# Deep Learning Networks for Target Recognition


by Roan Song

prepared for A. K. Mishra

Department of Electrical Engineering

University of Cape Town


Submitted to the Department of Electrical Engineering at the University of Cape Town in partial fulfilment of the academic requirements for a Bachelor of Science degree in Electrical and Computer Engineering.

**January 2017**

**Blank page**

# Declaration

1. I know that plagiarism is wrong. Plagiarism is to use another's work and pretend that it is one's own.
2. I have used the IEEE convention for citation and referencing. Each contribution to, and quotation in, this report from the work(s) of other people has been attributed, and has been cited and referenced.
3. This report is my own work.
4. I have not allowed, and will not allow, anyone to copy my work with the intention of passing it off as their own work or part thereof.

Signature: ...........................

Roan Song

Date: ...........................

# Abstract

Identifying targets present in radar images is critical to many military and commercial ventures. Using deep learning techniques for classification of radar imagery is an avenue that can use further exploration. Deep learning classifiers will be assessed and compared to naïve classifiers. The techniques applied in this report are applied to the MSTAR dataset of radar targets, which is publically available.

# Acknowledgements

A heartfelt ”thank-you” goes out to everyone who tried their best to understand what on earth I was actually doing in this report.

# Table of Contents

# List of Figures

# List of Tables

# Listings

# Chapter 1

# Introduction

This thesis aims to assess the effectiveness of deep learning techniques in the classification of radar imagery. Deep learning relies on the use of neural networks; interconnected layers of nodes sharing information and undergoing non-linear transformations that, through training and optimisation, can detect and extract features from a dataset without human supervision. Training a classifier on a known set of instances allows it to build a predictive model that can then be tested on unseen data, with varying degrees of accuracy. In the specific case of this report, the instances are radar images from the MSTAR dataset.

## 1.1 Background

The desire to mimic human brain function has driven the development of artificial intelligence (AI) and deep learning. The human brain can be viewed as a series of interconnected neurons, firing when undergoing different stimuli. This view led to the foundation of modern deep learning techniques in the 1940s, using multiple layers of artificial neurons. Due to hardware limitations, this approach saw neither success nor widespread adoption. The 21$^{st}$ century has seen renewed interest in the field due to increased computational capabilities. While the complexity of the human brain is currently beyond accurate emulation, the deep learning techniques used to approximate it have found their uses in commercial classification problems.

## 1.2 Motivation

Deep learning is used commercially in voice and image recognition, recommendation engines, artificial intelligence, and a host of other applications. Deep learning classifiers are characterised by relatively long training times and fast classification, making them suited to real-time applications; the time taken to train a predictive model is much greater than the time taken to classify a specific instance, but it can be done beforehand on known data. Acknowledging the success of deep learning techniques has encouraged this report to test the applicability of deep learning techniques in target acquisition and classification of radar imagery.

## 1.3 Objectives

This report aims to develop two classifiers suitable for use on the MSTAR dataset:

1. K-Nearest Neighbours classifier, supporting user-selected values of K
2. Multilayer Perceptron, supporting multiple hidden layers, where each layer's size and activation function can be specified.

Each classifier will be compared according to the following performance metrics:

- Training time
- Training accuracy
- Classification time
- Classification accuracy

The steps to be taken in this report:

- Understand the format of the MSTAR dataset

- Perform pre-processing of the MSTAR dataset
- Develop the KNN classifier
- Develop the Multilayer Perceptron
- Collect performance data for each classifier
- Compare and contrast each classifier based on their performance metrics
- Establish the merit of deep learning techniques in the context of target recognition

## 1.4 Scope and Limitations

### 1.4.1 Focus

This focus of this report is to:

- Review the appropriate academic literature regarding deep learning, and assess the current body of knowledge on the subject to understand where this report will be able to contribute
- Implement KNN and Multilayer Perceptron classifiers in Python
- Implement a training and testing regime for each classifier in Python
- Determine and comment on which classifier is most suitable for the task of radar target recognition

### 1.4.2 Scope

Within project scope:

- A literature review of the appropriate knowledge pertinent to this report
- Implementation of a K-Nearest Neighbour classifier

- Implementation of a Multilayer Perceptron classifier
- Optimising the aforementioned classifiers
- Implementation of all of the code in Python
- Comparing and contrasting the performance of each classifier
- Assessment of the viability of deep learning in target recognition

Outside project scope:

- very deep neural networks, such as the convolutional neural network
- complex dimensionality reduction of input images

### 1.4.3 Limitations

All computation will be performed on a desktop computer running Windows 10 Pro with an Intel i5-2500 processor (four cores @ 3.3-3.7GHz), Samsung Evo 850 SSD, and 8GB of DDR3 RAM .

## 1.5 Report Overview

This report will begin with a review of the academic literature pertaining to the field of deep learning and radar target recognition. This will be followed by a large design section, emphasising the importance of planning before implementation. This section decomposes the problem introduced by the report into achievable requirements and specifications, and details how each step will be validated and verified. The design section encompasses the software design and implementation, with examples of code given throughout. The report continues into a section that presents preliminary and final results for discussion. It shows how changes were made between stages of implementation, to achieve the final results. The report is concluded with a brief summary of the results, and recommendations for further development.

# Chapter 2

# Literature Review

## 2.1 Synthetic Aperture Radar

### 2.1.1 Description

SAR is used to create images of objects, such as vehicles (as in this report), or landscapes. The images are constructed by sending a radar signal from a moving platform, and the time taken for the signal to return to the antenna denotes the size of the aperture. The aperture can be physical, with a large antenna, or synthetic in the case of a moving aperture. Larger apertures allow for higher image resolution. SAR images consist of magnitude and phase data, from which elevation data can be calculated [1] . The classification of 2D SAR images, the type dealt with in this report, requires only the magnitude data to be preserved.

### 2.1.2 Relevance

The dataset chosen for this report is comprised of SAR imagery. Understanding the nature of this format allows the decision to strip the data of phase information and keep only magnitude data to be made.

## 2.2 The MSTAR Dataset

### 2.2.1 Description

The MSTAR Public Mixed Targets dataset is provided by the U.S. Airforce on the Sensor Data Management System (SDMS) site [2]. The dataset contains X-band synthetic aperture radar (SAR) image chips of 8 different targets. Each image has a resolution of 1 foot, and is captured in spotlight mode.

The target in each image is centered.

The targets in each class are rotated between 0°and 360°, with images given along the entire path of rotation. This gives a comprehensive view of each target. As Figure 2.1 shows, there is a large disparity in appearance between instances of the same class.

Targets are grouped by elevation angle. For each elevation angle there are between 195 and 274 images per class. Two elevation angles, 15°and 17°were chosen, as they were the two elevation angles for which each class had images (some classes had 45°, but were ignored). Over 8 classes there is a total of 4459 images to consider.

Information pertaining to each target, including its elevation, depression angle, and target type is contained in a header section of each file. The header is followed by magnitude and phase data of the SAR imagery. The SDMS provides tools for converting the raw data into TIFF and JPEG image formats. TIFF is an uncompressed image format, suffering none of the loss that the JPEG format has. It is thus the one used in this report. Converting from the raw data to an image file reduces the complexity of this study to that of image-based target recognition. The phase data present in the original file is safely ignored. [3].

The targets vary in size from 54x54 to 192x193, although images within each class are uniform in size.

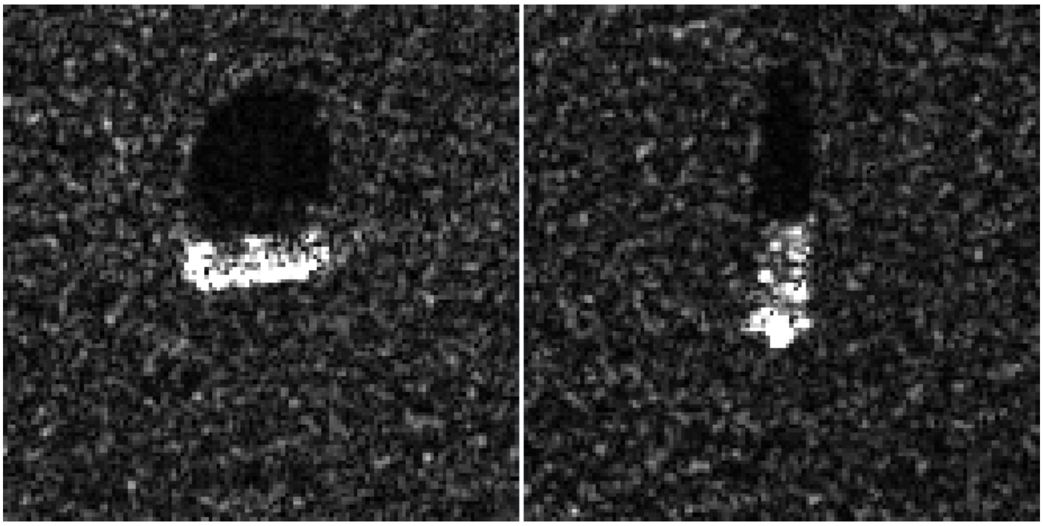

Figure 2.1: Rotational difference between two images of the same class

### 2.2.2 Relevance

The MSTAR dataset was suggested for this study by A. Mishra. It provides a generic SAR image chip dataset on which any classification method can be run. The dataset is comparatively small, with 195-274 images per class, and thus is suitable for machine learning on a consumer-grade desktop computer with reasonable classification time. The rotation factor present in each image introduces complications, but the images themselves are centered, which is very convenient.

## 2.3 Naïve Classification

### 2.3.1 Nearest Neighbour Classification

#### Description

The nearest neighbour classifier operates as follows: When given an input, the classifier compares this input to the training data set, and finds the one that is closest to the input. For example, if men and women were to be classified by their heights, a given input would be classified as either male or female based on the data point in the training data with the height closest to that of the input. This can be expanded to multiple features/dimensions by taking the Euclidean distance between the input and each instance in the training data set. For images, this amounts to comparing, pixel by pixel, each pixel value, and finding the L2 distance between them [4]. Note that there is no need to apply the square root to the distance; it is a monotonic operation, so it will not affect the ordering of the values, and will introduce additional computational complexity. The equation for calculating the L2 distance is:

$$d_2(a, b) = (a - b)^2$$

Each pixel usually has some relation to the pixels near to it, so there is the possibility for a better definition of 'distance' between images to be made [5, 6].

The nearest neighbour classifier has been shown to provide MSTAR classification rates (82-87%) through sufficient image processing and classifier development. SAR images are filled with 'clutter' surrounding the target, which was noted as affecting the success of classification [3].

#### Relevance

The nearest neighbour classifier is used in this study as an example of a naïve classifier. Results of this classifier provide a good benchmark against which subsequent classifier performance can easily be measured. The success of other parties in classifying the MSTAR targets using nearest neighbour methods lays a convincing foundation for future development. There is undoubtedly room for improvement, beginning with the elimination of clutter's effect on classification.

### 2.3.2 K-Nearest Neighbour Classification

#### Description

The K-Nearest Neighbour (KNN) classifier is an extension of the Nearest Neighbour classifier where, instead of selecting the single closest neighbour from the dataset to the input case, the 'K' nearest neighbours are selected, and the class most prevalent amongst the neighbours is taken as the predicted class. K is always an odd-valued integer, to prevent ties from occurring [4].

#### Relevance

KNN is a more robust form of Nearest Neighbour classification because larger values of K ignore outliers in the data. This regularisation effect can be optimised by choosing the best-performing value of K for a specific dataset.

## 2.4 Deep Learning

The objective of this study is to test the performance of deep learning-based classifiers on SAR image chip data. The success of naïve methods has already been proven [3], but lack the predictive power of a more sophisticated classifier. Neural networks and the application of deep learning are key to extracting features from the data to further improve classification rates.

### 2.4.1 Neural Networks

#### Description

A neural network is a system inspired by the perceived workings of the human brain; a system of neurons combine to perform tasks that exceed their individual capabilities. An input is passed through a series of neuron layers, each of which is tuned to identify characteristic features of the input between each layer, allowing for feature extraction and identification.

#### Relevance

The motivation behind using neural networks is simple; instead of specifying basic characteristics for a system to detect, the system is given an input and a matching output and is left to develop its own perceptions of what important feature link the two. Through optimisation and iteration this can become a very successful form of classification.

### 2.4.2 Multilayer Perceptron

#### Description

A multilayer perceptron is a neural network consisting of an input layer, one or more hidden layers, and an output layer. The input layer is mapped directly from an input instance; one feature per link. In the case of images, each pixel is a feature. The hidden layer neurons have non-linear activation function applied to their inputs, forming their outputs. Typically sigmoid (output range: 0 to 1) or hyperbolic tangent (output range: -1 to 1) are used. The output layer has neurons representing each class and is typically a logistic regression layer; a softmax function is applied to its outputs, making the sum of the layer's neurons' outputs equal to 1. The neuron with the highest output value denotes the predicted class [8].

Each neuron in every layer is linked to every neuron in the layer that follows it. Each neuron has a randomly chosen weight applied to its output when the classifier is initialised. These weights are subsequently optimised through back-propagation as the classifier is trained. This allows the network to develop relationships between neurons, eventually mapping an input to the output of the classifier with as little error as possible.

A single hidden layer is often sufficient for classification tasks, but more layers can be added as desired. Hidden layer 'depth' allows for more complex feature detection,a nd each layer can be made to serve a purpose as in convolutional neural networks (outside the scope of this report). The multilayer perceptron becomes difficult to optimise as the number of hidden layers grows, because the effect of each neuron on the output, and the effects of previous neurons become progressively more difficult to compute.

#### Relevance

Implementing a multilayer perceptron is the main focus of this report, as it provides an example of a deep neural network with a fairly simple implementation. Training time becomes a significant factor when using a deep neural network due to the time taken to complete back-propagation optimisation, so using a multilayer perceptron will likely force the development of more efficient methods of data pre-processing to speed up the training as much as possible.

## 2.5 Optimization and Training

A classifier is only operating efficiently when it is tuned to the data it is attempting to classify. Deep neural networks are initialised with random weights between their neurons, and at first use will perform worse on average than naïve classification methods. Through optimisation of these inter-neuron weights, however, the potential of deep neural networks can be reached, and classification results are expected to significantly improve [8]. Tuning the classifier to the dataset is crucial, but optimising too heavily may result in *overfitting* of the data, leaving the classifier with no predictive power on unseen data.

### 2.5.1 Theano

#### Description

Theano is a Python module geared towards machine learning applications. It can be used to create generic functions acting on 'TensorVariables' that act as placeholders for future parameters. C code of these functions is dynamically generated and compiled, resulting in much faster computation times than the Python interpreter can achieve. An additional benefit to using Theano is that it computes computational graphs for each compiled function, allowing easy calculation of derivatives with respect to parameters involved in its computation [9].

#### Relevance

The calculation of derivatives is essential for implementing the back-propagation algorithm used to train a Multilayer Perceptron.

### 2.5.2 Back-propagation

#### Description

Back-propagation is a system by which the effects of weights between neurons is adjusted through an iterative process. The base case is that of a single input, single output system. Varying the weight on the input directly effects the output. This change can be easily recognised, and the weight can be changed to more suitably link the input to the desired output. This involves developing a method of changing weights in a sensible manner. The most common form of this is through *gradient descent*, covered in Section 2.5.3, whereby the weights are adjusted corresponding to their perceived effect on the output state, and their rate of change. Back-propagation is not guaranteed to find a global minimum, and can settle on a local minimum instead, which can be somewhat alleviated through the use of random weights and multiple training rounds, before choosing the best version of the classifier that has been discovered.

One of the key issues with back-propagation is its computational complexity. With deep neural networks, the sheer number of weights and their possible combinations make discerning their impact on the output very difficult, and computationally infeasible to perfectly optimise.

#### Relevance

Back-propagation is a popular and successful technique, well-suited to neural networks with only a few layers. With enough time, it can help to optimise much larger networks, and potentially improve classification accuracy by a large margin. It alleviates the concern of trying to find the perfect network from the offset; it allows any network to be tuned to be better than it currently is.

### 2.5.3 Gradient Descent

#### Description

If the computational graph of a classifier's cost can be calculated, each parameter that contributes to the cost to be adjusted according to its contribution, with the aim of reducing the overall cost. The degree to which each parameter is adjusted is scaled by the *learning rate*. High learning rates lead to rapid change of parameter values, which can result in faster convergence to the optimal values, but can also end up in oscillation around these values if the adjustment is too large. Smaller learning rates can take longer to converge more safely, but can also result in local optimums being converged to, instead of the best possible set of values.

Given a weight $w$,a learning rate $a$, and the gradient of the cost with respect to the weight, $\Delta w$:

$$w = w - a * \Delta w$$

Given a large enough dataset, it may be infeasible to calculate the impact of every weight of every instance on the output, and so the gradient is approximated by taking a batch of instances and averaging the gradient of each, approximating a 'global gradient'. This is known as *stochastic gradient descent* [10].

#### Relevance

Stochastic gradient descent is used in this report in the Multilayer Perceptron to optimise the inter-neuron weight values.

### 2.5.4 Hyper-parameters

#### Description

Hyper-parameters are parameters that, when changed, modify the structure or operation of the neural network, without changing its core mechanics. Hyper-parameters under consideration in this project are:

- K in the KNN (**??**)
- Input Image size (100px vs 1000px)
- Learning Rate
- Network Shape (hidden layer size, and number of layers)
- L2 regularisation on inter-neuron weights
- Training, Validation and Testing ratios

#### Relevance

Optimising hyper-parameters is incredibly important when optimising a classifier. Sub-optimal hyper-parameters such as a learning rate that is too high could prevent an otherwise functional classifier from converging onto the optimal choice of inter-neuron weights and its classification accuracy could suffer.

### 2.5.5 Training, Validation, and Testing

Successful classification of a dataset is divided into three distinct steps:

1. Training

2. Model Validation
3. Testing

Training is the process of fitting a classifier - it involves running multiple iterations on a given set of inputs, comparing the output of the classifier to a known target dataset, and adjusting the parameters of the classifier (typically inter-layer weights and bias) through back-propagation. To find the best iteration of the classifier model, it is periodically tested on a different set of known data; the validation dataset. Testing on this intermediate dataset is used to provide performance metrics such as the mean error and the accuracy of classification, which is useful in selecting a model that is optimised to the desired set of parameters. The data from periodically testing on this validation set is used to tune the model, or implement early-stopping procedures. For example, if the desired level of classification accuracy has been achieved or if the mean error hasn't changed significantly after a number of epochs, the training can be stopped early. On the contrary, if the training is approaching its stated limit yet still improving classification accuracy, the number of epochs or 'patience' can be increased to allow for further iterations and tuning.

Once the training is complete, having achieved the desired level of classification accuracy, the chosen model can be tested on another dataset (typically a set of real-world instances) to see how it performs, providing the testing accuracy. The classifier is no longer tuned, and can be presented with a variety of inputs to simulate its real-world performance.

A dataset is typically split into three sections. 50% training, 25% validation, and 25% testing is a reasonable starting point, and the proportions can be seen as a hyper-parameter to be optimised. If the training set is too small, there is a risk that the model will not be able to successfully extract the features required for classification, and its testing accuracy will be low. If the validation or test sets are too small, the model validation and testing might not be representative of the classifier's performance on a larger test set.

### 2.5.6 The Confusion Matrix

Confusion matrices are useful tools for evaluating classifier performance. They can show the accuracy, misclassification rate, true/false positive rates, specificity, precision, and prevalence. They clearly show the number of correctly classified classes along the matrix's diagonal, and also shows how many of the incorrect classifications were attributed to which classes. The numbers can be changed to show the percent values of each; i.e. the classification accuracy of each class and how what percent of misclassifications went to each class.

An example of a confusion matrix is shown in Table IV

# Chapter 3

# Design

## 3.1 Design Context

This study is within the scope of an undergraduate level approach to both radar and machine learning, with an emphasis on machine learning; the techniques applied herewith are not limited to radar imagery, but attempts should be made to tailor the design to target recognition applications. With development this study should be adaptable to commercial use, and be helpful to people in the radar department who need assistance with radar target classification. Thus the study should develop an easily extensible framework for radar image classification, or at least guidelines to allow others to integrate with the work covered herein.

## 3.2 Feasibility Study / Concept Exploration

Does the classification of radar imagery lend itself to deep learning techniques, and if so, will the performance be better or worse than naïve classification methods? This is the question that best captures the analysis of the study's feasibility.

Consideration of this question needs to include the format of the data entering the system, the ability of the system to process such data, and

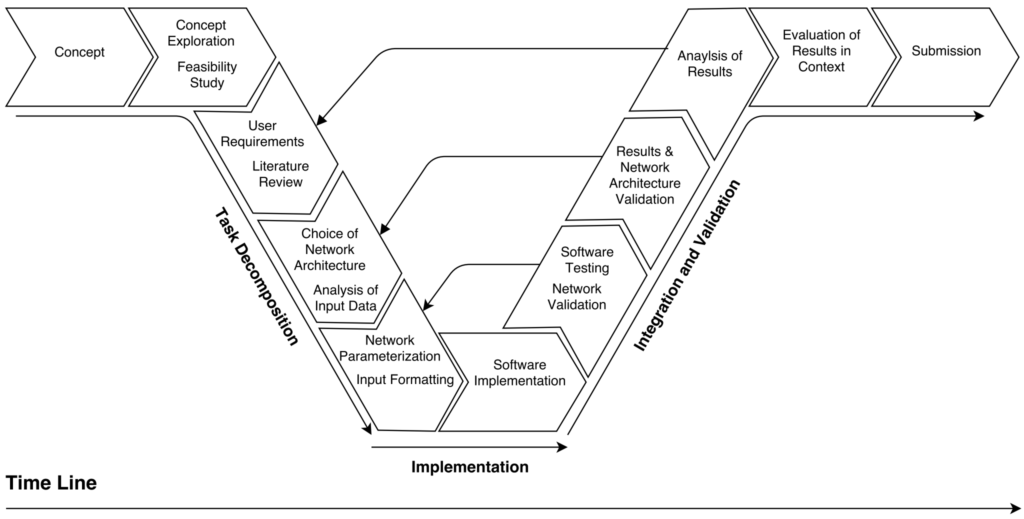


Figure 3.1: Vee Diagram

the effectiveness of the classification of the data.

The data consists of images of between 2916 (54x54) and 37054 (192x193) pixels in size. Each image contains one target, positioned at the centre of the image. This should allow for effective resizing of images to account for size discrepancies. Each pixel in an image constitutes an input. The computational cost of processing the largest image in the set versus the smallest will be at least 12.7 (37054/2916) times more expensive. Initial thoughts were that processing of such images will be made much more feasible if they can be reduced in size to match the smallest images present in the dataset, or at least be made as small as possible while retaining all of the information needed to classify each target. However, the shadow of each radar target often extends outside the 54x54 pixel range, yet could be necessary in classification. Preserving features inherent to each image is more important if the classifier can handle the largest images in a timely fashion.

Neural networks have a fixed structure; a collection of input neurons feeding their values through a series of hidden neuron layers before arriving at an output layer of neurons equal in size to the number of classes present in the data. The architecture of the network - the choice of number of hidden layers, the number of neurons in each hidden layer, and the number of neurons in the input layer are all subject to change during the development of the system. The training and operation of the system occurs through the adjustment of inter-neuron weights, with the structure of the system remaining constant.

The choice of size of the input layer is crucial; it must remain constant throughout the training and operation of the network. Since each pixel in an image forms of the input layer's neurons, all of the input images must be processed to contain the same number of pixels before any other work on the network's architecture can begin.

Once the pre-processing of the input images is complete, the structure of the neural network can be decided. This structure shall be changed and prototyped in order to try to find a good corresponding fit for the data. Having too many hidden layers will greatly increase the time taken by the back-propagation algorithm to optimise the weights of the system, and the likelihood of it settling at a local instead of global minimum increases with the complexity of the system.

The number of neurons in each hidden layer must also be chosen carefully; too many neurons in each layer will result in much longer optimization time (proportional to the increase in the number of inter-neuron weights created). Having too few neurons in a layer can result in the system being unable to extract the features key to classification, and too many neurons may lead to 'overfitting' of the training data, leaving the system with no predictive capability (an inability to classify data not present in the set of training instances).

## 3.3 Decomposition and Definition

This section is devoted to describing the study in terms of its requirements, operation, and implementation. An accompaniment explaining the verification and validity of each subsection will be in the next section.

### 3.3.1 Concept of Operations

Radar target classification is an inexact science; interpreting a radar image and comparing it to a known case is not as straightforward in all cases as one might expect. Weather conditions, environmental clutter, and image resolution all obscure the target to varying degrees, making intuitive classification ineffective. Computer-based classification through analysis of multiple targets and the application of deep learning techniques should in theory allow distorted images to be classified after the computer is trained to recognise features pertaining to each class. Naïve methods of classification lack predictive power - the ability to 'guess' effectively if the target is obscured or unrecognised. Deep learning methods are the solution that this study proposes.

### 3.3.2 User Requirements

The success of this report is based on the ability to correctly classify and recognise radar targets taken from the supplied dataset using deep learning techniques.

The following is required:

- Use the MSTAR dataset (Section 2.2)
- Develop a naïve classifier to use as a benchmark
- Use deep learning methodology to develop a classifier
- Indicate how the classifiers can be improved
- Comment on the performance of each classifier
- Report on the suitability of deep learning for target recognition

### 3.3.3 Design Specifications

Expanding upon the user requirements, the following design specifications have been derived:

**The system must be trained on the MSTAR database of radar images**

The images in the MSTAR dataset have undergone a level of pre-processing, making them suitable for rapid prototyping and development of classifiers.

**The system must have a testing accuracy of above 95%**

Correct identification of radar targets is the aim of this entire report, and as such is the most important performance metric to consider.

**The system must be resistant to noise**

Real-world conditions are not optimal; the classifier must be robust enough to handle the addition of Gaussian white noise to the dataset, without losing more than 1% classification accuracy.

**The system must have a training time of less than 10 hours**

Training time is taken as being freely available, but any classifier that takes longer than 10 hours to train on the MSTAR dataset will not be worth considering.

**The Nearest Neighbour classifier should be used as a benchmark for classifier comparison**

The NN classifier provides a good example of a naïve classifier, and should be beaten by any classifier that is somewhat optimised for the dataset in use. The NN classifier serves as a good benchmark to improve upon, as it gives a lower bound of expected results

**At least two different classifiers should be tested against the Nearest Neighbour classifier**

The chosen classifiers are the K-Nearest Neighbour classifier optimised to fit the dataset, and the Multilayer Perceptron, which satisfies the need for a deep learning classifier.

**Each classifier must be evaluated and compared**

The performance metrics to be used are training time, classification time, and classification accuracy.

### 3.3.4 High-Level Design

After analysis of the user requirements, the following areas of design need to be focused on:

- Identification/classification
- Image Processing/Preparation
- Dimensionality Reduction
- Naïve Classification (Nearest Neighbour as a benchmark)
- Deep Learning Classification (Multilayer Perceptron)
- Obtaining classifier performance metrics
- Classifier comparison
- Classifier optimisation

### 3.3.5 Detailed Design

#### Image Processing/Preparation

The MSTAR dataset is a compilation of image chips, all of which contain a header, as well as magnitude and phase data. The images are between 54x54 and 192x193 pixels in size, which suggests that some form of image processing should be performed to make sure that all images are the same size. The targets in each image chip are centred, suggesting that cropping each image to a size where the target (and its shadow - useful in classification) are left whole, and as much of the surrounding clutter as possible is removed.

An alternate approach is to retain the data inherent in the environmental clutter and pad the smaller images with zeros, keeping all images in the set at the size of the largest image in the set. While this preserves all of the image chip data, processing larger images leads to longer training and classification times.

A compromise is to ensure that all images are the size of the largest image in the set, and somehow reduce the clutter present in each image. Because the targets are substantially brighter than their surroundings, using some form of thresholding (setting values lower than a specified threshold to zero) should prove to be effective in lessening the impact of the clutter, if not completely removing it.

#### Dimensionality Reduction

Each pixel in an image is taken as an feature, forming a feature vector with a length equal to the total number of pixels in the image. The image cropping mentioned in Section 3.3.5 is very effective at reducing the size of this feature vector. If a 128x128 image is cropped to 64x64, the feature vector's length is reduced by factor of 4. This can be reduced further through the application of dimensionality reduction techniques, such as Principal Component Analysis, Locally-Linear Embedding, Sum of Means and non-linear methods, all of which are outside the scope of this report. As such, each image will be rescaled and padded with zeros to match the size of the largest image in the MSTAR dataset (192x193). This allows every possible feature in each image to be used, while allowing for dimensionality reduction to be implemented at a later stage if desired.

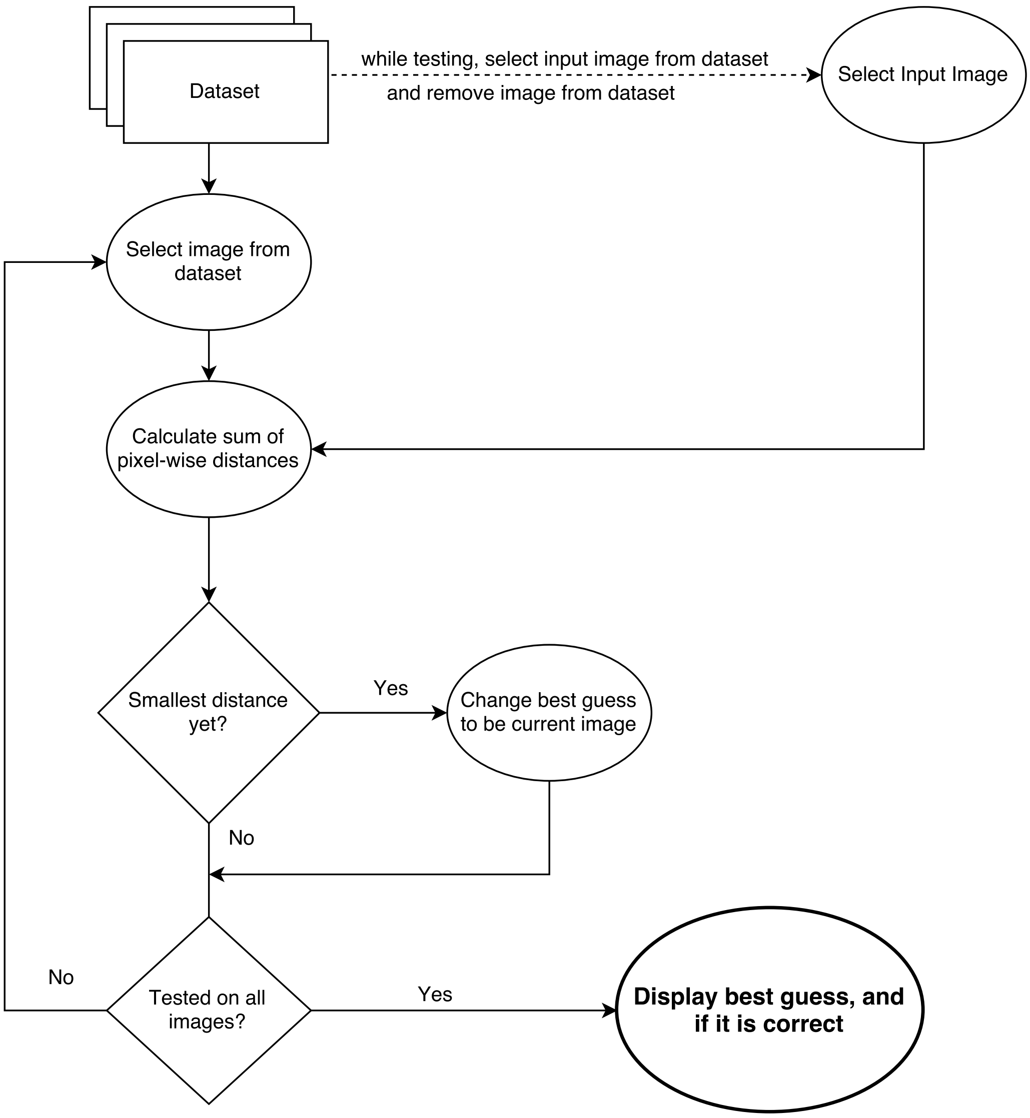


Figure 3.2: Nearest Neighbour Classification Flow Diagram

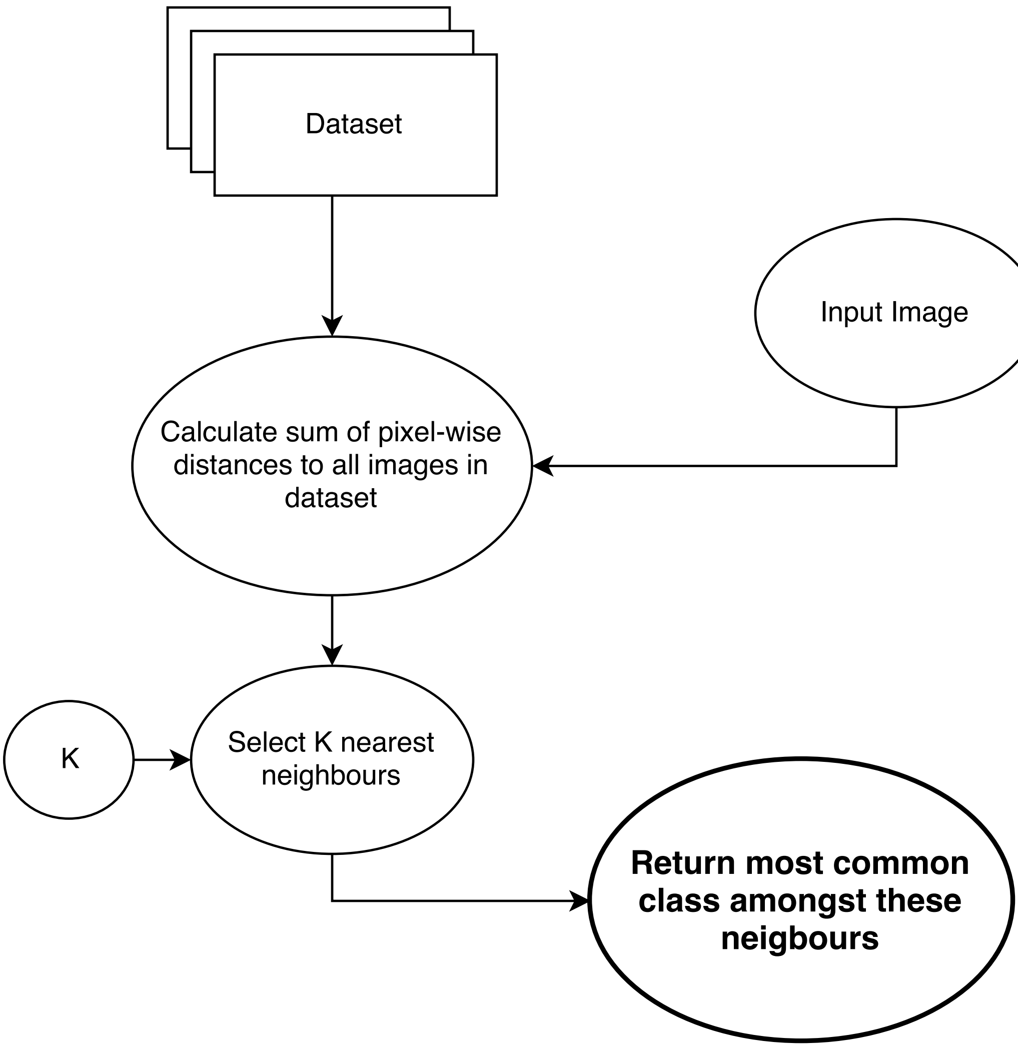


Figure 3.3: K-Nearest Neighbour Classification Flow Diagram

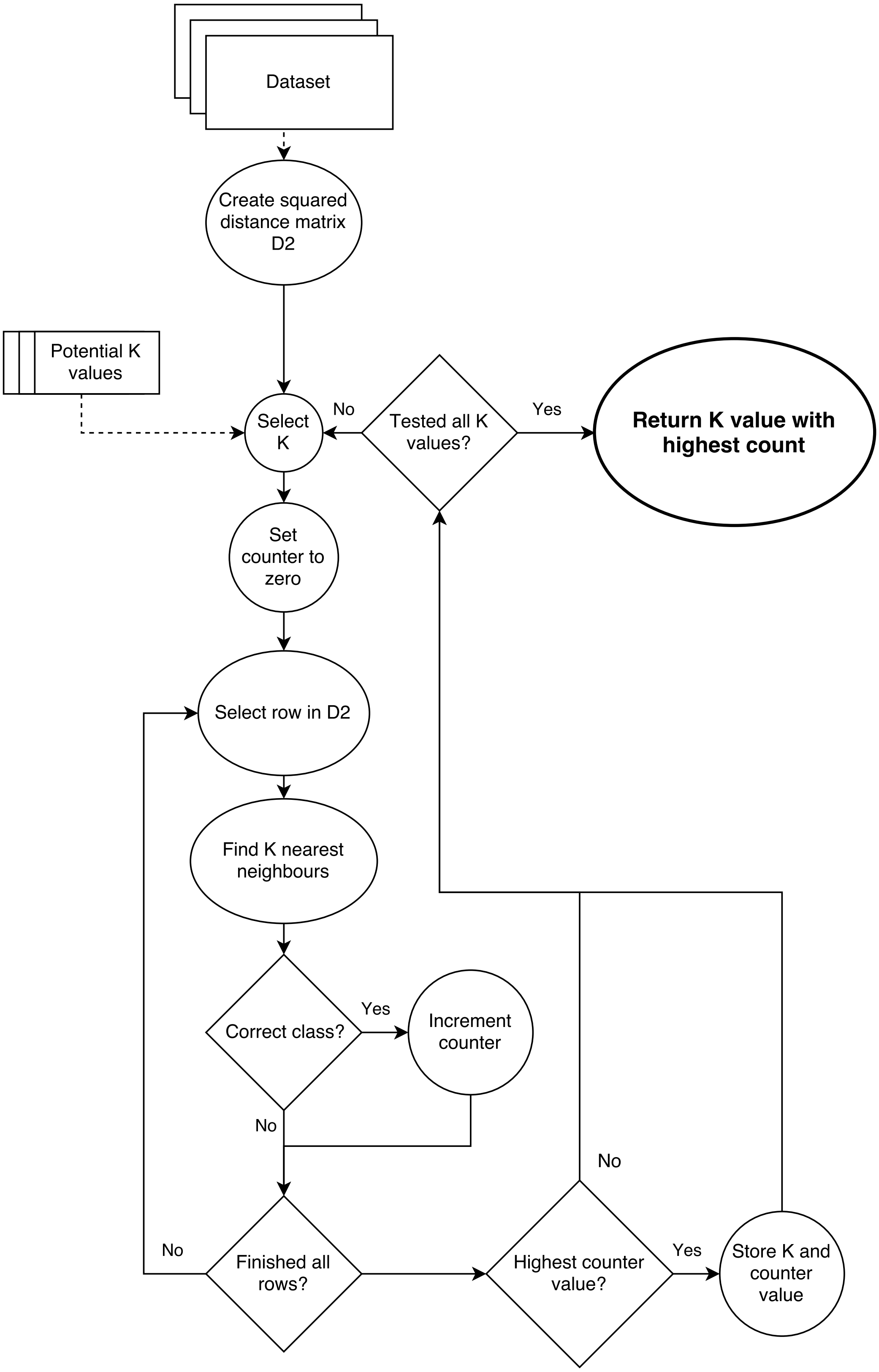




Figure 3.4: K-Nearest Neighbour Optimising for 'K' Flow Diagram

### Nearest Neighbour Design

The high-level design is shown in Figure 3.2. Its principles of operation are covered in Section 2.3.1. To implement this classifier, the following is needed:

- Access to the dataset
- A choice of input image
- A method to calculate and sum the pixel-wise distances
- A variable storing the smallest distance and tentative classification
- A method displaying the chosen class and whether or not it is correct

### K-Nearest Neighbour Design

While similar to the Nearest Neigbour design, KNN introduces its own complexities, most notably when optimising for K. The high-level design is shown in Figure 3.3, its principles of operation are covered in Section 2.3.2. The method of finding the best K value for a given dataset is shown in Figure 3.4.

### Multilayer Perceptron Design

Implementation of a multilayer perceptron in software can be divided into discrete sections as follows:

- Implement a wrapper class for the classifier
- Develop a logistic regression layer class to use at the output
- Develop an extensible hidden layer class
- Collate the important details of each layer in the wrapper class
- Implement a test method

## 3.4 Software Design

### 3.4.1 Guidelines

Developing the software for this report is done with the following objectives in mind:

- Comment code clearly
- Use logical code structure and layout
- Use Python as the chosen programming language
- Use Theano module for deep learning calculations
- Code self-contained classifier classes
- Difficult to generate data should be saved for future use
- Code reusable, adaptable methods
- Make performance metrics readily available
- Test while developing
- Provide a comprehensive 'README' for new users
- The code must be available online at github.com/roansong.

### 3.4.2 Testing and Optimisation

#### Testing

Each classifier has unique testing methods, but the performance metrics on which there are compared are the same. The metrics we are interested in are:

- Time taken to classify a single instance
- Time taken to train the classifier
- Classification accuracy of the classifier

#### Optimisation

Where possible, each classifier must be optimised to improve its classification accuracy and classification time. Training time is considered acceptable if it is less than ten hours, as specified in Section 3.3.2.

The K-Nearest Neighbour classifier must be optimised to find the value of K that best fits the dataset. The Multilayer Perceptron must be optimised for its learning rate, L1 regularisation, L2 regularisation, hidden layer size, and number of hidden layers.

#### Inline Testing

While implementing the software for this report, I ran into some early issues that resulted from insufficient planning. After taking a step back, it was decided that more steady progress would be made by implementing rigorous inline testing, i.e. incrementally testing the code after every slight modification, instead of only testing after the addition of a major feature and then trying to iron out any latent bugs. This approach leads to much simpler debugging.

## 3.5 Software Implementation

The implementation of the classifiers and techniques discussed in this report will be covered in this section.

#### Image Pre-processing

In a process outlined in Figure 3.8, the input image is converted to an array of unsigned 8-bit integers (ranging from 0 to 255). The elements of the array are 'normalised' by subtracting the mean of the array from each and then dividing each element by the standard deviation of the array. This ensures that the processed values lie centred around zero, and mostly between -1 and

1. Unsigned integers do not provide enough precision to represent this data, so 32-bit floating point numbers (floats) are used. The code for normalisation is shown in Listing 3.1.

```
def normalise(vector):
"""
Normalises a vector so that most of its values lie between -1 and 1
returns the vector mentioned above
vector --- the vector to be normalised (type: numpy array)
"""
return (vector - vector.mean(axis=0))/(vector.std(axis=0))
```

Listing 3.1: Normalisation method

Two additional methods of processing are made available: thresholding and noise addition. Thresholding is done before normalisation. For each image, pixels with values below the median value of the image are set to zero. This eliminates some of the noisy ‘clutter’ present in radar imagery. Noise addition is used to simulate real-world conditions by adding Gaussian noise to the image after normalisation to somewhat obscure the radar signal. The implementations of thresholding and noise addition are shown in Listings 3.2 and 3.3 respectively. The complete code is shown in Appendix A.1

```
if(threshold):
below_thresh = image < numpy.mean(image)
image[below_thresh] = 0
```

Listing 3.2: Thresholding

```
if(noise):
image += numpy.random.normal(0,1,image.shape)
```

Listing 3.3: Adding noise to an image

### Allocation of Data

Three different sets of data are needed to train a classifier. The training set, validation set, and testing set. Each set contains instances of data (in this case image vectors) and targets denoting the class each instance belongs to. Each set is filled with random images from the original dataset without replacement, with the size of the three sets being determined by a set of three numbers. The numbers do not have to correspond exactly to the number of images in each set - they represent ratios between the sizes. The code to generate the three sets, as well as a tuple of indices showing which instances from the original dataset are in each set is shown in Appendix A.2.

**K-Nearest Neighbours**

Developing a standalone Nearest Neighbour classifier and a separate K-Nearest Neighbour classifier is redundant; the NN can be obtained by setting K equal to 1 in the KNN. Only KNN needs to be implemented.

KNN operates by loading a dataset of input instances and targets, and calculating the squared distances between the input instance and every instance in the dataset. Each distance result is appended to a list. After every distance has been calculated, the list is sorted in ascending order, and the K lowest instances are taken. It is important to keep track of the target output associated with each input. These instances with the smallest squared distances are the 'neighbours' from which KNN derives its name. The most common class amongst the neighbours is taken as the predicted class. The ratio between the number of neighbours in agreement and K can be used to give a measure of confidence in the prediction.

To optimize K for the dataset, a different approach must be taken. It is inefficient to test every instance in the dataset against every other instance for every possible value of K. Instead, these distances are calculated only once. A squared distance array (D2) is formed by taking every instance in turn and calculating its distance to every other instance, with the knowledge that the distance from an instance A to instance B will be the same as from B to A. This halves the number of calculations necessary to calculate D2. D2 is symmetric along its diagonal, and all of its diagonals are equal to zero. This is because the distance from any instance to itself will always be zero.

Each row in D2 provides the squared distances between the instance corresponding to that row and every other instance in the data set. Sorting these distances in ascending order for every row gives the nearest neighbours to each instance. Removing the first instance in each sorted row removes the self-contribution factor of an instance to itself. Chosen values of K can be tested much more efficiently, going row by row and taking the first K instances, seeing if the majority of those K neighbours predicts the correct output, and tallying up a score for each value of K. The value of K with the highest score once every row has been visited is the value of K best optimised for the dataset. The complete KNN code is shown in Appendix B.1.

### The Multilayer Perceptron

Once each pixel has been processed, they can be sent to the input layer of the neural network, with each pixel representing a neuron as shown in Figure 3.8.

Each of these input layer neurons has a random weight applied, and is then fed into the neurons that form the first hidden layer. Every neuron in the input layer contributes to every neuron in the next layer. At each neuron in the first hidden layer, the 'net' value is formed by summing all of the values present at its input (i.e. all the weighted values passed along from the input layer). An activation function is applied to this net value to compress the range of values. There are many different activation functions that can be used, and for this study I narrowed down the choice to either the logistic function $\frac{1}{1+e^{-net}}$ or the hyperbolic tangent function $\frac{e^{net}-e^{-net}}{e^{net}+e^{-net}}$. The logistic function produces an output between 0 and 1, while the hyperbolic tangent function's output is between -1 and 1. I decided to use the hyperbolic tangent function, because the logistic function has a tendency to incur long training times if its values lie very close to 0, while the hyperbolic tangent function tends to move towards its extreme values more quickly.

The activation function, when applied to the net of the neuron, becomes the output of the neuron, feeding through to every neuron in the next layer with weights applied, repeating the same process until the output layer. This is shown in Figure 3.5 and more closely in Figure 3.6.

The output layer has as many neurons as there are classes. The output layer is a logistic regression layer, which has a softmax function applied to its outputs. This results in outputs that sum to 1, with larger values being

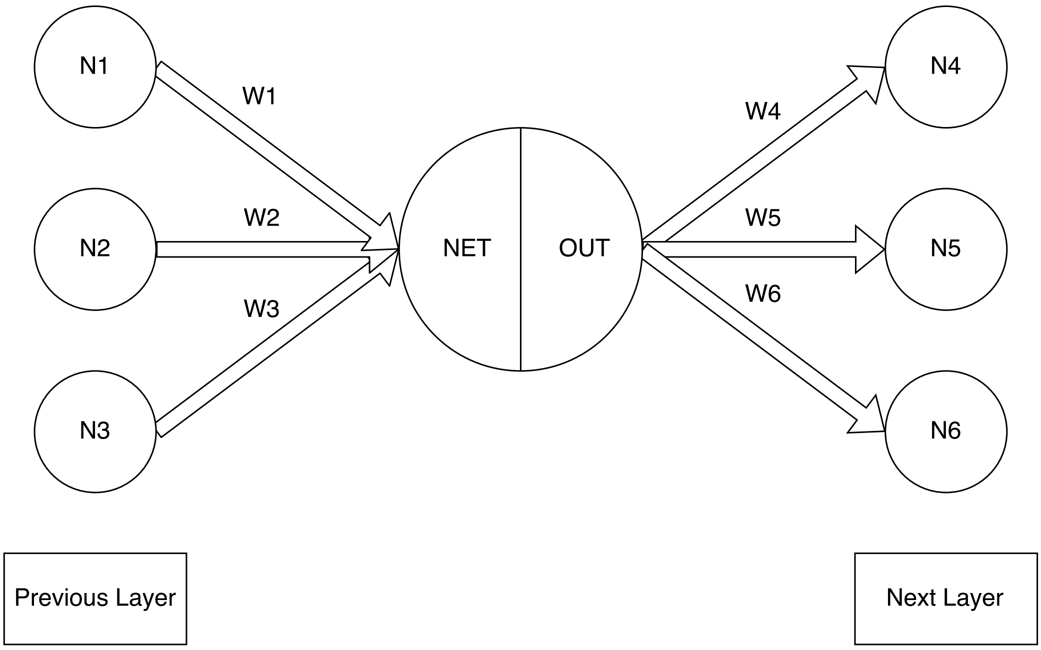


Figure 3.5: A Neuron in the Network

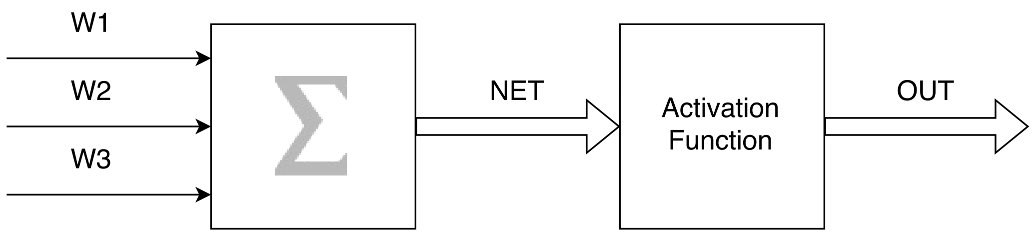


Figure 3.6: The Internal Workings of a Neuron

strongly emphasised. This correlates nicely with the one-hot vectors used to label classes [11]. The neuron with the highest value is taken to be the network's prediction. An example output would be:

$$[0.25, 0.2, 0.1, 0.05, 0.4]$$

There are five classes, and the fifth class has the highest value, thus the neural network has classified the input as belonging to that fifth class. Given that the maximum possible value for an output would be 1, 0.9 would show a large confidence in the classification. A value close to 0 shows strong disagreement, while 0.3 would show moderately low confidence.

To train the network, stochastic gradient descent is used, as described in Section 2.5.3. The gradient of the cost function with respect to the weights and biases of each layer is calculated and the weights and biases are updated proportionally to the learning rate. This is shown in Listing 3.4, and is easily achieved through the use of Theano (covered in Section 2.5.1).

```
gradients = [T.grad(cost,param) for param in self.parameters]
updates = [(param, param - learning_rate*gparam) for param, gparam in zip(
    parameters,gradients)]
```

Listing 3.4: Implementing Gradient Descent

### Training and Testing

The first step is to get to grips with the data - processing the data set in a way that makes sense to use, and allows different classification methods to be implemented easily on it.

The MSTAR dataset contains eight different targets. The dataset is sorted by depression angle, and by 'scene'. All of the targets have data corresponding to 15° and 17° elevation angles, for a total of 4459 images. The MSTAR dataset stores the information of each target in a header section of each file. This is inconvenient when reading in image files directly, so the header is discarded and images are classed according to their file extension. The file extensions and the classes they correspond to can be found in Table II. To simplify classification, a variant of a one-hot vector denoting the classes is attached to each target. The vector consists of a series of numbers, equal in length to the number of classes in the dataset. A '1' denotes that the instance is a member of the class corresponding to that entry in the vector, and the rest of the numbers are 0, showing that the instance is not in those classes. A file is created listing all of the filenames to be tested during the run of the algorithm. An example file with ten entries and two classes would look as follows:

HB03333.003.tiff 1 0
HB03334.003.tiff 1 0
HB03335.003.tiff 1 0
HB03337.003.tiff 1 0
HB03338.003.tiff 1 0
HB14931.025.tiff 0 1
HB14932.025.tiff 0 1
HB14933.025.tiff 0 1
HB14934.025.tiff 0 1
HB14935.025.tiff 0 1

This information is used to form the target set that corresponds to every set of instances.

### 3.5.1 K-Nearest Neighbours

The KNN is provided with the training, validation and testing datasets. Because KNN undergoes no validation phase, the validation set is concatenated to the training set to provide extra training data. The best value of K is obtained as detailed in Section 3.5. This value of K is then used to run every instance in the test set against all the images in the training set. This procedure is timed, and represents the training time metric of the classifier. The number of correct predictions is tallied and is used to calculate the classification accuracy. The time taken to classify a single instance against the dataset is taken as the classification time.

### 3.5.2 Multilayer Perceptron

Training a Multilayer Perceptron is more complex than training the KNN. Training the MLP revolves around minimising the loss function of the classifier, defined by the negative log likelihood $loss$ of the predicted output to the target output and the L1 and L2 norms of the network's weights, adjusted by the L1 and L2 regularisation factors $a$ and $b$. These factors sum to form the cost of the classifier, as shown below:

$$cost = loss + a * L1 + b * L2$$

This cost is minimised through training. It is important to note that the regularisation of the inter-neuron weights is thought to improve the classification accuracy, with smaller weights providing more general predictive capabilities. Minimising the cost function entails balancing loss and weight minimisation to prevent over-fitting of the training dataset.

Using the Theano module, the symbolic gradient of the cost with respect to each parameter that contributes to it is calculated, as shown in Listing 3.4.

Before training can commence, the parameters of the training session need to be chosen. The number of epochs specifies how many iterations over the dataset are to be done before returning. The batch size determines the number of batches and how many instances are in each batch when implementing stochastic gradient descent. Early stopping parameters are chosen, such as the option to stop training once testing error is zero, and the 'patience' of the training. The patience defines the number of instances to train on without improvement before stopping early. If the results obtained within this patience interval demonstrate improvement, the patience interval is extended. Improvement is measured by seeing if a new value falls under a threshold deemed significant (taken as 99.9% as default in this implementation).

During training, the error of the classifier is averaged over each training batch and the parameters of the system are adjusted in proportion to the learning rate. After training on every training batch once, the classifier is trained on the validation batches, averaging the errors returned for each batch. Note that the error is not the same as the cost - error is the zero-one loss between the predicted output and the target output, and is shown in Listing 3.5.

```
import theano.Tensor as T
def errors(self,target):
    """
    Returns the average error between the predicted class and the target
        class
    target --- correct output
```

```
    T.neq is the logical inequality function
    """
    return T.mean(T.neq(self.predicted_class,target))
```

Listing 3.5: Calculating Zero-one Error

If the improvement seen in the validation batches is significant, the patience value is increased. During validation the model parameters are also updated via stochastic gradient descent.

After validation, the test error or 'test score' is calculated by testing the model on data not in the training or validation sets. The error is the average of the zero-one errors over all test cases. If the test score has improved, the state of the model is saved by storing the inter-neuron weights on disk. If previously specified, if the training error is equal to zero, the training can stop, otherwise the classifier continues to train. There is a benefit to continuing in that the classifier cost can be reduced further, at the risk of over-fitting.

Once training is complete, the performance metrics of the classifier are stored. The training procedure is timed, a single instance is tested and timed to find the single-class classification time, and the classification accuracy of the classifier on the test set is noted.

Other metrics specific to the MLP are also stored; the shape of the classifier (how many hidden layers and their sizes), the best test score, final validation error, and final training error, as well as the final cost of the classifier.

### Testing During Development

The simplest way to find a classifier's efficacy is to test it on a wide variety of classes and on as many test instances as possible. To provide interim results, during the iterative phase of classifier development, only a subset of the dataset's images are used. This compromises the final accuracy of the classifier (it may perform differently on the full dataset), but brings with it the ability to test and train classifiers more quickly, due to the lower computational overheads. During the development this testing method allows for simple decisions regarding the direction of classifier implementation

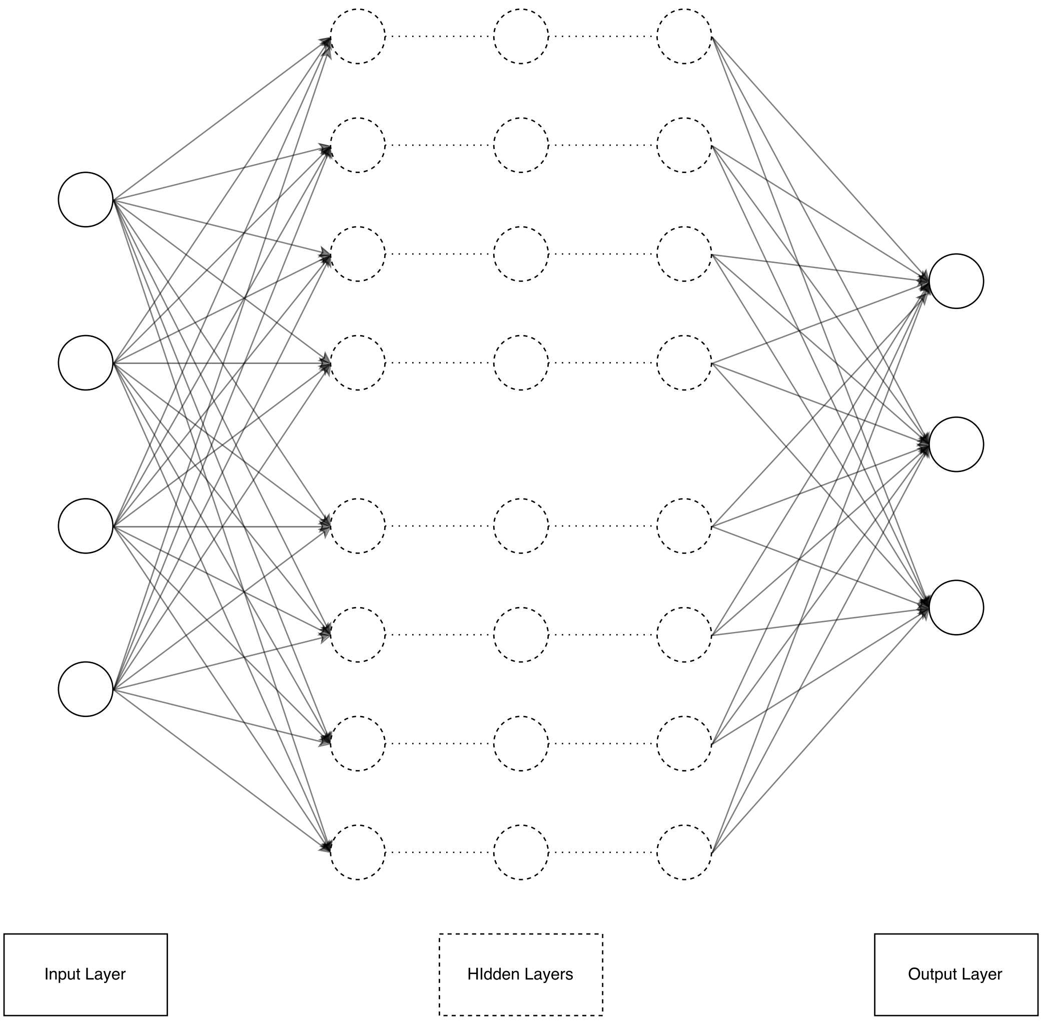


Figure 3.7: Multilayer Perceptron Overview

or optimisation to be made. In process of verifying the classifiers, the full dataset must be used to give an accurate picture of the classifier's performance and real-world implementation.

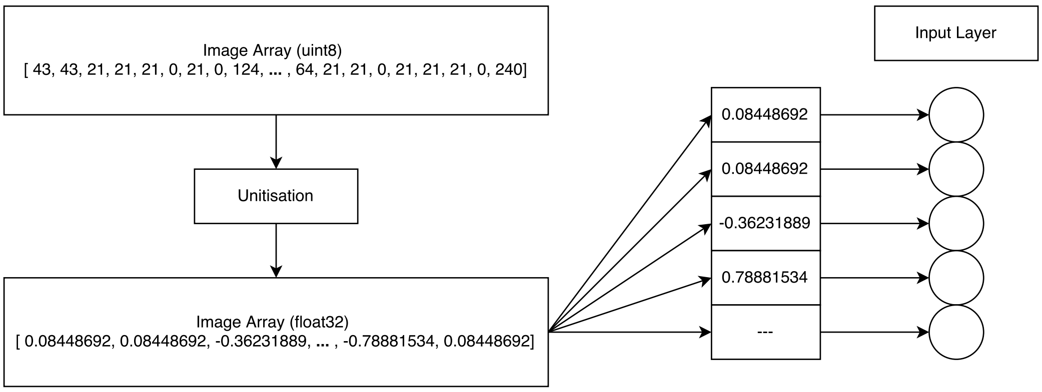


Figure 3.8: Multilayer Perceptron Input Layer

## 3.6 Integration and Recomposition

### 3.6.1 Subsystem Verification

The chosen method for verifying a classifier's efficacy once it is considered to be sufficiently optimised is simple; The classifier is tested on the dataset with a number of test instances as in Section 2.5.5. For the KNN, fully testing it entails removing one instance from the dataset, training the classifier on the remaining points, and using the removed instance as a test case. Once this has been done, the instance is replaced, another is taken, and the process is repeated until every instance in the dataset has been tested. The system's performance is based on the number of correct classifications made during the process. This is known as Leave-One-Out Cross Validation (LOOCV).

### 3.6.2 System Verification and Deployment

To confirm the results of each classifier, it is important to have a set of training instances on which the system can be trained. The system is then tested on another set of points whose classes are known. Once this has been tested and confirmed to have a desirable level of classification accuracy, the system will be ready for testing on previously unseen, real-world cases.

This is accomplished by dividing a set of known data points into a training set, a validation set, and a testing set. For example: 80% of the data points will be used to train the system, 10% to validate the model, which will then be tested on the remaining 10%. The process of cross-validation entails selecting a different training/test split each time (either systematically or at random) and performing the process again. This concept can be extended to where the system is trained on all but one of the instances and then tested against it, which is known as LOOCV ("Leave One Out" Cross-Validation). The system is tweaked until it reaches the level of classification required. Cross-Validation is an important tool for eliminating "overfitting" of the system to the training data. Mixing up the training and test cases ensures that the classifier is left with some ability to generalise, and not just repeat what it has been shown.

### 3.6.3 System Validation

If the objectives of the report have been accomplished, the system will be considered valid. The results found during this report have to be collated and analysed within the context of its requirements. This goes beyond confirming the individual results of each classifier and seeing if implementing deep learning techniques are indeed the 'smart choice' in the task of radar target recognition.

### 3.6.4 Operations and Maintenance

Considering the real-world application of a target recognition system, it should meet certain criteria to ensure ease of use and compatibility with various datasets. Providing adequate documentation to support the system is key.

The documentation must detail:

- The operating procedure of the system
- The expected input to the system (dataset and individual instance)
- The output format of the system (predicted class and performance metrics)

- Comprehensive troubleshooting (outside the scope of this report)

### 3.6.5 Changes and Upgrades

If the system is required to be maintained and expand its scope (by incorporating more, larger, and more complex images into its dataset), the input data will have to undergo pre-processing that is not currently implemented into the system, such as intelligent dimensionality reduction. The system may have to restructure some of its key classification methods and re-optimise hyper-parameters.

A classification method to consider for real-world use is a Convolutional Neural Network, as it can classify targets without the need for the target to be centered (as in the MSTAR dataset). This would be more suited to real-world applications, although the classification time is longer. This is not trivial, and lies outside the scope of this report.

# Chapter 4

# Results

This section of the report shows the results of the tests performed on the chosen classifiers

## 4.1 Preliminary Results

The preliminary results were calculated on a subset of the MSTAR dataset; 1291 images spanning 5 classes were selected. Each image was cropped/padded to 100x100 pixels in size and then normalised, forming an input of 10,000 data points. No further processing was performed on the images.

The data was divided as follows: 85% training, 5% validation, 10% testing. Instances are drawn randomly from the dataset without replacement. This is based on the idea that more training data results in a stronger classifier. The validation stage is for verifying the progress of the model and does not need too much data dedicated to it. Testing requires more data than validation - enough to confidently represent the contents of the entire dataset.

This resulted in a training set of 1097 images, a validation set of 64 images, and a test set of 129 images.

### 4.1.1 Nearest Neighbour

The Nearest Neighbour outperformed preliminary expectations, achieving 92.857% classification accuracy on a test set of 322 instances. Each instance was compared to a dataset of 967 instances. The classification time for a single instance is 12.38 seconds.

### 4.1.2 K-Nearest Neighbours

Before optimising for K it was expected that a K value greater than K = 1 would provide the best results. After optimising, however, it was determined that K = 1 is indeed the optimal value for K, so the results were the same as the Nearest Neighbour classifier. Generating the squared distance matrix and calculating the optimal value of K constitutes the training time of KNN, which took 1.3 hours when testing for K up to 239 on a set of 967 100x100 images. The results of this optimisation are shown in Figure 4.1

### 4.1.3 Multilayer Perceptron

The multilayer perceptron has the largest number of parameters that can be optimised, leading to a varied spread of results. The first parameters to optimise are the learning rate and the number of neurons in the hidden layer(s).

The learning rate controls how strongly the output error affects the shift in weight values every iteration. A smaller learning rate will typically cause the network to take longer to converge to its optimal weights, or get stuck in a local minimum, while a high learning rate can prevent the values from converging, resulting in oscillating values and no optimum being found.

The size of each hidden layer defines the number of connections between neurons. Too few connections can 'bottleneck' the classifier, leaving it unable to extract any meaningful features. Too many connections can allow the classifier to extract more features, at the cost of a longer training time spent optimising the larger number of weights. It is predicted that too

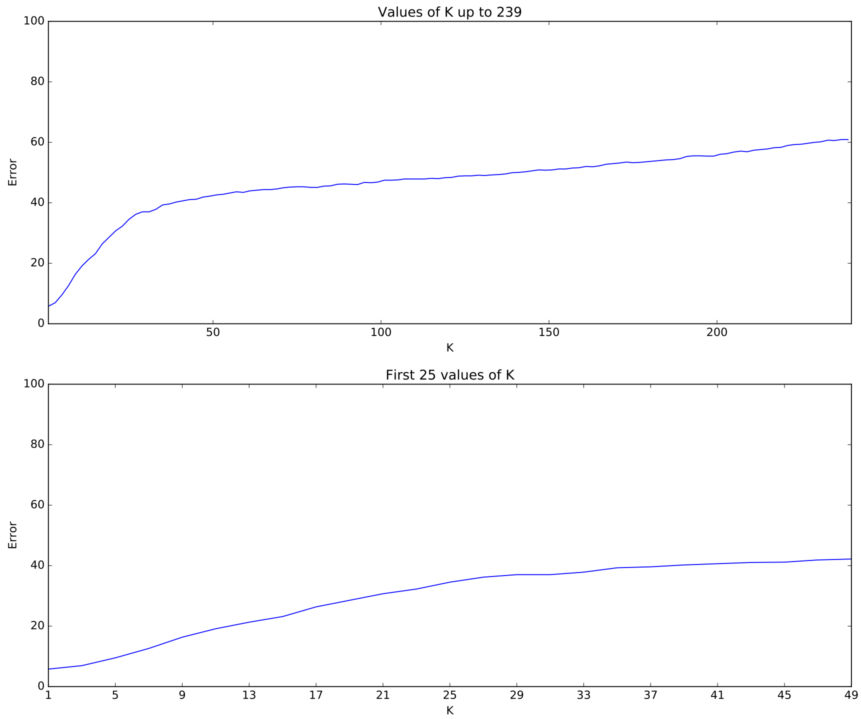


Figure 4.1: Optimising for K

many connections can also lead to over-fitting of the classifier to the training dataset.

Preliminary results show that learning rate must be set in indirect proportion to the number of hidden neurons to maintain training and classification accuracy.

#### Summary of Results

The preliminary results obtained from testing on a reduced dataset are presented in Table I. To assist with readability, important parameters are notated in brackets, for example MLP(10,5) denotes a Multilayer Perceptron with two hidden layers; 10 neurons in the first layer and 5 in the second. The training time for the KNN is optional, and should only be considered if an optimal K is found for the dataset.

The confusion matrix of the best-performing classifier is shown in Table IV, demonstrating that the classification accuracy is already high after the preliminary results

## 4.2 Preliminary Comments

Analysis of the preliminary results gives direction to the development of each classifier. It can immediately be seen that the KNN classifier is outclassed by even a roughly optimised Multilayer Perceptron.

### 4.2.1 K-Nearest Neighbours

The K-Nearest Neighbour classifier results are intended to function only as a low benchmark and point of reference, so its design remains unchanged. Comparing it to the Multilayer Perceptron shows that a Nearest Neighbour approach is infeasible for large datasets and large instances of data. No further testing needs to be performed to establish this.

| **Classifier** | **Training Time[s]** | **Classification Accuracy[%]** | **Single-Instance T** |
|---|---|---|---|
| KNN(k=1) | 0 | 92.86 | 12.38s |
| KNN(k=3) | 0 | 91.53 | 12.38s |
| KNN(k=1,trained) | 99.85 | 92.86 | 12.38s |
| MLP(10) | 11.22 | 97.00 | 0.1ms |
| MLP(20) | 42.97 | 96.33 | 0.074ms |
| MLP(5,1) | 29.07 | 82.33 | 0.012ms |
| MLP(5,5) | 7.58 | 98.33 | 0.087ms |
| MLP(10,5) | 28.35 | 97.67 | 0.091ms |
| MLP(10,10) | 12.71 | 97.67 | 0.072ms |
| MLP(5,5,5) | 8.32 | 92.00 | 0.064ms |
| MLP(10,10,10) | 20.77 | 91.67 | 0.068ms |

Table I: Preliminary Results

### 4.2.2 Multilayer Perceptron

The required hidden layer size is much smaller than anticipated; a single hidden layer with 5 neurons performs adequately. Optimisation of the weights according to their L2 norm proved crucial in improving predictive accuracy while reducing the cost of the classifier. A crude grid search was used to find the best L2 regularisation parameter, which was chosen as 0.1. The training time of each MLP is not directly proportional to the number of neurons. Single-class classification time for all MLP models is suitable for real-time application. Some of the classifiers have met and exceeded the design specifications in Section 3.3.3.

### 4.2.3 Data

The division of the dataset between training, validation and testing should remain at 85% training, 5% validation, 10% testing. Preliminary results have given no evidence that suggests a change in the allocation of data.

## 4.3 Final Results

### 4.3.1 Data Processing

The input images are rescaled to match at least the size of the largest images in dataset (192x193), and so are resized to 194x194. Thresholding is used to reduce the clutter present in each radar image. The threshold chosen is the median of each image. The full dataset is used, totalling 4459 images, 8 classes, with the training:validation:testing ratio at 85:5:10. This results in a training set of 3790 images, a validation set of 222 images, and a test set of 445 images.

### 4.3.2 K-Nearest Neighbours

As stated in Section 4.2.1, no further testing of the KNN classifier needs to be done to prove that it is inferior to the Multilayer Perceptron. However, to provide additional context, it was decided that the single-instance classification time should be measured on the full dataset. It takes 193.82 seconds to run a single instance against every instance in the training dataset. Definitely infeasible.

### 4.3.3 Multilayer Perceptron

The MLP performs exceedingly well on a larger dataset and with larger input images. Classification accuracy of 100% is achieved, meeting and exceeding the design specifications in Section 3.3.3.

### 4.3.4 Summary of Results

The final results of the report are shown in Table III. To assist with readability, important parameters are notated in brackets, for example MLP(10,5) denotes a Multilayer Perceptron with two hidden layers; 10

neurons in the first layer and 5 in the second. The confusion matrix corresponding to the best classifier is shown in Table V.

## 4.4 Final Comments

### 4.4.1 K-Nearest Neighbours

As stated in Section 4.2.1, the KNN classifier is not suitable for the task at hand. It scales too poorly with the size of the dataset, and does not provide a classification high enough to justify the long classification times that are introduced.

### 4.4.2 Multilayer Perceptron

The required hidden layer size is much smaller than anticipated; a single hidden layer with 5 neurons is enough to achieve 100% classification. The introduction of more neurons and hidden layers does not lead to improved accuracy, although one or two hidden layers of 5 to 10 neurons each appears to be the 'sweet spot'. Introducing too many layers or neurons can prevent the weights from converging to local minima during training, leading to the low classification accuracy in MLP(5,5,5) and MLP(10,10,10). They remained entirely untrained.

### 4.4.3 Data

The division of the dataset between training, validation and testing should remain at 85% training, 5% validation, 10% testing. Preliminary results have given no evidence that suggests a change in the allocation of data.

### 4.4.4 Classifier Robustness

The best-performing classifier was trained again on data with additive Gaussian noise. The classification accuracy was reduced from 100% to 99.52%. This is within the design specifications in Section 3.3.3.

### 4.4.5 Future Improvements

Moving forward, implementing more complex neural networks that can deal with non-centred images is critical. Convolutional neural networks are outside the scope of this report, but deal well with off-centre images.

| **Class** | **No. Images** | **Suffix** |
|---|---|---|
| 2S1 | 573 | 000 |
| BRDM | 572 | 001 |
| BTR 60 | 451 | 003 |
| D7 | 573 | 005 |
| SLICY | 572 | 015 |
| T62 | 572 | 016 |
| ZIL131 | 573 | 025 |
| ZSU_23_4 | 573 | 026 |

Table II: Input Data

| **Classifier** | **Training Time[s]** | **Classification Accuracy[%]** | **Single-Instance Time** |
|---|---|---|---|
| KNN(k=1) | — | — | 193.82s |
| MLP(5) | 16.47 | 100.00 | 0.174ms |
| MLP(10) | 23.24 | 100.00 | 0.156ms |
| MLP(20) | 45.22 | 96.33 | 0.156ms |
| MLP(5,1) | 749.58 | 92.14 | 0.978ms |
| MLP(10,5) | 66.19 | 99.76 | 0.185ms |
| MLP(10,10) | 66.64 | 99.76 | 0.206ms |
| MLP(5,2,1) | 745.72 | 92.14 | 0.175ms |
| MLP(5,5,5) | 7.93 | 11.00 | 0.127ms |
| MLP(10,10,10) | 11.60 | 11.00 | 0.226ms |

Table III: Preliminary Results

| | | Predicted Class | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 2S1 | BRDM | BTR 60 | D7 | SLICY | T62 | ZIL131 | ZSU_23_4 |
| Actual Class | 2S1 | 99.56 | 0.44 | 0 | 0 | 0 | 0 | 0 | 0 |
| | BRDM | 0 | 100 | 0 | 0 | 0 | 0 | 0 | 0 |
| | BTR 60 | 0 | 0 | 100 | 0 | 0 | 0 | 0 | 0 |
| | D7 | 0 | 0 | 0 | 100 | 0 | 0 | 0 | 0 |
| | SLICY | 0 | 0 | 0 | 0 | 100 | 0 | 0 | 0 |
| | T62 | 0 | 0 | 0 | 0 | 0 | 100 | 0 | 0 |
| | ZIL131 | 0 | 0 | 0 | 0 | 0 | 0 | 100 | 0 |
| | ZSU_23_4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 100 |

Table IV: Preliminary Confusion Matrix

| | | Predicted Class | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 2S1 | BRDM | BTR 60 | D7 | SLICY | T62 | ZIL131 | ZSU_23_4 |
| Actual Class | 2S1 | 100 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | BRDM | 0 | 100 | 0 | 0 | 0 | 0 | 0 | 0 |
| | BTR 60 | 0 | 0 | 100 | 0 | 0 | 0 | 0 | 0 |
| | D7 | 0 | 0 | 0 | 100 | 0 | 0 | 0 | 0 |
| | SLICY | 0 | 0 | 0 | 0 | 100 | 0 | 0 | 0 |
| | T62 | 0 | 0 | 0 | 0 | 0 | 100 | 0 | 0 |
| | ZIL131 | 0 | 0 | 0 | 0 | 0 | 0 | 100 | 0 |
| | ZSU_23_4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 100 |

Table V: Final Confusion Matrix

# Chapter 5

# Conclusions

The objective of this report was to assess the effectiveness of deep learning techniques in classifying radar images from the MSTAR dataset, with the goal of designing a classifier that can identify radar targets in real-time. This goal was achieved, and all design specifications were met. The Multilayer Perceptron classifier was able to achieve 100% classification accuracy against the MSTAR dataset, with access to 4459 images allocated between training and testing.

Even given the success of the Multilayer Perceptron, further development is always possible, through the collection of more data points and the implementation of more complex classifiers. The next logical step would be to implement a Convolutional Neural Network, which deals well with images that are not perfectly centred, as they are in the MSTAR datset.

Acknowledging the success of the Multilayer Perceptron in classifying MSTAR radar targets, I can heartily recommend that this classifier be considered in future target recognition activities.

# References

[1] J. C. Curlander and R. N. McDonough, *Synthetic aperture radar.* John Wiley & Sons New York, NY, USA, 1991.

[2] U. A. Force, "The mstar dataset," 2015. [Online]. Available: https://www.sdms.afrl.af.mil/index.php?collection=mstar

[3] R. Schumacher and K. Rosenbach, "Atr of battlefield targets by sar classification results using the public mstar dataset compared with a dataset by qinetiq, uk summary."

[4] J. M. Keller, M. R. Gray, and J. A. Givens, "A fuzzy k-nearest neighbor algorithm," *IEEE transactions on systems, man, and cybernetics*, no. 4, pp. 580–585, 1985.

[5] L. Wang, Y. Zhang, and J. Feng, "On the euclidean distance of images," Ph.D. dissertation, School of Electronics Engineering and Computer Science, Peking University, 2005. [Online]. Available: http://www.cis.pku.edu.cn/faculty/vision/wangliwei/pdf/IMED.pdf

[6] D. Michie, D. J. Spiegelhalter, and C. Taylor, "Machine learning, neural and statistical classification," 1994.

[7] I. Russel, "Definition of a neural network," 1996. [Online]. Available: http://uhaweb.hartford.edu/compsci/neural-networks-definition.html

[8] D. W. Ruck, S. K. Rogers, M. Kabrisky, M. E. Oxley, and B. W. Suter, "The multilayer perceptron as an approximation to a bayes optimal discriminant function," *IEEE Transactions on Neural Networks*, vol. 1, no. 4, pp. 296–298, 1990.

[9] T. D. Team, "Theano at a glance," 2016. [Online]. Available: http://deeplearning.net/software/theano/introduction.html

[10] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*. Springer, 2010, pp. 177–186.

[11] R. A. Dunne and N. A. Campbell, "On the pairing of the softmax activation and cross-entropy penalty functions and the derivation of the softmax activation function," in *Proc. 8th Aust. Conf. on the Neural Networks, Melbourne, 181*, vol. 185, 1997.

# Appendix A

# Image Loading and Processing

```
def get_images(w,h,file_list=None,num_classes=8,threshold=False,noise=False
    ):
  """
  Load images from a file, crop/pad them to a specific size, with
  some pre-processing options. returns an array of image vectors,
  an array of target vectors, and a dictionary linking the
  suffixes of each file to their respective target vector and the
  count of each class within the dataset.

  w       --- desired width of the images
  h       --- desire height of the images
  file_list  --- if specified, list of files to read image data
      from, otherwise uses default (default: None)
  num_classes --- number of classes into which images can be
      classified
  threshold  --- if True, set values under the median of each
          image to zero (default: False)
  noise    --- if True, add Gaussian noise to each image
          (default: False)
  """
  infile = str(num_classes) +'.txt'
  folder = 'tiffs'+str(num_classes)+'/'
  abspath = 'C:/Users/Roan Song/Desktop/thesis/'
  rng = np.random.RandomState(0)

  if(not file_list):
    dt = np.dtype([('filename','|S16'),
```

```
    ('labels',np.int32,(num_classes,))])
    infile = 'filenames8.txt'
    filedata = np.loadtxt(infile,dtype=dt)
    file_list = [a.decode('UTF-8')
    for a in filedata['filename']]
    file_list.sort(key=lambda x:x[-7:])

  suffixes = OrderedDict()

  for f in file_list:
    suffixes[f[-7:]] = suffixes.get(f[-7:], 0) + 1

  ind = 0
  for i in suffixes:
    suffixes[i] = {"count":suffixes[i],
    "label":one_hot(ind,num_classes)}
    ind += 1

  img_arr = np.zeros((len(file_list),h*w))
  target_arr = np.zeros((len(file_list),num_classes))
  i = 0
  for fname in file_list:
    img = mpimg.imread(abspath + folder + fname)
    IN_HEIGHT = img.shape[0]
    IN_WIDTH = img.shape[1]

    img = pad_img(img,h,w,IN_HEIGHT,IN_WIDTH)
    image = img.reshape(h * w)

    if(threshold):
      below_thresh = image < np.mean(image)
      image[below_thresh] = 0

    image = normalise(image)

    if(noise):
      image += rng.normal(0,1,image.shape)

    img_arr[i] = image
    target_arr[i] = suffixes[fname[-7:]]["label"]
    i+=1

  return img_arr,target_arr,suffixes
```

Listing A.1: Loading and Processing Images

```python
def gen_sets(data,targets,train,val,test):
  """
  Generate training, validation and test subsets from a given
  dataset. Returns the three sets and the indices of the
  original dataset which correspond to them

  data  --- full dataset to be split
  targets --- targets component of the dataset
  train  --- proportion allocated to the training set
  val   --- proportion allocated to the validation set
  test  --- proportion allocated to the test set

  Note: train, val and test do not have to sum to 1.
  The unit function is applied to them, ensuring that they sum to 1.
  """

  train,val,test = unit([train,val,test])

  training_set  = np.zeros((int(len(data)*train),2))
  validation_set = np.zeros((int(len(data)*val ),2))
  test_set    = np.zeros((int(len(data)*test ),2))
  rng = np.random.RandomState(0)
  indices = np.arange(len(data))

  temp = rng.choice(indices,size=len(training_set),replace=False)
  training_indices = temp
  training_set = (np.vstack(data[temp]),np.vstack(targets[temp]))
  # training_set = (data[temp],targets[temp])
  indices = np.delete(indices,temp)

  temp = rng.choice(indices,size=len(validation_set),replace=False)
  validation_indices = temp
  validation_set = (np.vstack(data[temp]),np.vstack(targets[temp]))
  indices = np.delete(indices,temp)

  temp = rng.choice(indices,size=len(test_set),replace=False)
  testing_indices = temp
  test_set = (np.vstack(data[temp]),np.vstack(targets[temp]))
  indices = np.delete(indices,temp)

  return training_set, validation_set, test_set,
  (training_indices,validation_indices,testing_indices)
```

Listing A.2: Generating sets

# Appendix B

# K-Nearest Neighbours

```python
class KNN():
  """
  A K-Nearest Neighbours classifier
  """
  def __init__(self,input,targets):
    """
    Initialisation method
    input   --- the dataset to be compared to
    targets --- the correct classes corresponding to each instance in the
        dataset
    """
    self.data = input
    self.targets = targets

  def initD2(self,filename=None,size=None,indices=None):
    """
    Method to initialise the squared distance array of the classifer
    This array stores the distances between every instance and every other
        instance

    filename --- a file from which the squared distance array can be
        imported (default: None)
    size    --- a size to which the squared distance array is to be cropped
        (default: None)
    indices --- indices of the dataset to be considered when creating the
        squared distance array (default: None)
    """
```

```
    if(filename==None):
      D2 = np.zeros((len(self.data),len(self.data)))
      for i in range(len(self.data)):
        for l in range(i,len(self.data)):
          cost = 0
          if(i != l):
            for j in range(len(self.data[i])):
              cost += pow(self.data[i][j] - self.data[l][j],2)
          D2[i][l] = D2[l][i] = cost

        u.progress_bar(i,len(self.data))
    else:
      D2 = np.load(filename)
      if(indices != None):

        temp = np.zeros((len(indices),len(indices)))
        for y in range(len(indices)):
          for x in range(len(indices)):
            temp[y,x] = D2[indices[y],indices[x]]

        D2 = temp

      elif(size):
        D2 = D2[:size,:size]

    self.D2 = D2

  def test(self,k_arr):
    """
    A method to test different values of K on the dataset
    returns an array of the results

    k_arr --- an array of K values to be tested
    """
    results = []
    correct = np.zeros((len(k_arr)))

    for img in range(len(self.data)):
      costs = sorted(list(zip(self.D2[img],self.targets.argmax(axis=1))))
      pred_lst = np.zeros((len(k_arr)))
      confidence = np.zeros((len(k_arr)))
      accuracy = np.zeros((len(k_arr)))
      ind = 0
```

```
      for k in k_arr:
        pred = list(zip(*costs[:k]))[1][1:]
        predicted_class = 0
        max = 0
        for i in pred:
          cnt = 0
          for l in pred:
            if(i == l):
              cnt += 1
          if(cnt > max):
            max = cnt
            predicted_class = i

        if(predicted_class == self.targets[img].argmax()):
          correct[ind] += 1
        confidence[ind] += max/k * 100
        accuracy[ind] = correct[ind]/len(self.data) * 100
        ind +=1


    for x in range(ind):
      results.append([k_arr[x],correct[x],accuracy[x],confidence[x]])

    self.results = results
    self.pred = list(zip(*costs[:k]))[1][1:]
    self.costs= list(zip(*costs[:k]))[0][1:]

    return np.array(results)

  def run(self,x,k,y=None):
    """
    A method to test a single instance against the dataset
    returns the predicted class, whether or not it is correct,
    the correct class, and a measure of confidence in the prediction

    x --- the input instance
    k --- the value of k determining how many neighbours to consider
    y --- the correct output if it is known (default: None)
    """
    temp = []

    for img in range(len(self.data)):
      if(np.equal(self.data[img],x).all()):
```

```
          continue
        cost2 = 0
        for px in range(len(self.data[img])):
          cost2 += pow(self.data[img][px] - x[px],2)
        temp.append((cost2,self.targets[img].argmax()))
      temp = sorted(temp)
      cost = list(zip(*temp[:k]))[0]
      pred = list(zip(*temp[:k]))[1]
      max = 0
      predicted_class = []
      for i in pred:
        cnt = 0
        for l in pred:
          if(i == l):
          cnt += 1
        if(cnt > max):
          max = cnt
          predicted_class = i
      confidence = max/k * 100
      if(y):

        return predicted_class, (y.argmax() == predicted_class), y.argmax(),
            confidence
      else:
        return predicted_class, confidence
```

Listing B.1: K Nearest Neighbours

# Appendix C

# Multilayer Perceptron

```
class Multilayer_Perceptron():
def __init__(self,input,shape,num_classes,rng):
  """
  A multilayer perceptron class

  input    --- a vector containing the input values
  shape    --- a tuple describing the shape of the classifier and its
      hidden layers
  each element in the tuple specifies the number of neurons per layer
  num_classes --- the number of classes in the dataset
  rng     --- seeded random number generator
  """

  self.hidden_layers = []
  self.hidden_layers.append(
  HiddenLayer(input=input,n_inputs=shape[0],n_outputs=shape[1],activation=
      None,rng=rng))
  for i in range(2,len(shape)):
    self.hidden_layers.append(
  HiddenLayer(input=self.hidden_layers[-1].output,n_inputs=shape[i-1],
      n_outputs=shape[i],activation=None,rng=rng)

  )

  self.output_layer = OutputLayer(self.hidden_layers[-1].output,shape[-1],
      num_classes)
  self.L1 = abs(self.output_layer.weights).sum()
```

```
self.L2 = (self.output_layer.weights**2).sum()
self.parameters = self.output_layer.parameters
for a in self.hidden_layers:
  self.L1 += abs(a.weights).sum()
  self.L2 += (a.weights**2).sum()
  self.parameters += a.parameters



self.neg_log_likelihood = self.output_layer.neg_log_likelihood



self.input = input
self.errors = self.output_layer.errors
self.predicted_class = self.output_layer.predicted_class
self.weights = [a.weights for a in self.hidden_layers]
self.weights.append(self.output_layer.weights)
self.shape = shape
self.rng = rng
```

Listing C.1: Multilayer Perceptron

```
class HiddenLayer():
  """
  This class represents a hidden layer of neurons
  It takes an array of inputs, applies an activation function to them, and returns the output
  """
  def __init__(self,input,n_inputs,n_outputs,weights=None,bias=None, activation=T.tanh,rng=np.random.RandomState(2)):
    """
    Initialise the hidden layer

    input    --- a vector containing the input values
    n_inputs  --- number of neurons feeding into the hidden layer
    n_outputs --- number of neurons in the next layer
    weights   --- weights applied to the inputs and outputs of the hidden layer (default: None)
    bias     --- bias applied to the output values (default: None)
    activation --- activation function to be applied to neuron inputs (default: tanh)
    rng     --- seeded random number generator (default: np.random.RandomState(2))
    """

    self.input = input
    if(not weights):
    weights = theano.shared(value=rng.uniform(-1,1,(n_inputs,n_outputs)), name = 'weights')
    if(not bias):
    bias = theano.shared(value=np.zeros((n_outputs,)),name='bias')

    self.weights = weights
    self.bias = bias
    output = T.dot(input,self.weights) + self.bias
    self.output = output if activation == None else activation(output)
    self.parameters = [self.weights,self.bias]
```

Listing C.2: Hidden Layer

```python
class OutputLayer():
  """
  This class is a logistic regression layer for use at the output of a neural network
  """
  def __init__(self,input,n_inputs,n_outputs):
    """
    Initialise the output layer

    input   --- a vector containing the input values
    n_inputs --- number of neurons feeding into the layer
    n_outputs --- number of classes
    """

    self.weights = theano.shared(value=np.zeros((n_inputs,n_outputs)),name='weights')
    self.bias = theano.shared(value=np.zeros((n_outputs,)),name='bias')
    self.output = T.nnet.nnet.softmax(T.dot(input,self.weights)+self.bias)
    self.predicted_class = T.argmax(self.output,axis=1)
    self.parameters = [self.weights,self.bias]
    self.input = input

  def neg_log_likelihood(self,target):
    """
    Returns the negative log likelihood between the classifier's output and a target

    target --- correct output
    """
    return -T.mean(T.log(self.output)[T.arange(target.shape[0]),target])

  def errors(self,target):
    """
    Returns the average error between the predicted class and the target class

    target --- correct output
    """
    return T.mean(T.neq(self.predicted_class,target))
```

Listing C.3: Output Layer