the basic idea is to reduce the number of different parameters so the network can be deeper [7]. When designing ConvNet, we should always calculate the trainable parameters, because if we have a lot of parameters to train and only a little data, ConvNet may not generalize well enough to the test set [7].

3.4 Forward Computation

This Section 3.4 is based mainly on the lecture material of Ion Petre [10].

Let us use the structure of LeNet-5, which is also presented later (3.6.1), as an example for our forward computation. So we have input image $\boldsymbol{X} \in \mathbb{R}^{h \times w \times c}$, where h, w, c are height, width, and number of channels respectively. Then we have a filter $\boldsymbol{F} \in \mathbb{R}^{f \times f \times c}$, where f is height and width, which typically are the same size and the number of channels is the same as in input image. Between these two matrices is the convolutional operator, we denote it by $*$. The filter $\boldsymbol{F}$ is multiplied element-wise with the input matrix and the output is stored for a given neuron in the output matrix. Let us see what the convolution operation looks like in mathematical terms:

$$z^t_{m,n} = (\boldsymbol{X} * \boldsymbol{F})^t_{m,n} = \sum_{i=1}^{f} \sum_{j=1}^{f} \sum_{c=1}^{n_c} X_{m+i-1,n+j-1,c} \cdot F^t_{i,j,c} \tag{59}$$

where $z^t_{m,n}$ is the output value stored in the neuron at (m, n), where $m, n = 1, \ldots, 28$, of the output matrix t, where $t = 1, \ldots, n_t$ is the number of used filter in the convolution operation. This equation makes a couple of assumptions, that padding is zero and stride is one so then $1 \leq m \leq h - f + 1$ and $1 \leq n \leq w - f + 1$. Letter $c = 1, \ldots, n_c$ represents here the number of the channel of the input image. The maximum values of i and j depend on how big a filter we have in this operation. Once the convolution operation has been done to the entire input matrix then the output matrices are activated by the activation function.

Next, the output matrices are going to the pooling layer. In LeNET-5 there is a pooling layer with a pooling filter size of 2×2 and stride is 2. From the region of the input matrix that the pooling filter is covering, we select the maximum value for our output matrix. Since the size of the filter is 2×2 and the stride is also 2, the pooling filter covers each neuron in the input matrices once. The same filter is used for all the input matrices, and the number of output matrices is the same as the input. In other words, in this case, the pooling filter is used 14×14 per each 6 matrices, so the output is 6 different 14×14 matrices. After the second convolutional and pooling layer, the 3-dimension matrix is converted to a flattened shape (same as a vector), and then there are a couple of fully connected layers that are working as we already presented in Section 2.4. After fully connected layers, the softmax function is used to obtain the probabilities for each neuron in the last layer, and the neuron with the highest probability is output from the program.

3.5 Training with Backpropagation

3.5.1 In Convolutional Layer

This Section 3.5.1 is based mainly on the Section *3.1.2 Backpropagation of Convolution Layers* of the book [7].

In a convolutional layer, backpropagation is broadly similar to fully connected layers, but of course, because there are fewer connections and weight sharing, there are also differences in backpropagation.

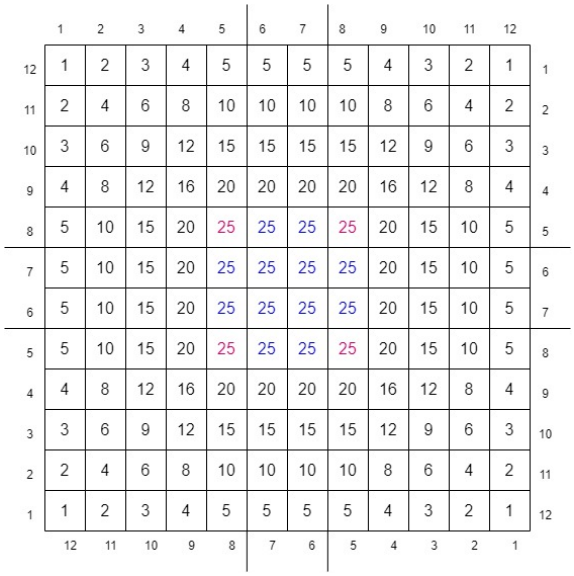


Figure 11: Number of outgoing connections of the input matrix.

The neurons in the input matrix have at least one connection from each neuron and no more than the number of neurons in one filter layer. So, mathematically speaking, outgoing connection (OC):

$$1 \leq OC \leq f \times f, \tag{60}$$

where OC is:

$$OC = \min(i, f) \cdot \min(j, f), \tag{61}$$

where $i, j \geq 1$ are a kind of coordinates from the nearest corner. If there is a stride, then we can think of a single coordinate displacement as being one stride long. Let us take a simple example. Consider the top left corner of the matrix. typically, only one neuron in a corner, at position $(1, 1)$, has only one connection, but if there is a stride, the area with only one connection increases, $(1 \times \text{stride}, 1 \times \text{stride})$. The bigger the stride, the fewer the connections. Figure 11 illustrates the outgoing connections of the input matrix in the convolution layer. The side has coordinates from one to twelve, which can also be thought of as strides. If the stride is greater than one, multiply the coordinate by the stride. We can see that there are more connections in the center of the matrix than in the sides. As a result, the things in the middle of the picture are more important than the things near the edges.

The number of incoming connections (IC) for each neuron in the output matrix is exactly equal to the number of neurons in all layers of the filter. This is mathematically speaking:

$$IC = f \times f \times c. \tag{62}$$

We can assume that the local gradient shown in Section 2.6 is calculated for the

unactivated output matrix of this convolution layer:

$$\delta_{m,n}^t = \frac{\partial E}{\partial z_{m,n}^t}, \tag{63}$$

where $\delta_{m,n}^t$ is a local gradient for the neuron $z_{m,n}$ in output matrix t. Now the gradient of error function with respect to $w_{i,j,c}^t$ is:

$$\frac{\partial E}{\partial w_{i,j,c}^t} = \sum_{m=1}^{M}\sum_{n=1}^{N} \frac{\partial E}{\partial z_{m,n}^t} \cdot \frac{\partial z_{m,n}^t}{\partial w_{i,j,c}^t} = \sum_{m=1}^{M}\sum_{n=1}^{N} \delta_{m,n}^t \cdot \frac{\partial z_{m,n}^t}{\partial w_{i,j,c}^t}, \tag{64}$$

where

$$z_{m,n}^t = (\boldsymbol{X} * \boldsymbol{F})_{m,n}^t \tag{65}$$

is a convolution operation between the input matrix of the convolution layer and the filter and $w_{i,j,c}^t$ is the weight value of the filter, so:

$$\frac{\partial z_{m,n}^t}{\partial w_{i,j,c}^t} = \frac{\partial (\boldsymbol{X} * \boldsymbol{F})_{m,n}^t}{\partial F_{i,j,c}^t}, \tag{66}$$

and then:

$$\frac{\partial E}{\partial w_{i,j,c}^t} = \sum_{m=1}^{M}\sum_{n=1}^{N} \delta_{m,n}^t \cdot X_{i+m-1,j+n-1,c}. \tag{67}$$

This can also be seen as a convolution operation between $\boldsymbol{X}$ and $\boldsymbol{\delta}$:

$$(\boldsymbol{X} * \boldsymbol{\delta})_{i,j,c}^t, \tag{68}$$

so if we convolve the incoming matrix $\boldsymbol{X}$ with the local gradients $\boldsymbol{\delta}$ of the outgoing matrix, we obtain a matrix containing the gradients of the error function with respect to the weights of the filter $\frac{\partial E}{\partial \boldsymbol{W}}$.

We also need to propagate the error to the previous layers, and for that, we also need to calculate $\frac{\partial E}{\partial \boldsymbol{X}}$. This can be done with the chain rule as before:

$$\frac{\partial E}{\partial X_{h,k,c}} = \frac{\partial E}{\partial z_{m,n}^t} \cdot \frac{\partial z_{m,n}^t}{\partial X_{h,k,c}} = \sum_{i=1}^{f}\sum_{j=1}^{f} \delta_{h-f+i,k-f+j}^t \cdot F_{1+f-i,1+f-j,c}^t. \tag{69}$$

where

$$\delta_{h-f+i,k-f+j}^t = \begin{cases} \delta_{h-f+i,k-f+j}^t, & \text{if } 1 \leq h-f+i \leq M \\ & \text{and } 1 \leq k-f+j \leq N \\ 0, & \text{else.} \end{cases} \tag{70}$$

This is same as the equation:

$$\boldsymbol{\delta}^t * flip(\boldsymbol{F}^t), \tag{71}$$

where $flip(\cdot)$ is a function that flips the matrix $\boldsymbol{F}^t$ backwards. So the gradient of the error function with respect to the incoming matrix of the convolution layer $\frac{\partial E}{\partial \boldsymbol{X}}$ is thus obtained by convolving the local gradients of the outgoing matrix $\boldsymbol{\delta}^t$ with a matrix containing the filter values of the convolution layer in inverse order $flip(\boldsymbol{F}^t)$.

3.5.2 In Pooling Layer

This Section 3.5.2 is based mainly on the Section *3.2.1 Backpropagation in Pooling Layer* of the book [7].

In the pooling layer, there is typically used maximum or average pooling. From the area that the pooling filter is covering only one neuron is connected to the output matrix in the case of the maximum pooling. In general, the stride is $s = f(= 2)$, so there are only $\frac{1}{f \cdot f}$ neurons in the input matrix that are connected to the output matrix. On the other hand, this connection is fully and direct from neuron to neuron. In average pooling, each neuron is connected to the output matrix, but in a sense, their share of the connection is only $\frac{1}{f \cdot f}$.

Again, we can assume that we already know the local gradients of the output matrix of this pooling layer:

$$\delta^t_{m,n} = \frac{\partial E}{\partial z^t_{m,n}}. \tag{72}$$

Now we have no trainable parameters to update in the pooling layer, so it is sufficient to compute the gradients of the error function with respect to the neurons in the input matrix.

$$\frac{\partial E}{\partial X^t_{h,k}} = \frac{\partial E}{\partial z^t_{m,n}} \cdot \frac{\partial z^t_{m,n}}{\partial X^t_{h,k}}, \tag{73}$$

where

$$z^t_{m,n} = \max(X^t_{ms-s+p,ns-s+q}), \tag{74}$$

where s is stride and $p, q = 1, \ldots, f$ runs every index of the pooling filter. Now

$$\frac{\partial z^t_{m,n}}{\partial X^t_{h,k}} = \begin{cases} 1, & \text{if } \max(X^t_{ms-s+p,ns-s+q}) = X^t_{h,k} \\ 0, & \text{else.} \end{cases} \tag{75}$$

Now we can write

$$\frac{\partial E}{\partial X^t_{h,k}} = \frac{\partial E}{\partial z^t_{m,n}} \cdot \frac{\partial z^t_{m,n}}{\partial X^t_{h,k}} = \delta^t_{m,n} \cdot 1 = \delta^t_{m,n}, \tag{76}$$

this means that when a neuron is selected based on its maximum value and is connected to the next layer, in backpropagation that neuron transfers the local gradient to the previous layer, and neurons that are not connected do not transfer the local gradient to the previous layer because these gradients are zero $\delta^t_{m,n} \cdot 1 = 0$.

Next, we consider the case of average pooling. In average pooling, the values of the output matrix are obtained as follows:

$$z^t_{m,n} = \frac{1}{f \cdot f} \sum_{p=1}^{f} \sum_{q=1}^{f} X^t_{ms-s+p,ns-s+q}, \tag{77}$$

and if we now take the gradient of the output matrix with respect to the input matrix, we get:

$$\frac{\partial z^t_{m,n}}{\partial X^t_{h,k}} = \frac{1}{f \cdot f} \cdot 1 = \frac{1}{f \cdot f}. \tag{78}$$

This can be applied to equation 76 to obtain:

$$\frac{\partial E}{\partial X^t_{h,k}} = \delta^t_{m,n} \cdot \frac{1}{f \cdot f}. \tag{79}$$

This means that from the local gradient $\delta^t_{m,n}$, of the output matrix neuron, there is $\frac{1}{f \cdot f}$ share for each neuron in the input matrix that is connected to that output matrix neuron. Also, if the stride $s < f$, many neurons in the input matrix can receive their share of several local gradients, because they are then connected to several neurons in the input matrix.

3.6 Examples

3.6.1 LeNet-5

This Section 3.6.1 is based mainly on the Section *3.3 LeNet* of the book [7] and on the Section *10.4 Hands-On LeNet-5* of the book [13] and also on the lecture material of Ion Petre [10].

As early as 1979, Kunihiko Fukushima proposed the ConvNet, an artificial neural network of complex and simple cells, very similar to the convolution and pooling layers we have today [7]. In 1989, LeCun and others proposed the weight sharing method and derived the modern convolution and pooling layers [7]. Then they designed a ConvNet called LeNet-5 [7]. LeNet-5 was designed to recognize handwritten digits and next, we present it here.

LeNet-5 is illustrated in Figure 12. It takes images of size $32 \times 32 \times 1$, which is the size of the input layer. After the input layer, there is a convolution layer with a stride of one and six filters of size $5 \times 5 \times 1$. The output of the first convolution is $28 \times 28 \times 6$, as calculated in Section 3.2.1. Next, these six matrices go to the pooling layer. The pooling layer is the maximum pooling layer with a stride of

two and a filter size of 2×2. The output of this pooling layer is $14 \times 14 \times 6$, as calculated in Section 3.2.2. Next, there is a second convolution layer with 16 different filters, and since the input of this layer is six different matrices, each filter is also six-dimensional. The stride is one, so the output of this convolution layer is $10 \times 10 \times 16$. Next, there is again the maximum pooling layer with a stride of two and filter size of 2×2. The output of the pooling layer is $5 \times 5 \times 16$. These matrices are converted to vector or flattened form because next we have three fully connected layers. So after conversion, we have 400 neurons fully connected to the next layer, which has 120 neurons fully connected to the next layer which has 84 neurons fully connected to the 10 neurons in the next layer. The last layer therefore has ten neurons which is the same number as the number of classes to be classified. The softmax function is used to select the neuron/class from the ten with the highest probability score.

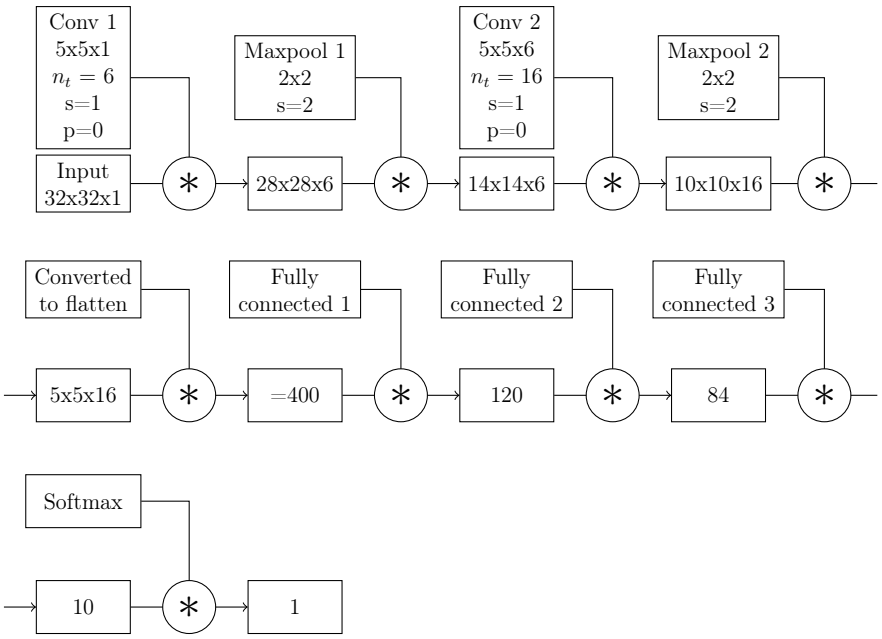


Figure 12: The structure of LeNet-5. The asterisk denotes the operations applied to the matrix along the path of the network structure. The upper boxes, connected to the asterisks, contain the necessary information about the operation applied to the matrix. For example, the first box tells the convolution operation and the filter size, number of filters, stride and padding.

Since the number of learning parameters is a very important piece of information about ConvNet, we next compute it as described in Section 3.2. These parameters per layer are also calculated in Figure 13. The first convolution layer contains 6, 5×5, filters. Each filter therefore has $5 \cdot 5$ parameters and one bias parameter, i.e. $6 \cdot (5 \cdot 5 + 1) = 156$ parameters. Pooling layers have no learning parameters, so nothing needs to be calculated. The second convolution layer has 16, $5 \times 5 \times 6$, filters and the bias parameter of each filter, i.e. $16 \cdot (5 \cdot 5 \cdot 6 + 1) = 2416$. The first fully connected layer has 400 different neurons and one bias, each with one connection to the next layer. This can also be thought of as 120 different filters of size 400. The first fully connected layer has $120 \cdot (400 + 1) = 48,120$ parameters. The

following layers are calculated in the same way. So there are $84 \cdot (120 + 1) = 10,164$ parameters in the second fully connected layer and $10 \cdot (84 + 1) = 850$ parameters in the third fully connected layer. The LeNet-5 network therefore has a total of $156 + 2416 + 48,120 + 10,164 + 850 = 61,706$ parameters.

Layer	Input	Output	# parameters
Input	32x32x1	32x32x1 (= 1024)	0
Conv. 1	32x32x1	28x28x6 (= 4704)	**6*(5*5*1+1)=156**
Pool 1	28x28x6	14x14x6 (= 1176)	0
Conv. 2	14x14x6	10x10x16 (= 1600)	**16*(5*5*6+1)=2416**
Pool 2	10x10x16	5x5x16 (= 400)	0
FC 1	400	120	**120*(400+1)=48,120**
FC 2	120	84	**84*(120+1)=10,164**
FC 3	84	10	**10*(84+1)=850**
Output	10	1	0

Figure 13: The layers of LeNet-5 and the input and output size of the layers and the number of trainable parameters on that layer.

3.6.2 AlexNet

This Section 3.6.2 is based mainly on the Section *3.4 AlexNet* of the book [7] and on the original article [1].

In 2012, Alex Krizhevsky and others trained the ImageNet dataset using the large ConvNet, which won an image classification competition [7]. The challenge was to identify 1000 different categories of the natural objects [7]. Afterward, this ConvNet became popular and was called AlexNet [7] and next, we present it here.

AlexNet is illustrated in Figure 14. It takes colour images as input, so the input matrix is three-dimensional. The size of the input images can be $224 \times 224 \times 3$, which is the size of the input layer.

The first layer after the input layer is the convolution layer. This layer contains 96 different filters of size $11 \times 11 \times 3$ and with a stride four. The output of the first convolution layer is $55 \times 55 \times 96$. This is followed by a maximum pooling layer with 3×3 filters and stride two. Then the neural network sort of splits in two in terms of depth. Then the second convolution layer takes 96 matrices in two parts so also the size of the filters is smaller. There are 256 filters of size $5 \times 5 \times 48$ and stride one. The first 128 filters are convolving the first 48 matrices and the second 128

filters second 48 matrices. This convolution layer also includes padding of size two, so the size of the input matrices after padding is 31×31. Then the second pooling layer is also the maximum type with a stride of two and filter size of 3×3. After pooling we have 256 matrices with a size of 13×13. The third convolution layer has 384 filters with a size of $3 \times 3 \times 256$ and a stride of one and a padding of one. The third convolution layer is not split, but its output is split because the fourth and fifth convolution layers are split. The fourth convolution layer is similar to the third, but the filters are 192-dimensional and the convolving is in two parts. Next is the fifth convolution layer, which is similar to the fourth, but with 256 filters. Then there is the third pooling layer, which is also the maximum type, with stride two and filter size 3×3. After this, we have 256 matrices of size 6×6. These matrices are converted to a vector of size 9216. This vector is fully connected to the next layer, which has 4096 neurons, between these layers there is also dropout in use. Then there is the second fully connected layer which also has 4096 neurons and also in this layer, there is dropout in use. The third fully connected layer is the classification layer with 1000 neurons. The softmax function gives us the class with the highest probability score out of these 1000 classes.

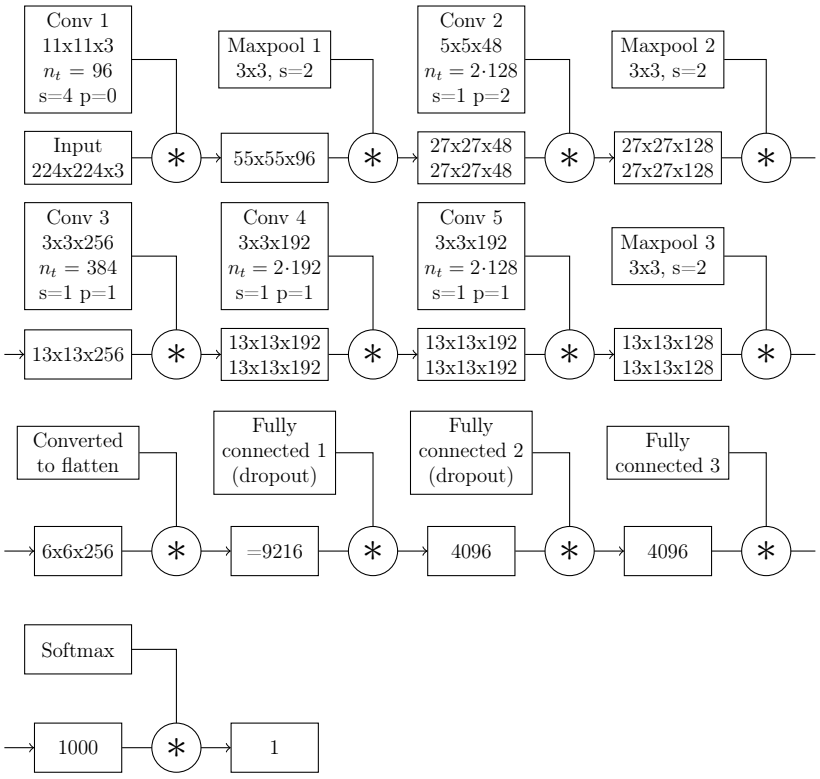


Figure 14: The structure of AlexNet. The asterisk denotes the operations applied to the matrix along the path of the network structure. The upper boxes, connected to the asterisks, contain the necessary information about the operation applied to the matrix. For example, the first box tells the convolution operation and the filter size, number of filters, stride and padding. The matrix inside the network path is also split and merged twice. After the first two fully connected layers there are also dropouts in use.

Let us now compute the learning parameters of AlexNet, which are also shown in Figure 15 below. The first convolution layer has 96 filters of size $11 \times 11 \times 3$ so there are $96 \cdot (11 \cdot 11 \cdot 3 + 1) = 34,944$ parameters in that layer. The second convolution layer has 256 filters in two parts, each with 128 filters, of size $5 \times 5 \times 48$ so that layer contains $128 \cdot (5 \cdot 5 \cdot 48 + 1) \cdot 2 = 307,456$ parameters. The third convolution layer has 384 filters of size $3 \times 3 \times 256$ so that layer consists of $384 \cdot (3 \cdot 3 \cdot 256 + 1) = 885,120$ parameters. The fourth convolution layer has 384 filters in two parts of size $3 \times 3 \times 192$ so there are $192 \cdot (3 \cdot 3 \cdot 192 + 1) \cdot 2 = 663,936$ parameters in that layer. The fifth convolution layer has 256 filters in two parts of size $3 \times 3 \times 192$ so there are $128 \cdot (3 \cdot 3 \cdot 192 + 1) \cdot 2 = 442,624$ parameters in that layer. Next, there are three fully connected layers, where the first fully connected layer has different weighted connections from 9216 neurons and one bias neuron to 4096 different neurons in the next layer, so there are $4096 \cdot (9216 + 1) = 37,752,832$ different parameters. The second fully connected layer follows the same procedure with $4096 \cdot (4096 + 1) = 16,781,312$ different parameters. The third fully connected layer has $1000 \cdot (4096 + 1) = 4,097,000$ different parameters. That makes a total of $34,944 + 307,456 + 885,120 + 663,936 + 442,624 + 37,752,832 + 16,781,312 + 4,097,000 = 60,965,224$ parameters for AlexNet.

Layer	Input	Output	# parameters
Input	224x224x3	224x224x3 (= 150,528)	0
Conv. 1	224x224x3	55x55x96 (= 290,400)	**96*(11*11*3+1)=34,944**
Pool 1	55x55x96	27x27x48 (= 34,992) 27x27x48 (= 34,992)	0
Conv. 2	27x27x48 27x27x48	27x27x128 (= 93,312) 27x27x128 (= 93,312)	**128*** **(5*5*48+1)*2=307,456**
Pool 2	27x27x128 27x27x128	13x13x256 (= 43,264)	0
Conv. 3	13x13x256	13x13x192 (= 32,448) 13x13x192 (= 32,448)	**384*** **(3*3*256+1)=885,120**
Conv. 4	13x13x192 13x13x192	13x13x192 (= 32,448) 13x13x192 (= 32,448)	**192*** **(3*3*192+1)*2=663,936**
Conv. 5	13x13x192 13x13x192	13x13x128 (= 21,632) 13x13x128 (= 21,632)	**128*** **(3*3*192+1)*2=442,624**
Pool 3	13x13x128 13x13x128	6x6x256 (= 9216)	0
FC 1	9216	4096	**4096*** **(9216+1)=37,752,832**
FC 2	4096	4096	**4096*** **(4096+1)=16,781,312**
FC 3	4096	1000	**1000*** **(4096+1)=4,097,000**
Output	1000	1	0

Figure 15: The layers of AlexNet and the input and output size of the layers and the number of trainable parameters on that layer.

Finally, a small comparison between LeNet-5 and AlexNet.

	Scale and Application	Architecture Complexity	Technological Innovations
LeNet-5	Small-scale tasks, simple datasets like MNIST.	Simpler, fewer layers and parameters.	Early use of CNNs, basic layers.
AlexNet	Large-scale tasks, complex datasets like ImageNet.	Deeper, more layers, significantly more parameters.	Introduced ReLU activation, dropout for regularization and GPU acceleration.

Figure 16: A small comparison of LeNet-5 and AlexNet.

4 Coding Neural Networks

Next, we present two standard libraries that are very popular in the field of machine learning.

4.1 Tensorflow Library

This section is written mainly from information found on the website of TensorFlow [17]. With TensorFlow, you can easily build AI-powered applications. TensorFlow supports devices such as desktops, mobile devices, web browsers and cloud services. It provides ready-to-use datasets. In TensorFlow 2.0.0, Keras was merged with TensorFlow [18]. By default, TensorFlow will try to run on the GPU if one is available, if not, on the CPU [19]. The library also provides the option to choose which GPU or CPU to use if there are multiple [19].

4.2 Keras Library

This section is written mainly from information found on the website of Keras [11]. Keras is a simple, flexible and powerful multi-framework deep learning API written in Python and can run on top of JAX, TensorFlow and PyTorch. Keras library provides support to for example computer vision (as in this thesis), natural language processing, generative deep learning and audio data. The API contains, for example, the implementation of neural network models, model layers, model optimizers, model activations, model metrics and losses, and dataset loaders. Pretrained models are also available, where the weights are preset.

5 Application

The code used in the following applications is for the time being available in my GitHub repository https://github.com/JarnoJappinen/ANN-CNN.

5.1 CIFAR-10

The CIFAR-10 database consists of 60,000 color images, $32 \times 32 \times 3$ in size. There are 10 different classes and every class has 6,000 images with 5,000 training and 1,000 testing images per class, the dataset is well balanced. The classes are: airplane, automobile (but not truck or pickup truck), bird, cat, deer, dog, frog, horse, ship,

and truck (but not pickup truck) [4]. Figure 17 shows some sample images and their labels from the CIFAR-10 training dataset. In the next sections, we try to train our model to recognize images that include those classes. First, we try it with a simple fully connected network and then a more complicated convolution network.

Figure 17: Random images from the CIFAR-10 training dataset.

5.1.1 With ANN

In the code, 50,000 test images are divided into 40,000 for training and 10,000 for validation. In our model, which comes from Ion Petre's GitHub [8], we have after the input layer three fully connected layers and an output layer. They have 128, 64, 32, and 10 neurons. They have the ReLU activation function in the hidden layers and the softmax in the output layer. This makes a total of 404,010 trainable parameters, as also shown in Figure 18.

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 3072)	0
dense (Dense)	(None, 128)	393344
dense_1 (Dense)	(None, 64)	8256
dense_2 (Dense)	(None, 32)	2080
dense_3 (Dense)	(None, 10)	330

Total params: 404,010
Trainable params: 404,010
Non-trainable params: 0

Figure 18: Parameters of the model for the CIFAR-10 dataset (ANN).

Figure 19 shows different metrics of the model. The graph on the left shows training loss with a blue line and validation loss with a red line. Initially, both are trending downwards, but already at around ten epochs the validation loss stops decreasing, and after fifty epochs it even starts to increase. At the same time, the training loss continues to decrease. In the middle graph, the training accuracy is shown by a blue line and the validation accuracy by a red line. Somewhere around ten epochs, the blue line continues to rise, but the red line slows down and soon after

fifty epochs it goes down more than it remains stable. In the third graph on the right, a blue line shows training true positives and a red line shows validation true positives. Again, at around ten epochs, the training line continues to rise, while the validation line starts to slow down. We can easily see that the model is overfitting, as the training values keep improving while the validation values are deteriorating.

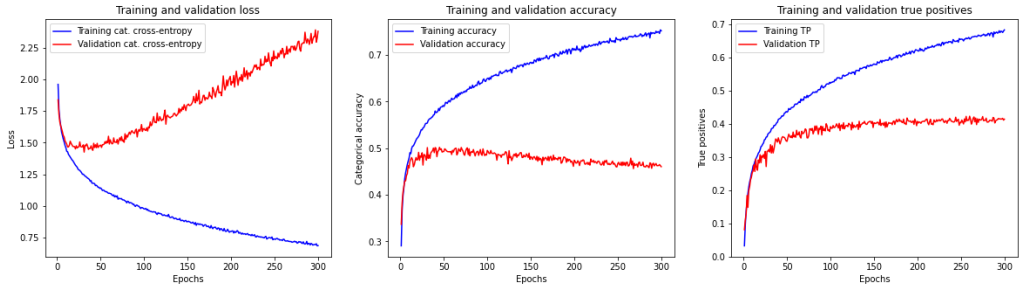


Figure 19: Metrics of the model.

Figure 20 is a confusion matrix of the validation data. It shows the percentage of predictions for each category. On the diagonal of the matrix are the correct predictions. In each row, the number on the diagonal is the largest, which is good, and five out of ten numbers on the diagonal are above 50 %.

```
[[52  5  5  5  4  3  2  6 10  8]
 [ 6 53  3  3  2  2  2  3  6 20]
 [ 9  3 35  8 14 10  8 10  2  2]
 [ 3  2 10 30  5 23 10  8  4  5]
 [ 7  2 15  8 35  9  9 12  3  2]
 [ 3  2  9 20  6 39  5 10  2  3]
 [ 2  2 10 11  9 11 47  4  1  3]
 [ 4  2  7  6  8  8  2 58  2  4]
 [13  8  3  2  2  2  1  2 58  8]
 [ 5 18  2  4  2  3  2  7  6 51]]
```

Figure 20: A confusion matrix of validation data, results are in percentages.

Figure 21 shows nine different randomly selected images from the validation dataset. Below the images are labeled with the prediction and prediction percentage and in parenthesis the actual labeling of the images. If the prediction is correct the label is blue and if the prediction is wrong the label is red. For each image, there is also a bar chart where the bars represent the probability of the categories for the image label.

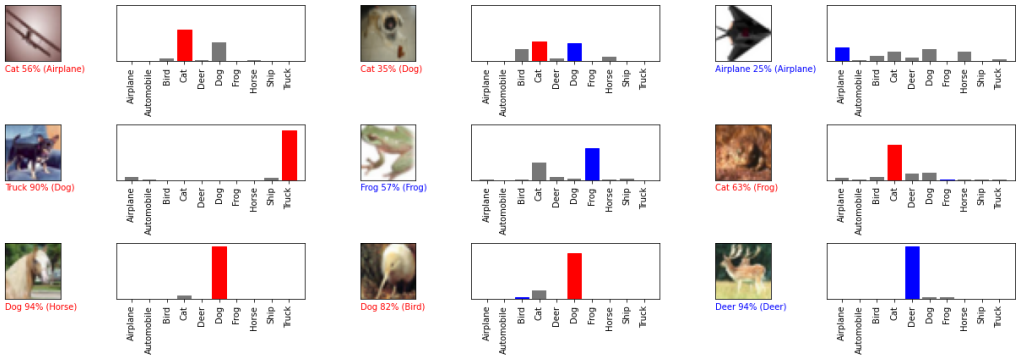


Figure 21: Random validation images and their predictions. Correct prediction in blue and incorrect prediction in red.

5.1.2 With CNN

Here we have the same dataset, divided in the same way as in Section 5.1.1 into training, validation and test subsets. The code and model follow the example of Ion Petre's course [9]. The summary of the model is shown in Figure 22. After the input layer, there are 16 layers. The first is a convolutional layer, which has 32 filters of size 3×3 with a padding that makes the output size the same size as the input size. The stride is one and the activation function is ReLU. Then there is a batch normalization layer, which contains two trainable parameters for each filter in the previous layer and two non-trainable parameters, since there are four batch normalization layers in total, two with 32 and two with 64 filters, the model has a total of $2 \cdot (32 \cdot 2) + 2 \cdot (64 \cdot 2) = 384$ non-trainable parameters. The next two layers are the same as layers one and two. Layer five is a max pooling layer with a filter size of 2×2 and the stride is two. Layer six is a dropout layer with a dropout ratio of 0.25, which means that for each round of training, 25 % of the output neurons in layer five are excluded from training. The next six layers are the same as the first six layers, but the convolutional layers have 64 filters instead of 32. Then layer 13 is for flattening and layer 14 is a fully connected layer with 64 neurons and the activation function is ReLU. Then again comes a dropout layer with a dropout ratio of 0.25. And finally, layer number 16 with 10 neurons, and the activation function is the softmax function.

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 32, 32, 32)        896
_________________________________________________________________
batch_normalization (BatchNo (None, 32, 32, 32)        128
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 32, 32, 32)        9248
_________________________________________________________________
batch_normalization_1 (Batch (None, 32, 32, 32)        128
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 16, 16, 32)        0
_________________________________________________________________
dropout (Dropout)            (None, 16, 16, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 16, 16, 64)        18496
_________________________________________________________________
batch_normalization_2 (Batch (None, 16, 16, 64)        256
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 16, 16, 64)        36928
_________________________________________________________________
batch_normalization_3 (Batch (None, 16, 16, 64)        256
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 8, 8, 64)          0
_________________________________________________________________
dropout_1 (Dropout)          (None, 8, 8, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 4096)              0
_________________________________________________________________
dense (Dense)                (None, 64)                262208
_________________________________________________________________
dropout_2 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                650
=================================================================
Total params: 329,194
Trainable params: 328,810
Non-trainable params: 384
```

Figure 22: Ion Petre's CNN model and its parameters.

Figure 23 shows the different metrics of Ion Petre's CNN model. The blue lines are the training results, they look very much the same shape as in Figure 19, but note that the results are better with a much smaller number of epochs. More importantly, however, the red lines showing the validation results are very different and much closer to the blue lines. In the graph on the left, the validation loss drops below 1.0 at around 10 epochs and then remains below 1.0. After that, it does not improve, but it does not get worse either. In the middle graph, the validation accuracy rises above 0.7 at 10 epochs, after which the accuracy does not improve much, but with ANN it does not exceed 0.5. In the right graph, the true positives in the validation dataset rise above 0.7, while with ANN they remain below 0.4.

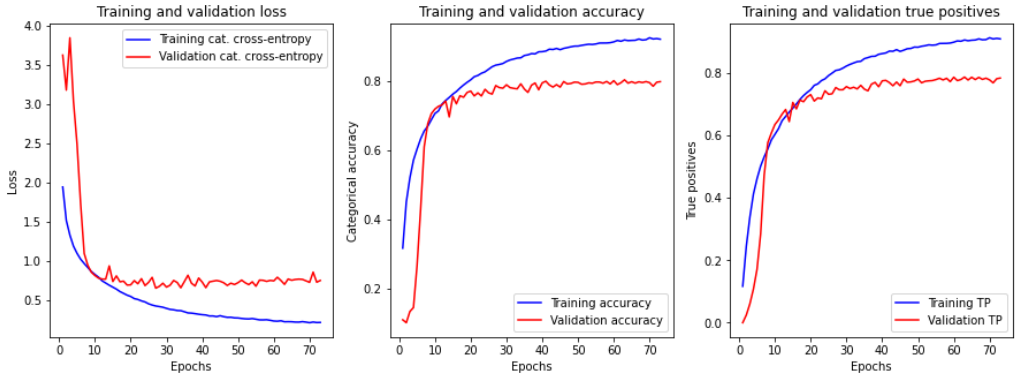


Figure 23: Metrics of the model.

Figure 24 is a confusion matrix of the validation data. Compared to Figure 20, the diagonal now has much higher percentages, all above 50 % and six out of ten above 80 %.

```
[[82  1  3  2  1  1  1  1  4  4]
 [ 1 89  0  0  0  0  0  0  1  8]
 [ 6  0 68  7  5  4  6  2  1  0]
 [ 1  1  5 66  3 15  5  3  1  1]
 [ 2  0  5  7 71  4  4  7  1  0]
 [ 0  0  3 19  2 70  1  3  0  1]
 [ 1  0  4  4  2  2 86  0  1  0]
 [ 1  0  3  4  2  5  0 84  0  1]
 [ 3  2  0  1  0  0  0  0 92  2]
 [ 2  4  0  1  0  0  0  0  0 91]]
```

Figure 24: A confusion matrix of validation data from the Ion Petre's model, results are in percentages.

Figure 25 has many more correct predictions than Figure 21. Seven predictions are correct, while ANN gave only three correct predictions out of nine. There is one interesting image in the middle left where the prediction is almost even with a cat and dog, it could well be a cat.

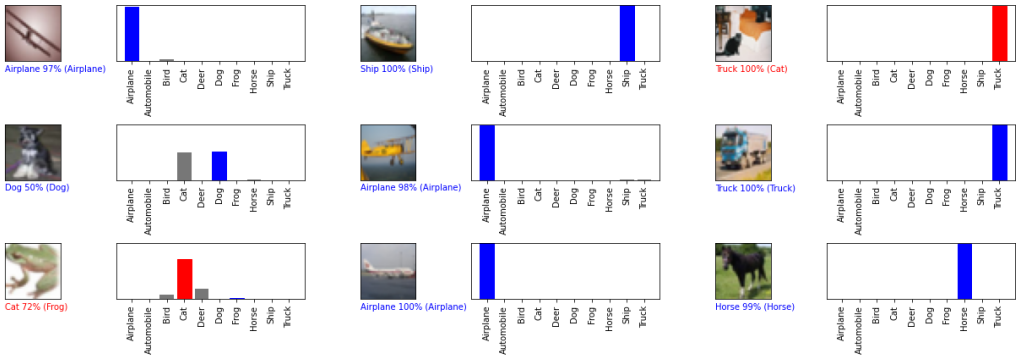


Figure 25: Random validation images and their predictions. Correct prediction in blue and incorrect prediction in red.

5.2 Fashion-MNIST

The Fashion-MNIST database consists of 70,000 grayscale images, $28 \times 28 \times 1$ in size. There are 10 different categories and every category has 7,000 images with 6,000 training and 1,000 testing images per category, also this dataset is well balanced [6]. The categories are: ankle boot, bag, coat, dress, pullover, sandal, shirt, sneaker, t-shirt/top, and trouser [12]. Figure 26 shows some sample images and their labels from the Fashion-MNIST training dataset. In the next sections, we try to train our model to recognize images that include those categories. First, we try it with a simple fully connected network and then a more complicated convolution network.

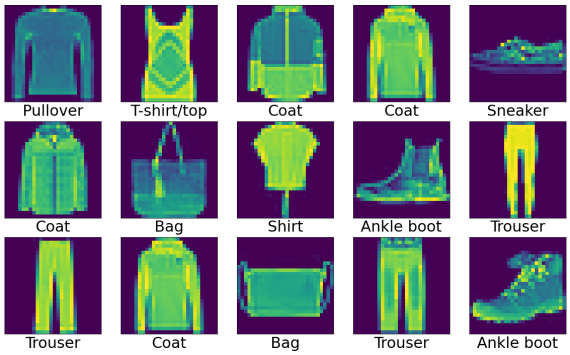


Figure 26: Random images from the Fashion-MNIST training dataset.

5.2.1 With ANN

In the code, 60,000 test images are divided into 48,000 for training and 12,000 for validation. In our model, which comes from Ion Petre's GitHub [8], we have after the input layer three fully connected layers and an output layer. They have 128, 64, 32, and 10 neurons. They have the ReLU activation function in the hidden layers and the softmax in the output layer. This makes a total of 111,146 trainable parameters, as also shown in Figure 18. This is the same model as in Section 5.1.1, with the only difference being the input layer, where fewer neurons are needed.

```
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
dense (Dense)                (None, 128)               100480
_________________________________________________________________
dense_1 (Dense)              (None, 64)                8256
_________________________________________________________________
dense_2 (Dense)              (None, 32)                2080
_________________________________________________________________
dense_3 (Dense)              (None, 10)                330
=================================================================
Total params: 111,146
Trainable params: 111,146
Non-trainable params: 0
```

Figure 27: Parameters of the model for the Fashion-MNIST dataset (ANN).

Figure 28 shows different metrics of the model. If we compare these graphs to the graphs in Figure 19 we can see that the lines look very similar, but we should notice that the values on the x and y axes are much better, so this dataset can be trained with the same model much faster to a better result. With this dataset, the validation accuracy rises above 0.875, and the validation true positives also rise above 0.85.

Figure 28: Metrics of the model for the Fashion-MNIST dataset (ANN).

Figure 29 is a confusion matrix of the validation data from the model for the Fashion-MNIST. Category seven, which is the shirt, has the biggest problem getting correct predictions and also that is more than 70 %. 11 % of those predictions have gone wrongly to the first category which is T-shirt/top. Three categories get more than 80 % and even six categories get more than 90 % correct predictions.

```
[[84  0  2  4  1  0  9  0  1  0]
 [ 0 97  0  2  0  0  0  0  0  0]
 [ 2  0 82  2  7  0  8  0  0  0]
 [ 2  0  1 91  3  0  3  0  0  0]
 [ 0  0  9  3 80  0  8  0  0  0]
 [ 0  0  0  0  0 96  0  3  0  1]
 [11  0  7  4  5  0 73  0  1  0]
 [ 0  0  0  0  0  2  0 96  0  3]
 [ 1  0  0  0  1  0  1  0 96  0]
 [ 0  0  0  0  0  1  0  4  0 96]]
```

Figure 29: A confusion matrix of validation data, results are in percentages.

Figure 30 shows that of the nine predictions, seven were correct and one was almost correct, but the shirt was incorrectly predicted as a T-shirt/top.

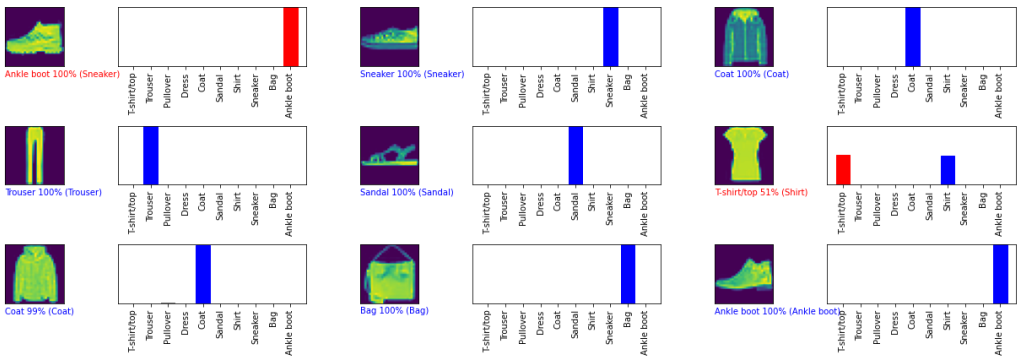


Figure 30: Random validation images and their predictions. Correct prediction in blue and incorrect prediction in red.

5.2.2 With CNN

The model used here is the same as the CIFAR-10 with CNN in Section 5.1.2. The summary of the model is shown in Figure 31. A smaller model could also have been used, such as LeNet-5 with the advantage of being a smaller model the speed of training could be faster. The dataset is the same and divided in the same way as in Section 5.2.1.

Layer (type)	Output Shape	Param #
conv2d_4 (Conv2D)	(None, 28, 28, 32)	320
batch_normalization_4 (Batch	(None, 28, 28, 32)	128
conv2d_5 (Conv2D)	(None, 28, 28, 32)	9248
batch_normalization_5 (Batch	(None, 28, 28, 32)	128
max_pooling2d_2 (MaxPooling2	(None, 14, 14, 32)	0
dropout_3 (Dropout)	(None, 14, 14, 32)	0
conv2d_6 (Conv2D)	(None, 14, 14, 64)	18496
batch_normalization_6 (Batch	(None, 14, 14, 64)	256
conv2d_7 (Conv2D)	(None, 14, 14, 64)	36928
batch_normalization_7 (Batch	(None, 14, 14, 64)	256
max_pooling2d_3 (MaxPooling2	(None, 7, 7, 64)	0
dropout_4 (Dropout)	(None, 7, 7, 64)	0
flatten_1 (Flatten)	(None, 3136)	0
dense_2 (Dense)	(None, 64)	200768
dropout_5 (Dropout)	(None, 64)	0
dense_3 (Dense)	(None, 10)	650

Total params: 267,178
Trainable params: 266,794
Non-trainable params: 384

Figure 31: Ion Petre's CNN model for the Fashion-MNIST dataset and its parameters.

This dataset is very well trained, as we can see in Figure 32, where the red line is very close to the blue line in each graph and these lines show very good results. The number of epochs needed is quite small, but still, for example, the validation accuracy is around 0.9 as are the true positives in this same line. These graphs are definitely the best of the four cases presented here.

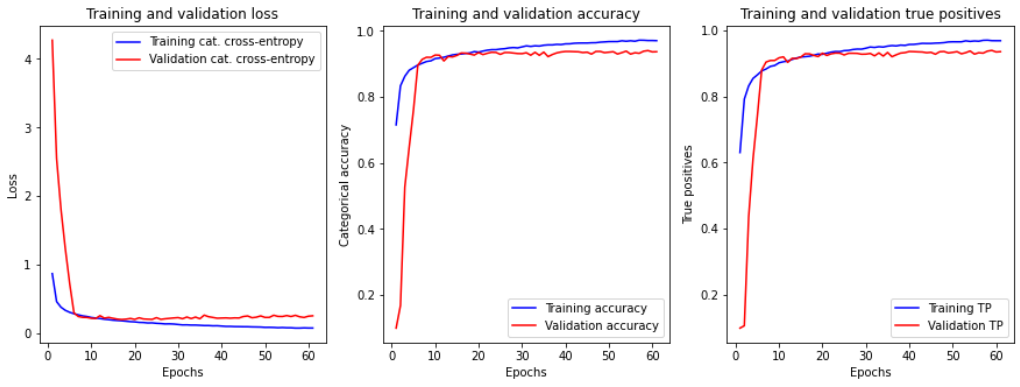


Figure 32: Metrics of the model for the Fashion-MNIST dataset (CNN).

As we can see in Figure 33, this time we have as many as eight out of ten in the ninety percent and above range of correct predictions, and even in the two worse

ones, eighty-six percent of the predictions have gone right. Such a model is already starting to be very usable.

```
[[86  0  2  1  0  0 11  0  0  0]
 [ 0 98  0  1  0  0  0  0  0  0]
 [ 1  0 90  1  3  0  5  0  0  0]
 [ 2  1  1 92  2  0  3  0  0  0]
 [ 0  0  2  1 93  0  3  0  0  0]
 [ 0  0  0  0  0 99  0  1  0  0]
 [ 6  0  3  2  3  0 86  0  0  0]
 [ 0  0  0  0  0  1  0 98  0  1]
 [ 0  0  0  0  0  0  1  0 98  0]
 [ 0  0  0  0  0  0  0  3  0 97]]
```

Figure 33: A confusion matrix of validation data from the Ion Petre's model, results are in percentages.

There is only one incorrect prediction in Figure 34, it has to be said that the picture looks more like an ankle boot than a normal sneaker, so it might not be such a bad mistake. Also, in the correct shirt prediction, some probability points have also gone to the T-shirt/top category, which contains quite similar-looking images.

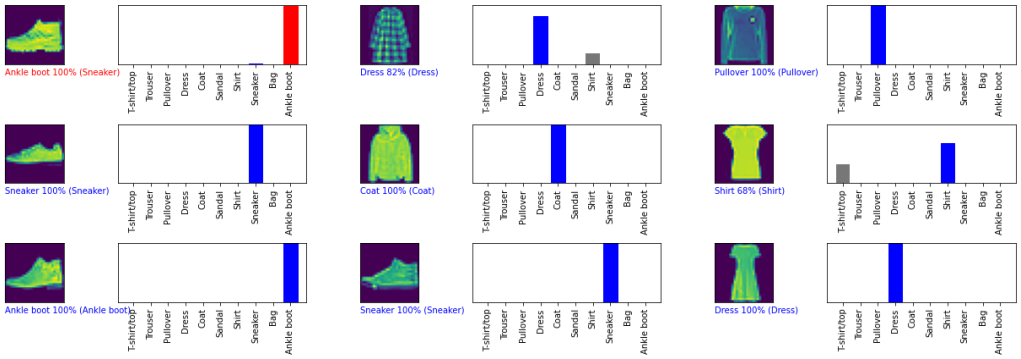


Figure 34: Random validation images and their predictions. Correct prediction in blue and incorrect prediction in red.

5.3 Summary

Here is a small summary of previous applications. As we have already seen and also in Figure 35 we can see that the CNN model gives better results. With CIFAR-10 we also had fewer trainable parameters with CNN than with the fully connected model (ANN). Fashion-MNIST also trains very well with ANN, unlike CIFAR-10. Fashion-MNIST has fewer trainable parameters than CIFAR-10 because it has a smaller input image, which especially affects the fully connected model. Fashion-MNIST images are also easier to train because they are slightly smaller and have only one colour channel.

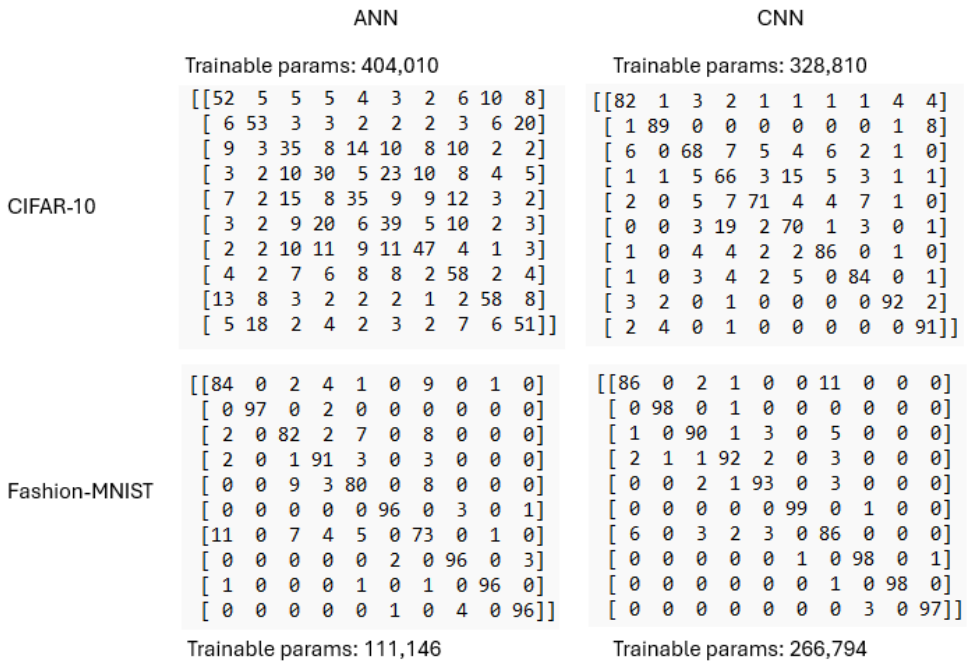


Figure 35: Confusion matrices and trainable parameters for each of the four cases.

6 Conclusions

One of the biggest difficulties with image recognition in the real world is that there are often many different objects in an image. Will the machine be able to recognize all of them or just a specific one? If there can be many objects in one image, there can always be a different object in many images. There are an infinite number of different things in the world to identify. The more versatile the recognition capability you want, the more classes you need to have in the model output layer. The number of neurons in the input and output layers greatly impacts the number of parameters. In these example works they have been quite moderate and still a lot of data has been needed to get even these results and the learning time of the model is quite considerable. Of course, once the model has been trained to its optimum state, the weights can be stored and the model can be used afterward. It is also important for the learning of the model that the images are either captured or processed in such a way that the actual object is as central as possible because it transfers the most information to the layers and that the object is large and there is little else in the image except the thing to be identified.

Just as with humans, you need to see certain things enough times to recognize them. Just as a human can learn to recognize a car with relative ease, but it takes more to recognize certain car models from each other, so it is the same with machine learning. The harder the task, the more data is required and still the recognition is harder. If the classes to be identified are particularly different in shape or colour, identification is much easier.

If the training data gives the model an accuracy close to 100 %, even if the model then overfits the training data, it means that once the model has seen all possible

images related to the topic, it will recognize them with good accuracy in the future. In other words, the size of the data is of great importance.

Training the model is affected by the number of layers. In this work, the fully connected networks had only input + 3 + output layers while the convolution models had input + 15 + output layers. If a fully connected network had a similar number of layers, the number of parameters would be absolutely enormous. Now there are quite a few of them, which makes it easier to do the maths for teaching the model.

References

[1] Alex Krizhevsky, et al. ImageNet classification with deep convolutional neural networks, in: International Conference on Neural Information Processing Systems, 2012, pp. 1097–1105.

[2] Author: Akash Ajagekar (SYSEN 6800 Fall 2021). "Adam". https://optimization.cbe.cornell.edu/index.php?title=Adam, retrieved 24.1.2024

[3] Author: Daniel Villarraga (SYSEN 6800 Fall 2021). "AdaGrad". https://optimization.cbe.cornell.edu/index.php?title=AdaGrad, retrieved 13.1.2024

[4] CIFAR-10 (Canadian Institute for Advanced Research, 10 classes). https://paperswithcode.com/dataset/cifar-10, retrieved 22.4.2024

[5] Ethem Alpaydin. Introduction to Machine Learning. Third edition. Cambridge, Massachusetts: MIT Press, 2014. Print.

[6] Fashion-MNIST. https://paperswithcode.com/dataset/fashion-mnist, retrieved 5.5.2024

[7] Hamed Habibi Aghdam and Elnaz Jahani Heravi. Guide to Convolutional Neural Networks: A Practical Application to Traffic-Sign Detection and Classification. Cham: Springer International Publishing AG, 2017. Print.

[8] Ion Petre. FoundML course assignments (Fall 2023). https://github.com/ionpetre/FoundML_course_assignments/blob/main/FML2w4_deep_learning.ipynb, retrieved 14.4.2024

[9] Ion Petre. FoundML course assignments (Fall 2023). https://github.com/ionpetre/FoundML_course_assignments/blob/main/FML2w5_cnn.ipynb, retrieved 1.5.2024

[10] Ion Petre. Lecture notes "Foundations of Machine Learning II", University of Turku, 2021

[11] Keras. https://keras.io/, retrieved 15.5.2024

[12] Know Your Data: Fashion-MNIST. https://knowyourdata-tfds.withgoogle.com/#dataset=fashion_mnist&tab=STATS&select=default_segment.fashion_mnist.label.value&expanded_groups=fashion_mnist, retrieved 5.5.2024

[13] Liangqu Long and Xiangming Zeng. Beginning Deep Learning with TensorFlow: Work with Keras, MNIST Data Sets, and Advanced Neural Networks. New York, New York: Apress L. P., 2022. Web.

[14] Nadav Cohen. Adam: A Method for Stochastic Optimization, Advanced Seminar in Deep Learning (67679), The Hebrew University of Jerusalem, October 18, 2015 https://moodle2.cs.huji.ac.il/nu15/pluginfile.php/316969/mod_resource/content/1/adam_pres.pdf, retrieved 9.1.2024

[15] Qualtrics. "The complete guide to regression analysis". https://www.qualtrics.com/experience-management/research/regression-analysis/, retrieved 1.2.2024

[16] Sebastian Ruder. "An Overview of Gradient Descent Optimization Algorithms." arXiv.org (2016): n. pag. Web.

[17] TensorFlow. https://www.tensorflow.org/, retrieved 16.5.2024

[18] TensorFlow 2.0.0 - Release 2.0.0. https://github.com/tensorflow/tensorflow/releases/tag/v2.0.0, retrieved 18.5.2024

[19] Use a GPU. https://www.tensorflow.org/guide/gpu, retrieved 21.5.2024