Using Computer Vision & Deep Neural Networks to Analyse Recursive Data Structures

Ross Byrne

15TH June, 2019

Advised by: Dr. John Healy, Dr. Sean Duignan

Department of Computer Science & Applied Physics

Galway-Mayo Institute of Technology

Abstract

Recursive data structures are fundamental to the solution of many problems in computer science. In particular, recursive structures based on graph theory have been successfully applied to a diverse range of problems including search, storage and machine learning. Despite their utility and widespread use in prototyping, design and teaching, little research has been conducted into how hand-drawn representations of graph structures can be automatically detected, parsed and analysed by computers.

This thesis presents research which investigates the feasibility of parsing a hand-drawn undirected labelled graph and translating it into a JSON representation that maintains its isomorphic properties. The JSON representation will include both the text from handwritten labels extracted from nodes and the relationships between nodes present on the graph. Following research of the literature surrounding artificial neural networks, deep learning and computer vision, a software prototype was designed and developed to investigate the feasibility of automated processing of hand-drawn graphs. This thesis presents the design of the prototype application, benchmarks its performance and evaluates its utility as a graph-processing tool.

Acknowledgements

I would first like to thank my thesis advisers Dr. John Healy and Dr. Sean Duignan for the years of guidance, mentorship and most importantly friendship. This would not have been possible without you.

*

I would also like to thank Galway-Mayo Institute of Technology for providing the opportunity for continuing my journey in education.

*

To my friends and family, thank you for the support and encouragement along the way.

Contents

1 Introduction **7**

1.1 JavaScript Object Notation (JSON) 8

1.2 Deep Learning 9

1.3 Computer Vision 10

1.4 Research Contribution 10

1.5 Thesis Structure 12

2 Literature Review **13**

2.1 Neural Networks 14

2.1.1 Artificial Neurons 15

2.1.2 Perceptrons 16

2.1.3 Multilayer Perceptrons 17

2.1.4 Activation Functions 18

2.2 Training A Neural Network 21

2.2.1 Back Propagation 23

2.2.2 Optimisation Methods 23

2.2.3 Loss Functions 26

2.2.4 Overfitting 28

2.2.5 Hyperparameters 29
2.3 Deep Learning 31
2.3.1 Restricted Boltzmann Machines (RBMs) 31
2.3.2 Deep Belief Networks (DBNs) 32
2.3.3 Autoencoders (AEs) 34
2.3.4 Generative Adversarial Networks (GANs) 35
2.3.5 Recurrent Neural Networks (RNNs) 36
2.3.6 Convolutional Neural Networks (CNNs) 38
2.4 Computer Vision 42
2.4.1 Image Processing 43
2.4.2 Feature Detection 47
2.4.3 Image Segmentation 50
2.4.4 Computer Vision Resources 54

3 System Design **57**

3.1 System Requirements 58
3.1.1 User Requirements 58
3.1.2 Project Constraints 59
3.2 Technologies 60
3.3 System Design 62
3.3.1 Handwritten Text Classification 64
3.3.2 Graph Parsing & Relationship Inference 65
3.3.3 Building the JSON Representation 67
3.4 System Implementation 69
3.4.1 Graph Processing Implementation 69

3.4.2 Graph Node Processing Implementation 78
3.4.3 Handwritten Text Classification Implementation 85
3.4.4 Training Data Pipeline Implementation 89
3.4.5 Training Text Classification CNN 91
3.4.6 Building the JSON Representation 93

4 System Evaluation 95
4.1 Graph Parsing Performance 96
4.1.1 Evaluation Data 96
4.1.2 Software Guidelines & Limitations 101
4.1.3 Results for Node & Edge Detection 109
4.2 Text Classification Accuracy 113
4.2.1 Evaluation Data 113
4.2.2 Handwritten Text Classification Results 116

5 Conclusion 120
5.1 Key Findings 123
5.2 Limitations & Future Research 124
5.2.1 Hand-drawn Graph Parsing 125
5.2.2 Handwritten Text Classification 126
5.3 Closing Remarks 127

Bibliography 129

Appendices 157

A 158

Chapter 1

Introduction

In the context of graph theory, a graph is a mathematical structure consisting of vertices or nodes that are connected by edges which model pairwise relations between objects. There is a distinction made between an undirected graph, where an edge links two vertices symmetrically, and a directed graph, where an edge links two vertices asymmetrically [1].

Graphs can be utilised for a large array of applications. Some such applications include the modelling of network topologies [2], data mining [3], image segmentation [4] and data searching algorithms such as Depth First Search (DFS) [5], Breadth First Search (BFS) [6], Best First Search [7] and A* Search [8]. Graphs can be employed in data analysis, calculating resource allocation and scheduling [9].

There are a number of applications for hand-drawn graphs, ranging from note taking, brainstorming, product planning on a whiteboard, describing how company stakeholders relate to a product's production, outlining the

sequential processes or steps involved in achieving a task or drawing entity relationship diagrams. Software solutions exists for achieving some of these tasks digitally, such as Visio or Draw.io but many of these activities are also performed on whiteboards or on paper.

This thesis presents a research project in the area of computer vision and deep learning and how these technologies can be leveraged for the development of productivity and accessibility enhancing tools.

Specifically, the research hypothesis proposes that computer vision and deep learning can be leveraged to generate a JSON representation of a hand-drawn undirected graph. This representation will include both the text from handwritten labels extracted from nodes and the relationships between nodes present on the graph. This JSON data can then be used for any purpose thereafter, such as storing the data in a database, generating a graph data structure in an existing application such as Visio or Draw.io, used for correcting college or university assignments or further analysed by machine learning algorithms.

1.1 JavaScript Object Notation (JSON)

JavaScript is a programming language designed to execute instructions on a web page in a web browser. It was first introduced in 1995 to add dynamic content to static web pages in the Netscape Navigator browser [10]. JavaScript Object Notation, or JSON (pronounced "Jason") is a data serialisation format capable of storing data in a schemaless fashion. This makes

it well suited for storing nested structures such as trees or recursive structures like graphs. JSON is widely utilised for data storage and communication on the web as its use is not limited to within the JavaScript programming language [10]. JSON is used with a variety of NoSQL databases such as key-value stores, document databases like MongoDB and CouchDB and graph databases like Neo4J [11, 12]. JSON is also a popular data transmission format utilised by REST application [13]. JSON mirrors the format of JavaScript objects and arrays, allowing for the storage and transmission of data objects containing key-value pairs of information.

1.2 Deep Learning

Deep learning is a subset of machine learning that involves the use of large artificial neural networks to solve problems using supervised or unsupervised training. Deep learning has been the driving force behind advancements in visual object recognition and detection [14], self-driving cars [15] and image [16], video [17], audio [18] and speech [19] processing, among others [20].

Deep learning is a promising candidate for performing handwritten text classification, which may be employed to extract and read the handwritten node labels attached to a hand-drawn graph [21]. Handwritten text classification will be a necessary step when generating a JSON representation of a given hand-drawn undirected labelled graph.

1.3 Computer Vision

Computer Vision is defined as "the study of enabling computers to understand and interpret visual information from static images and video sequences" [22]. Computer vision is a valuable tool which can be utilised when working with images. While deep learning has revolutionised the field of computer vision [20], it has been demonstrated to solve a wide range of problems, such as face detection [23, 24], fruit detection for harvesting [25], head pose estimation [26], hand gesture recognition [27], human gender recognition [28], pedestrian collision avoidance [29] and cancer classification [30]. As this research is in the domain of image processing, computer vision may be leveraged when processing images of hand-drawn undirected labelled graphs to provide the desired outcomes.

1.4 Research Contribution

The central objective of this research is to test the following hypothesis:

> "Can a hand-drawn undirected labelled graph be accurately parsed into a JSON representation that maintains its isomorphic properties?".

Specifically, this hypothesis requires the following research objectives to be addressed:

1. The viability of parsing a hand-drawn undirected graph's nodes.

2. The accurate interpretation of graph node relationships.

3. The extraction and classification of handwritten node labels.

4. The generation of a JSON representation of the processed graph.

This thesis endeavours to address these objectives by conducting a review of the current state-of-the-art research in the topics of artificial neural networks, deep learning and computer vision. Insights obtained through this review will then aid the design and implementation of a prototype software deliverable capable of being used to test and evaluate the presented research objectives and the overall research hypothesis.

The principle output of this research is a software prototype capable of receiving a captured image of a hand-drawn undirected labelled graph as an input that returns an accurate JSON representation of the graph as an output.

For the presented research objectives to be satisfied, the following issues must be addressed:

1. Given an image of a hand-drawn undirected labelled graph input, all graph nodes or vertices and graph edges should be accurately identified.

2. Based on the identified graph edges, relationships between nodes should be accurately inferred.

3. Labels contained within nodes present on the graph should be extracted, with the handwritten text being classified to the highest level of accuracy that is feasible.

4. The generated JSON representation of the hand-drawn labelled graph should be isomorphic with respect to the original hand-drawn image.

1.5 Thesis Structure

The remainder of this document is structured in the following manner:

Chapter Two reviews the literature surrounding artificial neural networks, deep learning and computer vision.

Chapter Three presents a detailed description of the prototype software application. This includes details relating to user requirements, system requirements and a walk-through of the designed software prototype source code.

Chapter Four evaluates and benchmarks the software prototype to identify whether the research objectives provided have been achieved and presents a discussion and analysis of the results achieved.

Chapter Five contains the conclusion to this research thesis, highlighting the strengths and shortcomings of the research. After presenting the key finds from this inquiry, potential avenues for related research are discussed.

In addition to the research presented, a link to a public GitHub repository is provided, containing the source code of the developed software prototype and related collateral resources.

Chapter 2

Literature Review

A substantial amount of literature was reviewed in order to make informed decisions throughout the entirety of this research project. Artificial intelligence remains a highly active area of research in the field of computer science. This has been the case for a number of decades. In recent years, it has grown in popularity due to advancements in both computational power and the advent of deep neural networks. At the same time, an increasing number of development frameworks have become readily available. Such frameworks including Deeplearning4j, TensorFlow, Keras and Theano make developing artificial intelligence software far more accessible.

Section 2.1 is a review of neural networks that covers some of the foundational concepts that have guided and shaped the field of artificial intelligence. It covers perceptrons, artificial neural networks and activation functions.

Section 2.2 is a review of training which examines the various aspects of training artificial neural networks. This includes the required steps for

training a neural network, the approaches typically used and a number of methods that can be leveraged to optimise the process.

Section 2.3 is a review of deep neural networks and covers some of the popular neural network architectures applied to deep learning. This section covers architectures such as RBMs, DBNs, Autoencoders, GANs, RNNs, LSTMs and CNNs.

Section 2.4 is a review of computer vision algorithms. This section covers classic computer vision techniques such as image processing, feature detection, image segmentation and computer vision software libraries and datasets.

2.1 Neural Networks

This section discusses the origins of current A.I. techniques and how they developed into the sophisticated designs employed today. The definition of a neural network according to the Merriam-Webster dictionary is, "a computer architecture in which a number of processors are interconnected in a manner suggestive of the connections between neurons in a human brain and which is able to learn by a process of trial and error". The two main focus points of this definition are: the architecture of a neural network being inspired by how the human brain operates and the network learning through trial and error. These two concepts are essential to defining and understanding what a neural network is.

The architecture of an artificial neural network (ANN), or neural network for short, is loosely modelled after the network of neurons in the human brain. The motivation for this was to model the human brain and it's functionality to create superior computers that mimic human intelligence. Work towards this concept began to gain traction in 1943 when Warren S. McCulloch and Walter H. Pitts developed a mathematical model of a biological neuron [31].

2.1.1 Artificial Neurons

Warren S. McCulloch and Walter H. Pitts developed the world's first mathematical model of a biological neuron. Their artificial neuron model is also known as a linear threshold gate [32] and accepts a number of inputs and provides a single output. The inputs are normalised to be the value 0 or 1. The output of the neuron model is also a value that is either 0 or 1, i.e. a binary output. The neuron uses a linear step activation function to produce its binary output value, allowing the neuron to act as a binary classifier that can be used to compute all of the binary functions with the use of one or more neurons. For a McCulloch and Pitts neuron to work, each input value has a weight, being a single value, added to the input. These input weights, similar to the input values, are normalised to be 0 or 1.

While the McCulloch and Pitts neuron was a breakthrough at the time, there are still a number of limitations with the design. These include the fact that the output of the neuron is binary and the weights associated with the inputs are fixed. Therefore, the weights can not be learned through a training program and so must be set manually [33].

2.1.2 Perceptrons

In the immediate period after the publication of the McCulloch and Pitts neuron, study of the brain and how biological neurons work proceeded. One such piece of research published was the book “The Organization of Behavior” by Donald Hebb in 1949. In this book, Hebb puts forward a theory that is now known as Hebbian Theory. This theory states that the more two biological neurons fire together in the brain, the stronger the connection between those two neurons become. He writes, “When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A’s efficiency, as one of the cells firing B, is increased.” [34] This process of strengthening connections between neurons aims to explain how learning occurs. This is now known as “Hebbian Learning”, and has been seen summarised as, “Cells that fire together wire together.” [35]. While this summary shouldn’t be taken literally, as it leaves out various nuances of the theory, it is an accessible summary for an uninformed reader and captures the essential kernel of Hebbian Learning.

After the publication of the McCulloch and Pitts neuron, the next notable advancement was the perceptron, created by Frank Rosenblatt et al in 1958 [36]. Rosenblatt’s perceptron built on the McCulloch and Pitts neuron and Hebb’s “Hebbian Learning” to provide a more advanced model of an artificial neuron aimed at binary classification. His model, unlike the McCulloch and Pitts neuron, was capable of learning its input weights and bias

through trial and error training using a four step training process that includes a perceptron learning rule. The perceptron has the ability to accept positive and negative numbers as inputs. This too is an improvement over the boolean inputs of the McCulloch and Pitts neuron. One of the main limitations of the perceptron is its ability to only solve linearly-separable functions. The shortcomings of the perceptron were discussed in detail in Marvin Minsky and Seymour Papert's 1969 book, "Perceptrons; an introduction to computational geometry" [37]. Many people at the time believed that the shortcomings of the perceptron applied to all neural networks and this lead to research in neural networks halting for a decade [38].

2.1.3 Multilayer Perceptrons

As previously stated, a single perceptron is unable to solve non linear problems. This limitation means a single perceptron is unable to solve an exclusive OR (XOR) function. What researchers at the time were unaware of, was that the use of multiple perceptrons connected together in a network could in fact solve XOR functions and many other non linear problems. Connecting multiple perceptrons together in this fashion creates what is now known as a multilayer perceptron. Multilayer perceptrons are considered members of the multilayer feed-forward neural network family. This type of neural network, sometimes called a multilayer perceptron (MLP), contains an input layer, one or more hidden layers and an output layer. The neurons in the input layer of a multilayer perceptron use linear activation functions. All other neurons in the network use non linear activation functions. Multilayer

perceptrons belong to the family of feed-forward artificial neural networks because values are fed from the input layer nodes, forwards through the network in one direction until they reach the output layer neurons [33]. Figure 2.1 shows a visualisation of what a feed-forward neural networks looks like.

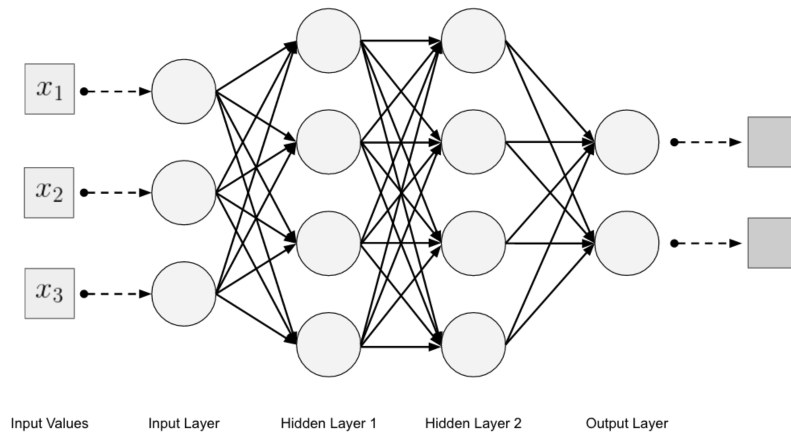


Figure 2.1: Neural Network diagram, adapted from [33].

2.1.4 Activation Functions

Each artificial neuron contains an activation function. The activation function mirrors the part of the biological neuron that decides whether or not to pass a received signal to the neurons it is connected to. The following section reviews the various activation functions found in neural networks and where in a network they can be employed.

Linear activation functions act as an identity function. The input value is not normalised to be within a set value range, essentially allowing the signal to pass through a neuron unchanged. Linear activation is usually found

in input neurons in the input layer of a neural network. Linear activation functions are described by Agostinelli et al [39].

A binary threshold activation function [40] can be used in neurons found in the hidden layer of a neural network. This activation function works in a manner similar to a biological neuron in the sense that it accepts a signal from a previous neuron and decides whether to pass the signal on to the next neuron. If the signal value is above a certain threshold value, the signal is propagated on. If the signal is below the threshold value, the neuron does not pass the signal on.

The sigmoid activation function [41] is a logistical activation function that normalises input values into a range between 0 and 1. Due to the range of its output values, given by the formula:

$$f(x) = \frac{1}{1 + e^{-x}}$$

it can be used to calculate probabilities during binary classification. This activation function can also be found in neurons in the hidden layer of a neural network. It provides the ability to learn complicated features allowing a neural network to solve non linear problems. A limitation of the sigmoid activation function is that it suffers from the vanishing gradient problem [38]. The vanishing gradient problem [42] is when the back propagated error used to update a neuron's weights during training becomes minuscule. This vanishing of the error gradient prevents a neuron's weights from being effectively updated, slowing or stopping training.

The tanh activation function [43] normalises its input values to be between the values -1 and 1. This is a wider range than the sigmoid activation function's range of 0 to 1, which allows it to better deal with negative input values. Tanh can also help solve the vanishing gradient problem [42] that the sigmoid activation function can suffer from [38].

The softmax activation function [44] normalises its input value to be in the range of 0 to 1. This activation function is used primarily in the output layer of a neural network when classifying multiple classes. Softmax is capable of classifying more than two classes, therefore separating itself from binary classifiers, which can only differentiate between two. When using softmax in the output layer of a neural network, each node represents a class the network is trained to classify. The output of each node is a probability between the values of 0 and 1. The total sum of all outputs from the output layer is equal to 1. When a classification is performed, the class representing the output node with the highest value or probability is the selected classification.

The rectified linear unit [45] is a state of the art activation function used in hidden layer neurons [46]. Also known as ReLU for short, they are now the more popular choice of activation function due to the quicker computation time when compared to other alternatives. The ReLU activation function eliminates the vanishing gradient problem that other activation functions, such as sigmoid, suffer from. ReLU activation normalises its input values employing the following logic. If the input value is less than 0, it sets the value to 0. Otherwise, it normalises any positive value to be in the range

of 0 and 1. Performance of ReLU activation can be further improved when parameters are added during training [47].

2.2 Training A Neural Network

Once a neural network has been designed, it must be trained so that it can "learn" how to solve a given problem. This training is in the form of trial and error learning and is analogous to how humans learn new concepts. Training a neural network is a complex procedure with a number of required discrete steps. It necessitates the availability of a large quantity of training data which consists of input and expected output values. Using image recognition as an example, an input value would be an image of a cat represented as a vector of numbers. The output value in the training data could be the word "cat". The goal is to train the neural network to output the correct value when given a pattern as input. Keeping with the cat example, the neural network should output the word "cat" when an image of a cat is given as an input.

With a dataset in the correct format of input values mapped to expected output values, the neural network can be trained in an iterative process. The process of training is as follows: a value is selected from the training data and is given as an input to the network's input layer. The value is then passed forward through the hidden layers of the network, changing as it is affected by the network's neurons. When the value is computed in the output layer of the network, it is evaluated against the expected output value in the training

dataset. If the output value does not match the expected output value, the weights and biases in the network are changed and the previous steps are repeated. This process iterates until the actual output value matches the expected output value. Once these converge, the neural network has been trained. In general, the larger and more complex a neural network is, the longer it will take to train. This is a characteristic of neural network training and is accentuated with large neural networks designed for image recognition [48, 49].

The goal of training a neural network is to find the optimal weights and biases for all the neurons in the network that allow for the most accurate results when evaluating the training data. These optimal weights are found by continuously modifying the network's weights and biases in a manner that minimises the errors that the network produces on the training data. Calculating the errors the network is generating on the entire training dataset is performed by a loss function. This is sometimes called a cost function. The loss function provides a metric for how well the network is performing and allows the training program to know whether a new set of weights and biases improve the network's accuracy or not. The loss or error value is zero when a network is perfectly trained. Networks that solve complex non-linear problems normally don't have a loss value of zero because it would either take too long to train or require too large a dataset. Therefore, most networks are trained to be as accurate as possible within an acceptable amount of time. This process of minimising the loss is known as optimisation.

2.2.1 Back Propagation

In order to train a feed-forward multilayer neural network, a method know as back propagation must be employed [50]. This is due to the complexity of attempting to train a neural network with an input layer, one or more hidden layers and an output layer. As a full explanation of the complexity of back propagation is beyond the scope of this literature review, the following is a high level explanation of how the algorithm works. In back propagation, each training iteration requires two steps, a forward and backward pass through the network. Training data is passed through the network during the forward pass and the loss is calculated based on the output. Then, to update the required weights in the neural network the error is propagated backwards through the network using the chain rule in calculus. This allows the weights to be proportionately updated based on their contribution to the total error. A more comprehensive review of back propagation is provided by Vogl et al [51] and J. Schmidhuber [52].

2.2.2 Optimisation Methods

There are a number of optimisation methods for training neural networks. The most popular form of optimisation is called gradient descent [53]. Gradient descent works similarly to a hill climbing search algorithm. Using the well known analogy of the mountain, the highest peak of the mountain represents the error or loss a network produces with its given set of weights and biases. The aim is to descend the mountain by testing a new set of weights and biases and choosing the set that reduces the loss, therefore resulting in a

descent of the “mountain”. The mountain in this analogy is the error space, which is created by all the possible sets of weights and biases and their associated loss values. See Figure 2.2 for a view of this error space. The goal of gradient descent is to optimise the weights and biases of a neural network so that the loss is as close to zeros as is feasible.

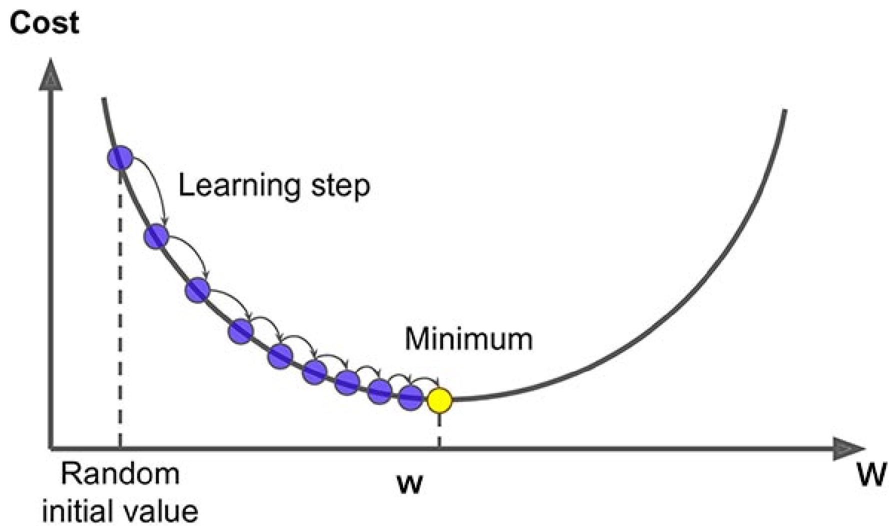


Figure 2.2: Gradient Descent diagram, adapted from [54].

Steepest descent [53, 55, 56] is a version of gradient descent that has a dynamic learning rate. The learning rate in gradient descent is the speed at which the algorithm descents to the bottom of the gradient, e.g. the bottom of the “mountain”. This lowest value is known as the minima. Steepest descent calculates the learning rate after each iteration to maximise the speed at which the loss reaches minima. The benefit of steepest descent is the speed increase gained while training. If the loss function is taking a steep path down towards the minima, it saves time by speeding up the descent until the

algorithm approaches the minima.

The gradient descent and steepest descent optimisation algorithms both calculate the gradient based on the entire training dataset. This is known as full batch training and becomes problematic when larger datasets are used. The larger the dataset the more expensive, in terms of time and space complexity, it becomes to calculate the gradient. Stochastic gradient descent [53, 57] is a version of gradient descent that solves this issue. Stochastic gradient descent uses mini batch training. A mini batch is a subset of the training dataset. It allows the algorithm to calculate the gradient based on a smaller subset of the data, in turn reducing time and space complexity of training with very large training datasets. Training using mini batches in now the most common approach to training due to the size of modern training datasets.

AdaGrad [58] is an optimiser that is a modified version of gradient descent. AdaGrad handles the learning rate differently, allowing for better learning on sparse data. The learning rate is the rate at which the optimiser descents the gradient during training.

AdaDelta [59] is a version of AdaGrad that handles the learning rate of the optimisation method more effectively. It overcomes an issue with AdaGrad that can result in the learning rate decreasing until it reaches zero when training for very long periods of time. When the learning rate reaches zero the training stalls and stops.

Adam [60] is a gradient-based optimisation method that is currently popular due to its performance in terms of time complexity. The goal of Adam is to combine the benefits of the AdaGrad and RMSProp [61] optimisation methods [60]. Adam's leading performance is due to its computationally efficient algorithm which manifests as an increase in the speed of convergence during the training of neural networks [62].

2.2.3 Loss Functions

Loss functions and their place in training is best described by M. Avendi when he says "Loss Functions are at the heart of any learning-based algorithm. We convert the learning problem into an optimization problem, define a loss function and then optimize the algorithm to minimize the loss function" [63]. Loss functions calculate the error a neural network is producing on training data during training. This is calculated by averaging the errors the network makes across the entire dataset. The calculated loss is used as a metric for identifying how far a neural network is from its ideal trained state.

Mean Squared Error (MSE) [64] is a simple and widely used loss function. It involves calculating the average square of the errors produced across a dataset. Mean squared error is used for training neural networks performing regression. Mean Absolute Error (MAE) [65] is an alternative to MSE. It involves calculating the average absolute error across the training dataset. Like MSE, Mean absolute error is used for regression problems.

Hinge [66, 67] is a loss function commonly used in the training of neural networks performing binary or hard classification. Binary classification meaning an output value of 0 = not cat, 1 = cat. MSE and MAE would not be appropriate to use in this case as they are used to calculate loss for regression, not classification problems [33].

Negative Log Likelihood is a logistic loss function often used when training neural networks that perform multi-class classification. Negative log likelihood can be seen used in training classifiers used for tasks such as colon and leukaemia cancer classification [30]. Cross Entropy is another loss function used when training a neural network to perform multi-class classification. Cross entropy originated in information theory but is mathematically equivalent to negative log likelihood, which originated in statistical modelling. While this often leads to confusion, both algorithms are interchangeable [33].

KL Divergence [68] is a loss function used during the training of neural networks performing reconstruction tasks such as image reconstruction [69]. KL divergence is an algorithm for calculating the divergence or difference between two data distributions. This makes it a suitable loss function for reconstruction, as the task of reconstruction is to reconstruct the data in a dataset. Therefore, a loss function which can assess the probability of differences between the training and reconstructed data is required.

2.2.4 Overfitting

Overfitting is a problem that occurs when a neural network is over trained on a given dataset. This results in a neural network working very well on the training data, but not generalising well leading to poor results with new unseen data. When designing a neural network for a given problem, if the network is not large enough it won't be able to learn to solve the problem. Conversely, if the network is designed to be too large it will lead to overfitting on the training data, where nodes in the network are effectively starved of information. Overfitting is a common issue encountered when training and there are a number of steps one can take to avoid it.

Data Splitting [70] is a basic approach to combating overfitting. It involves splitting the training dataset into two parts, a training dataset and a test dataset. A neural network is trained using the training dataset but then network performance is evaluated using the test dataset. This form of cross validation helps the network to generalise. Another approach is to split the data into three datasets, training, validation and test. The network is trained on the training data, weights and other parameters are updated based on the validation data and the network accuracy is assessed on the test data.

Data Regularisation can be performed in an attempt to avoid overfitting and promote generalisation once a given neural network is trained. Solazzi and Uncini explain the process as "the presence of noise in examples can lead to discover spurious structure in the data. Regularisation techniques impose smoothness constraints on the approximating set of functions f, excluding

high-frequency components. This allows to increase generalisation capability in approximation problems” [71].

The technique of early stopping has been shown to help avoid overfitting when training large neural networks. Caruana et al proposed the method of early stopping and show that neural networks trained with excessive capacity still generalise well if the training is stopped before overfitting occurs [72]. Dropout [73] is another approach to preventing neural networks from overfitting. Dropout helps large neural networks generalise well by randomly ‘dropping’ or removing neurons and their weights from the hidden layer of a network during training. This prevents neurons from co-adapting during training.

2.2.5 Hyperparameters

Training neural networks is a difficult task that varies depending on a number of factors such as network design, size or training data. This often adds an element of trial and error to training a neural network. To tailor training to a given neural network, there are a number of hyperparameters that can be set to fine tune the training process. While hyperparameters are typically selected from empirical iterative approaches, the automation of this process has been recently reported by Miikkulainen et al with promising results [74]. A number of different hyperparameters can be tuned to improve training.

Learning Rate controls the speed at which a network learns during training. When training with a form of gradient descent, learning rate controls the size of the 'step' taken when descending the gradient. It does this by effecting how much the weights change. If the learning rate is too large, the network will descent too fast and miss the global or local minima. If the learning rate is too small, the network will take an excessive amount of time to train. An in depth discussion of the importance of the learning rate hyperparameter and new methods of setting it are described by Smith [75]. Momentum is a training hyperparameter that acts like momentum in physics. It sets how fast the learning rate changes as the network trains. Momentum increases the speed at which a network can learn. The idea being, if the training is moving in the correct direction, the momentum increases the learning rate to speed up the descent down to the minima. Sutskever et al present a detailed review of the momentum hyperparameter and present results showing how careful turning of momentum can provide improvements to optimisation performance, stating "we observed that the use of stronger momentum (as determined by μ) had a dramatic effect on optimization performance, particularly for the RNNs" [76].

For optimisation methods that employ mini batching, such as stochastic gradient descent, the size of the mini batches can be configured. Masters and Luschi present results from a variety of tests using CNNs on popular benchmark datasets which states "the presented results confirm that using small batch sizes achieves the best training stability and generalization performance, for a given computational cost, across a wide range of experiments"

[77]. The number of training iterations or epochs, as they are also called, can be manually set if a method such as early stopping [72] is not in use. One epoch is one full pass over the entire training dataset. Setting the number of epochs correctly can be important because if a neural network is not trained for long enough it will not be accurate. If a neural network is trained for too long, overfitting will occur and it will not generalise well [72].

2.3 Deep Learning

Deep learning is a subset of machine learning. It involves the use of large artificial neural networks to solve problems using supervised or unsupervised training. Over the years as neural networks have become larger, with more neurons and connections, the term deep neural networks and deep learning have been adopted. Some network architectures, such as convolutional neural networks, have massive numbers of neurons in each layer. Deep learning is the application of such deep neural networks [78, 20]. In this section a number of common deep neural network topologies or architectures are discussed. These architectures are Restricted Boltzmann Machines (RBMs), Deep Belief Networks (DBNs), Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), Autoencoders (AEs), Long Short Term Memory Networks (LSTMs) and General Adversarial Networks (GANs).

2.3.1 Restricted Boltzmann Machines (RBMs)

While Restricted Boltzmann Machines or RBMs [79] are not deep neural networks, they are covered for historical purposes due to their role in various

deep neural network architectures. A RBM is a neural network composed of two layers of neurons, an input and hidden layer. The key characteristic of RBMs is that the hidden layer has less neurons in it than the input layer. This acts as a restriction on the input data, resulting in feature learning and dimensionality reduction. Deep RBMs with more than two layers can be found, with each subsequent layer having less neurons than the previous one. This is examined later in this section. In Figure 2.3, the layout of the neurons and connections in a RBM can be seen. RBMs can be used for the compression, reconstruction, generation and classification of data. Rastgoo et al [80] have used multiple RBMs for hand sign language recognition. Smith [81], exploited a modified RBM, known as a conditional restricted Boltzmann machine (CRBM), for music generation. Later in this section, the use of RBMs as a component of other deep neural networks will be examined, such as deep belief networks and autoencoders. Hinton [82] provides a detailed guide for training RBMs which includes suggestions for hyper-parameter settings such as momentum, learning rate and mini batch sizes, along with suggestions for the number of neurons in the hidden layers. When speaking about RBMs, Hinton states “their most important use is as learning modules that are composed to form deep belief nets” [82].

2.3.2 Deep Belief Networks (DBNs)

A Deep Belief Network (DBN) [84] is a deep neural network that incorporates RBMs as components in its architecture. A DBN consists of layers of RBMs which are stacked, followed by a feed-forward neural network to

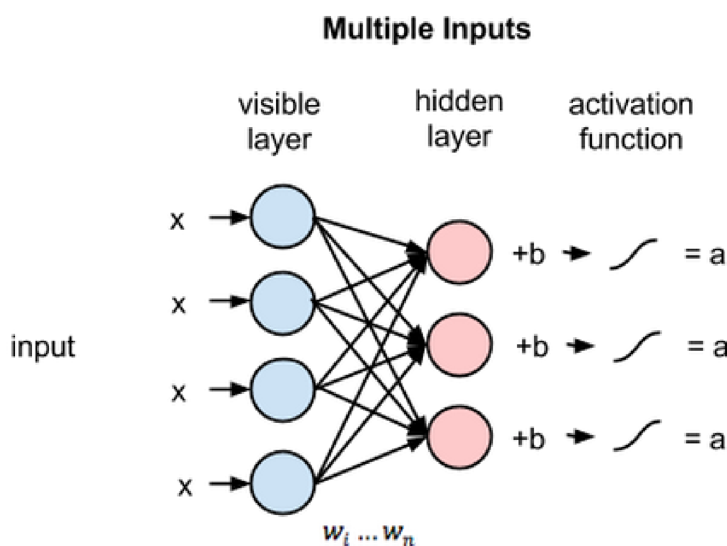


Figure 2.3: Restricted Boltzmann Machine, adapted from Skymind [83].

create a deep neural network capable of performing, *inter alia*, tasks such as prediction and classification [85]. In Figure 2.4 the network topology can be seen. Training a DBN takes place in two stages. First the RBMs in the network are pre-trained, then the entire network is fine tuned via supervised learning. During the pre-training stage features in the dataset are extracted and learned by the stacked RBMs. This occurs through unsupervised learning in the form of reconstruction. Reconstruction is explained in detail later in this section. After features are extracted in the RBM components of the network, normal supervised training takes place in order to fine tune the network to perform tasks such as classification. This method of training enhances the network's performance when compared to supervised training on its own. DBNs can be trained for many tasks such as image recognition [84] and speech recognition [86, 18]. DBNs have now since been replaced in many cases by convolutional neural networks or recurrent neural networks when it

comes to tasks such as image classification or speech recognition [20].

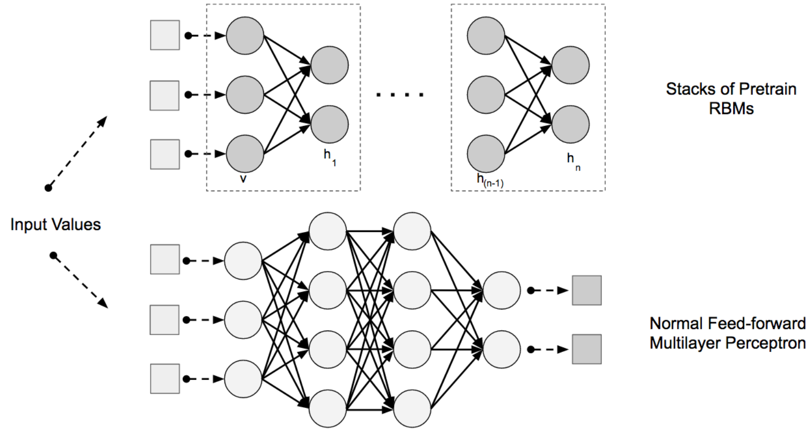


Figure 2.4: Deep Belief Network, adapted from [33].

2.3.3 Autoencoders (AEs)

Autoencoders (AEs) were first introduced by Rumelhart et al [50] as a solution to "backpropagation without a teacher" [87]. An AE is composed of two components, an encoder and a decoder. In a rudimentary AE, both of these components are implemented as RBMs. The first RBM receives an input and performs compression. This RBM is then connected to a second RBM placed in reversed order which performs the opposite of compression, reconstruction. Reconstruction is the process of taking a compressed representation of data and regenerating it as close to the original input data as possible. Therefore, it can be said that the encoder's task is data compression and the decoder's task is data reconstruction. Once trained, these two components can be used separately for a number of tasks such as semantic hashing [88], image search and retrieval [89] and denoising corrupted input data [90] among others. AEs are trained using unsupervised training which

means training data does not have to be labelled. The goal in training is to teach the encoder component to compress the input data and the decoder component to accurately reconstruct it. The architecture of an AE can be seen in Figure 2.5.

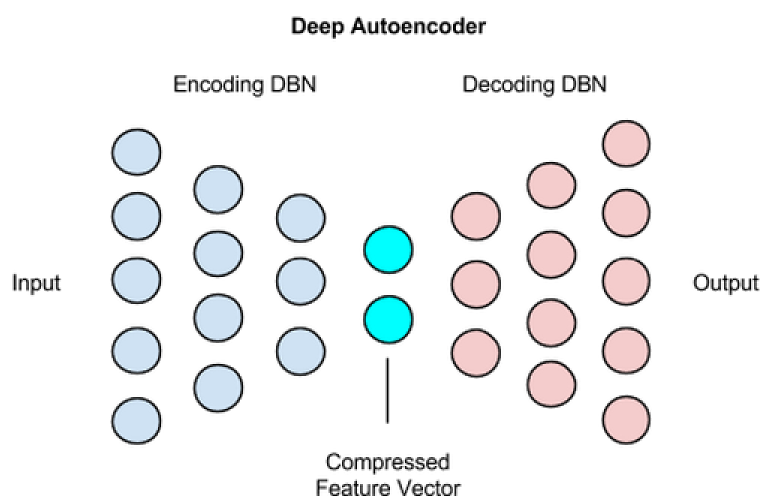


Figure 2.5: Autoencoder Architecture, adapted from [91].

2.3.4 Generative Adversarial Networks (GANs)

Generative Adversarial Networks (GANs) [92], like an AE, are composed of two components, a generative network and a discriminative network. In the original paper, [92] Goodfellow et al provide a succinct description of what a GAN aims to achieve when they state, "the generative model is pitted against an adversary: a discriminative model that learns to determine whether a sample is from the model distribution or the data distribution. The generative model can be thought of as analogous to a team of counterfeiters, trying to produce fake currency and use it without detection, while the

discriminative model is analogous to the police, trying to detect the counterfeit currency". Therefore, when training the generative network its task is to trick the discriminative network into classifying the generated content as real content. The generative network is trained until it is capable of generating content that successfully tricks the discriminative network. Although GANs are relatively new, having only been published in 2014, they are showing impressive results when used for generative tasks such as audio [93], image [94], video [95] and text to image [96] generation. For an overview of a GAN's components, see Figure 2.6.

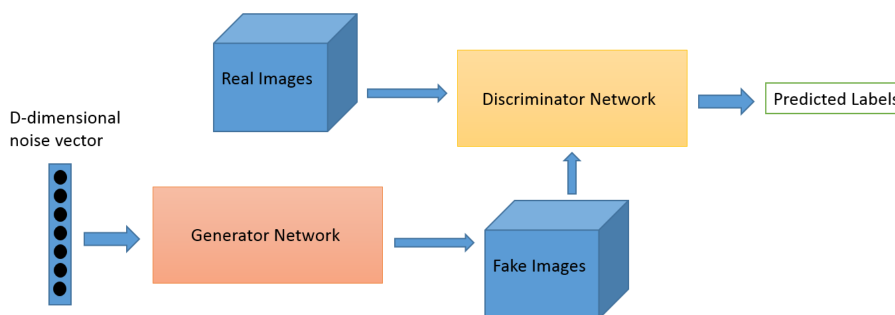


Figure 2.6: Components of a GAN, adapted from [97].

2.3.5 Recurrent Neural Networks (RNNs)

Recurrent neural networks (RNNs) [50] are feed-forward neural networks with key differences in network topology to typical feed-forward architectures, such as MLPs. In a standard feed-forward network, there are layers of neuron consisting of a single input and output layer and one or more hidden layers. Neurons in each layer are connected to all the neurons in the subsequent layer. The difference with RNNs is that the neurons within a layer

have connections between each other as well as connections to the neurons in the subsequent layer. These connections between neurons in the same layer are called recurrent connections. In standard feed-forward networks, neurons do not have connections to other neurons in the same layer. These recurrent connections allow for better handling of sequential data such as time series, speech and language data [20]. Figure 2.7 depicts the architecture of a layer in a RNN. Like feed-forward neural networks, RNNs are trained using back propagation. This however presents some issues, such as exploding or vanishing gradient problems when training for a large number of iterations [98]. RNNs can perform tasks such as regression [99], classification [99], speech recognition [100], prediction on time series data [101, 102] and predicting the next word in a sequence [103].

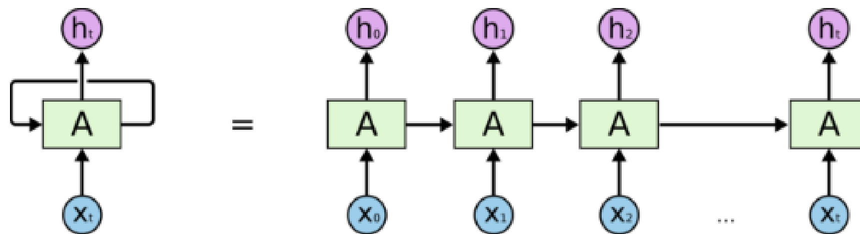


Figure 2.7: Recurrent Neural Network Architecture, adapted from [104]. On the left, is a 'rolled up' view of the RNN layer. x represents the input and h represents the output to the subsequent layer. On the right, is the 'unrolled' view of the RNN layer which shows the time steps t. Here, x represents the input at a given time step. h represents the outputs at the given time step to the subsequent layer. Information passes through the time steps through the recurrent connections. This allows the input at a given time step to be influenced by previous time steps. This is what makes RNNs ideal for handling sequential data [20].

Long Short-Term Memory Networks (LSTMs) [105] are a version of RNNs with modifications aimed at solving RNNs exploding and vanishing gradient problems. The more complex structure of a neuron in an LSTM can be seen in Figure 2.8. Deep LSTMs appear to be out performing standard RNNs for tasks such as speech recognition [19]. LSTMs are similar to RNNs except that their hidden layer neurons have a memory component that can save information for long periods of time.

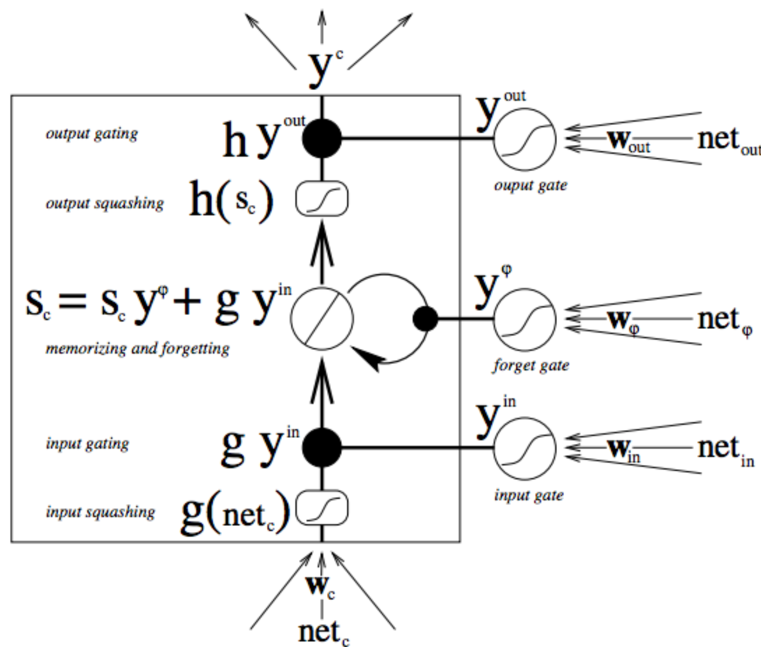


Figure 2.8: Long Short-Term Memory neuron architecture including input, forget and output gates, adapted from [106].

2.3.6 Convolutional Neural Networks (CNNs)

Convolutional neural networks (CNNs) [107] are a deep neural network architecture primarily aimed at image recognition and classification [108].

CNNs are composed of a number of different layer types that are designed to improve the performance of image processing neural networks. The three types of layers in a CNN are the input layer, feature extraction layers and classification layers. The input layer is the first layer that accepts input. The feature extraction layers include convolutional layers and pooling layers. The classification layers include fully connected layers and an output layer. A CNN is typically composed of an input layer, feature extraction layers and classification layers. A basic CNN would consist of layers in the following order: an input layer, followed by a convolutional layer, a pooling layer, a fully connected layer and an output layer [33]. Figure 2.9 shows the layout of the different layers in a CNN.

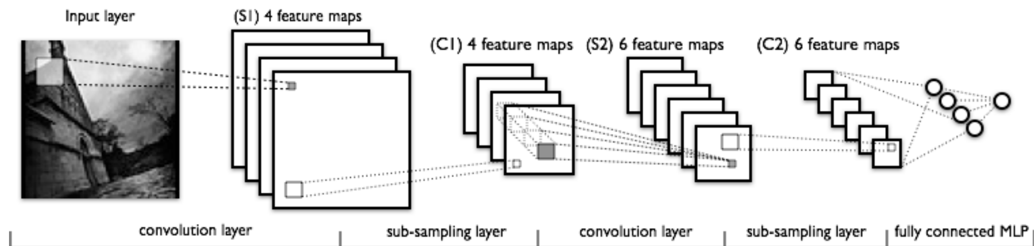


Figure 2.9: Convolutional Neural Network layout, adapted from [109].

Each layer in the CNN has neurons laid out in three dimensions representing width, height and depth. With input data for a CNN being an image, the input data contains the width, height and the number of channels. For colour images, there would be three channels for the RGB values of each pixel. The input layer can be visualised as a 3D rectangle [33].

Convolutional Layer

The convolutional layer in a CNN performs a convolution on the input data. This is the act of calculating the dot product between the input data and the weights from the connections connecting the input neurons to the subsequent layer. Input data can be data from the input layer or data output from another convolutional layer. A convolution creates a feature or activation map. As shown in Figure 2.9, feature maps stack on a third dimension creating a 3D volume of neurons [33, 78].

Pooling Layer

After a single or set of convolutional layers, a pooling layer is typically applied. Pooling, such as max, average or stochastic pooling [110, 111] is a form of dimensionality reduction. Boureau et al explain pooling layers as “a step of spatial ‘pooling’, where the outputs of several nearby feature detectors are combined into a local or global ‘bag of features’, in a way that preserves task-related information while removing irrelevant details” [110]. Although pooling is destructive, as it condenses the features learned by the network, it is necessary for reducing the size and complexity of a CNN [112]. Figure 2.10 gives a visual representation of max pooling. Lee et al propose a method of employing tree and max-average pooling in CNNs which yields state of the art performance on the MNIST [113] and CIFAR-10 [114] benchmark datasets [115].

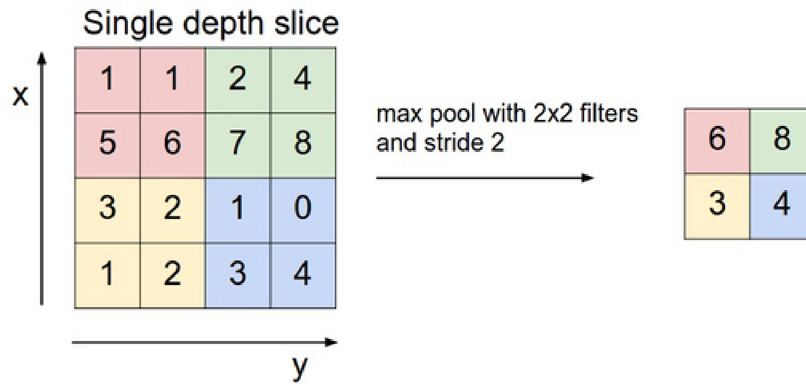


Figure 2.10: Max Pooling, adapted from [109].

Fully Connected Layer

A fully connected layer is a layer of neurons where each neuron has a connection to every neuron is the previous layer. This is similar to a normal feed-forward neural network. The fully connected layer in a CNN is responsible for the classification of the input image. Usually if a CNN is classifying an image based on a number of possible classes, softmax is used in the output layer [112].

There are a number of advanced CNN architectures that can be found, some of which produce state of the art results in image classification and object recognition. Such CNNs include VGG-Net [116] and AlexNet [16]. Other modified versions of CNNs, R-CNN [117], Fast R-CNN [118], Faster R-CNN [119], YOLO [120] and SSD [14] have all shown impressive results in object recognition.

2.4 Computer Vision

Computer vision is the application of the various deep neural networks previously covered, alongside more tradition machine vision techniques. Bebis et al define computer vision as "the study of enabling computers to understand and interpret visual information from static images and video sequences" [22]. Computer vision encompasses many tasks, *inter alia*, face detection [23, 24], fruit detection for harvesting [25], head pose estimation [26], hand gesture recognition [27], human gender recognition [28], pedestrian collision avoidance [29], cancer classification [30] and handwritten character classification [121]. This section provides a review of computer vision techniques, including previously covered deep neural networks, traditional machine vision algorithms such as edge, line and contour detection, image filtering, transformations and computer vision resources covering software libraries and datasets.

Before deep learning revolutionised the field of computer vision [20], there were a set of algorithms employed to achieve computer vision goals without the ease of automation through the training of neural networks. While many classic computer vision algorithms remain relevant, such as image processing techniques, many methods of feature engineering and detection have been superseded by feature learning via deep neural networks such as VGG-Net [116], AlexNet [16], Faster R-CNN [119], YOLO [120] and SSD [14]. This observation is supported by Szegedy et al who state, "Convolutional networks are at the core of most state-of-the-art computer vision solutions for a wide

variety of tasks" [122]. A feature is an attribute or property of the content analysed, where content refers to images or video [123]. Previous efforts were focused on manual feature engineering [25], which required manually compiling a collection of handcrafted features that one desired to detect. Manually created features are engineered by developing a set of heuristics that defined a feature. This can include defining heuristics based on the colour, shape or size [124] or the light reflectivity differences [25] of an object. Manually engineered features can also be defined examples of data distributions that match the distributions of the desired features in sample content [125].

The modern deep learning approach leads one to compile, or utilise existing, labelled or unlabelled datasets and train a deep neural network to perform automatic feature extraction for the purpose of feature classification or detection in new unseen content. Automatic feature extraction can require less domain specific knowledge and save time while still producing comparable results to manual feature extraction [126]. Automatic feature extraction is performed by training a neural network on a suitable dataset, as compared to a person manually engineering features by hand.

2.4.1 Image Processing

In classic computer vision, before attempting to detect features in an image, one must apply processing techniques to prepare the image in order to increase the quality of the results. The processing techniques covered are image manipulation, filtering and transformations. Image processing can be required, for instance, for resizing an image [127] or converting a captured

image to grayscale [128] when colour is not needed. Converting an image to grayscale is advantageous due to the reduced data required for storing an image. When an image is loaded in a computer vision program, the image is stored in a tensor where each pixel of the image is represented as an integer in the range 0 - 255. When a colour image is stored, this tensor contains an extra dimension for the red, green and blue (RGB) values of each pixel. When an image is converted to grayscale, only two dimensions for the width and height of the image is stored and the depth dimension that stores RGB values can be discarded. Modifying a captured image is the changing of the integers, in the range 0 - 255 that represent the pixels, which are stored in a two dimensional tensor [123].

Filters are another type of image processing that perform various tasks like reducing or adding noise [129], smoothing, sharpening or blurring images [130, 131], or gradient based edge detection [132]. Adding or removing noise from images can have an impact on the results of various computer vision tasks such as image classification. Koziarski & Cyganek define noise as "unwanted signal that affects the original one" [133]. They go on to state that "noise is usually modeled as a random multiplicative or additive component added to the pure signal". Koziarski & Cyganek [133] examine how noise can negatively affect image classification results and present two methods of dealing with noise during classification. These methods are: adding noise to training data and removing noise from images before classification. Koziarski & Cyganek find that both methods can provide significant improvements to classification accuracy, with the method of adding noise to training

data providing the greater improvement.

Figure 2.11: Gaussian filter example, adapted from [134]

A Gaussian filter [129] can be used for denoising, smoothing or blurring an image. Applying a Gaussian filter to an image removes high frequency components which result in the removing of strong edges, blurring the image [123]. Figure 2.11 provides a visual representation of an example of a Gaussian filter. Mathematically, a Gaussian function is given as:

$$f(x) = \frac{1}{\alpha\sqrt{2\pi}} exp(\frac{(x-\mu)^2}{2\alpha^2})$$

where μ is mean and α is variance [123]. Histogram equalisation [135, 136] is a processing technique for modifying brightness and contrast in images, which can expose details in an image that may be otherwise obscured. In Figure 2.12, the histogram equalisation can be seen improving the quality of an image. A Median filter [137, 138] is a method of noise reduction and removal, particularly effective at removing Salt and Pepper noise in images

[138]. Median filters divide an image into a number of regions, calculates the median pixel value for each region defined and sets all pixels to that value. This removes the noise from a region by eliminating the random peak values [112]. Figure 2.13 is a demonstration of a median filter applied to an image.

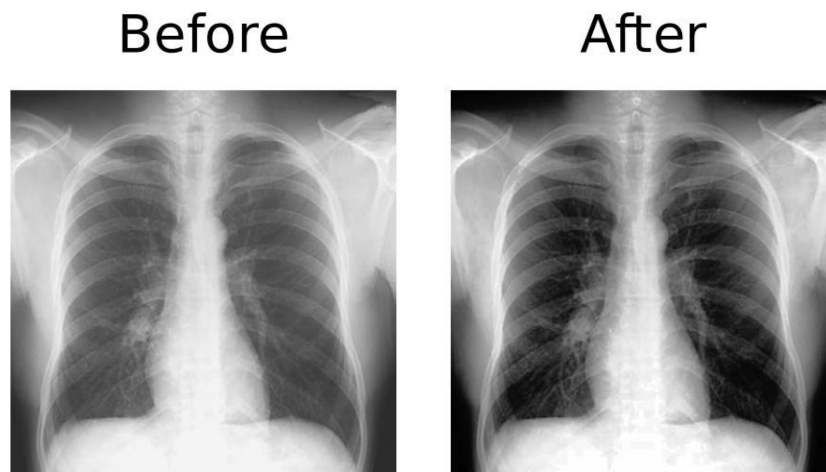


Figure 2.12: Histogram equalisation example, adapted from [139]

Figure 2.13: Median filter example, adapted from [140]

While there are a large number of transformations that can be applied to images, this review focuses on translation and rotation transformations.

These transformations are implemented as matrix multiplications, as images are represented as matrices. A translation is a transformation that moves an image in a given direction on the x and y axis of the image space. The transformation matrix for translation is given as:

$$T = \begin{bmatrix} 0 & 1 & t_x \\ 1 & 0 & t_y \end{bmatrix}$$

where t_x is translation in the x direction and t_y in the y direction [123]. A rotation transformation rotates an image by a given angle. The transformation matrix for rotation is given as:

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

where θ is the angle to rotate by [141].

2.4.2 Feature Detection

There are a variety of algorithms for feature detection in computer vision systems which include edge, line and contour detection. One such algorithm is the popular Canny Edge Detection [142]. Canny Edge Detection takes a multi-step approach to edge detection. The first step is to apply a Gaussian filter to smooth the image, then find the intensity gradients [143] of the image. Next, non-maximum suppression [144] is applied for edge thinning. A threshold is applied twice, once with a low value and once with a high value. This helps remove noise and falsely detected edges. Thresholds are

reviewed later in this section. Finally, edges are tracked by hysteresis [145], removing weak edges not connected to strong edges. The results of this algorithm can be seen in Figure 2.14 [146].

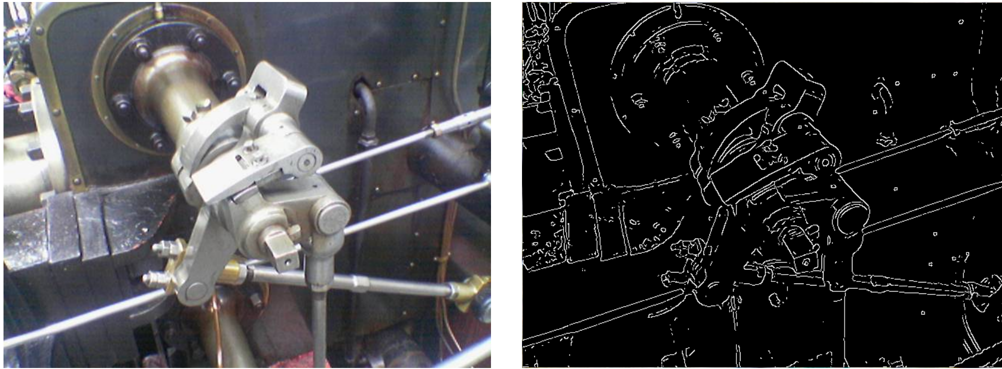


Figure 2.14: Example of Canny Edge Detection. On the left, the image before edge detection is applied. On the right, after edge detection is applied. Adapted from [146]

Harris Corner detection [147] by Harris & Stephens is a feature detection algorithm for detecting corners and edges in an image. The primary goal of this algorithm is to enable feature tracking in image sequences through edge and corner detection. Harris & Stephens state that tracking features in image sequences is not feasible when tracking edges alone. They state, “to enable explicit tracking of image features to be performed, the image features must be discrete, and not form a continuum like texture, or edge pixels (edgels). For this reason, our earlier work has concentrated on the extraction and tracking of feature-points or corners, since they are discrete, reliable and meaningful”. An example of Harris Corner Detection can be found in Figure 2.15.

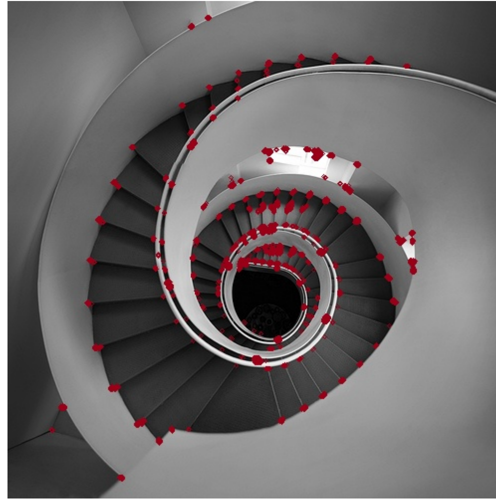


Figure 2.15: Harris Corner Detection example, adapted from [148]

Snakes by Kass et al [149] is an active contour model used for detection of lines, edges and contours. A contour is the boundary around an object in an image. Kass et al provide an explanation for Snakes when they state "a snake is an energy-minimizing spline guided by external constraint forces and influenced by image forces that pull it toward features such as lines and edges. Snakes are active contour models: they lock onto nearby edges, localizing them accurately" [149]. Active Contours Without Edges by Chan & Vese [150] is another active contour model similar to Snakes [149] but with key differences. Chan & Vese's algorithm does not rely on the gradient of an image to detect edges. Instead it utilities Mumford–Shah segmentation techniques [151]. Their algorithm can therefore "detect contours both with or without gradient, for instance objects with very smooth boundaries or even with discontinuous boundaries" [150]. Examples of both Snakes and Active Contours Without Edges can be seen in Figure 2.16. The popular software library OpenCV [152] utilises a border-following algorithm from Suzuki et al

[153] for contour detection. This border-following algorithm detects contours by detecting edges or borders between the image background and a feature in the image. Their border-following algorithm has a topological analysis capability, which allows for the detection of parent borders. This allows for the capture of all features in an image or just the outer most parent feature which may contain child features.

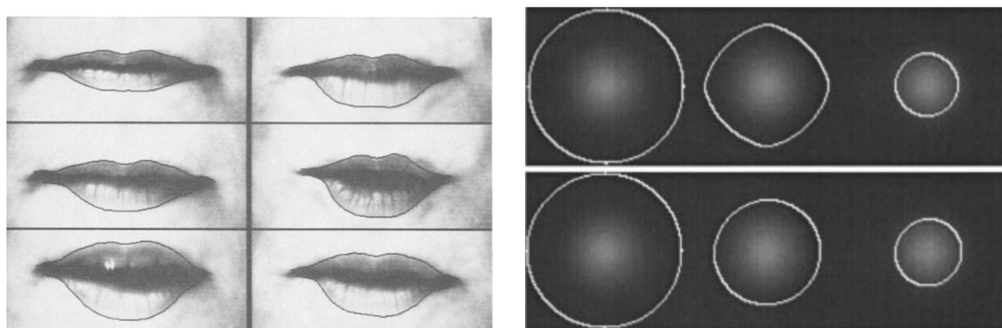


Figure 2.16: On the left, an example of Snakes by Kass et al [149]. On the right, Active Contours Without Edges by Chan & Vese [150].

2.4.3 Image Segmentation

Image segmentation is the act of clustering pixels in the same category. There are an array of segmentation methods that range in complexity. Segmentation has many uses in medicine, for applications such as processing MRI scans [154, 155, 156] and skin [157] and breast [158] cancer diagnosis. Segmentation can be achieved using a number of classic computer vision algorithms.

Watershed [159, 160, 161] is a morphological segmentation algorithm used on grayscale images. The goal of the algorithm is to identify contours to

enable segmentation of an image. Najman & Schmitt [162] have a succinct explanation of how Watershed segmentation works: “the idea of the watershed is to attribute an influence zone to each of the regional minima of an image (connected plateau from which it is impossible to reach a point of lower grey level by an always descending path). We then define the watershed as the boundaries of those influence zones”. An example of Watershed segmentation can be seen in Figure 2.17.

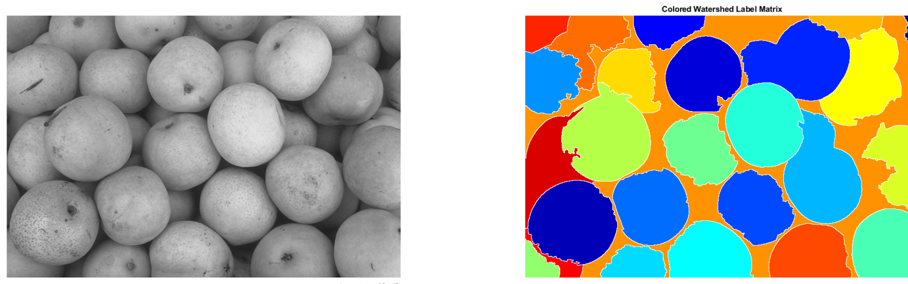


Figure 2.17: On the left, an image of fruit converted to grayscale. On the right, the same image with Watershed segmentation applied, adapted from [163].

Thresholding [164] is a simple form of segmentation that is used as an image processing technique to improve results when performing contour detection [142] and for document digitisation [165]. To apply a threshold to a grayscale image, a threshold value that is within the pixel value range must be selected. Any pixel in the image below the threshold is converted to white and any pixel above the threshold value is converted to black. Simple or Global Thresholding is the most straight forward version of thresholding, where the threshold value is applied to the entire image [166]. In Adaptive Thresholding [165], the algorithm calculates a threshold value for a number of small regions in the image. This accounts for differences in brightness across the image.

An example of this can be seen in Figure 2.18.

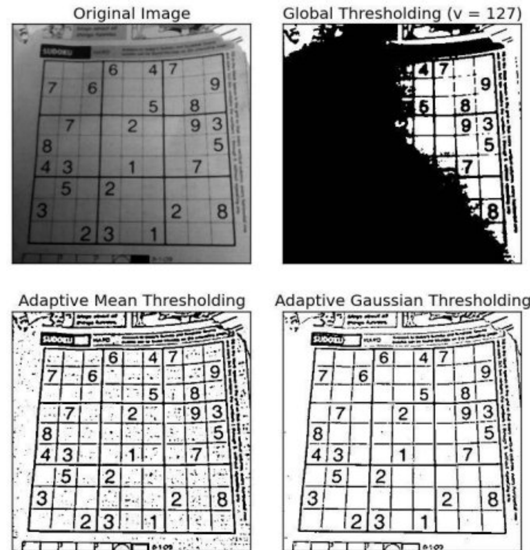


Figure 2.18: Examples of Global and Adaptive Thresholding, adapted from [167]

Clustering, like K-Means [168] or Fuzzy C-Means [169] is one of the simplest methods of image segmentation [170]. It works by clustering the pixels in an image into a set number of clusters. While this can be a limitation of clustering algorithms, Ng et al state that K-Means segmentation "is suitable for biomedical image segmentation as the number of clusters (K) is usually known for images of particular regions of human anatomy" [171]. K-Means Clustering can be seen in Figure 2.19.

In more recent publications, image segmentation through the use of deep neural networks such as Fully Convolutional Networks (FCNs) [172], which are similar to CNNs in architecture, have proven to be effective. An FCN is a CNN trained to perform pixel-wise semantic segmentation. Each pixel in an input image is classified as the background or an object from a set

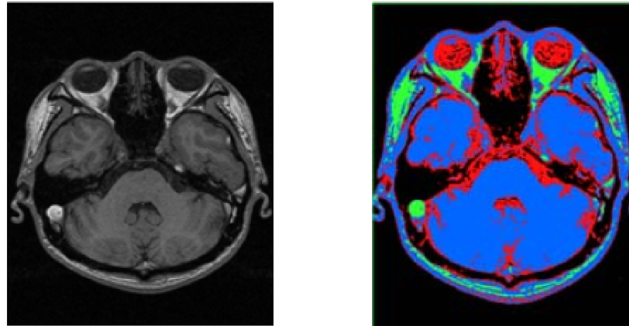


Figure 2.19: On the left, an MRI. On the right, an MRI after K-Means clustering. Adapted from [171]

of predefined categories. The architecture of a FCN can be seen in Figure 2.20. In a similar fashion to CNNs, each layer downsamples the input signal, using pooling layers, as it passes through the network to extract features. In the last layer of the network, the downsampled output is upsampled using a deconvolutional layer. This results in the final output image being the same size as the input image [112]. Long et al [172] show how they used the VGG-Net [116] and AlexNet [16] classifiers and "augment them for dense prediction with in-network upsampling and a pixelwise loss". They then train the FCN for segmentation by fine-tuning the network. U-Net [173], a modified FCN has shown promising results in biomedical segmentation applications. Ronneberger et al state that they modified the FCN [172] architecture to enable U-Net to work with very little training data and yield more precise segmentations. U-Net achieves this by having a more symmetrical network architecture that contains an equivalent number of upsampling layers as downsampling layers, when compared to the FCN's single upsampling layer reported by Long et al [172].

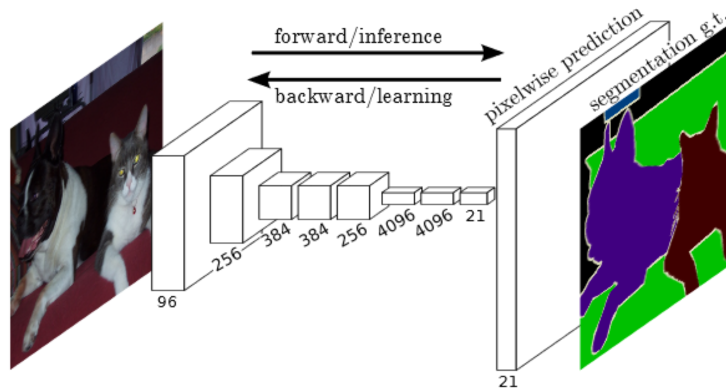


Figure 2.20: Architecture of a Fully Convolutional Network for semantic segmentation, adapted from [172]

2.4.4 Computer Vision Resources

A number of open source software libraries are available for developing deep neural networks and computer vision applications. A popular software library for computer vision is OpenCV [152]. OpenCV allows a developer to utilise the techniques reviewed in this section. For Optical Character Recognition (OCR), Tesseract OCR [174] is available. For more rich deep learning features like implementing the deep neural networks covered in this literature review, there are a set of deep learning frameworks. A number of these software libraries include TensorFlow [175], Keras [176], Torch [177], Caffe [178], Theano [179], Deeplearning4j [180], MXNet [181], CNTK [182] and Matlab [183]. There is also a selection of machine learning and data science oriented software libraries such as the SciPy ecosystem, which is a python-based ecosystem of open-source software for mathematics, science, and engineering [184]. The SciPy ecosystem contains a number of libraries useful in data science and machine learning, include the following. Pandas

[185], a data analysis tool. NumPy [186], a N-dimensional array library for creating efficient multi-dimensional containers of generic data. Scikit-Learn [187], a machine learning toolkit and Matplotlib [188], a 2D graph plotting library which can interface with Mathlab. Zacharias et al [189] provide a review of deep learning frameworks and state, based on metrics from GitHub, TensorFlow is the most popular open source deep learning framework, followed by Keras, Caffe, MXNet and then Theano.

There are a variety of datasets used for training and benchmarking deep neural networks. ImageNet [190] is a benchmark dataset for object category classification and detection. The dataset contains hundreds of object categories and millions of images. The dataset is used in the Large Scale Visual Recognition Challenge that has been running annually since 2010 [190]. MNIST [113] is a dataset of handwritten digits from 0 to 9, which contains 60,000 training samples and 10,000 test samples. The MNIST handwritten digits dataset is a subset of a larger dataset available from the National Institute of Standards and Technology (NIST). EMNIST [121] is an extension of the MNIST handwritten digit dataset. EMNIST or Extended MNIST is a dataset that contains digits, uppercase and lowercase handwritten letters. EMNIST is a normalised subset of the NIST Special Database 19 [191], which is the same database MNIST comes from. EMNIST contains a total of 697,932 training samples and 116,323 test samples. Fashion-MNIST [192] is a benchmark dataset that contains 70,000 images of fashion products from 10 categories. Like MNIST, Fashion-MNIST contains 60,000 training and 10,000 test samples. Fashion-MNIST is intended to serve as a direct drop-in

replacement for the original MNIST dataset for benchmarking machine learning algorithms. The CIFAR-10 and CIFAR-100 [114] are labelled datasets designed for training and benchmarking deep neural networks performing object recognition. The CIFAR-10 set has 6000 examples of each of 10 classes and the CIFAR-100 set has 600 examples of each of 100 non-overlapping classes [114].

Chapter 3

System Design

This chapter outlines the design of this research project's software component and discusses the various design decisions which were implemented to test the research hypothesis.

Section 3.1 is a review of the research project's software requirements. This section reiterates the problem this research project is aiming to solve. The system requirements from a user's point of view are laid out and examined.

Section 3.2 discusses the technologies utilised to develop this research project. This includes the programming language, software libraries and resources that were used in the prototype implementation.

Section 3.3 is a review of the overall design of this research project. This section covers the various working components of the software project, including descriptions of the function of each component and its role in the

prototype architecture.

Section 3.4 presents the implementation of this research project. This section covers the implementation details of each component of the project.

3.1 System Requirements

Before examining the system design of this research project's software component, this section outlines the initial project requirements and specification. The hypothesis underpinning this research is that computer vision and deep learning can be used to generate a JSON representation of a hand-drawn graph. This representation will include both the text from handwritten labels and the relationships between the nodes present on the graph. This JSON data can then be used for any purpose thereafter including storage, transformation and processing.

3.1.1 User Requirements

The proposed software program is required to convert an image of a hand-drawn graph into a JSON representation of the recursive data structure. The required functionality from a user's perspective is as follows: a given user must be able to draw a graph, be it a spider diagram or a brain storming bubble diagram, on a whiteboard or blank sheet of paper. The user can then capture an image of their drawing by digitising the image using some type of device. This image is given to the proposed software program, using any means that is deemed necessary. The proposed software program must

then parse the hand-drawn graph in the image. This process will infer the relationships between the nodes in the graph. The process will continue to interpret the handwritten text, if any is present, that is contained within each node in the hand-drawn graph. The results of this processing will then be compiled into a JSON representation that (1) describes the relationships between the nodes in the hand-drawn graph accurately and (2) contains an interpretation of the handwritten text from each graph node that is as accurate as is feasibly achievable. The scope of this proposed software program does not specify a use for the resulting JSON output, as the use for this output is dependent on contextual and situational requirements. Further integration of this software program into other services or programs is beyond the scope of this research project. However, there are obvious candidate use cases such as a usability tool for note taking software, integration with diagramming tools like Visio or draw.io or for the digitisation of handwritten notes which include hand-drawn graph diagrams.

3.1.2 Project Constraints

As this is a purely research based project and non-commercial, all software and accompanying resources required are to be open source or in the public domain. Based on the research conducted thus far, this constraint is not believed to be a limitation. This is due to the healthy and vibrant open source community around the research domain and the development of deep learning and computer vision projects. As demonstrated by the review of open source deep learning frameworks provided by Zacharias et al [189], there is a variety

of quality open source software frameworks available for developing deep learning software. Another project constraint is the availability of suitable hardware. In the case of training deep neural networks, the process can be a very demanding task requiring powerful hardware, primarily graphics card (GPU) compute power. The available hardware for this project is the following:

- An Intel i7 4790k @ 4.00GHz CPU.
- 16GBs of DDR3 RAM @ 2133 MHz.
- A Nvidia GeForce GTX 1080 8GBs GPU.

This limited suite of hardware resources was used to implement, train, test and benchmark all of the software developed to test the research hypothesis.

3.2 Technologies

To develop the functionality for the proposed software program, a variety of techniques were required. As the project's pivotal feature is interpreting a captured image of a hand-drawn graph, some form of image processing is clearly necessary. This lead to the selection of the software library OpenCV [152] for computer vision related image processing tasks. The second defining feature is the reading of handwritten labels present on a graph. To implement this functionality, an obvious candidate is deep learning, due to the state of the art results that can be achieved with these techniques. This lead to the selection of TensorFlow [175] and Keras [176] for deep neural network

development. This decision was informed by the review that Zacharias et al [189] provided of open source deep learning frameworks, where TensorFlow and Keras ranked as number one and two respectively on their list of most popular frameworks. When determining a dataset for training the text classification deep neural network, the dataset EMNIST [121] was selected. EMNIST, a relatively new dataset at the time of writing, provides a large selection of hand written characters and digits for training. The operating system chosen for development was Ubuntu 18.04 LTS and the programming language selected was Python. The primary motivation for these choices is the fact that all of the mentioned software can be developed in the Python programming language and will run on the Ubuntu operating system. Ubuntu 18.04 LTS is the operating system of choice due to it being free and open source, satisfying the project constraint of only using free open source software.

OpenCV (Open Source Computer Vision Library) [152] is a computer vision software library that contains "several hundreds of computer vision algorithms" [152]. OpenCV can be utilised to perform many computer vision tasks such as image processing, feature detection and segmentation. TensorFlow [175] is a machine learning and deep learning framework that allows for the development of deep neural networks. Keras [176] is a high level API abstraction, written in Python, that runs on top of TensorFlow, CNTK [182] or Theano [179]. Keras is designed to make developing deep neural networks faster by reducing the lines of code required to create a deep neural network. The EMNIST [121] dataset contains a collection of handwritten English char-

acters and digits. The digits are the same as the MNIST [113] handwritten digit dataset. EMNIST contains a total of 697,932 training samples and 116,323 test samples covering 62 classes.

3.3 System Design

With the technologies selected, the architecture of the processing steps involved in the graph image analysis was developed. A diagram of this set of processing steps, or work flow, for the prototype software can be seen in Figure 3.1. The five processing steps that the software can perform are:

1. Capture a graph drawn on a whiteboard or blank sheet of paper.
2. Perform computer vision processing to parse the graph.
3. Utilise a deep neural network to classify the extracted labels.
4. Generate a JSON representation of the parsed graph.
5. Store the JSON in a database for use thereafter.

With the processing steps of the software determined, the design was broken down into a number of components all responsible for a specific task. The prototype design consists of six separate components which include: a graph processing component, a node processing component, a text classification component, a text classification training component and a training data pipeline for text classification training data. The final component is the core service that manages the previous five components and generates

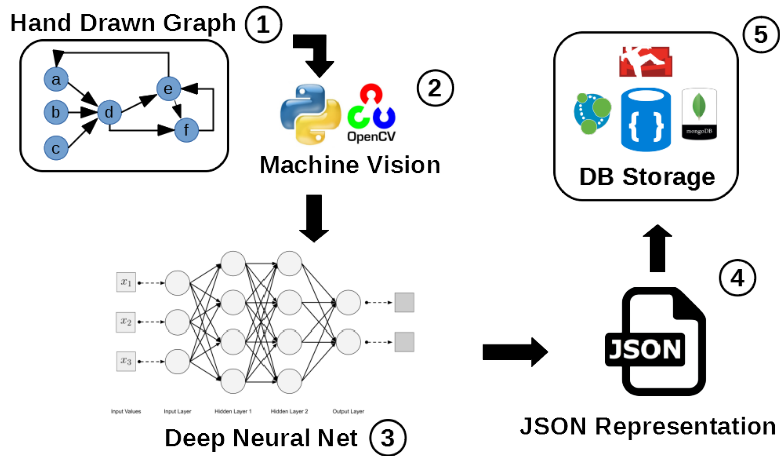


Figure 3.1: System diagram of the prototype software work flow.

the JSON representation of the drawn graph. A diagram of the prototype system design can be seen in Figure 3.2.

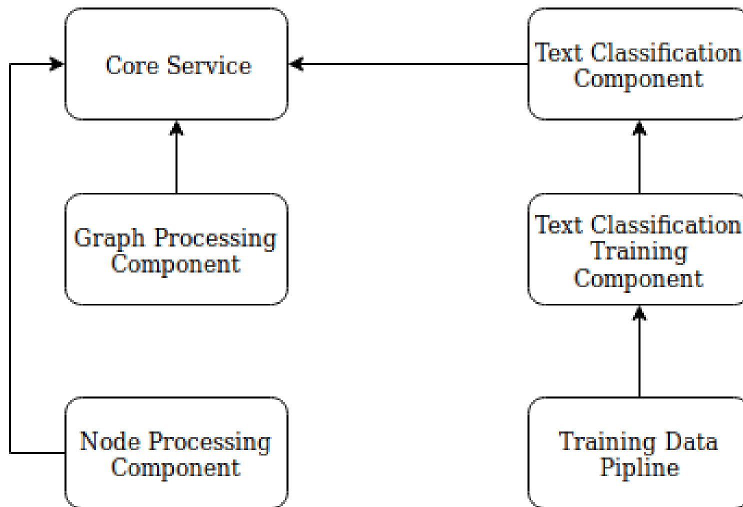


Figure 3.2: Diagram of system components.

A significant challenge of the prototype was deciding how the classification of the handwritten text within the nodes of the graph was going to be

achieved. Once this design challenge was overcome, the next major issue to address was how the graph was going to be parsed and the relationships between nodes inferred. To accomplish this, the design process was split up into discrete stages, namely achieving handwritten text classification, parsing the hand-drawn graph and inferring the relationships between nodes in the process. The ultimate stage was tying the components together and generating a JSON representation of the processed graph and then storing it in a database.

3.3.1 Handwritten Text Classification

The first major challenge to overcome was reading the handwritten labels on the user's hand-drawn graph. This classification problem has a number of solutions, including Optical Character Recognition (OCR) or deep learning. The Tesseract OCR library [174] was investigated as a solution to solve the text classification problem. After a period of testing, it was determined that while Tesseract works very well with document scanning and computer typed text recognition, it ultimately performed too poorly with handwritten text classification for it to be considered a viable option. This lead to the decision to utilise deep learning to solve the problem. This decision was aided by the state of the art results produced by Convolutional Neural Networks (CNNs) with classification tasks on image data. Various CNN implementations such as VGG-Net [116], AlexNet [16], Faster R-CNN [119], YOLO [120] and SSD [14] have all shown impressive results in the area of image recognition. The goal was to implement a CNN inspired by VGG-Net but modified to be

smaller. This is due to a variety of factors like the simplicity of the design, the reduced number of classes being classified and the limited hardware available to train a larger CNN in a feasible amount of time.

The next step was choosing a suitable dataset for training the proposed CNN to classify handwritten text. At first, the MNIST [113] handwritten digit dataset was used as a substitute during the proof-of-concept stage. This is due to the speed at which a neural network can be trained on the MNIST dataset. This stage of prototyping was important for getting a smaller scale CNN training and classifying from start to finish before spending a considerable amount of time training a larger network on a much larger dataset. As stated in the previous section, the EMNIST [121] handwritten character and digit dataset was selected as the main training resource for the handwritten text classification CNN. To load and use the images in the EMNIST dataset, the development of a data pipeline was necessary. This was another reason for first testing the CNN with MNIST data. Due to the popularity of the MNIST dataset and its status as a benchmark, TensorFlow contains a ready-made data module for using the MNIST dataset in training. This is not the case for the EMNIST dataset and thus a data pipeline was developed first, before training on the dataset could commence.

3.3.2 Graph Parsing & Relationship Inference

The second major challenge to overcome was the parsing of the hand-drawn graph and the inference of the relationships between the graph nodes. OpenCV [152] was utilised to achieve these development goals. The objective

of this stage of project design was to detect the nodes and edges of the hand-drawn graph and then to extract the characters from the handwritten labels in the detected nodes. These extracted characters would then be passed to the classification CNN for classification. The first task to complete was the extraction of labels from the detected nodes in the graph. This allows for the extension of the functionality of the software and the utilisation of the now trained CNN. For a given image with a handwritten label on it, the goal was to detect the characters in the image, determine the order in which they appear and infer where spaces may be, if any are present in the string of text. Based on the location of each detected character, an image of each character can then be captured and stored. Once stored, each character can be fed to the CNN for classification. Before any of the outlined operations can be undertaken the image must be processed. This processing may include converting the image to grayscale, Gaussian blurring and thresholding.

At this stage in the design process, handwritten text in an image can be extracted and classified. The next step was to parse a fully hand-drawn graph. This involves detecting the nodes and edges, allowing for the inference of the relationships between the nodes. Once the nodes are detected, they can then be passed to the handwritten label extraction component outlined in the previous paragraph. This step requires two distinct phases; detecting graph nodes and edges and then detecting which nodes are connected to the detected edges to record the relationships between the nodes. The first phase requires the detection of all objects in the image and the employment of a heuristic to differentiate the nodes and edges. The software must then

infer which nodes each edge is connected to. This can be achieved using the Euclidean distance [193] calculation. Images of each node can then be captured and stored, allowing them to be passed to the label extraction component. The node relationship data can also be stored and used later when constructing a JSON representation of the graph after the characters in the handwritten labels are classified.

3.3.3 Building the JSON Representation

The last stage of the design process was to combine all of the previously presented components together and construct a JSON representation of the hand-drawn graph. To briefly reiterate the proposed work-flow of the software, the process will consist of the following: an image of a hand-drawn graph is captured and sent to the core service component of the prototype. This service delegates the tasks required to each component. It passes the captured image to the graph processing component. The graph parsing component processes the image, by applying the necessary operations and filters to the image. This component then detects the nodes and edges in the image. The relationships between each node is inferred and stored. Images of the nodes are then cropped, stored in a collection and returned to the core service with the saved relationship information. The core service iterates through each node image, in the collection of nodes, and passes each node image to the node processing component which performs label extraction. See a sequence diagram of this process in Figure 3.3.

Label extraction extracts the characters from the image of the node, which

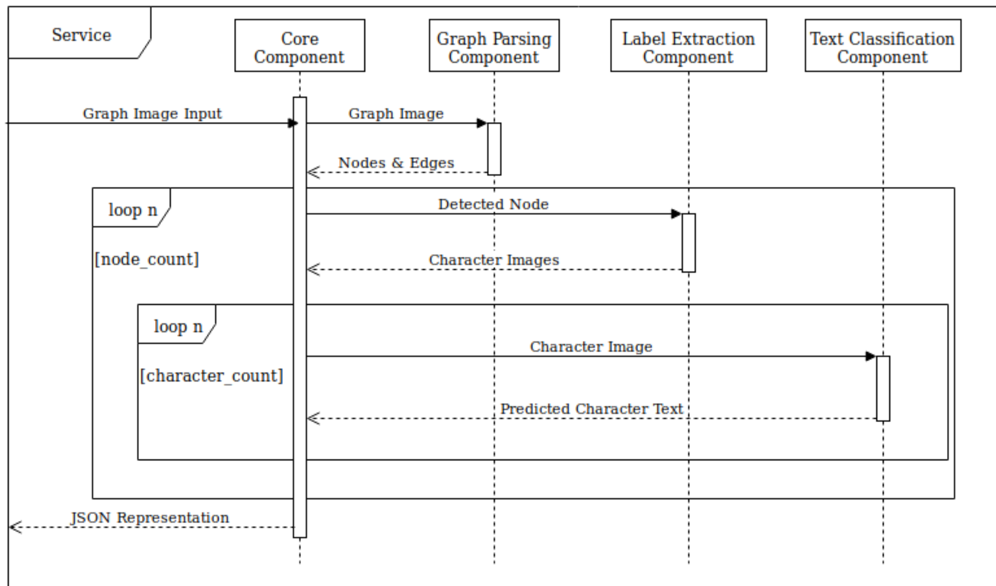


Figure 3.3: UML Sequence diagram of the system.

contains a string of handwritten characters. Each character is cropped into its own image and any potential spaces in the string of text are inferred using a distance-based heuristic. The images of each character are cropped and values used to denote spaces are saved, in correct sequential order, to a collection. This collection is then returned to the core service. The core service then passes each image of the character to the handwritten character classification CNN which returns the prediction of what the handwritten character in the image is. The values returned for each node image are saved in an object which represents their respective node. Once the labels are extracted from every node image and classified, each node object is linked to any node it has a relationship with based on the relationship information inferred by the placement of edges in the hand-drawn graph. Once all nodes are linked to their adjacent nodes, the JSON representation is constructed

and saved.

3.4 System Implementation

This section is a walk-through of the implementation details of the proposed software program. This section is segregated into six sections, one for each of the components the software prototype is decomposed into. While the previous section dealt with the design at a high level, the following section examines how the software was written. The first step is passing a captured image of a hand-drawn graph to the proposed software program. For demonstration purposes, this image is loaded based on a path to a file in the file system of the operating system. When integrating this software as a service for example, the captured image could be sent to a web server, via a HTTPS POST, which could pass the image to the proposed software program. When the captured image is loaded, it is passed to the graph processing component.

3.4.1 Graph Processing Implementation

The goal of the graph processing component is to parse the image of a hand-drawn graph and detect graph nodes and edges. When the image is received, the first set of operations performed are a number of image processing techniques to aid the detection of nodes and edges in the image. All of these image processing operations are performed using the OpenCV computer vision software library. First, the image is resized to the dimensions of 1000 by 500 pixels. This is done in order to instil consistency on the subsequent processing steps. Next, the image is converted to grayscale [128] and then

a Gaussian blur filter [129] is applied to the image to remove noise. The original image and the image after it has been converted to grayscale and filtered with a Gaussian blur, can be seen in Figure 3.4.

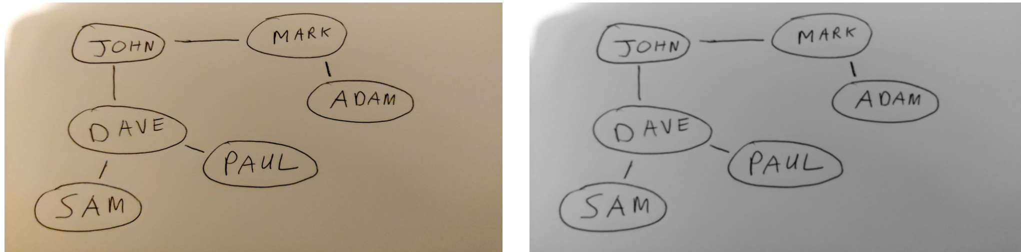


Figure 3.4: On the left, the original image loaded into the software. On the right, the same image converted to grayscale with a Gaussian blur applied to reduce noise.

After removing noise with the Gaussian blur filter, adaptive thresholding [165] is applied to the image. Removing noise from the image first, using a Gaussian filter, is necessary for yielding cleaner results from the adaptive threshold operation. The difference between the results obtained from adaptive thresholding without first removing noise and after removing noise can be seen in Figure 3.5 and Figure 3.6 respectively. This example expresses the importance of removing noise from images before attempting to perform feature detection.

The next step is to detect any contours [153] in the processed image after thresholding has been applied. Contours are detected using the *findContours()* function in OpenCV. When calling this function with the *RETR_TREE* parameter, a hierarchical tree showing which contours are contained within each other is returned with the list of all detected contours in the image. This contour hierarchy will enable the identification of root contours within the

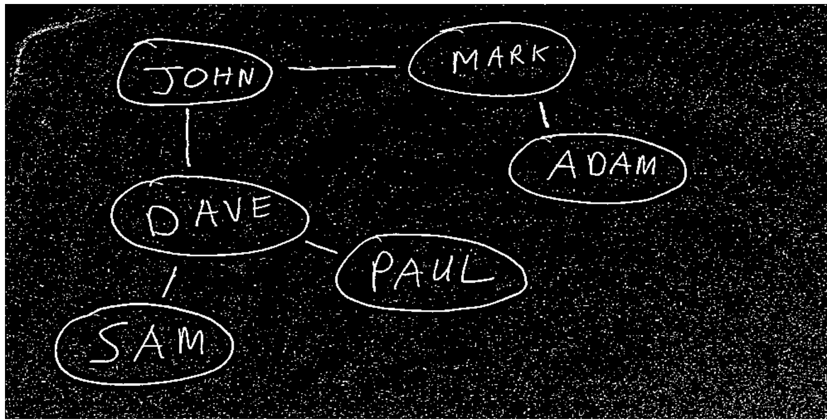


Figure 3.5: The grayscale image with adaptive thresholding applied before noise is removed with a Gaussian filter.

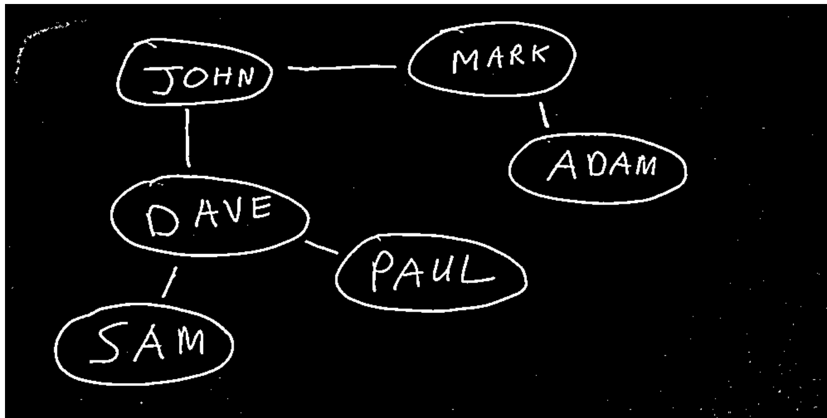


Figure 3.6: The grayscale image with adaptive thresholding applied after noise has been removed with a Gaussian filter.

image. This will enable the differentiation between the node and edge contours and the contours representing the handwritten labels contained within each node contour. The detected contours and the contour hierarchy are saved in a collection for further processing. The contours detected in the image of the hand-drawn graph can be seen in Figure 3.7. The first stage of processing the contours requires eliminating any noise that may have been detected. When looking at Figure 3.7, detected noise can be seen in the

bottom right corner of the image, among other places, in the form of spots. To solve this problem a heuristic is employed to eliminate potential noise from the list of contours detected in the image. The heuristic is: ignore any detected contours whose area is less than 40 pixels as these are most likely background noise that have persisted in the image following the adaptive thresholding. This heuristic was developed through trial and error and provides acceptable results. All other contours are added to a collection of candidate contours for further processing.

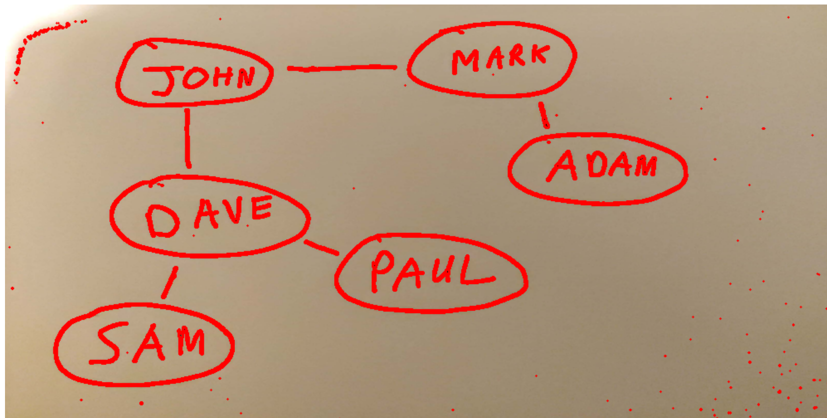


Figure 3.7: The original image with all detected contours superimposed, as red pixels. Detected noise can be seen in the top left and bottom right corners of the image.

After obtaining a collection of candidate contours, each candidate contour is evaluated, categorised and catalogued. The aim of this process is twofold: to separate the valid node and edge contours from everything else and to catalogue which contours are contained within each other. Cataloguing the hierarchy of contours is important for two reasons. The first is that only the outer most root parent contour will be either a node or an edge. The second reason is that all child contours of a node are required when extracting the

handwritten text labels from each node. It is at this stage that these child contours are identified. To catalogue the hierarchy of contours and identify the root parent contours, two lists are maintained. The first list is a collection of all contours identified as invalid. A contour is an invalid candidate for a node or edge if it is contained within another contour. The second list is a map, which relates each parent contour to all of its child contours. In this map, all candidate contours are saved as a parent and using the saved hierarchy information obtained when the contours were detected, all of the parent's child contours are saved. If a contour is a child of a parent, it is added to the list of child contours for that parent. At the same time, the child contour is also added to the list of invalid contours if it is not already present. This is because, as the child is contained within the parent, it is an invalid candidate for a node or edge contour. After this process is complete, all candidate contours are root parent contours which have a list of all of their child contours. The detected candidate contours, with all child and noise contours filtered out, can be seen in Figure 3.8.

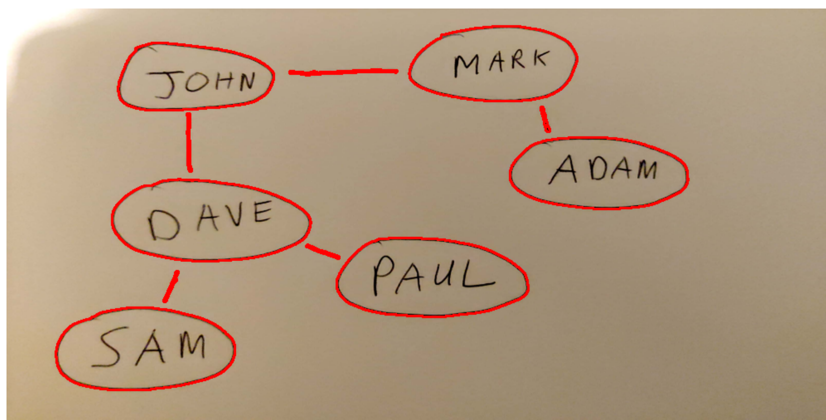


Figure 3.8: The original image with all detected root parent contours superimposed, as red pixels. Detected noise has been removed using the area-based heuristic.

After this exercise is completed, there are two lists which can be utilised to infer a great deal of information. At this point in the execution of the prototype, it can be assumed that any candidate contour which is not present in the list of invalid contours is either a node or an edge. It can also be assumed that, for a node contour, if that node contains a handwritten label, the contours that represent the handwritten text are contained within the list of children for that given node contour. The next task is to differentiate between node and edge contours. To separate nodes from edges, the geometric calculation for circularity is utilised. If the candidate contour is a perfect circle, its circularity would be 1. If the candidate contour was square, its circularity value would be 0.785. The equation for calculating the circularity [194] of a two dimensional geometric shape is defined as:

$$c = (4\pi a) \div p^2 \tag{3.1}$$

Where c is circularity, p is the perimeter and a is the area of the shape. To calculate the circularity of the candidate contours, the area and perimeter of the contour is calculated first and then the circularity is calculated. It was found that, through testing, any shape with a circularity value of 0.5 or higher can be considered a graph node and all other candidate contours can be considered graph edges. If a candidate is deemed a graph node, it is given an index number, based on the order in which it was detected, and it is saved in a list of graph nodes. If a candidate is deemed a graph edge, it is saved in a list of graph edges. In Figure 3.9, contours classified as nodes can be seen coloured as red and contours classified as edges can be seen coloured

green.

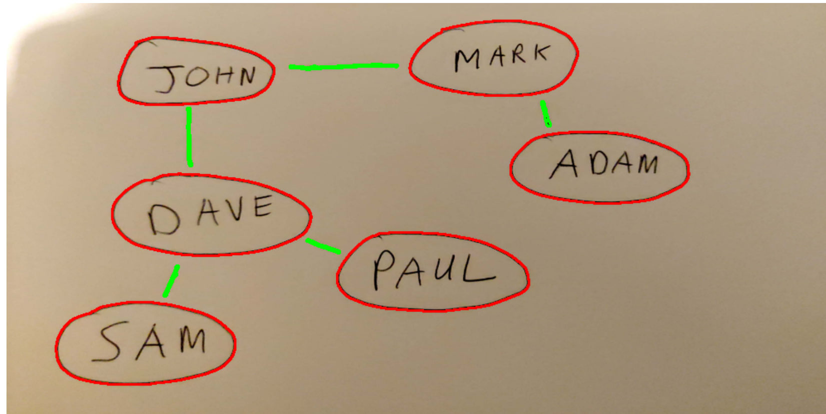


Figure 3.9: The original image with all detected node and edge contours superimposed. Contours classified as nodes can be seen coloured red. Contours classified as edges can be seen coloured green.

The next step is to infer which nodes are connected via edges, which will result in the interpretation of the relationships between nodes in the graph. To achieve this, the two ends of each hand-drawn edge must be found and then the closest node to each of those ends must be determined. It can then be assumed that, of all nodes, the node closest to one end of an edge is the first node that edge is pointing at. The node closest to the other end of that edge is the second node that edge is pointing at. Therefore, these two nodes can be considered connected by that given edge.

To find both ends of an edge contour, the two farthest points from each other are found on the contour. As a contour in OpenCV is a vector of points defining a shape in two dimensional space, the distance between each point, from every other point in the vector, is calculated. A list of objects, containing two points and the distance between them is then saved. Once

the distance between every point in the edge contour is calculated, the list of distances is sorted based on distance and the list is sorted in ascending order. See Figure 3.10 for a sequence diagram of this process. The set of points with the largest distance between them is then selected from the end of the ordered list. These two points are considered the two ends of the selected edge. Once both ends of an edge are defined, each saved node is iterated through. For each node, the centre X and Y coordinates are measured and, based on these coordinates, the distance between every node and each end of the selected edge is calculated and saved. Two lists are maintained, one list for each end of the selected edge. The list that represents end A, is populated with objects that contain a node and its respective distance from end A. The list that represents end B contains a similar list, of all the nodes and their distances from end B. When the distance of every node from each end of the selected edge is calculated, both lists are sorted in ascending order based on distance. The first node in each list is selected and these nodes will have the shortest distance to each end of the selected edge. Each node has an index value saved to it when it is identified. The indices of both nodes are saved onto the object that represents the node edge. This process is repeated for every node edge in the saved list of node edges.

The Euclidean distance [193] calculation, used to find the distance between two points, is used to find the distance between both ends of a detected edge and the distance between each node and each edge end. Euclidean distance is defined as:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{3.2}$$

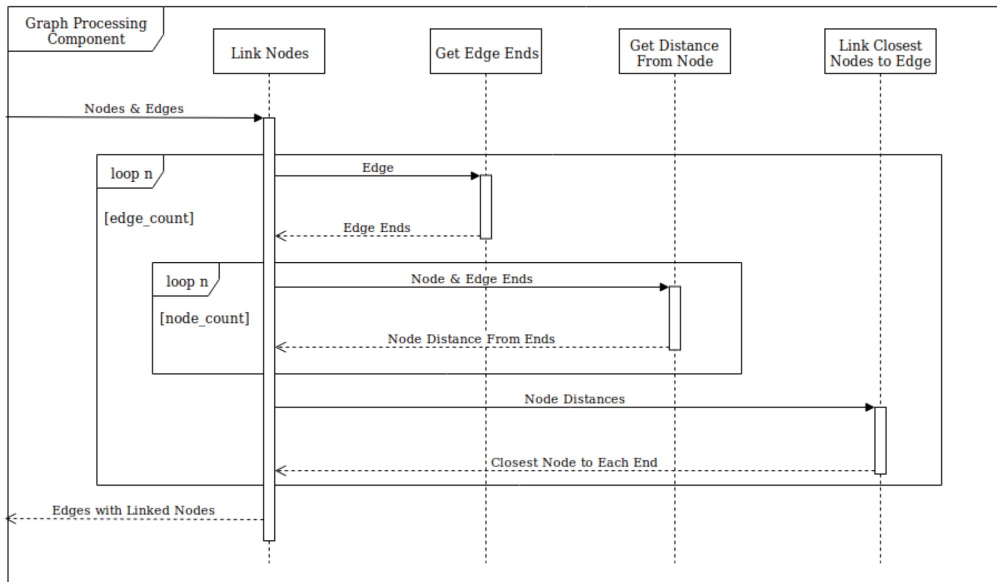


Figure 3.10: UML Sequence diagram, showing the process behind linking nodes to their edges.

Where x_1, y_1 are the X and Y coordinates of point 1, x_2, y_2 are the X and Y coordinates of point 2 and d is the distance between point 1 and point 2.

With this task completed, the program maintains a list of the following items.

1. A list of all detected nodes, which is populated with objects containing the node contour and an assigned index.
2. A list of all detected edges, which is populated with objects containing the edge contour, the index of connected node A and the index of connected node B.
3. A list containing a map of all candidate contours and their child contours.

The last task is to create a list of all graph nodes, including their node index and all of their child contours. This is achieved by iterating through each of the nodes saved in the list of detected nodes. Each node is compared against the contours contained in the list of parent contours. If a parent contour is found that matches the node contour, the node, its node index and its child contours are all saved in an object and added to a new list of graph nodes. Once this is complete, the graph processing component returns the list of graph nodes and the list of graph edges to the core component. At this point of execution, graph processing finishes and the core component contains just two lists: a list of all nodes including their node index and child contours, and a list of all edges including the edge contour and the indices of the two connected nodes. The relationships between nodes can now be mapped. The next task is to extract the handwritten labels from the nodes.

3.4.2 Graph Node Processing Implementation

This section discusses the implementation of the graph node processing component. The goal of the node processing component is to extract the handwritten characters from the labels contained within each of the nodes detected in the image of the hand-drawn graph. After the core service receives the detected graph nodes and edges from the graph processing component, each node is passed to the node processing component one at a time with the original captured image of the graph. Due to the node processing component's ability to function independently, the image of the graph is converted to grayscale, blurred using a Gaussian filter and adaptive thresholding is ap-

plied in the same manner as the previous component. Once this is complete, one of two processes are executed.

As stated, the node processing component can operate independently, so if a graph node is not provided to the component, it will look for text in the image. This takes place in a similar fashion to the graph processing component. Contours in the image are detected and saved to a list. This list is then filtered using the area-based heuristic that was presented earlier. If the calculated area of a contour is less than 40 pixels, it is ignored and all other contours are added to a list of candidate contours. Before saving the candidate contours, a rectangle around each contour is calculated using the *boundingRect()* function in OpenCV. This function calculates the X and Y coordinates, width and height of a rectangle that fits perfectly around a given contour. When saving the candidate contour, the rectangle information is also saved for later use. Therefore, the collection of candidate contours is a collection of objects that contain the following information: the X coordinate, Y coordinate, width and height of the calculated rectangle around the contour and the contour itself. A candidate contour in this context is a contour that may be a handwritten character contained within the image. An image of the contours detected in the image of the handwritten label can be seen in Figure 3.11.

If a detected graph node is passed to the node processing component, the node's child contours will be searched for candidate contours that may constitute the handwritten label contained within the image of the node.

Figure 3.11: On the left, an image of a handwritten label. On the right, the detected contours with red rectangles drawn around them. The image on the right highlights the first challenge of extracting the characters from a handwritten label. Multiple parts of a single letter are being detected. This can be seen where a rectangle is contained within another rectangle.

Before this can happen, one issue must be resolved. In the graph processing component, a detected node contour and all of the contours nested within it are saved. Due to how the contours are detected, the outline of the hand-drawn graph node is detected as a child contour. To overcome this issue, a heuristic based on the area of the node contour is employed. The area of the node contour is calculated and saved. Then, the area of each child contour is calculated and if the child contour is not 20% smaller than the parent node contour it is ignored. Therefore, unless child contours are 80% the area of the detected node or smaller, they are excluded from being a candidate. In practice, this developed heuristic provides a reasonable degree of accuracy. Figure 3.12 shows an example of handwritten letters within a node and the node itself being detected and an example of the area-based heuristic excluding the previously detected node contour. Once all valid candidate contours are saved into a collection, the next stage of processing can start.

The next stage of processing is to eliminate cases of multiple parts of a handwritten letter being detected. This issue has been shown in Figure 3.11. The solution is to implement a parent-child analysis which consists of

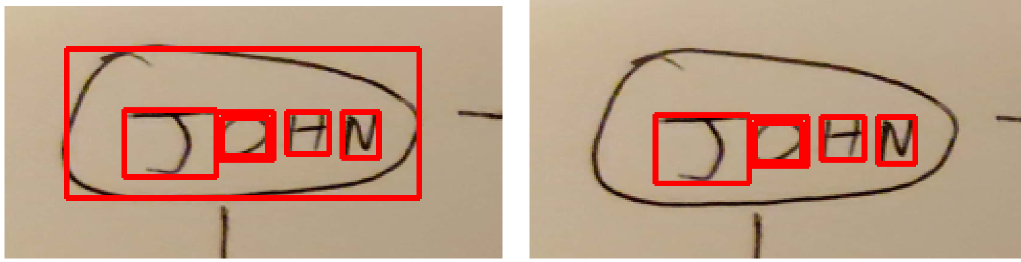


Figure 3.12: On the left, an image of the characters detected for a given node in the hand-drawn graph. The outline of the node itself is also detected as a child contour. On the right, the same contours are detected but the node contour itself is excluded due to the use of an area-based heuristic.

calculating, based on the calculated rectangle information, if one contour is contained within another. If a contour is contained within another at this point, it can be considered an invalid candidate and saved in a list of invalid contours. This process checks each candidate against all other candidate contours. A new list of valid characters is then created and any candidate that is not present in the list of invalid contours is added to it. This is a separate step that is necessitated by the nature of hierarchies. One cannot know if a contour is a parent until all potential child contours are identified. The result of this pruning can be seen in Figure 3.13.

Figure 3.13: On the left, an image of a handwritten label with multiple parts of a single letter are being detected. On the right, the same label after child contours have been eliminated.

To calculate if one contour rectangle is contained within another, which can be seen in Figure 3.13, the following boolean logic can be utilised:

$$j_1 = x_1 + w_1 \tag{3.3}$$

$$j_2 = x_2 + w_2 \tag{3.4}$$

$$k_1 = y_1 + h_1 \tag{3.5}$$

$$k_2 = y_2 + h_2 \tag{3.6}$$

$$W = ((j_1 \geq x_2) \wedge (x_2 \geq x_1)) \wedge ((j_1 \geq j_2) \wedge (j_2 \geq x_1)) \tag{3.7}$$

$$H = ((k_1 \geq y_2) \wedge (y_2 \geq y_1)) \wedge ((k_1 \geq k_2) \wedge (k_2 \geq y_1)) \tag{3.8}$$

$$C = W \wedge H \tag{3.9}$$

Where x_1, y_1, h_1 and w_1 are the X coordinate, Y coordinate, height and width of the first rectangle respectively and x_2, y_2, h_2 and w_2 are the X coordinate, Y coordinate, height and width of the second rectangle. It can be said that if C evaluates as *True*, the second rectangle is contained within the first rectangle.

At this point in execution, the prototype contains a list of all valid characters. The next stage of processing is to order the detected characters in the correct order, from left to right. This step is necessary, as the order in which contours are detected in an image is not guaranteed. The list of valid character contours is ordered in ascending order based on the X coordinate of the contour. This ensures that the characters are in the correct order, with

the first character being the first entry in the list, the second is the second entry and so on. The next step is to determine where the spaces in the text are, if any are present at all. This is achieved by calculating, based on the X coordinate and width of the rectangle surrounding the contour, how many pixels are between each detected letter. This calculation is given as:

$$d = x_2 - (x_1 + w_1) \tag{3.10}$$

Where x_1 and w_1 are the X coordinate and width of the first letter respectively, x_2 is the X coordinated of the second letter and d is the calculated distance. The distance is calculated for each character contour and saved in a collection of distances. A heuristic is then utilised to infer between which character a space may be. If the space between two characters is greater than the median space between characters, multiplied by 1.6, then there is a space between those two characters. This heuristic was fine tuned through empirical testing and it was found that a space between characters 0.6 times greater than the median space almost always differentiated the boundary between words in a handwritten label.

The last step is to prepare the detected characters for classification and then return them to the core service. The images of the detected characters need to be processed to aid the classification of the character. To do this, the images of the characters have an extra padding added to them. This extra padding of 6 pixels makes the character clearer, which in turn will make the classification of the character more accurate. When adding the extra

padding, the objective is to have the image square, so the height and width of the image are the same number of pixels. To accomplish this, a padding of 6 pixels is added to the largest dimension, i.e. if the image has a larger height than width, the padding is added to the top and bottom of the image. If the width is larger than the height, the padding is added to right and left of the image. Extra padding is then added to the opposite sides, based on the difference in pixels between the height and width of the image. This results in the image being transformed into a perfect square. An image of a character before and after the border is added can be seen in Figure 3.14.

Figure 3.14: On the left, an image of a detected character without a border added. On the right, the same character, with a border padding added to improve visibility and make the image square.

When the images of the characters have had their borders added, each image is added to a list of images, in the correct order based on their X coordinate. When adding these images into the list of images, for every space that was predicted using the median distance-based heuristic, a value

that is not an image is added to the list of images to denote that a space in the text is present. This will be used in the later stages of processing, after the images of the characters have been classified and a text string is constructed to represent the handwritten labels in each node.

3.4.3 Handwritten Text Classification Implementation

This section presents how the handwritten text classification component was implemented. After the handwritten labels from each graph node have been extracted and separated into separate characters, each character is passed to the text classification component, one by one, to be classified. To classify the handwritten characters, a deep convolutional neural network [107] was designed and trained. This section will examine the design of the CNN and the classification process. The subsequent sections will then outline the training of the CNN.

When an image of a handwritten character is passed to the text classification component, a number of operations take place. First, the model representing the deep convolutional neural network and the trained network weights are loaded from external files. The model of the deep convolutional neural network is saved to a file after training has been completed. The state of the deep neural network's weights is then saved to a separate file. Loading the model from a file in the classification component affords the flexibility of changing the model out for an updated version without having to edit the application code. Therefore, if any changes are made to the model, it can be updated while the application is live and not impact the operation of the

service. This flexibility also applies to the trained state of the model. As the weights are loaded from a file, if the network is trained on new data, the newly trained state can be loaded into the live neural network without changing application code. This can provide benefits such as improving the classification accuracy of the deep neural network while the service is live. The last file loaded is the set of character mappings, which relate the output node to its associated character in text format. Once these mappings are loaded, the program can output the character that the CNN has predicted, as a text string that is user readable.

The next step is to resize the input image of the character to match the size of the input layer in the CNN. As the dataset the CNN was trained on contained images of characters that are 28 by 28 pixels, this was the size chosen for the input layer. After the input image is resized to be 28 by 28 pixels, it is converted into a tensor. This tensor is two dimensional and 28 by 28, representing each pixel of the input image. The values in the tensor are converted to 32 bit floating point numbers and normalised to be in the range of 0 to 255. These values represent the grayscale values of each pixel. The last step is to pass the input to the model and predict which character is in the image. The model's softmax output returns the probability for each potential character. A max function is called to obtain the node with the highest probability and the node is passed to the output mapping to get the user readable text character. The model also outputs a confidence value between 0% and 100%. The predicted character text and the confidence value are added to an object and returned to the core service. This process

is repeated for each character extracted from the handwritten labels in every detected node.

The overall design of the convolutional neural network was adapted from the design of VGG-Net [116]. The designed neural network is a variation of the design of VGG-Net but with a much smaller number of nodes. For instance, the designed CNN contains 9 convolutional layers, whereas VGG-Net contains 13. A diagram of the architecture of the deep convolutional neural network designed and employed in this software project can be seen in Figure 3.15.

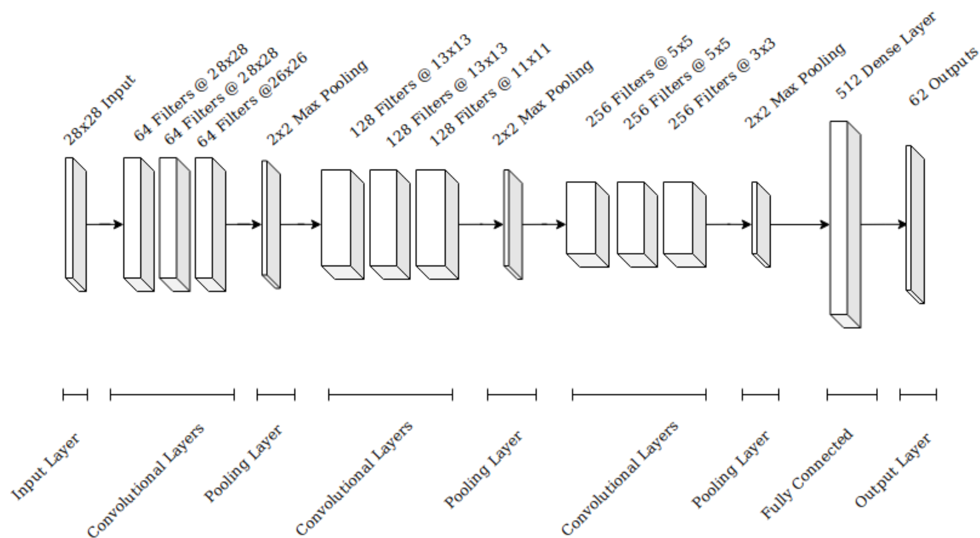


Figure 3.15: The architecture design of the deep convolutional neural network designed to classify handwritten characters.

The designed CNN contains an input layer, three convolutional layers, followed by a max pooling layer [110]. This pattern is repeated three times, creating a total of 9 convolutional layers and 3 max pooling layers. The last

max pooling layer is connected to a fully-connected layer which contains 512 nodes. This fully-connected layer is in turn connected to a softmax output layer which contains 62 nodes, representing the number of potential classes the network can classify. The first group of three convolutional layers have 64 filters or activation maps and have the same dimensions as the input layer. The dimensionality of the convolutional layers are reduced by the first max pooling layer, which operates with a filter of 2 by 2. This results in the next set of three convolutional layers having dimensions of 13 by 13 nodes. This second group of convolutional layers have an increased number of activation maps, enumerating to 128. After the second group of convolutional layers, there is another max pooling layer operating with a 2 by 2 filter. This pooling layer reduces the dimensions to 5 by 5 nodes. This third layer contains 256 activation maps. After the third pooling layer, which operates with a filter of 2 by 2, the dimensions are reduced to 1 by 1 nodes for 256 activation maps. This structure is flattened and connected to a fully-connected layer containing 512 nodes. These nodes are then connected to the output layer which contains 62 nodes, one for each classifiable class. The output layer utilises softmax [44] activation. Every other node in the convolutional neural network utilises the ReLU [45] activation function.

A detailed breakdown of the network layers, the output shapes of those layers and the number of parameters can be seen in Table 3.1. This information is summary information given from the saved Keras [176] model. Layers referring to dropout [73] are related to training and will be covered in the subsequent sections.

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 28, 28, 64)	640
conv2d_2 (Conv2D)	(None, 28, 28, 64)	36928
conv2d_3 (Conv2D)	(None, 26, 26, 64)	36928
max_pooling2d_1 (MaxPooling2D)	(None, 13, 13, 64)	0
dropout_1 (Dropout)	(None, 13, 13, 64)	0
conv2d_4 (Conv2D)	(None, 13, 13, 128)	73856
conv2d_5 (Conv2D)	(None, 13, 13, 128)	147584
conv2d_6 (Conv2D)	(None, 11, 11, 128)	147584
max_pooling2d_2 (MaxPooling2D)	(None, 5, 5, 128)	0
dropout_2 (Dropout)	(None, 5, 5, 128)	0
conv2d_7 (Conv2D)	(None, 5, 5, 256)	295168
conv2d_8 (Conv2D)	(None, 5, 5, 256)	590080
conv2d_9 (Conv2D)	(None, 3, 3, 256)	590080
max_pooling2d_3 (MaxPooling2D)	(None, 1, 1, 256)	0
dropout_3 (Dropout)	(None, 1, 1, 256)	0
flatten_1 (Flatten)	(None, 256)	0
dense_1 (Dense)	(None, 512)	131584
dropout_4 (Dropout)	(None, 512)	0
dense_2 (Dense)	(None, 62)	31806
Total params: 2,082,238 Trainable params: 2,082,238		

Table 3.1: Table containing the summary of the implemented deep convolutional neural network model, produced by Keras.

3.4.4 Training Data Pipeline Implementation

Before the designed convolutional neural network can be trained, a data pipeline is required for formatting the EMNIST [121] training dataset into the necessary format for training. The EMNIST dataset contains three versions of the data, a By_Field, By_Class and By_Merge dataset. The By_Class dataset was chosen as it has the largest number of classes, totalling at 62 classes. Cohen et al [121] state the By_Class dataset “represents the most useful organization from a classification perspective as it contains the seg-

mented digits and characters arranged by class. There are 62 classes comprising [0-9], [a-z] and [A-Z]. The data is also split into a suggested training and testing set" [121]. To prepare the chosen dataset for training, a number of steps must be taken.

First the dataset is loaded and the character mapping data is extracted and saved to a file for later use. This mapping data relates the training data samples to the character text values that they represent. This mapping will be loaded during both training and prediction in the text classification component. The next step is to extract the training data and training labels. These are saved to two collections and will be return when the data pipeline is in use. The training data contains the image data used to training the network. The training labels identify what class each sample belongs to. The testing images and testing labels are then extracted and stored in the same manner. The training images and labels are used during the training stage and the test images and labels are used to test the trained network on unseen data. The subsequent stage of processing is aimed at readying the extracted training and testing samples for use. The collection of training and testing images are reshaped to be contained within tensors of the following shape: [*number of sample images, image height, image width, 1*]. In this case, the width and height of all training data is 28 by 28 pixels. The 1 at the end of the tensor is to represent the single colour channel, as the images are grayscale. If the images were in full RGB colour, the last value in the tensor would be 3, i.e. one for each colour channel. When the image data is finished being reshaped, the images must be flipped due to the images

being transposed after being read from the dataset. This is a quirk with the dataset and must be rectified with every training and test image.

The final steps taken in the data pipeline is normalising the image data. Similar to when an image is passed to the text classification component, each image is converted to 32 bit floating point numbers and normalised to be in the range of 0 to 255. This operation is applied to both the training images collection and test images collection. Once this is complete, the training images, training labels, testing images, testing labels, the character mappings and the number of classes in the dataset are returned from the data pipeline. All of these values will be available for use where ever the pipeline is utilised. This component is utilised in the test classification training component.

3.4.5 Training Text Classification CNN

This section covers the component responsible for training the deep convolutional neural network designed for handwritten text classification. With the training data pipeline built, the first task before training can begin is loading the training dataset. The data pipeline component returns the training and testing images and labels, along with the character mappings and the number of classes. After the training data is loaded, a number of hyper-parameters are defined. The mini-batch [77] size will be 256, the number of training epochs will be 10, the shape of the input data will be (28, 28, 1) to match the shape of the training and testing data. The max pooling filter size is set to (2, 2) and the convolutional kernel size is set to (3, 3).

With the hyperparameters set, the network model is built using the Keras API. The architecture of the deep convolutional neural network can be seen in Figure 3.15. All nodes use ReLU activation and the output layer utilises softmax activation. After each max pooling layer, dropout [73] is applied during training. After the first and second pooling layer, a dropout of 25% is used. This will randomly deactivate 25% of the nodes during the training process. After the third pooling layer, a dropout value of 30% is used and a dropout of 50% after the fully-connected layer. During training, the categorical cross entropy loss function and Adam optimiser [60] are both employed. Refer to Table 3.1 for the Keras model summary.

Before training commences, if there are existing weights saved to a file from previous training sessions they are loaded into the model so training can continue. The training images and labels are then passed to the model as training data, the test images and labels are passed to the model as validation data and the model is then trained for 10 epochs using a mini-batch value of 256. As the EMNIST dataset contains 697,932 training samples and 116,323 test samples, training will take a long period of time and is discussed in the following chapter. Once the model is finished training, the model's accuracy on the training data and its accuracy on the test data is printed out. Finally, both the model and the model weights are saved to individual files for later use.

3.4.6 Building the JSON Representation

The core service handles the construction of the JSON representation of the hand-drawn graph. As previously covered, the graph processing component detects graph nodes and edge and returns them to the core service. Each node is then passed to the node processing component which extracts the handwritten labels from the nodes. The characters in the handwritten labels are processed and returned to the core service. The core service then passes each character image, one at a time, to the deep convolutional neural network to predict which characters are in the images. The text classification component returns the predicted character as a user readable text string. Once all of the characters are classified for a node, the text is added to the node object, with the inferred spaces being added to the text where required. After the handwritten labels for all detected nodes have been classified, the core service is left with two collections. The collection of nodes, including the classified text of the handwritten labels and the graph edges, including the indices of both nodes that are connected via that edge.

To construct the JSON representation of the hand-drawn graph, a JSON object is created with two arrays: nodes and links. An object for each node is added to the JSON array called nodes, with the node text and index as properties of the object. Then, for each edge, an object is added to the links array with the properties source and target, where the values are the two connected node indices. An example of JSON that would be generated from a simple hand-drawn graph can be seen below.

```
{
    'nodes':[
        {
            'index':0,
            'text':'SAM'
        },
        {
            'index':1,
            'text':'PAUL'
        },
        {
            'index':2,
            'text':'DAVE'
        },
        {
            'index':3,
            'text':'JOHN'
        }
    ],
    'links':[
        {
            'source':2,
            'target':0
        },
        {
            'source':1,
            'target':2
        },
        {
            'source':2,
            'target':3
        }
    ]
}
```

Chapter 4

System Evaluation

This chapter presents an evaluation of the research hypothesis, which includes examining the performance of the software prototype's ability to parse a hand-drawn graph, extract handwritten labels from the detected nodes and build a JSON representation. The key focus areas for this performance evaluation is the precision of the hand-drawn graph parsing and the accuracy of the handwritten text classification. For each of these facets of the hypothesis, the evaluation data, software limitations and results are examined and discussed. Following this presentation and discussion, the research objectives are reviewed and then evaluated to determine the degree to which they have been satisfied.

Section 4.1 is a review of the performance of the software at the task of parsing hand-drawn graphs. This includes the presentation of a hand-drawn graph dataset compiled for evaluation and benchmarking. The results of the evaluation are then examined and discussed in detail.

Section 4.2 presents a review of the performance of the handwritten text classification using a deep convolutional neural network. The results of the classification are compared to the accuracy achieved by the original publishers of the benchmark dataset used during training.

4.1 Graph Parsing Performance

This section evaluates the graph parsing component's performance by testing the developed prototype's graph parsing capabilities, showcasing what can be achieved and examining the software's limitations. This section examines the sample data used to evaluate the performance of the prototype, the limitations of the prototype to take into consideration and presents the results of the evaluation in a tabular format.

4.1.1 Evaluation Data

Due to the absence of any published objective benchmark datasets of hand-drawn graph images, a sample set of 29 hand-drawn graphs was created for the purpose of evaluation. The following description includes the sample data used for the evaluation of the graph parsing component. Each image contains a hand-drawn graph that the software must process. Each image has a graph code assigned to it, which can be found in the image caption under each image, e.g. Graph G-01, G-02, G-03, etc. This graph code will link the results of the testing to the image of the graph that the test results belong to. The test results are displayed in Table 4.2 in section 4.1.3.

Each graph image has been assigned a category, which signifies the area in which it is being evaluated. Each category, including category name, key and description are depicted in Table 4.1. Graphs are segregated into categories to test specific attributes of a set of graphs. These categories include what are classified as 'Normal' or 'Regular' (NR) graphs. Graphs belonging to this category are smaller graphs which cover a number of different styles of drawing. The Normal or NR category of graph images is a collection of graphs to test the prototype's baseline performance on relatively straight forward hand-drawn undirected graphs. Some graphs have small nodes, large nodes and long or short edges. Some examples even have arrows to denote direction. While the software prototype only supports undirected graphs, graphs with directed edges must also be tested. The aim is for a directed edge to still be detected and saved as an undirected edge.

Highly connected (HC) graph images are graphs with a large number of nodes and edges. The aim is to evaluate the prototype's ability to identify large numbers of nodes and edges correctly. Self-referencing (SR) graph images include graphs which contain edges where both ends are pointing at the same node. Self-referencing graph nodes can be utilised to communicate recursive patterns in data. This category of graphs evaluated whether or not these types of edges are supported by the software prototype. Filled-in-Node (FN) graph images are hand-drawn graphs where some or all nodes are filled, coloured or shaded in. These are a valid style of graph and therefore must be evaluated. Graphs with Curved Edges (CE) and Orphaned Nodes (ON) are also evaluated. Hand-drawn graphs which include long curving edges

and disconnected or orphan nodes are tested to define what properties of a hand-drawn graph are or are not supported.

There are a number of categories in which hand-drawn graphs fall into which are not supported. It is of paramount importance to clearly define which types of hand-drawn graphs cannot be accurately parsed. These categories include graphs with direct node to edge contact (NE), graphs in which edges are intersecting (IE) and graphs where labels appear in place of nodes (NN). The aim is to objectively evaluate a number of these types of graphs in order to clearly differentiate between valid and invalid hand-drawn graphs.

The sample graph images were drawn on blank A4 drawing paper and photographed in as even lighting conditions as possible. With multiple graphs per A4 page, the images of the graphs were then cropped to obtain a single graph per image. These images were then saved for testing. The following is a sample of the collection of graph images used during testing, including their graph number and category key.

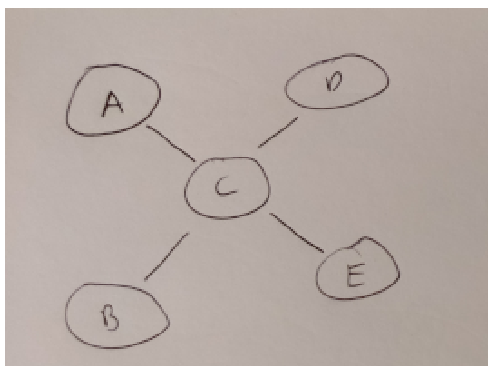


Figure 4.1: Graph G-01. Category NR.

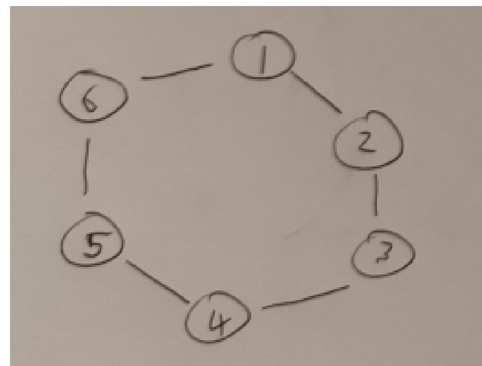


Figure 4.2: Graph G-02. Category NR.

Graph Image Categories

Key	Category Name	Description
NR	Normal	A normal example of a hand-drawn graph.
HC	Highly Connected	A graph with many nodes & edges.
SR	Self-Referencing	A graph with self referencing nodes.
FN	Filled-in-Nodes	A graph where nodes are filled in.
CE	Curved Edges	A graph where edges are curved.
ON	Orphan Nodes	Graphs with orphan or disconnected nodes.
NE	Node - Edge Contact	A graph where edges make direct contact with nodes.
IE	Intersecting Edges	Graphs where edges intersect each other.
NN	No Nodes	Graphs where labels are not contained in nodes.

Table 4.1: Each graph image belongs to a category aimed at testing a particular aspect of the drawn graph. The above table contains the categories each graph is separated into.

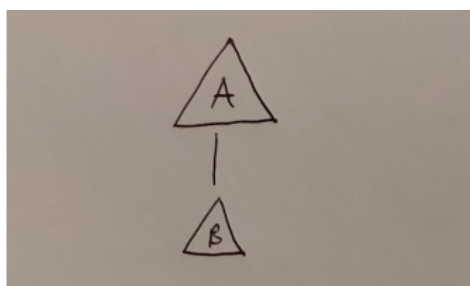


Figure 4.3: Graph G-03. Category NR.

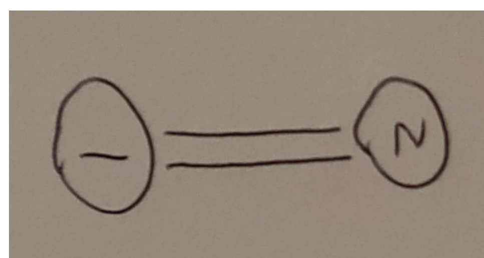


Figure 4.4: Graph G-04. Category NR.

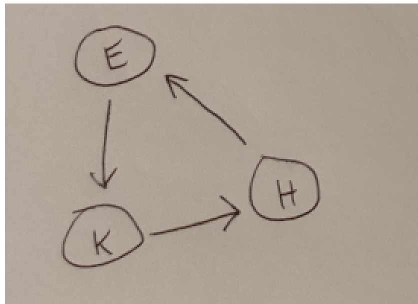


Figure 4.5: Graph G-05. Category NR.

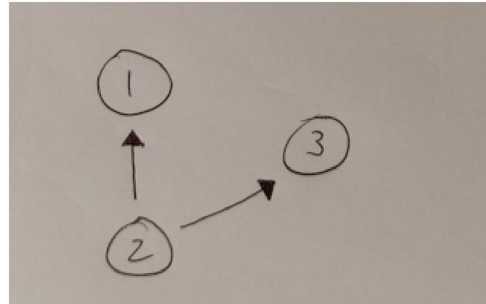


Figure 4.6: Graph G-06. Category NR.

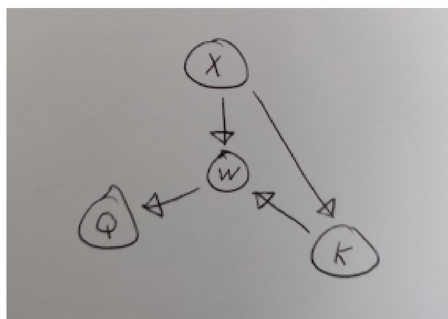


Figure 4.7: Graph G-07. Category NR.

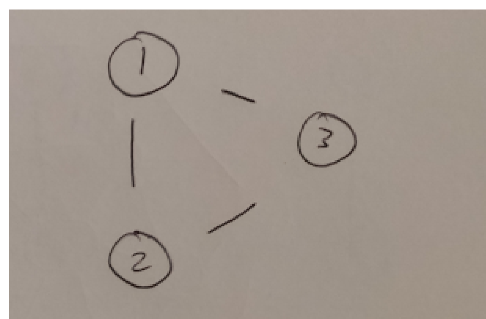


Figure 4.8: Graph G-08. Category NR.

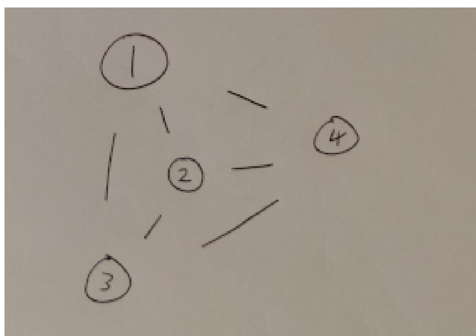


Figure 4.9: Graph G-09. Category NR.

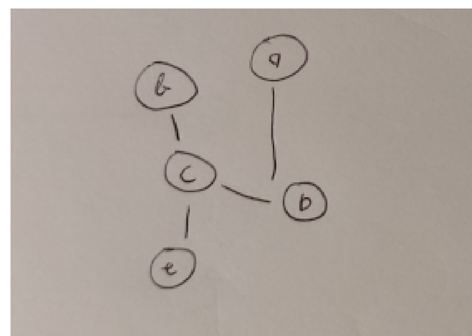


Figure 4.10: Graph G-10. Category NR.

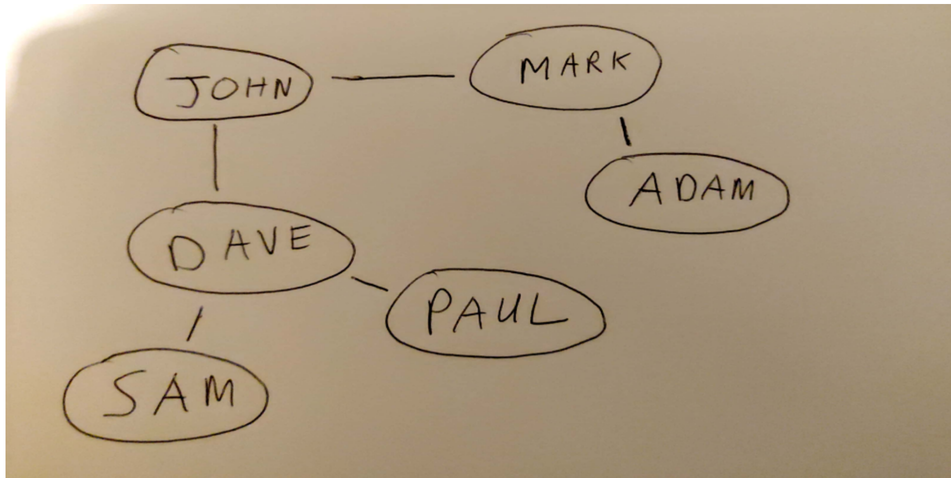


Figure 4.11: Graph G-11. Category NR.

4.1.2 Software Guidelines & Limitations

Before examining the results of the graph parsing system, the known limitations of this component of the prototype are reviewed. Based on the results displayed in Table 4.2, the graph parsing component performs consistently and accurately when working within a known set of guidelines. The development objective was to strike a balance between feasibility and functionality. There were a number of design decisions committed to during development to allow for improved robustness of the prototype. The following is a set of guidelines or rules that the prototype is constrained to operate within.

The first guideline to follow when drawing a parse-able graph, is related to graph node form. When drawing a graph, the shapes that denote the nodes must be fully enclosing shapes. This is necessary for node detection, as a non-enclosed node will be detected as an edge. An example of a fully enclosed node and a non-enclosed node can be seen in Figure 4.33. This decision was

```
{
    "nodes": [{
        "index": 0,
        "text": "SAM"
    }, {
        "index": 1,
        "text": "PAUL"
    }, {
        "index": 2,
        "text": "0AVE"
    }, {
        "index": 3,
        "text": "A0AM"
    }, {
        "index": 4,
        "text": "J0HN"
    }, {
        "index": 5,
        "text": "MARK"
    }],
    "links": [{
        "source": 2,
        "target": 0
    }, {
        "source": 1,
        "target": 2
    }, {
        "source": 2,
        "target": 4
    }, {
        "source": 3,
        "target": 5
    }, {
        "source": 5,
        "target": 4
    }]
}
```


Figure 4.12: Graph G-11 generated JSON.

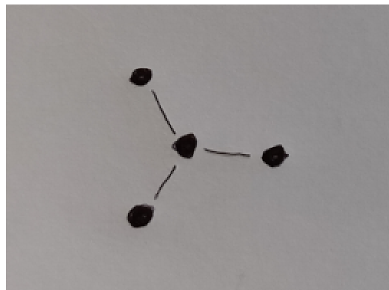


Figure 4.13: Graph G-12. Category FN.

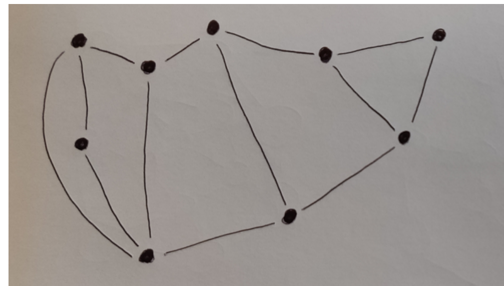


Figure 4.14: Graph G-13. Category FN.

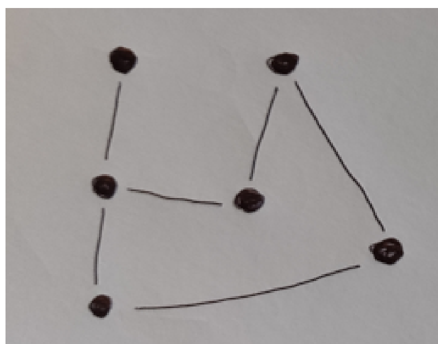


Figure 4.15: Graph G-14. Category FN.

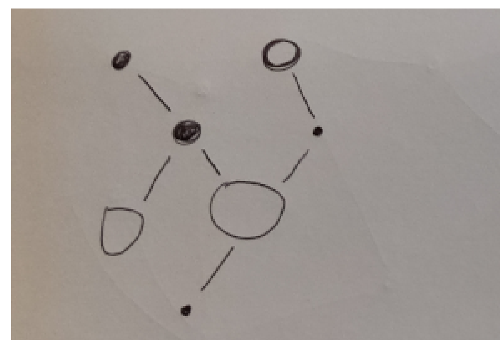


Figure 4.16: Graph G-15. Category FN.

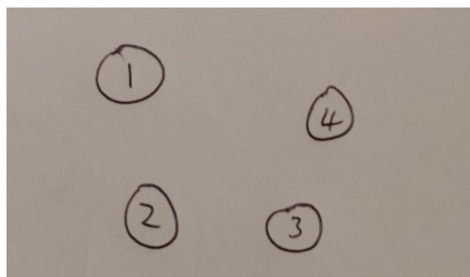


Figure 4.17: Graph G-16. Category ON.

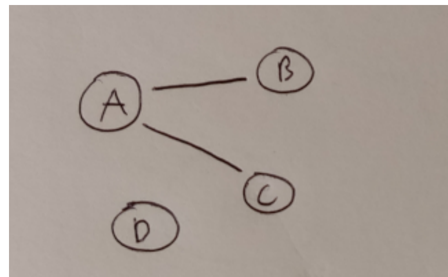


Figure 4.18: Graph G-17. Category ON.

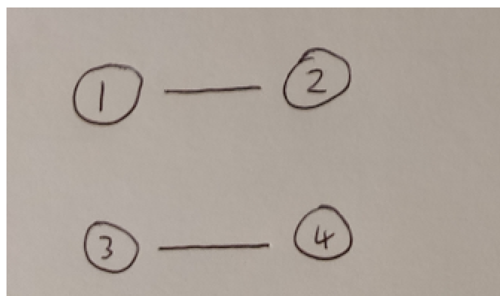


Figure 4.19: Graph G-18. Category ON.

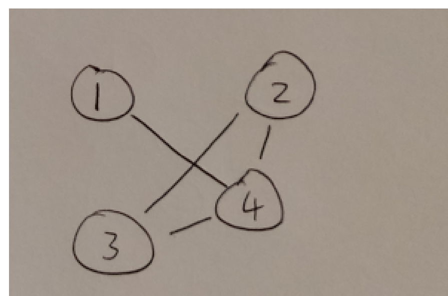


Figure 4.20: Graph G-19. Category IE.

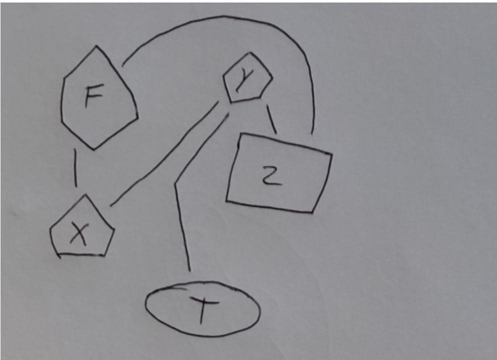


Figure 4.21: Graph G-20. Category CE.

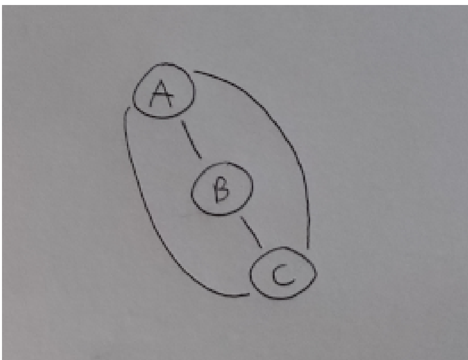


Figure 4.22: Graph G-21. Category CE.

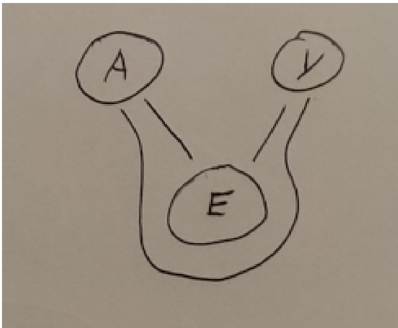


Figure 4.23: Graph G-22. Category CE.

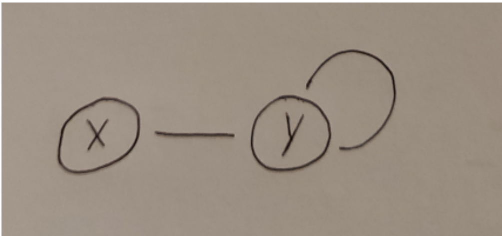


Figure 4.24: Graph G-23. Category SR.

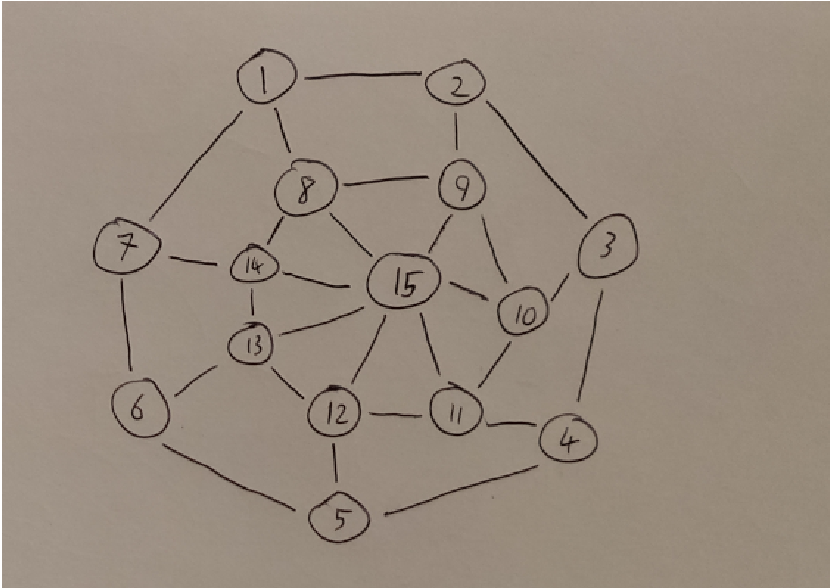


Figure 4.25: Graph G-24. Category HC.

```
{
    "nodes": [
        {"index": 0, "text": "5"},
        {"index": 1, "text": "4"},
        {"index": 2, "text": "1 2"},
        {"index": 3, "text": "1 1"},
        {"index": 4,"text": "6"},
        {"index": 5, "text": "B"},
        {"index": 6, "text": "10"},
        {"index": 7, "text": "1 5 "},
        {"index ": 8, "text ": "Q"},
        {"index ": 9, "text ": "7"},
        {"index ": 10, "text ": "3 "},
        {"index ": 11, "text ": "9 "},
        {"index ": 12, "text ": "8 "},
        {"index ": 13, "text ": "2 "},
        {"index ": 14, "text ": "1 "}
    ],
    "links ": [
        {"source ": 1, "target ": 0}, {"source ": 0, "target ": 2},
        {"source ": 0, "target ": 4}, {"source ": 1, "target ": 3},
        {"source ": 3, "target ": 2}, {"source ": 2, "target ": 5},
        { "source ": 5, "target ": 4}, {"source ": 3, "target ": 6},
        {"source ": 7, "target ": 2}, {"source ": 3, "target ": 7 },
        {"source ": 7, "target ": 5}, {"source ": 10, "target ": 1},
        {"source ": 5, "target ": 8}, {"source ": 6, "target ": 7},
        {"source ": 4, "target ": 9}, {"source ": 10, "target ": 6},
        {"source ": 7, "target ": 8}, {"source ": 8, "target": 9},
        {"source": 12, "target": 8}, {"source": 6, "target": 11},
        {"source": 11, "target": 7},  {"source": 7, "target": 12},
        {"source": 11, "target": 12}, {"source": 11, "target": 13},
        {"source": 9, "target": 14}, {"source": 12, "target": 14},
        {"source": 10, "target": 13}, {"source": 13,"target": 14}
    ]
}
```


Figure 4.26: Graph G-24 generated JSON.

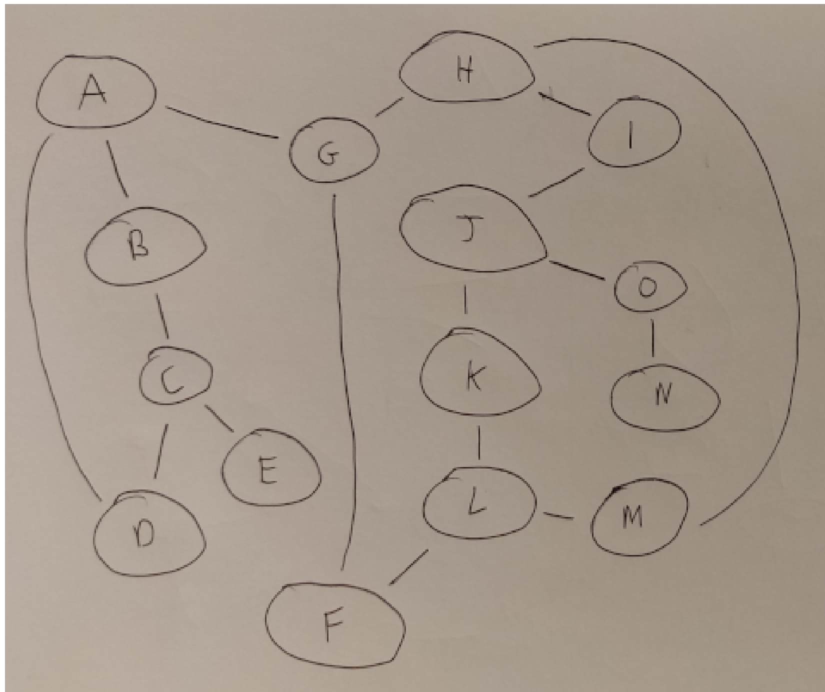


Figure 4.27: Graph G-25. Categories HC & CE.

made in an effort to improve the versatility of edge detection. Before this design decision was made, edge detection was brittle and lacked consistent robustness. This constraint allows for the detection of self referencing nodes, shown in Figure 4.24, and the detection of long curving edges as depicted in Figure 4.27.

The second guideline is related to graph edge placement. Edges between nodes must not physically touch the drawn nodes. If drawn edges make direct contact with nodes, the related node and edge will not be correctly detected. This is due to the contours which define the node and edge being detected as a single object, rather then two separate objects. Therefore, when the object that includes all directly connected nodes and edges is categorised as either

```
{
    "nodes": [
        {"index": 0, "text": "F"},
        {"index": 1, "text": "D"},
        {"index": 2, "text": "M"},
        {"index": 3, "text": "L"},
        {"index": 4, "text": "E"},
        {"index": 5, "text": "N"},
        {"index": 6, "text": "C"},
        {"index": 7, "text": "K"},
        {"index": 8, "text": "0"},
        {"index": 9, "text": "B"},
        {"index": 10, "text": "J"},
        {"index": 11, "text": "G"},
        {"index": 12, "text": "1"},
        {"index": 13, "text": "A"},
        {"index": 14, "text": "H"}
    ],
    "links": [
        {"source": 3, "target": 0}, {"source": 2, "target": 3},
        {"source": 3, "target": 7}, {"source": 6, "target": 1},
        {"source": 4, "target": 6}, {"source": 5, "target": 8},
        {"source": 6, "target": 9}, {"source": 7, "target": 10},
        {"source": 8, "target": 10}, {"source": 0, "target": 11},
        {"source": 12, "target": 10}, {"source": 9, "target": 13},
        {"source": 1, "target": 13}, {"source": 11, "target": 13},
        {"source": 14, "target": 11}, {"source": 12, "target": 14},
        {"source": 2, "target": 14}
    ]
}
```


Figure 4.28: Graph G-25 generated JSON.

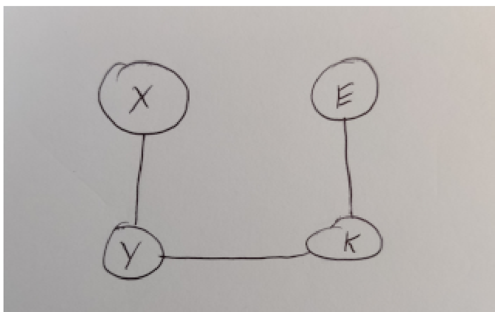


Figure 4.29: Graph G-26. Category NE.

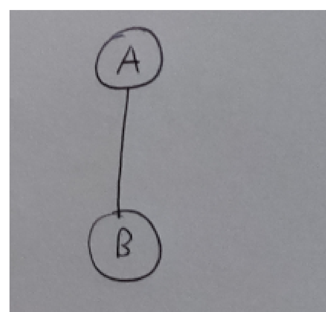


Figure 4.30: Graph G-27. Category NE.

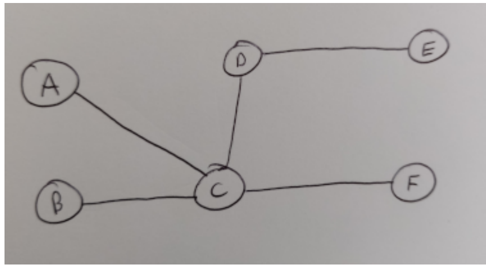


Figure 4.31: Graph G-28. Category NE.

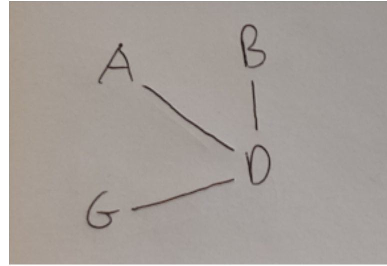
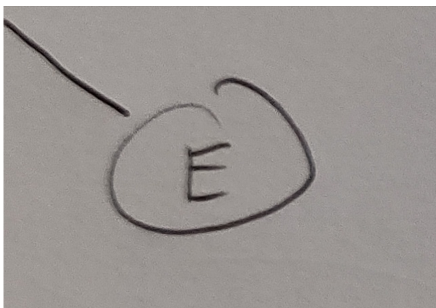


Figure 4.32: Graph G-29. Category NN.

a node or edge, it will fail the node test and it will be considered a graph edge. Examples of these graphs can be seen in Figures 4.29, 4.30 and 4.31.

Another guideline that must be followed for correct edge and relationship detection is the avoidance of graph edges that intersect one another. An example of this type of a graph can be seen in Figure 4.20. Hand-drawn graphs which do not follow this guideline, such as Figure 4.20, will fail to have its edges detected correctly and this is evident in the collected results. Graphs which do not adhere to the stated guidelines are not detected and parsed correctly. Their inclusion in the suite of benchmarking tests acts to illustrate the limitations of the prototype software application.



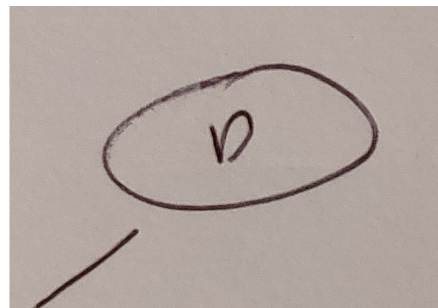


Figure 4.33: On the left, an example of a non-enclosing hand-drawn node. On the right, an example of a fully enclosing node. The image on the left will be detected as an edge, while the image on the right will be detected as a node.

4.1.3 Results for Node & Edge Detection

Results from the evaluation of the prototype are presented below in Table 4.2. The results are separated into columns representing the graph image code, the number of expected nodes, the number of detected nodes and the number of correctly detected nodes. The number of expected edges, the number of detected edges, the number of correctly detected edges and finally the total precision for each graph is also shown. Precision is "the degree to which repeated measurements under the same conditions give us the same results" [33] and is defined as: Precision = TP / (TP + FP), where TP is a True Positive prediction and FP is a False Positive prediction [33]. A True Positive prediction in this case is a graph node or edge that is detected, where the detection is correct. A False Positive prediction in this case is a graph node or edge that is detected, where the detection is incorrect.

An analysis of the results in Table 4.2 enables some clear conclusions to be drawn. Provided a hand-drawn graph follows the presented guidelines, the hand-drawn nodes, edges and therefore the relationships between each node, can be inferred from an image using this prototype software with near perfect precision. While there are a number of caveats, such as edge placement and shape, the software demonstrates the original goal set by the research hypothesis is achievable.

There are 7 of the 29 graphs which achieved a precision score lower than 100%, G-16, G-19, G-24, G-26, G-27, G-28 and G-29. The issues affecting these graphs can be categorised into 4 different problems within the graph

Graph Parsing Component's Evaluation Results

Graph	Nodes	Detected	Correct	Edges	Detected	Correct	Precision
G-01	5	5	5	4	4	4	100.00%
G-02	6	6	6	6	6	6	100.00%
G-03	2	2	2	1	1	1	100.00%
G-04	2	2	2	2	2	2	100.00%
G-05	3	3	3	3	3	3	100.00%
G-06	3	3	3	2	2	2	100.00%
G-07	4	4	4	4	4	4	100.00%
G-08	3	3	3	3	3	3	100.00%
G-09	4	4	4	6	6	4	80.00%
G-10	5	5	5	4	4	4	100.00%
G-11	6	6	6	5	5	5	100.00%
G-12	4	4	4	3	3	3	100.00%
G-13	9	9	9	13	13	13	100.00%
G-14	6	6	6	6	6	6	100.00%
G-15	7	7	7	6	6	6	100.00%
G-16	4	4	4	0	0	0	100.00%
G-17	4	4	4	2	2	2	100.00%
G-18	4	4	4	2	2	2	100.00%
G-19	4	3	3	4	3	2	83.33%
G-20	5	5	5	5	5	5	100.00%
G-21	3	3	3	4	4	4	100.00%
G-22	3	3	3	3	3	2	83.33%
G-23	2	2	2	2	2	2	100.00%
G-24	15	15	15	28	28	28	100.00%
G-25	15	15	15	17	17	17	100.00%
G-26	4	0	0	3	2	0	0.00%
G-27	2	0	0	1	2	0	0.00%
G-28	6	0	0	5	3	0	0.00%
G-29	4	0	0	3	7	3	42.86%

Table 4.2: Results of tests conducted on the graph parsing component's capabilities. The table columns represent the following values: the graph number associated with results (figures of graphs can be found above), the expected number of nodes, the number detected and the number correctly detected. Then, the expected number of edges in the graph, the number detected and the number correctly detected. Finally, the total precision of the graph parsing component for a given sample graph. Rows with poor performance are highlighted.

parsing process. The first issue that affects the parsing of graphs G-26, G-27 and G-28 (see Figures 4.29, 4.30 and 4.31), which have a precision score of 0%, is the graphs have edges which are directly connected to their nodes. As stated previously as a guideline, hand-drawn graphs cannot have edges that make direct contact with nodes. This is due to the contour detection in OpenCV [152] detecting the connected nodes and edges as a single object.

The second issue is related to the method of inferring which nodes an edge is pointing at. For each detected graph edge, the two farthest points from each other are calculated, as shown in Figure 4.34. For each of these two points, the closest node to each point is considered the node that edge is pointing at. Based on the results obtained, this method provides acceptable accuracy but it has clear limitations that can be observed in graphs G-09 and G-22 (see Figures 4.9 and 4.23), which obtained precision scores of 80% and 83.33% respectively. The issue with defining the ends of an edge as the two farthest points from each other, is that it is possible to draw an edge where the two farthest points are not the two actual ends of the edge. This is demonstrated in graph G-22 in Figure 4.23. It can be seen clearly in G-22 that the two farthest points from each other on the edge connecting the node labelled ‘A’ to the node labelled ‘Y’ are not the actual ends of the edge.

Another issue with the method chosen to connect nodes, is that once the two ends of an edge are calculated, the node closest to each end is selected. This is an issue due to the fact that the node closest to a given end of an edge might not be the node that the edge is actually pointing at. This can be seen

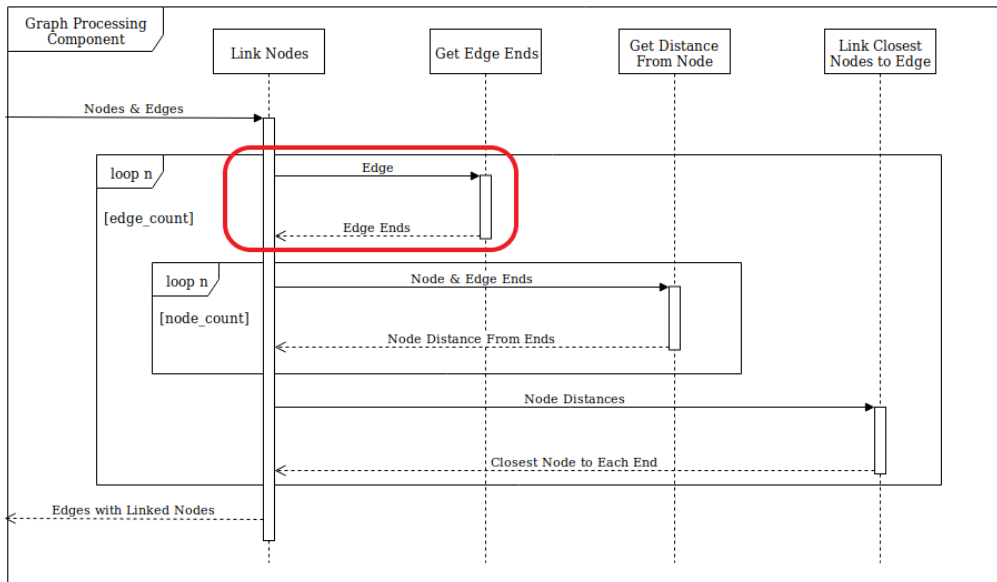


Figure 4.34: UML sequence diagram of the process of linking nodes via detected edges, with a focus on the process of detecting each end of a given edge.

in graph G-09 in Figure 4.9. The node in the centre of the graph, labelled '2' is the closest node to the edge that is connecting the nodes labelled '3' and '4'. This leads to the incorrect inference of which node the edge is pointing at, resulting in the wrong nodes being connected to each other. The root cause of this issue is due to the difference between an edge being *close* to a node and an edge *pointing* at a node. The current method presumes the closest node is the node an edge is pointing at but as demonstrated this may not be the case.

The third issue with parsing a graph is related to the placement of the edges connecting nodes. As stated previously, the edges connecting nodes in the hand-drawn graph cannot intersect each other. In the case of graph G-19 (see Figure 4.20) which obtained a precision score of 83.33%, the edge connecting

nodes ‘1’ and ‘4’ and the edge connecting nodes ‘2’ and ‘3’ are intersecting. The reason intersecting edges are not supported in the prototype is due to how contour detection works in OpenCV [152]. The intersecting edges are detected as a single object, not two individual objects. The last issue related to graph parsing is that node labels must be contained within a node object like a circle or oval. In the case of graph G-29 (see Figure 4.32), which obtained a precision score of 42.86%, the labels are not contained within a node object. While graph G-29 appears to be a valid graph, parsing such a graph is not supported by the prototype.

4.2 Text Classification Accuracy

This section is concerned with examining the accuracy of the handwritten text classification deep convolutional neural network. The deep CNN, the architecture of which can be seen in Figure 3.15, was trained on the EMNIST [121] handwritten character dataset.

4.2.1 Evaluation Data

The Extended MNIST (EMNIST) dataset was published by Cohen et al [121] as a more challenging classification benchmark than the widely popular MNIST [113] benchmark dataset. EMNIST is a collection of handwritten characters and digits that contains a much larger number of training and testing samples than the MNIST handwritten digit dataset. EMNIST is a variant of the full NIST Special Database 19 [191]. The EMNIST dataset contains a number of different configurations of the data. These configura-

tions can be seen in Table 4.3, adapted from [121]. By_Class was the chosen configuration for training because it "represents the most useful organization from a classification perspective as it contains the segmented digits and characters arranged by class. There are 62 classes comprising [0-9], [a-z] and [A-Z]. The data is also split into a suggested training and testing set" [121].

EMNIST Dataset Configurations

Name	Classes	No. Training	No. Testing	Validation	Total
By_Class	62	697,932	116,323	No	814,255
By_Merge	47	697,932	116,323	No	814,255
Balanced	47	112,800	18,800	Yes	131,600
Digits	10	240,000	40,000	Yes	280,000
Letters	26	124,800	20,800	Yes	145,600
MNIST	10	60,000	10,000	Yes	70,000

Table 4.3: Table containing the six configurations of the EMNIST dataset, adapted from [121]

The MNIST dataset segment in EMNIST is the same dataset as the popular MNIST handwritten digit dataset. An example of handwritten digits from MNIST is shown in Figure 4.35. The digits segment contains images of handwritten digits similar to MNIST but greater in number. The digits segment acts as a more challenging version of MNIST, containing 280,000 samples instead of the 70,000 samples in MNIST.

The letters segment of the EMNIST dataset contains 145,000 all uppercase characters, with no digits included. An example of a handwritten letter contained in the EMNIST dataset is shown in Figure 4.36. The letters segment "seeks to further reduce the errors occurring from case confusion by

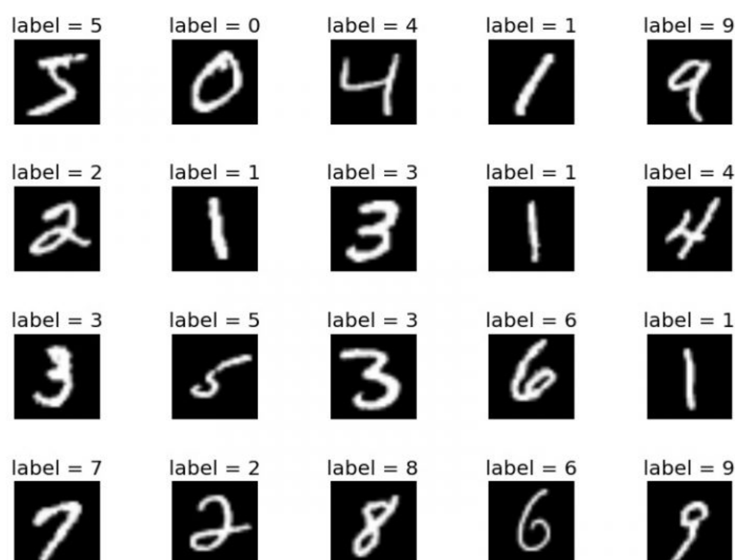


Figure 4.35: Example of the handwritten digits in the popular MNIST dataset, adapted from [195].

merging all the uppercase and lowercase classes to form a balanced 26-class classification task" [121]. The balanced segment contains handwritten digits and uppercase and lowercase characters. Some lowercase characters, such as 'c', 'u', 'z' etc. have been merged into the uppercase class due to character similarities. Cohen et al [121] made this decision to reduce classification errors where a lowercase character is classified as uppercase and vise versa.

The By_Class and By_Merge dataset segments are the two largest segments with a total of 814,255 samples each. Both segments contain handwritten digits and lowercase and uppercase characters. The only difference between the two segment is the number of classes. The By_Class segment contains the full 62 classes, which includes the digits 0 to 9 and all lowercase and uppercase characters. The By_Merge contains the same digits but similar

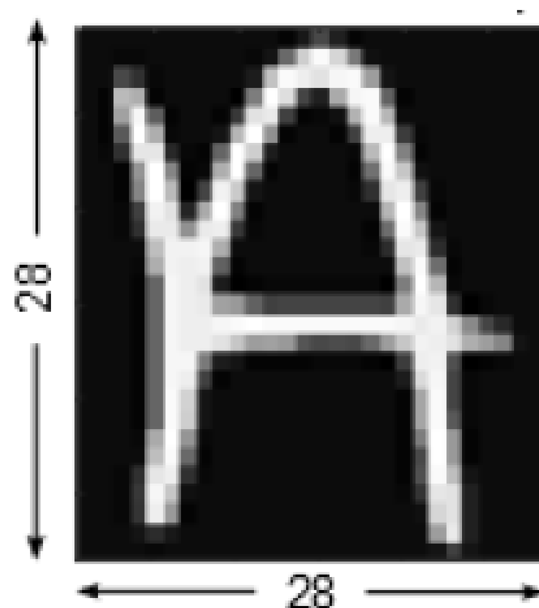


Figure 4.36: Example of the handwritten character in the EMNIST dataset, adapted from [121].

to the balanced segment merges some uppercase and lowercase characters to reduce the rate of classification error. This reduction in classification errors made in the event a lowercase character is classified as an uppercase character and vise versa is reflected in the collected results after training a deep CNN on the dataset. These results are displaced and discussed below in section 4.2.2.

4.2.2 Handwritten Text Classification Results

The following section includes the presentation of results obtained after the training of a deep convolutional neural network (deep CNN) on each segment of the EMNIST dataset. The deep CNN utilised during training is described in detail in section 3.4.3. The collected results are compared to the

results published by Cohen et al [121], the original publishers of the EMNIST dataset. The collected results are displayed in tables 4.4 and 4.5.

Table 4.4 compares the results published with the EMNIST dataset and the results obtained from training the deep CNN. The results published by Cohen et al [121] are the mean accuracy achieved on each segment of the EMNIST dataset after a given number of trials. Training was performed on the dataset for just one epoch, or iteration of the dataset, for a number of trials. Then, the mean accuracy was calculated for each segment. Following the method employed by Cohen et al, the deep CNN was trained under the same conditions. Table 4.4 displays the results published by Cohen et al, which were obtained using an OPIUM [196] classifier and the results of training on the deep CNN. All results are calculated as the mean accuracy achieved on each segment after 20 trials of training on the balanced segment and 10 trials on all other segments.

Training Results After One Epoch

Name	Classes	No. Training	No. Testing	OPIUM	Deep CNN
MNIST	10	60,000	10,000	96.22%	98.58%
Digits	10	240,000	40,000	95.90%	99.28%
Letters	26	124,800	20,800	85.15%	92.71%
Balanced	47	112,800	18,800	78.02%	84.95%
By_Merge	47	697,932	116,323	72.57%	89.69%
By_Class	62	697,932	116,323	69.71%	86.07%

Table 4.4: Training results for the EMNIST dataset segments. The name of the EMNIST dataset segment, the number of classes, training and testing samples, Cohen et al [121] published results and the results from the deep CNN are displayed respectively. All results are calculated as the mean accuracy achieved on each segment after 20 trials of training on the balanced segment and 10 trials on all other segments.

It can be seen in Table 4.4 that the deep CNN performs excellently on the EMNIST dataset. Examining the results obtained with the deep CNN, it appears the EMNIST Digits segments reached the highest accuracy of 99.28% after one epoch of training and the Balanced segment produced the lowest accuracy of 84.95%. The deep CNN out performed the OPIUM [196] classifier utilised by Cohen et al [121] in every segment of the dataset. It is believed the smaller sample size of the Balanced segment was the contributing factor in the segment scoring the lowest accuracy. This belief is also held when examining the higher accuracy score of the Digits segment over the MNIST dataset. The Digits segment has a total of 210,000 more samples than the MNIST segment.

Deep CNN Training Results

Name	Classes	No. Training	No. Testing	One Epoch	Five Epochs
MNIST	10	60,000	10,000	98.58%	99.35%
Digits	10	240,000	40,000	99.28%	99.60%
Letters	26	124,800	20,800	92.71%	94.59%
Balanced	47	112,800	18,800	84.95%	88.74%
By_Merge	47	697,932	116,323	89.69%	90.67%
By_Class	62	697,932	116,323	86.07%	87.13%

Table 4.5: Training results for the EMNIST dataset segments. The name of the EMNIST dataset segment, the number of classes, training and testing samples, the results from the deep CNN after one epoch of training and the results from the deep CNN after five epochs of training are displayed respectively. All results are calculated as the mean accuracy achieved on each segment after 10 trials of training.

Table 4.5 presents a comparison of the results obtained when training the deep CNN for one epoch and five epochs respectively. The results of training the deep CNN for one epoch are the same results seen in Table 4.4. The additional results are the mean accuracy calculated after training all dataset

segments for five trials for five epochs each. Due to hardware limitations, the deep CNN could not be trained more extensively as it would have taken too long. With extensive fine tuning of the deep CNN and longer training times, it may be possible to achieve a higher accuracy score. However, the results provided in Table 4.5 demonstrate the satisfaction of the research objective to achieve handwritten text classification which yields reasonable results.

The evaluation of the software prototype reveals a variety of information about its performance and accuracy. The precision of the graph parsing component, showcased in Table 4.2, clearly displays the software prototype's ability to parse a hand-drawn undirected labelled graph. While there are situations in which the software struggles to accurately parse hand-drawn graphs, these cases are rare when following the set of presented graph drawing guidelines. The classification accuracy of the trained deep convolutional neural network, displayed in Tables 4.4 and 4.5, similarly shows promising results. These results clearly demonstrate the prototype's accuracy in the task of handwritten text classification for hand-drawn graph node labels.

Chapter 5

Conclusion

Given the ubiquitous application of graph theory to solve problems spanning a multitude of discrete domains, research into developing usability and accessibility enhancing tools involving the utilisation of hand-drawn graphs is an attractive proposition. The aim of this research was to investigate the feasibility of using deep learning and computer vision to accurately parse a hand-drawn undirected labelled graph for the generation of a JSON representation that maintains its isomorphic properties.

For the research hypothesis to be objectively assessed, a number of core objectives were identified, scoped and satisfied by the design, construction, testing and evaluation of a prototype software artefact. In the first instance, the parsing of hand-drawn nodes contained within an image of a hand-drawn undirected labelled graph were required to be performed to a reasonable degree of precision. Secondly, the undirected edges present on the hand-drawn graph between graph nodes has to be accurately interpreted. Handwritten

text labels contained within nodes present on the hand-drawn graph should be extracted and classified to the highest level of accuracy that is feasible. Finally, the parsed graph information should be transformed into a JSON representation of the hand-drawn graph that maintains its isomorphic properties.

Research for this thesis began with an in-depth literature review of the state-of-the-art concepts, technologies and techniques in the areas of artificial neural networks, deep learning and computer vision. The rationale for this decision was to gain a greater understanding of the technologies required to successfully design and develop a software prototype capable of achieving the presented set of research objectives. Research commenced with a full review of artificial neural networks. Following this, the process of training artificial neural networks was studied from the ground up in order to obtain a full appreciation of the array of capabilities, challenges and nuances surrounding this complex subject. The subsequent stage of the literature review dealt with an examination of the various architectures employed when utilising deep learning to solve problems. Computer vision algorithms were then investigated in detail to ascertain the degree to which they can usefully be applied to test the research hypothesis.

Following an extensive literature review, the system design phase of this research project commenced with the analysis and definition of the set of user requirements for the software prototype. Following this, a collection of project constraints were elucidated in order to obtain a clear understanding

of what was required and realistically achievable. The design of the software prototype was heavily influenced by the ideas and insights presented in the literature review and consists of several discrete components, each of which is responsible for a defined parcel of work. These components include a service that parses images of hand-drawn undirected labelled graphs, detecting and capturing information about the nodes and edges present in the image. A component responsible for extracting the handwritten labels attached to graph nodes was constructed, along with a number of auxiliary components related to the classification of those handwritten labels. Finally, a handwritten text classification component, consisting of a deep convolutional neural network and its accompanying training system, was then presented.

The implementation of the designed software prototype was reviewed and benchmarked in order to test the performance and accuracy of the prototype and the underlying research hypothesis. A dataset containing hand-drawn graph images was developed, compiled and presented. The selection of sample hand-drawn graphs were segregated into a number of categories designed to test various aspects of the prototype's graph parsing and detection precision. Each of the graph samples were tested and results were presented and discussed alongside a detailed analysis. The handwritten character dataset utilised for the training of the developed deep convolutional neural network was then examined. This deep convolutional neural network was designed and developed to perform the classification of the handwritten text contained within the labels of the detected graph nodes. The accuracy of this deep neural network was benchmarked under various conditions and the results were

compiled and compared to the original publishers of the benchmark handwritten text dataset.

5.1 Key Findings

The evaluation of the developed software prototype has revealed a number of insights. Guidelines were developed to ensure a user of the software is aware of the type and style of hand-drawn undirected labelled graph that is parsable. When adherence to these guidelines is upheld, the software prototype can parse the nodes and edges of a given hand-drawn graph with near perfect precision. The classification accuracy of the handwritten labels contained within the nodes of a hand-drawn graph also show impressive results, with an accuracy rating greater than 80% on the most challenging segments of the benchmark handwritten text dataset.

The key objectives scoped to test the overall research hypothesis were the following:

1. The accurate parsing of a hand-drawn undirected graph's nodes.
2. The accurate interpretation of graph node relationships.
3. The precise extraction and classification of handwritten node labels.
4. The generation of an isomorphic JSON representation of the processed graph.

Based on the results obtained following the extensive testing and evaluation of the software prototype, the above research objectives have been broadly satisfied. The software prototype exhibits the ability to parse the nodes of a hand-drawn undirected labelled graph with near perfect precision. This ability includes the interpretation of the graph nodes relationships through the precise detection of hand-drawn edges. The developed deep convolutional neural network possesses the capability of handwritten text classification with up to 87% accuracy. This is believed to be an acceptable level of accuracy when taking feasibility, published benchmarks and project constraints into account. Consequently, the resulting JSON representation generated by the software prototype is shown to be highly accurate at maintaining the hand-drawn graph's isomorphic properties.

5.2 Limitations & Future Research

While the prototype software yields consistent results when following the defined rule set for graph creation, there is clearly room for improvement. This section discusses future research that may contribute towards superior performance and robustness. This future research has been divided into two main focus areas which are believed to be of primary importance. These two areas relate to improvements to the prototype's graph parsing capabilities and improvements to handwritten text classification.

5.2.1 Hand-drawn Graph Parsing

There are number of improvements that could be made to the prototypes graph parsing capabilities. At present, the required guidelines that must be followed for a hand-drawn graph to be successfully parsed are necessarily strict. The areas that require more development are the following:

1. Hand-drawn node detection.
2. Hand-drawn edge detection.
3. Node relationship inference based on edge placement.

Currently, hand-drawn nodes are only detected if they are fully enclosed shapes. This leads to incorrect node detection in the event of a node not being drawn correctly. While this design leads to more robust edge detection, there is potential room for improvement by leveraging more advanced machine vision techniques. Edge detection is also limited by the requirement of not having edges making direct contact with drawn nodes. Utilising a segmentation technique such as Watershed [159, 160, 161], a Fully Convolutional Network (FCN) for image segmentation [172] or a clustering algorithm such as K-Means [168] may provide superior results. This may overcome the difficulty in detecting nodes and edges using contour detection alone. Limitations with the current method of inferring which graph nodes an edge is pointing at relates to the difference between an edge being *close* to a node and an edge *pointing* at a node. In the event the software is upgraded in the future to detect edges between nodes, where the edges are directly making

contact with nodes, this issue will be resolved for those cases. With the current functionality of edges not making direct contact with nodes, a different approach to defining the ends of a detected edge and which nodes the edge is pointing at may be required.

A form of border-following, such as the one utilised in OpenCV's contour detection algorithm [153] or a corner detection algorithm such as Harris Corner Detection [147] may provide a better method of defining the ends of a detected edge. Corner detection could also be utilised to detect the direction an edge is pointing in, as directed graph edges are currently not supported. Improvements to detecting which node an edge is actually pointing at rather than the node closest to it could also be made. Applying some form of trajectory calculation to infer which node the edge is actually pointing at may yield improved results. These enhancements, in time, may allow inquiry into the possibility of processing directed graphs (digraphs) and even weighted directed graphs.

5.2.2 Handwritten Text Classification

Handwritten text classification and handwritten label extraction could be potentially improved by investigating different deep neural network architectures and topologies. Deep neural networks such as YOLO [120] and SSD [14] provide the capability of classifying multiple objects in an image. This feature could replace the label extraction steps in the software, due to it no longer being necessary to identify individual characters in an image of a graph node and pass them one by one to the neural network for classification. The

entire image of the hand-drawn graph could be passed to a YOLO or SSD style deep neural network and every detected character could be classified, with the locations of the classified characters being available for exploitation by the system. An upgrade such as this, would streamline the process of character detection and classification while reducing errors related to the improper detection and capture of a character that can currently occur. Other additions to the software could include an added post-processing step aimed at mitigating the errors in the handwritten text classification. A form of spelling correction after handwritten labels are classified could significantly improve the exhibited capabilities of the software prototype.

5.3 Closing Remarks

Future applications of the technologies examined in this thesis appear promising. There is still ample room for more extensive research into the utilisation of software which aims of provide a visual understanding and interpretation of potentially complex hand-drawn data structures. This is a category of software which can provide meaningful value to users in the form of usability and accessibility enhancing tools. Such tools may pave the way towards a future in which ease of use and accessibility are a priority, enabling the effortless communication, processing, storage and exploitation of complex hand-drawn data and ideas.

Publications

- Byrne, R., Healy, J., Duignan, S., McCaffery, K., ***Using a Deep Convolutional Network to Classify Handwritten Characters***, European GPU Technology Conference (GTC), 2018.

Bibliography

[1] Wikipedia contributors, “Graph theory — Wikipedia, the free encyclopedia.” https://en.wikipedia.org/w/index.php?title=Graph_theory&oldid=898058712, 2019. [Online; accessed 21-May-2019].

[2] K. L. Calvert, M. B. Doar, and E. W. Zegura, “Modeling internet topology,” *IEEE Communications magazine*, vol. 35, no. 6, pp. 160–163, 1997.

[3] D. J. Cook and L. B. Holder, “Graph-based data mining,” *IEEE Intelligent Systems and Their Applications*, vol. 15, no. 2, pp. 32–41, 2000.

[4] Z. Wu and R. Leahy, “An optimal graph theoretic approach to data clustering: Theory and its application to image segmentation,” *IEEE Transactions on Pattern Analysis & Machine Intelligence*, no. 11, pp. 1101–1113, 1993.

[5] R. Tarjan, “Depth-first search and linear graph algorithms,” *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[6] E. F. Moore, “The shortest path through a maze,” in *Proc. Int. Symp. Switching Theory, 1959*, pp. 285–292, 1959.

[7] H. Berliner, "The b* tree search algorithm: A best-first proof procedure," in *Readings in Artificial Intelligence*, pp. 79–87, Elsevier, 1981.

[8] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[9] S. Shirinivas, S. Vetrivel, and N. Elango, "Applications of graph theory in computer science an overview," *International journal of engineering science and technology*, vol. 2, no. 9, pp. 4610–4621, 2010.

[10] M. Haverbeke, *Eloquent javascript: A modern introduction to programming.* No Starch Press, 2014.

[11] A. Nayak, A. Poriya, and D. Poojary, "Type of nosql databases and its comparison with relational databases," *International Journal of Applied Information Systems*, vol. 5, no. 4, pp. 16–19, 2013.

[12] C. Chasseur, Y. Li, and J. M. Patel, "Enabling json document stores in relational systems.," in *WebDB*, vol. 13, pp. 14–15, 2013.

[13] C. Rodrigues, J. Afonso, and P. Tomé, "Mobile application webservice performance analysis: Restful services with json and xml," in *International Conference on ENTERprise Information Systems*, pp. 162–169, Springer, 2011.

[14] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. E. Reed, C. Fu, and A. C. Berg, "SSD: single shot multibox detector," *CoRR*, vol. abs/1512.02325, 2015.

[15] R. Hadsell, P. Sermanet, J. Ben, A. Erkan, M. Scoffier, K. Kavukcuoglu, U. Muller, and Y. LeCun, "Learning long-range vision for autonomous off-road driving," *Journal of Field Robotics*, vol. 26, no. 2, pp. 120–144, 2009.

[16] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[17] F. Ning, D. Delhomme, Y. LeCun, F. Piano, L. Bottou, and P. E. Barbano, "Toward automatic phenotyping of developing embryos from videos," *IEEE Transactions on Image Processing*, vol. 14, pp. 1360–1371, 2005.

[18] A.-r. Mohamed, G. E. Dahl, and G. Hinton, "Acoustic modeling using deep belief networks," *Audio, Speech, and Language Processing, IEEE Transactions on*, vol. 20, pp. 14 – 22, 02 2012.

[19] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *2013 IEEE international conference on acoustics, speech and signal processing*, pp. 6645–6649, IEEE, 2013.

[20] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[21] S. Espana-Boquera, M. J. Castro-Bleda, J. Gorbe-Moya, and F. Zamora-Martinez, "Improving offline handwritten text recognition

with hybrid hmm/ann models," *IEEE transactions on pattern analysis and machine intelligence*, vol. 33, no. 4, pp. 767–779, 2010.

[22] G. Bebis, D. Egbert, and M. Shah, "Review of computer vision education," *IEEE Transactions on Education*, vol. 46, no. 1, pp. 2–21, 2003.

[23] M.-H. Yang, D. J. Kriegman, and N. Ahuja, "Detecting faces in images: A survey," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 24, no. 1, pp. 34–58, 2002.

[24] E. Hjelmås and B. K. Low, "Face detection: A survey," *Computer vision and image understanding*, vol. 83, no. 3, pp. 236–274, 2001.

[25] A. Jimenez, R. Ceres, and J. Pons, "A survey of computer vision methods for locating fruit on trees," *Transactions of the ASAE*, vol. 43, no. 6, p. 1911, 2000.

[26] E. Murphy-Chutorian and M. M. Trivedi, "Head pose estimation in computer vision: A survey," *IEEE transactions on pattern analysis and machine intelligence*, vol. 31, no. 4, pp. 607–626, 2009.

[27] S. S. Rautaray and A. Agrawal, "Vision based hand gesture recognition for human computer interaction: a survey," *Artificial Intelligence Review*, vol. 43, no. 1, pp. 1–54, 2015.

[28] C. B. Ng, Y. H. Tay, and B.-M. Goi, "Recognizing human gender in computer vision: a survey," in *Pacific Rim International Conference on Artificial Intelligence*, pp. 335–346, Springer, 2012.

[29] T. Gandhi and M. M. Trivedi, "Pedestrian collision avoidance systems: A survey of computer vision based recent studies," in *2006 IEEE Intelligent Transportation Systems Conference*, pp. 976–981, IEEE, 2006.

[30] G. C. Cawley and N. L. C. Talbot, "Gene selection in cancer classification using sparse logistic regression with Bayesian regularization," *Bioinformatics*, vol. 22, pp. 2348–2355, 07 2006.

[31] W. S. McCulloch and W. H. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics*, 1943.

[32] S. I. Ele and W. Adesola, "Artificial neuron network implementation of boolean logic gates by perceptron and threshold element as neuron output function," *International Journal of Science and Research*, vol. 4, no. 9, pp. 637–641, 2013.

[33] J. Patterson and A. Gibson, *Deep Learning.* O'Reilly, 2017.

[34] D. Hebb, *The Organization of Behavior*. Psychology Press (2002), 1949.

[35] S. Lowel and W. Singer, "Selection of intrinsic horizontal connections in the visual cortex by correlated neuronal activity," *Science*, vol. 255, no. 5041, pp. 209–212, 1992.

[36] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, vol. 65, no. 6, p. 386–408, 1958.

[37] M. Minsky and S. Papert, *Perceptrons; an introduction to computational geometry.* MIT Press, Cambridge, MA, 1969.

[38] S. Pattanayak, *Pro Deep Learning with TensorFlow.* Apress, 2018.

[39] F. Agostinelli, M. D. Hoffman, P. J. Sadowski, and P. Baldi, "Learning activation functions to improve deep neural networks," *CoRR*, vol. abs/1412.6830, 2014.

[40] M. Rybarsch and S. Bornholdt, "Binary threshold networks as a natural null model for biological networks," *Physical Review E*, vol. 86, no. 2, p. 026114, 2012.

[41] C. Dugas, Y. Bengio, F. Bélisle, C. Nadeau, and R. Garcia, "Incorporating second-order functional knowledge for better option pricing," in *Advances in neural information processing systems*, pp. 472–478, 2001.

[42] F. Gers, *Long short-term memory in recurrent neural networks.* PhD thesis, Verlag nicht ermittelbar, 2001.

[43] C. Gulcehre, M. Moczulski, M. Denil, and Y. Bengio, "Noisy activation functions," in *International conference on machine learning*, pp. 3059–3068, 2016.

[44] B. Chen, W. Deng, and J. Du, "Noisy softmax: Improving the generalization ability of dcnn via postponing the early softmax saturation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5372–5381, 2017.

[45] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807–814, 2010.

[46] Y. Li and Y. Yuan, "Convergence analysis of two-layer neural networks with relu activation," in *Advances in Neural Information Processing Systems*, pp. 597–607, 2017.

[47] C. Zhang and P. C. Woodland, "Parameterised sigmoid and relu hidden activation functions for dnn acoustic modelling," in *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

[48] K. He and J. Sun, "Convolutional neural networks at constrained time cost," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[49] D. Nguyen and B. Widrow, "Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights," in *1990 IJCNN International Joint Conference on Neural Networks*, pp. 21–26 vol.3, June 1990.

[50] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," in *Parallel distributed processing: explorations in the microstructure of cognition, Vol. 1* (D. E. Rumelhart, J. L. McClelland, and C. PDP Research Group, eds.), ch. Learning Internal Representations by Error Propagation, pp. 318–362, Cambridge, MA, USA: MIT Press, 1986.

[51] T. P. Vogl, J. K. Mangis, A. K. Rigler, W. T. Zink, and D. L. Alkon, "Accelerating the convergence of the back-propagation method," *Biological Cybernetics*, vol. 59, pp. 257–263, Sep 1988.

[52] J. Schmidhuber, "Deep learning in neural networks: An overview," *CoRR*, vol. abs/1404.7828, 2014.

[53] S. Ruder, "An overview of gradient descent optimization algorithms," *CoRR*, vol. abs/1609.04747, 2016.

[54] "Gradient descent diagram." `https://saugatbhattarai.com.np/what-is-gradient-descent-in-machine-learning/`. Online; accessed 25 March 2019.

[55] A. A. Goldstein, "On steepest descent," *Journal of the Society for Industrial and Applied Mathematics Series A Control*, vol. 3, no. 1, pp. 147–151, 1965.

[56] J. Fliege and B. Svaiter, "Steepest descent methods for multicriteria optimization," *Math Methods Oper Res*, vol. 51, pp. 479–494, 08 2000.

[57] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010* (Y. Lechevallier and G. Saporta, eds.), (Heidelberg), pp. 177–186, Physica-Verlag HD, 2010.

[58] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *J. Mach. Learn. Res.*, vol. 12, pp. 2121–2159, July 2011.

[59] M. D. Zeiler, "ADADELTA: an adaptive learning rate method," *CoRR*, vol. abs/1212.5701, 2012.

[60] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014.

[61] T. Tieleman and G. Hinton, "Coursera: Neural networks for machine learning," *University of Toronto*, 2012.

[62] M. Alizadeh, J. Fernández-Marqués, N. D. Lane, and Y. Gal, "A systematic study of binary neural networks' optimisation," in *International Conference on Learning Representations*, 2019.

[63] M. R. Avendi, "Playing with loss functions in deep learning," *Medium*, 2018.

[64] F. Rousselle, C. Knaus, and M. Zwicker, "Adaptive sampling and reconstruction using greedy error minimization," *ACM Trans. Graph.*, vol. 30, pp. 159:1–159:12, Dec. 2011.

[65] M. Gabbouj and E. J. Coyle, "Minimum mean absolute error stack filtering with structural constraint and goals," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 38, pp. 955–968, June 1990.

[66] Y. Wu and Y. Liu, "Robust truncated hinge loss support vector machines," *Journal of the American Statistical Association*, vol. 102, no. 479, pp. 974–983, 2007.

[67] T. Gao and D. Koller, "Multiclass boosting with hinge loss based on output coding," *Proceedings of the 28th International Conference on Machine Learning, ICML 2011*, pp. 569–576, 01 2011.

[68] S. Kullback and R. A. Leibler, "On information and sufficiency," *The annals of mathematical statistics*, vol. 22, no. 1, pp. 79–86, 1951.

[69] S. Villena, M. Vega, S. Derin Babacan, R. Molina, and A. Katsaggelos, "Using the kullback-leibler divergence to combine image priors in super-resolution image reconstruction," *Proceedings - International Conference on Image Processing, ICIP*, pp. 893–896, 09 2010.

[70] Z. Reitermanova, "Data splitting," in *WDS*, vol. 10, pp. 31–36, 2010.

[71] M. Solazzi and A. Uncini, "Regularising neural networks using flexible multivariate activation function," *Neural Networks*, vol. 17, no. 2, pp. 247–260, 2004.

[72] R. Caruana, S. Lawrence, and C. Lee Giles, "Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping.," *Advances in Neural Information Processing Systems*, vol. 13, pp. 402–408, 01 2000.

[73] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 06 2014.

[74] R. Miikkulainen, J. Z. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy, and B. Hodjat, "Evolving deep neural networks," *CoRR*, vol. abs/1703.00548, 2017.

[75] L. N. Smith, "No more pesky learning rate guessing games," *CoRR*, vol. abs/1506.01186, 2015.

[76] I. Sutskever, J. Martens, G. E. Dahl, and G. E. Hinton, "On the importance of initialization and momentum in deep learning.," *ICML (3)*, vol. 28, no. 1139-1147, p. 5, 2013.

[77] D. Masters and C. Luschi, "Revisiting small batch training for deep neural networks," *CoRR*, vol. abs/1804.07612, 2018.

[78] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi, "A survey of deep neural network architectures and their applications," *Neurocomputing*, vol. 234, pp. 11–26, 2017.

[79] P. Smolensky, "Information processing in dynamical systems: Foundations of harmony theory," tech. rep., Colorado Univ at Boulder Dept of Computer Science, 1986.

[80] R. Rastgoo, K. Kiani, and S. Escalera, "Multi-modal deep hand sign language recognition in still images using restricted boltzmann machine," *Entropy*, vol. 20, no. 11, p. 809, 2018.

[81] B. D. Smith, "Musical deep learning: Stylistic melodic generation with complexity based similarity," *Proceedings of the Musical Metacreativity*

Workshop at the Eighth International Conference on Computational Creativity, 2017.

[82] G. E. Hinton, "A practical guide to training restricted boltzmann machines," in *Neural networks: Tricks of the trade*, pp. 599–619, Springer, 2012.

[83] "Restricted boltzmann machine diagram." `https://skymind.ai/wiki/restricted-boltzmann-machine`. Online; accessed 10 March 2019.

[84] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

[85] J. C. Cuevas-Tello, M. Valenzuela-Rendón, and J. A. Nolazco-Flores, "A tutorial on deep neural networks for intelligent systems," *CoRR*, vol. abs/1603.07249, 2016.

[86] A.-r. Mohamed, G. Dahl, and G. Hinton, "Deep belief networks for phone recognition," in *Nips workshop on deep learning for speech recognition and related applications*, vol. 1, p. 39, Vancouver, Canada, 2009.

[87] P. Baldi, "Autoencoders, unsupervised learning, and deep architectures," in *Proceedings of ICML workshop on unsupervised and transfer learning*, pp. 37–49, 2012.

[88] R. Salakhutdinov and G. Hinton, "Semantic hashing," *International Journal of Approximate Reasoning*, vol. 50, no. 7, pp. 969–978, 2009.

[89] A. Krizhevsky and G. E. Hinton, "Using very deep autoencoders for content-based image retrieval.," in *ESANN*, 2011.

[90] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P.-A. Manzagol, "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion," *Journal of machine learning research*, vol. 11, no. Dec, pp. 3371–3408, 2010.

[91] "Autoencoder diagram." `https://skymind.ai/wiki/deep-autoencoder`. Online; accessed 11 March 2019.

[92] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in Neural Information Processing Systems 27* (Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, eds.), pp. 2672–2680, Curran Associates, Inc., 2014.

[93] J. Engel, K. K. Agrawal, S. Chen, I. Gulrajani, C. Donahue, and A. Roberts, "Gansynth: Adversarial neural audio synthesis," *arXiv preprint arXiv:1902.08710*, 2019.

[94] Y. Wang, C. Wu, L. Herranz, J. van de Weijer, A. Gonzalez-Garcia, and B. Raducanu, "Transferring gans: generating images from limited data," in *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 218–234, 2018.

[95] Y. Pan, Z. Qiu, T. Yao, H. Li, and T. Mei, "To create what you tell: Generating videos from captions," *CoRR*, vol. abs/1804.08264, 2018.

[96] S. E. Reed, Z. Akata, X. Yan, L. Logeswaran, B. Schiele, and H. Lee, "Generative adversarial text to image synthesis," *CoRR*, vol. abs/1605.05396, 2016.

[97] "Generative adversarial network diagram." `https://skymind.ai/wiki/generative-adversarial-network-gan`. Online; accessed 11 March 2019.

[98] Y. Bengio, P. Simard, P. Frasconi, *et al.*, "Learning long-term dependencies with gradient descent is difficult," *IEEE transactions on neural networks*, vol. 5, no. 2, pp. 157–166, 1994.

[99] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.

[100] T. Mikolov, M. Karafiát, L. Burget, J. Černockỳ, and S. Khudanpur, "Recurrent neural network based language model," in *Eleventh annual conference of the international speech communication association*, 2010.

[101] J. T. Connor, R. D. Martin, and L. E. Atlas, "Recurrent neural networks and robust time series prediction," *IEEE transactions on neural networks*, vol. 5, no. 2, pp. 240–254, 1994.

[102] Y. Qin, D. Song, H. Cheng, W. Cheng, G. Jiang, and G. W. Cottrell, "A dual-stage attention-based recurrent neural network for time series prediction," *CoRR*, vol. abs/1704.02971, 2017.

[103] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, pp. 3111–3119, 2013.

[104] "Recurrent neural network diagram." `https://medium.com/explore-artificial-intelligence/an-introduction-to-recurrent-neural-networks-72c97bf0912`. Online; accessed 12 March 2019.

[105] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[106] "Long short-term memory diagram." `https://skymind.ai/wiki/lstm`. Online; accessed 12 March 2019.

[107] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[108] C. Poultney, S. Chopra, Y. L. Cun, *et al.*, "Efficient learning of sparse representations with an energy-based model," in *Advances in neural information processing systems*, pp. 1137–1144, 2007.

[109] "Convolutional nerual network schema diagram." `https://skymind.ai/wiki/convolutional-network`. Online; accessed 12 March 2019.

[110] Y.-L. Boureau, J. Ponce, and Y. LeCun, "A theoretical analysis of feature pooling in visual recognition," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 111–118, 2010.

[111] M. D. Zeiler and R. Fergus, "Stochastic pooling for regularization of deep convolutional neural networks," *arXiv preprint arXiv:1301.3557*, 2013.

[112] N. Buduma, *Fundamentals of Deep Learning.* O'Reilly Media, Inc., 2017.

[113] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, *et al.*, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[114] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," tech. rep., Citeseer, 2009.

[115] C.-Y. Lee, P. W. Gallagher, and Z. Tu, "Generalizing pooling functions in convolutional neural networks: Mixed, gated, and tree," in *Artificial intelligence and statistics*, pp. 464–472, 2016.

[116] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[117] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," *CoRR*, vol. abs/1311.2524, 2013.

[118] R. B. Girshick, “Fast R-CNN,” *CoRR*, vol. abs/1504.08083, 2015.

[119] S. Ren, K. He, R. B. Girshick, and J. Sun, “Faster R-CNN: towards real-time object detection with region proposal networks,” *CoRR*, vol. abs/1506.01497, 2015.

[120] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, “You only look once: Unified, real-time object detection,” *CoRR*, vol. abs/1506.02640, 2015.

[121] G. Cohen, S. Afshar, J. Tapson, and A. van Schaik, “Emnist: an extension of mnist to handwritten letters,” *arXiv preprint arXiv:1702.05373*, 2017.

[122] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, “Rethinking the inception architecture for computer vision,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2818–2826, 2016.

[123] A. Dadhich, *Practical Computer Vision*. Packt Publishing Ltd, 2018.

[124] N. Dalal and B. Triggs, “Histograms of oriented gradients for human detection,” *international Conference on computer vision & Pattern Recognition (CVPR’05)*, vol. 1, 2005.

[125] K. Sung and T. Poggio, “Example based learning for view-based human face detection (no. ai-m-1521),” *Massachusetts Inst Of Tech Cambridge Artificial Intelligence Lab*, 1994.

[126] A. Y. Appiah, X. Zhang, B. B. K. Ayawli, and F. Kyeremeh, “Long short-term memory networks based automatic feature extraction for photovoltaic array fault diagnosis,” *IEEE Access*, vol. 7, pp. 30089–30101, 2019.

[127] W. Dong, N. Zhou, J.-C. Paul, and X. Zhang, “Optimized image resizing using seam carving and scaling,” *ACM Transactions on Graphics (TOG)*, vol. 28, no. 5, p. 125, 2009.

[128] M. Grundland and N. A. Dodgson, “Decolorize: Fast, contrast enhancing, color to grayscale conversion,” *Pattern Recognition*, vol. 40, no. 11, pp. 2891–2896, 2007.

[129] S. K. Kopparapu and M. Satish, “Optimal gaussian filter for effective noise filtering,” *CoRR*, vol. abs/1406.3172, 2014.

[130] P. S. Heckbert, “Filtering by repeated integration,” in *ACM SIGGRAPH Computer Graphics*, vol. 20, pp. 315–321, ACM, 1986.

[131] B. Weiss, “Fast median and bilateral filtering,” *Acm Transactions on Graphics (TOG)*, vol. 25, no. 3, pp. 519–526, 2006.

[132] S. Di Zenzo, “A note on the gradient of a multi-image,” *Computer vision, graphics, and image processing*, vol. 33, no. 1, pp. 116–125, 1986.

[133] M. Koziarski and B. Cyganek, “Image recognition with deep neural networks in presence of noise–dealing with and taking advantage of

distortions," *Integrated Computer-Aided Engineering*, vol. 24, no. 4, pp. 337–349, 2017.

[134] "Gaussian filter example." `https://www.raywenderlich.com/167-uivisualeffectview-tutorial-getting-started`. Online; accessed 07 April 2019.

[135] T. K. Kim, J. K. Paik, and B. S. Kang, "Contrast enhancement system using spatially adaptive histogram equalization with temporal filtering," *IEEE Transactions on Consumer Electronics*, vol. 44, no. 1, pp. 82–87, 1998.

[136] J. A. Stark, "Adaptive image contrast enhancement using generalizations of histogram equalization," *IEEE Transactions on image processing*, vol. 9, no. 5, pp. 889–896, 2000.

[137] S. J. Shaffer, R. S. Dunbar, S. V. Hsiao, and D. G. Long, "A median-filter-based ambiguity removal algorithm for nscat," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 29, no. 1, pp. 167–174, 1991.

[138] S. Esakkirajan, T. Veerakumar, A. N. Subramanyam, and C. PremChand, "Removal of high density salt and pepper noise through modified decision based unsymmetric trimmed median filter," *IEEE Signal processing letters*, vol. 18, no. 5, pp. 287–290, 2011.

[139] "Histogram equalisation example." https://medium.com/@animeshsk3/back-to-basics-part-1-histogram-equalization-in-image-processing-f607f33c5d55. Online; accessed 07 April 2019.

[140] "Median filter example." https://stackoverflow.com/questions/18427031/median-filter-with-python-and-opencv. Online; accessed 07 April 2019.

[141] Wikipedia contributors, "Rotation matrix — Wikipedia, the free encyclopedia," 2019. [Online; accessed 7-April-2019].

[142] J. Canny, "A computational approach to edge detection," in *Readings in computer vision*, pp. 184–203, Elsevier, 1987.

[143] S. Birchfield, "Elliptical head tracking using intensity gradients and color histograms," in *Proceedings. 1998 IEEE Computer Society conference on computer vision and pattern recognition (Cat. No. 98CB36231)*, pp. 232–237, IEEE, 1998.

[144] A. Neubeck and L. Van Gool, "Efficient non-maximum suppression," in *18th International Conference on Pattern Recognition (ICPR'06)*, vol. 3, pp. 850–855, IEEE, 2006.

[145] E. C. Stoner and E. Wohlfarth, "A mechanism of magnetic hysteresis in heterogeneous alloys," *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences*, vol. 240, no. 826, pp. 599–642, 1948.

[146] Wikipedia contributors, "Canny edge detector — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Canny_edge_detector&oldid=886388901, 2019. [Online; accessed 8-April-2019].

[147] C. G. Harris, M. Stephens, *et al.*, "A combined corner and edge detector.," in *Alvey vision conference*, vol. 15, pp. 10–5244, Citeseer, 1988.

[148] "Harris corner detection example." `https://dzone.com/articles/corner-detection-opencv`. Online; accessed 08 April 2019.

[149] M. Kass, A. Witkin, and D. Terzopoulos, "Snakes: Active contour models," *International journal of computer vision*, vol. 1, no. 4, pp. 321–331, 1988.

[150] T. F. Chan and L. A. Vese, "Active contours without edges," *IEEE Transactions on image processing*, vol. 10, no. 2, pp. 266–277, 2001.

[151] D. Mumford and J. Shah, "Optimal approximations by piecewise smooth functions and associated variational problems," *Communications on pure and applied mathematics*, vol. 42, no. 5, pp. 577–685, 1989.

[152] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[153] S. Suzuki *et al.*, "Topological structural analysis of digitized binary images by border following," *Computer vision, graphics, and image processing*, vol. 30, no. 1, pp. 32–46, 1985.

[154] M. A. Balafar, A. R. Ramli, M. I. Saripan, and S. Mashohor, "Review of brain mri image segmentation methods," *Artificial Intelligence Review*, vol. 33, no. 3, pp. 261–274, 2010.

[155] V. Grau, A. U. J. Mewes, M. Alcaniz, R. Kikinis, and S. K. Warfield, "Improved watershed transform for medical image segmentation using prior information," *IEEE Transactions on Medical Imaging*, vol. 23, pp. 447–458, April 2004.

[156] X. Artaechevarria, A. Munoz-Barrutia, and C. Ortiz-de Solórzano, "Combination strategies in multi-atlas image segmentation: application to brain mr data," *IEEE transactions on medical imaging*, vol. 28, no. 8, pp. 1266–1277, 2009.

[157] G. A. Hance, S. E. Umbaugh, R. H. Moss, and W. V. Stoecker, "Unsupervised color image segmentation: with application to skin tumor borders," *IEEE Engineering in Medicine and Biology Magazine*, vol. 15, no. 1, pp. 104–111, 1996.

[158] Y.-L. Huang and D.-R. Chen, "Watershed segmentation for breast tumor in 2-d sonography," *Ultrasound in medicine & biology*, vol. 30, no. 5, pp. 625–632, 2004.

[159] S. Beucher, "Use of watersheds in contour detection," in *Proceedings of the International Workshop on Image Processing*, CCETT, 1979.

[160] F. Meyer and S. Beucher, "Morphological segmentation," *Journal of visual communication and image representation*, vol. 1, no. 1, pp. 21–46, 1990.

[161] R. Hirata Jr, F. C. Flores, J. Barrera, R. de Alencar Lotufo, and F. Meyer, "Color image gradients for morphological segmentation.," in *SIBGRAPI*, pp. 316–326, 2000.

[162] L. Najman and M. Schmitt, "Watershed of a continuous function," *Signal Processing*, vol. 38, no. 1, pp. 99–112, 1994.

[163] "Watershed segmentation example." `https://uk.mathworks.com/help/images/marker-controlled-watershed-segmentation.html;jsessionid=e29edad4ef391077c084a5fd58e3`. Online; accessed 08 April 2019.

[164] G. Kerkyacharian, D. Picard, L. Birgé, P. Hall, O. Lepski, E. Mammen, A. Tsybakov, G. Kerkyacharian, and D. Picard, "Thresholding algorithms, maxisets and well-concentrated bases," *Test*, vol. 9, no. 2, pp. 283–344, 2000.

[165] J. Sauvola and M. Pietikäinen, "Adaptive document image binarization," *Pattern recognition*, vol. 33, no. 2, pp. 225–236, 2000.

[166] P. K. Sahoo, S. Soltani, and A. K. Wong, "A survey of thresholding techniques," *Computer vision, graphics, and image processing*, vol. 41, no. 2, pp. 233–260, 1988.

[167] "Thresholding example." `https://docs.opencv.org/3.4.0/d7/d4d/tutorial_py_thresholding.html`. Online; accessed 08 April 2019.

[168] N. Dhanachandra, K. Manglem, and Y. J. Chanu, "Image segmentation using k -means clustering algorithm and subtractive clustering algorithm," *Procedia Computer Science*, vol. 54, pp. 764 – 771, 2015.

[169] M. M. Trivedi and J. C. Bezdek, “Low-level segmentation of aerial images with fuzzy clustering,” *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 16, no. 4, pp. 589–598, 1986.

[170] S. Ray and R. H. Turi, “Determination of number of clusters in k-means clustering and application in colour image segmentation,” in *Proceedings of the 4th international conference on advances in pattern recognition and digital techniques*, pp. 137–143, Calcutta, India, 1999.

[171] H. Ng, S. Ong, K. Foong, P. Goh, and W. Nowinski, “Medical image segmentation using k-means clustering and improved watershed algorithm,” in *2006 IEEE Southwest Symposium on Image Analysis and Interpretation*, pp. 61–65, IEEE, 2006.

[172] J. Long, E. Shelhamer, and T. Darrell, “Fully convolutional networks for semantic segmentation,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3431–3440, 2015.

[173] O. Ronneberger, P. Fischer, and T. Brox, “U-net: Convolutional networks for biomedical image segmentation,” in *International Conference on Medical image computing and computer-assisted intervention*, pp. 234–241, Springer, 2015.

[174] R. Smith, “An overview of the tesseract ocr engine,” in *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007)*, vol. 2, pp. 629–633, IEEE, 2007.

[175] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfel-

low, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *CoRR*, vol. abs/1603.04467, 2016.

[176] F. Chollet *et al.*, "Keras." `https://keras.io`, 2015.

[177] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," in *NIPS-W*, 2017.

[178] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[179] R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky, Y. Bengio, A. Bergeron, J. Bergstra, V. Bisson, J. Bleecher Snyder, N. Bouchard, N. Boulanger-Lewandowski, X. Bouthillier, A. de Brébisson, O. Breuleux, P.-L. Carrier, K. Cho, J. Chorowski, P. Christiano, T. Cooijmans, M.-A. Côté, M. Côté, A. Courville, Y. N. Dauphin, O. Delalleau, J. Demouth, G. Desjardins, S. Dieleman, L. Dinh, M. Ducoffe, V. Dumoulin, S. Ebrahimi Kahou, D. Erhan, Z. Fan, O. Firat, M. Germain, X. Glorot, I. Goodfellow, M. Gra-

ham, C. Gulcehre, P. Hamel, I. Harlouchet, J.-P. Heng, B. Hidasi, S. Honari, A. Jain, S. Jean, K. Jia, M. Korobov, V. Kulkarni, A. Lamb, P. Lamblin, E. Larsen, C. Laurent, S. Lee, S. Lefrancois, S. Lemieux, N. Léonard, Z. Lin, J. A. Livezey, C. Lorenz, J. Lowin, Q. Ma, P.-A. Manzagol, O. Mastropietro, R. T. McGibbon, R. Memisevic, B. van Merriënboer, V. Michalski, M. Mirza, A. Orlandi, C. Pal, R. Pascanu, M. Pezeshki, C. Raffel, D. Renshaw, M. Rocklin, A. Romero, M. Roth, P. Sadowski, J. Salvatier, F. Savard, J. Schlüter, J. Schulman, G. Schwartz, I. V. Serban, D. Serdyuk, S. Shabanian, E. Simon, S. Spieckermann, S. R. Subramanyam, J. Sygnowski, J. Tanguay, G. van Tulder, J. Turian, S. Urban, P. Vincent, F. Visin, H. de Vries, D. Warde-Farley, D. J. Webb, M. Willson, K. Xu, L. Xue, L. Yao, S. Zhang, and Y. Zhang, "Theano: A Python framework for fast computation of mathematical expressions," *arXiv e-prints*, vol. abs/1605.02688, May 2016.

[180] E. D. D. Team, "Deeplearning4j: Open-source distributed deep learning for the jvm, apache software foundation license 2.0.." `http://deeplearning4j.org`.

[181] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *CoRR*, vol. abs/1512.01274, 2015.

[182] F. Seide and A. Agarwal, "Cntk: Microsoft's open-source deep-learning toolkit," in *Proceedings of the 22Nd ACM SIGKDD International Con-*

ference on Knowledge Discovery and Data Mining, KDD '16, (New York, NY, USA), pp. 2135–2135, ACM, 2016.

[183] "Matlab optimization toolbox," 2014. The MathWorks, Natick, MA, USA.

[184] E. Jones, T. Oliphant, P. Peterson, *et al.*, "SciPy: Open source scientific tools for Python," 2001–. [Online; accessed ¡today¿].

[185] W. McKinney, "Data structures for statistical computing in python," in *Proceedings of the 9th Python in Science Conference* (S. van der Walt and J. Millman, eds.), pp. 51 – 56, 2010.

[186] S. van der Walt, S. C. Colbert, and G. Varoquaux, "The numpy array: A structure for efficient numerical computation," *Computing in Science Engineering*, vol. 13, pp. 22–30, March 2011.

[187] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, *et al.*, "Scikit-learn: Machine learning in python," *Journal of machine learning research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[188] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in science & engineering*, vol. 9, no. 3, p. 90, 2007.

[189] J. Zacharias, M. Barz, and D. Sonntag, "A survey on deep learning toolkits and libraries for intelligent user interfaces," *arXiv preprint arXiv:1803.04818*, 2018.

[190] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. S. Bernstein, A. C. Berg, and F. Li, "Imagenet large scale visual recognition challenge," *CoRR*, vol. abs/1409.0575, 2014.

[191] P. J. Grother, "Nist special database 19. nist handprinted forms and characters database," 2016.

[192] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," *CoRR*, vol. abs/1708.07747, 2017.

[193] Wikipedia contributors, "Euclidean distance — Wikipedia, the free encyclopedia." `https://en.wikipedia.org/w/index.php?title=Euclidean_distance&oldid=890387216`, 2019. [Online; accessed 14-May-2019].

[194] A. Balter, "How to calculate circularity." `https://sciencing.com/calculate-circularity-5138742.html`, 2019. [Online; accessed 14-May-2019].

[195] "Mnist example." `https://www.analyticsindiamag.com/top-10-popular-publicly-available-datasets-deep-learning-research/`. Online; accessed 30 May 2019.

[196] A. van Schaik and J. Tapson, "Online and adaptive pseudoinverse solutions for elm weights," *Neurocomputing*, vol. 149, pp. 233–238, 2015.

Appendices

Appendix A

Software GitHub Repository

The following is a link to the GitHub repository which contains the software prototype, implemented in the Python programming language.

`https://github.com/Ross-Byrne/Hand-Drawn-Graph-Parser`