Automated autoencoder neural network architecture design

Z Boonzaaier

orcid.org 0000-0009-5989-7049

Dissertation accepted in fulfilment of the requirements for the degree *Master of Science in Computer Science* at the North-West University

Supervisor: Prof JV du Toit

Graduation: July 2025

PREFACE

The AANNAD (Automated Autoencoder Neural Network Architecture Design) project has been a fascinating expedition into the world of artificial intelligence and has granted me but a mere glimpse into the vast potential that this field may hold. Born from a deep passion for neural networks, this research represents the first step in what I hope will be a long and fulfilling academic journey, filled with many more dissertations and contributions to this ever-evolving domain.

This project has been both a challenging and rewarding endeavour, which has caused not only my knowledge to expand, but also my skillset to advance. I am immensely grateful for the opportunity to undertake this research, and it is my sincerest wish that it serves as a meaningful contribution for future advancements in the vast field known as artificial intelligence.

However, this research would not have been possible without the support and guidance of several incredible individuals who played pivotal roles in its development. First and foremost, I want to express my deepest, indescribable gratitude to my wonderful, beautiful mother, Dalene, and my ever-supportive, ever vigilant father, Evert. Mom and Dad, your unwavering encouragement, patience, and love have been the foundation upon which I built this work. Through the trials and tribulations of this dissertation, you have been the wind at my back and my ever-present voices of encouragement, and I cannot express how fortunate I am to have such incredible parents. Your guidance, wisdom, and, of course, the steady stream of coffee have truly been the backbone of this research.

Next, I extend a heartfelt and special thank you to the best study leader I have ever had the pleasure of working with. Professor Tiny du Toit, your revisions, recommendations and corrections are in and of themselves indescribable. Without you I can say with the utmost certainty that none of this project would have gone as smoothly as it had. Through thick and thin, you had stuck with me and a consistent amount of effort, care and dedication to this research has been nothing short of inspiring.

Finally, I extend my appreciation to everyone who has contributed in any way, shape or form, whether through technical discussions, support, or encouragement. This work is a testament to the collective efforts of those who have inspired and guided me along the way.

I am endlessly grateful for the journey I have undertaken and for the lessons I have learned along the way.

ABSTRACT

Within recent years, there has been a sharp rise in the volume of raw data produced, analysed, and utilised across various fields. With the rapid evolution of technology, information systems, and data acquisition methodologies, this exponential growth necessitates advanced systems capable of efficiently processing, analysing, and interpreting large datasets. Autoencoders, a class of neural networks, have demonstrated significant potential in applications, such as dimensionality reduction, feature extraction, anomaly detection, thereby making them powerful tools for managing vast amounts of data. However, the manual design and development of autoencoder architectures are complex, time-consuming, and resource-intensive tasks that require expert knowledge, often leading to suboptimal models. The primary aim of this study is to develop an automated algorithm for constructing accurate autoencoder neural network architectures. The proposed Automated Autoencoder Neural Network Architecture Design (AANNAD) algorithm automates the hyperparameter selection process, iteratively generating and improving autoencoder models until a predefined resource limit is reached. This study follows a positivistic research paradigm and employs a Design Science Research methodology to evaluate the performance of the AANNAD algorithm across three datasets consisting of different data types originating from various application areas. These include the MNIST, SOCOFing, and credit card fraud detection datasets. The algorithm's performance is measured against traditional methods using reconstruction error metrics. The results demonstrate that the AANNAD algorithm consistently produces high-performing autoencoder architectures, often surpassing models specifically designed for their respective tasks. These findings suggest that the AANNAD algorithm may significantly streamline the development process of deep learning models, reducing time and human effort while maintaining or improving accuracy.

Keywords: architecture optimisation, autoencoder, AutoML, credit card fraud detection, deep learning, hyperparameter optimisation, MNIST, neural network, SOCOFing.

OPSOMMING

In die afgelope paar jaar was daar oor verskeie velde heen 'n skerp toename in die volume rou data wat geproduseer, geanaliseer en gebruik word. Met die vinnige ontwikkeling van tegnologie, inligtingstelsels en data-insamelingsmetodologieë, noodsaak hierdie eksponensiële groei gevorderde stelsels wat in staat is om groot datastelle doeltreffend te verwerk, te analiseer en te interpreteer. Outo-enkodeerders, 'n klas van neurale netwerke, het beduidende potensiaal getoon in toepassings, soos dimensievermindering, kenmerkonttrekking, anomalië-opsporing en meer, wat hulle kragtige instrumente maak vir die bestuur van groot hoeveelhede data. Die handmatige ontwerp en ontwikkeling van outo-enkodeerderargitekture is egter komplekse, tydrowende en hulpbronintensiewe take wat kundige kennis vereis en dikwels tot sub-optimale modelle lei. Die primêre doel van hierdie studie is om 'n outomatiese algoritme te ontwikkel vir die konstruksie van akkurate outo-enkodeerder neurale netwerkargitekture. Die voorgestelde Outomatiese Outo-enkodeerder Neurale Netwerkargitektuurontwerp (AANNAD) algoritme outomatiseer die hiperparameterseleksieproses en genereer en verbeter iteratief outo-enkodeerdermodelle totdat 'n voorafbepaalde hulpbronlimiet bereik is. Hierdie studie volg 'n positivistiese navorsingsparadigma en maak gebruik van ’n Ontwerp-wetenskaplike navorsingsmetodologie om die prestasie van die AANNAD algoritme oor drie datastelle, bestaande uit verskillende datatipes afkomstig van verskeie toepassingsareas, te evalueer. Dit sluit in die MNIST-, SOCOFing- en kredietkaartbedrogopsporingsdatastelle. Die algoritme se prestasie word gemeet teenoor tradisionele metodes met behulp van rekonstruksie foutmaatstawwe. Die resultate toon dat die AANNAD algoritme konsekwent hoë prestasie outo-enkodeerderargitekture lewer, wat dikwels modelle oortref wat spesifiek vir hul onderskeie take ontwerp is. Hierdie bevindinge suggereer dat die AANNAD algoritme die ontwikkelingsproses van diep leer modelle aansienlik kan bespoedig, wat tyd en menslike moeite verminder, terwyl akkuraatheid gehandhaaf of verbeter word.

Sleutelwoorde: AutoML, argitektuuroptimering, diep leer, hiperparameteroptimering, kredietkaartbedrogopsporing, MNIST, neurale netwerk, outo-enkodeerder, SOCOFing.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1 GENERAL INTRODUCTION AND PROBLEM CONTEXTUALISATION

1.1 Introduction

The rate of increase in data production and availability is presently on the high side, to the extent that the daily production is estimated at around 2.5 exabytes (Jayagopal & Basser, 2022). This increase can be attributed to technological advances, such as the availability of IoT devices, internet facilities, smartphones, users' activities on social media, an increase in research activities and many more. Processing huge data sizes to extract useful information requires more effective and efficient systems (Pinaya *et al.*, 2020).

Feature engineering, defined as the process of constructing features from raw data, is crucial for many machine learning algorithms (Zhu *et al.*, 2022). The goal is to transform raw data into features that can better express the nature of a situation or dataset. Recently, the implementation of feature engineering has shifted from leveraging human-based information and knowledge to automated methods (Zhu *et al.*, 2022).

Different approaches have been considered in data processing and analysis, and the list is not limited to data mining (Gupta & Chandra, 2020), data science (Provost & Fawcett, 2013) and machine learning (Rahmaty, 2023). The machine learning approach has sub-classes, such as supervised, unsupervised, and reinforcement learning. Among the subclasses, the unsupervised learning approach can automatically generate patterns from a large dataset without formal assistance (Rahmaty, 2023). Apart from the stated machine learning categories, machine learning can also be classified into classical and deep learning. While classical machine learning depends primarily on handcrafted feature engineering, deep learning, on the other hand, is an advanced neural network that automatically extracts its features from a pool of data samples (Picon *et al.*, 2020). Therefore, unsupervised deep learning can extract features from large data samples without formal assistance.

An autoencoder is an unsupervised deep learning model that is prominent and proficient for automatic feature engineering from large datasets; that is, an autoencoder can automatically learn useful features and representations from data, simplifying feature engineering in various machine learning practices and studies (Pinaya *et al.*, 2020). Autoencoders primarily function by taking data in its original or high-dimensional space and projecting it into a new lower-dimensional space from which it can be accurately restored (Sewak *et al.*, 2020). The concept of autoencoders has been a fundamental part of neural network research since the 1980s (Pinaya *et al.*, 2020).

Autoencoders consist of hidden nodes that are constructed in varying hierarchical layers (Sewak *et al.*, 2020). These different layers comprise input layers, hidden layers and output layers. The weight values, densities, and layouts of these components significantly determine the functionality of a neural network. Therefore, the configuration of these components creates a neural network architecture that can be used for various purposes, depending on the dataset provided.

Autoencoders have found applications across various domains. In healthcare, they are used for anomaly detection in medical imaging and aiding in early disease detection (El-Shafai *et al.*, 2022). In finance, autoencoders help in fraud detection by identifying unusual patterns in transaction data (Misra *et al.*, 2020). Image processing has significantly benefited from autoencoders, particularly in tasks, such as image denoising, compression, and generation (Theis *et al.*, 2022). Generally, autoencoders can reduce dimensionality and be used at the feature extraction phase for many machine learning applications.

Data variation, redundancy, and anomalies are inevitable, given the vast amounts of information autoencoders process. These must be addressed within the autoencoder architecture to utilise the information effectively. While autoencoders have shown remarkable progress, most implementations require adjustments within their various architectural features for optimal functioning (Sewak *et al.*, 2020).

Like other deep networks, the accuracy and convergence of autoencoders are heavily dependent on hyperparameters, such as the number of hidden layers and the number of neurons utilised within these layers for network construction (Nikbakht *et al.*, 2021). Since hyperparameter tuning determines the performance of autoencoders, it suffices to find a means of creating an autoencoder with appropriate hyperparameters for a task choice. This research focuses on developing an automated autoencoder architecture design algorithm to generate accurate autoencoders to ease the developmental lifecycle efficiently.

The remainder of the chapter will be presented as follows. In Section 1.2, the research problem statement will be described. The research questions guiding this study will be introduced in Section 1.3, followed by the primary aim and secondary research goals that aid in addressing the research questions in Section 1.4. In Section 1.5, the paradigm and research design underpinning the study will be explained, outlining the research methodology used to develop and test the proposed Automated Autoencoder Neural Network Architecture Design (AANNAD) algorithm. Ethical considerations related to the study will be addressed in Section 1.6. An outline of the chapter division for the research will be provided in Section 1.7. Finally, a summary of the chapter will be given in Section 1.8.

1.2 Research problem statement

An autoencoder inherits features of deep learning models, such as layer-by-layer learning, that come with randomly creating layers with the associated number of neurons. The main autoencoder hyperparameters are layers, neurons, and activation functions (Berahmand *et al.*, 2024). However, the adequate hyperparameters for an optimal model cannot be predetermined (Berahmand *et al.*, 2024). The manual method of setting autoencoder hyperparameters comes with challenges, such as difficulty finding the optimal structure, time-wasting on hyperparameter tuning, network scaling, reusability challenges, and others (Chen & Guo, 2023). The manual approach to creating an autoencoder may confine the model to non-generalisation because it may require retraining or fine-tuning to adapt to a new environment.

Considering the limitations of the manual method of developing an autoencoder model, a method that will automatically develop an accurate autoencoder model is needed.

1.3 Research questions

The following three research questions will guide the study on how to develop an accurate automated autoencoder model:

- What are the required resources or information for developing the automatic design of the autoencoder?
- How can a novel algorithm be developed to automate the hyperparameter selection process for an accurate autoencoder neural network architecture?
- What will be the mode of evaluating the designed automatic autoencoder algorithm?

The different research goals determined crucial for answering these research questions are discussed in the following section.

1.4 Research goals

Based on the research questions, the primary aim of this study is the following:

- To develop a novel automated autoencoder neural network architecture construction algorithm to create accurate autoencoder architectures using Design Science Research and a positivism approach.

The secondary objectives that provide solutions to the research questions are as follows:

1. To perform a literature study on deep learning, autoencoder architectures and the automated design of neural network architectures.

2. To create a novel algorithm to automatically design an accurate autoencoder neural network.
3. To determine the success of the automated construction algorithm by considering applicable metrics.

The different research design principles implemented within the study, including the research paradigm and the research methods implemented, are described in the following section.

1.5 Research design

The research conducted in this study is grounded in the positivist paradigm, which emphasises observable and measurable outcomes (Kankam, 2019). In line with this, the study follows the Design Science Research (DSR) methodology, a problem-solving approach aimed at creating and evaluating artefacts to address specific challenges (Simon, 1996; Vom Brocke *et al.*, 2020). Both of these concepts are explored within the following subsections.

1.5.1 Research paradigm

A research paradigm is considered a system of fundamental beliefs and theoretical frameworks concerning epistemology, ontology, methodology, methods and axiology (Rehman & Alharthi, 2016). Epistemology describes how certain information is determined or becomes known. It also explains how we know the truth or reality (Kivunja & Kuyini, 2017). Ontology is more concerned with the assumptions that humans make when believing that a concept makes logical sense. Methodology is a broad term referring to the research methods, approaches, and procedures used during research. Methods refer to the specific means to collect and analyse data, such as interviews or questionnaires. Finally, axiology refers to the various ethical considerations that need to be taken into account when planning research ventures. It considers the philosophical approach to making decisions of value or the right choices.

The research paradigm utilised in this study is positivism. This paradigm was proposed initially by Comte (1856) and defines a perception of research grounded in factual and quantifiable information as the predominant scientific method of investigation. Comte proposed that experimentation, observation, and reason, based on experience, should be the basis for understanding not only human behaviour, but also the behaviour of various systems. The positivistic research paradigm assumes that reality exists independently of humans and is not mandated by senses and emotions, but follows a collection of immutable laws.

This study follows the positivist paradigm because it emphasises observable and measurable outcomes and largely implements statistical analysis to validate findings. The research method employed in this study is outlined in the following subsection.

1.5.2 Research method

This study follows a Design Science Research (DSR) methodology to construct the AANNAD algorithm, which will be tested and compared to various baseline studies on three datasets (Simon, 1996; Vom Brocke *et al.*, 2020). DSR focuses mainly on creating and evaluating artefacts, such as models, methods, or algorithms that can improve the environment in which they are applied. The goal of DSR is twofold: to create functional solutions to specific problems and to generate design knowledge that deepens the understanding of why and how these different artefacts can enhance or influence their application contexts.

DSR places considerable focus on analysing the academic knowledge base relevant to a given problem to solve the problem. The design activities related to DSR also comprise building a solution and evaluating the presented solution, typically in various iterations. This study employs the DSR methodology to develop the AANNAD algorithm, which automates the design of accurate autoencoder neural network architectures. The DSR methodology was chosen as its core principles involve iterative cycles of building and evaluating artefacts to ensure that it meets the research objectives and effectively solves identified problems. In the following section, ethical considerations related to this study are discussed.

1.6 Ethical considerations

The proposed research has been presented to the Faculty of Natural and Agricultural Sciences' ethics committee for ethical clearance. The study was approved as a zero/no-risk study in the Faculty of Natural and Agricultural Sciences of the North-West University under the reference number NWU-00573-21-A9.

Every dataset utilised in this study has been cleared of personal information and is available in the public domain. If any additional ethical concerns arise during the study, the appropriate ethics committee will be contacted before proceeding. The chapter division for the research is provided next.

1.7 Outline of chapters

The study is divided into the following six chapters.

Chapter 1 - General introduction and problem contextualisation

Chapter 1 is used to introduce the study by providing background information, the research questions, the study's primary aim, and the secondary objectives that are intended to aid in answering the research questions. This chapter is additionally utilised to outline the research

paradigm, the chosen research methodology, and the ethical considerations relevant to the study. A chapter outline is provided to give an overview of the structure of the dissertation.

Chapter 2 – Artificial intelligence and deep learning

An overview of deep learning, beginning with its origins and historical development, is presented in Chapter 2. Key concepts, such as neural networks, learning structures (supervised, unsupervised, and reinforcement learning), and the components of neural networks are also discussed. The chapter contains information regarding various types of neural network (e.g., feedforward, convolutional, and recurrent networks) and their applications in modern machine learning tasks. Additionally, the challenges and goals faced by deep learning systems, such as overfitting, generalisation, and computational efficiency, are explored. Various topics related to the training of a neural network are discussed.

Chapter 3 – Autoencoders

In Chapter 3, the structure, functionality, and applications of autoencoders are discussed. The chapter documents how autoencoders work, including their encoding and decoding processes and types, such as sparse, denoising, and contractive autoencoders. The areas where autoencoders can be implemented, such as dimensionality reduction, feature extraction, anomaly detection, and generative modelling, are also explored. Finally, a discussion regarding asymmetric autoencoders is presented.

Chapter 4 – Automated neural network architecture design

The literature surrounding the automated construction of neural network architectures, focusing on approaches, such as neural architecture search and hyperparameter optimisation, is explored in Chapter 4. The AANNAD algorithm is also introduced in this chapter. It describes the methodology for automatically generating and optimising accurate autoencoder architectures and explains the evolutionary strategies employed within the AANNAD algorithm.

Chapter 5 – Experimental design

Within Chapter 5, the experimental design implemented to evaluate the AANNAD algorithm is discussed. This chapter covers the acquisition and pre-processing of three distinct datasets and the steps involved in applying the AANNAD algorithm to these datasets, including training, evaluation, and comparison to baseline studies. The chapter also includes information regarding the performance metrics used, such as reconstruction error and computational efficiency, and analyses the algorithm's results in terms of accuracy and robustness. The insights gained from these experiments and the successes encountered are discussed to evaluate the algorithm.

Chapter 6 – Conclusion

The final chapter of the dissertation summarises the research and its findings. The research questions, main aim, and secondary objectives are revisited to determine whether these were successfully met. Key contributions made by the study are discussed, and suggestions on potential areas for future research are made. Furthermore, recommendations for improving the AANNAD algorithm and its applicability to more complex or real-time data environments are addressed.

1.8 Summary

The exponential growth of raw data necessitates advanced systems like deep neural networks to process and comprehend large datasets efficiently. As a subclass of these networks, autoencoders offer significant potential for various applications, including dimensionality reduction, feature extraction, and anomaly detection. In this chapter, the study on developing an accurate automated autoencoder neural network architecture design algorithm is introduced. The following chapter contains a literature review of important and relevant background information on deep learning, which forms the basis of the autoencoder model used in the study.

CHAPTER 2 ARTIFICIAL INTELLIGENCE AND DEEP LEARNING

2.1 Introduction

The study of artificial intelligence, artificial neural networks and deep learning originated from humanity's ambitions to study and simulate the functionality of the human brain, effectively focusing on the production of an imitation mind (Lennox, 2020). As this study focuses on the construction and implementation of deep learning architectures, it is integral to better understand the innovations made in terms of relevant topics. As such, Chapter 2 will contain an introduction to the history and functionality of neural networks and deep learning.

The remainder of this chapter is structured as follows. Section 2.2 contains the history of neural networks, followed by the comprehensive discussion of neural network components and training procedures in Section 2.3. Section 2.4 encompasses the goals and the applications of neural networks and deep learning discussion, and the chapter summary is provided in Section 2.5.

2.2 History

To better understand deep learning, it is beneficial to possess at least a high-level context of the historical events of deep learning. Therefore, this section contains an overview of some of the more important historical events related to the development of deep learning in the modern sense.

There have been roughly three different evolutionary time frames of deep learning (Peters, 2017). These are as follows:

- In the 1940s-1960s, the first wave of deep learning, known as cybernetics, occurred;
- In the 1980s-1990s, the second wave of deep learning, known as connectionism, occurred, and finally
- In 2006-the present, the most recent and current wave, referred to as deep learning, came about.

Figure 2-1 is a graphical representation of the three historical deep learning research waves. According to the Google Books database, these waves are denoted by the recurrent utilisation of words or concepts, such as neural networks, connectionism, cybernetics and deep learning (Google, 2024). The information presented in Figure 2-1 covers the period from 1940 to 2022, the maximum range of time available at the time of writing.

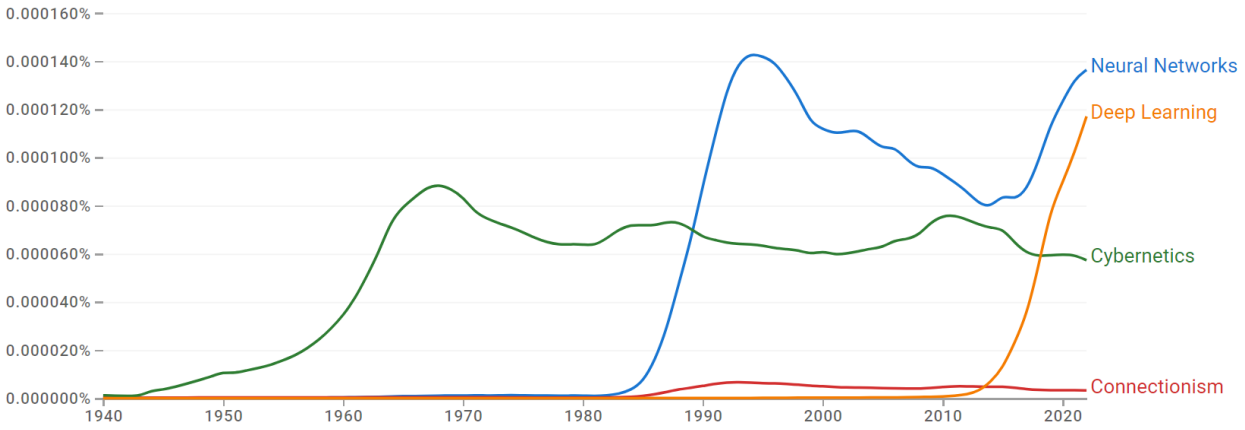

Figure 2-1: Timeline of the three historical deep learning waves (Google, 2024)

Next, an overview of the three different evolutionary time frames of deep learning will be presented chronologically.

2.2.1 Deep learning known as cybernetics (the 1940s-1960s)

The study, documentation and speculation on the functionality of the human brain had been the subject of intrigue for the scientific community for thousands of years, with the earliest scientific mention of the brain being written in the 17th century BCE within the Edwin Smith Surgical Papyrus (Agarwal *et al.*, 2018). Since then, considerable progress has been made in understanding the brain's biology, psychology, and chemistry. This naturally led to the desire to replicate the brain's learning capabilities, ultimately creating artificial neural networks and deep learning.

It is a common misconception that deep learning was a relatively new field when research that significantly impacted the formation of modern-day deep learning already occurred in the 1940s (Hagan *et al.*, 2014). The concept of deep learning would only appear to be a relatively new area of study as it had been viewed in an unpopular light for several years before its current resurgence. Additionally, the term deep learning has been known by numerous other synonyms, but it has only recently been dubbed the moniker of deep learning.

The first artificial neural network was portrayed in 1943 by neurophysiologist Warren McCulloch and mathematician Walter Pitts through a simple electrical circuit (McCulloch & Pitts, 1943). Due to the developments made by Pitts and McCulloch, concepts and practices that would later lead to the creation of deep learning as known today had continuously evolved, despite two infamous artificial intelligence winters. An example of the electrical circuit designed by McCulloch and Pitts is shown in Figure 2-2.

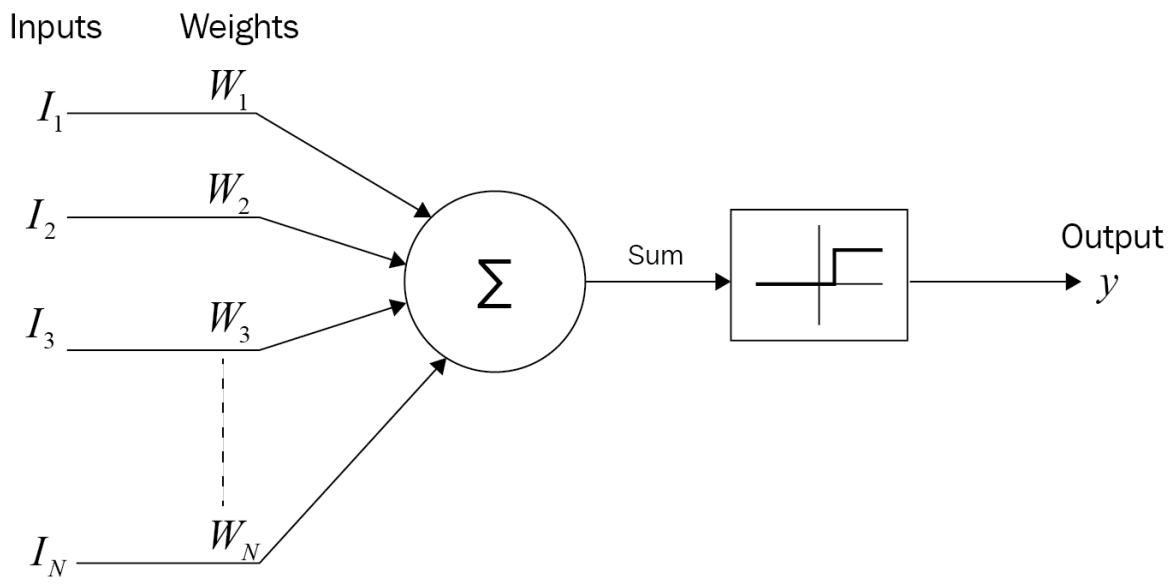

Figure 2-2: A McCulloch-Pitts neuron (Rothman, 2020)

In its most basic form, the inputs (denoted by I) are values that are received from a dataset or other neurons. These values are multiplied by the weights (represented by W), which determines the connection between two neurons. Additionally, a bias value may be added to the calculated total value (denoted by Σ). The weights and biases are the only modified values within a neural network. The neuron then applies an activation function to the value and usually produces a number between one and zero to represent a probability. However, it should be noted that other functions may exist which provide information in a different form.

The research and developments of McCulloch and Pitts would lead to further research on connectionism and neural networks. In 1949, a Canadian psychologist, Donald Hebb, took the idea even further. Hebb proposed that neural pathways strengthen over each successive use, especially between neurons that tend to fire simultaneously (Hebb, 1949). This hypothesis would later be considered a precursor to quantifying the brain's complex processes. Based on the research that Hebb had performed, two integral components of modern-day neural networks had been realised:

- Threshold Logic — This is the act of continuously converting various inputs to discrete outputs, and
- Hebbian Learning — This learning model is primarily based on neural plasticity, often summarised by the phrase: "Cells that fire together, wire together."

These concepts later led to the creation of the first Hebbian network in 1954. Around this time, a psychologist at Cornell named Frank Rosenblatt had been working on understanding the decision systems present within the optical sensors of a fly, which would have adverse effects on its fight

or flight responses. The research conducted here mainly focused on the following: "If the input is equal to X, then the output would resemble Y" principle on the fly's actions. To better understand and quantify this process, Rosenblatt (1961) proposed a perceptron algorithm for supervised learning of binary classifiers. Supervised learning is an approach to creating and training an artificial intelligence model. The input data provided to the learning algorithm is labelled to correspond to a specific output. Rosenblatt's research later led to the first successful perceptron implementation in 1958, creatively named the Mark I Perceptron.

The Mark I Perceptron functioned on a simple input-output relationship modelled on a McCulloch-Pitts neuron. Reflecting the work that Frank Rosenblatt had performed on the decision-making process of a fly, the neuron would take in input signals, modify them by a weighted sum and return a value of 0 if the input value had been perceived to be below a certain threshold, and a value of 1 if the input value had been perceived as being above the threshold. Figure 2-3 is a picture of the first known implementation of a Mark I Perceptron.

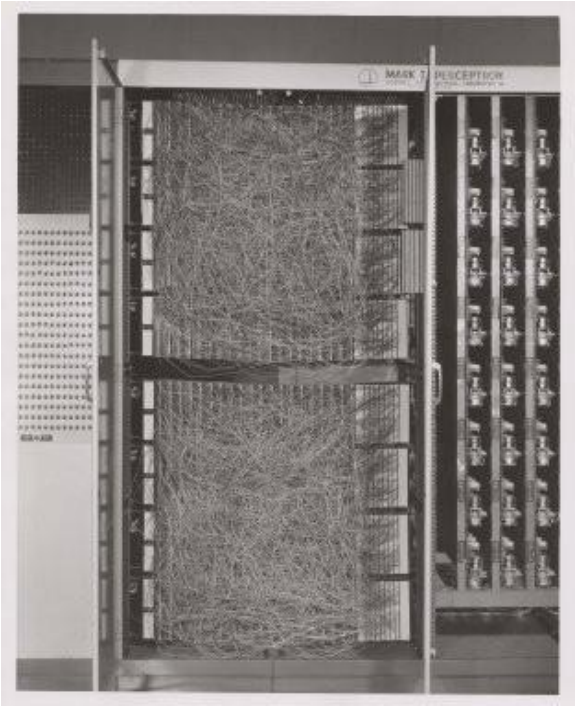

Figure 2-3: The first known implementation of a Mark I Perceptron (Laboratory, 2018)

After successfully implementing the first perceptron, more focus and research were put into developing more advanced neural networks. Two Stanford researchers, Bernard Widrow and Marcian Hoff , successfully created the first functional neural network (Widrow & Hoff, 1960). They had developed an artificial neural network to eliminate noise heard on telephone lines.

The earliest efforts in developing the concepts and practices as discovered by McCulloch and Pitts came from two researchers, Alexey Grigoryevich Ivakhnenko and Valentin Grigor'evich Lapa, in 1965 (Ivakhnenko & Lapa, 1966). They utilised models with polynomial activation functions that were then analysed statistically. The activation function defines the output of that node given an input or set of inputs. From each layer, the best statistically chosen features were forwarded to the next layer.

Unfortunately, after the work performed by Grigoryevich and Lapa, the first artificial intelligence winter occurred in 1974 (Muthukrishnan *et al.*, 2020). An artificial intelligence winter is a period of reduced funding and interest in artificial intelligence research (Jiang *et al.*, 2022). Professor Sir James Lighthill was asked to evaluate the state of artificial intelligence research in the United Kingdom. His report, the Lighthill report, criticised artificial intelligence's failure to achieve its "grandiose objectives." (Agar, 2020). He concluded that there was nothing being done in artificial intelligence that could not be done in other sciences. He was also adamant about stating that many of artificial intelligence's most successful algorithms could not solve real-world problems and were only suitable for solving simple versions of said problems. Due to severe criticism and a drastic decrease in funding, deep learning and artificial intelligence development had nearly ceased entirely, save for a few individuals conducting research without funding.

One such individual was Kunihiko Fukushima, who had developed the first iteration of convolutional neural networks (Fukushima, 1988). This type of neural network consists of multiple pooling and convolutional layers. In 1979, he developed an artificial neural network called the Neocognitron. This architecture used a hierarchical, multi-layered design, which allowed the computer to observe and learn visual patterns. The Neocognitron resembled modern versions of similar neural networks, but had been trained with a reinforcement strategy of recurring activation in multiple layers, which gained strength over time. Additionally, Fukushima's design allowed essential features to be adjusted manually by increasing the weight of specific connections. An example of these utilisations is the implementation of inference. A modern Neocognitron can not only identify patterns with missing information (for example, an incomplete image), but can also complete the image by adding the missing information. Following this, the second wave of deep learning occurred.

2.2.2 Deep learning known as connectionism (the 1980s-1990s)

In the 1980s, neural network research re-emerged in its second wave, with the concept of connectionism as the spearhead (Bengio *et al.*, 2017). The idea of neural networks originated from the creation of models meant to simulate the functionality of the human brain, with the original study being called connectionism (Garson, 1997). Connectionism refers to a concept in cognitive sciences that aims to explain intellectual behaviours and abilities by utilising artificial

neural networks. Consequently, neural networks have initially been intended to function effectively as simplified models of the brain.

During the early 1980s, most scientists concerned with connectionism would focus their research on models of symbolic reasoning. Despite the popularity of the subject at the time, symbolic models had been challenging to describe in terms of how the human brain would have been able to implement them using neurons. Consequently, connectionists started studying different models of cognition that could be grounded within the realm of neural implementations. This research would later revive the research conducted by Donald Hebb in the 1940s.

Several core concepts that first appeared during the 1980s connectionism movement remain active, even in modern deep learning practices (Bengio *et al.*, 2017). One such idea is distributed representation. Distributed representation is a concept that states that many features must represent every input that is present within a system (Liu *et al.*, 2023). Distributed representation additionally states that every feature present within a system should form part of the representation of multiple inputs. Additionally, connectionism was the origin of backpropagation, which had been used to increase the productivity of supervised learning algorithms by calculating the gradient of the error function for the neural network's weights. During this time, backpropagation had started to increase in popularity. Though the popularity of backpropagation has fluctuated over the years, it is still one of the more dominant approaches to training deep learning models in modern times.

At this point, the second artificial intelligence winter would occur in the late 1980s (Jiang *et al.*, 2022). This second occurrence was due to overestimations and exaggerations of artificial intelligence capabilities at that time (Hirsch-Kreinsen, 2024). As a result of this, companies and people would place significant investments into the implementation of artificial intelligence. Unfortunately, the exaggerated expectations of artificial intelligence at the time would not be met, leading to the discontent of the previously hopeful investors. The disappointment in the unrealised potential had reached the point that artificial intelligence had been regarded as pseudoscience.

However, the artificial intelligence winter did cause researchers to become considerably more conservative. This resulted in a paradigm shift within artificial intelligence research to more established theories, such as statistics-based implementations. Additionally, this paradigm shift has also caused the development of public benchmark datasets that have been utilised and updated, even in recent times. These datasets would be integral to rigorous artificial intelligence research, testing and advancement.

Following the end of the second artificial intelligence winter, the next significant evolutionary step regarding deep learning precursors occurred in 1999. Researchers and companies started

utilising graphics processing units (GPUs) to increase the computational power of neural networks (Pandey *et al.*, 2022). From here, neural networks could compete against industry-standard information processing systems, such as support vector machines. Furthermore, neural networks also gained an advantage over other computational paradigms by continuously improving as more training data was presented to the system as input data. An overview of the more recent developments in deep learning will be discussed in the following section.

2.2.3 The modern resurgence of deep learning (2006-the present)

The third and current wave of deep learning initially occurred in 2006 when a researcher named Geoffrey Hinton showed in his research that a class of neural networks referred to as deep belief networks could efficiently undergo training (Hinton *et al.*, 2006). This would be done by utilising a strategy referred to as greedy layer-wise pre-training. Based on the research performed by Hinton, several researchers would use the same approach to train various types of neural networks. Greedy layer-wise pre-training would quickly be adopted and systematically assist researchers in improving generalisations on test examples. As a result, researchers now had the abilities and resources needed to train deeper neural networks than previously feasible. They could now concentrate most of their attention on the theoretical importance of depth. As a result, the term *deep learning* has drastically increased in popularity.

At this time, deep neural networks had the capability to outperform competing artificial intelligence systems created by utilising technologies related to machine learning and hand-designed functionality. Due to this sharp increase in interest, this time frame has been dubbed the third wave of popularity of neural networks and is still in effect. Initially, this wave had begun with a focus on the capabilities of deep learning models to generalise from smaller datasets and new unsupervised learning techniques. However, in more recent years, more interest has been placed on the capabilities of deep learning models to leverage labelled datasets of considerable size and the re-implementation of much older supervised learning algorithms.

Due to our society's digitisation, modern deep learning neural networks have become more popular than ever. Furthermore, many humans utilise different devices daily, generating enormous amounts of data. As these devices become increasingly networked, the centralisation and curation of data into a dataset ideal for utilisation within various machine learning applications become easier. Due to the rise of Big Data, machine learning has become significantly more accessible. The primary challenge of statistical estimation—generalising and adapting efficiently to new data after observing only a limited subset of a larger dataset—has now been considerably reduced.

In recent years, deep learning has experienced tremendous growth in popularity, usefulness, functionality, effectiveness and possible utilisations (Bengio *et al.*, 2017). Examples of the different tasks include pattern recognition tasks, such as object detection, speech recognition and even machine translation. Furthermore, with the deep architecture nature that deep learning possesses, there is a possibility of solving considerably more complicated artificial intelligence tasks in the future. As a result of this, researchers have begun applying deep learning techniques to a wide variety of different modern domains and functions. Examples of these more recent implementations include the applications of a recurrent neural network to de-noise speech signals, stacked autoencoders to discover clustering patterns of gene expressions, and sentiment analysis from multiple modalities simultaneously (Adnan *et al.*, 2021).

The general structure of neural networks will be discussed in the following section. Accompanying this discussion will be a more in-depth look into the functionality and design of autoencoder neural networks, which form the focus of this study.

2.3 Neural network structure

Neural networks were initially developed to simulate the nervous systems of humans to improve machine learning tasks (Aggarwal, 2018). This was done by modelling and treating the computational modules within a machine learning model similar to that of a human neuron. The primary goal of neural network technology is to achieve a more low-level understanding of the human brain and construct machines with a computational architecture that simulates the computational power of the human mind.

In more recent years, different neural network architectures have changed and advanced various application areas of machine learning, such as pattern recognition, natural language processing and computational learning (Liu *et al.*, 2017). Due to the ability of neural networks to improve their functionality and exhibit superior performance when provided with enough structured data, they have also become abundant in practical applications and critical decision processes, such as autonomous driving, medical image analysis and even novel knowledge discovery techniques (Prevedello *et al.*, 2019). However, due to this abundance of application areas, neural networks do not subscribe to a singular architectural layout or utilise a set of universal components. As a result, numerous neural network architectures are currently in circulation, with an example of some of the most popular architectures shown in Figure 2-4.

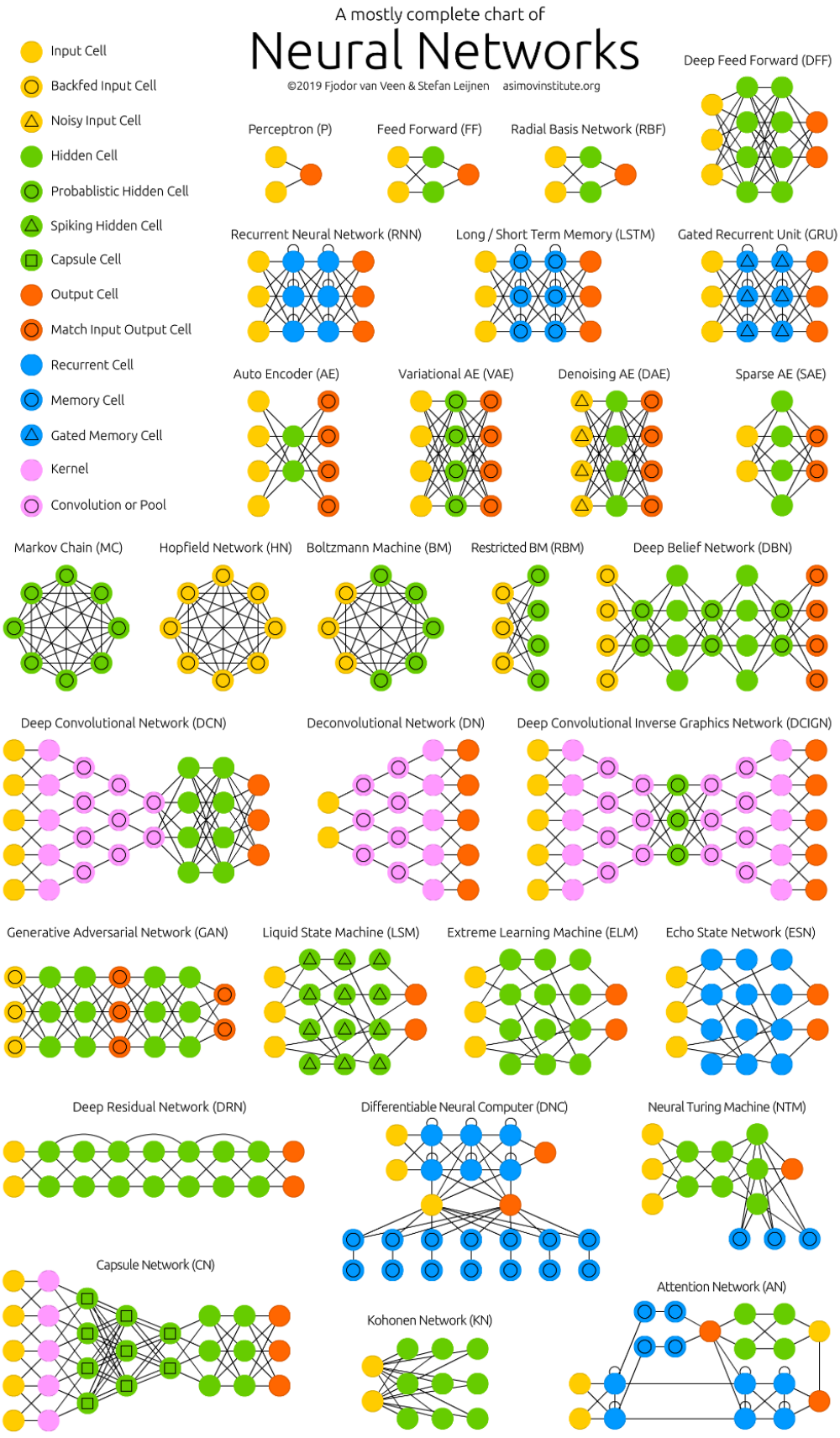

Figure 2-4: Collection of different neural network architectures (Van Veen, 2019)

Composing a complete list of all neural networks in circulation is practically impossible, as new neural network architectures are invented almost daily. Different neural network architectures also consist of various types and quantities of components, such as neurons and hidden layers. Therefore, Figure 2-4 is simply a graphical representation of some of the most utilised neural network architectures accompanied by their most commonly found components. However, it is essential to note that this study mainly focuses on the automated design of autoencoder architectures. Therefore, an overview of the components of neural networks is presented in the following section.

2.3.1 An introduction to the components of artificial neural networks and biological neurons

The human mind is a highly complex decision-making system that consists of millions of units, referred to as neurons, interconnected in various intricate ways (Angelov *et al.*, 2021). It consists of numerous neuron layers that interact with one another in parallel (Mijwel, 2021). This parallel interaction would mean that every single neuron receives some input from various other neurons within previous layers (Bataineh, 2015). These neurons then provide different outputs to many other neurons within the next layer. The human mind consists of the capabilities of solving problems, innovations, art and many more concepts and it is due to these capabilities that we strive to understand and mimic the mind.

The human brain possesses powerful abilities to make proper decisions when facing different tasks, such as performing mathematical calculations, postures, and motions (Bataineh, 2015). This ability is developed by solving and experiencing similar previous tasks. An elementary example would be a child learning that fire is hot as soon as he/she touches it, and because of this, he/she would attempt to avoid touching it in the future.

Based on this learning paradigm, researchers have attempted to artificially duplicate the brain's underlying units in hopes of recreating the ability of the brain to solve different complex practical problems. Just as there are multiple connections within a human brain, artificial neural networks also include units, referred to as neurons, distributed within different layers (Mijwel, 2021). Examples of similar structures can be seen in Figures 2-5 and 2-6.

An artificial neural network's units (parameters) are adjusted and adapted to predict a system output by utilising data and information provided by a system. Similar to biological neurons, an artificial neural network consists of three main parts:

1. Network inputs. The network inputs are represented within a biological neuron by the dendrites in Figure 2-5. The equivalent component in an artificial neural network is the input layer in Figure 2-6.
2. Hidden layers with multiple neuron connections. This is represented by the cell body in Figure 2-5 and by the hidden layers in Figure 2-6.
3. Network outputs. This is represented by the terminal axon in Figure 2-5 and the output layer in Figure 2-6.

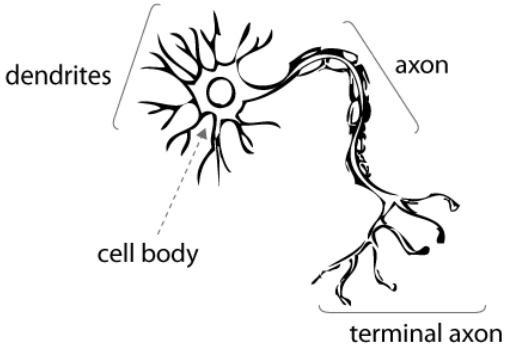

Figure 2-5: A human brain neuron (Bataineh, 2015)

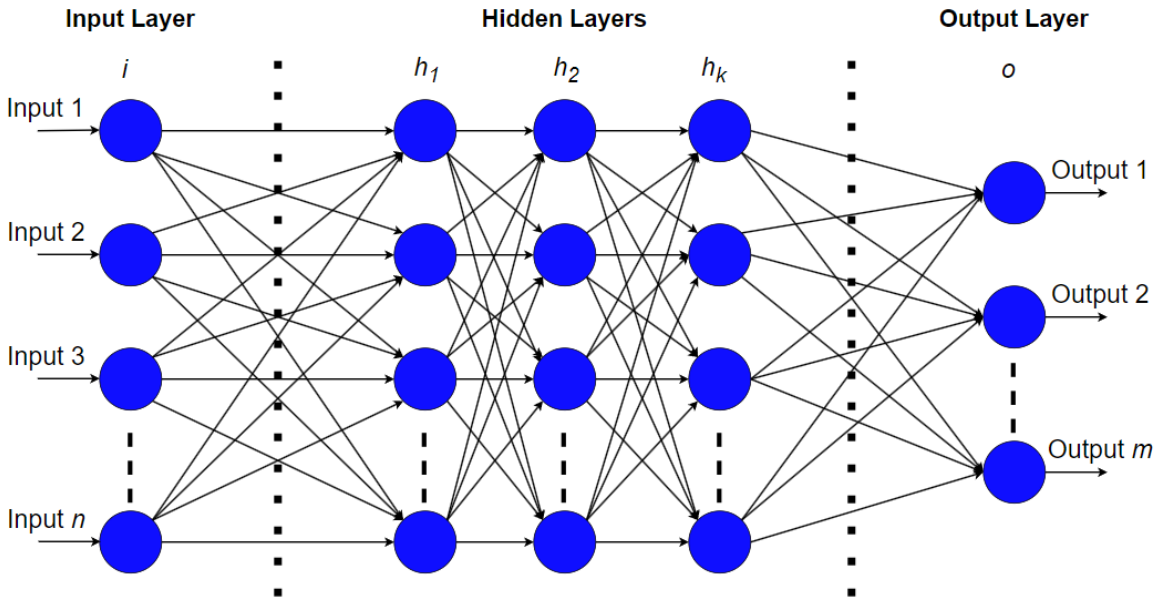

Figure 2-6: An artificial neural network

The hidden layers within an artificial neural network are where the main mathematical calculations occur. These calculations process the various inputs and attempt to provide the most suitable outputs given the structure of the neural network and the information provided. The axon (shown in Figure 2-5) consists of a threshold value, which triggers an output signal as soon as the signal supplied to the axon is larger than its threshold value. Like the axon, there are multiple connections within an artificial neuron called connection weights. The weight value assigned to

each weight is decided by a learning process performed to remember or learn a task (Mijwel, 2021). The most common paradigms utilised in the training of neural networks are discussed in the following section.

2.3.2 Neural network learning structures

Deep learning is considered a type of machine learning algorithm based on artificial neural networks (Janiesch *et al.*, 2021). Therefore, it is essential to possess a decent understanding of the basic principles related to machine learning. Machine learning algorithms are defined as algorithms that possess the ability to learn (Mahesh, 2020). A formal definition of the algorithms studied in the machine learning field, as stated by the American computer scientist Tom Mitchell, concludes that "a computer program is said to learn from experience *E* with respect to some class of tasks *T* and performance measure *P*, if its performance at tasks in *T*, as measured by *P*, improves with experience *E"* (Mitchell, 1997). There are a near-infinite number of experiences, tasks, and measures. In this section, the possible combinations of these parameters will not be detailed; instead, the focus will be on the principles behind some of the most popular machine learning paradigms.

Before discussing learning algorithms, it is essential to note that these algorithms either function using labelled data or unlabelled data (Serrano, 2021). Data refers to any information utilisable by machine learning algorithms to achieve a set goal and usually appears in labelled or unlabelled data. Labels are often subjective to the application of the data. For instance, if the algorithm attempts to predict a particular feature (the age of a pet, for example) based on other features, then that feature (the age as a number) is a label. As such, labelled datasets are datasets accompanied by labels, and unlabelled datasets come without labels. An example of labelled and unlabelled data can be seen in Figure 2-7.

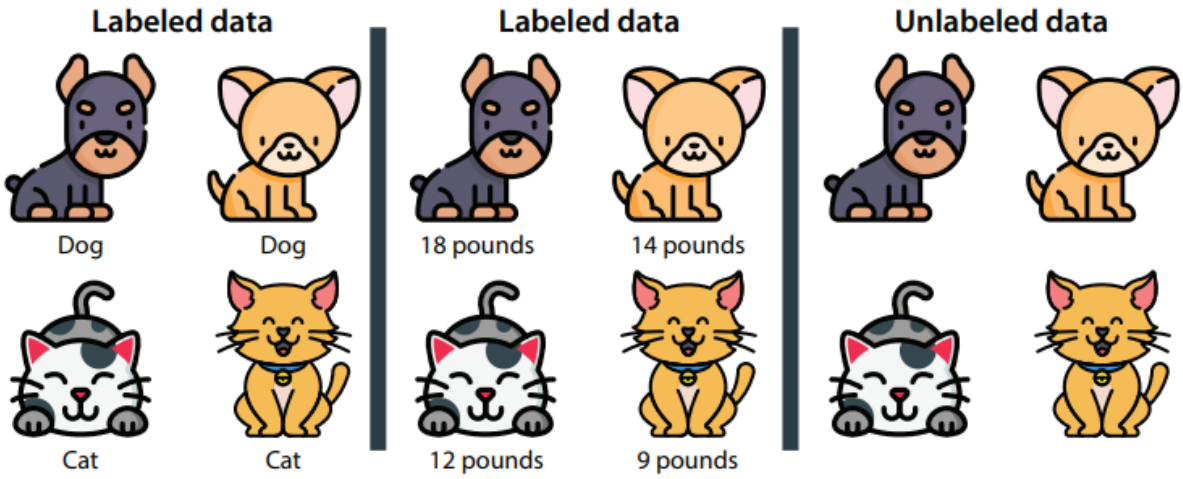

Figure 2-7: Examples of labelled and unlabelled data (Serrano, 2021)

This label or tag can appear in a specific type, as seen in the leftmost collection in Figure 2-7, or a number that denotes some characteristic of the data, as seen in the middle set of Figure 2-7. The rightmost set of Figure 2-7 represents unlabelled data.

Machine learning models are grouped into categories based on how they process and react to specific data. Figure 2-7 will be utilised to explain the three most utilised learning algorithms currently in circulation. These algorithms include

- Supervised learning,
- Unsupervised learning, and
- Reinforcement learning.

The different algorithms are explained in the following three subsections.

Supervised learning

The most commonly utilised machine learning paradigm is supervised learning (Serrano, 2021). In their most basic form, supervised learning algorithms function by constructing predictive models when presented with a large set of labelled training data. In most instances of supervised learning applications, the goal of the supervised learning algorithm is to predict the value of labels.

As an example, consider Figure 2-7. The dataset on the leftmost side contains images of cats and dogs, with accompanying labels stating either "Cat" or "Dog". For this dataset, a supervised learning algorithm would utilise previous data to predict the label of newer data points. If the algorithm is presented with a new image of a dog or a cat, the model will predict whether the provided image is a dog or cat, effectively labelling the latest data. Supervised learning can be found in the most common application areas, such as recommendation systems, text processing and image recognition. A more detailed dissection of supervised learning is provided in Chapter 3, as it relates mainly to the subject of the automatic setup of autoencoder neural network architectures.

Unsupervised learning

Unsupervised learning is considered another vastly utilised learning paradigm (Serrano, 2021). However, this learning paradigm differs from supervised learning in that it uses unlabelled data instead of labelled data. Therefore, unsupervised learning algorithms aim to extract as much utilisable information from a provided, unlabelled dataset as possible.

Fewer application areas of unlabelled datasets exist; however, a relatively small amount of information can still be extracted from unlabelled datasets. By utilising the rightmost image set from Figure 2-7 as an example, it is clear that this dataset contains images of dogs and cats and

is not accompanied by any labels. As this dataset does not have any labels, an unsupervised learning algorithm would not be able to discern precisely whether or not an image is of a cat or a dog, but would be able to determine whether or not two images are similar or different from one another. Unsupervised learning is mainly utilised in market segmentation, genetics, medical imaging and video recommendations.

Reinforcement learning

Reinforcement learning is yet another type of machine learning algorithm. Reinforcement learning algorithms are not presented with any datasets, but are instead expected to perform a task (Serrano, 2021). Within most reinforcement learning applications, the model is placed inside an environment and provided with an agent to interact with said environment. The agent is assigned a goal or a set of goals that it needs to complete within the environment. This environment is then populated with rewards (usually an increase in a score metric) and punishments (usually a decrease in a score metric) which help the agent make the correct decisions and take the right actions. Reinforcement learning is currently being utilised in several cutting-edge technological advances, the most prominent being within robotics, self-driving vehicles and games. The following section contains information regarding some of the different types of neural network and how and when they are utilised.

2.3.3 Types of neural network

As seen in Figure 2-4, there are many different types and architectures of neural networks. With numerous structures, neural networks are typically classified into more significant categories, with underlying subcategories depending on specific characteristics. These characteristics include the number of layers, neurons, structure, data flow, and density.

As the variety of neural networks is vast, only the most utilised neural network types in their basic form will be explored in this section. These neural networks include perceptrons, feedforward neural networks, convolutional neural networks, and recurrent neural networks. Finally, autoencoders will be discussed in greater detail, as they are the primary type utilised in this study.

Perceptrons

Rosenblatt (1958) initially proposed the perceptron model, and even today, it still stands as a fundamental and iconic concept relating to artificial intelligence (Islam, 2023). Perceptrons are considered the most commonly used type of neural network (Almeida, 2020). Perceptrons utilise weighted inputs, to which they then apply the activation function to obtain an output value as the final result. An example of this may be seen in Equation 2-1:

$$f(x) = \{^{1,\ if\ \boldsymbol{W}.x+b>0}_{0,\ otherwise}\}f. \tag{2-1}$$

Within Equation 2-1, $f(x)$ represents the activation function, $X = \{x_1, x_2\}$ are the inputs, $\boldsymbol{W} = \{w_1, w_2\}$ are the weights and b is a bias value. An example of the basic architecture of a perceptron can be seen in Figure 2-8.

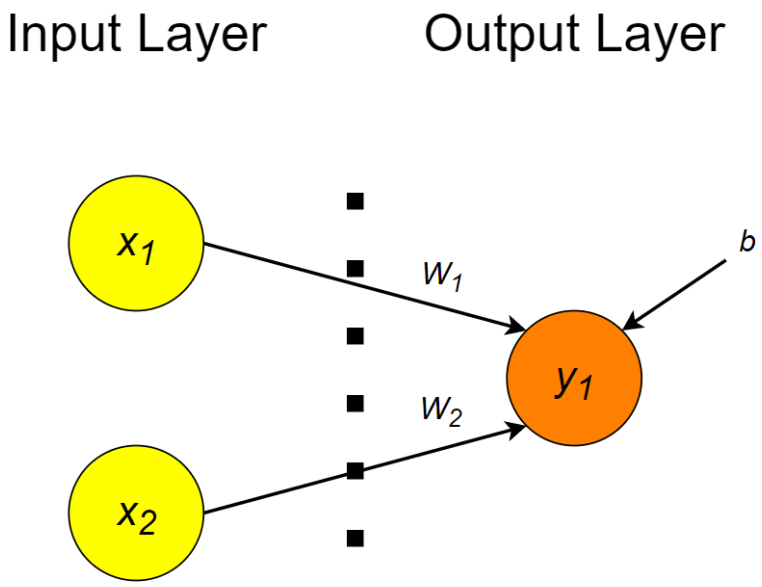

Figure 2-8: A basic perceptron neural network

The perceptron consists of an input layer and an output layer. The input layer contains two input neurons x_1 and x_2. The values of these neurons are affected by the weights w_1 and w_2, respectively. Additionally, a bias value b may be added to the output layer in order to increase the accuracy of the perceptron. The output layer consists of a single output neuron y_1 which then produces the output of the model.

Supervised learning algorithms typically train perceptrons to classify data into two categories. As such, perceptrons are also often referred to as binary classifiers. It should, however, be noted that perceptrons are limited because they can only be applied to linearly separable problems (Mendez Lucero *et al.*, 2022). As a result, perceptrons fail when presented with more complex problems, such as nonlinear scenarios, like a Boolean *XOR* problem (Islam, 2023).

Feedforward neural networks

Feedforward neural networks are similar to perceptrons in that they transform patterns from input to output data (Leijnen & Veen, 2020). Feedforward neural networks are considered one of the simplest forms of neural networks utilised today. Every neuron from a single layer of the network is connected with every neuron of the next layer. To be considered a feedforward neural network, the minimal components comprising a neural network are two input neurons connected to a single output neuron (similar to the base form perceptron as seen in Figure 2-8). Theoretically, where a feedforward neural network is created with an infinite number of neurons contained within a single,

nonlinear hidden layer, any conceivable relation between input data and output data or patterns can be studied and learned, given enough time (Leijnen & Veen, 2020). This property of neural networks is often referred to as universal approximation.

As with most other neural network architectures, feedforward neural networks may have varying densities of components, such as neurons and layers. The following is a description of typical component layouts found within feedforward neural network architectures.

Single input neurons

To effectively explain the components and computational process of single input neurons, Figure 2-9 is presented.

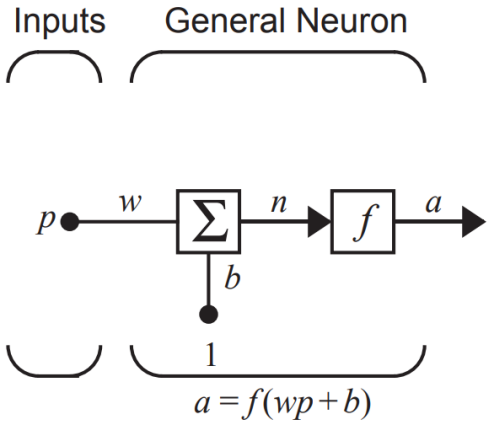

Figure 2-9: A basic, single-input feedforward neural network (Hagan *et al.*, 2014)

Within single input neurons, a scalar input p is typically multiplied by a scalar weight w to create a value wp (Hagan *et al.*, 2014). This value is then sent to a summation function. With other inputs, such as those deemed necessary by the neural network designer, the value 1.0 is additionally multiplied by a bias value b and also sent to the summation function. The summation function then produces a net input value, n, which is sent to a transfer (activation) function f which in turn produces the scalar neuron output a. The mathematical representation of the calculation of the neuron output within Figure 2-9 is shown in Equation 2-2 as

$$a = f(wp + b). \tag{2-2}$$

As an example, the following values are assigned to the following components: $w = 3$, $p = 2$, and $b = -1.5$. This results in the following calculation:

$$a = f(3(2) - 1.5) = f(4.5).$$

Nodes within feedforward neural networks typically represent numbers within an algorithm (Guilhoto, 2018). It should be noted that the variables w and b are adjustable scalar parameters

of the neuron, and the designer of said neuron chooses the transfer function utilised within a neuron. The parameters w and b may also be adjusted by some learning rule so that the neuron input/output relationship meets some specific goal as set by the designer of the neuron.

Multiple input neurons

In most implementations, a neuron will have more than one input. An example of a neuron with R inputs can be seen in Figure 2-10.

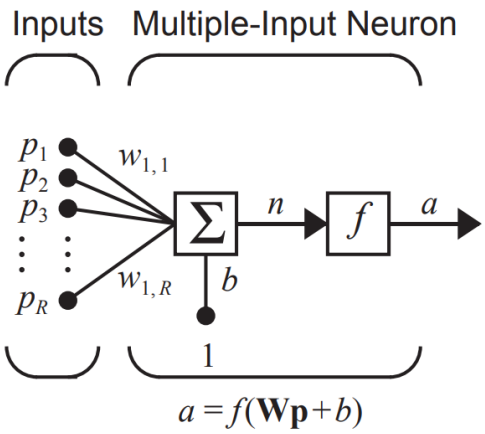

Figure 2-10: Neuron with multiple inputs (Hagan *et al.*, 2014)

In the neuron represented in Figure 2-10, there is a set of individual inputs, namely $p_1, p_2, \dots, p_R$ which are each weighted by their corresponding weight value, $w_{1,1}, w_{1,2}, \dots, w_{1,R}$. These weights are collectively represented by a weight matrix $\boldsymbol{W}$. A weight matrix is most commonly defined as shown in (2-3):

$$\boldsymbol{W} = \begin{bmatrix} w_{1,1} & w_{2,1} & \dots & w_{R,1} \\ w_{1,2} & w_{2,2} & \cdots & w_{R,2} \\ \vdots & \vdots & \ddots & \vdots \\ w_{S,1} & w_{S,2} & \cdots & w_{s,R} \end{bmatrix}. \tag{2-3}$$

Like the single input neuron, the multiple input neuron consists of a bias value b, which is summed with the weighted inputs to form the net input n. Equation 2-4 represents this process as follows:

$$n = w_{1,1}p_1 + w_{1,2}p_2 + \cdots + w_{1,R}p_R + b. \tag{2-4}$$

The above equation may also be written in matrix form, as shown in Equation 2-5:

$$n = \boldsymbol{W}\boldsymbol{p} + b. \tag{2-5}$$

Neural networks are often described in terms of matrices where the matrix $\boldsymbol{W}$ for the single neuron example only consists of a single row. With this, the neuron output for the multiple-neuron model can be represented as shown in Equation 2-4.

Layered neurons

In most neural network-based problems, a neural network consisting of a single neuron, even if said neuron contains many inputs, may not be sufficient to solve a problem. As such, neurons are utilised in collections that operate in parallel with one another, often referred to as a layer. A single-layer network that consists of S neurons is shown in Figure 2-11.

In the network represented in Figure 2-11, each of the R inputs is connected to each of the neurons. Additionally, the applicable weight matrix consists of S rows. The layer of S neurons includes the weight matrix, the summation functions, the bias scalars $b_1, b_{2,} \dots, b_S$, the transfer functions and the output scalars $a_1, a_{2,} \dots, a_S$. The inputs are considered a layer in themselves.

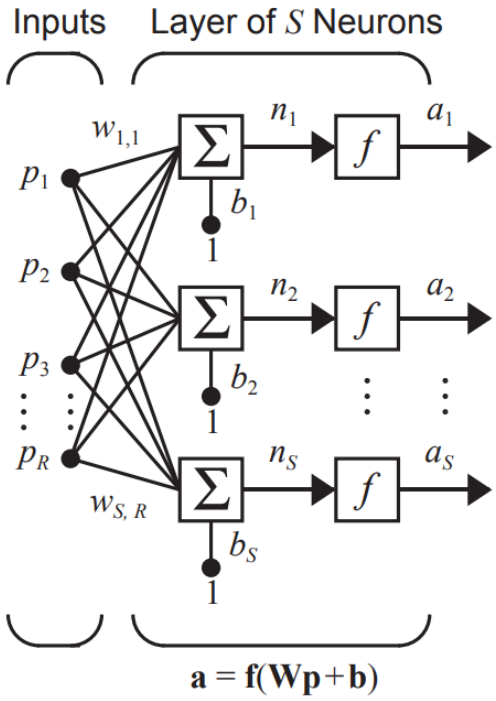

Figure 2-11: Single-layer network (Hagan *et al.*, 2014)

Every element of the input layer, $p_1, p_2, \dots, p_R$ is connected with each neuron through the weight matrix $\boldsymbol{W}$, with every neuron consisting of a bias value b_i, a summation function, a transfer function f and an output value a_i. Combined, the output values form an output vector $\boldsymbol{a}$. Additionally, it is common practice within neural network implementations that the number of input neurons is different from the number of neurons within subsequent layers, thus meaning that $R \neq S$.

Multiple layers of neurons

Neural network architectures additionally allow for the utilisation of multiple layered neuron layers. To better explain this concept, Figure 2-12 is presented. A layer that produces a network's final output is called the output layer. Additional layers, excluding the input layer, are called hidden layers. The neural network architecture in Figure 2-12 consists of an input layer, two hidden layers (layers 1 and 2), and an output layer (layer 3). Each hidden layer, as well as the output layer, consists of its weight matrix $\boldsymbol{W}$, its bias vector $\boldsymbol{b}$, a net input vector $\boldsymbol{n}$ and an output vector $\boldsymbol{a}$. The presented neural architecture contains R inputs, S^1 neurons in the first layer, S^2 neurons in the second layer and S^3 neurons in the third layer. The number of neurons within different layers can differ from one another.

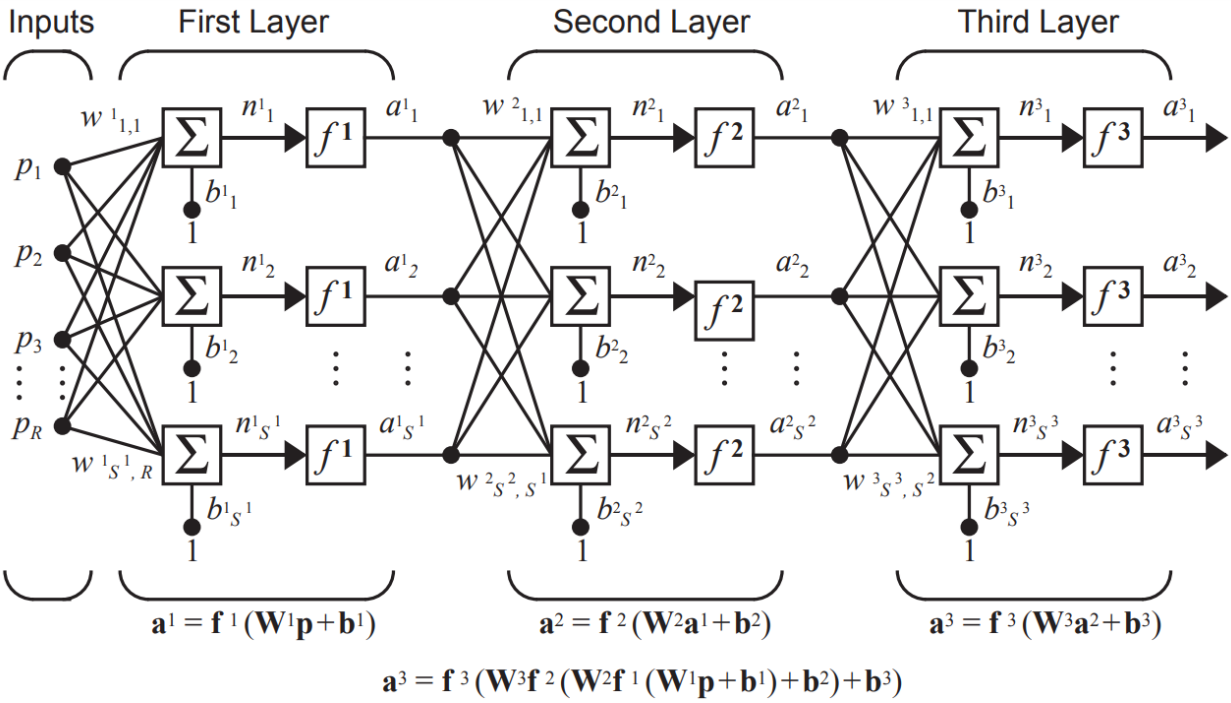

Figure 2-12: Architecture with multiple layered neuron layers (Hagan *et al.*, 2014)

The output values produced by the first layer are utilised as the input values for the second layer, the second layer output as the third layer input and so on for as many layers that the neural network comprises. Thus, layer two may be viewed as a single-layer neuron that consists of $R = S^1$ inputs, $S = S^2$ neurons and a weight matrix $\boldsymbol{W}^2$ with dimensions $S^2 \times S^1$. The input layer to layer two is denoted as $\boldsymbol{a}^1$ and the output as $\boldsymbol{a}^2$.

Multilayer neural networks are typically more powerful than single-layer networks. For example, a neural network comprising two layers that utilise the sigmoid function within the first layer and a linear second layer can be trained to approximate most functions arbitrarily well. In contrast, a single-layer neural network would be unable to do this.

Basic feedforward neural networks are advantageous in that they are typically less complicated and easy to maintain compared to other more resource-intensive neural networks, such as convolutional neural networks. Additionally, feedforward neural networks excel at tasks that are presented with noisy data. Some of the more popular application areas of feedforward neural networks are feature recognition, computer vision and speech recognition.

Convolutional neural networks

Convolutional neural networks (CNNs) differ primarily from typical feedforward neural networks in that they consist of pooling layers utilised for approximate scanning of patterns that are usually correlated according to space (Leijnen & Veen, 2020). CNNs are most commonly used in image processing applications; however, they can also be applied to other data modalities. An example of a typical CNN layout can be seen in Figure 2-13. The example neural network consists of a single input layer, a single feature mapping layer and two convolution layers, each followed by its respective pooling layer.

CNNs consist of three primary arrangements of neurons instead of the standard two-layer paradigm followed in less complex neural network architectures (Cong & Zhou, 2023). The initial layer is referred to as the convolutional layer. Within this layer, neurons only process information from a small part of the data fed into the neural network. Input features are then determined in different batches, which allows the networks to process the data in parts. In addition, this allows the network to compute the data over multiple iterations to process the provided input data fully.

The second layer is referred to as the pooling layer. Pooling layers are typically utilised to reduce the dimensionality of the feature maps that the convolutional layers had created. After the pooling layer has concluded the dimensionality reduction operations, the altered data is typically sent to additional iterations of convolution and pooling layers as the need arises.

The third layer is referred to as the fully connected layer. The fully connected layer is typically utilised to classify the data processed within the different convolutional and pooling layers. This data is then presented to the output layer nodes after a bias value is added to the data. This layer is similar to the hidden layers found in other neural network architectures. The nodes found within this layer are typically interconnected with every node of the next layer.

CNNs are hierarchical structures that attempt to solve a supervised learning application by passing an input signal x through a series of simple non-linearities and convolutional operations. Starting with an input signal x every subsequent layer x_n is computed as shown in Equation 2-6:

$$x_n = pC_n x_{n-1}, \tag{2-6}$$

where, C_n is a linear operator, and p is a non-linearity. Typically, within a CNN, C_n represents a convolution and p is a pooling layer. Additionally, it is recommended to consider the operator C_n as a stack of convolutional filters, the layers are filter maps, and each layer can be written as a sum of the convolutions of the previous layer.

Some of the more prominent advantages of CNNs are that they can be used for tasks that do not require many parameters, as they require fewer parameters to learn compared to other neural network models. However, this comes with a stark disadvantage: CNNs take comparatively more time to train than different architectures and are relatively slower in terms of operating speed, as the applications of CNNs are typically more complex in structure and contain various layers of convolution, depending on the number of hidden layers and data presented to the neural network.

Recurrent neural networks

The most straightforward manner of comprehending the structure of recurrent neural networks is to imagine them as feedforward networks with additional connections within layers (Leijnen & Veen, 2020). As a result of these numerous intricate connections, recurrent neural networks are not stateless, and the order and timing in which different inputs are structured contribute to the functionality of the neural network. This implies that recurrent neural networks can find structure over time. Recurrent neural networks are designed with the intent to interpret sequential or temporal information. To do this, data points that appear within a sequence are utilised to produce more accurate predictions. An example of a basic recurrent neural network structure can be seen in Figure 2-14.

The neural network presented in Figure 2-14 consists of three input variables, such as words in a sentence. Additionally, the neural network contains six hidden nodes. These nodes may consist of various hidden states, $h(t)$, that act as the memory of the neural network, where t represents the time step. The hidden states would calculate the value of the hidden node, based on the previous state of said node. Additionally, recurrent neural networks also consist of hidden connections parameterised by a weight matrix $\boldsymbol{U}$, hidden-to-hidden recurrent connections parameterised by a weight matrix $\mathbf{W}$, and hidden-to-output connections parameterised by a weight matrix $\boldsymbol{V}$. The variable b represents the bias term, which adjusts the output of the network and acts as an offset, enabling the model to better accommodate data that is not perfectly centred. The functionality of a recurrent neural network is expressed in Equation 2-7:

$$h(t) = f\big(\boldsymbol{U}x(t) + \boldsymbol{W}h(t-1)\big)(\mathbf{V} + \mathrm{b}). \tag{2-7}$$

Recurrent neural networks are often utilised in areas where sequential data is plentiful (Serrano, 2021). These include text processing, text-to-speech implementations, image tagging implementations, sentiment analysis and translation services.

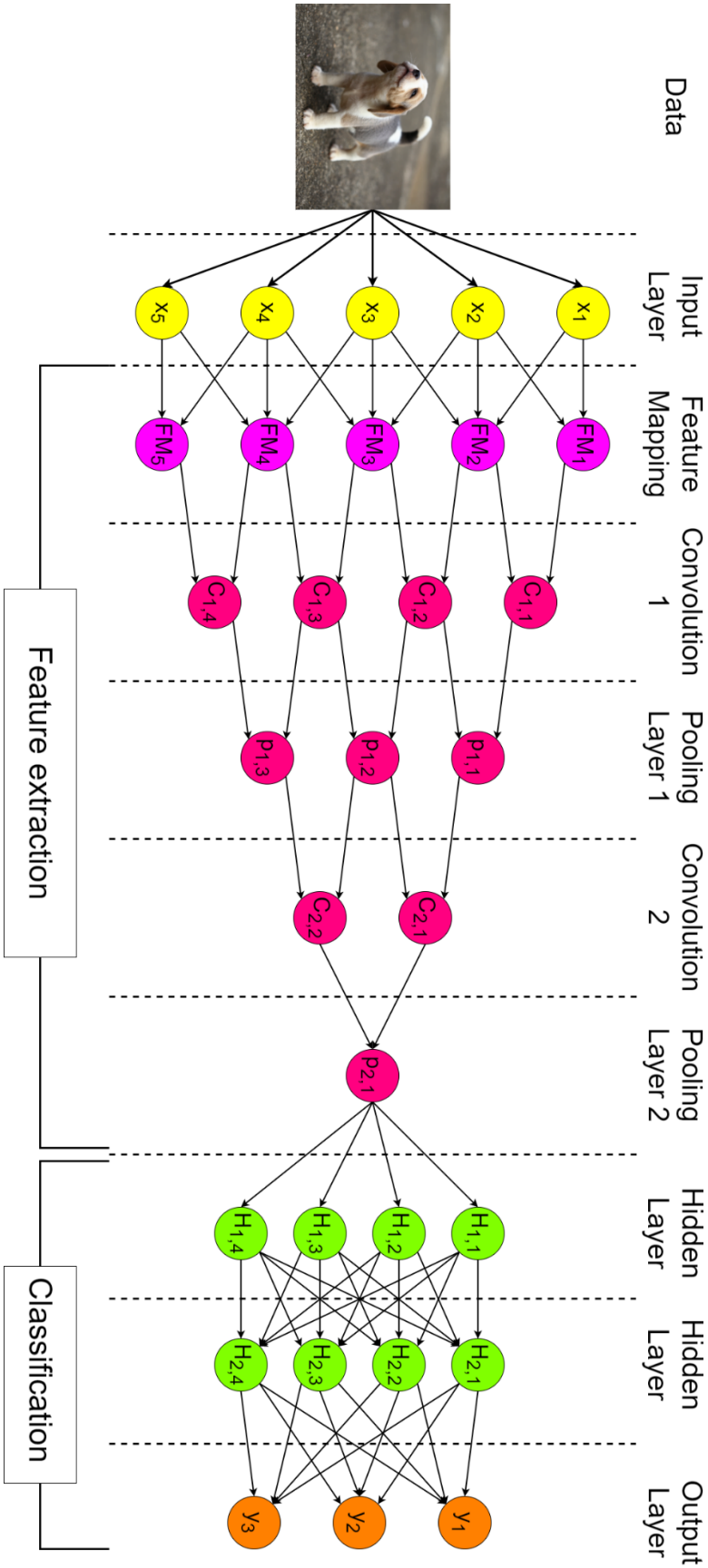

Figure 2-13: Convolutional neural network structure

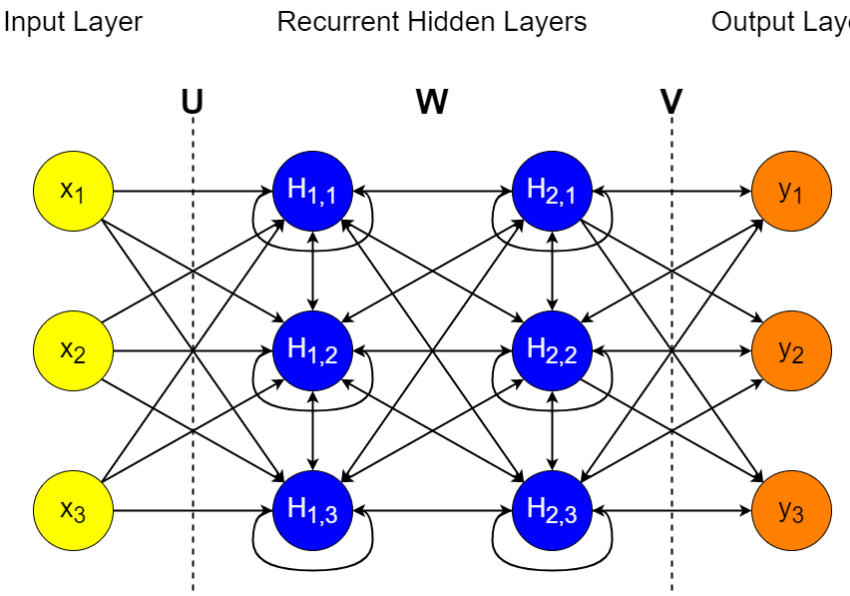

Figure 2-14: Basic structure of a recurrent neural network

The following section contains an overview of the autoencoder neural network architecture. As autoencoders form the basis of the study, a more in-depth explanation of the structure of autoencoders is presented in Chapter 3.

Autoencoders

Autoencoders are primarily utilised to compress or encode information and then reconstruct or decode this information by transforming it into smaller hidden layers of symmetrical surrounding layers (Leijnen & Veen, 2020). This allows researchers to utilise the resemblance of the input provided to the neural network and the output provided by the neural network to be compared to measure the autoencoder's accuracy. An example of a basic architecture of an autoencoder can be seen in Figure 2-15. A discussion of the application areas and goals of neural networks is presented in the following section.

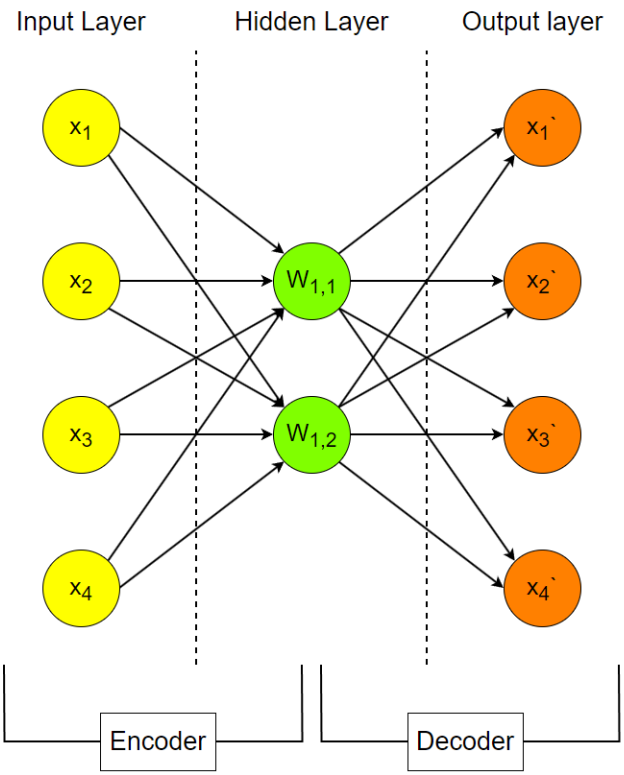

Figure 2-15: Basic structure of an autoencoder neural network

2.4 Application areas, goals and challenges faced by neural networks

From the beginning of time, humans have been building ever-evolving tools to comprehend and control the environment around us. In recent years, artificial intelligence and neural networks stand as cornerstones in the revolutionising of how machines may learn and interact with the aforementioned environment (Chapman, 2023). In the modern world, artificial intelligence is considered a flourishing field with a plethora of active research topics and numerous practical applications applicable to the field. Deep learning, considered a practical branch of artificial intelligence, has also significantly benefited from the sharp increase in resources related to artificial intelligence as more funding and research efforts for artificial intelligence applications occur, leading to a rise in the application of neural networks across various areas.

The utilisation of neural networks has gained popularity in areas with high demands for classification, clustering, pattern recognition and prediction. The effectivity of these neural networks is often measured in terms of data-based analytical factors, such as accuracy, convergence, fault tolerance, latency, performance, processing speed, scalability and volume (Mohd Amiruddin *et al.*, 2020; Serey *et al.*, 2023).

Neural networks have capabilities for high-speed processing within sizable parallel implementations (Bilski *et al.*, 2021; Liu *et al.*, 2024). This has also led to a considerable increase in the need for research and development in the field of neural networks, specifically in

numerically based problems, with the main focus being placed on the utilisation of properties, such as adaptivity, self-learning, non-linearity, fault tolerance and advancements in input to an output mapping (Wang *et al.*, 2022).

It is due to these data analysis factors that neural networks can be successfully implemented in various areas. Some of the areas of application include computer vision, text processing, action recognition, medicine, security, natural language processing, transportation, engineering, recommender systems, reinforcement learning, and robotics, as well as a plethora of other application areas. (de Santana Correia & Colombini, 2022; Moayedi *et al.*, 2020; Naranjo-Torres *et al.*, 2020; Zacarias-Morales *et al.*, 2021).

As the modern world steadily transitions into a more digital era, the applicable areas of neural network technologies are more than likely to increase. However, despite the numerous application areas of neural networks, an increasing need to adopt a systematic approach to developing neural networks is becoming all the more present as common key challenges and issues arise (Abiodun *et al.*, 2018).

These challenges include, but are not limited to, the improvement of the design of robust neural network models, the advancement of transparency within neural network models and the access to knowledge extracted from trained neural networks, the improvement of the extrapolation ability of neural networks, more effective approaches to uncertainty, data behaviour analysis problems, and problems regarding supervised classification.

Despite these generally experienced challenges, new technology and techniques are implemented to rise to these challenges as research is conducted within artificial intelligence and neural networks. This correlates with the objective of this study, which is to provide a systematic approach to the automated construction of autoencoders, which could result in a reduction of the severity of complications faced concerning the development of autoencoder neural networks by not only lessening the probability for possible human error, but also having the possible benefit of reducing the developmental lifecycle of new autoencoder neural network architectures. Although sentient machines might still be out of reach, neural networks with the capabilities of improving people's lives are implementable today.

The training of different neural network architectures is crucial for the successful implementation of these models. Therefore, the various training algorithms regarding neural networks are critical to this study. The following section contains information regarding practical approaches utilised within neural network training.

2.5 Neural network training

It has been widely recognised that the efficient training of a neural network is crucial to the performance of that neural network (Yang *et al.*, 2021). In general terms, neural network training consists of determining appropriate weights for the different connections within a neural network (Gurney, 2018). Over the past decade, multiple advancements have been made that improve machine learning and training large, deep neural network architectures. Within this section, an overview of the training process of a relatively basic neural network is presented. In Chapter 3, a more comprehensive overview of the training regime of the autoencoder neural network is discussed.

As stated, neural network models are grouped into categories, based on how they process and react to specific data; however, as the study focuses on autoencoders and the automated design of autoencoder architectures, the methodologies behind supervised learning will be described in the following section.

2.5.1 Training of supervised learning models

Training and optimising supervised learning models are approachable through different means. In terms of training, backpropagation is considered the algorithm most often used in training neural networks (Lillicrap *et al.*, 2020), while first-order optimisation algorithms are a collection of algorithms that utilise gradients to optimise a learning model, specifically, the stochastic gradient descent algorithm (Goodfellow *et al.*, 2016). The following is a summary of these two algorithms.

Backpropagation

Within most instances of a neural network, the network is utilised to accept an input x and produce an output y by using the input. The input variables provide the preliminary information passed on to the hidden nodes at each layer, producing the outputs. This concept is referred to as forward propagation (Goodfellow *et al.*, 2016). The base concept behind backpropagation is the adjustment of the different weights found within a neural network, based on the error rate observed within the previous epoch (Rumelhart *et al.*, 1985). An epoch is a terminology that describes when an entire dataset passes forward and backwards through a neural network only once. To explain the processes that take place within backpropagation, Figure 2-16 is referenced.

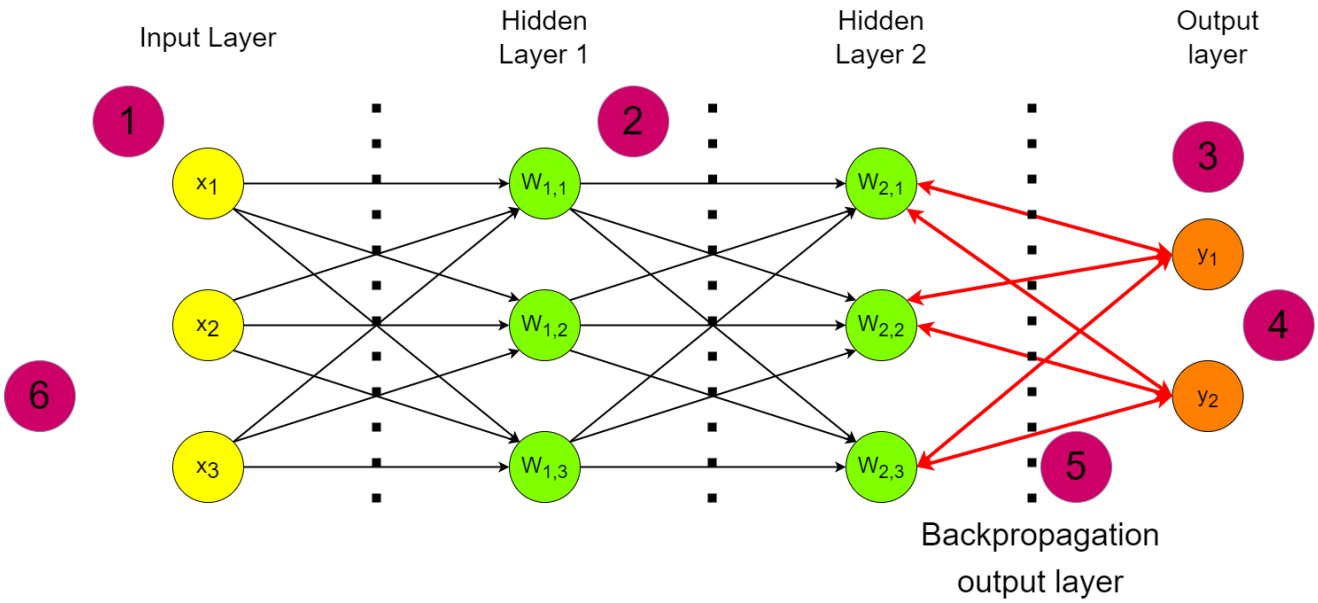

Figure 2-16: Backpropagation steps

Backpropagation is performed in the following steps:

1. Initially, input values are received by the input layer neurons.
2. Inputs are modelled by utilising the weighted values of the hidden layers.
3. The output from each of the neurons contained within the input layer, hidden layer and output layer is calculated and stored within the neurons within the output layer.
4. The errors of the outputs are calculated by utilising the calculation shown in Equation 2-8:

$$Error = Actual\ output - Desired\ output. \quad (2\text{-}8)$$

5. The information stored within the neurons of the output layer is compared with the calculated error, after which the weights of the hidden layers are adjusted to reflect the desired output better.
6. The process is repeated until a desirable output error is achieved.

Backpropagation forms one of the most utilised supervised learning training algorithms today. Often used with backpropagation and present in most deep learning applications is the stochastic gradient descent algorithm, which forms part of the iterative first-order optimisation class of optimisation (Goodfellow *et al.*, 2016).

Iterative first-order optimisation

In terms of neural network training, an iterative process is where new sensitivity values are determined, and the network is continuously re-tuned until a satisfactory objective value is found (Zhang *et al.*, 2023). Optimisation algorithms that only utilise the gradient, such as gradient

descent, are first-order optimisation algorithms (Goodfellow *et al.*, 2016). To help explain the process of iterative first-order optimisation, consider Figure 2-17.

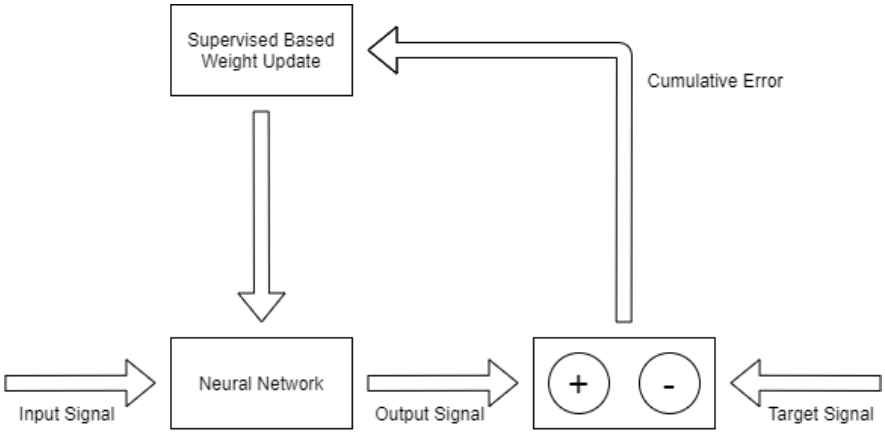

Figure 2-17: Supervised learning training diagram (Dike *et al.*, 2018)

Within this supervised learning model training algorithm, the trainer utilises a set of feedback stimuli for which the output is already known (Dike *et al.*, 2018). For a period of the initial training process, the desired output is constantly related to the provided output signal as input signals are fed into the neural network. Afterwards, the actual signal, the output signal (denoted by +) and the target signal (represented by -) are compared.

After some iterations of training have concluded, the trainer utilises an iterative first-order optimisation algorithm, referred to as the slope descent rule, that uses the error among the actual output and the target information to calculate the nearest match between the target signal (-) and the actual out signal (+). This value, called the cumulative error, is then used to update the neural network weights.

As an example, consider a scenario where a neural network needed to be trained to discern whether or not an image is of a bird or a cat. First, the algorithm is trained on multitudes of pictures of birds and cats. The algorithm is also fed images of rats and cats to supervise the algorithm to capture the classification of images precisely. After the algorithm has adapted to the pictures and is capable of categorising them, it can then be used on new datasets and be able to discern between images of cats and birds within the unseen images. The following section contains an explanation of the concept of activation functions utilised within neural networks.

2.5.2 Activation functions

The term activation function has not always been the industry standard terminology associated with the meaning it has today, with earlier literature referring to activation functions as transfer or output functions (Apicella *et al.*, 2021). Activation functions have the capability of modelling linearly or non-linearly. For linear implementations, a linear mapping of an input function to output

is performed within the hidden layers of the neural network whereafter the final prediction for each applicable value is calculated. The training of neural networks is additionally heavily affected by the activation functions used. These functions also allow neural network models to learn complex representations by providing the neural network with the concept of non-linearity. Activation functions are applied to the linearly transformed input data provided to a neural network for non-linear implementations.

An activation function determines whether or not a neuron produces enough value to be considered activated (Montesinos López *et al.*, 2022). If a value is presented to an activation function that does not meet the requirements set by the activation function, then the activation function does not fire. Multiple different activation functions have adverse effects on the success of a neural network (Misra, 2019).

In earlier literature, two different activation functions, *Sigmoid* and *TanH*, were widely used and are still regarded as popular choices today (Dubey *et al.*, 2022). The *Sigmoid* function is typically mathematically represented as shown in Equation 2-9:

$$\frac{1}{e^{x}+e^{-x}}. \tag{2-9}$$

The *TanH* function is typically mathematically represented as shown in Equation 2-10:

$$\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}. \tag{2-10}$$

In the above equations, the value of x typically represents the input to a neuron. When plotted on a graph, the *TanH* function creates an S-shaped curve between -1 and 1, whilst the *Sigmoid* function varies in the range of (0,1). A graphical representation of both functions is shown in Figure 2-18 and Figure 2-19.

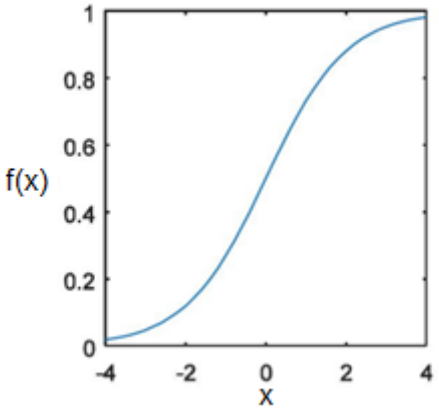

Figure 2-18: A graphical representation of the *Sigmoid* function (Gomar *et al.*, 2016)

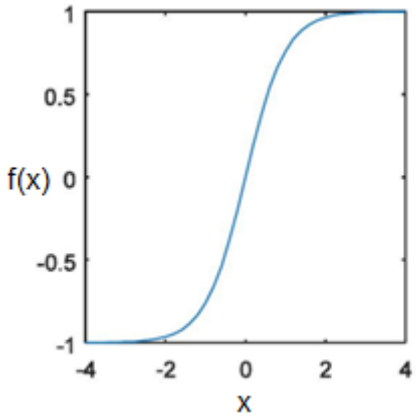

Figure 2-19: A graphical representation of the *TanH* function (Gomar *et al.*, 2016)

Multiple activation functions have been developed, each with its pros and cons (Szandała, 2021). In contrast to the two main probability-inspired saturated activation functions, namely the *Sigmoid* and the *TanH* functions, a less probability-inspired, unsaturated linear activation function known as the *Rectified Linear Unit* (*ReLU*) function, has also become widely utilised (Abdel-Nabi *et al.*, 2023). The *ReLU* implementation also showed better generalisation and improved coverage than *Sigmoid* and *TanH*. The mathematical representation of the *ReLU* implementation is shown in Equation 2-11:

$$f(x) = \max(0, x). \tag{2-11}$$

Applying the *ReLU* function to the output of a linear transformation yields a non-linear transformation. However, the function remains very close to linear because it is considered a piecewise linear function with two linear pieces. Figure 2-20 is a graphical representation of the *ReLU* activation function.

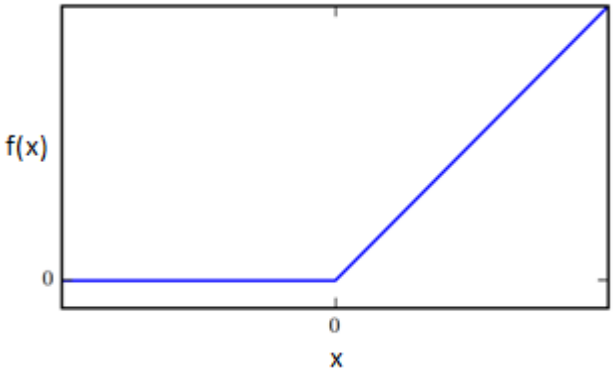

Figure 2-20: A rectified linear activation function

The *ReLU* activation function has grown so popular that it is considered the default activation function used across the deep learning community (Zheng *et al.*, 2020). Within the following section, a summary of the chapter is provided.

2.6 Summary

Artificial intelligence has a rich history, albeit relatively short. Artificial intelligence development and implementations have gone through numerous fluctuations in the past, with the first wave of artificial intelligence, the age of cybernetics, occurring in the 1940s-1960s. After this, an artificial intelligence winter occurred, followed by the second wave of artificial intelligence, called the age of connectionism, between the 1980s and the 1990s. This period was followed by the second artificial intelligence winter and the most recent age of artificial intelligence, deep learning.

During the age of deep learning, neural networks mimicking the structure and functionality of the biological mind have seen an increase in development. These neural networks can be applied to numerous different fields for a vast number of utilisations. This is achievable due to the plethora of different architectures that neural networks may adopt, including, but not limited to, perceptrons, feedforward neural networks, convolutional neural networks, recurrent neural networks and autoencoders. These architectures can model labelled or unlabelled data, using supervised, unsupervised and reinforcement learning algorithms. In the next chapter, a more in-depth study regarding autoencoders is presented, as the concept of autoencoders forms one of the cornerstones of the study.

CHAPTER 3 AUTOENCODERS

3.1 Introduction

Deep learning has emerged as the leading AI technology for data processing, and the type of deep learning model employed depends on the proposed task, the nature of its requirements and the data structure. An autoencoder is one of the deep learning models with a strong affinity for tasks including feature extraction, dimensionality reduction, and anomaly detection. Although an autoencoder is an unsupervised model, it has been diversely employed to enhance some supervised models' performances (Chen *et al.*, 2023; Chen *et al.*, 2024).

The above research exemplifies the usefulness of autoencoders within an important field. In this study, an improvement in the developmental lifecycle of autoencoders may contribute to artificial intelligence by streamlining the generation of relevant neural network structures and yield benefits for additional fields that utilise autoencoders. In contrast to Chapter 2, which contained a broader view of artificial intelligence and neural networks, Chapter 3 presents a more focused overview concerning autoencoders.

The structure of the chapter is as follows. Initially, the history and general purpose of autoencoders are explored within Section 3.2, after which the general framework and components of the autoencoder neural network will be discussed in Section 3.3. Next, in Section 3.4, the different methods of achieving regularisation within autoencoders are addressed. Various applications and utilisations of autoencoders will be considered in Section 3.5. Finally, Section 3.6 concludes the chapter with a summary regarding its contents.

3.2 Background

The concept of autoencoders was initially introduced in the 1980s by Rumelhart *et al.* (1985). To address the problem of backpropagation without a teacher, the teacher uses the input data provided to a neural network. Autoencoders, combined with the Hebbian learning rules, provide one of the fundamental techniques for unsupervised learning (Baldi, 2012). The main purpose of autoencoders is to model an input signal in an unsupervised manner, which can be used for various applications, such as clustering.

Autoencoders, as do many deep learning applications, play an integral role in unsupervised learning. Autoencoders are considered learning circuits that transform input data into output data while achieving the least possible amount of distortion within the input data (Baldi, 2012). Autoencoders can be defined as follows: Autoencoders are a specific type of neural network primarily utilised to encode an input signal into a representation that is both compressed and

meaningful, which the neural network is then tasked to decode in such a manner that the reconstructed output is comparatively as similar to the original initial signal as possible (Rokach *et al.*, 2023). A graphical representation of this process is shown in Figure 3-1.

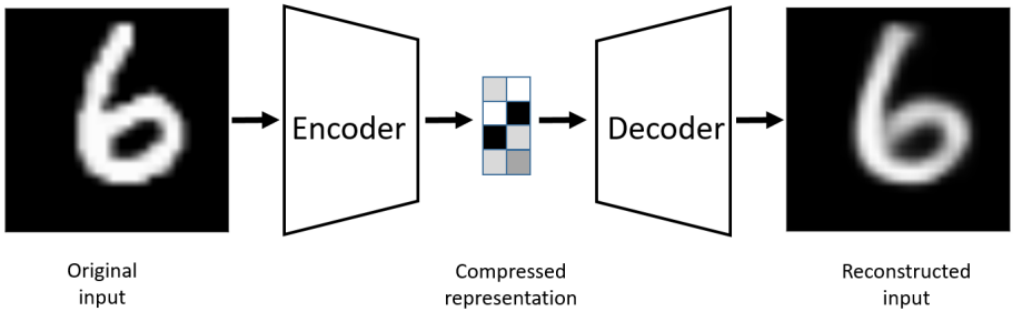

Figure 3-1: A graphical representation of the functionality of an autoencoder neural network

In the above figure, a hand-drawn image of the number 6 is presented to the autoencoder as input data. On a very high level, the input data is compressed by the encoder module of the neural network, finally becoming a compressed representation of the original data within the bottleneck layer of the autoencoder. From the bottleneck, the compressed representation of the input data is then reconstructed by utilising the decoder module of the neural network. The resulting image is a reconstructed version of the input data. This process is further described in greater detail in the following section.

3.3 A general autoencoder framework

A general framework of an autoencoder is presented in Figure 3-2. The autoencoder architecture consists of two components, namely an encoder module and a decoder module, which are mirrored architectures of one another (Dawani, 2020). The encoder and decoder components are connected, using a bottleneck layer (also referred to as a latent-space representation, code, or compression), which typically has considerably fewer dimensions than the encoder or decoder.

The encoder component takes in an input signal of a high dimension. Then, it reduces it to a lower dimensional representation by evaluating and learning the patterns within the signal. This process is represented by Equation 3-1:

$$z = f_\theta(x), \tag{3-1}$$

where f_θ represents the encoder module, x the input signal presented to the network and z the resulting compressed representation that had been learned within the bottleneck layer.

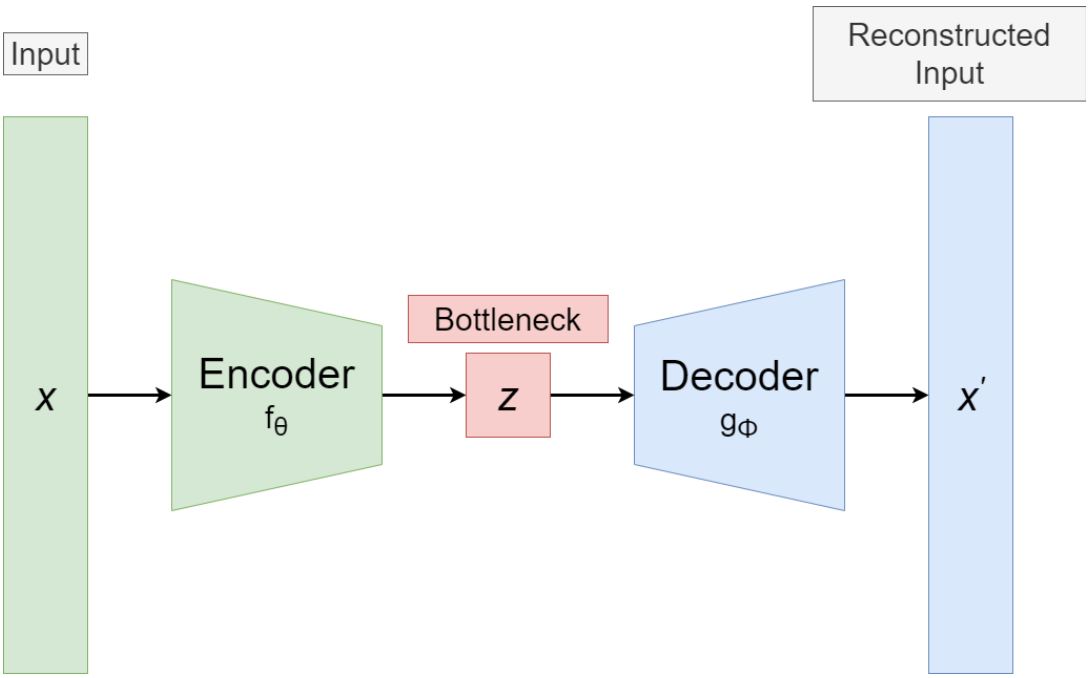

Figure 3-2: A basic autoencoder architecture

The resulting compressed representation contains all of the primary information of the original input signal, but within a lower dimensionality. The decoder module receives this lower dimensional representation and attempts to reconstruct the original input signal as accurately as possible. This process is represented as shown in Equation 3-2:

$$x' = g_{\Phi}(z), \tag{3-2}$$

where the decoder module g_{Φ} utilises the compressed representation z to create the reconstruction of the original input signal presented to the network, x'. Equations 3-1 and 3-2 can be combined to represent the autoencoder as follows:

$$x' = g_{\Phi}(f_{\theta}(x)). \tag{3-3}$$

The goal of the autoencoder is to recreate the original input signal as accurately as possible from the compressed representation, meaning: $x \approx x'$ (Dawani, 2020). Autoencoders consist of encoder and decoder components that mirror one another's structure. The weights of the encoder and decoder components do not need to be mirrored; however, they are trained together to output the reconstructed data as close to the original signal as possible.

Traditionally, autoencoders are symmetrical in structure; however, research has shown that in certain situations, autoencoders that consist of asymmetrical structures may result in comparable or even superior representation for classification tasks, as demonstrated in the research conducted by Sun *et al.* (2016). To determine the similarity between the original and the

reconstructed signals, the mean squared error (MSE) loss function is typically utilised (Rokach *et al.*, 2023). This may be mathematically represented in Equation 3-4:

$$L_{AE}(\theta, \Phi) = \frac{1}{n}\sum_{i=1}^{n}\left(x_i - g_{\Phi}\left(f_{\theta}(x_i)\right)\right)^2, \tag{3-4}$$

where x_i denotes the input, n represents the number of data points, Φ represents the decoded or predicted parameter of the applicable input value and θ represents the initial parameters presented to the encoder module of the autoencoder.

The smaller the MSE value, the closer the reconstructed output signal x' is to the original input signal x (Dawani, 2020). This subclass of the autoencoder is often called an under-complete autoencoder, as the bottleneck layer is considerably smaller than the dimensions utilised within the input and output layers. The function of the bottleneck is to map a high-dimensional space to a lower-dimensional space by learning a manifold. A manifold is a topological space resembling that of Euclidean space. This manifold may also be represented by utilising a vector field and visualising the applicable data clusters. The vector field is then utilised by autoencoders in order to learn to reconstruct inputs. Every data point could be located within this manifold and projected back into a space of a higher dimension in order to reconstruct it.

As an example, the MNIST dataset is utilised (Deng, 2012). The MNIST dataset of handwritten digits ranging from 0 to 9 is among the most popular for comparing learning techniques and training pattern recognition methods. The dataset consists of a training dataset of 60000 images and a test set of 10000 images. An example of this dataset can be seen in Figure 3-3.

The encoder component of an autoencoder receives this data as an input signal. It encodes it into a lower-dimensional latent bottleneck layer, which contains a compressed representation of this higher-dimensional input. This may typically be represented visually in a two-dimensional space, as seen in Figure 3-4.

In most autoencoder implementations, a bottleneck that is of a smaller density than that of the input of the encoder module is present within the autoencoder structure (Yong & Brintrup, 2022). Typically, autoencoders learn the identity function within most autoencoder applications if no constraints are placed to regulate the autoencoder. When this occurs, the autoencoder effectively reproduces any inputs provided to it regardless of the content of the input signal, whether it is valid or anomalous. To prevent this occurrence, imposing a bottleneck within the autoencoder architecture with a lower dimensionality of the input signal is common, resulting in an under-

complete autoencoder architecture. As such, the output of the encoder module should present a signal much lower than the dimension of the input signal. This implies that an autoencoder with a large bottleneck layer size may not effectively compress input information, and an autoencoder with a bottleneck layer size that is too small may not reproduce the inputs accurately.

Figure 3-3: A small subset of data contained within the MNIST dataset (Dawani, 2020)

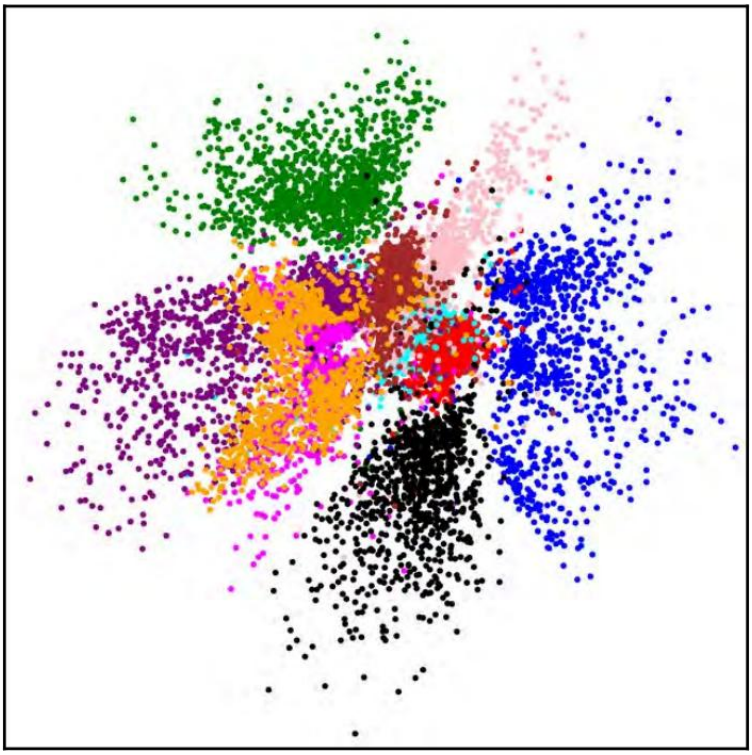

Figure 3-4: Two-dimensional space representation of the MNIST dataset (Dawani, 2020)

Figure 3-4 is an example of the capability of an autoencoder to map different features of entities contained within a dataset to a two-dimensional space. Within the figure, every colour represents a different digit of the MNIST dataset. The capabilities and effectiveness of an autoencoder may additionally be altered by implementing a commonly used practice, referred to as regularisation. In this study, regularisation will be utilised to make slight alterations to the learning algorithm of the neural network to promote better generalisation within the neural network. This process is expanded upon in the following section.

3.4 Regularisation in autoencoders

Autoencoders and neural networks, in general, are powerful tools capable of learning complex functions across various application areas and domains (Aggarwal, 2018). However, this potential for learning often leads to a significant challenge in neural network training: overfitting. Overfitting occurs when a neural network performs excellent prediction on the training dataset, but performs poorly on unseen test data. This happens because the learning process often memorises random artefacts of the training data that do not generalise well to the test data. In extreme cases, this is referred to as memorisation. Conversely, the ability of a network to provide useful predictions for instances it has not yet seen is known as generalisation.

Over the years, various strategies have been developed to help neural networks generalise better, collectively called regularisation (Moradi *et al.*, 2020). One standard method of regularisation involves modifying the dimensions of the representation by reducing them, creating a bottleneck structure within the autoencoder. This ensures regularisation and achieves a lower-dimensional representation of the input signal, which is commonly used in applications like feature extraction and data compression.

If the autoencoder is given too much modelling power, either through excessive training time or by having hidden layers larger than the input layer, it can still overfit or even memorise the input data. This results in the model effectively copying the input data across all hidden layers, thereby learning patterns that do not generalise beyond the training set (Aggarwal, 2018; Bourlard & Kabil, 2022).

The bias-variance trade-off is crucial in developing effective autoencoder architectures (Rokach *et al.*, 2023). Bias refers to the error introduced by approximating a complex problem with a simplified model. In contrast, variance refers to how sensitive a model's predictions are to fluctuations in the training data. As the model becomes more complex, its variance increases, and its bias decreases. The bias-variance trade-off aims to build a model that is neither too complex nor too general, achieving an acceptable balance between bias and variance. This forms one of the objectives of the automated autoencoder construction algorithm presented in this study.

Three of the most popular autoencoder implementations used to achieve a desirable balance in this trade-off will be explored in this section.

3.4.1 Sparse autoencoders

One method utilised to achieve an acceptable trade-off balance between variance and bias applies a sparsity constraint to a neural network. Sparse autoencoders are characterised by a limited number of neural nodes that are simultaneously active (Berahmand *et al.*, 2024). The decreased number of active nodes is due to sparse autoencoders attempting to model sparse representations of various input data by utilising a sparsity constraint within their loss function. The objective of sparse autoencoders is to minimise the disparity between the reconstructed data produced by the autoencoder and the original input data, all while following the constraints placed on the sparsity of the latent representation.

The constraints are particularly advantageous in scenarios where the number of hidden units exceeds the number of input units, as they enable the discovery of meaningful structures within the data while preventing the network from simply learning the identity function (Chen & Guo, 2023). This can be achieved through various methodologies, including L1 regularisation and Kullback-Leibler (KL) divergence, both of which introduce a sparsity penalty to the loss function.

L1 regularisation typically functions by adding a sparsity penalty $\Omega(h)$ to the reconstruction error. This can be represented as seen in Equation 3-5:

$$L\left(x, g\left(f(x)\right)\right) + \Omega(h). \tag{3-5}$$

Within the above equation, L represents the primary loss function, whereas h represents the hidden layer activations or parameters that are to be regularised. KL-divergence functions by measuring the difference, often denoted by the operator || between two probabilistic distributions, typically denoted by the variables P and Q, and constrains the average activation of a neuron over a collection of samples. This is shown in Equation 3-6:

$$KL(P||Q). \tag{3-6}$$

3.4.2 Denoising autoencoders

Denoising autoencoders are able to be utilised either as regularisation methods or as standalone autoencoders that excel at error correction. In contrast to sparse autoencoders, which add a penalty to the cost function of an autoencoder, denoising autoencoders affect the cost function by altering the connected reconstruction error term. Within traditional autoencoders, the neural network attempts to minimise a function (3-7)

$$L\left(x, g\left(f(x)\right)\right), \tag{3-7}$$

where L is a loss function value which denotes the difference between x and $g(f(x))$. This leads the autoencoder to learn to merely act as an identification function if the autoencoder can act as such. Conversely, a denoising autoencoder attempts to minimise a function (3-8)

$$L(x, g(f(x'))) \tag{3-8}$$

where x' is a copy of the input signal x that had been purposefully corrupted by distortion or noise. This leads the autoencoder to learn not only to reconstruct the input, but also to undo the corruption of the signal. An example of this is shown in Figure 3-5. A disrupted input image is encoded within the figure into the representation, which the neural network would then decode.

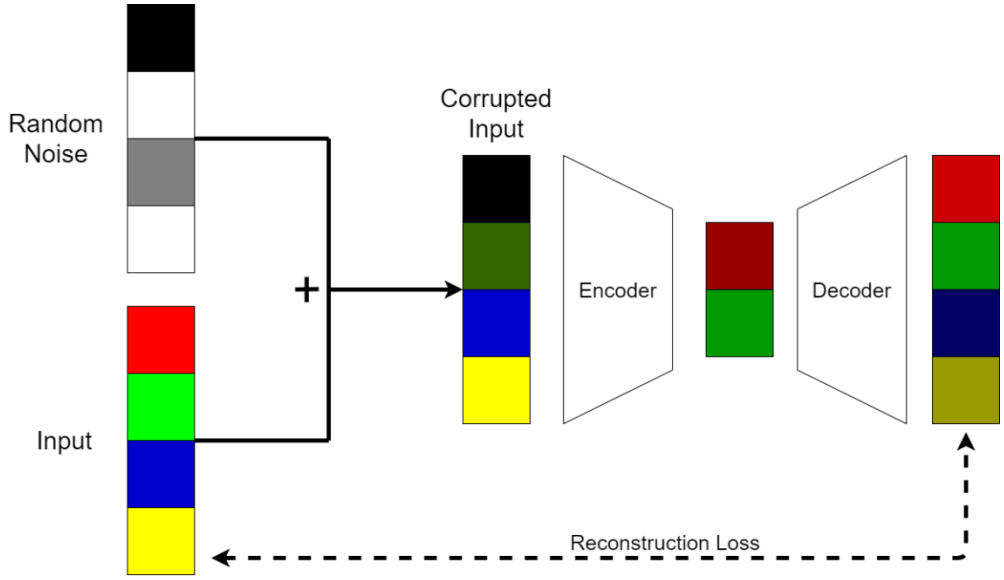

Figure 3-5: A denoising autoencoder example

Denoising autoencoders can remove corruption from a signal, which presents another example of how to achieve a suitable bias-variance trade-off. Another method used for achieving such a trade-off is to utilise contractive autoencoders.

3.4.3 Contractive autoencoders

In contrast to denoising autoencoders, where prominence is placed on training the encoder module to be resistant against outliers within the input signal, contractive autoencoders place prominence on ensuring that the feature extraction capabilities of the encoder module are less susceptible to small outliers by ensuring that the encoder module disregards changes made to the input signal which are considered to be of less importance to the reconstruction of the input signal by the decoder. This is done similarly to how sparse autoencoders add a penalty value of $\Omega(h)$ to their loss function (as seen in Equation 3-5). However, in the case of contractive

autoencoders, the $\Omega(h)$ value is calculated differently. Within contractive autoencoders, the penalty $\Omega(h)$ is calculated as the squared Frobenius norm of the Jacobian matrix of partial derivatives associated with the encoder function, denoted by $\Sigma\left|\left|\frac{\partial f(h)}{\partial h}\right|\right|$. This is represented in Equation 3-9:

$$\Omega(h) = \Sigma\left|\left|\frac{\partial f(h)}{\partial h}\right|\right|_F^2. \tag{3-9}$$

The $\Omega(h)$ value is only applied to training examples, and it enforces the autoencoder to model features that capture information concerning the training distribution, thus applying regularisation to the autoencoder. In the following section, the applications of autoencoders are considered.

3.5 Applications of autoencoders

There are many ways to use autoencoders for learning representations. Some of the most popular applications of autoencoders are discussed in this section.

3.5.1 Autoencoders as generative models

In recent years, generative autoencoder models have become vastly popularised (Zhang *et al.*, 2020). A generative model typically simulates how data might be generated in the real world (Kingma & Welling, 2019). Generative modelling may be considered an auxiliary task, capable of providing information regarding a possible immediate future, which may assist in the construction of useful abstractions of the world and may ultimately assist in multiple downstream prediction tasks. Variational autoencoders have been extensively utilised as generative models and can be viewed as consisting of two coupled, yet independently parameterised models, the encoder or recognition model and the decoder, or the generative model. These two models support each other, with the encoder providing an estimate of the hidden variables needed for the decoder to improve its internal settings. In turn, the decoder assists the encoder in learning useful features from the data, such as different class labels.

Figure 3-6 represents a sub-collection of images found within the MNIST database. To show the generative capabilities of autoencoders, a set of hand-drawn images, as seen in Figure 3-6, had been sourced from the MNIST dataset and fed into an autoencoder neural network to be utilised as input material. The autoencoder was then tasked with learning the data and reproducing randomised samples of the data as a new, smaller dataset. The results of this experiment are shown in Figures 3-7.

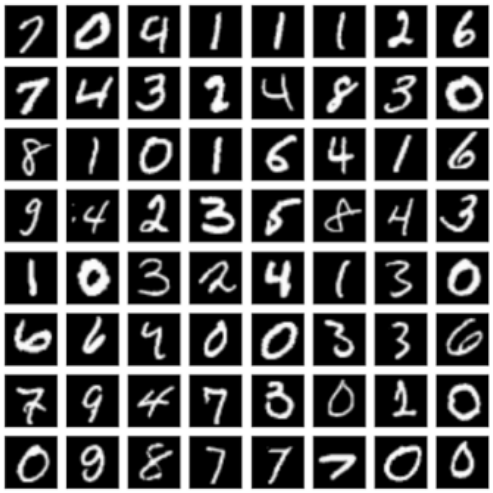

Figure 3-6: Samples from the original MNIST dataset (Rokach *et al.*, 2023)

Figure 3-7: Dataset as generated by an autoencoder neural network when presented with the MNIST dataset

As seen by the experiment's data, autoencoders possess the capability of creating similar, yet new data to that of their original input signal source. Another use case for autoencoders is that of a classification tool.

3.5.2 Autoencoders utilised for classification

Whilst autoencoders are typically utilised for unsupervised learning applications, they are also capable of being utilised in semi-supervised applications (Bank *et al.*, 2023). These applications are basically employed for improving classification results. When autoencoders are used for this purpose, they are usually synonymous with feature extractors that are added to classification networks.

Classification applications mainly take place in semi-supervised environments in which large datasets are provided for supervised learning tasks. However, only a smaller subset of the data is labelled. The core assumption within these applications is that the sample data with labels correlate to some latent representation that the latent layer of autoencoders could approximate.

Autoencoders utilised for classification are typically trained in an unsupervised manner by utilising a stacked layer training paradigm in which the layers of an autoencoder are stacked together, ultimately creating a deeper autoencoder. Parallel to this training, or after this training has commenced, the encoder module will be utilised as the initial section of a classification model whilst the decoder module is put aside. During this time, the weights of the module may either be fine-tuned or left to be static. Another strategy is to utilise the output features of the encoder to train a support vector machine.

If a domain has high dimensionality and the stacked layer training paradigm is unfeasible, another solution would be to train each layer as a linear layer before adding the non-linearity. In this case, even when denoising the input signal, a closed-form solution for each layer exists. This implies that no iterative process is needed. Another approach would be to utilise autoencoders as a regularisation technique for classification networks.

3.5.3 Autoencoders utilised for clustering

Clustering is considered to be an unsupervised technique in which the goal is to split data into groups in such a way that data fragments that are contained within a single group are similar to one another and not to fragments that are contained within other groups (Rokach *et al.*, 2023). Most clustering algorithms are relatively susceptible to the dimensions of the data they work with and, as such, suffer from the curse of dimensionality, which states that the difficulty of a machine learning problem increases as the problem's dimensionality increases. This concept may be further simplified to conclude that as the number of features or dimensions grows, the number of data samples would also need to grow exponentially. If data consists of a low-dimensional latent representation, autoencoders could be utilised to determine the representation for the data, typically comprised of considerably fewer features.

Initially, the autoencoder would be trained to use the same method mentioned in the above section. After this, the decoder is put aside, and the latent representations of every data point are stored and utilised within any given clustering algorithm as the input. The main disadvantage of using a typical autoencoder architecture for clustering is that the latent representations are constructed and trained not for the clustering application, but solely for reconstruction purposes. As such, autoencoders utilised for this purpose are typically revised to a degree.

Clustering may also be performed similarly to that of the k-means algorithm (Rokach *et al.*, 2023; Song *et al.*, 2013). However, the embeddings will be retained at every iteration of this implementation. Therefore, as training would commence within this neural network, an argument would be affixed to the loss function of the autoencoder that penalises the distance between the cluster and embedding centre.

A deep convolutional embedded clustering algorithm has been proposed by Guo *et al.* (2017). The proposed autoencoder structure model embedded features end-to-end, after which a clustering-oriented loss would be directly constructed on embedded features to perform cluster assignment and feature refinement jointly. This results in the simultaneous minimisation of the reconstruction loss of the convolutional autoencoder and the clustering loss.

3.5.4 Autoencoders utilised for anomaly detection

Anomaly detection is considered an unsupervised task in which the objective of the autoencoder is to observe and learn a profile while being provided with examples of data that conform to the requirements set by the researchers (Rokach *et al.*, 2023). The autoencoder is then expected to identify entities within the input data that do not conform to the profile of acceptable variation and label the entities as anomalies. This application of autoencoders has found great success within application areas, such as system monitoring and fraud detection.

Using autoencoders for anomaly detection is based on the assumption that a trained autoencoder can evaluate and learn the subspace in which acceptable entities can be found. Once the autoencoder has been trained, anomalies correlate to a high reconstruction error value, while standard samples correlate to a low reconstruction error.

An example of anomaly detection is provided in a study performed by Beggel *et al.* (2019) in which a robust anomaly detection algorithm for images had been adjusted, utilising a likelihood model. This allowed potential anomalies to be identified and rejected pre-emptively during the training phase of the autoencoder's lifecycle, resulting in an anomaly detection algorithm that is considerably more robust to the presence of outliers during training.

3.5.5 Autoencoders utilised for recommendation systems

Recommendation systems are systems or models that are tasked with predicting a user's preferences or affinities to different entities (Roy & Dutta, 2022). These systems are abundant in environments, such as e-commerce platforms, application library platforms, online audio and video platforms, and numerous others. Examples are plentiful, and recommendation systems are applied to concepts, such as movies, books, vacations, or electronic products daily.

The fundamental role of recommendation systems is to reduce the probability of information overload and to provide personalised suggestions that may assist users in the decision-making process. One of the more utilised methods of achieving this is using a concept called collaborative filtering (Sedhain *et al.*, 2015). Within collaborative filtering, a user's preferences are deduced by utilising the preferences of other users. This information may be filtered according to many criteria, such as age or ethnicity, and fed to the autoencoder as an input signal. The autoencoder would then be tasked with recommending items based on the patterns present in the data. As an example, consider the film viewing patterns of a demographic. If a specific piece of media is popular with Western viewers between a certain age bracket, of a particular gender, and in a specific class bracket, an autoencoder would recognise and further observe this pattern and either identify additional viewers of a similar demographic, or additional media that is similar to the original. These pieces of media (as well as similar pieces of media) are more likely to be advertised or pushed onto additional Western viewers of the same age bracket, gender and class bracket. Within this application, the veiled assumption is that human preferences are typically considerably correlated. This implies that in most scenarios, people shown to have corresponding biases in the past will also have corresponding proclivities in the future.

A final example of recommendation systems can be seen in the work of Sedhain *et al.* (2015) who used autoencoders as recommendation systems and proposed a novel autoencoder framework for collaborative filtering called AutoRec. This model consists of two variants, namely item-based AutoRec (I-AutoRec) and user-based AutoRec (U-AutoRec). Within the item-based AutoRec, the autoencoder is tasked with learning a lower-dimensional representation of item proclivities for specific items, whilst in user-based AutoRec, the autoencoder is tasked with learning a lower-dimensional representation of user proclivities.

3.5.6 Autoencoders utilised for dimensionality reduction

Autoencoder implementations for dimensionality reduction often represent data, such as images, text, and repositories, by utilising sparse high-dimensional parameters (Rokach *et al.*, 2023). Whilst many applications have adapted to work directly within this high-dimensional space, this is not always the case, and often leads to the problem with the curse of dimensionality. This led to the utilisation of autoencoders to reduce the dimensionality of data by learning a lower-dimensional manifold and then applying it to said data.

The two most used linear dimensionality reduction methods are principal component analysis and linear discriminant analysis (Rani *et al.*, 2022). Principal component analysis is an unsupervised linear dimension reduction technique (Nanga *et al.*, 2021). The main function of principal component analysis is to reduce the dimensions of a dataset while preserving as much variability as possible. Conversely, linear discriminant analysis is a supervised linear dimension reduction

technique which utilises the linear combination of features as a linear classifier for the extraction of features and dimension reduction.

Whilst autoencoders are utilised solely to perform dimensionality reduction within larger datasets, dimensionality reduction is considered a standard feature within most autoencoders. The act of projecting an initial input signal into a bottleneck representation of a lower dimension is considered a task of dimensionality reduction.

3.6 Asymmetric autoencoders

As mentioned previously, traditionally, autoencoders consist of a symmetrical design regarding the encoder and decoder modules. As such, they have an equal number of encoder and decoder nodes (Majumdar & Tripathi, 2017). The symmetric autoencoder architecture is a by-product of how greedily stacked autoencoders were originally designed and trained (Kim *et al.*, 2020; Majumdar & Tripathi, 2017).

Asymmetric autoencoders refer to autoencoder structures that do not possess an equal number of encoding or decoding nodes. They may also refer to structures that consist of a different number of encoder and decoder layers. Asymmetric autoencoders are typically used in applications where the encoding or decoding module is pre-prepared for another task; thus, the corresponding module is unutilised.

In recent years, asymmetric autoencoders have seen an increase in utilisation in several fields, such as intrusion detection (Ji *et al.*, 2020), text simplification (Zhao *et al.*, 2020) and image compression (Kim *et al.*, 2020). Additionally, recent studies have shown that asymmetric autoencoders are capable of achieving lower reconstruction errors than their deeper symmetric counterparts (Gilbert *et al.*, 2024).

3.7 Summary

In Chapter 3, an exploration of autoencoders is provided. An emphasis has been placed on autoencoder structure, functionality, and applications in machine learning and data processing. The chapter begins with an overview of autoencoder history, tracing the origins of autoencoders to the 1980s when they had been developed as a technique for unsupervised learning, particularly suited for extracting meaningful representations from data. Chapter 3 introduces the general framework of autoencoders, detailing the roles of the encoder, decoder, and bottleneck layers, which work in tandem in order to transform input data into compressed representations and to then reconstruct them with minimal distortion.

Additionally, Chapter 3 is used to discuss different types of autoencoder designed to address specific challenges, such as sparse autoencoders, denoising autoencoders and contractive autoencoders. The chapter further explores applications of autoencoders across various domains, such as generative models, classification, clustering, anomaly detection, recommendation systems and dimensionality reduction. The concept of asymmetric autoencoders, which deviate from the traditional symmetric structure by having distinct encoder and decoder designs, is also discussed to optimise performance for specific tasks.

CHAPTER 4 AUTOMATED NEURAL NETWORK ARCHITECTURE DESIGN

4.1 Introduction

The past decade has seen a drastic increase in machine learning research and applications, especially in deep learning, as deep learning has become one of the most effective, supervised and time/cost-effective machine learning approaches being utilised today (Dargan *et al.*, 2020). Deep learning can be applied to various fields, such as adaptive testing, biological image classification, cancer detection, computer vision, facial recognition, handwriting recognition, natural language processing, object detection, smart city applications, speech recognition, stock market analysis, and various dynamic information utilisation domains. Every machine learning system consists of different hyperparameters, and the selection and fine-tuning of these hyperparameters is often a computational challenge (Ali *et al.*, 2023). The hyperparameters of an algorithm regulate the learning process of that algorithm and often determine whether or not a machine learning implementation will perform optimally.

As a result of the sheer number of different elements to be considered when constructing a deep learning implementation, a difficulty barrier may form, discouraging new or non-expert users from utilising these more complex implementations. As such, within more complex implementations of artificial neural networks, the adjustment and alteration of architecture may be tedious, as there are numerous options and combinations of variables, making an exhaustive search of all hyperparameters unfeasible (Franceschi *et al.*, 2024). Consequently, the design and alteration of more complex autoencoder architectures for a given implementation are typically entrusted to the experience of an applicable researcher or practitioner. However, the architectures these professionals provide may not always be the optimal configuration for the applicable task.

It is widely accepted within the field of neural networks that there is no best autoencoder architecture for all cases, as the characteristics of the data to be processed vary (Charte *et al.*, 2019). As such, this study aims to produce a novel algorithm to automate the hyperparameter selection of an accurate autoencoder neural network architecture. In addition to the literature study on deep learning and autoencoder architectures, the information gathered by this research will then be utilised. The algorithm will be tested by sourcing several datasets. Finally, the automated construction algorithm will be applied to the sourced datasets. The datasets to be utilised within this study will be manually selected, based on criteria, such as availability of the dataset, research performed using the dataset, the popularity of the dataset, quality of the data, and quantity of data. These datasets will then be presented to the algorithm to generate a possible list of applicable autoencoder architectures. These architectures will then be further evolved and

mutated to produce an automatically procured architecture for the dataset. The performance of this architecture would ideally function comparably to the architectures that the researchers mentioned above, or professionals would have proposed. Within this chapter, a description of the automated design process of autoencoder neural network architectures is presented.

The remainder of the chapter is structured as follows. An introduction to the automation of the construction of machine learning models is presented in Section 4.2, after which some of the different methods utilised in automated neural network architecture design are explored and discussed within Section 4.3. Section 4.4 will contain a more in-depth discussion of the automated design of autoencoder-based neural networks. The proposed algorithm for automated autoencoder neural network architecture design (AANNAD) and a brief discussion of the functionality of the proposed algorithm will be addressed in Section 4.5. Finally, Section 4.6 concludes the chapter with a summary of its contents.

4.2 Automated machine learning

Recent research concerning machine learning has produced promising results regarding neural network structures and learning methods (Real *et al.*, 2020). However, the tasks of manually designing and developing neural network architectures are time-consuming, resource-intensive and often rely heavily on expert design experience (Lin *et al.*, 2024). Additionally, numerous deep learning systems currently in use can only achieve their state-of-the-art performance by utilising a number of highly complex models and investing large quantities of GPU power, time, and data.

To achieve sufficient performance in a machine learning implementation, the correct neural architectures, training procedures, regularisation methods, and hyperparameters must be selected and be in place (Hutter *et al.*, 2019). This process of hyperparameter selection often requires deep knowledge of machine learning algorithms and appropriate hyperparameter optimisation techniques which have evolved over the years.

The neural networks that had previously been trained in the past are not necessarily ineffective. However, the possibility of a more efficient neural network than that which is currently utilised within any given scenario may exist. Consequently, if the initial design and training aspects of the lifecycle of neural networks could be automated, it could streamline the implementation and effective utilisation of neural networks by making them more approachable for researchers, as well as deep learning practitioners and possibly decrease the time it would take to design and implement a neural network.

The field of automated machine learning (AutoML) aims to achieve these goals in a data-driven, objective, and automated manner by allowing the user to input a dataset, which the AutoML implementation would then analyse to determine an acceptable approach to utilise said

information (Hutter *et al.*, 2019). Therefore, by successfully implementing an AutoML system, more complex machine learning implementations may be achievable.

Several modern AutoML frameworks have emerged over the years and have been widely implemented in various areas. Auto-sklearn, which had first been proposed by Feurer *et al.* (2015), leverages Bayesian optimisation and ensemble learning to automate model selection and hyperparameter tuning, particularly for structured data. TPOT, first proposed by Olson and Moore (2016), utilises genetic programming to evolve entire machine learning pipelines. AutoKeras, which had been built on top of TensorFlow and Keras, enables neural architecture search with a user-friendly interface and supports image, text, and tabular tasks (Jin *et al.*, 2019). H2O AutoML, which had been built on the H2O open source, distributed machine learning platform, is a supervised learning algorithm which boasts high scalability, full automation and is capable of training extensive selections of candidate models and stacked ensembles (LeDell & Poirier, 2020).

Alongside these frameworks, research has continued to push the boundaries of search efficiency. Techniques such as reinforcement learning-based neural architecture search (Zoph & Le, 2022), differentiable architecture search like DARTS (Liu *et al.*, 2019), and efficient one-shot methods such as ENAS (Pham *et al.*, 2018) have drastically reduced the computational costs traditionally associated with architecture search. Meta-learning approaches, which aim to learn across tasks and generalise architecture and hyperparameter configurations, have also shown promise in accelerating AutoML by leveraging prior knowledge (Hospedales *et al.*, 2021).

In recent years, many efforts have been dedicated to automating deep neural network architecture design processes (Talbi, 2021). Some of the most prominent steps of these processes include data preparation, feature engineering, model generation and finally, model estimation (He *et al.*, 2021). The combination of these processes is referred to as the AutoML pipeline. A graphical representation of the AutoML pipeline is shown in Figure 4-1.

The processes presented within this pipeline may be utilised to aid in the task of building effective autoencoder architectures. In the following subsections, these different components of the AutoML pipeline within the lifetime of an automatic architecture design process will be discussed.

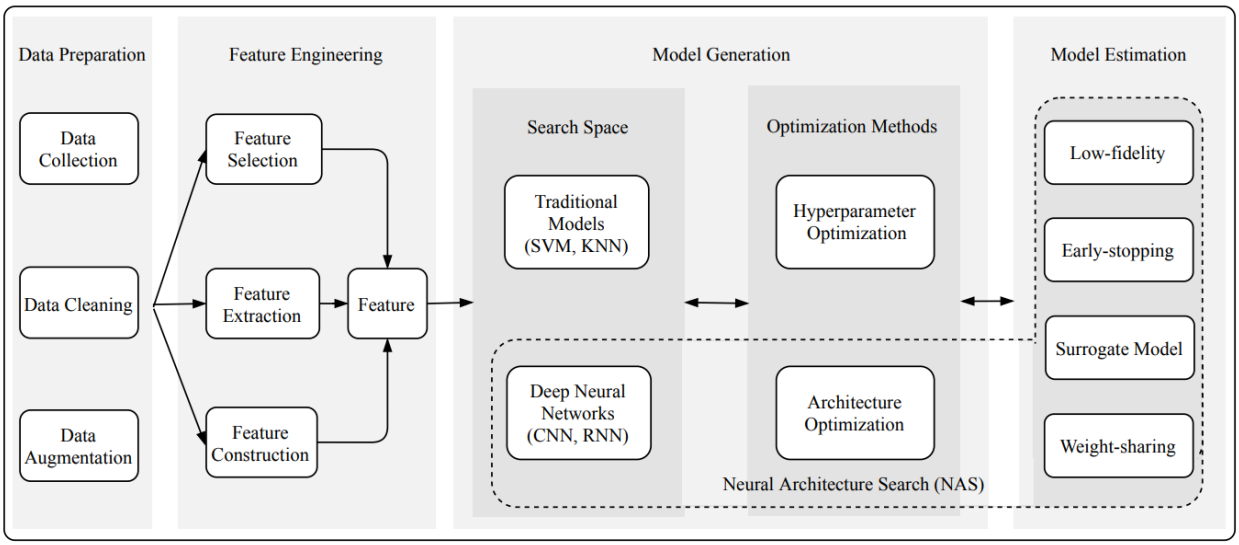

Figure 4-1: An overview of the AutoML pipeline (He *et al.*, 2021)

4.2.1 Data preparation

The initial step of the AutoML pipeline is data preparation. A flowchart of the decisions made during data preparation is shown in Figure 4-2. Data preparation can be divided into three main aspects: data collection, augmentation and cleaning.

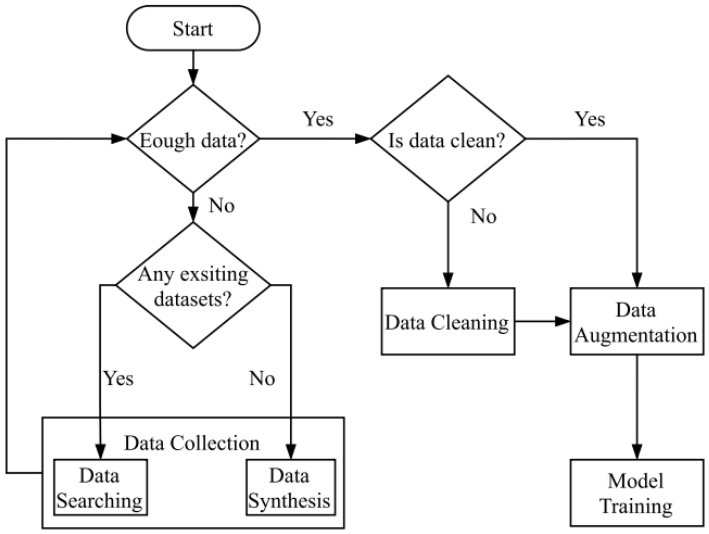

Figure 4-2: Data collection, augmentation and cleaning process (He *et al.*, 2021)

Data collection

When training a machine learning model for a specific task, it must be determined if enough data exists (Whang *et al.*, 2023). If not, the next step is to collect data by searching for existing datasets or creating new ones through synthetic data. This is a crucial step in building or expanding a

dataset. Once the data is acquired, it must be cleaned to remove any noise that could negatively impact the model's training. This is often done by satisfying integrity constraints within the data, such as key constraints, domain constraints, referential integrity constraints, and functional dependencies. After the data has been cleaned or deemed suitable, it can be augmented before the model training begins. These concepts will be discussed in more detail in the following sections.

Good-quality datasets are crucial in machine learning applications (McDonald, 2021). Numerous open datasets have been created and made available to the public. An example of this is shown within the early development of automated machine learning in which the MNIST dataset had been widely utilised. This led to the creation of several larger datasets, such as CIFAR-10, CIFAR-100 and ImageNet (Deng *et al.*, 2009). Since then, internet distribution has made many more datasets available to the public. The acquisition of existing datasets has become more accessible, with a variety of datasets being retrieved from websites, such as Elsevier Data Search, Kaggle and Google Dataset Search (GOODS). However, it should be noted that in some instances, the acquisition of publicly released datasets concerning specific data or for particular tasks, such as those related to private medical matters, may prove more complicated than the acquisition of others.

Data searching is one of the most widely used methods for obtaining data. With the internet serving as a vast source of information in recent years, searching through publicly available resources can yield datasets that may be useful in some contexts, but not others (He *et al.*, 2021). However, it is worth mentioning that this methodology may pose several problems. The first problem is that the search results may not match the required dataset's applicable features. Thus, all unrelated data would need to be filtered out of the dataset in a process referred to as data cleaning, discussed in Section 4.2.1 – “Data cleaning” before it could be utilised. Secondly, the data depicted in the search results may be incorrectly labelled or even completely unlabelled. This problem is typically fixed with the implementation of a learning-based self-labelling method. Lastly, due to the complexity of the data content of the search results, an inadequate number of descriptive labels are often also present.

Another methodology for obtaining data is data synthesis (He *et al.*, 2021). Synthetic data has gained prominence as a valuable tool across several industries (Buggineni, 2023). For some machine learning tasks, creating synthetic data rather than utilising pre-existing data, is often advantageous due to resource restrictions and even data scrutiny and biases (Buggineni, 2023; Chen *et al.*, 2021). Data synthesis poses numerous benefits, such as the capability of sharing synthetic data with other parties without the risk of information leakage. Secondly, there are no one-to-one relationships between certain synthetic and actual data records, meaning re-identification attacks (in which private or sensitive data is reverse-engineered) are impossible.

Data augmentation

The second strategy of data preparation is data augmentation. To a certain degree, data augmentation may be considered an extension of data acquisition, as the process can produce new data, based on existing data (He *et al.*, 2021). In addition, data augmentation may also serve as a method of regularisation, utilised to prevent overfitting during model training (Ding *et al.*, 2023). Data augmentation is typically implemented across three types of data: image, audio, and text. Examples of image data augmentation include rotation, altered scaling, randomised cropping and reflection.

When combined, data acquisition and augmentation can produce vast amounts of data for research. A wide range of pre-existing datasets for machine learning applications has made data acquisition considerably more accessible for this study. The concept of data cleaning is discussed in the following section.

Data cleaning

Acquired data may contain distortions or imperfections that may impede the training of a model. These distortions include, but are not limited to, corrupt, duplicate, incomplete, inaccurate, or noisy data points (Côté *et al.*, 2024). Data cleaning is detecting or repairing these distortions in a dataset to improve its quality. Traditionally, data cleaning was manual and often required expert knowledge. As such, multiple methodologies have been proposed to streamline the process. However, most of the proposed methods still relied on the input of data scientists to design and specify certain aspects of the data cleaning algorithms.

Traditional data cleaning operations typically have the objective of addressing data quality issues within a specific dataset (Neutatz *et al.*, 2021). When presented with a dataset containing errors, data cleaning operations seek to identify and/or repair those errors to derive a cleaned dataset that may be utilised in various ways.

Data cleaning often involves the removal of faulty data points, the addition of non-faulty data points into the dataset, or the alteration of less accurate or inaccurate data values with more suitable accurate or representative values (Hameed & Naumann, 2020). Examples of these operations include removing duplicate or unnecessary data from a dataset, filling in missing values that are not present within some data points, and removing whitespace from a dataset.

The data used in this study does not require further cleaning since there are no errors, inconsistencies or missing values that can skew the results or make the data difficult to analyse. Instead, the data will only need to be standardised. Standardisation of data refers to the process of adopting common data formats, models, and protocols across various platforms and organisations (Yella & Kondam, 2022). Data standardisation often reduces the risks of errors

within operations that utilise a dataset, reduces costs and reduces delays caused by unstandardised data (Zeb *et al.*, 2021). Examples of text-based data standardisation include capitalising letters, removing punctuation marks, and changing alphanumeric values to numeric values and *vice versa*. After applicable data has been obtained, prepared, and cleaned, it is ready for the next step of the AutoML pipeline, namely feature engineering. For autoencoder systems, which depend on clean input to learn accurate representations, this preparatory step is especially important and well-suited to automation.

4.2.2 Feature engineering

It could be argued that in most machine learning applications, the data and features utilised within their lifecycles determine the boundaries of machine learning processes (He *et al.*, 2021). Different algorithms and models can only approximate these boundaries. Based on this context, feature engineering consists of concepts that maximise the quality and amount of utilisable and relevant information, or features from data so that machine learning processes may use the information mentioned above. Feature engineering is defined as the task of improving the predictive modelling performance on a dataset by transforming its feature space (Nargesian *et al.*, 2017). Feature engineering comprises three sub-divisions: feature selection, feature extraction, and feature construction.

Feature selection

The objective of feature selection is to build a more comprehensible feature subset, based on the initial feature set by minimising the amount of non-relevant or redundant features, resulting in better model performance and clean and understandable data (He *et al.*, 2021). This often results in a simplified feature set and model. Furthermore, the selected features are correlated with the objective values of the data.

Within feature selection, the original dataset is utilised to determine a subdivision of features, based on the feature relevance and redundancy of the original set (Venkatesh & Anuradha, 2019). Irrelevant features refer to features that are not required for accurate prediction. Feature relevancy is measured by considering the characteristics of the data contained within the feature, not by its value. Redundant features refer to features that imply the correspondence of another feature. Removing these redundant features by utilising feature selection not only reduces computational cost and storage requirements, but also avoids significant loss of information or degradation of learning performance (Li *et al.*, 2017). Based on the degree of relevance and feature redundance, feature subsets may be divided into four categories: irrelevant features, redundant and weakly relevant features, weakly relevant and non-redundant features, and finally, strongly relevant features (Yu & Liu, 2004). Note that an optimal feature subset would consist of weakly relevant, but non-redundant features and strongly relevant features.

Feature selection methodologies are divided into three categories, based on the interactions with learning models (Venkatesh & Anuradha, 2019). These three categories are filter, wrapper and embedded methods. Filter method features are selected, based on statistical measurements. The filter method is independent of the system's learning algorithm and utilises less computational time than the other two techniques. The functionality of a wrapper method is dependent on the classifier of a system, from which the best subset of features is selected. The cross-validation wrapper methods and repeated learning steps are computationally more expensive than filter methods; however, they are considerably more accurate than filter methods. Finally, the embedded method utilises hybrid and ensemble learning methodologies for feature selection. As the embedded method is a collective decision, it presents a more favourable performance rate than the other two methods. A brief comparison of filter, wrapper and embedded methods is shown in Table 4-1.

Table 4-1: Strengths and weaknesses of the different feature selection methods (Venkatesh & Anuradha, 2019)

Feature selection method	Strengths	Weaknesses
Filter method	• Computationally faster than the Wrapper and Embedded methods • Independent of the learning algorithm • Suitable for low-dimensional data	• It does not consider the correlation between the features. • It does not consider the correlation between classifiers. • Fails to recognise the patterns properly during the learning phase
Wrapper method	• It considers the correlation between the features and class labels. • More accurate than the Filter method • The technique also considers the dependencies between the features.	• Causes overfitting • Computationally more complex • Iteratively evaluates the selected feature subset. • Some features may not be evaluated when dropped at the initial stage. • There is searching overhead.
Embedded method	• Computationally more efficient than the Wrapper method • More accurate than the Filter and Wrapper methods	• Computationally costlier than the Filter method • Not suitable for high dimensional data • Poor generality

There are five main stages of feature selection: determining search direction, strategy, evolution criteria, stopping criteria and finally, validation of the final results. This process can be seen in Figure 4-3.

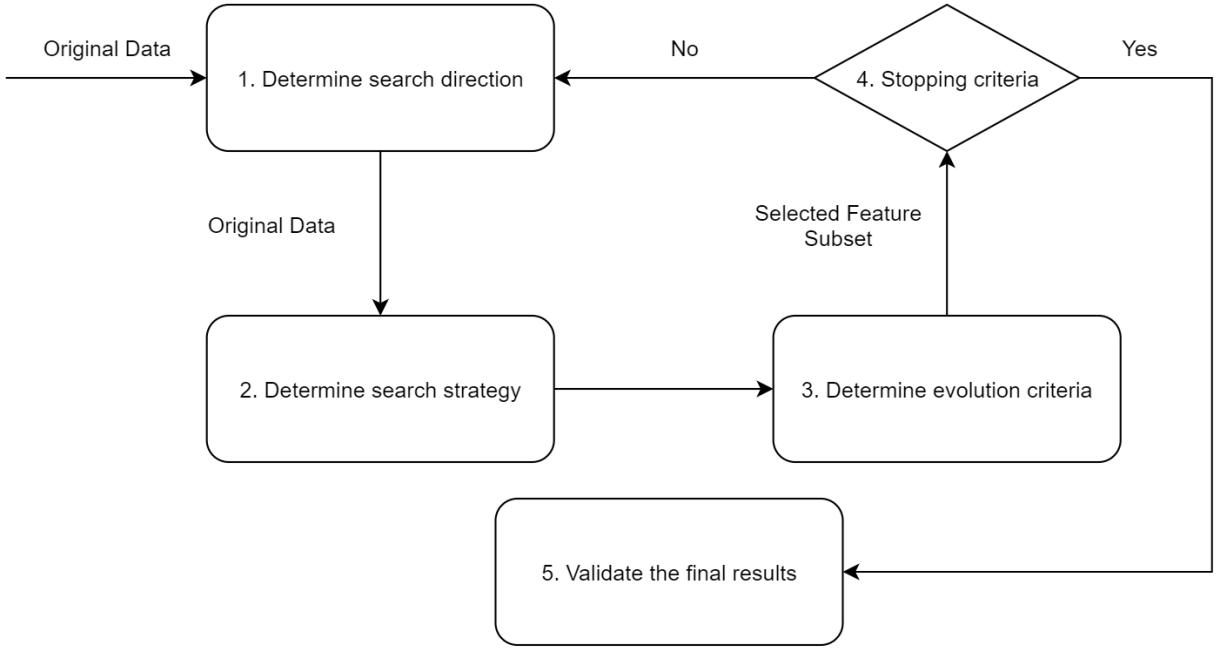

Figure 4-3: Stages of the feature selection process (Venkatesh & Anuradha, 2019)

Initially, the search direction is specified: forward, backward, or random search (Venkatesh & Anuradha, 2019). Within forward search, the search process is initiated with an empty set of features, with new features added recursively in every set. Conversely, the backward search starts with a complete set of features and removes features iteratively until only the desired subset of features remains. The final variant of the search direction is random search, which constructs a feature subset by iteratively adding and removing features. A search strategy may be applied once the search direction has been decided. Search strategies can be divided into three types: exponential search, sequential search and randomised selection strategy (Agrawal *et al.*, 2021).

Within exponential search, also referred to as complete search, the strategy attempts to find all possible feature subsets (Islam *et al.*, 2022). Then, the exponential search strategies attempt to obtain the optimal subset from this collection. It should, however, be noted that complete search strategies are considered exhaustive and come with the disadvantage that the optimal solution is not guaranteed to be found. The issue lies in the computational cost and time required to explore all possible feature subsets, which can make them impractical in many real-world scenarios. An example of a complete search strategy is the branch and bound technique.

Sequential search algorithms function by adding and removing features sequentially (Venkatesh & Anuradha, 2019). Sequential forward selection is a sequential search technique in which features are sequentially assigned to empty candidates until the criterion is not altered (Islam *et al.*, 2022). Sequential forward selection aims to determine an acceptable subset of features that result in optimal computational performance while reducing overfitting by removing irrelevant information. In contrast, the sequential backward selection approach aims to reduce the dimensionality of the initial feature subspace with a minimum reduction in system performance. Sequential backwards selection aims to eliminate features from the provided feature list until a predefined number of features is reached.

Random search utilises randomness to explore a search space whilst also preventing an algorithm from being trapped within a local optimum (Agrawal *et al.*, 2021). A heuristic search will not always produce the most optimal solution, like a complete search. However, it often finds a feasible solution within an acceptable space and time complexity. Heuristic search methods also run comparatively faster than other search strategies. Examples of heuristic search strategies include simulated annealing, random generation, and metaheuristic algorithms.

Stopping criteria determine when a feature selection process should terminate (Venkatesh & Anuradha, 2019). A well-functioning stopping criterion would lower computational complexity during the determination of an optimal feature subset and, ideally, prevent overfitting. The decisions made in the previous stages typically influence the selection of an ideal stopping criterion. Examples of common stopping criteria include a predefined number of features, a predefined number of iterations, a percentage of advancement over two successive iteration steps, and a stopping criterion, based on the evaluation function.

Feature set validation techniques are used to validate the results of a feature selection process. These techniques include, but are not limited to, a confusion matrix, cross-validation, the Jaccard similarity-based measure, and the Rand index. The most commonly used method is cross-validation. In addition, after feature selection has been concluded, the subset may be subjected to feature extraction.

Feature extraction

Feature extraction may be considered a dimensionality reduction procedure that seeks an optimal transformation from input data into a feature vector that can be utilised for input data for a machine learning algorithm (Storcheus *et al.*, 2015). One of the significant goals of feature extraction is to increase the accuracy of a learning model by extracting salient features (features that are understandable and relevant to the learning algorithm) from a provided set of input data, whilst potentially removing noisy and redundant information from the data. Additional objectives that may be achieved by utilising feature extraction may include low-dimensional representations for

data visualisation and data compression to optimise data storage requirements, whilst increasing training and inference speed. Deep learning models, such as autoencoders, can serve as automated feature extractors and may be used in task-specific modelling, including object detection, image classification, natural language processing, and speech recognition (Berahmand *et al.*, 2024). In contrast to feature selection, feature extraction modifies the initial features (He *et al.*, 2021). The core mapping function of a feature extraction algorithm may be implemented in multiple ways, with the more well-known approaches being independent component analysis, principal component analysis, linear discriminant analysis (LDA), non-linear dimensionality reduction and isomap.

Feature construction

Feature construction is constructing new features from the basic feature space determined from the feature selection process or raw data (He *et al.*, 2021). This aims to improve the ability of the initial features within a dataset to be representative. The feature construction process is, in large part, typically performed by utilising human expertise. One of the most used methodologies for feature construction is pre-processing transformations, such as normalisation, feature discretisation and further standardisation if necessary. Additionally, the transformation of different feature types may vary. For example, Boolean-based features are often subjected to concepts, such as conjunctions, disjunctions and negation, while numerical features are often subjected to minimum, maximum, addition, subtraction and mean calculations. Therefore, it is typically impossible to manually explore all possible feature construction options for a feature set.

4.2.3 Model generation

Model generation primarily consists of search space and optimisation methods (He *et al.*, 2021). The search space of the model generation process refers to the model structures that can be developed and optimised for utilisation within a given scenario. The models that may be utilised for this purpose are broadly divided into two categories: traditional machine learning models and deep neural networks. Optimisation methodologies utilise two parameters: hyperparameters specified before training starts, such as learning rate, filter size, and the number of layers for deep neural networks, and model parameters, such as neural network weights. In this study, neural architecture search is employed to optimise the hyperparameter selection. Neural architecture search algorithms are typically used to find a relatively well-performing architecture when presented with certain restrictions (Baymurzina *et al.*, 2022). The process of neural architecture search is categorised into three dimensions, as shown in Figure 4-4: search space, architecture optimisation (search strategy) and model evaluation (performance estimation strategy) (Elsken *et al.*, 2019).

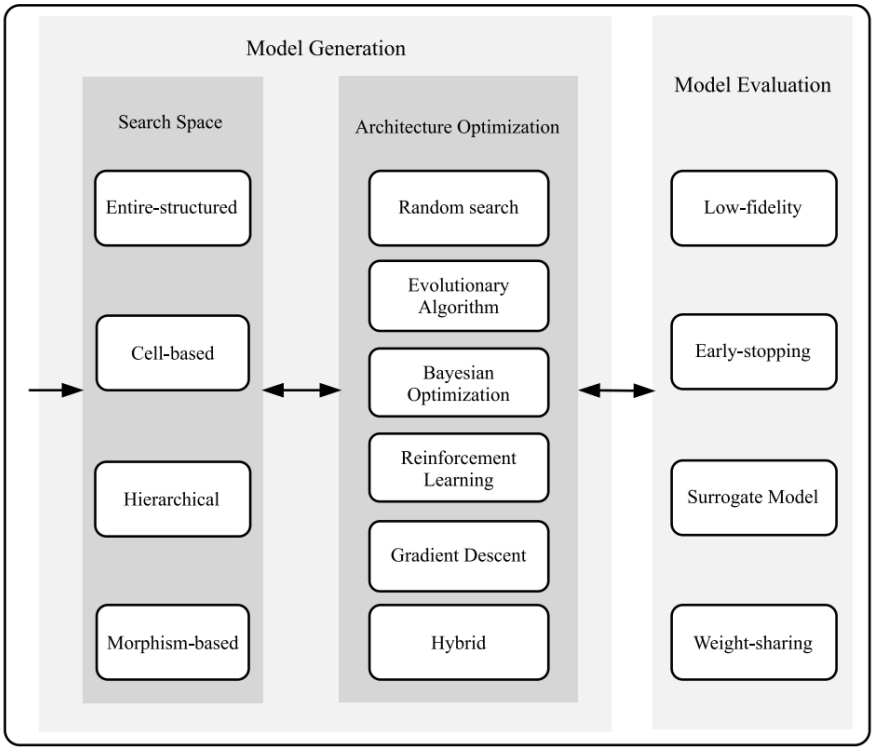

Figure 4-4: An overview of the neural architecture search pipeline

The search space defines the elements of neural network architectures. The four most utilised search space strategies employed are discussed in Section 4.2.3 – “Search space”. In addition, architecture optimisation defines methods of guiding the search to most effectively locate model architectures with acceptable performance once the search space has been determined, as discussed in Section 4.2.3 – “Architecture optimisation”. Finally, once a model has been generated, its performance is evaluated through the model evaluation step, addressed in Section 4.2.4.

Search space

The search space of a neural architecture search algorithm defines which architectures can be represented in principle (Elsken *et al.*, 2019). The search space size can be reduced by incorporating prior knowledge concerning typical architectural properties well-suited for similar tasks. While reducing the search space can improve neural architecture search algorithms, it is essential to note that this may introduce some human bias and limit the discovery of superior architectural building blocks. The most commonly utilised search spaces are entire or chain-structured, cell-based, hierarchical, and morphism-based (Ganepola & Wirasingha, 2021). Two simplified examples of an entire-structured neural architecture search space can be seen in Figure 4-5.

Within an entire-structured search space/chain-structured search space, each search space is built by assembling a predefined amount of nodes on top of one another (Elsken *et al.*, 2019).

Every node is a layer, as represented by $L1 - L4$ in the image below, with each layer performing a specific operation within the neural architecture. The left-hand representation shows a simpler structure to implement. On the other hand, the right-hand structure is more complex, allowing for arbitrary skip connections and exploration of intricate neural architectures. It should, however, be noted that entire-structured neural architectures possess two disadvantages: they are computationally expensive and contain insufficient transferability, meaning that a model that is generated on a smaller dataset might not fit a larger dataset, which would demand the generation of a new model for the larger dataset.

The search space utilised within this study's algorithm is entirely structured. However, the simplified structure on the left is used in the experiments done for this research. This is because the problems modelled are not complex enough to warrant the use of arbitrary skip connections. This structured approach aligns naturally with the symmetrical design of autoencoders, which would make it a strong candidate for automation.

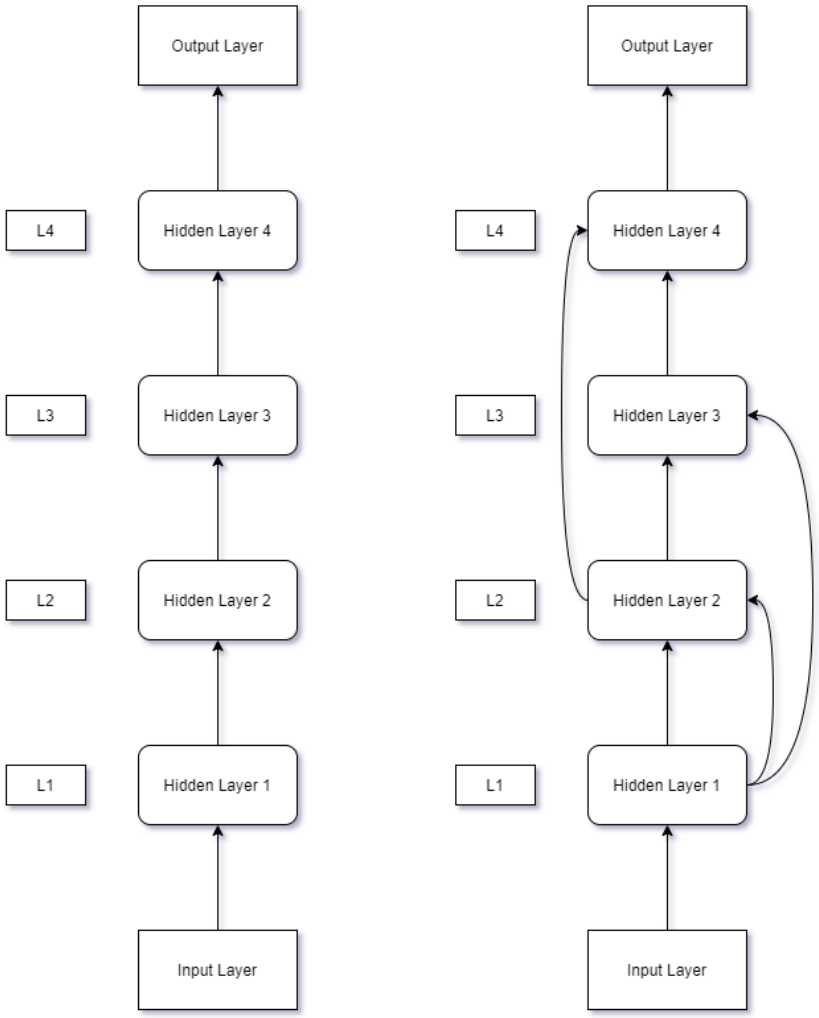

Figure 4-5: Two examples of entire-structured neural architectures in the form of basic autoencoder neural networks

The cell-based search architecture, initially proposed by Zoph *et al.* (2018), was developed to enhance the transferability of generated models (Wang & Zhu, 2024). This neural network architecture consists of a predetermined number of repeated cell structures, reflecting the observation that numerous human-designed models with a good performance rate are also constructed by assembling a consistent amount of modules. For instance, the ResNet collection, which includes variants, such as ResNet152, ResNet101, and ResNet50, is built by assembling multiple bottleneck modules. Throughout academic texts, this recurring module is often referred to as a cell, block or motif; in this particular study, it is denoted as a cell. A typical structure of a cell-based search architecture implementation can be seen in Figure 4-6. Due to the performance of cell-based search architectures, cell-based neural architecture search approaches have undergone additional development.

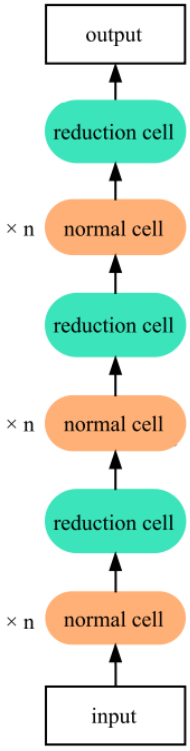

Figure 4-6: Example of a cell-based search architecture (He *et al.*, 2021)

The core concept of hierarchical-based search spaces is that starting from a small set of primitives, such as convolutional and pooling operations at the bottom level of the hierarchy, higher-level computation graphs or motifs are formed by using lower-level motifs as their building blocks (Liu *et al.*, 2018). The motifs at the top of the hierarchy are stacked multiple times to form the final neural network. This approach enables search algorithms to implement powerful hierarchical modules where any change in the motifs is immediately propagated across the whole network. An example of a hierarchy-based search space can be seen in Figure 4-7.

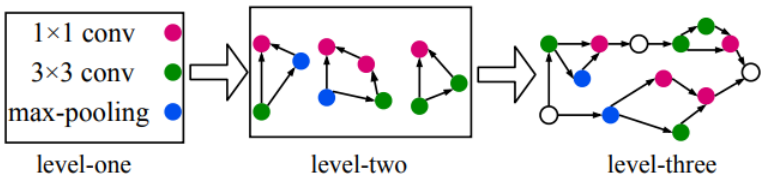

Figure 4-7: Graphical representation of a three-levelled hierarchical architecture (He *et al.*, 2021)

Different levels of primitive operations are assembled into higher-level cells. Level one contains primitive operations, such as 1x1 and 3x3 convolutional operators and 3x3 max-pooling operators. These primitive operations are then used to construct level two cells. From here, the level two cells are used as primitive operations to generate level three cells. Following this paradigm, a single motif corresponding to a neural network's complete architecture would represent the highest-level cell.

Within morphism-based search spaces, earlier versions of the search space would design new neural network architectures by utilising an identity morphism (often referred to as an IdMorph) transformation between the layers of a neural network. This later evolved into the morphism-based search space paradigm, when child networks were able to assume the knowledge from their well-trained parent network, growing into a more robust network within a shortened training time (He *et al.*, 2021). A graphical representation of a morphism-based search space operation is shown in Figure 4-8.

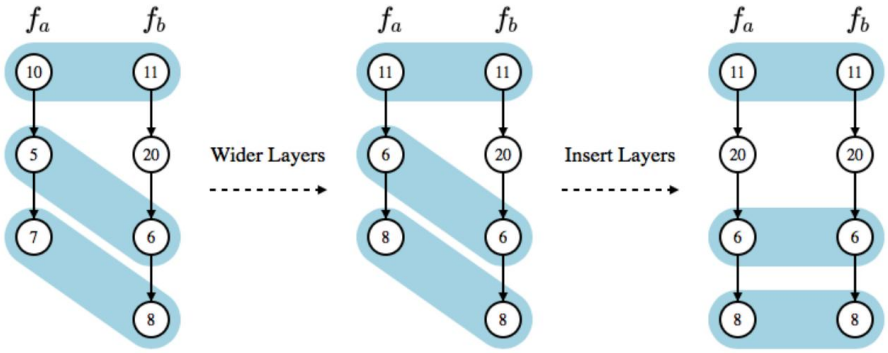

Figure 4-8: Examples of possible Morphism-based search space operations (Jin *et al.*, 2019)

The concept of network morphism refers to the technique utilised to morph and alter the architecture of a neural network, but to keep its functionality (Jin *et al.*, 2019). Within a morphism-based search space, a new architecture is created by utilising a trained neural network and modifying it, using network morphism operations, such as incorporating a skip connection or adding a layer.

In the above figure, two neural networks, f_a and f_b are presented. By utilising a morphism-based search space, it is shown that network f_a can be turned into a network f_b by first increasing the weight of all nodes within the network, then adding another layer to the network entirely. After the initial alteration, the new architecture would require a few more epochs of training towards a possible improvement in terms of performance. The algorithm produced during this study makes use of a modified paradigm of the morphism-based search space.

Optimisation methods

Once a search space has been defined, a search is conducted to determine the best-performing architecture and hyperparameters. The process that follows this is referred to as architecture optimisation.

Architecture optimisation

Different methodologies may be utilised in this search, each with advantages and disadvantages. Examples of these search strategies include, but are not limited to, evolutionary algorithms, reinforcement learning, gradient descent models, surrogate model-based optimisation, and grid and random search (Elsken *et al.*, 2019).

Evolutionary algorithms are designed to function similarly to biological evolution. An evolutionary algorithm is considered to be a generic population-based metaheuristic optimisation algorithm (He *et al.*, 2021). When compared to the alternative, more traditional optimisation algorithms, such as exhaustive methods, evolutionary algorithms are considered a mature global optimisation method that has a broad range of application areas and high robustness and with the ability to effectively address some of the more complex tasks that a traditional optimisation algorithm would struggle to solve, without the limitation of the nature of the task acting as a limiter. Evolutionary algorithms typically comprise four main steps: selection, crossover, mutation and update, as seen in Figure 4-9.

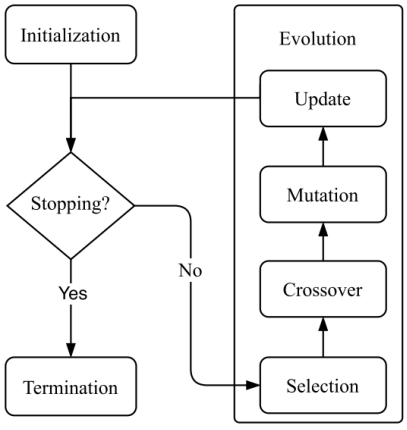

Figure 4-9: Overview of the evolutionary algorithm

Within the selection step, a portion of the generated networks is selected to maintain comparatively better-performing networks and eliminate less well-performing ones. The selection step may make use of different selection strategies to achieve this (Yadav & Sohal, 2017). Some examples of these strategies include roulette wheel selection, rank selection and tournament selection.

The roulette wheel selection strategy functions by assigning each individual a probability of selection proportional to its fitness value, with higher fitness increasing the likelihood of being chosen (Yadav & Sohal, 2017). The rank selection strategy, by contrast, bases selection probabilities on the relative ranking of individuals rather than their raw fitness. This is mainly done to ensure diversity and prevent highly fit individuals from dominating, leading to slower convergence. Finally, tournament selection randomly selects a subset of individuals, with the fittest in the group winning and being added to the pool of networks that might reproduce. The selection pressure can be adjusted by changing the tournament size, allowing a methodology to balance diversity and convergence speed.

After the selection process has concluded, the crossover process takes place. Crossover operations are utilised in order to generate offspring by combining the genetic information of two or more parent nodes (Katoch *et al.*, 2021). This process is comparable to biological reproduction and crossover, in which genetic information is shared. Multiple crossover methodologies may be utilised for specific purposes. During the three operations mentioned above, many new networks are generated and removed, based on the resources available to the system.

After the abovementioned steps have concluded, various new networks will be generated and need to be culled. This final process of culling the collection of acceptable best-performing evolution networks is called the update step. Another methodology utilised within architecture optimisation is reinforcement learning. A graphical representation of reinforcement learning can be seen in Figure 4-10.

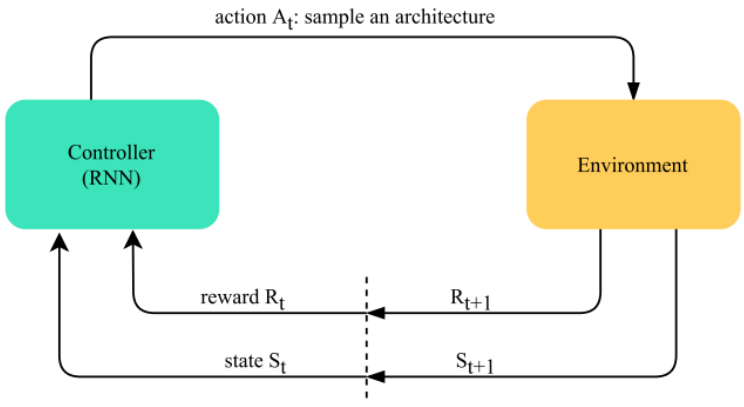

Figure 4-10: Overview of reinforcement learning utilised within neural architecture search (He *et al.*, 2021)

The core concept of reinforcement learning is that an agent interacts with an environment and learns an optimal policy through trial and error (Naeem *et al.*, 2020). A reinforcement learning agent would interact with an environment over time. With each time step t the agent would observe a state S_t within the state space of S. The agent then selects an action, a from an action space A, following a policy of $\pi(a_t|s_t)$, which represents the agent's behaviour at that time. A mapping from the state S_t to action a_t receives a scalar reward r_t afterwards, transitioning to the S_{t+1}, based on the reward function of $R(s,a)$, determined by the environment dynamics or model. The agent aims to maximise the reward it receives from a particular action.

Within this example, the agent refers to the controller recurrent neural network, which executes the action of sampling a new architecture from the provided search space (He *et al.*, 2021). The environment would then refer to utilising a standard neural network training paradigm to evaluate and train one of the networks produced by the controller, after which the correlating results (such as accuracy or error rate) would be returned. This process is repeated until suitable architectures are selected or the system's computational resources are depleted.

While evolutionary and reinforcement learning algorithms utilise a discrete search space to sample neural architectures, gradient descent utilises a continuous and differentiable search space to search for applicable neural architectures, using a *SoftMax* function to relax the discrete space. To obtain a local minimum of a differential equation, gradient descent acts as a first-order iterative optimisation algorithm (Haji & Abdulazeez, 2021). Several variations, based on the gradient descent optimisation technique, have been developed recently to enhance its efficiency further. These variants are Batch Gradient Descent, Stochastic Gradient Descent and Mini-batch gradient descent. A graphical representation of the basic concept of a gradient descent implementation is shown in Figure 4-11.

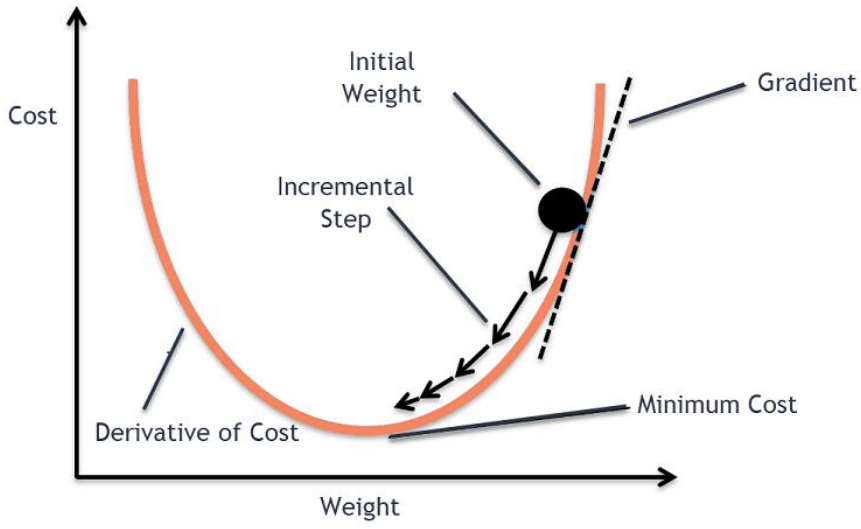

Figure 4-11: Gradient Descent (Haji & Abdulazeez, 2021)

Gradient descent is a way to minimise an objective function $J(\theta)$ parameterised by a model's parameters $\theta \in R^d$ by updating the parameters in the opposite direction of the gradient of the objective function $\nabla_\theta J(\theta)$ in regard to the parameters of the current model. The learning rate η is utilised to determine the magnitude of the weight changes leading to a localised minimum. In Figure 4-11, the initial weight follows the incremental steps shown until the minimum cost is achieved by following the direction of the slope created by the objective function.

Surrogate model-based optimisation is another group of architecture optimisation methods, and it is further discussed in Section 4.2.4 – "Surrogate-based methodology". The concepts of grid and random search also relate to hyperparameter optimisation, which are discussed in the following subsection.

Hyperparameter optimisation

After identifying the most promising neural architecture with the highest potential for achieving good performance, a new hyperparameter set must be designed and utilised to retrain or fine-tune the architecture (Elshawi *et al.*, 2019).

Every machine learning model consists of two types of parameters, namely hyperparameters that the model designer must manually set before training and normal parameters that are optimised and altered during the training of the machine learning model (Waring *et al.*, 2020). The hyperparameters are considered the different settings that can control the behaviour of the machine learning model in some way, often in a manner that is unique to the particular model being used.

The most basic task of automated machine learning is to automatically set and alter these hyperparameters to optimise the performance of the model. The performance of most machine learning methods can depend significantly upon these hyperparameter settings; thus, it is one of the most critical tasks in machine learning. Automated hyperparameter tuning techniques can be classified into two main categories: black-box optimisation and multi-fidelity optimisation (Elshawi *et al.*, 2019). A graphical representation hereof can be seen in Figure 4-12.

Black box optimisation techniques include, but are not limited to, grid search, random search, Bayesian optimisation, simulated annealing and genetic algorithms.

Grid search is a basic, yet straightforward solution, utilised within hyperparameter optimisation. This approach evaluates every combination of hyperparameters to identify the optimal set. However, grid search can be computationally expensive and often becomes infeasible due to the curse of dimensionality (Elshawi *et al.*, 2019). This challenge arises as the number of comparisons

grows exponentially with the increase in hyperparameters, making the implementation of grid search more complex and time-consuming.

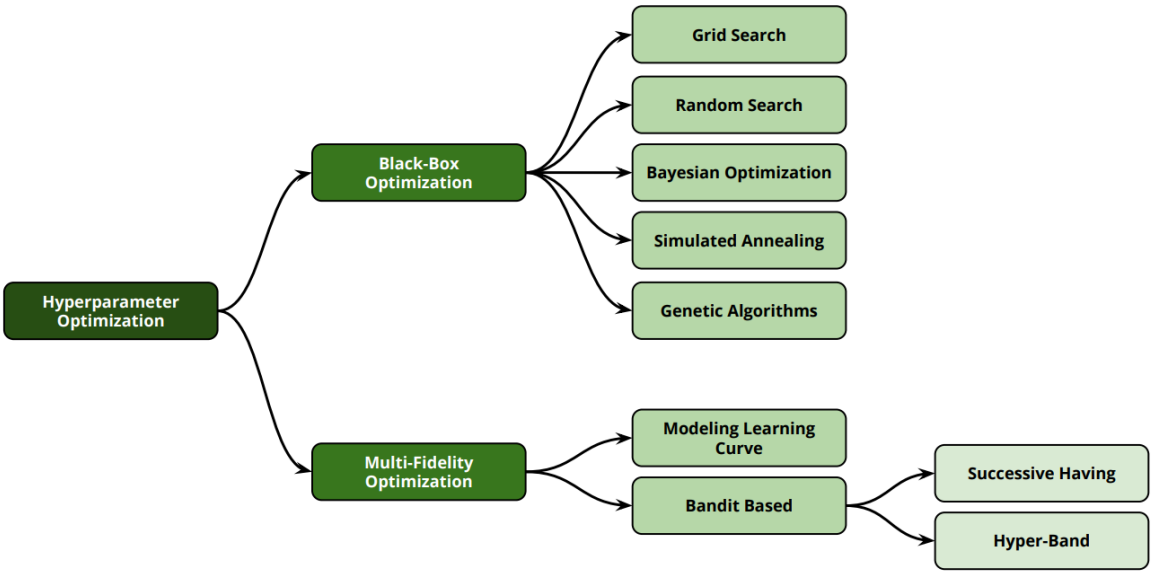

Figure 4-12: A Taxonomy for the Hyperparameter Optimisation Techniques (Elshawi *et al.*, 2019)

An alternative to grid search is random search. The different possible configurations are randomly sampled in a random search until a set computation or resource budget has been exhausted. When presented with a computational budget, a random search tends to obtain more ideal solutions than a grid search implementation. One of the core advantages of the grid and random search strategies is that both can be parallelised across numerous workers, a rather crucial trait when faced with big data.

Bayesian optimisation algorithms are effective methodologies typically used for optimising black-box functions (Terayama *et al.*, 2021). Bayesian optimisation is especially valuable in scenarios where evaluating the objective function is resource-intensive and where human intuition alone may not significantly improve model accuracy (Victoria & Maragatham, 2021). This optimisation approach consists of two main components: a Bayesian statistical model that approximates the objective function and an acquisition function that determines where to sample next (Frazier, 2018). The Bayesian model often takes the form of a Gaussian process, which provides a probabilistic estimate of the objective function across the search space. The process begins by collecting initial data points, whereafter the Bayesian optimisation algorithm constructs a surrogate model. This process is typically Gaussian in nature. The acquisition function then uses this surrogate model to decide the next sampling point by balancing exploring new areas and exploiting known promising areas. After collecting new sample data, the Gaussian process model is updated, using Bayes' theorem, refining the surrogate model's accuracy. This iterative process

continues, with the acquisition function guiding each step until a predetermined stopping criterion is met, such as a time constraint, processing budget, or convergence threshold.

Simulated annealing is a probabilistic optimisation technique most commonly used in combinatorial optimisation problems (Sarfi, 2023). Simulated annealing is based on the metallurgy technique of cooling a material to reduce its defects and structural integrity, after which it is carefully heated. Simulated annealing aims to determine the global optimum of a given optimisation problem. This is done by generating a sequence of different solutions that, over time, converge on the global optimum for that given problem. Various steps are conducted within simulated annealing (Elshawi *et al.*, 2019). Initially, simulated annealing techniques sample a single value, the current state, to be applied to all hyperparameters, after which the model's performance is tested. After this initial step, the technique randomly updates the value of one of the hyperparameters by choosing a value from the immediate collection of hyperparameters to obtain a neighbouring state. Thirdly, the approach evaluates the performance of the model, based on the neighbouring states. It then compares the performance information collected from the adjacent and current states. Finally, the user can accept or reject the neighbouring state as one of the current states, based on predefined criteria.

Genetic algorithms discussed in Section 4.2.3 may also be utilised during hyperparameter optimisation. These operations include, but are not limited to, mutation, survival of the fittest, and crossover breeding (Lambora *et al.*, 2019).

Multi-fidelity optimisation is an optimisation technique that attempts to decrease the evaluation costs of a model by combining numerous low-cost, low-fidelity evaluations with a small amount of expensive high-fidelity evaluations (Elshawi *et al.*, 2019). This optimisation methodology is essential in real-world applications, as applications that utilise large datasets might take considerable time to train but one hyperparameter. Additionally, within multi-fidelity optimisation, the ability to evaluate samples from different levels exists. In multi-fidelity optimisation, two different evaluation functions may be used: low-fidelity and high-fidelity. Where low-fidelity evaluations only evaluate a smaller subset of the dataset and are less expensive than high-fidelity evaluations, high-fidelity evaluations output precise evaluations concerning the entire dataset.

The modelling learning curves methodology is a model optimisation methodology in which the learning curves are modelled during hyperparameter optimisation. A learning curve is considered a graphical representation that provides insight into the learning rate of a neural network by plotting a generalisation of its performance against some resource, typically the number of epochs or the number of training examples provided to the neural network (Viering & Loog, 2023). Learning curves may function as an important tool for model selection, reducing the computational complexity of a model's training, using hyperparameter tuning, and even predicting the effects of

additional training data on the model. Another important concept of this methodology is learning curve extrapolation. Learning curve extrapolation is utilised to predict whether or not to terminate the training of a configuration to reduce data collection costs. The learning process is terminated if the predicted configuration's performance is not accepted or performs worse than the performance of the top-performing models that had been trained before.

Bandit-based algorithms also fall under the category of multi-fidelity optimisation and have been shown to function as a powerful tool in deep learning optimisation tasks (Elshawi *et al.*, 2019). Bandit-based methodologies may be categorised into two subcategories: Successive halving and Hyper-Band. Successive halving is a powerful multi-fidelity methodology that uses a given time budget variable, B. All potential configurations are evaluated according to the time budget variable B and ranked, based on performance. This collection of configurations is then halved, with the half containing the configurations that did not perform as well as the others removed. Finally, the time budget variable B is doubled, and the previous steps are repeated until only one configuration remains. Hyper-Band is another bandit-based multi-fidelity exploration technique. It attempts to optimise a search space when a selection is randomly made from a collection of configurations. Like successive halving, the Hyper-Band methodology also receives a budget variable B, partitioning it into several combinations of configurations, with budgets assigned to each configuration. The successive halving methodology is then utilised on each random sample configuration.

After hyperparameter optimisation has concluded and a neural network is successfully created, the next step is to evaluate the model and its effectiveness. This process is discussed in the following sections.

4.2.4 Model evaluation

The simplest methodology for model evaluation is to train the neural network on a training partition of a dataset and then evaluate its performance on a validation partition (Hutter *et al.*, 2019). This methodology may require considerable computational resources and time. Therefore, this section summarises several potential algorithms typically used to accelerate and improve model evaluation.

Low-fidelity optimisation

The time it takes to train a model correlates with the model's size and the dataset's characteristics. Therefore, model evaluation may be accelerated by applying low-fidelity optimisation to different areas of the model. Fidelity is the measurement of accuracy between model approximations and black box predictions (Molnar, 2020). Thus, lower-fidelity optimisation refers to a representation of the data that is less costly to produce, altered, or of less accurate quality.

Low-fidelity optimisation can be used in several different ways; for example, the number of images or the resolution of images utilised within image classification tasks can be decreased, such as done in the research performed by Klein *et al.* (2017), in which model evaluation speed is significantly increased by training models on subsets of the training set. Additionally, low-fidelity optimisation could also be implemented by reducing the model size by training with fewer filters per convolutional layer, as done in the research conducted by Zoph *et al.* (2018).

The datasets utilised within this study include the MNIST dataset, the SOCOFing dataset and the CCFD dataset. More information regarding these datasets is provided in Chapter 5 of the study. These datasets comprise different data forms to ensure the proposed algorithm can function in various environments. These different types of data include, but are not limited to, image-based data, biometric data and structured data, and as such, may be subjected to low-fidelity optimisation to increase the learning rate of the tasks, whilst reducing the processing cost. However, low-fidelity optimisation is not performed within this study, as one of the study's main aims is accuracy within the proposed architectures, requiring extensive training with high-fidelity data.

Weight sharing

Within the early years of neural architecture search development, algorithms often sampled large quantities of architectures from a search space, training each from scratch to validate its performances (Xie *et al.*, 2021). Unfortunately, these approaches often demanded considerable computational power, preventing these architectures from being utilised or transplanted to other applications. To mitigate this problem, researchers proposed re-utilising some of the weights of previously optimised architectures or sharing computation among different, but relevant, architectures sampled from the collective search space. This reduced search and evaluation costs by several orders of magnitude. An example of this can be seen in the research performed by Pham *et al.* (2018), in which different parameters would be shared across child networks. This led to a thousandfold increase in the efficiency of the neural network design process utilised within their research.

Whilst weight sharing yielded promising results within larger and more complex datasets and implementations, the concept is not feasible within the scope of this study. As a result, weight sharing is not implemented within the proposed algorithm.

Surrogate-based methodology

The surrogate-based methodology (also called meta models) presents another powerful tool that can be utilised to approximate black box functions, or problems in which the objective functions are considerably expensive to evaluate and derivative information is not available (Vu *et al.*,

2017). The core idea behind surrogate-based methodologies is to iteratively construct surrogate models that approximate the black box functions and utilise them to search for the optimal solution. An example of the surrogate method being employed is put forward by Luo *et al.* (2020), in which they propose a semi-supervised neural architecture search approach called SemiNAS. SemiNAS trains an initial accuracy predictor (the surrogate) by utilising a small subset of the architecture and accuracy data pairs. Afterwards, it uses the achieved accuracy predictor to predict the accuracy of a more significant number of architectures without evaluating them. Additionally, the generated data pairs are added to the original dataset to improve the predictor further.

Early stopping

Early stopping has been utilised in classical machine learning applications to prevent overfitting and perform regularisation (Aggarwal, 2018). The effective functionality of early stopping is reducing the size of the parameter space to a smaller neighbourhood within the initial values of the parameters.

In terms of model evaluation, early stopping functions similarly. It saves processing time and resources by preventing evaluations predicted to deliver poor performance results on the validation set. Early stopping also addresses the concerns of a phenomenon referred to as learning speed slow-down, in which the accuracy of an algorithm ceases to improve after a set number of epochs and may even worsen in its performance (Ying, 2019). When applied to autoencoders, it prevents wasted computation on models that fail to improve reconstruction quality early on.

4.2.5 Model validation

Once a model has been successfully created and trained, it needs to undergo the process of validation. Model validation within the context of the study is defined as the process of determining the degree to which a model is an accurate representation of the real-world system from the perspective of the intended use of the model (Ke *et al.*, 2020). One of the core concepts of model validation comes in the form of loss functions. The different loss functions are discussed within the following subsections.

Loss functions

One of the more important topics within machine learning is the concept of loss functions, as they not only play a role in constructing different machine learning algorithms, but are also imperative to improving their performance (Wang *et al.*, 2020). Hence, the use of inappropriate loss functions within a given implementation affects the effectiveness of an algorithm to some extent. As such, acceptable loss functions or a collection of loss functions must be used.

The specific objective within a neural network is to identify the ideal parameter values, such as the values of weights that minimise the loss value calculated between the expected and the predicted output of a neural network (Sewak *et al.*, 2020). Due to the importance of loss functions to machine learning tasks' functionality, many different functions have been developed and utilised over the years for various machine learning applications. In Table 4-2, a summary of different loss functions used in different tasks is presented.

The most popular loss functions will be discussed in the following subsections. The proposed algorithm also uses these regression loss functions to construct and measure the accuracy of the constructed autoencoders as certain loss functions are particularly suited to evaluating reconstruction quality in autoencoders.

Table 4-2: Different loss functions across machine learning application areas

Applications	Loss function
Regression problems	• Mean Squared Error • Mean Absolute Error • Hubber loss
Classification problems	• Binary cross-entropy • Categorical cross-entropy

Mean squared error

The mean squared error loss function (MSE, Quadratic or L2 loss) is the most commonly implemented generic statistical loss function (Cai *et al.*, 2024). The MSE can be calculated as seen in Equation 4-1:

$$L_{MSE} = MSE = \frac{1}{n}\sum_{i=1}^{n}|x_i - \tilde{x}_i|. \tag{4-1}$$

In Equation 4-1, n indicates the number of observations within the training set, $\tilde{x}_i$ the predicted values, and x_i the observed values.

The MSE is interpreted as the distance between the observed and predicted values. The produced values are always positive and indicate the accuracy of a neural network.

Mean absolute error

The mean absolute error (MAE, L1 loss) is another widely used metric for evaluating different models (Hodson, 2022). The MAE can be calculated as follows, as shown in Equation 4-2:

$$L_{MAE} = MAE = \frac{\sum_{i=1}^{n} |x_i - \tilde{x}_i|}{n}, \tag{4-2}$$

where n indicates the number of data points in the dataset, the predicted values are denoted by $\tilde{x}_i$ and the observed values are denoted by x_i. Although the MAE is more robust against outliers than the MSE, the latter is more used, as it severely penalises larger errors.

Huber loss

The Huber loss is a relatively robust loss function often utilised for various regression-based tasks (Meyer, 2021). It was suggested over half a century ago and is still widely utilised. The Huber loss function is calculated as shown in Equation 4-3:

$$Huber = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{2}(x_i - \tilde{x}_i)^2, \tag{4-3}$$

where n indicates the number of data points in the dataset, the predicted values are denoted by $\tilde{x}_i$ and the observed values are denoted by x_i. The Huber loss is often considered to lie between the functioning of the MSE and the MAE and is relatively robust against outliers.

Binary cross-entropy

Binary cross-entropy (BCE) loss, or log loss, is used for classification problems (Ruby & Yendapalli, 2020). The BCE loss function is expressed as shown in Equation 4-4:

$$BCE = -\frac{1}{n}\sum_{i=1}^{n} x_i log\tilde{x}_i \ + (1 - \tilde{x}_i)\log(1 - \tilde{x}_i), \tag{4-4}$$

where n indicates the number of data points in the dataset, the predicted values are denoted by $\tilde{x}_i$ and the observed values are denoted by x_i.

Categorical cross-entropy

The final loss function discussed is categorical cross-entropy (CCE). CCE is one of the standard and most commonly utilised loss functions in training deep neural network models (Feng *et al.*, 2021; Gowdra *et al.*, 2021). Equation 4-5 shows a commonly used expression for CCE:

$$CCE = \sum_{i=1}^{n} p(x_i) \log q(x_i), \tag{4-5}$$

where n indicates the number of data points in the dataset, $p(x_i)$ indicates the probability of the actual class for sample x_i, $q(x_i)$ represents the predicted probability of the correct class for sample x_i and the summation should be implemented across all samples of x within the dataset.

After the conclusion of the abovementioned sections on the components utilised within an automated machine learning process, the proposed algorithm for automatically constructing an autoencoder will be presented next.

4.3 Proposed automated autoencoder algorithm

The proposed algorithm for automated autoencoder neural network architecture design (AANNAD) was designed with modularity in mind. As such, the different modules of the algorithm, ranging from the autoencoder string generation to the autoencoder training, mutation, and saving, are described in the subsequent subsections.

4.3.1 Standardisation of autoencoder representation

A syntax for presenting autoencoder architectures in terms of a text representation was developed and utilised during the design of the proposed algorithm. This text-based representation is in the format of a chromosomic layout, allowing an evolutionary and mutative approach to be followed to develop and improve the presented autoencoder architectures. An example of the representation is shown in Table 4-3.

Table 4-3: String-based autoencoder representation

784[5]631;552;550;550;520#490=000

The proposed algorithm utilises each part of the chromosome to configure a different part of the autoencoder. Each discerning part is delimited by a specific character to improve the human

readability of the proposed architecture. An overview of each section of the autoencoder chromosome can be seen in Table 4-4.

Table 4-4: Explanation of chromosome contents

Name	Purpose	Character or symbol denominator	Example in regard to Table 4-3
Input and output layer size	This section of the chromosome denotes the input and output vector sizes.	This information is denoted before the first square open bracket: [	The input size within the above example is 784.
Number of hidden layers	Denotes the number of hidden layers within the encoder and decoder components of the autoencoder, respectively. Both encoder and decoder components are mirror images of one another.	This information is denoted between two square brackets: [ ]	The number of hidden layers within the above example is five, meaning that the encoder will consist of five hidden layers, and the decoder will consist of five. This number does not include the input layer, the output layer or the bottleneck.

Table 4-4: Explanation of chromosome contents (continued)

Name	Purpose	Character or symbol denominator	Example in regard to Table 4-3
Hidden layer configuration	The hidden layer configuration denotes the number of nodes inside each hidden layer. For example, given the hidden layer structure X; Y; Z, the encoder component consists of X nodes in the first hidden layer, Y nodes in the second layer, and Z nodes in the third hidden layer. The decoder component then starts off with Z nodes in the first layer, Y nodes in the second layer, and X nodes in the third layer.	This information is located between the closed square bracket] and the hash symbol #. Each configuration of the hidden layers is separated by a semicolon symbol ;	In relation to the above example, the layout of the encoder and decoder sub-architectures are as follows: 631-552-550-550-520-Bottleneck-520-550-550-552-631
Bottleneck Configuration	The bottleneck configuration denotes the number of nodes in the bottleneck layer of the autoencoder.	The bottleneck configuration can be found between the # symbol and the = sign.	The number of nodes within the above example is 490.
A placeholder for future information		The = symbol denotes this information at the end of the chromosome string.	

With the chromosome notation of the proposed algorithm discussed, the following subsections are utilised to describe the functionality of the algorithm in terms of pseudocode.

4.3.2 Autoencoder chromosome generation

Initially, the algorithm loads the required dataset for the problem with which it is presented. This dataset is then pre-processed and split into three different subsets of datasets for subsequent modelling. More details regarding this process are discussed in Chapter 5.

Initial autoencoder structure generation

After the preparation mentioned above and pre-processing have concluded, the algorithm proceeds with the first part of its core function, which is to randomly generate a collection of potential autoencoder architecture chromosomes, based on the dimensions of the presented dataset. This process is presented in terms of pseudocode as follows:

Module 1: Initial Autoencoder Chromosome Structure Creation

Input: None

Output: None

```
1  while the length of array_autoencoder_architectures not equal to number_of_initial_architectures do
2      random_autoencoder_string ← input size of the dataset as a string, plus an opening bracket "["
3      num_encoder_hidden_layers ← random value between 3 and 10
4      add num_encoder_hidden_layers to random_autoencoder_string and append "]"
5      bottleneck_dimensions ← random value between 1 and the input size of the dataset
6      changing_limitation ← input size
7      for k in range (num_encoder_hidden_layers) do
8          changing_limitation ← random value between bottleneck_dimensions and changing_limitation
9          add changing_limitation to random_autoencoder_string
10         if k less than num_encoder_hidden_layers – 1 do
11             add special character denominator to random_autoencoder_string
```

Module 1: Initial Autoencoder Chromosome Structure Creation (continued)

```
12    add "#" and bottleneck_dimensions followed by "=000" to random_autoencoder_string
13    if random_autoencoder_string not in array_autoencoder_architectures do
14        append random_autoencoder_string to array_autoencoder_architectures
15  print array_autoencoder_architectures
```

The pseudocode presented in Module 1 is executed to produce ten initial autoencoder chromosomes. Each chromosome is tested to ensure its uniqueness and is designed to consist of completely randomised components.

Compiling and training of the first set of autoencoders

After the initial autoencoder chromosomes have been produced, they are compiled into complete autoencoder architectures and trained. This process is represented in pseudocode as follows:

Module 2: Initial compiling and training of autoencoders

Input: None

Output: autoencoder_Info, history_Info, **and** loss

```
1   loss ← an empty array
2   autoencoder_Info ← an empty string
3   history_Info ← an empty string
4   for k in range (len(array_autoencoder_strings)) do
5       early_stopping_callback ← an instance of EarlyStopping
6       autoencoder ← create_autoencoder(array_autoencoder_strings[k])
7       optimizer_adam ← an instance of Adam optimiser with learning rate 1e-6
8       compile the autoencoder with optimiser_adam and mean_squared_error as the loss
        function
```

Module 2: Initial compiling and training of autoencoders (continued)

9	autoencoder.metrics ← ["mean_squared_error", "mean_absolute_percentage_error"]
10	**print** a summary of the autoencoder
11	**train** the autoencoder using training data, with callbacks including early_stopping_callback
12	**append** the loss list with the loss of this architecture
13	autoencoder_Info ← current autoencoder
14	history_Info ← current history
15	**save** autoencoder
16	**print** "Saving autoencoder: " concatenated with array_autoencoder_strings[k] + ".h5"
17	**return** autoencoder_Info, history_Info **and** loss

With the completion of Module 2, the user is presented with a collection of ten autoencoder architectures that have been trained, validated and tested. The presented architectures are also saved to allow the user to reload any of the architectures with their applicable weights intact for additional testing or implementation. These first ten autoencoder chromosomes are saved in an array named *array_tested_autoencoders* to ensure that a previously tested autoencoder would not be trained and tested again. Additionally, these initial ten autoencoders make up the initial values of the *array_current_best_autoencoders* array.

Selection and mutation of presented autoencoder strings

With the initial construction of autoencoder chromosomes and training of autoencoder architectures complete, the presented algorithm then randomly selects different chromosomes to mutate and architectures to train. This process is represented in pseudocode as follows:

Module 3: Post-training mutation selection

Input: None

Output: None

```
for k in range (100) do:
    mutation_candidate ← randomly selected chromosome from array_current_best_autoencoders
    print mutation_candidate
    mutation_candidate ← mutate_autoencoder_string(mutation_candidate)
    if mutation_candidate not in array_tested_autoencoders do
        append array_tested_autoencoders with mutation_candidate
        print mutation_candidate
        autoencoder, history, mutation_performance ← compile_mutated_autoencoder_models(mutation_candidate)
        print mutation_performance
        if the performance of the least well-performing autoencoder in array_current_best_autoencoders < mutation_performance do
            delete array_current_best_autoencoders[-1]
            append array_current_best_autoencoders with mutation_performance
    sort array_current_best_autoencoders by performance
    print array_current_best_autoencoders
```

The pseudocode presented in Module 3 mutates 100 different autoencoders before presenting the user with the final list of best autoencoder architectures and chromosomes. This final list size may be altered to produce as many or as few autoencoders as required.

Mutation of presented autoencoder strings

Within the previous module, a method to mutate an autoencoder chromosome had been implemented, called *mutate_autoencoder_string*. This method alters and mutates existing autoencoder chromosomes within the array, containing the selection of the best autoencoder architectures discovered during the algorithm's lifecycle. This process may be represented in pseudocode as follows:

Module 4: Mutate autoencoder chromosomes

Input: received_autoencoder_string

Output: mutated_string

1 autoencoder_array ← received_autoencoder_string dissected into its different chromosome sections

2 autoencoder_array[-1] ← 0 (Resetting the loss value)

3 input_size ← autoencoder_array[0]

4 encoder_decoder_layers ← autoencoder_array[1]

5 mutated_string ← received_autoencoder_string

6 mutation ← Random value between 1 and 3

7 **if** mutation = 1 **do**

8 | **increase** the number of hidden layers within the chromosome

9 | **update** mutated_string

10 **if** mutation = 2 **do**

11 | **decrease** the number of hidden layers within the chromosome, ensuring at least one hidden layer remains

12 | **update** mutated_string

13 **if** mutation = 3 **do**

14 | **decrease** the number of nodes within the bottleneck of the chromosome

Module 4: Mutate autoencoder chromosomes (continued)

```
15      update mutated_string

16  return mutated_string
```

As seen in the above pseudocode, one of three random mutations may occur to alter a chromosome: increasing the number of hidden layers, decreasing the number of hidden layers, and finally decreasing the size of the bottleneck layer. With every mutation completed, an array is tested to ensure that the specific autoencoder chromosome has not yet been generated to promote program efficiency.

Efficient and adaptive design of autoencoder architectures for resource-constrained environments

The AANNAD algorithm was designed to function in a time- and resource-limited environment. To manage the complexity of the generated architectures, the search space for possible architectures has been limited to a maximum of 10 encoder and decoder layers. However, the algorithm is designed to allow this number to be increased or decreased, based on the problem's characteristics. The generated architecture can also assign dropout layers following each encoder layer, with a dropout rate of 0.15. This helps prevent the generated autoencoders from overfitting the dataset. The algorithm randomly determines whether an architecture is created with or without dropout layers.

4.4 Different partitions of the utilised datasets

During the lifecycle of each experiment, the provided dataset that the algorithm utilises is divided into three different partitions: a training subset, a validation subset, and a testing subset. This methodology is often referred to as the holdout method, a widely utilised methodology within machine learning (Xiong *et al.*, 2020). More specific information regarding each experiment's data division is discussed in Chapter 5. The training subset is used to train (fit) the autoencoder model to reproduce the data. The validation subset provides an unbiased evaluation of a model's performance during training on the training subset. However, this evaluation becomes more biased as the model's knowledge gained from the validation subset starts to become more integrated into its configuration. The testing dataset provides an unbiased evaluation of the final version of an autoencoder model. It is only used once after the model has been completely trained and is used in all of the experiments within this study to compare different autoencoder architectures.

4.5 Summary

Within Chapter 4, the automated design of neural network architectures was discussed, focusing on how manual design challenges can be overcome through automation, particularly by utilising AutoML. The chapter introduced the key components of AutoML, such as data preparation, feature engineering, model generation, and evaluation and validation. These steps are essential for building effective machine learning models.

The chapter was then used to explore concepts, such as data preparation, with more intricate information being provided regarding data collection, augmentation, and cleaning to ensure the data is ready for model training. This was followed by an in-depth review of feature engineering, which included techniques for feature selection, extraction, and construction, all aimed at optimising model performance.

Model generation was covered next, and methods for optimising neural network architectures and hyperparameters, including search spaces and optimisation techniques, were discussed. Model evaluation was further expanded upon, with techniques, such as low-fidelity optimisation, early stopping, and weight sharing further explained.

Additionally, within this chapter, the proposed AANNAD algorithm was presented. The algorithm utilises evolutionary strategies, such as mutation and selection, to generate and refine autoencoder architectures. The process of creating these models, from autoencoder chromosome generation to compiling, training, and evaluating them, has been described in detail within this chapter. Lastly, the division of datasets into training, validation, and testing subsets and the importance of dataset partitions in the algorithm’s performance were addressed.

CHAPTER 5 EXPERIMENTAL DESIGN

5.1 Introduction

Experimentation is an important component when making stronger claims regarding generalisations, as it proves a concept through the replication of findings (Greenhill *et al.*, 2020). Experimentation is a fundamental concept in scientific and engineering practices. A well-designed set of experiments is meant to yield an empirical model of a process, which facilitates understanding and prediction of its behaviour. Thus, experimentation forms an integral part of any scientific research.

In the previous chapters, a contextual background and overview of concepts and techniques relating to the novel algorithm presented in Chapter 4 were provided. The purpose of Chapter 5 is to describe the experimental design and a series of experiments that investigate the performance and generalisability of various autoencoder configurations created by the novel algorithm when applied to different datasets.

The remainder of the chapter is structured as follows. The general setup of the experimental process, accompanied by a brief explanation of the three core concepts associated with the experimental design of the proposed algorithm, is presented in Section 5.2. The subsequent sections are composed of more specific details on the datasets used within each experiment and the results of these experiments. More specifically, in Sections 5.3, 5.4 and 5.5, the selection, implementation, and reasoning behind utilising different datasets and information regarding applying the Automated Autoencoder Neural Network Architecture Design (AANNAD) algorithm when modelling these datasets are explained. Section 5.6 concludes the chapter with a summary of the contents discussed.

5.2 Overview of experimental design

The experiments conducted within this study were designed to assess the performance of the proposed AANNAD algorithm by testing its capabilities to produce accurate autoencoder models for different datasets. By investigating this algorithm, valuable insights into the practical implementation of autoencoders and the design of autoencoder architectures may be gained. The experimental design of the conducted experiments comprises multiple aspects, including software and hardware specifications, data acquisition, and finally, data pre-processing. These facets are discussed in a broader overview, with more in-depth discussions relating to each of the experiments presented in the following sections.

5.2.1 Software and hardware utilised in the experiments

The Google Collaboratory platform (Google Colab) (Google, 2023) has been utilised to implement the AANNAD algorithm. Google Colab is a hosted Jupyter notebook service designed for machine learning applications. The proposed algorithm was implemented using the Python programming language (version 3.10.12) with the Keras API (version 3.5.0) and TensorFlow deep learning framework (version 2.17.1). An implementation of the algorithm can be found in Appendix A.

All development and experiments had been completed on the same hardware. When creating a Google Colab instance, a virtual machine is provided, which consists of an Intel Xeon CPU with two vCPUs (virtual CPUs) and 13GB of RAM. However, Colab additionally offers access to GPU services for machine learning models. The default GPU for Google Colab is an NVIDIA Tesla K80 with 12GB of VRAM. After numerous tests, it was decided that the GPU services had delivered more favourable results than the CPU-based option. As such, the GPU services were utilised throughout the experimentations of the study.

5.2.2 Data acquisition

Across all experiments, data was acquired utilising publicly available repositories. These datasets were selected explicitly due to corresponding research that utilised each. This was a crucial factor in determining which datasets to use, as the results of the proposed algorithm were compared to the results and performance of the prior experiments conducted on each dataset.

5.2.3 Data pre-processing

Data may be subject to corruption, noise, inconsistency and missing values (Maharana *et al.*, 2022). Poor quality data can drastically affect the accuracy of an autoencoder and may lead to false predictions. As a result, ensuring that a dataset is of the highest quality is imperative. One of the most utilised methodologies for securing higher-quality data is called data pre-processing. This methodology improves the ease with which knowledge may be extracted from a dataset by implementing methods, such as cleaning, transformation, integration and reduction. Specific data pre-processing performed within each of the experiments is discussed within their respective subsections following Section 5.2. The following sections contain more in-depth explanations and discussions regarding the experiments conducted with the AANNAD algorithm when applied to specific datasets.

5.3 MNIST experiments

The first experiment used the Modified National Institute of Standards and Technology (MNIST) dataset (Deng, 2012). The observations made and knowledge gained from the experiments

performed on the MNIST dataset are discussed in this section. This compares the results obtained from the experimentation and a previously conducted study.

The MNIST dataset is described in Section 5.3.1. In Section 5.3.2, the pre-processing performed on the dataset is addressed. In Section 5.3.3, prior research conducted on the dataset is considered. The process of selecting the best-performing autoencoder architecture is described in Section 5.3.4. The best-performing autoencoder architecture found by the AANNAD algorithm is discussed in Section 5.3.5. In Section 5.3.6, experimental results are considered. Finally, the section is concluded in Section 5.3.7.

5.3.1 Dataset description

The MNIST dataset is a well-known dataset that consists of a collection of handwritten digit images and is often extensively utilised in optical character recognition research, as well as machine learning research (Deng, 2012). The dataset had initially been proposed, and based on the freely available nature of the dataset, it had become a standard when testing machine learning algorithms. The dataset consists of 70000 data points, typically split in a ratio of 60000 data points for training and 10000 for testing purposes. The dataset comprises two different data sources: the NIST Special Dataset 1 (collected from high-school students) and the NIST Special Dataset 3 (collected from Census Bureau employees).

5.3.2 Data pre-processing of the MNIST dataset

The MNIST dataset contains images that consist of 28x28 pixels. The input and output layers utilised within the study comprised 784 nodes, one for each pixel. For the AANNAD algorithm experiments that utilised the MNIST dataset, a 78.57% (training), 7.14% (validation), and 14.29% (test) split was performed. The testing dataset was utilised as is for the experiment, with the training dataset being split into training and validation datasets.

The images in the MNIST dataset were initially presented as greyscale, with the values of the pixels within each image that can range between 0 (black) to 255 (white) inclusively. A graphical representation of the different values of a greyscale can be seen in Figure 5-1.

To avoid larger gradient values that may affect the training of the autoencoders, the value of every pixel was divided by 255, effectively transforming the value of each pixel to range between 0 and 1 inclusively.

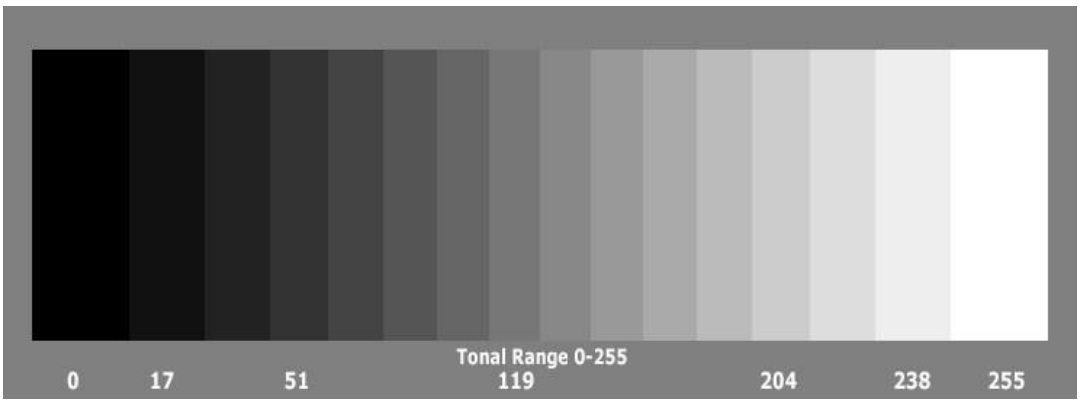

Figure 5-1: MNIST data greyscale range

5.3.3 Prior research conducted with the MNIST dataset

The research conducted by Charte *et al.* (2019) was utilised as a comparative baseline, as it used a similar chromosomal data structure to what is utilised within the presented study. Evolutionary approaches to automating the configuration of autoencoder architectures have been developed and utilised. These approaches include a genetic algorithm in which a population of configurations was programmed to evolve through a crossover operator to produce new architectures and an evolutionary strategy that allowed only the best-performing architectures to produce other architectures exclusively through mutation.

The presented AANNAD algorithm shares some concepts with the algorithm proposed by Charte *et al.* (2019). Both algorithms use an evolutionary approach to develop architectures further, and both algorithms use a chromosomal structure when describing the composition of the presented autoencoder architectures. The chromosomal structure utilised in the evolutionary algorithm presented by Charte *et al.* (2019) may consist of up to 14 genes which describe the different components of autoencoder architectures. These genes are shown in Figure 5-2.

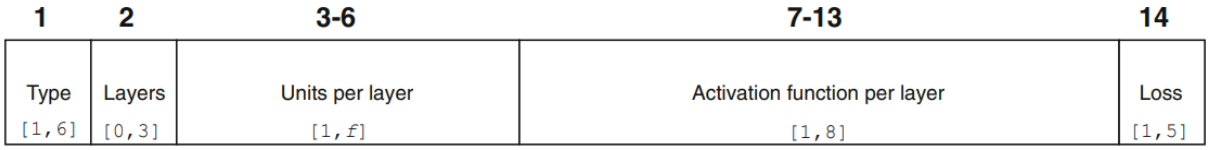

Figure 5-2: Chromosomal structure used by Charte *et al.* (2019)

The notation depicted in the above figure is further discussed in Table 5-1.

Table 5-1: The purpose of each gene as utilised by Charte *et al.* (2019)

Name	Purpose	Values
Type	Determines the type of autoencoder to implement.	1. Basic 2. Denoising 3. Contractive 4. Robust 5. Sparse 6. Variational
Layers	Determines the number of additional layers in the encoder and decoder.	(0) Only a coding layer, (1–3) Additional layers in both coder and decoder
Units	Determines the number of units per layer, with f being the number of features within the dataset.	The first integer (gene 3) configures the number of units in the outer layer, while the last one (gene 6) sets the coding length.
Activation function per layer	Determines the activation function to be implemented within each layer, both for the encoder and decoder.	1. *Linear* 2. *Sigmoid* 3. *Tanh* 4. *Relu* 5. *Selu* 6. *Elu* 7. *Softplus* 8. *Softsig*n
Loss	Determines the loss function used to evaluate the autoencoders during fitting.	1. Mean squared error 2. Mean absolute error 3. Mean absolute percentage error 4. Binary cross-entropy 5. Cosine proximity

Through the abovementioned research, a specific error metric has been utilised to determine the accuracy of each architecture. The mean square percentage error (MSPE) performance measurement used by Charte *et al.* (2019) was also utilised within the AANNAD experiments in order to compare results. The MSPE may be calculated as shown in Equation 5-1:

$$MSPE = \frac{\sum_{j=1}^{N}\left(\frac{A_j - P_j}{A_j}\right)^2}{N} \quad (5\text{-}1)$$

where N represents the number of data points within the dataset, A_j represents the actual value of a data point j contained within the dataset, and P_j represents the predicted value of the data point j generated by the autoencoder (Fomby, 2008).

5.3.4 Selection of the best-performing architecture

The selection process for the AANNAD algorithm takes place continuously during the algorithm's execution. At the same time, statistics are recorded, such as the number of training epochs per architecture and architecture performance in various performance metrics, such as the MSE and MSPE.

As soon as an autoencoder architecture has been trained and validated, its performance is compared with that of its predecessors, and it is ranked, with the best-performing architectures appearing at the top of the list and the less-well-performing architectures appearing at the bottom. Within the MNIST dataset experiments, the ranking of each architecture was determined by the MSPE rate, as described in the previous section.

5.3.5 Description of the best-performing architecture

The AANNAD algorithm generated very accurate autoencoder architectures within four hours. During that time, 30 different architectures were examined and compared, with the best-performing architecture reaching an MSPE rate of 0.2447 on the validation set. This information is shown in Table 5-2.

Table 5-2: Time frame of different algorithms trained on the MNIST dataset

Approach utilised	Running time of strategy	Configurations tested	Best MSPE achieved
AANNAD algorithm	4 hours	30	0.2447

The best-performing architecture is shown in Figure 5-3. The architecture in the above figure comprises an input layer and an output layer, each with 784 nodes. There are four hidden layers within the encoder module $(EH_1, EH_2, \cdots, EH_4)$ and four hidden layers within the decoder module $(DH_1, DH_2, \cdots, DH_4)$. The composition of the hidden layers is 587, 546, 335 and 279 nodes, respectively. A bottleneck is also present within the architecture which consists of 217 hidden nodes (BN). A graph depicting the training of the above architecture in terms of the MSE loss over the course of 140 epochs is shown in Figure 5-4.

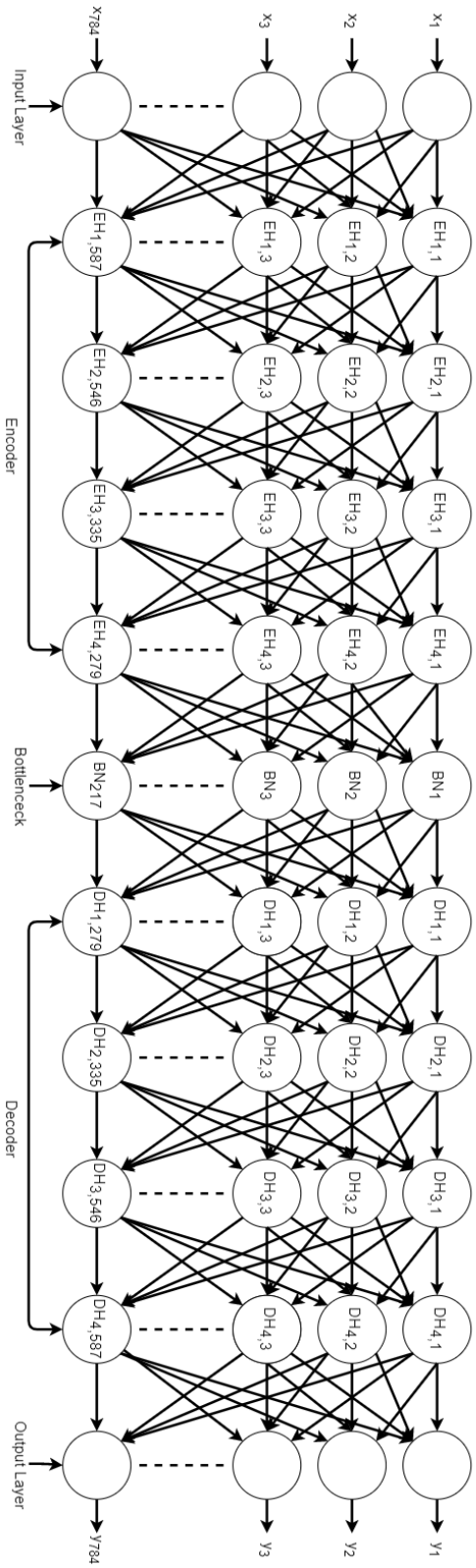

Figure 5-3: Best-performing autoencoder found

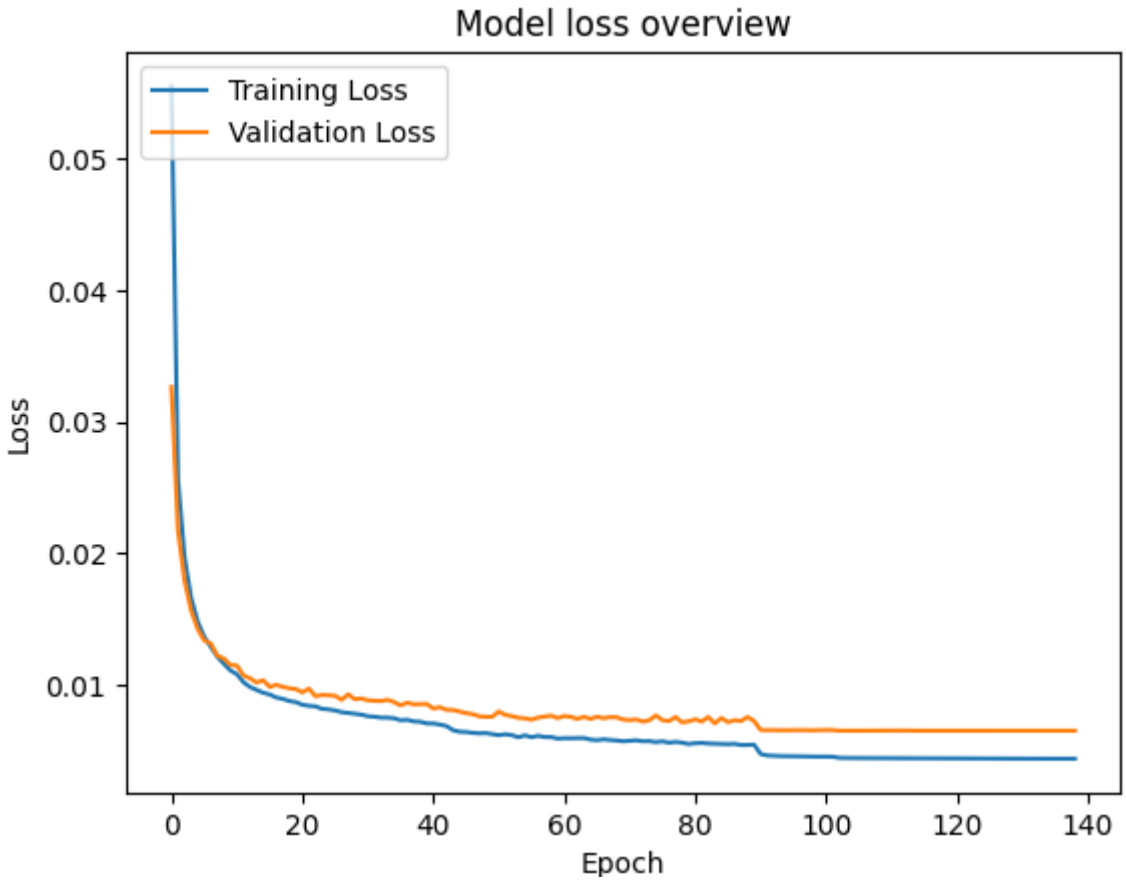

Figure 5-4: Training and validation loss of the architecture

As seen in the above figure, overfitting did not occur during the best-performing architecture training, which stopped when a plateau was reached. It should also be noted that the best-performing architecture found by the AANNAD algorithm performed better when constructed without dropout layers. When constructed without dropout layers, an MSPE of 0.2447 was achieved; however, when constructed with dropout layers between each encoder hidden layer, an MSPE of 1.5414 was achieved. Within the following section, the results of the best-performing AANNAD algorithm-produced architecture compared to the baseline study conducted by Charte *et al.* (2019) are discussed.

5.3.6 Discussion of experimental results

The best autoencoder model found by Charte *et al.* (2019) had an MSPE rate of 0.560. The AANNAD algorithm produced a number of autoencoder architectures that outperformed the best architecture discovered by Charte *et al.* (2019). A comparison between the AANNAD algorithm and the evolution strategy is presented in Table 5-3.

Table 5-3: Comparisons of different algorithms based on the MNIST dataset

Approach utilised	Running time of strategy	Configurations tested	Best MSPE achieved
Charte *et al.* (2019) evolution strategy	4 h 40 m	4 001	0.560
AANNAD algorithm	4 h	30	0.2447

As seen in the above table, the AANNAD algorithm produced fewer autoencoder architectures when compared to that of the evolution strategy utilised by Charte *et al.* (2019). However, despite producing far fewer architectures than its evolution strategy, the AANNAD algorithm produced a better-performing architecture based on MSPE.

A random selection of 10 MNIST digits from the testing subset and their corresponding reconstruction (prediction) by the AANNAD algorithm is displayed in Figure 5-5.

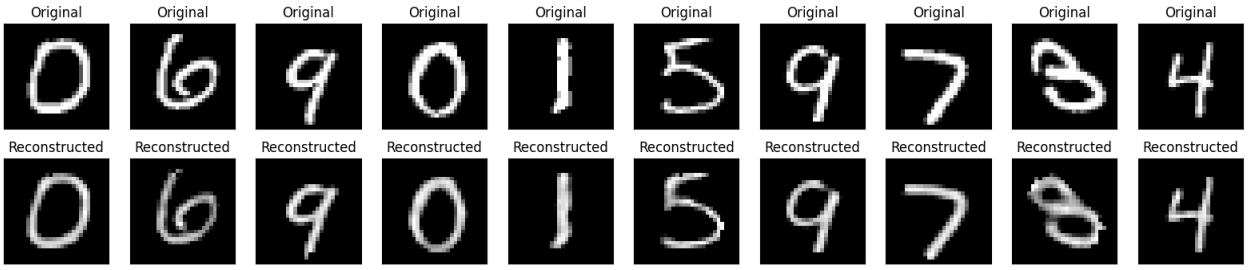

Figure 5-5: Reconstructed images of AANNAD

Figure 5-5 is a graphical representation of the AANNAD algorithm's reconstructive capabilities. Based on the quantifiable results discussed in the previous subsections, it produces recognisable reconstructions of the images contained within the MNIST dataset. The following subsection concludes the experiments conducted on the MNIST dataset.

5.3.7 Conclusion

Based on the presented results, the AANNAD algorithm outperformed the best autoencoder model Charte *et al.* (2019) found on the MNIST dataset. Within the following section, an experiment utilising a second dataset is presented.

5.4 SOCOFing experiments

Within recent years, biometric identification systems have begun seeing a sharp increase in utilisation, with fingerprint recognition systems being applied widely to adopt accurate and reliable

biometric identification between individuals. However, in many systems that implement deep learning models, problems are encountered with overlapping patterns and poor-quality images (Saponara *et al.*, 2021). This means that systems capable of recreating biometric images, specifically fingerprints, from concepts, such as overlapping patterns and poor original image quality may benefit various systems concerned with biometric information.

The second set of experiments conducted utilised a dataset constructed from the Sokoto Coventry Fingerprint (SOCOFing) dataset (Saponara *et al.*, 2021). A description of the SOCOFing dataset is provided in Section 5.4.1, and details of the pre-processing methods employed appear in Section 5.4.2. Previous investigations conducted on the SOCOFing dataset are summarised in Section 5.4.3. The methodology used to identify the most suitable autoencoder architecture is explained in Section 5.4.4, and information about the best architecture identified by the AANNAD algorithm is presented in Section 5.4.5. The experimental findings are reported in Section 5.4.6 and concluding remarks in Section 5.4.7.

5.4.1 Dataset description

The SOCOFing dataset is a biometric fingerprint database that had originally been designed for academic research purposes (Shehu *et al.*, 2018). The dataset is composed of 6000 different fingerprint images taken from 600 different African research subjects. There are ten fingerprints per subject; all subjects are 18 or older. All original images were acquired, based on impressions collected with Hamster Plus (HSDU03PTM) and SecuGen (SDU03P) sensor scanners. Additionally, the SOCOFing dataset contains unique attributes, such as labels for gender, different hand and finger names, and purposefully altered versions of each of the 6000 fingerprint images, which comprised three different levels of alteration intensity. The dataset is freely available for non-commercial research purposes.

5.4.2 Data pre-processing of the SOCOFing dataset

The images contained within the SOCOFing dataset are composed of 96x103 pixels. As such, it would require input and output layers comprising 9888 nodes to process the entirety of the images, as the AANNAD system would focus on developing autoencoder architectures that follow a mirrored encoder-decoder shape. An example of a subset of images from the SOCOFing dataset is shown in Figure 5-6.

Figure 5-6: A sample of five left-handed fingerprints from the SOCOFing dataset

Unfortunately, due to resource constraints, experiments attempting to reconstruct entire images were too computationally expensive. Consequently, alternative methods of processing and reconstructing the biometric images have been considered. The chosen method includes determining the accuracy of the AANNAD algorithm by utilising smaller subsets of the images to train and test the algorithm. Based on the research done in the comparative study conducted by Saponara *et al.* (2021), two different subsets were generated in different formats. One subset consisted of 50x50 pixel images, while another comprised 10x10 pixel images. The different pixel sub-partitions of the image dataset are discussed in Section 5.4.3.

Within the SOCOFing dataset experiments, a 70% (training), 20% (validation) and 10% (test) random split was performed. Like the images within the first experiment, the images contained within the SOCOFing dataset were initially presented as greyscale, with the values of the pixels within each image that could range between 0 (black) to 255 (white), inclusively. To avoid larger gradient values that may affect the training of the autoencoders, the value of every pixel was divided by a value of 255, effectively transforming the value of each pixel to range between 0 and 1, inclusively.

5.4.3 Prior research conducted with the SOCOFing dataset

The research that had been conducted by Saponara *et al.* (2021) utilised different neural network architectures to recreate a graphical representation of various fingerprint-based datasets. An optimised convolutional neural network and an optimised sparse autoencoder achieved the most promising results obtained during their experiments. The sparse autoencoder would be utilised for comparison, since the autoencoder architecture forms the basis of this study.

The presented AANNAD experiment with the SOCOFing dataset follows methodologies implemented in the research conducted by Saponara *et al.* (2021). During the pre-processing steps of the experiments, the larger biometric images that consist of 96x103 pixels are normalised by transforming them into 100x100 pixel images. These 100x100 pixels are then divided into subsets of images that consist of different pixel fragments of the original image. Some examples of the fragmentation structures implemented include four images of 50x50 pixels, 16 images of

25x25 pixels, 25 images of 20x20 pixels and 100 images of 10x10 pixels. Two of these sub-partitions had been utilised in the experiments of the AANNAD algorithm, namely the 50x50 and the 10x10 pixel partitions. These smaller subsets are employed during the training and validation of the different autoencoder architectures. The architectures are constructed to recreate the smaller pixel images, re-stitching them together to determine the accuracy of the autoencoder architecture.

5.4.4 Selection of the best-performing architecture

As with the previous experiment, metrics of each architecture are continuously recorded during the execution of the AANNAD algorithm to determine the most accurate architecture for the given dataset. These metrics include the number of epochs taken to train, learning rate, mean square error and training and validation loss throughout training.

The testing set will be subjected to recreation as soon as an autoencoder architecture has been trained and validated on each of the smaller 10x10 or 50x50 pixel sub-images. To determine the accuracy of the model, both the original 10x10 and 50x50 pixel images and their recreated counterparts produced by the AANNAD algorithm are re-stitched together. This process results in the re-creation of the original 100x100 pixel images and the 100x100 pixel images generated by the autoencoder architecture. The difference between these images is then measured in terms of MSE, recorded, and utilised to determine the best architecture by comparing the MSE scores of the recreated images of all architectures.

5.4.5 Description of the best-performing architecture

When applied to the SOCOFing dataset, utilising the 50x50 pixel and the 10x10 pixel partitioning methodologies, the AANNAD algorithm generated architectures comparable to that of the study conducted by Saponara *et al.* (2021), even presenting architectures with better results regarding the MSE metric during its 12-hour running period. During that time, numerous architectures were examined and compared, with the best-performing architectures achieving an MSE rate of 0.05387 and 0.01885, respectively, on the testing set. These results are shown in Table 5-4.

The architecture implemented with the 50x50 pixel sub-partitioning experiments comprised an input layer and an output layer, each with 2500 nodes. The encoder module consists of two hidden layers (EH_1, EH_2) and another two hidden layers within the decoder module (DH_1, DH_2). The composition of the hidden layers is 2453 and 2400 nodes, respectively. A bottleneck is also present within the architecture, which consists of 2357 hidden nodes (BN). To prevent overfitting within this experiment, dropout layers were inserted between each input layer, with a dropout rate of 15%. A graph depicting the training of this architecture in terms of the MSE loss for 451 epochs is shown in Figure 5-7.

Table 5-4: Performance results of different architectures trained on the SOCOFing dataset with different image pixel subsets

Approach utilised	Running time of strategy	Number of architectures tested	Methodology utilised	Best MSE achieved
AANNAD algorithm	12 hours	10	50x50 pixel partitioning	0.05861
AANNAD algorithm	12 hours	6	10x10 pixel partitioning	0.01885

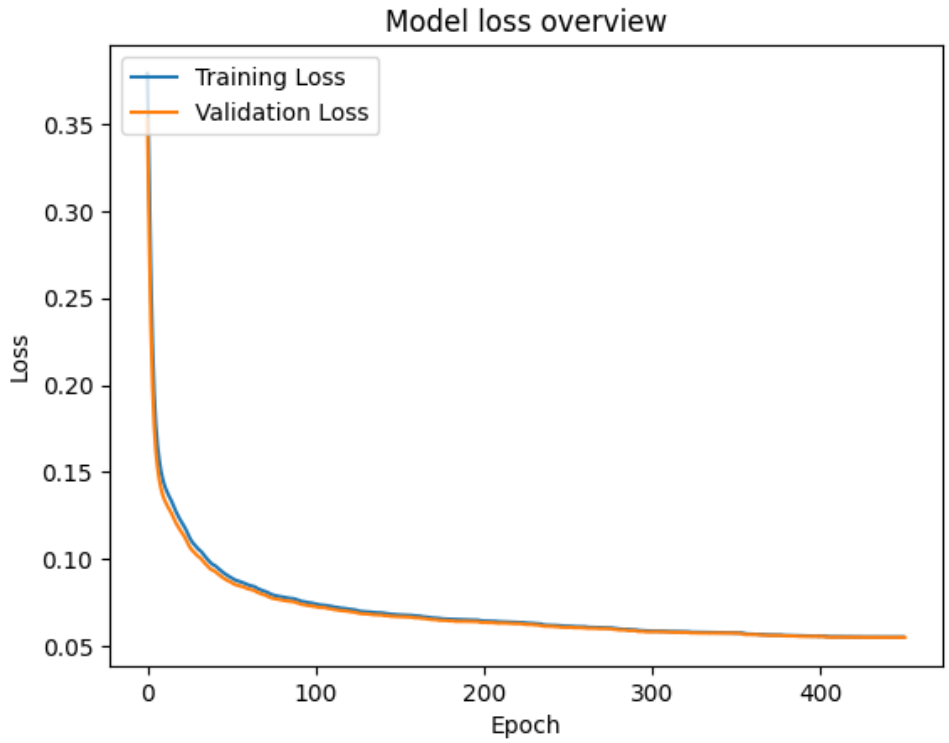

Figure 5-7: Training and validation loss of architecture implementing the 50X50 partitioning images

The best-performing architecture's training and validation loss development regarding the 50x50 image partition is shown in Figure 5-7. The validation loss, represented by the orange line, and the training loss, represented by the blue line, decrease throughout 451 epochs. Nearing epoch 100, a plateau forms and increases in intensity as training proceeds. This indicates that the point where further training would induce overfitting is being reached. This point is reached at epoch 451 when the AANNAD algorithm ceased further training of this architecture.

A graphical representation of the architecture implemented in the 50x50 pixel partitioning experiments of the SOCOFing dataset may be seen in Figure 5-8.

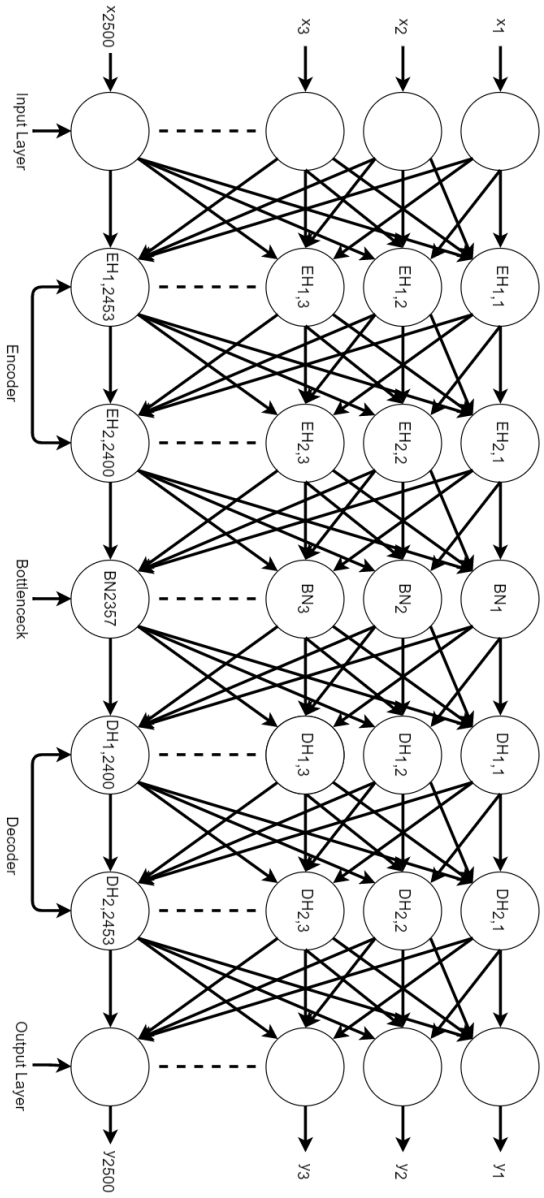

Figure 5-8: Best-performing autoencoder structure for the 50x50 partitioning methodology

The architecture implemented with the 10x10 pixel sub-partitioning experiment had been implemented with input and output layers consisting of 100 nodes each. The encoder module of this experiment consisted of a single hidden layer (EH_1) and a single hidden layer within the decoder module (DH_1). These hidden layers were composed of 80 nodes each. A bottleneck is also present within the architecture, which consists of 70 hidden nodes (BN). A graph depicting the training of the 10x10 pixel architecture in terms of the MSE loss throughout 1 532 epochs is shown in Figure 5-9. In contrast to the experiments conducted with the 50x50 pixel sub-partitioning, this experiment did not utilise any dropout layers.

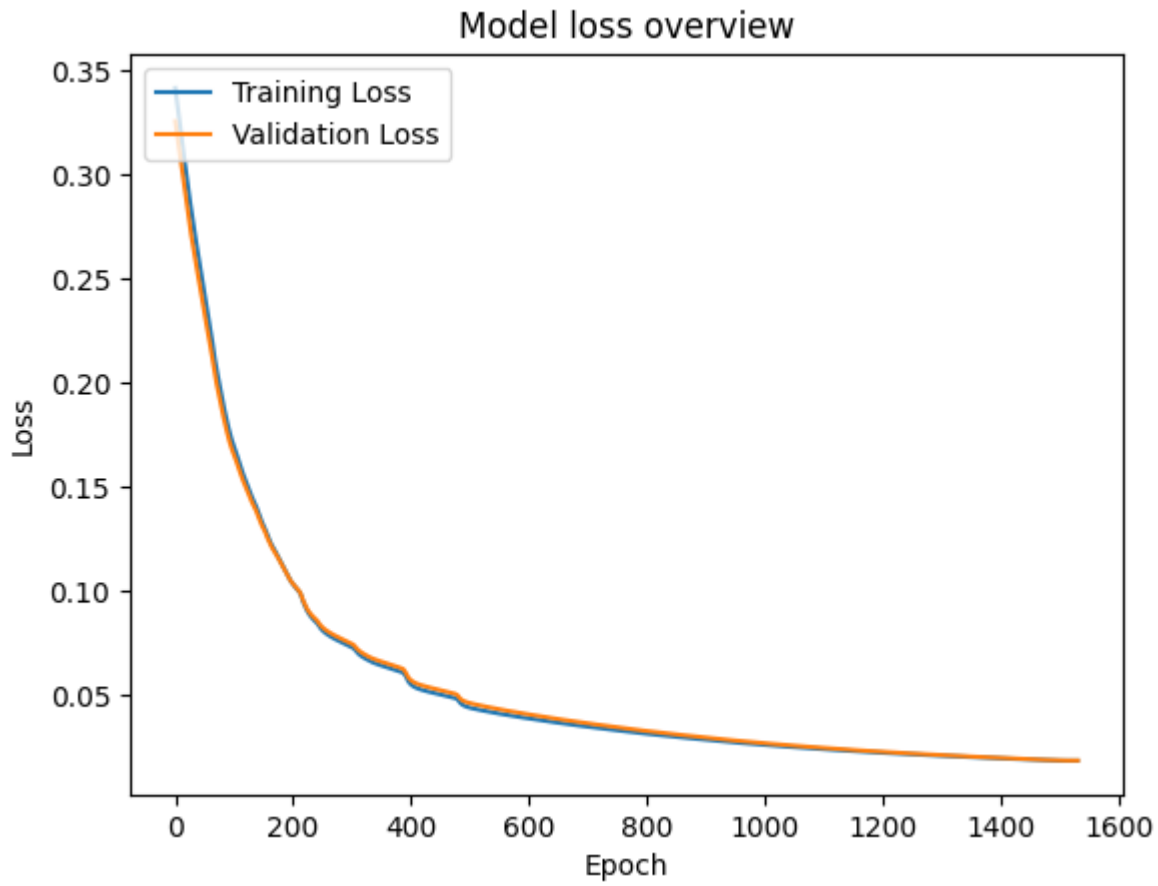

Figure 5-9: Training and validation loss of architecture implementing the 10x10 partitioning

As seen in Figure 5-9, the validation loss, represented by the orange line, and the training loss, represented by the blue line, decrease for the 1532 epochs it had taken to train this specific autoencoder architecture. Around epoch 600, both losses start to plateau and continue to increasingly plateau as the epochs continue. This indicates that the architecture is nearing the point where further training would induce overfitting. This point would be reached at epoch 1532 when the AANNAD algorithm had ceased further training of this architecture. A graphical representation of the architecture generated during the experiments concerning the 10x10 pixel partitioning subsets can be seen in Figure 5-10.

In the following section, the results of the best-performing architectures are compared to the baseline study conducted by Saponara *et al.* (2021).

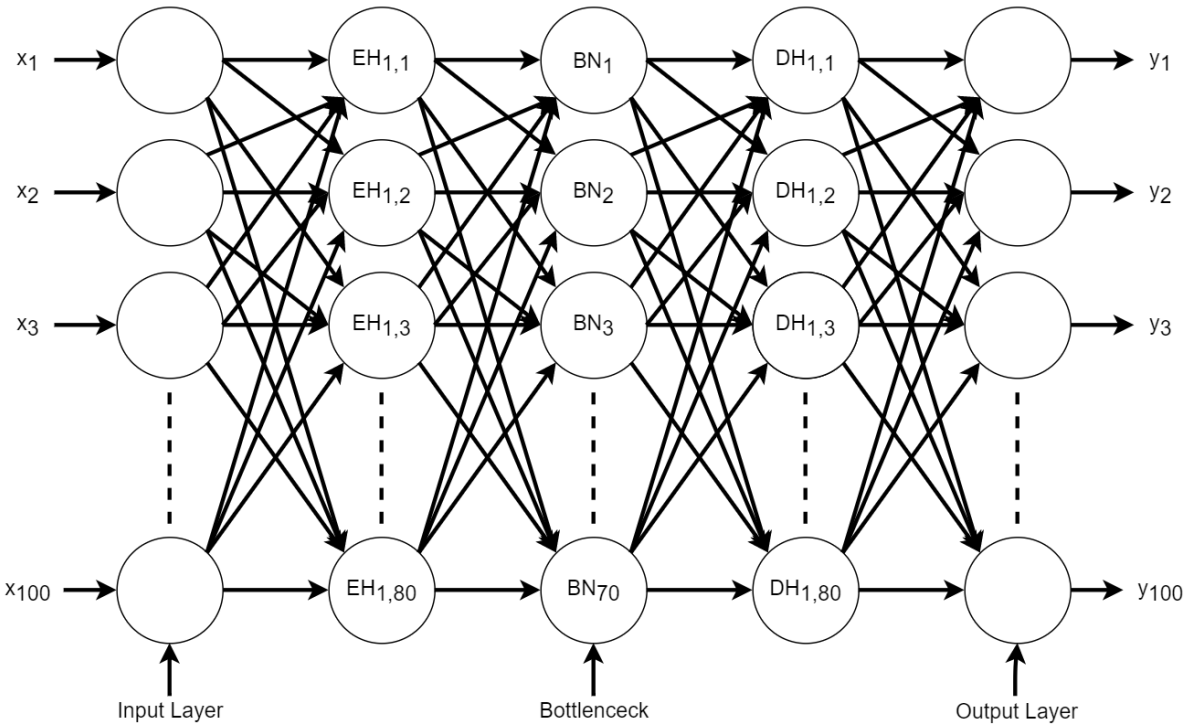

Figure 5-10: Best-performing autoencoder structure for the 10x10 partitioning methodology

5.4.6 Discussion of experimental results

The sparse autoencoder model that was implemented in the research conducted by Saponara *et al.* (2021) consisted of an input layer, an encoder, a latent representation (hidden units), a decoder and an output layer (decoder). The latent representation contained 50 neurons. This architecture had been allowed to train for 1000 epochs and implemented a scarcity regularisation factor of 0.05 (Saponara *et al.*, 2021). This sparse autoencoder model achieved an MSE rate of 0.069 when provided with the 50x50 pixel cropped tiles. The AANNAD algorithm produced several autoencoder architectures that outperformed the best architecture discovered by Saponara *et al.* A comparison between the AANNAD algorithm and the fingerprint recreation sparse autoencoder concerning the 50x50 pixel partitioning is presented in Table 5-5:

Table 5-5: Comparisons of different algorithms based on the 50x50 pixel image subset of the SOCOFing dataset

Approach utilised	Number of Epochs	Methodology utilised	Best MSE achieved
Sparse Autoencoder Neural Network	1000	50x50 pixel partition	0.069
AANNAD algorithm	451	50x50 pixel partition	0.05387

When presented with the 10x10 pixel partition, the sparse autoencoder neural network developed by Saponara *et al*. (2021) achieved an MSE rate of 0.033, whilst the AANNAD algorithm achieved an MSE of 0.01885. A comparison between the AANNAD algorithm and the fingerprint recreation sparse autoencoder concerning the 10x10 pixel partition is presented in Table 5-6:

Table 5-6: Comparisons of different algorithms based on the 10X10 pixel subset partition of the SOCOFing dataset

Approach utilised	Number of Epochs	Methodology utilised	Best MSE achieved
Sparse Autoencoder Neural Network	1000	10x10 pixel partition	0.033
AANNAD algorithm	1532	10x10 pixel partition	0.01885

As seen in the above tables, the AANNAD algorithm produced architectures capable of comparatively better results, based on the MSE score when compared to that of the sparse neural network developed by Saponara *et al.* (2021). A random selection of 10 fingerprints from the testing subset and their corresponding reconstruction (prediction) by the AANNAD algorithm is displayed in Figure 5-11.

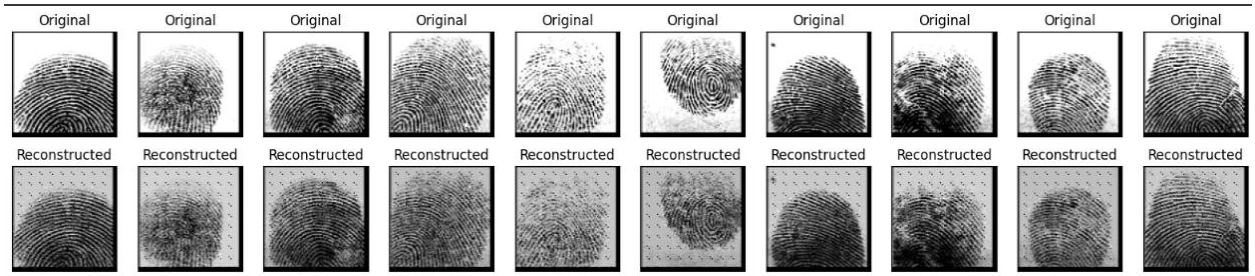

Figure 5-11: Reconstructed images of SOCOFing

Figure 5-11 is a graphical representation of the AANNAD algorithm's reconstructive capabilities. Based on the quantifiable results discussed in the previous subsections, recognisable reconstructions of the images contained within the SOCOFing dataset are produced. The following subsection concludes the experiments conducted on the SOCOFing dataset.

5.4.7 Conclusion

Based on the presented results, the AANNAD algorithm outperformed two of the best sparse autoencoder models found by Saponara *et al.* (2021) on the SOCOFing dataset. Within the following section, an experiment utilising the third dataset is presented.

5.5 Credit card fraud detection experiments

The third experiment utilised the Credit Card Fraud Detection (CCFD) dataset (Bruxelles, 2013). The dataset is described in Section 5.5.1. The pre-processing methods are explained in Section 5.5.2, and earlier investigations related to the dataset are presented in Section 5.5.3. Section 5.5.4 details the approach used to identify the best autoencoder architecture, and this architecture, which was identified by the AANNAD algorithm, is presented in Section 5.5.5. Experimental results are reported in Section 5.5.6, and concluding observations appear in Section 5.5.7.

5.5.1 Dataset description

The CCFD dataset comprises different transactions made by different credit cards during September of 2013 by various European cardholders (Bruxelles, 2013). This dataset presents transactions within two days, during which 492 fraudulent transactions occurred out of 284 807 transactions. Consequently, the dataset is highly unbalanced, with fraudulent transactions making up only 0.173% of all transactions.

5.5.2 Data pre-processing of the CCFD dataset

The CCFD dataset consists only of numerical input variables transformed by principal component analysis (Worldline, 2013). As a result of its nature, all values and features were pre-processed when the dataset was constructed to be completely confidential. Each data point within the dataset consists of 31 features, including the transaction time, amount, kind of transaction (fraudulent or legitimate), and, finally, confidential features labelled from V1 to V28 (Chang *et al.*, 2022).

After the data had been obtained, it was first pre-processed to ensure that the dataset did not contain any missing or incomplete values. Next, all duplicate values were additionally removed from the dataset, resulting in a total of 1081 duplicate values being removed. Finally, the information regarding the time of the transactions for all of the records was removed, and two additional separate versions of the data were made, namely one containing only fraudulent transactions and one containing only non-fraudulent transactions.

To be able to compare this experiment's results to the results obtained by Chang *et al.* (2022), the same procedure that they implemented to handle the imbalance in the data was followed. After the duplicate results had been removed, 5000 randomly selected transactions were extracted from the remaining 283 726 data points and combined with the remaining 473 fraudulent transactions to create a more balanced and smaller dataset containing 5473 data points. For the AANNAD algorithm experiments conducted on the CCFD dataset, a 70% (training), 20% (validation), and 10% (test) split was performed.

5.5.3 Prior research conducted with the CCFD dataset

Chang *et al.* (2022) conducted different experiments to identify accurate models for fraud detection platforms. The study performed five different experiments, utilising logistic regression, decision trees, k-nearest neighbours, a random forest, and an autoencoder machine learning technique. For the current experiment, the results of the AANNAD algorithm are compared with those of the autoencoder model.

An autoencoder model consisting of two encoder layers was utilised within the experiments. The layers contained 14 and 7 nodes, respectively. It should be noted that there is no mention of the composition of the decoder nor the composition of the bottleneck layer within the research of Chang *et al.* (2022). Three different sub-experiments were conducted with this model, with differing batch sizes and learning rates.

Each of the three experiments was tested against two datasets after training had concluded to gauge the capability of the system to detect fraudulent transactions. One of these datasets only consisted of fraudulent transactions, whilst the other consisted of non-fraudulent transactions. The experiments conducted by Chang *et al.* (2022) show that data points resulting in higher MSE values indicate possible fraudulent activity and would require additional investigation, whilst data points that return a lower MSE value are of a smaller risk of being fraudulent. The MSE values obtained when validating architectures on these different datasets are one of the determining factors in selecting the best-performing architectures by the AANNAD algorithm, as discussed in the following subsection. An MSE threshold of 3.0 was implemented to flag transactions that might be fraudulent (Chang *et al.*, 2022). This means that if the autoencoder produces an MSE above 3.0 for a data point, it would require further investigation, as it presents a higher chance of being a fraudulent transaction.

5.5.4 Selection of the best-performing architecture

Similarly to previous experiments, various metrics are calculated during the execution of the AANNAD algorithm. Within the experiments with the CCFD dataset, these metrics include the MSE value, based on fraudulent transactions, the MSE value, based on non-fraudulent

transactions and the training and validation loss throughout training. Similar to the research conducted by Chang *et al.* (2022), the MSE values based on fraudulent and non-fraudulent transactions are obtained after training has concluded and tested on datasets that contain solely fraudulent and non-fraudulent transactions.

Initially, training and validation datasets are created from the dataset containing 5000 non-fraudulent transactions and 473 fraudulent transactions. After each architecture had been trained and validated on their respective datasets, they were ranked according to their performance on the validation dataset based on MSE. After the number of architectures requested by the user was ranked, or until a time constraint was reached, the best-performing architecture was then tested on a dataset containing only non-fraudulent transactions and a dataset containing only fraudulent transactions. These results were then recorded as the final benchmark of performance from the best-performing autoencoder architecture of the AANNAD algorithm.

5.5.5 Description of the best-performing architecture

The AANNAD algorithm generated various architectures when applied to the CCFD dataset, with the most promising architectures consisting of between three and five hidden layers within the encoder and decoder module, respectively. The experimental results are shown in Table 5-7. These architectures represent comparable results to the architectures that had been reported by Chang *et al.* (2022). Some of the architectures generated by the AANNAD algorithm even outperform models constructed by Chang *et al.* (2022) when based on the MSE performance measure. During its 12-hour run time, numerous different architectures were evaluated, the results of which are shown in Table 5-7.

Table 5-7: Results of the best-performing architecture determined by the AANNAD algorithm applied to the CCFD dataset

Approach utilised	Running time of the algorithm	Number of architectures tested	MSE achieved based on the non-fraudulent test set	MSE achieved based on a fraudulent test set
AANNAD algorithm	12 hours	19	0.8513	21.7179

When the best autoencoder model found is applied to the fraudulent test subset, it produces a much higher MSE value than the non-fraudulent test subset. Within a typical autoencoder test, this would mean that the autoencoder architecture had performed comparatively worse; however,

within the context of this experiment, it would mean that the architecture produced by the AANNAD algorithm is much more sensitive to fraudulent transactions. Both the AANNAD model and the model produced by Chang *et al.* (2022) function on the premise that transactions that are more likely of a fraudulent nature should produce larger MSE rates. Consequently, a technique that produces a considerably higher MSE rate when presented with fraudulent transactions would more reliably detect and ultimately lead to the investigation and prevention of those transactions. The best autoencoder architecture found by the AANNAD algorithm consisted of input and output layers that each have 29 nodes. The encoder architecture consists of two hidden layers (EH_1, EH_2) and another two hidden layers within the decoder architecture (DH_1, DH_2). The composition of the hidden layers in the encoder module is 29 and 26 nodes, respectively. The bottleneck consists of 25 hidden nodes (BN). A graph depicting the training of this architecture in terms of the MSE loss for 13752 epochs is shown in Figure 5-12.

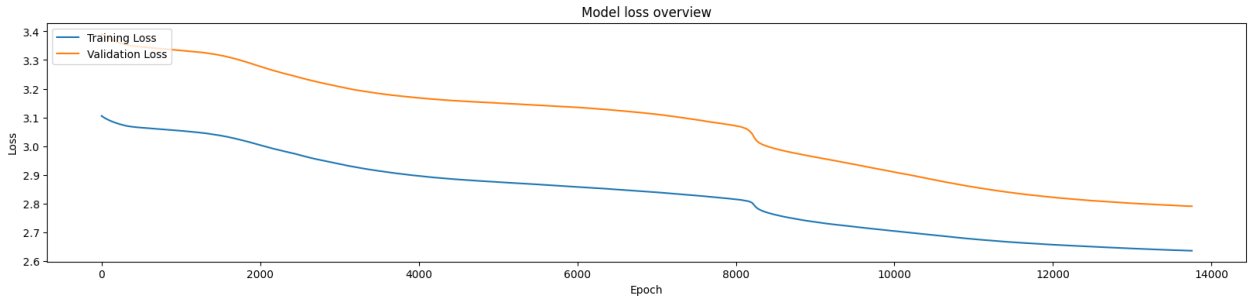

Figure 5-12: Training and validation loss of the architecture tested on a dataset containing fraudulent and non-fraudulent transactions

As seen in Figure 5-12, the training loss, represented by the blue line, and the validation loss, represented by the orange line, both steadily decrease over the course of the 13752 epochs it had taken to train this specific autoencoder architecture. Near epoch 12000, a slight plateau could be seen forming in both the training loss, as well as the validation loss, indicating that training is nearing the point where further training would induce overfitting and that the architecture would not benefit from further training. This point would be reached at epoch 13752, at which point the AANNAD algorithm had ceased further training of this architecture.

A graphical representation of the architecture implemented is shown in Figure 5-13. In the following section, the results of the best-performing architecture are compared to those of the baseline study conducted by Chang *et al.* (2022).

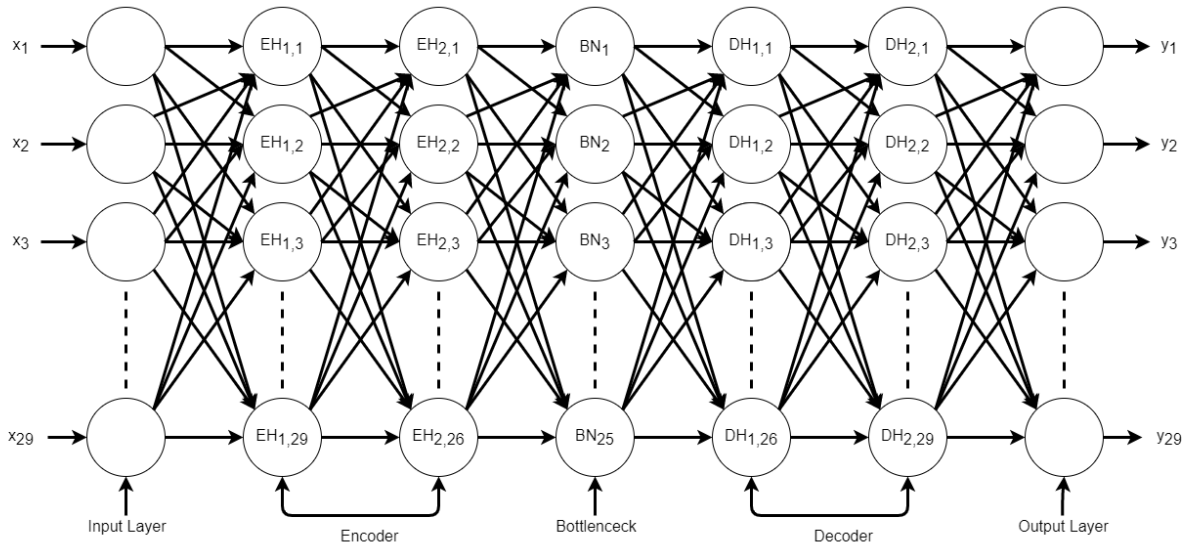

Figure 5-13: Best-performing autoencoder model for the CCFD dataset

5.5.6 Discussion of experimental results

The autoencoder that had been developed in the research conducted by Chang *et al.* (2022) was allowed to train for 80 epochs, and once their autoencoder models had been trained, they achieved an MSE between 0.867 and 0.904 when presenting the model with transactions of a non-fraudulent class, while achieving an MSE between 4.976 and 5.010 when presenting the model with transactions of the fraudulent class. A comparison between the AANNAD algorithm and the three different autoencoder experiments is presented in Table 5-8.

Table 5-8: Comparisons of different techniques when presented with both fraudulent and non-fraudulent credit card activities

Approach utilised	Number of Epochs	Best MSE achieved, based on a non-fraudulent dataset	Best MSE achieved, based on a fraudulent dataset
Chang *et al.* (2022) autoencoder experiment A	80	0.878	4.978
Chang *et al.* (2022) autoencoder experiment B	80	0.867	4.976
Chang *et al.* (2022) autoencoder experiment C	80	0.904	5.010
AANNAD algorithm	13752	0.851	21.718

The above table shows that the AANNAD algorithm produced an architecture capable of more accurately classifying non-fraudulent credit card transactions. It is also considerably more sensitive to fraudulent transactions. This points out potential fraudulent transactions much more accurately because of the larger MSE produced when presented, as both the AANNAD algorithm and the algorithm presented by Chang *et al.* (2022) rely on fraudulent transactions to produce a high MSE in order to distinguish them from non-fraudulent transactions. The following subsection concludes the experiments conducted on the CCFD dataset.

5.5.7 Conclusion

Based on the presented results, the AANNAD algorithm outperformed the best autoencoder model that had been developed by Chang *et al.* (2022) on the CCFD dataset. Within the following section, a discussion of the results obtained by the three experiments utilising the AANNAD algorithm is presented.

5.6 Discussion of results

Each of the experiments presented in this study utilised datasets from different domains to test the accuracy and robustness of the AANNAD algorithm across various applications. These datasets included MNIST for handwritten digit recognition, SOCOFing for fingerprint reconstruction, and Credit Card Fraud Detection (CCFD) for anomaly detection. The following discussions provide insights from these experiments, emphasising the AANNAD algorithm's robustness across diverse datasets and its accuracy compared to other baseline studies.

The first set of experiments, which utilised the MNIST dataset, demonstrated that the AANNAD algorithm's chromosomal evolutionary strategy could potentially generate and evolve different autoencoder architectures that may perform comparatively well regarding relatively small image-based datasets. This experiment had been compared to the research conducted by Charte *et al*. (2019). The next dataset against which the AANNAD algorithm had been tested, was the SOCOFing dataset with its biometric fingerprint images. In this experiment, the AANNAD algorithm outperformed the results of the baseline study that Saponara *et al.* (2021) developed. These results indicate that the AANNAD algorithm may show potential in tasks regarding the generation and evolution of autoencoder architectures for alternative, biological image-based data, specifically data consisting of fingerprints. The CCFD experiment showed that, with adequate pre-processing, the AANNAD algorithm could perform well with highly imbalanced data. Additionally, the algorithm demonstrated a potential for generating architectures that are considerably more sensitive to anomalies when compared to the results obtained by Chang *et al.* (2022). This heightened sensitivity is crucial for fraud detection systems, emphasising the algorithm's potential for real-world anomaly detection scenarios, based on transactional data.

Despite variations in data types across the different datasets, ranging from multi-dimensional images to transactional data, the algorithm consistently generated and optimised various autoencoder architectures that performed well. This cross-dataset performance underscores the robustness of the AANNAD algorithm.

5.7 Summary

The experimental design and results of applying the AANNAD algorithm to various datasets were presented in this chapter. The experimental design included details on the software and hardware used during the experiments. It also briefly addressed the steps followed in acquiring and pre-processing the data, with more detailed explanations provided for each dataset. Each experiment, which utilised a different dataset, was discussed in terms of the dataset's characteristics, specific pre-processing procedures applied, and prior research used as a comparative baseline to evaluate the functionality of the AANNAD algorithm. The process of selecting the best-performing architecture in each experiment was also explained, followed by a detailed description of the chosen architectures and a discussion of the observed results.

The datasets used in this study, namely MNIST, SOCOFing, and Credit Card Fraud Detection, were chosen to represent different types of data and application areas. The AANNAD algorithm demonstrated good performance compared to the techniques in the comparative studies across these diverse datasets.

The AANNAD algorithm's capability to automate the design of autoencoder neural network architectures, with experiments providing evidence of its potential to enhance performance in specific application areas, was illustrated in this chapter. These findings suggest that the AANNAD algorithm could be a promising machine learning technique for researchers and practitioners.

CHAPTER 6 CONCLUSION

6.1 Introduction

The primary aim of this study was to develop the novel Automated Autoencoder Neural Network Architecture Design (AANNAD) algorithm capable of constructing accurate autoencoder architectures for various scenarios. The AANNAD algorithm was designed and tested to achieve this aim, using three distinct datasets, each representing different data types. The algorithm's accuracy was evaluated by comparing its performance against three baseline studies, one for each dataset, conducted by other researchers. Experimental results demonstrated that the AANNAD algorithm may be an effective approach for constructing accurate autoencoder architectures when applied to datasets similar to those used in this study.

The remainder of the chapter is structured as follows. The research questions are revisited, and the study's objectives are evaluated in Section 6.2. The contributions made by the research conducted within this study are discussed in Section 6.3. Finally, in Section 6.4, possible future work that could contribute to the research's subsequent development is presented.

6.2 Evaluation of research objectives

As outlined in Chapter 1, this study's three guiding research questions have been stated as: *What are the required resources or information for developing the automatic design of the autoencoder? How can a novel algorithm be developed to automate the hyperparameter selection process for an accurate autoencoder neural network architecture?* and finally*, what will be the mode of evaluating the designed automatic autoencoder algorithm?* Therefore, the study's primary aim was *to develop a novel automated autoencoder neural network architecture construction algorithm to create accurate autoencoder architectures using Design Science Research and a positivism approach*. Three secondary objectives have been presented to achieve this primary aim and answer the proposed research questions. In the following subsections, how each of the three secondary objectives was accomplished will be explained.

6.2.1 Objective 1: To perform a literature study on deep learning, autoencoder architectures and the automated design of neural network architectures

A comprehensive literature review was conducted throughout Chapters 2, 3, and 4. The foundational concepts of artificial intelligence and deep learning were introduced in Chapter 2. An introduction to these concepts was presented in Section 2.1, while the history of deep learning, covering its evolution from cybernetics (1940s-1960s) through connectionism (1980s-1990s) to its current resurgence, was explored in Section 2.2. The structure of neural networks was

examined in Section 2.3, explaining key components, learning structures, and various types of networks, such as perceptrons, feedforward networks, convolutional neural networks, recurrent neural networks, and autoencoders. The application areas and challenges of neural networks were also addressed in Section 2.4 and concluded with an overview of neural network training methodologies, including supervised learning, unsupervised learning, and reinforcement learning in Section 2.5.

Chapter 3 focused explicitly on autoencoders. The concept of autoencoders was introduced in Section 3.1, followed by an explanation of the functionality of autoencoders in unsupervised learning in Section 3.2. Section 3.3 has been used to outline a general framework for autoencoder architectures, while various regularisation techniques used for sparse, denoising, and contractive autoencoders have been reviewed in Section 3.4. The applications of autoencoders in areas, such as generative modelling, classification, clustering, anomaly detection, recommendation systems, and dimensionality reduction have been discussed in Section 3.5. Finally, asymmetric autoencoders have been explored in Section 3.6, contrasting the traditional symmetric models.

Chapter 4 covered the topic of automated neural network architecture design. Automated machine learning (AutoML) as a broader concept has been introduced in Section 4.1. Section 4.2 has been used to elaborate on the core components of AutoML, including data preparation, feature engineering, model generation, evaluation, and validation. Section 4.3 focused on the development of the AANNAD algorithm. This chapter also provided details on partitioning datasets used for testing the algorithm in Section 4.4.

6.2.2 Objective 2: To create a novel algorithm to automatically design an accurate autoencoder neural network

The AANNAD algorithm described in Section 4.3 centred on automating the generation of autoencoder architectures through an evolutionary strategy. The algorithm began with creating an initial population of diverse autoencoder models, which were trained and evaluated using sub-partitions of the various datasets. After evaluating the models based on performance metrics, such as the reconstruction error, the algorithm employed mutation and selection techniques to refine and improve the top-performing architectures. The mutation phase involved altering key architecture components, such as the number of hidden layers, nodes per layer, and dropout rates. In contrast, the selection phase identified the most successful models to progress to subsequent iterations. These foundational steps formed the core of the algorithm's design.

6.2.3 Objective 3: To determine the success of the automated construction algorithm by considering applicable metrics

Three distinct datasets, all available in the public domain, were sourced and used to evaluate the performance of the AANNAD algorithm. These datasets were described in Sections 5.3, 5.4 and 5.5, respectively. The first dataset, MNIST (Modified National Institute of Standards and Technology), consists of image-based data of hand-drawn digits (LeCun *et al*., 2010). The second dataset, SOCOFing (Sokoto Coventry Fingerprint), comprises biometric fingerprint images (Saponara *et al*., 2021). Lastly, the CCFD (Credit Card Fraud Detection) dataset contains transactional data related to credit card usage and fraud detection (Chang *et al*., 2022). The diversity of these datasets ensured that the AANNAD algorithm was tested across various scenarios, assessing its ability to process and model different data types.

The AANNAD algorithm has been applied to the three datasets. The algorithm demonstrated its versatility by generating autoencoder architectures capable of accurately reconstructing the features from each dataset. For the MNIST dataset in Section 5.3, which contains image-based data of hand-drawn digits, the AANNAD algorithm automatically constructed autoencoder architectures that excelled in reconstructing and compressing the visual information, as evidenced by its low reconstruction error. In Section 5.4, where the algorithm was applied to the SOCOFing dataset, it handled the complexities of biometric fingerprint data, successfully generating autoencoder architectures that preserved the intricate details of the fingerprint images during the reconstruction process. Finally, in Section 5.5, the AANNAD algorithm processed the CCFD dataset, which contains transactional data, demonstrating its ability to model structured, non-image data by accurately reconstructing patterns in the transactional information, further confirming its robustness across different data types.

The success of the AANNAD algorithm was measured using several key metrics, including reconstruction error, the number of training epochs, and the time taken to complete training. These metrics were compared against baseline studies conducted by other researchers, as discussed in Chapter 5. Specifically, the MNIST dataset results were compared in Section 5.3 to the baseline study by Charte *et al.* (2019). For the SOCOFing dataset, the algorithm's performance was compared in Section 5.4 to the study by Saponara *et al.* (2021). Lastly, for the CCFD dataset, the results were evaluated in Section 5.5 against the baseline established by Chang *et al.* (2022). Across all datasets, the comparative results demonstrated that the AANNAD algorithm performed competitively, often surpassing the baseline models in terms of accuracy. These findings suggest that the AANNAD algorithm may generate higher-performing autoencoder architectures than the techniques employed in the comparative studies.

6.3 Contributions of the study

This study introduces a novel AANNAD algorithm for automating the construction of accurate autoencoder neural networks, simplifying the complexity of hyperparameter tuning, architecture creation, and selection. The research also contributes in the following ways:

- An overview of the literature on deep learning, neural networks, autoencoders, and the concept of AutoML was conducted. This may serve as a helpful literature study for future researchers in the field.
- The research demonstrated the AANNAD algorithm's robustness and adaptability across various domains, validating its ability to effectively handle diverse datasets and problem types. The AANNAD algorithm may be presented as a versatile tool with potential applications in multiple fields by showcasing this success. This may, in turn, serve as a proof of concept that automated autoencoder architecture development may address a broad range of challenges and, in turn, may provide a foundation for future advancements in AutoML systems and encourage the exploration of similar implementations for complex, real-world problems.
- The AANNAD algorithm's ability to autonomously evolve architectures tailored to specific datasets highlights its possible versatility and practical application across different datasets.
- By automating traditionally labour-intensive tasks, the AANNAD algorithm makes autoencoder neural network design more accessible while possibly achieving high accuracy across diverse datasets.

6.4 Future work

Several potential directions for future research could enhance the AANNAD algorithm as follows:

- Future research could expand the algorithm's capacity to process larger and more complex datasets.
- The autoencoder architectures generated by the AANNAD algorithm in this study were limited to symmetrical structures. Expanding the algorithm to support asymmetric architectures could enhance performance and provide flexibility for various applications.
- Future iterations of the algorithm could automate the design of additional types of neural networks, such as convolutional and recurrent neural networks, further broadening its versatility and use in different machine learning tasks.
- Optimising the algorithm to enhance its computational performance, particularly regarding resource usage and training time, would make it more practical for real-world applications, especially when working with large-scale datasets.

- Investigating advanced techniques like weight sharing (Choi *et al.*, 2021) and surrogate-based methodologies (Cui *et al.*, 2021) could improve the algorithm's efficiency by expanding upon concepts, such as feature compression and optimising the evolutionary methodology utilised, making it more effective in identifying accurate network designs.

6.5 Summary

The research questions and objectives set out at the beginning of this study were reviewed in this final chapter. It demonstrated how they were addressed through the development and application of the AANNAD algorithm. The contributions of this research, which include automating the design of autoencoder architectures and successfully applying the algorithm across diverse datasets, were also highlighted. Additionally, several avenues for future research were proposed to improve the algorithm’s performance and adaptability further. The results obtained during the study show that the AANNAD algorithm may be a competitive algorithm for automating accurate autoencoder neural network design, often outperforming previously designed models. These findings suggest that the algorithm may be valuable in the ongoing advancement of autoencoder neural network architecture optimisation.

REFERENCE LIST

Abdel-Nabi, H., Al-Naymat, G., Ali, M.Z. & Awajan, A. 2023. HcLSH: a novel non-linear monotonic activation function for deep learning methods. *IEEE Access*, 11:47794-47815.

Abiodun, O.I., Jantan, A., Omolara, A.E., Dada, K.V., Mohamed, N.A. & Arshad, H. 2018. State-of-the-art in artificial neural network applications: A survey. *Heliyon*, 4(11):e00938.

Adnan, M.M., Rahim, M.S.M., Rehman, A., Mehmood, Z., Saba, T. & Naqvi, R.A. 2021. Automatic image annotation based on deep learning models: a systematic review and future challenges. *IEEE Access*, 9:50253-50264.

Agar, J. 2020. What is science for? The Lighthill report on artificial intelligence reinterpreted. *The British Journal for the History of Science*, 53(3):289-310.

Agarwal, Y., Ish, P., Chaudhry, N. & Sethi, R.S. 2018. The neurology timeline: Of demon holes, sacred pathways and great triumphs. *Astrocyte*, 5(1):10.

Aggarwal, C.C. 2018. *Neural Networks and Deep Learning*: A Textbook. 1st ed. Cham: Springer. ISBN: 978-3-319-94462-3.

Agrawal, P., Abutarboush, H.F., Ganesh, T. & Mohamed, A.W. 2021. Metaheuristic algorithms on feature selection: A survey of one decade of research (2009-2019). *Ieee Access*, 9:26766-26791.

Ali, Y.A., Awwad, E.M., Al-Razgan, M. & Maarouf, A. 2023. Hyperparameter search for machine learning algorithms for optimizing the computational complexity. *Processes*, 11(2):349.

Almeida, L.B. 2020. Multilayer perceptrons. In: Fiesler, E. & Beale, R., eds. Handbook of Neural Computation. 1st ed. New York: Institute of Physics Publishing Ltd and Oxford University Press. pp. C1.2:1-C1.2:30.

Angelov, P.P., Soares, E.A., Jiang, R., Arnold, N.I. & Atkinson, P.M. 2021. Explainable artificial intelligence: an analytical review. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 11(5):e1424.

Apicella, A., Donnarumma, F., Isgrò, F. & Prevete, R. 2021. A survey on modern trainable activation functions. *Neural Networks*, 138:14-32.

Baldi, P. 2012. Autoencoders, unsupervised learning, and deep architectures. In: Proceedings of ICML Workshop on Unsupervised and Transfer Learning. Bellevue, Washington, USA: JMLR Workshop and Conference Proceedings. pp. 37-49.

Bank, D., Koenigstein, N. & Giryes, R. 2023. Autoencoders. In: Rokach, L., Maimon, O. & Shmueli, E., eds*. Machine Learning for Data Science Handbook: Data Mining and Knowledge Discovery Handbook*. Cham: Springer International Publishing. pp. 353-374.

Bataineh, M.H. 2015. *New neural network for real-time human dynamic motion prediction*. The University of Iowa.

Baymurzina, D., Golikov, E. & Burtsev, M. 2022. A review of neural architecture search. *Neurocomputing*, 474:82-93.

Beggel, L., Pfeiffer, M. & Bischl, B. 2019. Robust anomaly detection in images using adversarial autoencoders. In: Joint European Conference on Machine Learning and Knowledge Discovery in Databases. Springer. pp. 206-222.

Bengio, Y., Goodfellow, I. & Courville, A. 2017. *Deep learning*. Cambridge, MA, USA: 1. MIT press.

Berahmand, K., Daneshfar, F., Salehi, E.S., Li, Y. & Xu, Y. 2024. Autoencoders and their applications in machine learning: a survey. *Artificial Intelligence Review*, 57(2):28.

Bilski, J., Rutkowski, L., Smoląg, J. & Tao, D. 2021. A novel method for speed training acceleration of recurrent neural networks. *Information Sciences*, 553:266-279.

Bourlard, H. & Kabil, S.H. 2022. Autoencoders reloaded. *Biological cybernetics*, 116(4):389-406.

Bruxelles, W.a.t.M.L.G.o.U.L.d. 2013. Credit Card Fraud Detection [Dataset]. Université Libre de Bruxelles, Machine Learning Group. DOI: 10.23721/100/1504318.

Buggineni, V. 2023. Utilizing Synthetic Data Generation Techniques to Improve the Availability of Data in Discrete Manufacturing for AI Applications: A Review and Framework. Athens, Georgia: University of Georgia.

Cai , Z., Cui, Z., Lassance, N. & Simaan, M. 2024. The Economic Value of Mean Squared Error: Evidence from Portfolio Selection. SSRN Electronic Journal. Available at: https://ssrn.com/abstract=4856139 or http://dx.doi.org/10.2139/ssrn.4856139.

Chang, V., Di Stefano, A., Sun, Z. & Fortino, G. 2022. Digital payment fraud detection methods in digital ages and Industry 4.0. *Computers and Electrical Engineering*, 100:107734.

Chapman , A.D. 2023. Neural Networks. The Autodidact's Toolkit. New York: The Autodidact's Toolkit.

C harte, F., Rivera, A.J., Martínez, F. & del Jesus, M.J. 2019. Automating autoencoder architecture configuration: An evolutionary approach. In: International Work-Conference on the Interplay Between Natural and Artificial Computation. Almería, Spain: Springer. pp. 339-349.

Chen, K., Liu, Z., Hong, L., Xu, H., Li, Z. & Yeung, D.-Y. 2023. Mixed autoencoder for self-supervised visual representation learning. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 22742-22751.

Chen, R.J., Lu, M.Y., Chen, T.Y., Williamson, D.F. & Mahmood, F. 2021. Synthetic data in machine learning for medicine and healthcare. *Nature Biomedical Engineering*, 5(6):493-497.

Chen, S. & Guo, W. 2023. Auto-encoders in deep learning—a review with new perspectives. *Mathematics*, 11(8):1777.

Chen, X., Ding, M., Wang, X., Xin, Y., Mo, S., Wang, Y., ... Wang, J. 2024. Context autoencoder for self-supervised representation learning. *International Journal of Computer Vision*, 132(1):208-223.

Choi, J.S., Kim, J. & Ko, J.H. 2021. A Weight-Sharing Autoencoder with Dynamic Quantization for Efficient Feature Compression. In: M. Lee, T. Kim, & Y. Park, eds. Proceedings of the 2021 International Conference on Information and Communication Technology Convergence (ICTC). IEEE, pp. 1111-1113.

Comte, A. 1856. *A General View of Positivism.* London: Smith, Elder & Co.

Cong, S. & Zhou, Y. 2023. A review of convolutional neural network architectures and their optimizations. *Artificial Intelligence Review*, 56(3):1905-1969.

Côté, P.-O., Nikanjam, A., Ahmed, N., Humeniuk, D. & Khomh, F. 2024. Data cleaning and machine learning: a systematic literature review. *Automated Software Engineering*, 31(2):54.

Cui, M., Li, L., Zhou, M. & Abusorrah, A. 2021. Surrogate-assisted autoencoder-embedded evolutionary optimization algorithm to solve high-dimensional expensive problems. *IEEE Transactions on Evolutionary Computation*, 26(4):676-689.

Dargan, S., Kumar, M., Ayyagari, M.R. & Kumar, G. 2020. A survey of deep learning and its applications: a new paradigm to machine learning. *Archives of Computational Methods in Engineering*, 27:1071-1092.

Dawani, J. 2020. *Hands-On Mathematics for Deep Learning: Build a Solid Mathematical Foundation for Training Efficient Deep Neural Networks.* Birmingham, UK: Packt Publishing Ltd. ISBN: 978-1-83864-729-2.

De Santana Correia, A. & Colombini, E.L. 2022. Attention, please! A survey of neural attention models in deep learning. *Artificial Intelligence Review*, 55(8):6037-6124.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K. & Fei-Fei, L. 2009. Imagenet: A large-scale hierarchical image database. In: 2009 IEEE conference on computer vision and pattern recognition. Ieee. pp. 248-255.(See 4.9 in Referencing Guide]

Deng, L. 2012. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE signal processing magazine*, 29(6):141-142.

Dike, H.U., Zhou, Y., Deveerasetty, K.K., & Wu, Q. 2018. Unsupervised Learning Based on Artificial Neural Network: A Review. In: Proceedings of the 2018 IEEE International Conference on Cyborg and Bionic Systems (CBS), Shenzhen, China, IEEE, pp. 322-327.

Ding, Y., Liu, C., Zhu, H. & Chen, Q. 2023. A supervised data augmentation strategy based on random combinations of key features. *Information Sciences*, 632:678-697.

Dubey, S.R., Singh, S.K. & Chaudhuri, B.B. 2022. Activation functions in deep learning: A comprehensive survey and benchmark. *Neurocomputing*, 503:92-108.

El-Shafai, W., El-Nabi, S.A., El-Rabaie, E.-S.M., Ali, A.M., Soliman, N.F., Algarni, A.D., ... Fathi, E. 2022. Efficient Deep-Learning-Based Autoencoder Denoising Approach for Medical Image Diagnosis. *Computers, Materials & Continua*, 70(3).

Elshawi, R., Maher, M. & Sakr, S. 2019. Automated machine learning: State-of-the-art and open challenges. *arXiv preprint arXiv:1906.02287.*

Elsken, T., Metzen, J.H. & Hutter, F. 2019. Neural architecture search: A survey. *The Journal of Machine Learning Research*, 20(1):1997-2017.

Feng, L., Shu, S., Lin, Z., Lv, F., Li, L. & An, B. 2021. Can cross entropy loss be robust to label noise? In: Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence. Yokohama, Japan. pp. 2206-2212.

Feurer, M., Klein, A., Eggensperger, K., Springenberg, J.T., Blum, M. & Hutter, F. 2015. Efficient and robust automated machine learning. In: Advances in Neural Information Processing Systems. Vol. 28.

Fomby, T. 2008. Scoring measures for prediction problems. *Department of Economics, Southern Methodist University, Dallas, TX*.

Franceschi, L., Donini, M., Perrone, V., Klein, A., Archambeau, C., Seeger, M., ... Frasconi, P. 2024. Hyperparameter Optimization in Machine Learning. *arXiv preprint arXiv:2410.22854.*

Frazier, P.I. 2018. Bayesian Optimization. In: Recent Advances in Optimization and Modeling of Contemporary Problems. INFORMS TutORials in Operations Research. pp. 255-278.

Fukushima, K. 1988. Neocognitron: A hierarchical neural network capable of visual pattern recognition. *Neural networks*, 1(2):119-130.

Ganepola, V.V.V. & Wirasingha, T. 2021. Automating generative adversarial networks using neural architecture search: A review. In: 2021 International Conference on Emerging Smart Computing and Informatics (ESCI). Pune, India: IEEE. pp. 577-582.

Garson, J. 1997. *Connectionism*. Stanford Encyclopedia of Philosophy: Metaphysics Research Lab, Stanford University.

Gilbert, M.S., De Campos, M.L. & Campista, M.E.M. 2024. Asymmetric Autoencoders: An NN alternative for resource-constrained devices in IoT networks. *Ad Hoc Networks*, 156:103412.

Gomar, S., Mirhassani, M., & Ahmadi, M. 2016. Precise Digital Implementations of Hyperbolic Tanh and Sigmoid Function. In: Proceedings of the 2016 50th Asilomar Conference on Signals, Systems and Computers, Pacific Grove, CA, USA, IEEE, pp. 1586-1589. DOI: 10.1109/ACSSC.2016.7869646.

Goodfellow, I., Bengio, Y. & Courville, A. 2016. *Deep learning*. MIT press.

Google. 2023. Colaboratory, Frequently Asked Questions. Available at: https://research.google.com/colaboratory/faq.html#gpu-availability [Accessed 15 December 2023].

Google. 2024. *Books Ngram View of the terms neural networks, connectionism, cybernetics and deep learning* [Accessed 2024].

Gowdra, N., Sinha, R., MacDonell, S. & Yan, W. 2021. Maximum Categorical Cross Entropy (MCCE): A noise-robust alternative loss function to mitigate racial bias in Convolutional Neural Networks (CNNs) by reducing overfitting. In: International Conference on Learning Representations (ICLR).

Greenhill, S., Rana, S., Gupta, S., Vellanki, P. & Venkatesh, S. 2020. Bayesian optimization for adaptive experimental design: A review. *IEEE access*, 8:13937-13948.

Guilhoto, L.F. 2018. An overview of artificial neural networks for mathematicians. University of Chicago.

Guo, X., Liu, X., Zhu, E. & Yin, J. 2017. Deep clustering with convolutional autoencoders. In: Neural Information Processing: 24th International Conference, ICONIP 2017, Guangzhou, China, November 14-18, 2017, Proceedings, Part II 24. Springer. pp. 373-382.

Gupta, M.K. & Chandra, P. 2020. A comprehensive survey of data mining. *International Journal of Information Technology*, 12(4):1243-1257.

Gurney, K. 2018. *An Introduction to Neural Networks*. Boca Raton, FL, USA: CRC Press. ISBN: 978-1-138-41562-3.

Hagan, M.T., Demuth, H.B., Beale, M.H., & De Jesús, O. 2014. *Neural Network Design.* Stillwater, OK, USA: Martin Hagan. ISBN: 978-0-9717321-1-7.

Haji, S.H. & Abdulazeez, A.M. 2021. Comparison of optimization techniques based on gradient descent algorithm: A review. *PalArch's Journal of Archaeology of Egypt/Egyptology*, 18(4):2715-2743.

Hameed, M. & Naumann, F. 2020. Data preparation: A survey of commercial tools. *ACM SIGMOD Record*, 49(3):18-29.

He, X., Zhao, K. & Chu, X. 2021. AutoML: A survey of the state-of-the-art. *Knowledge-Based Systems*, 212:106622.

Hebb, D.O. 1949. The Organization of Behavior: A Neuropsychological Theory. New York: John Wiley & Sons.

Hinton, G.E., Osindero, S. & Teh, Y.-W. 2006. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527-1554.

Hirsch-Kreinsen, H. 2024. Artificial intelligence: A “promising technology”. *AI & SOCIETY*, 39(4):1641-1652.

Hodson, T.O. 2022. Root mean square error (RMSE) or mean absolute error (MAE): When to use them or not. *Geoscientific Model Development Discussions*, 2022:1-10.

Hospedales, T., Antoniou, A., Micaelli, P. & Storkey, A. 2021. Meta-learning in neural networks: A survey. IEEE Transactions on Pattern Analysis and Machine Intelligence, 44(9):5149–5169. DOI: 10.1109/TPAMI.2021.3079209.

Hutter, F., Kotthoff, L. & Vanschoren, J. 2019. Automated Machine Learning: Methods, Systems, Challenges. 1st ed. Cham: Springer Nature. ISBN: 978-3-030-05318-5.

Islam, M.N. 2023. Introduction to the Perceptron and Its Applications. Jahangirnagar University, Bangladesh.

Islam, M.R., Lima, A.A., Das, S.C., Mridha, M.F., Prodeep, A.R. & Watanobe, Y. 2022. A comprehensive survey on the process, methods, evaluation, and challenges of feature selection. *IEEE Access*, 10:99595-99632.

Ivakhnenko, A.G. & Lapa, V.G. 1965. Cybernetic Predicting Devices. Translated and prepared by U.S. Joint Publications Research Service. CCM Information.

Janiesch, C., Zschech, P. & Heinrich, K. 2021. Machine learning and deep learning. *Electronic Markets*, 31(3):685-695.

Jayagopal, V. & Basser, K. 2022. Data management and big data analytics: Data management in digital economy. In: *Research anthology on Big Data analytics, architectures, and applications*: IGI Global. pp. 1614-1633.

Ji, S., Ye, K. & Xu, C.-Z. 2020. A network intrusion detection approach based on asymmetric convolutional autoencoder. In: Cloud Computing–CLOUD 2020: 13th International Conference, Held as Part of the Services Conference Federation, SCF 2020, Honolulu, HI, USA, September 18-20, 2020, Proceedings 13. Springer. pp. 126-140.

Jiang, Y., Li, X., Luo, H., Yin, S. & Kaynak, O. 2022. Quo vadis artificial intelligence? *Discover Artificial Intelligence*, 2(1):4.

Jin, H., Song, Q. & Hu, X. 2019. Auto-keras: An efficient neural architecture search system. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. Anchorage, Alaska, USA. pp. 1946-1956.

Kankam, P.K. 2019. The use of paradigms in information research. *Library & Information Science Research*, 41(2):85-92.

Katoch, S., Chauhan, S.S. & Kumar, V. 2021. A review on genetic algorithm: past, present, and future. *Multimedia tools and applications*, 80:8091-8126.

Ke, F., Yuchen, Z. & Ping, M. 2020. An adaptive sequential experiment design method for model validation. *Chinese Journal of Aeronautics*, 33(6):1661-1672.

Kim, J.-H., Choi, J.-H., Chang, J., & Lee, J.-S. 2020. *Efficient Deep Learning-Based Lossy Image Compression via Asymmetric Autoencoder and Pruning.* In: Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), Barcelona, Spain, IEEE, pp. 2063-2067. DOI: 10.1109/ICASSP40776.2020.9053102.

Kingma, D.P. & Welling, M. 2019. An introduction to variational autoencoders. *Foundations and Trends® in Machine Learning*, 12(4):307-392.

Kivunja, C. & Kuyini, A.B. 2017. Understanding and applying research paradigms in educational contexts. *International Journal of higher education*, 6(5):26-41.

Klein, A., Falkner, S., Bartels, S., Hennig, P. & Hutter, F. 2017. Fast Bayesian Optimization of Machine Learning Hyperparameters on Large Datasets. In: Proceedings of the 20th International Conference on Artificial Intelligence and Statistics. PMLR, 54:528-536.

Laboratory, C.A. 2018. **https://digital.library.cornell.edu/catalog/ss:550351** Date of access: 14 April 2022.

Lambora, A., Gupta, K. & Chopra, K. 2019. Genetic algorithm: A literature review. In: 2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon). Faridabad, India: IEEE. pp. 380-384.

LeDell, E. & Poirier, S. 2020. H2O AutoML: Scalable Automatic Machine Learning. In: H2O World Conference.

Leijnen, S. & Veen, F.v. 2020. The Neural Network Zoo. *Proceedings*, 47(1):9. **https://www.mdpi.com/2504-3900/47/1/9** [Accessed 29 March 2024].

Lennox, J.C. 2020. 2084: Artificial Intelligence and the Future of Humanity. Grand Rapids, Michigan: Zondervan. ISBN: 978-0-310-11313-1.

Li, J., Cheng, K., Wang, S., Morstatter, F., Trevino, R.P., Tang, J. & Liu, H. 2017. Feature selection: A data perspective. *ACM computing surveys (CSUR)*, 50(6):1-45.

Lillicrap, T.P., Santoro, A., Marris, L., Akerman, C.J. & Hinton, G. 2020. Backpropagation and the brain. *Nature Reviews Neuroscience*, 21(6):335-346.

Lin, C.-H., Chen, T.-Y., Chen, H.-Y. & Chan, Y.-K. 2024. Efficient and lightweight convolutional neural network architecture search methods for object classification. *Pattern Recognition*, 156:110752.

Liu, F., Li, H., Hu, W. & He, Y. 2024. Review of neural network model acceleration techniques based on FPGA platforms. *Neurocomputing*:128511.

Liu, H., Simonyan, K., Vinyals, O., Fernando, C., & Kavukcuoglu, K. 2018. Hierarchical Representations for Efficient Architecture Search. In: International Conference on Learning Representations (ICLR) 2018 Conference Track Proceedings.

Liu, H., Simonyan, K. & Yang, Y. 2019. DARTS: Differentiable Architecture Search. In: International Conference on Learning Representations (ICLR).

Liu, W., Wang, Z., Liu, X., Zeng, N., Liu, Y. & Alsaadi, F.E. 2017. A survey of deep neural network architectures and their applications. *Neurocomputing*, 234:11-26.

Liu, Z., Lin, Y., & Sun, M. 2023. *Representation Learning for Natural Language Processing.* 2nd ed. Cham, Switzerland: Springer Nature. ISBN: 978-3-031-12345-6.

Luo, R., Tan, X., Wang, R., Qin, T., Chen, E. & Liu, T.-Y. 2020. Semi-supervised neural architecture search. *Advances in Neural Information Processing Systems*, 33:10547-10557.

Maharana, K., Mondal, S. & Nemade, B. 2022. A review: Data pre-processing and data augmentation techniques. *Global Transitions Proceedings*, 3(1):91-99.

Mahesh, B. 2020. Machine learning algorithms: A review. *International Journal of Science and Research (IJSR).[Internet]*, 9(1):381-386.

Majumdar, A., & Tripathi, A. 2017. *Asymmetric Stacked Autoencoder.* In: Proceedings of the 2017 International Joint Conference on Neural Networks (IJCNN), Anchorage, AK, USA, IEEE, pp. 911-918. DOI: 10.1109/IJCNN.2017.7966012.

McCulloch, W.S. & Pitts, W. 1943. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115-133.

McDonald, A. 2021. Data quality considerations for petrophysical machine-learning models. *Petrophysics*, 62(06):585-613.

Mendez Lucero, M.-A., Karampatsis, R.-M., Bojorquez Gallardo, E. & Belle, V. 2022. Signal perceptron: On the identifiability of Boolean function spaces and beyond. *Frontiers in Artificial Intelligence*, 5:770254.

Meyer, G.P. 2021. *An Alternative Probabilistic Interpretation of the Huber Loss.* In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Nashville, TN, USA, pp. 5261-5269. DOI: 10.1109/CVPR46437.2021.00521.

Mijwel, M.M. 2021. Artificial neural networks advantages and disadvantages. *Mesopotamian Journal of Big Data*, 2021:29-31.

Misra, D. 2019. Mish: A self regularized non-monotonic activation function. *arXiv preprint arXiv:1908.08681*,

Misra, S., Thakur, S., Ghosh, M. & Saha, S.K. 2020. An autoencoder-based model for detecting fraudulent credit card transaction. *Procedia Computer Science*, 167:254-262.

Mitchell, T.M. 1997. Machine Learning. New York: McGraw Hill. ISBN: 978-0-07-042807-2.

Moayedi, H., Mosallanezhad, M., Rashid, A.S.A., Jusoh, W.A.W. & Muazu, M.A. 2020. A systematic review and meta-analysis of artificial neural network application in geotechnical engineering: theory and applications. *Neural Computing and Applications*, 32:495-518.

Mohd Amiruddin, A.A.A., Zabiri, H., Taqvi, S.A.A. & Tufa, L.D. 2020. Neural network applications in fault diagnosis and detection: an overview of implementations in engineering-related systems. *Neural Computing and Applications*, 32(2):447-472.

Molnar, C. 2020. *Interpretable machine learning*. Lulu. com.

Montesinos López, O.A., Montesinos López, A., & Crossa, J. 2022. *Fundamentals of Artificial Neural Networks and Deep Learning.* In: Multivariate Statistical Machine Learning Methods for Genomic Prediction, 1st ed., Cham, Switzerland: Springer, pp. 379-425. DOI: 10.1007/978-3-030-89010-0_12.

Moradi, R., Berangi, R. & Minaei, B. 2020. A survey of regularization strategies for deep models. *Artificial Intelligence Review*, 53(6):3947-3986.

Muthukrishnan, N., Maleki, F., Ovens, K., Reinhold, C., Forghani, B. & Forghani, R. 2020. Brief history of artificial intelligence. *Neuroimaging Clinics of North America*, 30(4):393-399.

Naeem, M., Rizvi, S.T.H. & Coronato, A. 2020. A gentle introduction to reinforcement learning and its application in different fields. *IEEE access*, 8:209320-209344.

Nanga, S., Bawah, A.T., Acquaye, B.A., Billa, M.-I., Baeta, F.D., Odai, N.A., ... Nsiah, A.D. 2021. Review of dimension reduction methods. *Journal of Data Analysis and Information Processing*, 9(3):189-231.

Naranjo-Torres, J., Mora, M., Hernández-García, R., Barrientos, R.J., Fredes, C. & Valenzuela, A. 2020. A review of convolutional neural network applied to fruit image processing. *Applied Sciences*, 10(10):3443.

Nargesian, F., Samulowitz, H., Khurana, U., Khalil, E.B., & Turaga, D.S. 2017. *Learning Feature Engineering for Classification.* In: Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI), Melbourne, Australia, pp. 2529-2535. DOI: 10.24963/ijcai.2017/352.

Neutatz, F., Chen, B., Abedjan, Z. & Wu, E. 2021. From Cleaning before ML to Cleaning for ML. *IEEE Data Eng. Bull.*, 44(1):24-41.

Nikbakht, S., Anitescu, C. & Rabczuk, T. 2021. Optimizing the neural network hyperparameters utilizing genetic algorithm. *Journal of Zhejiang University Science A*, 22(6):407-426.

Olson, R.S. & Moore, J.H. 2016. TPOT: A Tree-based Pipeline Optimization Tool for Automating Machine Learning. In: Workshop on Automatic Machine Learning. pp. 66–74.

Pandey, M., Fernandez, M., Gentile, F., Isayev, O., Tropsha, A., Stern, A.C. & Cherkasov, A. 2022. The transformational role of GPU computing and deep learning in drug discovery. *Nature Machine Intelligence*, 4(3):211-221.

Peters, M.A. 2017. Deep learning, the final stage of automation and the end of work (Again)? *Psychosociological Issues in Human Resource Management*, 5(2):154-168.

Pham, H., Guan, M., Zoph, B., Le, Q., & Dean, J. 2018. *Efficient Neural Architecture Search via Parameters Sharing.* In: Proceedings of the 35th International Conference on Machine Learning (ICML), Stockholm, Sweden, PMLR, pp. 4095-4104.

Picon, A., Alvarez-Gila, A., Irusta, U. & Echazarra, J. 2020. Why deep learning performs better than classical machine learning. *Dyna Ing. Ind*, 95:119-122.

Pinaya, W.H.L., Vieira, S., Garcia-Dias, R. & Mechelli, A. 2020. Autoencoders. In: Machine Learning. Elsevier. pp. 193-208.

Prevedello, L.M., Halabi, S.S., Shih, G., Wu, C.C., Kohli, M.D., Chokshi, F.H., ... Flanders, A.E. 2019. Challenges related to artificial intelligence research in medical imaging and the importance of image analysis competitions. *Radiology: Artificial Intelligence*, 1(1):e180031.

Provost, F. & Fawcett, T. 2013. Data science and its relationship to big data and data-driven decision making. *Big data*, 1(1):51-59.

Rahmaty, M. 2023. Machine learning with big data to solve real-world problems. *Journal of Data Analytics*, 2(1):9-16.

Rani, R., Khurana, M., Kumar, A. & Kumar, N. 2022. Big data dimensionality reduction techniques in IoT: Review, applications and open research challenges. *Cluster Computing*, 25(6):4027-4049.

Real, E., Liang, C., So, D., & Le, Q. 2020. *AutoML-Zero: Evolving Machine Learning Algorithms from Scratch.* In: Proceedings of the 37th International Conference on Machine Learning (ICML), Vienna, Austria, PMLR, pp. 8007-8019.

Rehman, A.A. & Alharthi, K. 2016. An introduction to research paradigms. *International journal of educational investigations*, 3(8):51-59.

Rokach, L., Maimon, O. & Shmueli, E. 2023. Machine Learning for Data Science Handbook: Data Mining and Knowledge Discovery Handbook. 3rd ed. Cham: Springer.

Rosenblatt, F. 1958. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386.

Rosenblatt, F. 1961. Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms. Washington, D.C.: Spartan Books.

Rothman, D. 2020. *Artificial Intelligence By Example: Acquire Advanced AI, Machine Learning, and Deep Learning Design Skills.* 2nd ed. Birmingham, UK: Packt Publishing Ltd. ISBN: 978-1-83882-498-3.

Roy, D. & Dutta, M. 2022. A systematic review and research perspective on recommender systems. *Journal of Big Data*, 9(1):59.

Ruby, U. & Yendapalli, V. 2020. Binary cross entropy with deep learning technique for image classification. *Int. J. Adv. Trends Comput. Sci. Eng*, 9(10),

Rumelhart, D.E., Hinton, G.E., & Williams, R.J. 1985. *Learning Internal Representations by Error Propagation.* Technical Report, University of California, San Diego, CA, USA.

Saponara, S., Elhanashi, A. & Zheng, Q. 2021. Recreating fingerprint images by convolutional neural network autoencoder architecture. *IEEE Access*, 9:147888-147899.

Sarfi, A. 2023. *On Using Simulated Annealing in Training Deep Neural Networks.* Montreal, Canada: Concordia University.

Sedhain, S., Menon, A.K., Sanner, S., & Xie, L. 2015. AutoRec: Autoencoders Meet Collaborative Filtering. In: Proceedings of the 24th International Conference on World Wide Web (WWW 2015 Companion), Florence, Italy, ACM, pp. 111-112. DOI: 10.1145/2740908.2742726.

Serey, J., Alfaro, M., Fuertes, G., Vargas, M., Duran, C., Ternero, R., ... Sabattin, J. 2023. Pattern recognition and deep learning technologies, enablers of industry 4.0, and their role in engineering research. *Symmetry*, 15(2):535.

Serrano, L. 2021. *Grokking Machine Learning.* New York, NY, USA: Simon and Schuster. ISBN: 978-1-61729-591-1.

Sewak, M., Sahay, S.K. & Rathore, H. 2020. An overview of deep learning architecture of deep neural networks and autoencoders. *Journal of Computational and Theoretical Nanoscience*, 17(1):182-188.

Shehu, Y.I., Ruiz-Garcia, A., Palade, V. & James, A. 2018. Sokoto coventry fingerprint dataset. *arXiv preprint arXiv:1807.10609*,

Simon, H.A. 1996. *Models of My Life.* Cambridge, MA, USA: MIT Press. ISBN: 978-0-262-69191-1.

Song, C., Liu, F., Huang, Y., Wang, L., & Tan, T. 2013. *Auto-Encoder Based Data Clustering.* In: Proceedings of the 18th Ibero-American Congress on Pattern Recognition (CIARP), Havana, Cuba, Springer, pp. 117-124. DOI: 10.1007/978-3-642-41822-8_15.

Storcheus, D., Rostamizadeh, A., & Kumar, S. 2015. *A Survey of Modern Questions and Challenges in Feature Extraction.* In: Proceedings of the NIPS 2015 Workshop on Feature Extraction: Modern Questions and Challenges, Montreal, Canada, PMLR, pp. 1-18.

Sun, Y., Mao, H., Guo, Q. & Yi, Z. 2016. Learning a good representation with unsymmetrical auto-encoder. *Neural computing and applications*, 27:1361-1367.

Szandała, T. 2021. Review and comparison of commonly used activation functions for deep neural networks. *Bio-inspired neurocomputing*:203-224.

Talbi, E.-G. 2021. Automated design of deep neural networks: A survey and unified taxonomy. *ACM Computing Surveys (CSUR)*, 54(2):1-37.

Terayama, K., Sumita, M., Tamura, R. & Tsuda, K. 2021. Black-box optimization for automated discovery. *Accounts of Chemical Research*, 54(6):1334-1346.

Theis, L., Shi, W., Cunningham, A. & Huszár, F. 2022. Lossy image compression with compressive autoencoders. In: International conference on learning representations.Be morespecific]

Van Veen, F. 2019. The Neural Network Zoo. Available at: https://www.asimovinstitute.org/neural-network-zoo. Date of access: 14 March 2022.

Venkatesh, B. & Anuradha, J. 2019. A review of feature selection and its methods. *Cybernetics and information technologies*, 19(1):3-26.

Victoria, A.H. & Maragatham, G. 2021. Automatic tuning of hyperparameters using Bayesian optimization. *Evolving Systems*, 12(1):217-223.

Viering, T. & Loog, M. 2023. The Shape of Learning Curves: A Review. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, 45(06):7799-7819.

Vom Brocke, J., Hevner, A. & Maedche, A. 2020. Introduction to design science research. *Design science research. Cases*:1-13.

Vu, K.K., d'Ambrosio, C., Hamadi, Y. & Liberti, L. 2017. Surrogate-based methods for black-box optimization. *International Transactions in Operational Research*, 24(3):393-424.

Wang, D., Ha, M. & Zhao, M. 2022. The intelligent critic framework for advanced optimal control. *Artificial Intelligence Review*, 55(1):1-22.

Wang, Q., Ma, Y., Zhao, K. & Tian, Y. 2020. A comprehensive survey of loss functions in machine learning. *Annals of Data Science*:1-26.

Wang, X. & Zhu, W. 2024. Advances in neural architecture search. *National Science Review*, 11(8):nwae282.

Waring, J., Lindvall, C. & Umeton, R. 2020. Automated machine learning: Review of the state-of-the-art and opportunities for healthcare. *Artificial intelligence in medicine*, 104:101822.

Whang, S.E., Roh, Y., Song, H. & Lee, J.-G. 2023. Data collection and quality challenges in deep learning: A data-centric ai perspective. *The VLDB Journal*, 32(4):791-813.

Widrow, B. & Hoff, M.E. 1960. Adaptive switching circuits. In: IRE WESCON Convention Record. Los Angeles, CA: Institute of Radio Engineers, pp. 96–104.

Xie, L., Chen, X., Bi, K., Wei, L., Xu, Y., Wang, L., ... Zhang, X. 2021. Weight-sharing neural architecture search: A battle to shrink the optimization gap. *ACM Computing Surveys (CSUR)*, 54(9):1-37.

Xiong, Z., Cui, Y., Liu, Z., Zhao, Y., Hu, M. & Hu, J. 2020. Evaluating explorative prediction power of machine learning algorithms for materials discovery using k-fold forward cross-validation. *Computational Materials Science*, 171:109203.

Yadav, S.L. & Sohal, A. 2017. Comparative study of different selection techniques in genetic algorithm. *International Journal of Engineering, Science and Mathematics*, 6(3):174-180.

Yang, S., Tian, Y., He, C., Zhang, X., Tan, K.C. & Jin, Y. 2021. A gradient-guided evolutionary approach to training deep neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 33(9):4861-4875.

Yella, A. & Kondam, A. 2022. Big Data Integration and Interoperability: Overcoming Barriers to Comprehensive Insights. *Advances in Computer Sciences*, 5(1).

Ying, X. 2019. *An Overview of Overfitting and Its Solutions.* In: Journal of Physics: Conference Series, Vol. 1168, No. 2, Article 022022. IOP Publishing. DOI: 10.1088/1742-6596/1168/2/022022.

Yong, B.X. & Brintrup, A. 2022. Do autoencoders need a bottleneck for anomaly detection? *IEEE Access*, 10:78455-78471.

Yu, L. & Liu, H. 2004. Efficient feature selection via analysis of relevance and redundancy. *The Journal of Machine Learning Research*, 5:1205-1224.

Zacarias-Morales, N., Pancardo, P., Hernández-Nolasco, J.A. & Garcia-Constantino, M. 2021. Attention-inspired artificial neural networks for speech processing: A systematic review. *Symmetry*, 13(2):214.

Zeb, A., Soininen, J.-P. & Sozer, N. 2021. Data harmonisation as a key to enable digitalisation of the food sector: A review. *Food and Bioproducts Processing*, 127:360-370.

Zhang, S., Vassiliadis, V.S., Dorneanu, B. & Arellano-Garcia, H. 2023. Hierarchical multi-scale parametric optimization of deep neural networks. *Applied Intelligence*, 53(21):24963-24990.

Zhang, Z., Zhang, R., Li, Z., Bengio, Y., & Paull, L. 2020. Perceptual Generative Autoencoders. In: Proceedings of the 37th International Conference on Machine Learning (ICML 2020), Vienna, Austria, PMLR, pp. 11298-11306.

Zhao, Y., Chen, L., Chen, Z., & Yu, K. 2020. Semi-Supervised Text Simplification with Back-Translation and Asymmetric Denoising Autoencoders. In: Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 34, No. 05, pp. 9668-9675.

Zheng, Q., Yang, M., Tian, X., Wang, X. & Wang, D. 2020. Rethinking the Role of Activation Functions in Deep Convolutional Neural Networks for Image Classification. *Engineering letters*, 28(1),

Zhu, G., Xu, Z., Yuan, C., & Huang, Y. 2022. DIFER: Differentiable Automated Feature Engineering. In: I. Guyon, M. Lindauer, M. van der Schaar, F. Hutter, & R. Garnett, eds. Proceedings of the First International Conference on Automated Machine Learning, 25–27 July 2022, Johns Hopkins University, Baltimore, MD, USA. PMLR, pp. 17/1–17. DOI: 10.48550.

Zoph, B. & Le, Q.V. 2022. Neural Architecture Search with Reinforcement Learning. In: International Conference on Learning Representations (ICLR).

Zoph, B., Vasudevan, V., Shlens, J., & Le, Q.V. 2018. Learning Transferable Architectures for Scalable Image Recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Salt Lake City, UT, USA, IEEE, pp. 8697-8710. DOI: 10.1109/CVPR.2018.00907.

APPENDIX A: PYTHON CODE FOR THE AANNAD ALGORITHM

```
## 00111111
## This program serves as a prototype for the master's dissertation of
## Zander Boonzaaier (ID: 28749995)

# Importing general libraries
import re  # Regular expressions for text manipulation
import numpy as np  # Numerical computing library
import pandas as pd  # Data manipulation and analysis library
import random  # Random number generation
import time  # Time management utilities
import sys  # System-specific parameters and functions
import matplotlib.pyplot as plt  # Plotting library for visualizing data
import datetime  # Date and time manipulation utilities

# Importing TensorFlow and Keras libraries for building neural networks
import tensorflow as tf
import tensorflow.keras as keras
from tensorflow.keras.layers import Input, Dense, Dropout  # Layer types for
neural networks
from tensorflow.keras.models import Model, Sequential  # Neural network models
import tensorflow.keras.backend as K  # Backend operations for low-level
manipulation

# Additional imports from Keras for optimization and model saving
from keras.optimizers import Adam  # Adam optimizer for training the model
from keras.saving import register_keras_serializable  # For registering custom
objects with Keras
from keras.models import load_model  # To load pre-trained models

# Importing scikit-learn (sklearn) libraries for machine learning utilities
from sklearn import datasets  # Pre-built datasets in sklearn
from sklearn.metrics import confusion_matrix  # Evaluation metric for
classification accuracy
from sklearn.model_selection import train_test_split  # Utility to split datasets
into training and testing sets
from sklearn.neighbors import KNeighborsClassifier  # K-Nearest Neighbors
algorithm for classification

# Install required libraries (pyyaml and h5py) to save models in HDF5 format
!pip install pyyaml h5py

# Ensure that file-based operations are supported (i.e., for saving models,
working with directories)
import os

## This cell imports and initializes the necessary metrics for the model
configuration
```

```
# Input size for the model
input_size = 784  # Each image in the MNIST dataset consists of 28x28 pixels (784
total pixels)

# Training configuration
training_epochs = 1000000000  # Number of training epochs (currently set to an
extremely high value)
activation_function_encoder = 'relu'  # Activation function for the encoder
layers
activation_function_decoder = 'relu'  # Activation function for the decoder
layers

# Dataset size and train-test-validation (TTV) ratios
# In this example, we use a 55,000 training, 5,000 validation, and 10,000 testing
set split
# The split ratio is based on a study:
https://ieeexplore.ieee.org/abstract/document/9051147
dataset_size = 70000  # Total dataset size for MNIST
training_ratio = 0.7  # 70% of the data is used for training
testing_ratio = 0.2  # 20% of the data is used for testing
validation_ratio = 0.1  # 10% of the data is used for validation
## This section defines a custom loss function referred to as Mean Squared
Percentage Error (MSPE)

def mean_square_percentage_error(y_true, y_pred):
    """
    Custom loss function: Mean Squared Percentage Error (MSPE)
    MSPE is used to measure the accuracy of predictions as a percentage.

    Parameters:
    y_true: Actual true values
    y_pred: Predicted values from the model

    Returns:
    loss: The MSPE loss value, which will be minimized during training
    """

    # Calculate the difference between the true values and the predicted values
    loss = y_true - y_pred
    print(loss)  # Debugging: Print the loss (difference) for inspection

    # Avoid division by zero by using tf.math.divide_no_nan
    loss = tf.math.divide_no_nan(loss, y_true)  # Divide the error by the true
values to compute the percentage error
    print(loss)  # Debugging: Print the percentage error for inspection

    # Compute the mean of the squared percentage errors
    loss = K.mean(K.square(loss))
```

```
    ## Alternatively, this line can be used for a more concise implementation:
    ## loss = K.mean(K.square(((y_true - y_pred) / y_true)))

    return loss

## This cell is used to import and normalize the MNIST dataset

# Dataset utilized in this version: MNIST
from keras.datasets import mnist

# Load the MNIST dataset (returns training and test sets)
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# The loaded images are rescaled from 0-255 to 0-1
# Grayscale images in MNIST have pixel values from 0 to 255 (inclusive)
# Rescaling them to the range 0-1 to normalize the data for the model
x_train = x_train.astype('float32') / 255.  # Normalize training data
x_test = x_test.astype('float32') / 255.  # Normalize testing data

# Flatten the 28x28 images into 1D arrays of 784 pixels (28x28 = 784)
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))

# Create a validation set by selecting the last 5000 samples from the training
dataset
x_validation = x_train[-5000:]  # Validation data
y_validation = y_train[-5000:]  # Validation labels (though not used in
unsupervised learning)

# Reduce the training set to 35,000 samples (excluding validation data)
x_train = x_train[0:35000]
y_train = y_train[0:35000]

# In the MNIST experiment:
# The x_ datasets (x_train, x_test, x_validation) contain the image data.
# The y_ datasets (y_train, y_test, y_validation) contain the corresponding
labels.
# Since autoencoders are unsupervised, the labels are not used during training.

# Print the shapes of the datasets to confirm the splits
print("The training set consists of:", x_train.shape)
print("The testing set consists of:", x_test.shape)
print("The validation set consists of:", x_validation.shape)

# Validate that the maximum value of the pixel data is indeed 1, as intended
after normalization
print(np.max(x_train), np.max(x_test), np.max(x_validation))  # Ensure max value
is 1
```

```
# Note: We do not use the labels (y_ datasets) for training, as autoencoders work
unsupervised.

def initialize_autoencoder_architectures(number_of_initial_architectures):
    """
    This function generates a list of random autoencoder architectures.

    Parameters:
    number_of_initial_architectures (int): The number of autoencoder
architectures to generate.

    Returns:
    arr_autoencoder_architectures (list): A list of autoencoder architecture
strings.
    """

    # Initialize an empty list to store autoencoder architecture strings
    arr_autoencoder_architectures = []

    # Continue generating architectures until the required number is reached
    while len(arr_autoencoder_architectures) < number_of_initial_architectures:
        # Start constructing the autoencoder architecture string
        random_autoencoder_string = str(input_size) + "["

        # Randomly determine the number of encoder hidden layers (between 3 and
10)
        num_encoder_hidden_layers = random.randint(3, 10)
        random_autoencoder_string += str(num_encoder_hidden_layers) + "]"

        # Randomly determine the bottleneck dimensions (smallest layer)
        bottleneck_dimensions = random.randint(1, input_size)
        changing_limitation = input_size  # This value is updated to "shrink" the
autoencoder

        # Construct the encoder architecture by adding hidden layers
        for k in range(num_encoder_hidden_layers):
            # Shrink the layer size as we move toward the bottleneck
            changing_limitation = random.randint(bottleneck_dimensions,
changing_limitation)
            random_autoencoder_string += str(changing_limitation)

            # Add semicolons between layer sizes, but not after the last layer
            if k < num_encoder_hidden_layers - 1:
                random_autoencoder_string += ";"

        # Append bottleneck information and final formatting
        random_autoencoder_string += "#" + str(bottleneck_dimensions) + "=000"

        # Only add the architecture string if it is not already in the list
        if random_autoencoder_string not in arr_autoencoder_architectures:
```

```
            arr_autoencoder_architectures.append(random_autoencoder_string)

    # Print the generated autoencoder architectures for debugging purposes
    print(arr_autoencoder_architectures)

    # Return the list of autoencoder architecture strings
    return arr_autoencoder_architectures
@register_keras_serializable()
def get_lr_metric(optimizer):
    """
    Custom metric to track the learning rate during model training.

    Parameters:
    optimizer: The optimizer being used for model training. It is expected to
have a 'learning_rate' attribute.

    Returns:
    learning_rate: A function that returns the current learning rate of the
optimizer.
    """

    def learning_rate(y_true, y_pred):
        # Return the learning rate from the optimizer (does not depend on y_true
or y_pred)
        return optimizer.learning_rate

    return learning_rate
def compile_initial_autoencoder_models(arr_autoencoder_strings):
    """
    This function compiles and trains a series of autoencoder models based on a
list of architecture strings.

    Parameters:
    arr_autoencoder_strings (list): A list of strings representing different
autoencoder architectures.

    Returns:
    autoencoder_info: The last trained autoencoder model.
    history_info: The training history of the last autoencoder model.
    arr_trained_autoencoders (list): A list containing the performance details of
trained autoencoder models.
    """

    # List to store trained autoencoders and their performance information
    arr_trained_autoencoders = []
    autoencoder_info = ""
    history_info = ""

    # Loop through each autoencoder architecture string to create, compile, and
train models
```

```
    for k in range(len(arr_autoencoder_strings)):

        # First, a collection of callbacks is created
        print("Creating callbacks...")

        # Early stopping to prevent overfitting
        early_stopping_callback = tf.keras.callbacks.EarlyStopping(
            monitor="loss", min_delta=0.00001, patience=15, verbose=1, mode="min"
        )

        # TensorBoard callback for logging
        log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
        tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir,
histogram_freq=1)

        # Learning rate reduction on plateau
        reduce_lr = tf.keras.callbacks.ReduceLROnPlateau(
            monitor='loss', factor=0.2, patience=10, verbose=1, min_lr=1e-9,
min_delta=0.0001
        )

        print("Callbacks have been successfully created")

        # Create the autoencoder model from the architecture string
        autoencoder = create_autoencoder(arr_autoencoder_strings[k])

        # Initialize the Adam optimizer with a small learning rate
        optimizer_adam = Adam(learning_rate=1e-6)  # Initial learning rate for
autoencoders

        # Custom metric to track the learning rate during training
        lr_metric = get_lr_metric(optimizer_adam)

        # Compile the autoencoder with the necessary loss function and metrics
        autoencoder.compile(
            optimizer=optimizer_adam,
            loss="mean_squared_error",
            metrics=[lr_metric, mean_square_percentage_error,
"mean_squared_error", "mean_absolute_percentage_error"]
        )

        # Train the autoencoder using the training data and callbacks
        history = autoencoder.fit(
            x=x_train,
            y=x_train,
            epochs=training_epochs,
            batch_size=256,
            shuffle=True,
            validation_data=(x_validation, x_validation),
            callbacks=[reduce_lr, early_stopping_callback, tensorboard_callback],
```

```
            verbose=1
        )

        # Print the model summary for debugging purposes
        print(autoencoder.summary())

        # Record the performance of the autoencoder using the custom write
function
        autoencoder_performance = write_autoencoder_string(
            arr_autoencoder_strings[k],
history.history['mean_square_percentage_error'][-1]
        )
        arr_trained_autoencoders.append(autoencoder_performance)

        # Update the last autoencoder and training history for potential return
        autoencoder_info = autoencoder
        history_info = history

        '''# Evaluate the model on the test data using `evaluate`
        # Uncomment the following section if you want to evaluate on test data
after training
        print("Evaluate on test data")
        results = autoencoder.evaluate(x_test, x_test, batch_size=256)
        print("Test loss:", results)'''

        # Generate predictions for the test data (reconstruction of images)
        print("\n\nReconstructing testing images...\n")
        decoded_imgs = autoencoder.predict_on_batch(x_test)

        # Call a function to visualize the reconstructed images
        reconstruct_images(x_test, autoencoder, history, decoded_imgs)

        # Save the trained autoencoder model to an .h5 file
        savepath = autoencoder_performance + ".h5"
        autoencoder.save(savepath)
        print("Autoencoder saved at: " + savepath)

    # Return the last trained autoencoder, its training history, and the list of
all trained autoencoders
    return autoencoder_info, history_info, arr_trained_autoencoders
## This function creates a Keras model for autoencoders
## It receives the dimensions of the encoding layers, the image size, and the
number of layers.

def create_autoencoder(received_autoencoder_string):
    """
    Creates an autoencoder model based on the provided architecture string.

    Parameters:
```

```
    received_autoencoder_string (str): A string representing the autoencoder
architecture.
                                       Example format: "784[2]392;196#64=000"

    Returns:
    model (tf.keras.Model): The constructed autoencoder model.
    """

    # First, we extract the values from the string and convert them to usable
integers.
    # Example string: "784[2]392;196#64=000" -> input size, number of layers, and
bottleneck dimensions.
    autoencoder_array = re.split(r"[^a-zA-Z0-9\s]", received_autoencoder_string)
    autoencoder_array = list(map(int, autoencoder_array))

    input_size = autoencoder_array[0]  # Input layer size (e.g., 784 for MNIST)
    encoder_decoder_layers = autoencoder_array[1]  # Number of encoder/decoder
layers

    # Create an array to hold the dimensions of each encoder/decoder layer
    encoder_decoder_values = [0] * encoder_decoder_layers
    for k in range(encoder_decoder_layers):
        encoder_decoder_values[k] = autoencoder_array[k + 2]

    # Extract the bottleneck dimensions (smallest layer)
    bottleneck_dimensions = autoencoder_array[-2]

    # Define the input layer of the autoencoder
    input_img = Input(shape=(input_size,), name='Input_Layer')

    print("Number of layers in this architecture: " +
str(encoder_decoder_layers))

    # Randomly decide whether to include dropout layers, based on the specified
probability
    dropout_layers_active = random.randint(0, 1)  # 0 means no dropout, 1 means
dropout is active
    if dropout_layers_active == 1:
        print("This architecture includes dropout layers.")
    else:
        print("This architecture does not include dropout layers.")

    # Initialize a Sequential model to build the autoencoder
    model = Sequential()

    ## Encoder Construction ##
    if encoder_decoder_layers == 1:
        # If there's only 1 layer, directly add the bottleneck layer
        model.add(Dense(bottleneck_dimensions,
activation=activation_function_encoder, name='Bottleneck_Layer'))
```

```
    else:
        # Add the first encoder layer
        model.add(Dense(encoder_decoder_values[0],
activation=activation_function_encoder, name='Encoder_Layer_1'))
        if dropout_layers_active == 1:
            # Add dropout layer if active
            model.add(tf.keras.layers.Dropout(rate=0.15))

        # Add the remaining encoder layers
        for k in range(encoder_decoder_layers - 1):
            model.add(Dense(encoder_decoder_values[k+1],
activation=activation_function_encoder, name='Encoder_Layer_' + str(k+2)))
            if dropout_layers_active == 1:
                # Add dropout after each encoder layer if active
                model.add(tf.keras.layers.Dropout(rate=0.15))

        # Add the bottleneck layer
        model.add(Dense(bottleneck_dimensions,
activation=activation_function_encoder, name='Bottleneck_Layer'))

    ## Decoder Construction ##
    if encoder_decoder_layers == 1:
        # If there's only 1 layer, directly decode to the input size
        model.add(Dense(input_size, activation=activation_function_decoder,
name='Decoder_Layer_1'))
    else:
        # Add the first decoder layer
        model.add(Dense(encoder_decoder_values[-1],
activation=activation_function_decoder, name='Decoder_Layer_1'))

        # Add the remaining decoder layers
        for k in range(encoder_decoder_layers - 1):
            model.add(Dense(encoder_decoder_values[-2-k],
activation=activation_function_decoder, name='Decoder_Layer_' + str(k+2)))

        # Add the output layer that reconstructs the input size
        model.add(Dense(input_size, activation=activation_function_decoder,
name='Output_Layer'))

    # Return the constructed model
    return model
def write_autoencoder_string(received_autoencoder_string, loss_calculated):
    """
    This function updates the autoencoder string by appending the calculated loss
value.

    Parameters:
    received_autoencoder_string (str): The original autoencoder string (e.g.,
"784[2]392;196#64=000").
```

```
    loss_calculated (float): The loss value calculated during the training
process.

    Returns:
    updated_string (str): The updated autoencoder string with the new loss value.
    """

    # Split the original string at the '=' sign, keeping the first part
    updated_string, sep, old = received_autoencoder_string.partition('=')

    # Append the new loss value to the string
    updated_string = updated_string + "=" + str(loss_calculated)

    return updated_string

def mutate_autoencoder_string(received_autoencoder_string):
    """
    This function mutates an autoencoder architecture string in one of three
ways:
    1. Increases the number of hidden nodes (adds a new hidden layer).
    2. Decreases the number of hidden nodes (removes a hidden layer).
    3. Decreases the density of the bottleneck layer.

    Parameters:
    received_autoencoder_string (str): The original autoencoder string.

    Returns:
    mutated_string (str): The mutated autoencoder string.
    """

    # Split the autoencoder string into usable parts
    autoencoder_array = re.split(r"[^a-zA-Z0-9\s!.]",
received_autoencoder_string)

    # Reset the loss value in the string (since it's mutated, the previous loss
is irrelevant)
    autoencoder_array[-1] = 0

    # Convert the list elements into integers
    autoencoder_array = list(map(int, autoencoder_array))
    input_size = autoencoder_array[0]  # Input size (e.g., 784 for MNIST)
    encoder_decoder_layers = autoencoder_array[1]  # Number of encoder/decoder
layers

    # Initialize mutated string as the original one
    mutated_string = received_autoencoder_string

    # Randomly decide which mutation to apply (1 = increase nodes, 2 = decrease
nodes, 3 = change bottleneck size)
    mutation = random.randint(1, 3)
```

```
    if mutation == 1:  # 1. Increase the number of hidden nodes
        autoencoder_array[1] += 1  # Increase the number of hidden layers

        # Update the autoencoder string to reflect the new number of layers
        regex = r'\[.*?\]'
        mutated_string = re.sub(regex, '[' + str(autoencoder_array[1]) + ']',
received_autoencoder_string)

        # Add a new random hidden layer size between the bottleneck and the
previous layer
        mutated_string, sep, old = mutated_string.partition('#')
        mutated_string = mutated_string + ";" +
str(random.randint(autoencoder_array[-2], autoencoder_array[-3])) + "#" +
str(autoencoder_array[-2]) + "=000"

    elif mutation == 2:  # 2. Decrease the number of hidden nodes
        if autoencoder_array[1] > 1:  # Ensure at least one hidden layer remains
            autoencoder_array[1] -= 1  # Decrease the number of hidden layers

            # Update the autoencoder string to reflect the new number of layers
            regex = r'\[.*?\]'
            mutated_string = re.sub(regex, '[' + str(autoencoder_array[1]) + ']',
received_autoencoder_string)

            # Remove the last hidden layer from the string
            mutated_string, throw_away = mutated_string.rsplit(';', 1)
            mutated_string = mutated_string + "#" + str(autoencoder_array[-2]) +
"=000"

    elif mutation == 3:  # 3. Decrease the bottleneck layer size
        if autoencoder_array[-2] > 1:  # Ensure the bottleneck is not reduced to
zero
            autoencoder_array[-2] -= 1  # Decrease the bottleneck layer size

            # Update the autoencoder string to reflect the new bottleneck size
            regex = r'\#.*?\='
            mutated_string = re.sub(regex, '#' + str(autoencoder_array[-2]) +
'=', received_autoencoder_string)

            # Ensure the loss value remains reset after mutation
            mutated_string, throw_away = mutated_string.rsplit('=', 1)
            mutated_string = mutated_string + "=000"

    # Return the mutated autoencoder string
    return mutated_string
def compile_mutated_autoencoder_models(mutated_string):
    """
    Compiles, trains, and evaluates a mutated autoencoder model based on a
provided architecture string.
```

```
    Parameters:
    mutated_string (str): The mutated autoencoder architecture string.

    Returns:
    autoencoder_info (tf.keras.Model): The trained autoencoder model.
    history_info (tf.keras.callbacks.History): The training history of the
autoencoder.
    mutation_performance (str): The updated autoencoder string with the final
loss value.
    """

    # Initialize variables to store information about the autoencoder and its
performance
    loss = []
    autoencoder_info = ""
    history_info = ""

    ## First, create a collection of callbacks for training
    print("Creating callbacks...")

    # Early stopping: stops training if the loss does not improve for several
epochs
    early_stopping_callback = tf.keras.callbacks.EarlyStopping(
        monitor="loss", min_delta=0.00001, patience=15, verbose=1, mode="min"
    )

    # TensorBoard callback: logs data for visualization during training
    log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
    tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir,
histogram_freq=1)

    # ReduceLROnPlateau: reduces the learning rate if loss stops improving
    reduce_lr = tf.keras.callbacks.ReduceLROnPlateau(
        monitor='loss', factor=0.2, patience=10, verbose=1, min_lr=1e-9,
min_delta=0.0001
    )

    print("Callbacks have been successfully created.")

    # Create the autoencoder model based on the mutated architecture string
    autoencoder = create_autoencoder(mutated_string)

    # Set up the Adam optimizer with a very small initial learning rate
    optimizer_adam = keras.optimizers.Adam(lr=1e-6)
    lr_metric = get_lr_metric(optimizer_adam)

    # Compile the autoencoder with necessary loss function and metrics
    autoencoder.compile(
        optimizer=optimizer_adam,
```

```
        loss="mean_squared_error",
        metrics=[lr_metric, mean_square_percentage_error, "mean_squared_error",
"mean_absolute_percentage_error"]
    )

    # Train the autoencoder using the training data
    history = autoencoder.fit(
        x=x_train,
        y=x_train,
        epochs=training_epochs ,
        batch_size=256,
        shuffle=True,
        validation_data=(x_validation, x_validation),
        callbacks=[reduce_lr, early_stopping_callback, tensorboard_callback],
        verbose=1
    )

    # Display a summary of the autoencoder architecture
    print(autoencoder.summary())

    # Record the performance of the mutation
    mutation_performance = write_autoencoder_string(
        mutated_string, history.history['mean_square_percentage_error'].pop()
    )
    autoencoder_info = autoencoder
    history_info = history

    '''# Evaluate the model on test data (optional)
    print("Evaluate on test data")
    results = autoencoder.evaluate(x_test, x_test, batch_size=256)
    print("Test loss, test accuracy:", results)

    # Reconstruct the testing images
    print("\n\nReconstructing the testing images\n")
    decoded_imgs = autoencoder.predict_on_batch(x_test)
    reconstruct_images(x_test, autoencoder, history, decoded_imgs)'''

    # Save the trained autoencoder model
    savepath = mutation_performance + ".h5"
    autoencoder.save(savepath)
    print("Autoencoder saved at: " + savepath)

    # Return the trained autoencoder model, its training history, and the
mutation performance
    return autoencoder_info, history_info, mutation_performance
def reconstruct_images(original_dataset, autoencoder, history, decoded_imgs):
    """
    This function visualizes the original and reconstructed images side by side,
    and also plots the training and validation loss across epochs.
```

```
    Parameters:
    original_dataset (numpy array): The original dataset used for training (e.g.,
MNIST).
    autoencoder (tf.keras.Model): The trained autoencoder model.
    history (tf.keras.callbacks.History): The training history of the autoencoder
model.
    decoded_imgs (numpy array): The reconstructed images produced by the
autoencoder.
    """

    # Number of images to display
    n = 10

    # Create a figure to display the original and reconstructed images
    fig = plt.figure(figsize=(20, 4))

    for i in range(n):
        # Display the original image
        ax = plt.subplot(2, n, i + 1)
        plt.imshow(original_dataset[i + 10].reshape(28, 28))  # Offset index for
better variety
        plt.title("Original")
        plt.gray()  # Set the color map to grayscale
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)

        # Display the reconstructed image
        ax = plt.subplot(2, n, i + 1 + n)
        plt.imshow(decoded_imgs[i + 10].reshape(28, 28))
        plt.title("Reconstructed")
        plt.gray()
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)

    plt.show()

    # Plot the loss across epochs for both training and validation
    plt.plot(history.history['loss'])  # Training loss
    plt.plot(history.history['val_loss'])  # Validation loss
    plt.title('Model Loss Overview')
    plt.ylabel('Loss')
    plt.xlabel('Epoch')
    plt.legend(['Training Loss', 'Validation Loss'], loc='upper left')
    plt.show()
def load_autoencoder_model(filepath, optimizer):
    """
    Loads a pre-trained autoencoder model from the specified file path.

    Parameters:
    filepath (str): The path to the saved autoencoder model file (.h5 format).
```

```
    optimizer (tf.keras.optimizers.Optimizer): The optimizer used in the model,
needed for the learning rate metric.

    Returns:
    model (tf.keras.Model): The loaded autoencoder model with custom metrics.
    """

    # Retrieve the learning rate metric using the provided optimizer
    lr_metric = get_lr_metric(optimizer)

    # Load the model, ensuring that the custom objects (like the learning rate
metric and custom loss function) are included
    return load_model(filepath, custom_objects={
        'learning_rate': lr_metric,
        'mean_square_percentage_error': mean_square_percentage_error
    })

def main_function():
    """
    Main function to initialize, train, mutate, and evaluate autoencoder models.
    The process includes generating initial autoencoder architectures, compiling
them,
    mutating the best-performing models, and testing the final results.
    """

    ## Number of initial autoencoders to create
    int_number_of_initial_architectures = 25

    ## Number of final autoencoders to keep after mutation
    int_number_of_final_architectures = 5

    ## Initialize a list to store the autoencoder architectures
    arr_autoencoder_strings = []

    print("Welcome to the demonstration\n")
    print("First, we create the initial set of autoencoder architectures\n")
    arr_autoencoder_strings =
initialize_autoencoder_architectures(int_number_of_initial_architectures)  ##
MODULE 1

    print(x_test)  ## Display x_test (validation of data)

    print("\n\nNext, we compile and train the initial set of autoencoders\n")
    autoencoder, history, arr_trained_autoencoders =
compile_initial_autoencoder_models(arr_autoencoder_strings)  ## MODULE 2
    arr_current_best_autoencoders = arr_trained_autoencoders

    ## Begin the mutation process. We will mutate for about 100 iterations (or
time-limited)
    arr_tested_autoencoders = arr_current_best_autoencoders
```

```
    print(arr_current_best_autoencoders)

    # Record the start time to enforce the time limit of 4 hours
    start_time = time.time()
    time_limit = 4 * 60 * 60  # 4 hours in seconds

    for k in range(int_number_of_final_architectures):
        # Check the elapsed time to enforce the time limit
        elapsed_time = time.time() - start_time
        if elapsed_time > time_limit:
            print("Time limit of 4 hours reached. Stopping further processing.")
            break

        # Select a random candidate from the current best autoencoders
        str_mutation_candidate = np.random.choice(arr_current_best_autoencoders)
        print(str_mutation_candidate)

        # Mutate the selected autoencoder
        str_mutation_candidate =
mutate_Autoencoder_String(str_mutation_candidate)

        # Ensure the mutated autoencoder hasn't been tested before
        if str_mutation_candidate not in arr_tested_autoencoders:
            arr_tested_autoencoders.append(str_mutation_candidate)
            print(str_mutation_candidate)

            # Compile and train the mutated autoencoder
            autoencoder, history, mutation_performance =
compile_mutated_autoencoder_models(str_mutation_candidate)
            print(mutation_performance)

            # Check if the mutation performs better than the current best, and
update the best list
            if float(arr_current_best_autoencoders[-1].split('=')[-1]) <
float(mutation_performance.split('=')[-1]):
                del arr_current_best_autoencoders[-1]  # Remove the worst-
performing autoencoder
                arr_current_best_autoencoders.append(mutation_performance)  # Add
the new better-performing mutation

            # Sort the best autoencoders by their performance (loss value)
            arr_current_best_autoencoders.sort(key=lambda x: x.split('=')[1])
            print(arr_current_best_autoencoders)

    print("\n\n\n\nNOW FOR THE BIG LOADING AND TESTING PHASE!!\n")
    str_best_architecture = arr_current_best_autoencoders[0]
    print("THE BEST ARCHITECTURE WAS: " + str_best_architecture)
    print("Checkpoint 1")
```

```
    # Prepare the best architecture's file path
    str_best_architecture = str_best_architecture + ".h5"
    print("Checkpoint 2")

    # Load the best-performing autoencoder model
    optimizer_adam = Adam(lr=1e-6)
    reconstructed_model = load_autoencoder_model(str_best_architecture,
optimizer_adam)

    print("Checkpoint 3")

    # Evaluate the best autoencoder on the test data
    print("Evaluate on test data")
    results = reconstructed_model.evaluate(x_test, x_test, batch_size=256)
    print("Test loss, test accuracy:", results)

    # Reconstruct and display the testing images
    print("\n\nReconstructing the testing images\n")
    decoded_imgs = reconstructed_model.predict_on_batch(x_test)
    reconstruct_images(x_test, reconstructed_model, history, decoded_imgs)

main_function()
```

APPENDIX B: CONFIRMATION OF LANGUAGE EDITING

This serves to confirm that I, Isabella Johanna Swart, registered with and accredited as professional translator by the South African Translators' Institute, registration number 1001128, language edited the following dissertation (the Reference List was edited in accordance with the Harvard style of the NWU Referencing Guide):

Automated autoencoder neural network architecture design

by

Z Boonzaaier

Dr Isabel J Swart

Date: 21 February 2025

23 Poinsettia Close
Van der Stel Park
Dormehlsdrift
GEORGE
6529
Cell: 082 718 4210
e-mail: isaswart@telkomsa.net