Implementation of Deep Learning Algorithms for Object Recognition

Süvaöppe Algoritmide Rakendamine Objektide Tuvastamiseks

MASTER THESIS

Student: Aydin Zengin
Student code: 156351
Supervisor: Dr Robert Hudjakov

Tallinn, 2018
AUTHOR’S DECLARATION

Hereby I declare, that I have written this thesis independently.
No academic degree has been applied for based on this material. All works, major viewpoints and data of the other authors used in this thesis have been referenced.

“.......” ................... 201....

Author: ..............................
/signature /

Thesis is in accordance with terms and requirements

“.......” ................... 201....

Supervisor: .............................
/signature/

Accepted for defence

“.......” ...................201....

Chairman of theses defence commission: .................................................................
/name and signature/
TTÜ Department’s title

THESIS TASK

Student: Aydin Zengin, MAHM156351
Study programme, Main speciality: MAHM, Mechatronics
Supervisor: Phd., Robert Hudjakov
Consultants: N/A

Thesis topic:
(in English) Implementation of Deep Learning Algorithms for Object Recognition
(in Estonian) Süvaõppe algoritmide rakendamine objektide tuvastamiseks

Thesis main objectives:
1. Implementation of deep learning algorithms to recognize minor differences of the products
2. Modifying and creating the artificial neural networks (ANN) for this purpose
3. Evaluation to get better performance and applicability

Thesis tasks and time schedule:

<table>
<thead>
<tr>
<th>No</th>
<th>Task description</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Literature review</td>
<td>25.02.2018</td>
</tr>
<tr>
<td>2.</td>
<td>Finding a suitable ANN for our case</td>
<td>09.03.2018</td>
</tr>
<tr>
<td>3.</td>
<td>Getting know about the ANN which we use</td>
<td>16.03.2018</td>
</tr>
<tr>
<td>4.</td>
<td>Creating and labelling the dataset that we will test</td>
<td>23.03.2018</td>
</tr>
<tr>
<td>5.</td>
<td>Modifying and testing the algorithm to get a better result</td>
<td>06.04.2018</td>
</tr>
<tr>
<td>6.</td>
<td>Results and conclusion</td>
<td>20.04.2018</td>
</tr>
<tr>
<td>7.</td>
<td>Compiling the thesis</td>
<td>18.05.2018</td>
</tr>
</tbody>
</table>

Language: English

Deadline for submission of thesis: 25.05.2018

Student: Aydin Zengin
Supervisor: Dr Robert Hudjakov
CONTENT

LIST OF FIGURES

LIST OF SYMBOLS

LIST OF ABBREVIATIONS

PREFACE

1. INTRODUCTION

1.1 MOTIVATION

1.2 TASK DEFINITION

1.3 THESIS GOAL

1.4 LITERATURE REVIEW

1.5 OUTLINE OF THE THESIS

2. AN EFFICIENT ARTIFICIAL NEURAL NETWORK FOR OBJECT RECOGNITION

2.1 PERCEPTRON

2.2 SIGMOID NEURONS

2.3 FEED-FORWARD NETWORKS

2.3.1 HIDDEN LAYERS

3. CONVOLUTIONAL NEURAL NETWORKS (CNNS)

3.1 CONVOLUTION

3.1.1 CONVOLUTION ARITHMETIC

3.2 POOLING

3.2.1 POOLING ARITHMETIC

3.3 CALCULATION OF OUTPUT VALUES OF CONVOLUTION AND POOLING LAYER OF THE NETWORK

3.4 RELU (RECTIFIED LINEAR UNIT)

3.5 STOCHASTIC GRADIENT DESCENT WITH MOMENTUM (SGDM)
3.6 **Optimization**

3.7 **L2 (Gaussian) Regularization**

3.8 **Fully Connected Layer**

3.9 **The Activation Function of the Neural Network**

3.10 **Dropout**

4. **Experiments**

4.1 **Creating a Dataset for the Input Stage**

4.2 **Modifying the ANN with Valid Parameters**

4.3 **Feature Extraction from the Input**

4.4 **Results**

5. **Conclusions**

5.1 **Future Work**

5.2 **Summary**

BIBLIOGRAPHY

APPENDICES

A. **Datasets**

A.1 **Decent Wrench Dataset**

A.2 **Defective Wrench Dataset**

B. **Coding**

B.1 **Object Detection Algorithm with CNN in MATLAB**
LIST OF FIGURES

FIGURE 2.1 PERCEPTRON (NIELSEN, 2015) .......................................................................................................................... 17
FIGURE 2.2 STEP FUNCTION AND SIGMOID FUNCTION, RESPECTIVELY (NIELSEN, 2015) ................................................................. 20
FIGURE 2.3 FEED-FORWARD NETWORK STRUCTURE (STANFORD UNIVERSITY) ........................................................................................................ 21
FIGURE 2.4 TWO-LAYER NEURAL NETWORK (BISHOP, 2013) .......................................................................................................................... 21
FIGURE 3.1 TYPICAL CNN ARCHITECTURE (ZHO et al., 2017) ......................................................................................................................... 24
FIGURE 3.2 ILLUSTRATION OF COMPUTING THE OUTPUT VALUE OF THE CONVOLUTION ................................................................. 27
FIGURE 3.3 POOLING WITH USING AVERAGE POOLING OPERATION ........................................................................................................... 29
FIGURE 3.4 POOLING WITH MAX POOLING OPERATION ............................................................................................................................. 29
FIGURE 3.5 BASIC NEURON MODEL WITH ACTIVATION FUNCTION (ALOM et al., 2018) ................................................................. 33
FIGURE 3.6 BEHAVIOUR OF SGD WITHOUT MOMENTUM (ON THE LEFT SIDE) AND BEHAVIOUR OF SGD WITH MOMENTUM (ON THE
RIGHT SIDE) (SARGUR & N.) ................................................................................................................................................. 35
FIGURE 3.7 BACK-PROPAGATION ILLUSTRATION (CIRESAN et al., 2011) ........................................................................................................ 36
FIGURE 3.8 (A) NETWORK WITHOUT DROPOUT, (B) DROPOUT APPLIED NETWORK, (SRIVASTAVA et al., 2014) .......................................................... 39
FIGURE 3.9 IMPORTANCE OF DROPOUT (SRIVASTAVA et al., 2014) ........................................................................................................ 39
FIGURE 4.1 ROOT FILE FOR DATASTORE WHERE THE SYSTEM TAKES THE DATA AS INPUT ........................................................................... 42
FIGURE 4.2 CREATING DATASTORE VARIABLES .................................................................................................................................... 43
FIGURE 4.3 LABELLING THE IMAGES WITH THE NAME OF THE FOLDER WHERE THEY BELONG ........................................................................ 43
FIGURE 4.4 DATASTORE FOLDER WITH ITS SUBFOLDER WITH THEIR FILES ........................................................................................................ 44
FIGURE 4.5 SPLITTING DATA FOR TRAINING AND TESTING ..................................................................................................................... 44
FIGURE 4.6 IMAGE PRE-PROCESSING FOR INPUT LAYER .................................................................................................................................. 45
FIGURE 4.7 CALLING ALEXNET IN A VARIABLE ................................................................................................................................. 47
FIGURE 4.8 BASIC INFORMATION ABOUT ALEXNET .................................................................................................................................. 47
FIGURE 4.9 MODIFYING THE NN ARCHITECTURE FOR OUR APPLICATION ................................................................................................... 47
FIGURE 4.10 INPUT LAYER INFORMATION .................................................................................................................................................. 48
FIGURE 4.11 OUTPUT LAYER OF THE ALEXNET ...................................................................................................................................... 48
FIGURE 4.12 MODIFYING THE NETWORK LAYERS ....................................................................................................................................... 48
FIGURE 4.13 NEW FULLY CONNECTED LAYER ........................................................................................................................................... 49
FIGURE 4.14 SCALING AND RESIZING THE WEIGHTS FOR THE SECOND CONVOLUTIONAL LAYER ........................................................................ 49
FIGURE 4.15 TRAINING OPTIONS ............................................................................................................................................................ 51
FIGURE 4.16 TRAINING OPTION’S PARAMETERS ......................................................................................................................................... 51
FIGURE 4.17 TRAINING THE NETWORK AND CLASSIFYING THE TEST DATA .................................................................................................... 51
FIGURE 4.18 EVALUATING THE RESULTS OF PREDICTIONS ON TEST DATA ...................................................................................................... 52
FIGURE 4.19 FIRST CONVOLUTIONAL LAYER WEIGHTS .................................................................................................................................... 53
FIGURE 4.20 TRAINING AND VALIDATION WITH 6 EPOCHS AND 90 ITERATIONS .......................................................................................... 54
FIGURE 4.21 TRAINING USING MINIBATCH 36, MAX EPOCH 8, LEARNING RATE 1E-5 ......................................................... 54
FIGURE 4.22 TRAINING USING MINIBATCH 36, MAX EPOCH 10, LEARNING RATE 1E-4 ...................................................... 55
FIGURE 4.23 TRAINING USING MINIBATCH 8, MAX EPOCH 8, LEARNING RATE 1E-5 ......................................................... 55
FIGURE 4.24 THE PREDICTION RESULT OF THE TEST DATASET WITH THE CONFUSION MATRIX ........................................... 56
FIGURE 4.25 GRAPHICAL REPRESENTATION OF THE CONFUSION MATRIX ........................................................................ 56
FIGURE A.1 DECENT DATASET (1-35/485) ...................................................................................................................... 66
FIGURE A.2 DECENT DATASET (36-63/485) .................................................................................................................. 67
FIGURE A.3 DECENT DATASET (63-91/485) .................................................................................................................. 67
FIGURE A.4 DECENT DATASET (91-119/485) ............................................................................................................... 68
FIGURE A.5 DECENT DATASET (120-147/485) ............................................................................................................. 68
FIGURE A.6 DECENT DATASET (148-176/485) ............................................................................................................. 69
FIGURE A.7 DECENT DATASET (177-204/485) ............................................................................................................. 69
FIGURE A.8 DECENT DATASET (205-232/485) ............................................................................................................. 70
FIGURE A.9 DECENT DATASET (233-260/485) ............................................................................................................. 70
FIGURE A.10 DECENT DATASET (261-288/485) ............................................................................................................ 71
FIGURE A.11 DECENT DATASET (289-316/485) ............................................................................................................ 71
FIGURE A.12 DECENT DATASET (317-344/485) ............................................................................................................ 72
FIGURE A.13 DECENT DATASET (345-372/485) ............................................................................................................ 72
FIGURE A.14 DECENT DATASET (373-400/485) ............................................................................................................ 73
FIGURE A.15 DECENT DATASET (401-428/485) ............................................................................................................ 73
FIGURE A.16 DECENT DATASET (429-456/485) ............................................................................................................ 74
FIGURE A.17 DECENT DATASET (457-485/485) ............................................................................................................ 74
FIGURE A.18 DEFECTIVE DATASET (1-36/130) ............................................................................................................. 75
FIGURE A.19 DEFECTIVE DATASET (37-60/130) ............................................................................................................. 75
FIGURE A.20 DEFECTIVE DATASET (61-84/130) ........................................................................................................... 76
FIGURE A.21 DEFECTIVE DATASET (85-108/130) ....................................................................................................... 76
FIGURE A.22 DEFECTIVE DATASET (109-130/130) ................................................................................................... 77
FIGURE A.23 IMPORTING A TESTING IMAGE ............................................................................................................ 79
FIGURE A.24 FIRST CONVOLUTIONAL LAYER WEIGHTS ............................................................................................ 81
FIGURE A.25 SHOWING RANDOM IMAGES FROM BOTH FILES ................................................................................... 83
FIGURE A.26 SHOWING RANDOMLY TWENTY-FIVE TRUE RESIZED IMAGES TO SEE THE RESULT .................................. 84
FIGURE A.27 SHOWING RANDOMLY TWENTY-FIVE FALSE RESIZED IMAGES TO SEE THE RESULT .............................. 84
FIGURE A.28 THE ACCURACY OF THE TRAINING AND TESTING WITH 6 EPOCHS AND 90 ITERATIONS .................................. 86
FIGURE A.29 PLOTTING THE CONFUSION MATRIX .................................................................................................... 88
FIGURE A.30 PLOTTING THE TRAINING LOSS RESULT ................................................................................................ 88
LIST OF SYMBOLS

\( x_j \)  
Input of a perceptron or sigmoid neuron

\( w_j \)  
Weight of a perceptron or sigmoid neuron

\( b \)  
Bias

\( \sigma \)  
Sigmoid function

\( p \)  
Proportion

\( \varepsilon \)  
Learning rate

\( \alpha \)  
Momentum parameter

\( \vartheta \)  
Initial parameter

\( v \)  
Initial velocity

\( g \)  
Gradient

\( l \)  
Number of layers of the network

\( h_k \)  
The output of hidden layer

\( \bar{y} \)  
The network’s output

\( I \)  
Input

\( k \)  
Kernel

\( p \)  
Padding

\( s \)  
Stride
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANN</td>
<td>Artificial Neural Network</td>
</tr>
<tr>
<td>CNN</td>
<td>Convolutional Neural Network</td>
</tr>
<tr>
<td>CPU</td>
<td>Control Processing Unit</td>
</tr>
<tr>
<td>DL</td>
<td>Deep Learning</td>
</tr>
<tr>
<td>DNN</td>
<td>Deep Neural Network</td>
</tr>
<tr>
<td>FC</td>
<td>Fully Connected Layer or Fully Connected Network</td>
</tr>
<tr>
<td>GPU:</td>
<td>Graphics Control Unit</td>
</tr>
<tr>
<td>IBVS:</td>
<td>Image-Based Visual Servoing</td>
</tr>
<tr>
<td>MLP:</td>
<td>Multilayer Perceptrons</td>
</tr>
<tr>
<td>NN:</td>
<td>Neural Networks</td>
</tr>
<tr>
<td>PBVS:</td>
<td>Position Based Visual Servoing</td>
</tr>
<tr>
<td>RGB:</td>
<td>Red Green Blue</td>
</tr>
<tr>
<td>RGB-D:</td>
<td>Red Green Blue-Depth</td>
</tr>
<tr>
<td>SVM:</td>
<td>Support Vector Model</td>
</tr>
<tr>
<td>OCR:</td>
<td>Optical Character Recognition</td>
</tr>
</tbody>
</table>
PREFACE

In English:

The author would like to thank his supervisor, Dr Robert Hudjakov, who gave him the opportunity to work with him. His great patience has given him the motivation. His academic and industrial connections and knowledge enlightened him during this work. The author also would like to thank lecturers who made this thesis work easier with those subjects. This thesis is provided by Department of Electrical and Power Engineering and Mechatronics by Tallinn University of Technology.

Finally, the author would like to thank his family, who believed in him and helped him in many aspects tremendously during this period with their great motivational support.
1. INTRODUCTION

Object recognition has been a very popular task lately, and it has been solving tasks from many different areas in a better way day by day. Existing solutions are good enough to use, and they are efficient too, but the problem is that these are algorithms susceptible to small changes in some of its parameters, i.e. lighting conditions, the way the object desired to be recognised is positioned, the orientation of the object. New machine learning techniques which use deep learning, might be efficient and time-saving to use in this task.

The existing solutions have boundaries that cannot develop while keeping its essential features as it is. Hence these essential features limit the development. For instance, visual servoing can handle object recognition task via marking each object, and this makes it time costly and inefficient to use in an industrial application, considering the high probability of a significant volume of objects to be recognised in such a task in an industrial environment.

Now, we live in a world in which technology is developing at a very fast pace, and with advanced and powerful graphics processing units, machine vision problems are getting more accessible and comfortable to solve using deep learning algorithms to recognise and classify any desired object. Using deep learning algorithms in industrial or scientific applications, means the outcome will be a better and faster operation thanks to a less need for interruption or configuration by human labour as the outcome. The acquired results might flourish in a variety of fields. The desired outcome for machine vision for now is for the product to be correctly recognised by the algorithm, even when it is not in the desired position or marked with an identifier, or is blocked by some distraction.

Nowadays, powerful GPUs make the process time shorter and rise the result rates, but energy consumption is higher than underdeveloped systems. However, in this thesis work, we will ignore this energy consumption. The reason behind this is that the consumption is outside of the scope of this paper, and for further readings, provided articles can be useful for the reader (Ahamed & Magoules, 2014), (Magoules et al., 2014), (Huzmiev & Chipirov, 2016).

Artificial neural networks (ANNs) are becoming more influential and robust, and these networks make more manageable solutions to most of the problems. Moreover, solving tasks generally take much more time with traditional methods than ANNs. Therefore, instead of marking products with markers, deep learning (DL) algorithms will be used in this thesis task. Instead of choosing DL for this task, we could also use edge-based segmentation for object detection, and colour-based
descriptor could be used for classification. However, choosing DL makes this task data driven. With enough data to train the system, we expect the result to be better than other options and more accessible to implement. The other advantage of the DL is that object’s colour, shape, orientation or lightening condition loses its importance to some extent. With an extensive dataset of an object, the system will not lose its capability to recognise it or its the prediction rate will be affected much when above-mentioned conditions occur, thanks to having a deep perception from many features it learned.

In this thesis work, the aim is to apply one object recognition task with an ANN where the task is to specifically distinguish the same object in different states. One class is “good” condition and the other one “defective”. This kind of system could be applicable in quality assurance and classification of similar products with further developments.

1.1 Motivation

Even though, the ANN has a long history; the convolutional neural network (CNN) is not a new concept. Decades ago it got involved in our lives and started reshaping our daily tasks such as document detection, handwritten digit recognition (Y. LeCun, 1998), (Y. LeCun et al., 1989). Still, it is not in the position that it deserves to be in. There are dozens of applications that it is used in, but robotic applications. Since I have had interest in this field for a while, I attended lectures in Intelligent Control System, and this field became even more important to me. Those days the world was shaken with news which was about a computer program, that was developed with deep neural networks (DNNs), that, for the first time, won a game against a professional player, European Go champion, by five games to zero (Silver et al., 2016). My growing interest in this field and this news gave me the inspiration for this thesis work.

1.2 Task definition

In this thesis work, with using DL method, object detection will be done. The basic idea behind implementing a deep learning algorithm for object detection is checking the ability of the algorithm
to recognise same products which might have minor/major defects or none. Since NN are quite
good at recognising different products, this option is promising. With the result, the possibility of
using it for quality control will be evaluated. Since creating a new NN is expensive and requires too
much computing time to reach flexible configuration, the most suitable solution will be researched,
tested and evaluated for its applicability.

1.3 Thesis goal

Herein I focus on the applicability of DL algorithms for similar objects recognition task. My goal is
to use a DL algorithm based on similar objects and recognise them either as a correct product or
faulty product. The challenge here is that traditional techniques have boundaries to recognise
similar products if there is not much difference between products. With using DL algorithms, this
chance is higher than traditional techniques. However, is it still applicable? Alternatively, how is
going to be the accuracy and efficiency of the system? These will be answered at the end of the
thesis.

1.4 Literature review

Object recognition with ANNs has already been used in daily life tasks for quite long time and
getting more popular and efficient each day. It handles problems with ease if we compare them
with traditional methods. For instance, in the USA, LeNet, which has been created by Yann LeCun
(Yann LeCun et al., 1998), has been reading 10%-20% of all USA bank checks for character
recognition, which means several millions of checks per day. This shows how NNs can be more than
useful for daily tasks. The absence of this, wouldn’t cause a problem but would make the task more
challenging and increase the cost of the solution economically.

Many tasks have been completed and so much time has been spent on building better techniques
for traditional methods to get to this stage. Why we should appreciate the traditional techniques is
that we still use them and they were needed to get to this point. ANNs still implement some of the
same methods of traditional techniques such as support vector machines (SVM) (Burges, 1998),
histograms of oriented gradients (Dalal & Triggs, 2005), naive Bayes classifier, edge detection, gradient descent optimisation algorithm.

ANNs generally employ supervised learning algorithms. Supervised learning algorithms need the training data to be labelled with the desired result so that the ANN will make a connection between the input and the result. With the learned features from the training data, test data can be evaluated and results predicted. ANNs are also used for unsupervised learning. The main issue with unsupervised learning is that it generally requires more data than supervised learning algorithms. The reason is, in unsupervised learning method, the aim is to find interesting patterns. The system will just be fed with input, and it will make a prediction what might be the connection between inputs. Therefore, there should be many inputs. In our case, we are implementing supervised learning with our labelled input data.

Biological inspiration (Hubel & Wiesel, 1968) initiated the process of formation for CNN. As we already know that CNNs have been proposed for solving image and visual task many years ago (Yann LeCun et al., 1998), it is still not popular in engineering fields yet (P. Y. Simard, Steinkraus, & Platt, 2003), but there are many types of research about it.

There is some literature which has a similar topic with this thesis (Stenroos, 2017), (Gutstein, 2010), (Mohammed, 2014), (Alina, 2015), (Hartemink, 2012), but either the method they use, or the priority of the topic makes our topic novel.

1.5 Outline of the thesis

In the first chapter, the main idea is introduced, the motivation of this thesis is explained. Thesis’ task definition is clarified, and the goal is specified as well. Overviewed of the literature, which has already become a milestone in this field, and previous researcher related to this thesis task. Basically, in the overview part, we explained the essential concepts and methods which are related to the topic. Those will be compared with this thesis work to improve the result.

Based on the defined tasks in the first chapter and drawn boundaries, the most efficient solution will be searching for in the second chapter. It is going to be more specific and deeper side of the
solutions will be evaluated. At the very end of this chapter, we will have a particular type of NN that we will progress within the next chapters.

Since the chosen solution is already too wide to explain in the third chapter, explanation of the specific solution will take place in the third chapter with its boundaries and specific parameters. Implementation of the solution is going to be another task too. The result of this chapter is going to get the finalised algorithm to progress with experimentation section.

In the fourth, experimentation chapter, the chosen type of NN will be used to develop the network for our task. Testing the finally implemented solution is going to be a task at the end of this chapter, and of course, evaluation is a must in this section. “What could be better?” and “How to make it more efficient?” are going the be answered in this part of the research.

In the concluding chapter, future works will be mentioned based on what could be done differently in this thesis work. Moreover, of course, the summary is another must at the very the end of this thesis.
2. AN EFFICIENT ARTIFICIAL NEURAL NETWORK FOR OBJECT RECOGNITION

Deep learning is a technique of machine learning. It has a history which starts 70 years ago. What we refer when we say deep learning is ANNs. Learning features from input using ANNs is called deep learning. Nowadays, this kind of networks became quite robust and efficient algorithms that they make very accurate predictions. There are some cases it is promising that they are even better than humans such as in (Silver et al., 2016).

To recognise an image using traditional techniques, first, there should be a task that has to recognise each object in the images to be able to make a connection between images, but the difficulty here is vectorial or matrix representation of each image required in order to progress with recognition. While traditional techniques are applicable for small datasets, wide datasets are not an issue for ANNs. Traditional techniques are not good enough to get more information from mid-level and high level when feature extraction (Zhou et al., 2017). Moreover, with calculated, and learned features, classifier needs to be trained to be able to recognise new images in either with traditional techniques or ANNs. Once new images are received by the algorithm, these need to be represented mathematically by using vectors or matrixes, after that, the learned features of the new images will be forwarded to the classifier. Using learned features in the classifier, classification part makes predictions. What artificial neural network provides is that it has loss function to evaluate and make the error approximate zero. It is kind of in-house mechanism that combines all the algorithms and techniques together. However, with deep learning algorithm, it provides end-to-end learning approach, which means that deep learning receives image as an input, as a raw image, and features are learned directly from raw images. To learn the features of each image properly, it uses cost function to approximate the error to zero. After that learned features are forwarded to next layers in the network to make a classification. Therefore, it is called end-to-end learning. It would be easier to explain in a way with assuming neural network as a black box. Input enters the black box, black box first represents images in a matrix format, recognise the background, the shape of the images and the colour concentrations of the images, and then make a pattern using common features. With this pattern, a new image is used as an input for classification progress. As an output of black box, the class of the new object is predicted. That is the reason for calling deep learning end-to-end. The mathematical side of the chosen NN will be explained in the next chapter.
CNN is a type of NN that is used mostly in image recognition field due to it learns features from raw images and has a robust architecture. All kinds of NNs are made of layers, and CNN uses the same logic, which means information goes through those layers. The idea behind CNN is proposed by (Fukushima, 1980), and developed by (Yann LeCun et al., 1998). After these two articles, it became more popular, powerful and significant the more it was developed. As a type of neural network, CNN, has a similar structure as the human brain. Moreover, biological neural networks are much more complicated and have much more connections than artificial neural networks.

In the next following subsections, the basic structure of ANNs are explained, and then in the next chapter chosen ANN, CNN, will take place.

2.1 Perceptron

Human brains have two hemispheres, and each of them has a primary visual cortex, V1, it has 140 million neurons, and there are connections between them. The human brain is such a gorgeous formation that does not just have one cortex but has five of them. So that for our brains it is quite easy to handle image processing. When we say deep learning, we refer to machine learning technique, the reason behind that is deep learning uses DNN (deep neural network) which is consist of layers to make a prediction. Alternatively, in the other way, deep learning has explained a set of techniques to learn features and classify in neural networks (Nielsen, 2015).

Frank Rosenblatt, who is best known in the field of artificial intelligence, developed an electronic device which has the ability to learn, and it is called perceptron. The most common in this field is known as a sigmoid neuron. Perceptors take the binary data with the connection it has with its surrounding as inputs and generates a single binary output as shown in Figure 2.1.

![Figure 2.1 Perceptron (Nielsen, 2015)]
As it is shown in Figure 2.1, the perceptron has three inputs as called \( x_1 \), \( x_2 \), and \( x_3 \). At this point, Rosenblatt involved weights as a parameter to calculate the output. Weights for corresponding \( x \) are \( w_1 \), \( w_2 \), and \( w_3 \) in Figure 2.1.

\[
output = \begin{cases} 
0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\
1 & \text{if } \sum_j w_j x_j > \text{threshold} 
\end{cases}
\]  

(2.1)

Where \( x_i \) – input,

\( w_i \) – weight.

The threshold value is a value that we define in the algorithm. It can be between 0 and 1. For instance, if the mean value of the inputs is bigger than the threshold, the output is going to be 1, and vice versa. Keeping the value low means output is going to be most likely 1 if the system is not sure about the learned pattern and it will result in wrong classification. Keeping it high also creates a problem and reduces the success rate.

The modern networks do not use just one perceptron but use many of them as it is shown in Figure 2.3. It helps to solve complex problems with ease and resembles the structure of the human brain. The perceptrons of the first layer which is the input layer, are responsible for making decisions weighing the inputs. The second layer is going to be weighing the result of the first layer’s decisions and so on. However, the task is not just weighing the previous layer’s decision. Each layer that comes next makes a more complicated decision than the previous one. In this way, more complex tasks become more natural.

Equation (2.1) can be expressend as shown in Equation (2.2), and in this equation output is a depended variable. \( x \), as an input, can be any data. Therefore it is an independent variable. \( w \) and \( b \) are other parameters of the output function.

\[
w.x \equiv \sum_j w_j x_j
\]  

(2.2)

\[
output = \begin{cases} 
0 & \text{if } w.x + b \leq 0 \\
1 & \text{if } w.x + b > 0 
\end{cases}
\]

Where \( w \) - vectorial representation of weights,
\[ x - \text{vectorial representation of inputs,} \]
\[ b - \text{bias.} \]

Bias can be put in a way that measures of how easy it is to get the perceptron to output a 1. Within this explanation, one can understand that with a significant bias, it is easier to get the perceptron to output a 1 (Nielsen, 2015). What is changed in Equation (2.1) is that bias involved in the equation as \( b \equiv -\text{threshold} \) and new Equation (2.2) is derived. The other role of the bias is that absence of input, the x value will be zero, and there will not be an output. Absence of output will decrease NN’s capability of learning. Therefore bias is there and it is one of the important parameters.

Perceptrons can be used as an elementary logical function as giving weights to inputs and to biases themselves, in this way it behaves such as OR, NAND, or AND. Since NAND gates are universal computation units, perceptrons will also behave like a universal computational unit.

### 2.2 Sigmoid neurons

The problem with perceptrons is that we can create neural networks using it and it automatically generates and updates the weight and bias values to make an output pattern out of input that we feed the system with. Sometimes it recognises the input image wrongly that that one did not give the desired result and get misclassified. By changing the bias or weight value of a perceptron might correct that particular misclassified result but we can get lost all the correct results because of changing that weight or bias. So when perceptrons are not an option at some points, sigmoid neurons can come into play in the network, and in fact, those are similar with perceptrons in most ways.

What sigmoid neuron does is when we change the bias or weight value of it, it just changes the output of it. In this way, the other weight and biases will be safe.

The main difference between perceptron and sigmoid neuron is that while perceptrons take input as 0 or as 1, sigmoid neurons take a value between 0 and 1 range, which makes the output different too. The difference can be noticed in Figure 2.1. On the left side, perceptron behaves as a step
The step function gives output as 0 or 1. The sigmoid function as presented on the right side of gives a value in a range between 0 and 1.

![Step function and sigmoid function](image)

**Figure 2.2** Step function and sigmoid function, respectively (Nielsen, 2015)

If we think the perceptron as shown in Figure 2.1 as a sigmoid neuron, then the Equation (2.2) can be expressed as Equation (2.3) for sigmoid neuron's output.

\[ \sigma(z) \equiv \frac{1}{1 + e^{-z}} \]  

(2.3)

\[ \text{output} \equiv \frac{1}{1 + \exp(-\sum_j w_j x_j - b)} \]

Where \( \sigma \) represents sigmoid function.

If there are small changes in the weights \( \Delta w_j, \Delta b \) small changing is bias will create small changing on output function, and it can be represented by using Equation (2.4) (Nielsen, 2015).

\[ \Delta \text{output} \approx \sum_{j} \frac{\partial \text{output}}{\partial w_j} \nabla w_j + \frac{\partial \text{output}}{\partial b} \nabla b \]  

(2.4)

Where \( \partial \) represents partial derivative.
2.3 Feed-forward networks

The necessary explanation of feed-forward network is that it takes the output of the previous layer as an input for current layer, which means the data goes straight from the input layer until the output layer as it can be seen in Figure 2.3. No feedback that goes back in this type of network, there is no connection between perceptrons within the same layer neither. If there is a feedback, then it is called recurrent neural network type in which the behaviours of the perceptrons and sigmoid neurons change with feedback from the neuron in the same layer. Feed-forward network is also known as multilayer perceptron. Since it is not suitable for classification tasks, to limit this thesis’ scope, we will not go deep into it.

Figure 2.3 Feed-forward network structure (Standford University)

Feed-forward networks are very good at classification problems and predictions. Since the primary task of this thesis is classifications, the feed-forward network type of neural network that we will use. To avoid misunderstanding, CNN uses a feed-forward type of network but uses back-propagation to train network through, and back-propagation do not have anything to do with the type of network or architecture of the network. It updates the weights, and it will be explained in the next section.

Figure 2.4 Two-layer neural network (Bishop, 2013)
The linear model is represented in Equation (2.5) for classification.

\[ y(x, w) = f\left( \sum_{j=1}^{M} w_j \emptyset_j(x) \right) \]  

(2.5)

Where \( f() \) represent a nonlinear function,

\( \emptyset_j(x) \) is basis function,

\( w_j \) is coefficients (it will be updated during training).

According to (Bishop, 2013), basis functions are used by neural networks, and each basis function itself is a nonlinear function of input’s linear combination. Moreover, the coefficients are adaptive parameters in the linear combination. Activations equation is shown in Equation (2.6).

\[ a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \]  

(2.6)

Where \( j=1, \ldots, M \),

\( x_i, \ldots, x_D \) is the inputs,

superscript (1) shows the layer of the network,

\( w_{ji}^{(1)} \) refers weights,

\( w_{j0}^{(1)} \) refers biases.

This formula is essential to know the activations of a layer. Each weight and bias are created differently for each input. When we multiply the weight of a input with input itself and sum up with bias then we get an activation function for just one input. If we sum all the activation function for all the inputs, then we will get the activation function of the layer. It can be applied for all layers.

If we use these activations in nonlinear activation function, it will give us Equation (2.7).

\[ z_j = h(a_j) \]  

(2.7)

Where \( h() \) activation function.

Output unit activations can be calculated by using Equation (2.8).
\[ a_k = \sum_{j=1}^{M} w_{kj}^{(2)} z_j + w_{k0}^{(2)} \]  

(2.8)

Where \( k=1,\ldots,K \), and \( K \) indicates the outputs numbers in total.

To get the network outputs, Equation (2.10), we can use Equation (2.9).

\[ y_k = \sigma(a_k) \]  

(2.9)

Where \( \sigma() \) is sigmoid function,

\[ \sigma(a) = \frac{1}{1 + \exp(-a)}. \]

\[ y_k(x, w) = \sigma \left( \sum_{j=1}^{M} w_{kj}^{(2)} h \left( \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \right) + w_{k0}^{(2)} \right) \]  

(2.10)

2.3.1 Hidden layers

The reason for calling them hidden layer is because they are not either input layer or output layer. Since they are not connected to the real world, those are called as hidden layers. Hidden layers are responsible for finding a feature from inputs. The working mechanism of the hidden layers resembles human learning mechanism. First, we see an object and our brain evaluate this information, but before doing that in the first place we recognise borders and corner, and then other features recognised. In hidden layers, simple features such as borders, and corners get recognised and then on the other layers in hidden layers start recognising more complex features.

CNN is a feed-forward network type. First inputs are received in the first layer then feature extraction occurs between the input layer and the output layer. Finally, classifications come true in the output layer. As it is shown in Figure 2.3, all information passes through layers, including hidden layers, from the first to the last.

While choosing the number of hidden layers, input difficulty should be considered. If the input is complex, then more hidden layers required.
3. CONVOLUTIONAL NEURAL NETWORKS (CNNs)

CNN is the type of ANN that we used in this thesis work as mentioned in the previous chapter. CNNs are quite powerful networks to classify the raw image. The reason for choosing it because it is the most efficient network that can learn from raw training data or inputs as mentioned in the previous section. As one can understand from this statement that, if the amount of training data increases, the accuracy of the system will increase too. Therefore, the input is an essential parameter for NNs. However, it is not always easy to gather enough data.

![Typical CNN architecture](image)

In this section, the chosen CNN architecture is going to be evaluated. The main idea behind this ANN is that establishing a connection between training data, if the network trained within a logical way, then the system will be capable of making a correct decision on a new data. This concept is explained in many ways, but broadly speaking, CNN is a feed-forward neural network as a structurally but uses back-propagation method (Y. LeCun et al., 1989) to train the network through. One of CNN architectures is shown in the Figure 3.1. If we take a closer look at Figure 3.1, we see that feature learning or feature extraction part is consist of convolutional layers and pooling layers. After weighted features, these get connected to fully connected layer or layers.

Training of the artificial neural networks is an essential part in order to use it. During training, the connection’s weights are adjusted using the input data. These weights play quite essential rules in the efficiency of the network. Moreover, if the weights are correct, we get the correct and accurate result. Input data is always different for different applications; weights are depended on the input data, which means learned weights cannot be implemented to another dataset, even if we can, the result is going to be quiet less. For instance, we have two ANNs, and these have the same...
architecture, but we train these two same ANNs with different datasets. As an outcome, we will have a different weight of connections.

There are many different ANNs in the field which are a solution for different problems. Those ANNs might have similar or different architecture, and the architecture of an ANN is chosen by an architect. As a rough sketch, architect defines the type and size of the ANN, put the different layers in order. Since building a new CNN up from scratch is possible but requires to know many parameters than using pre-trained networks it is not always an economical solution for tasks. There are pre-trained neural networks exist, and those are available to use in tasks such as AlexNet, Caffe (Caffe), VGG16 (Simonyan & Zisserman, 2014), and GoogleNet (Szegedy et al., 2015). The most suitable for our application is AlexNet. What considered while choosing is that if it is available for academic and industrial use, the input size (if it is too small then it is not applicable), and the speed. Moreover, we also needed to consider using this network in a personal notebook. It had to be light and quick to get a response.

The chosen CNN, AlexNet, has twenty-five layers which are input layer, five convolution layers, seven ReLU layers, two normalisation layers, three pooling layers, three fully connected layers, two dropout layers, probability or softmax layer, and final, classification, layer.

In the following section, these layers and their rules will be explained.

3.1 Convolution

CNN takes the images from first input layer and progress with convolution layer as a next step as shown in Figure 3.1. What convolution layer does is it takes a raw image as an input, and the image gets convolved. Convolution is done by using multiple learned kernels using shared weights. Convolution layer consists of parameters such as the size of maps, number of maps, and kernel size. Every layer has equal size of M maps \((M_x, M_y)\). Kernel size \((K_x, K_y)\). The convolution layer takes this kernel and shift over the input image starting from left most until right down (Zhou et al., 2017). In the following part, it will be explained using an example and other parameters that is used for this operation.
The kernel can be a matrix or vector which slides over the input. As it is shown in Figure 3.2, kernel represented as a 3×3 matrix with dark grey colour. In the figure, input represented as a 5×5 matrix with no colour and outputs represented as a 2×2 matrix with light gold colour. To calculate output, there are a couple of more parameters which affect the result. To slide and overlap the kernel matrix on the input matrix, we use a concept which is called as stride, and a 1×1 stride is used. It means that slide the kernel 1 row to the right after each calculation until the end of the matrix, and after that slide 1 column down and repeat the same method on the new row. After repeating this sliding until the right down corner of the input matrix, we generate a new matrix as shown with light gold colour in Figure 3.2. Padding is a concept that we did not use it in this example but what it does it that it adds zero to the input matrix for example if we define as 1×1. If we add 2×2 padding, then we will have an 8×8 matrix with having zeros on the first, second, seventh and eighth rows, and columns.

It may seem that using padding is unnecessary for some. The reason to use it of course to extend the size of the input. In this way, we will not have any problem with unusual kernels or sliding window size. Let’s give a simple example of using a 2×2 strides instead of 1×1 in Figure 3.2. The output matrix is a 2×2 matrix, but the problem is going to be fitting the kernel on the input matrix because 1 column of the kernel will not match with input matrix. Such in a case like this, using padding is the easiest way. The other way is that we could use more strides or kernel sizes. With making the stride matrix or kernel matrix bigger, we decrease the feature amount that we will gather from that window. Therefore instead of increasing stride size or kernel size, padding is a better solution. If we take a look at the layers in Appendix B, 9th section, 6th, 10th, 12th, 14th we will see that we use padding a lot, and in the other layers, we do not need.
Figure 3.2 Illustration of computing the output value of the convolution

<table>
<thead>
<tr>
<th>3</th>
<th>1</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3</th>
<th>1</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3</th>
<th>1</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3</th>
<th>1</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Input: $i_1 = 4$, and $i_2 = 5$

Kernel: $k_1 = 3$, and $k_2 = 3$

Stride: $s_1 = 1$, and $s_2 = 1$

Padding: $p_1 = 0$, and $p_2 = 0$

3.1.1 Convolution arithmetic

Depending on the size of the input, kernel, stride, and padding, output matrix size can be calculated by using following equations (Dumoulin & Visin, 2016).

- $s=1$, and $p=0$ (without using zero padding but one stride)

For any input and kernel sizes, with $s=1$ and $p=0$ (which means no zero padding is applied. Therefore there is no place in the equation), output size calculation formula is shown in Equation (3.1).

$$o = (i - k) + 1 \quad \text{(3.1)}$$
• **s=1, and p (with zero padding and one stride)**

For any input, kernel, and padding sizes, with s=1, output size calculation formula is shown in Equation (3.2).

\[ o = (i - k) + 2p + 1 \]  \hspace{1cm} (3.2)

• **Half or same padding**

In this thesis work we do not need to use half (same) padding but what it does is that it makes the output matrix same as input matrix as shown in Equation (3.3). It is applicable for any input, kernel odd (if \( k = 2n + 1 \)), s=1, and \( p = \left\lfloor \frac{k}{2} \right\rfloor = n \).

\[ o = i + 2 \left\lfloor \frac{k}{2} \right\rfloor - (k - 1) = i + 2n - 2n = i \]  \hspace{1cm} (3.3)

• **Full padding**

Full padding is a method when the output needs to be bigger than input size. When we need it, we can use Equation (3.4) for any input, and kernel size, \( p = k - 1 \), and s=1.

\[ o = i + 2(k - 1) - (k - 1) = i + (k - 1) \]  \hspace{1cm} (3.4)

• **p=0, and s (without using zero padding but using any strides)**

For any input, kernel, and s size, the output can be calculated by using Equation (3.5).

\[ o = \left\lfloor \frac{i - k}{s} \right\rfloor + 1 \]  \hspace{1cm} (3.5)

• **p, and s (with any padding and stride sizes)**

For any input, kernel, padding, and stride sizes, Equation (3.6) can be used to calculate output size.

\[ o = \left\lfloor \frac{i + 2p - k}{s} \right\rfloor + 1 \]  \hspace{1cm} (3.6)
3.2 Pooling

The rule of pooling is to decrease the size of feature maps. In a way, it summarises the shared input regions using average or the maximum value. The principle is almost same with discrete convolutions as shown in Figure 3.2. It works with a sliding window over the input matrix and feeds the pooling function with data which it takes from that window (Dumoulin & Visin, 2016).

If we stick to the same example than the representation of the pooling will be like as shown in Figure 3.3 using the 3x3 average value of the window, and in Figure 3.4 it uses 3x3 max pooling operation. In both operations, it uses 1x1 strides.

![Figure 3.3 Pooling with using average pooling operation](image1)

![Figure 3.4 Pooling with max pooling operation](image2)
Since we use square inputs, kernel size, strides and padding, we can define these parameters as shown in Equation (3.7), (3.8), (3.9), (3.10) and progress with these for the further calculations.

\[
\text{square inputs } (i_1 = i_2 = i), \quad (3.7)
\]

\[
\text{square kernel size } (k_1 = k_2 = k), \quad (3.8)
\]

\[
\text{square strides } (s_1 = s_2 = s), \quad (3.9)
\]

\[
\text{square padding } (p_1 = p_2 = p). \quad (3.10)
\]

### 3.2.1 Pooling arithmetic

The derived equation can also be used for pooling arithmetic. The difference here is that pooling does not use padding. Therefore, \(p\) parameter will not involve in the equation. The calculation of output matrix size can be done using Equation (3.11).

\[
o = \left[ \frac{i - k}{s} \right] + 1 \quad (3.11)
\]

### 3.3 Calculation of output values of convolution and pooling layer of the network

- Taking the data from the input layer to the first convolution layer (conv1)

Let’s call the input image as layer 0, and it has a 227x227x3 size. On the first layer, which is convolution layer, takes the 227x227x3 size image as input and uses stride as \(s=4\), using kernel as \(k=11x11x3\), and without using padding. With these parameters and Equation (3.5), the result can be seen in Equation (3.12). First convolution layer applies 96 filters and the memory represented in a 55x55x3x96 size. The weight matrix is 11x11x3x96.

\[
o = \left[ \frac{227 - 11}{4} \right] + 1 = 55 \quad (3.12)
\]
• **Taking the input from conv1 to the first pooling layer (pool1)**

After first convolution layer, first pooling layer is responsible for processing the input which applies max-pooling operation and takes a 55x55x96 matrix as an input and also applies a kernel as k=3, and stride as s=2. Output matrix is calculated by using Equation (3.11), and the output is going to be as shown in Equation (3.13). Memory size is 27x27x96.

\[
o = \left\lfloor \frac{55 - 3}{2} \right\rfloor + 1 = 27
\]  

(3.13)

• **Taking the input from pool1 to the next convolution layer (conv2)**

Second convolution layer takes the data from the first pooling layer as a 27x27x96 matrix and applies kernel k=5, stride s=1 and padding p=2 Equation (3.2) can be used to calculate the second layer’s output size and output size is shown in Equation (3.14). Second convolution layer applies 256 filters. Memory size is 27x27x256x3.

\[
o = (27 - 5) + 2 \times 2 + 1 = 27
\]  

(3.14)

• **Taking the input from conv2 to the next pooling layer (pool2)**

Second pooling layer takes the input in a 27x27x256 and applies max-pooling operation using a k=3 kernel, and s=2 stride. Output size can be calculated by using Equation (3.2) again, and the result is shown in Equation (3.15). Memory size is 13x13x256.

\[
o = \left\lfloor \frac{27 - 3}{2} \right\rfloor + 1 = 13
\]  

(3.15)

• **Taking the input from pool2 to the next convolution layer (conv3)**

The third convolution layer takes the 13x13x256 matrix as input and applies 384 filters and kernel k=3, padding p=1, and stride s=1. Equation (3.2) gives the result as in Equation (3.16). Memory is 13x13x384x2.

\[
o = (13 - 3) + 2 \times 1 + 1 = 13
\]  

(3.16)
• **Taking the input from conv3 to the next convolution layer (conv4)**

The fourth convolution layer takes the 13x13x384 matrix from the third convolution layer as input and applies kernel k=3, stride s=1 and padding p=1. The output matrix can be calculated by using Equation (3.2), and the result is going to be same with the previous layer as 13x13x384x2. Doing this, original size is kept, and more filter applied.

• **Taking the input from conv4 to the next convolution layer (conv5)**

The fifth convolution layer will take the input as 13x13x384x2 from the forth convolution layer and applies 256 filters along with kernel k=3, stride s=1, and padding p=1. The output size is calculated by using the same Equation (3.2), and the result is shown in Equation (3.17). Memory is 13x13x256x2.

\[ o = (13 - 3) + 2 \times 1 + 1 = 13 \]  \hspace{1cm} (3.17)

• **Taking the input from conv5 to the last pooling layer (pool5)**

As the last layer in this set of layers is going to be third and last pooling layer which is called pool5. This layer takes the input as 13x13x256x2 size from the fifth convolution layer and applies max-pooling operation with kernel k=3, and stride s=2. The output size can be calculated by using Equation (3.11), and the result is presented in Equation (3.18). Memory is 6x6x256.

\[ o = \left(\left\lfloor \frac{13 - 3}{2} \right\rfloor + 1 \right) = 6 \]  \hspace{1cm} (3.18)

### 3.4 ReLU (rectified linear unit)

ReLU is an activation function and it is used to introduce non-linear in the network. It takes place after each convolution and fully connected layer. Instead of using Tanh or sigmoid function, ReLU preferred in the network and applied on the hidden layers. The reason behind that is because it significantly trains the network faster than others (Krizhevsky, Sutskever, & Geoffrey E., 2012). The main problem with sigmoid function is that it gets saturated when it becomes tiny and it is called vanishing gradient problem. While using ReLU, it avoids saturation issues.
ReLU gives 0 output if the input smaller than zero, negative. If it is greater than 0, it gives equal output value with the input value. The mathematical representation of ReLU is shown in Equation (3.19).

\[ f(x) = \max(x, 0) \]  

(3.19)

According to Figure 3.5, the output of the neuron is presented in Equation (3.26), and Equation (3.31) (Alom et al., 2018).

\[ v_k = \sum_{j=1}^{m} w_{kj} = x_j \]  

(3.30)

\[ y_k = \varphi(v_k + b_k) \]  

(3.31)

Where  

- \( x_j \) is input,  
- \( w_{kj} \) is weight,  
- \( v_k \) is linear combination of input,  
- \( \varphi(\cdot) \) is activation function (it is ReLU in our case),  
- \( b_k \) is bias,  
- \( y_k \) is the output value.
3.5 Stochastic gradient descent with momentum (sgdm)

'Sgdm' is a short form of stochastic gradient descent with momentum. It is an optimizer of the network. It is a parameter that is used for our task too. The momentum is a value that can be adjusted between 0 to 1. In our case, it is defined as 0.9000. This parameter is responsible for updating iteration parameters, and it introduces velocity variable within stochastic gradient descent algorithm. With adding momentum to stochastic gradient descent, we decreased the possibility of having oscillation. Boris Polyak proposes this method and for more in-depth reading see the article (Polyak, 1964). This method is a first-order optimisation algorithm, and it is used to find where the local minima are. The pseudo code of the stochastic gradient algorithm with momentum is explained in the Algorithm I. Equation (3.33) shows the sgdm update. $\gamma(\theta_l - \theta_{l-1})$ part of the equation comes from momentum. Absence of this part, it is equation of gradient descent.

$$\theta_{l+1} = \theta_l - \alpha \nabla E(\theta_l) + \gamma(\theta_l - \theta_{l-1})$$  

(3.32)

Where $\gamma$ is the contribution of the previous gradient step to the current iteration, $l$ is iteration number, $E(\theta)$ is loss function, $\nabla E(\theta)$ is the gradient of the loss function.

**Algorithm I: Stochastic gradient descent algorithm with momentum**

**Inputs:** Learning rate $\epsilon$, momentum parameter $\alpha$, initial parameter $\theta$, initial velocity $v$

**Outputs:** Optimum $\theta$ which minimises $\epsilon$

While stopping criterion not met do

Sample a minibatch of $m$ examples from the training set $\{x^{(1)}, \ldots, x^{(m)}\}$ with corresponding targets $y^{(i)}$

Compute gradient estimate: $g \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(x^{(i)}; \theta), y^{(i)})$

Compute velocity update: $v \leftarrow \alpha v - \epsilon g$

Apply update: $\theta \leftarrow \theta + v$

end while
A more natural way to explain the behaviour of momentum is going to be described using figures. If we take a closer look on the left side of Figure 3.6, we will see that defining a constant value for sgd cause taking small step sizes (also known as learning rate) each time and it make the process a lot longer for each iteration. It is a bit more accurate than sgd with momentum, but in general, the absence of momentum makes the algorithm less efficient, more time consuming and it makes it unpractical in many applications. However, with momentum, it takes less time to find the lowest part or local minima as shown on the right side of Figure 3.6. The reason behind that it takes more steps, and less time to get there, but the issue with sgd with momentum is that it makes it a bit inefficient. Missing the local minima is possible. Since the iterations take a lot shorter, and accuracy of the algorithm preferably better than other ones, it makes it more reasonable to use. Therefore in our case, we have between sixty to ninety iterations while training, sgd with momentum is the chosen one and defined as 0.900 as default for each iteration.

Figure 3.6 Behaviour of sgd without momentum (on the left side) and behaviour of sgd with momentum (on the right side) (Sargur & N.)

3.6 Optimization

The main reason to make optimisation is to find weights, \( w \), which is going to minimise the loss function (CS231). Lost function is calculated by using error which comes from wrongly weighed the weights, and then backpropagation gives feedback to the previous layer to update the weight values as shown in Figure 3.7. There are many optimisers for a variety of applications. Since we already use backpropagation algorithm, it is entirely compatible to use stochastic gradient descent algorithm with together as explained in Efficient BackProp article by Yann Lecun in 1998 (Yann, 1998), for information the back-propagation is proposed by (Rumelhart, Hinton, & Williams, 1986).
With training dataset, back propagation algorithm is used to train feed-forward neural networks to create input, output pattern. Briefly, it uses the dataset to learn parameters between them and uses the same parameters for new images. In this sense, learning means to find the closest set of weights or finding the lowest cost error. To minimise the error, it uses gradient descent technique.

Figure 3.7 Back-propagation illustration (Cireşan et al., 2011)

For further reading on back-propagation and activation functions see the article (Sibi, Jones, & Siddarth, 2013), and pseudo code of back-propagation algorithm is shown in Algorithm II (Alom et al., 2018).

Algorithm II: Back-propagation

**Inputs:** $l$ represents the number of layers of a network, $\sigma_i$ represents the activation function, $h_i = \sigma_i(W_i^T)h_{i-1} + b_i$ the equation gives the output of hidden layer and $\hat{y} = h_i$ gives the network’s output

The gradient is computed by the equation of $\delta \leftarrow \frac{\partial_e(y_\hat{y})}{\partial y}$

**For** $i \leftarrow l$ **to** 0 **do**

Calculation of the current layer’s gradient:

$$\frac{\partial_e(y_\hat{y})}{\partial W_i} = \frac{\partial_e(y_\hat{y})}{\partial h_i} \frac{\partial h_i}{\partial W_i} = \delta \frac{\partial h_i}{\partial W_i}$$

$$\frac{\partial_e(y_\hat{y})}{\partial b_i} = \frac{\partial_e(y_\hat{y})}{\partial h_i} \frac{\partial h_i}{\partial b_i} = \delta \frac{\partial h_i}{\partial b_i}$$

Gradient descent is applied by using $\frac{\partial_e(y_\hat{y})}{\partial W_i}$ and $\frac{\partial_e(y_\hat{y})}{\partial b_i}$ equations

**For** the lower layer

$$\delta \leftarrow \frac{\partial_e(y_\hat{y})}{\partial h_i} \frac{\partial h_i}{\partial h_{i-1}} = \delta \frac{\partial h_i}{\partial h_{i-1}}$$

End
3.7 L2 (gaussian) regularization

Regularizations are handy terms to reduce overfitting. Since overfitting affects the result and makes the system unstable, we use L2 regularization, which is also called weight decay. Either we have lots of data or not, using regularization plays quite an essential role in the classification part (P. Murphy, 1991).

\[
f'(e) = NLL(w) + \lambda w^Tw \\
g'(w) = g(w) + \lambda w \\
H'(w) = H(w) + \lambda I
\]

3.8 Fully connected layer

The output of neurons of the last pooling is connected to all the fully-connected layer’s neurons. Therefore, it is called as fully connected layer. In our case, since the first fully connected layer takes the input from the last pooling layer, the input of the fully connected layer is a 6x6x256 matrix. Which means if we multiply these numbers, we will get 6*6*256=9216 pixels. These pixels will feed all the fully connected layers. All fully connected layers have 4096 neurons, but the last one has two-fully-connected layer as we modified that layer. The reason of having two fully connected layers is that it is always connected with the number of classes.

3.9 The activation function of the neural network

There are two classifiers which are most commonly used by the researchers and in the industry. One of them is support vector machine, called SVM with a short form, and the other one is softmax. SVM is used for supervised learning and algorithms which uses data for classification and regression field. SVM is proposed by (Vapnik & Lerner, 1963). For further reading on SVM pattern recognition see the article (Burges, 1998). While SVM generates similarity score, softmax generates similarity probability. On the other hand, softmax is a standard function to use in CNNs (Zhou et al., 2017).
Therefore, softmax is a function that we use between last fully connected layer and classification or final layer. Softmax calculates the loss function with using cross entropy.

Since we use ReLU in the network, softmax is possibly better options than others. The other advantage of softmax is that ReLU will not cause any problems such as vanishing. The role of softmax is that it squashes the output of the previous layer’s output to fit in a range between 0 and 1. As shown in Equation (3.26), softmax divides each output value by the total of output values, that should equal to 1.

\[
\begin{bmatrix}
1.2 \\
0.9 \\
0.4
\end{bmatrix} \rightarrow \text{SOFTMAX} \rightarrow \begin{bmatrix}
0.46 \\
0.34 \\
0.20
\end{bmatrix}
\] (3.26)

As the mathematical representation of softmax is given in Equation (3.37).

\[
\sigma(z_j) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}
\] (3.37)

The output of softmax function gives the probability of having a relevant object in the input that received in the first layer, or not.

### 3.10 Dropout

Dropout is proposed by (Hinton et al., 2012), (Srivastava et al., 2014) to reduce overfitting as a crucial problem in deep learning. The network can be trained using such a wide dataset, in the end, we can get an insufficient result because of overfitting. Dropout is used to reduce the overfitting of the training set. How it does is it drops some units and their connections as shown in Figure 3.8. In the figure, there are two hidden layers and crossed neurons are dropped. Therefore, it is called dropout. In supervised learning task like ours, dropout is an essential parameter. For further reading on dropout see the article (Srivastava et al., 2014). In our system, we used two dropouts after the first and second fully connected layers. The value was 50%, and we dropped 4096/2=2048 neuron out. Since the last fully connected layer has two neurons, there is no need to apply dropout to that layer.

38
Figure 3.8 (a) Network without dropout, (b) Dropout applied network, (Srivastava et al., 2014)

In Figure 3.9, we can see how dropout makes differences. Classification error is reduced around 1% which also makes the ANN robust. Dropout also forces the network to use all the neurons randomly. Absence of dropout, one neuron can learn the same feature over and over again, and in the end the possibility of learning for new features for that neuron reduced.

Let’s assume that we have L hidden layers.

Where \( l \in \{1, ..., L\} \) is representation of hidden layers of network,

\( z^l \) is the vectoral representation of the input into layer \( l \),

\( y^l \) is the vectoral representation of the output from layer \( l \),

\( 0^{th} \) layer is the input layer. Therefore \( y^0 = x \). And \( x \) is the input,

\( w^{(l)}, and b^{(l)} \) is the weights and biases at layer \( l \) respectively,

\( l \in \{1, ..., L - 1\}, i \) represents any hidden layers.

39
The feed-forward operation is written as shown in Equation (3.38), (3.39) for standard network (without dropout) (Srivastava et al., 2014).

\[
z_i^{(l+1)} = w_i^{(l+1)}y_i^{(l)} + b_i^{(l+1)} \quad (3.38)
\]

\[
y_i^{(l+1)} = f(z_i^{(l+1)}) \quad (3.39)
\]

Where \( f \) is activation function, and \( f(x) = 1/(1 + \exp(-x)) \).

Once dropout is applied, then the feed-forward operation can be shown as shown in Equation (3.40), (3.41), (3.42), (3.43) (Srivastava et al., 2014).

\[
r_j^{(l)} \sim \text{Bernoulli}(p) \quad (3.40)
\]

\[
\tilde{y}^{(l)} = r^{(l)} \ast y^{(l)} \quad (3.41)
\]

\[
z_i^{(l+1)} = w_i^{(l+1)}\tilde{y}^{(l)} + b_i^{(l+1)} \quad (3.42)
\]

\[
y_i^{(l+1)} = f(z_i^{(l+1)}) \quad (3.43)
\]

Where \( \ast \) is for element-wise multiplication.
4. EXPERIMENTS

There are couple of parameters which can affect the efficiency of the system such as running the algorithm by using CPU instead of GPU, having not enough dataset, choosing the mini-batch size bigger than your GPU can afford, overfitting/underfitting which comes from training the system longer or shorter than optimum time(wrong number of epochs), respectively.

Having the most accurate algorithm that is available for use is not an issue in some cases if it is not applicable to our problem. In this section, the experimental results will be evaluated, and parameters will be changed to get the optimum result. Before going to represent the results, it is useful to provide the specification about the system that is used to run this algorithm on. This information might be useful for the reader to provide a better understanding. The laptop that we use has an Intel(R) Core(TM) i5-7300HQ CPU at 2.50GHz up to 3.50GHz (Intel), 8GB DDR4 memory with 2400MHz, and Nvidia GeForce GTX 1050 GPU (Nvidia), 4GB DDR5.

Matlab (matrix laboratory) is used to run the algorithm as a numerical computing environment using toolboxes (Matlab, 2018). To use the GPU, and Matlab efficient, there are some additional toolboxes that we used for Matlab, and those are provided in the bibliography section.

In this section, setting the algorithms up will be presented. The basic idea in this section is to show how we can implement a pre-trained network to our system and the results. Since CNN algorithm can be divided into three part such as input, feature extraction using input data, and classification with features that it learned from the previous step. In this logic this experimental part shaped.

4.1 Creating a dataset for the input stage

To give a better understanding to the reader, we put the steps in pseudo code shape and it is provided as Algorithm III. In this section, the dataset will be created for the experimental part. Created data will take place in the algorithm, and before progressing with image pre-processing will be done.
Algorithm III: Creating a dataset for the input layer

**Inputs:** File location where the images will be taken from, *path*

**Outputs:** General dataset folder, *datastore*; images that will be used for training, *trainingdatastore*, images that will be used for testing *testingdatastore*

READ *path* // read the path of the dataset where it is placed

*Filename = path* // save the file path in the filename variable

READ *filename*

CREATE *trainingdatastore*

\[ \text{datastore} \leftarrow \text{filename} \] // create datastore with reading data from filename

READ folder names, which are placed in *trainingdatastore* and save those as *labels of those images*

SAVE name of the *labels* into two variables, which means into two different classes // in our cases correct products and false products

CREATE *trainingdatastore* and *testingdatastore*

\[ \text{testingdatastore} \leftarrow \text{trainingdatastore}^\ast p \] // \( p \); proportion, take the defined proportion of trainingdatastore randomly by the user and save as trainingdatastore and do not train the network with those images

CALL the *datastore* for image pre-processing

READ images from *datastore*

If the image is different from the 227-by-227 pixel

THEN resize the image

SAVE the new images in the same path

ENDIF

IF the image does not have three colour channels

THEN convert to 3 colour channels

ENDIF

As occasions require a new dataset, the script needs to take the input data from a dataset which we defined. Since we use wrenches as an object to recognise, root file defined to use it for creating “wrenchdatastore” in the script as shown in Figure 4.1.

```matlab
filename=('C:\Users\aydin\OneDrive\All About Master Thesis\MATLAB FILES AND DATASET\WrenchDataset') \*The file location where datastore is
```

Figure 4.1 Root file for datastore where the system takes the data as input
Now, the system knows where the data is stored, but it does not have defined value on “wrenchdatasetstore”. To create a dataset using root file, we need to give access to take the dataset from a defined folder with its subfolders. In our case, “IncludeSubfolders” command will work for this application. In Figure 4.2, it is shown that it takes the datastore from a defined place with subfolder and extracts the file names.

```
% Command to display the images in the current folder and
% Imports in network

wrenchdatasetstore=imageDatstore('C:\Users\syahid\OneDrive\All About Master Thesis\MATLAB FILES AND DATASET\Wrench Dataset', 'Inc
ludeSubfolders', true); %Creating a datastore for all products
filenames=wrenchdatasetstore.Files; %Extracting the file names from datastore
wrenchnames=wrenchdatasetstore.Labels; %Command to show labels

labels=labels_CountEachLabel(wrenchdatasetstore); %Counts each label which are in datastore

images=readimage(decentwrenchdatasetstore, 5); %Reading an image for testing the

```

Figure 4.2 Creating datastore variables

Already taken datastore master file and subfolders of its are ready to progress with. What we need to do is that we need to read names of the folder. We need to do it to label the images in those files with less effort because in our case, we take more than six hundred images from datastore. In some applications, it goes up to thousands of images. Therefore, it is not even an issue to label each image one by one. In this sense, we need to use more this solution as a practical solution for now and for further applications. ‘LabelSource’ and ‘foldernames’ commands read the information from folder name and put on each image as a label as it is shown in Figure 4.3. Doing this, we got rid of labelling more than six thousand images. Wrenchdatasetstore folder has two subfolders which are called as “DecentWrenches” which consist of 484 wrench images, and “DefectiveWrenches” which has 130 wrench images as shown in Figure 4.4. Each decent wrench image has brand new or almost new wrench object in it, and defective wrench images have low quality, broken, crushed or rusty wrench objects in it. Training the ANN with the proper dataset is a critical part as a starting point. Once we use not well-labelled inputs, then the ANN will be confused and decrease the result by increasing confusion matrix. Confusion matrix will be explained in the following part.

```

wrenchdatasetstore=imageDatstore('C:\Users\syahid\OneDrive\All About Master Thesis\MATLAB FILES AND DATASET\Wrench Dataset', 'Inc
ludeSubfolders', true, 'LabelSource', 'foldernames'); %Creating a datastore for all products, and taking label names from folde
rs

wrenchnames=wrenchdatasetstore.Labels; %Command to show labels

```

Figure 4.3 Labelling the images with the name of the folder where they belong

All the images are taken from (IMAGENET), (Krizhevsky, Sutskever, & Hinton, 2012), and undoubtedly, those are not pre-proceeded. Available to use in the academic environment. Those
have different sizes, colour and brightness adjustment, and not necessarily natural photos. The
dataset that we gathered is presented in the Appendix A.1 and Appendix A.2.

Figure 4.4 Datastore folder with its subfolder with their files

Having the datastore for training, we can develop our CNN algorithms, but there is still missing part
which is we do not have a dataset to validate or test our algorithm. If we do not have it, then we
will not be able to test the performance of our network. Therefore, we also need another dataset
to test the system. We can do it two ways. One of these ways is that we can create another folder
and call it as “testdatastore” and use it, or we can split some of our images to validate or test the
system on the command base. The reason behind not choosing the first way is that we do not have
so many images in the dataset that we can put some of them into another folder. If we do that, we
will have fewer data to train the system, and it will decrease the efficiency of our system. Therefore,
the second way is much more convenient to apply. It is easy to split some of the images for training,
and another plus is that once we need to change the dataset or product, there is going to be less
work to do. As shown in Figure 4.5, we can create test dataset within the one-line code.

Figure 4.5 Splitting data for training and testing

If we take a look closer to the code which is presented in Figure 4.5, we will see that we split
randomly six of ten images for training and four of ten images for testing. It is always possible to
change the proportion of train/test image, and this is another plus if we need to train the network
with more or fewer data. With having adjustable parameters in the algorithm, we will be able to reach the ideal success rate in the end. It is also possible to use value instead of a portion.

As we have a defined dataset folder which is consist of two subfolders of labelled images, we can progress with. AlexNet has twenty-five layers, and the first layer of it is an input layer. It means that first layer should correspond to inputs. Our dataset consisted of different image sizes as it is mentioned before. Predefined input size for AlexNet is $[227, 227, 3]$. The input size of AlexNet can be extracted from the network with running the code which is provided in Figure 4.9. The answer is shown as $[227, 227, 3]$ in Figure 4.10. It means the first layer of the architecture needs an image with the size of 227 by 227. Moreover, three stands for colour channels. The issue here is that there is going to be an error if the input is corresponded by the first layer without changing any parameters. To adapt the input size to the network, we can either change the first layer parameter of AlexNet or make image processing to get the desired image size. The preferred solution is image pre-processing. Because our dataset consists of many different images sizes, and 227 by 227 is suitable for our application.

```matlab
wrench datastore.ReadFcn = @readPreprocessedImage; % Reading the images which are stored in the datastore for preprocessing
function Iout = readAndPreprocessedImage(filename) % Creating a function for further application, defining variables, and resizing images as 227 by 227 for the network
    I = imread(filename);
    Iout = imresize(I, [227, 227]);
    if ismatrix(I) % Converting the grayscale image to colour if there are
        I = cat(3, I, I, I);
    end
    Iout = imresize(I, [227, 227]); % Saving the new images in required size
end
```

Figure 4.6 Image pre-processing for input layer

Image pre-processing should take the image from the dataset and save it there. Doing this, we will be able to use the same input size for training and testing. The code is shown in Figure 4.6. If we take closer to the code, we will see that it also generates grey images to colourful image with using the command:

```matlab
If ismatrix(I)
    I = cat(3, I, I, I);
end
```

Converting grey images to colourful images is good because more input improves results. Cropping and stretching an image is also possible to use in some application.
Now we can progress with exist dataset with desired parameters and image size, but before that, there are some parameters should be changed in some layers.

### 4.2 Modifying the ANN with valid parameters

Creating neural network requires so many data and computing time to get a reasonable result. For instance, AlexNet is trained with more than one million images in almost seven days without stopping. Since it takes so much time and energy, it is not something that everyone can afford with one GPU card, neither can be a solution in the industry with ease.

There is always an option that we can create a neural network with few layers and reasonable parameters, but then the results will not be satisfactory, or it will take some time to get good results. It might not be a good solution with increased project time and budget.

<table>
<thead>
<tr>
<th>Algorithm IV: Modifying pre-trained network to our system</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs:</strong> Pre-trained network, <em>AlexNet</em></td>
</tr>
<tr>
<td><strong>Outputs:</strong> Modified network, <em>deepnet</em></td>
</tr>
<tr>
<td>READ <em>AlexNet</em></td>
</tr>
<tr>
<td>SAVE</td>
</tr>
<tr>
<td><em>deepnet ← AlexNet</em> // saving AlexNet network as deepnet to modify layers*</td>
</tr>
<tr>
<td>READ <em>layers of AlexNet</em> // AlexNet consists of twenty-five layers*</td>
</tr>
<tr>
<td>CREATE a new <em>fully connected layer</em> for the twenty-third layer*</td>
</tr>
<tr>
<td>CREATE a new last layer, <em>classification layer</em>, for the twenty-fifth layer*</td>
</tr>
<tr>
<td>REPLACE the twenty-third layer with a newly created <em>fully connected layer</em></td>
</tr>
<tr>
<td>REPLACE the last layer with newly created <em>classification layer</em></td>
</tr>
<tr>
<td>READ the information from the first layer to clarify input size</td>
</tr>
<tr>
<td>READ the information from the layer for <em>training options and parameters</em></td>
</tr>
<tr>
<td>SAVE the modified network</td>
</tr>
</tbody>
</table>

Since AlexNet is an open and free CNN algorithm that everyone can use, it is another option. On the other hand, AlexNet uses so many parameters that we do not need such as thousand of the fully connected layer which will decrease the performance of our algorithm if we use it without
changing it. What suggested is that to create new layers and implement those into AlexNet. With this solution, it is going to be the most satisfactory solution for this task. Modification of pre-trained network for our case is explained in pseudo code in Algorithm IV.

First of all, to use AlexNet, it should be called by typing AlexNet in Matlab environment. For further applications, we added into a variable as shown in Figure 4.7, and the system gives general information about AlexNet as shown in Figure 4.8.

```matlab
deepnet=alexnet  %loading pretrained network
```

Figure 4.7 Calling AlexNet in a variable

deepnet =

SeriesNetwork with properties:

Layers: [25x1 nnet.cnn.layer.Layer]

Figure 4.8 Basic information about AlexNet

AlexNet uses pre-defined input size and input information can be extracted from layers as shown in Figure 4.9. Firstly, layers should be extracted using “deepnet” variable. Then we will be able to define new layers and replace those layers with AlexNet layers. As shown in Figure 4.9, output layer is defined as a new variable to replace in the twenty-fifth layer of Alexnet in the next section.

```matlab
layers=deepnet.Layers;  %Saving layers for further use
inlayer=layers(1)  %Extracting the first layer to read features of it
inputsize=inlayer.InputSize  %Extracting the input size for further use to resize the images in datastore
outlayer=layers(25)  %Extracting the last layer to modify in order to get better performance
categorynames=outlayer.ClassNames;  %Extracting the class names
```

Figure 4.9 Modifying the NN architecture for our application
inlayer =

    ImageInputLayer with properties:

        Name: 'data'
        InputSize: [227 227 3]

    Hyperparameters
        DataAugmentation: 'none'
        Normalization: 'zerocenter'

Figure 4.10 Input layer information

The reason to replace the last layer of the AlexNet, which is classification layer, is because it uses a thousand categories (as shown in Figure 4.11) to classify, and we use two different classes. Therefore, with replacing the layer(25) with a new one, basically, we will get rid of thousand fully connected layers and speed up the process time.

outlayer =

    ClassificationOutputLayer with properties:

        Name: 'output'
        ClassNames: [1000x1 cell]
        OutputSize: 1000

    Hyperparameters
        LossFunction: 'crossentropyex'

Figure 4.11 Output layer of the AlexNet

Since we have defined the layers that we are going to replace, we can progress with defining the new layers. As it mentioned already the twenty-third layer will be two fully connected layer and last layer will be classification layer for our application.

    fullyconnectednewlayer=fullyConnectedLayer(2) %2 fullyconnected layer for 2 kind of wrench label
    layers(23)=fullyconnectednewlayer; %modifying the fully connected layer with a new one
    layers(25)=classificationLayer %modifying the last layer with actual classes

Figure 4.12 Modifying the network layers

As shown in Figure 4.12, the new layers took place in the network. The response of the algorithm is shown in Figure 4.13 with new output size.
As the last step, for information, we extract weight representation from the network and call it as ‘First convolutional layer weights’ as shown in Figure 4.14. Weights of the first layer can be observed in Figure 4.19. As mentioned in the third chapter, in the first convolution layer, system detects ninety-six weights.

```matlab
weight1=deepnet.Layers(2).weights; % Code, in order to get weights for the second convolutional layer
weight1=mat2gray(weight1); % Scaling and resizing the weights for visualization
fig2=figure
montage(weight1)
title('First convolutional layer weights')
```

The final network structure is defined, and the entire structure of the network is shown in Table 4-1. During training and testing this structure used. In the next section, we will show the setting of the training and testing options.
Table 4-1 Modified layers of the network that will be used for the further applications

<table>
<thead>
<tr>
<th>LAYERS</th>
<th>NAME OF THE LAYER</th>
<th>TYPE OF THE LAYER</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>‘data’</td>
<td>Image Input</td>
</tr>
<tr>
<td>2</td>
<td>‘conv1’</td>
<td>Convolution</td>
</tr>
<tr>
<td>3</td>
<td>‘relu1’</td>
<td>ReLU</td>
</tr>
<tr>
<td>4</td>
<td>‘norm1’</td>
<td>Cross-Channel Normalization</td>
</tr>
<tr>
<td>5</td>
<td>‘pool1’</td>
<td>Max Pooling</td>
</tr>
<tr>
<td>6</td>
<td>‘conv2’</td>
<td>Convolution</td>
</tr>
<tr>
<td>7</td>
<td>‘relu2’</td>
<td>ReLU</td>
</tr>
<tr>
<td>8</td>
<td>‘norm2’</td>
<td>Cross-Channel Normalization</td>
</tr>
<tr>
<td>9</td>
<td>‘pool2’</td>
<td>Max Pooling</td>
</tr>
<tr>
<td>10</td>
<td>‘conv3’</td>
<td>Convolution</td>
</tr>
<tr>
<td>11</td>
<td>‘relu3’</td>
<td>ReLU</td>
</tr>
<tr>
<td>12</td>
<td>‘conv4’</td>
<td>Convolution</td>
</tr>
<tr>
<td>13</td>
<td>‘relu4’</td>
<td>ReLU</td>
</tr>
<tr>
<td>14</td>
<td>‘conv5’</td>
<td>Convolution</td>
</tr>
<tr>
<td>15</td>
<td>‘relu5’</td>
<td>ReLU</td>
</tr>
<tr>
<td>16</td>
<td>‘pool5’</td>
<td>Max Pooling</td>
</tr>
<tr>
<td>17</td>
<td>‘fc6’</td>
<td>Fully Connected</td>
</tr>
<tr>
<td>18</td>
<td>‘relu6’</td>
<td>ReLU</td>
</tr>
<tr>
<td>19</td>
<td>‘drop6’</td>
<td>Dropout</td>
</tr>
<tr>
<td>20</td>
<td>‘fc7’</td>
<td>Fully Connected</td>
</tr>
<tr>
<td>21</td>
<td>‘relu7’</td>
<td>ReLU</td>
</tr>
<tr>
<td>22</td>
<td>‘drop7’</td>
<td>Dropout</td>
</tr>
<tr>
<td>23</td>
<td>‘fc8’</td>
<td>Fully Connected</td>
</tr>
<tr>
<td>24</td>
<td>‘prob’</td>
<td>Softmax</td>
</tr>
<tr>
<td>25</td>
<td>‘output’</td>
<td>Classification Output</td>
</tr>
</tbody>
</table>

4.3 Feature extraction from the input

We are already able to read and pre-process images from defined datastore as explained in the first section. We have also modified the network, and now we have a proper network structure which is fully compatible with our application and inputs. As we know that feature extraction requires input data and network structure, having these essential parameters, we can progress with feature extraction part.

Before progressing with feature extraction, we need to define some more parameters which are called training options. Adjusting training option is also part of feature extraction, because with training options parameters, we will define the efficiency of the algorithm, and we always change or interrupt these options. As shown in Figure 4.15, we introduced stochastic gradient descent with...
momentum (sgdm). Sgdm already explained with its crucial parameters in the third chapter. Beside sgdm, we also specified initial learning rate as 0.001. If we take a closer look at Figure 4.16, we will see that regularization value is defined as a default value, and momentum is 0.900 as mentioned in the previous chapter.

```matlab
trainingopt = trainingOptions('sgdm', ...
   'MiniBatchSize',8, ...)
   'MaxEpochs',8, ...
   'InitialLearnRate',1e-5, ...
   'ValidationData',wrenchValid, ...
   'ValidationFrequency',4, ...
   'ValidationPatience',Inf, ...
   'Verbose',false, ...
   'Plots','training-progress');
```

Figure 4.15 Training options

```
trainingopt =
    TrainingOptionsSGDM with properties:
    
    Momentum: 0.9000
    InitialLearnRate: 1.0000e-03
    LearnRateScheduleSettings: [1x1 struct]
    L2Regularization: 1.0000e-04
    GradientThresholdMethod: 'l2norm'
    GradientThreshold: Inf
    MaxEpochs: 30
    MiniBatchSize: 128
    Verbosity: 1
    VerbosityFrequency: 50
    ValidationData: []
    ValidationFrequency: 50
    ValidationPatience: 3
    Shuffle: 'once'
    CheckpointPath: ''
    ExecutionEnvironment: 'auto'
    WorkerLoad: []
    OutputFcn: []
    Plots: 'none'
    SequenceLength: 'longest'
    SequencePaddingValue: 0
```

Figure 4.16 Training option's parameters

```matlab
[deepnet, info] = trainNetwork(wrenchTrain, layers, trainingopt);
testpredictions = classify(deepnet, wrenchTest);
```

Figure 4.17 Training the network and classifying the test data

Deepnet is the modified pre-trained AlexNet. In Figure 4.17 on the first line, we train the system using our created training dataset, modified layers and the training options that we defined. Once
we run the code, CNN starts learning. And then we are able to use the learned features from the training dataset to test the novel dataset, which is test dataset.

As the last step, we need to see the result of the test prediction. To do that we define calculations on the script and add a graph for the graphical representation as shown in Figure 4.18.

```matlab
actuallabels = wrenchTest.Labels; % Extracting the test labels
numberofcorrect = nnz(testpredictions == actuallabels) % Calculating the correct prediction amount
percentageofcorrect = numberofcorrect/numel(testpredictions) % Success rate
[wrenchconf, wrenchnames] = confusionmat(wrenchTest.Labels, testpredictions) % Defining and calculating the confusion matrix
fig6 = heatmap(wrenchnames, wrenchnames, wrenchconf) % Visual representation of confusion matrix
```

Figure 4.18 Evaluating the results of predictions on test data

In the last section, evaluation part takes place with the results.

### 4.4 Results

In this section, results will be evaluated. The response of the system will be provided, and the behaviour of the system will be explained. Since this is an experimental part, we will be checking the parameters to make the system better.

In Figure 4.19, we can see the first convolutional layer weights which we explained in the third chapter, the first section.
Almost in every chapter, we mentioned the importance of the dataset. Dataset is the most crucial part of ANN and one cannot be sure that if the dataset would lead them to underfitting or overfitting. To avoid this problem, we used an algorithm which eliminates this probability of overfitting, regularisation.

There are many ways to improve the result with mentioned methods in the previous chapter such as changing the learning rate, using another type of activation functions, changing the momentum value for stochastic gradient descent momentum, using svm instead of softmax.

When we run the code, which is provided in Figure 4.17, we get the result as shown in Figure 4.20. It takes one and half minute to train and test the network. During this training, it makes 6 epochs and 90 iterations. The accuracy of the network reached the maximum accuracy at the second epoch, but then it decreased to 90% in the end. While this was happening, loss dramatically dropped under 0.2 in the second epoch, and in the end it was almost 0.2. Test accuracy increased until 80% in the second epoch, where the training accuracy reached the max, and then it kept steady. The test loss stayed at 0.6 at the end of the progress. The result seems promising but it is not, because of the difference between validation error and train error. In good fit systems, validation errors should be slightly bigger than training error. Therefore this configuration does not work stable.
• Training result which is shown in Figure 4.20 has almost 81% validation accuracy. Training error is quite low that we would use this configuration unless validation error was not that much different than training error. Therefore this is not a good fit that we can use.

Figure 4.20 Training and validation with 6 epochs and 90 iterations

• The result which is shown in Figure 4.21 has 77% validation accuracy, and validation error is slightly higher than training error. This configuration might be applicable if there is no better result.

Figure 4.21 Training using minibatch 36, max epoch 8, learning rate 1e-5
• Figure 4.22 shows training which has 77% validation accuracy, and it is pretty quick too. Validation error is almost doubled the training error which is not a good result.

Figure 4.22 Training using minibatch 36, max epoch 10, learning rate 1e-4

• With almost 81% validation and good fit validation-training error, the result which is shown in Figure 4.23 has the most applicable configuration. To predict the test dataset, this configuration will be used.

Figure 4.23 Training using minibatch 8, max epoch 8, learning rate 1e-5
With the configuration that tested and got the most applicable result in Figure 4.23, we will use this configuration while testing our novel test dataset. To predict test dataset we run the second line of Figure 4.17, and prediction starts. At this point, the network is going to use the learned features on the new datasets. Each image will be handled from the beginning of the script. Firstly, the image size will be checked. If the size is not correct that image resizing will take place. After that with using convolution and pooling as explained in the third chapter will take place. In this stage, the network will be looking the similar patterns on the new images. If the network finds the similar pattern with the learned one, then classification will happen.

The prediction result of the all test dataset, which is consist of hundred and thirty-seven images, is shown in Figure 4.24. The result almost 82% successfully.

numberofcorrect =
137

percentageofcorrect =
0.8135

wrenchconf =
70 14
17 67

Figure 4.24 The prediction result of the test dataset with the confusion matrix

![Confusion Matrix](image)

Figure 4.25 Graphical representation of the confusion matrix
As shown in Figure 4.25, the confusion matrix shows the system’s weakness. This result was expected. Since we trained system with using the same products but minor/major mistakes, it is natural that we got this confusion matrix. Usually, CNNs are used for recognition of different kind of products, which does not make this much confusion at some points.

The good thing with this system is it showed the capability of making a decision on similar products with small differences, and it is a ready-working algorithm that can be used for any kind of products, and classification problems. Being flexible and applicable makes the system promising and adaptable to industrial tasks.
5. CONCLUSIONS

In the first section of this chapter, we will refer possible future works with referring literature. This part is going to be the literature step of the future works. In the second section of the chapter, the summary will take place, and it is going to be the last section of this thesis work.

5.1 Future work

CNNs are quite compatible and powerful machines that it can be used in industrial applications with ease. The result of this thesis work is promising. The system is robust enough to use it in daily tasks such as a quality check, classification of similar products. Once the products get changed in the application, the system will be capable of learning futures from new product images. Those images can be taken by camera right above the conveyor belt or over the manipulator, and in this way, the system will be fed with using new data. The operator will create a folder which consists of similar or desired product classes. After having a dataset, the system gets trained with the new dataset, and will create new weigh and bias values to learn features from the new dataset, right after that system will be able to recognise new images without training the system. This system possibly gets employed by many industrial tasks. With stereo camera over the robot or on the hand of the robot, this algorithm can be run to pick and place the object (Jiang, Wang, Cheng, Peach, & Feng, 2015; Leeper & Hsiao, 2010; Mahler et al., 2016).

There are a variety of methods which uses different techniques and does not need to have many image inputs to make feature extraction such as one-shot learning by (Vinyals, Blundell, Lillicrap, Kavukcuoglu, & Wierstra, 2016). If this can be applied in this thesis work, then needing a dataset will not be a big issue anymore. Having/creating less dataset will decrease the processing time.

The other future work can be dropping some layers of the network which is used for this work. In that way, the newly modified network will be a bit more complex. To make the network faster and more efficient, this is needed. Approximately fifty-nine million weights are in three fully connected layers, which is %96 of the network. For the last fully connected layer we decreased a thousand neurons to two neurons and had made the network quick to use it in our case. However, it still can be even faster than this and more accurate.
5.2 Summary

The aim of this thesis was to recognize objects using a deep learning algorithm. The task at hand was to be able to recognize defects occurring in wrenches. The results of this could be applied to detection of defects for any given object.

This thesis consists of five main chapters. In the first chapter, advantages and disadvantages of the traditional methods and deep learning algorithms were mentioned. ANN is studied to find the applicable solution to this thesis topic. The most applicable deep learning algorithm is searched for the defined task. In this section, CNN is chosen as a further ANN.

In the third chapter, CNN is studied and explained why it is the best DL application for our task. The theory of CNN is explained and pre-trained CNN algorithm, AlexNet explained.

In the forth chapter, decided pre-trained network is modified to our task. The layers of the of the AlexNet is explained and modified to our dataset which is created for this thesis work. To train the modified ANN with the created dataset, training options were defined. In this section, modified AlexNet is tested on the wrench dataset and the results were presented. To get better result, defined parameters were changed and results were improved.

Applicable results have been achieved. The algorithm has been designed to take any raw image in any sizes as input. The result and designed algorithm has been tested in many ways with changing parameters. The most applicable configuration of the network is saved and used to recognise novel test dataset. Based on the result and accuracy of the system it is defined as a promising result.

In the conclusion part, future work is defined. Possible areas that this topic can be used is explained and introduced literature research for the future work. In the second part of the conclusions, the summary of the thesis is explained.
BIBLIOGRAPHY


APPENDICES

A. Datasets

A.1 Decent wrench dataset

Figure A.1 Decent dataset (1–35/485)
Figure A.2 Decent dataset (36-63/485)

Figure A.3 Decent dataset (63-91/485)
Figure A.4 Decent dataset (91-119/485)

Figure A.5 Decent dataset (120-147/485)
Figure A.6 Decent dataset (148-176/485)

Figure A.7 Decent dataset (177-204/485)
Figure A.8 Decent dataset (205-232/485)

Figure A.9 Decent dataset (233-260/485)
Figure A.10 Decent dataset (261-288/485)

Figure A.11 Decent dataset (289-316/485)
Figure A.12 Decent dataset (317-344/485)

Figure A.13 Decent dataset (345-372/485)
Figure A.14 Decent dataset (373-400/485)

Figure A.15 Decent dataset (401-428/485)
Figure A.16 Decent dataset (429-456/485)

Figure A.17 Decent dataset (457-485/485)
A.2 Defective wrench dataset

Figure A.18 Defective dataset (1-36/130)

Figure A.19 Defective dataset (37-60/130)
Figure A.20 Defective dataset (61-84/130)

Figure A.21 Defective dataset (85-108/130)
Figure A.22 Defective dataset (109-130/130)
function DeepNetObjectRecognition

1. Testing the program

img1 = imread('n03218446_339.jpeg'); %Importing an image
fig1 = imshow(img1) %view image

fig1 =

Image with properties:

   CData: [300x400x3 uint8]
   CDataMapping: 'direct'

Use GET to show all properties
2. Defining a path where the dataset is

```matlab
filename = ('C:\Users\aydin\OneDrive\All About Master Thesis\MATLAB FILES AND DATASET\WrenchDataset') %The file location where datastore is
```

```matlab
filename =
'C:\Users\aydin\OneDrive\All About Master Thesis\MATLAB FILES AND DATASET\WrenchDataset'
```

3. Importing pre-trained neural network

```matlab
deepnet = alexnet %loading pre-trained network
```

```matlab
deepnet =
SeriesNetwork with properties:
Layers: [25×1 nnet.cnn.layer.Layer]
```

4. Network architecture

```matlab
layers = deepnet.Layers; %Saving layers for further use
inlayer = layers(1) %Exctracting the first layer to read features of it
inputsize = inlayer.InputSize %Extracting the input size for further use to resize the images in datastore
outlayer = layers(25) %Extracting the last layer to modify to get better performance
categorynames = outlayer.ClassNames; %Extracting the class names
weight1 = deepnet.Layers(2).Weights; %Code, in order to get weights for the second
convolutional layer

weight1 = mat2gray(weight1); % Scaling and resizing the weights for visualization
fig2 = figure
montage(weight1)
title('First convolutional layer weights')

inlayer =

ImageInputLayer with properties:

    Name: 'data'
    InputSize: [227 227 3]

Hyperparameters
DataAugmentation: 'none'
Normalization: 'zerocenter'

inputsize =

    227 227 3

outlayer =

ClassificationOutputLayer with properties:

    Name: 'output'
    ClassNames: [1000×1 cell]
    OutputSize: 1000

Hyperparameters
LossFunction: 'crossentropyex'

fig2 =

Figure (5) with properties:

    Number: 5
    Name: '
    Color: [0.9400 0.9400 0.9400]
    Position: [680 558 560 420]
    Units: 'pixels'

Use GET to show all properties
5. Creating a datastore

```matlab
%ls *.jpeg; %Command to display the images in the current folder and imports in network
wrenchdatastore = imageDatastore(filename, 'IncludeSubfolders', true); %Creating a datastore for all products
filenames = wrenchdatastore.Files; %Extracting the file names from decent product store
wrenchnames = wrenchdatastore.Labels; %Command to show labels
numberoflabels = countEachLabel(wrenchdatastore) %Counts each label which are in datastore
```

6. Separating and preparing the testing data

```matlab
%load wrenchdatastore %Command to load the datastore in Matlab environment
wrenchdatastore = imageDatastore(filename, 'IncludeSubfolders', true, 'LabelSource', 'foldernames'); %Creating a datastore for all products, and taking label names from folders
wrenchnames = wrenchdatastore.Labels; %Command to show labels

tbl = countEachLabel(wrenchdatastore)
minSetCount = min(tbl{:,2});
wrenchdatastore = splitEachLabel(wrenchdatastore, minSetCount, 'randomize');
countEachLabel(wrenchdatastore)
```

ans = 2×2 table

<table>
<thead>
<tr>
<th>Label</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
7. Pre-processing the images(inputs) for the training, validation and testing part

```matlab
wrenchdatastore.ReadFcn = @(filename)readAndPreprocessImage(filename); %Reading the images which are stored in the datastore for preprocessing

function Iout = readAndPreprocessImage(filename) %Creating a function for further application, defining variables, and resizing images as 227 by 227 for the network

I = imread(filename);

Iout = imresize(I, [227 227]);
if ismatrix(I) %Converting the greyscale image to colourful if there are
    I = cat(3, I, I, I);
end
Iout = imresize(I, [227 227]); %Saving the new images in required size

end

DecentWrenches = find(wrenchdatastore.Labels == 'DecentWrenches', 1);
DefectiveWrenches = find(wrenchdatastore.Labels == 'DefectiveWrenches', 1);

fig3 = figure
subplot(1,2,1);
imshow(readimage(wrenchdatastore, DecentWrenches))
subplot(1,2,2);
imshow(readimage(wrenchdatastore, DefectiveWrenches))
```

fig3 = Figure (6) with properties:

- **Number**: 6
- **Name**: ''
- **Color**: [0.9400 0.9400 0.9400]
- **Position**: [680 558 560 420]
- **Units**: 'pixels'

Use GET to show all properties
8. Splitting data for training and testing

```matlab
[wrenchTrain, wrenchTest] = splitEachLabel(wrenchdatastore, 0.50, 'randomized'); %Taking randomly some of the training images proportionally for testing
[wrenchTrain, wrenchValid] = splitEachLabel(wrenchdatastore, 0.30, 'randomized');
trainingLabels = wrenchTrain.Labels; %Creating a variable as training labels so that we will have ready variables for further use
testingLabels = wrenchTest.Labels; %Creating a variable as testing labels so that we will have ready variables for further use

numberOfTrainingImages = numel(wrenchTrain.Labels);
rndTrain = randperm(numberOfTrainingImages, 25)
fig4 = figure
for i = 1:25
    subplot(5, 5, i)
    trainExampleImages = readimage(wrenchTrain, rndTrain(i));
    imshow(trainExampleImages)
end

numberOfTestingImages = numel(wrenchTest.Labels);
rndTest = randperm(numberOfTestingImages, 25)
fig5 = figure
for i = 1:25
    subplot(5, 5, i)
    testExampleImages = readimage(wrenchTest, rndTest(i));
    imshow(testExampleImages)
end
```

Figure A.25 Showing random images from both files
9. Modifying and replacing some of the network layers

```matlab
fullyconnectednewlayer = fullyConnectedLayer(2) % 2 fully connected layer for 2 kinds of wrench label
layers(23) = fullyconnectednewlayer; % modifying the fully connected layer with a new one
layers(25) = classificationLayer % modifying the last layer with actual classes
```
fullyconnectednewlayer =

FullyConnectedLayer with properties:

    Name: ''

Hyperparameters
  InputSize: 'auto'
  OutputSize: 2

Learnable Parameters
  Weights: []
  Bias: []

Use properties method to see a list of all properties.

10. Setting up the training options

%pixelRange = [-30 30];
%imageAugmenter = imageDataAugmenter('RandXReflection', true, 'RandXTranslation',
  pixelRange, 'RandYTranslation', pixelRange);
%augdatastoreTrain = augmentedImageDatastore(inputSize(1:2), wrenchTrain,
  'DataAugmentation', imageAugmenter);
%augdatastoreValidation = augmentedImageDatastore(inputsize(1:2), wrenchTest);

trainingopt = trainingOptions('sgdm', ...
  'MiniBatchSize', 10, ...
  'MaxEpochs', 6, ...
  'InitialLearnRate', 1e-4, ...
  'ValidationData', wrenchValid, ...
  'ValidationFrequency', 3, ...
  'ValidationPatience', Inf, ...
  'Verbose', false, ...
  'Plots', 'training-progress');

%netTransfer = trainNetwork(wrenchTest, layers, trainingopt);

%[YPred, scores] = classify(netTransfer, wrenchTest);
featureLayer = 'fc7';
trainingFeatures = activations(deepnet, wrenchTrain, featureLayer, 'MiniBatchSize', 32,
  'OutputAs', 'columns'); %In this code minibatchsize is defined as 32 and the reason behind
  that is that CNN and image data fit into our GPUs. If the GPU is not performing well, that
  would be better to keep this number low.
%net.trainParam.show=15
%net.trainParam.epochs=1;
11. Classifier

classifier = fitcecoc(trainingFeatures, trainingLabels, 'Learners', 'Linear', 'Coding', 'onevsall', 'ObservationsIn', 'columns'); %With in this code SVM classifier will be trained with using CNN features and SVM classifier will use fast linear solver, and be set 'ObservationsIn' to 'columns' to match the arrangement used for training features.

12. Evaluating the classifier

testFeatures = activations(deepnet, wrenchTest, featureLayer, 'MiniBatchSize', 32); %Extracting test features using the CNN

13. Performing the training

[deepnet, info] = trainNetwork(wrenchTrain, layers, trainingopt);
testpredictions = classify(deepnet, wrenchTest);

Figure A.28 The accuracy of the training and testing with 6 epochs and 90 iterations.
14. Evaluating the trained network with test performance

 actuallabels = wrenchTest.Labels;  %Extracting the test labels
 numberofcorrect = nnz(testpredictions == actuallabels)  %Calculating of the correct prediction amount
 percentageofcorrect = numberofcorrect/numel(testpredictions)  %Success rate
 wrenchconf = confusionmat(wrenchTest.Labels,testpredictions)  %Defining and calculating the confusion matrix
 [wrenchconf, wrenchnames] = confusionmat(wrenchTest.Labels, testpredictions)  %Calculating the confusion matrix with class labels

 fig6 = heatmap(wrenchnames, wrenchnames, wrenchconf)  %Visual representation of confusion matrix

 numberofcorrect =
  84

 percentageofcorrect =
  0.8077

 wrenchconf =
  42  10
  10  42
15. Investigation the predictions

```matlab
fig7 = plot(info.TrainingLoss)

% [testpredictions, scores] = classify(deepnet, wrenchdatastore);
% bar(scores)
% highscores = scores>0.01;
% bar(scores(highscores))
% xticklabels(categorynames(highscores))
```

Figure A.29 Plotting the confusion matrix

Figure A.30 Plotting the training loss result

End