2018•2019
Faculteit Industriële ingenieurswetenschappen
master in de industriële wetenschappen: elektronica-ICT

Masterthesis

Development of AI classification system for sorting food applications

PROMOTOR :
Prof. dr. ir. Luc CLAESEN

PROMOTOR :
dr. ir. Gert POESEN

Bram Verstraeten

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: elektronica-ICT

2018•2019
Faculteit Industriële ingenieurswetenschappen
master in de industriële wetenschappen: elektronica-ICT

Masterthesis

Development of AI classification system for sorting food applications

PROMOTOR :
Prof. dr. ir. Luc CLAESEN

PROMOTOR :
dr. ir. Gert POESEN

Bram Verstraeten
Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: elektronica-ICT

Acknowledgements

This research and thesis were created in the first semester of the academic year of 2018-2019. Conducting this research was a learning phase from the beginning to the end. Due to the short time in which it was created it was very intensely but a pleasure to conduct. With this thesis I discovered the beauty and efficiency of Deep Learning applications which will certainly open new doors in the future.

Therefore I would like to thank Key Technology for granting me the opportunity the conduct this research. I hope this research will contribute to their product in the future and generates new opportunities for them.

Secondly I would like to thank dr. ir. Gert Poesen for his great guidance, time and insight in the world of sorting machines. As well as proposing the subject of this thesis.

Thirdly I would like to thank prof. dr. ir. Luc Clasen for his support and guidance during this research. Without his insightful advice and counseling this thesis would not have been the same.

Finally I would like to thank all people around me and specifically all my family and friends for their support. No only for this semester but also for all the years before. Without them I would not have succeeded in creating this thesis.

Abstract

The sorting of green beans in bulk food sorting applications with common image processing methods has a performance with is below the highest quality requirements of the customer. In this research a new approach to correctly sort green beans is introduced. The selected methods make use of advanced machine learning and Deep Learning algorithms. These algorithms use convolutional layers and fully connected layers to correctly classify if the object is good or bad. This was done with ordinary RGB images but also images that have extra information channels that contain information gathered by lasers. The training was done locally with a Python environment that runs with TensorFlow and Keras. These libraries ensure that a GPU could be used to speed up the training. In this research multiple network architectures where trained. These are a custom created, ResNet-18, ResNet-50 and MobileNetV2 architecture. The results for these networks are very divergent. Because the data set was very skewed, with 90% good and 10% bad examples, three metrics are monitored. The first two metrics are the accuracy and Loss. The thirth metric is the F1-score. The networks are trained on the original data set and on a data set where data augmentation is applied. This means that the object in the images is rotated and displaced to generated a bigger data set.

Abstract (Dutch)

Het sorteren van groene bonen in bulk sorteer applicaties met normale beeld verwerkingsmethodes heeft een prestatie die lager ligt dan de hoogste kwaliteit eisen van de klant. Een nieuwe aanpak voor het sorteren van groene bonen is geïntroduceerd in dit onderzoek. De geselecteerde methodes maken gebruik van geavanceerde machine learning of Deep Learning algoritmes. Deze algoritmes gebruiken convolutionele and standaard neuraal netwerk lagen om de objecten correct te classificeren tussen goed en slecht. Dit werd gedaan op gewone RGB foto's maar ook op foto's die extra informatie kanalen bevatten die bestaan uit informatie verzameld door lazers. De training werd lokaal uitgevoerd in een Python omgeving die TensorFlow en Keras gebruikt. Deze bibliotheken zorgen ervoor dat een GPU gebruikt kon worden om het trainen te versnellen. In dit onderzoek worden meerdere netwerk architecturen getraind. Dit is een zelf gemaakte, ResNet-18, ResNet-50 en MobileNetV2 architectuur. De resultaten voor deze netwerken lopen sterk uiteen. Omdat de data set scheef verdeeld is, 90% goede en 10% voorbeelden, worden 3 metrische variabelen gemonitord. De eerste twee variabele zijn de nauwkeurigheid en de schade. De derde variabele is de F1-score. De netwerken worden getraind op de originele data set en op de data set waar data aangroei is op toegepast. Dit betekent dat de objecten in de foto's worden geroteerd of verplaatst om een grotere data set te genereren.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The new breakthroughs in machine learning provides a new basis for the computer vision sector. Previously computer vision was accomplished with digital image processing where a variety of filter operations were applied to an image. This sequence of filters is defined by the designer of the algorithm. For complex shapes this process can very time consuming. Therefor a new method is needed. Most likely, machine learning can provide a solution. This relatively new technology makes it possible to classify objects with a complex shape or colour scheme on an easy way. The new algorithm will determine which operations it will need to correctly classify the images.

The food industry is a sector which is highly dependent on computer vision. To create a new method to sort products in this sector two aspects are very important, speed and accuracy. Speed is very important because this way a machine can sort a larger amount of product which will benefit the company. The accuracy is even more important than speed because a high quality control is imposed on the food industry. This means that a batch of products may not contain any foreign objects which do not belong there after the sorting process. If the accuracy of the sorting process is increased the amount of waste will decrease. This results in a better efficiency for the sorting plant.

This research focuses on developing a new way of sorting green beans. This is done in cooperation with Key Technology. Key Technology is a global firm with offices in the United States, Mexico, Australia, the Netherlands and Belgium. They develop machines that sort food products for food processor companies. Their newest machine, the VERYX digital sorting platform, is the platform on with the desired algorithm should run. This machine is capable of sorting a high throughput of bulk food products. This machine gathers information with multiple cameras and lasers. These cameras and lasers gather images with up to 12 channels. For this research a maximum of 8 channels was used. These channels are red, green, blue, infrared, infrared scatter, infrared anti-scatter, fluorescence and SWIR (Short-wavelength infrared).

The current implementation shows still room for significant improvement because the highest quality requirement of the customer is not reached. The current algorithm is capable of removing all foreign objects in the product stream but makes too much mistakes with other defects. The other defects that need to be sorted are stalk, beans with the stem still on and beans with stem and stalk on. These defects are all coming from the bean plant and are non toxic. If they enter the food chain they cause no harm. Therefor a zero tolerance on these defects is not necessary. The current accuracy of the algorithms in place ranges between 65% to 80%. This is still too low. Therefor another algorithm approach is explored.

The goal of this research is as stated earlier to develop an algorithm that can successfully classify green beans with a high accuracy. This means that all foreign material needs to be classified correctly, for other defects the goal is that minimal 80% is classified correctly. The requested market goal is that also all foreign material is classified correctly and that the other defects are classified correctly in 95% of the time. The speed of the algorithm will be disregarded in this research.

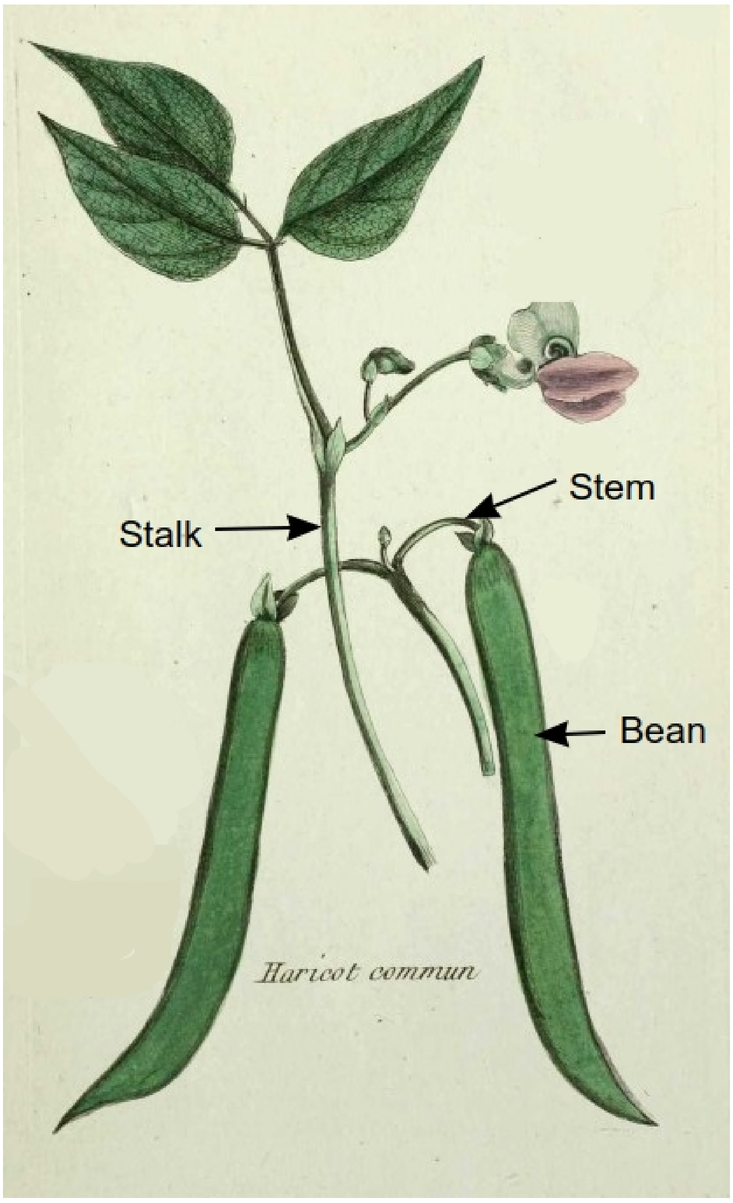


Figure 1.1: Figure of a bean with stem and stalk [1]

Chapter 2

Literature Study

2.1 Machine Learning Basics

Machine learning is the science of getting computers to act without being explicitly programmed. – Andrew Ng [2]

The quote above, from [2], gives one of many definitions for machine learning. The quote basically says that it is possible to make an algorithm without explicitly programming decision boundaries.

This is accomplished by creating a mathematical model of the problem without calculating the variables that are used. These variables are then 'learned' by the algorithm to make a reasonable guess of what the outcome should be.

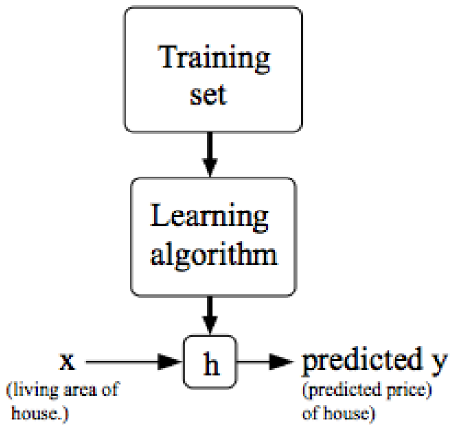


Figure 2.1: Machine learning principle [2].

Figure 2.1 shows the work flow of a machine learning algorithm. This figure shows that a learning algorithm uses a training set for 'learning' a hypothesis, h, that will predict the outcome, y, for an input, x [2].

There are two ways of learning a hypothesis. The first is supervised learning which means that the algorithm is trained with a labeled data set. The second learning method is unsupervised learning. In this case the algorithm gets a data set that is unlabeled. In this case the algorithm can recognize similar data but does not know what it is [2].

Because unsupervised learning cannot identify objects, it will no longer be in the scope of this research.

2.1.1 Terminology

Table 2.1 shows the basic terminology that is used in machine learning.

h or $h(\theta)$	The hypothesis of the model
θ_j	The jth feature to learn
θ	The feature space that need to be learned
$x_j^{(i)}$	The value of feature j in the ith training example
X	The space of input values
$y^{(i)}$	The output value of the ith training example
Y	The space of output values
$(x^{(i)}, y^{(i)})$	The ith training's example
m	The number of training examples
n	The number of features to learn

Table 2.1: Table of the used terminology.

2.1.2 Linear regression

Linear regression is the most simple form for predicting continuous outputs. It uses a linear function to approach the learning problem. The basic formula, according to [2], is :

$$h_\theta(x) = \theta_0 + \theta_1 * x_1 + \theta_2 * x_2 + \theta_3 * x_3 + \cdots + \theta_n * x_n \tag{2.1}$$

Equation 2.1 can be simplified. This can be done when θ en x are considered as two $(n+1) \times 1$ matrices.

$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} \qquad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \\ \vdots \\ \theta_n \end{bmatrix}$$

Note that x_0 is used in the x-matrix because the matrix dimensions otherwise do not match. x_0 will therefore always be one as can be seen in formula 2.1, $\theta_0 = \theta_0 * x_0$ if $x_0 = 1$. According to [2] equation 2.1 is transformed into:

$$h_\theta(x) = \theta^T * x \tag{2.2}$$

2.1.3 Polynomial regression

A lot of machine learning problems can be solved with linear regression. However, if the nature of the problem is not linear it can be difficult to find a decent solution in linear regression.

When the problem is not linear anymore. It is better to use a polynomial function. Equation 2.3 shows an example of such a polynomial function.

$$h_\theta(x) = \theta_0 + \theta_1 * x_1 + \theta_2 * x_1^2 + \theta_3 * x_1^3 \tag{2.3}$$

A polynomial regression can be considered as a special case of a linear regression. This can be done when the x-matrix is used like [2]:

$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_1^2 \\ x_1^3 \\ \vdots \\ x_1^n \end{bmatrix}$$

When the x-matrix is constructed like the example above equation 2.2 can be used to describe the hypothesis [2].

2.1.4 Logistic regression

Linear regression and polynomial regression are very good at solving problems with prediction continuous values. But when a binary classification problem is introduced the performance of these algorithms drops. They predict values between $-\infty$ and $+\infty$. The model also reacts too much on possible outliers which is not desirable [2].

To solve this, the sigmoid function is introduced. According to [2] equation 2.4 defines this function while figure 2.2 gives a graphical representation.

$$g(z) = \frac{1}{1 + \exp^{-z}} \tag{2.4}$$

This sigmoid function maps the output of any of the previous hypothesis, which had possible values between $-\infty$ and $+\infty$, between 0 and 1. So according to [2] the hypothesis becomes:

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1 + \exp^{-\theta^T x}} \tag{2.5}$$

The output of this hypothesis can be considered the probability that the output is 1. The only thing that remains is assigning 1 or 0 as output. This is done by choosing a threshold to which the output of the hypothesis will be subjected. For example, if the threshold is 0.5 and $h_\theta(x) = 0.7$ then the output of the algorithm will be 1. If $h_\theta(x) = 0.3$ then the output will be 0 [2].

For classification problems with multiple outputs there needs to be a logistic regression function for every possible output. These logistic regressions are computed in parallel. The function that outputs the highest probability will determine what the discrete output will be [2].

2.1.5 Neural networks

Neural networks are a further elaborated method of making predictions. This method is based on mimicking the human brain. Figure 2.3 shows such a basic neural network. This neural network has 3 layers, an input layer, one hidden layer and an output layer. The depth of a neural network is determined by the number of hidden layers. These layers are not visible for the end-user of the algorithm [2].

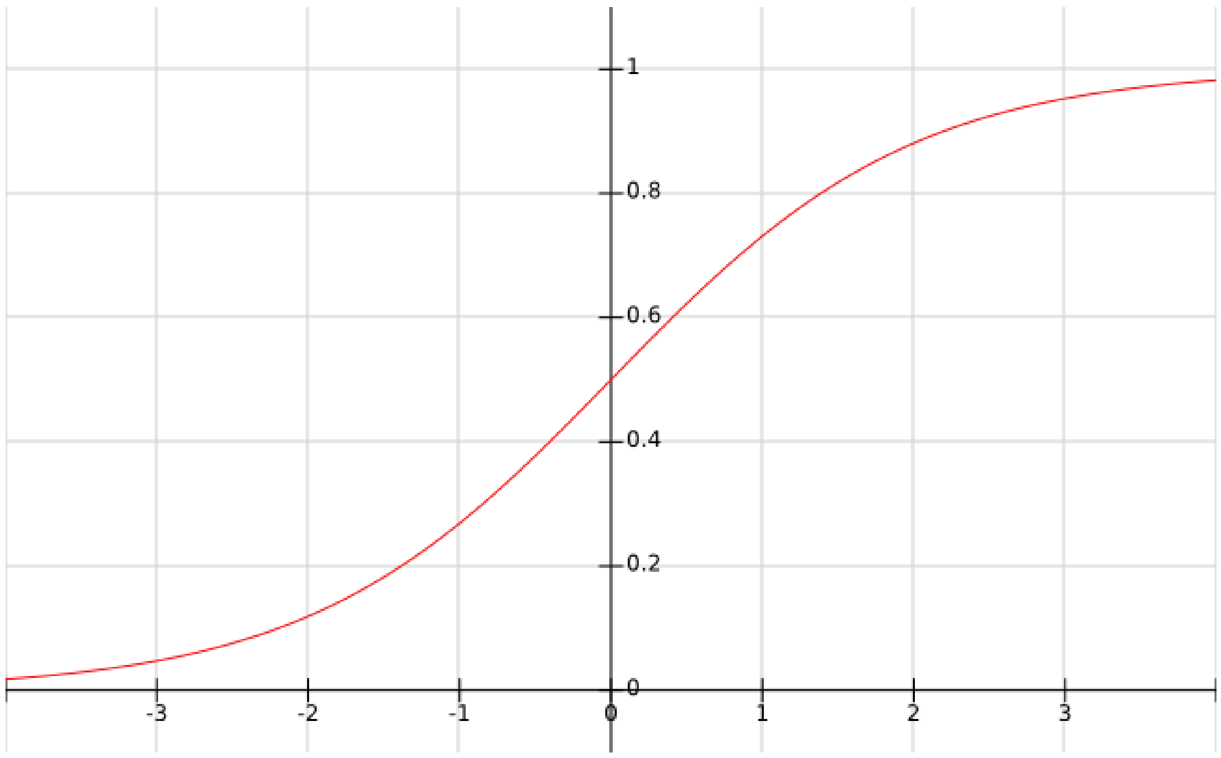


Figure 2.2: Sigmoid function.

For neural networks there is some extra terminology used. $a_i^{(l)}$ is the 'activation' of unit i in layer l. And $\Theta^{(l)}$ is the matrix of weights between layer l and $l+1$. Consequently, $\Theta_{ji}^{(l)}$ is the weight that connects the node i in layer l to the node j in layer $l+1$ [2].

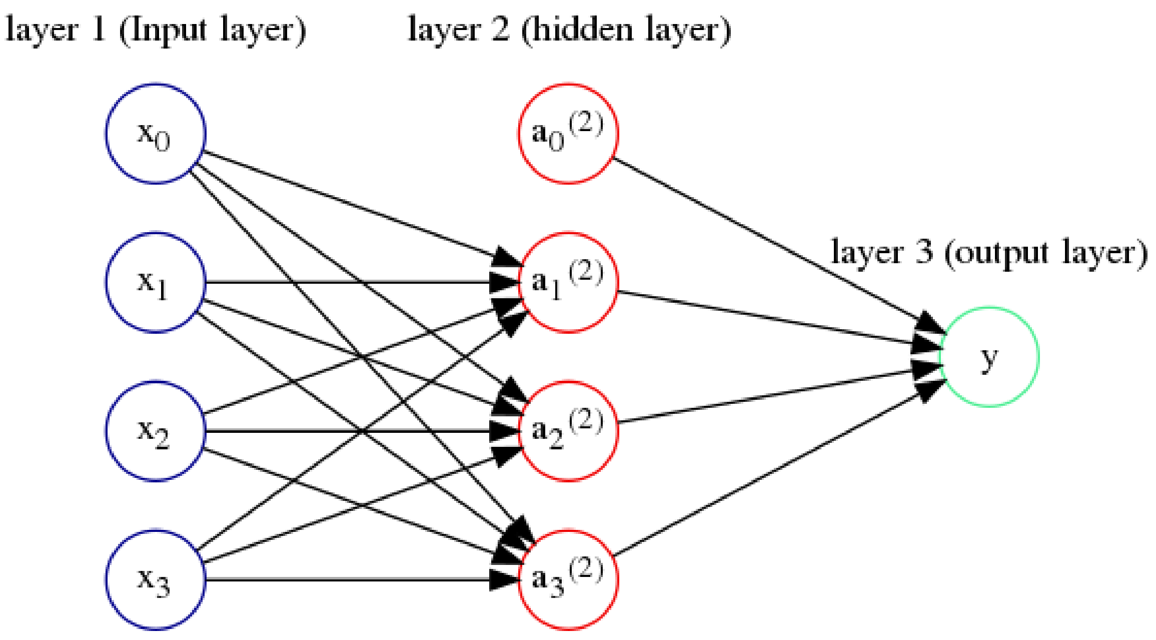


Figure 2.3: Basic neural network with bias nodes.

The output y for figure 2.3 is computed as follows:

$$\begin{aligned}
a_1^{(2)} &= g(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3) \\
a_2^{(2)} &= g(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3) \\
a_3^{(2)} &= g(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3) \\
h_\Theta(x) = a_1^{(3)} &= g(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)})
\end{aligned}$$

This can be written as an operation with matrices, then this will become:

$$\begin{aligned}
a^{(2)} &= g(\Theta^{(1)} * x) \\
h_\Theta(x) &= g(\Theta^{(2)} * a^{(2)})
\end{aligned}$$

In the example above the activation function used is the sigmoid function. But this can be any continuous function where the output is a real number [15].
The bias nodes, in this example x_0 and $a_0^{(2)}$, are shown in figure 2.3 but in reality these are not shown. The same neural network but without the bias nodes is shown in figure 2.4.

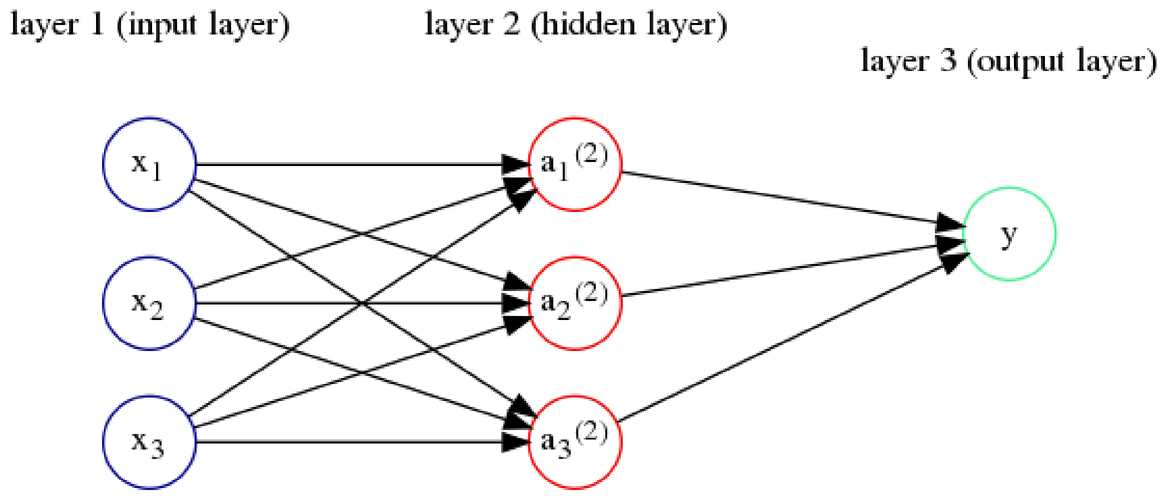


Figure 2.4: Basic neural network.

A neural network is not bound to have only one output node. The amount of output nodes can be the amount of classes it needs to recognize. This means that a neural network can represent very complex functions without the need of a complicated mathematical model [2].

2.1.6 Gradient descent

In the previous subsections different sorts of hypotheses are explained. These hypotheses all have some variables that are represented by θ or Θ. To use these hypotheses these variables need a value. These values define the model and are challenging to calculate for complex hypotheses. Gradient descent is a way to 'learn' these variables with the help of a labeled data set [2].

The first step is to initialize the variables. For linear, polynomial or logistic regression these variables can be initialized as zeros. For a neural network this needs to be random values. This is needed because otherwise different nodes could learn to recognize the same features. Another advantage of the use of random values is a faster learning speed [2].

Step two is computing the cost of the hypothesis. This is done by running the data set through the algorithm and calculating the error. The error is computed by calculating the difference between the estimated value and the exact value of the labeled data set. The cost is then found by calculating the mean squared error of all the error values of the data set. For the logistic regression this method cannot be used because there are a lot of local optima. Therefore a new method is used. If the label equals 0 then the cost for that example will be 0 if the predicted value also equals 0. If the label is 1 then the cost will be 0 if the cost equals 1. The total cost is then computed by averaging the cost of each example. Note that the output of a logistic regression is not limited to binary values. The output can contain values between 0 and 1. The selection of the threshold value or which logistic model to use is done after the model is trained [2].
According to [2] the cost for linear regression and logistic regression are respectively given by equations 2.6 and 2.7. The cost for polynomial regression is the same as for linear regression.

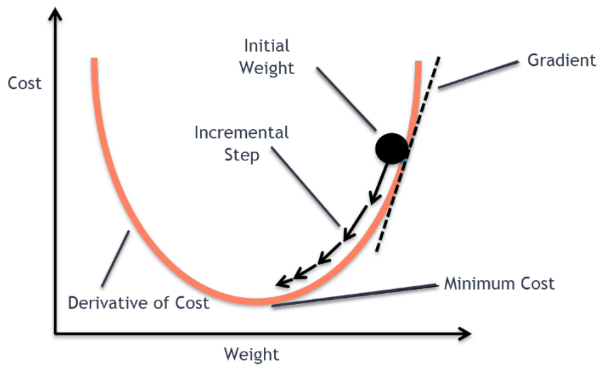


Figure 2.5: Example of gradient descent [3].

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 \tag{2.6}$$

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log(h_\theta(x^{(i)})) + (1-y^{(i)})\log(1-h_\theta(x^{(i)}))] \tag{2.7}$$

The third step is to compute the gradients of the cost function and update the variables bases on these gradients. For linear, polynomial and logistic regression this is done with equation 2.8 as reported by [2]. These steps need to be repeated until the cost reaches a minimum. This process is called gradient descent [2].

$$\theta_j := \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta) = \theta_j - \frac{\alpha}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} \tag{2.8}$$

2.1.7 Back propagation

The cost of a neural network, as stated by [2], is given by equation 2.9. In this equation K stands for the number of output nodes, y_k is the labeled value of the kth output node and $(h_\Theta(x^{(i)}))_k$ is the computed value of the kth output node.

$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}[y_k^{(i)}\log((h_\Theta(x^{(i)}))_k) + (1-y_k^{(i)})\log(1-(h_\Theta(x^{(i)}))_k)] \tag{2.9}$$

For multi-layer neural networks gradient descent is efficiently implemented with the back propagation algorithm. This means that the network is run through back to front and the gradients of each parameter are computed per layer [2].

The first step is making a forward propagation of the network. This means that all the 'a' and output values, y, in the network are computed for an input x.
Secondly the error values of the last layer are computed. This is done with equation 2.10 [2].

$$\delta^{(L)} = a^{(L)} - y \tag{2.10}$$

In equation 2.10 $\delta^{(l)}$ stands for the error value in layer l and L is the total number of layers. δ is going to be a matrix with the same dimension as the matrix of activations from that layer.

Next the error values of the other layers are computed with equation 2.11. In this equation g' stands for the derivative of the activation function, g, and $.*$ is the element-wise multiplication [2].

$$\delta^{(l)} = ((\Theta^{(l)})^T\delta^{(l+1)}).*g'(z^{(l)}) \tag{2.11}$$

When all the error values are computed, the gradient for every variable is computed with equation 2.12 as stated in [2]. These steps need to be computed for every training example. This means that $\Delta_{ji}^{(l)}$is the sum of all the gradients of $\Theta_{ji}^{(l)}$ [2].

$$\Delta_{ji}^{(l)} := \Delta_{ji}^{(l)}.*a_i^{(l)}\delta_j^{(l+1)} \tag{2.12}$$

The last step to computing the partial derivatives of the Θ variables is computing the mean of the gradients. The importance to do this as a separate step is handled further on in the literature study. Equation 2.13 shows this step [2].

$$D_{ji}^{(l)} = \frac{\partial}{\partial\Theta_{ji}^{(l)}}J(\Theta) = \frac{1}{m}\Delta_{ji}^{(l)} \tag{2.13}$$

This process needs to be repeated, just as gradient descent, until a minimum for the cost function is found [2].

2.1.8 Evaluation of an algorithm

The previous section shows how an algorithm can be built and trained. This is done with a data set that is called the training set. The algorithm will build his model to match this training set. This process is evaluated by the cost on the training set. This however does not tell how the algorithm will react to new data. This is because a normal data set, and thus also the training set, has outliers and these outliers will influence the training of the algorithm. That is why a part of the data set is kept apart, this data set is called the test set. After training the algorithm with the training set, the test set is used to make a final evaluation of the algorithm. This evaluation is commonly made with the cost or the accuracy of the algorithm. The accuracy is calculated by dividing the correct classified examples by the total amount of examples. This tells something more about how good it classifies [2].

2.1.8.1 Bias and variance

Bias and variance are two keywords that denote how a model is doing. If a model has high bias it means that it is underfitting. The model is not complex enough to model the data acceptably. High variance on the other hand means that the model is overfitting. The calculated model is too complex for the data. All outliers are perfectly matched with the model but that means that the error on new data is a lot bigger [2].

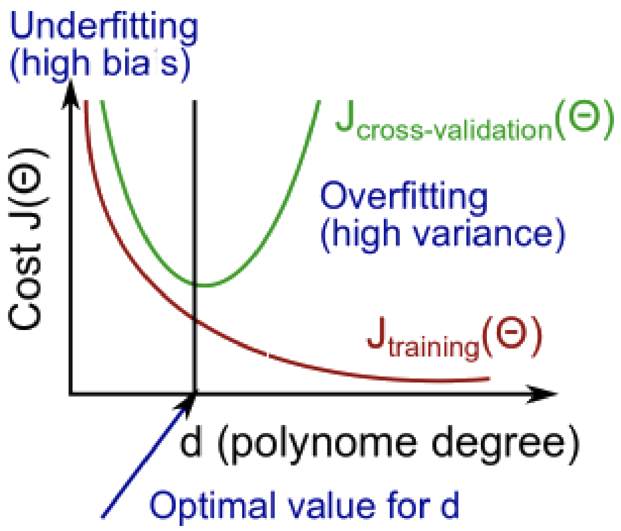


Figure 2.6: Example of high bias (cost of training and cross-validation are close together but high) and high variance (cost of training is very low but the cost of the cross-validation is high) [2].

Figure 2.6 shows an example of how to detect high bias or high variance. It shows the relation between the cost of the model and the polynomial degree. On the left side of the optimal polynomial degree the model has a high bias. In other words, the model is not complex enough. On the right side of the optimal polynomial degree the model has a high variance. The model is too complex to fit the data [2].

Figure 2.7a shows the influence of the training set size on a high bias model. The figure shows that using a larger data set does not solve the problem. This is because a larger data set does not make a more complex function. High bias can only be solved by using more features or by adding more polynomial features [2].

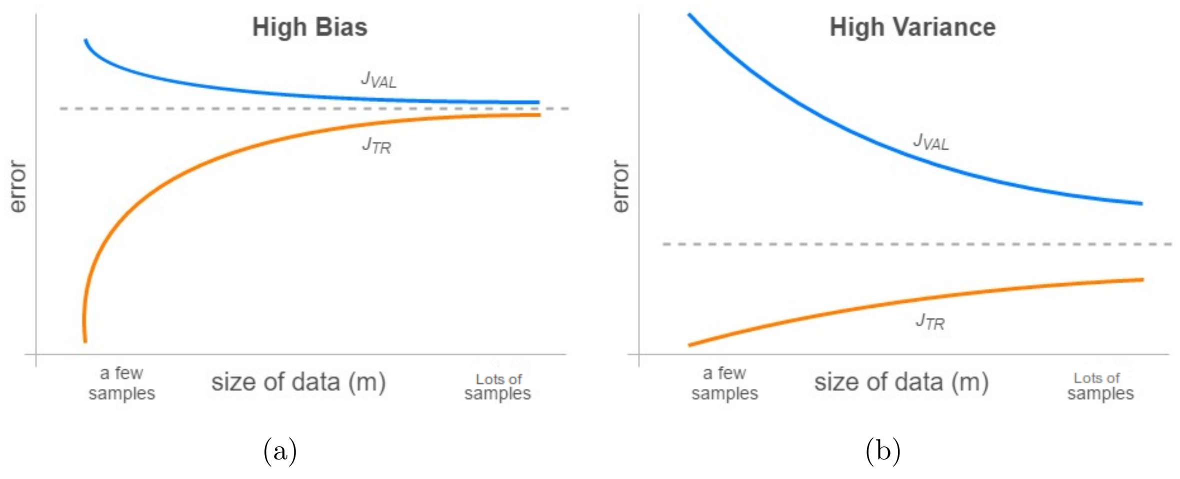


Figure 2.7: (a) Influence of training set size on high bias [4] (b) Influence on training set size on high variance [4].

Figure 2.7b shows the influence of the training set size on a high variance problem. In this figure it can be seen that getting a larger training set does affect the performance of this example, in a positive way. This could be predicted because the impact of outliers is smaller in a large training set. Another way to solve the model from overfitting is getting a smaller set of features [2].

2.1.8.2 Skewed data set

The accuracy of the algorithm is not always a good measure to evaluate the model. If the data set provided does not contain enough examples of one particular class the data set is called a skewed data set. This can have a serious influence on the algorithm. For instance if there is a data set of 100 values (m = 100), 99 of those values are '1' and 1 value is '0'. Then the algorithm is inclined to always output '1'. This has as a result that 99 of the 100 examples are classified correctly and thus the algorithm has an accuracy of 99%. This seems like a good algorithm to classify these classes but it will always fail to detect the '0' class [2].

A better approach on this is to use the F1-score. For calculating the F1-score the precision and recall of the algorithm is needed. The precision and recall are calculated with a confusion matrix as can be seen in figure 2.8 [16].

According to [16] the recall, precision and F1-score are given by equation 2.14, 2.15 and 2.16.

$$recall = \frac{\#TruePositives}{\#TruePositives + \#FalseNegatives} \tag{2.14}$$

$$precision = \frac{\#TruePositives}{\#TruePositives + \#FalsePositives} \tag{2.15}$$

$$F1 - score = \frac{2 * Recall * Precision}{Recall + Precision} \tag{2.16}$$

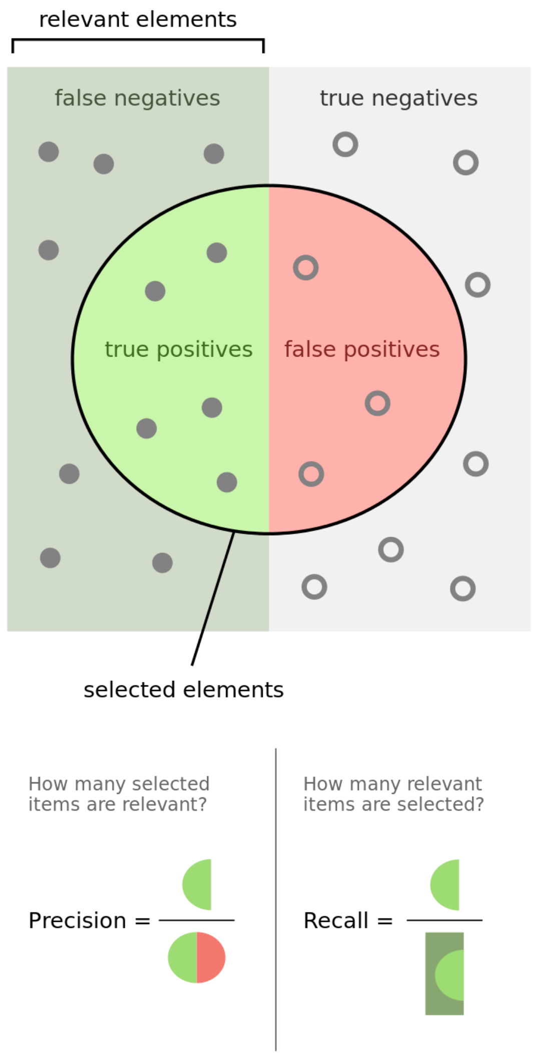


Figure 2.8: Confusion matrix with precision and recall visualized [5].

2.1.9 Regularization

Another way of solving problems with high bias or variance is introducing regularization parameters in the model. This regularization parameter can suppress or reinforce the high polynomial parameters in the model [2].

According to [2] the cost and the gradient update for linear and polynomial functions, shown in equation 2.6 and 2.8, becomes:

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 \boxed{+\lambda\sum_{j=1}^{n}\theta_j^2} \tag{2.17}$$

$$\begin{aligned} \theta_0 &:= \theta_0 - \frac{\alpha}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_0^{(i)} \\ \theta_j &:= \theta_j - \alpha[(\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}) \boxed{+\frac{\lambda}{m}\theta_j]} \qquad j \in \{1, 2 \dots n\} \end{aligned} \tag{2.18}$$

The cost for a logistic regression becomes like equation 2.19, the gradient update is the same as for linear functions (equation 2.18) [2].

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log(h_\theta(x^{(i)})) + (1 - y^{(i)})\log(1 - h_\theta(x^{(i)}))] \boxed{+\frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2} \tag{2.19}$$

For Neural networks only the last step changes. Adding regularization is the reason why the mean needs to be computed separately. According to [2] equation 2.13 becomes:

$$\begin{aligned} D_{ji}^{(l)} &= \frac{1}{m}(\Delta_{ji}^{(l)} \boxed{+\lambda\Theta_{ji}^{(l)}}) && \text{if } i \neq 0 \\ D_{ji}^{(l)} &= \frac{1}{m}\Delta_{ji}^{(l)} && \text{if } i = 0 \end{aligned} \tag{2.20}$$

Note that with the introduction of regularization a new parameter is introduced. This parameter is called λ. With this parameter high bias and high variance problems can be solved. If the problem is high bias, then the value of λ should decrease. If the problem is high variance, then the value of λ should increase [2].

The best method to determine the best value for λ is using an extra data set. This is called the cross-validation set. The model is still trained with the training set without regularization. After the training the regularization is added and the best value for λ is when the cost on the cross-validation set reaches its minimum. The model is afterwards still evaluated with the test set [2].

2.2 Convolutional neural networks

As stated before, neural networks are useful methods for making a classification algorithm. But there are limits. A large neural network (a deep network with many parameters) is prone to overfitting. This is also the case for recognizing objects in pictures [8].

Take for example that for a classification task an object in an RGB picture of 300×300 pixels needs to be recognized. This means that the number of input features equals $300 * 300 * 3 = 270000$. Thus every pixel is an input. Figure 2.9 shows the network that is used for calculating the number of trainable features. This network has one input layer of n features (n = 270000 in this example), a hidden layer with m (m = 438080 this was randomly chosen) features and an output layer with one output feature. The matrix containing the trainable features between layer 1 and layer 2 has a size of $m \times (n + 1)$ which in this case is 438080×270001. The matrix with trainable features between layer 2 and 3 has the size of 1×438081. The total number of trainable features is then:

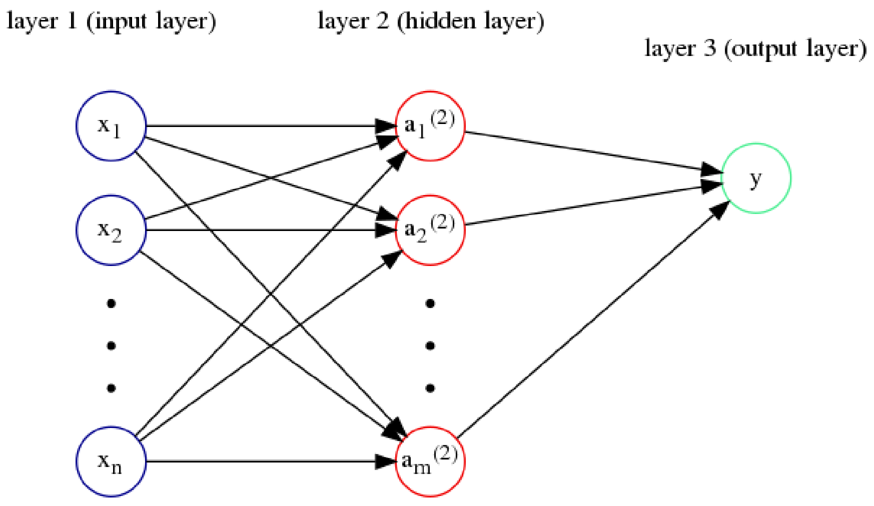


Figure 2.9: Neural network with 3 layers, n input features and 1 output feature.

$$438080 * 270001 + 1 * 438081 = 118282476161$$

Having this much features to train means that the chances of the model overfitting are very high, or the provided data set needs to be immense. This is the reason why convolutional neural networks have been developed [8].

A convolutional neural network is based on the same principle as used in digital image processing. Namely the use of a filter, which is convoluted with the image. For a convolutional network this means that only variables of the filter need to be learned. Take for example the first layer of a convolutional neural network where the input size is $300 \times 300 \times 3$ and there are 5 filters used with a size of $5 \times 5 \times 3$. Then the number of features that need to be trained is $(3 * (5 * 5) + 1) * 5 = 380$ (the +1 in this equation stands for 1 bias node for each filter). This is a lot less then the 118282038080 features that need to be trained in the first layer of the fully connected neural network for the same layer. This means that a convolutional neural network is less prone to overfitting and can be trained substantially faster with a smaller data set [8].

2.2.1 Convolution

A convolution is mathematical method that is widely used in signal and image processing. It is used to filter or extract patterns out of signals or images. In image processing the image is convoluted with a 'kernel' of a 'filter'. This 'filter' is another smaller matrix that is designed to detect patterns or filter the image [6].

$$\underbrace{\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & \dots \\ 6 & 7 & 8 & 9 & 10 & \dots \\ 6 & 7 & 8 & 9 & 10 & \dots \\ 1 & 2 & 3 & 4 & 5 & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}}_{\text{Image}} \circledast \underbrace{\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}}_{\text{Filter}} = \begin{bmatrix} x00 & x10 & x20 & \dots \\ x01 & x11 & x21 & \dots \\ x02 & x12 & x22 & \dots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

$x00 = 1*0+2*1+3*0+6*1+7*-4+8*1+6*0+7*1+8*0 = -5$
$x10 = 2*0+3*1+4*0+7*1+8*-4+9*1+7*0+8*1+9*0 = -5$
$x01 = 6*0+7*1+8*0+6*1+7*-4+8*1+1*0+2*1+3*0 = -5$

⋮

The example above shows how the convolution of two matrices is calculated. The filter is placed over the image and the element wise product and cumulative sum of the product is made. The filter then slides a place to the right and the same operations are executed. When the filter reaches the right border it is slid back to the left border and a place down. The process is repeated until the filter reaches the bottom right corner. Figure 2.10 gives a visual representation.

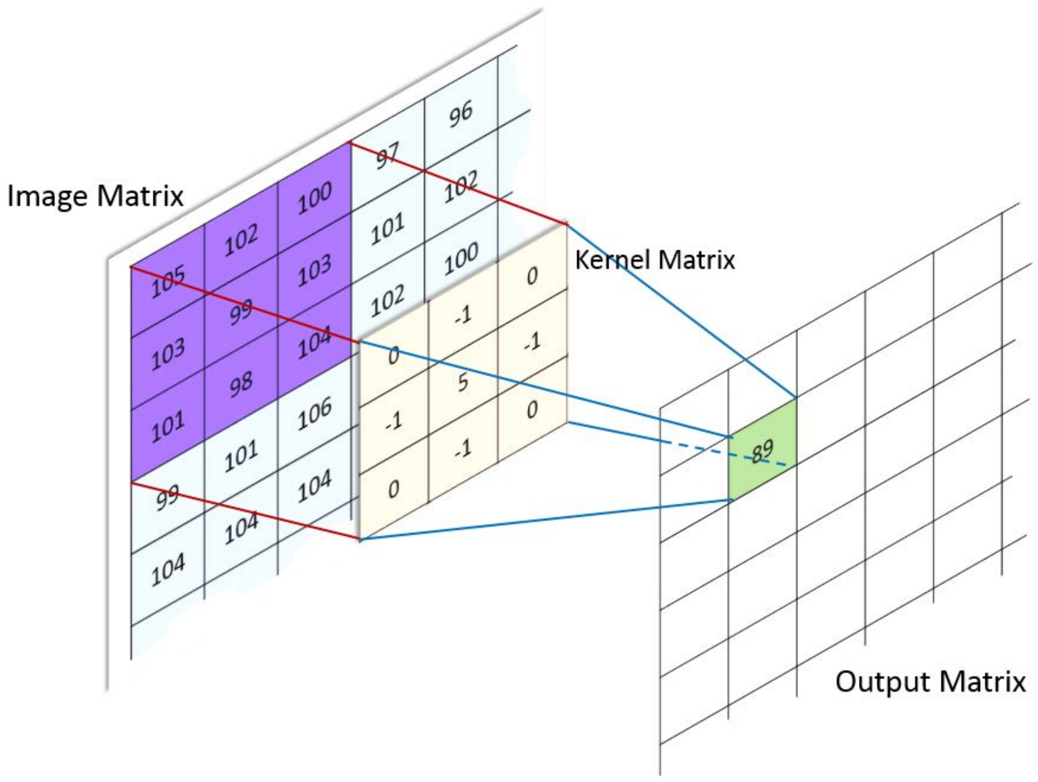


Figure 2.10: Visual representation of a convolution [6].

2.2.2 Padding and stride

If the filter used has a size bigger then 1×1 then the dimensions of the output matrix are smaller then the dimension of the input matrix. This means that the dimensions of the output size are dependent on the filter size. To counter this effect padding can be added to the image. This means that around the picture a series of zeroes, or other values (e.g. derived from values of neighbouring elements), are added. This has as a consequence that there are two ways of convolution usage. Namely, 'Valid' convolution and 'Same' convolution. With valid convolution no padding is added, the height and width of the output will decrease. In the case of same convolution, the amount of padding is just enough the keep the dimension of the output the same is the input dimension [8].

Another variable that can influence the output dimensions is the stride that is used. The stride is the amount of pixels that the filter shifts. When the convolution used is 'same' convolution but the stride is 2 then the output dimension will be about half of the original dimension [8].

$$n_H^{(l)} = \lfloor \frac{n_H^{(l-1)} + 2P^{(l)} - f^{(l)}}{S^{(l)}} + 1 \rfloor \tag{2.21}$$

Equation 2.21, according to [8], is used to calculate the output dimension size of one convolutional layer. $n_H^{(l)}$ stands for the output height, $n_H^{(l-1)}$ is the input height, $f^{(l)}$ is the filter size, $P^{(l)}$ is the amount of padding and $S^{(l)}$ is the value of the stride. The equation for the width is the same as equation 2.21 and the number of output channels is the same as the number of filters that are used.

Example: If the matrix A is the input image, filter size of 2 and the stride is 2, how much padding need to be added to have a same convolution?

$$\begin{bmatrix} x00 & x10 & x20 & x30 \\ x01 & x11 & x21 & x31 \\ x02 & x12 & x22 & x32 \\ x03 & x13 & x23 & x33 \end{bmatrix}$$

Matrix A = 4×4

$$4 = \lfloor \tfrac{4+2P^{(l)}-2}{2} + 1 \rfloor$$
$$P^{(l)} = 2$$

This means that on all sides of the matrix A 2 pixels of zeroes need to be added. Matrix A is transformed to matrix B before the convolution is executed.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & x00 & x10 & x20 & x30 & 0 & 0 \\ 0 & 0 & x01 & x11 & x21 & x31 & 0 & 0 \\ 0 & 0 & x02 & x12 & x22 & x32 & 0 & 0 \\ 0 & 0 & x03 & x13 & x23 & x33 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Matrix B = 8×8

2.2.3 Pooling layer

In the previous sections it is shown that that a convolutional network significantly reduces the amount of features to train. But if 'same' convolution is used, then the number of input features does not decline and makes for a higher computationally cost. Therefore pooling is used. A pooling layer reduces the height and width dimensions of the input matrix. It can do this in two ways [8]:

- max pooling: computing the maximum of the values in each filter
- average pooling: computing the average of the values in each filter

Figure 2.11 gives an example of such a max pooling layer. It uses a 2×2 filter with a stride of 2. For calculating the output dimensions equation 2.21 can be used. Important is that the number of channels does not change when using pooling [8].

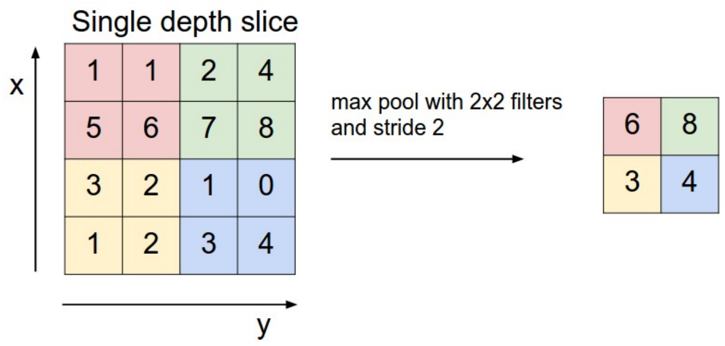


Figure 2.11: Max pooling [7].

2.2.4 1×1 convolution

In a classic convolutional neural network the height and width dimension will decrease while the number of channels will increase. A 1×1 convolution can help control that number output channels. A 1×1 convolution can also be seen as a fully connected layer inside the convolution model that is only applied to the corresponding elements in each channel, because it makes the element wise product of the elements in the input matrix with the filter and then applies an activation function. The output of this convolution is a matrix where the height and width dimensions are preserved but the number of channels is changed to the number of applied filters [8].

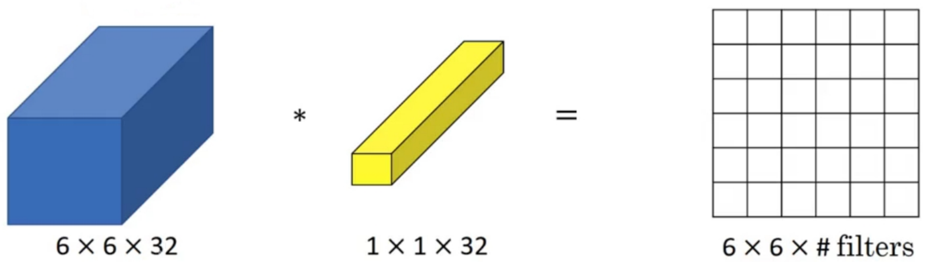


Figure 2.12: example of a 1×1 convolution [8].

2.2.5 Residual networks

Complex problems ask for relative deep convolutional networks. But with the classic approach this is not always possible. A classic network can result in a larger error if the number of layers is increased. This can be seen in figure 2.13. A solution for this problem is found in residual networks. This is a new topology where certain convolutional layers can be jumped with a skip connection [8].

Figure 2.14 shows two possible implementations of a ResNet, residual network, block. The first is the identity block, show in figure 2.14a. Here the skip connection is clearly shown. The second block, is the convolutional block shown in figure 2.14b. The convolutional and identity block have both the same purpose. But the identity block can only be used if the input and output dimensions of both connections are the same. The convolutional block is used if the input and output dimension do not match [8].

Equations 2.22 and 2.23 respectively show the calculations for a classic convolutional

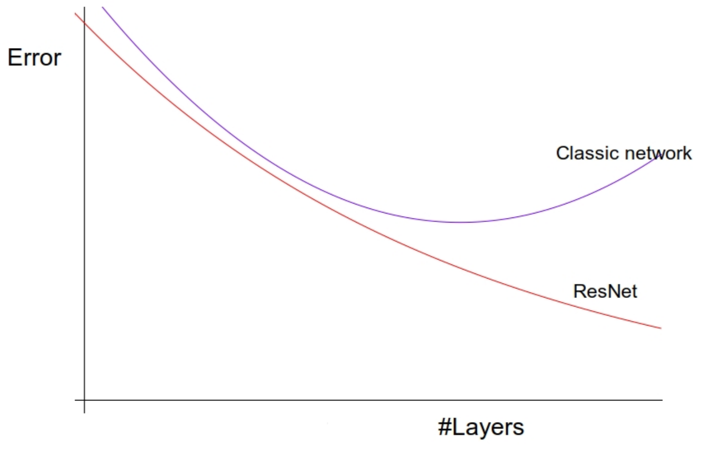


Figure 2.13: Error of convolutional nets in function of the number of layers.

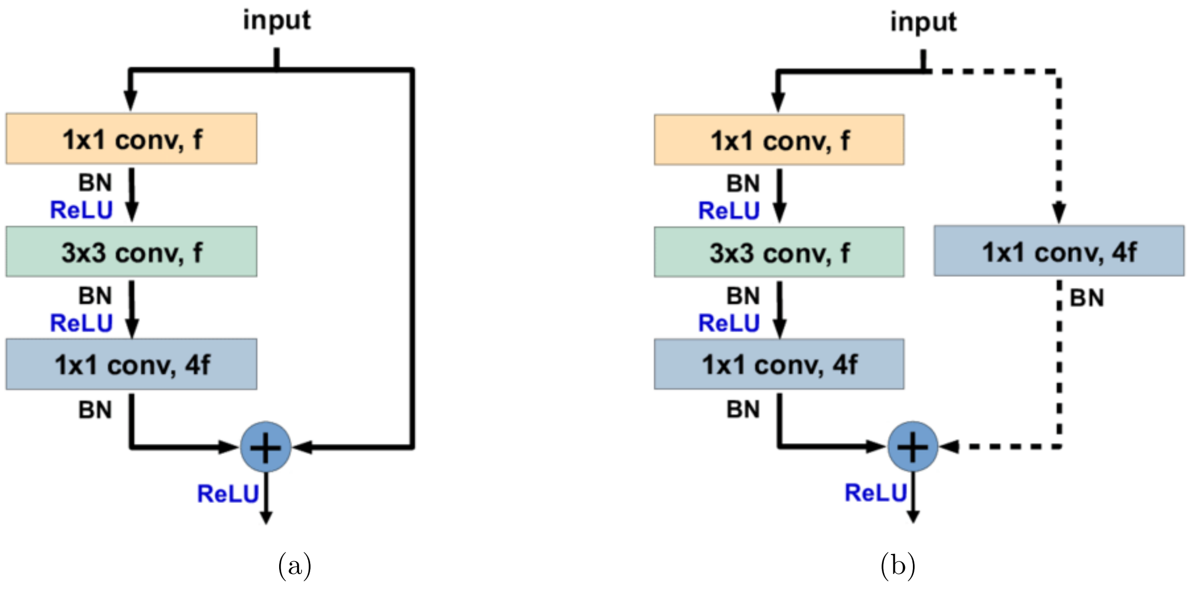


Figure 2.14: (a) residual identity block (b) residual convolutional block [9].

block and a residual block. In these equations $W^{(l+1)}$ stands for the weights of layer $l+1$ and $b^{(l+1)}$ is the bias unit for that layer.

$$
\begin{aligned}
z^{(l+1)} &= W^{(l+1)}a^{(l)} + b^{(l+1)} \\
a^{(l+1)} &= g(z^{(l+1)}) \\
z^{(l+2)} &= W^{(l+2)}a^{(l+1)} + b^{(l+2)} \\
a^{(l+2)} &= g(z^{(l+2)})
\end{aligned}
\tag{2.22}
$$

$$
\begin{aligned}
z^{(l+1)} &= W^{(l+1)}a^{(l)} + b^{(l+1)} \\
a^{(l+1)} &= g(z^{(l+1)}) \\
z^{(l+2)} &= W^{(l+2)}a^{(l+1)} + b^{(l+2)} \\
a^{(l+2)} &= g(z^{(l+2)} + a^{(l)})
\end{aligned}
\tag{2.23}
$$

These equations are valid for classic neural networks but they show the principle that is used.

These residual networks (ResNets) do not hurt the loss . This is because if they would hurt the performance, the back propagation would make the weight for $W^{(l+1)}$ and $W^{(l+2)}$ equal to zero. Thus $a^{(l+2)} = g(0 + a^{(l)}) = g(a^{(l)}) = a^{(l)}$ [8].

2.2.6 Inception networks

Another way to enhance the sorting performance of the classic convolution networks is using inception blocks. The basic idea of the inception blocks is using multiple filters on one matrix. An example of this is shown in figure 2.15. Typically the operations used are a 1×1 convolution, 3×3 convolution, 5×5 convolution and a max pooling layer. But this is not a rule [8].

The idea is that the network will compute all these operations and chooses the features that it likes the most. The algorithm chooses its own topology [8].

In figure 2.15 the convolutions are preceded by a 1×1 convolution and the max pooling block is followed by a 1×1 convolution. This is not necessary to implement but it reduces the computationally cost. These 1×1 convolutions are meant to reduce the number of features that the other convolutions have as input. The 1×1 convolution after the max pooling is there for matching the output dimensions with the other operations so that the different outputs can be concatenated [8].

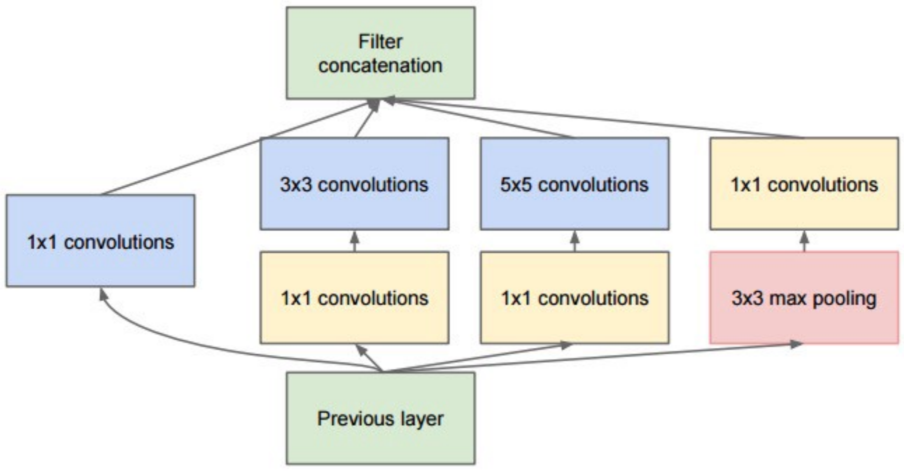


Figure 2.15: Example of an inception block [10].

2.2.6.1 Side branches

With the introduction of inception blocks, the main problem of deep learning is reintroduced. This means when performing back propagation the gradients that reach the middle or first layers are not substantial enough to make the learning efficient. The previous solution was using residual blocks. But with inception blocks this is not possible to implement because of the difficult output dimensions [8].

The proposed solution is the use of side branches. These are auxiliary classifiers that are located on other places in the network. These help enhancing the loss that is back propagated to these layers. The total loss that is used in these layers is a weighted sum of the loss of the auxiliary classifiers and the loss that is back propagated. Figure 2.16 shows such an inception net with side branches. The name of that model is the GoogleLeNet [11].

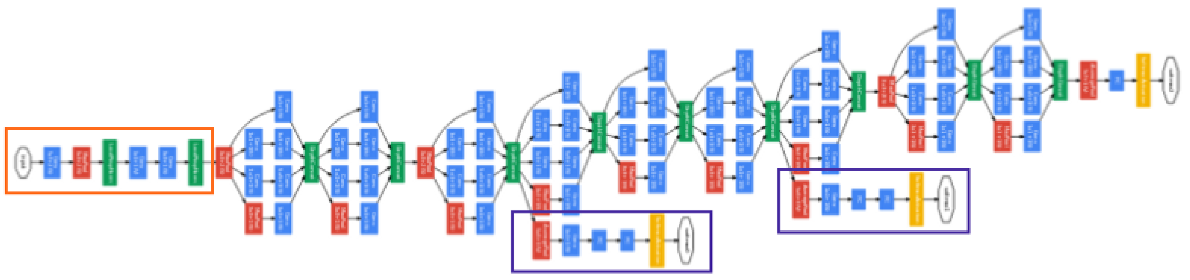


Figure 2.16: GoogLeNet, an example of an inception net with side branches [11].

2.2.7 Back propagation

The back propagation for a convolutional neural network uses the same principle as for a fully connected neural network. This principle says that the each layer passes the computed gradients to the previous layer which will use these gradients to compute their gradients [8].

For a convolutional neural network the equations and steps to perform back propagation are not trivial and are mostly handled by frameworks like TensorFlow or Theano. Which makes that the developer does not need to know these formulas. Therefore these will not be handled in this research [8].

2.3 Current technology

The current method for doing big batch image classification in the industry are based on hand-crafted low-latency image processing. Thus meaning that a sample product is sent through the machine and several threshold values are manually selected or changed to differentiate good from bad product. This has as advantage that the designers and production workers keep complete control over the sorting process and the amount of sample product needed can be kept very low. The disadvantage is that complex products with complex shapes and colours are very difficult to sort with a high accuracy.

Although most systems do not use machine learning based algorithms to classify the product, the idea to do so is not new. At Key Technology a new algorithm has been developed and is ready to unroll in the field. This algorithm has very good results with accuracy's over 99%, which means it has a precision and recall close to 1. This means that this algorithm can very precisely distinguish good from bad product. The model is based on a multiple layer neural network. This means that a large amount of weights need to

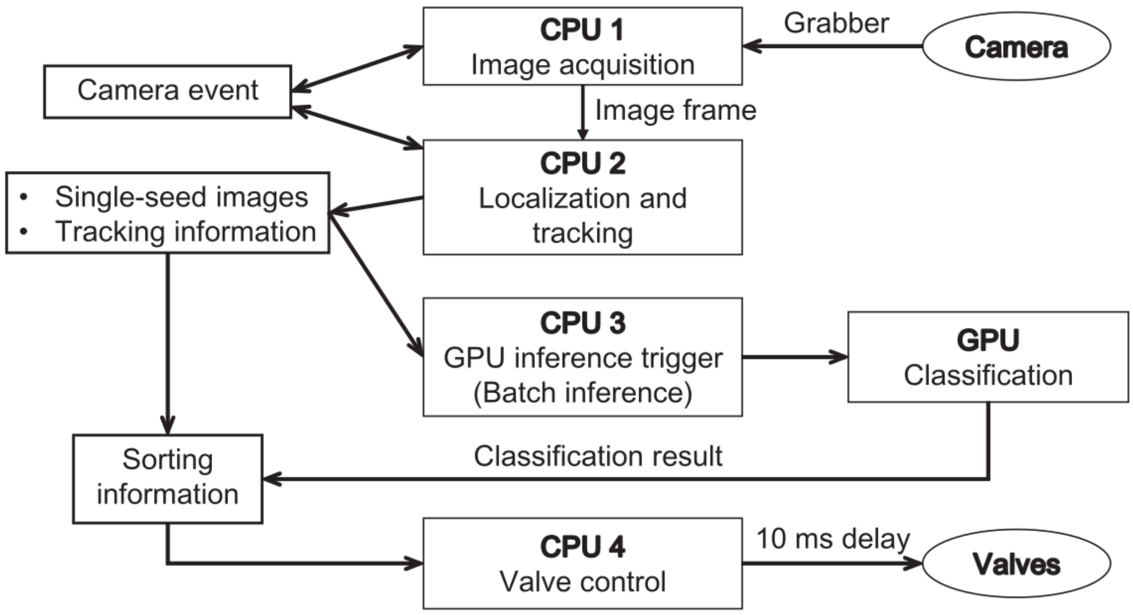


Figure 2.17: Hardware setup as proposed by [12]

be trained but nevertheless training of this algorithm is very quick. The big difference of this algorithm and the one developed in this research is the target machine. The machine of the neural network is the Veo2. This machine aligns the product in different streams that cause that different objects do not overlap with each other and every area of interest can only contain one object. The application has been designed to classify corn cobs.

Another interesting method is presented in [12]. In this research paper another method was proposed. The researchers only use the deep learning algorithm to classify the objects. The detection of objects in the images is done using naive low-latency image processing. This has as advantage that the latency of the algorithms is lower and there is an opportunity to parallelize the process. The hardware structure used is shown in figure 2.17. The model that is used in this research is the ResNet-18 model. This network is further explained in section 3.3.2. The results that this setup achieves are very good with a high purity and high throughput.

This machine also aligns the product in different streams and is designed to sort hemp from weed seeds [12].

Chapter 3

Method

3.1 Data set

For Deep Learning and machine learning applications assembling the data set is one of the most important tasks that needs to be performed. This means that a lot of attention goes in the construction of the data set.

Constructing the data set has been done in two main steps, data acquisition and data augmentation. The data acquisition has already been done by Key Technology because there are seasonally restrains in which the beans are processed in the field. The data augmentation is done in multiple steps that are explained below. This is done to enlarge the data set and make the model more stable.

The data provided by Key Technology consists of 291 image files. These have both images of the full stack setup with 12 channels, and images with only the red, green, blue and infrared channels. Figure 3.1 shows an example of the full stack setup. The only difference with the RGB setup is that the background is blue instead of black. The images from the RGB setup are only used if the final model is trained with 3 channels. For comparing the models the full stack images are used with either 8 channels or 3 channels.

3.1.1 KIF-format

The data set consists out of images saved in the KIF-format (Key Image Format). This is a custom file format created by Key Technology. This is done because the existing formats do not allow using more than 3 or 4 channels. Thus a new format has been defined: the KIF format. This format consists out of 3 files. An image_data.bin, line_headers.bin and meta_data.yaml. The image_data.bin contains the binary information of the picture. The meta_data.yaml contains the dimensions of the image, the channels used, the order of the channels and other data about the image. The line_headers.bin has no relevant use in this research.

3.1.2 Data augmentation

The images in the original data set contain multiple objects in one image. This can be seen in figure 3.1. Because object detection is not needed for this algorithm the individual objects need to be separated. For extracting the objects in the images a Python script

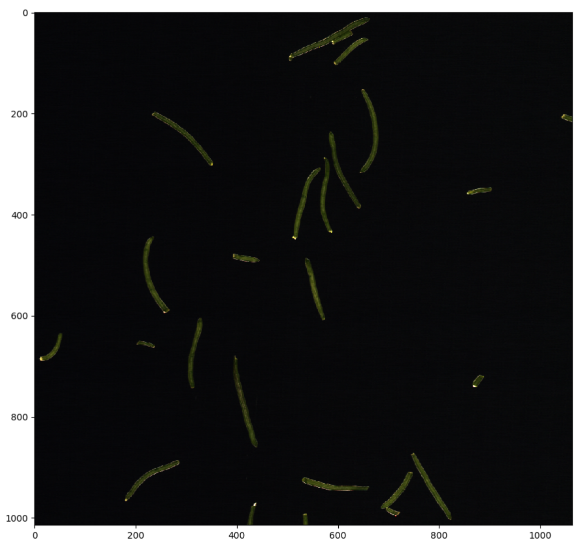


Figure 3.1: Example image from the data set.

has been created. This script uses the red channel to recognize the present objects. The sequence of operations is listed below.

1. Binarizing the image.
2. Filter the present noise with a Gaussian filter.
3. Generate a list of connected objects that are present in the image. This is done to detect and count the objects.
4. Segment the images into smaller images. These segmented images should contain only one object.
5. Save the segmented images according to the KIF-format.

Figure 3.2 shows the same image as is shown in figure 3.1. But in figure 3.2 the detected objects are surrounded by a bounding box. Important to note is that the objects that touch the edges of the picture are not used. This is done because they do not always show the complete object and cannot be sorted accurately. In the sorter these objects are stitched with objects from the neighbor images. But the data set provided contains screen-shots from a continuous image pipeline.

Another flaw of this script is that it recognizes a cluster of objects as one object. These clusters are separated from the other pictures because it is much more difficult to classify these pictures accurately. The reason is that only one defect in a cluster makes this cluster as a negative sample even if the other objects are good. This can cause confusion for the algorithm and this is why these images are not used at this stage of the research. Figure 3.3 shows an example of an segmented image.

The images that are generated from the segmentation are of a variable size. The height and width of these images can therefore not be predicted. For a convolutional network

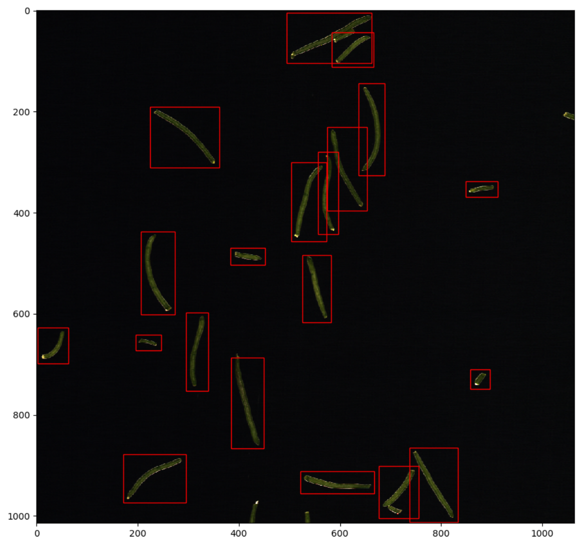


Figure 3.2: Example image with bounding box around detected objects.

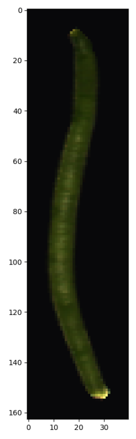


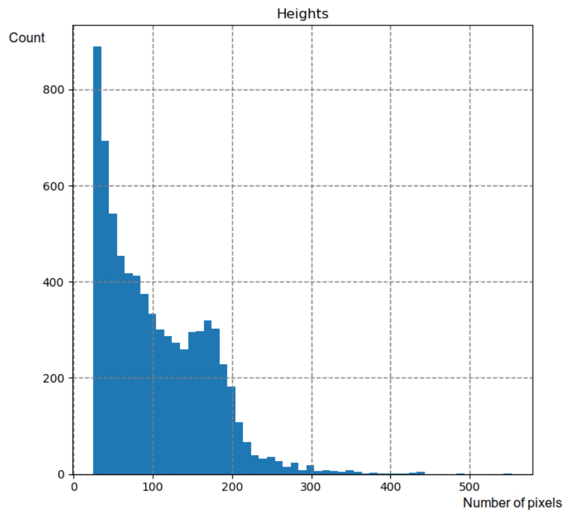
Figure 3.3: Example image of a single cut out bean.

this is not a problem because the computation is done using filters that move over the image. The result of this operation is that the output matrix has a variable size. This, however, is a problem for making a classification algorithm. Because the output matrix of the convolutional layers need to be converted to a predefined amount of output features. The consequence is that the input size needs to be defined before the network can be constructed.

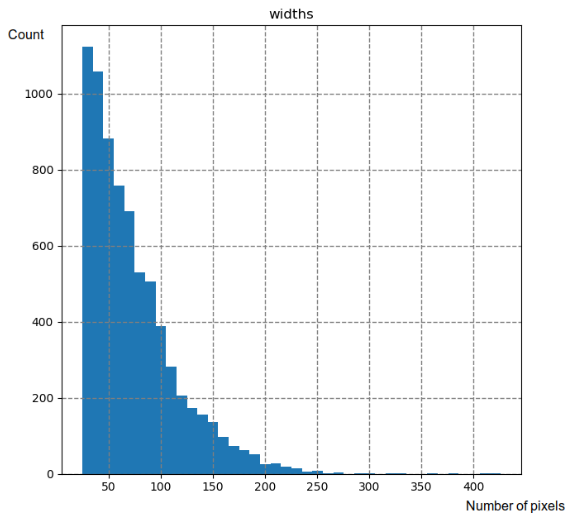
This problem can be solved by adding padding to the images until a predefined size is reached. The value that is used to fill the padding is dependent to the value that lays on the edges of the image. So if the image has a blue background, the padding value will also represent blue. It is certain that the values on the edge of the image are equal to the background data because the object in the images will never touch the edges of the images. This is ensured in the algorithm that segments the individual objects out of the complete images.

The next task is determining the preferred image size. The easiest way is adding padding until the original size of the original images is reached. Such a large image is represented in figure 3.1. Enlarging the image to this size is not good practice because in most cases the amount of background is too much. The consequence is that the algorithm needs to make a lot of computations on data that only contains background data. This makes the algorithm slow and does not contribute to the performance or accuracy.

To pick another size for the images two histograms are generated. The first histogram, shown in figure 3.4a, displays the value of the heights of the segmented images. The second histogram, figure 3.4b, displays the value of the widths.



(a)



(b)

Figure 3.4: histogram of (a) heights of the pictures (b) widths of the pictures.

From these histograms a preferred size of 250 $\times$ 250 pixels is picked. This size does not include all images in the data set. But it includes the most images without getting to large. To use the other images in training, a window of 250$\times$ 250 will be picked from these images. Although the effect of doing this on the performance of the algorithm is not clear. The benefit of using this method is, as mentioned before, that the speed of the algorithm will be higher.

The next steps in the data augmentation process are done inline with the training of the network. This is done because otherwise too much storage space would be needed to store all image data. This would increase the cost because new storage solutions would be needed. The speed of the training would also suffer because too much disk operations would be needed to load the data into the algorithm.

The next step in for the data augmentation process is rotating the objects in the images. This may be done because it is known that these objects are flying when the image is made. It is therefore unknown in which position or angle these objects will pass by the camera. Important to note is that the images do not contain background objects. If the images would contain background objects, such as trees or other objects, it would not be possible to rotate the images. The objects in the images are rotated in an angle that is randomly chosen between 0°and 360°.

The final step is displacing the objects. In the images that are used by the previous operations the object is centered around the center point of the image. If only these images are used for training the network it is possible that the network will learn that objects only occur in the middle of the image. This is not eligible, therefore the object in the images are displaced. This is done by shifting the object in a horizontal and vertical manner with a random value.

3.2 Hardware & Software setup

The second task is assembling an hardware setup and software platform to accommodate the learning. Because training a deep learning network requires a lot of computing resources a graphics processing unit (GPU) is required to speed up the process.

The specific hardware setup consists out of a Lenovo Think-pad with an Intel i7 vPro processor with 8 cores, a 16GB RAM memory and an NVidia Quadro K2100M GPU. The other hardware specifications do not have any importance in this research.

The software setup is made in the Ubuntu 16.04 LTS 64-bit operating system. For building and compiling the models Python 3.6 is used. For the deep learning models additional packages are used. These packages are TensorFlow and Keras.

TensorFlow is an open-source software library from Google. This library is specifically built for high performance numerical computations. This means that it is ideal for using in deep learning. TensorFlow also manages the communication with the GPU and supports all NVidia GPUs with compute capabilities higher than or equal to 3.0 [17].

The second package, Keras, is also an open-source library. This library is capable of running on top of TensorFlow. Keras is developed for fast prototyping with deep learning models [18].

Both of these libraries can be used in Python. Other libraries used are numpy, scipy, etc.

3.3 Models

3.3.1 Very basic Network

The very basic network has been created to validate the hardware and software setup. This has been done to ensure that possible problems with future models would not originate from the hardware and software. Therefore no real results are expected from this model.

The model structure is displayed in appendix A. This model consists out of 2 layers with some additional operations. In this network the padding used is always 'valid' to quickly reduce the size of the image.

- First layer: input layer with input size of 250 × 250 × 3 or 250 × 250 × 8

- First operation: Batch normalization. This normalization is computed on the entire batch, this means that the mean and standard deviation is computed over the entire batch of inputs
- Second operation: Maximum Pooling with a filter size of 3 × 3 and a stride of 2 × 2. This reduces the size to 124 × 124 × 3 or 124 × 124 × 8.
- Second layer: 1 × convolution to reduce the amount of channels to 1.
- Thirth operation: Maximum Pooling with a filter size of 6 × 6 and a stride of 4 × 4. The output size of this operation is 30 × 30 × 1
- Fourth operation: Flatten the matrix to 1 long vector of 900 values.
- Thirth layer: Fully connected layer: 900 inputs to 2 outputs.

3.3.2 ResNet-18

This network has been used because it was also used in [12]. But the network that is used in this research can still differ slightly from that used in [12] because the complete structure is never mentioned.

The model that has been implemented is shown in appendix B. This model is a residual network of 18 layers deep. Because this network is quite deep it should be able to learn to recognize complex functions.

The model is built with 2 important blocks, an identity block and a convolutional block. The identity block is a residual block where the input dimensions match the output dimensions. This means that the padding used in this block is 'same' padding to preserve the image dimensions. The identity block consists out of 2 convolutions in series. The filter size of the convolutions in this block is always 3 × 3 and the stride is always 1.

The convolutional block is used when the dimensions should be reduced and thus the input and output do not have the same dimensions. This block also has 2 convolutions in series. The first convolution has 'valid' padding, a filter size of 3 × 3 and a stride of 2 × 2. The second convolution has 'same' padding, also a filter size of 3 × 3 and a stride of 1 × 1. With these convolutions a thirth convolution runs in parallel. This is a 1 × 1 convolution with a stride of 2 × 2 and 'same' padding.

The actual structure that is used is listed below:

- Input layer
- Zero Padding (increase image size with 3 in every direction)
- 7 × 7 convolution with 64 filters, a stride of 2 × 2 and 'valid' padding
- Max Pooling with a filter size of 2 × 2 and a stride of 2 × 2
- Identity block with 64 filters
- Identity block with 64 filters
- Convolutional block with 128 filters
- Identity block with 128 filters

- Convolutional block with 256 filters
- Identity block with 256 filters
- Convolutional block with 512 filters
- Identity block with 512 filters
- Average Pooling with a filter size of 7×7 and a stride of 1×1
- Flatten the image to a long vector
- Fully connected layer with 2 outputs

3.3.3 ResNet-50

The ResNet-50 network is like the ResNet-18 model a residual network build with identity and convolutional blocks. The main difference between the ResNet-50 and ResNet-18 architecture is the amount of layers. The ResNet-18 architecture is 18 layers deep while the ResNet-50 architecture is 50 layers deep.

The larger quantity of layers should mean that the ResNet-50 model will be able to learn to recognize more complex features. This means that very complex shapes and colour schemes will be used to classify the product. The disadvantage is that training will take longer and more data will be needed to train the ResNet-50 model than to train the ResNet-18 model.

The convolutional block is build out of 3 convolutions in series and 1 convolution in parallel with the other 3. The first of the three convolutions is a 1×1 convolution with a stride that is an argument in the convolutional block. The second convolution is 3×3 convolution with a 1×1 stride, and the thirth convolution is also a 1×1 convolution with a 1×1 stride. The convolution in parallel is a 1×1 convolution with a the same stride as the first convolution.

The identity block has only three convolutions. The first convolution is a 1×1 convolution with a 1×1 stride. The first convolution is a 3×3 convolution with 1×1 stride, and the thirth convolution is a also a 1×1 convolution with a 1×1 stride.

The amount of filters in both the convolutional or identity block is given in a list of 3 numbers. In the convolutional block the amount of filters in the parallel convolution is the same as the amount of filters in the last convolution, this to match the dimensions. In ResNet-50 model structure is listed below:

- Input layer
- Zero Padding (increase image size with 3 in every direction)
- 7×7 convolution with 64 filters, a stride of 2×2 and 'valid' padding
- Max Pooling with a filter size of 3×3 and a stride of 2×2
- Convolutional block with [64, 64, 256] filters and stride of 1×1
- Identity block with [64, 64, 256] filters
- Identity block with [64, 64, 256] filters

- Convolutional block with [128,128,512] filters and stride of 2 × 2
- Identity block with [128,128,512] filters
- Identity block with [128,128,512] filters
- Identity block with [128,128,512] filters
- Convolutional block with [256,256,1024] filters and stride of 2 × 2
- Identity block with [256,256,1024] filters
- Identity block with [256,256,1024] filters
- Identity block with [256,256,1024] filters
- Identity block with [256,256,1024] filters
- Identity block with [256,256,1024] filters
- Convolutional block with [512,512,2048] filters and stride of 2 × 2
- Identity block with [512,512,2048] filters
- Identity block with [512,512,2048] filters
- Average Pooling with a filter size of 2 × 2 and a stride of 1 × 1
- Flatten the image to a long vector
- Fully connected layer with 2 outputs

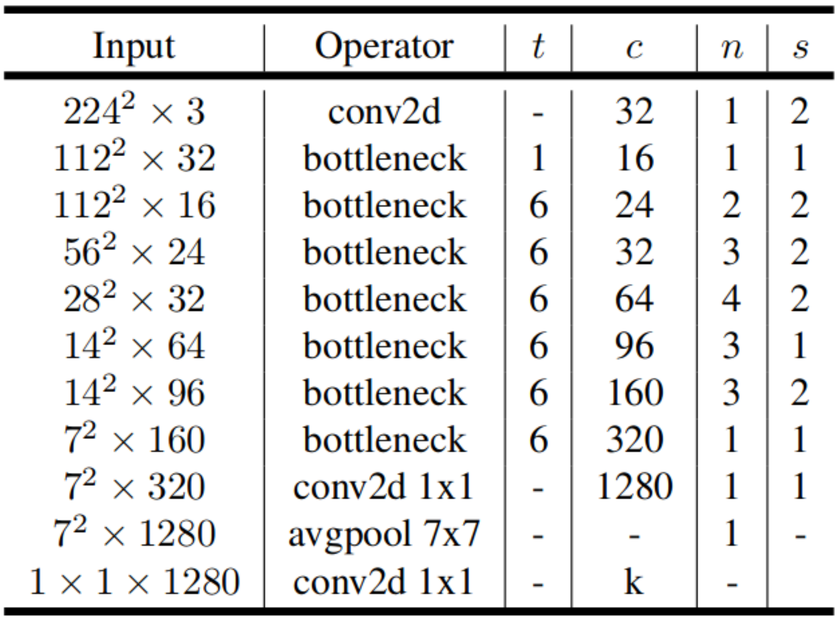
This network is relative deep and thus would need a lot of data to train it. Therefore transfer learning is used. This means that the network is trained on another data set. The result of this training is found on the internet. The weights of this pre-trained network are loaded and only the last layer (the fully connected layer) is trained with the custom data set.

The data set that is used to train the algorithm the first time is the ImageNet data set. Because this data set consists out of ordinary RGB images the pre-trained network cannot be used to classify the custom data set with 8 channels. Thus only 3 channels can be used with the pre-trained model.

3.3.4 MobileNetV2

The MobileNetV2 architecture is also implemented as standard in the Keras library. The pretrained weights for the ImageNet data set are also available in the Keras library. This model is the second version of the MobileNet architecture which was developed by Google and uses even fewer parameters. This architecture is designed to run on mobile devices thus having a lower computational load than other models. This can speed up the classification process which would benefit the amount of product that can be sorted in a specific time frame [13].

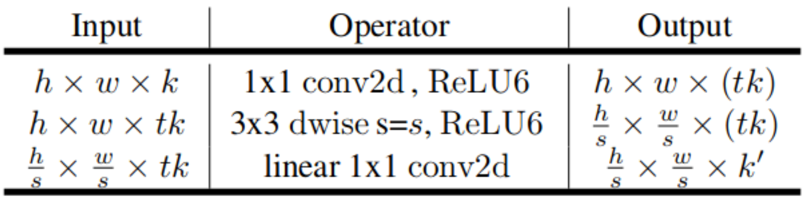
The complete architecture is displayed in figure 3.5. The main difference with other residual networks is the structure of the residual block. The research team at Google succeeded in using less parameters to train the network without losing accuracy. For further information on the exact MobileNetV2 architecture reference is made to [13].



Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	

Figure 3.5: MobileNetV2: Each line describes a sequence of 1 or more identical (modulo stride) layers, repeated n times. All layers in the same sequence have the same number c of output channels. The first layer of each sequence has a stride s and all others use stride 1. All spatial convolutions use 3 X 3 kernels. The expansion factor t is always applied to the input size [13].

In figure 3.5 the basic building block is the bottleneck block. The MobileNetV2 architecture is the first network that uses the bottleneck block. The bottleneck building block contains three other convolutional blocks. This is visible in figure 3.6. In this figure the dwise or depth-wise convolution stands for a convolution that has the same amount of filters than input channels [13].



Input	Operator	Output
$h \times w \times k$	1x1 conv2d , ReLU6	$h \times w \times (tk)$
$h \times w \times tk$	3x3 dwise s=s, ReLU6	$\frac{h}{s} \times \frac{w}{s} \times (tk)$
$\frac{h}{s} \times \frac{w}{s} \times tk$	linear 1x1 conv2d	$\frac{h}{s} \times \frac{w}{s} \times k'$

Figure 3.6: Bottleneck block with k input channels, k' output channels, expansion rate t and a stride s [13].

The MobileNetV2 architecture is a deep neural network because it has multiple convolutional layers. But the slight difference in the residual block structure has as advantage that it has far less parameters to train which has a computationally benefit over other networks of the same size this make them ideal for using in mobile devices and potential big batch sorting applications [13].

3.4 Experiments

All experiments are done on a data set that consists out of 4568 samples. Because of this small size no validation set is used. Thus the data set is separated in a training set and a test set. This is done with a ratio of 80/20. The training set contains 80% of the data set and the test set contains 20%. This separation is done without overlapping examples.

The data augmentation is done inline with the training. It is therefore very challenging the predict how many extra samples this creates. This is dependent on the amount of epochs that is trained on. An epoch means that the entire training set is used once in the training. When using data augmentation every sample of the training set generates one augmented sample. This means that the amount of samples that is used during every epoch is the same as without data augmentation.

Because the data set is already quite small for Deep Learning applications only 2 classes are used to classify. Namely 'good' and 'bad' product. This classification only needs 1 output feature to make this classification. It would then output '0' for good product and '1' for bad product. This method is not used. The chosen method was to use 2 output features. This was done because when a larger data set is used and the different defects need to be classified it would be easier to transform the networks and algorithms. The use of 2 output features results in another output vector. The output vector for good product is [1 0] and the output vector for bad product is [0 1].

3.4.1 Very basic network

The Very basic architecture is developed to test of the software and hardware setup works correctly. Nonetheless the network was trained on images with 3 and 8 channels. To do this some slight modifications are made. The differences between the two networks are found in the input layer and the second layer (1×1 convolution). The input layer has an input size of $250 \times 250 \times 3$ or $250 \times 250 \times 8$. In the convolution layer the difference is a bit more significant. The filter sizes are $1 \times 1 \times 3$ or $1 \times 1 \times 8$. To initialize the network random values are used. This network was trained during 10 or 100 epochs.

3.4.2 ResNet-18

The ResNet-18 network was custom implemented and thus not integrated via the Keras library. Also no decent pretrained weights where found and thus the network is initialized with random values and is trained during 10 or 100 epochs

This network was also trained on images with 3 or 8 channels. The difference between these two networks is found in the input layer and the first convolution. The input layer has an input size of $250 \times 250 \times 3$ or $250 \times 250 \times 8$ like the very basic architecture. The first convolutional layer has a filter size of $7 \times 7 \times 3$ or $7 \times 7 \times 8$.

3.4.3 ResNet-50

The ResNet-50 network was also custom implemented but for this network pretrained weights on the ImageNet data set where found. This means that transfer learning is possible. Because this pretrained weights are trained on images with 3 channels it is challenging to implement images with 8 channels. Therefore when using transfer learning only images with 3 channels are used.

The network is thus initialized in two different ways. The first one is initializing the network with random values. This makes that the network can be trained on images with 3 or 8 channels. The difference between these two networks is the same as with the ResNet-18 networks. Namely the input layer has a size of $250 \times 250 \times 3$ or $250 \times 250 \times 8$. And the first convolution has filter sizes of $7 \times 7 \times 3$ or $7 \times 7 \times 8$.

The second method to initialize this network is loading the pretrained weights. This is done for all layers except the last (fully-connected) layer. For this layer no weights can be loaded because the output is customized. The last layer is thus initialized with random values. This network is trained with two methods. The first method is training only the last layer. The second method is training the last layer and also retraining all the other layers.

This network is only trained during 10 epochs because of the long training time.

3.4.4 MobileNetV2

The MobileNetV2 networks is the only network that is loaded from the Keras library. This means that for this pretrained weights are also available to make transfer learning possible.

This network is also initialized with random values or pretrained values. When the network is initialized with random values it is trained on images with both 3 or 8 channels. When the network is initialized with pretrained weights it is only trained on images with 3 channels. Although a difference is made with training all layers or only the last layer.

This network is trained during 10 epochs and 100 epochs.

Chapter 4

Results

This chapter will discuss the architectures and their different implementations that are mentioned in chapter 3. The result in this chapter are visualized with graphs. The numerical values are summarized in tables that are found in appendix C.

4.1 Very Basic architecture

Figure 4.1 shows the accuracy and F1-score of the very basic architecture which was trained during 10 epochs and on images with 3 channels. In this figure it can be seen that the accuracy is straight away above the 90%. This is not unexpected because the data set consists out of 90% good examples and 10% bad examples. Therefore the F1-score is more important.

Both graphs show a lot of fluctuations. This is normal. These fluctuations are present because mini-batch gradient descent is used. This means that not all images are used to make a training step. Only a few are used. In the case of this network the batch size is 32 samples. The approach makes for faster learning but introduces the fluctuations.

Figure 4.2 shows the accuracy and F1-score of the very basic architecture which was trained during 10 epochs and on images with 8 channels. These graphs show also the same fluctuations as in figure 4.1. The batch size is also 32 samples. This batch size is continued for all Very basic networks.

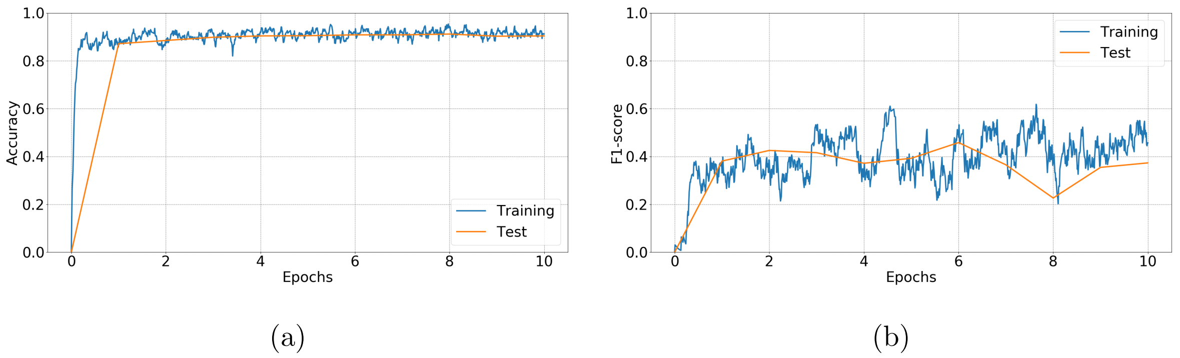
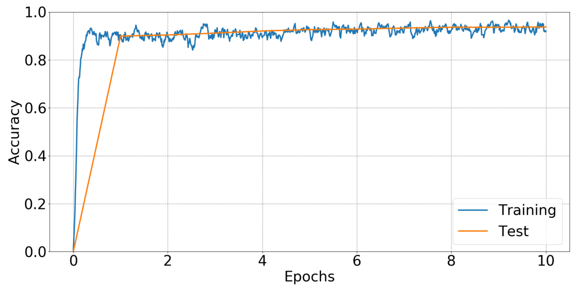
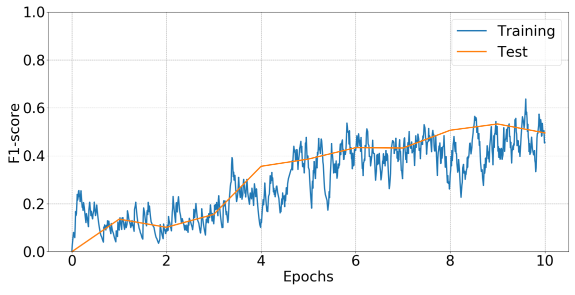


Figure 4.1: Metrics of the very basic architecture trained on 10 epochs with 3 channels and no data augmentation (a) Accuracy (b) F1-score



(a)



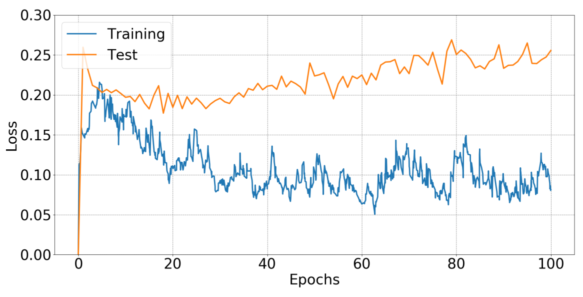
(b)

Figure 4.2: Metrics of the very basic architecture trained on 10 epochs with 8 channels no data augmentation (a) Accuracy (b) F1-score

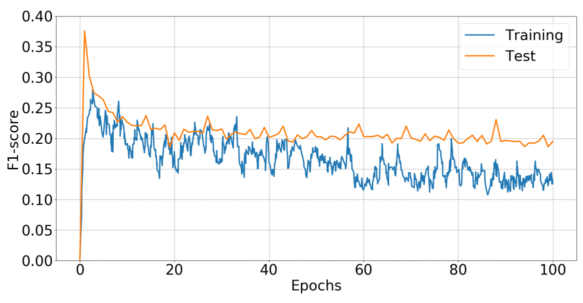
Figure 4.3 visualizes the loss of the very basic networks that are trained with images with 3 and 8 channels. The networks for figure 4.3 are also trained during 100 epochs.

In figure 4.3 it is clearly shown that the model is overfitting the training set. This is clear because the loss on the training set is falling while the loss on the test set is falling slower or even climbing.

Figures 4.4a and 4.4b respectively show the accuracy and F1-score on the test set of the very basic model trained during 100 epochs on images with 3 and 8 channels.

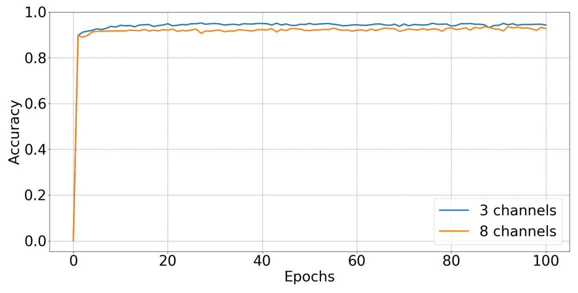


(a)

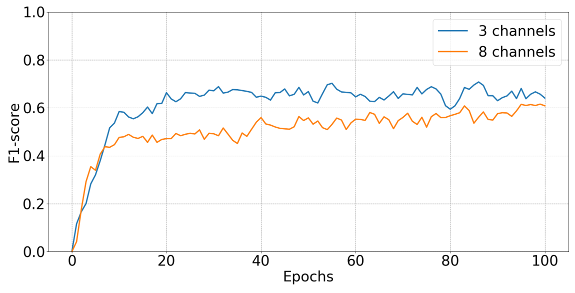


(b)

Figure 4.3: Loss of the very basic architecture trained on 100 epochs and no data augmentation (a) 3 channels (b) 8 channels



(a)



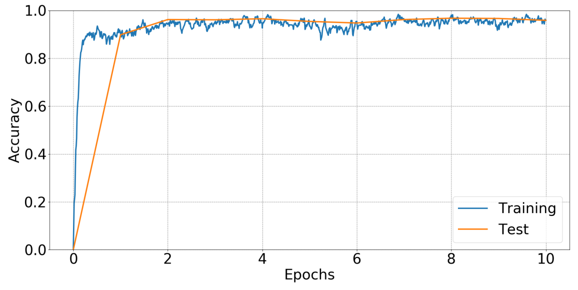
(b)

Figure 4.4: Metrics of very basic network with 3 and 8 channels trained on 100 epochs and no data augmentation (a) Accuracy (b) F1-score

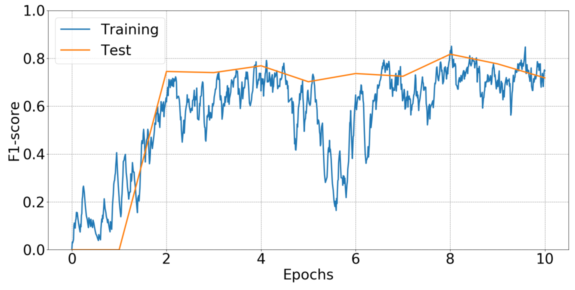
4.2 ResNet-18 architecture

Figures 4.5 and 4.6 show the accuracy and F1-score of the ResNet-18 network trained during 10 epochs and trained on images with respectively 3 and 8 channels. The batch size for all ResNet-18 networks is 32 samples.

In these images the fluctuations are also clearly visible during the training of the network. And with this network is the accuracy directly very high like expected.

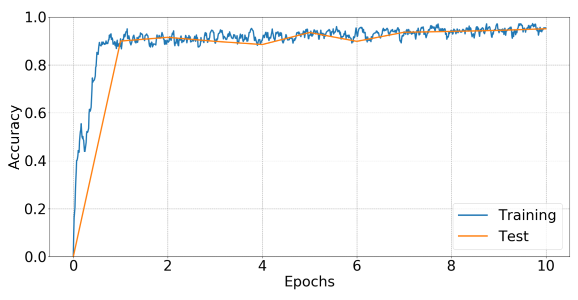


(a)

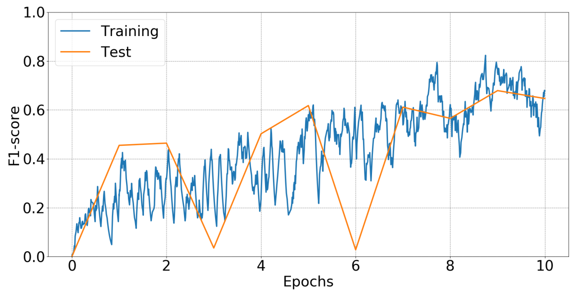


(b)

Figure 4.5: Metrics of the ResNet-18 architecture trained on 10 epochs with 3 channels and no data augmentation (a) Accuracy (b) F1-score



(a)

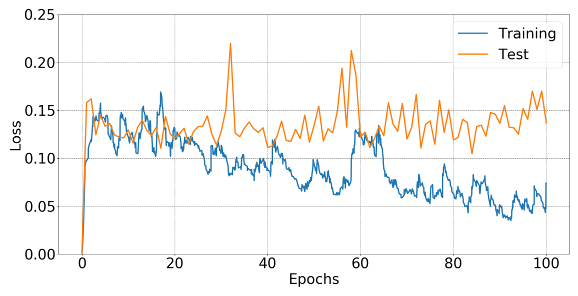


(b)

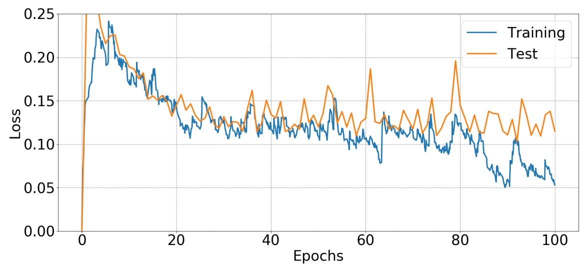
Figure 4.6: Metrics of the ResNet-18 architecture trained on 10 epochs with 8 channels and no data augmentation (a) Accuracy (b) F1-score

In figure 4.7 the loss of the ResNet-18 networks is visualized for the networks that are trained during 100 epochs and trained on images with respectively 3 or 8 channels. In figure 4.7a overfitting is clearly visibly. In figure 4.7b the overfitting is also present, but it is a bit more subtle. If training would continue this overfitting would become more and more visible.

Figures 4.8a and 4.8b show respectively the accuracy and F1-score on the test set of the ResNet-18 that was trained during 100 epochs on images with 3 and 8 channels.

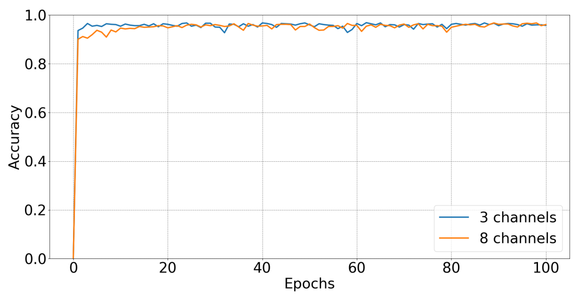


(a)

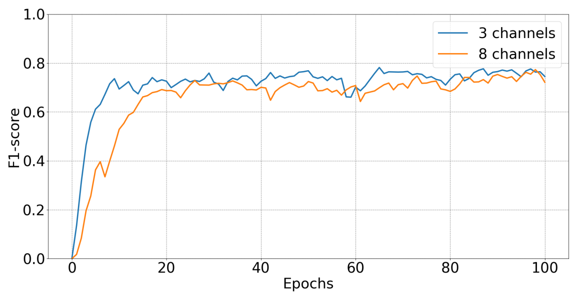


(b)

Figure 4.7: Loss of the ResNet-18 architecture trained on 100 epochs and no data augmentation (a) 3 channels (b) 8 channels



(a)

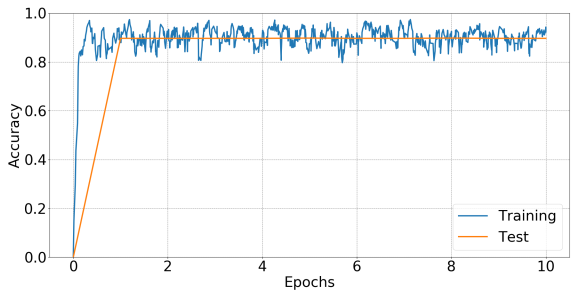


(b)

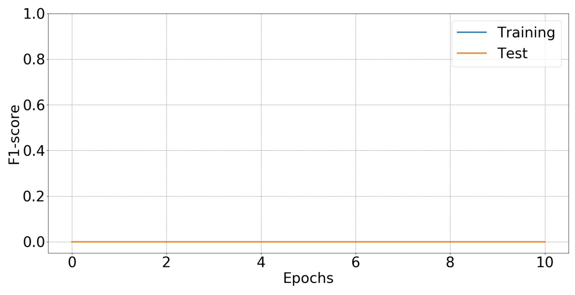
Figure 4.8: Metrics of ResNet-18 network with 3 and 8 channels trained on 100 epochs and no data augmentation (a) Accuracy (b) F1-score

4.3 ResNet-50 architecture

Figures 4.9 and 4.10 show the accuracy and F1-score of the ResNet-50 architecture that was trained during 10 epochs and on images with respectively 3 and 8 channels. In these images it is clearly shown that the F1-score stays at zero and the accuracy fluctuates around the same value. Namely 89.7% which is approximately the ratio of good to bad values in the data set. The batch size for all ResNet-50 networks is 8 because of memory constrains. This was due to a lack of RAM on the GPU.

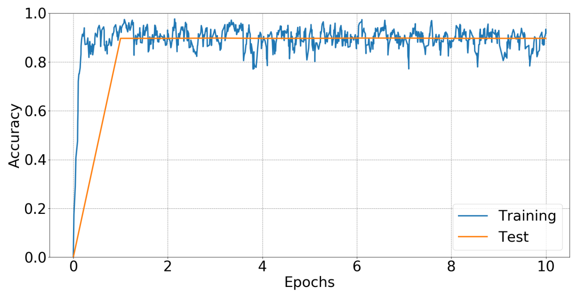


(a)

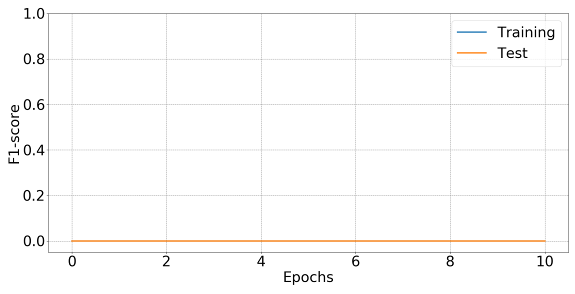


(b)

Figure 4.9: Metrics of the ResNet-50 architecture trained on 10 epochs with 3 channels and no data augmentation (a) Accuracy (b) F1-score



(a)



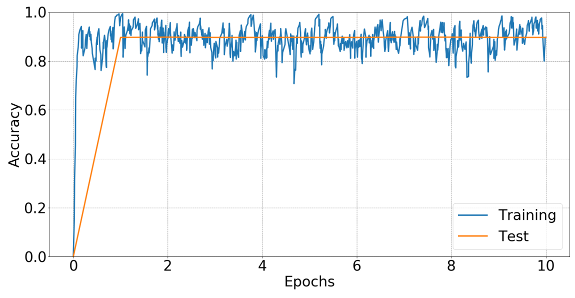
(b)

Figure 4.10: Metrics of the ResNet-50 architecture trained on 10 epochs with 8 channels and no data augmentation (a) Accuracy (b) F1-score

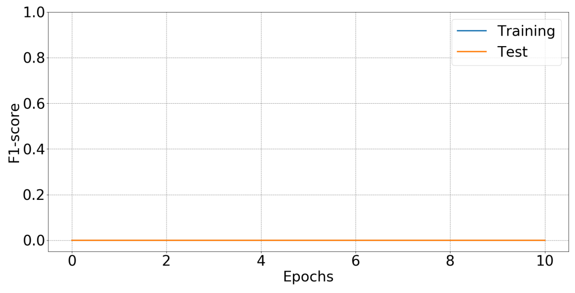
Figures 4.11 and 4.12 show the accuracy and F1-score of the ResNet-50 network initialized with pretrained weights, trained during 10 epochs and respectively only the last layers or all layers are retrained.

Although is was expected that this pretrained network would perform better than the network with randomly initialized values no increase of performance is detected. The F1-score is still zero and the accuracy fluctuates around 89.7%.

These models are only trained on 10 epochs because of the training time. To train during 1 epoch for any ResNet-50 model approximately 20 minutes are needed. For training a ResNet-50 network during 100 epochs more than 2000 minutes are needed.

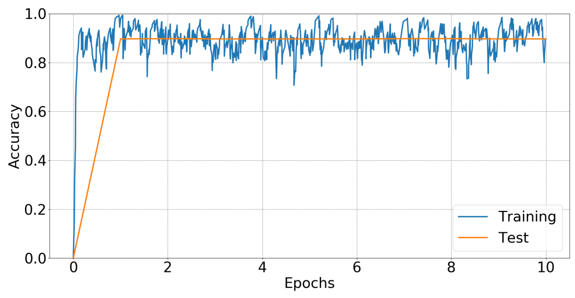


(a)

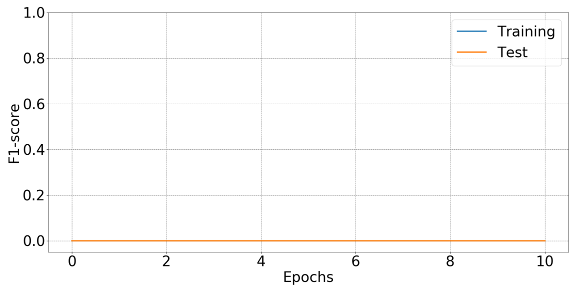


(b)

Figure 4.11: Metrics of the pretrained ResNet-50 architecture trained on 10 epochs with 3 no data augmentation and only the last layer is retrained (a) Accuracy (b) F1-score



(a)



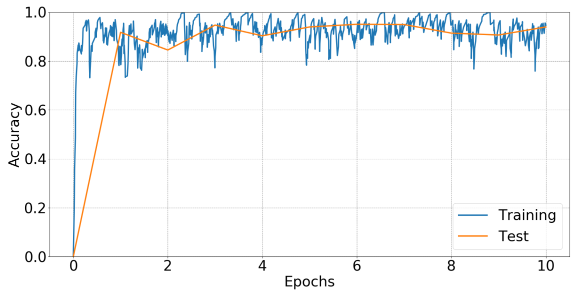
(b)

Figure 4.12: Metrics of the pretrained ResNet-50 architecture trained on 10 epochs with 3 no data augmentation and complete network is retrained (a) Accuracy (b) F1-score

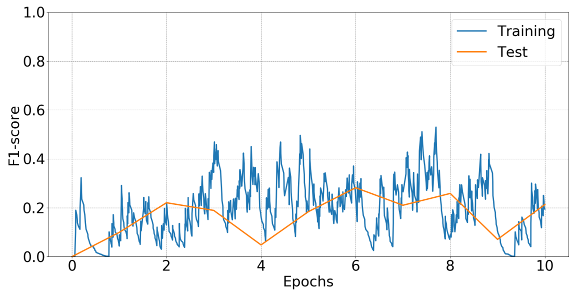
4.4 MobileNetV2 architecture

Figures 4.13 and 4.14 show the accuracy and f1-score of the MobileNetV2 networks which are trained during 10 epochs on images with respectively 3 and 8 channels. In these images it is clear that these models perform very good on accuracy but poorly on the F1-score. This eventually means that the algorithm does not perform very well.

The batch size for the MobileNetV2 network is 4. This is also needed because of the shortage of RAM on the GPU.

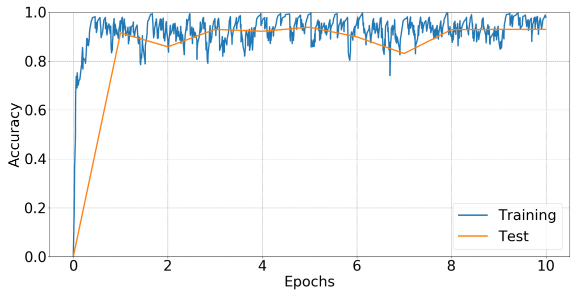


(a)

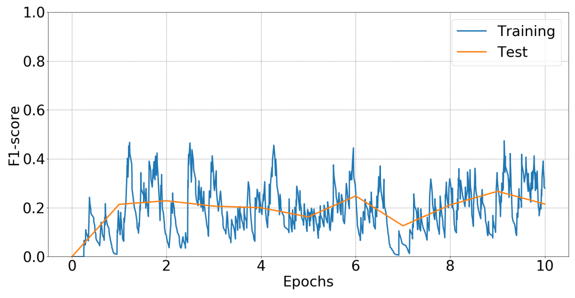


(b)

Figure 4.13: Metrics of the MobilenetV2 architecture trained on 10 epochs with 3 channels and no data augmentation (a) Accuracy (b) F1-score



(a)

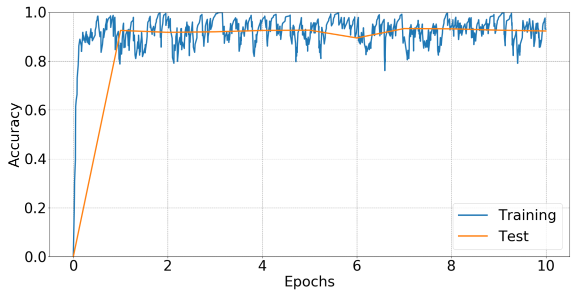


(b)

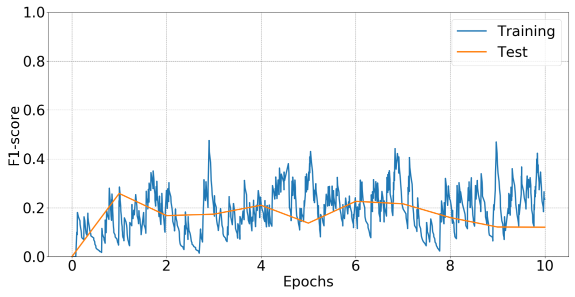
Figure 4.14: Metrics of the MobilenetV2 architecture trained on 10 epochs with 8 channels and no data augmentation (a) Accuracy (b) F1-score

In figures 4.15 and 4.16 show also the accuracy and F1-score of the MobileNetV2 networks. But these figures are from the networks that are initialized with pretrained weights and respectively from the networks where only the last layer or all the layers are retrained.

These figures show approximately the same as in the previous figures for the MobileNetV2. Namely that the accuracy is relatively high but the F1-score is not very high.

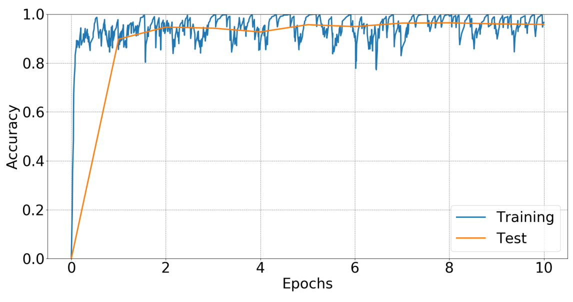


(a)

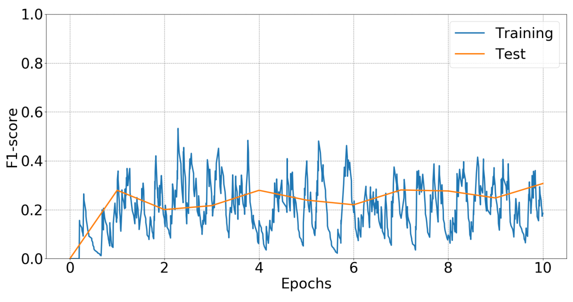


(b)

Figure 4.15: Metrics of the pretrained MobilenetV2 architecture trained on 10 epochs with no data augmentation and only the last layer is retrained (a) Accuracy (b) F1-score



(a)



(b)

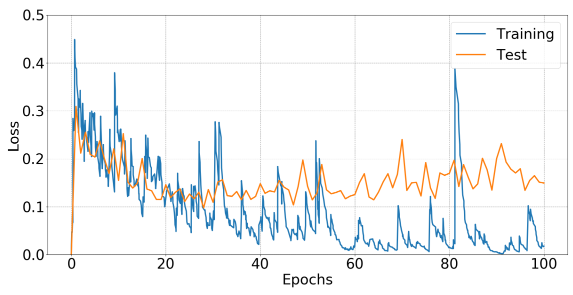
Figure 4.16: Metrics of the pretrained MobilenetV2 architecture trained on 10 epochs with no data augmentation and the complete network is retrained (a) Accuracy (b) F1-score

From the four MobileNetV2 networks two networks are chosen that are trained during 100 epochs. The two networks that were chosen are the MobileNetV2 network that was trained on images with 8 channels and the MobileNetV2 network that was initialized with pretrained weights with all layers retrained. These were chosen because they seem to climb the most stable.

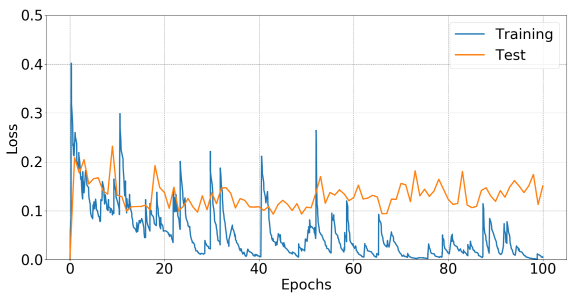
Figure 4.17 visualizes the loss of the two networks mentioned above. Figure 4.17a shows the loss of the model trained on images with 8 channels and figure 4.17b shows the loss of the pretrained model with all layers retrained.

In these figures the overfitting is clearly visible. This another demonstration that the data set is not sufficient for a deep learning application.

Figures 4.18a and 4.18b show the accuracy and F1-score of the test set for the two MobileNetV2 models that were trained during 100 epochs.

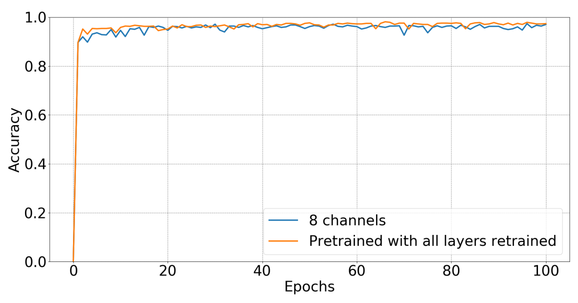


(a)

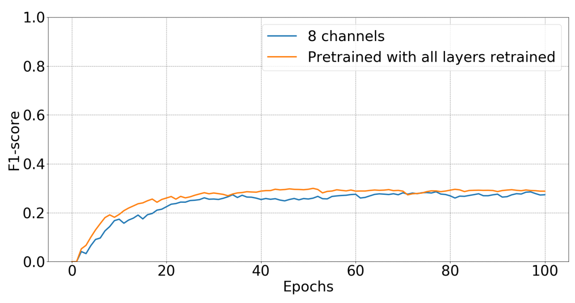


(b)

Figure 4.17: Loss of the MobilenetV2 architecture trained on 100 epochs with no data augmentation (a) 8 channels (b) pretrained with all layers retrained



(a)



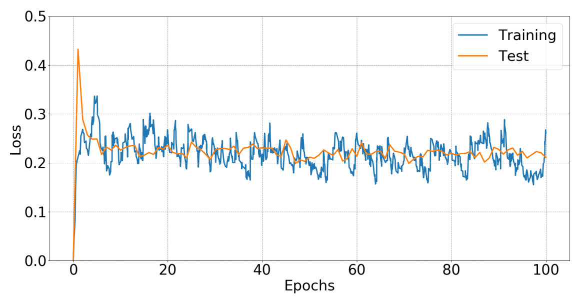
(b)

Figure 4.18: Metrics of the MobilenetV2 architecture trained on 100 epochs and no data augmentation with 8 channels or pretrained with all layers retrained (a) Accuracy (b) F1-score

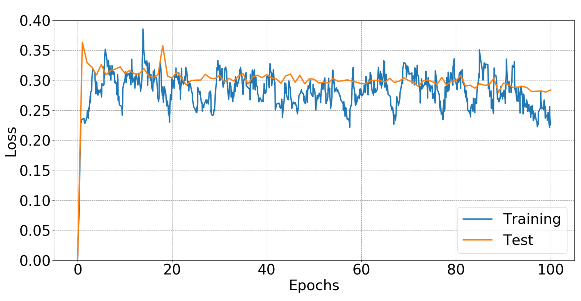
4.5 Data augmentation

Overfitting is visible with all models that are trained during 100 epochs. Therefore data augmentation is a way of reducing this overfitting. This data augmentation introduces 'new' data samples. For every sample that is used during training one augmented sample is generated. Therefore the amount of generated augmented samples is dependent on the amount of epochs the model is trained.

The training with data augmentation is only performed on the Very basic architecture and the ResNet-18 architecture. Figure 4.19 shows the loss of the training with data augmentation on the very basic models. This training was done during 100 epochs and on images with 3 channels and 8 channels. Figure 4.20a shows the accuracy on the test set of this training and figure 4.20b shows F1-score on the test set of the same training.

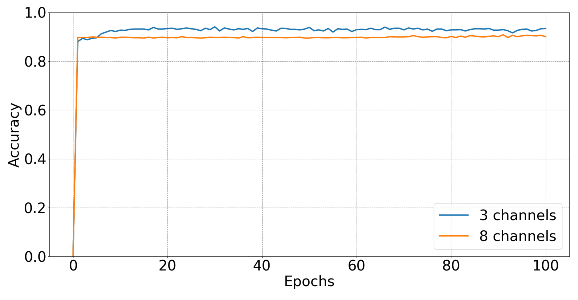


(a)

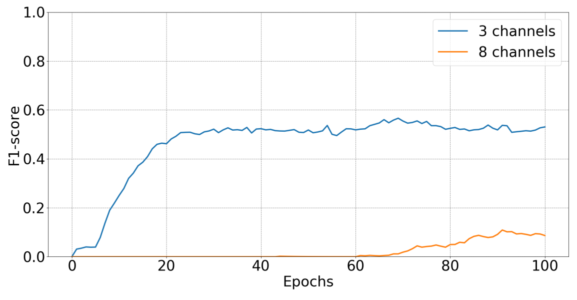


(b)

Figure 4.19: Loss of the very basic architecture trained on 100 epochs and no data augmentation (a) 3 channels (b) 8 channels



(a)

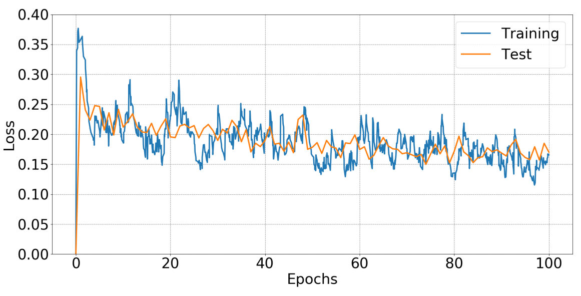


(b)

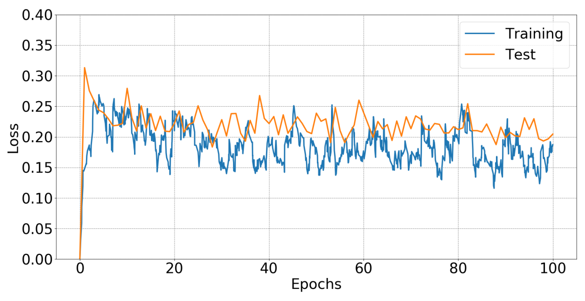
Figure 4.20: Metrics of very basic network with 3 and 8 channels trained on 100 epochs and with data augmentation (a) Accuracy (b) F1-score

Figures 4.21 shows the loss of the ResNet-18 models which are trained with data augmentation on 100 epochs for respectively images with 3 or 8 channels. And figures 4.22a and 4.22b respectively show the accuracy and F1-score on the test set of the same training.

From these figures it is clear that the overfitting is gone with the introduction of the data augmentation. This clearly shows why data augmentation is a necessary step in this research.

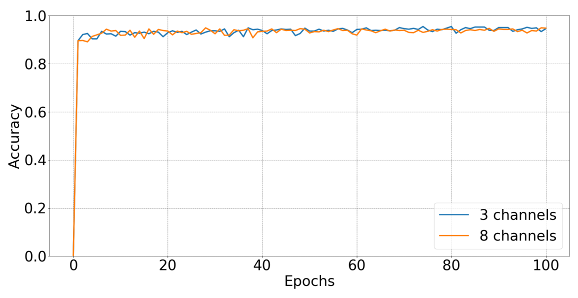


(a)

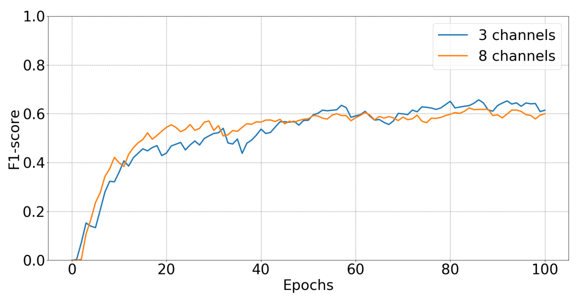


(b)

Figure 4.21: Loss of the ResNet-18 architecture trained on 100 epochs and no data augmentation (a) 3 channels (b) 8 channels



(a)



(b)

Figure 4.22: Metrics of ResNet-18 network with 3 and 8 channels trained on 100 epochs and with data augmentation (a) Accuracy (b) F1-score

Chapter 5

Discussion

	Accuracy	F1-score
Predict all [1 0]	0.900	0.000
Predict all [0 1]	0.182	0.100
Random prediction with 90% [1 0] and 10% [0 1]	0.828	0.105

Table 5.1: Relevant metric values

In table 5.1 metrics for 3 cases are listed. These values are convenient to interpret the result that are listed in chapter 4.

From the results it clearly shows that when the data set is used without data augmentation overfitting is occurring relatively fast. This was as expected and confirms that the data set is too small to train an accurate network.

The results also show that the Very basic network has better results than expected. This is strange because only 2 layers are used that can isolate meaningful features from the images. This is normally too small to make a good classification. But when comparing with table 5.1 it shows that this network is better then a random prediction.

The ResNet-50 model is, in both the pretrained version or the version that is initialized with random values, to big to make a decent prediction. When comparing with table 5.1 it shows that the ResNet-50 model always outputs [1 0] which will cause a 90% accuracy but an F1-score of 0.0. When this network is trained with a large enough data set it is expected that the pretrained versions will perform the best.

With all models it shows that when the model is trained on images with 8 channels the performance is lower. This is normal because this model has more features to train. But the model with 8 channels seems to raise and stay more stable than the other models.

When looking at table 5.1 it is clear that all networks except the ResNet-50 models perform better than a random classifier which predicts [1 0] in 90% of the cases.

When comparing all networks (that are trained during 100 epochs), which is done in figures 5.1 and 5.2, it show that the MobileNetV2 has the best accuracy but when F1-score is considered it is very low. The ResNet-18 is the best all-round model. It has the highest F1-score and a good accuracy.

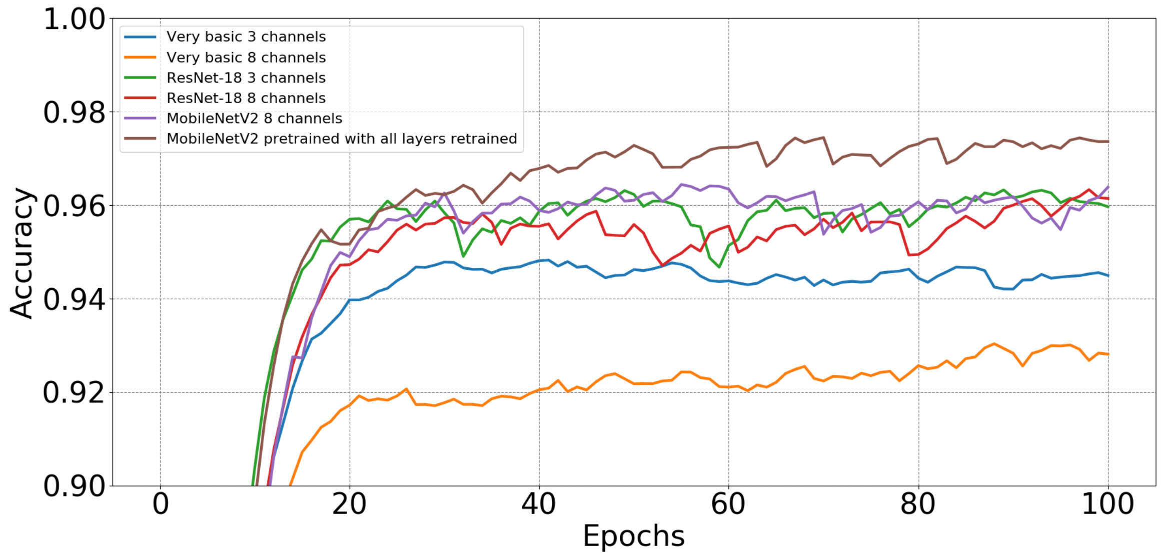


Figure 5.1: Comparison of accuracy of all the networks that where trained with 100 epochs and no data augmentation

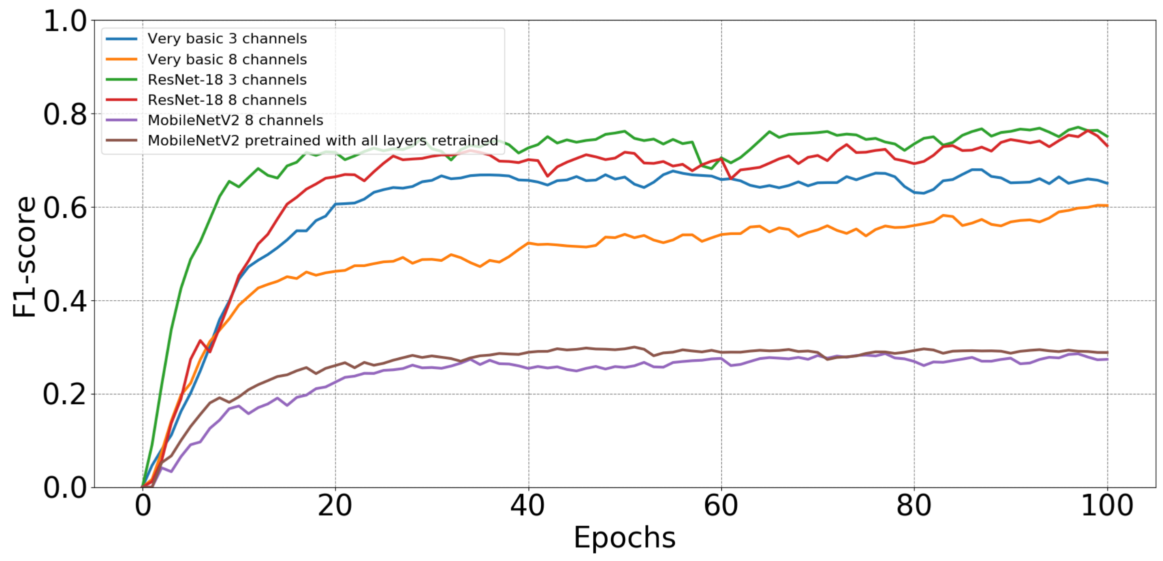


Figure 5.2: Comparison of F1-score of all the networks that where trained with 100 epochs and no data augmentation

5.1 Low results

Although the results are not bad these results do not render an algorithm that could be used directly. This has multiple factors.

The first and probably the most important factor is that the data set is too small. This results in a very high risk of overfitting and probably a quite low robustness.

A second reason is that after segmentation the data set was sorted by hand. This was done by employees at Key Technology. This can result and wrong sorted samples. When wrongly sorted samples are inserted in a very big data set the impact will be very low. But because the data set is small the impact of wrongly sorted samples will be a lot bigger.

A thirth explanation is that the data set is biased too much. For a good learning the best ratio of good and bad samples is 50/50. In this case the ratio is 90/10. Which is strongly biased. Due to the small data set size it is also not very evidently to keep the 50/50 ratio in the mini batches because the bad samples would be used a lot more.

A last possible reason for these result is given by [19]. In this research it is mentioned that the performance of convolutional network suffers from noise filtering before these images are used to train the network. This was not done in this research but with segmenting the images a lot of noise in the backgrounds was lost because these were refilled in the segmenting algorithm. Therefore noise is only present in the pixel values that denote the object in the image [19].

Chapter 6

Conclusion

In this research four different architectures are used to classify images of green beans. This is done on images with 3 channels or 8 channels. And this was realized with using a strict structure and generating results from every step that was made.

These networks differ greatly in results and structure. With a network dept ranging between 2 to 50 layers. The ResNet-50 architecture ended with an F1-score of 0 which means it fails to successful classify the images. But this does not mean this network will not be usable in the future. With a data set that is significantly bigger the algorithm could perform better. Because of this poor F1-score no further experiments were done on this network. The very basic architecture produced better results than expected. The mobileNetV2 architecture reached the highest accuracy but with a relatively low F1-score. Because of the skewed data set the highest priority lays on the F1-score. Thus this result is not satisfactory. The ResNet-18 architecture produced the most stable results with a relatively high accuracy and F1-score. This network will be a good starting point for further research.

6.1 Future work

This research makes the first steps for developing a computer vision algorithm for Key Technology. The 4 different architectures developed and tested are not ready to be used in the field. Therefore future research is needed.

The first and most important step is to develop a method the efficiently collect a representative data set which could be used to train the network. With this data set the results from the networks will be more representative to the real data and classification results will benefit.

The second step needed is developing a better classification algorithm and test its robustness in a range of circumstances. A good starting point in this future research is the ResNet-18 architecture. If this network is customized to specifically sort food products the expected result could be very good.

The last step is to speed up the algorithm in the sorting machine without hurting performance. Because these algorithms need a lot of computations the speed of it can be too slow to sort the same volume of product. Therefor a few hardware adjustments may be needed. A recommendation is adding a dedicated GPU to the hardware of the machine.

Bibliography

[1] A. Comaills, “BEAN POD CUT IPHYM Herbalism Phaseolus vulgaris L.” [Online]. Available: https://www.soin-et-nature.com/pt/plantas-medicinais/5182-bean-pod-cut-iphym-herbalism-phaseolus-vulgaris-l.html

[2] Andrew NG, “Machine Learning — Coursera,” 2018. [Online]. Available: https://www.coursera.org/learn/machine-learning

[3] “Gradient descent explained - Learn ARCore - Fundamentals of Google ARCore [Book].” [Online]. Available: https://www.oreilly.com/library/view/learn-arcore-/9781788830409/e24a657a-a5c6-4ff2-b9ea-9418a7a5d24c.xhtml

[4] Ebc, “Diagnosing Learning Algorithms — ebc.” [Online]. Available: http://www.ebc.cat/2017/02/19/diagnosing-learning-algorithms/

[5] “File:Precisionrecall.svg - Wikimedia Commons.” [Online]. Available: https://commons.wikimedia.org/wiki/File:Precisionrecall.svg

[6] M. L. Guru, “Image Convolution.” [Online]. Available: http://machinelearninguru.com/computer{_}vision/basics/convolution/image{_}convolution{_}1.html

[7] F.-F. Li, J. Johnson, and S. Yeung, “CS231n Convolutional Neural Networks for Visual Recognition.” [Online]. Available: http://cs231n.github.io/convolutional-networks/

[8] A. Ng, K. Katanforoosh, and Y. B. Mourri, “Convolutional Neural Networks - deeplearning.ai — Coursera.” [Online]. Available: https://www.coursera.org/learn/convolutional-neural-networks/home/info

[9] E. R. S. de Rezende, G. C. S. Ruppert, A. Theophilo, and T. Carvalho, “Exposing Computer Generated Images by Using Deep Convolutional Neural Networks,” nov 2017. [Online]. Available: http://arxiv.org/abs/1711.10394

[10] E. Culurciello, “Neural Network Architectures – Towards Data Science.” [Online]. Available: https://towardsdatascience.com/neural-network-architectures-156e5bad51ba

[11] B. Raj, “A Simple Guide to the Versions of the Inception Network.” [Online]. Available: https://towardsdatascience.com/a-simple-guide-to-the-versions-of-the-inception-network-7fc52b863202

[12] Y. J. Heo, S. J. Kim, D. Kim, K. Lee, and W. K. Chung, “Super-High-Purity Seed Sorter Using Low-Latency Image-Recognition Based on Deep Learning,” *IEEE Robotics and Automation Letters*, vol. 3, no. 4, pp. 3035–3042, oct 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8391727/

[13] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, “MobileNetV2: Inverted Residuals and Linear Bottlenecks,” 2018. [Online]. Available: http://arxiv.org/abs/1801.04381

[14] “(PDF) Hybrid Binary Networks: Optimizing for Accuracy, Efficiency and Memory.” [Online]. Available: https://www.researchgate.net/publication/324471879{_}Hybrid{_}Binary{_}Networks{_}Optimizing{_}for{_}Accuracy{_}Efficiency{_}and{_}Memory

[15] S. Sharma, “Activation Functions: Neural Networks – Towards Data Science.” [Online]. Available: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

[16] S. Narkhede, “Understanding Confusion Matrix – Towards Data Science.” [Online]. Available: https://towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62

[17] “TensorFlow.” [Online]. Available: https://www.tensorflow.org/

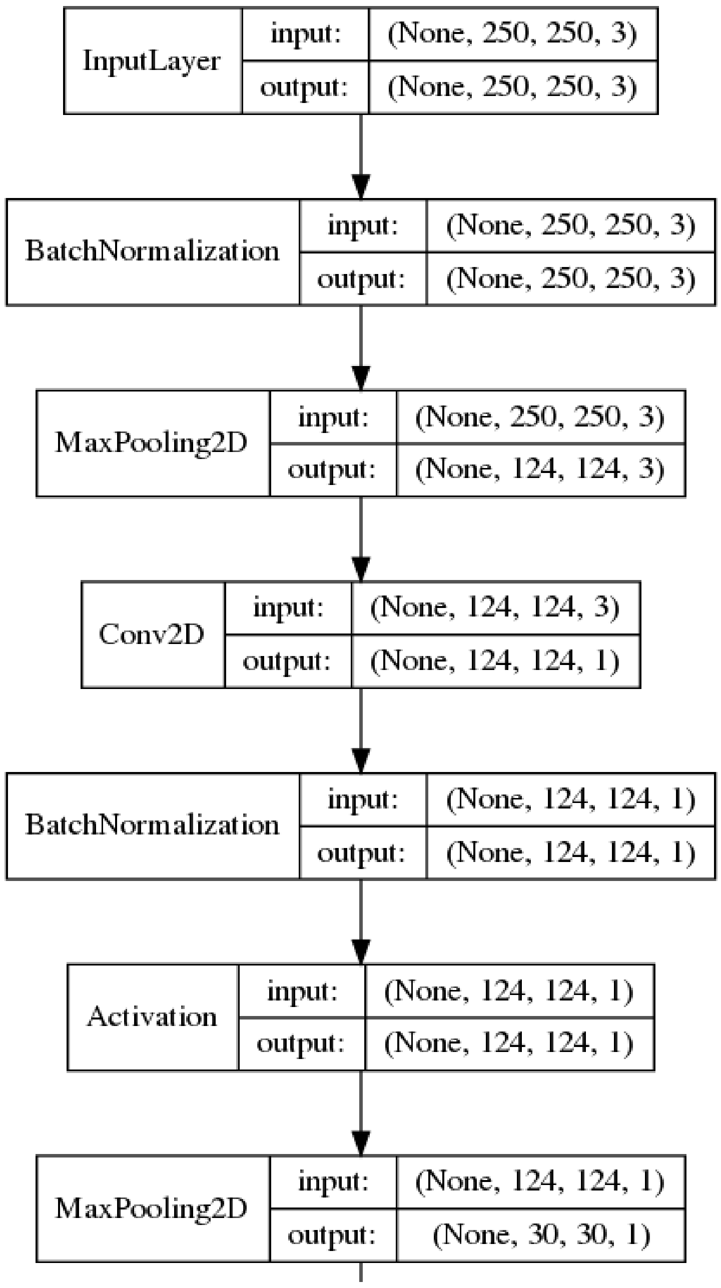
[18] “Home - Keras Documentation.” [Online]. Available: https://keras.io/

[19] J. Jo and Y. Bengio, “Measuring the tendency of CNNs to Learn Surface Statistical Regularities,” no. 1, 2017. [Online]. Available: http://arxiv.org/abs/1711.11561

Appendices

Appendix A

Very basic network architecture



InputLayer
input: (None, 250, 250, 3)
output: (None, 250, 250, 3)
BatchNormalization
input: (None, 250, 250, 3)
output: (None, 250, 250, 3)
MaxPooling2D
input: (None, 250, 250, 3)
output: (None, 124, 124, 3)
Conv2D
input: (None, 124, 124, 3)
output: (None, 124, 124, 1)
BatchNormalization
input: (None, 124, 124, 1)
output: (None, 124, 124, 1)
Activation
input: (None, 124, 124, 1)
output: (None, 124, 124, 1)
MaxPooling2D
input: (None, 124, 124, 1)
output: (None, 30, 30, 1)

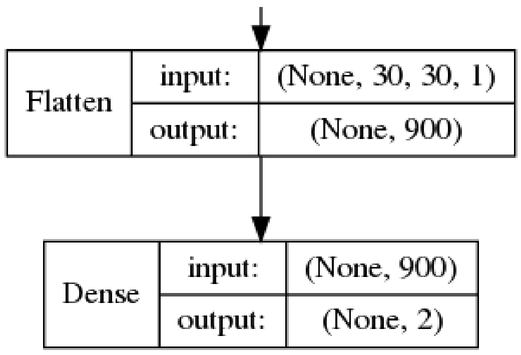


Figure A.1: Network architecture of very basic network

Appendix B

ResNet-18 network architecture

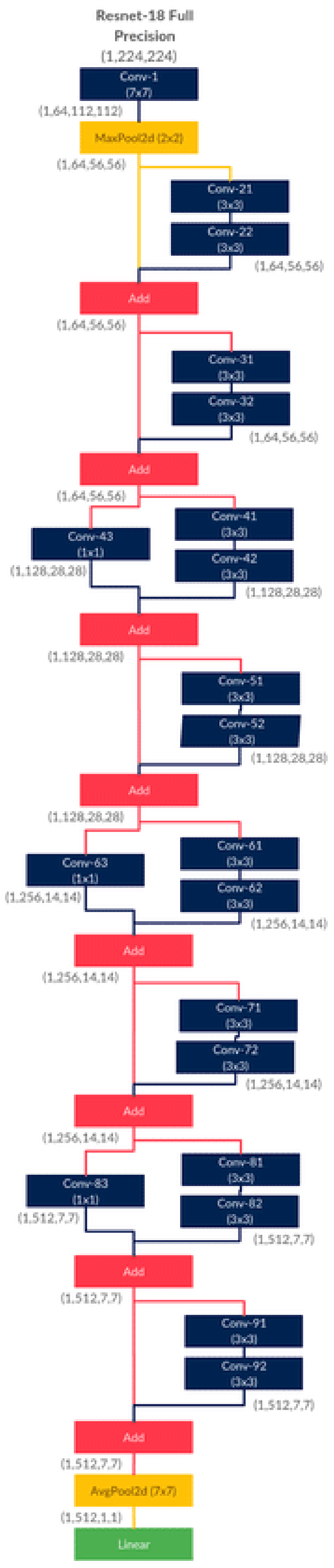


Figure B.1: Block diagram of the ResNet-18 architecture [14]

Appendix C

tables

C.1 Very basic Network

		Before Training	10 Epochs	100 Epochs
Training	Accuracy	0.749	0.918	0.969
	Loss	1.137	0.214	0.084
	F1-score	0.079	0.437	0.819
Test	Accuracy	0.734	0.904	0.946
	Loss	1.121	0.267	0.245
	F1-score	0.076	0.396	0.669

Table C.1: Results for the Very basic architecture with 3 channels and no data augmentation

		Before Training	10 Epochs	100 Epochs
Training	Accuracy	0.202	0.993	0.954
	Loss	2.158	0.177	0.126
	F1-score	0.185	0.473	0.720
Test	Accuracy	0.180	0.936	0.927
	Loss	2.174	0.219	0.195
	F1-score	0.182	0.543	0.597

Table C.2: Results for the Very basic architecture with 8 channels and no data augmentation

C.2 ResNet-18

		Before Training	10 Epochs	100 Epochs
Training	Accuracy	0.422	0.949	0.991
	Loss	0.729	0.143	0.035
	F1-score	0.224	0.719	0.908
Test	Accuracy	0.398	0.968	0.962
	Loss	0.733	0.129	0.125
	F1-score	0.213	0.796	0.742

Table C.3: Results for the ResNet-18 architecture with 3 channels and no data augmentation

		Before Training	10 Epochs	100 Epochs
Training	Accuracy	0.503	0.954	0.989
	Loss	0.762	0.145	0.038
	F1-score	0.237	0.698	0.907
Test	Accuracy	0.521	0.945	0.967
	Loss	0.767	0.160	0.121
	F1-score	0.231	0.613	0.745

Table C.4: Results for the ResNet-18 architecture with 8 channels and no data augmentation

C.3 ResNet-50

		Before Training	10 Epochs
Training	Accuracy	0.104	0.897
	Loss	3.390	1.657
	F1-score	0.168	0.000
Test	Accuracy	0.105	0.897
	Loss	3.347	1.661
	F1-score	0.171	0.000

Table C.5: Results for the ResNet-50 architecture with 3 channels and no data augmentation

		Before Training	10 Epochs
Training	Accuracy	0.103	0.897
	Loss	3.887	1.657
	F1-score	0.168	0.000
Test	Accuracy	0.102	0.897
	Loss	3.874	1.661
	F1-score	0.169	0.000

Table C.6: Results for the ResNet-50 architecture with 8 channels and no data augmentation

		Before Training	10 Epochs
Training	Accuracy	0.125	0.897
	Loss	2.226	1.655
	F1-score	0.154	0.000
Test	Accuracy	0.137	0.897
	Loss	2.190	1.661
	F1-score	0.158	0.000

Table C.7: Results for the pretrained ResNet-50 architecture with no data augmentation and only the last layer is retrained

		Before Training	10 Epochs
Training	Accuracy	0.127	0.898
	Loss	2.212	1.651
	F1-score	0.156	0.000
Test	Accuracy	0.122	0.897
	Loss	2.189	1.661
	F1-score	0.148	0.000

Table C.8: Results for the pretrained ResNet-50 architecture with no data augmentation and all layers are retrained

C.4 MobileNetV2

		Before Training	10 Epochs
Training	Accuracy	0.897	0.938
	Loss	0.378	0.184
	F1-score	0.000	0.248
Test	Accuracy	0.897	0.939
	Loss	0.376	0.187
	F1-score	0.000	0.213

Table C.9: Results for the MobileNetV2 architecture with 3 channels and no data augmentation

		Before Training	10 Epochs	100 Epochs
Training	Accuracy	0.251	0.948	0.995
	Loss	0.783	0.156	0.016
	F1-score	0.134	0.263	0.341
Test	Accuracy	0.264	0.933	0.969
	Loss	0.783	0.195	0.144
	F1-score	0.142	0.200	0.290

Table C.10: Results for the MobileNetV2 architecture with 8 channels and no data augmentation

		Before Training	10 Epochs
Training	Accuracy	0.263	0.931
	Loss	1.066	0.197
	F1-score	0.160	0.148
Test	Accuracy	0.273	0.929
	Loss	1.058	0.248
	F1-score	0.164	0.141

Table C.11: Results for the pretrained MobileNetV2 architecture with no data augmentation and only the last layer is retrained

		Before Training	10 Epochs	100 Epochs
Training	Accuracy	0.314	0.971	0.995
	Loss	0.976	0.081	0.011
	F1-score	0.144	0.144	0.346
Test	Accuracy	0.328	0.958	0.973
	Loss	0.981	0.121	0.139
	F1-score	0.153	0.322	0.292

Table C.12: Results for the pretrained MobileNetV2 architecture with no data augmentation and all layers are retrained

C.5 Very basic network with data augmentation

		Before Training	100 Epochs
Training	Accuracy	0.543	0.939
	Loss	1.650	0.208
	F1-score	0.166	0.597
Test	Accuracy	0.548	0.931
	Loss	1.551	0.218
	F1-score	0.177	0.564

Table C.13: Results for the Very basic architecture with 3 channels and data augmentation

		Before Training	100 Epochs
Training	Accuracy	0.376	0.905
	Loss	0.897	0.259
	F1-score	0.220	0.121
Test	Accuracy	0.367	0.898
	Loss	0.842	0.292
	F1-score	0.219	0.046

Table C.14: Results for the Very basic architecture with 8 channels and data augmentation

C.6 ResNet-18 with data augmentation

		Before Training	100 Epochs
Training	Accuracy	0.459	0.954
	Loss	0.736	0.164
	F1-score	0.062	0.697
Test	Accuracy	0.480	0.949
	Loss	0.737	0.168
	F1-score	0.079	0.648

Table C.15: Results for the ResNet-18 architecture with 3 channels and data augmentation

		Before Training	100 Epochs
Training	Accuracy	0.620	0.954
	Loss	0.642	0.163
	F1-score	0.279	0.673
Test	Accuracy	0.610	0.940
	Loss	0.642	0.214
	F1-score	0.277	0.550

Table C.16: Results for the ResNet-18 architecture with 8 channels and data augmentation