Bachelor Thesis Project

A COMPARATIVE STUDY OF FFN AND CNN WITHIN IMAGE RECOGNITION

The effects of training and accuracy of different artificial neural network designs

Bachelor Degree Project in Information Technology
Basic level 30 ETCS
Spring term 2019

Linus Lindahl & Magnus Knutsson

Supervisor: Niclas Ståhl
Examiner: Juhee Bae

Abstract

Image recognition and -classification is becoming more important as the need to be able to process large amounts of images is becoming more common. The aim of this thesis is to compare two types of artificial neural networks, *FeedForward Network* and *Convolutional Neural Network*, to see how these compare when performing the task of image recognition.

Six models of each type of neural network was created that differed in terms of width, depth and which activation function they used in order to learn. This enabled the experiment to also see if these parameters had any effect on the rate which a network learn and how the network design affected the validation accuracy of the models. The models were implemented using the API *Keras,* and trained and tested using the dataset CIFAR-10.

The results showed that within the scope of this experiment the CNN models were always preferable as they achieved a statistically higher validation accuracy compared to their FFN counterparts.

Keywords - *Machine learning, Supervised learning, FeedForward Network, Convolutional Neural Network, CIFAR-10, Keras, Activation Function*

Acknowledgement

We would like to thank our supervisor, Nicklas Ståhl, at the University of Skövde for the help and support given during this study. He always had good advice and always had time for a meeting whenever we needed it. Without him the work would not have been possible.

Table of content

1 Introduction

The amount of data that is created and shared is larger than ever and growing, and because of social media a large part of this data is in the form of images. The large quantity of data makes it more prudent than ever to automate the processing of these images as there is no feasible way that they can be manually monitored and classified. A way to handle large amounts of data is to use machines that for example, learn to classify images into different classes. In order for a machine to learn to automate such processes a technology called *machine learning* is utilized.

Machine learning is a very large, and at the time of writing, rapidly growing field. There are many ways and techniques to enable a machine to 'learn' but for the scope of this thesis the focus will be image classification using a form of *supervised learning*, namely *artificial neural networks*. In this thesis two different variations of neural networks will be used, basic *FeedForward Networks* (FFN) and *Convolutional Neural Networks* (CNN).

Six models will be created of each of these types of neural networks. The models will differ in terms of their width, depth and which *activation function* they use in their learning process. The aim of the thesis is to answer the following research questions:

1. To what extent does the width and depth of a FFN affect the validation accuracy of the models used in the experiment?
2. To what extent does the activation functions Sigmoid, TanH, ReLU and Leaky ReLU affect the validation accuracy of a FFN?
3. To what extent does the width and depth of a CNN affect the validation accuracy of the models used in the experiment?
4. To what extent does the activation functions Sigmoid, TanH, ReLU and Leaky ReLU affect the validation accuracy of a CNN?
5. Which of the activation functions is able to learn with the fewest epochs on the models used in the experiment?
6. Which architectural models used in this study is able to achieve the highest validation accuracy on the trained models FFN or CNN?

To answer these questions an experiment was conducted. The FFN and CNN models were implemented in the programming language *python* using the *Keras* API and trained using the dataset *CIFAR-10*. The training and testing was performed using the same hardware to keep the results as comparable as possible. To maximize the usage of the dataset *k-fold cross-validation* was used. To keep the tests fair and replicable, all random values were seeded the same, to maintain a degree of controllability.

Paired t-tests were conducted to compare the results and determine whether any potential differences between the models were large enough to be statistically significant.

2 Background

This chapter covers the theoretical background and introduces the technical terms and aspects brought up in the thesis. Machine learning in general will be described, as well as specific techniques and related aspects such as different activation functions.

2.1 Intro to machine learning

Russel & Norwig (2010) describes learning as “An agent is learning if it improves its performance of future tasks after making observations about the world.”
Domingos (2012) describes the term machine learning as the process in which a system studies data and learns to discern and categorize patterns. This is meant in the general sense as it is unlikely that the data that has been studied will reappear when the system or program is later tested or applied in a real world setting.

There are different uses for machine learning, one example is *classifiers*. Classifiers learn to discern different types of input into predetermined output classes. In other words if a classifier is built to separate images into two categories, images containing cats and images not containing cats, then the classifier needs to learn which patterns identifies cats in images in order to make predictions of whether images contain cats or not. It is important to note that the images used to train the classifier will not be used to evaluate its performance, because the wanted behavior is that the classifier should be able to correctly classify images it have not seen before. If the performance were evaluated on the training images there is a risk that the results would appear to be very accurate, but when the classifier would try to evaluate new unseen images, the classifier could potentially perform poorly since it knows only exactly the images from the training images.

2.2 Supervised Learning

Supervised learning is when a machine, program, neural network for instance is trained to recognize certain elements or artifacts based on a labeled *training set*. A labeled dataset is a collection of data that is labeled, for example images of cats would carry the label ‘cat’ and images of dogs would have the label ‘dog’. This is an example of a task that can be done through the use of supervised learning called image recognition (LeCun et al., 2015).
According to Russel & Norwig (2010) an agent monitors input in form of a labeled training set and by comparing these labels to the agent’s predictions, the program can be tuned in order to make more accurate predictions.

An example of this could be an artificial neural network with the desired functionality to identify images of cars and images of cats. After this neural network is implemented it's not very likely to have a high success rate in identifying or separating cars and cats in any image it is shown, as it is still untrained. The neural network will take an image as input, for example in the form of values for each pixel in the image, and pass these values through the network. At the end of said network there are two possible outputs, cat or car, and the desire is for the correct output for said image to have the highest output value. On an untrained network the output values can be completely random and to mitigate this the network must be trained. The training is done by assembling a dataset of labeled images containing the desired outputs. These images are then used to train the network as a ground truth for what a car is and what a cat is.

Russel & Norwig (2010) also describes other forms of learning called unsupervised learning and reinforcement learning. In reinforcement learning the behavior of the agent results in either positive or negative feedback. Based on the desired behavior of the agent, its actions are either rewarded or punished. This will have the effect of the agent repeating the behavior that nets rewards and avoid behavior that results in negative feedback. In unsupervised learning the agent receives no feedback, but is rather left to its own devices to find patterns or clusters that might appear in the data.

2.2.1 Training a machine using supervised learning

In order for a machine to learn it needs to be trained on the type of data it will be analyzing later. This is done by feeding a large amount of training examples through the system. When training a new machine learning system a sound strategy is to set some of the data aside in order to be utilized later as a *validation set*, meaning that the program is not trained or tuned using this data. This data is rather used for final validation as a measure of accuracy of the programs predictions. In case the amount of available data is relatively small in size it might be unfeasible or undesired to set aside a part of the data to use for validation, because it reduces the size of the training set. To mitigate this problem the entire data set can be split into multiple smaller parts, one or more parts are set aside to form a validation sets and the model is trained of the remaining subsets. This process can be repeated where the subset that is set aside changes. This sort of segmentation and usage of the training data is called *cross-validation* (Domingos 2012).

2.2.2 K-fold cross-validation

K-fold cross-validation is used to make more use out of a dataset, where all parts of the dataset serves two purposes, both as training set and validation set (Russell and Norvig, 2010). In k-fold cross-validation the dataset is divided into *k* equally sized parts. During training one of the dataset parts will be set aside to be used as a validation set, and the network will train on the remainder of the *k* dataset parts. This is done *k* times, once for every dataset part. Domingos (2012) further details that the model, or classifier, needs to be restarted after each configuration. The average accuracy from the *k* iterations will be used as the accuracy for the network. Figure 2.1 below illustrates 5 fold cross-validation.

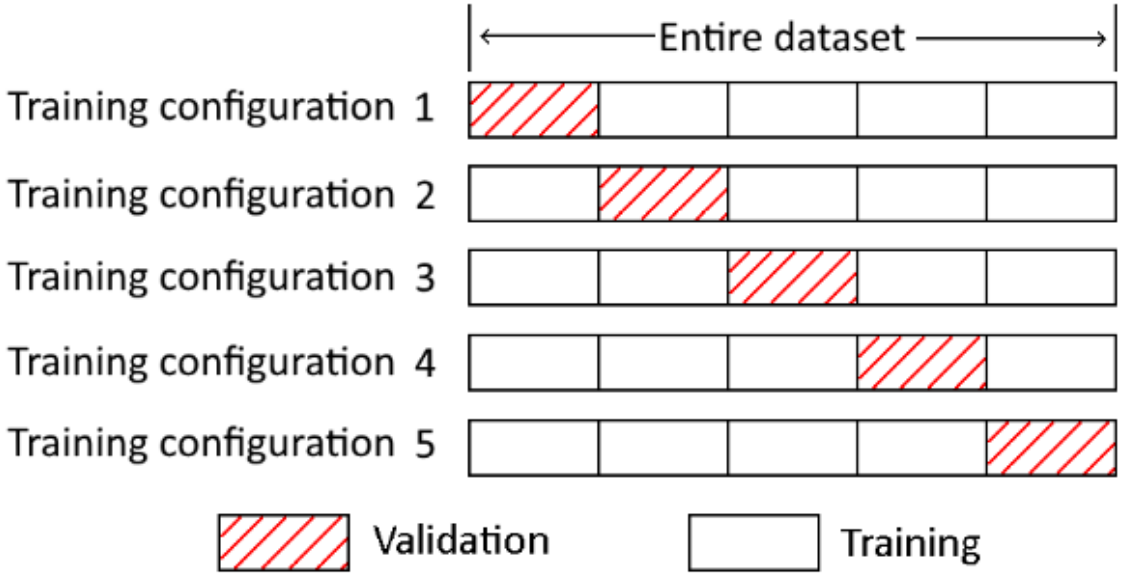


Figure 2.1 Illustration of 5 fold cross-validation

2.3 FeedForward Network

Artificial neural networks are inspired by biological neural networks. Kong et al. (2017) describes an artificial neural network as a network that consists of neurons that are placed in layers. All of these neurons each have a bias value and an activation function. Demuth & Beale (2001) furthers explains that whenever two or more neurons receive input from the same sources this constitutes a layer. The number of neurons in a layer or the number of layers in an artificial neural network can vary. Chauvin and Rumelhart (2009) describes the most basic form of an artificial neural network as a *FeedForward Network* (*FFN)*. Whilst there technically exist other types of neural networks that could also be described as FFN´s, for the sake of this paper the term FFN will refer to a fully connected FeedForward Neural Network. Figure 2.2 shows a FFN with two hidden layers. Layers in a model that are not either the input or output layer are called a hidden layers.

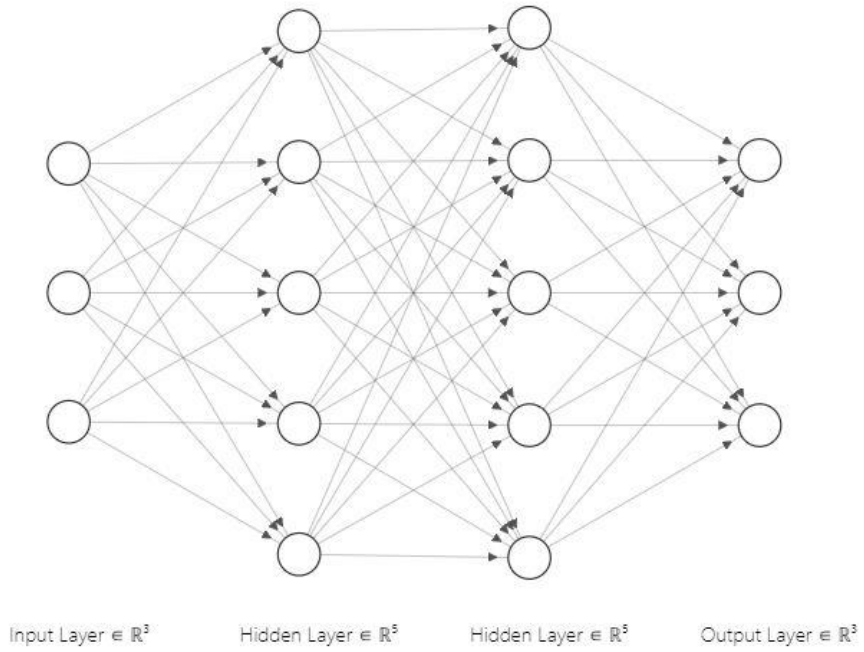


Figure 2.2 Illustration of a FFN with an input layer, two hidden layers and an output layer. The arrows indicate data flowing from the left of the image to the right, i.e. from the previous layer to a neuron in the next layer.

Russel & Norwig (2010) makes the comparison between the biological neural network of the human brain and how the recreation of such a network can be attempted using computer science and logic, the result being an artificial neural network. Russel & Norwig (2010) describes it as "A neural network is just a collection of units connected together; the properties of the network are determined by its topology and the properties of the 'neurons'".

Demuth & Beale (2001) describes that FFN's have areas of application such as pattern recognition and -identification for instance. By adjusting the values between neurons and layers, a network can be trained to perform different tasks. A network can be trained by input that is fed through the network in order to classify the input as anything in a target range of outputs. For example if the input was an image, the input would be each individual pixel. Training can be done in steps where the training data is divided into *batches* and after each batch the weights and biases of the neurons are updated.

LeCun et al. (2015) explains that before the FFN is trained its edges have *weights* and the *neurons* have a *bias* that are random values which will very likely lead to incorrect outputs. By feeding labeled images from the training set through the network the output of the network can be compared to the desired output and by how much the neural networks prediction was off. After a batch the network makes some calculations of how close the predictions of the neural networks were to the correct and desired output. Based on these calculations, the weights and biases of the neurons are altered by using a technique called *backpropagation,* which will be explained in more detail in a later section. This is done as an attempt to minimize the difference between the predictions and their corresponding label, or rather to create a network that makes good generalizations. An example of training a neural net could be a dataset of 100 training images where backpropagation is performed after each batch of 20 images and running through all 100 images constitutes one *epoch*. Note however that a training set of 100 images would be considered very small.

2.3.1 Neuron

The neurons in a neural network are interconnected with each other and when data is passed forward in the network a neuron will receive an input from neurons in the previous layer. Along with this input the neuron will also receive a value that is the weight of the edge that connects the two neurons. The *activation function* use the sum of the products of the inputs and weights in addition to the neurons own bias to compute a new output that is forwarded to the neurons in the next layer (Kong et al. 2017). Different activation functions will be described later in the report.

2.3.2 Activation function

Neurons in a neural network can have activation functions associated with them. The purpose of the neuron is to take multiple inputs and produce a single output. The way the neuron does this is by taking the input parameters and multiply each parameter with its specific weight. The purpose of the weight is to determine the influence of the given input and how it affects the output. These products are then summed together and the bias of the neuron is added. Then the total value is fed through an activation function (Russell and Norvig, 2010).

$$f(\sum_{i} input_i weight_i + bias)$$

The purpose of the activation function is to allow the neural network to create non-linear patterns in order to group data points together. If the neural network had only linear functions, then the entire network could be rewritten as a single linear function.
For instance:

- if $f(x) = 2x$
- and $g(x) = 5x$,
- then $f(g(x)) = 10x$

This would have the same effect as a single function $h(x) = 10x$. In other words if the neural network had only linear functions, then the hidden layers in the model would have no effect, because all the hidden layers could be replaced by a single function from the input layer to the output layer. In Figure 2.3 below, cats are marked by x and dogs are marked as o, and it illustrates the difference between the decision boundary of linear neural networks and non-linear neural networks. The decision boundary decides if the data point is a cat or a dog.

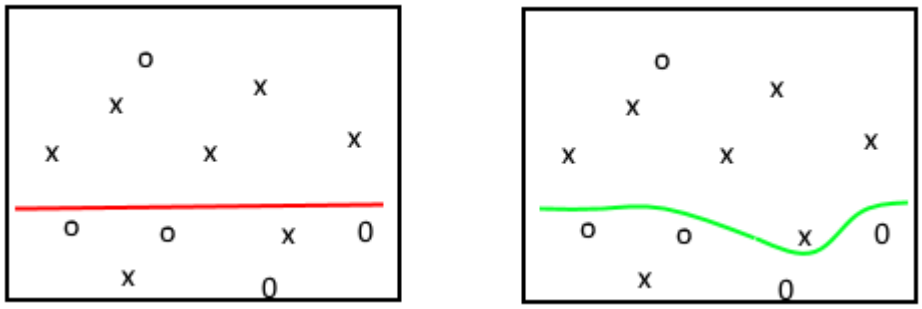


Figure 2.3 The left image shows a linear decision boundary and the right image illustrates a non-linear decision boundary, separating cats from dogs, cats are marked as x, and dogs are marked as o.

Sigmoid

The Sigmoid function can take input from negative infinity to positive infinity and compress this into a spectrum ranging from 0 to 1 (Demuth, H. & Beale, M. 2001). The formula for Sigmoid is $f(x) = \frac{1}{1+e^{-x}}$ and Figure 2.4 below shows a representation of the Sigmoid function.

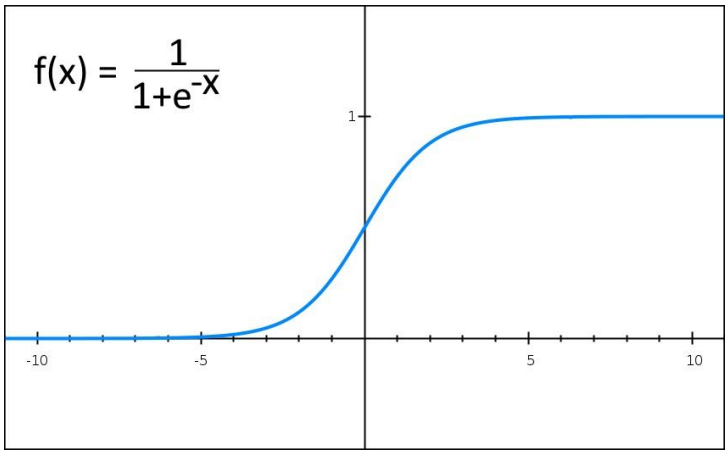


Figure 2.4 Sigmoid activation function

TanH

TanH refers to the *hyperbolic tangent* and this function is similar to the Sigmoid function, however the range for TanH is from −1 to 1, unlike the Sigmoid function that has an output range from 0 to 1 ("Neural Networks", 2019). In Figure 2.5 below there is a visual representation of the TanH function.

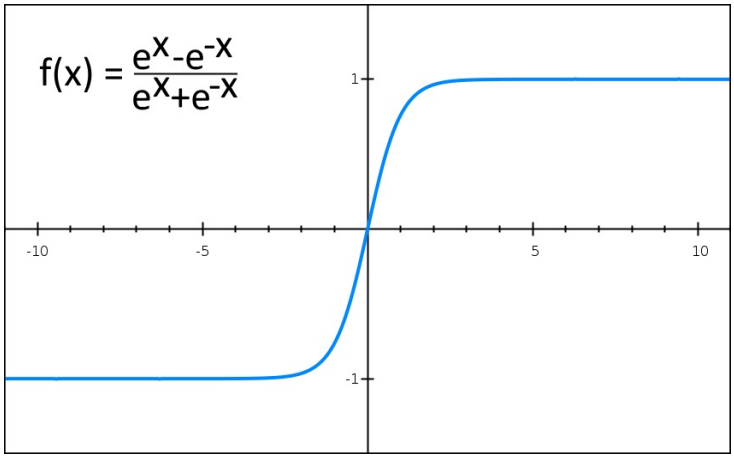


Figure 2.5 TanH activation function

ReLU

ReLU is an abbreviation of *Rectified Linear Unit (*Krizhevsky, A., Sutskever, I., & Hinton, G. E., 2012*)*. If ReLU receives a positive input the function outputs the same value, but if the input is negative or zero, it outputs zero instead. The formula for ReLU is thereby $f(x) = max(0, x)$. ReLU is also said to train faster than Sigmoid and TanH because ReLU does not suffer from saturation, this will be discussed further in chapter 2.3.6 Vanishing gradient. Figure 2.6 below show a representation of ReLU.

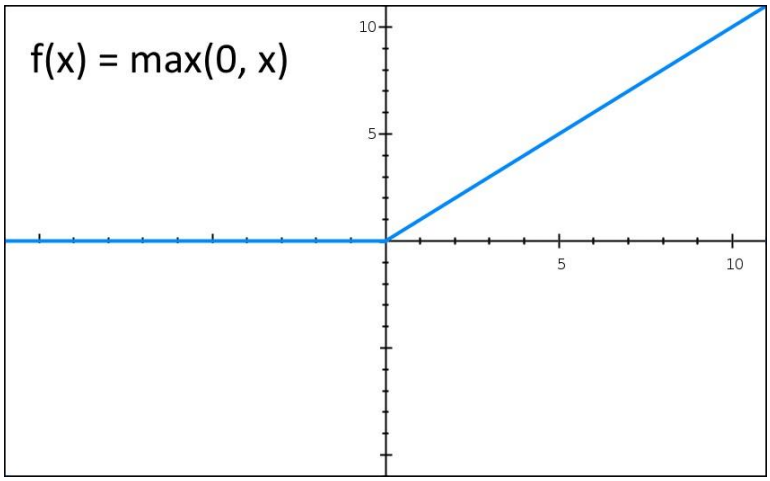


Figure 2.6 ReLU activation function

Leaky ReLU

Leaky Rectified Linear Unit (Leaky ReLU) is very similar to ReLU (Maas, A. Hannun, A and A Ng, 2013). The difference is that Leaky ReLU does not have a gradient of zero for negative values, but instead use a small gradient for negative values.

The reason for the small gradient is to prevent neurons from potentially dying during training. This can occur if, during training, the weights to a neuron is changed in a way that it will not activate again, which in turn will make the neuron dead, since the only time the weights are updated are if they gave an output. Figure 2.7 below illustrates Leaky ReLU.

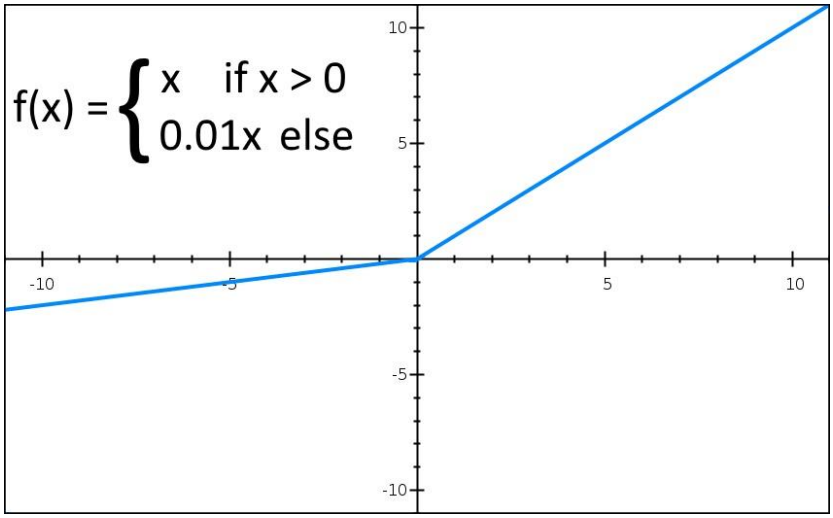


Figure 2.7 Leaky ReLU activation function (for the purpose of illustration the image uses 0.1x as a negative gradient)

2.3.3 Loss function

When using supervised learning in conjunction with neural networks, data is fed through the network and a prediction about the data is made and turned into an output. Kong and Takatuska (2017) writes that in order for a network to learn a loss function is utilized. This loss function calculates the difference between the predicted output and the correct output. By incrementally changing the weights and biases after a batch, the hope is to reach a global minima of the loss function, where the network to the best of its abilities is able to generalize the data and make correct predictions.

2.3.4 Backpropagation

In order for machines to learn, they use backpropagation, where the *gradient* of the loss function with regard to the output is used to calculate the gradient for the input parameters (Lecun et al. 1998).

It was previously believed that the existence of local minima in the gradient would affect the performance of learning for the neural network when using gradient based learning. But in practice neural networks does not seem to suffer from getting stuck in local minima for the gradients (Lecun et al. 1998).

Stochastic gradient descent is one example of gradient learning that can be used to update the parameters in machine learning. This is done by doing small iterative updates of the parameters with the help of the loss function and backpropagation.

Backpropagation is used to determine the effect of changing the inputs to a function with regards to the functions output (Cs231n.github.io, 2019). With other word when backpropagating the goal is to determine the gradient of the inputs to the function. To calculate the gradient of the input parameters to the function the *chain rule* can be used. The chain rule determines how each individual step backwards through the function is derived relative to the output value. These derivatives are the gradients for each step.

The *chain rule* is an important concept to grasp in order to understand backpropagation. Before introducing the chain rule the concept of composite expressions need to be explained. Herman and Strang (2016) explains that a composite expression is when one function is passed as an argument to another function. Strang (n.d.) gives an example with two functions:

- $f(x) = x^4$
- $g(x) = x^3$
- $z = (f\ o\ g)(x) = f(g(x))$

It is important to note that $f(g(x)) \neq f(x)g(x)$ because:

- $f(x)g(x) = x^7$
- $f(g(x)) = x^{12}$

The expression $z = f(g(x))$ can be rewritten into two expressions:

- $y = g(x) = x^3$ and $z = f(y) = y^4$

Thereby the expression can be rewritten as the expression:

- $z = (x^3)^4 = x^3 * x^3 * x^3 * x^3 = x^{12}$

Herman and Strang (2016) suggest usage of the chain rule in order to derive a composite expression. The chain rule states that in order to find the derivative for the composite expression, derive the outer expression while using the inner expression and multiply it by the derivative for the inner expression. The formula for the chain rule looks like the following:

$$h(x) = (f o\ g)(x) = f(g(x))$$
$$h'(x) = f'(g(x))g'(x)$$

2.3.5 Backpropagation example

Andrej Karpathy at Stanford University explains *backpropagation* in the form of logical gates, where each gate is a mathematical calculation (Cs231n.github.io, 2019). Karpathy present a simple Sigmoid activation function with two inputs values, two associated weights, one for each input, and a bias. An illustration can be seen in Figure 2.8 below.

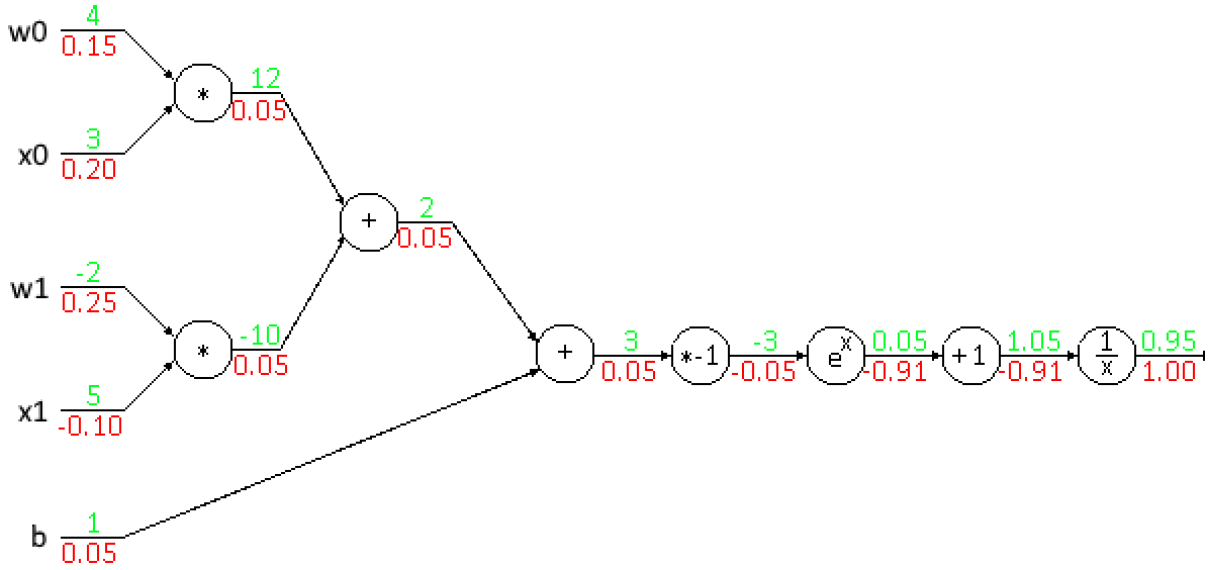


Figure 2.8 Sigmoid activation function. The input to this activation function is x0 and x1, the weight associated with x0 is w0 and the weight associated with x1 is w1, b represents the bias. The green numbers are the input and the red numbers are the derivative for the gate

To get a better understanding of how backpropagation works, a short forward and backward pass through the example data will be explained. The process begins with the forward pass through the figure. Only the green numbers will be relevant in the forward pass.

- Input $x0$ will be multiplied by $w0$ which is the weight of $x0$ $(3 * 4 = 12)$
- input $x1$ will be multiplied by $w1$ which is the weight of $x1$ $(5 * -2 = -10)$
- These two products are then added together $(12 + (-10) = 2)$
- The bias b is then added to the previous sum $(2 + 1 = 3)$
- This sum is then passed through the Sigmoid activation function

The first thing that is done in the Sigmoid function is that the exponential expression needs to be calculated so the value of the exponent needs to be inverted by multiplying the previous value with negative one.

- $(3 * (-1) = -3)$

Next the exponential expression is calculated ($e^{-3} \approx 0.05$) then one is added to calculate the denominator of the Sigmoid expression $(0{,}05 + 1 = 1.05)$. The next steps is the division of the Sigmoid function $(1/1{,}05 \approx 0.95)$. Now the forward pass is done and the graph produces approximately the value 0.95.

With the forward pass done the process can start with the backward pass to calculate the gradients of the input parameters. Derivative rules can be found in Appendix A - Derivation rules. The gradient of the output will come from the next layer in the model and the final layer returns a value from the loss function, but for simplicity the output used in this example will be 1.00. The calculations will be illustrated from Figure 2.9 to Figure 2.17. The derivative to the expression $\frac{1}{x}$ is used, and with the help of the chain rule, the derivative for this expression can be calculated by multiplying it with the derivative of the previous step backwards through the graph $((\frac{-1}{1.05^2})(1.00) \approx -0.91)$.

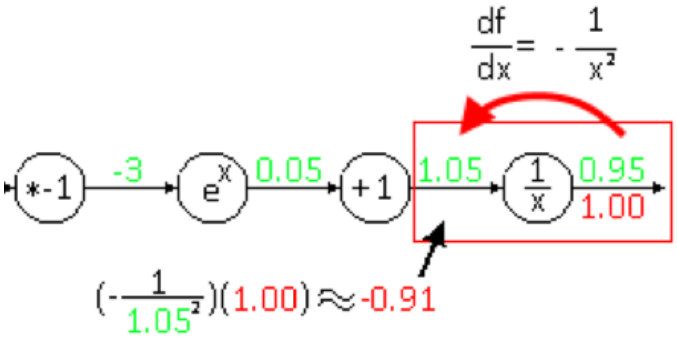


Figure 2.9 Backpropagation part 1

The process continues to work its way backwards to the +1 expression of the activation function and derive this expression. The usage of the chain rule continues, giving:

- $((1.00)(-0.91) = -0.91)$

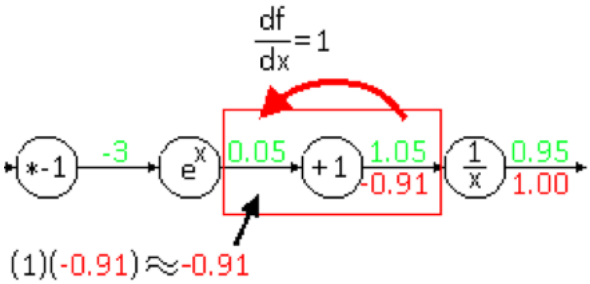


Figure 2.10 Backpropagation part 2

The derivative for e^x is used and multiplied by the previously calculated derivative:

- $((e^{-3})(-0.91) \approx -0.05)$

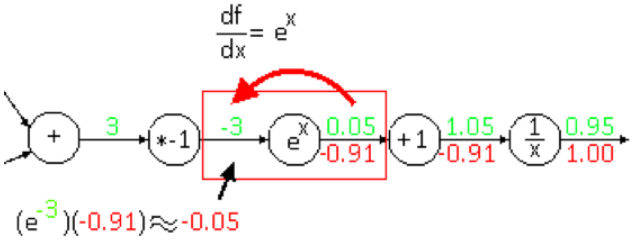


Figure 2.11 Backpropagation part 3

Next the derivative is used for the expression *-1 and continue with the chain rule:

- $((-1)(-0.05) = 0.05)$

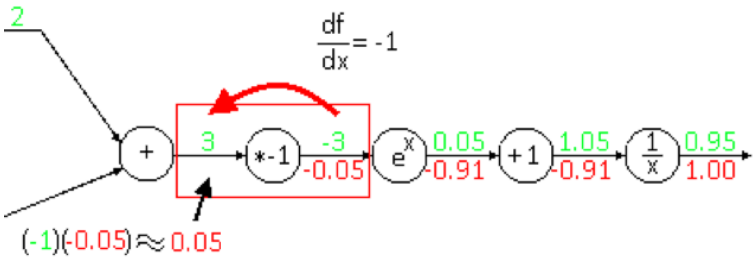


Figure 2.12 Backpropagation part 4

Now the Sigmoid function have been derived and the backpropagation continues to explore the gradients of the different inputs to the function, starting with the bias *b*. The gradient does not change during an addition operation so the gradient for *b* is:

- $((1)(0{,}05) = 0.05)$

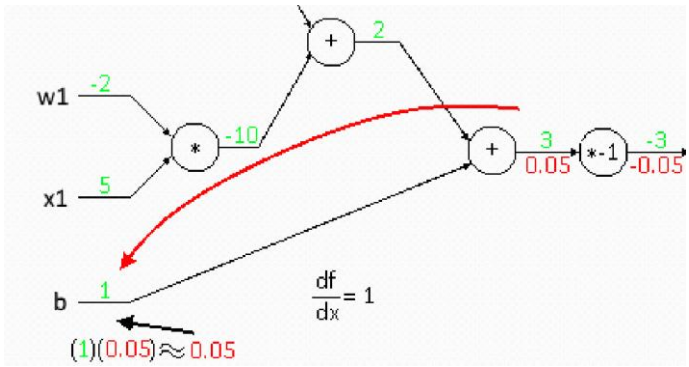


Figure 2.13 Backpropagation part 5

The same gradient also flows backwards to towards the other input variables.

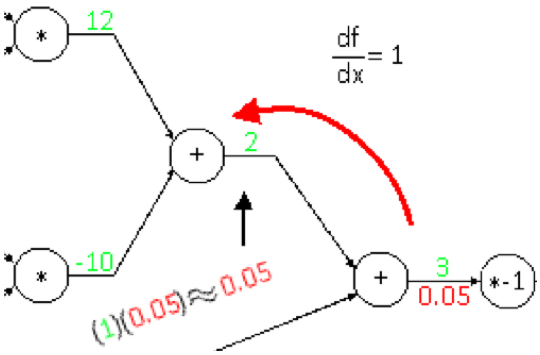


Figure 2.14 Backpropagation part 6

There is another addition gate so the gradient does not change and gets passed along to both inputs to the gate $((1)(0.05) = 0.05)$.

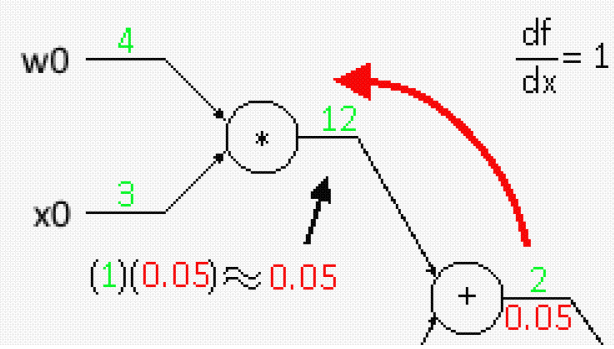


Figure 2.15 Backpropagation part 7

The next step is the multiplication gate between $x0$ and $w0$ and since this is a multiplication gate the gradient will depend on the inputs counterpart to the gate, so the gradient for $w0$ is $((3)(0.05) = 0.15)$ and the gradient for $x0$ is $((4)(0.05) = 0.20)$.

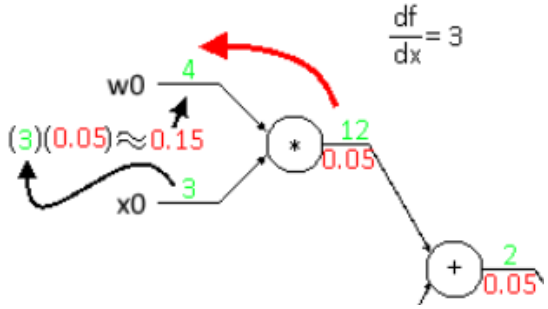


Figure 2.16 Backpropagation part 8

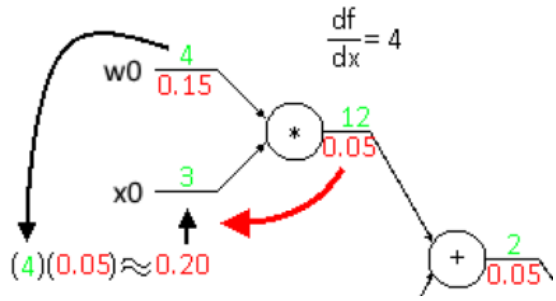


Figure 2.17 Backpropagation part 9

The gradients for the gate between $x1$ and $w1$ can also be calculated on the same principle, the gradient for $w1$ is $((5)(0.05) = 0.25)$ and the gradient for $x1$ is $((-2)(0.05) = -0.10)$.

2.3.6 Vanishing gradient

The vanishing gradient problem is when the gradient or in other words the derivative of the neurons decay from layer to layer during *backpropagation*. The problem is when small numbers close to 0 is multiplied with another small number between 0 and 1 the result will be an even closer to 0. This would mean that if the gradient for a neuron is close to 0 during backpropagation, this neuron will send an even smaller gradient towards the neurons in the previous layer in the model. This indicate that there will be regions where the neuron is saturated and very inefficient at learning and this influence other neurons making them slower at learning as well. The activation functions Sigmoid and TanH both have this problem. Figure 2.18 below gives a representation of where Sigmoid and TanH exhibits saturated behavior.

ReLU and Leaky ReLU were introduced to solve this problem and does not exhibit this behavior because these activation functions are not forced to be within a certain range, so the gradient will not decay for larger positive and negative numbers like Sigmoid and TanH(Kong, S. & Takasuka, M. 2017).

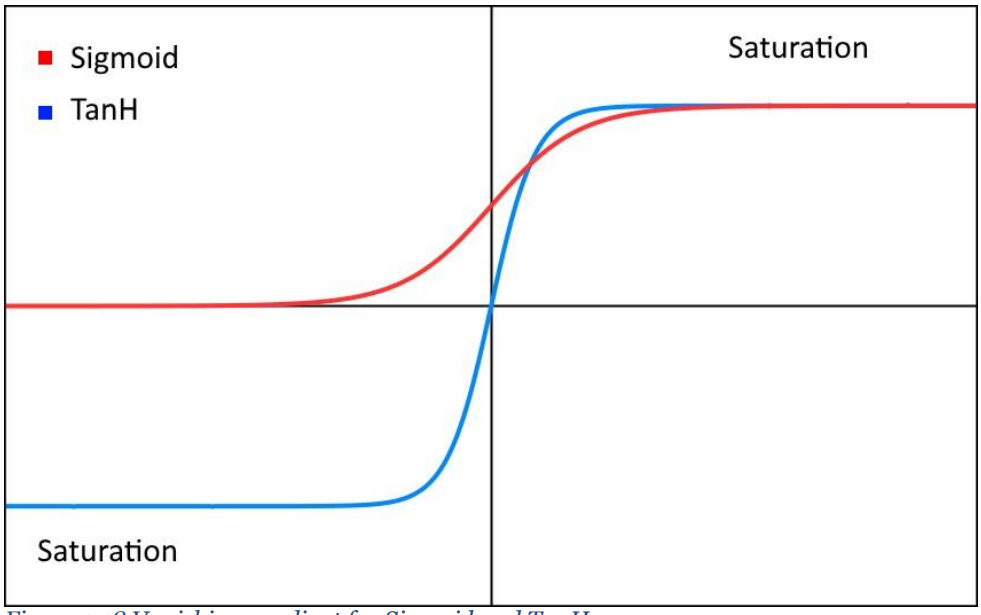


Figure 2.18 Vanishing gradient for Sigmoid and TanH

2.3.7 Overfitting

When training a network it is important to have a sufficient amount of data, or there is a risk of *overfitting* the network. Chauvin and Rumelhart (2009) describes that if there is insufficient data to train on the network might appear to be very successful in its learning, but in reality the network is only starting to memorize the data. For this reason it's very important to validate the accuracy of a network using a validation set of data that the network has not previously seen. For example if a network is trained using k-fold cross validation and the size of the training set is too small, then over the course of the training the network might begin to recognize the individual images rather than, for example, the patterns that the network is supposed to detect and generalize. Chauvin and Rumelhart (2009) summarizes it as "It is possible for a sufficiently large network to merely 'memorize' the training data. We say that a network has truly 'learned' the function when it performs well on unseen cases".

2.4 Convolutional Neural Network

A *Convolutional Neural Network (CNN*) is a type of artificial neural network architecture used for processing images and the strength of this model is that it abstracts the image in different layers of abstraction. The *CNN* normally alternate between *convolution filters* and *pooling layers*. The convolution filters work in a similar fashion as the human brain is believed to function, where the convolution filter looks at small regions of the input and abstracts these regions as straight lines or circles for instance. The convolution filters further down in the architecture detect less abstract features and they can also detect where the feature is located within the image. The convolution filters are trained on a set of training data and they learn through backpropagation. The CNN architecture usually produces class labels or probabilities as output (Greenspan, van Ginneken and Summers, 2016). The pooling layers reduce the size of the image representation by looking in small regions of the image and the pooling layer turns each small region into a single value representing the presence of the given feature. Figure 2.19 below illustrates a CNN.

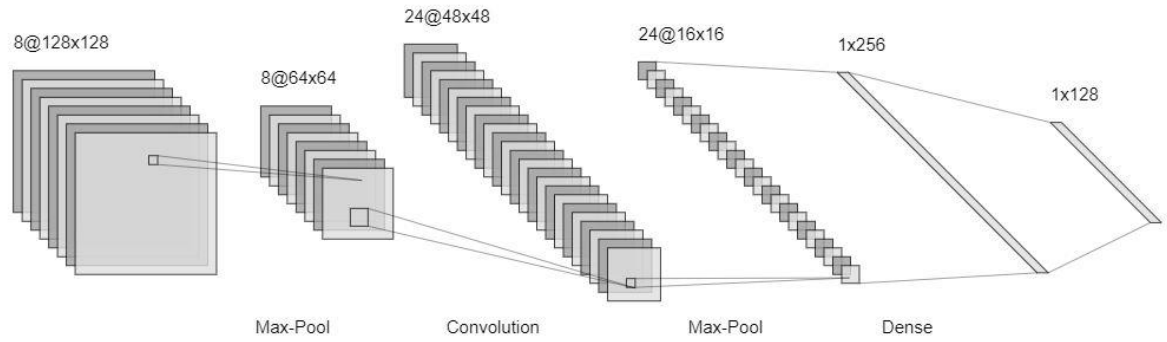


Figure 2.19 Illustration of a Convolutional Neural Network

Lecun et al. (1998) explains that a CNN abstracts the image in multiple steps, the early convolutional layers might extract simple patterns like angled edges and corners, while deeper layers bind these features together into more complex patterns. They also explain that a convolutional layer are divided into multiple planes. Each plane has a receptive field which is a small rectangular region in the input to that layer, the weights for this receptive field are then reused for all other location of the input. In other words a single plane reuses the same weights for different locations of the input, so it can detect the same pattern regardless of where it appears in the input. The results from the regions in a single plane are stored in a single feature map and the convolutional layer will have the same number of feature maps as it has planes. The different feature maps detect different patterns and have individual weights associated with them.

Lecun et al. (1998) further explains that CNN´s do not only use convolutional layers, but they also use subsampling layers. The subsampling layers are used after convolutional layers to reduce the resolution of the image and also reduce the spatial distance between relevant features. In other words, subsampling layers take input from feature maps and each feature map is subsampled into a new feature map of smaller size. This will produce the same number of feature maps as the amount of feature maps in the previous layer. The subsampling divides the input feature map into many non-overlapping areas in the input feature map. These result from these areas are then stored in the output feature map. It is worth noting that the position in the input feature map is store relative to the other non-overlapping areas in the feature map.

2.5 Related Work

In the paper *Gradient Based Learning Applied to Document Recognition* (1998) Lecun et al. benchmarked different neural networks and architectures on the MNIST dataset. The authors chose however to measure the results in error rate, but for the sake of simplicity their results will be described in terms of validation accuracy here. Validation accuracy is a measurement in percentage of the correct predictions among all predictions on the validation set. Lecun et al. (1998) compared different layer structures of FFN's and found that deeper and wider models resulted in a higher validation accuracy. When testing the different network structures there were some test-configurations where the dataset was manipulated and some where the dataset were not manipulated. The choice was made to focus on the results were the structure of the network was tested rather than the manipulation of the dataset, as it seemed more relevant for this thesis project. While the focus is not on the dataset, this is still a point of interest since CIFAR-10 which is a full color dataset will be used in this study. MNIST and CIFAR-10 both have ten possible output classes but CIFAR-10 has more intricate classes such as frog and deer, compared to handwritten digits. The FFN where the dataset was not manipulated that achieved the lowest validation loss was chosen as the inspiration for the FFN base case used in this study. This was done to see if even further variation in width and depth would yield even better results. The choice of activation function was also of interest since Lecun et al. (1998) used the activation function TanH which could potentially suffer from the vanishing gradient problem. An activation function that did not suffer from the vanishing gradient problem introduced by Nair & Hinton (2010) was the activation function called *Rectified Linear Unit* or ReLU. This was later improved by Maas et al. (2013) with the creation of Leaky ReLU. In this study the activation functions ReLU and Leaky ReLU will also be tested and these are not affected by the vanishing gradient. The design and thought pattern behind our base case is further explained in chapter 4.1. The main purpose of *Gradient Based Learning Applied to Document Recognition* was not to just compare variations of FFN's but also test a CNN on the same dataset and compare the results. This CNN was called LeNet-5. In *Gradient Based Learning Applied to Document Recognition* LeNet-5 was able to achieve a higher validation accuracy than all of their FFN's on the MNIST dataset.

Kong et al. compares different activation functions in their paper *Hexpo: A Vanishing-Proof Activation Function* (2017) but the choice of activation functions differs from ours. No study could be found where the effect of TanH, Sigmoid, ReLU and Leaky ReLU was tested and compared on both FFN's and CNN's.

No work could be found where the effect of more filters per convolution layer are compared to only adding more layers to the models for CNN's. There has however been work to investigate the effect of adding more layers to CNN models. In the paper *Deep Residual Learning for Image Recognition* (2016) He et al. benchmarks CNN's models on different image datasets, CIFAR-10 being one of them. He et al. creates residual networks and plain CNN's models. A residual network is essentially a plain CNN with shortcuts every third layer in the model. These shortcuts sends the output to the next layer in the model and also to the layer that is two layers deeper in the model. He et al. created both plain CNN's and residual networks that were 20, 32, 44 and 56 layers deep respectively and all these models used ReLU between convolution layers. That study showed that the plain CNN model with 20 layers achieved the best validation accuracy among the plain CNN's, with a validation accuracy of approximately 90%. In the paper *Implementation of Deep Feedforward Neural Network with CUDA backend for Efficient and Accurate Natural Image Classification* (2017) von Hacht benchmarks different designs of CNN's on the dataset CIFAR-10. Von Hacht created three different models referred to as model A, model B and model C. Model A is based on LeNet 5 created by Yann LeCun in the paper *Gradient Based Learning Applied to Document Recognition* (1998), model B is based on model A, but model B has 3 times the amount of filters for each convolutional layer, the fully connected layers after the convolution layers were however not unaffected. Model B had more neurons after the convolution layers compared to model A. Model C was essentially model B with two layers of convolution before the maxpooling layers instead of one layer of convolution before maxpooling. Von Hacht found that the model C with multiple convolution layers achieved a validation accuracy of 72.88% which is higher than the validation accuracy of the other models.

In the study conducted in this paper it was not feasible to benchmark models that where up to 20 layers deep because of the time it would take to train the models. So the decision to make models that were feasible to train and evaluate within the time constraints of this thesis project, was made. Furthermore this thesis project aims to investigate the effect of only changing the number of convolutional and maxpooling layers versus more filters per convolutional layer for CNN's and also investigate the effect of using different activation functions.

In the paper *RMDL: Random Multilevel Deep Learning for Classification* (2018) Kowarsi et al. combines a *deep neural network*, a CNN and a *Recurrent Neural Network* and they were able to achieve state of the art result on the dataset MNIST, with a validation accuracy of 99.79%. While this was the best current result on the MNIST dataset, learning and combining all these techniques were out of scope for this thesis project.

In the paper *Fractional Max-Pooling* (2015) Benjamin Graham achieves state of the art results on the dataset CIFAR-10 with a validation accuracy of 96.53%. This is however achieved while augmenting the dataset. In this thesis the focus was on the model rather than augmenting the dataset.

Seeing how the state of the art vary in their results, with MNIST achieving a higher validation accuracy due to its simpler images, it is fair to assume that a neural network generally would achieve a higher validation accuracy using the dataset MNIST compared to dataset CIFAR-10.

3 Problem

The following chapter describes the motivation and aim of the study as well as the research questions and the hypotheses that were tested in this thesis. The chapter also describes the objectives that needed to be fulfilled in the study. Finally the chapter will motivate the choice of method when conducting the study and also a discussion regarding alternative methods and why these were not applied in the study.

3.1 Aim

The aim of this study was to benchmark and compare different variations of artificial neural networks, namely FFN and CNN. In this study twelve neural networks were created, six FFN and six CNN. These networks were created, not only to compare FFN vs. CNN but also to test if a variance in depth, width or activation function had any effect on the validation accuracy of a neural network. These different variations of the networks were trained and tested on the dataset CIFAR-10 and the results were compared and benchmarked.

3.2 Motivation

The motivation behind this study stems from the fact that machine learning, at the time of writing, were a growing field and image classification could come to play a large role in the future of communication. Due to the recent surge in recent years in internet-connectivity and mainly due to social media the amount of shared data were larger than ever and the bulk of the data are in form of images. The increase in available data makes the task of manually screening and classifying all images infeasible. This is where the automation of this process using classifiers could significantly improve performance in such tasks ("CS231n Winter 2016: Lecture1: Introduction and Historical Context", 2016).

3.3 Research Questions

The research questions this study aims to answer are the following:

1. To what extent does the width and depth of a FFN affect the validation accuracy of the models used in the experiment?
2. To what extent does the activation functions Sigmoid, TanH, ReLU and Leaky ReLU affect the validation accuracy of a FFN?
3. To what extent does the width and depth of a CNN affect the validation accuracy of the models used in the experiment?
4. To what extent does the activation function Sigmoid, TanH, ReLU and Leaky ReLU affect the validation accuracy of a CNN?
5. Which of the activation functions is able to learn with the fewest epochs on the models used in the experiment?
6. Which architectural models used in this study is able to achieve the highest validation accuracy on the trained models FFN or CNN?

3.4 Hypotheses

The hypotheses for this study were

1. Deeper models are able to achieve higher validation accuracy for both FFN´s and CNN´s
2. ReLU and Leaky ReLU are able to train with fewer epochs compared to Sigmoid and TanH for both FFN's and CNN's
3. Leaky ReLU achieves the highest validation accuracy among all the activation functions for both FFN and CNN
4. Models with more neurons/filters per layer achieves higher validation accuracy
5. CNN is able to achieve an overall higher validation accuracy compared to the FFN counterparts

RQ1: will be answered by hypotheses 1 and 4
RQ2: will be answered by hypothesis 3
RQ3: will be answered by hypotheses 1 and 4
RQ4: will be answered by hypothesis 3
RQ5: will be answered by hypothesis 2
RQ6: will be answered by hypothesis 5

3.5 Objectives

In order to conduct the study a set of objectives needed to be set up and performed. The distribution between the authors of the work can be found in Appendix B - Work distribution. In this study the objectives were as follows:

1. Research the field and related work
2. Research the Keras API
3. Create a FFN model and variations on this model in terms of activation functions, layer-depth and layer-width
   3.1. Create a base case FFN
   3.2. Create a FFN model with more layers than the base case
   3.3. Create a FFN model with more neurons per layer than the base case
   3.4. Create FFN models that have the same structure as the base case but use different activation functions
4. Create a CNN model and variations on this model in terms of activation functions, layer-depth and layer-width
   4.1. Create a base case CNN
   4.2. Create a CNN model with more convolution and pooling layers than the base case
   4.3. Create a CNN model with more filters per layer than the base case
   4.4. Create CNN models that have the same structure as the base case but use different activation functions
5. Train and test all the models using the dataset CIFAR-10
6. Evaluate the FFN results from the experiments in terms of the validation accuracy achieved and epochs of training needed of each individual model
7. Evaluate the CNN results from the experiments in terms of the validation accuracy achieved and epochs of training needed of each individual mode

3.6 Method

In this subchapter the choice of method will be discussed as well as alternative methods and how these could have been utilized instead.

3.6.1 Method of choice: Experiment

According to Wohlin et al. (2012) an experiment is when a few select variables are changed and these variations of variables are tested in a controlled environment in order to see if there is support for a predetermined hypothesis or not. An experiment can never prove a hypothesis but only support it or not, which is a weakness in this particular method. Despite the controlled environment of the experiment there are countless things that could go wrong which could affect the result of the experiment. Wohlin et al. describes an experiment as:

> "...experiments are often done by implementing a model of some system and running simulations to see how the model is affected by different variables." Wohlin et al., 2012, p.65

This quote describes the goal of this study almost to the letter. In order to benchmark and compare various neural networks these networks need to be tested under identical circumstances which strengthens the case for the usage of experiments as the preferred method. In order to produce reliable and comparable results there are a number of threats to validity that needs to be taken into consideration, these will be discussed in chapter 3.7 Validity.

3.6.2 Alternative methods

A *case study* is conducted in a real life setting or environment where the problem is present (Wohlin et al., 2012). This means that the researcher might have less control over external and internal parameters when conducting the study. This poses a drawback that it might be hard to generalize the findings to a broader domain area, since the findings might be case specific. Another drawback is that if two different techniques are compared in the case study, then it might be difficult to say which technique performs better in another setting, which means that it might be difficult to generalize the results to a broader domain. The aim of this study is to benchmark FFN, CNN and different activation functions within the image recognition domain. It is therefore preferable to have the maximum amount of control when conducting the study. To reach conclusive results it is important that the different networks are tested in identical conditions using identical data. Therefore the methodological approach case study was discarded in order to increase the level of control of the benchmarking environment.

A *survey* is used to draw conclusions about the population from which the samples were collected (Wohlin et al., 2012). The survey is generally done before or after some event occur. This could for instance be after a new tool has been used, or before some change in the workflow for employees at a company. The survey can be conducted in forms of interviews and/or questionnaires where a small group of the target audience are selected and their opinions are used to draw conclusions about what the opinions are of entire target audience. Since the aim of the study is not to understand the population, but benchmark FFN, CNN and activation functions within the image recognition domain this methodological approach was discarded.

3.6.3 Dataset of choice: CIFAR-10

CIFAR-10 is dataset that was created by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The dataset contains 60 000 labeled images of 10 different categories automobiles, cats and dogs etc. CIFAR-10 has 50 000 images for training and 10 000 images for validation. The training-set contains 5 000 images of each category and the validation set contains 1 000 images of each category. Each image in the dataset is associated with only one of the categories in the dataset (Krizhevsky, n.d.). 10 samples from each category can be seen in Figure 3.1 below.

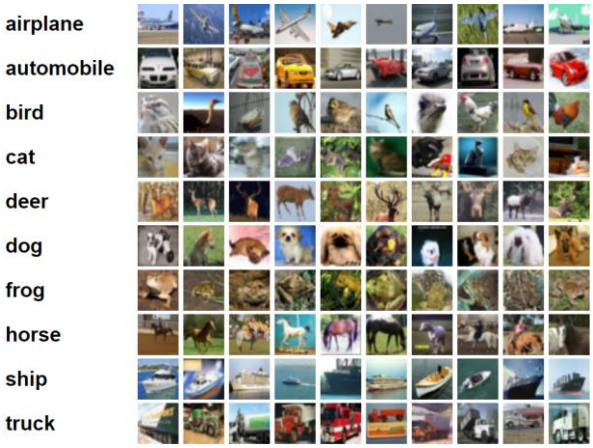


Figure 3.1 CIFAR-10 illustration

3.7 Validity

Wohlin et al. (2012) describe that there are four overarching groups of validity threats: *Conclusion, Internal, Construction* and *External*. Each category of validity threats handles different aspects of the validity of a study. Conclusion validity focus on the relationship between the results from the experiment and conclusions that could be drawn, and what could threaten the validity of those conclusions. Internal validity is about threats to the correlation between the cause and effect of the experiment, there might also be aspects that affect the outcome that was unknown during experiment. Construction validity focus on the threats to the validity with regards to the theoretical background in relation to the experiment, as well as the design of the experimental setting. External validity is about the ability to generalize the result to a broader domain and what threats are associated with this.

Within the group of conclusion validity there are a number of potential threats. Since the study was conducted through an experiment *reliability of measures* were a potential conclusion threat. This meant that the instrumentation and the layout of the experiment needed to be of a high standard and the results should be replicable. *Reliability of treatment implementation* meant that all involved researchers followed the same procedure in order to make sure that each model is treated equally (Wohlin et al., 2012).

Internal validity threats in the study could potentially include *instrumentation*. Instrumentation is when artifacts used in the experiments are poorly designed in such a way that they might have a negative outcome in the experiment. (Wohlin et al., 2012)

Threats to the construction validity in the study includes *inadequate preoperational explication of construct, mono-operational bias* and *experimenter expectancies*. Inadequate explication of construct means that clear definitions of constructs used in the experiment are missing. For example if two approaches are compared to examine if one is better, what does 'better' mean? Mono-operational bias means that only one case or subject is examined. This single case or subject might not represent the domain in an accurate way. Experimenter expectancies means that the bias of the researcher could affect the result, for example the researcher could choose to interpret the result in a way that match their expectations (Wohlin et al., 2012).

Threats to the external validity in the study are *interaction of setting and treatment* and *interaction of history and treatment*. Interaction of setting and treatment means that the tools and practices used in the experiment mimics real world applications and standards. If the conditions in the experiment are too different from the real world, then the results might not reflect reality. Interaction of history and treatment is when the time the experiment is conducted might affect the result. For example if an experiment is run on a computer using the operating system Microsoft Windows, there might be hidden downloads and updates occurring in the background, unbeknownst to the researcher (Wohlin et al., 2012).

4 Implementation

This chapter describes how the neural networks were implemented and what decisions were made along the way.

4.1 Designing a common Base case (Objective 3.1 & 4.1)

A problem in this study was that the techniques FFN and CNN are fundamentally different in many ways. This meant that it was difficult to create two neural networks that were as equal and comparable as possible using these two techniques. As the focus of this study was to benchmark different techniques against each other, and not to test the researchers ability to design neural networks, the choice was made to base the models that would become the base cases on previous work.

A paper from the University of Chalmers *Implementation of Deep Feedforward Neural Network with CUDA backend for Efficient and Accurate Natural Image Classification,* von Hacht (2017) described a similar study where different depth and width of a CNN was benchmarked and compared. In that paper the model for the used CNN was depicted and it referred to another earlier paper called *Gradient Based Learning Applied to Document Recognition* (1998) by Lecun et al.
Since the paper by von Hacht (2017) referred back to Lecun et al. (1998) who described models of both FFNs and CNNs, the decision was made to design the base cases for the experiment in this thesis based on these two papers. The decision was made to recreate the CNN model from von Hacht (2017) with some slight modifications and this CNN was based on the earlier paper (Lecun et al., 1998) whom also described an assortment of FFNs. To make the models as comparable as possible, the choice was to use the same initial source when choosing a FFN base case. Lecun et al. (1998) described the model structure of several FFNs and based on their findings, the model that performed best was chosen or as the FFN base case.

LeNet 5 was used as a basis for creating the CNN in the report from von Hacht (2017). Some modifications was done by von Hacht to accommodate for images in 3 colors instead of black and white images as found in the MNIST dataset. The activation functions found in the paper was replaced by ReLU and the subsampling layers was replaced by max pooling. With this in mind FFNs from Lecun et al. (1998*)* was used as described earlier. LeCun et al. (1998) benchmarked MNIST black and white images so some modifications were made such as an increase of the layer sizes to accommodate the three color channels required for CIFAR-10. The activation functions were also replaced with ReLU in order to keep the base cases as comparable a possible. All models used softmax to calculate its output.

4.2 Keras (Objective 2)

The decision was made early on to use the API *Keras* when implementing the models. Keras is a high level API that is capable of running atop of *Tensorflow*, the machine learning library used in the study ("Keras", 2019). The reason Keras was chosen was twofold; the high level nature enables fast and easy implementation of models despite limited experience of machine learning and the python programing language, and Keras was also able to run on a CPU or a GPU.

Due to the high level nature of Keras it was possible to create entire layers of a neural network in only one line of code. This did not only simplify the design process, but it also produced code that was easy to read and understand, even if the reader lacked knowledge of the programming language python.

4.3 Shared architecture

In an attempt to keep the files organized and as short and readable as possible all the models were written in separate files. These files could then be loaded into another shared file when it was time to train. The shared file used for training the networks were simply called *Training.py*. The source code for this file can be found in Appendix D - Source Code. *Training.py* did the following things:

- Defined parameters for the size of the images and color depth, and these were shared between all the models
- The random seed was initialized in order to make all models initialization and training comparable
- A model template was loaded and a new model was created according to specification
- Training was done in this file through k-fold cross-validation and the *k* was defined within this file
- Since CIFAR-10 has its own pre-determined training- and validation-set that did not match up with the desired k, the dataset was merged and then resized to fit the desired k-fold cross-validation
- The model was saved when it achieved its lowest validation loss

Results.py was created to abstract the results from the training and thereby make it safe to run this script without affecting the results from the trained models. The source code for this file can be found in *Appendix D* - Source Code. *Results.py* did the following things:

- Load the validation accuracy, training accuracy, validation loss and training loss for each epoch during training for all models
- Create average training validation accuracy, training accuracy, validation loss and training loss for each epoch for each model, this data was used to draw the graphs
- Produce a txt-file for each model where the average validation accuracy, training accuracy, validation loss and training loss was saved, as well as a confusion matrix with the predictions for all the images in the data set, for all k folds used.
- Performed paired t-tests to see if there was any statistical difference between the training needed for the model, as well as compare if there was a statistical difference in validation accuracy achieve by the fully trained models.

A file called *Statistics.xlsx* was used to plot the graphs used in the report by loading the txt-files produced by *Results.py*. *Statistics.xlsx* also visualized the tables and confusion matrices used in this report.

4.4 Publicly available data

The implementation and results produced in this study was made publicly available at *Github*[1]. The models were however too large to fit in a Github repository, so these were made publicly available on G*oogle drive*[2]. The source files for training can also be found in Appendix D - Source Code.

4.5 FeedForward Network (Objective 3)

In the paper *Gradient Based Learning Applied to Document Recognition* Lecun et al. (1998) describes several models of FFNs and specifies their layer size and their performance. The FFN that achieved the highest performance was a model that had one input layer, two hidden layers and one output layer. An illustration of the FFN that was used by Lecun can be seen in Table 4.1 below.

Table 4.1 Lecun's FFN – The difference from the base case used in this thesis is the amount of neurons per hidden layer and the activation function used. These have been marked in bold.

Input	Layer type
32 x 32	Input
500	Fully connected, **TanH**
150	Fully connected, **TanH**
10	Output, Softmax

4.5.1 FFN Base (Objective 3.1)

The model from *Gradient Based Learning Applied to Document Recognition* was chosen as the base case for the FFNs in this study, it did however need some modifications. Since Lecun et al. (1998) had tested their models on the MNIST dataset this network was designed with a dataset consisting of black and white images in mind and not a full color image dataset like CIFAR-10. A colored dataset means that the input needs to be split into 3 color channels red, green and blue (*RGB*). This meant that the input layer was given the resolution of the image times 3 to compensate for the 3 colors, however the variables are defined in *Training.py* and therefore they were not specified in the model template. To further compensate for the increased size of the input, the hidden layers in the FFN base case are tripled in size in comparison to the FFN presented by Lecun et al. (1998) due to the three color channels. The output layer is still the same size in both networks since both MNIST and CIFAR-10 has 10 different classes as potential outcome. An illustration of the FFN Base model can be seen in Table 4.2 below.

Table 4.2 FFN Base

Input	Layer type
32 x 32 x 3	Input
1500	Fully connected, ReLU
450	Fully connected, ReLU
10	Output, Softmax

[1] https://github.com/a15magknf16linli/A-COMPARATIVE-STUDY-OF-FFN-AND-CNN-WITHIN-IMAGE-RECOGNITION
[2] https://drive.google.com/open?id=1nlzaW7kCEkgTnWjSFBDsIzCK4GcylPON

4.5.2 FFN Deep (Objective 3.2)

The deeper variant of the base case had one more hidden layer of 150 neurons added to it. The decision for the size of the extra layer was based on the models used in the paper by Lecun et al. (1998) where the size of layers often decreased when placed later in a network. Thus an extra hidden layer with the size of 150 neurons were added to the model. An illustration of the model FFN Deep can be seen in Table 4.3 below.

Table 4.3 Deep FFN – The difference from the base case used in this thesis is the one additional hidden layer with 150 neurons have been added. This new layer have been marked in bold.

Input	Layer type
32 x 32 x 3	Input
1500	Fully connected, ReLU
450	Fully connected, ReLU
150	**Fully connected, ReLU**
10	Output, Softmax

4.5.3 FFN Wide (Objective 3.3)

When designing the model templates that were used to build the variations in terms of width and depth the base case was used as a starting point. The wide network is the same as the base case with the difference that the number of neurons in the hidden layers have been doubled. The reasoning for this was simply to increase the number of neurons by a factor of 2. An illustration of the model FFN Wide can be seen in Table 4.4 below.

Table 4.4 Wide FFN – The difference from the base case used in this thesis is the amount of neurons per hidden layer, as they have been doubled. These have been marked in bold

Input	Layer type
32 x 32 x 3	Input
3000	Fully connected, ReLU
900	Fully connected, ReLU
10	Output, Softmax

4.5.4 FFN Sigmoid, TanH, Leaky ReLU (Objective 3.4)

When creating the variations in terms of activation function the base case was entirely reused with the only difference being that the hidden layers used a different activation function, meaning that the activation function was changed to either Sigmoid, TanH or Leaky ReLU. This was done in order to keep the models as comparable as possible. An illustration of the model FFN Sigmoid can be seen in Table 4.5 below.

Table 4.5 Sigmoid FFN – The difference from the base case used in this thesis is the activation function used in the hidden layers. These have been marked in bold. Note that the net structure is identical for the other activation functions.

Input	Layer type
32 x 32 x 3	Input
1500	Fully connected, **Sigmoid**
450	Fully connected, **Sigmoid**
10	Output, Softmax

4.6 Convolutional Neural Network (Objective 4)

In the paper *Implementation of Deep Feedforward Neural Network with CUDA backend for Efficient and Accurate Natural Image Classification,* von Hacht (2017) created three variations of CNNs for the CIFAR-10 dataset. These models were based on the LeNet 5 CNN presented in the paper *Gradient Based Learning Applied to Document Recognition* (Lecun et al., 1998). The smallest of these models was chosen as the base case for this experiment. An illustration of the model created by von Hacht can be seen in Table 4.6 below.

Table 4.6 CNN August von Hacht - The differences from the CNN Base case is that von Hacht used dropout layers in his network. This have been marked in bold letters.

Input size	Layer type	parameters
3 x 32 x 32	Convolution	5 x 5, 32 filter, stride = 1, padding = 2
32 x 32 x 32	ReLU	
32 x 32 x 32	Max pooling	Pooling size = 2, stride = 2, padding = 0
32 x 16 x 16	Convolution	5 x 5, 64 filter, stride = 1, padding = 2
64 x 16 x 16	ReLU	
64 x 16 x 16	Max pooling	Pooling size = 2, stride = 2, padding = 0
4096	Fully connected	Output = 1024
1024	ReLU	
1024	**Dropout**	**P = 0,5**
1024	Fully connected	Output = 10
10	Softmax	

4.6.1 CNN Base (Objective 4.1)

The model CNN Base is identical to the model created by von Hacht, except that the base case in this experiment does not use any dropout layer. A representation of the CNN Base model used in this experiment can be seen in Table 4.7 below.

Table 4.7 CNN Base

Input size	Layer type	parameters
3 x 32 x 32	Convolution	5 x 5, 32 filter, stride = 1, padding = 2
32 x 32 x 32	ReLU	
32 x 32 x 32	Max pooling	Pooling size = 2, stride = 2, padding = 0
32 x 16 x 16	Convolution	5 x 5, 64 filter, stride = 1, padding = 2
64 x 16 x 16	ReLU	
64 x 16 x 16	Max pooling	Pooling size = 2, stride = 2, padding = 0
4096	Fully connected	Output = 1024
1024	ReLU	
1024	Fully connected	Output = 10
10	Softmax	

4.6.2 CNN Deep (Objective 4.2)

The model CNN Deep was created to make it possible to observe the effect of training- and validation accuracy achieved when having more convolutional layers together with max pooling layers. The decision was made to add an additional layer of convolution to the model together with a max pooling layer, which can be seen in Table 4.8 below. The decision for using 128 filters in the added convolution layer was that it´s common that new layers of convolution after a pooling layer has twice the amount of filters as the previous convolution layer. Thus the same approach was used here.

Table 4.8 CNN Deep - The differences from the CNN base case are three added layers. These have been marked in bold letters.

Input size	Layer type	parameters
3 x 32 x 32	Convolution	5 x 5, 32 filter, stride = 1, padding = 2
32 x 32 x 32	ReLU	
32 x 32 x 32	Max pooling	Pooling size = 2, stride = 2, padding = 0
32 x 16 x 16	Convolution	5 x 5, 64 filter, stride = 1, padding = 2
64 x 16 x 16	ReLU	
64 x 16 x 16	Max pooling	Pooling size = 2, stride = 2, padding = 0
64 x 8 x 8	**Convolution**	**5 x 5, 128 filter, stride = 1, padding = 2**
128 x 8 x 8	**ReLU**	
128 x 8 x 8	**Max pooling**	**Pooling size = 2, stride = 2, padding = 0**
2048	Fully connected	Output = 1024
1024	ReLU	
1024	Fully connected	Output = 10
10	Softmax	

4.6.3 CNN Wide (Objective 4.3)

The model CNN Wide was created to make it possible to observe the effect of training- and validation accuracy achieved when having more filters per convolutional layer. This model used twice as many filters as the model CNN Base for each layer of convolution. The reasoning for this was that for each layer of convolution the amount of filter increase by a factor of two, so the same was used here. An illustration of the model CNN Wide can be seen in Table 4.9 below.

Table 4.9 CNN Wide - The differences from the CNN base case is that the amount of filter used have been doubled. These have been marked in bold letters.

Input size	Layer type	parameters
3 x 32 x 32	Convolution	5 x 5, **64** filter, stride = 1, padding = 2
64 x 32 x 32	ReLU	
64 x 32 x 32	Max pooling	Pooling size = 2, stride = 2, padding = 0
64 x 16 x 16	Convolution	5 x 5, **128** filter, stride = 1, padding = 2
128 x 16 x 16	ReLU	
128 x 16 x 16	Max pooling	Pooling size = 2, stride = 2, padding = 0
8192	Fully connected	Output = 1024
1024	ReLU	
1024	Fully connected	Output = 10
10	Softmax	

4.6.4 CNN Sigmoid, TanH, Leaky ReLU (Objective 4.4)

The model CNN Sigmoid was identical to CNN Base, but used the Sigmoid activation function instead of the ReLU activation function and the model can be seen in Table 4.10. The model CNN TanH replaced the activation functions to TanH in the same way as Sigmoid replaced ReLU to Sigmoid, otherwise it is identical to the model CNN Base. The model CNN Leaky ReLU replaced the activation functions to Leaky ReLU in the same manner as the models CNN Sigmoid and CNN TanH.

Table 4.10 CNN Sigmoid - The differences from the CNN base case is the activation function used in the hidden layers. These have been marked in bold. Note that the net structure is identical for the other activation functions

Input size	Layer type	parameters
3 x 32 x 32	Convolution	5 x 5, 32 filter, stride = 1, padding = 2
32 x 32 x 32	**Sigmoid**	
32 x 32 x 32	Max pooling	Pooling size = 2, stride = 2, padding = 0
32 x 16 x 16	Convolution	5 x 5, 64 filter, stride = 1, padding = 2
64 x 16 x 16	**Sigmoid**	
64 x 16 x 16	Max pooling	Pooling size = 2, stride = 2, padding = 0
4096	Fully connected	Output = 1024
1024	**Sigmoid**	
1024	Fully connected	Output = 10
10	Softmax	

4.7 Training (Objective 5)

In the interest of validity all training were performed on the same hardware. At the time of writing the desktop used could be classified as high-end. Since Keras has the ability to train using the GPU (Graphics Processing Unit) the hardware can have a significant impact on the real world time that is required to train. The GPU used in the experiment was a MSI GeForce RTX 2080 Ti 11GB GAMING X TRIO. On this set-up the training took between 4 and 9 microseconds per sample, which made 100 epochs and cross-validation with a k of 10 feasible. In an effort to prevent hidden background activity in Microsoft Windows, the training was done while disconnected from the internet.

Hardware specs:
Motherboard: MSI Z370 GAMING PRO CARBON
CPU: Intel Core i7 8700K 3.7 GHz 12MB
GPU: MSI GeForce RTX 2080 Ti 11GB GAMING X TRIO
Memory: Corsair 16GB (2x8GB) DDR4 2666Mhz CL16 Vengeance
Hard Drive: Samsung 970 EVO 250GB
Power supply: Corsair Vengeance 750M 750W

Since python 3.7 did not support Keras, a new environment had to be set up. This was done using Miniconda, where a temporary environment in Python 3.6 was set up. In this temporary environment *Keras-gpu* was installed along with the library S*klearn*, which was needed for stratified k-fold cross validation. The code was written and executed in the IDE Spyder.

5 Results

This chapter will present and analyze the results that were gathered in this study. First the results from both the FFN and CNN will be presented in the shape of graphs and confusion matrices. In the next subchapter the data will be discussed and which conclusions that can be drawn based on the results and the research questions will be answered.

5.1 Presentation of the data

The data that was gathered was summarized into graphs, tables and confusion matrices and these will be presented in this subchapter. The graphs will cover all of the FFNs and CNNs both in terms of their validation accuracy and validation loss. The graphs were created by taking every epoch of each *k* in the training and printing the average in a graph. This results in two graphs per model. One graph shows the training- and validation accuracy, and another separate graph shows the training- and validation loss. For each model a confusion matrix was created. The confusion matrices shows the rate of which predictions were correct as well as which output classes were mixed up. The matrices were assembled by taking the sum of every separate *k* and averaging these. In order to prove or disprove any statistical significance in the collected data, paired t-tests was conducted. A paired t-test is a test where two correlating points are compared to each other, such as the same epoch in two different models.

5.1.1 FeedForward Network - Results (Objective 6)

Figure 5.1 to Figure 5.6 shows the graphs for both the training- and validation accuracy as well as the training- and validation loss of the FFNs. When studying these graphs it's possible to discern that none of the FFN's reached 100% training accuracy and the validation accuracy ended up around 50% with the exception of FFN Sigmoid which can be seen in Figure 5.4, where the validation accuracy ended around 35%.

The loss of the FFN's are more varied than the accuracy. The training loss kept decreasing in an almost linear fashion, with the exception of Sigmoid which did not decrease at such a high rate. The training loss for the models: FFN Base, FFN Deep and FFN Wide can be seen in Figure 5.1, Figure 5.2 and Figure 5.3, they all ended below 0.5 after 100 epoch and Sigmoid again performed the worst ending with a training loss of around 2.0. Another shared trait between FFN Base, FFN Deep and FFN Wide was that the validation loss decreased in the beginning, but after a certain point the loss began to increase and kept increasing to the point where the loss was higher or equal after 100 epochs compared to what it was at the start of the training. The validation loss of Sigmoid, TanH and Leaky ReLU, all kept decreasing in general but reached their lowest before the 100th epoch.

In Table 5.1 the confusion matrix for the FFN Base case is presented. Confusion matrices for the other models can be found under Appendix C - Confusion matrices. These matrices contain the predicted classes for all images when they are used as validation images for all ten configurations of k-fold cross-validation. The matrices display a percentage of how the models predicted and of how often the prediction was correct as well as incorrect on the validation set. Since it's a matrix it's also possible to see how often an output class was confused with another class, as well as which classes are often confused for one and other.

By viewing the confusion matrix for the base case it was possible to discern that the output class that was accurately predicted the most times was the images of ships which was accurately predicted 66.52% of the times. The lowest percentage for correct predictions were the output class cats, with 28.73%. A trend between the models that could be observed in the confusion matrices was that all the models had the same class as the highest accurate prediction, ship. All models also had the same class as the second highest accurate prediction, automobile. The lowest accuracies were of the images of animals where the models FFN Base, FFN Wide, FFN TanH and FFN Leaky ReLU all had cat as the lowest while the model FFN Deep and FFN Sigmoid had dog as its least accurately predicted output class.

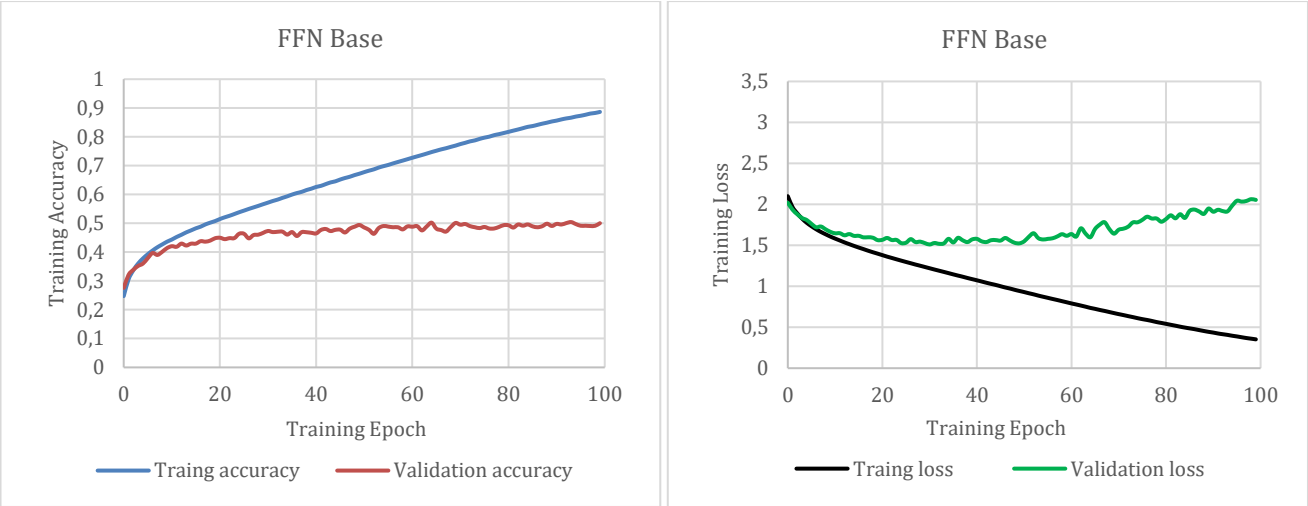


Figure 5.1 ***FFN Base*** *- The graphs show the improvement in accuracy and loss for each epoch during training*

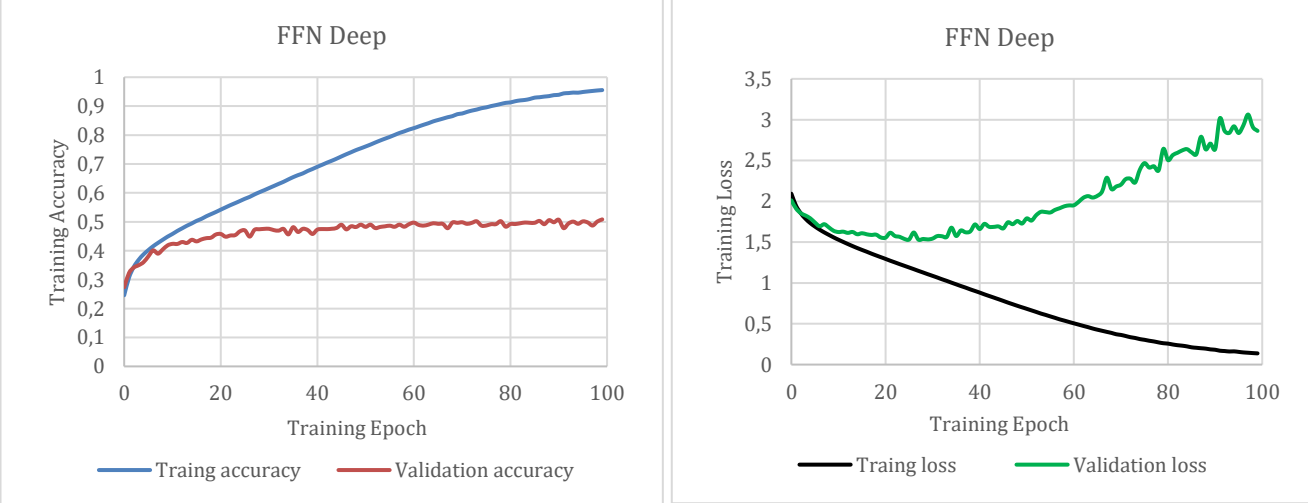


Figure 5.2 ***FFN Deep*** *- The graphs show the improvement in accuracy and loss for each epoch during training*

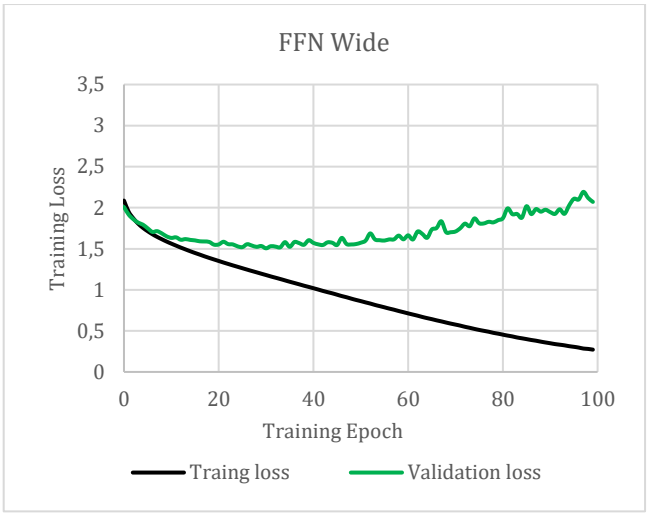
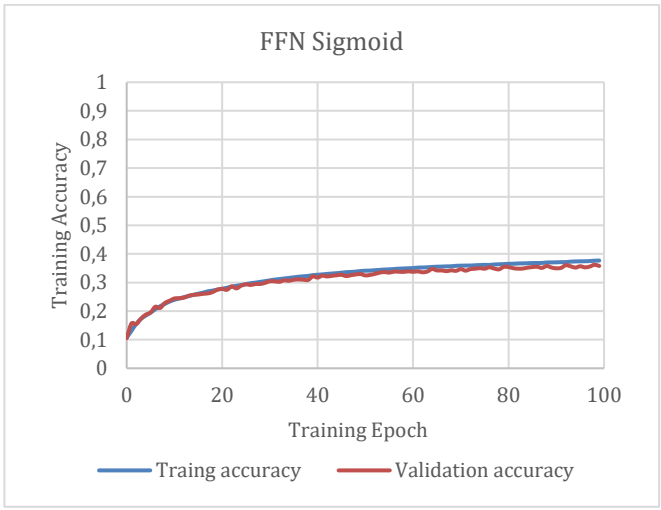


Figure 5.3 ***FFN Wide*** *- The graphs show the improvement in accuracy and loss for each epoch during training*



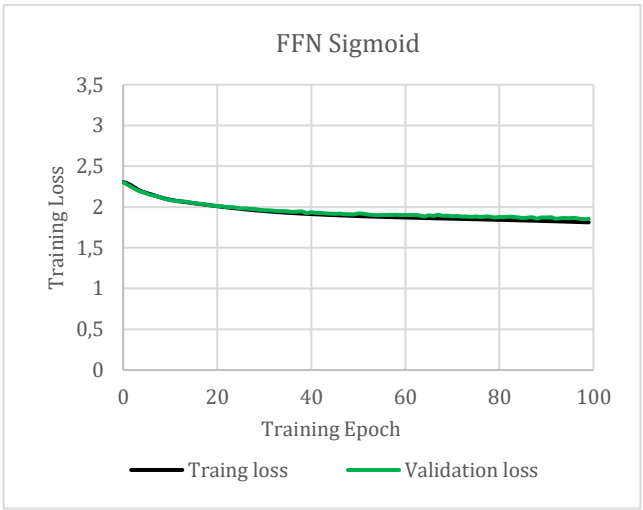
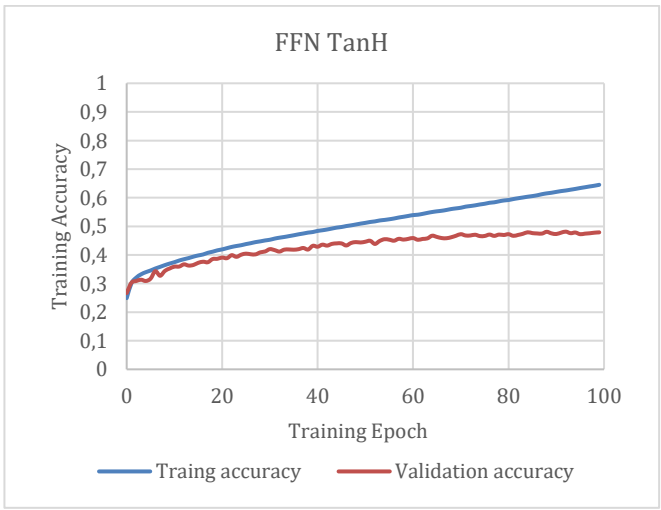


Figure 5.4 ***FFN Sigmoid*** *- The graphs show the improvement in accuracy and loss for each epoch during training*



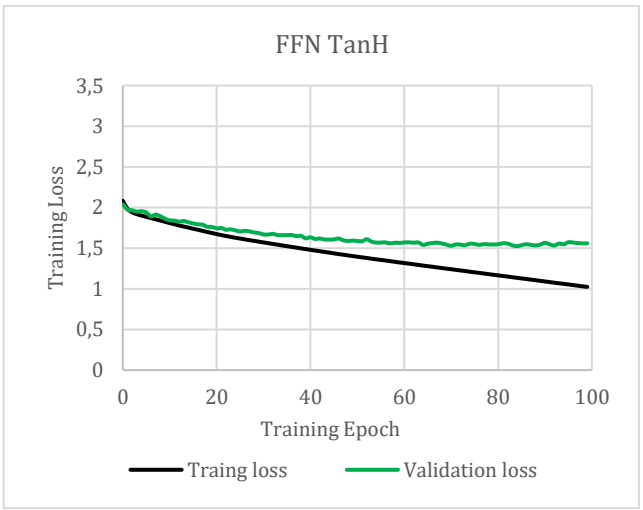


Figure 5.5 ***FFN TanH*** *- The graphs show the improvement in accuracy and loss for each epoch during training*

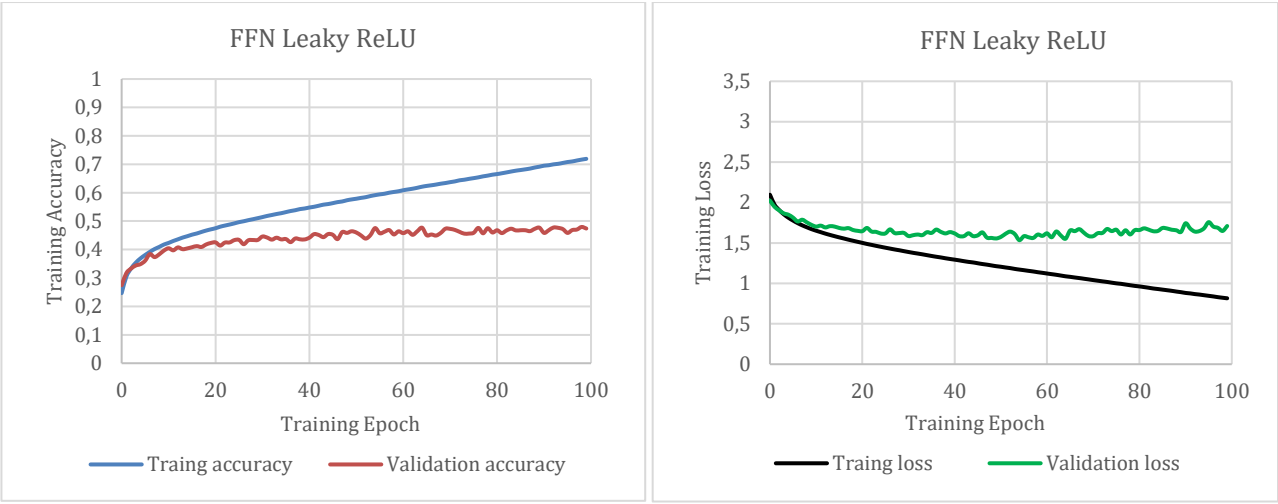


Figure 5.6 ***FFN Leaky ReLU*** *- The graphs show the improvement in accuracy and loss for each epoch during training*

Table 5.1 FFN Base confusion matrix actual vs predicted, the rows are the actual values, and the columns are the predicted values

	P Airplane	P Automobile	P Bird	P Cat	P Deer	P Dog	P Frog	P Horse	P Ship	P Truck
Airplane	54.37	3.13	9.45	1.63	5	1.68	3.25	3.32	12.62	5.55
Automobile	3.25	62.6	1.65	1.98	1.35	1.22	2.85	2.02	7	16.08
Bird	8.13	2.42	41.67	6.87	12.12	8.37	8.63	6.4	2.67	2.73
Cat	3.35	2.4	11.57	28.73	7.32	18.4	13.85	5.97	3.4	5.02
Deer	7.5	1.68	16.38	4.62	40.87	6.07	6.98	8.77	4.58	2.55
Dog	2.48	1.75	13.52	18.28	7.9	33.08	10.1	7.65	2.07	3.17
Frog	2.75	2.57	9.07	8.83	6.93	7.35	52.62	3.28	2.83	3.77
Horse	3.75	1.38	6.58	4.67	8.07	6.58	2.93	60.38	1.22	4.43
Ship	8.98	6.08	2.55	1.83	3.47	1.27	2.07	1.45	66.52	5.78
Truck	4.1	14.72	1.77	2.87	1.62	2.4	3	4.63	6.93	57.97

5.1.2 Convolutional Neural Network - Results (Objective 7)

The left graph Figure 5.7 to Figure 5.12 show the correlation between the average training accuracy per epoch of the model in relation to the average validation accuracy per epoch of the same model, while the right graph represent the relation between the average loss on the training data per epoch of the model in relation to the average validation loss per epoch for the same model.

Figure 5.7 shows the performance of the model CNN Base during training. The left graph shows that the training accuracy increased for each successive epoch and after approximately 30 epochs the model achieved 100% accuracy on the training data. The same pattern can be seen in Figure 5.8 CNN Deep, Figure 5.9 CNN Wide and Figure 5.11 CNN TanH. The Figure 5.12 CNN Leaky ReLU seems to have needed approximately 40 epochs to achieve 100% accuracy on the training data, which was slightly higher compared to the previous CNN models.
The model CNN Sigmoid which can be seen in Figure 5.10 appears to have had a different learning pattern compared to the previously discussed models. For the first few epochs CNN Sigmoid achieved little to no improvement in its training accuracy. Then there was a slight burst where the model improved its accuracy on the training data significantly for just a few epochs. Then the improvement per epoch slowed down significantly again to a slower but even pace where the training accuracy increased a little bit for each epoch. The model was only able to achieve approximately 50% after 100 epochs.

The validation accuracy can be seen in the left graph in Figure 5.7 for the CNN Base model. This graph show that this model was able to achieve just under 70% accuracy on the validation data after approximately 30 to 40 epochs, and then stabilized for the remainder of the training. The same trends could be seen in Figure 5.8 CNN Deep, Figure 5.9 CNN Wide, Figure 5.11 CNN TanH and Figure 5.12 Leaky ReLU however these models were all able to achieve a slightly higher validation accuracy of approximately 70%.
The model CNN Sigmoid which can be seen in Figure 5.10 seemed to have had a different rate of improvement in validation accuracy compared to the other models. However the accuracy for the training data and the validation data seemed to improve in the same manner for each epoch and followed each other closely for the entire training period.

The training loss for the model CNN Base which can be seen in the right graph in Figure 5.7 decreased for each epoch and approached 0 after approximately 30 epochs. The same pattern could be seen in Figure 5.8 CNN Deep, Figure 5.9 CNN Wide and Figure 5.11 CNN TanH. The Figure 5.12 CNN Leaky ReLU had a similar pattern for its training loss compared to the previously mentioned models but needed approximately 40 epochs for the training loss to approach 0 instead of 30.
The model CNN Sigmoid which can be seen in Figure 5.10 seemed to be very different from the previously mentioned models with regards to training loss. The graph indicate that the training loss decreased very slowly for the approximately first 20 epochs. Then the training loss started to decrease at a slightly higher rate for the remainder of the training, even though the speed of decline had increased, it is still low compared to the other models. The model was able to achieve a training loss of approximately 1.5 after 100 epochs.

The validation loss for the model CNN Base which can be seen in the right graph of Figure 5.7 reached a validation loss of approximately 1.0 within the first 20 epochs. After the first 20 epochs the validation loss increased relatively quickly until it reached epoch 40 where the validation loss was approximately 2.0. After epoch 40 the validation loss continued to increase but started to plateau and reached a validation loss of approximately 2.4 after 100 epochs. The same pattern can be seen in Figure 5.8 CNN Deep. In both Figure 5.9 CNN Wide and Figure 5.12 CNN Leaky ReLU a similar pattern can be seen compared to the previous models but they plateaued at a validation loss of approximately 2.0 after 100 epochs. Figure 5.11 CNN TanH also followed the same trend as the previously mentioned models, but plateaued at a validation loss of approximately 1.5

The validation loss for the model CNN Sigmoid seen in Figure 5.10 followed a different pattern compared to the other models, where the validation loss closely followed the training loss for all 100 epochs of training. This model was able to achieve a validation loss of approximately 1.5.

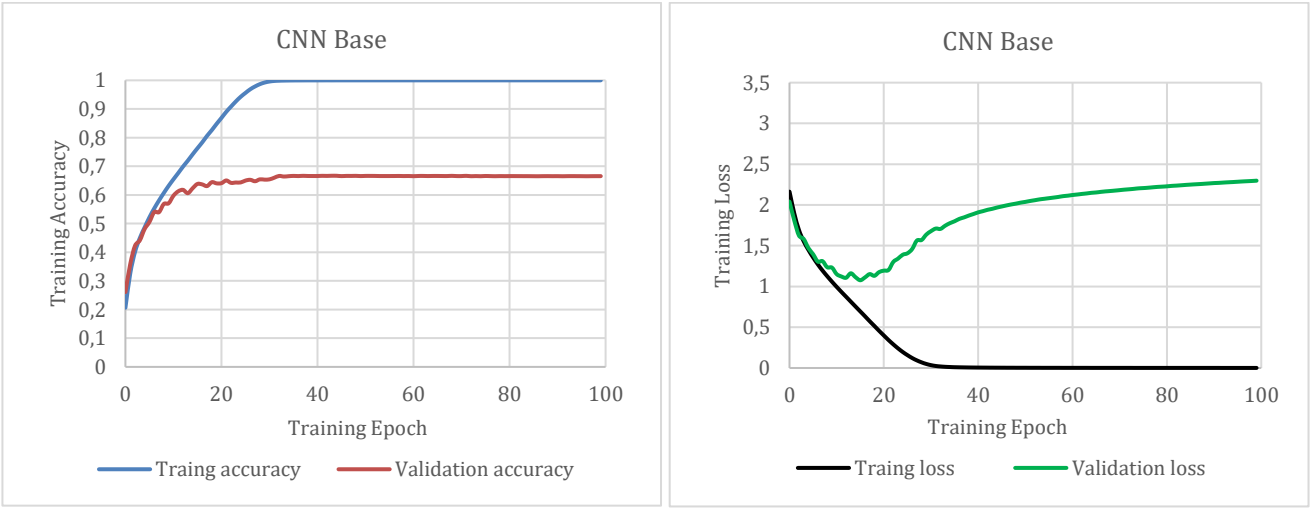


Figure 5.7 ***CNN Base*** *- The graphs show the improvement in accuracy and loss for each epoch during training*

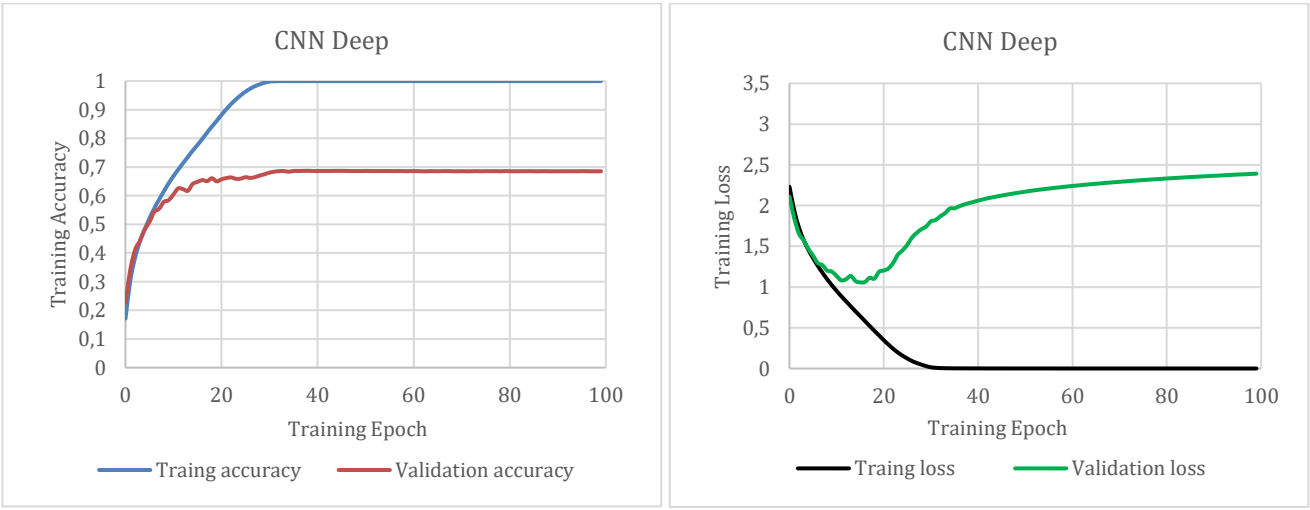
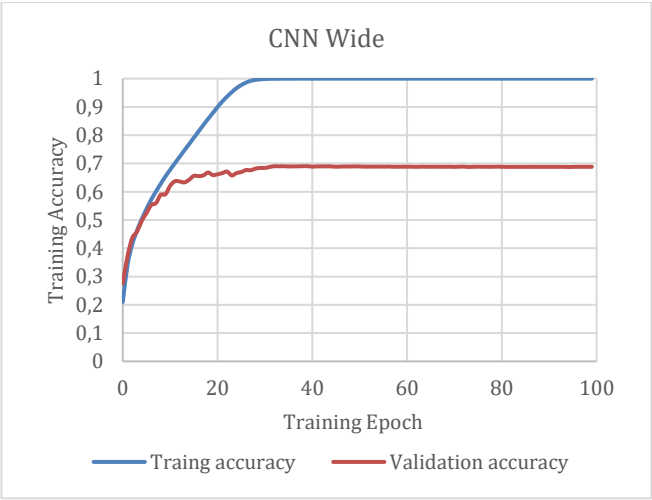


Figure 5.8 ***CNN Deep*** *- The graphs show the improvement in accuracy and loss for each epoch during training*



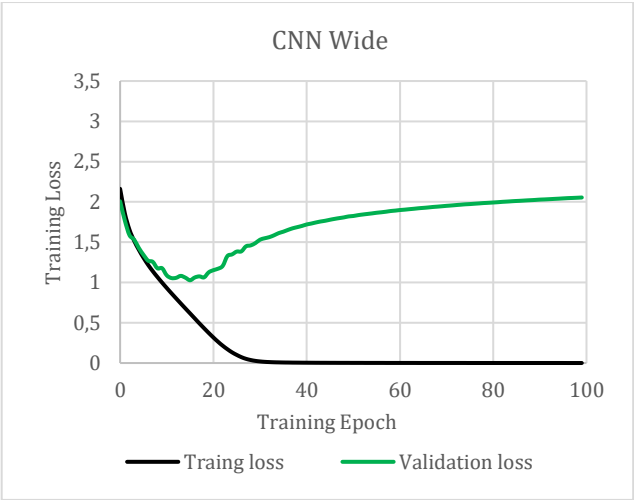
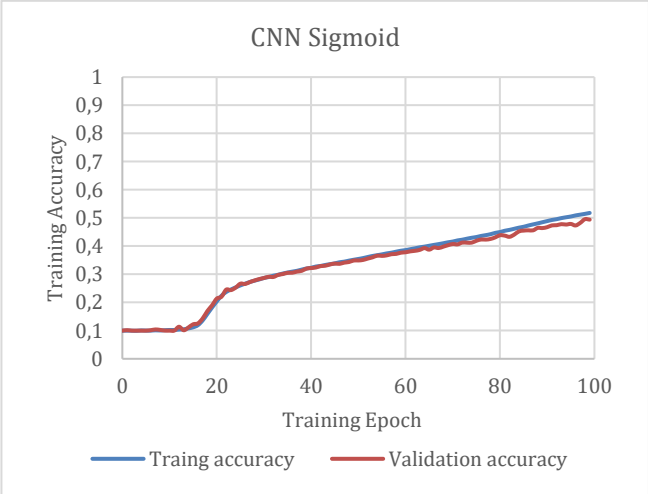


Figure 5.9 ***CNN Wide*** *- The graphs show the improvement in accuracy and loss for each epoch during training*



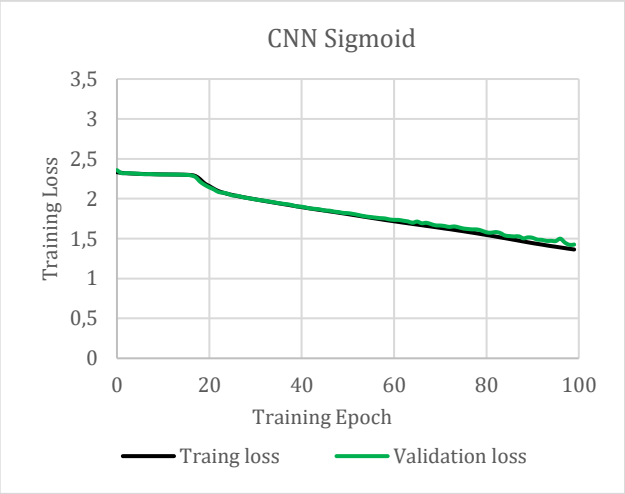
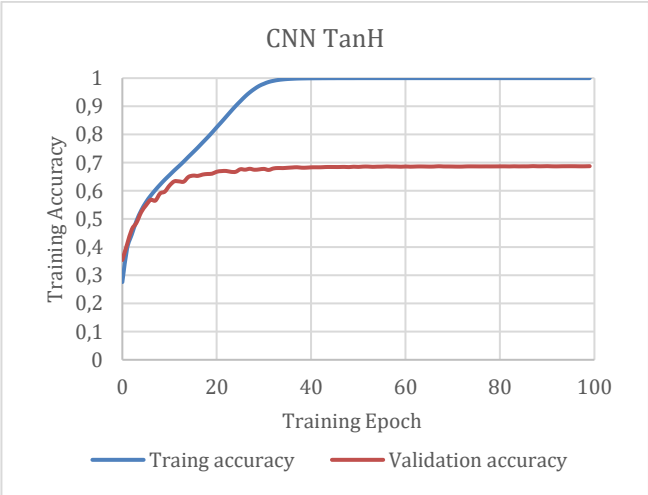


Figure 5.10 ***CNN Sigmoid*** *- The graphs show the improvement in accuracy and loss for each epoch during training*



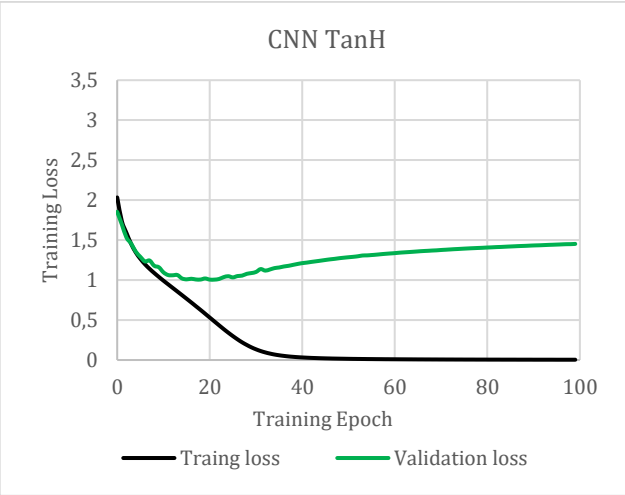


Figure 5.11 ***CNN TanH*** *- The graphs show the improvement in accuracy and loss for each epoch during training*

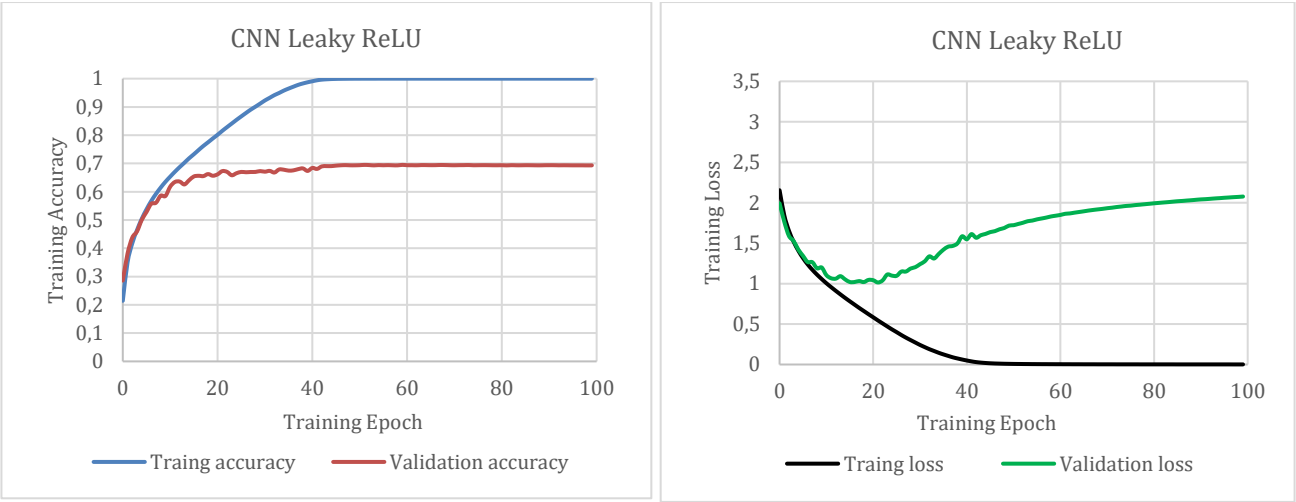


Figure 5.12 ***CNN Leaky ReLU*** *- The graphs show the improvement in accuracy and loss for each epoch during training*

5.2 Analysis of the data

In order to make conclusive statements regarding the collected data, some further processing was required. When studying the graphs some unexpected trends became apparent in terms of the validation loss when training the models. In several cases the validation loss decreased at the start of the training, but after a certain point, began to increase again. Rather peculiarly, the validation accuracy kept increasing despite an increase in the validation loss. In light of this, the decision was made to deem a model as fully trained at the point of lowest validation loss, rather than at the point of highest validation accuracy.

The choice was made to only save the model when it achieved its lowest validation loss, i.e. its universal minima. This was done because when the validation loss was at its lowest point the model was able to generalize the images as good as possible on the validation data. Even if the models appears to keep learning even after this point, since its validation accuracy continued to rise this simply meant that the model was more certain in its predictions. This resulted in correct predictions with a higher degree of certainty, but also incorrect predictions with a higher degree of certainty. Hence the validation loss grew higher and thus the model was saved at its universal minima. In other words, when the model achieves its minimum validation loss, the model was done training and these results were used to compare the models to each other.

Validation accuracy (for each model)

1. Look at each of the 10 folds for the given model
2. Determine the epoch when the validation loss was the lowest for each individual fold (model done training)
3. Determine the validation accuracy for each individual fold when the model was done training (lowest validation loss)
4. Sum all the validation accuracies for each individual fold and divide the result by the number of folds

Statistical evidence

In order to determine whether there was any statistical evidence that there was any difference in the amount of epochs needed to train the model, two tailed paired t-tests were performed with a p-value of 0.001. The t-tests were done for all combinations of models meaning that the model FFN base was tested against every other model, and the same process was done for each of the models in the experiment. Since 10 fold cross-validation was used in this experiment each model was trained 10 times, and each fold consisted of 100 epochs. The choice to perform paired t-tests was made since the epochs made up the data points in the experiment and epoch 1 in fold 1 was closely connected with epoch 1 in the other folds, and thus the individual epochs could be compared. This meant that when all the models were finished training there were a total of 10 epochs, one per fold, where the validation loss was at its lowest and the model was deemed to be trained. All models used the same random-seed which indicate that they split the dataset in the same way, thereby the first result for one model is comparable to the first result of any other model.

The initial plan was to have a significance level of 0.05, meaning a 5% chance to incorrectly discard the null-hypothesis for the statistical tests. However since multiple t-test was conducted a compensation was needed for each individual test in order achieve an overarching significance level of 0.05. The decision was made to use *bonferroni correction*. Bonferroni correction states that the overall significance level should be divided by the amount of tests done, and this significance level should be used for each individual test in order to achieve the overall significance level.
Each model was compared to every other model, 6 FFN and 6 CNN, and this resulted in 66 t-tests. The t-test was done once to compare the amount of epochs needed to train the model and once again to compare the achieved validation accuracy for each model. The total number of t-tests done was thereby 66+66 = 132. This meant that according to the bonferroni correction the significance level that should be used is:

- $0.05/132 = 0.000378$

However in the t-table used in this experiment (Box, Hunter & Hunter, 2012) the minimum significance level for a two tailed t-test was 0.001. So the significance level of 0.001 was used for each t-test, this would however mean that the overall significance level would be slightly higher than the initial plan and result in a significance level of:

- $0.001 * 132 = 0.132$

5.2.1 FeedForward Network - Analysis (Objective 6)

In order to analyze the results from the experiment in terms of the FFNs, statistical t-test were used in order to test the hypotheses of this study. Table 5.2 shows the models average validation accuracy and an average of how many epochs it took to achieve the lowest validation loss.

In Table 5.3 the result from the paired t-tests for the amount of epochs needed to train each model is presented. Table 5.4 on the other hand present the results from the t-test where the validation accuracy for different FFN models are presented.

H1: Deeper models are able to achieve higher validation accuracy for both FFN´s and CNN´s

(Not supported)

This hypothesis was that a deeper model would result in a higher validation accuracy compared to a shallower model. By studying the graphs in Figure 5.1 to Figure 5.6 it seemed apparent that the model FFN Deep and every other FFN model, with the exception of FFN Sigmoid, were able to achieve similar validation accuracy. The Table 5.2 shows that the deep model achieved a validation accuracy of 49.4% which was lower than both the wide model which had a validation accuracy 50.4% and the TanH model with 48.8% validation accuracy. Despite this, according to the paired t-test as can be seen in Table 5.4 with a p-value of 0.001 the differences between the outcomes were too small to be statistically significant. The exception was Sigmoid, which achieved a significantly lower validation accuracy compared to the other models. This hypothesis was thus not supported.

H2: ReLU and Leaky ReLU are able to train with fewer epochs compared to Sigmoid and TanH for both FFN's and CNN's

(Not supported)

The hypothesis states that the activation functions ReLU and Leak ReLU trains faster than other activation functions, meaning that they would reach their universal minima of validation loss after fewer epochs than their counterparts. When studying Table 5.2 it was possible to see that the model using the ReLU activation function, which is named FFN Base in the table had finished its training after 35 epochs compared to FFN Sigmoid which kept learning until epoch 98, and FFN TanH kept learning until epoch 84. Table 5.3 shows that the difference in number of epochs required to train FFN Base compared to both FFN Sigmoid and FFN TanH were large enough to be statistically significant with a p-value of <0.001. However FFN Leaky ReLU kept learning until epoch 69 which was faster than both FFN TanH and FFN Sigmoid, but the difference between FFN Leaky ReLU and FFN TanH was not large enough for it to be statistically significant.

Training epochs:

ReLU needs less training than Sigmoid = **True**
ReLU needs less training than TanH = **True**
Leaky ReLU needs less training than Sigmoid = **True**
Leaky ReLU needs less training than TanH = **False**

This hypothesis was thus not supported.

H3: Leaky ReLU achieves the highest validation accuracy among all the activation functions for both FFN and CNN

(Not supported)

The different models performed quite similar in terms of validation accuracy, with the exception of Sigmoid. The Table 5.2 shows that all models achieved a similar validation accuracy with a variance of roughly 2%, except for Sigmoid which was around 10% below the others in terms of validation accuracy when fully trained. Table 5.4 also shows that according to the paired t-test the differences between most of the models were not large enough to be statistically significant. The exception was Sigmoid, where the difference were large enough to be statistically significant with a p-value of <0.001 in the performed t-test.

Model accuracy:

Leaky ReLU achieves higher validation accuracy than ReLU = **False**
Leaky ReLU achieves higher validation accuracy than Sigmoid = **True**
Leaky ReLU achieves higher validation accuracy than TanH = **False**

This hypothesis was thus not supported.

H4: Models with more neurons/filters per layer achieves higher validation accuracy *(Not supported)*

The only FFN model in the experiment that differed from the others in terms of numbers of neurons per layer was the model FFN Wide, which had twice the number of neurons in its hidden layers compared to its counterparts. Table 5.4 shows that the differences in validation accuracy between the FFN Wide and the other FFN models were not large enough to be statistically significant according to the performed t-tests, with a p-value of 0.001, with the exception of Sigmoid. Hence there was no support for this hypothesis.

Interesting observations

By looking at statistics of the confusion matrices Appendix C - Confusion matrices it's possible to discern that the highest accurate prediction for the FFN Deep model is the class 'ship' with 64.67% correct prediction. However both the base model and the Leaky ReLU model had a higher percentage of correct prediction with their prediction of ships with 66.52% and 68.1% respectively. This was just for the output class with the highest accurate prediction. In fact all of the FFN models had the output class 'ship' as their most accurately predicted, with the class 'automobile' as the second highest. All the models struggled the most with the images containing animals such as the output classes 'cat' and 'dog'. It should be noted that these patterns might say more about the dataset used than the actual models however.

Table 5.2 Training needed and validation accuracy for the FFN models

	Epochs needed to train the model	Validation accuracy
FFN Base	35	49.9%
FFN Deep	28	49.4%
FFN Wide	35	50.4%
FFN Sigmoid	98	36.6%
FFN TanH	84	48.8%
FFN Leaky ReLU	69	50.5%

Table 5.3 ***Training t-test*** *- The results were compared against the t-value of 4.781, meaning that if the result of the t-test was between -4.781 and 4.781 there were no statistical difference between two models (marked in white). If the result was higher or lower than the t-value it the model was deemed to be statistically different with a p-value of < 0.001 (orange if better, purple if worse).*

	Base	Deep	Wide	Sigmoid	TanH	Leaky
Base	nan	2.918	-0.768	-43.091	-22.734	-8.488
Deep	-2.918	nan	-3.407	-61.412	-20.347	-8.505
Wide	0.768	3.407	nan	-39.695	-21.318	-7.410
Sigmoid	43.091	61.412	39.695	nan	5.756	6.502
TanH	22.734	20.347	21.318	-5.756	nan	2.947
Leaky	8.488	8.505	7.410	-6.502	-2.947	nan

Table 5.4 ***Validation accuracy t-test*** *- The results were compared against the t-value of 4.781, meaning that if the result of the t-test was between -4.781 and 4.781 there were no statistical difference between two models (marked in white). If the result was higher or lower than the t-value it the model was deemed to be statistically different with a p-value of < 0.001 (orange if better, purple if worse).*

	Base	Deep	Wide	Sigmoid	TanH	Leaky
Base	nan	1.053	-1.532	38.582	2.891	-1.780
Deep	-1.053	nan	-2.150	60.236	1.906	-2.356
Wide	1.532	2.150	nan	41.146	3.815	-0.252
Sigmoid	-38.582	-60.236	-41.146	nan	-46.209	-38.912
TanH	-2.891	-1.906	-3.815	46.209	nan	-3.934
Leaky	1.780	2.356	0.252	38.912	3.934	nan

5.2.2 Convolutional Neural Network - Analysis (Objective 7)

In order to analyze the results and draw conclusions from the results, statistical two-tailed paired t-tests were performed to test the hypotheses of this experiment. Table 5.5 presents the number of epochs needed to train each CNN model as well as the achieved validation accuracy for these models. In Table 5.6 the result from the paired t-tests for the amount of epochs needed to train each model is presented. Table 5.7 on the other hand present the results from the t-test where the validation accuracy for different CNN models are presented.

H1: Deeper models are able to achieve higher validation accuracy for both FFN´s and CNN´s

(Not supported)

The hypothesis was that deeper models with more convolutional layers would yield an overall higher validation accuracy than the base case model. To calculate the validation accuracy for each model the accuracy for each 10 folds for that model were summed and divided by 10 to give a representative validation accuracy for each model.
This experiment showed that CNN Base yielded an overall validation accuracy of 64.4% and the model CNN Deep yielded a validation accuracy of 65.5%. However in the t-test where the validation accuracy for the CNN Base model was compared to the validation accuracy of the CNN Deep the results showed that with a p-value of 0.001, the difference in validation accuracy between the models were not large enough to be statistically significant. This t-test result can be seen in Table 5.7. Since this experiment showed that there was no statistical difference between the base case and the deeper model, the null hypothesis was accepted and thereby this hypothesis was not supported.

H2: ReLU and Leaky ReLU are able to train with fewer epochs compared to Sigmoid and TanH for both FFN's and CNN's

(Not supported)

The hypothesis was that ReLU and Leaky ReLU would be able to complete its training with fewer epochs compared to Sigmoid and TanH. This experiment showed that the model CNN Base which used ReLU was on average able to complete its training within 15 epochs, which can be seen in Table 5.5 below. The model CNN Leaky ReLU needed 17 epochs to complete its training on average. The model CNN Sigmoid needed 99 epochs in order to train on average and the model CNN TanH was done training after 19 epochs on average.
In order to test this hypothesis paired t-tests with a p-value of 0.001 was used were the amount of epochs needed to train the model was compared. The results from these t-tests can be seen in Table 5.6. The t-tests showed that ReLU was able to train with fewer epochs compared to Sigmoid, however there were no statistical difference between the number of epochs needed for ReLU and TanH. The t-tests also showed that Leaky ReLU needed fewer epochs to train compared to Sigmoid, however there was no statistical difference between the amount of epochs needed to train for Leaky ReLU compared to TanH.
Training epochs:

ReLU needs less training than Sigmoid = **True**
ReLU needs less training than TanH = **False**
Leaky ReLU needs less training than Sigmoid = **True**
Leaky ReLU needs less training than TanH = **False**

Since the above t-tests did not show that ReLU and Leaky ReLU used fewer epochs to train compared to Sigmoid and TanH, the null hypothesis was accepted, meaning that there was no support for this hypothesis.

H3: Leaky ReLU achieves the highest validation accuracy among all the activation functions for both FFN and CNN

(Not supported)

The hypothesis was that the activation function Leaky ReLU would be able to achieve a higher validation accuracy compared to the ReLU, Sigmoid and TanH. In this experiment the model CNN Base which used ReLU was able to achieve a validation accuracy of 64.4% which can be seen in Table 5.5. The model CNN Sigmoid was able to achieve a validation accuracy of 49.8%, the model CNN TanH achieved a validation accuracy of 67.1% and the model CNN Leaky ReLU was able to achieve a validation accuracy of 67.2%.

In order to test this hypothesis paired t-tests with a significance level of 0.001 were used where the validation accuracy for the models was compared. The results from these t-tests can be seen in Table 5.7. The results from the t-tests showed that the model CNN Leaky ReLU was able to achieve a higher validation accuracy than the model CNN Base that used ReLU. The t-tests also showed that the model CNN Leaky ReLU was able to achieve a higher validation accuracy compared to the model CNN Sigmoid. However the t-tests showed that there was no statistical difference for the achieved validation accuracy for the model CNN Leaky ReLU compared to the model CNN TanH.

Model accuracy:

Leaky ReLU achieves higher validation accuracy than ReLU = **True**

Leaky ReLU achieves higher validation accuracy than Sigmoid = **True**

Leaky ReLU achieves higher validation accuracy than TanH = **False**

Since the above t-tests did not show that Leaky ReLU was able to achieve a higher accuracy than the other models the null hypothesis was accepted, meaning that there was no support for this hypothesis.

H4: Models with more neurons/filters per layer achieves higher validation accuracy

(Supported)

- *CNN Wide achieved a higher validation accuracy compared to CNN Base*

The hypothesis was that more filters per convolutional layer would yield a higher validation accuracy compared to the base model used in this experiment. The model CNN Base was able to achieve a validation accuracy of 64.4% and the model CNN Wide achieved a validation accuracy of 66.1%.

In order to test this hypothesis paired t-tests with a significance level of 0.001 were used where the model CNN Base was compared to CNN Wide. The result from this t-test showed that the model CNN Wide was able achieve a higher validation accuracy compared to the model CNN Base. Since the t-test showed that there was a statistical difference where the model CNN Wide was able to achieve higher validation accuracy compared to the model CNN Base, the null hypothesis was rejected, and thereby this hypothesis was supported.

Interesting observations

When the graphs in Figure 5.7 to Figure 5.12 were analyzed the model CNN Sigmoid sticks out in more ways than one. Firstly the derivative for the validation loss never changes from negative to positive, which seem to indicate that CNN Sigmoid could get a lower validation loss if it was allowed to train for more epochs, which in turn seemed to indicate that the model was not fully trained after 100 epochs. The graphs for CNN Sigmoid also seemed to confirm the effect of the vanishing gradient problem, because for approximately the first 20 epochs the validation loss and training loss made little to no improvements. This could also be seen in the training and validation accuracy which made little to no improvement for the first few epochs. In other words it took a few epochs to get the Sigmoid activation function to start learning and thereby improving the performance of the model. It was however interesting to see that the vanishing gradient could not be observed in the model CNN TanH which was expected since they are both sigmoidal functions so it was expected that they would share this behavioral pattern.

It was also interesting to see that the model CNN Sigmoid trained slower than all the other CNN models and it also achieved the lowest validation accuracy compared to the other CNN models.

Table 5.5 Training needed and validation accuracy for the CCN models

	Epochs needed to train the model	Validation accuracy
CNN Base	15	64.4%
CNN Deep	14	65.5%
CNN Wide	15	66.1%
CNN Sigmoid	99	49.8%
CNN TanH	19	67.1%
CNN Leaky ReLU	17	67.2%

Table 5.6 ***Training t-test*** *- The results were compared against the t-value of 4.781, meaning that if the result of the t-test was between -4.781 and 4.781 there were no statistical difference between two models (marked in white). If the result was higher or lower than the t-value it the model was deemed to be statistically different with a p-value of < 0.001 (orange if better, purple if worse).*

	Base	Deep	Wide	Sigmoid	TanH	Leaky
Base	nan	2.025	1.561	-128.028	-4.417	-3.254
Deep	-2.025	nan	-1.627	-186.377	-5.403	-5.779
Wide	-1.561	1.627	nan	-212.979	-6.034	-5.449
Sigmoid	128.028	186.377	212.979	nan	144.102	150.970
TanH	4.417	5.403	6.034	-144.102	nan	2.043
Leaky	3.254	5.779	5.449	-150.970	-2.043	nan

Table 5.7 ***Validation accuracy t-test*** *- The results were compared against the t-value of 4.781, meaning that if the result of the t-test was between -4.781 and 4.781 there were no statistical difference between two models (marked in white). If the result was higher or lower than the t-value it the model was deemed to be statistically different with a p-value of < 0.001 (orange if better, purple if worse).*

	Base	Deep	Wide	Sigmoid	TanH	Leaky
Base	nan	-2.218	-5.410	45.318	-7.542	-8.156
Deep	2.218	nan	-1.983	58.939	-3.202	-3.932
Wide	5.410	1.983	nan	92.479	-3.341	-3.762
Sigmoid	-45.318	-58.939	-92.479	nan	-45.482	-46.989
TanH	7.542	3.202	3.341	45.482	nan	-0.384
Leaky	8.156	3.932	3.762	46.989	0.384	nan

5.2.3 FeedForward Network vs. Convolutional Neural Network - Analysis

In order to compare how the FFN models performed compared to the CNN models a series of paired t-tests were performed. For the interest of completeness all models were compared to all other models. This was done for the amount of training the models needed as well as the validation accuracy the models was able to achieve upon completing training. The results of the t-tests for the number of epochs needed to train a model can be seen in Table 5.8 below and the results for the validation accuracy t-tests can be seen in Table 5.9.

Table 5.8 ***Training t-test*** *- The results were compared against the t-value of 4.781, meaning that if the result of the t-test was between -4.781 and 4.781 there were no statistical difference between two models (marked in white). If the result was higher or lower than the t-value it the model was deemed to be statistically different with a p-value of < 0.001 (orange if better, purple if worse).*

	F Base	F Deep	F Wide	F Sigmoid	F TanH	F Leaky	C Base	C Deep	C Wide	C Sigmoid	C TanH	C Leaky	Epochs
F Base	nan	2.918	-0.768	-43.091	-22.734	-8.488	11.767	12.387	11.905	-40.567	8.870	9.412	35
F Deep	-2.918	nan	-3.407	-61.412	-20.347	-8.505	9.000	10.805	11.075	-65.422	7.169	7.448	28
F Wide	0.768	3.407	nan	-39.695	-21.318	-7.410	11.029	11.989	11.400	-37.650	8.561	9.355	35
F Sigmoid	43.091	61.412	39.695	nan	5.756	6.502	108.018	140.150	122.376	-1.809	98.409	96.600	98
F TanH	22.734	20.347	21.318	-5.756	nan	2.947	25.464	26.393	26.902	-6.194	30.046	25.109	84
F Leaky	8.488	8.505	7.410	-6.502	-2.947	nan	11.329	12.192	11.789	-6.415	10.334	10.617	69
C Base	-11.767	-9.000	-11.029	-108.018	-25.464	-11.329	nan	2.025	1.561	-128.028	-4.417	-3.254	15
C Deep	-12.387	-10.805	-11.989	-140.150	-26.393	-12.192	-2.025	nan	-1.627	-186.377	-5.403	-5.779	14
C Wide	-11.905	-11.075	-11.400	-122.376	-26.902	-11.789	-1.561	1.627	nan	-212.979	-6.034	-5.449	15
C Sigmoid	40.567	65.422	37.650	1.809	6.194	6.415	128.028	186.377	212.979	nan	144.102	150.970	99
C TanH	-8.870	-7.169	-8.561	-98.409	-30.046	-10.334	4.417	5.403	6.034	-144.102	nan	2.043	19
C Leaky	-9.412	-7.448	-9.355	-96.600	-25.109	-10.617	3.254	5.779	5.449	-150.970	-2.043	nan	17

Table 5.9 ***Validation accuracy t-test*** *- The results were compared against the t-value of 4.781, meaning that if the result of the t-test was between -4.781 and 4.781 there were no statistical difference between two models (marked in white). If the result was higher or lower than the t-value it the model was deemed to be statistically different with a p-value of < 0.001 (orange if better, purple if worse).*

	F Base	F Deep	F Wide	F Sigmoid	F TanH	F Leaky	C Base	C Deep	C Wide	C Sigmoid	C TanH	C Leaky	Val acc
F Base	nan	1.053	-1.532	38.582	2.891	-1.780	-27.577	-30.606	-40.741	0.207	-30.882	-28.381	49.9
F Deep	-1.053	nan	-2.150	60.236	1.906	-2.356	-33.350	-44.777	-59.638	-1.055	-41.519	-36.036	49.4
F Wide	1.532	2.150	nan	41.146	3.815	-0.252	-26.918	-28.840	-37.189	1.312	-28.070	-27.056	50.4
F Sigmoid	-38.582	-60.236	-41.146	nan	-46.209	-38.912	-80.886	-81.021	-143.147	-52.036	-100.634	-84.009	36.6
F TanH	-2.891	-1.906	-3.815	46.209	nan	-3.934	-35.966	-32.210	-53.216	-2.780	-54.725	-40.420	48.8
F Leaky	1.780	2.356	0.252	38.912	3.934	nan	-30.593	-46.676	-46.710	2.964	-32.919	-33.337	50.5
C Base	27.577	33.350	26.918	80.886	35.966	30.593	nan	-2.218	-5.410	45.318	-7.542	-8.156	64.4
C Deep	30.606	44.777	28.840	81.021	32.210	46.676	2.218	nan	-1.983	58.939	-3.202	-3.932	65.5
C Wide	40.741	59.638	37.189	143.147	53.216	46.710	5.410	1.983	nan	92.479	-3.341	-3.762	66.1
C Sigmoid	-0.207	1.055	-1.312	52.036	2.780	-2.964	-45.318	-58.939	-92.479	nan	-45.482	-46.989	49.8
C TanH	30.882	41.519	28.070	100.634	54.725	32.919	7.542	3.202	3.341	45.482	nan	-0.384	67.1
C Leaky	28.381	36.036	27.056	84.009	40.420	33.337	8.156	3.932	3.762	46.989	0.384	nan	67.2

H5: CNN is able to achieve an overall higher validation accuracy compared to the FFN counterparts

(supported)

- *CNN models achieved a higher validation accuracy compared to the FFN counterparts*

The hypothesis was that CNN´s would be able to achieve a higher validation accuracy once completely trained compared to the FFN counterparts. Table 5.10 below shows the validation accuracy achieved for each FFN and CNN models side by side as well as which architecture was able to achieve the highest validation accuracy and if there was any statistical difference between the two models using a significance level of 0.001. The results in Table 5.10 were derived from Table 5.9. In order to determine whether there was any statistical difference in favor of one of the architectures, the lower left hand quarter of Table 5.9 could be used and it showed that the CNN model was able to achieve higher accuracies than the FNN counterparts.

Since all CNN models were able to achieve a higher accuracy than their FNN counterparts the null hypothesis was rejected, meaning that there was statistical evidence that support this hypothesis.

Table 5.10 Comparison of validation accuracy for the FFN and CNN models

	FFN val acc	CNN val acc	Statistically best
Base	49.9%	64.4%	CNN
Deep	49.4%	65.5%	CNN
Wide	50.4%	66.1%	CNN
Sigmoid	36.6%	49.8%	CNN
TanH	48.8%	67.1%	CNN
Leaky	50.5%	67.2%	CNN

Interesting observations

When comparing whether there were any statistical difference in the amount of epochs of training needed between all the models, some interesting conclusions could be drawn. Looking at the lower left quarter of Table 5.8 it was clear that all CNN models except CNN Sigmoid trained faster than all FFN models. It was however interesting to see that there was no statistical difference in the amount of epochs of training needed between CNN Sigmoid and FFN Sigmoid. This was most likely due to the fact that these two models could still improve with more training, meaning that they were not trained to their full potential within 100 epochs.

When comparing whether there were any statistical difference in the validation accuracy achieved on the trained models using a significance level of 0.001, some interesting patterns could be seen in the data. Looking at the lower left quarter of Table 5.9, it was easy to see that there were statistical evidence that suggest that all CNN models except for CNN Sigmoid was able to achieve a higher validation accuracy compared to all FFN models. The validation accuracy achieved for CNN Sigmoid was however not statistically significant to suggest that there was any difference in validation accuracy compared to all FFN models except for FFN Sigmoid. It was hard to draw any conclusions from these similarities, because these FFN models are fully trained, CNN Sigmoid on the other hand was not fully trained and could potentially learn more with more training. There were however statistical evidence to suggest that CNN Sigmoid was able to achieve a higher validation accuracy than FFN Sigmoid.

5.3 Conclusion

This subchapter will summarize the conclusions that were drawn from the results of the experiment and this data was used to answer the research questions written in chapter 3.3. This subchapter will not present any new data and everything presented here could be inferred from earlier chapters, but a short section answering the research questions have been added for ease of reading.

RQ1: To what extent does the width and depth of a FFN affect the validation accuracy of the models used in the experiment?

The model FFN Base was able to achieve a validation accuracy of 49.9%, the model FFN Deep was able to achieve a validation accuracy of 49.4% and the model FFN Wide was able to achieve a validation accuracy of 50.4%. The difference between the models in terms of width and depth turned out to have a very small effect on the achieved validation accuracy in the experiment and the differences were not large enough to be statistically significant using a significance level of 0.001.

Deeper model: *no difference* compared to base model
Wider model: *no difference* compared to base model

RQ2: To what extent does the activation functions Sigmoid, TanH, ReLU and Leaky ReLU affect the validation accuracy of a FFN?

The model FFN Sigmoid was able to achieve a validation accuracy of 36.6%, the model FFN TanH was able to achieve a validation accuracy of 48.8%, the model FFN Base which used ReLU achieved a validation accuracy of 49.9% and the model FFN Leaky ReLU was able to achieve a validation accuracy of 50.5%.

The models using ReLU, Leaky ReLU and TanH did not achieve validation accuracies where the differences were large enough to be statistically significant using a significance level of 0.001. However when comparing the model FFN Sigmoid to every other activation function in the experiment, Sigmoid achieved a validation accuracy that was statistically lower than the other activation functions.

Sigmoid: *Lowest validation accuracy*
ReLU, TanH and Leaky ReLU: *Higher validation accuracy than Sigmoid*

RQ3: To what extent does the width and depth of a CNN affect the validation accuracy of the models used in the experiment?

The model CNN Base was able to achieve a validation accuracy of 64.4%, the model CNN Deep achieved a validation accuracy of 65.5% and the model CNN Wide was able to achieve a validation accuracy of 66.1%. The difference between the validation accuracy between CNN Base and CNN Deep was too small to be statistically significant using a significance level of 0.001. This meant that the results from this study could not be used to show that there was any difference in validation accuracy when using a deeper model compared to the base model used in this experiment.

When comparing the validation accuracy of the model CNN Base and CNN Wide the difference was large enough to be statistically significant using a significance level of 0.001 and it showed that CNN Wide was able to achieve a slightly higher validation accuracy compared to the base model.

Deeper model: *no difference* compared to base model
Wider model: *slightly higher validation accuracy* compared to base model

RQ4: To what extent does the activation function Sigmoid, TanH, ReLU and Leaky ReLU affect the validation accuracy of a CNN?

The model CNN Sigmoid was able to achieve a validation accuracy of 49.8%, the model CNN TanH achieved a validation accuracy of 67.1%, the model CNN Base which used ReLU was able to achieve a validation accuracy of 64.4% and the model CNN Leaky ReLU was able to achieve a validation accuracy of 67.2%.

The results from this study indicated that the activation function Sigmoid achieved the overall lowest validation accuracy among the different activation functions used on the CNN models in this experiment.
The result also indicated that the activation function ReLU was able to achieve a significantly higher validation accuracy compared to the activation function Sigmoid, while achieving a slightly lower validation accuracy compared to TanH and Leaky ReLU.
The result from this study also indicated that the activation function TanH and Leaky ReLU was able to achieve a higher validation accuracy compared to Sigmoid and ReLU, but there were no statistical difference in the validation accuracies achieved by TanH and Leaky ReLU using a significance level of 0.001.

Sigmoid: *Lowest validation accuracy*
ReLU: *Second lowest validation accuracy*
TanH and Leaky ReLU: *Highest validation accuracy*

RQ5: Which of the activation functions is able to learn with the fewest epochs on the models used in the experiment?

When comparing the amount of training epochs needed for the FFN Models, the results indicated that the activation function ReLU required less training compared to Sigmoid, TanH and Leaky ReLU. The activation function Sigmoid required more training than the other activation functions for all FFN models, and there was no statistical difference in the amount of training needed for TanH compared to Leaky ReLU for the FFN models using a significance level of 0.001.

When comparing the amount of training epochs needed for the CNN models, the results indicated that the activation function Sigmoid, needed significantly more training compared to the other activation functions for the CNN models. The results also indicated that there was no statistical difference in the amount of training needed for ReLU, TanH and Leaky ReLU using a significance level of 0.001.

FFN:

ReLU: *fewest epochs* needed
TanH & Leaky ReLU: *more training than ReLU* needed
Sigmoid: *most epochs* needed

CNN:

ReLU, TanH & Leaky ReLU: *fewest epochs* needed
Sigmoid: *most epochs* needed

Since ReLU was trained fastest for both the FFN models and CNN models, the overall fastest learner among the activation functions was ReLU.

Overall fastest learner: *ReLU*

RQ6: Which architectural models used in this study is able to achieve the highest validation accuracy on the trained models FFN or CNN?

The CNN models were able to achieve higher validation accuracy than their FFN counterparts, which was discussed in chapter 5.2.3 FeedForward Network vs. Convolutional Neural Network - Analysis.

CNN higher accuracy than FNN

6 Discussion

This chapter will contain a summary of the findings and the study as a whole. It will also contain some conclusions in terms of related work, the validity of the study and some ethical aspects of the study and finally some thoughts on potential future work.

6.1 Summary

In this thesis several models of two different types of neural networks, FFN and CNN, were built and tested against each other to see if there were any statistical difference in their performance when performing image classification. These models also varied in terms of their depth, width and which activation functions they used. These models were all trained and tested on the dataset CIFAR-10 using 10 fold cross validation, and then compared against each other. By doing this the hope was to not only to compare FFN and CNN against each other but also if the different architectural styles and activation functions had any impact on the validation accuracy. In order to see if there were any statistical difference in the results, several two tailed paired t-tests were performed.

The results of the experiment showed that the CNN's achieved a higher validation accuracy than the FFN's in every comparable variation of the models. When comparing the variations in terms of width and depth, only one of the models achieved a higher validation accuracy, CNN Wide. In the other models the depth and width of the networks did not have any statistically significant difference in validation accuracy.

In the variation of the models one of the changing variables were the activation functions used to learn. In this thesis project four different activation functions were tested: ReLU, Leaky ReLU, TanH and Sigmoid. The results showed that ReLU was the overall fastest learner for both FFN and CNN. In terms of the achieved validation accuracy the Sigmoid function performed the worst for both FFN and CNN. For the FFN´s there was no statistically significant difference between the other activation functions, and for CNN the activation function TanH and Leaky ReLU achieved the highest validation accuracy.

After the experiment had been conducted and the results had been evaluated, our recommendation would be when performing image classification, to use CNN´s and to use the activation function ReLU. However we do not have any concrete advice for how to construct the CNN´s in forms of width and depth of the network, but from our findings it would seem that more convolutional filters per convolution layer improves the accuracy slightly.

6.2 Comparison to previous work

The second chapter in this report covers background and it contains related work, terms and concepts that have been used throughout the study. There are however two papers that has influenced the study more than others. These paper are *Implementation of Deep Feedforward Neural Network with CUDA backend for Efficient and Accurate Natural Image Classification* (2017) by von Hacht and *Gradient Based Learning Applied to Document Recognition* (1998) by Lecun et al.

These two papers were the inspiration and used as a basis for the models that was used as base cases for the FFNs and CNNs used in this study. The task of creating a FFN and a CNN that are directly comparable are a challenging task. However, a good way to setup an experiment containing a FFN or a CNN, is to mimic the design of previous studies. The thought process behind the experiment design can be found in chapter 4 Implementation.

Lecun et al. in *Gradient Based Learning Applied to Document Recognition* (1998) found an improvement in validation accuracy for their FFN's. The FFNs used in this thesis project did not achieve differences in validation accuracy of the evaluated networks that were large enough to be statistically significant. The exception was the usage of Sigmoid as activation function which achieved a statistically lower validation accuracy compared to the other models. Furthermore the FFN's Lecun et al. used all achieved a higher validation accuracy than the FFNs used in this thesis project, however the datasets used in these studies were different. Lecun et al. achieved a validation accuracy between 95.3% and 97.5% on their FFN's while our FFN's achieved approximately between 35% and 50% in validation accuracy. This was most likely due to the differences in datasets and the modifications that were made to the layer architecture in order to accommodate these more complex images. After seeing these results it would be interesting to test our networks using the MNIST dataset to see if our FFN's would be able to achieve the same validation accuracy as Lecun et al. In a similar fashion our CNN´s were not able to achieve a validation accuracy as high as LeNet-5, but it did perform better than the FFN's used in our study.

The results found in this thesis project also showed that the CNN models was able to achieve an overall higher validation accuracy than their FFN counterparts. The same correlation between FFN and CNN models could be found in the paper *Gradient Based Learning Applied to Document Recognition* (1998) by Lecun et al. This seemed to indicate that it was in fact the choice of model architecture, CNN or FFN, that influenced the increased validation accuracy of the models. It also seemed to indicate that this correlation was not closely tied to the dataset MNIST used by Lecun et al. (1998)

In the paper *Implementation of Deep Feedforward Neural Network with CUDA backend for Efficient and Accurate Natural Image Classification*, von Hacht (2017) created three CNN models for the CIFAR-10 dataset and by increasing the number of filters in the convolutional layers by a factor of 3, the validation accuracy went from 70.39% to 72.64%. Von Hacht also found that both increasing the number of filters and the number of convolutional layers before max pooling in the same model further increased the validation accuracy to 72.88%. A similar pattern could be seen in this thesis project, where the CNN model with twice as many filters per convolution layer increased the validation accuracy of the model from 64.4% to 66.1%. This seemed to indicate that increasing the number of filters in the convolution layers led to a slight increase in the achieved validation accuracy.

In the paper *Deep Residual Learning for Image Recognition* (2016) He et al. found that models trained and evaluated on the CIFAR-10 dataset with fewer convolution layers were able to achieve a higher validation accuracy than their deeper counterpart. The model with the fewest amount of layers had 19 layers of convolution and by increasing the number of convolutions to 31, the validation accuracy decreased from approximately 91% to approximately 90%. He et al. (2016) created even deeper models, but the deeper models became the worse they performed. These findings are different from the findings in this thesis project, because when comparing the effect of adding a layer of convolution to the models used in this thesis project no difference in validation accuracy could be observed and the models CNN Base and CNN Deep achieved approximately 65% validation accuracy. It is however worth noting that the CNN models used in this thesis project used 2 and 3 layers of convolutions respectively, and there is quite a large difference between 2 and 3 layers of convolution compared to 19 and 31 layers of convolution. It might be the case that adding more layers of convolutions to the models used in this thesis would have increased the validation accuracy.

6.3 Validity

Several decisions and actions was taken during the course of the thesis project to improve the overall validity, however some potential threats still remain.

In terms of conclusion validity the reliability of measures was an issue. An effort was made to design the experiment in such a way that each model would be trained under the same circumstances. The code architecture was set up in such a way that the models was trained and evaluated using the same files and the models themselves followed a shared template. The reliability of treatment implementation was also considered. Since there existed several random elements when creating a neural network, the experiment was set up in such a way that these randoms would all have an identical seed. This was done in the hope of creating replicable results. The models were also trained using the same dataset and with the same k-fold cross validation.

Internal validity also had some aspects that needed to be considered. In terms of instrumentation the dataset used in the study, CIFAR-10, could be brought into question. All that was known about this dataset was the number of images, the resolution, the types of output classes and the equal partitioning of these classes. The potential poor quality of this dataset could be classified as a threat to the validity of this study. Another factor that could potentially be an instrumentation threat is the code used to build the neural networks. Due to the lack of experience and knowledge of the researchers the code could potentially contain errors or suboptimal solutions.

Some considerations were also made regarding construction validity. To combat inadequate preoperational explication of constructs some guidelines were determined for when a model was finished training and which measurement was interesting. The choice was to select the validation accuracy from the point of training where the validation loss was the lowest, since this was when the model generalized each category of images as much as possible. A potential threat to the validity was mono operational bias, since the experiment was only run once and only with one dataset. The reason this was done however was to somewhat combat experimental expectancies. By only running the experiment once with one dataset, there was only one result to choose from and no potential threat that a personal bias affected the choice of results.

A potential threat to the external validity was interaction of history and treatment. To counteract this potential threat all the training was performed on the same hardware under the same circumstances, for example the computer was disconnected from the internet to prevent potential disturbances. In terms of the threat, interaction of setting and treatment, all the tests were used using the same code base written using the Keras API. However if this could be considered to mimic real world application and standards cannot be guaranteed, due to the limited knowledge and experience of the researchers.

6.4 Ethics

Whenever machine learning is discussed there are some ethical aspects that should be considered.

If a task can be automated, meaning a machine can perform said task with adequate results, a one-time expense for a machine could be more attractive to a business compared to human personnel that require salary, healthcare and similar expenses over the course of time. By making humans obsolete in such a way could potentially cause many to lose their employment. Self-driving cars could long term mean that long distance transportation would no longer require human chauffeurs. Machine learning could potentially result in a shift similar to the early industrialization when factory workers were replaced by machines. Society has adapted since, and the machines and people coexist in society, but a shift may still be imminent.

Another ethical aspect to consider is the potential malicious use of technology. Technology such as image- and face recognition can be very effectively utilized in surveillance. However could the very same technology be used for illegal surveillance or gathering of data.

6.5 Future Work

The scope of this study was limited both by time constraints and the knowledge and competence of the researchers. If given additional time there are ways the study could potentially be improved. There are also the potential to base future studies on the results found here.
The task of creating a FFN and a CNN which are comparable is very challenging, and perhaps it's not possible to create base cases of these types of networks that are entirely equal. If given more time, this topic could be researched further in an attempt to further strengthen the validity of the study. There might also exist a task that is more suited for comparing two networks other than image classification.

Even without radically changing the experimental design there is still room for development. Given the current code and model structure it might be of interest to change the dataset from CIFAR-10 to another dataset to see if trends and behaviors persist or if the current results are closely tied to CIFAR-10. There are also a lot of variables and parameters within the current experiment that could be tweaked, such as number of layers, activation functions used, epochs trained etc.

Bibliography

Box, G., Hunter, J., & Hunter, W. (2012). *Praktisk statistik och försöksplanering*. Lund: Studentlitteratur.

Cs231n.github.io. (2019). *CS231n Convolutional Neural Networks for Visual Recognition*. [online] Available at: http://cs231n.github.io/optimization-2/ [Accessed 18 Feb. 2019].

CS231n Winter 2016: Lecture1: Introduction and Historical Context. (2016). Retrieved from https://www.youtube.com/watch?v=NfnWJUyUJYU&list=PLkt2uSq6rBVctENoVBg1TpCC7OQi31AlC

Chauvin, Y. and Rumelhart, D. (2009). *Back propagation*. New York: Psychology Press.

Demuth, H., & Beale, M. (2001). *Neural network toolbox*. Natick, Mass: MathWork.

Domingos, Pedro M. "A few useful things to know about machine learning." *Commun. Acm* 55.10 (2012): 78-87.

Greenspan, H., van Ginneken, B. and Summers, R. (2016). Guest Editorial Deep Learning in Medical Imaging: Overview and Future Promise of an Exciting New Technique. *IEEE Transactions on Medical Imaging*, 35(5), pp.1153-1159.

Von Hacht, A. (2017) "*Implementation of Deep Feedforward Neural Network with CUDA backend for Efficient and Accurate Natural Image Classification",* Gothenburg, Sweden, University of Chalmers

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770-778).

Herman, E. and Strang, G. (2016). *Calculus*. OpenStax

Home - Keras Documentation. (2019). Retrieved from https://keras.io/

Kong, S., & Takatsuka, M. (2017, May). Hexpo: A vanishing-proof activation function. In *2017 International Joint Conference on Neural Networks (IJCNN)* (pp. 2562-2567). IEEE.

Krizhevsky, A. CIFAR-10 and CIFAR-100 datasets. Retrieved from https://www.cs.toronto.edu/~kriz/cifar.html

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (pp. 1097-1105).

Lecun, Y., Bottou, L., Bengio, Y. and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), pp.2278-2324.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton. "Deep learning." *nature* 521.7553 (2015): 436.

Maas, A. L., Hannun, A. Y., & Ng, A. Y. (2013, June). Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml* (Vol. 30, No. 1, p. 3).

Nair, V., & Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)* (pp. 807-814).

Neural Networks. (2019). Retrieved from http://ufldl.stanford.edu/wiki/index.php/Neural_Networks [Accessed 4 April. 2019].

Russell, S. and Norvig, P. (2010). *Artificial intelligence A Modern Approach*. 3rd ed.

Strang, G. Retrieved from https://ocw.mit.edu/ans7870/resources/Strang/Edited/Calculus/Calculus.pdf [Accessed 6 June. 2019].

Wohin, Claes et al. (2012). *Experimentation in software engineering*. Springer Scinece & Buisness Media.

Appendix A - Derivation rules

Logical expression	Derivative
$f(x) = \frac{1}{x}$	$\frac{df}{dx} = -\frac{1}{x^2}$
$f_c(x) = c + x$	$\frac{df}{dx} = 1$
$f(x) = e^x$	$\frac{df}{dx} = e^x$
$f_a(x) = ax$	$\frac{df}{dx} = a$

Appendix B - Work distribution

Objective	Contributor
1	Linus & Magnus
2	Linus & Magnus
3	Magnus
4	Linus
5	Linus & Magnus
6	Magnus
7	Linus

Appendix C - Confusion matrices

FFN BASE

	P Airplane	P Automobile	P Bird	P Cat	P Deer	P Dog	P Frog	P Horse	P Ship	P Truck
Airplane	54.37	3.13	9.45	1.63	5.0	1.68	3.25	3.32	12.62	5.55
Automobile	3.25	62.6	1.65	1.98	1.35	1.22	2.85	2.02	7.0	16.08
Bird	8.13	2.42	41.67	6.87	12.12	8.37	8.63	6.4	2.67	2.73
Cat	3.35	2.4	11.57	28.73	7.32	18.4	13.85	5.97	3.4	5.02
Deer	7.5	1.68	16.38	4.62	40.87	6.07	6.98	8.77	4.58	2.55
Dog	2.48	1.75	13.52	18.28	7.9	33.08	10.1	7.65	2.07	3.17
Frog	2.75	2.57	9.07	8.83	6.93	7.35	52.62	3.28	2.83	3.77
Horse	3.75	1.38	6.58	4.67	8.07	6.58	2.93	60.38	1.22	4.43
Ship	8.98	6.08	2.55	1.83	3.47	1.27	2.07	1.45	66.52	5.78
Truck	4.1	14.72	1.77	2.87	1.62	2.4	3.0	4.63	6.93	57.97

FFN DEEP

	P Airplane	P Automobile	P Bird	P Cat	P Deer	P Dog	P Frog	P Horse	P Ship	P Truck
Airplane	53.25	3.72	10.12	1.88	6.85	1.0	2.98	3.08	11.45	5.67
Automobile	3.18	64.58	1.62	2.2	1.65	0.9	2.92	2.0	6.63	14.32
Bird	8.6	2.43	39.72	7.17	16.25	5.92	8.5	6.25	2.5	2.67
Cat	3.78	2.27	11.32	30.13	9.35	14.18	15.3	5.38	3.6	4.68
Deer	7.42	1.57	15.33	5.12	45.57	3.53	6.77	8.18	4.23	2.28
Dog	2.52	1.72	13.18	19.12	10.35	29.48	11.15	6.95	2.68	2.85
Frog	3.25	2.63	8.72	9.23	8.0	5.0	53.9	2.78	2.82	3.67
Horse	3.4	1.42	7.23	5.25	10.52	5.45	3.3	57.5	1.5	4.43
Ship	10.28	6.38	2.15	2.05	3.9	1.03	2.37	1.0	64.67	6.17
Truck	4.4	17.22	1.82	3.58	2.02	1.98	3.32	3.72	6.3	55.65

FFN WIDE

	P Airplane	P Automobile	P Bird	P Cat	P Deer	P Dog	P Frog	P Horse	P Ship	P Truck
Airplane	56.55	3.67	8.05	1.52	4.75	1.68	3.8	3.23	11.42	5.33
Automobile	3.47	63.6	1.88	1.6	1.13	1.32	3.08	1.85	6.18	15.88
Bird	8.87	2.6	41.2	6.67	10.42	8.68	10.23	6.23	2.27	2.83
Cat	3.53	2.6	11.77	28.85	6.3	18.95	14.97	5.38	3.1	4.55
Deer	8.4	1.95	16.13	5.03	38.03	6.63	8.68	8.87	3.75	2.52
Dog	2.37	1.67	13.03	17.42	6.85	35.02	11.1	7.28	2.0	3.27
Frog	2.95	2.57	8.93	8.62	6.0	5.65	56.23	3.22	2.35	3.48
Horse	3.57	1.47	6.7	5.15	7.75	6.97	3.27	59.8	1.07	4.27
Ship	9.67	6.38	2.42	2.0	3.5	1.42	2.73	0.92	65.0	5.97
Truck	4.15	15.27	1.7	3.12	1.53	2.12	3.5	3.73	5.67	59.22

FFN SIGMOID

	P Airplane	P Automobile	P Bird	P Cat	P Deer	P Dog	P Frog	P Horse	P Ship	P Truck
Airplane	36.92	4.58	13.62	2.93	6.98	1.33	3.82	5.07	17.6	7.15
Automobile	4.35	50.43	1.75	3.58	2.35	2.62	5.12	5.03	8.63	16.13
Bird	13.45	4.73	33.72	6.83	9.4	3.87	13.13	7.25	4.75	2.87
Cat	5.97	6.48	12.03	22.82	7.85	8.8	17.65	6.43	5.37	6.6
Deer	14.5	3.6	17.78	5.05	25.73	3.93	10.83	9.72	5.02	3.83
Dog	5.28	5.95	16.83	15.7	9.28	15.67	15.15	7.6	4.8	3.73
Frog	5.2	6.6	11.08	11.68	6.98	4.98	38.3	5.22	5.68	4.27
Horse	6.42	4.5	7.58	5.98	11.12	3.8	6.43	43.47	3.45	7.25
Ship	17.22	6.93	2.8	2.87	2.85	1.85	2.43	1.77	52.02	9.27
Truck	6.22	19.13	1.6	2.63	1.73	1.57	4.57	6.02	9.7	46.83

FFN TanH

	P Airplane	P Automobile	P Bird	P Cat	P Deer	P Dog	P Frog	P Horse	P Ship	P Truck
Airplane	57.03	4.03	7.7	1.78	3.73	2.13	3.55	3.98	11.22	4.83
Automobile	4.13	62.77	1.62	1.73	1.3	1.8	3.07	2.68	6.28	14.62
Bird	9.97	3.2	38.15	6.63	11.12	9.18	9.67	6.75	2.95	2.38
Cat	4.1	2.87	10.1	26.52	5.93	21.3	14.43	5.88	3.95	4.92
Deer	8.8	2.12	15.0	5.2	37.0	7.55	7.63	9.55	4.75	2.4
Dog	3.43	2.38	11.62	15.57	6.77	36.95	10.27	7.37	2.77	2.88
Frog	3.83	2.87	7.98	9.05	5.47	8.87	51.93	3.55	3.22	3.23
Horse	4.37	2.1	6.32	4.97	7.35	7.95	3.28	57.65	1.63	4.38
Ship	11.92	6.03	2.08	1.9	3.35	1.93	1.83	1.75	63.52	5.68
Truck	4.88	15.53	1.53	3.02	1.58	3.17	3.5	4.68	5.97	56.13

FFN Leaky ReLU

	P Airplane	P Automobile	P Bird	P Cat	P Deer	P Dog	P Frog	P Horse	P Ship	P Truck
Airplane	53.35	3.55	9.7	2.08	4.67	1.57	3.88	3.28	13.28	4.63
Automobile	2.92	64.07	2.1	2.27	1.75	1.25	3.05	2.02	7.25	13.33
Bird	8.37	2.37	40.28	7.73	12.4	7.48	9.58	6.45	3.32	2.02
Cat	3.28	2.45	9.93	32.73	7.13	16.67	14.72	5.72	3.6	3.77
Deer	7.28	1.7	14.38	6.33	42.22	5.32	7.7	8.55	4.67	1.85
Dog	2.42	1.93	11.7	20.48	7.95	33.35	10.25	6.77	2.62	2.53
Frog	2.73	2.58	8.4	10.4	6.02	5.57	55.38	3.15	3.23	2.53
Horse	3.33	1.5	6.2	6.05	8.62	6.28	3.05	59.57	1.42	3.98
Ship	8.35	6.05	2.78	2.08	3.43	1.23	2.1	1.2	68.1	4.67
Truck	3.7	15.4	1.83	3.75	2.03	2.28	3.97	4.45	7.02	55.57

CNN BASE

	P Airplane	P Automobile	P Bird	P Cat	P Deer	P Dog	P Frog	P Horse	P Ship	P Truck
Airplane	71.93	1.78	5.52	1.72	2.65	1.17	1.25	1.33	9.03	3.62
Automobile	2.63	75.6	1.15	1.42	0.8	0.68	1.67	0.58	5.7	9.77
Bird	8.57	1.32	49.98	7.73	10.65	7.38	6.83	3.63	2.18	1.72
Cat	2.87	1.6	8.28	44.25	7.72	16.8	9.02	4.28	2.32	2.87
Deer	4.87	0.82	9.13	6.63	57.38	4.77	5.28	7.4	2.45	1.27
Dog	1.88	0.95	7.08	17.68	6.35	51.78	5.08	6.3	1.27	1.62
Frog	1.58	1.97	4.57	6.38	4.72	3.53	73.05	0.78	1.88	1.53
Horse	2.05	0.62	3.8	4.65	7.92	6.55	1.32	70.2	0.53	2.37
Ship	7.17	3.78	1.27	1.1	1.17	0.53	0.73	0.33	80.65	3.27
Truck	4.25	11.47	1.28	2.45	1.05	1.52	1.7	1.53	5.32	69.43

CNN DEEP

	P Airplane	P Automobile	P Bird	P Cat	P Deer	P Dog	P Frog	P Horse	P Ship	P Truck
Airplane	69.95	1.78	5.38	1.97	4.4	1.35	0.88	1.1	9.13	4.05
Automobile	1.85	77.27	0.82	1.12	0.7	0.8	1.22	0.65	5.05	10.53
Bird	8.0	1.23	51.43	6.98	12.23	8.25	4.2	4.08	2.25	1.33
Cat	2.55	1.33	6.92	44.13	9.35	20.5	6.05	4.5	2.23	2.43
Deer	3.98	0.7	7.62	5.88	62.67	6.18	2.67	7.65	1.63	1.02
Dog	1.27	0.8	5.9	15.48	7.4	57.23	2.85	6.33	1.17	1.57
Frog	1.37	1.95	4.38	7.83	6.32	5.28	67.58	1.4	1.93	1.95
Horse	1.23	0.43	3.22	4.08	8.42	6.97	0.73	72.73	0.38	1.8
Ship	5.92	3.93	1.55	1.32	1.75	0.65	0.53	0.32	80.6	3.43
Truck	3.2	10.87	1.27	2.55	1.57	1.78	1.03	1.93	4.5	71.3

CNN WIDE

	P Airplane	P Automobile	P Bird	P Cat	P Deer	P Dog	P Frog	P Horse	P Ship	P Truck
Airplane	69.3	1.85	6.83	2.37	2.77	1.12	1.1	0.97	10.45	3.25
Automobile	2.35	78.1	1.58	1.17	0.85	0.75	1.2	0.45	5.32	8.23
Bird	6.0	1.23	57.87	5.85	8.83	8.8	4.9	2.97	2.27	1.28
Cat	2.15	1.3	9.82	43.85	7.03	22.07	6.23	3.18	2.3	2.07
Deer	3.52	0.63	10.9	6.28	59.43	6.53	3.55	6.07	2.13	0.95
Dog	1.13	0.75	8.33	14.6	5.92	58.5	3.67	4.77	1.27	1.07
Frog	1.28	1.72	6.15	6.93	4.08	5.17	70.78	0.73	1.77	1.38
Horse	1.5	0.45	4.78	4.33	7.4	7.72	0.85	70.65	0.67	1.65
Ship	5.32	3.52	1.65	1.13	1.22	0.52	0.53	0.32	83.28	2.52
Truck	3.37	11.95	1.82	2.42	1.05	1.98	1.37	1.52	5.48	69.05

CNN SIGMOID

	P Airplane	P Automobile	P Bird	P Cat	P Deer	P Dog	P Frog	P Horse	P Ship	P Truck
Airplane	52.63	3.33	9.7	1.63	4.57	1.38	3.33	3.78	13.8	5.83
Automobile	2.38	61.63	0.93	1.38	1.38	1.0	4.52	2.37	7.23	17.17
Bird	8.03	1.97	40.35	7.15	10.65	8.73	11.12	7.13	2.13	2.73
Cat	2.43	2.43	11.82	28.72	6.6	17.97	15.77	6.62	2.05	5.6
Deer	7.38	2.07	17.9	4.9	36.13	5.18	9.93	10.78	3.32	2.4
Dog	1.65	1.6	13.32	16.3	6.15	37.33	9.95	9.15	1.37	3.18
Frog	1.93	3.1	7.7	7.35	5.17	4.93	60.92	3.1	1.63	4.17
Horse	2.62	1.25	6.37	4.48	8.07	7.93	3.75	60.15	0.8	4.58
Ship	9.92	7.97	3.2	1.28	2.78	0.92	2.65	1.25	62.65	7.38
Truck	3.88	16.82	1.83	2.55	1.03	1.83	3.98	3.77	6.85	57.45

CNN TanH

	P Airplane	P Automobile	P Bird	P Cat	P Deer	P Dog	P Frog	P Horse	P Ship	P Truck
Airplane	71.08	2.08	5.33	2.02	3.03	1.08	1.4	1.23	8.4	4.33
Automobile	2.08	77.98	1.02	1.05	0.75	0.77	1.42	0.77	3.48	10.68
Bird	7.08	1.08	53.65	6.67	9.9	7.18	6.47	4.72	1.72	1.53
Cat	2.52	1.22	6.87	46.83	6.92	17.98	8.63	4.75	2.02	2.27
Deer	3.17	0.75	8.23	5.95	61.1	4.72	4.93	8.35	1.82	0.98
Dog	1.23	0.73	6.25	16.55	6.78	55.57	4.33	6.55	0.83	1.17
Frog	1.27	1.2	4.9	6.88	4.08	3.57	74.67	1.1	1.27	1.07
Horse	0.9	0.48	2.98	4.28	6.83	5.47	0.95	75.67	0.48	1.95
Ship	5.9	3.7	1.27	1.23	1.27	0.55	1.02	0.67	81.23	3.17
Truck	3.22	10.37	1.23	1.88	1.32	1.32	1.17	2.23	4.02	73.25

CNN Leaky ReLU

	P Airplane	P Automobile	P Bird	P Cat	P Deer	P Dog	P Frog	P Horse	P Ship	P Truck
Airplane	71.17	1.93	5.13	2.42	2.53	1.15	1.47	1.42	8.53	4.25
Automobile	2.17	77.48	1.0	1.17	0.6	0.62	1.75	0.68	4.0	10.53
Bird	7.92	1.25	54.02	7.55	8.57	6.42	6.42	4.5	1.75	1.62
Cat	2.58	1.25	6.78	49.95	5.78	15.05	8.83	4.97	1.92	2.88
Deer	4.02	0.8	7.73	7.55	58.43	3.75	4.8	9.57	1.97	1.38
Dog	1.42	0.73	6.55	18.5	5.15	53.28	4.47	7.23	1.05	1.62
Frog	1.38	1.6	4.13	7.08	3.25	2.73	75.5	1.45	1.35	1.52
Horse	1.27	0.42	3.0	4.77	6.5	5.1	1.32	75.27	0.43	1.93
Ship	6.38	3.43	1.12	1.28	1.13	0.55	1.08	0.57	81.2	3.25
Truck	3.25	9.02	1.25	2.22	0.97	0.98	1.37	1.98	3.4	75.57

Appendix D - Source Code

In this appendix the python-source code used in the experiment will be presented. The code will be displayed one file at a time. These files are also publicly available at https://github.com/a15magknf16linli/A-COMPARATIVE-STUDY-OF-FFN-AND-CNN-WITHIN-IMAGE-RECOGNITION

In those instances when the line of code has been too long an arrow will be used to indicate that the following code is a continuation of the line above.

Source Files:

Training.py

```
#Enum for choosing behavior
from enum import Enum
class model_type(Enum):
    FFN_base = 0
    FFN_deep = 1
    FFN_wide = 2
    FFN_Sigmoid = 3
    FFN_TanH = 4
    FFN_Leaky_ReLU = 5
    CNN_base = 6
    CNN_deep = 7
    CNN_wide = 8
    CNN_Sigmoid = 9
    CNN_TanH = 10
    CNN_Leaky_ReLU = 11

############################
# parameters for the model #
############################
input_x = 32
input_y = 32
color_depth = 3
batch_size = 32
epochs = 100

##########################################
# parameters for training and choosing model #
##########################################
k_fold_split = 10
model_architecture = model_type.FFN_Leaky_ReLU

#support libraries
import numpy as np
import tensorflow as tf
from sklearn.model_selection import StratifiedKFold
import pickle

#import models
import sys
sys.path.insert(0, '../ModelTemplates')
import FFNBaseModel
import FFNDeepModel
import FFNWideModel
import FFNLeakyReLuModel
import FFNSigmoidModel
import FFNTanHModel
import CNNBaseModel
import CNNDeepModel
import CNNWideModel
import CNNSigmoidModel
import CNNTanHModel
import CNNLeakyReLUModel

#seed random
```

```
import random as rn
random_seed = 123
rn.seed(random_seed)
np.random.seed(random_seed)
tf.set_random_seed(random_seed)

def mergeDataSet():
    image_data_set = np.concatenate((x_train, x_test))
    image_labels = np.concatenate((y_train, y_test))

    return (image_data_set, image_labels)

def createFilePath(epoch, folder, k, fileEnding):
    #Create the filename.<model_architecture>_epoch(<epoch>)k(<k>)seed(<seed>).pkl
    filePath = '../' + str(folder) + '/'
    filePath += str(model_architecture).split('.')[1]
    filePath += '_epoch(' + str(epoch) + ')'
    filePath += 'k(' + str(k) + ')'
    filePath += 'seed(' + str(random_seed) + ')'
    filePath += '.' + str(fileEnding)

    return filePath

def kFoldCrossValidation(model_architecture):
    kFolds = StratifiedKFold(k_fold_split)
    print('### ' + str(k_fold_split) + ' fold crossvalidation')
    print('### ' + str(model_architecture))
    print('### epochs ' + str(epochs))
    print('### batch size ' + str(batch_size) + '\n')

    #create new model
    model = createModel(model_architecture)(random_seed, input_x, input_y, color_depth)
    filePath = createFilePath('untrained', 'models', k_fold_split, 'HDF5')
    model.save(filePath)
    print('\n### untrained model saved\n')
    del model

    model_performance = []
    k = 0
    #K fold cross validation
    for train_index, test_index in kFolds.split(X, Y):
        x_train = X[train_index]
        y_train = Y[train_index]
        x_test = X[test_index]
        y_test = Y[test_index]

        #load the untrained model
        model = tf.keras.models.load_model(filePath)
        print('### untrained model loaded')
        print('### Fold = ' + str(k) + ' ###\n')

        #save only the best performing model after each epoch
        bestModelFilePath = createFilePath(epochs, 'models', k, 'HDF5')
        callbacks = [tf.keras.callbacks.ModelCheckpoint(filepath = bestModelFilePath,
                                                        monitor = 'val_loss',
                                                        verbose= 1,
                                                        save_best_only = True,
                                                        period = 1)]

        #train the model
        history = model.fit(x = x_train,
                  y = y_train,
                  validation_data = (x_test, y_test),
                  epochs = epochs,
                  batch_size = batch_size,
                  callbacks = callbacks)

        #load the best trained model
        del model
        model = tf.keras.models.load_model(bestModelFilePath)

        #find the predicted class for each image in the test set
        predictions = model.predict(X[test_index])
        predictions = np.argmax(predictions, axis = 1)
```

```
        #find the actual class in the test set and reshape it to the same format as
        # predictions
        actual = Y[test_index].reshape(Y[test_index].shape[0],)

        #save the design of the model as a string
        modelSummary = []
        model.summary(print_fn = lambda x : modelSummary.append(x))
        modelSummary = "\n".join(modelSummary)

        #store the training history, prediction of the test set on trained model and actual
        # class of the test set, and the design of the model
        model_performance.append({'history' : history.history, 'predictions' : predictions,
    'actual' :     actual, 'summary' : modelSummary})
        k = k + 1

        #dealocate memory
        del history
        del predictions
        del actual
        del modelSummary
        del model

    return model_performance

def createModel(model_architecture):
    #returns a functions
    switcher = {
            model_type.FFN_base : FFNBaseModel.createModel,
            model_type.FFN_deep : FFNDeepModel.createModel,
            model_type.FFN_wide : FFNWideModel.createModel,
            model_type.FFN_Sigmoid : FFNSigmoidModel.createModel,
            model_type.FFN_TanH : FFNTanHModel.createModel,
            model_type.FFN_Leaky_ReLU : FFNLeakyReLuModel.createModel,
            model_type.CNN_base : CNNBaseModel.createModel,
            model_type.CNN_deep : CNNDeepModel.createModel,
            model_type.CNN_wide : CNNWideModel.createModel,
            model_type.CNN_Sigmoid : CNNSigmoidModel.createModel,
            model_type.CNN_TanH : CNNTanHModel.createModel,
            model_type.CNN_Leaky_ReLU : CNNLeakyReLUModel.createModel
            }

    return switcher.get(model_architecture)

def saveResults(model_architecture, results):
    filePath = createFilePath(epochs, 'ModelOutputs', k_fold_split, 'pkl')

    #write to disk
    results = np.array(results)
    with open (filePath, 'wb') as output:
        pickle.dump(results, output)

    print(str(filePath) + ' saved!')

print('*********')
print('* start *')
print('*********')

#load the dataset
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()

#normalize the data
x_train = tf.keras.utils.normalize(x_train, axis = 1)
x_test = tf.keras.utils.normalize(x_test, axis = 1)

#merge the train and test sets and merge the corresponing labels
(X, Y) = mergeDataSet()

#Train the model architecture
results = kFoldCrossValidation(model_architecture)
saveResults(model_architecture, results)

print('done')
```

Results.py

```
#Enum for choosing behavior and loading all the models
from enum import Enum
class model_type(Enum):
    FFN_base = 0
    FFN_deep = 1
    FFN_wide = 2
    FFN_Sigmoid = 3
    FFN_TanH = 4
    FFN_Leaky_ReLU = 5
    CNN_base = 6
    CNN_deep = 7
    CNN_wide = 8
    CNN_Sigmoid = 9
    CNN_TanH = 10
    CNN_Leaky_ReLU = 11

##########################################
# Parameters to specify the models to use #
##########################################
epoch = 100
k = 10
seed = 123

########################
# Statistics one tailed #
########################
# significance level 0.05 / 132 = 0.0000378 value (column 0.001 degree of freedom 9 used)
tKFold = 4.781

#support libraries
import sys
sys.path.insert(0, '../ModelOutputs')
import numpy as np
import pickle
from scipy import stats

def getConfusionMatrix(histories):
    #create empty table
    table = np.zeros((10,11))

    #fill table with the predicted and actual results for all k-folds
    for model in histories:

        actual = model['actual']
        predictions = model['predictions']
        for i in range(0, len(actual)):
            row = actual[i]
            column = predictions[i]
            table[row][column] += 1
            #total amount of images of the given category
            table[row][10] += 1

    #convert the table entries into pecentages
    for row in range(0, 10):
        for column in range(0,10):
            table[row][column] /= table[row][10] / 100
            table[row][column] = round(table[row][column], 2)

    #Predicted categories (Cifar10 categories)
    columns =
    'P.Airplane,P.Automobile,P.Bird,P.Cat,P.Deer,P.Dog,P.Frog,P.Horse,P.Ship,P.Truck,Total
'


    #Actual categories (Cifar10 categories)
    rows =    ['Airplane', 'Automobile', 'Bird', 'Cat', 'Deer', 'Dog', 'Frog', 'Horse',
    'Ship', 'Truck']

    #build the table with comma separation
    results = ',' + columns + '\n'
    for i in range(0, 10):
        results += rows[i]
```

```
        for j in range(0, 11):
            results += ',' + str(table[i][j])
        results += '\n'

    return results

def getModelHistory(epoch, k, seed, model_architecture):
    #history dataStructure
    #    [k] k-fold model
    #        ['history'] training loss, acc, val_loss, val_acc
    #        ['predictions'] predicted classes on the test set once trained
    #        ['actual'] actual classes of the test set
    #        ['summary'] string with model design

    model = str(model_architecture).split('.')[1]
    filePath = '../ModelOutputs/'
    filePath += model + '_epoch(' + str(epoch) + ')k('
    filePath += str(k) + ')seed(' + str(seed) + ').pkl'

    with open (filePath, 'rb') as file:
        modelHistory = pickle.load(file)

    return modelHistory

def getAverageTrainingHistory(history):
    averageHistory = {'acc' : np.zeros(epoch), 'val_acc' : np.zeros(epoch), 'loss' :
    np.zeros(epoch),      'val_loss' : np.zeros(epoch)}

    #sum acc, val_acc, loss, val_loss
    for model in history:
        averageHistory['acc'] += np.array(model['history']['acc'])
        averageHistory['val_acc'] += np.array(model['history']['val_acc'])
        averageHistory['loss'] += np.array(model['history']['loss'])
        averageHistory['val_loss'] += np.array(model['history']['val_loss'])

    #average accuracy, validation accuracy, loss, validation loss
    averageHistory['acc'] /= k
    averageHistory['val_acc'] /= k
    averageHistory['loss'] /= k
    averageHistory['val_loss'] /= k

    return averageHistory

def saveStatistics(results, model_architecture):
    model = str(model_architecture).split('.')[1]
    filePath = '../ModelOutputs/' + str(model) + '(statistics).txt'
    with open (filePath, 'w') as output:
        output.write(results)

def getResults(model_architecture):
    #get the model history for each k
    histories = getModelHistory(epoch, k, seed, model_architecture)

    #get a confusion matrix with actual image label vs predicted image label
    confusionMatrix = getConfusionMatrix(histories)

    #get the average training performance
    averageHistory = getAverageTrainingHistory(histories)

    trainingAcc = averageHistory['acc']
    trainingValAcc = averageHistory['val_acc']
    trainingLoss = averageHistory['loss']
    trainingValLoss = averageHistory['val_loss']

    #create a table epochs as columns, rows: training acc, val acc, training loss, val loss
    training = 'epohs'
    for i in range(0,100):
        training += ',' + str(i)

    training += '\nTraing accuracy'
    for i in trainingAcc:
        training += ',' + str(i)

    training += '\nValidation accuracy'
```

```
    for i in trainingValAcc:
        training += ',' + str(i)

    training += '\nTraing loss'
    for i in trainingLoss:
        training += ',' + str(i)

    training += '\nValidation loss'
    for i in trainingValLoss:
        training += ',' + str(i)

    output = str(model_architecture) + '\n\n'
    output += training + '\n'
    output += '\n\n' + confusionMatrix

    #save the performance and confusion matrix to disk
    saveStatistics(output, model_architecture)


    return {'histories' : histories, 'averageHistory' : averageHistory}

def pairedTTest(sampleA, sampleB):
    #A two tailed paired t-test returning a tuple <t-value, p-value>
    tp = stats.ttest_rel(sampleA, sampleB)

    results = {'t*' : tp[0], 'p*' : tp[1]}

    return results

def getEpochsDoneTraining(models, archType):
    modelTypes = ['base', 'deep', 'wide', 'Sigmoid', 'TanH', 'Leaky']

    bestKFoldEpochs = {}
    for i in range(0, len(modelTypes)):
        allKFolds = []
        for j in range(0, k):
            epochs = models[archType][modelTypes[i]]['histories'][j]['history']['val_loss']
            bestEpoch = np.argmin(epochs)
            allKFolds.append(bestEpoch)
        bestKFoldEpochs[modelTypes[i]] = allKFolds

    return bestKFoldEpochs

def getTrainingValAcc(models, archType, bestKFoldEpochs):
    modelTypes = ['base', 'deep', 'wide', 'Sigmoid', 'TanH', 'Leaky']

    trainedKFoldValAcc = {}
    for i in modelTypes:
        allKFolds = []
        for j in range(0, k):
            valAcc = models[archType][i]['histories'][j]['history']['val_acc']
            valAcc = valAcc[bestKFoldEpochs[i][j]]
            allKFolds.append(valAcc)
        trainedKFoldValAcc[i] = allKFolds

    return trainedKFoldValAcc

def getTTestTable(title, samples, measure):
    title = title + '\n'

    header = ''
    for key in samples:
        header += ',' + key
    header += ',' + measure + '\n'

    rows = ''
    for i in samples:
        row = i
        for j in samples:
            modelA = samples[i]
            modelB = samples[j]
            t = pairedTTest(modelA, modelB)['t*']
            row += ',' + str(t)
        average = float(np.sum(samples[i])/k)
```

```
        rows += row + ',' + str(average) + '\n'

    table = title + header + rows + '\n\n'
    return table

def allModels():
    #models datastructure
    #   ['FFN'] -> same as CNN
    #   ['CNN']
    #       ['base'] -> same as Leaky
    #       ['wide'] -> same as Leaky
    #       ['deep'] -> same as Leaky
    #       ['Sigmoid] -> same as Leaky
    #       ['TanH'] -> same as Leaky
    #       ['Leaky']
    #           ['averageHistory']          !! average training among all k folds !!
    #               ['acc']
    #               ['val_acc']
    #               ['loss']
    #               ['val_loss']
    #           ['histories']               !! k-fold training k = 10 !!
    #               [k]                     !! training history for fold k (0-9) !!
    #                   ['history']
    #                       ['acc']
    #                       ['val_acc']
    #                       ['loss']
    #                       ['val_loss']
    #                   ['predictions'] predicted classes on the test set once trained
    #                   ['actual'] actual classes of the test set
    #                   ['summary'] string with model design

    #build models datastructe by loading the results from files
    models = {'CNN' : {}, 'FFN' : {}}
    for archType in model_type:
        modelType = str(archType).split('.')[1].split('_')
        temp = getResults(archType)

        #networktype = FFN/CNN, variant = base, wide, deep, Sigmoid, TanH, Leaky
        networkType = modelType[0]
        variant = modelType[1]
        models[networkType][variant] = temp

    #Training statistics
    #FFN
    ffnTrain = getEpochsDoneTraining(models, 'FFN')
    ffnValAcc = getTrainingValAcc(models, 'FFN', ffnTrain)
    ffnTrainTable = getTTestTable('FFN training epochs', ffnTrain, 'epochs')
    ffnValAccTable = getTTestTable('FFN accuracy', ffnValAcc, 'acc')

    #CNN
    cnnTrain = getEpochsDoneTraining(models, 'CNN')
    cnnValAcc = getTrainingValAcc(models, 'CNN', cnnTrain)
    cnnTrainTable = getTTestTable('CNN Training epochs', cnnTrain, 'epochs')
    cnnValAccTable = getTTestTable('CNN accuracy', cnnValAcc, 'acc')

    #FFN and CNN training
    allTraining = {}
    for i in ffnTrain:
        key = 'f.' + i
        allTraining[key] = ffnTrain[i]
    for i in cnnTrain:
        key = 'c.' + i
        allTraining[key] = cnnTrain[i]
    allTrainingTable = getTTestTable('all training', allTraining, 'epochs')

    #FFN and CNN val acc
    allAccuracy = {}
    for i in ffnValAcc:
        key = 'f.' + i
        allAccuracy[key] = ffnAcc[i]
    for i in cnnValAcc:
        key = 'c.' + i
        allAccuracy[key] = cnnAcc[i]
    allAccuracyTable = getTTestTable('all accuracy', allAccuracy, 'acc')
```

```
    #t
    content = 't value' + ',' + str(tKFold) + ',' + str(-tKFold) + '\n\n'
    #FFN train
    content += ffnTrainTable
    #FFN acc
    content += ffnValAccTable
    #CNN train
    content += cnnTrainTable
    #CNN acc
    content += cnnValAccTable
    #FFN & CNN train
    content += allTrainingTable
    #FFN & CNN acc
    content += allAccuracyTable

    saveStatistics(content, 'output.Training')

allModels()
print('output calculated')
```

Model Templates:

CNNBaseModel.py

```
import random as rn
import numpy as np
import tensorflow as tf

def createModel(random_seed_value, input_x, input_y, color_depth):


#seed random
    rn.seed(random_seed_value)
    np.random.seed(random_seed_value)
    tf.set_random_seed(random_seed_value)


#create the model
    model = tf.keras.models.Sequential()


#Convolution 1
    model.add(tf.keras.layers.Conv2D(filters = 32, kernel_size = 5, padding = 'same', strides
    = 1,    input_shape = (input_x, input_y, color_depth), name = 'Convolution_1'))

#ReLU 2
    model.add(tf.keras.layers.Activation(tf.nn.relu, name = 'ReLU_2'))

#Max pooling 3
    model.add(tf.keras.layers.MaxPooling2D(pool_size = 2, strides = 2, name =
    'Max_pooling_3'))

#Convolution 4
    model.add(tf.keras.layers.Conv2D(filters = 64, kernel_size = 5, padding = 'same', strides
    = 1, name      = 'Convolution_4'))

#ReLU 5
    model.add(tf.keras.layers.Activation(tf.nn.relu, name = 'ReLU_5'))

#Max pooling 6
    model.add(tf.keras.layers.MaxPooling2D(pool_size = 2, strides = 2, name =
    'Max_pooling_6'))

#Flatten 7
    model.add(tf.keras.layers.Flatten(name = 'Flatten_7'))

#Fully connected 8
    model.add(tf.keras.layers.Dense(units = 1024, name = 'Fully_connected_8'))

#ReLU 9
    model.add(tf.keras.layers.Activation(tf.nn.relu, name = 'ReLU_9'))

#Fully connected 10
    model.add(tf.keras.layers.Dense(units = 10, name = 'Fully_connected_10'))

#Soft Max 11
    model.add(tf.keras.layers.Activation(tf.nn.softmax, name = 'Soft_max_11'))


#Compile model
    model.compile(loss = tf.keras.losses.sparse_categorical_crossentropy, optimizer = 'sgd',
    metrics =      ['accuracy'])
    return model
```

CNNDeepModel.py

```
import random as rn
import numpy as np
import tensorflow as tf

def createModel(random_seed_value, input_x, input_y, color_depth):


#seed random
    rn.seed(random_seed_value)
    np.random.seed(random_seed_value)
    tf.set_random_seed(random_seed_value)


#create the model
    model = tf.keras.models.Sequential()

#Convolution 1
    model.add(tf.keras.layers.Conv2D(filters = 32, kernel_size = 5, padding = 'same', strides
    ➡ = 1,   input_shape = (input_x, input_y, color_depth), name = 'Convolution_1'))

#ReLU 2
    model.add(tf.keras.layers.Activation(tf.nn.relu, name = 'ReLU_2'))

#Max pooling 3
    model.add(tf.keras.layers.MaxPooling2D(pool_size = 2, strides = 2, name =
➡       'Max_pooling_3'))

#Convolution 4
    model.add(tf.keras.layers.Conv2D(filters = 64, kernel_size = 5, padding = 'same', strides
    ➡ = 1, name      = 'Convolution_4'))

#ReLU 5
    model.add(tf.keras.layers.Activation(tf.nn.relu, name = 'ReLU_5'))

#Max pooling 6
    model.add(tf.keras.layers.MaxPooling2D(pool_size = 2, strides = 2, name =
➡       'Max_pooling_6'))

#Convolution 7
    model.add(tf.keras.layers.Conv2D(filters = 128, kernel_size = 5, padding = 'same',
    ➡ strides = 1,   name = 'Convolution_7'))

#ReLU 8
    model.add(tf.keras.layers.Activation(tf.nn.relu, name = 'ReLU_8'))

#Max pooling 9
    model.add(tf.keras.layers.MaxPooling2D(pool_size = 2, strides = 2, name =
        'Max_pooling_9'))

#Flatten 10
    model.add(tf.keras.layers.Flatten(name = 'Flatten_10'))

#Fully connected 11
    model.add(tf.keras.layers.Dense(units = 1024, name = 'Fully_connected_11'))

#ReLU 12
    model.add(tf.keras.layers.Activation(tf.nn.relu, name = 'ReLU_12'))

#Fully connected 13
    model.add(tf.keras.layers.Dense(units = 10, name = 'Fully_connected_13'))

#Soft Max 14
    model.add(tf.keras.layers.Activation(tf.nn.softmax, name = 'Soft_max_14'))


#Compile model
    model.compile(loss = tf.keras.losses.sparse_categorical_crossentropy, optimizer = 'sgd',
    ➡ metrics =      ['accuracy'])
    return model
```

CNNWideModel.py

```
import random as rn
import numpy as np
import tensorflow as tf

def createModel(random_seed_value, input_x, input_y, color_depth):


#seed random
    rn.seed(random_seed_value)
    np.random.seed(random_seed_value)
    tf.set_random_seed(random_seed_value)


#create the model
    model = tf.keras.models.Sequential()

#Convolution 1
    model.add(tf.keras.layers.Conv2D(filters = 64, kernel_size = 5, padding = 'same', strides
    = 1,   input_shape = (input_x, input_y, color_depth), name = 'Convolution_1'))

#ReLU 2
    model.add(tf.keras.layers.Activation(tf.nn.relu, name = 'ReLU_2'))

#Max pooling 3
    model.add(tf.keras.layers.MaxPooling2D(pool_size = 2, strides = 2, name =
    'Max_pooling_3'))

#Convolution 4
    model.add(tf.keras.layers.Conv2D(filters = 128, kernel_size = 5, padding = 'same',
    strides = 1,   name = 'Convolution_4'))

#ReLU 5
    model.add(tf.keras.layers.Activation(tf.nn.relu, name = 'ReLU_5'))

#Max pooling 6
    model.add(tf.keras.layers.MaxPooling2D(pool_size = 2, strides = 2, name =
    'Max_pooling_6'))

#Flatten 7
    model.add(tf.keras.layers.Flatten(name = 'Flatten_7'))

#Fully connected 8
    model.add(tf.keras.layers.Dense(units = 1024, name = 'Fully_connected_8'))

#ReLU 9
    model.add(tf.keras.layers.Activation(tf.nn.relu, name = 'ReLU_9'))

#Fully connected 10
    model.add(tf.keras.layers.Dense(units = 10, name = 'Fully_connected_10'))

#Soft Max 11
    model.add(tf.keras.layers.Activation(tf.nn.softmax, name = 'Soft_max_11'))


#Compile model
    model.compile(loss = tf.keras.losses.sparse_categorical_crossentropy, optimizer = 'sgd',
    metrics =      ['accuracy'])
    return model
```

CNNLeakyReLUModel.py

```
import random as rn
import numpy as np
import tensorflow as tf

def createModel(random_seed_value, input_x, input_y, color_depth):


#seed random
    rn.seed(random_seed_value)
    np.random.seed(random_seed_value)
    tf.set_random_seed(random_seed_value)


#create the model
    model = tf.keras.models.Sequential()

#Convolution 1
    model.add(tf.keras.layers.Conv2D(filters = 32, kernel_size = 5, padding = 'same', strides
    = 1,   input_shape = (input_x, input_y, color_depth), name = 'Convolution_1'))

#Leaky ReLU 2
    model.add(tf.keras.layers.LeakyReLU(name = 'Leaky_ReLU_2'))

#Max pooling 3
    model.add(tf.keras.layers.MaxPooling2D(pool_size = 2, strides = 2, name =
    'Max_pooling_3'))

#Convolution 4
    model.add(tf.keras.layers.Conv2D(filters = 64, kernel_size = 5, padding = 'same', strides
    = 1, name      = 'Convolution_4'))

#Leaky ReLU 5
    model.add(tf.keras.layers.LeakyReLU(name = 'Leaky_ReLU_5'))

#Max pooling 6
    model.add(tf.keras.layers.MaxPooling2D(pool_size = 2, strides = 2, name =
    'Max_pooling_6'))

#Flatten 7
    model.add(tf.keras.layers.Flatten(name = 'Flatten_7'))

#Fully connected 8
    model.add(tf.keras.layers.Dense(units = 1024, name = 'Fully_connected_8'))

#Leaky ReLU 9
    model.add(tf.keras.layers.LeakyReLU(name = 'Leaky_ReLU_9'))

#Fully connected 10
    model.add(tf.keras.layers.Dense(units = 10, name = 'Fully_connected_10'))

#Soft Max 11
    model.add(tf.keras.layers.Activation(tf.nn.softmax, name = 'Soft_max_11'))


#Compile model
    model.compile(loss = tf.keras.losses.sparse_categorical_crossentropy, optimizer = 'sgd',
    metrics =      ['accuracy'])
    return model
```

CNNSigmoidModel.py

```
import random as rn
import numpy as np
import tensorflow as tf

def createModel(random_seed_value, input_x, input_y, color_depth):


#seed random
    rn.seed(random_seed_value)
    np.random.seed(random_seed_value)
    tf.set_random_seed(random_seed_value)


#create the model
    model = tf.keras.models.Sequential()

#Convolution 1
    model.add(tf.keras.layers.Conv2D(filters = 32, kernel_size = 5, padding = 'same', strides
    = 1,   input_shape = (input_x, input_y, color_depth), name = 'Convolution_1'))

#Sigmoid 2
    model.add(tf.keras.layers.Activation(tf.nn.sigmoid, name = 'Sigmoid_2'))

#Max pooling 3
    model.add(tf.keras.layers.MaxPooling2D(pool_size = 2, strides = 2, name =
    'Max_pooling_3'))

#Convolution 4
    model.add(tf.keras.layers.Conv2D(filters = 64, kernel_size = 5, padding = 'same', strides
    = 1, name      = 'Convolution_4'))

#Sigmoid 5
    model.add(tf.keras.layers.Activation(tf.nn.sigmoid, name = 'Sigmoid_5'))

#Max pooling 6
    model.add(tf.keras.layers.MaxPooling2D(pool_size = 2, strides = 2, name =
    'Max_pooling_6'))

#Flatten 7
    model.add(tf.keras.layers.Flatten(name = 'Flatten_7'))

#Fully connected 8
    model.add(tf.keras.layers.Dense(units = 1024, name = 'Fully_connected_8'))

#Sigmoid 9
    model.add(tf.keras.layers.Activation(tf.nn.sigmoid, name = 'Sigmoid_9'))

#Fully connected 10
    model.add(tf.keras.layers.Dense(units = 10, name = 'Fully_connected_10'))

#Soft Max 11
    model.add(tf.keras.layers.Activation(tf.nn.softmax, name = 'Soft_max_11'))


#Compile model
    model.compile(loss = tf.keras.losses.sparse_categorical_crossentropy, optimizer = 'sgd',
    metrics =      ['accuracy'])
    return model
```

CNNTanHModel.py

```
import random as rn
import numpy as np
import tensorflow as tf

def createModel(random_seed_value, input_x, input_y, color_depth):


#seed random
    rn.seed(random_seed_value)
    np.random.seed(random_seed_value)
    tf.set_random_seed(random_seed_value)


#create the model
    model = tf.keras.models.Sequential()

#Convolution 1
    model.add(tf.keras.layers.Conv2D(filters = 32, kernel_size = 5, padding = 'same', strides
    ➡ = 1,   input_shape = (input_x, input_y, color_depth), name = 'Convolution_1'))

#TanH 2
    model.add(tf.keras.layers.Activation(tf.nn.tanh, name = 'TanH_2'))

#Max pooling 3
    model.add(tf.keras.layers.MaxPooling2D(pool_size = 2, strides = 2, name =
➡       'Max_pooling_3'))

#Convolution 4
    model.add(tf.keras.layers.Conv2D(filters = 64, kernel_size = 5, padding = 'same', strides
    ➡ = 1, name      = 'Convolution_4'))

#TanH 5
    model.add(tf.keras.layers.Activation(tf.nn.tanh, name = 'TanH_5'))

#Max pooling 6
    model.add(tf.keras.layers.MaxPooling2D(pool_size = 2, strides = 2, name =
➡       'Max_pooling_6'))

#Flatten 7
    model.add(tf.keras.layers.Flatten(name = 'Flatten_7'))

#Fully connected 8
    model.add(tf.keras.layers.Dense(units = 1024, name = 'Fully_connected_8'))

#TanH 9
    model.add(tf.keras.layers.Activation(tf.nn.tanh, name = 'TanH_9'))

#Fully connected 10
    model.add(tf.keras.layers.Dense(units = 10, name = 'Fully_connected_10'))

#Soft Max 11
    model.add(tf.keras.layers.Activation(tf.nn.softmax, name = 'Soft_max_11'))


#Compile model
    model.compile(loss = tf.keras.losses.sparse_categorical_crossentropy, optimizer = 'sgd',
    ➡ metrics =      ['accuracy'])
    return model
```

FFNBaseModel.py

```
import random as rn
import numpy as np
import tensorflow as tf

def createModel(random_seed_value, input_x, input_y, color_depth):

    #seed random
    rn.seed(random_seed_value)
    np.random.seed(random_seed_value)
    tf.set_random_seed(random_seed_value)


#Creation of the model

    model = tf.keras.models.Sequential()

    model.add(tf.keras.layers.Flatten(input_shape = (input_x, input_y, color_depth), name =
    ⇨ 'Flatten_1'))

    model.add(tf.keras.layers.Dense(1500, name = 'Fully_connnected_2'))
    model.add(tf.keras.layers.Activation(tf.nn.relu, name = 'ReLu_3'))

    model.add(tf.keras.layers.Dense(450, name = 'Fully_connected_4'))
    model.add(tf.keras.layers.Activation(tf.nn.relu, name = 'ReLu_5'))

    model.add(tf.keras.layers.Dense(10, name = 'Fully_connected_6'))
    model.add(tf.keras.layers.Activation(tf.nn.softmax, name = 'Softmax_7'))


#Compile the model
    model.compile(loss=tf.keras.losses.sparse_categorical_crossentropy,
                  optimizer = 'sgd',
                  metrics = ['accuracy'])

    return model
```

FFNDeepModel.py

```
import random as rn
import numpy as np
import tensorflow as tf

def createModel(random_seed_value, input_x, input_y, color_depth):

    rn.seed(random_seed_value)
    np.random.seed(random_seed_value)
    tf.set_random_seed(random_seed_value)

    model = tf.keras.models.Sequential()

    model.add(tf.keras.layers.Flatten(input_shape = (input_x, input_y, color_depth), name=
    ➡ 'Flatten_1'))

    model.add(tf.keras.layers.Dense(1500, name = 'Fully_connected_2'))
    model.add(tf.keras.layers.Activation(tf.nn.relu, name='ReLu_3'))

    model.add(tf.keras.layers.Dense(450, name = 'Fully_connected_4'))
    model.add(tf.keras.layers.Activation(tf.nn.relu, name='ReLu_5'))

    model.add(tf.keras.layers.Dense(150, name = 'Fully_connected_6'))
    model.add(tf.keras.layers.Activation(tf.nn.relu, name = 'ReLu_7'))

    model.add(tf.keras.layers.Dense(10, name = 'Fully_connected_8'))
    model.add(tf.keras.layers.Activation(tf.nn.softmax, name = 'Softmax_9'))


    model.compile(loss = tf.keras.losses.sparse_categorical_crossentropy,
                  optimizer = 'sgd',
                  metrics = ['accuracy'])

    return model
```

FFNWideModel.py

```
import numpy as np
import random as rn
import tensorflow as tf

def createModel(random_seed_value, input_x, input_y, color_depth):

    rn.seed(random_seed_value)
    np.random.seed(random_seed_value)
    tf.set_random_seed(random_seed_value)

    model = tf.keras.models.Sequential()

    model.add(tf.keras.layers.Flatten(input_shape = (input_x, input_y, color_depth), name =
    ➡ 'Flatten_1'))

    model.add(tf.keras.layers.Dense(3000, name = 'Fully_connected_2'))
    model.add(tf.keras.layers.Activation(tf.nn.relu, name = 'ReLu_3'))

    model.add(tf.keras.layers.Dense(900, name = 'Fully_connected_4'))
    model.add(tf.keras.layers.Activation(tf.nn.relu, name = 'ReLu_5'))

    model.add(tf.keras.layers.Dense(10, name = 'Fully_connected_6'))
    model.add(tf.keras.layers.Activation(tf.nn.softmax, name = 'Softmax_7'))

    model.compile(loss = tf.keras.losses.sparse_categorical_crossentropy,
                  optimizer = 'sgd',
                  metrics = ['accuracy'])

    return model
```

FFNLeakyReLuModel.py

```
import numpy as np
import random as rn
import tensorflow as tf

def createModel(random_seed_value, input_x, input_y, color_depth):

    rn.seed(random_seed_value)
    np.random.seed(random_seed_value)
    tf.set_random_seed(random_seed_value)

    model=tf.keras.models.Sequential()

    model.add(tf.keras.layers.Flatten(input_shape= (input_x, input_y,color_depth), name =
        'Flatten_1'))

    model.add(tf.keras.layers.Dense(1500, name = 'Fully_connected_2'))
    model.add(tf.keras.layers.LeakyReLU(name = 'Leaky_ReLU_3'))

    model.add(tf.keras.layers.Dense(450, name = 'Fully_connected_4'))
    model.add(tf.keras.layers.LeakyReLU(name = 'Leaky_ReLU_5'))

    model.add(tf.keras.layers.Dense(10, name = 'Fully_connected_6'))
    model.add(tf.keras.layers.Activation(tf.nn.softmax, name = 'Softmax_7'))

    model.compile(loss = tf.keras.losses.sparse_categorical_crossentropy,
                  optimizer = 'sgd',
                  metrics = ['accuracy'])
    return model
```

FFNSigmoidModel.py

```
import numpy as np
import random as rn
import tensorflow as tf

def createModel(random_seed_value, input_x, input_y, color_depth):

    rn.seed(random_seed_value)
    np.random.seed(random_seed_value)
    tf.set_random_seed(random_seed_value)

    model=tf.keras.models.Sequential()

    model.add(tf.keras.layers.Flatten(input_shape= (input_x, input_y,color_depth), name =
        'Flatten_1'))

    model.add(tf.keras.layers.Dense(1500, name = 'Fully_connected_2'))
    model.add(tf.keras.layers.Activation(tf.nn.sigmoid , name = 'Sigmoid_3'))

    model.add(tf.keras.layers.Dense(450, name = 'Fully_connected_4'))
    model.add(tf.keras.layers.Activation(tf.nn.sigmoid, name = 'Sigmoid_5'))

    model.add(tf.keras.layers.Dense(10, name = 'Fully_connected_6'))
    model.add(tf.keras.layers.Activation(tf.nn.softmax, name = 'Softmax_7'))

    model.compile(loss = tf.keras.losses.sparse_categorical_crossentropy,
                  optimizer = 'sgd',
                  metrics = ['accuracy'])
    return model
```

FFNTanHModel.py

```
import numpy as np
import random as rn
import tensorflow as tf

def createModel(random_seed_value, input_x, input_y, color_depth):

    rn.seed(random_seed_value)
    np.random.seed(random_seed_value)
    tf.set_random_seed(random_seed_value)

    model=tf.keras.models.Sequential()

    model.add(tf.keras.layers.Flatten(input_shape= (input_x, input_y,color_depth), name =
        'Flatten_1'))

    model.add(tf.keras.layers.Dense(1500, name = 'Fully_connected_2'))
    model.add(tf.keras.layers.Activation(tf.nn.tanh , name = 'TanH_3'))

    model.add(tf.keras.layers.Dense(450, name = 'Fully_connected_4'))
    model.add(tf.keras.layers.Activation(tf.nn.tanh, name = 'TanH_5'))

    model.add(tf.keras.layers.Dense(10, name = 'Fully_connected_6'))
    model.add(tf.keras.layers.Activation(tf.nn.softmax, name = 'Softmax_7'))

    model.compile(loss = tf.keras.losses.sparse_categorical_crossentropy,
                  optimizer = 'sgd',
                  metrics = ['accuracy'])
    return model
```