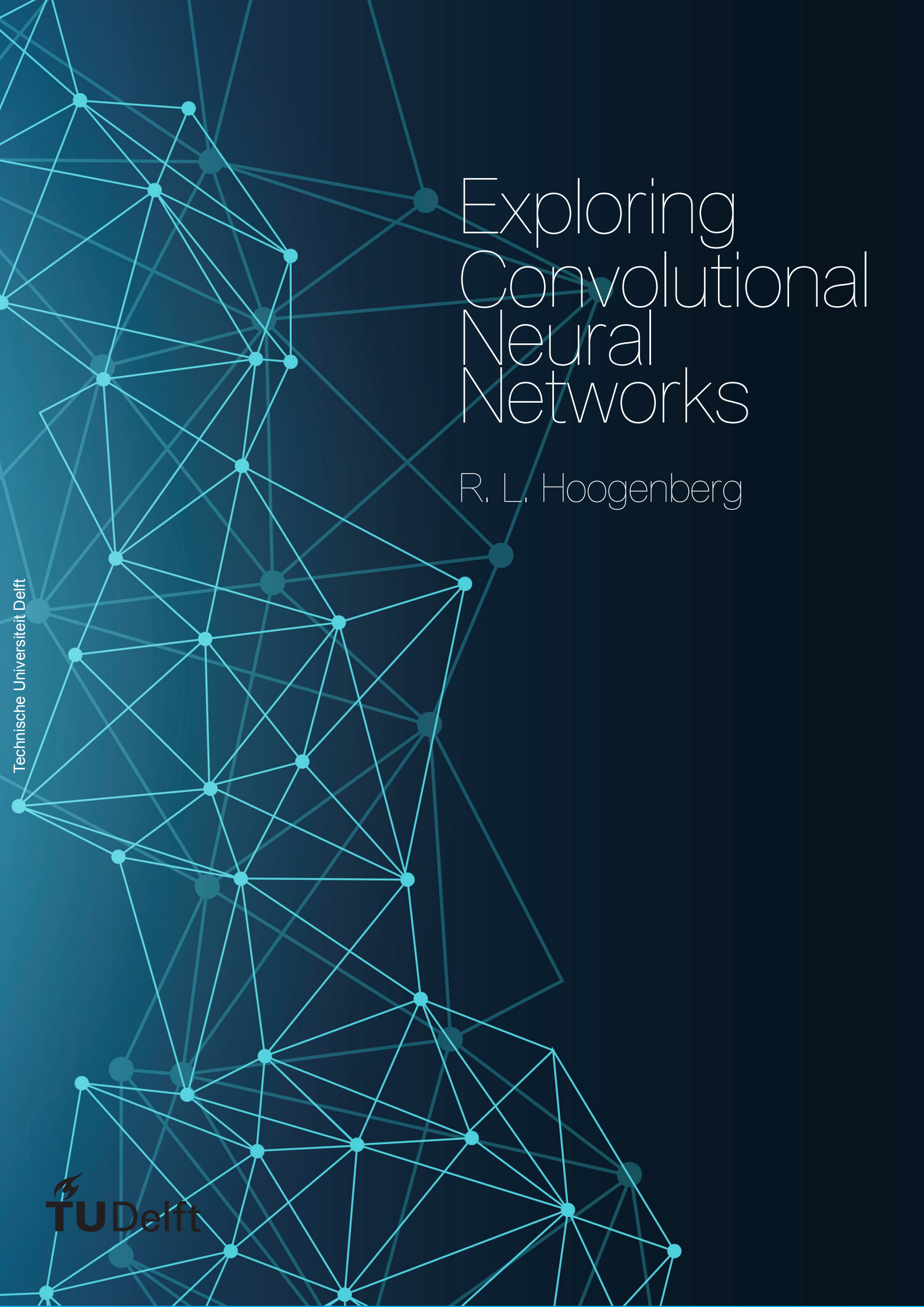

Exploring Convolutional Neural Networks

R. L. Hoogenberg

Technische Universiteit Delft

TU Delft

Exploring Convolutional Neural Networks

by

R. L. Hoogenberg

Title page image by courtesy of rawpixel.com/Freepik [16].

TU Delft

Abstract

In this thesis we have looked into the complexity of neural networks. Especially convolutional neural networks (CNNs), which are useful for image recognition, are looked into. In order to better understand the process in the neural networks, in the first half of this report a mathematical foundation for neural networks and CNNs is constructed. After this, during different experiments on a simple CNN, a better understanding of the variables that can be chosen for the network is gained.

In the first experiment the network was trained on two translated images of a cat in order to test the common believe that CNNs are translation invariant. The result of this experiment refuted this believe and led to the understanding of translation equivariance, with which translation invariance often is confused. After this, the network was trained to learn the difference between a cat and a dog. From this, it appeared that the network can become translation invariant if the right training data is chosen. Also, a look was taken at the kernels of the convolution layer and the feature maps they produce. Finally, there was looked into an other transformation of the images, namely rotation. This led to the idea that the resolution of the images has influence on classifying the rotated cats and dogs.

Contents

1 Introduction

Machine learning is becoming a more and more important part of our lives. It is already used in a lot of places around us. We can use it to recognise our voice, to make our cars drive by itself or even in diagnosing diseases. There is no doubt that machine learning will play an ever bigger role in our lives in the future.

One of the fields of machine learning that can be explored is supervised learning on artificial neural networks. In this field, a lot of labeled training examples are fed through a network and analysed. Based on the labels of the examples, the network tries to learn the classification. The extent to which it can perform the classification, really depends on the complexity and the architecture of the network. For example, special layers can be used to perform certain tasks. Therefore, in designing neural networks it is important to know which tasks are learned, and how the design of the network influences its performance. The aim of this thesis is to explore these concepts.

The structure of the report will be as follows. In chapter 2 the basics of neural networks are introduced. Also a mathematical foundation for neural networks is created. A special type of neural network, which is called the convolutional neural network (CNN), is explored in chapter 3. In chapter 4 multiple experiments will be done on a very simplistic CNN in order to investigate the limits of these networks. Finally, chapter 5 gives the conclusions and the recommendations for further research.

2

Basics of neural networks

In this chapter we will look into what neural networks are and how we can use them to classify objects. For this, we will look into the building blocks of a neural network, the neurons. We will look into how they are connected and what mathematics is involved to make the network learn.

2.1. Building blocks

The architecture of the artificial neural network (ANN) is based on the architecture of the biological brain. Just like the brain, an ANN is build out of neurons. In figure 2.1 a schematic drawing of a biological neuron and an artificial neuron can be seen. The biological neuron receives multiple electrical signals from other neurons via the dendrites. Inside the neuron these signals are processed and then a new signal is sent through the axon to other neurons. Similarly, the artificial neuron receives multiple input values from other neurons, which are processed and one output will be sent forward to other neurons.

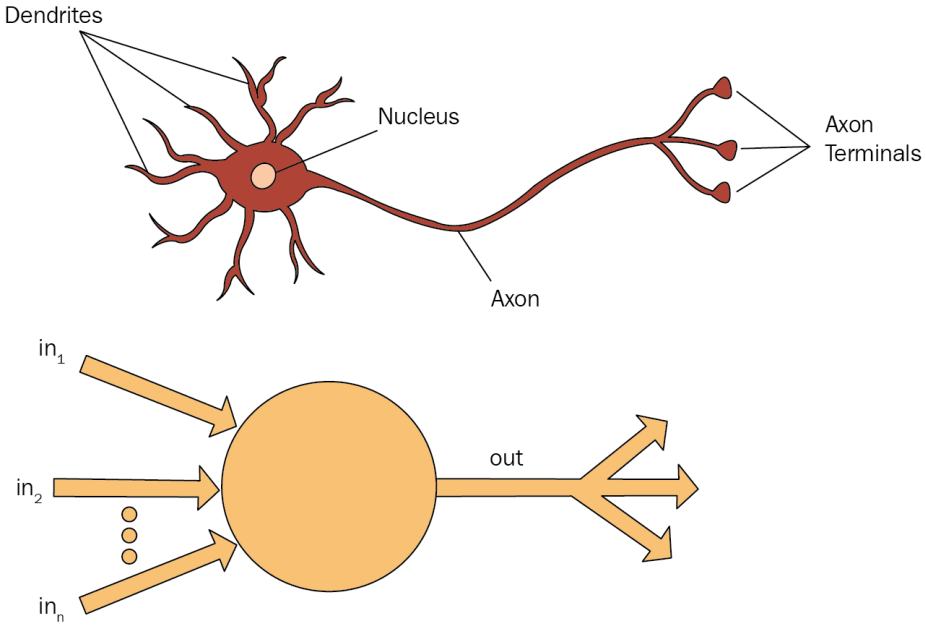

Figure 2.1: The comparison between a biological neuron (schematic) and an artificial neuron. Similar to how the biological neuron receives electrical signals via the dendrites and outputs a signal via the axon, the artificial neuron receives multiple input values and sends out one output value that can be transmitted to multiple other neurons. By courtesy of [14].

A common way to build up an ANN is by putting neurons in layers and then connecting these layers. In figure 2.2 a network can be seen that could be created. The first layer in such a network is called the

input layer. This layer contains the external data, for example the pixel values of an image. The last layer will be called the output layer. This layer produces the final result, which could be a probability that a certain object is in the image. All layers in between are called the hidden layers. This is where most of the calculations are done and where the network learns most.

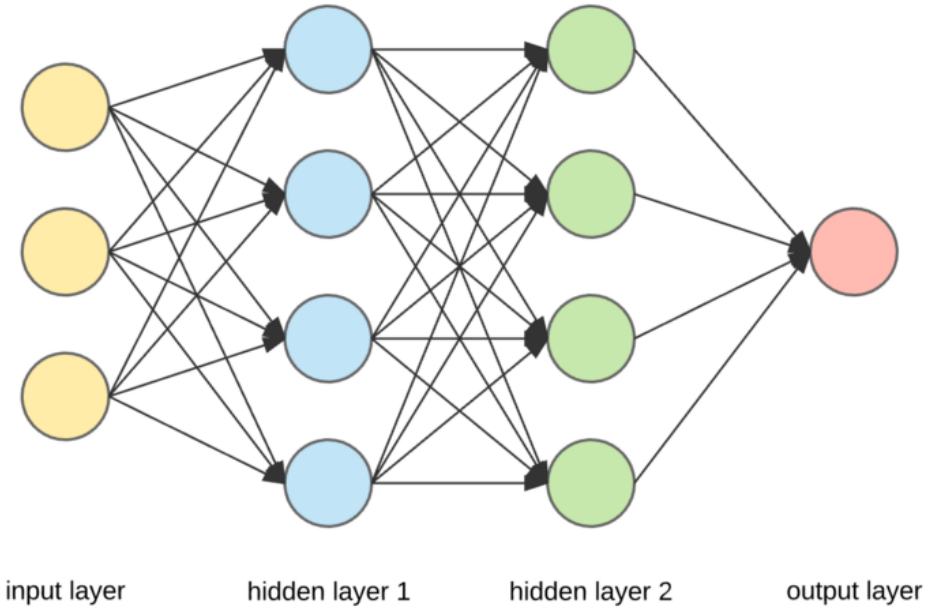

Figure 2.2: A dense ANN containing an input layer, two hidden layers and an output layer. By courtesy of [7].

2.2. The mathematical formulation of the neurons

The ANN can be described solely by mathematical calculations. To get a better understanding of how this works, we will look again at a single neuron in the network. Say that this neuron is in a certain layer, that is not the input layer, and is connected to the previous layer which has n neurons. Since it is connected to multiple neurons from this layer, it can receive several values x_j from them, in which $j \in \{1, 2, \dots, n\}$. These values x_j are processed by multiplying each value with a corresponding weight w_j. In this way the importance of each input can be altered, which is useful since some of the inputs will be more important in the classification than others. If the neuron is not connected to a certain neuron from the previous layer and therefore it does not receive that input value x_j, then $w_j = 0$. After this, all the products are summed, together with an extra value b, which is called the bias. This bias is an extra way to alter the output of the neuron. The sum is now passed into some activation function, which gives the final output of this neuron. This output will be transmitted to neurons in the next layer. Say y is taken as the output of the neuron and f as the activation function, then the process in a neuron can be described by

$$y = f\left(\sum_{j=1}^{n} w_j x_j + b\right) \tag{2.1}$$

The most simple network is a network that consists of only dense layers. In dense layers all neurons from one layer are connected to all of the neurons in the previous layer. The network that is shown in figure 2.2 consists of only dense layers. Say that there is a dense layer with m neurons. Then for each of the neurons in the layer equation 2.1 should hold. The output of each neuron in this layer y_i with $i \in \{1, 2, \dots, m\}$ can therefore be described by

$$y_i = f_i\left(\sum_{j=1}^{n} w_{i,j} x_j + b_i\right) \tag{2.2}$$

A more elegant way to write this is by using matrix notation. In this way the output of a whole layer can be described in terms of the input of this layer. Therefore, let $\mathbf{x}$ be a vector containing all of the

input values x_j and $\mathbf{f}$ be a vector of activation functions f_i. The bias values b_i are placed in a vector $\mathbf{b}$. Finally, all the weights $w_{i,j}$ are stored in a matrix W, which will look as follows.

$$W = \begin{pmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1} & w_{m,2} & \cdots & w_{m,n} \end{pmatrix} \tag{2.3}$$

The output of the layer $\mathbf{y}$ is now easily calculated by the following equation, making equation 2.2 hold for all $i \in \{1, \ldots, m\}$.

$$\mathbf{y} = \mathbf{f}(W\mathbf{x} + \mathbf{b}) \tag{2.4}$$

Most of the time, however, the activation function will be chosen the same for neurons in a layer. We then have that for all $i, k \in \{1, 2, \ldots, m\}$ that $f_i = f_k$. In this case we do not need a vector of activation functions.

The weights and biases in the whole network are often referred to as the trainable variables. Later we will see why this is the case.

2.3. Activation function

In the neural network, the purpose of the activation function for each neuron is to define if the neuron will transmit the information it receives and to what extent. In the comparison of the ANN with the biological brain, the activation function can be seen as the rate of action potential firing in the biological neuron [12]. The simplest function that could be taken is a function for which the neuron simply will fire or not. The Heaviside step function H is such a function and is defined as

$$H(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x < 0 \end{cases} \tag{2.5}$$

In this case the output of the neuron will be 0 or 1, depending on the fact if the weighted sum is above or below a certain threshold, which is 0 in this case.

However, there are a lot more activation functions that could be used. The most common used functions in neural networks are the following.

2.3.1. Sigmoid function

Instead of using the harsh Heaviside step function, it might be preferred to use a much smoother function. In this way, a weighted sum that is just below the threshold will still influence the neurons in the next layer. A function that is a lot smoother is the sigmoid function.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.6}$$

This function will have a S-shaped curve. It takes the input that could range from $-\infty$ to $+\infty$ and will output a value that is in the range between 0 and 1.

2.3.2. Rectified Linear Unit function

The Rectified Linear Unit (ReLU) function is one of the most used activation function, due to its simplicity and efficiency [8]. The function is defined to take the maximum of 0 and the input value.

$$ReLU(x) = max\{0, x\} \tag{2.7}$$

Since the function is unbounded, it is possible for the output to become very large.

2.3.3. Hyperbolic Tangent function

Just as the sigmoid function, if the output needs to be restricted to a certain range, the hyperbolic tangent function can be used.

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.8}$$

The function has a range between -1 and 1 and therefore the output can also be negative.

2.3.4. Softmax function

The softmax function is often used in the output layer of a neural network, since it will output a probability distribution on the classes that we want to classify. What is special about this activation function is that the function not only takes one weighted sum as the input, but instead takes a vector of all the sums for each of the m neurons in the output layer. In this way, the function can transform this vector such that the values will be between 0 and 1. These values can be interpreted as probabilities [19].

Say that $\mathbf{z}$ is the vector of weighted sums that are calculated from the values of the previous layer, the weights between the layers and the biases. Then the softmax function is formulated as follows

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{m} e^{z_j}} \tag{2.9}$$

The term in the denominator is used to normalize the ith weighted sum to a probability for a certain class.

2.4. Loss function

In order to know how well a neural network performs, there need to be a kind of measure that tells the difference between the output of the network and the output that is desired. For this purpose, a loss function is used.

There are multiple loss functions that can be used. Two of the most famous functions are the mean squared error loss and the cross-entropy loss.

2.4.1. Mean Squared Error Loss

The mean squared error (MSE) loss measures the difference between the output of the network and the desired output by calculated the average of the squares of the differences. Say that for a neural network that will do a classification on n objects, the desired output of a neuron in the last layer is y_i and the actual output of this neuron is $\hat{y}_i$, then the MSE loss is defined as follows

$$MSE(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{2.10}$$

In here $\mathbf{w}$ will be a vector that contains all the trainable variables in the network. For simple networks with only dense layers, these are all the weights and biases.

2.4.2. Cross-entropy Loss

Another measure that could be used is cross-entropy. This is a measure that comes from the field of information theory and defines the difference between two probability distributions p and q over the same set of events [1].

In machine learning this can be used to measure the difference between the desired and actual output of the network. Since in machine learning often classification is done on a finite number of classes, the probability distributions p and q are discrete, for which the cross entropy loss is defined by

$$H(p, q) = -\sum_{i} p_i \log(q_i) \tag{2.11}$$

In here, p_i is taken as the desired probability output at a neuron in the last layer. The actual probability for this neuron that is calculated in the softmax function will be q_i.

2.5. Optimizers

Now that it is defined how the network will process the input data and how the performance of the network is measured, still a way is needed to train the network on the input data. From the loss function it becomes clear that training of a neural network is nothing more than just minimizing the loss. Indeed, if the loss is close to 0 the network's output is really close to the desired output. In order to find a minimum for the loss, good values for the trainable variables need to be found. These variables will initially be chosen arbitrarily after which they are updated constantly in the training process.

2.5.1. Gradient Descent

One of the strategies that can be used for the training is the gradient descent method. This method makes use of the gradient of the loss function. Say that $J(\mathbf{w})$ is chosen as the loss function for the neural network, which takes as input the vector $\mathbf{w}$, that contains all the trainable variables, then the gradient of this function is defined as

$$\nabla J(\mathbf{w}) = \left(\frac{\partial J(\mathbf{w})}{\partial w_1}, \frac{\partial J(\mathbf{w})}{\partial w_2}, \ldots, \frac{\partial J(\mathbf{w})}{\partial w_n} \right) \tag{2.12}$$

In finding a minimum for the loss function, the gradient is used to make small steps in the direction that decreases the loss. Since the gradient gives the slope of the function at a point, this direction will be the opposite direction of the gradient. This can mathematically be formulated as

$$\mathbf{w} = \mathbf{w} - \eta \nabla J(\mathbf{w}) \tag{2.13}$$

In this equation, the vector $\mathbf{w}$ will be constantly updated during the process. The size of the steps η that will be taking is referred to as the learning rate of the optimizer. Choosing the learning rate to be large is useful if a quick learning of the network is preferred. However, this will potentially lead to a lower accuracy, since the network will find a minimum of the loss function less precise. It is also possible in this case that the loss will oscillate around the minimum. Taking a small learning rate, on the other hand, has the advantage that it is much more precise and therefore it can come really close to the minimum of the loss function. This could lead to a higher accuracy, but the training process will then take much longer. In figure 2.3 a visual explanation of the gradient descent method can be seen.

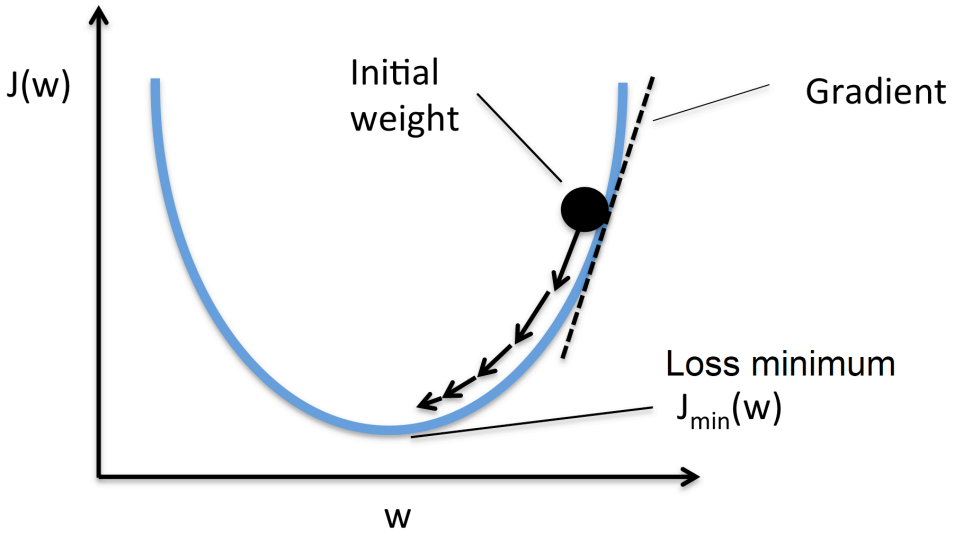

Figure 2.3: The process of gradient descent for a network in which only one trainable variable w can be altered. Each time a small step in the opposite direction of the gradient is taken. In reality often there are a lot of variables that can be changed and therefore the loss function will be much more difficult. Adapted from [15].

Since the network usually has a lot of trainable variables, the loss function will be very hard to visualize . The function in general has multiple minima, in which the optimizer could end up. Therefore, most of the time when the optimizer has found a minimum, it probably is a local minimum and not the global minimum. As a result, the key of machine learning could also be described as finding the best local minimum for the loss function.

In the normal version of gradient descent, the trainable variables are updated after the gradient is calculated from the whole training data. This has the disadvantage that for a large data set, the training process takes a lot of time. Therefore, an adapted version of gradient descent could be used. This variant is called stochastic gradient descent (SGD). It updates the trainable variables after every single input of the training data. In this way, a much quicker convergence is reached, but the path that is taken towards the minimum of the loss function will be oscillating a lot. Therefore often a combination of the normal version and SGD is chosen, which is called mini-batch gradient descent. In this case the training data is split in batches and the trainable variables are updated after every batch.

2.5.2. Momentum

The gradient descent method can be improved by taking momentum into account for the optimizer during training [5]. This is beneficial, since then the closer we go to the minimum of the loss function, the smaller the steps are that will be taken. In this way the minimum can be found much more accurate. A way how momentum can be implemented in the loss function is shown in equation 2.14.

$$\begin{aligned} M_n &= \alpha M_{n-1} + \eta \nabla J(\mathbf{w}) \\ \mathbf{w} &= \mathbf{w} - M_n \end{aligned} \tag{2.14}$$

In the equation α will be the momentum term and can be set to 0.9, for example.

2.5.3. Learning rate

Another way to improve the optimizer is by changing the learning rate η based on the variables. For the gradient descent method this rate is constant, but altering this on the trainable variables gives much more flexibility in the learning process. A lot of optimizers have been made over the last years that use adaptive learning rates. Some examples are the Adagrad, the AdaDelta and the RMSProp optimizer [2] [5]. Also one of the best optimizers there is nowadays has such learning rate. This optimizer is called the Adaptive Moment Estimation (Adam) optimizer, which can be seen as a combination between the RMSProp optimizer and the SGD optimizer with momentums of the first and second order [2] [10].

2.6. Design of neural networks

In designing a neural network, there are a lot of parameters that need to be specified. The network can be made very complex by using a lot of neurons and a lot of layers. However, making the network larger, does not necessarily mean it will perform better. One of the problems to take into account in designing the neural network is the phenomenon called overfitting. In this case the network is able to classify really well on the data it was trained on, but not on data it has never seen before. This phenomenon can best be explained by means of another situation, where this appears in.

When a set of data points is given, a polynomial function can be fitted in order to find the relationship between the points. For this, the degree of the polynomial need to be specified. When the degree equals 1, then just a linear function is fitted. In increasing the degree, the polynomial will go through the data points ever better. Ultimately, when the degree of the polynomial is high enough, it will be able to go through all of the data points. In figure 2.4 this effect can be seen. However, it can also be seen that the polynomial with high degree has become very wild and can not be used to predict any values in between the data points. Since often the data points do have some noise in it, for example due to an experimental setup, the best fit does not need to go through all of them.

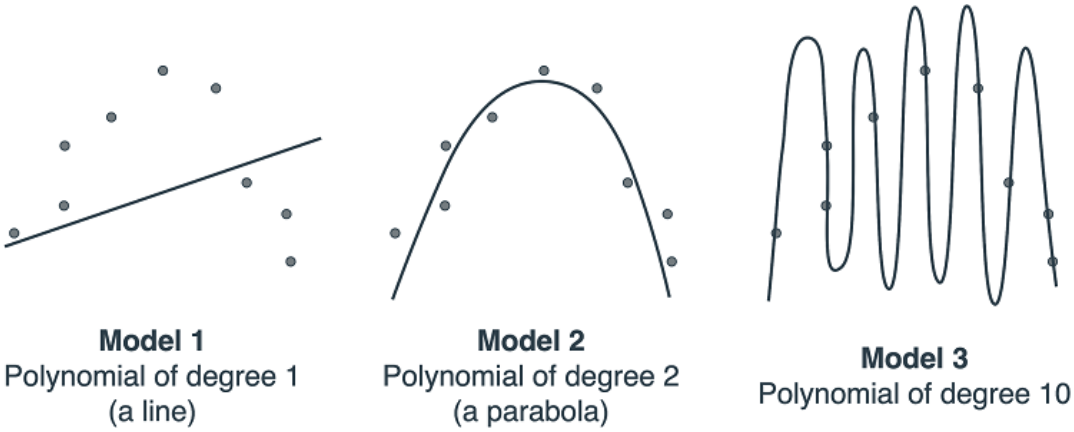

Figure 2.4: Fitting polynomials with different degrees on a set of data points. The higher the degree of the polynomial, the better it will fit the data points. However, when the degree is too high, the polynomial can not be used to predict values in between the data points. By courtesy of [17].

Similarly, for neural networks it is possible to use a lot of neurons and layers in order to get maximum flexibility. In this way, the network will become very good in classification on the training data, but has

not learned to generalize. Therefore, new data, which it has not seen, will give bad results. In this case, we say that the network has overfit the training data.

Another problem that could appear in choosing a large neural network, is that the training process will take a lot of time. This has to do with the fact that the larger the size of the network is, the more calculations there need to be done. Therefore, in designing a neural network, there is a trade off between the speed of the training process and the accuracy by which the classification is performed.

Consequently, the size of the network and the parameters that need to be chosen really depend on the task that the network is supposed to do. In order to prevent overfitting and to help the training process to go quicker, special types of layers, other than dense layers, have been designed. In chapter 3 two of these layers will be viewed.

3

Convolutional Neural Networks

A type of network that is especially useful for image recognition is the convolutional neural network (CNN). These networks are able to extract features from the data by using filters, which are usually called kernels. This is done in the convolution layers of the network. In this chapter, we will take a look into this type of network and try to understand the workings of the convolution layer. Furthermore, we will look into another layer of the network, which is called the pooling layer.

3.1. Convolution operator

The mathematical tool CNNs make use of is the convolution operator. The operator takes two functions and produces a third function, describing how one function is changed by the other. For two functions $f, g : \mathbb{R} \to \mathbb{R}$ the convolution between the functions $f * g$ is defined by

$$(f * g)(t) := \int_{-\infty}^{\infty} f(\tau) g(t - \tau)\, d\tau \tag{3.1}$$

Visually, this can be seen as sliding a flipped version of g over f. During the sliding, at every time t the area of the overlap between the two functions is calculated. The areas now in the entire time domain make the convolution of the two functions.

For functions f and g that are defined on the set of integers $\mathbb{Z}$, a discrete version of convolution is used. This is defined by

$$(f * g)(n) = \sum_{m=-\infty}^{\infty} f(m) g(n - m) \tag{3.2}$$

3.2. Convolution layer

In neural networks, the convolution is done in the convolution layer. The input of this layer often is a multidimensional array of data, which will be adapted during the convolution with a kernel. This kernel is an array of the same dimension and contains the trainable variables of the layer. In order to understand how the convolution layer works between the input and the kernel, it is best to look at the convolution layer in an image recognition network.

For image recognition the input of the layer will be an image that can be seen as a two-dimensional array in which every value corresponds to a pixel of the image. The idea for the convolution is that the kernel will alter the image in a way that some parts become highlighted. If a good kernel is used, then these parts show useful patterns in the image such as lines and curves that can be recognized as parts in the objects. These patterns are often called features. Therefore, the output of the convolution layer will be called the feature map.

Since images are two-dimensional and pixels in a grid are used to represent an image, a two-dimensional discrete convolution is needed. Taking an image I and a kernel K we can define the convolution between these to as

$$(I * K)(x, y) = \sum_{m} \sum_{n} I(m, n) K(x - m, y - n) \tag{3.3}$$

Visually, this means that the kernel will be flipped in both the horizontal and vertical direction and then slided over the image. At each overlap now the dot product between the image pixels and the kernel will be taken. Say that, for example, there is an input image and a kernel as in figure 3.1. Then the kernel will be flipped in both directions, which gives the array $\left[\begin{smallmatrix} 0 & 1 \\ 2 & 3 \end{smallmatrix}\right]$. Now the output is obtained by taking the sum over the products of the corresponding entries from the input and the flipped kernel. The following calculations are done to give the output in figure 3.1.

$$
\begin{aligned}
0 \cdot 0 + 1 \cdot 1 + 3 \cdot 2 + 4 \cdot 3 &= 19 \\
1 \cdot 0 + 2 \cdot 1 + 4 \cdot 2 + 5 \cdot 3 &= 25 \\
3 \cdot 0 + 4 \cdot 1 + 6 \cdot 2 + 7 \cdot 3 &= 37 \\
4 \cdot 0 + 5 \cdot 1 + 7 \cdot 2 + 8 \cdot 3 &= 43
\end{aligned}
$$

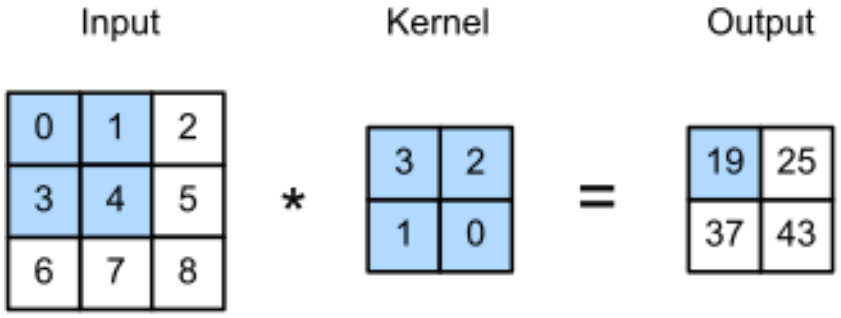

Figure 3.1: The convolution between an input image and a kernel. The convolution has valid padding and strides set to 1 for both the height and the width. Adapted from [20].

To extract useful features from the images it is important to choose the size of the kernel carefully. The larger the kernels are chosen, the larger the features potentially are that are highlighted. However, in choosing a larger kernel, also more variables need to be learned by the network. This could lead to a much longer training time and make the training a lot more difficult. Therefore, often smaller kernels are used. Also odd sized kernels are preferred above even sized kernels. This has to do with the fact that for odd sized kernels the output of the dot product can be placed in the middle of the window. For even sized kernel it is not clear where the output needs to be placed, since there is no middle element. This will make the implementation harder. A size that is, therefore, often used for the kernel is 3×3.

During the training of the neural network, multiple images will go through the convolution layer. Since the images usually are in colour, multiple channels are needed to store the colour of every pixel. Most of the time, three channels are used that correspond to red, green and blue. Also, in general the convolution layer will perform a convolution between the image and multiple kernels. In this way the layer is able to extract more features from the image. By all this, it is important to keep track of the shape of the input and the shape of the kernels. The shape of the input can be defined as $N_I \times H_I \times W_I \times C$. In here, N_I denotes the number of images that are in the input data, H_I the height of the images and W_I the width of the images. The number of input channels will be denoted by C. The shape of the kernels can be defined as $H_K \times W_K \times C \times N_K$. In here, H_K and W_K will be the height and the width of the kernel respectively. In order to deal with the multiple channels in the input images, the kernels need to be defined on the same number of channels C. For each image, the number of kernels N_K that will be used will lead to multiple outputs. These outputs are taken as the different channels for the feature map of the image. Therefore, the feature map will have a shape of $N_I \times H_F \times W_F \times N_K$, where H_F and W_F are the height and the width of the feature map. From section 3.4 it will become clear how these follow from the height and width of the input image and the kernel. Note that the shape of the feature map is similar to the shape of the input image. Therefore the feature map can be seen as a new image. Also, because of this, multiple convolution layer can be stacked in the network that will take the feature map from the previous layer as the input.

After the convolution in the convolution layer also for each kernel a bias value could be added to the output. Like the values in the kernels, the bias values can also be trained. Since in the convolution the same kernel is used at every position in the image, this means that in a convolution layer there are

$N_K \cdot (H_K \cdot W_K \cdot C + 1)$ trainable variables. Here the $+1$ comes from the bias value that is added per kernel. This is far less than we would have in a dense layer, which makes the convolution layer very useful. After the bias is added, the output will be put through the activation function and the resulting feature map will be transmitted to the next layer.

3.3. Convolution vs Cross-correlation

In a lot of machine learning libraries cross-correlation is used, when convolution is mentioned [6]. Cross-correlation is an operator that is quite similar to convolution and can be used in a lot of the same situations. It is defined by

$$(f \star g)(t) := \int_{-\infty}^{\infty} f(\tau) g(t + \tau)\, d\tau \tag{3.4}$$

The difference with convolution visually is that the function g is not flipped before sliding it over the function f. From the definition it becomes clear that convolution of two functions f and g is equal to the cross-correlation of f and $-g$. Since convolution is commutative, meaning that $f * g = g * f$, similarly the convolution between f and g is equal to the cross-correlation of $-f$ and g.

In image recognition, a two dimensional discrete cross-correlation can be used, which is defined by

$$(I \star K)(x, y) = \sum_m \sum_n I(m, n) K(x + m, y + n) \tag{3.5}$$

When cross-correlation is used instead of convolution, the kernel is not flipped horizontally and vertically before the operator is applied. Therefore, the convolution between the image and the kernel is equal to the cross-correlation between the image and the flipped kernel. For example, taking the cross-correlation with the kernel from figure 3.2, which is the flipped version of the kernel in figure 3.1, will give the same output as in figure 3.1.

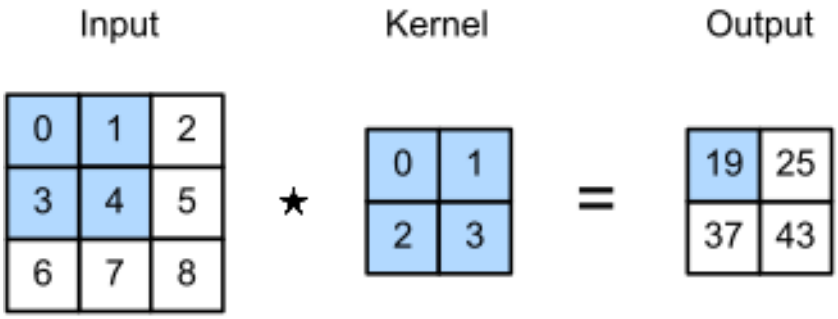

Figure 3.2: Two-dimensional cross-correlation with valid padding and strides set to 1 for both the height and the width. Adapted from [20].

3.4. Padding and stride

Besides the number of kernels that are used and the size of these kernels, there are more parameters that can be changed in the convolution layer. Two of these parameters are the padding and the stride.

3.4.1. Padding

The padding of the convolution layer specifies what happens at the boundaries of the input image. In case we allow the kernel to only lie entirely in the image, as in figure 3.1, the convolution layer is said to have valid padding. The height H_F and the width W_F of the output of the convolution will then be $H_I - H_K + 1$ and $W_I - W_K + 1$, respectively. This means that the feature map will be smaller than the input image, if a kernel is chosen larger than 1×1. This has the consequence that pixels close to the boundary of the image become less important, certainly when multiple convolution layers are used [9].

Another type of padding that can be chosen, is called same padding. In this case, the height and width will remain the same during the convolution. This means that $H_F = H_I$ and $W_F = W_I$. This is achieved by adding a number of rows and columns of zeros to the input image. As said before, often

even sized kernels are not used. In order to have a convolution with same padding between the input image and the kernel from the figures, the output of the convolution need to have a shape of 3×3. To obtain this, one extra row and column with zeros need to be added to the input image. However, there are multiple ways that this can be done, which yield different outputs. To avoid this odd sized kernels are preferred. Same padding is especially useful if we want to use multiple convolutional layers without having to worry about the output getting smaller and smaller.

As a third, we could take full padding. In this case we will slide the kernel over the image, such that every value of the kernel will overlap once with every value in the image. With this padding no information will be lost from the boundaries. However, as can be seen in figure 3.3, the output will be larger than the input. This padding is therefore more computationally expensive, certainly when multiple of these convolution layers are used.

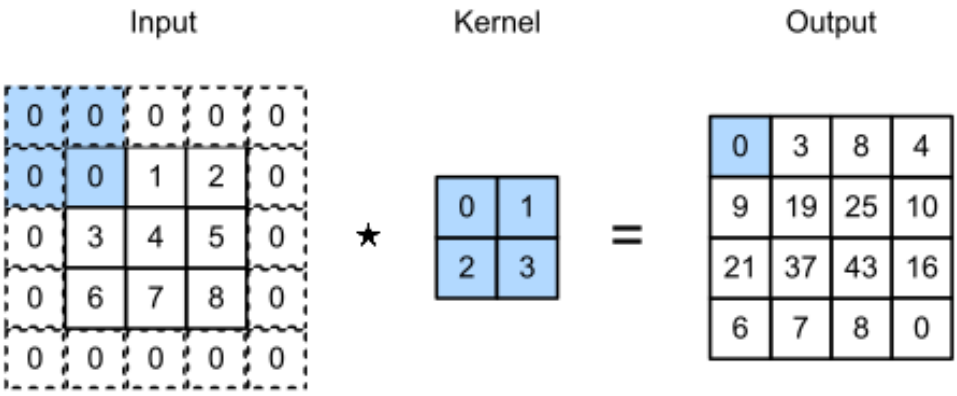

Figure 3.3: Two-dimensional cross-correlation with full padding. Adapted from [20].

3.4.2. Stride

Like padding, the stride also describes how the kernel slides over the image during the convolution or cross-correlation. The stride is defined for both the height and the width as an integer, which tells how large the steps are that the kernel takes in each direction. In normal convolution both strides are set by default to 1. In figure 3.4 an example of cross-correlation can be seen with stride for the height equal to 3 and stride for the width equal to 2.

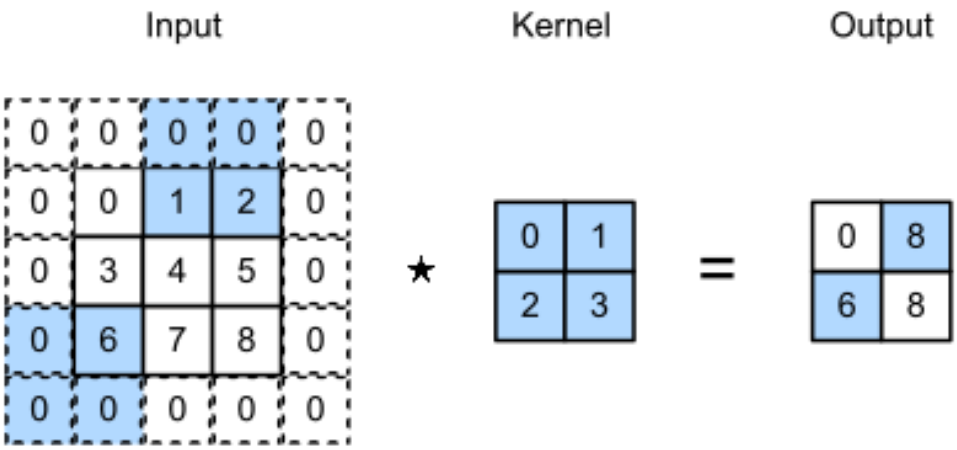

Figure 3.4: Cross-correlation with strides of 3 and 2 for the height and width, respectively. Adapted from [20].

If the input image has a high resolution, then strides can be used to downsample the input. In this way the speed of the learning process can be increased. However, it is possible that small features in the image will be lost.

3.5. Pooling layer

After a convolution layer often a pooling layer is chosen. Similar to the strides, the pooling layer can be used to lower the resolution of the input. This is done by placing a pooling window with fixed size on the input in different locations. For each location the values that appear in the window are replaced with one value by taking a summary over the values. Contrary to the convolution layer, the pooling layer does not contain trainable variables.

There are different kind of pooling layers. Two of the most common kinds of pooling are maximum pooling (MP) and average pooling (AP). In a maximum pooling layer, we will take the maximum value that appears in the pooling window. In average pooling the average over the values in the pooling window will be taken. In figure 3.5 both pooling methods can be seen on a 4×4 input. Since the pool size is 2×2, the size of the output of the pooling layer will be 2×2.

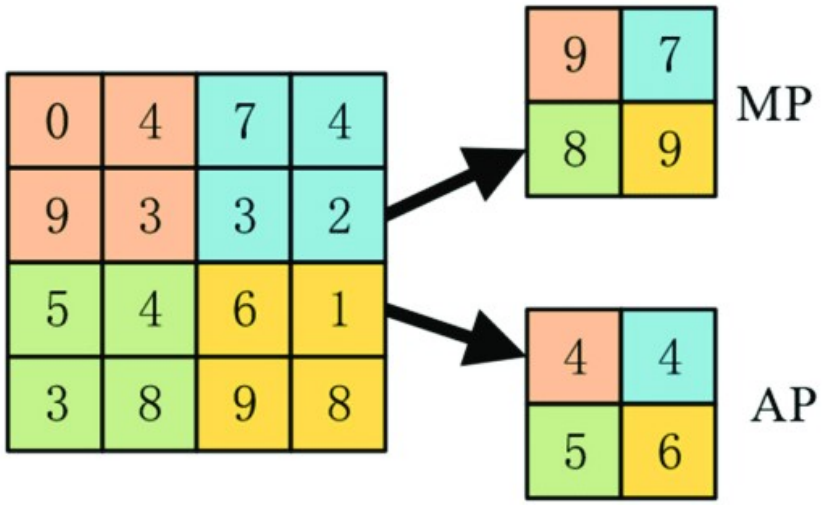

Figure 3.5: Maximum pooling (MP) and average pooling (AP) on a 4×4 input image with a 2×2 pooling window. The pooling has valid padding and strides equal to 2. By courtesy of [18].

Just like in the convolution layer, padding and strides can be specified for the pooling to tell what happens at the edges and how large the steps are that are taken. Usually for pooling the padding is valid and the strides are chosen similar to the size of the pooling window.

Apart from downsampling, the pooling layer can also be used to make the input data invariant to small translations [6]. This means that if the input is shifted with a small amount, the output of the network does not change. For example, since in maximum pooling the maximum value over a certain area is taken, the information about where that value was in the area is lost.

4

Experimenting on a simple CNN

Convolutional neural networks have become one of the most important type of neural network for image recognition. Since for every location in the image in a CNN the same kernels are applied, it is intuitive to believe that this will make the CNN invariant to translations. This means that after shifting the object to be recognized in the image, it will still be classified as the object. In other words, the classification of the object is not depending on the location of the object. In this chapter we will explore if this intuition is right by doing an experiment on images in which a cat appears in different locations. Furthermore, the network will be learned to recognise the difference between a cat and a dog. There will be looked into the different layers of the network and where this ability comes from. After this, also there will be looked into rotating the cats and dogs and how well a simple CNN performs on these, depending on the number of kernels that will be used in the convolution layer. All of the experiments will be done in Google's machine learning library Tensorflow [11].

4.1. Testing translation invariance

In the first experiment we would like to investigate if CNNs are indeed translation invariant, which is commonly believed. Therefore, we will train a simple CNN on two images of a cat appearing in different locations. It is expected that the location of the cat will be neglected and that the network sees both training images as the same, if the CNN is indeed translation invariant. This means that the network will not prefer one of the two training images above the other, when a cat appearing in a third position is put through the network.

4.1.1. Creating a data set

To set up the experiment, a data set is needed. For this an image of a cat is chosen, which can be seen in figure 4.1a. However this image has a high resolution, which is not necessary for our purpose and will only slow down the training process. Therefore, this image is rescaled to a 65×65 pixels image, which can be seen in figure 4.1b.

Another way to improve the training speed is by making the image a grayscale image. For this experiment, this will be done, since the colour of the cat is not relevant in showing the translation invariant property. By doing this, only one channel has to be used to store pixel values instead of three channels for each red, green and blue. This will reduce the time needed for the training.

After this, the cat will be placed on a white background of 200×200 pixels in various locations. It is placed in 10 different locations both horizontally and vertically, making a total of 100 images. This means that the distance between the locations of two adjacent cats is 15 pixels. Since the cat is 65×65 pixels, this also means that cats which appear close to each other, partially overlap. The images are named from 0 to 99 from left to right and top to bottom. This has the advantage that it is easy to imagine from the number of the image where the cat is located inside the image. The first number shows the row and the second number shows the column that the cat appears in, where images 0 to 9 can be seen as images 00 to 09. As an example in figure 4.2 on the left, image 22 can be seen where the cat appears in the second row and the second column, when the counting of the rows and columns starts at 0.

(a) The original image of 2079×2571 pixels.

(b) The rescaled 65×65 pixels version of figure 4.1a.

Figure 4.1: The image of a cat that we will use during the experiment. Since the resolution is very high and not needed for the experiment, the image is rescaled. By courtesy of [3].

4.1.2. Training data and test data

To train the network, for this experiment two images are chosen from the data set. These two images are called the training data. As mentioned in chapter 3.4, boundary conditions can have influence on the classification of objects. Therefore, the two images are chosen a bit away from the boundary. Also, in order to avoid overlap of the cats it is preferred that the cats are far away from each other in the images. In figure 4.2 images 22 and 77 can be seen that are chosen to be the training data.

In order to see if the network is indeed translation invariant, the testing will be done on all of the 100 images from the data set. This is called the test data. It is expected that if the network is indeed translation invariant, that none of the images will give a clear preference of being image 22 or image 77.

It is not very common to include the training data in the test data, since the network has already seen this data. Besides, since neural network are prone to overfitting, testing on the training data will most of the time not give useful information. However, in this case we will be able to see a nice result by doing this.

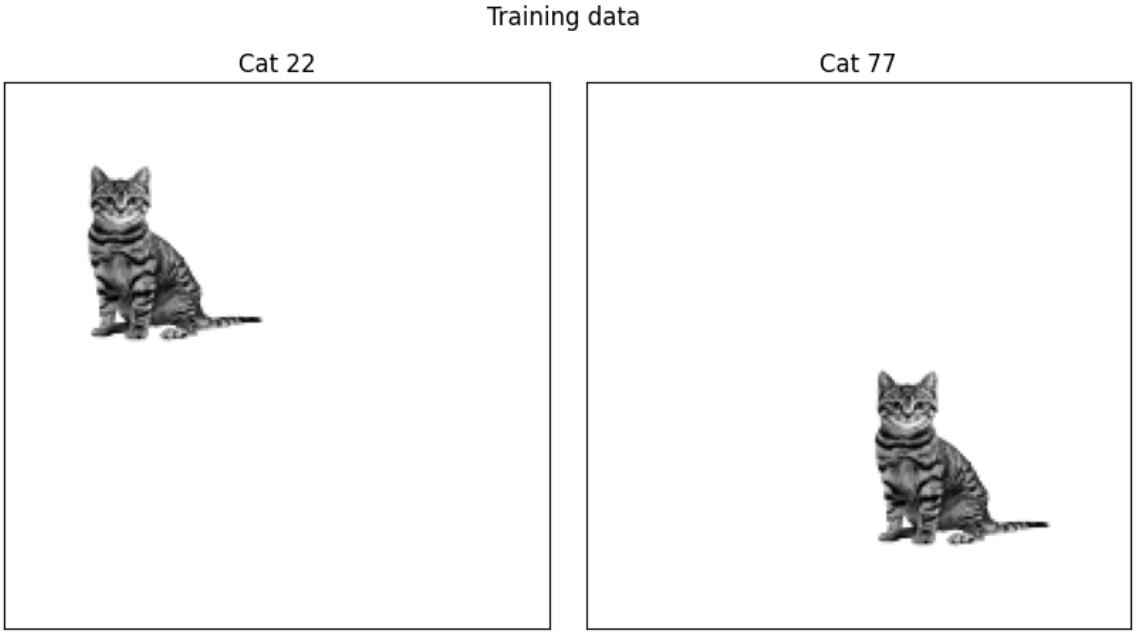

Figure 4.2: Images 22 and 77 that will be used as the training data. The images are 200×200 pixels and are grayscale to reduce the training time.

4.1.3. Constructing the network

For the experiment also a specific network should be chosen. Since we want to investigate the limits of CNNs and we want to come to clear conclusions, we will make use of a CNN that is very simple. The network will consist only of a two-dimensional convolution layer, a two-dimensional pooling layer, a flatten layer and a final dense layer.

An image that will be put through the network will first come across the convolution layer. In this layer there are 8 kernels with which convolution is done. The kernels have a size of 3×3, the strides are set to be 1 for both the width and the height and the padding is chosen to be valid. These settings for strides and padding are most common. Because of these parameter settings in the convolution layer, the output of this layer will be 8 feature maps with a width and height of 198. The activation function used will be the ReLU activation function.

After this, each of the feature maps is passed through the maximum pooling layer. In this layer the pooling window has a size of 2×2 and has equal strides. Therefore, the output of this layer will be 8 two-dimensional maps of half the size both horizontally and vertically, meaning each item in the data now has a shape of $99 \times 99 \times 8$.

The flatten layer is now used to transform the three-dimensional data in to a one dimensional array in order to pass the information to the dense layer.

The dense layer will be the output layer and consists of 2 neurons. In the dense layer the softmax function is used as the activation function. In this way the two neurons will give a probability for the input image of being image 22 or image 77.

In figure 4.3 an overview of the described network can be seen. On the left an image will be put into the network and on the right the probabilities on image 22 and image 77 will be given.

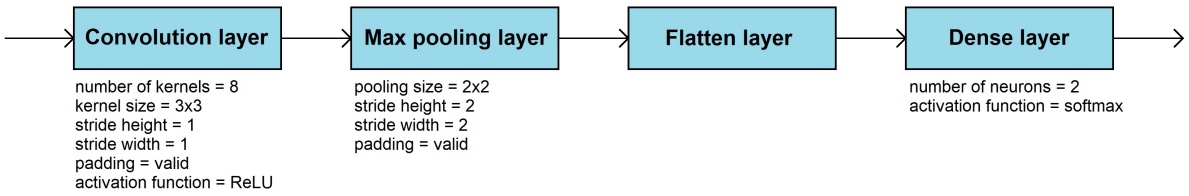

Figure 4.3: An overview of the network that is used in this experiment. Also the parameters in each layer are specified. On the left an image of 200×200 pixels with the cat will pass through the network and on the right the probabilities of being image 22 and image 77 will be given.

For training of the network, we will make use of the cross-entropy loss function and the Adam optimizer. Since the training data consists of only two images and the optimizer only takes small steps in the right direction, we have to go over the training data a lot of times in order to find good values for the trainable variables. The process in which all of the training data is passed through the network once is called an epoch. The network will be trained on 100 epochs, meaning that the the two images will go 100 times through the model each. In order to keep the training balanced and avoid walking back and forward between the two training images, the batch size is set to 2. Since the batch size now is the same as the number of images in the training data, the loss will be calculated and the trainable variables will be updated after each epoch.

4.1.4. Results

Taking this setting for the experiment and the common believe that CNN are translation invariant, we expect that the network will not be able to learn the difference between the two training images. Furthermore, we expect that for each image of the test data the network can not decide if it should be classified as image 22 or image 77.

However, after putting all of the test data through the network, the location of the cat in the image does seem to be important for classifying the image as image 22 or image 77. In figure 4.4 the probabilities of being image 22 for all of the test data can be seen. The probabilities are plotted in a 10 by 10 grid from left to right and top to bottom. In this way, the row and column the probability appears in correspond to the image where the cat appears in that location. For example, the probability that is given in the second row and second column corresponds to image 22 that was in the training data. It can be seen that the closer the cat appears in the image to where the cat appears in image 22, the

more likely it is that the network will classify the image as image 22. Also, if it becomes closer to where the cat appears in image 77 the probability of being classified as image 22 will become smaller. Since there are only two cases to classify on and probabilities should add up to 1, this is the same as saying that the probability of being classified as image 77 will become higher. This result makes us question the believe that CNNs are translation invariant, since from this experiment it seems that the network can learn location in an image.

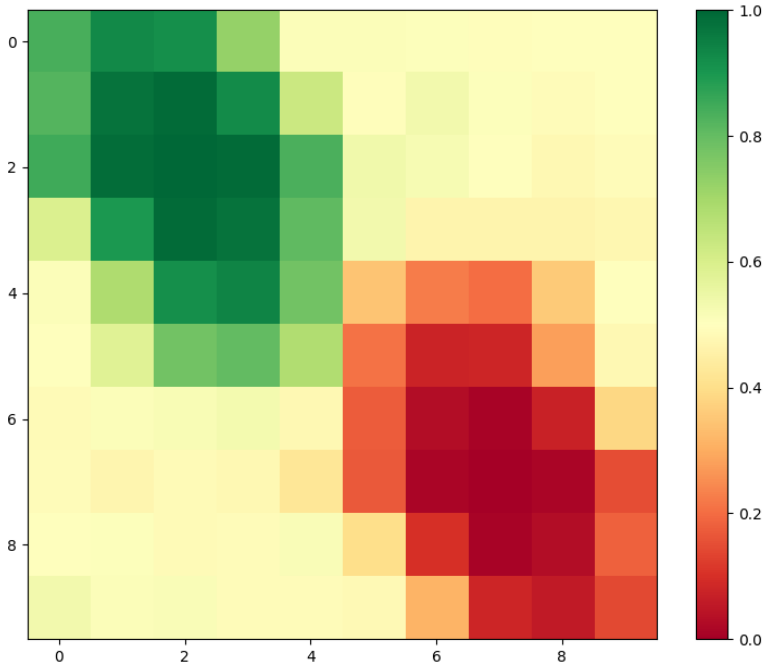

Figure 4.4: The probabilities of being image 22 for the test data plotted in a 10 by 10 grid. The location in the grid corresponds with the location where the cat appears in the image. The greener a pixel is, the higher the probability of being image 22. Since the network only classifies on 2 images, automatically more red means a higher probability of being image 77. It is clear that the closer towards positions 22 and 77 the cat appears, the more likely it is that the network will classify an image from the test data as one of these two images. Images that are too far away from both positions do not have a clear classification.

4.2. Translation in/equivariance

From section 4.1 it becomes clear that in general CNNs are not translation invariant. However, CNNs do have a property that translation invariance often is confused with, which is called translation equivariance. In understanding CNNs, it is important that the distinction between these two is known.

4.2.1. Definitions

Translation invariance and translation equivariance can be described by the following two definitions (adopted from [13]).

Definition 4.2.1 (Translation invariance) *A function f is said to be invariant to a group of translations G if for any g element of G:*

$$f(g(I)) = f(I) \tag{4.1}$$

Definition 4.2.2 (Translation equivariance) *A function f is said to be equivariant to the translations of group G if for any g element of G:*

$$f(g(I)) = g(f(I)) \tag{4.2}$$

For the convolution in CNNs, this means that it is said to be translation invariant if the output is not changed, whether or not the object in the input is translated or not. From the previous section we have seen that this is not the case. On the other hand, convolution is said to be translation equivariant if the order in which the convolution and the translation is applied, does not matter. That is, convolution commutes with translation. This means that applying first a convolution with a kernel to the input data and then a translation is equal to applying first the translation and then the convolution.

4.2.2. Showing translation equivariance of convolution

The translation equivariance of convolution can easily be shown by performing a convolution on image 22 and image 77 from section 4.1. Since image 22 actually is a translation of image 77 and vice versa, convolution on both images should give a similar output that only differ in translation.

As an example, the following simple edge detection kernel K can be used.

$$K = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \tag{4.3}$$

In figure 4.5 the convolution of images 22 and 77 with the kernel K are given. It is clear from the feature maps that after the convolution the cats look the same. Translating one of the two cats to the position of the other, will therefore give the same image. This shows the equivariant property of the convolution layer.

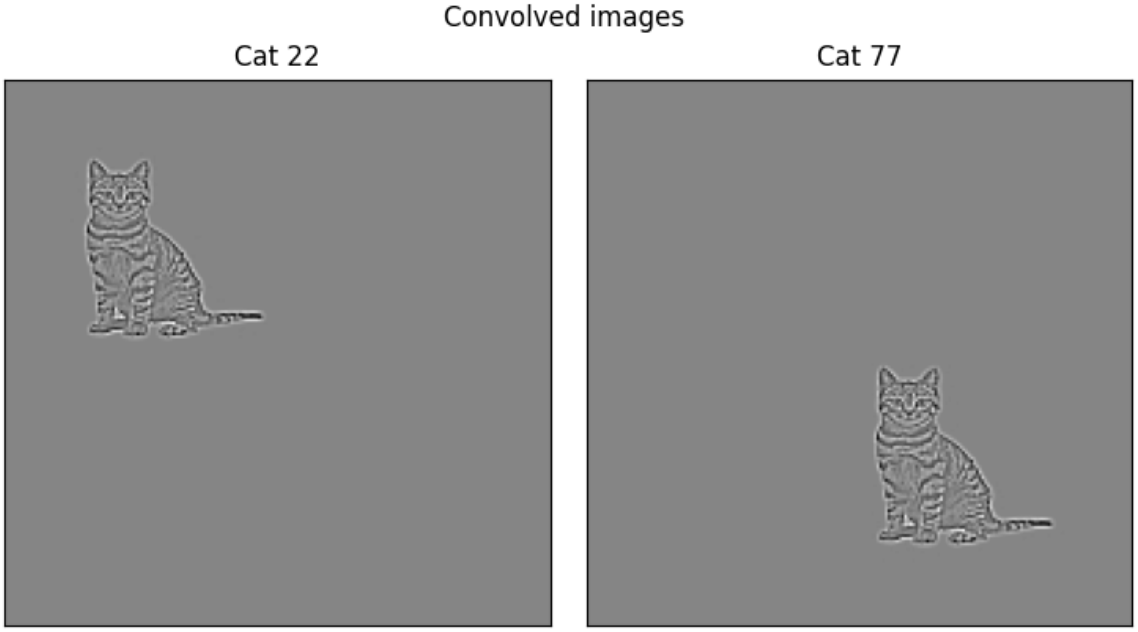

Figure 4.5: Images 22 and 77 after convolution with the edge detection kernel from 4.3. The cats are the same, except from their location in the image, showing the translation equivariant property.

4.2.3. Passing on location

The consequence of the convolution being translation equivariant instead of invariant, now explains the result that was found in section 4.1. An image that is passed through the network first comes across the convolution layer. This layer will perform a convolution, but will still pass on the location of the cat in the image. After this, the data reaches the pooling layer. Despite that this layer is by construction invariant to very small translations, still for larger translations again the location of the cat is passed on. The flatten layer now makes an one-dimensional array of the input, but in this array still the location of the cat is preserved. The trainable variables in the final layer, the dense layer, which is used to classify the image as image 22 or 77, now are trained using the location of the cat. In testing with all the 100 images these learned weights and biases now give the effect we see in the experiment.

4.3. Learning the difference between a cat and a dog

In this experiment we would like to see if there is a way in which the CNN from section 4.1 can be trained to be translation invariant. For this, together with the data set of cats, a data set with dogs is used. The network will now be trained on parts of both data sets in order to recognize the difference between the cat and the dog.

4.3.1. Data set of a dog

Similar to the data set of the cat, for this experiment also a data set of 100 images of a dog is created. Since we do not want the network to learn the difference between the cat and the dog based on their posture, the dog is chosen to sit in the same way as the cat. Figure 4.6a shows the image of the dog that will be used. This image will be rescaled to a 65×65 pixels image and placed on a white background of 200×200 pixels. The 100 images are again numbered from the top left to the bottom right, making the location of the dog correspond to the number of the image.

Since cats and dogs exist in several colours, it is not preferred that the network learns the difference between a cat and a dog based on the colour of the animal. Therefore, again instead of using all 3 channels for red, green and blue, only 1 channel is used to store grayscale values of the pixels. This also helps to speed up the training process and will keep the experiment simple.

(a) The original image of 312×312 pixels.

(b) The rescaled 65×65 pixels version of figure 4.6a.

Figure 4.6: The image of a dog that will be used during the experiment. Since the resolution is very high and not needed for the experiment, the image is rescaled to the same size as the cat. By courtesy of [4].

4.3.2. Training on cats and dogs in the top half

In the first half of this experiment, the network from section 4.1 will be trained on the data where the cats and dogs appear in the top half of the image, i.e. the images where the cats and dogs appear in the top 5 rows. These are 50 images for both the cat and the dog, which are numbered from 0 to 49.

In order to see the performance of the network during training, it is useful to have a validation set. This set exists of images that are similar to the images in the training data, but that the network has never seen before. After each epoch also the loss and accuracy of the validation set is calculated. Furthermore, this validation set can be used to see if the network is overfitting the training data. If the loss and accuracy of the validation set differ a lot from the loss and accuracy of the training data, the network is not learning the important general features in the images. Instead, the network tends to just memorize the training data and therefore will perform really badly on images from the validation set, which it has never seen before.

In recognising the difference between the cat and the dog, it is preferred to make use of a validation set. In this way, it can be seen if the simple CNN is able to learn the distinction between the animals. The validation set is made by choosing arbitrarily 20% of the training data. These images will now be removed from the training data and therefore only 80 images remain for training and 20 images for validation.

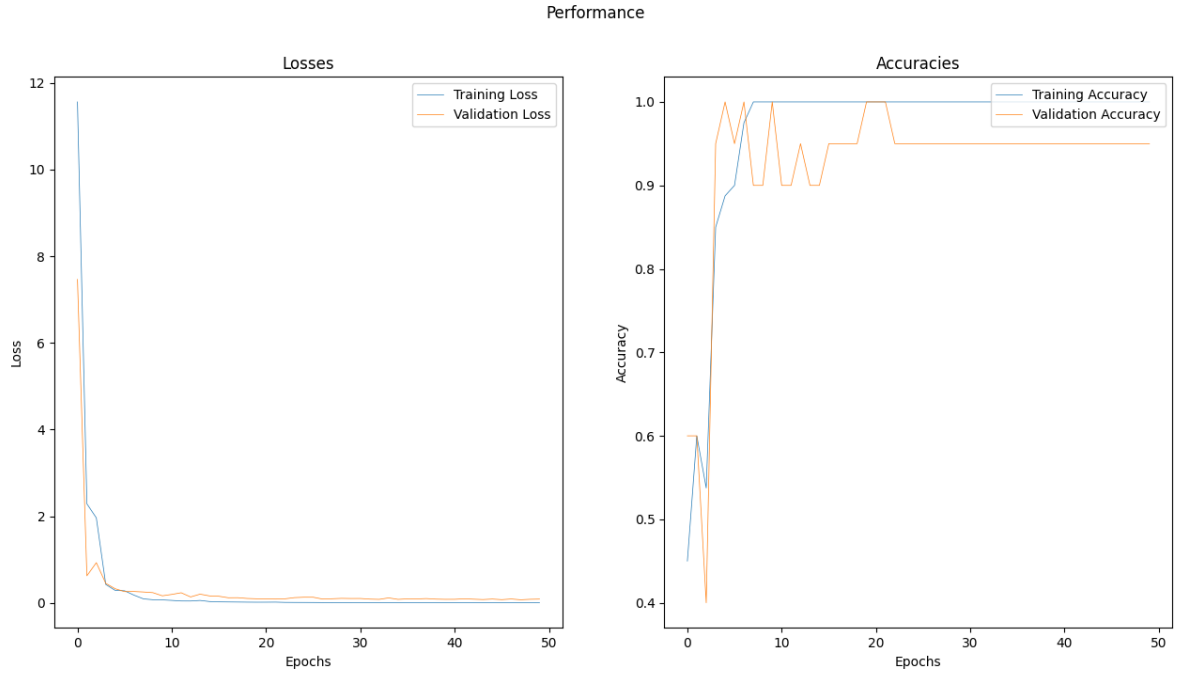

Figure 4.7: The loss and accuracy for both the training data (in blue) and the validation data (in orange) after training the network from section 4.1 on the images where the cat and the dog appear in the top half of the image plotted against the number of epochs.

The network will be trained for 50 epochs and with a batch size of 4. This results in updating the weights an biases 20 times per epoch. In figure 4.7 the loss and accuracy for both the training data and the validation data can be seen after each epoch. It is seen that the loss of the training data quickly drops to almost 0 and also its accuracy reaches 100% after 7 epochs. The loss of the validation data is a bit higher but also close to 0. Its accuracy reaches 95% after 50 epochs. This indicates that the network is indeed able to recognise the difference between a cat and a dog, since not only the training data but also the validation data gives good results.

Since the training is only done on the cats and dogs in the top half, it is interesting to see what happens when the network with the learned variables is tested on the images where the cat and dog appear in the bottom rows. These images will be the test data for this experiment. In figure 4.8 the probabilities for these images can be seen in which red means that there is a probability of 0 for the right animal and green means that the probability for the right animal is 1. From the axes of the plot it can be determined the probability of what image the pixel shows. For example, the pixels on the top left corresponds to image 50 for the cat and the dog.

The network seems to be very good in classifying the images in which a dog appears. However, for the images in which the cat appears it still seems that for most cases the network is certain of seeing the dog. Only for a few cats that appear close to the top half the network tends more to the right probability. This effect is remarkable. However, it comes as no surprise that the network is not able to classify between cats and dogs in the lower half, given that the CNN is only translation equivariant and not invariant. The effect could be explained by the way the network has learned the difference between the cat and the dog in the top half. It is plausible that the network has learned to classify every image as a dog, except for images where clearly the cat appears in the top half. This way the network will always have an accuracy of more than 50% for the training data. Since for the test images the cat is in the lower half, the network will simply classify them as the dog.

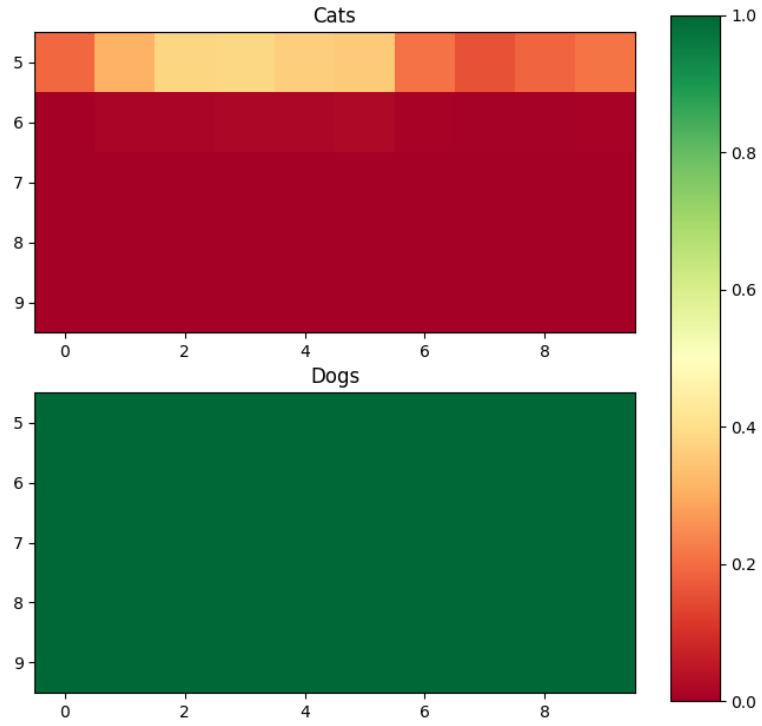

Figure 4.8: The probabilities of being a cat or a dog for the images of the test data in which the cat and dog appear in the bottom half of the image. It can be seen that the network performs really well on the images of the dog. However, also most of the images of the cat are classified as the dog.

4.3.3. Training on even columns

In the second half of this experiment, the training data and validation data will be chosen differently. Instead of using the images with the cat and the dog in the top half, the images where the cat and the dog appear in an even column are used. Again 20% of this data will be used as the validation set and 80% as the training data. In figure 4.9 it can be seen by the losses and accuracies that again the network is able to learn the difference between the cat and the dog.

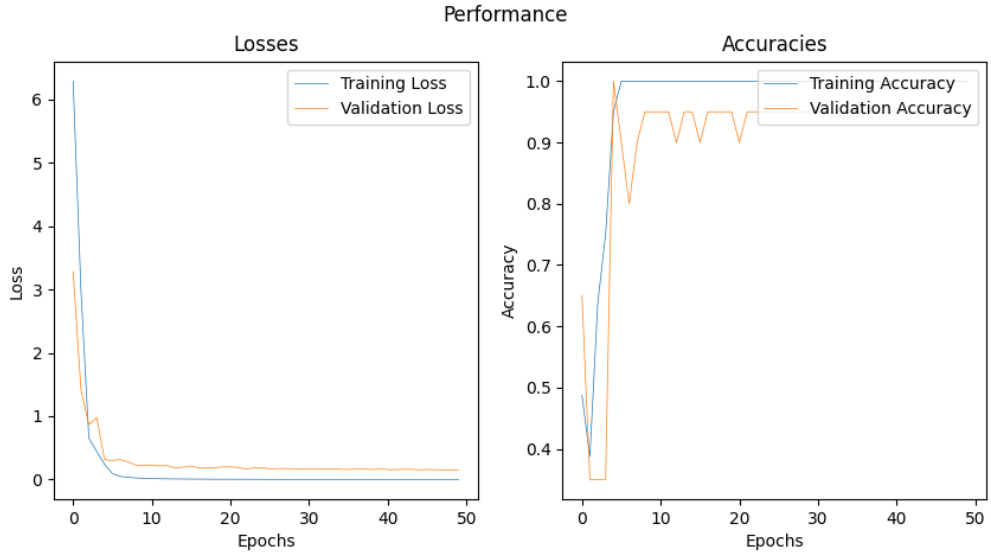

Figure 4.9: The loss and accuracy for both the training data (in blue) and the validation data (in orange) after training the network from section 4.1 on the images where the cat and the dog appear in an even column plotted against the number of epochs

The testing can now be done on the images where the cat and dog appear in the odd columns. In figure 4.10 the probabilities for these images can be seen. Just as in the split between top and bottom, the network is able to recognise all the images with a dog in it. However, in this case also for the images with a cat in an odd column it performs really well. Only column 9, in which the cat appears close to the right boundary, shows some difficulty.

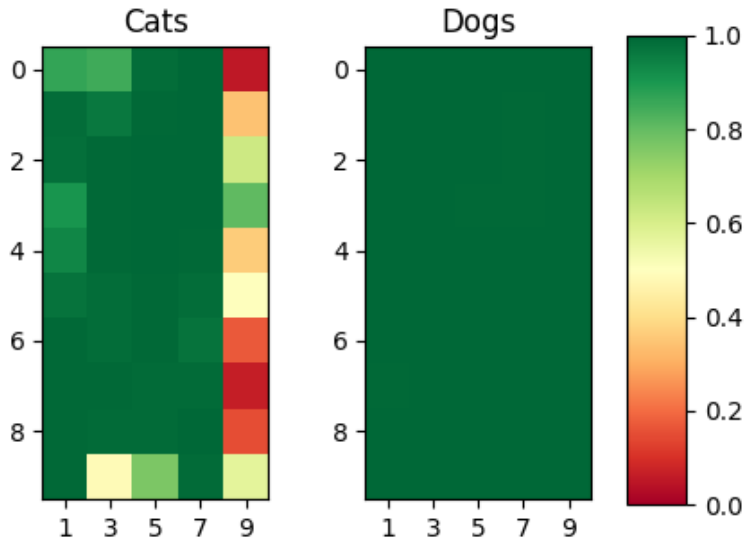

Figure 4.10: The probabilities of being a cat or a dog for the images of the test data, in which the cat and dog appear in odd columns. It can be seen that the network performs really well on the images of the dog. Also, most of the images of the cat give good result, except from the images in the column 9.

The reason why the network has become almost translation invariant could be the fact that cats from the test data are in this set up of the experiment closer to the cats from the training data. Indeed, for cats that appears in an odd column, 80% of the adjacent cats in the even columns are in the training data. Furthermore, since the cats only differ 15 pixels horizontally and the fact that the width of the cats is 65 pixels, the overlap between the two cats is 50 pixels. In this way, all of the locations where the cat appears in the test data are already used in the training. This is contrary to when the training was done on the cat and the dog in the top half of the image. The weights that corresponded to locations in the lower part of the image could there just be arbitrarily chosen.

The smaller distance between the cats could also be an explanation for the fact that column 9 shows bad results. This column is the upper right column of the image and therefore it only has a column of images on the left which could be in the training data. All of the other columns on which testing is done, have a column left and right from it whose images are in the training data.

Apart from column 9, in figure 4.10 it can also be seen that some of the probabilities for the cats that are not close to the boundaries are slightly lower than the rest. For example image 31 of the cat does have even columns 0 and 2 adjacent to train on, but appears to be predicted slightly more uncertain. This could be due to how the validation set is chosen. Since the validation set is chosen arbitrarily from the even columns, it could be that in some of the images the cats appear close to each other. In this area the network now has learned less how the cat looks like. If an image from the test data with a cat in this area is put trough the network, it will have more difficulty to classify this image. This could be the case for image 31 of the cat.

If the validation data is not used and therefore training is done on all of the even columns, it can be seen in figure 4.11 that these small variations in probabilities disappear. Only the images in column 9 still seem to be hard to classify. However, this could be expected, since they still only have the images in column 8 close to learn the classification.

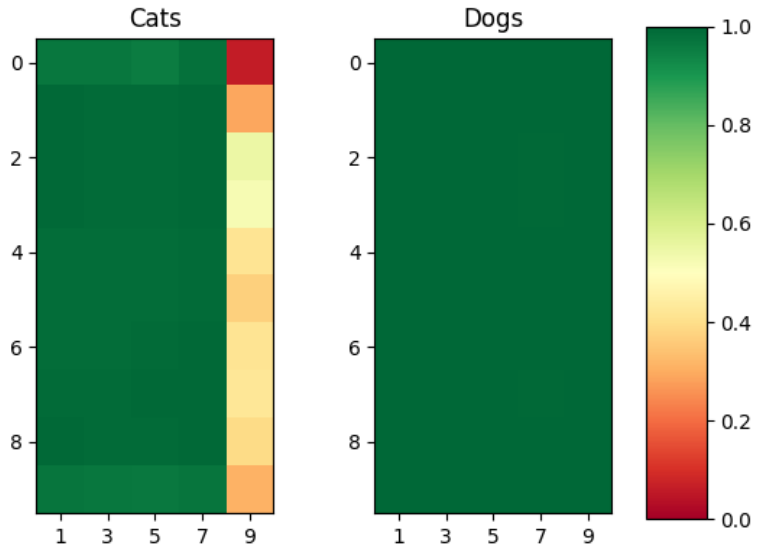

Figure 4.11: The probabilities of being a cat or a dog for the images of the test data, in which the cat and dog appear in odd columns, without using a validation set. It can be seen that the small variations in probability from figure 4.10 are gone. However, still the ninth column performs badly. Also in row 0 and 9 a small decrease in probability can be seen.

Another idea that emerged is that the translation invariance has occurred from the pooling layer. In theory, the pooling layer will make the network invariant for small translations. To test this idea, the pooling layer is removed from the network and the same experiment is done again without the validation set. It can be seen in figure 4.12 that in this situation the network has still learned the classification very well. Only for the images where the cat appears at the boundaries a clear decrease can be found. We can conclude that the pooling layer helps the network in becoming translation invariant, but is not the main reason for the appearance of translation invariance in this experiment.

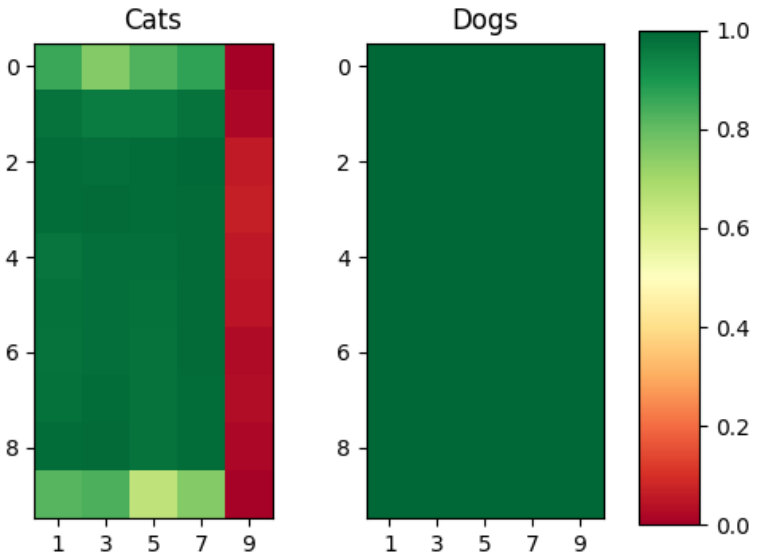

Figure 4.12: The probabilities of being a cat or a dog for the images of the test data, in which the cat and dog appear in odd columns, without using a validation set and a pooling layer. It can be seen that at the boundaries the networks performance decreases.

4.3.4. Kernel and feature map representation

In order to really understand what the network has learned in the training process, we could look into the kernels and the feature maps they produce after the convolution. We will take the network with the pooling layer from the previous experiments and train them such as in section 4.3.3 without the

validation set.

To see how the kernels look like, they can be plotted as images of 3×3 pixels. In these images the grayscale of each pixel corresponds to the learned weight in that position of the kernel. This will make it easier to see patterns in the kernels. In figure 4.13 the 8 kernels that the network has learned after 50 epochs can be seen. Also in each pixel the value of the weight is given.

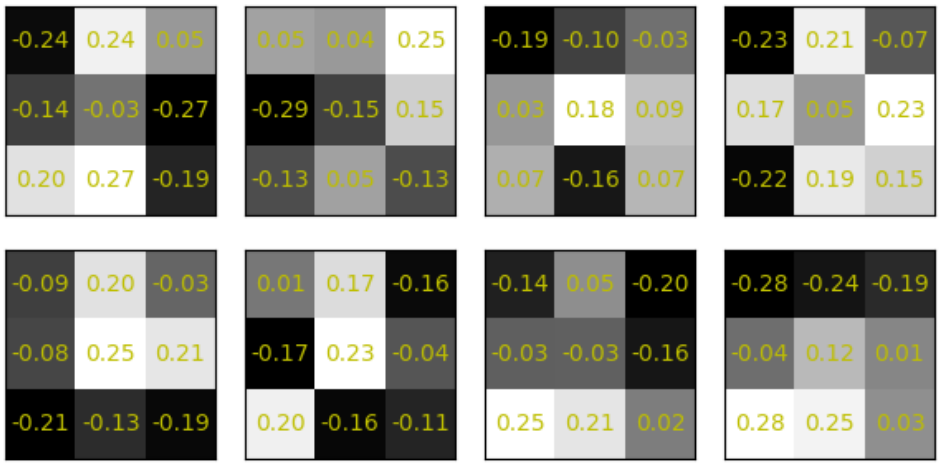

Figure 4.13: The 8 kernels of size 3×3 that the network from section 4.3.3 has learned. The grayscale of each pixel corresponds to the weight in it. It can be seen that some kernels show a nice pattern.

In some of the kernels nice patterns can be seen. For example, the kernel on the bottom left seems to have a L-shape and the kernel on the bottom right seems to have some horizontal lines in it. In order to understand even better how the kernels alter the images during the convolution, it is useful to also look at the feature maps they produce. If image 22 of the cat is taken, for example, and put through the learned network, the following feature maps are found.

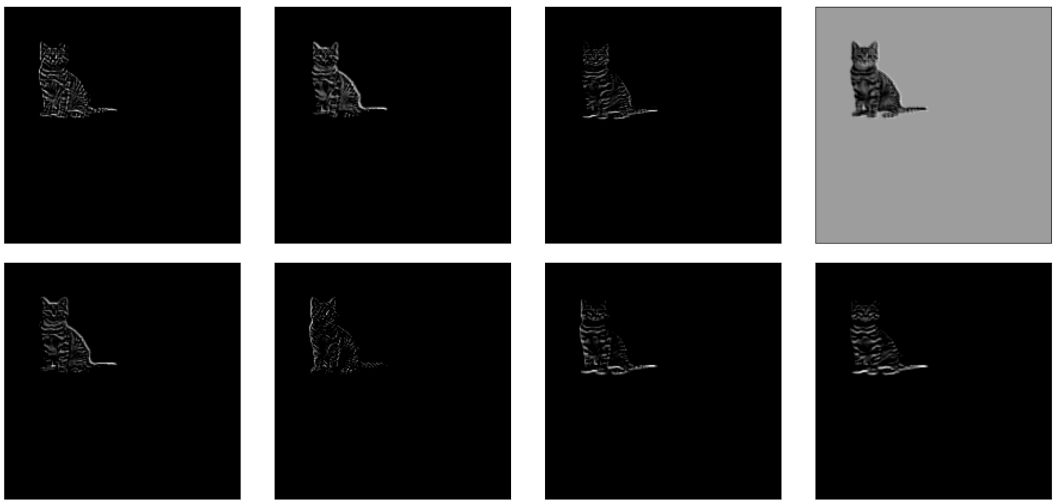

Figure 4.14: The 8 feature maps of image 22 that are made by convolution with the kernels from figure 4.13.

In figure 4.14 it can be seen that each of these kernels highlight other parts of the cat. For example the kernel on the bottom left seems to highlight the diagonal back of the cat. The horizontal lines from the kernel on the bottom right seems to highlight the horizontal bottom of the cat. After the convolution layer the maximum pooling layer will make these highlighted areas brighter, since the whiter the pixel is, the higher the value. This can be seen in figure 4.15.

The real difference between the cat and rhe dog is now learned in the last dense layer of the network. After flattening of the feature maps, it learns that if a certain combination of pixels is highlighted, the

image should be classified as a cat or a dog.

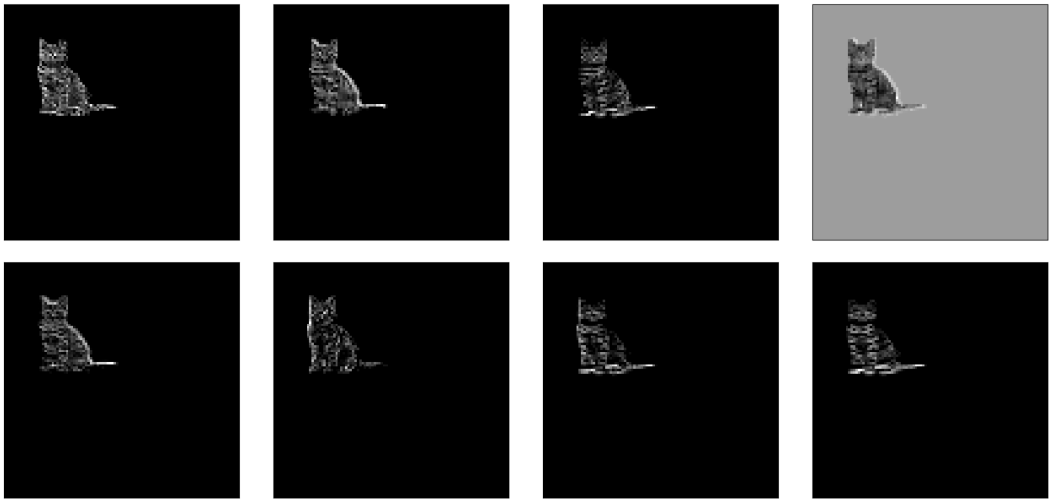

Figure 4.15: The 8 feature maps of image 22 after putting it through the maximum pooling layer in the network.

4.4. Rotating the cat and the dog

Now that we have explored translation, the question may arise if the network is to some extent invariant to other transformations, such as rotation. Since from the previous experiment we got invariance for small translations, it would be interesting to see if this also leads to invariance in small rotations. In this experiment, we would like to investigate for what angles of rotation the network loses its ability to classify accurately.

4.4.1. Data set

For this experiment the same data set as before is used, together with a new data set which consists of images with rotated cats and dogs. In this data set, the cat and the dog are placed in the middle of the images and rotated by 5 degrees every time, making a total of 72 rotated cats and 72 rotated dogs. In figure 4.16 two of the images can be seen, where the cat is rotated 0 and 45 degrees.

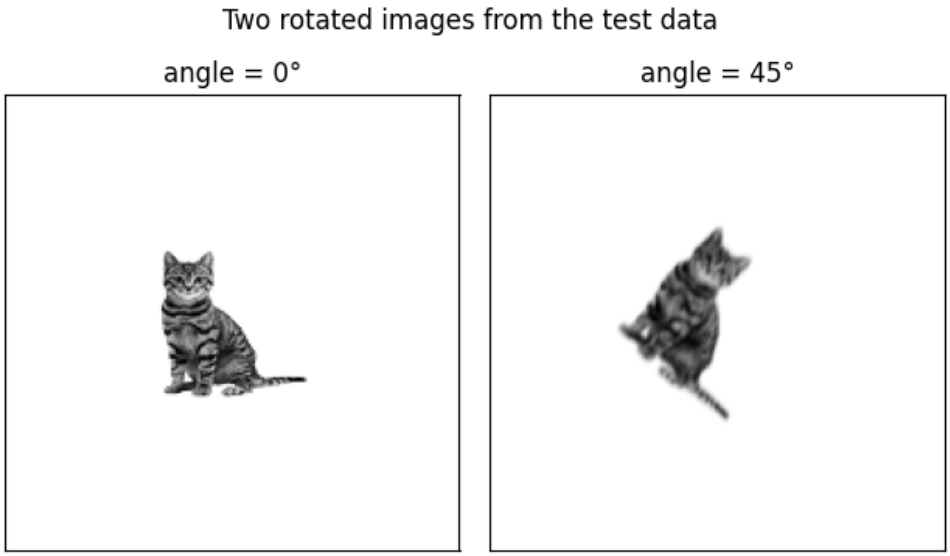

Figure 4.16: Two of the images from the data set of rotated cats where the cat is rotated 0 degrees and 45 degrees. It can be seen that due to the low resolution of the image and the fact that pixels are square that a rotation of 45 degrees will lead to a more blurred image of the cat.

4.4.2. Rotation equivariance

To show that the equivariance property of convolution does not hold for rotation, we will perform a convolution on two images that are rotated in different angles. These angles will be 0 and 90 degrees. For the convolution to be rotation equivariant, the feature maps of these images should be the same after one of the maps is rotated to the same angle as the other map.

The kernel that we will use is the Sobel edge detection kernel G_x. This kernel is used to only pick up the vertical edges that are in an image.

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \tag{4.4}$$

In figure 4.17 the feature maps of the two images can be seen. In both feature maps it can be seen that the vertical lines in the cats are highlighted. For example, for the cat that is rotated 90 degrees, it can be seen that the tail of the cat appears brighter. However, for the cat that is rotated with 0 degrees, this is not the case. A similar result can be seen between the ears of the cat. If the cat that is rotated 90 degrees now is rotated back to 0 degrees, the cats will appear very different. Therefore the convolution operator is not rotation equivariant.

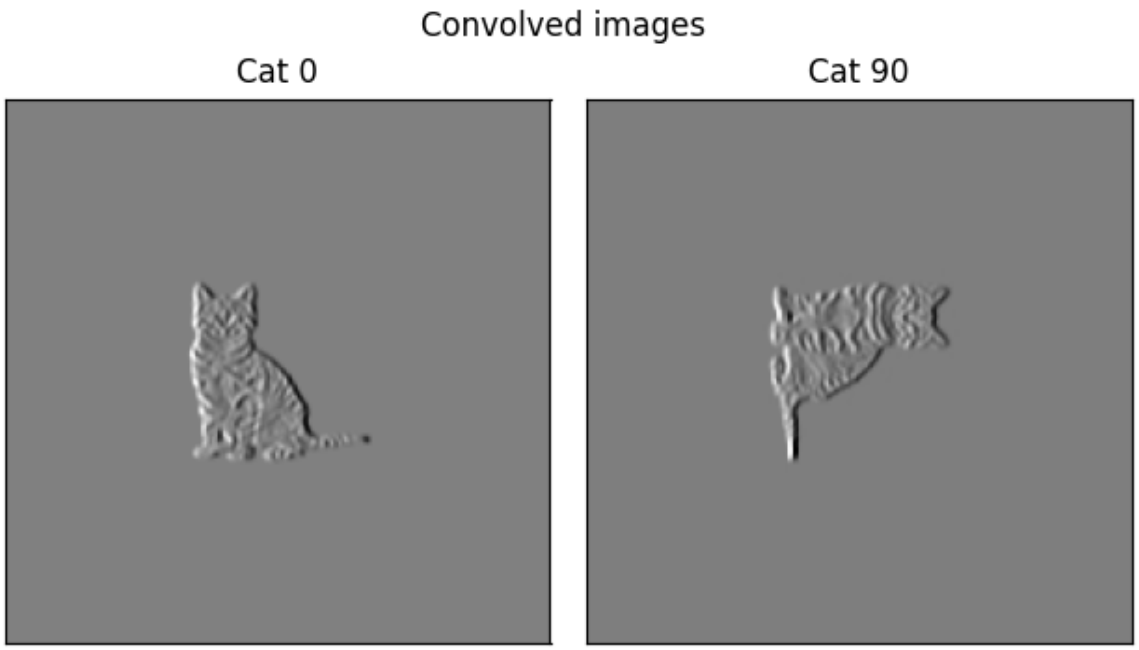

Figure 4.17: Two images of a cat rotated 0 and 90 degrees after a convolution with the Sobel edge detection kernel G_x.

4.4.3. Training and testing the network

The network that is used in this experiment is again the same network as in the previous experiments. For the training of this network the data set of translated cats and dogs will be used. However, 20% of this data will be used to validate if the network is able to learn the difference between the cat and the dog. Therefore, in total there are 160 images to do the training on and 40 images to do the validation. The batch size will be set to 10 for this experiment, making the network update the weights and biases 16 times per epoch. After the training we can see in figure 4.18 that the network indeed has learned the difference between the cat and the dog based on the translations.

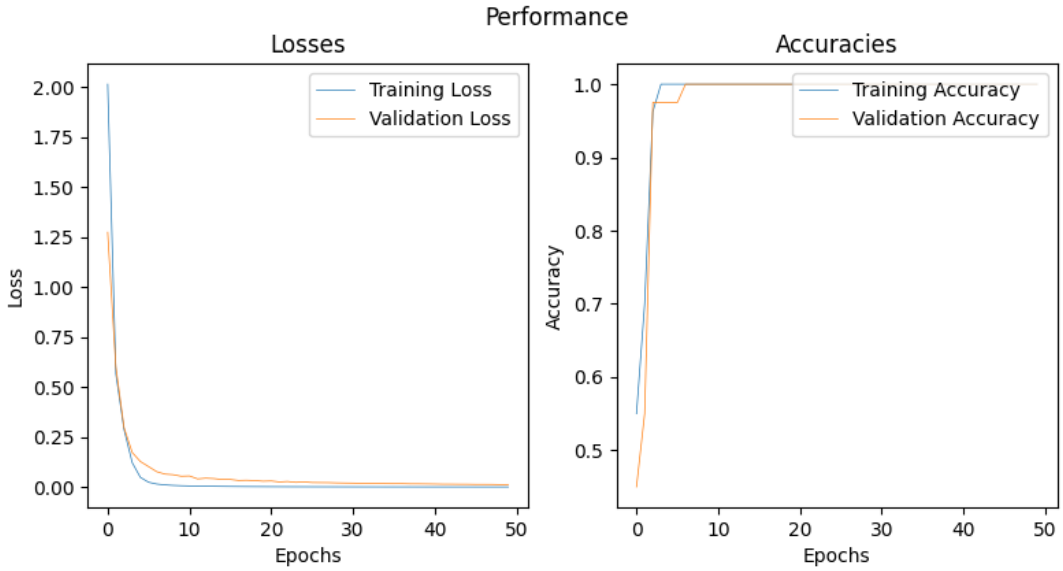

Figure 4.18: The loss and the accuracy for both the training data (in blue) and the validation data (in orange) after training the network on the translated images of the cat and the dog plotted against the number of epochs.

The testing will be done on the 144 rotated images of the cat and the dog. In this way we can see if the network can still recognise the cats and dogs regardless of the rotation. In figure 4.19 the probabilities for the rotation of the cat and the dog are plotted against the angle they appear in. It can be seen that for the dogs again the network performs really well. For every angle the network is more than 99% sure it is a dog. For the cats, however, it can be seen that the network has a lot of trouble to classify the rotated images. At 0, 90, 180 and 270 degrees high spikes can be seen. This is an interesting result. It means that at these angles the network is very sure that the image is indeed a cat, but if it is rotated only slightly by 5 degrees, the probability drops an enormous amount. Between the spikes, at angles 45, 135, 225 and 315 degrees even low troughs can be seen. This indicates that at these rotations the network has higher confidence of seeing the dog instead of the cat.

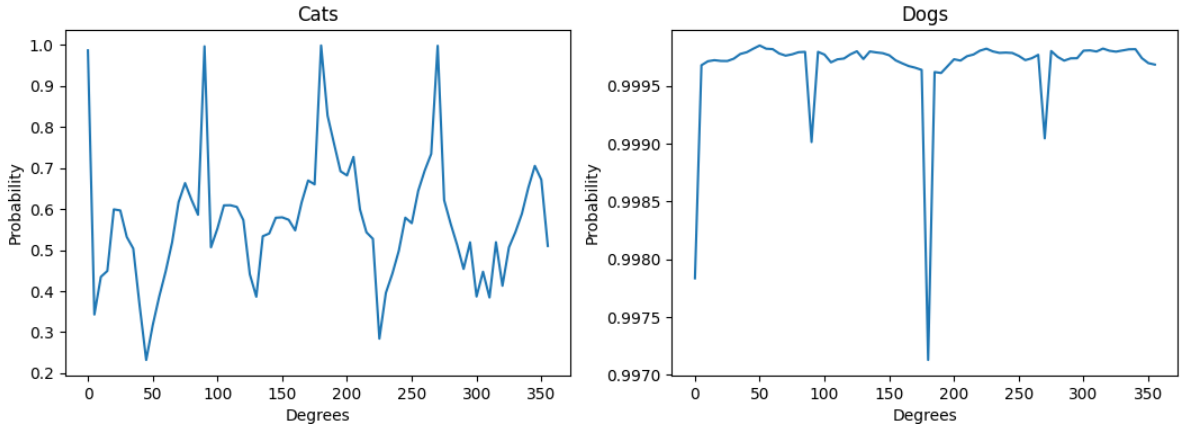

Figure 4.19: The probabilities of being a cat or a dog for the rotated animals in the test data plotted against the angle in which they appear. It can be seen that again the dog performs really well. However, the probability of the cat shows high spikes at 0, 90, 180 and 270 degrees and low troughs at 45, 135, 225 and 315 degrees.

4.4.4. Experimenting on the number of kernels

In an attempt to understand the spikes and troughs, we are interested if this behaviour is influenced by the number of kernels that is used in the convolution layer of the network. Therefore the experiment from section 4.4.3 will be done again but now for different amounts of kernels in the convolution layer. An idea that we could have is that the spikes and troughs may disappear if the number of kernels is increased, since then the network can extract more features from the images and therefore may be

better in distinguishing the cat from the dog. This experiment is also a nice way to see how many kernels are needed to see the difference between the cat and the dog.

In figure 4.20 the validation loss and accuracy can be seen for networks in which a different number of kernels is used plotted against the number of epochs. From the figure, it is clear that for networks in which two or more kernels are used after 50 epochs a validation accuracy of more than 90% is reached. Therefore, it can be concluded that with only two kernels the network is already able to classify the cat and the dog in translated images. From the loss and accuracy in the figure, it can also be seen that the more kernels that are used in the network, the quicker the network seems to learn the difference between the cat and the dog. The loss drops quicker with more kernels and also the accuracy increases quicker with more kernels. This feels really intuitive, since with more kernels the network has a higher chance to learn a kernel that extracts useful features. However, in using more kernels than strictly necessary, overfitting of the network can lead to a low accuracy for the validation data. In this experiment, this seems not to be the case. Probably multiple kernels will look quite similar to each other and function to extract the same features.

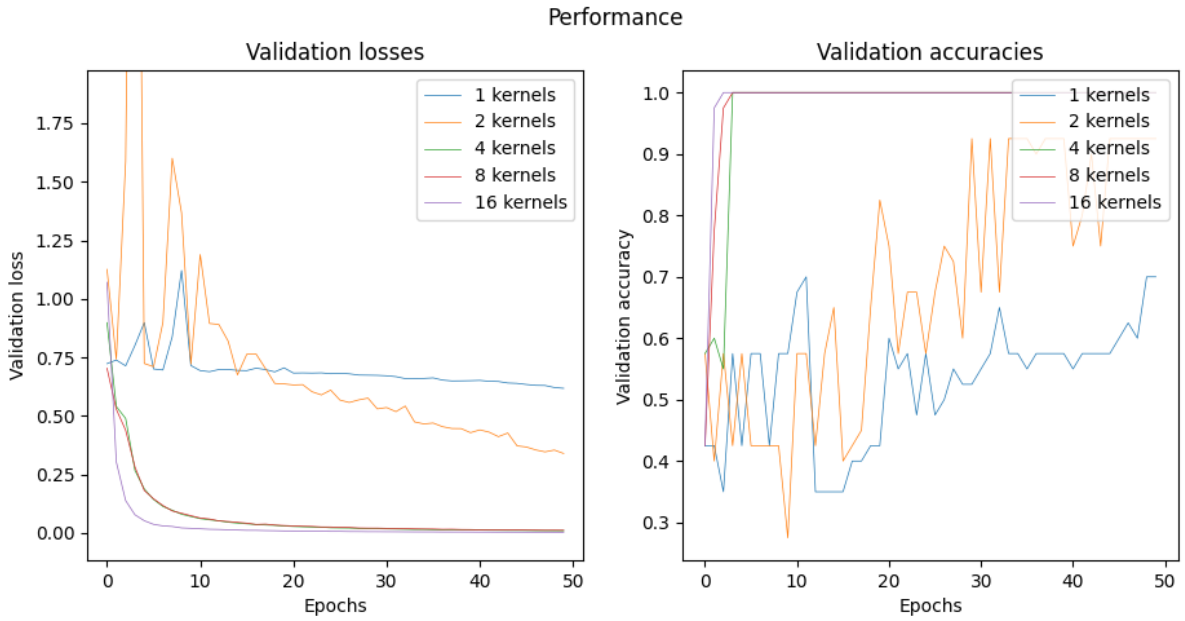

Figure 4.20: The loss and the accuracy of the validation data after training the network with different amount of kernels on the translated images of the cat and the dog plotted against the number of epochs. It can be seen that with using only 2 kernels the network already is able to classify the cat and the dog with more than 90% accuracy. Also it can be seen that using more kernels will lead to a quicker learning of the network.

In figure 4.21 the same image as in figure 4.19 can be seen, but now for different amount of kernels. For the dogs it seems that the more kernels that are used the closer to a probability of 1 the troughs get. For the cats, it can be seen that the probability still shows spikes at 0, 90, 180 and 270 degrees for the networks that use two or more kernels. It seems that there is not a clear correlation between the number of kernels and the height and width of the spikes. Also, since the training process has some randomness, it is very difficult to see a correlation if there is any.

An explanation that seems more plausible for the high spikes and low troughs could be an effect that occurs due to the low resolution of the images. Since the pixels are square, only at a rotation of a right angle the exact pixel values as in the original image will be used. In all of the other rotations the pixel values need to be calculated by performing an operation on the original pixel values. This leads to a blurred version of the cat or dog in these images. In figure 4.16, for example, it can be seen that the cat has lost a lot of its stripes when it is rotated with 45 degrees.

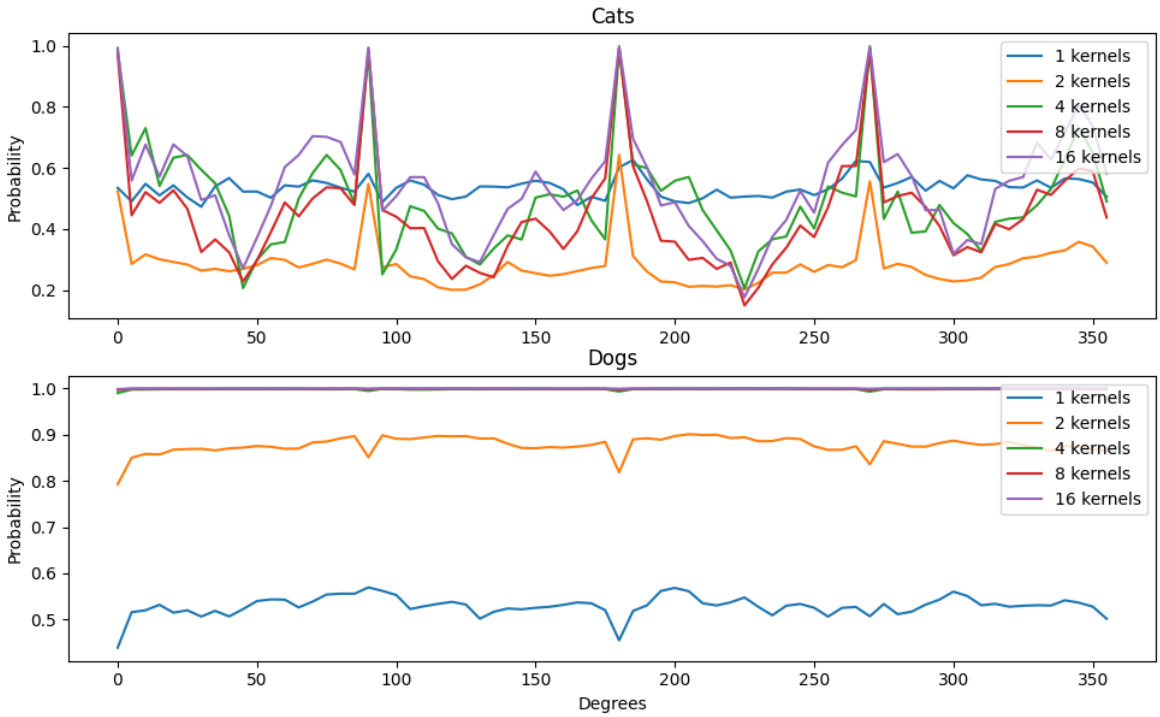

Figure 4.21: The probabilities of being a cat or a dog for the rotated images of the test data after training with different amount of kernels plotted against the angle in which the cat and dog are rotated in the image. It seems that for the dog the more kernels that are use the more the troughs go to a probability of 1. However, for the cats the spikes seem to be the same at angles 0, 90, 180 and 270 degrees for the networks in which 2 or more kernels are used.

In order to see how the network is already able to classify the difference between the cat and dog with just two kernels, we will take a look at the two kernels and their feature maps. In figure 4.22 the kernels can be seen that are learned based on the translated images of the cat and the dog. For both kernels some horizontal, but also vertical elements can be seen. The feature maps that the kernels produce can be seen in figure 4.23. It seems that the feature maps have to some extent reversed colors of each other. Clearly the background is a different color, but also for the cats it seems that this is the case. For example, on the chest of the cat on the left there are two white stripes, but on the cat on the right these stripes are black. A similar contrast can be seen for the tail.

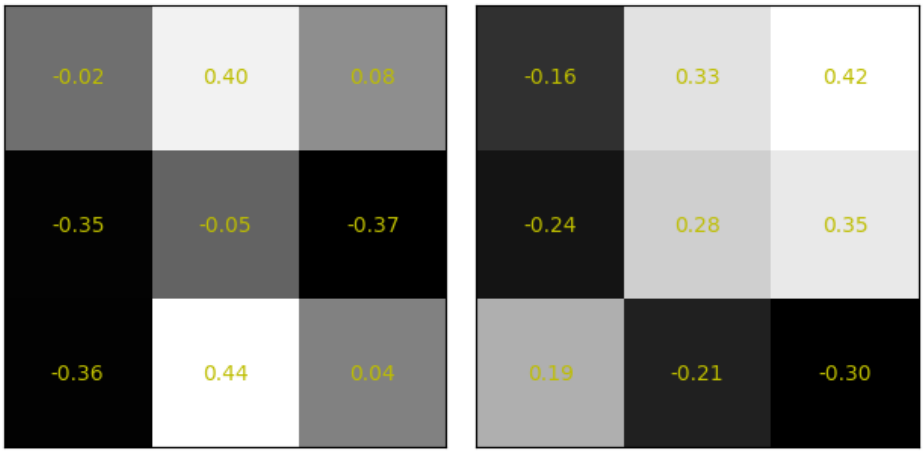

Figure 4.22: The 2 kernels that the network has learned based on the translated images of the cat and the dog.

Figure 4.23: The feature maps that are produces by performing a convolution with the kernels from figure 4.22.

5

Conclusion and Discussion

In this thesis we have looked into supervised learning on artificial neural networks. We tried to understand the working of every part of the neural networks and how the complexity has effect on their performance. In special, the convolutional neural network (CNN) is explored. By doing multiple experiments, we tried to gain a deeper understanding how the CNNs work and what their limits are.

5.1. Mathematical foundation

The idea for the artificial neural network comes from the structure of biological neural networks in the brain. In order to express an artificial neural network, in chapter 2 a mathematical foundation is build. Neurons that are arranged in connected layers will process input values by multiplying them with weights and putting their sum along with an extra value, the bias, in an activation function. The output value will then be transmitted to other neurons in the succeeding layers. The loss function will be used as a measure of the performance of the network and an optimization strategy is chosen to update the weights and make the network smarter.

In chapter 3 the CNNs were examined. These make use of the convolution operator in order to extract features from the input. For image recognition this type of network is especially useful and the convolution layer will make use of a two-dimensional discrete convolution. A kernel, which is trained, will alter the input image by sliding a flipped version of itself over the image. The parameters padding and strides can be defined in the convolution layer to tell how this sliding is done. After the convolution, a pooling layer can be used to lower the resolution of the input.

5.2. Experimenting on the CNNs

The experiments on a simple CNN are done in chapter 4. The first experiment showed that, after training the simple CNN on two images of translated cats, the network was not translation invariant. This led to the understanding of some other property that CNNs do have, which is called *translation equivariance*. This tells that the convolution commutes with translation. This property is shown in the second experiment by doing a convolution on two translated cats and comparing the outputs, which are called feature maps. In the third experiment, we tried to teach the same simple CNN to recognise the difference between a cat and a dog. It is shown that the network can do this very quickly, but fails on images that are translated too far from the training data, due to the difference between invariance and equivariance. However, this same experiment showed that the network is able to become translation invariant if the cats and the dogs are chosen to be more distributed in the training data. Also, in this experiment a look is taken into the kernels of the convolution layer and the feature maps they produce. In the last experiment, we have seen in an example that the network is not equivariant to rotation. Furthermore, the network is trained on the translated cats and dogs and was then tested on rotated cats and dogs. It is shown that at angles of 0, 90, 180 and 270 degrees the network is able to recognize the cats and dogs very well, but at other angles will lose this ability very quickly. In an attempt to explain this, the number of kernels was varied. However, it turned out that this does not have a big influence on this phenomenon. In addition, it is seen from this experiment that with only 2 kernels the network is already able to learn the difference in translated cats and dogs.

5.3. Reflecting on the experiments

During this research, we have tried to keep the experiments as simple as possible. We have used a simple CNN, but also simple data images. In this way, we tried to keep the experiments manageable and the results easy to interpret. However, after the experiments, it was observed that the images could be made even more simple. The images of the cat and the dog, we have used, are actually not that easy. The shape of the animals is very detailed and also the texture of the fur is very complex. For example, the cats have a lot of stripes. It turns out that as humans we tend to very easily underestimate the complexity of images, since we are very good in classifying images ourselves.

Because the images are so complex, it is possible that the network has learned to differentiate the cat and the dog from the horizontal stripes of the cat. This should not be done by the network, since there also exist cats that do not have stripes. Instead, the network should learn the difference, for example, by means of the position of the eyes and ears relative to each other. From the experiments it turned out that the location is preserved and therefore it should be able to learn such features.

Also, the fact that the dog seems to score a lot better than the cat in the experiments is remarkable. Since the classification is only done on two animals, one of the two does not need to be learned. The network could just always predict one animal, except for the images in which clearly the other animal appears. In this way, the network already has a high accuracy, but this clearly is not the preferred way to differentiate the cat and the dog.

Furthermore, in the experiments we have paid little attention to the effects of the valid pooling that was chosen in the convolution layer. Nevertheless, in the third experiment a drop in probability can be seen for images where the cat and dog appear close to the boundaries. We have explained this by the fact that for a position that is close to the boundary, there are less images in the training data that have a cat or dog near that position. However, the valid padding that is chosen could also play a role in this.

From the last experiment, the hypothesis emerged that the spikes in probabilities for 0, 90, 180 and 270 degrees is an effect of the resolution of the image. Since the images that we have used, have low resolution and because of the fact that pixels are square, at other angles of rotation the image seems to be blurred. However, it is still remarkable that the network is able to classify the images at angles 90, 180 and 270 degrees, since we have shown that convolution is not equivariant to rotation.

5.4. Recommendations for further research

Instead of using the complex images of a cat and a dog, we could have used simple geometric shapes, such as squares, circles and triangles. This would hopefully make it much easier to understand why the network has learned some specific kernels and which kernels are relevant for the classification. In doing such an experiment, it would also be interesting to see how the network will deal with polygons. For example, the question arises at which number of angles a polygon will be classified as a circle.

Also, in order to prevent the network from learning the stripes of the cat, experiments can be done on a data set with multiple different cats and dogs. If some of the cats do have stripes, but others do not, then the hope is that the network will unlearn to classify on the stripes.

In order to prevent the network from always predicting the dog for situations it was not trained on, the same experiments could be done on more different animals. In this way, the network is challenged more and the effect that dogs are always classified well, may disappear.

Moreover, our hypothesis that the resolution of the images has influence on the height of the spikes from the last experiment, could be tested. In increasing the resolution of the image, we expect that the spikes will become smaller.

Besides, experiments can be done on the parameters in the network that we have not explored. For example, it would be interesting to see how the size of the kernel in the convolution layer influences the features that appear in the feature maps. Also various paddings and strides in the convolution layer can be used. This way we will get an even deeper understanding of all the parameters in the network.

Lastly, next to translation and rotation, also other transformations could be applied to the data. For example, the effect of scaling or mirroring the images could be investigated.

Bibliography

[1] Jason Brownlee. *A Gentle Introduction to Cross-Entropy for Machine Learning*. https://machinelearningmastery.com/cross-entropy-for-machine-learning/. 2020.

[2] Vitaly Bushaev. *Adam — latest trends in deep learning optimization.* https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c. 2018.

[3] *Cat PNG image with transparent background*. http://pngimg.com/image/50514.

[4] *Dog Png - Sit Dog*. https://www.seekpng.com/ipng/u2a9o0u2r5e6r5e6_dog-png-sit-dog/.

[5] Sanket Doshi. *Various Optimization Algorithms For Training Neural Network*. https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6. 2019.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[7] Shadab Hussain. *Building a Convolutional Neural Network: Male vs Female*. https://towardsdatascience.com/building-a-convolutional-neural-network-male-vs-female-50347e2fa88b. 2019.

[8] Bharath K. *Understanding ReLU: The Most Popular Activation Function in 5 Minutes!* https://towardsdatascience.com/understanding-relu-the-most-popular-activation-function-in-5-minutes-459e3a2124f. 2020.

[9] Osman Semih Kayhan and Jan C. van Gemert. "On Translation Invariance in CNNs: Convolutional Layers Can Exploit Absolute Spatial Location". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2020.

[10] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[11] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[12] Laurens C. F. van Mieghem. *Music Genre Detection with Neural Networks*. 2020.

[13] Coenraad Mouton, Johannes C. Myburgh, and Marelie H. Davel. "Stride and Translation Invariance in CNNs". In: *Artificial Intelligence Research*. Ed. by Aurona Gerber. Cham: Springer International Publishing, 2020, pp. 267–281. ISBN: 978-3-030-66151-9.

[14] Karthikeyan NG, Arun Padmanabhan, and Matt R. Cole. *Mobile Artificial Intelligence Projects*. Packt Publishing, 2019. ISBN: 9781789344073.

[15] Sebastian Raschka. *Gradient Descent and Stochastic Gradient Descent*. http://rasbt.github.io/mlxtend/user_guide/general_concepts/gradient-optimization/. 2020.

[16] rawpixel.com/Freepik. *Blue neural network illustration Free Vector*. https://www.freepik.com.

[17] Luis Guillermo Serrano. *Grokking Machine Learning*. Manning Publications, 2021. ISBN: 9781617295911.

[18] Shuihua Wang et al. "Multiple Sclerosis Identification by 14-Layer Convolutional Neural Network With Batch Normalization, Dropout, and Stochastic Pooling". In: *Frontiers in Neuroscience* 12 (Nov. 2018), p. 818. DOI: 10.3389/fnins.2018.00818.

[19] Thomas Wood. *What is the Softmax Function?* https://deepai.org/machine-learning-glossary-and-terms/softmax-layer.

[20] Aston Zhang et al. *Dive into Deep Learning*. https://d2l.ai. 2020.