UNIVERSITY OF MANCHESTER

Character Recognition Using Neural Networks

A third year project report submitted to the University of Manchester for the degree of
BSc (Hons) Computer Science with Business and Management with IE
in the School of Computer Science

Author: Erkki Lukk

Supervisor: Dr. Richard Neville

April 2013

Contents

List of Figures

List of Tables

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Abstract

Character recognition, a subdomain of pattern recognition, has been a subject of research since the early 20th century. The motivation for developing modern optical character recognition (OCR) systems originated from the challenge whereby manual processing of paper documentation by major corporations became a significant bottleneck in the organisations' efficiency. Therefore, a solution for automating this process through computerisation was needed. One way of computerising character recognition is to utilise an artificial neural network (ANN) for building the classification models.

The aim of this project was to research the domains of character recognition, artificial neural networks, associated areas of software engineering and human computer interaction in order to build a software package that could recognise handwritten character samples of the English alphabet. The Agile software development methodology was adopted after domain specific research was undertaken, and it provided the framework for developing five prototypes of increasing complexity – from a simple perceptron to a multilayer backpropagation ANN. Extensive ANN training and testing was conducted on a dataset of handwritten English character samples to optimise the ANN model parameters.

The outcome of the training and testing process showed that on the given dataset up to twelve classes of characters can be classified with an accuracy of over 75%. Increasing the character classes over twelve resulted in an unacceptably high error rate. The results could be improved by further processing of the character samples; and carrying out extensive sensitivity analysis for finding the optimal ANN structure. Future improvements, such as parallelisation of the learning algorithm, could be beneficial to speed up the learning process.

Acknowledgements

I would like to thank my supervisor Dr. Richard Neville for his support and mentoring throughout this project.

1 Introduction

1.1 Chapter Overview

The chapter introduces the problem domain under inspection, briefly discusses existing solutions, proposes a solution and puts forward the scope for this project. An outline of the project structure and writing conventions will also be provided at the end of the chapter.

1.2 Problem Summary

This project investigates and researches utilising and building computer software for recognising handwritten characters using *artificial neural networks* (ANN). The process of recognising handwritten characters is commonly known as *optical character recognition* (OCR) [1].

The task of recognising other people's handwriting is fairly simple for humans given that they comprehend the character set of the given document. In the case of handwriting that is difficult to read for any reasons, the human brain learns to decipher the characters when presented with more examples of the same handwriting. The task that our brain faces is to map the imperfect (or unseen) written characters into classes of known characters.

The aforementioned process can be computerised with the help of character recognition software. While the human brain possesses complex cognitive abilities that facilitate learning, the same task is relatively complex to achieve on a computer. The diverse inputs to the software system place high generalisation[1] requirements to the ANN model in order the software to be of value to the user. The ideal software system should be able to classify a wide array of characters written in various styles into their corresponding alphabetic classes.

1.3 Existing Solutions

There is a wide variety of advanced commercial OCR solutions on the market, either as an end user product or a software development kit. Microsoft has built OCR capabilities into its various products such as Microsoft Office Document Imaging [2] and OneNote [3]. Tesseract is an OCR engine that is currently maintained by Google. Originally, it was developed by Hewlett Packard (HP) in the 1980s and then released as open source software in 2005 [4]. Leadtools OCR software development kit provides a building block for developers to integrate OCR functionality into their applications [5]. This is far from an exhaustive list, but highlights the fact that the OCR software market has various mature products available.

[1] Generalisation is a property of an ANN whereby the model performs well when classifying unseen data [14].

1.4 Project Proposal

The aim of the project is to research the domain of character recognition using artificial neural networks and develop a software package that performs character recognition on the characters of the English alphabet. The piece of software should be accessible via a graphical user interface. This project does not try to add ground breaking improvements to the problem domain due to the fact that there are already very advanced solutions available commercially and as free software. The effort to innovate in the area would not fit into the time frame of the current project. Therefore it targets to be a learning aid to the author to gain deeper knowledge of the domain and provide experience of building a robust piece of software.

1.5 Report Structure

This report consists of eight core chapters and documents the phases undertaken during the project. The following is the outline of the chapters with a short summary of each.

1. Introduction:

   The chapter introduces the problem domain under inspection, briefly discusses existing solutions, proposes a solution and puts forward the scope for this project. An outline of the project structure and writing conventions will also be provided at the end of the chapter;
2. Background:

   This chapter will provide the relevant background information that is necessary to introduce the research topic. Pattern recognition, which is the wider domain that the project falls into, is being discussed. A brief look into the history of character recognition will augment and form and adjunct to the paragraph from introduction chapter on the current solutions on the market;
3. Research:

   This chapter documents all the subdomains in this project, but specifically gives an introduction to the theoretical background to each of them. The areas covered include general research, software engineering, software design, algorithms and data structures;
4. Requirements:

   The chapter describes the process of requirements engineering (RE) and what it entails. It then presents the table of stakeholders, the context diagram, use cases, the use case diagram and functional and non-functional requirements;
5. Design:

   The chapter provides an introduction to object oriented analysis, design methods and presents various artefacts of the design phase of the project such as domain model, class diagram and class responsibility collaborator (CRC) cards. It also discusses the choice of technology platform for this project and provides details of data persistence strategy for the application;
6. Implementation and Results:

   The chapter introduces the implementation strategy and describes briefly the tools that were used for this project. The majority of the chapter is

devoted to documenting the prototypes that led to the final product. Each prototype will be documented in two parts: I) description - outlines the implemented features and algorithms; and ii) results – conveys and analyses the outcome of the prototype;

7. Testing:

   The chapter introduces the topic of software testing and discusses various methods for functional and non-functional testing, the levels of testing and automating the testing process. A reference to the test case table of automated test for the project will be provided; and

8. Conclusion:

   The final chapter of the report summarises the work undertaken for this project, reflects on challenges faced, outlines the new skills attained and highlights the main achievements. Furthermore, evaluation of the overall results and future prospects of the project will be provided.

1.6 Writing Conventions

IEEE 2006 square brackets referencing style, where [n] refers to the n^{th} entry in the listing of bibliography, is being used in this project paper. The notation for cross-referencing uses the numbered chapters and sub-chapters to refer to the relevant part in the paper itself. For instance, 4.2 would refer to the second sub-chapter in the fourth chapter. Foot notes are being utilised throughout the report to clarify various concepts. They are noted by a superscript number that refers to the relevant comment at the bottom of the page.

1.7 Closing Remarks

This chapter gave an introduction to the report and outlined the main topics covered. The following chapter will provide further details on the domain of character recognition and how it has evolved to this date.

2 Background

2.1 Chapter Overview

This chapter will provide the relevant background information that is necessary to introduce the research topic. Pattern recognition, which is the wider domain that the project falls into, is being discussed. A brief look into the history of character recognition will augment and form and adjunct to the paragraph from previous chapter on the current solutions on the market (see §1.3 that discusses existing solutions on the market).

2.2 Introduction to Pattern Recognition

> "Pattern recognition is the scientific discipline whose goal is the classification of objects into a number of categories or classes," *[6]*.

The objects specified in the quote can be anything from images and sound waves to more complex data sets. The types of input data create basis for the various subdomains of pattern recognition: speech recognition; machine vision; data mining; and character recognition - among others [6].

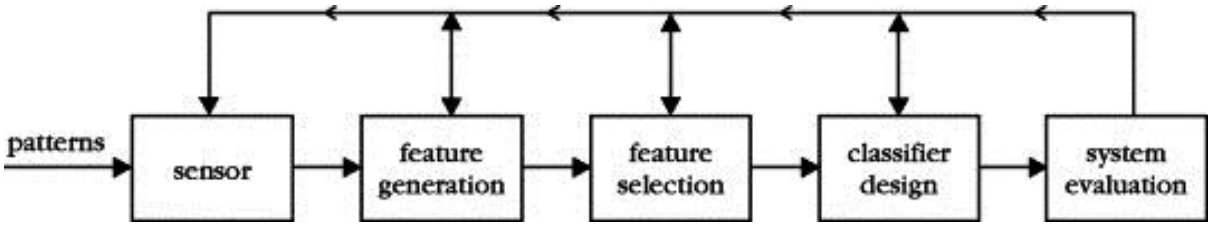

Figure 2.1: The general stages of pattern recognition, [6].

Figure 2.1 outlines the general pipeline[2] of a pattern recognition system. First, the input pattern is digitised - with the help of a sensor. That could be a microphone for capturing sound - waves or a camera - for visual images. The next stage is to generate feature vectors[3] from the data, for instance, pixel values for an image - which are two dimensional in nature, are transposed to a one dimensional vector – that can be clamped on to the input of an ANN after any normalisation has been undertaken. Usually the raw set of features is too large for processing and contains an unnecessary amount of noisy data [7]. A low resolution picture of one megapixel may already contain one million features – i.e. pixel values; or sets of features. Therefore, analytics must be performed in order to select the appropriate number of features that will actually be used for classification. The process of feature selection aims to pick out the features that best characterise the distinctive properties of the object being classified [8]. After feature selection, the subset of features is then presented vector by

[2] Pipeline is used here in the meaning of flow of actions.
[3] Feature vector is a representation of the object that is being classified in an OCR system [7].

vector to the input of the classifier that performs classification based on its parameters. The system evaluation stage will assess how well the whole model generalises in the real world with unseen data [6].

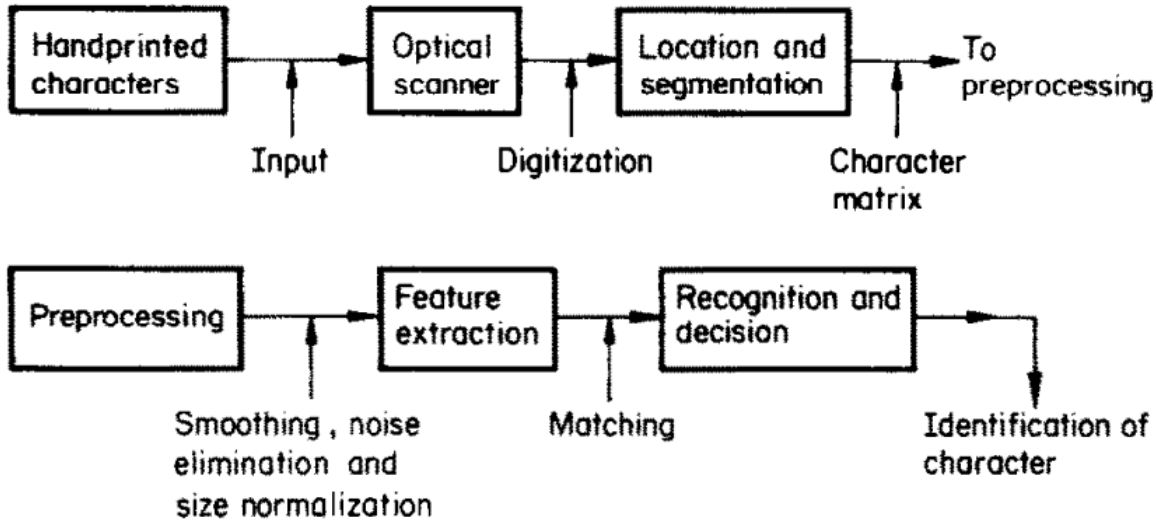

Figure 2.2: An example of a pipeline of an optical character recognition system, [7].

Figure 2.2 provides a more specific abstraction of an optical character recognition system than the previously presented pattern recognition pipeline (see figure 2.1). Note it is the preprocessing stage that normalises the data set and eliminates noise from digitisation stage before proceeding to the next step of feature extraction. The preprocessing stage is important to reduce the complexity of feature extraction and facilitate accurate classification in the recognition phase [7] [9] .

2.3 A Brief History of Character Recognition

Character recognition is a problem that has been under investigation since the early 20th century [10]. The first devices from the time mainly focussed on making written text audible to the blind [10] [11]. Optophone, a device that could make written text audible by marking different letters with different sound signals, was described by Dr. Edmund Fournier d'Albe in 1914 [12]. OCR as known in the current day, however, had it advent with the invention of the digital computer [6] [8] [10].

In 1950s, the first commercially viable character recognition systems were produced by the Intelligent Machines Research Corporation and sold to clients such as Reader's Digest, Standard Oil and United States Air Force for digitising hand written documents [10] [11]. An important driver of OCR technology in the early days was the vast amount of paper based data such as bank cheques and government reports, which became impossible to handle manually [8]. The rapid development of OCR technology in the 1970s can largely be contributed to standardisation of paper, ink and print fonts. This made possible high recognition rates at a feasible cost and revolutionised the data input industry [1]. The later research in the domain has been moving from character recognition to word recognition that utilises word context to improve classification accuracy [13].

2.4 Introduction to Neural Networks

2.4.1 Neurobiological background

The computational model of artificial neural network is inspired by the neurobiological neurons that are the building blocks of the human brain [14]. Figure 2.3 illustrates the functioning of the neurobiological neuron. The dendrites connecting to the cell body accept input signals, also referred to as spike trains, from other neurons through junctions called synapses, which are either inhibitory[4] or excitatory[5] [14]. The communication between neurons is facilitated with the help of electrical signals received from dendrites. Signals from different neurons have different strength and they will be integrated in the cell body. A simplified model could then describe the following process as comparing the integrated signal strength to a threshold and deciding if to fire an output signal through the axon. The axon is the connection to another neuron. The same process takes place then in the other neurons that receive signal from this neuron.

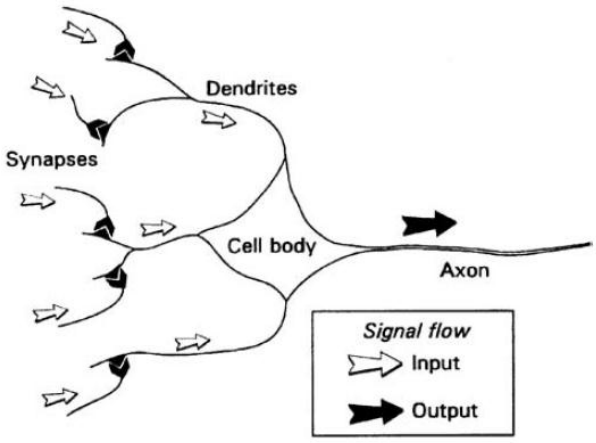

Figure 2.3: A model of the neurobiological neuron, [14].

2.4.2 Artificial Neural Networks

An artificial neural network is a computational model consisting of a set of interconnected simple processing units that pass signals between each other to produce output [14]. These processing units, which are referred to as neurons similarly to the neurobiological counterparts, are the basic computational units of neural networks [15]. In this paper from now on, the author uses the terms, artificial neural networks and neural networks, referring to the computation model of neural networks and not to the neurobiological counterpart.

Figure 2.4 shows how a single neuron operates in a neural network. The input signal denoted by the vector $x_0, x_1 \ldots . x_n$ is being multiplied element wise by the weight vector $w_{j0}, w_{j1} \ldots w_{jn}$ of the neuron. Note that $x_0(x_\theta)$ is a special input element called the bias (θ) that has always value of -1 (or 1 depending on the implementation) [15]. The output of the neuron is calculated by using an activation-output (a-o) function; which is a mathematical a-o function of the weighted sum of the inputs [13]. The activation function is usually a nonlinear mathematical a-o function; the a-o is

[4] Inhibitory synapse reduces the input signal strength [81].
[5] Excitatory synapse increases the input signal strength [81].

normally a sigmoidal function which endows the neural network with the ability – if the neural network consists of three layers - to solve nonlinearly separable problems[6] [15].

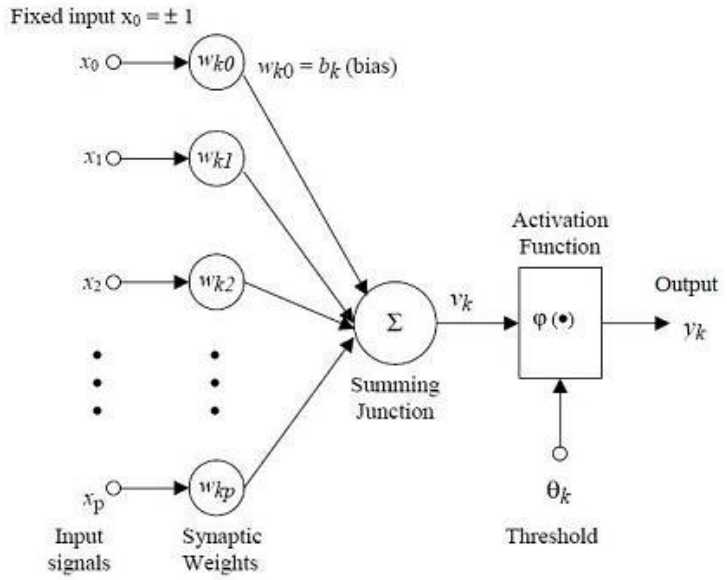

Figure 2.4: A diagram illustrating the functioning of a single neuron, [16].

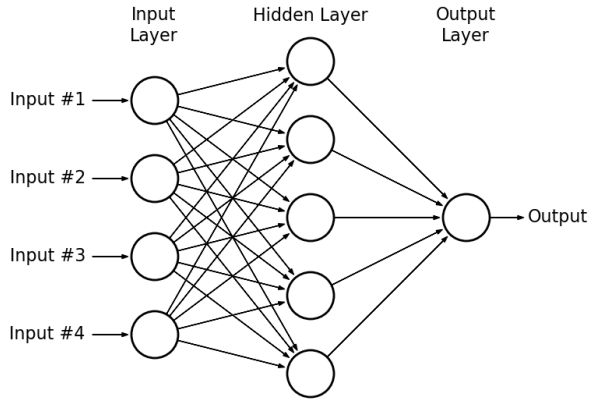

Figure 2.5: An example of a three-layer neural network, [17].

Figure 2.5 shows an example of a three-layer neural network with four neurons in the input layer, five in the hidden layer and one neuron in the output layer; a 4-5-1 ANN topology. All the neurons in the network as per the diagram in figure 2.3. The network is fully connected, meaning that every neuron in a layer has a connection to all the neurons in the following layer [14]. This is an example of a feedforward network, meaning that input signal travels only from left to right in the network diagram (see figure 2.5) [14]. Once the output is calculated at the output layer, it can be compared to the expected target value if it is available. The network is called a supervised neural network if the

[6] Nonlinearly separable problem is a problem where data points belonging to different classes in the dataset cannot be separated by a hyperplane. Linearly separable problems fulfil the aforementioned condition [14].

target values are available. The difference between the expected target and the actual output provides necessary delta (δ) that is utilised to adjust the weight values of the neurons. This process is called *training or learning* [14].

2.5 Training Artificial Neural Networks

The training of neural networks refers to the process of modifying the weight values of the neurons in order to adapt the network function to perform a specific function (function mapping task). In general, the neural network training is divided into two classes – supervised and unsupervised training [14] [18].

2.5.1 Supervised Learning

Supervised training algorithms assume that the correct classification of input data is known beforehand – hence there is a set of target vectors that the ANN can be trained to map. This enables comparison of the network output to the actual classification (class, or target class) and modifying the weights accordingly. Examples of supervised training algorithms are the perceptron, delta rule and back propagation, which will be inspected in detail in the research chapter [14]. The process of training is also referred to as learning.

2.5.2 Unsupervised Learning

Unsupervised learning algorithms do not need the actual class of the input data and operate on the structure of the input data. An example of unsupervised learning algorithm is clustering. Clustering attempts to divide the training samples into a number of groups based on their similarity. Unseen data samples will be later classified into these groups based on their characteristics [14]. The training algorithms used in this project belong all to the supervised group and therefore unsupervised training will not be discussed in the research chapter.

2.6 Closing Remarks

The background chapter provided a high level introduction into pattern recognition, looked briefly into the history of character recognition and laid the foundation for understanding neural networks. The next chapter will cover the theoretical research carried out for this project.

3 Research

3.1 Chapter Overview

This chapter documents all the subdomains in this project, but specifically gives an introduction to the theoretical background to each of them. The areas covered include general research, software engineering, software design, algorithms and data structures.

3.2 General Introduction into Research

Research entails systematic creative work in order to improve the knowledge base, find answers to a problem or come up with novel applications. Research can be divided into three large categories: basic, applied and experimental development. Applied research, as the most relevant methodology for the current project, is concerned with acquiring new knowledge for reaching a specific goal. Another well-known categorisation is qualitative versus quantitative research. Quantitative research makes use of analysing numeric datasets to reach its aims, whereas qualitative research relies on various non-numeric sources. This project mainly applies qualitative research methods in order to find out about the possible ways of producing software [19] [20]. Assessing and analysing the source materials is a crucial part of conducting research [21]. The research undertaken for the current project relies mostly on reviewing relevant books, articles and online resources.

3.3 General Introduction into Software Engineering

One of the artefacts of the current project is a software package. Therefore, the project needs to make use of existing software engineering knowledge. According to IEEE, software engineering is defined as:

> *"The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software," [22].*

Numerous methodologies, outlined in literature and practiced in industry, approach the software engineering process from different angles. The following paragraphs provide introduction to some of the more widespread approaches.

3.4 Software Development Methodologies

A software development methodology defines the structure and activities that should be undertaken during a software project. Software development methodology can also be referred to as software development life cycle (SDLC). An SDLC is based on four basic phases:

1. Planning;
2. Analysis;
3. Design; and
4. Implementation.

The difference between various methodologies lies in how they approach each of the phases and what is being emphasised. On high level, the methodologies can be grouped into sequential and iterative processes [23]. Sequential processes go through a number of predefined stages that lead to a completed product. An example of a sequential process is the Waterfall method. Iterative processes use a sequential process as a building block. The underlying sequential model is being repeated to produce incrementally growing product until the requirements are being met. An example of an iterative process is the Unified Process [24].

3.4.1 Waterfall

The waterfall method is a sequential process whereby a project is divided into a number of stages corresponding to [23]:

1. System engineering;
2. Requirement analysis;
3. Design;
4. Implementation or coding;
5. Verification or testing; and
6. Maintenance.

Every stage is dependent on full completion of the preceding stage and cannot be revised later on during the project. The main arguments for using the waterfall methodology are that it has: a clear structure to the project, specific deliverables that denote milestones, testability of deliverables against requirements and ease of defining progress. The drawbacks of using the methodology are inflexibility to changes after agreeing on requirements, and the difficulty of defining all the requirements early in the project and relatively long time gap before seeing actual working software [23]. In practice, however, the business environment can change during the project lifetime, which makes the approach obsolete by current day standards. Past evidence shows that the Waterfall methodology is associated with high rates of failure and inefficiency [25].

3.4.2 Spiral

The spiral software development methodology addresses the inflexibility of the waterfall process by introducing incremental development of the product and supporting documentation. It uses the same phases as the waterfall method, but in an iterative manner. It enables revising errors from previous phases and focussing on architecturally riskier parts earlier in the project. It enables to deliver most important functionality for the client first and also take into account feedback received in the following iterations. The negative impact of the Spiral approach is that it relies heavily on the software engineers' ability to assess riskiness and the iterative nature makes it difficult to foresee the actual length of the project [23].

3.4.3 Prototyping

Prototyping is an iterative software development methodology that focuses on delivering partial working models of the project to the client. Clients then provide feedback on the prototype[7] and revisit the requirements. Analysis, design and implementation phases are being performed at the

[7] Prototype is a rapidly built model of software that implements a subset of the final set of features used for feedback purposes [24].

same time to produce prototypes with minimal required features. The process is being repeated until the stakeholders approve the solution. The system can be further refined afterwards to meet the full acceptance criteria [23].

Prototyping addresses the issue whereby clients do not know in detail what functionality they need at the beginning of the project. Interacting with early prototypes enables them to better understand their requirements. Customer feedback is the central aspect of this approach. It provides the developers with information that guides them to build functionality that users actually care about. The approach prioritises delivering software over spending time on excessive documentation [24].

One of the main disadvantages is that the focus on building quick prototypes can lead to weak architectural designs. There is also a tendency for the simplistic and inefficient prototyped features to make it into the final version of the software [23].

3.4.4 Agile Development

Agile refers to a group of software development methodologies that aim to produce versions of working software throughout the project life cycle. The four ideas highlighted by the agile manifesto reflect the general approach taken [25]:

1. Individuals and interactions over processes and tools;
2. Working software over comprehensive documentation;
3. Customer collaboration over contract negotiation; and
4. Responding to change over following a plan.

The agile principles further stress the importance of producing working software in iterative manner, close collaboration with the customers, acceptance of change throughout the project and working in self-organising teams. Modelling and documentation should not be omitted, but used wisely only to support the development process [26].

The critics claim that agile methods lead to “hacking” culture where modelling and analysis are not embraced enough. Lack of documentation can make the system difficult to audit. It is also questionable if agile methodology is suitable for developing large scale critical systems [24].

3.4.5 Extreme Programming (XP)

Extreme Programming is an agile software development methodology aiming to minimise formal analysis and design processes focussing on code delivery, as specified in [27]:

> *“A philosophy of software development based on the values of communication, feedback, simplicity, courage, and respect,” [27].*

User stories are captured in order to understand requirements. This requires the clients to be heavily involved in the development process in order to clarify any questions that the developers might face. Developers make heavy use of test driven development, paired programming and refactoring. All this makes the methodology very nimble to respond to changes in user requirements [24] [27].

Extreme Programming assumes strong coherent teams of developers for the model to work. This might make the wide-spread practice of hiring external contractors questionable. Another criticism

to the methodology is that it has been proved a successful approach for small projects, but large mission critical systems need more formal processes to capture all the details [24]. Extreme Programming deems the costs of communicating requirements through a number of various team roles, as in more structured methodologies, too high and therefore it cannot handle a project with very large requirements base [28].

3.4.6 Unified Process

The Unified Process is an iterative software development methodology that consists of four phases: inception, elaboration, construction and transition. Each of the iterations is timeboxed[8] and results in a usable software artefact that implements a subset of project requirements. The various users of the software will be involved in testing and feedback sessions throughout the project so that the end product would meet the client's expectations [25].

3.4.7 Kanban

Kanban is a task management technique which defines a pipeline that all the project's subtasks need to flow through. Kanban itself is not a software development methodology, but merely a tool for managing work load. Therefore, Kanban should be used in conjunction with an existing software development methodology. It enables easy visualisation of the overall project state, tracking of individual tasks and encourages incremental development approach. The main characteristics of a Kanban system are that there is always a limit to work in progress and there a signal for pulling in new work should be in place. Kanban can be implemented (drawn) on a physical whiteboard with sticky notes as tasks or as a software package [29].

3.4.8 Methodology Chosen

The agile methodology in combination with Kanban workflow control was used for this project as it enables rapid development of working software and provides the necessary flexibility to address a complex problem domain effectively. Timeboxing iterations ensures keeping focus on writing usable pieces of software rather than building software that is constantly in development stage. Agile is also the methodology taught in software engineering courses at the University of Manchester and therefore the author can build on the experience gained during studies.

3.5 Introduction to Algorithms and Data Structures

An algorithm is a set of instructions to accomplish a specific task [30]. Algorithms are generally platform independent and can be written down in an informal natural language or formal mathematical notation, whichever is more suitable for the occasion [31]. A data structure is a way of storing and organising data so that specific algorithms could operate efficiently on it [32]. Examples of data structures are arrays, maps, trees and graphs. Algorithms and data structures are inherently related, because data structures would not be of any use without algorithms to operate on them [31]. The basic operations like data retrieval, data insertion and sorting all perform based on a certain kind of algorithm. These algorithms can be simple like element insertion into an array or complex like lookup from a hash table. Nevertheless, algorithms facilitate efficient usage of data structures. Two of the most utilised data structures in this project are discussed next.

[8] Timeboxing is the process fixing the length of a software project iteration that results in working software. Can involve descoping requirements, if necessary, to meet the deadline [25].

3.5.1 Arrays

An array is an ordered set of elements consisting of a fixed number of elements and each element is accessible via an index [33] [34]. Therefore the operations, such as select and insert, always perform in constant time $O(1)$. Arrays can be one-dimensional, whereby each element is the stored value object, or multidimensional, whereby each element of the array is itself an array. Multidimensional arrays provide convenient representation for matrixes. The importance of arrays lies in the fact that they are a suitable data structure for a wide range of problems and also serve as a building block to many other data structures [33].

3.5.2 Dictionaries

A dictionary, also known as a map or a lookup table, is a collection of key and value pairs, where all the key values must be unique [35]. A dictionary is an abstract data type[9] that allows the following operations: insertion, removal, lookup and modification [30] [35]. More operations can be available depending on the specific implementation. The efficiency of the operations is similarly dependent on the specific implementation. Dictionaries are a suitable data structure when building an index, where the key to an element, contrary to arrays, is itself a complex data type.

3.6 Supervised Algorithms for Training Neural Networks

Supervised learning entails classifying training data using the given neural network and adapting the weights of the network neurons according to the classification error [14]. Training continues until the error on the training set is sufficiently small. It is necessary, however, to avoid over fitting - adapting the neural network to perform exceptionally well on the training set, but resulting in poor generalisation[10] [36]. Next, three supervised learning algorithms for neural networks are being discussed.

3.6.1 The Perceptron

The perceptron is a learning algorithm for single layer neural networks. For simplicity, let's assume that we are dealing with a neural network consisting of a single neuron. The neuron takes a Boolean vector v labelled t as input. The neuron has a weight vector w that is of length $n + 1$, where n refers to the number of elements in the training vector v. The extra weight, which is constantly set to -1, is necessary to adjust the threshold θ of the neuron similarly to the weights. This allows the activation function outlined below to use 0 as the threshold mark instead of θ. Before the training starts, all the elements of weight vector w need to be initialised to random numbers between 0 and 1. The symbols defined in this paragraph are used in the same context for describing other algorithms in next chapters and will not be repeated. The training of the neuron then takes place according to the steps outlined in Table 3.1 [14].

[9] Abstract data type is an abstract data structure that is defined purely by operations that can be implemented on it [82] [83].

[10] Generalisation is the property whereby a neural network generalised on a training data set also performs reasonably well on unseen data [36].

Table 3.1: The high-level algorithm for the perceptron.

	STEPS OF THE ALGORITHM
1:	$initialise\ weight\ vector\ w$
2:	$\boldsymbol{do}$
3:	$\quad\boldsymbol{for\ each}\ training\ vector\ pair\ (v, t)$
4:	$\quad\quad evaluate\ the\ output\ y\ of\ the\ neuron\ where\ v\ is\ input\ to\ the\ neuron$
5:	$\quad\quad\boldsymbol{if}\ y \neq t\ \boldsymbol{then}$
6:	$\quad\quad\quad adjust\ weight\ vector\ w\ by\ adding\ a\ fraction\ of\ the\ error\ to\ w$
7:	$\quad\quad\boldsymbol{end\ if}$
8:	$\quad\boldsymbol{end\ for}$
9:	$\boldsymbol{while}\ y \neq t\ for\ all\ training\ vectors$

The output y of the neuron is a calculated using the value a of the threshold activation function according to equation (3.1) [14].

$$a = w \cdot v;\ y = \begin{cases} a \geq 0 \rightarrow 1 \\ a < 0 \rightarrow 0 \end{cases} \tag{3.1}$$

Adjusting the weight vector w takes place by adding a fraction of the classification error to it. The fraction is referred to as learning rate and denoted as α. Learning rate is usually a number in the range of $0 < \alpha < 1$. The modification of the weight vector then takes place according to equation (3.2), where subscript i denotes the i[th] element of the vector [14].

$$w_i = w_i + \alpha(t - y)v_i \tag{3.2}$$

The perceptron is the simplest neural network training algorithm and can solve only linearly separable problems. In case of a nonlinearly separable problem, it is necessary to use a nonlinear activation function and a more advanced learning algorithm [14].

3.6.2 The Delta Rule

The delta rule is a more advanced learning algorithm than the perceptron. It uses gradient descent, which is an optimisation algorithm, to minimise the neural network error by adjusting the weights of the neurons. Gradient descent does guarantee, however, reaching the global minima of the error function and can be stuck in a local minima. One way of escaping this problem is to present the training data samples in a random order. The high level algorithm for the delta rule is as described in Table 3.2 [14].

Table 3.2: The high-level algorithm for the delta rule

	STEPS OF THE ALGORITHM
1:	$initialise\ weight\ vector\ w$
2:	$\boldsymbol{do}$
3:	$\quad\boldsymbol{for\ each}\ training\ vector\ pair\ (v, t)$
4:	$\quad\quad calculate\ the\ weight\ adjustment\ when\ v\ is\ input\ to\ the\ neuron$
5:	$\quad\quad adjust\ each\ of\ the\ weights$
6:	$\quad\boldsymbol{end\ for}$
7:	$\boldsymbol{while}\ the\ rate\ of\ change\ of\ the\ error\ is\ not\ sufficiently\ small$

The stopping criterion of sufficiently small rate of change is subjective and needs to be fixed by experimentation. It is now necessary do define the rule for updating weights and calculating the error function value e. These two differ depending on the type of the activation function chosen.

3.6.2.1 Case: Linear Activation Function

Linear[11] activation functions can solve only linearly separable problems and are therefore of limited use in the context of neural networks. The weight vector update uses the difference of the target value t and activation a to determine the size of the weight change as stated in equation (3.3) [14].

$$w_i = w_i + \alpha(t - a)v_i \tag{3.3}$$

The error e on classifying a pattern is defined as a squared error of the difference of the target value t and activation a. In case of more than one neuron, the error value needs to be summed over all the M neurons as outlined in equation (3.4) [14].

$$e = \frac{1}{2}\sum_{i=1}^{M}(t_i - a_i)^2 \tag{3.4}$$

3.6.2.2 Case: Nonlinear Activation Function

Nonlinear functions, such as sigmoid[12], are widely used with neural networks, because they can solve nonlinearly separable problems. When using a nonlinear activation function, the weight update rule gets an extra term that is the derivation of the activation function. In addition to that, activation a has been replaced with the neuron output y [14]. The complete weight update rule is described in equation (3.5).

$$w_i = w_i + \alpha\partial'(a)(t - y)v_i \tag{3.5}$$

The error function value e is adjusted similarly to use the neuron output y instead of activation a as stated in equation (3.6) [14].

$$e = \frac{1}{2}\sum_{i=1}^{M}(t_i - y_i)^2 \tag{3.6}$$

3.6.3 Back propagation

Backpropagation is an advanced training algorithm for multilayer[13] neural networks with hidden layers. It is named after the process of propagating the errors in the output layer to the hidden layers. The high level algorithm for backpropagation outlined in Table 3.3 [14].

[11] Linear function is defined as $f(x) = ax + b$, where a and b are constants [84].

[12] Sigmoid is a function defined as $f(x) = \frac{1}{1+e^{-x}}$ [84].

[13] Multilayer refers to consisting of more than one layer.

Table 3.3: The high-level algorithm for backpropagation.

	STEPS OF THE ALGORITHM
1:	*initialise weight vector w*
2:	***do***
3:	***for each*** *training vector pair* (v, t)
4:	*classify the training vector pair* (v, t)
5:	*calculate the difference between network output y and target t*
6:	***for each*** *neuron in output layer*
7:	*adjust the neuron weights using gradient descent*
8:	***end for***
9:	***for each*** *neuron in hidden layer*
10:	*calculate the weight change*
11:	*adjust the weights of the neuron using gradient descent*
12:	***end for***
13:	***end for***
14:	***while*** *training error is higher than wanted*

The general rule for adjusting the weights of a neuron is outlined in equation (3.7). The subscript k refers to the index of the neuron and subscript i to the index of the weight. The difference between adjusting a neuron in output and hidden layers is in the definition of the delta term δ_k [14].

$$w_{ki} = w_{ki} + \alpha\delta_k v_{ki} \tag{3.7}$$

The weight adjustment for the output neurons is identical to the delta rule and achieved by fitting equation (3.8) into equation (3.7) [14].

$$\delta_k = \partial'(a_k)(t_k - y_k) \tag{3.8}$$

The complicated part of the algorithm is the weight adjustment rule for the neurons in the hidden layers. There is access to the errors made at the output layer, but not in the hidden layer. The errors made at the output layer are influenced by the input from the hidden layer and therefore should be appropriately credited to the hidden layer. It is an example of a credit assignment problem, whereby it is necessary to assign the right amount of blame to the hidden layer for causing errors in the output layer. For a single neuron k in the hidden layer, the solution is to take all the output layer neurons I_k that neuron k has connection to and sum up the error δ^j made in the output neuron j and multiply it to the weight w_{jk} that connects neuron k to neuron j. The result needs to be multiplied by the derivation $\partial'(a_k)$ of the activation function over the value of the activation a of the neuron k as outlined in equation (3.9) [14].

$$\delta_k = \partial'(a_k) \sum_{j \in I_k} \delta^j w_{jk} \tag{3.9}$$

Lastly, there is a problem of deciding when to stop training. The condition has to be dependent on the classification error of the network, which for backpropagation is defined exactly like for the delta rule (see §3.6.2.2). One way is to train until the change of network error is sufficiently small. Another option is to set an arbitrary small target error value. The third solution is to train until the network

output vector has values, which are closest to the actual Boolean values. For example, all values larger than 0.5 mark output of 1 and the rest mark the output of 0 [14].

3.6.4 Momentum

Generally, the weight adjustments made are governed by the learning rate α. If the learning rate is too high, then the error function might not converge and keeps oscillating. Whereas too small value will result in very slow training. One way of approaching the problem of balancing between these extremes is to introduce an extra term to the weight adjustment rule. The extra term will add a fraction of the previous weight change to the current weight change. This way the weight adjustment gains "momentum" when moving down a steady slope on the error plane and also slows down when oscillation happens, because the weight changes with different signs will cancel each other out. Equation (3.10) represents the momentum term, where n stands for the n^{th} change in weight w [14].

$$\Delta w(n) = \alpha\delta(n)v(n) + \lambda\Delta w(n-1) \tag{3.10}$$

3.7 Closing Remarks

This chapter covered the majority of research carried out for this project and provided the theoretical background for design and implementation phases. The next chapter will discuss and document the requirements gathering phase of the project.

4 Requirements

4.1 Chapter Overview

The chapter describes the process of requirements[14] engineering (RE) and what it entails. It then presents a sequence of deliverables and abstract requirement (or design) diagrams and tables, temporally ordered to aid the development and comprehension of the requirements, these are: i) the table of stakeholders; ii)the context diagram; iii) use cases and associated use case diagram; iv) functional; and v) non-functional requirements tables.

4.2 Requirements Engineering Overview

Requirements engineering is a systematic process of determining, formulating, prioritising and documenting the goals that the project is expected to meet [37] . The four core activities of RE [37]:

1. Elicitation – the gathering of requirements;
2. Documentation – describing and recording the requirements;
3. Validation and negotiation – checking and discussing the requirements; and
4. Management – dealing with change and uncertainty.

Elicitation of requirements is the core activity of RE and takes place through various methods such as interviews, observation, surveys, brainstorming, workshops and document analysis. It is necessary to agree on mutual rights and responsibilities to facilitate cooperation and communication. The main sources for requirements are stakeholders, existing systems and documents [37]. Requirements elicitation provides input for documentation. To start this process, stakeholder analysis §4.3 was undertaken as one of the first steps in the process for this project.

Some of the most common formats for eliciting, compiling and documenting requirements are data tables, various UML diagrams and descriptive text documents. It is of primary importance that these documentations are comprehensive and of good quality. Principles such as unambiguity, consistency, clear structure, extensibility and completeness should be followed [37]. Documentation provides common understanding of the project goals and sets ground for communication between the project stakeholders [38]; in this context we present the stakeholder analysis §4.3; context diagram §4.4; use case modelling §4.5 and §4.6; functional requirements table §4.7; and non-functional requirements table §4.8, in order to achieve these goals.

Validation of requirements involves checking with the client that the documented requirements indeed meet their expectations. It is important to make sure that the requirements provide an adequate basis for continuing with the design and implementation phases [38].

[14] Requirement is a capability that a system has to provide, needed by a user to achieve an objective [37] [85].

It is inevitable however, that requirements change during the project lifecycle and therefore it is important to manage the change process [38]. In light of the chosen agile SDLC, the RE process takes place throughout the project – with more emphasis at the start of the project and then gradually less, when requirements are finalised and signed off. Furthermore, requirements can be revisited after initial documentation [25] [39]. This is in contrast to the waterfall model that freezes the requirements before actual development starts (see §3.4 for comparison of software development methodologies).

Nevertheless, requirements engineering is a crucial part of a software project to get right. Research shows that 60% of errors in the project originate from RE phase. Furthermore, an error can be 20 times more costly to fix if found in development phase or even up to 100 times more costly if discovered in acceptance testing phase [37].

4.3 Stakeholder Analysis

> "A stakeholder in the architecture of a system is an individual, team, organization, or classes thereof, having an interest in the realization of the system," *[40]*.

Defining the stakeholders will enable the author to address their expectations, issues, and general requirements in general; and hence manage the risks of the project not meeting its goals [40]. Stakeholders are also the main sources of requirements for a project [37]. The current project is limited in scope[15] and is being carried out in an academic environment and therefore has a low number of stakeholders outline in table. In industry, a project could have tens of stakeholders with conflicting interests. Assessing and mapping these possible conflicts early on will assist the implementer of the project in making critical decisions.

Table 4.1: List of Stakeholders

NAME	ROLE	INVOLVEMENT
Erkki Lukk	Author	Implements the project according to the project plan.
Richard Neville	Supervisor	Provides strategic support and guidelines in order to meet the project's goals.
Liping Zhao	Second Marker[16]	Provides feedback and evaluation of the successfulness of the project
Erkki Lukk	User	Uses the system to create, train and test neural network structures.

Table 4.1 describes the stakeholders of the current project and their involvement. There are four stakeholders outlined – the author, supervisor, second marker and user. The impact on the project progress is influenced the most by the author and the supervisor as they have direct access to the code base and the documentation. The author implements the project, whereas the supervisor only provides strategic advice without getting directly involved with implementation. The second marker will review and assess the project via various formats such as the project seminar, project demo and

[15] The scope of the project is discussed in §1.4.
[16] Second marker is a person from academic staff who is responsible for marking the project.

submission of final report. The user accesses the software system for creating, training and testing neural network structures. Other stakeholders, such as third year project course leader, could be documented, but their impact to the project is almost negligible and therefore they have been left out from the stakeholder table.

4.4 Context Diagram

A context diagram is a UML diagram whose aim is to capture the environment that the software is operating in [24]. It maps the important external actors and their interactions with the system indicating the direction of the flow of information and data [40]. Context diagram describes the system on a high level of abstraction and provides a limited amount of details about the system and the external actors. Context diagram is a suitable tool for facilitating communication between the project stakeholders and helping to decide what is in scope of the system and what is not. Context diagram can be used for identifying any external dependencies or services that might be needed and assessing the impact of them [41]. It also and enables mapping of historic or legacy information (i.e. standard forms, or data storage) to actors representing these items The diagram is often produced before moving on to more formal artefacts such as use cases and use case diagrams [42].

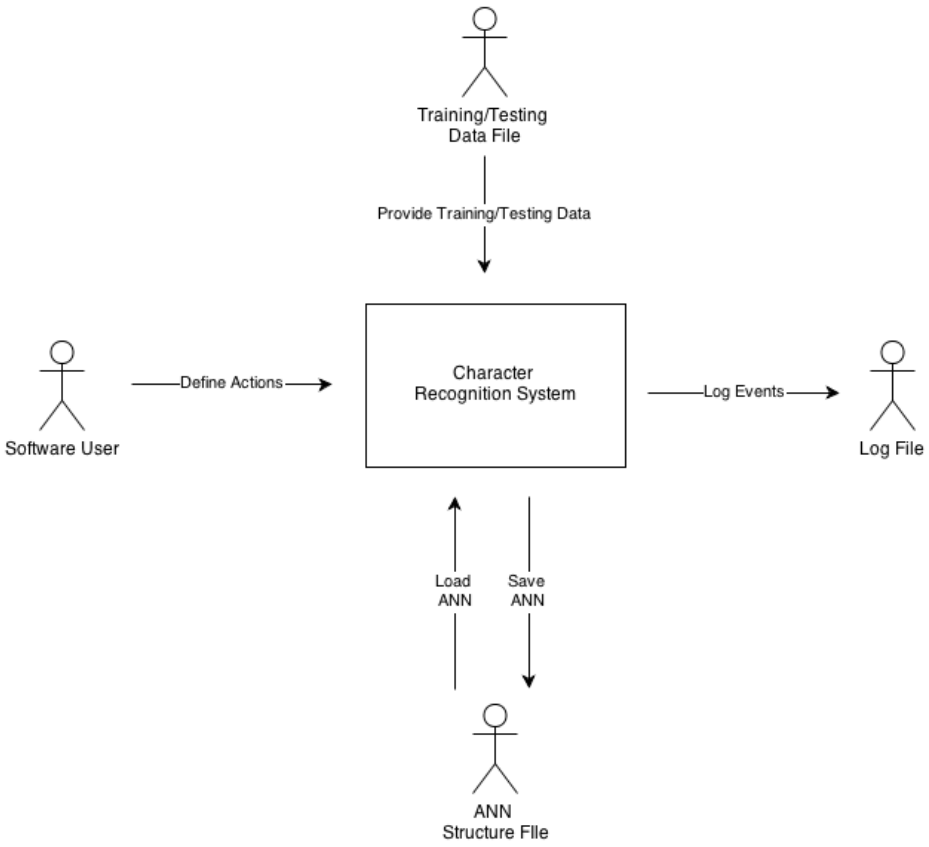

Figure 4.1: A context diagram outlining the main actors and their interaction with the system.

Figure 4.1 presents the context diagram for the current project. At the centre of the diagram is the OCR system that receives testing and training data from a file. The software user defines the sequence of actions being performed on the OCR system. The OCR system model can be saved to a configuration file and loaded from the same file at a later time. All the important events of the system are being logged to a log file.

4.5 Use Case Modelling

Use cases are simple user stories that capture a user using the system to achieve a specific goal [25]. Each use case consists of a name and the description of the action, which essentially is a story of what the user intends to achieve and how the system enables it (the use case scenario) [43]. Use cases can be written in various levels of detail and formality, also in different formats [25]:

1. Brief – short one paragraph summary to provide overview of scope;
2. Casual – informal summary, possibly a few paragraphs; and
3. Fully Dressed – a structured and detailed description of possible action flows.

The main benefits of defining use cases are: engaging users in the RE process, providing developers with the users' perspective to the system and gaining understanding of the system under development. Use cases also usually serve as direct input for testing the software [44].

Table 4.2: Outline of the use cases, their intent and description.

NAME	INTENT	DESCRIPTION
UC1: Create ANN	Allows the user to create a neural network structure.	User defines the number of layers and the number of neurons in each layer. After which an ANN of the defined size gets created.
UC2: Load Training / Testing Data	Allows the user to load a dataset that can be used to train and test an ANN.	User selects the data patterns file, selects the data labels file and names the dataset. Then presses load button to finalise loading. After that the loaded data sets are available for training and testing the ANN.
UC3: Train ANN	Train the given neural network to perform a function based on the training data.	Select training data file, choose learning algorithm and start the training process. The training process finishes when the network converges or when the user manually stops the training process.
UC4: Test ANN	Present the trained neural network with data patterns to test the generalisation of the model.	Select the testing data and let the ANN classify them to see the output and classification accuracy. Testing stops when all test examples get classified or the user stop the process manually.
UC5: Save ANN	Present the trained neural network with data patterns to test the generalisation of the model.	Select save option from program menu, choose location, choose file name and press save. The user gets notified if the saving was successful.
UC6: Load ANN	Allows the user to load a pre-existing neural network model.	The user selects the load option from the menu, specifies the location and the name of the configuration file and loads it. After loading the ANN is available for training and testing.
UC7: View ANN Structure	Allows the user to see the structure and weight values of the ANN.	The user expands the tree view of the neural network that consists of list of layers. Then each layer can be expanded to see the neurons in the layer. Each neuron can be expanded to see the weights of the neuron.

Table 4.2 list the seven main use cases of the character recognition software under development. The use cases have been described in a brief style to provide a high level understanding of what the system needs to achieve. The use cases defined here will provide input to functional and non-functional requirements.

4.6 Use Case Diagram

Use case diagrams provide a visual summary of written use cases and how the system actors relate to the use cases [25] [45]. Use case diagrams are a good communication tool for describing the behaviour of the system and providing a high level context [43]. Use case diagrams should not be produced without relevant use case descriptions, because as stated before, they only provide the visual summary, but no actual detail of the action flow of information and what tasks are performed on them. Figure 4.2 presents the use case diagram for the character recognition system under development. The diagram has been derived from the use cases defined in Table 4.2.

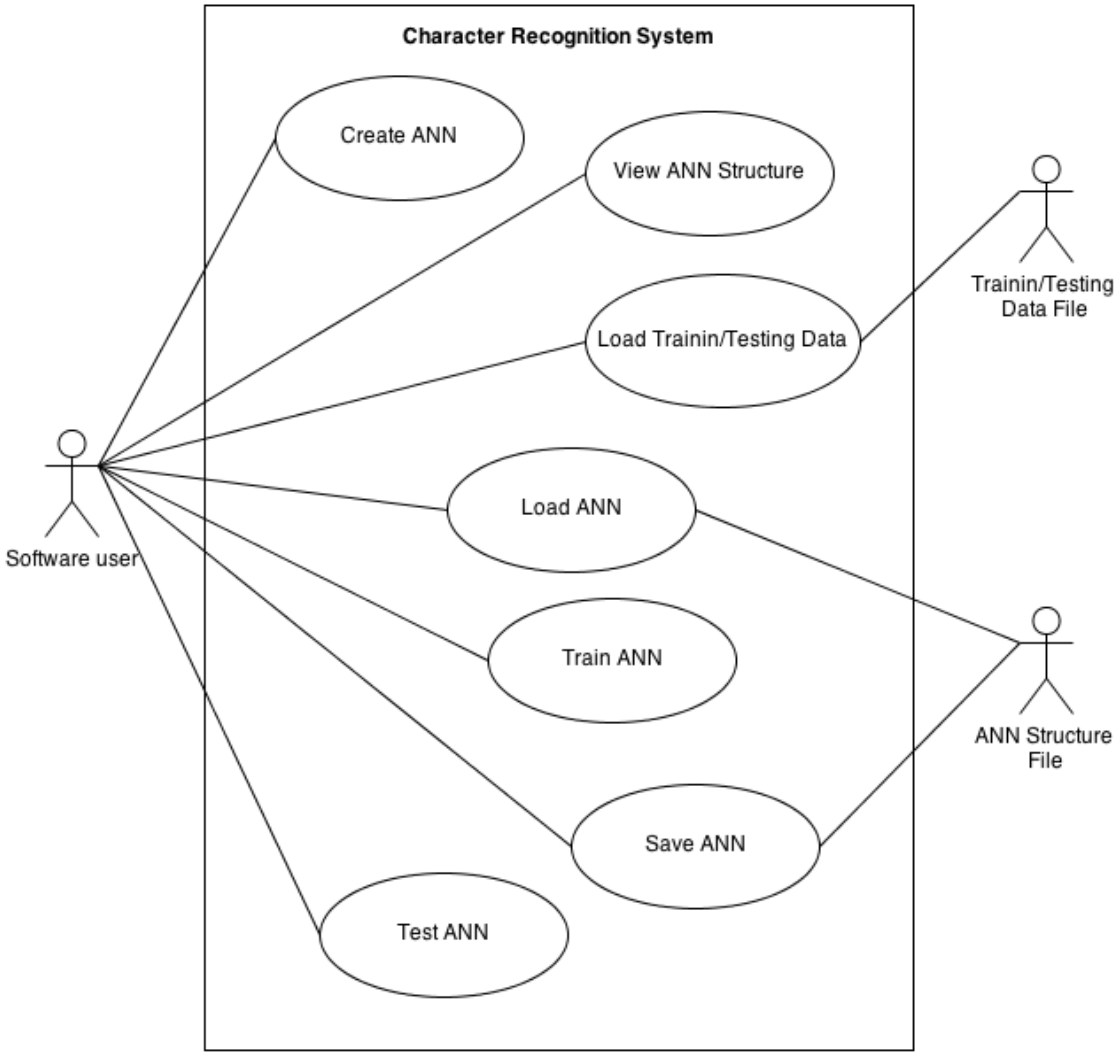

Figure 4.2: Use case diagram of the character recognition system.

4.7 Functional Requirements Table

Functional requirements describe the features that the system needs to perform; essentially they are the specifications of the system – the functionality that has to be delivered to the client [46]. They should be written from a technological platform independent view point and focus on describing the functionality itself. Functional requirements also serve as input to writing test cases, which is a requirement in the testing and validation phase (see §7) [47]. Table 4.3 lists the functional requirements for this project together with the assessment of their risk and priority.

Risk is defined in this project as an assessment by the author of possibility of failure to deliver the requirement due to design or implementation complexity. Low risk means that the author does not anticipate complications with delivering the feature, however medium and high risk indicate the anticipation of challenges during implementation. The priority shows the importance of the feature for delivering an acceptable system – therefore high priority features should be implemented first.

Table 4.3: Listing the functional requirements, their risk and priority.

ID	DESCRIPTION	RISK	PRIORITY
FR1	Means to create a fully connected ANN.	LOW	HIGH
FR2	Means to define the number of layers in an ANN.	LOW	HIGH
FR3	Means to define the number of neurons in an ANN layer.	LOW	HIGH
FR4	Means to load a data set for training and testing the ANN.	LOW	HIGH
FR5	Means to train an ANN using a supervised feedforward training algorithm.	LOW	HIGH
FR6	Means to test a trained ANN by presenting a data pattern.	LOW	HIGH
FR7	Means to view the classification errors of the ANN.	LOW	HIGH
FR8	Means to view the classification accuracy of the ANN.	LOW	HIGH
FR9	Means to stop ANN training before convergence.	LOW	HIGH
FR10	Means to stop ANN training before convergence.	LOW	HIGH
FR11	Means to persist the structure and properties of an ANN.	HIGH	MEDIUM
FR12	Means to display the weights of the neurons of the selected ANN.	MEDIUM	MEDIUM
FR13	Means to load a persisted ANN and its properties.	MEDIUM	MEDIUM
FR14	Means to view the structure of the ANN.	MEDIUM	MEDIUM

4.8 Non-functional Requirements Table

Non-functional requirements aid in specifying the qualitative attributes such as responsiveness, scalability, usability and availability of the system [47]. Non-functional requirements tend to be overlooked in the requirements engineering process, because they do not specify a particular functionality. They are important, however, because the clients' perspective depends heavily on the aforementioned qualities and can therefore influence the overall project success [37]. Table 4.4

outlines the non-functional requirements together with their risk and priority defined for this project. Risk and priority have been used in the same context as described in §4.7.

Table 4.4: Listing the non-functional requirements, their risk and priority.

ID	DESCRIPTION	RISK	PRIORITY
NFR1	The user needs to be notified if errors happen in the program.	LOW	HIGH
NFR2	User needs to be provided with details and a course of action expected if errors happen in the program.	LOW	HIGH
NFR3	The user needs to have an overview at all times about what the software is doing at a particular moment.	LOW	HIGH
NFR4	The program should not take more than 5 seconds to start up.	LOW	HIGH
NFR5	The program should be able to load data sets of at least 100 MB.	MEDIUM	MEDIUM
NFR6	The software must be able to scale to create ANNs of at least three layers and 200 neurons per layer.	MEDIUM	MEDIUM
NFR7	The GUI needs to be responsive at all times when the program is being used.	MEDIUM	MEDIUM

4.9 Closing Remarks

This chapter provided overview of the requirements engineering process theory and presented the relevant artefacts produced for this project. The next chapter will document the design phase of the project and outline the techniques used.

5 Design

5.1 Chapter Overview

The chapter provides an introduction to object oriented analysis, design methods and presents various artefacts of the design phase of the project such as domain model, class diagram and class responsibility collaborator (CRC) cards. It also discusses the choice of technology platform for this project and provides details of data persistence strategy for the application.

5.2 Object Oriented Analysis and Design

Object oriented analysis and design is a software engineering paradigm that facilitates modelling systems by defining objects or concepts and their interactions in the problem domain [25]. The analysis phase involves building a conceptual model of the problem domain and producing various artefacts such as use cases, domain models, interaction diagrams and user interface mockups[17] to address the requirements of the system in question [23] [25]. The de facto tool for building these models is the Unified Modelling Language (UML). UML is a visual general purpose modelling language for specifying and documenting system artefacts. The benefits of using UML include formality of structure, scalability and comprehensiveness [48]. The relevant design phase artefacts will be discussed next.

5.3 Domain Model

A domain model is a visual representation of the important real-life objects in the problem domain. The domain model or diagram shows the main concepts, their attributes and associations between them together with any constraints. Therefore, it also serves as a visual dictionary. Domain model does not represent software classes or objects, but quite often the resulting class names are inspired by the object names in the domain model to facilitate low representational gap between real life and software objects. A method for finding the domain objects is called noun phrase identification. It involves looking through documentation that has already been produced, such as context diagrams, use cases, requirements lists and algorithms; and highlighting noun phrases to consider them as candidate for a domain object. This kind of linguistic analysis is not perfect, because words can have ambiguous meanings, but nevertheless serves as a helpful technique [25].

Figure 5.1 presents the abstract domain model for the character recognition system under development; where ‘abstract’ in this content refers to outline or template – as really no methods or data structures are defined – this aligns to the “top down design” methodology. Noun phrase techniques were used to discover the domain objects. For instance, the description of use case “UC1: Create ANN” (cross-reference Table 4.2 in the requirements chapter §4.5) hinted domain

[17] Mockups are rough sketches or conceptual visual designs.

objects such as artificial neural network, layer and neuron[18]. Other domain objects were identified in a similar way, consulting the documented requirements. The central object in the domain model in Figure 5.1 is the artificial neural network (ANN). An ANN nominally consists of at least two layers – one input layer and one output layer; and each layer consists of one or more neurons. Each neuron has an activation function that it uses for turning input signals into output. An ANN is trained with a learning algorithm (cross-reference discussion about learning algorithms in the research chapter §3.6) using training data. Finally, the ANN can be tested using testing data.

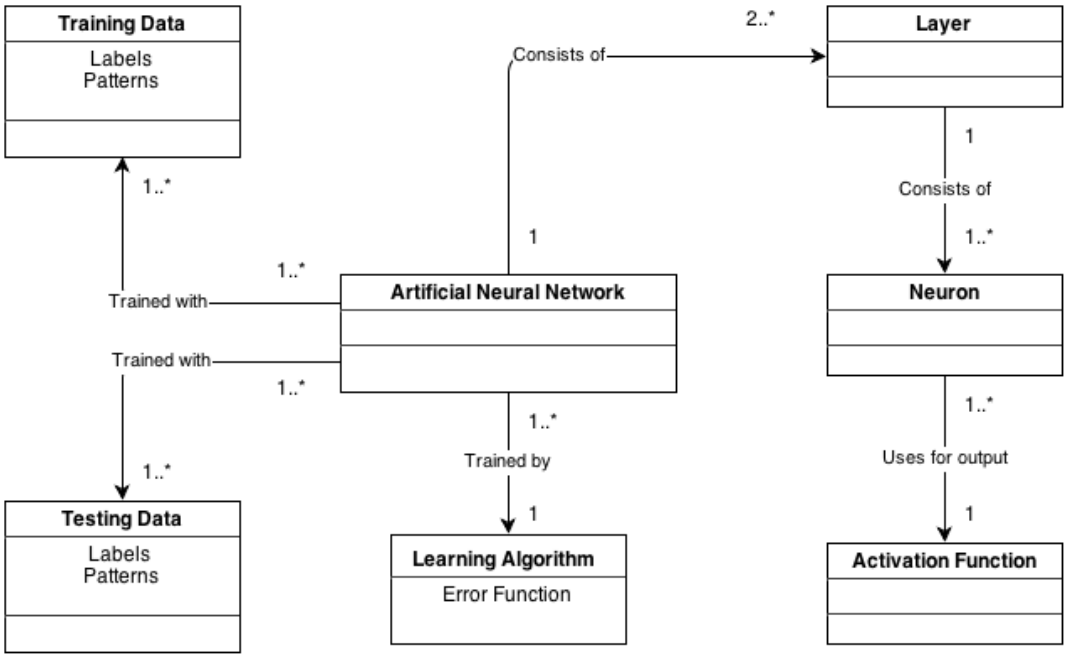

Figure 5.1: The top level domain model of the ANN system.

The next step in the design phase is to develop the conceptual domain model into a model that represents the actual software classes. The first stage involves creating CRC cards to define the classes, their responsibilities and collaborating classes. The CRC cards can then be used to build the class diagram that describes the software classes and their associations in detail. Both, CRC cards and the class diagram, will be inspected further after discussing software design patterns, which are a good aid for evolving a conceptual model into a model of software classes.

5.4 Software Design Patterns

Software design patterns are well-known named solutions for commonly occurring software design problems [23] [25]. The names of design patterns also facilitate effective communication and make them easier to remember, because solutions to complex problems are described by a short phrase. Two of the most useful resources for design patterns are the Gang of Four patterns [49] and GRASP[19] principles. A selection of software patterns, together with examples, that the author has used for designing the system classes for this project are explained next.

[18] These nouns occur in the description of the use case and can be considered as good candidates for domain objects.

[19] GRASP is an abbreviation of General Responsibility Assignment Software Patterns [25].

5.4.1 High Cohesion

High cohesion is a design principle that encourages defining objects with tightly related and limited amount of responsibilities. Following the principle of high cohesion facilitates the way to maintainable and reusable software classes [25]. For instance, a software class *Layer* representing a neural network layer has to keep track of neurons. One way of designing the *Layer* class would be to build all the functionalities of neurons into the class. However, this design would break the high cohesion guideline, because *Layer* would also be accountable for functionalities associated with neurons. Therefore, a good design would be to extract the neuron-related functionality to a class called *Neuron* and the rest to *Layer*.

5.4.2 Low Coupling

Low coupling is a design principle that suggests designing objects in a way that they would not depend on or need to have knowledge of many other objects. The meaning of 'many' can vary depending on software context, but it is advisable to aim for lower number of dependencies so that changes to other objects would not cause a chain reaction of changes in the system. Let us inspect the designing of classes for the neural network structure. The classes *NeuralNetwork*, *Layer* and *Neuron* have been identified as necessary software classes. One way of designing the interaction between these classes is to make the *NeuralNetwork* class responsible for keeping track of layers and neurons (Figure 5.2). An alternative approach would be to design the interaction so that *NeuralNetwork* only knows about *Layer* and *Layer* encapsulates the instances of *Neuron* (Figure 5.3). In this context, the latter design is a low coupling design, because every class needs to know about only one other class, whereas the former design made *NeuralNetwork* responsible for knowing about both *Layer* and *Neuron* classes; also known as a highly coupled design.

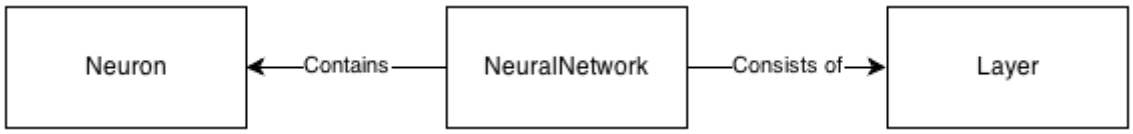

Figure 5.2: A high coupling design of an ANN structure.

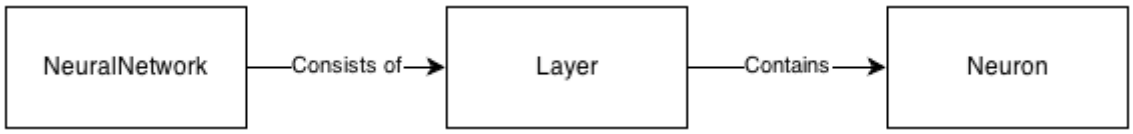

Figure 5.3: A low coupling design of a car repair.

5.4.3 Protected Variations

Protected variations is a design principle that advises creating a stable interface to points in system, where the implementation has a potential to cause changes [25]. Consider an example of activation functions in the current project. There are various activation functions that might be utilised, such as linear, threshold and sigmoid. Therefore, creating an interface for activation functions would comply with protected variations principle so that the implementations can be changed without affecting the rest of the system.

5.4.4 Polymorphism

Polymorphism is a design principle that offers a solution to problems such as how to create pluggable software objects and use a various types of objects with similar functionality [25]. One way of achieving it is to design an abstract object with abstract methods from which all the various

implementations can inherit from. For instance, training algorithms have some high-level features in common, but vary in the low-level implementation. Following the polymorphism principle, it would then be a good design to create an abstract *TrainingAlgorithm* class and let the concrete implementations such as *Perceptron* and *BackPropagation* inherit from it. Polymorphism is very much similar to the protected variations pattern, but approaches it from an inheritance point of view.

5.5 Class Responsibility Collaborator Cards

Class Responsibility Collaborator (CRC) cards are informal diagrams that display the class name together with its responsibilities and collaborators [25]. The responsibilities are high level functionalities that the class has to fulfil, whereas the collaborators are the other classes that the class needs to interact with in order to fulfil its responsibilities. CRC cards are a useful technique for designing classes from the domain model, because they concentrate on high level functionalities rather than concrete implementations [24].

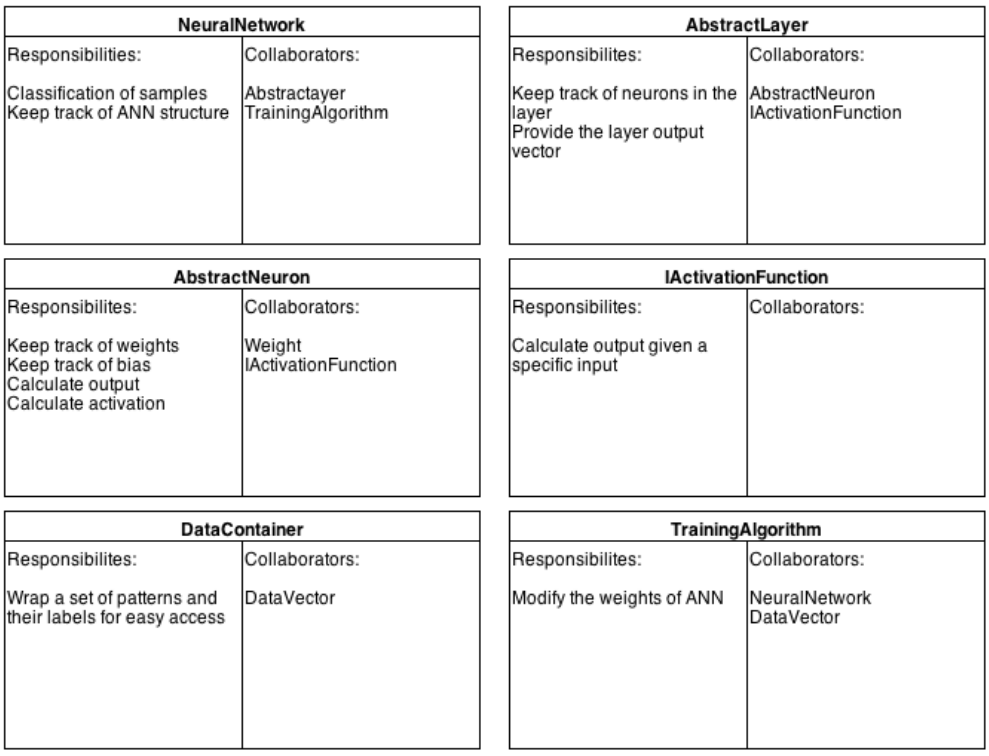

Figure 5.4: Selection of CRC cards.

Figure 5.4 presents a selection of CRC cards produced during the project for aiding class design. The names for classes have been inspired from the domain model outlined in Figure 5.1 to provide low representational gap between the real life and software objects. The central class for the whole project is *NeuralNetwork*, inspired by its neurobiological counterpart. Its main task is to classify data samples and serve as a wrapper to the whole ANN structure. It collaborates with implementations of *AbstractLayer* class to provide the classification functionality and with *TrainingAlgorithm* so that the ANN would be adjusted to classify examples of testing data. CRC cards are a good platform for building the class diagram, which is a more specific software class model. The class diagram developed for the project will be discussed next.

5.6 Class Diagram

Class diagram is a static model of the system architecture that presents the software classes and their interactions. Various other details such as attributes and methods can be included in the diagram [24].

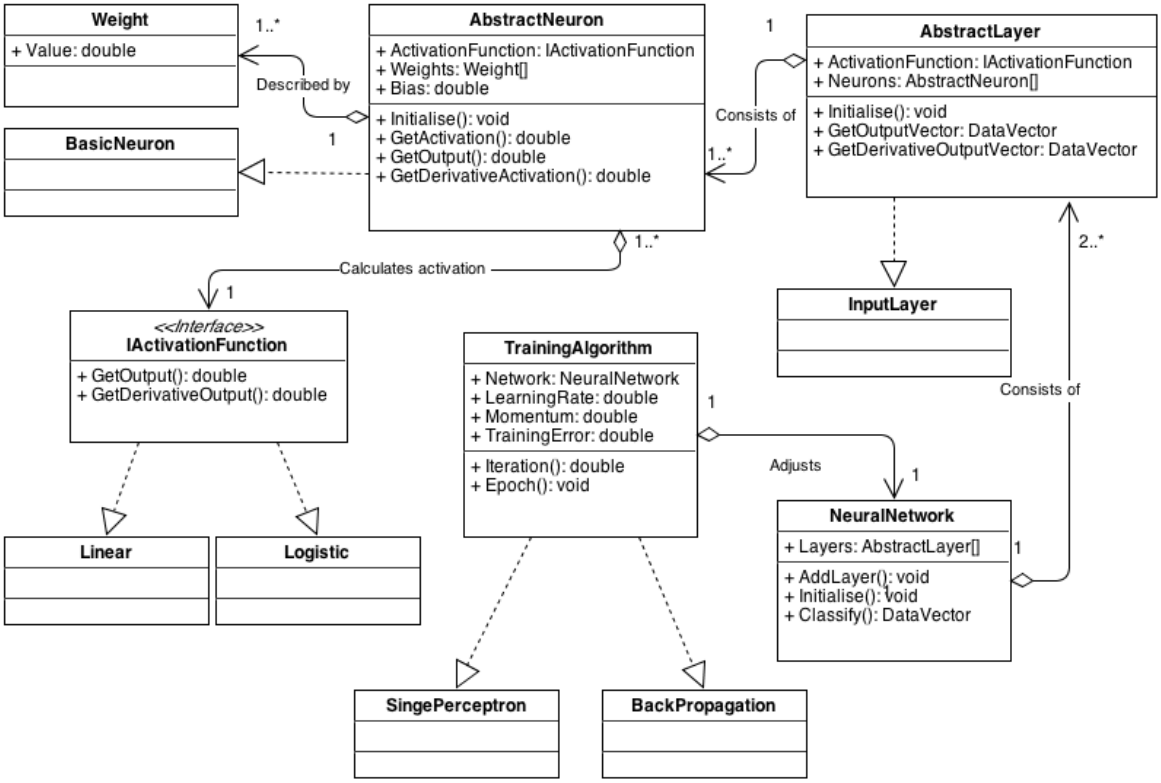

Figure 5.5: Class diagram outlining a selection of core classes.

Figure 5.5 shows the class diagram outlining a selection of core classes for the current project. The class names have been inspired by the objects from the domain model outlined in Figure 5.1. The class diagram shows some of the concrete implementations of abstract classes. For instance, *BasicNeuron* that inherits from *AbstractNeuron* class. There is also an implementation of *InputNeuron* class, which performs functions that are characteristic to the neurons only in the input layer, but has been left out of the diagram for keeping the diagram readable. The inheritance from *AbstractNeuron* is an example of leveraging the Polymorphism design pattern. The interface *IActivationFunction* has several implementations in the actual project, but only two – *Linear* and *Logistic* – have been outlined here. The protected variations design pattern has been used here to provide a common interface to various activation functions.

After the software classes have been designed, it is necessary to think about how the users will be interacting with the software. Therefore, the topic of user interface design will be discussed next.

5.7 User Interface Design

User interface (UI) is the part of the system that facilitates interaction between the software and the user. A good quality UI should be user friendly, attractive and easy to use to cater for user satisfaction [23] . The three golden rules that should be followed when designing a UI are [50]:

1. Place the user in control;
2. Reduce the user's memory load; and
3. Design for consistency in UI.

The first principle states that user should always be in control; meaning that the user can determine the course of action and understand at each moment what is happening in the system. The second principle refers to the fact that a good UI should not expect the user to memorise a lot to be able to effectively use the software. Instead, the UI should aim to help with focussing on the primary task that the user is trying to achieve [51]. Lastly, consistency enables the user to generalise knowledge about one part of the interface to the whole. For instance, a command in context of the UI should behave similarly in all contexts. In visual interfaces, it could mean that all the elements of the UI should have a similar style [23].

The two categories of user interfaces utilised during this project are textual (TUI) and graphical user interfaces (GUI). TUI, also referred to as command line, provides interaction with the user via text. TUI is nowadays considered an outdated approach because of its limitations, but it does serve as a quick way of interaction for developing product prototypes. All the early prototypes in this project will also leverage TUIs. GUIs are the modern approach to UI design and consist of graphical elements such as windows, menus and buttons that can be manipulated [52]. The final prototype of this project is expected to be manipulated via a GUI.

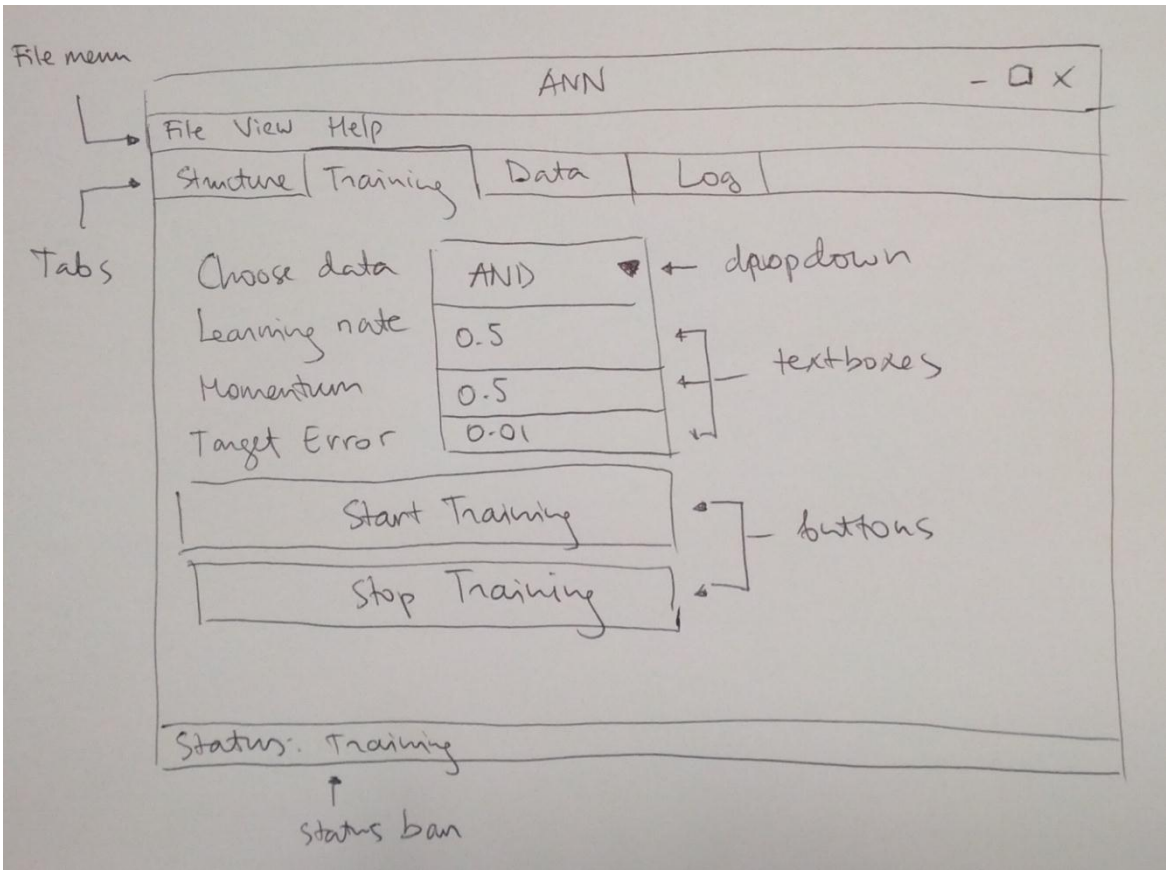

Figure 5.6: Early sketch for the software GUI.

Figure 5.6 shows a mockup of the planned GUI to the system. The training tab, which controls the selection of training process and its parameters, of the UI is focussed. Similar mockups were

produced for designing the rest of the tabs. A useful technique that the author has leveraged for designing user interaction flow with the system is called hierarchical task analysis.

5.8 Hierarchical Task Analysis

Hierarchical task analysis (HTA) is a technique for defining task division into subtasks. Tasks are fine grained into as many subtasks as deemed necessary in order to provide enough detail. The tasks higher in the hierarchy depict logical units of work, whereas the lower level subtasks express the atomic physical tasks that the user can execute. HTA facilitates understanding of a system on a suitable abstraction level, enables effective comparison of different systems and encourages good design. However, HTA is limited to fairly simple tasks as the notation does not scale well, making HTA not a suitable choice for large complex tasks. Furthermore, HTA does not enable modelling parallel or overlapping tasks [53]. HTA can be implemented as a numbered list or a graphical diagram of boxes and connectors. The graphical method is preferred in project, because the author deems the visual representation a better communication tool. Figure 5.7 outlines the HTAs of the core tasks for this project – train ANN, load data, load ANN and create ANN. The discussion of HTA concludes the review of design techniques. Next, technology platform for the project needs to be decided on.

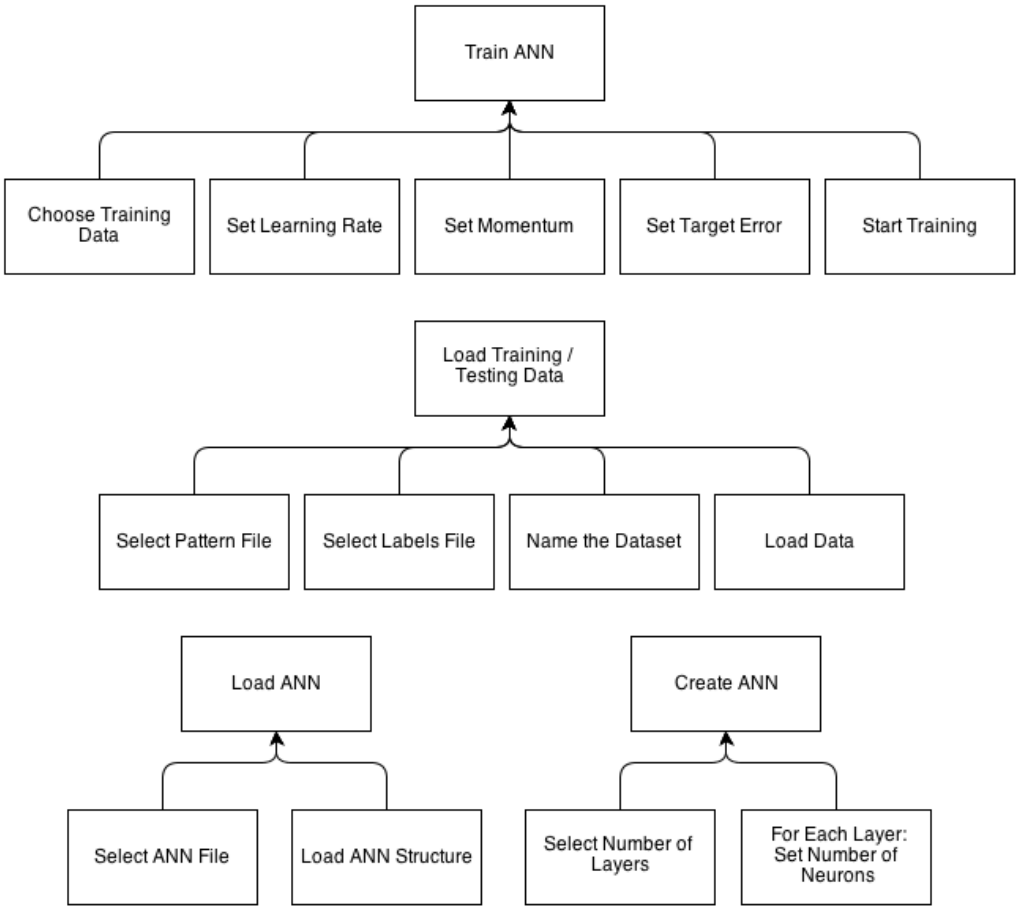

Figure 5.7: Selection of HTA diagrams for tasks in the OCR system.

5.9 Technology Platform

It is necessary to choose a technology platform and a programming language in order to implement the designs in software. The design approach taken in this project was object oriented; therefore the

programming language should also support object orientation in order to turn the designs into software effectively.

The author considered three popular object oriented programming languages: Java, C# and Python. Initially, the author used Python for quick prototyping of ANN algorithms, but decided to drop it after the first prototype, because of lack of convenient development tools and difficult code base management due to its script-like nature [54].

Java was considered as a suitable candidate for implementation for its multiplatform support, robustness and availability of advanced development tools. C# was preferred over Java, however, because C# provides a more advanced Windows Presentation Foundation (WPF) GUI framework than the one's available for Java – Swing and Abstract Window Toolkit (AWT) being the most popular ones [55] [56]. C#, contrary to Java, is not a multiplatform language and supports only Windows operating system, but platform dependence was not a primary goal for this project and therefore did not influence the decision of choosing the programming language. Therefore, the C# programming language with WPF framework was chosen for this project.

5.10 Data Persistence Strategy

Functional requirement FR4 refers to the need to store the training and testing data sets on the hard drive, so that they could be loaded to the application at convenience (see Table 4.3). The initial way of storing the data files was a simple comma-separated values (CSV) text file where each row represented a classification sample. The reason for choosing CSV as the file structure was the ease of creation and later parsing in the application. A sample consisted of $n + 1$ integers, where the first n integers were features and the last one the label of the sample. This structure was sufficient for early prototypes, where all labels were 1-dimensional vectors, however, later prototypes needed multi-dimensional vectors as labels and this approach was not suitable any more. The solution was to divide the samples and the labels into two separate files so that the rows in both files would match – one consisting of feature vectors and the other label vectors.

Functional requirements FR11 and FR13 place a demand to be able to store an ANN structure built and trained with the application. The process of storing an object's state to hard drive is called serialization and in C# there are two main formats that an object can be serialized to – Extensible Markup Language (XML) or binary format. The benefit of using XML is that the files are human readable. In order to use XML serialization it is necessary for each object that is being serialized to have a constructor with no parameters. This would have meant that the class structure needed to be modified just for object persistence. Therefore binary format was chosen for serialization, because it did not place any constraints on the class structure. The benefit of using binary format is also the fact that the file size is much smaller than when using XML format. In today's terms where disk space is cheap and in abundance, this characteristic did not however play a major role in decision making [57].

5.11 Closing Remarks

The chapter discussed object oriented design and analysis methods and presented the relevant artefacts produced for this project. Technology platform and persistence solutions for the application were also determined. The next chapter will document the implementation phase and provide detail about the prototypes built.

6 Implementation and Results

6.1 Chapter Overview

The chapter introduces the implementation strategy and describes briefly the tools that were utilised for this project. The majority of the chapter is devoted to documenting the prototypes that led to the final product. Each prototype will be documented in two parts:

1. Description - outlines the implemented features and algorithms; and
2. Results – conveys and analyses the outcome of the prototype.

6.2 Implementation Strategy

The project was implemented according to agile methodology that encourages iterative approach to software development (see §3.4.4). Five timeboxed iterations[20] were carried out; each of which resulted in a prototype. The prototypes were built by adding more advanced features to the main ANN library and building simple test projects on top of that library. This approach enabled code reuse throughout implementations, but also meant that changes to the main library could potentially have an effect on the previous test projects, which added complexity to the code base management process. Extensive testing (see §7) and good software design principles helped to overcome this problem. Implementation tools will be briefly discussed next after which a summary of each prototype will be provided.

6.3 Implementation Tools

6.3.1 Integrated Development Environment

Integrated Development Environment (IDE) is a software development tool that aids with tasks such as source code management, debugging, testing, code compilation and build management [58]. Visual Studio Professional 2012 was used for this project as it is the latest version of the most popular C# IDE. The IDE provides excellent debugging tools, which were extensively utilised during the development process.

6.3.2 Version Control

Version control is the process of recording changes to the project source code so that any changes could be reversed. The simplest way of achieving this is to make copies of the project folder to another location, but it does not provide features such as listing all the changes and reversing single files [59]. Git [59], which is distributed version control software, was used for maintaining the code base for this project. The author did not have any experience with Git before and used this project as an opportunity to learn about it.

[20] Iteration length was chosen to be 3 weeks in order to avoid lengthy periods of code being in development stage.

6.4 Prototype 1: The Perceptron

6.4.1 Description

The perceptron is the simplest neural network training algorithm that facilitates solving linearly separable problems. Table 6.1 describes the perceptron algorithm in detail. The prototype implemented functional requirements FR1 - FR6 and used the command line for user interaction.

Table 6.1: Low-level algorithm of perceptron training.

	STEPS OF THE ALGORITHM
1:	$\boldsymbol{function}\ PerceptronTrainingIteration(ANN, v, t)$
2:	$y \leftarrow Classify(ANN, v)$
3:	$error \leftarrow AbsoluteDifference(y, t)$
4:	$\boldsymbol{if}\ error > 0\ \boldsymbol{then}$
5:	$\Delta w \leftarrow error \times learningRate$
6:	$\boldsymbol{for\ each}\ weight\ w\ \boldsymbol{do}$
7:	$w_i \leftarrow w_i + \Delta w \times v_i$
8:	$\boldsymbol{end\ for}$
9:	$bias \leftarrow bias + \Delta w \times (-1)$
10:	$\boldsymbol{end\ if}$
11:	$\boldsymbol{return}\ error$
12:	$\boldsymbol{end\ function}$

6.4.2 Results

Figure 6.1 shows utilising the perceptron algorithm to perform AND function on a 2-1 topology [21] neural network. It can be seen that higher learning rate in this case leads to slower convergence. The performance in general is as expected and the network converges without any problems. Notice that convergence happens when the error value is 0, because threshold activation function with discrete output of 1 or 0 is used. Later prototypes that use continuous activation functions show a smooth error curve that usually converges at an above zero value.

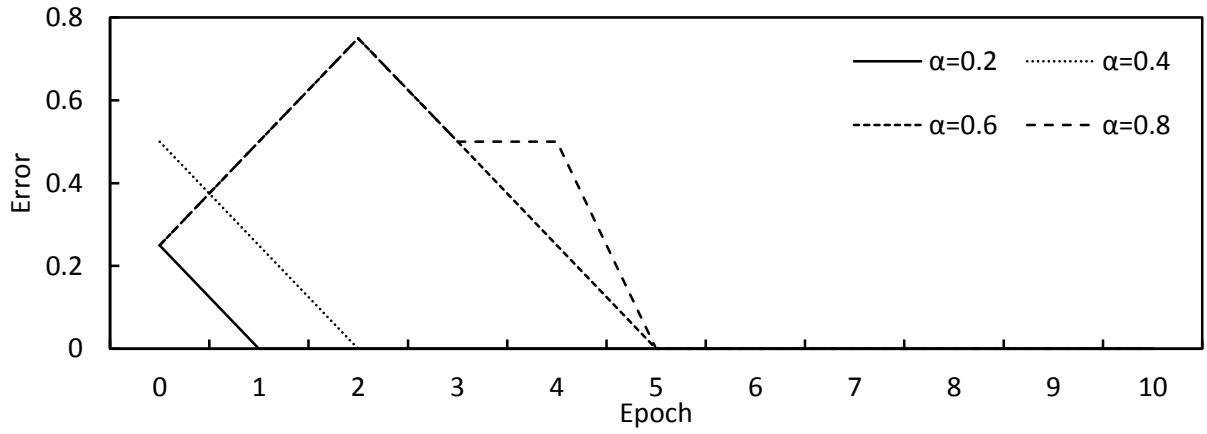

Figure 6.1: Diagram of perceptron training to perform AND function.

[21] The 2-1 topology refers to an ANN structure of 2 neurons in the input and 1 neuron in the output layer.

6.5 Prototype 2: The Delta Rule (Linear Activation)

6.5.1 Description

Delta rule is a more advanced neural network training algorithm that uses gradient descent to minimise the error function. Table 6.2 outlines the delta rule training in detail. The prototype implemented functional requirements FR1 – FR7 and used a command line for user interaction.

Table 6.2: Low-level algorithm of delta rule training using a linear activation function.

STEPS OF THE ALGORITHM	
1:	$\boldsymbol{function}\ LinearDeltaRuleTrainingIteration(ANN, v, t)$
2:	$y \leftarrow Classify(ANN, v)$
3:	$iterationError \leftarrow 0$
4:	$\boldsymbol{for\ each}\ neuron\ n\ \boldsymbol{do}$
5:	$error \leftarrow t_n - activation_n$
6:	$\Delta w \leftarrow error \times learningRate$
7:	$\boldsymbol{for\ each}\ weight\ w\ in\ neuron\ n\ \boldsymbol{do}$
8:	$w_i \leftarrow w_i + \Delta w \times v_i$
9:	$\boldsymbol{end\ for}$
10:	$bias_n \leftarrow bias_n + \Delta w \times (-1)$
11:	$iterationError \leftarrow iterationError + error^2 / 2$
12:	$\boldsymbol{end\ for}$
13:	$\boldsymbol{return}\ iterationError$
14:	$\boldsymbol{end\ function}$

6.5.2 Results

Figure 6.2 describes training a 2-1 topology neural network to perform OR function. Learning rates $\propto \in [0.2, 0.4, 0.6, 0.8]$ have been used to show different error values for convergence. The graph converges at a relatively high error value when $\alpha = 0.8$, but it is caused only because of the peculiarity of the algorithm that uses the difference between the activation and target class as the error value. The classification performance at the high error level is nevertheless 100%.

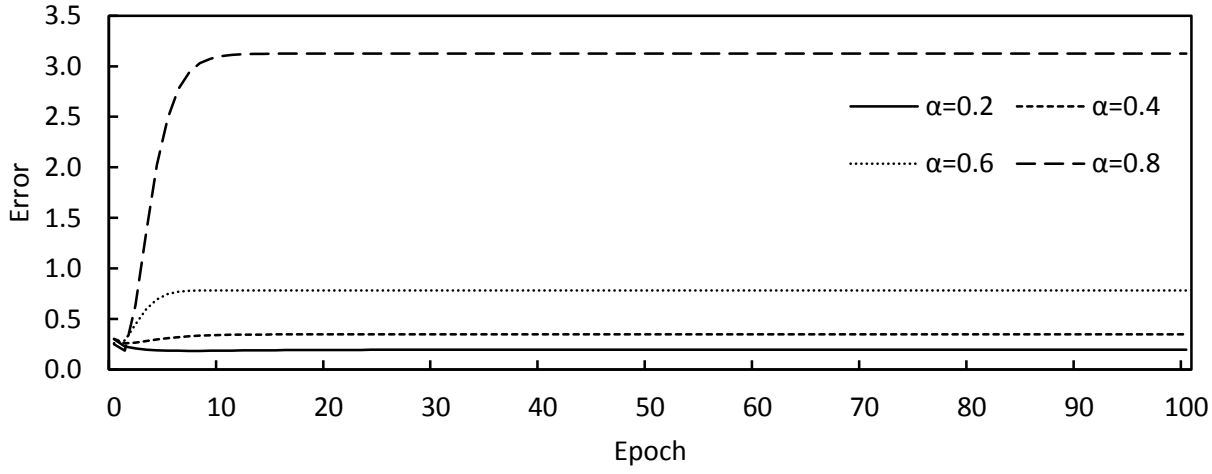

Figure 6.2: Diagram of delta rule training to perform OR function.

6.6 Prototype 3: The Delta Rule (Sigmoid Activation)

6.6.1 Description

Prototype builds on prototype 2 by enabling usage of non-linear sigmoid activation function. Table 6.3 outlines the low-level abstraction of the delta rule training algorithm. The prototype implemented functional requirements FR1 – FR8 and used a command line for user interaction.

Table 6.3: Low-level algorithm of delta rule training using a sigmoid activation function.

STEPS OF THE ALGORITHM	
1:	$\boldsymbol{function}\ SigmoidDeltaRuleTrainingIteration(ANN, v, t)$
2:	$y \leftarrow Classify(ANN, v)$
3:	$iterationError \leftarrow 0$
4:	$\boldsymbol{for\ each}\ neuron\ n\ \boldsymbol{do}$
5:	$error \leftarrow t_n - y_n$
6:	$\Delta w \leftarrow error \times learningRate \times Derivative\ (activation_n)$
7:	$\boldsymbol{for\ each}\ weight\ w\ in\ neuron\ n\ \boldsymbol{do}$
8:	$w_i \leftarrow w_i + \Delta w \times v_i$
9:	$\boldsymbol{end\ for}$
10:	$bias_n \leftarrow bias_n + \Delta w \times (-1)$
11:	$iterationError \leftarrow iterationError + error^2 / 2$
12:	$\boldsymbol{end\ for}$
13:	$\boldsymbol{return}\ iterationError$
14:	$\boldsymbol{end\ function}$

6.6.2 Results

Figure 6.3 describes training a 2-1 topology neural network to perform NAND (not AND) function. The sensible learning rates $\alpha \in [0.2, 0.4, 0.6]$ converge relatively quickly. As a contrast, an extremely high learning rate $\alpha = 30$ is also demonstrated. The error converges at a high level and the network does not learn to perform the NAND function.

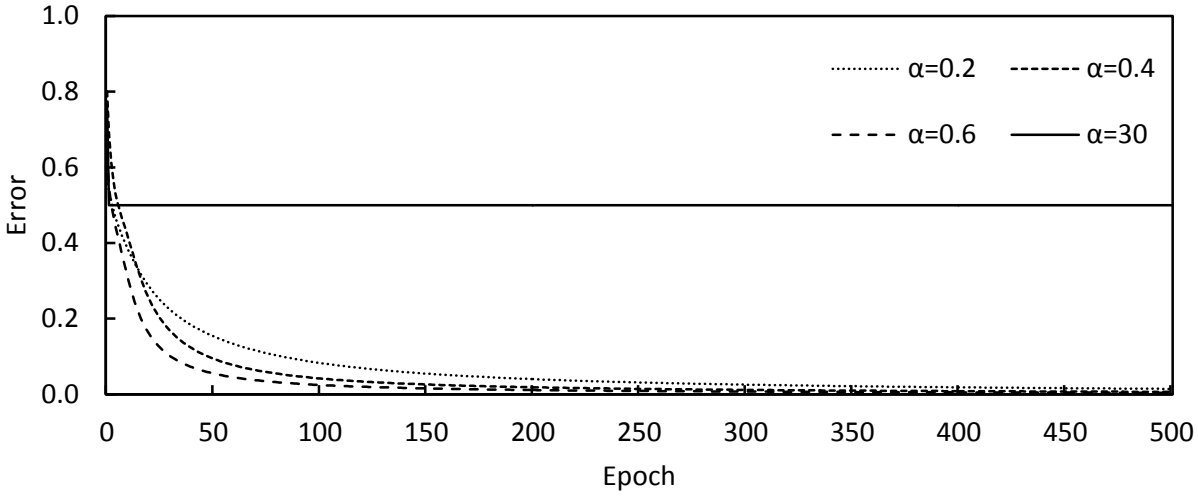

Figure 6.3: Diagram of delta rule training to perform NAND function.

6.7 Prototype 4: Backpropagation

6.7.1 Description

Backpropagation is the most advanced neural network training algorithm implemented in this project and will be used also in the final prototype for character recognition. Table 6.4 outlines the algorithm for backpropagation training in detail. The algorithm consists of two phases – *forward pass* (step 2), where the pattern is classified using the network; and *backward pass* (steps 3-23), where the error deltas are being propagated to output and hidden layers. The prototype implemented functional requirements FR1 – FR8 and used a command line for user interaction.

Table 6.4: Low-level algorithm of backpropagation training.

	STEPS OF THE ALGORITHM
1:	$\boldsymbol{function}\ BackPropagationTrainingIteration(ANN, v, t)$
2:	$y \leftarrow Classify(ANN, v)$
3:	$iterationError \leftarrow 0$
4:	$\boldsymbol{for\ each}\ neuron\ n\ in\ output\ layer\ \boldsymbol{do}$
5:	$error \leftarrow t_n - y_n$
6:	$\Delta w \leftarrow error \times learningRate \times Derivative\ (activation_n)$
7:	$\boldsymbol{for\ each}\ weight\ w\ \boldsymbol{do}$
8:	$w_i \leftarrow w_i + \Delta w \times v_i$
9:	$\boldsymbol{end\ for}$
10:	$bias_n \leftarrow bias_n + \Delta w \times (-1)$
11:	$iterationError \leftarrow iterationError + error^2 / 2$
12:	$\boldsymbol{end\ for}$
13:	$\boldsymbol{for\ each}\ layer\ l\ in\ hidden\ layers\ \boldsymbol{do}$
14:	$\boldsymbol{for\ each}\ neuron\ n\ in\ layer\ l\ \boldsymbol{do}$
15:	$errorBlame \leftarrow GetOutputErrorBlame(n)$
16:	$derivative \leftarrow Derivative\ (activation_n)$
17:	$\Delta w \leftarrow errorBlame \times learningRate \times derivative$
18:	$\boldsymbol{for\ each}\ weight\ w\ in\ neuron\ n$
19:	$w_i \leftarrow w_i + \Delta w \times v_i$
20:	$\boldsymbol{end\ for}$
21:	$bias_n \leftarrow bias_n + \Delta w \times (-1)$
22:	$\boldsymbol{end\ for}$
23:	$\boldsymbol{end\ for}$
24:	$\boldsymbol{return}\ iterationError$
25:	$\boldsymbol{end\ function}$

6.7.2 Results

Figure 6.4 describes training a 2-2-1 topology[22] neural network to perform XOR function. XOR is a non-linearly separable problem and therefore needs a hidden layer in the network together with a sigmoid activation function. The training converges in a smooth manner when using a low learning rate $\alpha = 0.2$, but learning rate $\alpha = 0.8$ causes significant oscillation (between A and B) on the error plane before error function conversion. Therefore, $\alpha = 0.8$ is probably the highest learning rate that

[22] The 2-2-1 topology refers to 2 neurons in the input, 2 neurons in the hidden and 1 neuron in the output layer.

can be set for solving XOR using the given parameters. Setting a high learning rate $\alpha = 5$ indeed shows that the network does not converge to a low error value and does not learn the XOR mapping.

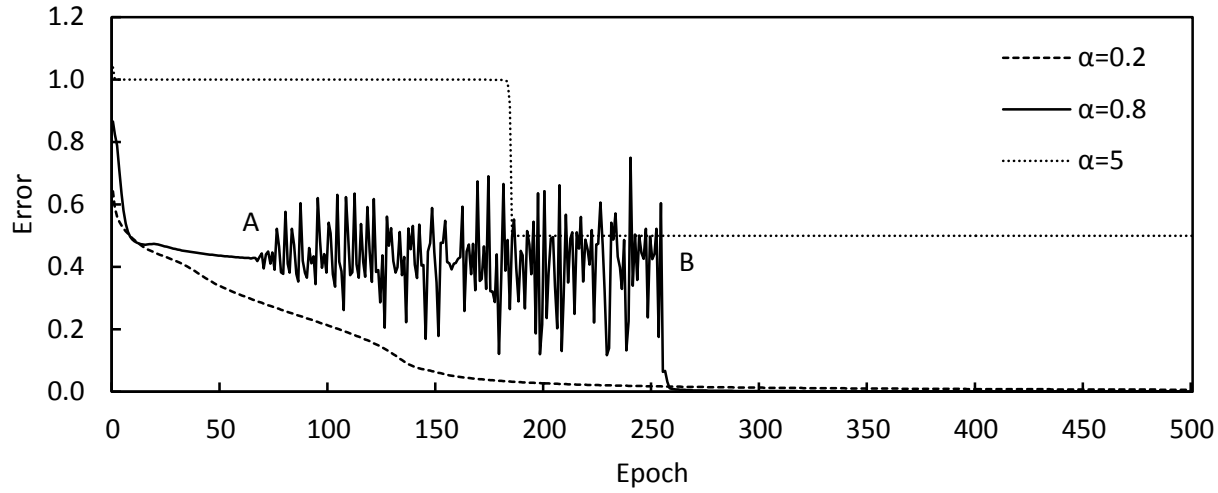

Figure 6.4: Diagram of backpropagation training to perform XOR function.

6.8 Prototype 5: Graphical User Interface

6.8.1 Description

The development work for the final prototype was mostly concerned with building an effective and efficient GUI for the existing neural network library built throughout the previous four iterations. The prototype implemented functional requirements FR1 – FR14 and non-functional requirements NFR 1 – NFR7. Figure 6.5 shows the final GUI design that implements the tasks from HTA (see §5.8). It has been divided into four tabs:

1. Structure (refer to A) – enables viewing the ANN structure and neuron weights;
2. Training and Testing (refer to B) – enables controlling and setting the parameters training (refer to E - H) and testing (refer to K) processes;
3. Loaded Data (refer to C) – enables viewing the loaded data sets; and
4. Log (refer to D) – provides the log for events in the GUI.

Non-functional requirement NFR7 prescribes that the GUI needs to be responsive at all times. To address the requirement, the training and testing processes run on separate threads from the GUI. In principle, this means that the ANN that is being trained can be tested simultaneously without stopping the training process. The results would not be accurate however, because the training process keeps adjusting the ANN's weights, but it could give indication of its accuracy on the testing data. Functional requirements FR9 and FR10 have been addressed in the design of the GUI by exposing buttons that enable stopping the training (refer to J) and testing (refer to M) process at any time, after which the process can be continued again. These buttons are disabled when the software is idle, but become active during training or testing process respectively. For further detail, see Appendix B that provides a comprehensive walkthrough of the final prototype.

The backpropagation training algorithm was improved in this prototype with an addition of momentum term (see §3.6.4 for mathematical definition) for enhancing the training time.

Presenting the training patterns in a random order using the Fisher–Yates shuffle algorithm was implemented in order to prevent gradient descent from getting stuck in local minima instead of the global minima [60]. Table 10.1 in Appendix A outlines the shuffling algorithm in detail.

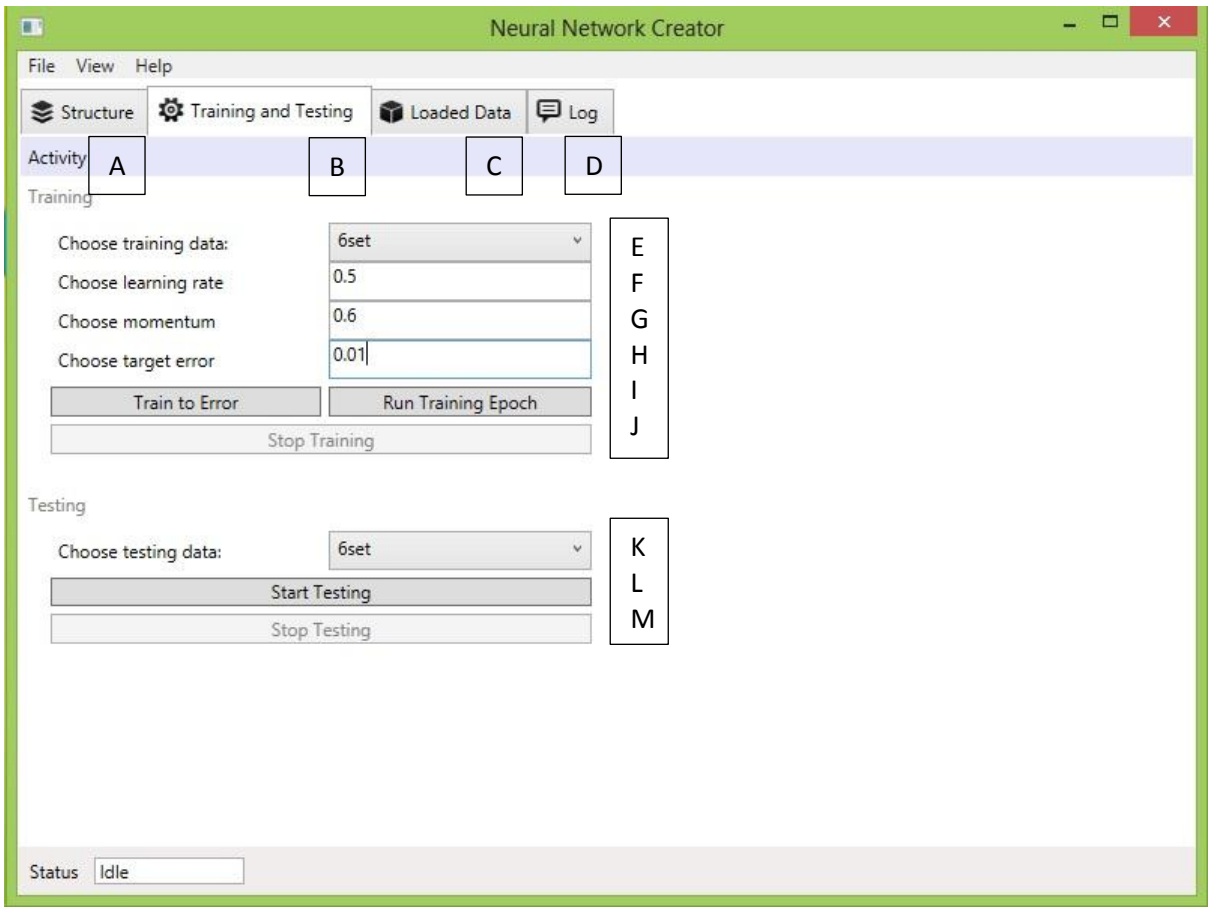

Figure 6.5: Screenshot of the training and testing tab in the GUI.

6.8.2 Results

Table 6.5 outlines a selection of the test results for performing character recognition using a normalised character set, where each character is represented by a 16x8 retina [61]. The training and testing accuracies listed in Table 6.5 have been summarised in Figure 6.6. The columns of Table 6.5 denote the following details of the testing process:

1. Classes – number of different characters classes in test with range in brackets;
2. Training samples per class – number of samples for each character used for training;
3. Testing samples per class - number of samples for each character used for testing;
4. Topology – number of neurons in each layer (input-hidden-output);
5. α – learning rate (refer to F in Figure 6.5);
6. λ – momentum value (refer to G in Figure 6.5);
7. Epochs – the number of training cycles;
8. $r_{training}$ – accuracy of the ANN on the training character set; and
9. $r_{testing}$ – accuracy of the ANN on the testing character set;

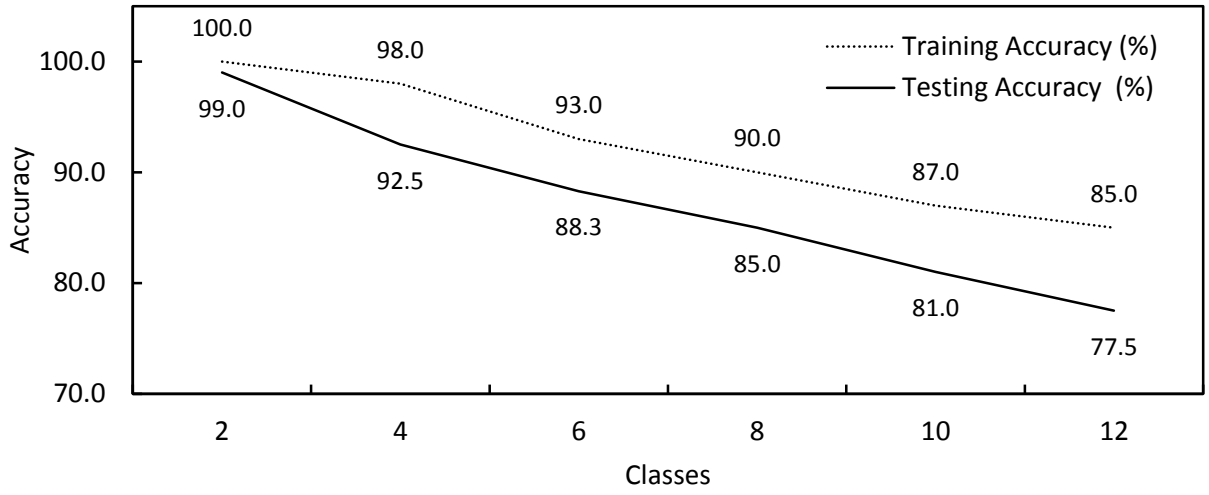

Figure 6.6: Training and testing accuracy of the ANN vs. number of character classes.

Figure 6.6 shows that the accuracy of the ANN decreases in line with increasing the number of character classes. This observation can be explained by the fact that introduction of similar character classes, such as 'b' and 'd' or 'i' and 'j', can cause an increase in misclassification.

Table 6.5: Results of character recognition tests.

CLASSES	TRAINING SAMPLES PER CLASS	TESTING SAMPLES PER CLASS	TOPOLOGY	α	λ	EPOCHS	$r_{training}$	$r_{testing}$
2 [a-b]	50	10	128-4-2	0.1	0.7	100	100.0%	99.0%
4 [a-d]	50	10	128-10-4	0.1	0.7	500	98.0%	92.5%
6 [a-f]	50	10	128-16-6	0.1	0.7	500	93.0%	88.3%
8 [a-h]	50	10	128-24-8	0.05	0.5	1000	90.0%	85.0%
10 [a-j]	50	10	128-28-10	0.05	0.5	3000	87.0%	81.0%
12 [a-l]	50	10	128-36-12	0.02	0.3	5000	85.0%	77.5%

Testing results of up to twelve classes has been provided in Table 6.5, because the introduction of classes after twelve caused the accuracy of the ANN to fall below 75%, which the author considers the threshold value for classification performance that is acceptable. The deteriorating performance can be caused by the large variation of character examples and also the fact that the characters samples are not always centred in the image retina. The author believes that processing the feature vectors to strip the white space around the characters would improve the performance significantly.

The choice of network topology had a material influence on the ANN performance and therefore *sensitivity analysis*[23] was carried out to find the optimal range of number of neurons for the hidden layer. The analysis showed that the number of hidden layers should be at least double the number of output layer neurons. The upper constraint was found to be 42 neurons for the hidden layer, after which training time increased unacceptably and the performance on test sets deteriorated due to *over-fitting*[24] the ANN to the training set resulting in poor *generalisation*. The value of learning rate and momentum was also found necessary to decrease with introducing more character classes, because the training algorithm needs to take 'smaller steps' on the complex error plane in order to avoid oscillation.

6.9 Closing Remarks

The chapter provided overview of the implementation phase of the project outlining the descriptions and results of each of the five prototypes built. The next chapter will discuss the topic of software testing and present a selection of testing results for this project.

[23] Sensitivity analysis is the process of assessing the impact of a variable when other variables remain constant [14].

[24] Over-fitting occurs when the ANN model is too much fine-grained to the training data, but performs poorly on unseen data [14].

7 Testing and Evaluation

7.1 Chapter Overview

The chapter introduces the topic of software testing and discusses various methods for functional and non-functional testing, the levels of testing and automating the testing process. A reference to the test case table of automated test for the project will be provided.

7.2 Introduction to Software Testing

Software testing is the activity of evaluating a capability of a software system and assessing if it meets the specified requirements [62]. In a wider perspective, software testing can also serve a multitude of other aims – building confidence in robustness of the system, discovering errors, measuring performance and verifying the correctness of the system [62]. Therefore, it can be said that testing is mostly concerned with ensuring software quality[25].

While in more traditional software development methodologies testing took place mainly after the development had finished, the more contemporary methodologies such as agile, encourage testing as a parallel process to development (see §3.4.1 and §3.4.4). Some approaches, for instance Test Driven Development (TDD), even prescribe writing tests before writing any system code [63]. TDD can be practiced separately as a software development methodology or incorporated as a practice into other methodologies. Agile, the methodology chosen for this project (see §3.4.4), suggests TDD as a good practice, but does not enforce it [25]. Some of well-known testing practices will be discussed next.

7.3 Functional Testing

Functional testing is concerned with comparing the specifications that describe *'what the system should do'* to the actual activities that the system performs. Predetermined inputs are used to compare expected outputs to the actual ones produced by the system. The biggest problem with this approach is that increasing complexity of requirements causes exponential growth of the possible test cases and covering all possible system states becomes impossible. There are a number of techniques that address this issue, such as partitioning – dividing input space into sections in which values can be considered to be equal. This way only one sample of each partition needs to be tested [64]. The following paragraphs introduce methods of functional testing.

7.3.1 Black-box Testing

Black-box testing is a process whereby software is tested by utilising an interface; without knowing the inner workings. The testing is carried out purely by determining whether the system output corresponds to the expected output or not [23].

[25] Quality – in this software context it refers to meeting the requirements set [62].

7.3.2 White-box Testing

White-box testing makes use of inner properties of software objects to verify compliance with requirements [23]. The software tester chooses paths through the system structure to test with the aim of covering as much as possible of the code base; and possible paths through the code. The amount of code base covered by tests is called *test coverage* [65].

7.3.3 Regression Testing

Regression testing is any kind of software testing with the aim of discovering new bugs[26]; it is carried out after amending the code base. It is common to run only a subset of all tests, because full testing of a large system can take a very long time to run [66]. There are two ways how regression testing can be used. One way is to rerun a test after fixing an error that was exposed by it. The second one concerns running all tests after making a fix in order to make sure that the change did not break the rest of the system [67].

7.4 Non-functional Testing

Non-functional testing is concerned with the domains of testing that do not assess a specific user action on the system. The requirements tested are related to scalability, usability and performance of the system [23]. The following paragraphs introduce methods of non-functional testing.

7.4.1 Scalability Testing

Scalability testing can be used to assess the ability of the system to deal with large amounts of data or requests. There might be requirements that the system needs to support a certain number simultaneous users or has to be able to process files up to a certain size. These requirements can then be tested to those predefined benchmarks. The process is also referred to as load or stress testing [23].

7.4.2 Usability Testing

Software usability testing is a testing method whereby the software product is tested on the end-users to evaluate the ease of use. It involves observing users while they interact with the product in order to assess various indicators such as efficiency, accuracy and emotional response. The ways of assessing these indicators include interviews, questionnaires and specialist software aids [68].

7.4.3 Performance Testing

Performance testing is a testing routine that is utilised to determine system responsiveness under certain conditions. The aim of performance testing is to ensure end-user satisfaction while using the system. The key indicators of performance testing are divided into two groups: service oriented (availability, response time); and efficiency oriented (throughput, utilisation). These indicators should be taken into account during requirements gathering process so that the system can be tested against these benchmarks later during the project [69].

[26] Bug refers to an error in software functionality.

7.5 Levels of Testing

7.5.1 Unit Testing

Unit testing is the lowest level of testing and involves evaluating the correctness of methods or subroutines in a software program [23]. Test cases need to be devised first, in order to be able to cover all possible states, operations and attributes of a software object [70]. A well-written unit test should adhere to the following guidelines [71]:

1. It should be automated and repeatable;
2. It should be easy to implement;
3. Once it's written, it should remain for future use;
4. Anyone should be able to run it;
5. It should run at the push of a button; and
6. It should run quickly.

The benefits of unit tests are that they highlight problems early on in the code development cycle so that they get fixed before higher levels of testing are undertaken [71]. Unit tests can also serve as basis for integration testing, which will be discussed next.

7.5.2 Integration Testing

Integration testing takes a wider grasp on the system than unit testing. It tests how various modules function together [23]. It is important to execute integration tests from early on in the project in order to avoid major errors; which may occur between different modules of the software system [72]. Integration testing means testing interfaces, which might be internal or external to the software system. There are various types of integration testing and different sources take different approaches. Some of the most common ones are briefly explained next.

Big bang integration is an approach, whereby all modules of the software are being integrated in one go and tested together. It is a cost-effective method, but feasible only on small systems, because it is extremely difficult to attribute the error to any specific component [23]. Big bang integration can be suitable also to larger systems if majority of the components are already mature and stable [73].

Bottom-up integration testing is a process of testing the subcomponents separately first and then integrating them into bigger subsystems alongside testing until all the modules have been proved to work together. The main advantage of this approach is that testing of subsystems can take place simultaneously and no test stubs need to be written to mock actual software functionality [23].

Top-down integration tests the main functionalities of the system first and then lowers the level of abstraction to test the subroutines [73]. It is assumed that all the interfaces have been defined beforehand so that test stubs can be written for the missing subroutines. That is also one of the drawbacks of the approach – fixing interfaces might not be possible early in the project and writing testing stubs requires extra effort [23] .

7.5.3 System Testing

System testing is carried out to evaluate the software system against the requirements specification document [23]. It is run against the complete integrated system, which means that this could also include the hardware component in addition to the software [73]. During system testing, both functional and non-functional tests get executed. System testing helps to find out about defects that do not appear when testing just a single subsystem [74].

7.5.4 Acceptance Testing

Acceptance testing is carried out by the system's end users to evaluate if the system meets the requirements and is ready for use in a production environment [75]. Acceptance testing is often carried out executing predefined scripts or getting users to perform the use cases that were defined in the requirements gathering process [73].

7.6 Test Automation

Automated tests are programs that are written for testing other programs and therefore planning for maintenance and expansion of automated tests is essential [76]. Automated tests are usually executed after making changes to the code base and before merging changes from a code branch to the main repository. Writing of the tests takes place depending on the software development methodology either before any actual code is written (test driven development) or after writing the code (waterfall approach).

The modern approach is to write tests before any actual code, then see the tests failing and after that fulfil the test requirements by writing all the necessary code so that all the tests pass. Tests can be automated on various levels such as unit, integration and system testing levels [77]. The main benefits of utilising automated tests is that once they have been written it is very little effort to run them again. Also, they highlight the errors straight away if new changes break existing code base. On the other hand, there is a cost for planning, writing and maintaining the tests [76].

Table 7.1 outlines a selection of all 23 automated tests written for this project. All the tests had to pass in order to regard the project to be successfully delivered. The columns of Table 7.1 have been defined as follows:

1. ID – the unique test identifier;
2. Description – explanation of the test case;
3. Input – description of the input data;
4. Expected output – the expected outcome given the inputs; and
5. Result – indicates if the system passed the test.

Tests T1 - T7 validate the architectural constraints of the neural network model. Exceptions are expected for erroneous input, such as negative number of neurons (see T4) or neuron weights (see T7). Tests T8 – T13 validate the activation functions used in the project. For instance, tests T8 – T10 partition the possible input state space to the bipolar sigmoid activation function; which maps the input values into the range $[-1, 1]$, into three value partitions and validates each. These tests were built throughout the project and got executed after each major amendment to the code base.

Table 7.1: A selection of automated tests for the project.

ID	DESCRIPTION	INPUT	EXPECTED OUTPUT	RESULT
T1	Test that the defined number of layers get actually created for the ANN.	An ANN with 2 layers.	Layer count of 2.	PASS
T2	Add more layers to the ANN than defined at creation time.	An ANN with defined layer capacity of 2, add 3 layers.	Exception (2 layers expected).	PASS
T3	Test that neurons do get properly added to a layer of ANN.	An ANN layer that gets 3 neurons added.	Neuron count of 3.	PASS
T4	Try creating a layer with a negative number of neurons.	Neuron count of -1.	Exception (count of greater than 90 expected).	PASS
T5	Check that the input layer's output vector is of correct size.	Datavector of length 2.	Output vector of length 2.	PASS
T6	Provide a hidden layer with an input vector of unexpected (wrong) length.	Datavector of length 2.	Exception (Datavector of length 1 expected).	PASS
T7	Create a neuron with a negative number of weights.	Weight count of -1.	Exception (weight count of greater than 0 expected).	PASS
T8	Test the correctness of the bipolar sigmoid activation function.	100	1	PASS
T9	Test the correctness of the bipolar sigmoid activation function.	-100	-1	PASS
T10	Test the correctness of the bipolar sigmoid activation function.	0	0	PASS
T11	Test the correctness of the threshold activation function.	10	1	PASS
T12	Test the correctness of the threshold activation function.	-5	0	PASS
T13	Test the correctness of the threshold activation function.	0	1	PASS

7.7 Closing Remarks

The chapter provided an overview of software testing theory and referenced the results of automated testing for the project. The next chapter will conclude the report and discuss topics such as project evaluation, skills attained and project future.

8 Conclusion

8.1 Chapter Overview

The final chapter of the report summarises the work undertaken, discusses the project management approaches taken and evaluates the results that were achieved. An outline of the new skills attained and possible future developments of the project will also be provided.

8.2 Project Summary

The aim of the project was to develop a software package that would perform character recognition utilising artificial neural networks. There were five prototypes built according to the agile SDLC which was adopted for this project (see §3.4.4). The approach of starting with a simple neural network model and evolving it through prototypes (see §6.4 - §6.8) to a complex solution proved to be the right approach, as a gradual up take of the neural network domain knowledge (by the author) took place in parallel with producing working software. The final product provided a GUI for creating, training and testing multilayer artificial neural networks using the backpropagation training algorithm together with a momentum term for speeding up the training process.

In addition to functional features, concepts like scalability, usability and performance were kept in mind while designing the product in order to provide good user experience while interacting with the software. This was ensured by getting feedback from technical and non-technical users, which indicated the author to simplify the GUI to the neural network library so that the user would not be overwhelmed with detail.

8.3 Project Evaluation

The project met all the specifications in use cases, functional and non-functional requirements that were defined throughout the project. Ideally, the neural network model could classify between 26 characters of the English alphabet, however effective results for this project were achieved for up to 12 different character classes at time. The main causes for this are the training and testing samples which had not been sufficiently pre-processed and also lack of computing facilities to run extensive tests for determining the most optimal neural network structure. In order to improve the classification performance, it might be a good idea to test other publicly available OCR datasets and also carry out further sensitivity analysis on various neural network setups. This would require usage of dedicated high-end hardware, which the project did not have access to. All in all, in spite of the limitations, the project was successfully completed with all the necessary features delivered.

8.4 Skills Attained

Extensive research was conducted for the project and a complex implementation were all challenges that the author faced at the start of the project; which meant a steep learning curve had to be ascended in order for the author to attain all the necessary skills and domain knowledge. The following lessons were learned during the project:

1. Deep knowledge of the theory of artificial neural networks and the associated training algorithms was gained (see §2.4, §2.5 and §6.4 - §6.7). One of the key initial phases, to be aware of, is selection of the correct parameter set and the neural network structure; it is this process that needs a lot of experimentation and undertaking of sensitivity analysing;
2. The Agile SDLC methodology together with Kanban project management provided an effective framework for ensuring that the project was delivered in a timely fashion (see §3.4.8). It was found that timeboxed iterations resulted in working software which had a positive psychological effect on the developer;
3. Accurate modelling of the requirements (see §4) and design artefacts (see §5) had a material impact on the implementation phase. The process encouraged the developer to see software from the user's perspective and make design decisions without being overwhelmed with the implementation details;
4. Effective user interface design empowers the user and provides just enough information at a time. This was achieved with a tabbed GUI, in this project, that divides the controls into logical units without confusing the user with too many options at one time (see §11.1 for GUI walkthrough);
5. Fluency in C# and XAML programming languages together with Git version control was achieved; and
6. Writing the current report taught the author how to compose and format an extensive technical document.

8.5 Further Development

The author believes that parallelisation of the backpropagation training algorithm would be a great improvement that could be addressed in future developments of the project. Parallelisation would enable to make use of multiple cores on a single machine or even distribute the computation across many physical machines [78]. The simplest way of achieving parallelisation is pattern partitioning - dividing the training data set into batches that are run on separate threads. The threads need to be synchronised after each iteration and the summed weight deltas from the threads applied to the neural network. More complex strategies could distribute copies of the structure to a set of processors [79].

This project implemented a feedforward neural network; however other types of neural networks exist. For instance, Hopfield networks are a kind of recurrent neural network, which try to match an input pattern to an already stored pattern [14]. The neurons in a Hopfield network are connected in a loop (recurrently) and the process of computing the outputs finishes when the output pattern of the network evolves to the one that generated it through a series of state space transition [80]. Hopfield networks are especially useful for restoring the original pattern from a noisy or damaged input pattern.

Other than fundamental developments to the neural network architecture, aspects of user experience could be enhanced by visualising the learning process. Furthermore, porting the application to other platforms or implementation in a cross-platform programming language such as Python or Java would make the application accessible to a wider audience.

8.6 Closing Remarks

This chapter concluded the report on the author's work on the 3rd year project at the University of Manchester, outlined the skills learned, evaluated the outcome and discussed the future prospects of the project. The listing of bibliography and appendixes with further details on the implementation results follow the chapter.

9 Bibliography

[1] M. Cheriet, N. Kharma, C.-L. Liu and C. Suen, Character Recognition Systems: A Guide for Students and Practitioners, Wiley-Interscience, 2007.

[2] Microsoft, “About Microsoft Office Document Imaging,” Microsoft, [Online]. Available: http://office.microsoft.com/en-gb/help/about-microsoft-office-document-imaging-HP001077103.aspx. [Accessed 6 4 2013].

[3] Microsoft, “Exploring ink in OneNote,” Microsoft, [Online]. Available: http://office.microsoft.com/en-gb/onenote-help/exploring-ink-in-onenote-HA001078456.aspx. [Accessed 6 4 2013].

[4] R. Smith, “An Overview of the Tesseract OCR Engine,” in *Document Analysis and Recognition, 2007. ICDAR 2007. Ninth International Conference on Document Analysis and Recognition*, 2007.

[5] LEAD Technologies, Inc., “LEADTOOLS OCR SDK,” LEAD Technologies, Inc., [Online]. Available: http://www.leadtools.com/sdk/ocr/default.htm. [Accessed 6 4 2013].

[6] S. Theodoridis and K. Koutroumbas, Pattern Recognition, 4th Edition, Academic Press, 2008.

[7] R. H. Davis, J. Lyal and l, “Recognition of handwritten characters — a review,” *Image and Vision Computing,* vol. 4, no. 4, p. 208–218, 1986.

[8] V. K. Govindan and A. Shivaprasad, “ Character Recognition - a Review,” *Pattern Recognition,* vol. 23, no. 7, pp. 671-683, 1990.

[9] W. Bieniecki, S. Grabowski and W. Rozenberg, “Image Preprocessing for Improving OCR Accuracy,” *Perspective Technologies and Methods in MEMS Design, 2007. MEMSTECH 2007. International Conference on,* pp. 75,80, 23-26 , 2007.

[10] M. J, “An Overview of Character Recognition Methods,” *Pattern Recognition,* vol. 19, no. 6, pp. 425-430, 1986.

[11] H. F. Schantz, The history of OCR, optical character recognition, Recognition Technologies Users Association, 1982.

[12] E. E. F. d'Albe, “On a Type-Reading Optophone,” *Proceedings A,* vol. 90, pp. 373-375, 1914.

[13] S. Mori, C. Suen and K. Yamamoto, “Historical review of OCR research and development,” *Proceedings of the IEEE,* vol. 80, no. 7, pp. 1029 - 1058, 1992.

[14] K. Gurney, An Introduction to Neural Networks, CRC Press, 1997.

[15] R. Begg, J. Kamruzzaman and R. Sarker, Neural Networks in Healthcare, IGI Global, 2006.

[16] D. Rios, “Neuro AI - Intelligent systems and Neural Networks,” Learn artificial neural networks , [Online]. Available: http://www.learnartificialneuralnetworks.com. [Accessed 15 4 2013].

[17] AstroML, “Neural Network Diagram,” AstroML: Machine Learning and Data Mining for Astronomy, [Online]. Available: http://astroml.github.io/book_figures/appendix/fig_neural_network.html. [Accessed 6 4 2013].

[18] M. Mohri, A. Rostamizadeh and A. Talwalkar, Foundations of Machine Learning, The MIT Press, 2012.

[19] OECD, “Frascati Manual: Proposed Standard Practice for Surveys on Research and Experimental Development, 6th edition,” OECD Publishing, 2002.

[20] L. Blaxter, C. Hughes and M. Tight, How to Research, Open University Press, 2010.

[21] J. Weyers and K. McMillan, How to Write Dissertations & Project Reports (Smarter Study Skills), Prentice Hall, 2011.

[22] IEEE Computer Society, Guide to the Software Engineering Body of Knowledge, IEEE Computer Society Press, 2005.

[23] J. Mishra and A. Mohanty, Software Engineering, Pearson Education India, 2011.

[24] A. Dennis, B. H. Wixom and D. Tegarden, Systems Analysis and Design with UML, 4th Edition, John Wiley & Sons, 2012.

[25] C. Larman, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, Third Edition, Prentice Hall, 2004.

[26] J. Shore and S. Warden, The Art of Agile Development, O'Reilly Media, 2007.

[27] K. Beck, Extreme Programming Explained: Embrace Change, Second Edition, Addison-Wesley Professional, 2004.

[28] P. McBreen, Questioning Extreme Programming, Addison-Wesley Professional, 2002.

[29] D. J. Anderson and D. G. Reinertsen, Kanban: Successful Evolutionary Change for Your Technology Business, 2010: Blue Hole Press.

[30] S. Harris and J. Ross, Beginning Algorithms, Wrox, 2005.

[31] R. Sedgewick and K. Wayne, Algorithms, Fourth Edition, Addison-Wesley Professional, 2011.

[32] W. McAllister, Data Structures and Algorithms Using Java, Jones & Bartlett Learning, 2010.

[33] A. Sharma, Data Structures Using C, Pearson Education India, 2011.

[34] P. H. Dave and H. B. Dave, Design and Analysis of Algorithms, 2007: Pearson Education India.

[35] M. T. Goodrich, R. Tamassia and D. M. Mount, Data Structures and Algorithms in C++, Second Edition, John Wiley & Sons, 2011.

[36] P. Rosin and F. Fierens, “Improving neural network generalisation,” *Geoscience and Remote Sensing Symposium, 1995. IGARSS '95. 'Quantitative Remote Sensing for Science and Applications', International ,* vol. 2, pp. 1255 - 1257, 1995.

[37] K. Pohl and C. Rupp, Requirements Engineering Fundamentals, Rocky Nook, 2011.

[38] K. E. Wiegers, Software Requirements, Second Edition, Microsoft Press, 2003.

[39] S. Withall, Software Requirement Patterns, Microsoft Press, 2007.

[40] N. Rozanski, Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, Second Edition, Addison-Wesley Professional, 2011.

[41] M. Godinez, E. Hechler, K. Koenig, S. Lockwood, M. Oberhofer and M. Schroeck, The Art of Enterprise Information Architecture: A Systems-Based Approach for Unlocking Business Insight, IBM Press, 2010.

[42] F. Armour and G. Miller, Advanced Use Case Modeling: Software Systems, Addison-Wesley Professional, 2000.

[43] K. Bittner and I. Spence, Use Case Modeling, Addison-Wesley Professional, 2002.

[44] D. Leffingwell and D. Widrig, Managing Software Requirements: A Use Case Approach, Second Edition, Addison-Wesley Professional, 2003.

[45] C. Britton and J. Doake, A Student Guide to Object-Oriented Development, Butterworth-Heinemann, 2004.

[46] T. Lowdermilk, User-Centered Design, O'Reilly Media, Inc., 2013.

[47] S. Robertson and J. Robertson, Mastering the Requirements Process: Getting Requirements Right, Third Edition, Addison-Wesley Professional, 2012.

[48] R. Miles and K. Hamilton, Learning UML 2.0, O'Reilly Media, Inc., 2006.

[49] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-

Oriented Software, Addison-Wesley Professional, 1994.

[50] F. Tsui and O. Karam, Essentials of Software Engineering, 2nd Edition, Jones & Bartlett Learning, 2010.

[51] J. Johnson, Designing with the Mind in Mind: Simple Guide to Understanding User Interface Design Rules, Morgan Kaufmann, 2010.

[52] E. Butow, User Interface Design for Mere Mortals, Addison-Wesley Professional, 2007.

[53] Y. Rogers, H. Sharp and J. Preece, Interaction Design: Beyond Human-Computer Interaction, 3rd Edition, John Wiley & Sons, 2011.

[54] Safari Content Team, Python Bibliography, O'Reilly Media, Inc., 2011.

[55] P. Sarang, Java Programming, McGraw-Hill, 2012.

[56] A. Nathan, WPF 4 Unleashed, Sams, 2010.

[57] E. Gunnerson and N. Wienholt, A Programmer's Guide to C# 5.0, Fourth Edition, Apress, 2012.

[58] B. Johnson, Professional Visual Studio 2012, Wrox, 2012.

[59] S. Chacon, Pro Git, Apress, 2009.

[60] R. Durstenfeld, “Algorithm 235: Random permutation,” *Communications of the ACM,* vol. 7, no. 7, p. 420, 1964.

[61] B. Taskar, “OCR dataset,” University of Pennsylvania, [Online]. Available: http://www.seas.upenn.edu/~taskar/ocr/. [Accessed 20 4 2013].

[62] B. Hetzel, The Complete Guide to Software Testing, 2nd Ed, John Wiley & Sons, 1993.

[63] K. Beck, Test Driven Development: By Example, Addison-Wesley Professional, 2002.

[64] J. Pan, “Software Testing,” Carnegie Mellon University, 1999. [Online]. Available: http://www.ece.cmu.edu/~koopman/des_s99/sw_testing/. [Accessed 5 4 2013].

[65] A. Spillner, T. Linz and H. Schaefer, Software Testing Foundations, Third Edition, Rocky Nook, 2011.

[66] G. J. Myers, C. Sandler, T. Badgett and T. M. Thomas, The Art of Software Testing, Second Edition, John Wiley & Sons, 2004.

[67] C. Kaner, J. Falk and H. Q. Nguyen, Testing Computer Software, 2nd Edition, John Wiley & Sons, 1999.

[68] C. M. Barnum, Usability Testing Essentials, Morgan Kaufmann, 2010.

[69] I. Molyneaux, The Art of Application Performance Testing, O'Reilly Media, Inc., 2009.

[70] I. Sommerville, Software Engineering (9th Edition), Addison-Wesley, 2010.

[71] R. Osherove, The Art of Unit Testing: with Examples in .NET, Manning Publications, 2009.

[72] J. Bender and J. McWherter, Professional Test-Driven Development with C#, Wrox, 2011.

[73] S. Desikan and G. Ramesh, Software Testing: Principles and Practices, Pearson Education India, 2006.

[74] R. Schlesinger, Developing Real World Software, Jones & Bartlett Learning, 2010.

[75] R. Cimperman, UAT Defined: A Guide to Practical User Acceptance Testing, Addison-Wesley Professional, 2006.

[76] R. Black, Advanced Software Testing - Vol. 1, Rocky Nook, 2008.

[77] D. J. Mosley and B. A. Posey, Just Enough Software Test Automation, Prentice Hall, 2002.

[78] M. L. M. Pethick, P. Werstein and Z. Huang, “Parallelization of a Backpropagation Neural Network on a Cluster Computer,” University of Otago, 2003. [Online]. Available: http://www.cs.otago.ac.nz/postgrads/mliddle/pubs/pethick2003.pdf. [Accessed 22 4 2013].

[79] M. Mohamad, M. Y. M. Saman and M. S. Hitam, “Parallel Training for Back Propagation in Character Recognition,” University Malaysia of Terengganu, 2012. [Online]. Available: http://www.taibahu.edu.sa/iccit/allICCITpapers/pdf/p89-mohamad.pdf. [Accessed 22 4 2013].

[80] D. Graupe, Principles of Artificial Neural Networks, World Scientific, 1997.

[81] S. Kulkarni, Machine Learning Algorithms for Problem Solving in Computational Applications, IGI Global, 2012.

[82] B. Liskov and S. Zilles, “Programming with abstract data types,” *Proceedings of the ACM SIGPLAN symposium on Very high level languages,* pp. 50-59, 1974.

[83] R. Connor, “Abstract data type,” in *Encyclopedia of Computer Scienc*, John Wiley and Sons, 2000, pp. 1-5.

[84] J. R. Hass, M. D. Weir and G. B. Thomas Jr., University Calculus, Early Transcendentals 2nd Edition, Pearson, 2011.

[85] H. van Vliet, Software Engineering: Principles and Practice, John Wiley & Sons, 2008.

10 Appendix A

10.1 Fisher–Yates Shuffle Algorithm

Table 10.1 outlines the algorithm for Fisher-Yates array shuffle, $RandomNumberInRange(n)$ refers to a random number generating function that returns a random number in range $[0..n]$ [60] .

Table 10.1: The steps for the Fisher-Yates array shuffling algorithm.

	STEPS OF THE ALGORITHM
1:	$\boldsymbol{function}\ Shuffle(array)$
2:	$n \leftarrow Length(array)$
3:	$error \leftarrow AbsoluteDifference(y, t)$
4:	$\boldsymbol{while}\ n > 1\ \boldsymbol{do}$
5:	$n \leftarrow n - 1$
6:	$k \leftarrow RandomNumberInRange(n + 1)$
7:	$value \leftarrow array[k]$
8:	$array[k] = array[n]$
9:	$array[n] = value$
10:	$\boldsymbol{end\ while}$
11:	$\boldsymbol{end\ function}$

11 Appendix B

11.1 Walkthrough of Prototype 5

The screenshots outlined in Figure 11.1 to Figure 11.9 provide a walkthrough of the application. Figure 11.1 is the first screen that the user sees when the application is started. The user is provided with to options – to create a new network or load an existing one. If the user chooses to create a new network then they are asked to define the number of layers for the ANN (Figure 11.2) and the number of layers for each layer (Figure 11.3) after which a confirmation is displayed (Figure 11.4). The user is then directed to the *'Structure'* tab, where the ANN structure and properties can be viewed (Figure 11.5). It is then necessary to load a data set to perform training and testing on the ANN (Figure 11.6). The user needs to locate the input patterns, the labels and name the new data set (Figure 11.7). The training and testing processes can then be initiated in the *'Training and Testing'* tab (Figure 11.8). All the important events such as training epochs get logged into the *'Log'* tab (Figure 11.9).

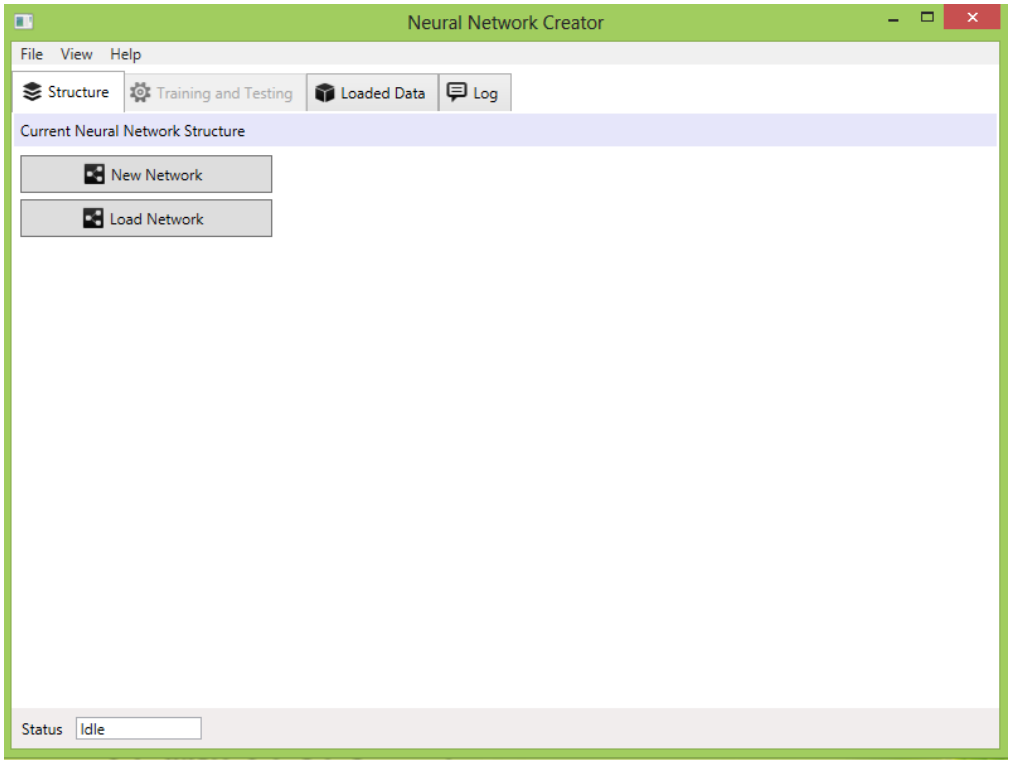

Figure 11.1: The first screen that the user sees after starting the application.

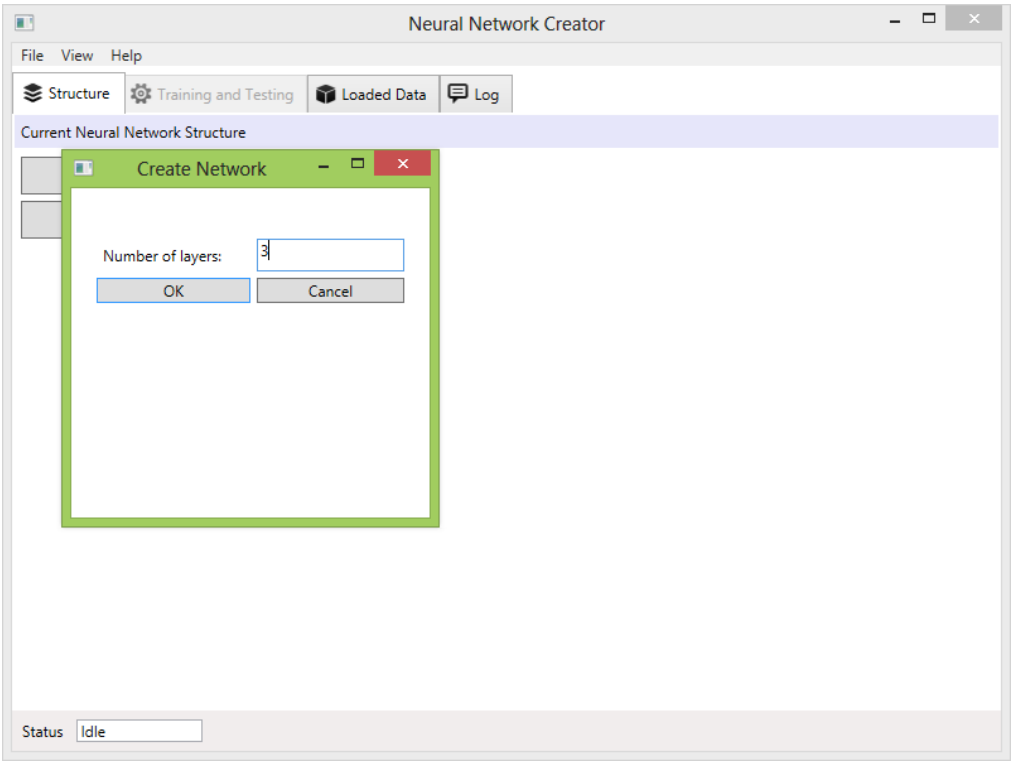

Figure 11.2: The prompt after clicking "New Network" asking to choose the number of layers.

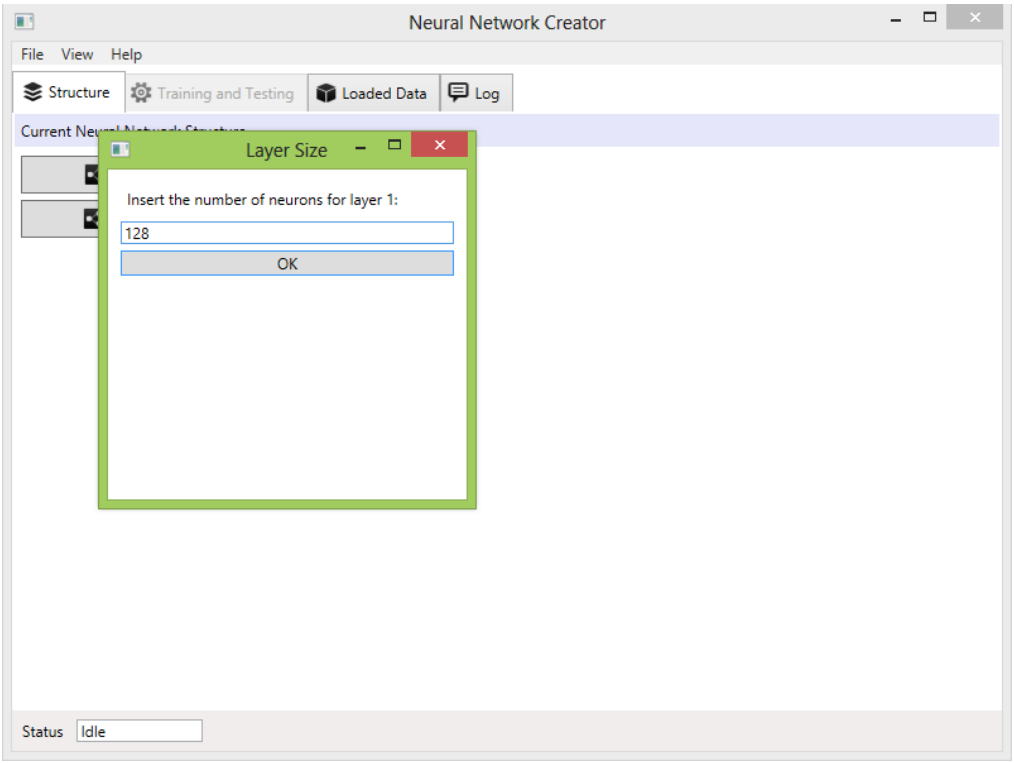

Figure 11.3: This prompt to choose the number of neurons is displayed for each layer.

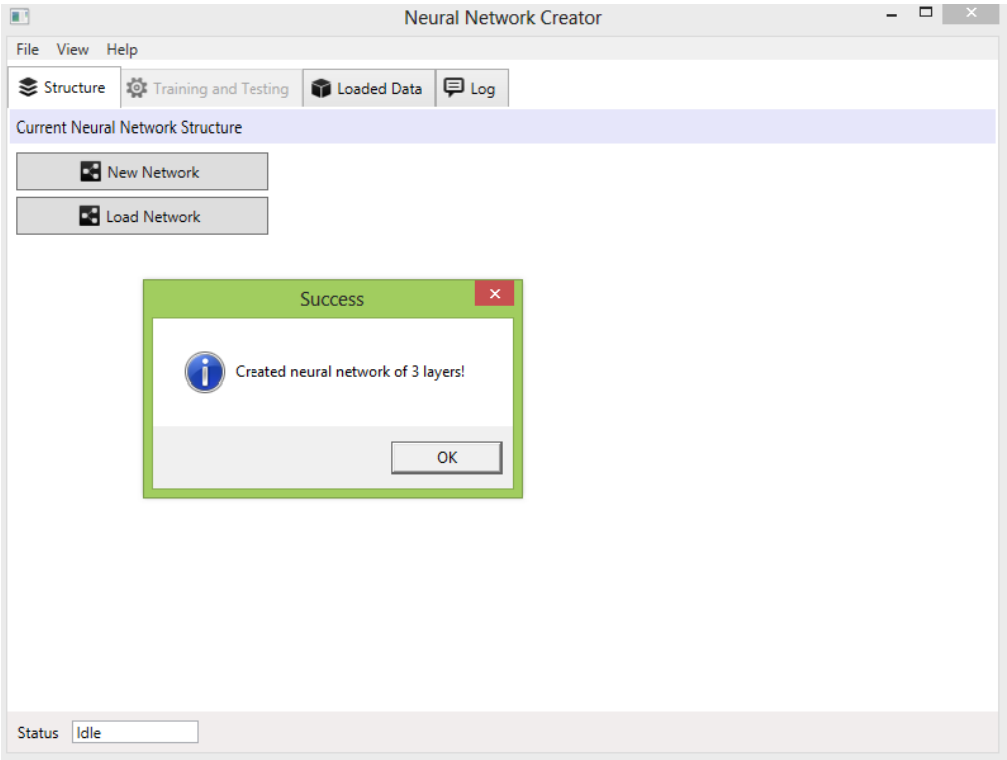

Figure 11.4: The screen after successful network creation.

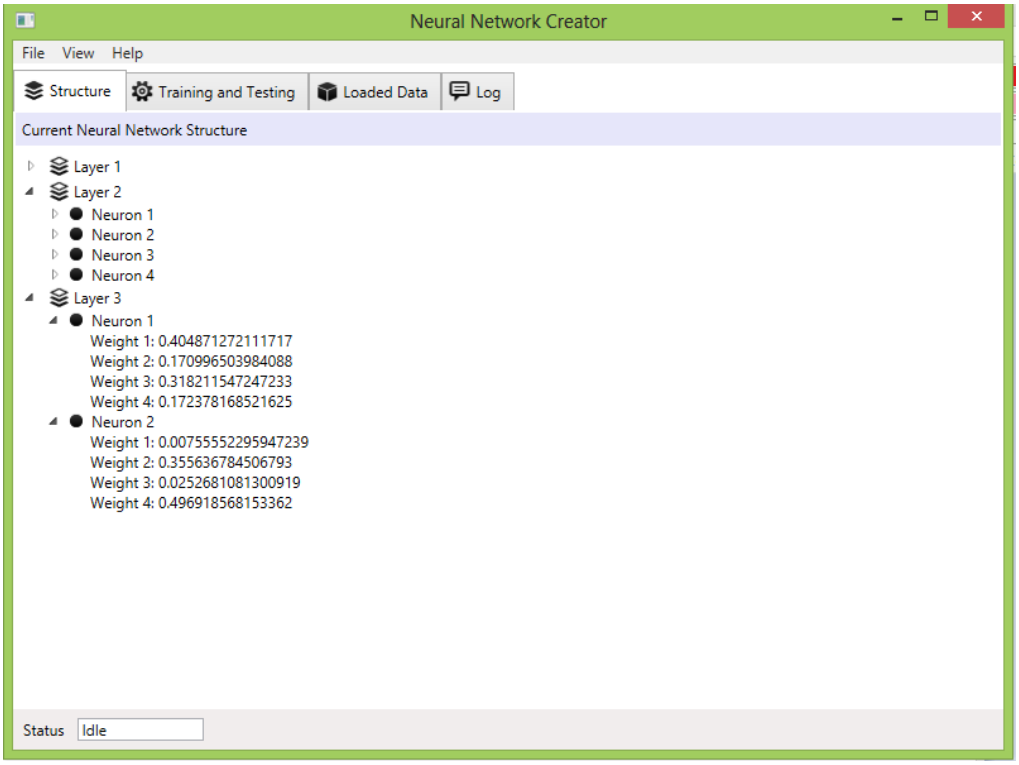

Figure 11.5: The structure and weights of the created network.

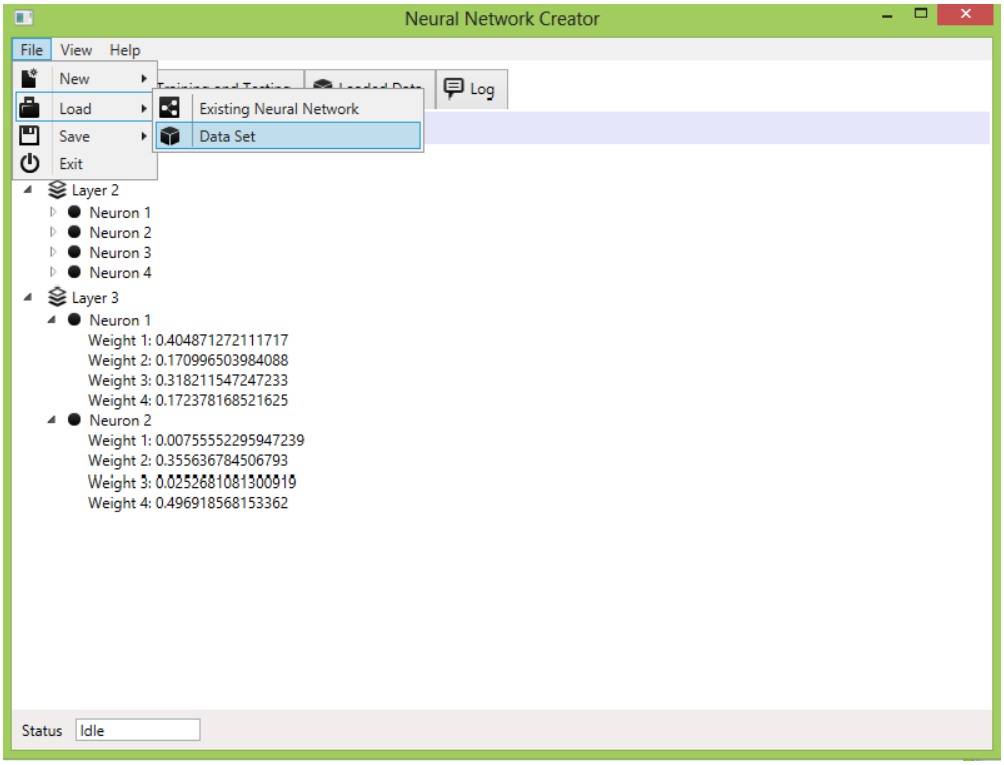

Figure 11.6: Loading of a new dataset.

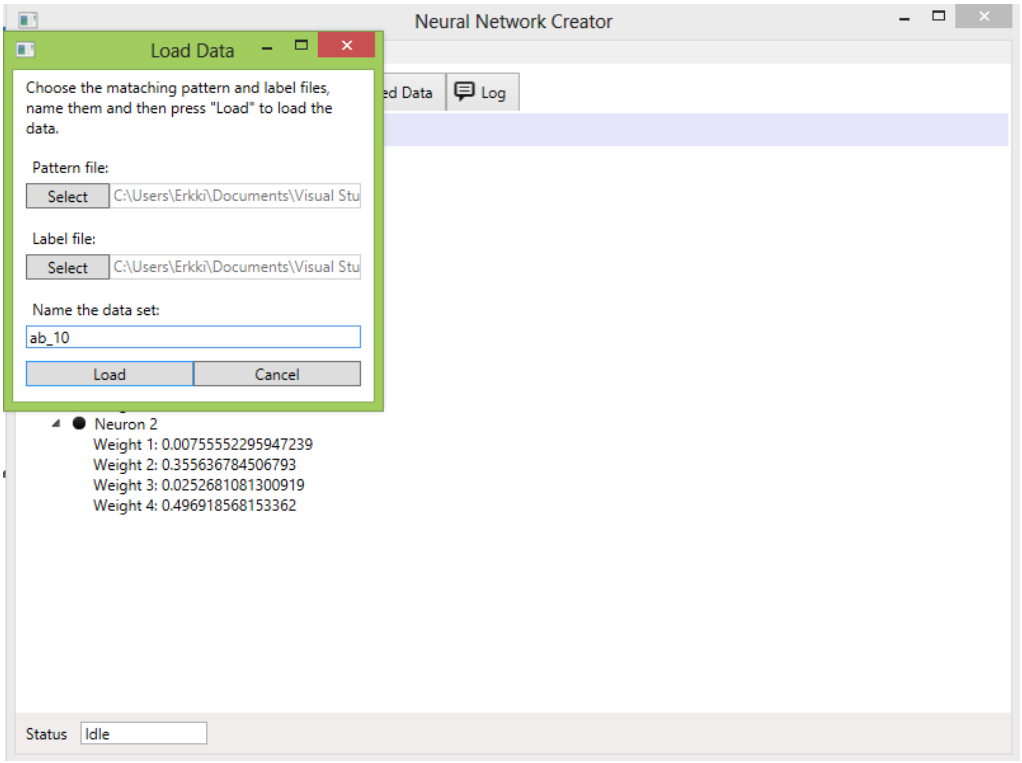

Figure 11.7: Select the patterns, labels and name the dataset.

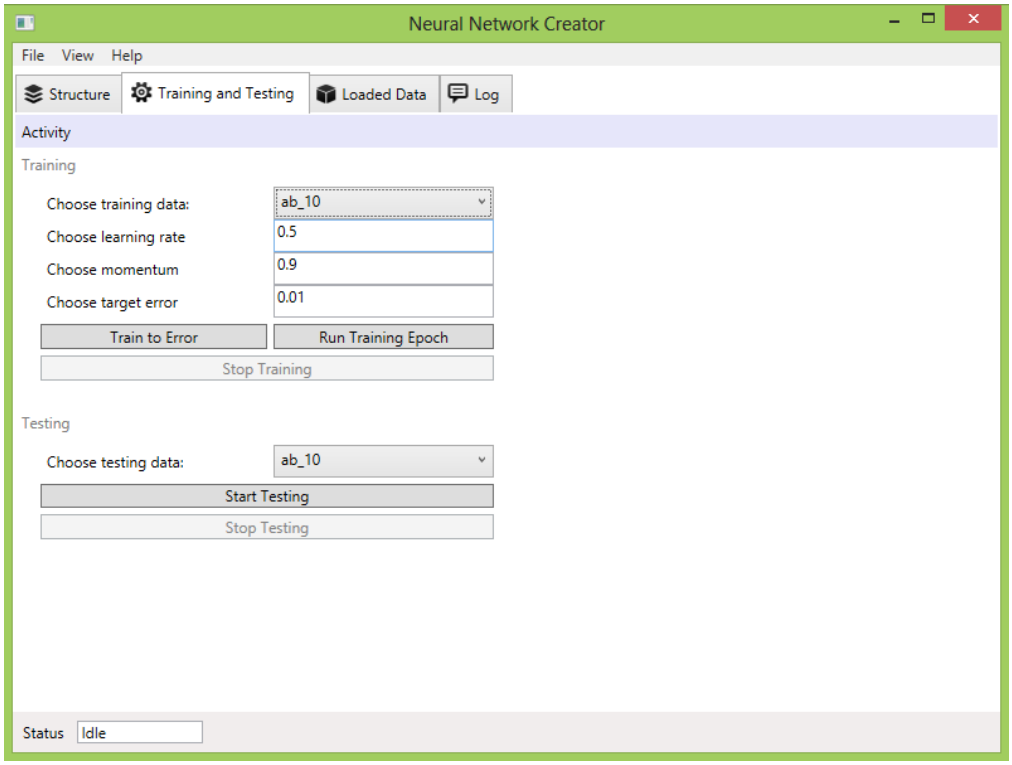

Figure 11.8: The training and testing screen.

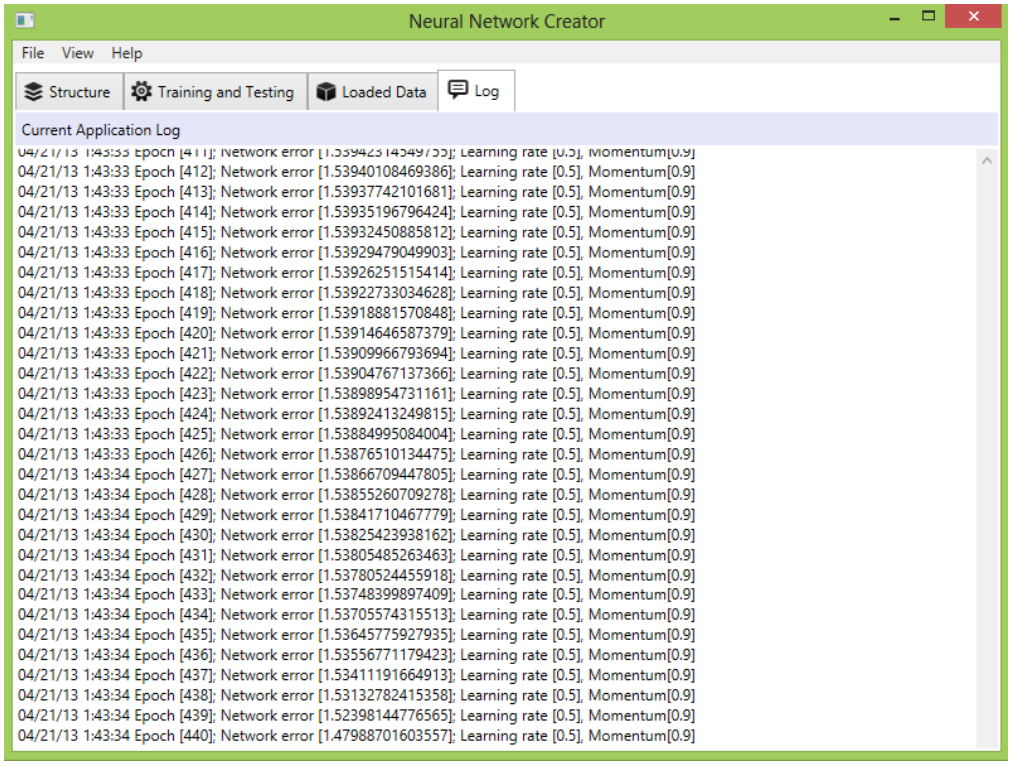

Figure 11.9: All the important events in the application get logged to the log tab.