Expertise
and insight
for the future

Mikko Mäkelä

Understanding, Building and Analyzing Basic Convolutional Neural Networks

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communications Technology

Bachelor's Thesis

26 May 2019

Author	Mikko Mäkelä
Title	Understanding, building and analyzing basic convolutional neural networks
Number of Pages	37 pages
Date	26 May 2019
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Information and Communications Technology
Instructors	Aarne Klemetti, Researching Lecturer

The purpose of this thesis was to go through the basics of a convolutional neural network and Artificial Intelligence in general and build a guide for them. The theory is explained in a way that a beginner who is interested in AI but does not know the details can jump in and learn about neural networks, how to build and analyze them without technical jargon.

First, the study covers a brief history of artificial intelligence. Second, in-depth theory of neural networks is studied. Third, the theory is put into the test by building a convolutional neural network using Zalando's Fashion-MNIST training material to train the network for image recognition. Python is used to configure the network together with the power and ease of use of Tensorflow and Keras. Finally, the model building process is introduced step by step with clear instructions to demonstrate how not only to build a convolutional network, but also how to improve it.

As a result of this study a convolutional neural network was successfully trained to classify pieces of clothing. The network was tested against the test data, which revealed a flaw in the design of the network. Further analysis revealed that the network was performing worse against test data. The phenomenon is known as overfitting and is one of the main improvement points discussed. Building the network in a more efficient way is also proposed as a potential future improvement idea.

Keywords	Convolutional neural network, AI, Keras, Tensorflow, Python

Tekijä Otsikko Sivumäärä Aika	Mikko Mäkelä Yksinkertaisten neuroverkkojen ymmärtäminen, rakentaminen ja analysointi 37 sivua 26.5.2019
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Suuntautumisvaihtoehto	Tietoverkot
Ohjaaja	Tutkijaopettaja Aarne Klemetti

Insinöörityön tarkoitus oli laatia opas tekoälyn ja tarkemmin konvolutionaalisten neuroverkkojen maailmaan asiasta kiinnostuneelle aloittelijalle ilman turhaa ammattikielistä terminologiaa, joka voi vaikeuttaa asiaan perehtymistä. Asioihin pyrittiin paneutumaan tarpeellisen syvällisesti pysymällä turvallisen lähellä pintaa, mutta silti menemällä tarpeeksi syvälle, jotta oppiminen ja materiaali olisi kiinnostavaa.

Laadittuun oppaaseen kuuluu teoriaosuus, jossa käydään läpi lyhyesti tekoälyn historia ja tarkemmin neuroverkkojen teoriaa. Seuraavaksi syvennytään konvolutionaalisiin neuroverkkoihin ja niiden käyttöön kuvantunnistuksessa. Jokainen työssä käytetty termi avataan termille sopivalla tasolla.

Insinöörityön osana rakennettiin kuvantunnistukseen käytettävä konvolutionaalinen neuroverkko. Mallin rakentamiseen käytettiin Python-kieltä, ja se yhdistettiin Tensorflow'n ja Kerasin tehokkaisiin tekoälyyn erikoistuneisiin ohjelmakirjastoihin. Ohjelmointiosuus käydään läpi askel askeleelta, jotta lukija ei vain pystyisi tekemään saman itse perusohjelmointitaidoilla, vaan myös kykenisi haluessaan parantamaan mallia.

Kun malli oli valmis ja koulutettu Zalandon Fashion-MNIST-koulutusmateriaalilla, sen toimivuus ja tehokkuus testattiin ja tulokset analysoitiin. Työssä saatiin onnistuneesti rakennettua neuroverkko, joka oppi luokittelemaan vaatekappaleita oppimateriaalin avuin. Neuroverkon tehokkuutta analysoitiin testimateriaalin avulla, josta paljastui verkon heikkous: tuntematto-man datan luokittelu. Parannusehdotuksia olivat ylisovittamisen vähentäminen ja verkon rakenteen parantaminen.

Avainsanat	neuroverkko, tekoäly, Keras, Tensorflow, Python

1 Introduction

Artificial intelligence is an intriguing topic. It is utilized in every technical field possible with use cases increasing rapidly. For the average person, learning about neural networks and how they work can be cumbersome. To understand how machine learning and convolutional neural networks work, one has to go through a lot of jargon and an endless number of math functions that often do not help in understanding the basics of the technology. Even the most beginner friendly articles and books manage to fit in a lot of special words and expressions that easily become confusing. This thesis aims to bridge the gap between the jargon of the machine learning world and the actually useful basic knowledge that can open the door to the world of AI and neural networks.

This study takes a brief look at the history of artificial intelligence and neural networks: where the concepts came from and where they are now. Next, the basics of artificial intelligence, machine learning and neural networks are introduced in more detail. Moreover, more specific terminology related to image recognition and machine learning is covered and finally, a convolutional neural network for image recognition is programmed.

In the case study part of this thesis, Zalando's dataset, Fashion-MNIST is used as the basis for a convolutional neural network in order to train the model for pattern recognition purposes. The process of building a neural network and all the basic theory behind and around it is introduced step by step. After checking the results against test data, possible future improvement ideas are discussed.

By the end of this thesis the core concepts and ideas behind machine learning and more specifically image recognition should be clearer and building, analyzing and tweaking a basic neural network model should be feasible. The goal of this thesis is not only to invite the reader into the vast and sometimes cryptic world of machine learning, but to also spark interest in learning more.

2 A Brief history of Artificial Intelligence

2.1 The roots of Artificial Intelligence

Artificial Intelligence, AI, is everywhere now. But where did it come from? It hasn't been in the spotlight for too long, but with today's easy to learn, difficult to master technology and incredibly available knowledge it is possible for anyone to learn the basics and more. Most if not all of this information is available for free on the internet for anyone to consume, and just like with plain old programming, you can get very far with self-taught knowledge. It was not, of course, always like this; the modern concepts of AI were actually invented a surprisingly long time ago.

Around the change of the millennium AI started seeing uses in the logistics industry along with other uses such as data mining and medical diagnosis. AI research was started a bit before this though, in mid 1950's at Dartmouth college in New Hampshire, United states.[1] So, AI has come quite a long way and is at a major turning point in the late 2010's with the use cases going up by the week and hobbyists and researchers alike pushing the technology to new limits every passing year. Today when you open the home screen of your mobile phone, you can already see a dozen apps that all use AI, not to mention all the algorithms that connect your phone to the internet and other devices. Almost every competent application or service has a self-learning algorithm or two behind it to make using it more convenient or effective.

2.2 Artificial intelligence in the 2010s

One of the major moments of modern AI being brought to the eyes of the mainstream was when IBM's Watson competed in the Jeopardy! -TV show. It beat the other contestants, Jeopardy! legends Brad Rutter and Ken Jennings, with ease and showed the true capability and power of modern AI in a way any average Joe could understand.[2] This was back in 2011, which does not seem like a long time, but for technology, and especially such a hot concept like AI, it feels almost like an eternity ago. Since then we got Siri, Google's DeepMind and massive advancements in self-driving cars and disaster response robots[3]. AI even learned to play some video games better than humans.[4]

While the 2011 Jeopardy champion Watson is a question answering system, there are other types of AI that are progressing in complexness alongside it. Deep learning allows for even more incredible feats with the help of neural networks. This thesis focuses on pattern recognition (image recognition). Simply put, image recognition is when a computer is given an image, and the computer makes its best guess as to what that image contains.[5] This is done through a process of giving the machine a set of data to learn from, adjusting the process so the learned information is relevant to what we are trying to recognize and then applying the learned information to data outside from the training dataset. This process will be explained in depth in the coming chapters.

2.3 Image recognition: A form of AI

Image recognition is becoming more and more widely used in the modern day. It can be used in conjunction with security and street cameras to track peoples' movements, or car's license plates. It can also be used in self-driving cars to detect incoming threats or objects on the road. It can be used to automatically sort and categorize pictures in your digital albums, or to detect your facial features to unlock your smart device faster. The applications for image recognition, just like any type of AI, are seemingly endless.

This thesis takes a look at some of the more common ways for an amateur hobbyist to begin their image recognition programming journey. The information used in the theory part are from sources anyone has access to and the methods used in the case study part of this thesis are simple enough that most people with basic programming knowledge can reproduce the results.

3 Artificial Intelligence

The following chapter aims to explain the terminology and process of an artificial neural network from the perspective of image recognition in a simple yet effective manner. The neural network structure most used with image recognition, the convolutional neural network (CNN) will be used as the basis for explaining not only terminology related specifically to the CNN, but also AI and neural networks as a whole. But before we get to the nitty gritty of things, let's start from the very basics and get more familiar with the concept of AI.

3.1 What is artificial intelligence?

The term “Artificial Intelligence” simply refers to machines that work and show intelligence in a similar way to humans.[6] While AI possessing machines have yet to pass the Turing test[7], their biology and intelligence in regard to learning is still comparable to us. Artificial intelligence has become sort of a blanket term for all of the different fields of research related to machine intelligence, which there are quite a few. Some of the more well-known and researched parts of AI are for example speech recognition, machine learning and problem solving.[6]

A lot of the terms related to machine learning have quite unsurprising meanings behind them since they are often named very aptly, so a lot of them are self-explanatory and can be guessed as to what they are about. We will still go through the most important and defining ones since everyone should be on the same page as we get deeper into the subject.

3.2 Machine learning, a form of AI

Machine learning, as the name might already suggest, refers to the art of a machine learning from the data input into it. This is in contrast to a more classic approach to solving problems through machines, programming. Essentially, instead of coding a machine to do a specific task and only that task, a machine is instead programmed to learn and adapt to changing situations. This is achieved through the use of various algorithms that are specialized in improving, describing and predicting the outcomes of data. The learning part comes into play when the machine is fed training data, which in return helps it produce more and more precise models and predictions.[8] This learning allows machines to train and become well versed in the data they are supposed to be analyzing before actually being deployed and exposed to foreign data from outside the learning material.

3.3 Datasets, the basis of learning

Datasets are large pieces of information specifically meant for training a machine. The MNIST is one of these datasets, consisting of a database of handwritten numbers, ranging from 0 to 9. They are labeled, so a handwritten 0 would be labeled as “zero” in plain text. There are a total of 70,000 of these labeled images, 60,000 of which are training

images and 10,000 which are testing images in the MNIST database. This dataset is one of the most if not the most famous way of training a machine for image processing tasks.[9] The MNIST learning model is considered to be offline, which means that the dataset is a model that does not change and is a standard, set in stone. In contrast to this, some datasets are online and continuously adapt to the changing environment.[10] While it makes sense for something like a database of handwritten numbers to not change, some other datasets like ones based on a database of tweets or other online posts can be updated with new data on a regular basis.

The aforementioned way of teaching a machine with a database of labeled images is specifically for image recognition, as recognizing the features of handwritten characters can be utilized in a variety of image and pattern recognizing tasks. There are other types of training datasets for other types of machine learning purposes. If the goal of a machine is to perform natural language progressing tasks, then they could be fed a large amount of data containing synonyms, idioms and other complex forms of word for the machine to learn from, such as a database of tweets.

We have now established what machine learning is at the top level, the data that they learn from and how they use that learned information to analyze data. But *how* does machine learning work? How do they learn from the training data that they are ingesting? What is the actual process that connect the aforementioned steps into a functioning system? Let's dive deeper into neural networks.

3.4 Neural networks

A neural network is actually something we all have inside of our brain. It is a network of neurons and synapses that perform a specific task when a chemical or an electrical impulse activates them.[11] This is the exact same principle that is used in machine learning, or more accurately, in something called deep learning. These biological models based on the human nervous system that paved the way for modern artificial neural networks are some of the oldest concepts in AI. For machine learning, this network of neurons is split into multiple layers of different types that all have different functions within the greater network itself.

A machine's neural network has to have at least three layers for it to be able to function properly: an input layer, at least one hidden layer and an output layer.[12] The first layer,

the input layer, and the last layer, the output layer, are quite easily explained. The data is fed into the input layer and it comes out of the output layer as values. But what then happens in the hidden layer(s)? Any layer that isn't the input or the output layer is considered a "hidden layer". The hidden layer is a layer where the input data is worked on by the machine based on the weights applied on the artificial neurons of our network, or as they are called in this context, nodes. These nodes form connections between each other in a way similar to how our brain's neurons form synapses. This artificial neural network is in fact called just that, an artificial neural network (ANN).

Going deeper with artificial neural networks

An ANN can have thousands or millions of these simple nodes. Connections are formed and broken as data is being input into the network, and thus the machine learns what is important information and what is not while analyzing training data. Image recognition is one of the applications that most commonly uses neural networks in its design, because recognizing the relevant patterns needed from images is quite a complicated task in the complex world of artificial intelligence. For example, in facial recognition the machine needs to know which features to look for in a face and needs to adapt to different lighting situations or even facial hair. This process is quite complex in that it has a lot of variables and features that need to be considered, thus a beefy application of neural network is very useful. While the ANN is just a catch-all phrase for any kind of artificial neural network, there are many types of structures you can build these networks in. We will focus on the one mostly used in image recognition, the convolutional neural network.

The convolutional neural network

A convolutional neural network (CNN) is one of the most common neural networks used for image related training and analysis. The purpose of the convolution operation is to reduce the size of the image and to create a feature map. The feature map is created when during the operation a feature detector scans through the image looking for the most important features. This operation technically results in a loss of data since the resulting feature map is smaller in size compared to the original image, but the purpose as the name might suggest is to only keep the relevant features and get rid of the rest. Thus, the data that is lost is not really relevant in the first place, so while we save the important parts, we also get rid of the parts we don't want. The design of a CNN has an

input and an output layer and multiple hidden layers. These hidden layers include convolutional layers, which emulate the response of a neuron to visual stimuli.[13] In a CNN after the input layer the image first gets processed in the multiple connected convolutional hidden layers. The first hidden layer will capture for example rough edges and color, while the latter ones would be responsible for finer features and detail.[14]

3.5 Layers

As previously explained, every ANN has an input, an output and one or multiple hidden layers. While the input and output layers have the same purpose most of the time no matter what kind the neural network is, there are many types of hidden layers for many different purposes. Generally speaking, the more hidden layers a neural network has the more the performance decreases, as in the computational cost increases. For the majority of simple machine learning tasks, one hidden layer is good enough.[15] This is however not true for the type of neural network we are going to be analyzing and using in this thesis, which will be touched on in the latter chapters. As for what the number of neurons in each layer is, it depends a bit on the data that is being read, but there are some good rule of thumb practices that can be taken here. Generally, the number of neurons in the input layer should match the number of columns in the data, for example, if an image that is being analyzed is 28 x 28 pixels, then the input layer should have at least 28 neurons.

The output layer

The number of nodes in an output layer should also match the data in a way that makes sense. However, output layers are a bit different in the way that the data that eventually comes out of the output layer at the end of the training process should also be able to be turned into a format that humans can look at and understand. If a neural network is supposed to return a single value, for example the price or something, then it makes sense for the output layer to have just a single node that gives us that piece of data.[15] If the purpose of the neural network is to however, for example, try to figure out which piece of clothing is in an image it is analyzing, then we should have one node per each class label in the data. These labels can for example be “Sneaker” or “T-shirt”. The output should be setup like this because then we can get the probabilities for each of the classes in the final result.[16] All of these nodes add up to 1, or 100%. So, an item could have a

95% chance to be a shoe, a 5% chance to be a sneaker, and then the rest of the nodes would be at 0%.

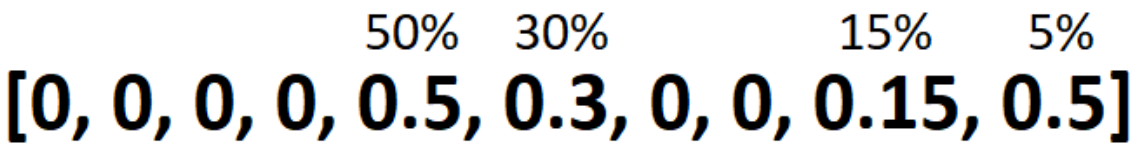

Image 1. How an output layer outputs the probabilities of the classes.

The flatten operation

Hidden layers can also be used in ways other than pure learning. This is where the fact that multiple hidden layers being useful for an image recognizing network comes into play. There is a very useful layer type for image recognition called flatten. The flatten layer turns the format of the images from a normal 2d image (for example 28 x 28 pixels) to a 1d-array of 28 * 28 = 784 pixels. The only purpose of this layer and the end result of the 784 pixel array is to reformat the data into a more efficient form.[16] The 1d-array of 784 pixels is digested by the network way easier than the clunkier original data format. Flatten is often used in a convolutional neural network as the first step, even before any other image analyzing is performed. This is so that the process can be optimized as much as possible, since the computational cost of the training process should be kept as low as possible. Optimization and the fight against slow algorithms will become a recurring theme in other parts of the training process as well.

Pooling

Another layer used in convolutional neural networks is a pooling layer. Pooling helps the neural network with detecting the object it is trying to recognize no matter the orientation, size, lighting or other outside factors that might make the conditions unideal for recognition.[17] The end result of pooling, teaching the neural network how to recognize patterns despite differing conditions, is called “spatial variance”. In recognizing pieces of clothing this would be very important, especially if they are being detected while worn. A good spatial variance is crucial when leaving the training environment and starting analyzation of foreign, real world data. No handwritten number or t-shirt will look exactly the same or be in the exact position of the training images.

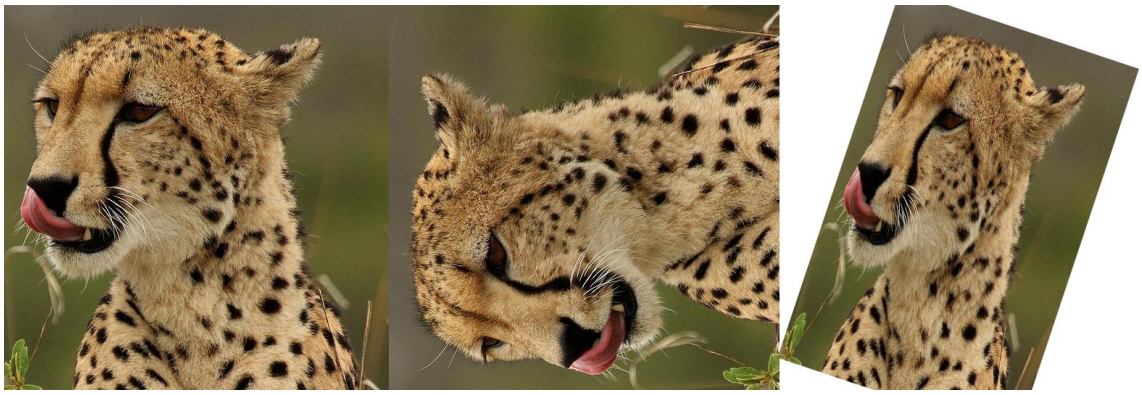

Image 2. Example of a situation where spatial variance is needed.

The fully-connected layer

As the different types of hidden layers pile on, the process continues. A layer also used in CNNs and image recognition in general is a fully-connected layer. This layer is the meat on the bones of the neural network, doing the work required for the neural network to learn based on the training data. The fully-connected layer connects every neuron in the previous convolutional layer to every neuron on the fully-connected layer, hence the name.[13] The fully connected layer takes the input from the previous layer, and outputs the probabilities for each of the classes in the system. This is why the fully-connected layer is always the last hidden layer right before the output layer. The filters have created activation maps in the previous convolutional layers and detected features that represent the different classes in the data. These activation maps will be explained more in depth later on, but for now imagine a map of numbers that represents the image, where concentrations of higher values represent different detected features. For example, if the neural network is recognizing animals and the previous layer has high values for spots, paws and other feline features, the fully connected layer could give a high probability to a cheetah, and maybe a lower one for a bird or a crocodile.[18]

There are many types of layers used in CNNs, and while there are even more used in other networks and applications, we will not get into every single one. Not only are there many types of layers, there are also multiple ways of implementing them. We will talk about these different “archetypes” of layers in the next segment, where the performance and computational costs of the layers we use get optimized for different use cases.

3.6 Types of convolutional layers used in CNNs

The 2D convolutional layer

The type of layer architecture that was explained in the previous section is called a 2D convolutional layer. The 2D convolutional layer is the most common type of layer used in image recognition for its simplicity and very good results, but this has drawbacks in certain implementations. The way the 2D convolutional layer works, is that it uses a filter to slide on top of the image from left to right, top to bottom, scanning it and inputting values for the resulting activation map. It is good at detecting the simple features of an image such as edges and other simple shapes, but this comes at a performance cost. This is not a problem for a powerful desktop, or especially a data center, but since performance can be a problem on less powerful systems such as smart phones or IoT devices, researchers started looking at alternative ways to get similar results and performance from a layer with less computational power required.[19]

The dilated convolutional layer

One of these is a Dilated or Atrous convolutional layer. The way the dilated convolutional layer manages to save in performance is by adding a new feature to the basic 2D convolution layer called the dilation rate. The value of the dilation rate determines the distance between the individual pixels of a filter. So, when we increase the dilation rate of the filter, the area it can cover with the same amount of pixels is higher, thus the performance of the operation doesn’t take a hit even though the filter works more efficiently. If a filter was 3 x 3 pixels before, adding a dilation rate of 1 to it would increase the field that it covers to 5 x 5 pixels, or a total of 25 pixels, while the actual filter only has a total of 9 pixels it is working with.

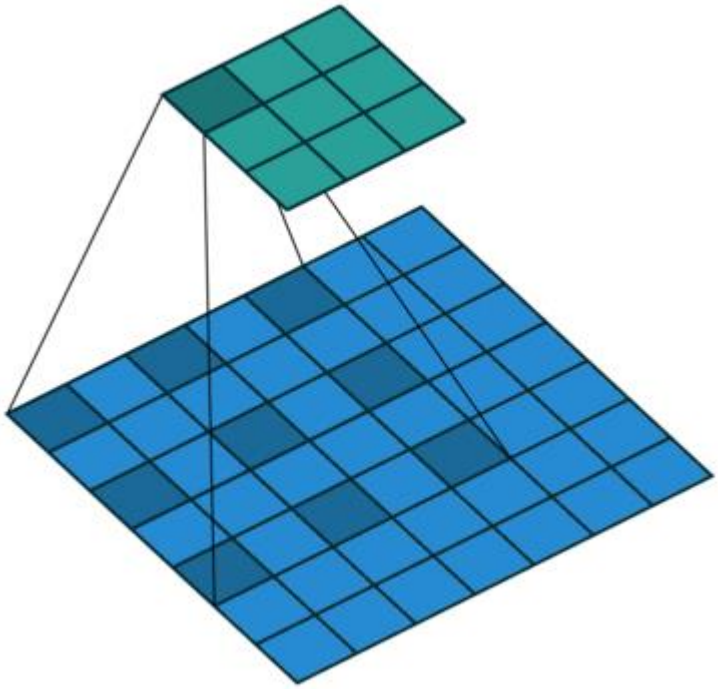

Image 3. A filter with a dilation rate of 1.[40]

Dilated convolutions are useful when detecting details in high resolution images, since they cover a large area.[19] Thus dilated convolution applications are best at getting a wider context of the input, while also keeping track of finer details and keeping the computational cost down. However, since we are using one pixel to represent a larger area, there is a chance for information loss.

The separable convolutional layer

Where dilated convolution decreases the computer cost of the convolution operation, the separable convolution layer decreases it even further. There are two main types of separable convolution layers: spatial and depthwise. The way the spatial separable convolution layer saves in computational costs is by multiplying two smaller convolutions together. A normal filter could have for example 3 x 3 pixels and 9 parameters attached to it, while the same convolution can be added up from a 1 x 3 and a 3 x 1 filter. This reduces the number of parameters down to 6 from the previous 9.[19]

$$\begin{bmatrix} 3 & 6 & 9 \\ 4 & 8 & 12 \\ 5 & 10 & 15 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix} \times \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

Image 4. Separating a 3 x 3 filter into a 1 x 3 and a 3 x 1 filter.

The spatial separable convolution is a bit less commonly used compared to the depthwise separable convolution since not all machine learning applications support splitting the convolution operation in two. The way the latter works is a bit more complicated, but regardless it is more commonly used in machine learning. The depthwise separable convolution is named so because it deals with depth of the input image, the RGB (Red Green Blue) values. So, while an image can be 28 x 28 pixels, it still has three different values for red, green and blue, so technically the image is 28 x 28 x 3 in size. The depthwise separable convolution still separates the convolution into smaller parts, but it does this by splitting the depth layers into three different channels. So instead of having one 28 x 28 x 3 image, we instead have three 28 x 28 x 1 images that get convolved by the filter at the same time and then multiplied together for the output. The actual convolution, called the pointwise convolution, uses a 1 x 1 x 3 filter, and in the end combines the three separated depths into one 28 x 28 x 1 output.

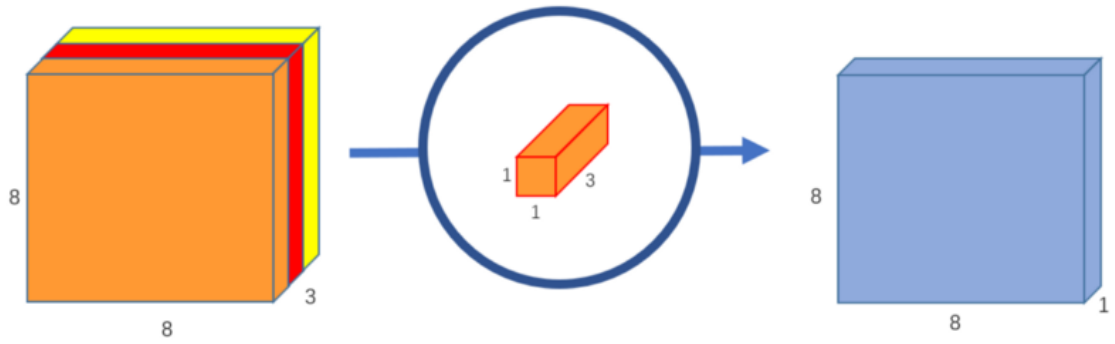

Image 5. A pointwise convolution where a 3 channel image is transformed to 1 channel.[20]

The way this operation saves up computational costs compared to the original 2D convolution has to do with the amount of multiplications they do compared to each other. The separable convolutional layer manages to process the images with a way lower

amount of calculations compared to the normal 2D convolution. This reduced amount of calculations, again, does not of course come free of charge. Because the number of parameters is reduced compared to the 2D convolution, there is a chance some crucial data could be lost. In the right hands however the depthwise separable convolution can increase efficiency while retaining a more than acceptable accuracy.[20]

Going backwards: the up-sampling operation

While it is useful for the machine learning process to convolve an image to a smaller set of values, it can be equally as useful to be able to turn the image back to its original size. The operation that allows this is called up-sampling, or in other words, turning a low resolution picture to a high resolution picture. The CNN layer that can perform this operation is called a transposed convolution layer (also known as deconvolution layer). This operation is similar, but not quite the same as going backwards in the convolution operation. While the convolution operation condenses a larger amount of data to one value, the transposed convolution operation turns the one value to multiple, for example 9 values.[21]

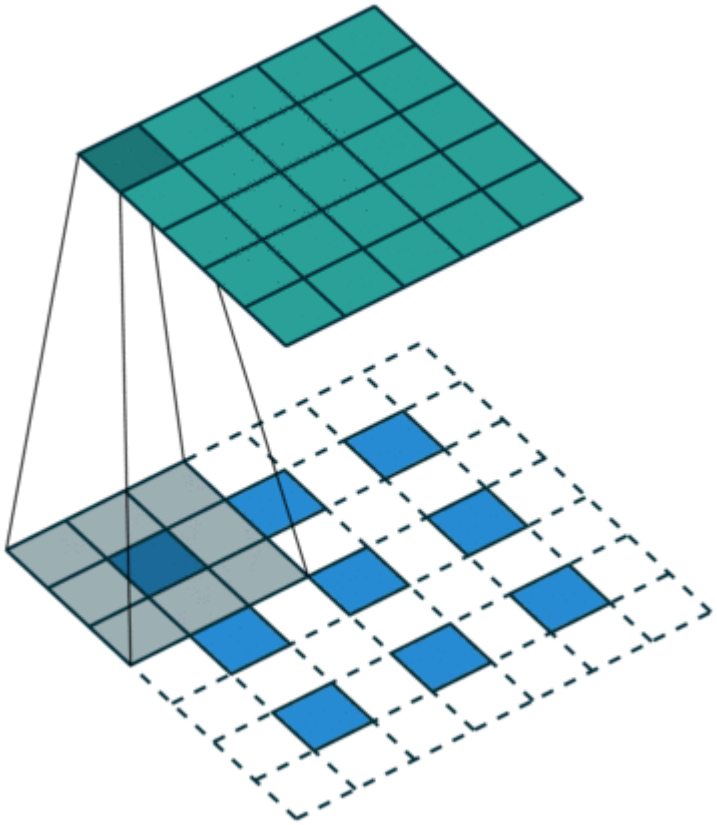

Image 6. The filter in a transposed convolutional layer.[41]

The way the layer works is that the operation simply spreads out the information from the original input (blue) onto a larger array while padding the rest of the values with zeroes (white). This padding can manifest in checkerboard-like patterns on the images which have undergone the transposed convolution operation,[22] thus while data is not lost since nothing is removed, the output will not be as clean as the original high-resolution image would have been. Since the information is spread out and padded it will not have the same values as the original input image, but more importantly it does mean that the output image will be the same resolution as the input.[23] A use case for the up-sampling operation in regard to image manipulation with AI is for example denoising pictures. One first down-samples a grainy picture, and then up-samples it back up with some clever use of algorithms to end up with a way clearer output than the original image.[24]

Making savings, is it worth it?

In conclusion, the trusty 2D convolutional layer is the best option if computational power is not an issue, for example in desktop environments or other even more powerful setups. This provides the highest amount of accuracy and reliability but may become too slow in mobile or IoT situations. This of course also depends on the size of the network itself, as a massive network might be slow even on desktop systems. The dilated and depthwise convolutional layers are more efficient in situations where saving in costs is required but they come at a slight loss in accuracy. Thus, the efficiency and accuracy should be balanced accordingly based on the expectations of the learning process and also the capabilities of the hardware being used.

3.7 Machine learning terminology

The reason why there has been so much discussion specifically about the convolutional neural network so far is because they are what we will mainly be using in our research part in latter chapters. A lot of the terminology applies more generally than just to CNN's, but it is still good to note that especially the following parts are written with that specific neural network structure in mind. Now that we are done with the more general AI concepts, it's time to go through some more advanced training related terminology.

Weights

As previously explained in chapter number 3.4.1 a neural network has millions of nodes, or neurons. The connections between these determine how a machine learns certain aspects about the data it analyzes. These connections can also be weighted. What this means is that the network learns to consider some of the connections to be more important than others. Let's take identifying a piece of clothing from a picture as an example again. Our goal is to identify which piece of clothing it is (t-shirt, sweatshirt, pants etc.). So, we could have multiple types features that are related to clothing, for example silhouette, relative size and color. These three types of data should not all carry the same importance, as it is quite obvious that while color can be useful for determining the type of an item (for example leather shoes might be black or brown more commonly than purple) it should not be as significant as the silhouette. Then, the output will be more accurate than just a simple $x_1 + x_2 + x_3$ formula where the values of the features would be added up with no weights applied to them. The training process through which the

computer adjusts its filter values (weights) and what features are actually significant is called backpropagation.[18]

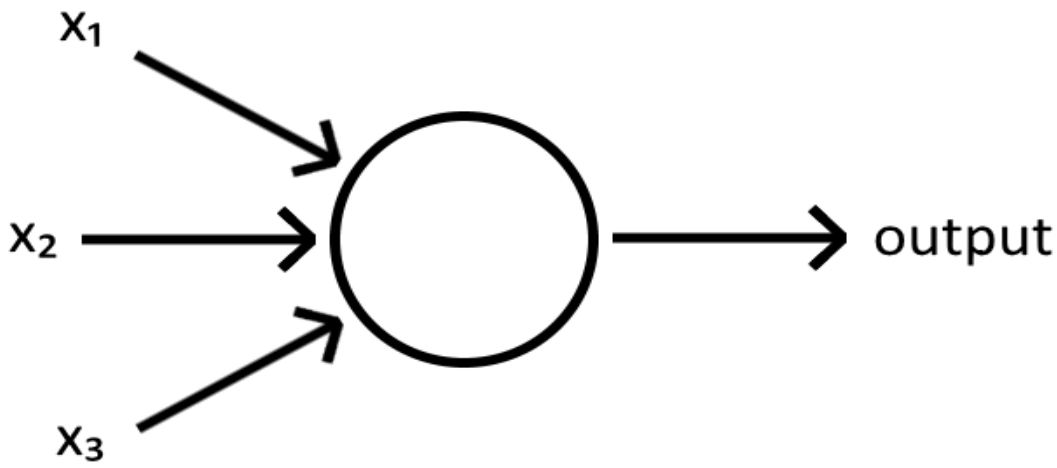

Image 7. Three weighted values going into a neuron (node) and coming out as the output

Forwards- and back again, the propagation operation

When data goes through the layers and ends up in the output and gets compared to the training labels, the whole operation is called forward propagation. The word propagation comes from biology, which means growing new plants from seeds and other various parts of plants. When the data is met with some error at the output, as in it wasn't able to correctly identify any of the classes from the picture, it is sent back through the network in order to correct the error.[25] The way this, backpropagation, is done is when the data is propagated backwards through the network, the derivatives of the errors are found for each weight and then the value of this is subtracted from the weight's values. This way the weights get adjusted to proper values, making the machine appreciate some features more than others. The name backpropagation comes from the way the operation works, as it has the same principle as forward propagation, but is performed in reverse.

Filter related terminology

Let's say our training images are 28 x 28 pixels. What happens to this image when it enters the first convolutional layer is that it gets passed through by a filter, scanning one area at a time. The area that the filter is currently targeting is called a receptive field, which matches the size and location that the filter is currently in.[26] The location of the

filter on the field also matters when it comes to the importance of a feature, which is called the center location. Center location refers to the fact that the closer the receptive field is to the center of the image, the more it affects the outcome of the output, meaning that the center pixels are considered more important to the recognizing feature. Depth should also be taken into account for the filter. Since an image also contains values for each pixel, the filter should have a depth of 3 (for red, green and blue). A filter in this case then would be for example 5 x 5 x 3 pixels, and it passes through the whole image from top to bottom until it has scanned all the pixels of the image.[18] The act of the filter passing through the field of pixels is called a stride, and the movement itself is called convolving. The length of the stride can be any number of pixels, but it should at least be short enough to cover all the pixels in the image. A stride of 1 would mean that the receptive field moves by one pixel with each stride.

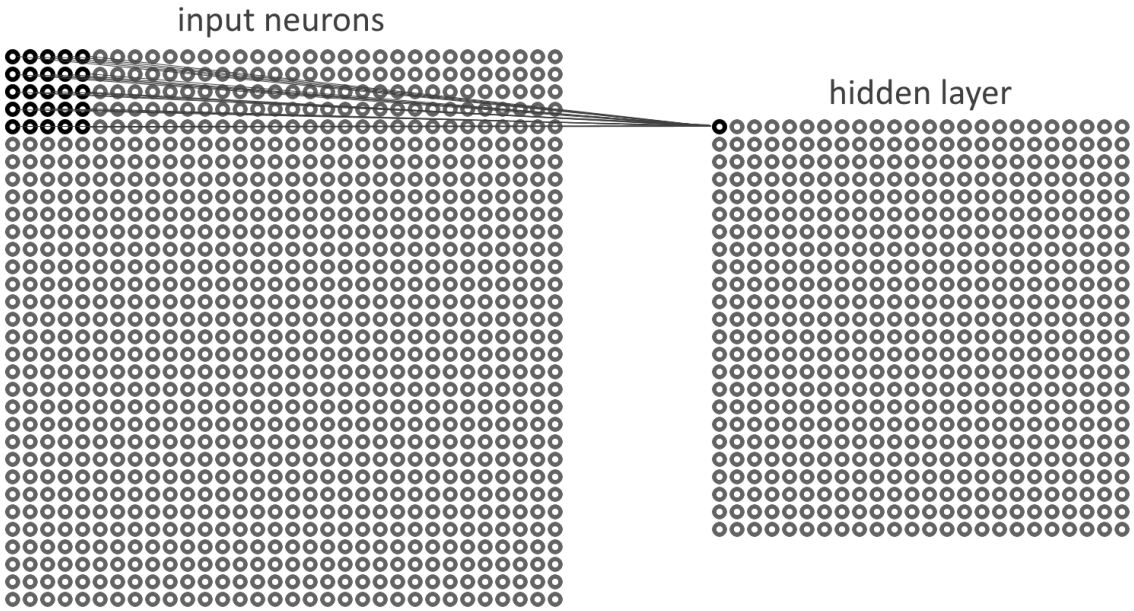

Image 8. A 5 x 5 filter producing an activation map into the hidden layer

As the filter convolves through the input data, our 28 x 28 pixel image, it is reading the data from the pixels as values, and then multiplies the values in the receptive field in between each stride. After this convolution operation is done, we are left with the activation map. The activation map will be a 24 x 24 x 1 array of numbers, since the 5 x 5 filter can move to 576 different pixels total in our original input image.[18] It is also possible to use more than one filter on an image. In this case the output would be 24 x 24 x 2 (for two filters), which means that the output values would preserve the original images features more accurately. In regard to image recognition, it is very beneficial to have multiple

filters, since one filter generally focuses on one feature. An object could have tens of features that make up the whole picture, no pun intended, thus the more filters we can have then technically the end result should be more accurate as well.

What are features to a machine?

Convolutions have been used in the past in image and signal processing before machine learning took use of them. Some of these filters might be familiar to users of Photoshop or other image editing software. Some filters used in image processing are for example Identity, which returns the same value as the input, Edge detection which as the name would suggest detects edges of objects in the image. Sharpen and blur either sharpen or make the image blurrier.[27] In image processing there are some set filters that are used, but the use cases of filters in machine learning are a bit more specific and out there, and the value of them is learned during the training process.[19] When detecting clothing pieces, humans might have certain cues they would detect a t-shirt by. These could be the neck opening shape, the length of the sleeves compared to the body of the clothing piece and so on. A machine, however, is capable of finding very specific features of clothing pieces that humans might not even take into consideration. There could be for example a curve detector filter than detects different curve shapes from an image and would find that useful when determining if a piece of clothing is a t-shirt or not. The filters in image recognition can get very abstract, since the detector is a machine and not a human.

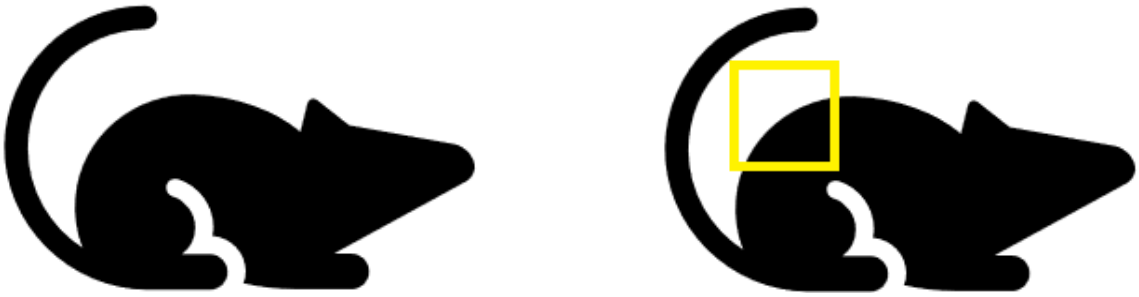

Image 9. Example image of a rat and a visualization of how a “curve detector” filter could work.

What these filters represent can be described in real world terms as feature identifiers.[18] These features were touched upon in our previous segment about different weights for different features in between neurons. It is because of these features that it makes sense to have multiple filters, so we can cover as much ground as possible. In a

2D convolution layer a random distribution is chosen for each filter, and these distributions can be for example Normal or Gaussian, which causes the resulting filters to get trained a bit differently from each other. This leads to them detecting slightly different features in the images, thus making the image recognition more concise and precise. Each filter generates its own feature map, and in the end all these feature maps get added up to a single output. The 2D convolution layer is the best at detecting these simple features in the first few convolutional layers of a CNN.[19] When you go past the first convolutional layer and you have identified some high level features such as curves, sleeves or buttons, the activation maps generated by the filters get passed on to the next convolutional layer. In the next layer, and the ones beyond that, the features get a lot more complex and specific, as we are now looking at features already detected by the first layers instead of just the original image. After this, the output of the last convolutional layer gets passed on to the fully connected layer, which outputs the probabilities of the different classes. Now that our neural network is trained it is ready for inference.

Welcome to graduation: Inference

From the Oxford English dictionary, the definition of inference is “a conclusion reached on the basis of evidence and reasoning”, which is quite accurately what it means in our case of AI as well. The evidence in our case is the training that we performed before this step, and the reasoning is the weights and backpropagation used to transform those weights to something reasonable. Our convolutional neural network is now a trained and is ready to use the tools and methods it learned in the real world.

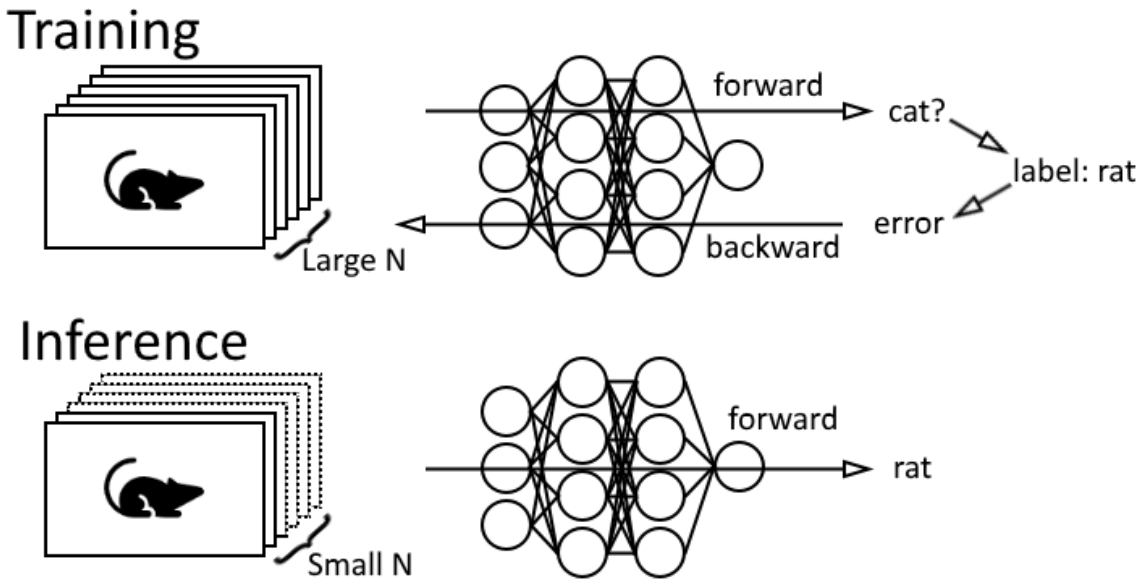

Image 10. The training operation compared to inference

As one can tell by the amount of different operations, terms, calculations and outcomes a convolutional neural network, and AI in general possess, it is quickly clear how complicated and intricate the whole process is. Math and other deep calculations and meanings were omitted from the previous chapters, so they can act as an entry point to the world of AI for anyone curious. Now that we have gone through the whole training process of the system from start to finish, and the most important terminology related to it, we can finally get to training our very own image recognition network.

4 Case study: Fashion-MNIST as a basis for image recognition

As has already been established in this thesis, the goal is not to program the deepest neural network possible, but instead to dive just deep enough where anyone is able to understand the content and hopefully themselves get interested in the world of AI. Our goal is to write a convolutional neural network and train it using Zalando's Fashion-MNIST to hopefully recognize pieces of clothing. We will use the methods and practices established in the previous chapters to achieve this goal.

In the following chapters we will touch on the backbone of our whole operation, the Python programming language, and the tools and modules that go along with it. These will be the means, along with the basic knowledge of neural networks, that will allow us to build a network of our very own.

4.1 Our dataset of choice: Fashion-MNIST

Fashion-MNIST is a dataset of images of clothing put together by a team at a research branch of Zalando. The dataset contains 60,000 training images and 10,000 test images, made to replace the original MNIST dataset. The reason for this was because the original MNIST is not that efficient at benchmarking algorithms. It's too easy, meaning that most of the handwritten digits can be distinguished from each other pretty well by just one pixel. This basically means that even poor algorithms can get pretty good scores on MNIST, compared to Zalando's Fashion-MNIST which is a lot harsher.[28]

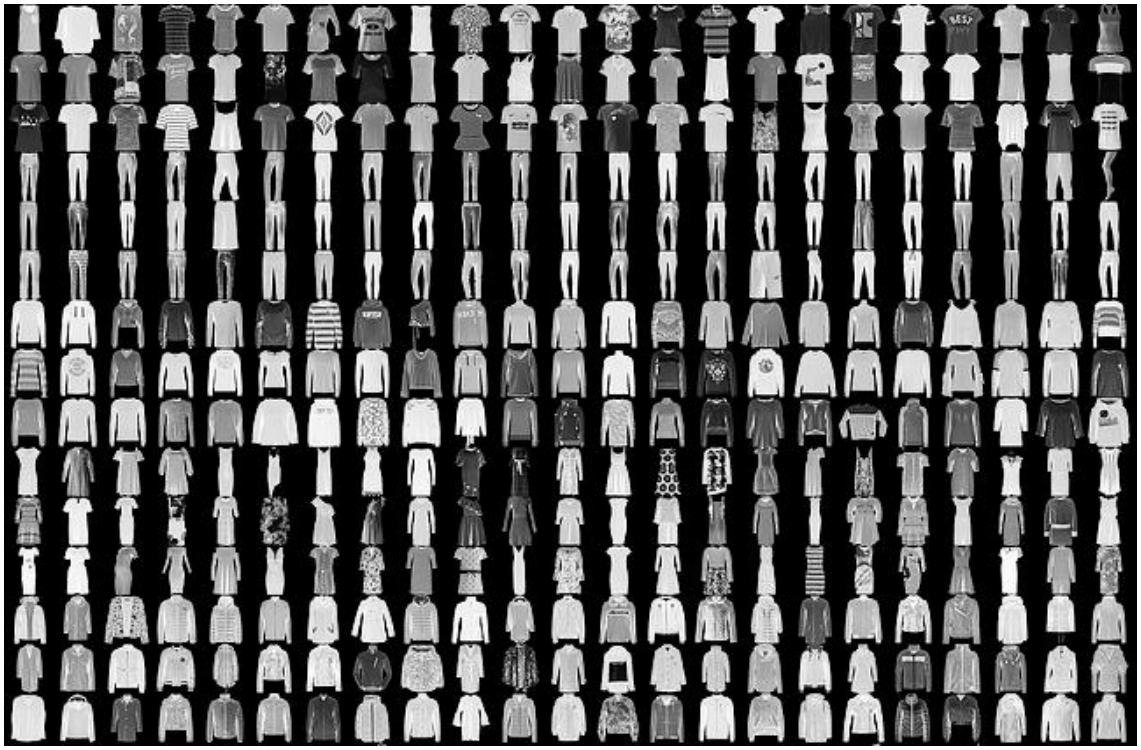

Image 11. A few of the 70,000 images of clothing within the Fashion-MNIST dataset.

The data is split into 10 classes, which consist of T-shirt/top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag and Ankle boot. You can already tell how this could become difficult, since for example a sneaker and an ankle boot share quite a few similarities and overlapping pixels with each other. Compared to the original MNIST, where the most difficult task could be differentiating the number 1 from the number 7, or an 8 from a 6. Thus, the filters must develop more complex features and get a bit more creative with how they tackle this problem, which in return results in an overall smarter network.

We chose Fashion-MNIST for multiple reasons to be our dataset of choice. First, because it is a derivative of the famous and original MNIST which was touched upon in chapter 3.3. Secondly, it is a database of images which naturally goes hand in hand in our goal of training a neural network for image recognition. Thirdly, because the fashion industry is so old fashioned, it is quite refreshing that some new and trendy technological applications breathe fresh air into it.

4.2 Python and the tools that go with it

The programming language used in the project is Python. Python is an open source programming language developed by Python Software Foundation. It is an easy to learn and very powerful language, used very commonly and possibly the most out of any for AI applications. The combination of beginner friendliness and effectiveness is exactly what we are looking for in our case study, and luckily Python is also the real deal when it comes to AI, so starting out and sticking with it is perfect. We are going to write and run our Python code through the Google Colaboratory environment, which will be explained in a bit more in-depth below.

To begin, in Python 3, you have to import the modules you want to use in your code. Modules are long pieces of code that can be loaded without having to write that part of the code all over again. The modules that we will import for image recognition activities are TensorFlow, NumPy and Matplotlib. We will also go over these modules and what they do in a little bit more detail, to get a better grasp of what is actually happening when we start working on our code.

Colaboratory

The Google Colaboratory is a free Jupyter notebook environment that runs completely in the Google Drive cloud.[29] Colaboratory allows you to write code in easy blocks just like a Jupyter notebook and you can execute them individually as well. Since the code is in the cloud, it is naturally also accessible anywhere with access to the internet, and it can also be collaborated on by multiple people at the same time, hence the name. We will of course be flying solo on this one, but the option is there.

The reason we are using Colaboratory is simply because it’s convenient. The notebook is specifically designed for quick experimentation and prototyping which is exactly what

we are looking for. It is also easy to use, since the graphical UI is very user friendly. A nice bonus to add on top of this is that you get a powerful GPU to use and a total of 12 GB of ram for free. This means that the processing power of the system you're using doesn't matter, since all the power comes from Google's side.

NumPy

NumPy is a widely used library for Python that lets us use its arrays and matrices. These arrays will help us with the datasets that will be used in the machine learning progress. Datasets come in many formats, but for image recognition purposes they will of course be a collection of images. These images need to be sorted and analyzed in some way before being able to use them properly and putting them in an array is the perfect way to do that. Python also has arrays, but they are too simple for the needs of the neural network.[30] NumPy has a lot faster computation and is a lot easily manipulated in ways that large datasets require.[31] This makes NumPy a very potent tool for any kind of machine learning scenario.

Matplotlib

Matplotlib is what will allow us to visualize the results of our training from the output. The great thing about the plotting library provided with Matplotlib is that it is very simple to use, but the results are still great. You can generate many kinds of charts with just a few lines of code, which will come in handy when we want to see different probabilities of classes for a training image, or for example training and validation accuracy.

Not only are the charts generated by Matplotlib nice to look at, they also bridge the gap between just looking at the code and the values it generates and actually visually seeing what it is doing. We are working with image recognition after all, seeing the images themselves and what is happening to them is quite important.

TensorFlow

TensorFlow is an open source platform for machine learning. It covers the whole process of machine learning from start to finish and is usable by both beginners and experts. Even though anyone with some Python knowledge and a bit of time can start using TensorFlow, it is still used by some of the biggest companies in the world like Google and

Intel for their most demanding machine learning tasks.[32] This makes TensorFlow absolutely perfect for our purposes.

We will be building our machine learning model with an API called Keras. Keras will allow us to configure and train our convolutional neural networks that we are going to use. Keras takes pride in being easy to use and “being able to go from idea to result with the least possible delay”. The guiding principles of the API are user friendliness, modularity, easy extensibility and the fact that you can use it with Python.

Synergy through the tools: it just works

The great thing about all these tools is that they are designed to go hand-in-hand. Keras is the official frontend of TensorFlow, and the arrays it uses are designed to be used with NumPy. NumPy and Matplotlib work perfectly with TensorFlow and it is shown through the usage of them through their guides as well.[16] This makes it very convenient for anyone to quickly start building and experimenting with neural networks.

We have gone through Python and the modules we will be using with it. It’s time to look at some of the training related terminology.

4.3 Building our network

While we are going to move step by step in our neural network building and explain things along the way, we still assume that the reader has some knowledge of the Python language and programming in general. Aspects related to AI will be explained at a greater depth, but most basic Python steps will be omitted from the text. It should still be possible to follow the process regardless of programming skill due to the concepts explained in the previous chapters.

This chapter will be divided into sections based on the major steps of programming a convolutional neural network, where these steps will be analyzed, and relevant previous chapters referenced.

To begin with, before anything else we will import TensorFlow, NumPy and Matplotlib in our code. Importing basically just means that we will call upon those modules, and now they’re fully usable by us for the rest of the code. The part of Matplotlib we will call upon

specifically is pyplot, which provides a MATLAB-like plotting framework for our data. We will also import Keras from within TensorFlow separately, so we are able to train our models using it. After this we will preprocess the data, configure the network itself with all of its layers, compile it and then begin the training. Before we get to the first part of programming, we'll take a look at the optimization algorithm that will make all the training possible, the gradient descent.

The gradient descent

The gradient descent is what handles the optimization of the parameters in our model. The weights, which measure how important a feature is, are these parameters. During the process of the training of our model, the gradient descent gets adjusted along the way and with each iteration moves slightly more towards the local minimum. In other words, it is trying to minimize our cost function as far as possible.[33] This cost function is the difference between the values our model is suggesting versus the actual value of the item, which means the higher the cost function the more wrong our machine is. To put it simply, the gradient descent is trying to minimize the distance between the model's best guesses and the dataset's correct answers. This is what allows us to train our model and make it learn from its mistakes. We will talk about our choice of optimizing algorithm (optimizer) in a bit more depth later on.

Importing our dataset and preprocessing the data

The first thing after importing our modules is going to be loading our dataset into the program. Luckily for us, Keras comes with a couple built in datasets, and one of them happens to be Fashion-MNIST. This means that we can simply use the keras.datasets command to add our dataset of choice to the program.

```
fashion_mnist = keras.datasets.fashion_mnist
(image_train, label_train), (image_test, label_test) = fash-
ion_mnist.load_data()
```

Listing 1. Loading test data, training data and their labels.

At this point we will also load our training images, their corresponding labels, the test images and their corresponding labels. The training images and labels contain the previously mentioned 60,000 training images which we will train our data with, and the test images and labels the 10,000 images that we can test the model against.

Since the Fashion-MNIST images are in a grayscale format, we don't have to worry about RGB values at all, but we still have to take into account the pixel values of the image's pixels. The pixel value for grayscale images ranges from 0 to 255 and represents how bright that pixel is.[34] The next and final preprocessing step is to divide these values by themselves, so they get scaled to be between 0 and 1.

```
image_train = image_train / 255
image_test = image_test / 255
```

Listing 2. Dividing the pixel values of the dataset's images by themselves

This is done because we want to normalize the pixel values of the images. By doing this, one can get rid of some lighting and shadows that could make recognition harder.[35] We don't really have to worry about this with our data, since it is in greyscale, however scaling the values has some other benefits as well. Having the raw, 0 to 255 unsigned integer values can slow down the training process quite a bit. Since this operation is a good default data preparation, it can be done in cases where one is not sure what kind of preparation to use.[36]

Another step we are going to have to take to make the data fit our model properly is reshaping the input data. Since the model expects the 2D convolutional layers we are going to use to have 4 dimensions, we are going to have to specify this as well.

```
image_train = image_train.reshape(image_train.shape[0],28,28,1)
image_test = image_test.reshape(image_test.shape[0],28,28,1)
input_shape = (28,28,1)
```

Listing 3. Reshaping the input data to fit our model

These four dimensions are: total number of training/test images, depth of the image, height of the image and width of the image. Originally our data had just two dimensions, the height and the width (28 x 28). We also determine the variable input_shape to have the depth, height and width dimensions as well, which will be used when configuring the layers.

We are going to have to do one more thing before we are done with preprocessing, which is giving the classes actual readable labels. Fashion-MNIST doesn't come with label names, so those should be defined also. It's not very useful to us if the classes are just numbers.

```
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'San-
dal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```

Listing 4. Giving the classes proper names

Now we are finally ready for the exciting next step. It's time to start setting up the network itself.

Layers on layers: building the network

Keras comes with two base models for a network: The sequential model and the functional model. The sequential model is a linear stack of layers, which means that the layers flow linearly from one to the next. First comes input, then the data gets passed on to the first hidden layer, then the next one, until we get to the output layer, and finally the output data. The functional API allows for a lot more complex models, for example multi-input and multi-output and shared layers.

We will be using the sequential model for our purposes, since it fits our needs well. We don't need the more complex features of the functional model and chaining together layers is a great way to build an image classification network.

```
model = keras.Sequential([

    keras.layers.Conv2D(32, (3, 3), activation=tf.nn.relu, input_shape=in-
put_shape),
    keras.layers.Conv2D(64, (3, 3), activation=tf.nn.relu),
    keras.layers.MaxPooling2D(pool_size=(2,2)),
    keras.layers.Dropout(0.25),
    keras.layers.Flatten(),
    keras.layers.Dense(128,activation=tf.nn.relu),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(10,activation=tf.nn.softmax)

])
```

Listing 5. Setting up the sequential model and the layers within it

Our input layer will be a 2D convolutional layer with 32 filters and a filter size of 3 x 3 pixels. We will use the Rectified Linear Unit (ReLU) as its activation function. The activation function of a layer determines how that layer's neurons should respond to the inputs they receive. When a neuron receives an input, it has a function inside it that determines if it should be activated, thus passing the data on to the next one. ReLU is the most often used activation function in deep learning even though it is very simple. The function returns 0 if the input is negative, but it activates with any positive input.[37]

The last argument for our input layer is input_shape, where we use the input shape variable we set in the preprocessing phase.

Our second layer is a similar 2D convolutional layer compared to the first one, with the same filter size and the same activation function. The difference apart from the obvious where we don't have to determine the input size is that the amount of filters has doubled from 32 to 64. Like previously explained in chapter 3.4.2, the first hidden layer is responsible for rough details while the rest handle finer features. A rule of thumb when designing a network is to have an increasing amount of filters in successional 2D convolutional layers. Each successive layer can have even four times the amount of filters compared to the previous one, which helps the network learn features in a hierarchical sense.[19]

Now that we are past the feature grabbing layers it's time to move to the pooling layer. The only argument for our MaxPooling2D layer is pool_size, which is determined at 2 x 2. As explained in chapter 3.5.3 this means that the network will be better at recognizing features that might be squished or differently sized due to the object in the image being at an angle. What this actually does to the data is that if we consider our data to be a table of numbers, the 2 x 2 max pool operation will take the highest values, which represent the most important features, and keep them while getting rid of the rest.[38] Effectively a pool_size of 2 x 2 will divide the vertical and horizontal dimensions in half.

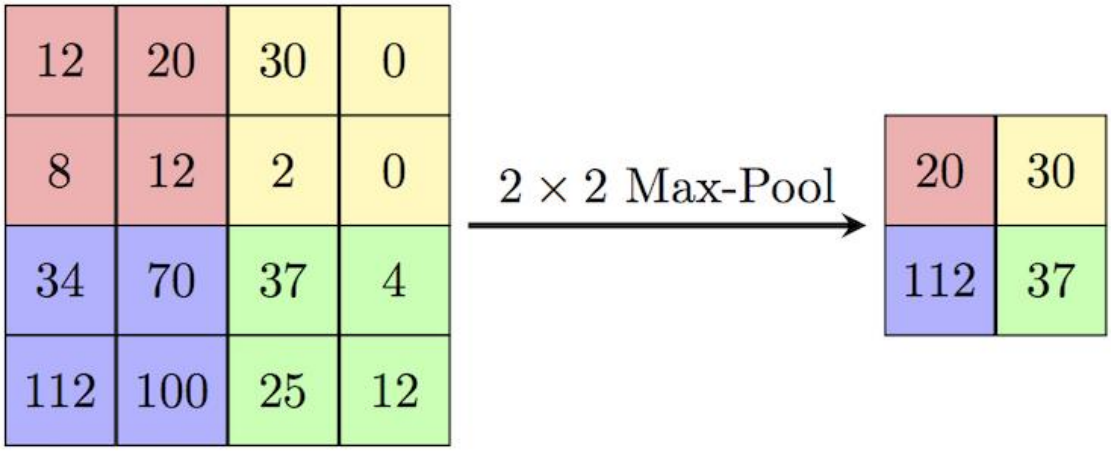

Image 12. A visualization of a 2 x 2 max pool operation.[38]

Next comes our first of two Dropout layers. For this layer our only argument is the dropout rate at 0.25. What the dropout layer, and the dropout operation within it, does is that it drops a fraction of the input units, in the case of our first dropout layer 0.25. The reason

we want to get rid of 25% of our inputs is because it will help us in the long run by preventing overfitting. Overfitting happens when the trained model gets very good results with the training data, but responds poorly with lower accuracy to completely new, foreign data. This happens when the model gets too used to the small nuances of the training data and thus expects to find those with foreign data as well. This can be helped by getting rid of some of the parameters, and the dropout layer does just that.

Since our next convolutional layer is the Dense layer, let's do a small familiar operation first. We will drop in a flatten layer before our densely connected layer, explained in chapter 3.5.2. Now that our data is in the form of a 1-dimensional pancake we can move on to our powerhouse: the densely connected convolutional layer. The arguments for this layer are 128 neurons total, and the activation function for them which is the now familiar ReLU. The trainable parameters of our network will be set in this dense network and will be updated accordingly during backpropagation. We are almost at the output layer, but first we will insert another dropout layer, this time with a value of 0.5 to prevent overfitting even more effectively.

For the output layer we have another dense Keras layer, this time with a neuron count of 10 and a new activation function, the softmax. The neuron count is set at 10 because at the output, we want the amount of values coming out of the network to match the numbers of classes we have, which is 10. What the softmax function does for us is that it outputs the probabilities of the 10 classes as explained in chapter 3.5.1. In short, this output will give us a value for all of the 10 classes, the value being how likely the model thinks the image is any of those classes.

We have now built our network, so the only thing left to do with it is to train it using Fashion-MINST.

4.4 Time to learn: Training the model

Since our network is now complete, we are almost ready to begin the training process. There are just two more small steps before the grand finale and being able to tell just how accurate our model is. We first need to compile the model and then fit it to the training data.

Compiling the model

We have configured our model, however before training it we also need to configure the learning process itself. This is done through the compile method. In the compiling step of our model building journey we are faced with a few unfamiliar terms, so we will go through those and how exactly they interact with our data.

```
model.compile(optimizer=tf.train.AdamOptimizer(), loss='sparse_categori-
cal_crossentropy', metrics=['accuracy'])
```

Listing 6. Configuring the optimizer

The compile step has three arguments, the optimizer, a loss function and a metric. The optimizer is what handles, well, the optimization of the weights in our network. The way this is done is by taking advantage of the previously talked about backpropagation algorithm and its statistics and previous historical values to optimize a function. The goal of the optimization is to make the training process faster.

The second argument is the loss function. The lower the loss, the better the model, thus this function is what the model will try to minimize as much as possible during training. Loss is a sum of all the errors made in the training or the validation sets, comprised of all the different examples of the model predicting what an image may be. A loss of 0 means that the model is perfect, but this would probably mean that the training set was way too easy, and also that we would have some problems with overfitting. For detecting this we are going to use sparse_categorical_crossentropy in our argument. What this specific loss function does is it assumes that each sample image has exactly one class it corresponds to, which is true in our dataset's case. It then compares the label predicted by our model to the real label belonging to the sample and calculates the loss based on this.[39][40]

The third and last argument for the compile step is metrics. This argument, as the name might already suggest, monitors how well or poorly the model does on the training and testing steps. The way it works is similar to the loss function, except it's not used while the training is happening, but after it. For our purposes we are using accuracy, which shows the fraction of the images that get classified correctly by the training process. One might assume from this that loss + accuracy = 1, but unlike accuracy, loss is actually not a percentage. The math behind the number generated by loss is a bit complicated, but generally the lower it is the better for us.

Fitting and training the model

To start training the model, we will use the fit function. Let's go through the arguments once again and then we can get to training and studying the results.

```
model.fit(image_train, label_train, verbose=1, batch_size=128, epochs=20)
```

Listing 7. Fitting the model and determining the batch size and epochs

We use 5 arguments total for the function: the NumPy array target for training data, the target for training label data, verbose, batch size and the number of epochs we want to run. The first two arguments are simply our training data and the labels, and the verbose argument determines how we want to visually view the progress of our training, with the value 1 giving us an animated progress bar.

We set the batch_size argument at 128. What the batch size determines is the number of samples for the training process to go through until the gradient gets updated with new data. If not determined, batch_size defaults at 32. The reason the data needs to be divided into batches is because you can't pass the entire dataset through the network at once. The bigger the size of a batch is, the longer it takes to compute the gradient for it, but we should be fine with a size of 128.

For epochs, we have set this value at 20. An epoch is one complete iteration through the whole provided training data, which means that we will go through the training data a total of 20 times before the training is finished. The reason why we need to go through the data more than once is because with each epoch the training parameters get adjusted into the right direction slightly. With each iteration, the parameters get nudged towards the area what the model believes at that time to be the correct one, based on not only the current but also the previous epochs. Some models might find use in having the epoch counts in the hundreds or thousands, but for our data the accuracy we gain and the loss we get rid of starts to get negligible after 15 or so iterations.

Now we can finally click the big red button and run our code.

```
Epoch 1/20
60000/60000 [==============================] - 154s 3ms/sample - loss: 0.5257
- acc: 0.8150
Epoch 2/20
60000/60000 [==============================] - 153s 3ms/sample - loss: 0.3412
- acc: 0.8774
Epoch 3/20
60000/60000 [==============================] - 152s 3ms/sample - loss: 0.2921
- acc: 0.8939
Epoch 4/20
60000/60000 [==============================] - 152s 3ms/sample - loss: 0.2625
- acc: 0.9038
Epoch 5/20
60000/60000 [==============================] - 152s 3ms/sample - loss: 0.2371
- acc: 0.9129
Epoch 6/20
60000/60000 [==============================] - 153s 3ms/sample - loss: 0.2176
- acc: 0.9194
Epoch 7/20
60000/60000 [==============================] - 153s 3ms/sample - loss: 0.2047
- acc: 0.9234
```

Listing 8. Epochs 1 to 7 running with progress bar, duration, current loss and current accuracy shown

There are a couple noteworthy things in the visual representation of the learning process provided by Keras. On the left you can see the current epoch that is running and the sample it is currently processing. Next to that is the progress bar and on the right of that the estimated time the current epoch will finish in. On the right you can see the current value of loss, and the accuracy value. The loss and accuracy vary significantly between different models, datasets and algorithms used, so there is no generally “good” amount of loss or accuracy to aim for. If the amount of epochs is big, for example a thousand, then there will come a point where the amount of accuracy gained and the reduction of loss plateaus. This makes further iterations nearly useless and means that the amount of epochs was set too high.

4.5 Studying the results

We have finished all 20 of our epochs and now we can start looking at how good exactly our model is at learning through the Fashion-MNIST dataset. We will go through where it succeeded, where it struggled and what the possible improvement points could be for future tinkering. Then we will test the model on the test dataset and see if there is any overfitting or other unexpected results.

Training results: loss and accuracy through the iterations

Let's take a look at our final loss and accuracy, and how it evolves through the epochs. After the first epoch, our loss was 0.5207 and accuracy 0.8150 and after the final 20th epoch they were 0.0874 and 0.9667 respectively. This is a quite impressive score for just 20 epochs, since we managed to identify the piece of clothing correctly ~96.7% of the time. However, by taking overfitting into account, we should know whether it is it a bit too good or did we do just enough.

For the first 5 epochs, we can see that on average our loss went down by 0.0712 and our accuracy went up by 0.0246 per epoch. As we can see the progress is still quite fast in the first few epochs and a lot of ground is covered. If we compare this to our last 5 epochs, where our loss went down on average by 0.00435 and our accuracy went up by 0.00175 per epoch, we can see that while there is some slowdown, there was more than enough gains per epoch to justify the total of 20 that we had.

So far so good, but it's time to test our model against the test dataset, will the amount of loss and accuracy be similar, or did we overdo it?

The test dataset

To test our trained network there is a few things we can do. We can test it against the whole test dataset and print out the loss and accuracy. What we want out of this is that the loss would be as close as possible to the loss of the last epoch of the training process. This would mean that we didn't overfit or underfit the data, since it reacts in the same way to the test data as it did to the training data. We can also test it against a single sample and see the values it gives to the different classes. While doing this we can print pictures using Matplotlib, allowing us to visualize the predictions behind the samples.

```
score = model.evaluate(image_test, label_test, verbose=0)
print('Test loss:', score[0], 'Test accuracy:', score[1])
```

Listing 9. Piece of code for showing the total test loss and accuracy

First of all, we can see the total test loss and accuracy with a pretty simple command. This is done by the evaluate command, with the two significant arguments being the NumPy array of test data and the NumPy array of label data.

```
Test loss: 0.2311066864579916
Test accuracy: 0.9353
```

Listing 10. The test loss and accuracy after training

The end result is a loss of 0.2311 and an accuracy of 0.9353. This means that our model performed worse on the test data compared to the training data. This could be due to overfitting, where our model got too used to the nuances of the training data and failed to perform against new data, or our model could be too unoptimized. It's possible that the model didn't spend enough time with the training data, and thus didn't have enough knowledge to do as well against the test samples. We could have also tried to perform more dropout to further negate the potential overfitting.

Then, let's take a few test samples and see how our model performs on them. We will use a lengthy piece of code from the basic classification tutorial of TensorFlow (Appendix 1) to help with this. We will look at a couple of test images and see if we agree with out model's interpretation of the clothing pieces.

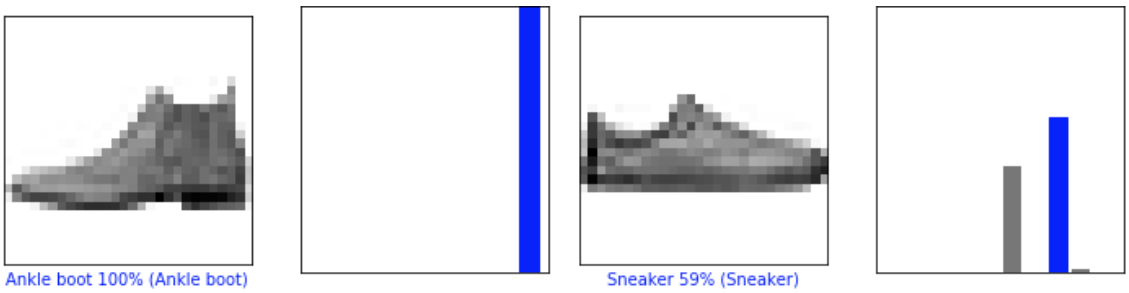

Image 13. The test image with the model's best guess, % of how sure it is about it and other classes' percentages

In Image 12 we are looking at the image of an ankle boot and an image of a sneaker. As we can see our model got both of the guesses correct, but there are some key differences in the two cases. Our model was 100% sure about the ankle boot, but the sneaker only got a score of 59%. As we can see from the bar graph on the right, the other strongly considered choice, in this case sandal, was only a bit behind. This could be due to multiple things, but a good guess would be that the silhouette of a sandal and a low-top sneaker are quite similar, and our model didn't develop strong enough features to differentiate them from each other well enough.

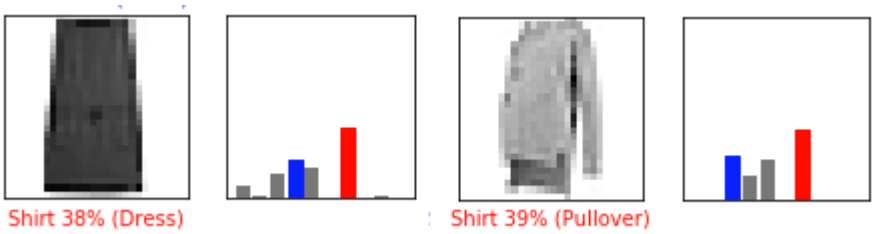

Image 14. A couple of the test images the model struggled with the most

Let's look at some of the test images the model struggled with. In Image 13 are presented two pieces of clothing that the model didn't really seem to be able to figure out. The first image's true label is "Dress", but the model guessed it as a shirt. As we can see from the bar graph, "Shirt" was the second highest guess percentage-wise, but "Dress" was chosen as strongest. Similarly, with the test image on the right, "Shirt" was wrongly chosen over "Pullover". We can see that there were quite a few classes that were up for consideration from the amount of bars present on the graph.

Interestingly these test images are a bit hard to figure out even for a human. The pullover really does look like a shirt, and it could also be a jacket since the low resolution makes it quite hard to figure out exactly what it is. The pullover is also at an angle, which is where better spatial variance could have helped out. The dress on the other hand is just a featureless black box. If there wasn't a context of clothing for these test images, one might think it is a door or a bed. This is where we can notice the difference between Fashion-MNIST and the original MNIST, these "out of the left field" images are what throws the model off easily and forces it to figure out harder problems than the handwritten numbers of MNIST would be able to offer.

So how could we attempt to fix these problems and, perhaps with another round of training, be able to do better? Let's look at some possible solutions and improvements we could implement.

Future improvement ideas

Just like any program or craft, the model could be tinkered with for eternity until it is as close to perfect as we could get it. Since our model was made to show the process of building a convolutional neural network from start to finish, we will not do that. However, we can still theorize how the model could be made better.

The glaring problem with our model currently is the fact that loss is quite a bit higher on the test data compared to our training results. This could be due to overfitting, where as previously mentioned we could increase dropout to further reduce the difference of loss when testing. It could also be because the amount of epochs was not appropriate for the data. More epochs could mean that the model gets more optimized and thus does better when dealing with new data, but it could also even increase the overfitting, thus doing the exact opposite of what we want. Changes could also be made to the layer structure, and even the layers themselves to further increase the effectiveness of our model.

As can be quickly seen, there can be many reasons why a network performs the way it does, and it's not always easy to pinpoint where improvements could be made. With more experience better models could be built with less time, but nothing is ever going to be perfect and there will always be a better option hidden away just out of reach.

5 Conclusion

The goal before beginning the work on this thesis was not only to learn about AI, neural networks, machine learning and programming a model from scratch, but to also be able to write about the subject confidently enough to justify a thesis. A satisfactory outcome was achieved on all of these points, and not only was the door to the world of machine learning opened, during the writing of this thesis the interest deepened.

The theory part was kept simple to understand for anyone regardless of skill level in the subject and was written clear of jargon. All the terms used were explained in appropriate length and information not relevant to the aim of the thesis was attempted to be toned down as much as possible.

The thesis should give a good basis for getting familiar with machine learning and neural networks, and even more specifically convolutional neural networks. The model building and training portion should be a great starting point for anyone interested in trying the process themselves. Since all the major programming steps are explained, not only should building the model be possible, but also understanding why it is built the way explained in the thesis and in general.

The work done has some potential for future developing. A way to continue this thesis to something bigger would be to continue improving the machine learning model created

and use it for inference, and perhaps even turn it into something where it could detect clothes while they are being worn.

The internet is the home of AI progress and is where all of the sources in this thesis were found. A great hobby can start with a simple Google search.

References

1 formal.stanford.edu [Internet], 2006 [cited 2019 May 25] Available from: http://www-formal.stanford.edu/jmc/slides/dartmouth/dartmouth/node1.html

2 TechRepublic [Internet], 2013 [cited 2019 May 25] Available from: https://www.techrepublic.com/article/ibm-watson-the-inside-story-of-how-the-jeopardy-winning-supercomputer-was-born-and-what-it-wants-to-do-next/

3 Forbes [Internet], 2016 [cited 2019 May 25] Available from: https://www.forbes.com/sites/gilpress/2016/12/30/a-very-short-history-of-artificial-intelligence-ai/

4 Verizon Media. Engadget [Internet], 2018 [cited 2019 Mar 25] Available from: https://www.engadget.com/2018/08/06/openai-five-dumpsters-dota-2-veterans/

5 GeeksforGeeks [Internet], 2019 [cited 2019 May 26] Available from: https://www.geeksforgeeks.org/pattern-recognition-introduction/

6 techopedia.com [Internet], 2019 [cited 2019 Feb 16] Available from: https://www.techopedia.com/definition/190/artificial-intelligence-ai

7 Todorović A, Has The Turing Test Been Passed? [Internet], 2015 [cited 2019 Mar 25] Available from: http://isturingtestpassed.github.io/

8 Hurwitz J & Kirsch D. What is machine learning?. Machine Learning for Dummies. 2018. p. 4.

9 The MNIST Database [Internet], 2019 [cited 2019 May 26] Available from: http://yann.lecun.com/exdb/mnist/

10 Hurwitz J & Kirsch D. Iterative learning from data. Machine Learning for Dummies. 2018. p. 5.

11 ScienceDaily [Internet], 2019 [cited 2019 May 26] Available from: https://www.sciencedaily.com/terms/neuron.htm

12 Hurwitz J & Kirsch D. Neural networks and deep learning. Machine Learning for Dummies. 2018. p. 17-18.

13 CS231n Convolutional Neural Networks for Visual Recognition [Internet], 2019 [cited 2019 May 26] Available from: http://cs231n.github.io/convolutional-networks/

14 Saha S, Towards Data Science [Internet], 2018 [cited 2019 Mar 3] Available from: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

15 doug & sapo_cosmico, Stack Exchange [Internet], 2018 [cited 2019 Mar 3] Available from: https://stats.stackexchange.com/questions/181/how-to-choose-the-number-of-hidden-layers-and-nodes-in-a-feedforward-neural-netw

16 Tensorflow [Internet], 2019 [cited 2019 Feb 12] Available from: https://www.tensorflow.org/tutorials/keras/basic_classification

17 SuperDataScience [Internet], 2018 [cited 2019 Mar 5] Available from: https://www.superdatascience.com/blogs/convolutional-neural-networks-cnn-step-max-pooling

18 Deshpande A, A Beginner's Guide To Understanding Convolutional Neural Networks [Internet], 2016 [Cited 2019 Mar 2] Available from: https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/

19 Chatterjee S, Different Kinds of Convolutional Filters [Internet], 2017 [Cited 2019 Mar 16] Available from: https://www.saama.com/blog/different-kinds-convolutional-filters/

20 Wang C-F, Towards Data Science [Internet], 2018 [cited 2019 Mar 16] Available from: https://towardsdatascience.com/a-basic-introduction-to-separable-convolutions-b99ec3102728

21 Shibuya N, Towards Data Science [Internet], 2017 [cited 2019 Mar 17] Available from: https://towardsdatascience.com/up-sampling-with-transposed-convolution-9ae4f2df52d0

22 Odena A, Dumoulin V, Olah C, Deconvolution and Checkerboard Artifacts [Internet], 2016 [cited 2019 Mar 17] Available from: https://distill.pub/2016/deconv-checkerboard/

23 Dao D. Stack Exchange [Internet], 2016 [cited 2019 Mar 17] Available from: https://datascience.stackexchange.com/questions/6107/what-are-deconvolutional-layers

24 Lai A, Using AI to Clean up Your Grainy Photos! [Internet], 2019 [cited 2019 Mar 25] Available from: https://blog.goodaudience.com/using-ai-to-clean-up-your-grainy-photos-fb5c01002a24

25 Dabbura I, Coding Neural Network—Forward Propagation and Backpropagtion[sic] [Internet], 2018 [cited 2019 Mar 20] Available from: https://towardsdatascience.com/coding-neural-network-forward-propagation-and-backpropagtion-ccf8cf369f76

26 Abhigoku10, Topic DL03: Receptive Field in CNN and the Math behind it [Internet], 2018 [cited 2019 Mar 10] Available from: https://medium.com/@abhigoku10/topic-dl03-receptive-field-in-cnn-and-the-math-behind-it-e17565212a20

27 NatureFocused [Internet], 2019 [cited 2019 May 26] Available from: https://www.naturefocused.com/articles/photography-image-processing-kernel.html

28 Zalando Research. Fashion-MNIST Research Project by Kashif Rasul & Han Xiao [Internet], 2019 [cited 2017 Oct 19] Available from: https://research.zalando.com/welcome/mission/research-projects/fashion-mnist/

29 Colaboratory. Welcome to Colaboratory! [Internet], 2019 [cited 2019 Mar 27] Available from: https://colab.research.google.com/notebooks/welcome.ipynb#scrollTo=-Rh3-Vt9Nev9

30 Upadhyay N, Artificial Intelligence series_part 2: NumPy walkthrough [Internet], 2018 [cited 2019 Feb 24] Available from: https://medium.com/datadriveninvestor/artificial-intelligence-series-part-2-numpy-walkthrough-64461f26af4f

31 Vasudev R, Introduction to Numpy -1 : An absolute beginners guide to Machine Learning and Data science. [Internet], 2017 [cited 2019 Feb 24] Available from: https://hackernoon.com/introduction-to-numpy-1-an-absolute-beginners-guide-to-machine-learning-and-data-science-5d87f13f0d51

32 Tensorflow [Internet], 2019 [cited 2019 Mar 29] Available from: https://www.tensorflow.org/about/

33 ML Cheatsheet. Gradient Descent [Internet], 2017 [cited 2019 Mar 30] Available from: https://ml-cheatsheet.readthedocs.io/en/latest/gradient_descent.html

34 Fisher R, Perkins S, Walker A & Wolfart E, Pixel Values [Internet], 2003 [cited 2019 Mar 28] Available from: https://homepages.inf.ed.ac.uk/rbf/HIPR2/value.htm

35 Sinha U, Normalized RGB [Internet], 2019 [cited 2019 Mar 28] Available from: http://aishack.in/tutorials/normalized-rgb/

36 Brownlee J, How to Manually Scale Image Pixel Data for Deep Learning [Internet], 2019 [cited 2019 Mar 28] Available from: https://machinelearningmastery.com/how-to-manually-scale-image-pixel-data-for-deep-learning/

37 Becker D, Rectified Linear Units (ReLU) in Deep Learning [Internet], 2019 [cited 2019 Mar 29] Available from: https://www.kaggle.com/dansbecker/rectified-linear-units-relu-in-deep-learning

38 Computersciencewiki [Internet], 2019 [cited 2019 Mar 29] Available from: https://computersciencewiki.org/index.php/Max-pooling_/_Pooling

39 frenzykryger. Stack Exchange [Internet], 2018 [cited 2019 Mar 29] Available from: https://datascience.stackexchange.com/questions/41921/sparse-categorical-crossentropy-vs-categorical-crossentropy-keras-accuracy

40 Wei L. Multi-hot Sparse Categorical Cross-entropy [Internet], 2018 [cited 2019 Mar 29] Available from: https://cwiki.apache.org/confluence/display/MXNET/Multi-hot+Sparse+Categorical+Cross-entropy

41 Dumoulin V & Visin F, A guide to convolution arithmetic for deep learning [Internet], 2018 [Cited 2019 Mar 3] Available from: https://arxiv.org/abs/1603.07285

Appendices

Appendix 1. Code for matplotlib visualizations

```
def plot_image(i, predictions_array, true_label, img):
  predictions_array, true_label, img = predictions_array[i], true_label[i],
img[i]
  plt.grid(False)
  plt.xticks([])
  plt.yticks([])

  plt.imshow(img, cmap=plt.cm.binary)

  predicted_label = np.argmax(predictions_array)
  if predicted_label == true_label:
    color = 'blue'
  else:
    color = 'red'

  plt.xlabel("{} {:2.0f}% ({})".format(class_names[predicted_label],
                                100*np.max(predictions_array),
                                class_names[true_label]),
                                color=color)

def plot_value_array(i, predictions_array, true_label):
  predictions_array, true_label = predictions_array[i], true_label[i]
  plt.grid(False)
  plt.xticks([])
  plt.yticks([])
  thisplot = plt.bar(range(10), predictions_array, color="#777777")
  plt.ylim([0, 1])
  predicted_label = np.argmax(predictions_array)

  thisplot[predicted_label].set_color('red')
  thisplot[true_label].set_color('blue')
```

```
i = 0
image_test = image_test.reshape(image_test.shape[0],28,28)
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions, label_test, image_test)
plt.subplot(1,2,2)
plot_value_array(i, predictions,  label_test)
plt.show()
```