University of Maribor

Faculty of Electrical Engineering
and Computer Science

Sašo Pavlič

Construction of deep neural networks using swarm intelligence to detect anomalies

Gradnja globokih nevronskih mrež s pomočjo inteligence rojev za detekcijo anomalij

Master's thesis

Maribor, September 2021

University of Maribor

Faculty of Electrical Engineering
and Computer Science

Construction of deep neural networks using swarm intelligence to detect anomalies

Gradnja globokih nevronskih mrež s pomočjo inteligence rojev za detekcijo anomalij

Student:	Sašo Pavlič
Study programme:	Second cycle bologna study program of Informatics and technologies of communication
Module of study program:	Security of information systems and security management
Mentor:	doc. dr. Sašo Karakatič
Proofreader:	Bojan Keevill

Zahvala

Najprej bi se rad zahvalil svojemu mentorju, doc. dr. Sašu Karakatiču za konstantno strokovno pomoč in usmeritve v procesu raziskovanja in pisanja magistrskega dela.

Iskreno bi se rad zahvalil svoji celotni družini, ki me je v času študija podpirala in navdihovala na moji izobraževalni poti. Prav tako bi se rad zahvalil svojim prijateljem in sodelavcem.

Acknowledgment

First of all, I would like to thank my mentor, doc. dr. Sašo Karakatič for constant professional support and guidance in the process of researching and writing a master's thesis.

I would like to sincerely thank my entire family for supporting and inspiring me on my educational journey during my studies. I would also like to thank my friends and co-workers.

Gradnja globokih nevronskih mrež s pomočjo inteligence rojev za detekcijo anomalij

Ključne besede: iskanje arhitektur nevronske mreže, strojno učenje, inteligenca rojev

UDK: 004.85(043.2)

Povzetek

Umetna inteligenca (angl. Artificial intelligence) postaja vse bolj dovršena in se vedno bolj uporablja v storitvah, ki jih uporabljamo vsak dan. Aplikacije, ki so se bile še pred desetletji skoraj neizvedljive, predvsem zaradi izgradnje njihove logike (samovozeča vozila, predlagane vsebine, sinteza govora …), postajajo danes izvedljive z algoritmi, ki so zmožni sami zgraditi model odločanja za podano težavo. Umetna inteligenca bo v prihodnosti poglavitno orodje, ki ga bomo uporabljali za reševanje vse bolj zahtevnih vsakodnevnih težav.

Pomembno vlogo pri reševanju teh težav ima strojno učenje (angl. Machine learning), ki z globokim učenjem (angl. Deep learning) gradi globoke nevronske mreže (angl. Deep neural networks). Tovrstne mreže temeljijo na posnemanju poenostavljenega delovanja bioloških možganov in so zelo učinkovite za reševanje določenih težav, kajti same prilagajajo parametre nevronov ob učenju. Vendar pa je uspešnost učenja odvisna predvsem od tega, kako arhitekt zasnuje arhitekturo globoke nevronske mreže in kako so nastavljeni parametri mreže. S temi nastavitvami omejimo gradnjo nevronske mreže na izkušnje arhitekta, namesto da bi algoritem sam ugotovil, kakšne nastavitve so najbolj primerne za podano težavo. Znanost je za reševanje te težave začela aplicirati algoritme po vzoru iz narave (angl. Nature inspired algorithms) za izgradnjo nevronskih mrež z nevroevolucijo (angl. Neuroevolution). Proces nevroevolucije išče in optimizira ustrezno arhitekturo nevronskih mrež za reševanje specifične težave. Eden od algoritmov po vzoru iz narave so algoritmi inteligence rojev, ki s posnemanjem vedenja delcev (npr. mravelj) v naravi iščejo najboljšo možno rešitev za podano težavo. Pri nevroevoluciji je rešitev arhitektura modela nevronske mreže. Takšen proces je inovativen predvsem za težave pri izgradnji arhitekture nevronskih mrež, pri katerih poznamo le vhodne podatke (začetek) in končno stanje (cilj), ne pa procesa, ki ga moramo izvesti na naši poti. Takšen način imenujemo nenadzorovano učenje (angl. Unsupervised learning). Konkretni predstavnik takšnega nevronskega modela je avtomatski kodirnik (angl. Autoencoder), ki sprejme vhodne podatke, izvede proces in vrne izhodne podatke, cilj tega modela je, da so si vhodni in izhodni podatki čim bolj podobni, kajti zanimajo nas le koraki

procesa. S tem lahko opazujemo, kako se je model naučil predelati podatke, da so si čim bolj podobni kljub izvajanju operacij nad njimi. Če pride do prevelikega odstopanja, pa lahko to ovrednotimo kot anomalijo (angl. Anomaly detection). S tem procesom učenja iskanja arhitektur nevronskih mrež lahko ustvarimo računalniške sisteme, ki delujejo tako kot živa bitja. Primer: »Kako živo bitje opredeli spremembe v okolju kot razlog za strah ali užitek?«.

V magistrskem delu se bomo osredotočili na spoznavanje in implementacijo sistema za avtomatsko gradnjo arhitektur nevronskih mrež. Naš program nosi ime AutoDaedalus. Ta program temelji na uporabi inteligence rojev, s katero definiramo algoritem za odločitev komponent v nevronski mreži. Te komponente so lahko vse od tipa nevronov, strukture nivojev, aktivacijskih funkcij do dimenzije izhoda nevronov. Tip arhitekture nevronske mreže, ki jo v našem primeru gradi AutoDaedalus, je avtokoder. Ta arhitektura je prepoznana po tem, da se vhodni podatki zakodirajo v latentni prostor, nato pa dekodira nazaj v izhodne podatke. AutoDaedalus v svojem iskanju najuspešnejše arhitekture nevronskega modela uporablja matriko, kot je razlika med vhodom in izhodom. Na koncu imamo cilj, da dobimo najboljši nevronski model za rekonstrukcijo vhodnih podatkov. S tem ko se je nevronski model naučil ustrezno zakodirati in dekodirati dan tip podatkov, ga lahko uporabimo za iskanje anomalij. Razlog za to je verjetnost, da nevronski model ne bo znal ustrezno zakodirati in dekodirati tipa podatkov, za katerega ni bil naučen. V našem primeru smo nevronske modele učili na podatkih MNIST, v katerem lahko najdemo slike ročno napisanih številk od 0 do 9. Če smo nevronski model učili na 99 % slikah enic (1) in 1 % slik ničel (0), smo želeli, da se model nauči označiti enke (1) kot normalne in ničle (0) kot anomalije v podatkih.

Magistrsko delo je razdeljeno na 8 poglavij. Začne se s poglavjem, kjer se osredotočimo na spoznavanje umetnih nevronskih mrež, pri čemer se podrobneje spoznamo s strojnim učenjem in evolucijsko gradnjo nevronskih mrež. V tem poglavju želimo pridobiti potrebno znanje za učenje tovrstnih mrež na podatkih. Ob tem spoznamo različne tehnike učenja in kakšni so potrebni procesi, ko se model nevronske mreže uči na učni množici. Ker so nevronske mreže tesno povezane z globokim učenjem, spoznamo tudi to tehniko. Poglavje nadaljujemo s podrobnejšim pregledom nevronskih mrež in nevronov, ki jih sestavljajo. Ogledamo si, iz katerih komponent je sestavljen nevron in kako potujejo podatki skozi njega. Pri tem spoznamo, da nevronsko mrežo sestavljajo posamezni nivoji (vhodni, skriti, izhodni), ki so skupek nevronov. Vsak nevron se vede kakor električno stikalo, ki se vključuje ali izklaplja, kadar podatki potujejo skozi nevronsko mrežo. To vedenje je določeno kot rezultat aktivacijske funkcije, s katero lahko nadzorujemo celotno vedenje nevronske mreže. Ker pa je ta tok podatkov pomemben za učenje nevronske mreže, spoznamo prav

tako različne arhitekture nevronske mreže. Te arhitekture se razlikujejo po svoji strukturi in po toku podatkov skozi nevrone. Ker nas v magistrski nalogi zanima predvsem gradnja nevronskih mrež, si v naslednjem podpoglavju ogledamo, kako poteka evolucijski proces gradnje. Vse se začne z iskalnim prostorom, ki definira vse možne generirane arhitekture za izgradnjo in optimizacijo nevronskega modela. Pregledamo razliko, kako poteka iskanje arhitekture med človekom (znanstvenikom) in avtomatskim sistemom (angl: neural architecture search (NAS). Nadaljujemo strategijo iskanja, s katero se ustvarjajo kandidati za arhitekturo nevronskega modela. V naši nalogi podrobneje spoznamo konkreten primer algoritma (angl: ant colony optimisation (ACO)), ki ga uporabimo za iskanje arhitektur. Kot tretjo komponento NAS-a spoznamo evolucijsko strategijo, s katero si pomagamo, ko želimo dobiti povratne informacije za optimizacijo iskalne strategije. Takrat moramo izmeriti, oceniti ali predvideti uspešnost vsakega otroka arhitekture.

V tretjem poglavju preidemo k spoznavanju anomalij in načinov, kako jih lahko prepoznamo v podatkih. V nekaj predstavljenih primerih si ogledamo, kakšne tipe anomalij poznamo in kako se med seboj razlikujejo. Prav tako ugotovimo, da se detekcija anomalij razlikuje od tipa strojnega učenja, kjer je odvisno, kakšni so podatki za učenje nevronskega modela. Podatki lahko vsebujejo jasno označeno mejo med normalnimi primerki in anomalijami ali pa je ta meja nepoznana. Na podlagi tega se odločimo, kateri tip strojnega učenja bo uporabljen, kajti od tega je odvisen izhod algoritma.

V četrtem poglavju podamo primer praktične rešitve za odkrivanje anomalij s strojnim učenjem. Predstavimo poseben tip nevronske mreže, to je avtokoder. Kakor že predhodno omenjeno je glavna značilnost tega tipa nevronske mreže, da je sposoben originalne podatke zakodirati, nato pa jih dekodirati nazaj v originalno obliko. Kakovost nevronskega modela avtokoderja se pokaže ob primerjavi rekonstrukcije z originalom. Izbira avtokoderja tako sovpada s težavo zaznavanja anomalij. V našem delo se osredotočimo na odkrivanje anomalij, kjer ne poznamo meje med normalnimi podatki in anomalijami. Proces je relativno preprost, nevronski model avtokoderja naučimo na neki množici podatkov. Za to množico je potrebno, da je večina podatkov normalnih, nekaj pa anomalij. Ob tem procesu se bo model naučil zelo dobro obdelati normalne podatke, za anomalije pa se ne bo dobro izkazal. S takšno uporabo lahko model nevronske mreže učimo brez nadzora. V tem poglavju prav tako spoznamo, da je uspešnost obdelave podatkov odvisna od globine avtokoderja (števila nivojev) in tipa. Še vedno pa je treba arhitekturo avtokoderja ročno izdelati, zato si v naslednjem podpoglavju ogledamo, kako lahko to storimo s pomočjo inteligence rojev. V našem primeru z uporabo algoritma ACO. Algoritem uporablja mravlje kot reprezentacijo

nevronskega modela. Arhitektura, ki je zgrajena iz več nivojev, pa je reprezentacija poti, ki jo mravlja prehodi. Tako mravlje vedno začnejo na vhodnem nivoju, nato pa glede na odločitev algoritma vsaka izbere svoj naslednji tip nivoja z atributi (velikost, aktivacijska funkcija) in tako naprej, dokler ni dosežena *X* globina nevronskega modela. Enak proces se ponovi na strani koderja pa tudi dekoderja. Mravlja ob hoji skozi vozlišča za sabo spušča feromon. Več mravelj, ki se bo odločilo za dano pot, bo spustilo več feromona na tleh, posledično pa bo gostota večja, kar bo privabilo še več mravelj. Ta ideja izvira iz naravnega vedenja mravelj v koloniji. Kadar mravlje iščejo hrano, se izkaže, da je najbolje, da mravlje skupaj najdejo najkrajšo pot med gnezdom in hrano. V našem umetnem primeru se algoritem nadaljuje tako, da kadar vse mravlje opravijo svoje delo (generiranje arhitekture avtokoderja), se ta pretvori v ustrezen objekt v knjižnici Keras. Na tem objektu lahko nato izvedemo učenje in vrednotenje uspešnosti. Na podlagi rezultatov lahko določimo najuspešnejšo mravljo, kar v našem primeru predstavlja model avtokoderja, ki ustvari najmanjšo razliko med originalom in rekonstrukcijo vhoda.

Nadaljujemo s petim poglavjem, kjer smo predstavili implementacijo programa AutoDaedalus. Na začetku smo začeli s predstavitvijo razloga za implementacijo avtomatskega iskanja nevronskih arhitektur v primerjavi z ročno izdelanim. Navedli smo omejitve pri izdelavi, ki so bile razdeljene na strojno opremo, ki smo jo uporabljali pri implementaciji in pri poznejšem testiranju, ter človeške vire. AutoDaedalus je prav tako omejen pri tipu arhitektur nevronskih mrež, ki jih lahko izdela. To je le avtokoder arhitektura. Naj omenimo, da je tip avtokoderja omejen na plitki in globoki model. S tem smo si določili začetne meje, v katerih bo deloval naš program. V naslednjem podpoglavju so omenjena orodja, okvirji in paketi programske opreme, ki je bila uporabljena. Ker je strojno učenje, ki poteka na grafičnih karticah, hitrejše, smo namestili ustrezno programsko opremo za učenje na grafičnih karticah. Ob nadaljevanju smo si podrobneje ogledali celotni tok programa AutoDaedalus, kjer smo ugotovili, da se vse začne z uporabnikom, ki s pomočjo konfiguracijske datoteke nastavi parametre za iskanje arhitektur. S tem omejimo iskanje arhitektur znotraj predvidenega območja. Nadaljujemo pripravo nabora podatkov, kjer je treba določiti, kateri primerki podatkov bodo normalni in kateri anomalije. Razmerje se lahko določi s parametrom v konfiguracijski datoteki. Na koncu postopka dobimo nabor podatkov razdeljen na učno in testno množico. Tok programa AutoDaedalus se nadaljuje z inicializacijo potrebnih objektov za knjižnici Tensorflow in Keras. Nato pa preidemo na glavni del programa, kjer smo uporabili odprtno kodno rešitev DeepSwarm, ki uporablja algoritem ACO. Kot že omenjeno smo ta del kode spremenili za naše potrebe. DeepSwarm je pristojen predvsem za vodenje iskanja nevronskih arhitektur, shranjevanja kreiranih modelov, prikazovanje infografike in zaganjanje ACO-algoritma na podlagi konfiguracijske datoteke.

DeepSwarm skrbi, da kadar je zgenerirana arhitektura modela, se dodajajo določeni nivoji. Ti so potrebni, da je možno na koncu model koderja in dekoderja združiti v en sam model avtokoderja. Tega uporabimo za učenje na učni množici. Skozi trening modela se nam izpisujeta natančnost in izguba. Ob koncu učenja pa se model ovrednoti z matrikami za izračun matrike zmede, F-mere in krivulje ROC-AUC. S pomočjo izpisanih metrik lahko ocenimo, kako se je končna arhitektura avtokoderja obnesla za prepoznavanje anomalij in kateri nevronski model je bil pri tem bil najuspešnejši.

V šestem poglavju preidemo do eksperimentalnega dela, kjer smo želeli preizkusiti, kako se naša implementacija programa AutoDaedalus obnese v primerjavi z ročno zgrajeno arhitekturo avtokoderjev. Na eksperimentalni del smo se pripravili tako, da smo zasnovali dva različna eksperimenta. V prvem smo želeli preizkusiti obe metodi na podatkovni množici, kjer so enke (1) normalni podatki, ničle (0) pa anomalije. V drugem eksperimentu pa so bili normalni podatki med 1 in 9 in le ničle (0) anomalije. Pri tem smo določili 0.9 kot kvantil dovoljene napake med originalnimi podatki in anomalijami. Obe metodi smo preizkušali med seboj do maksimalne globine 5. Generirani nevronski modeli so se razlikovali predvsem po številu nevronov v nivoju in aktivacijskih funkcijah glede na metodo. Med primerjavo metod na prvem eksperimentu smo ugotovili, da se uspešnost modelov ni kaj bistveno razlikovala in da sta obe metodi dosegali odlične rezultate glede na rezultate vseh matrik. Na koncu je bila ročna metoda za malenkost uspešnejša. V nasprotju z drugim eksperimentom so razlike postale hitreje vidne. Tukaj je šlo za bistveno težji eksperiment, kajti s tako preprosto arhitekturo nevronske mreže pri dani težavi se hitro pokaže, da se model ni sposoben naučiti tako kakovostno kot v prejšnjem eksperimentu razlikovati med normalnimi podatki in anomalijami. Zato sta imela odločilno vlogo izbira zaporedja nivojev in kombinacija aktivacijskih funkcij v nevronskih modelih. V tem eksperimentu je AutoDaedalus zgradil boljšo arhitekturo za prepoznavanje anomalij. To se je zelo poznalo na številu pravilno identificiranih anomalij, ki je bilo v najboljšem modelu več kot dvakrat večje kot v najboljšem modelu pri ročni metodi. Kadar pa primerjamo vse generirane nevronske modele po obeh metodah, pa so bili po matrikah F1-mera in AUC vrednosti modelov po naši metodi boljši za 3,5 %.

V predzadnjem poglavju so diskusija, komentiranje eksperimentov in sprejemanje hipotez. Glavne iztočnice iz tega poglavja so predvsem, da lahko sistemi za avtomatsko kreiranje arhitektur nevronskih modelov izdelajo vsaj enako dobre modele, v nekaterih primerih pa celo boljše. Ti sistemi nam odpirajo vrata, ki nas velikokrat ovirajo, kadar pride do tega, da se boljša rešitev lahko skriva izven logičnih konceptov in sprejetih praks pri načrtovanju nevronskih arhitektur. To je

pomembno predvsem takrat, ko je vključeno nenadzorovano učenje, kajti težko je zasnovati uspešno arhitekturo, če je že razumevanje podatkov nejasno.

V zadnjem poglavju sledijo komentiranje in zaključki vseh poglavij. Nekaj besed namenimo tudi vključevanju pridobljene teorije v našo implementacijo in poznejše testiranje skozi eksperiment. Na koncu želimo sporočiti bralcu, da so možnosti razvoja in uporabe sistemov NAS, osnovanih na inteligenci rojev, še velike.

Construction of deep neural networks using swarm intelligence to detect anomalies

Keywords: neural architecture search, machine learning, swarm intelligence

UDK: 004.85(043.2)

Abstract

The design of neural network architecture is becoming more difficult as the complexity of the problems we tackle using machine learning increases. Many variables influence the performance of a neural model, and those variables are often limited by the researcher's prior knowledge and experience. In our master's thesis, we will focus on becoming familiar with evolutionary neural network design, anomaly detection techniques, and a deeper knowledge of autoencoders and their potential for application in unsupervised learning. Our practical objective will be to build a neural architecture search based on swarm intelligence, and construct an autoencoder architecture for anomaly detection in the MNIST dataset.

Table of Contents

Table of Figures

Table of Tables

List of abbreviations

AI Artificial intelligence

ML Machine learning

DL Deep learning

NN Neural network

ANN Artificial neural network

DNN Deep neural network

CNN Convolutional neural network

NAS Neural architecture search

SL Supervised learning

UL Unsupervised learning

RL Reinforcement learning

GA Genetic algorithm

EC Evolutionary computation

EA Evolutionary algorithm

ACO Ant colony optimisation

ACS Ant colony system

EvoAE Evolving autoencoders

Tanh Hyperbolic tangent

ReLU Rectified Linear Unit

API Application programming interface

ROC Receiver operating characteristic

AUC Area under the curve

TPR True positive rate

FPR False positive rate

MSE Mean squared error

MSE Mean absolute error

TP True positive

FN False negative

FP False positive

TN True negative

TPR True negative rate

FNR False negative rate

TNR True negative rate

FPR False positive rate

CUDA Compute Unified Device Architecture

RAM Random Access Memory

CPU Central Processing Unit

GPU Graphical Processing Unit

SW Software

Chapter I

1 INTRODUCTION

Artificial intelligence (AI) is becoming more sophisticated and is deployed in services that we use every day. Applications that were unfeasible a decade ago, due to the complex nature of their logic (self-driving vehicles, personalised content, speech synthesis, etc.), are now becoming feasible with algorithms capable of building a decision model for a given problem. Artificial intelligence will be the primary tool we employ in the future to solve everyday challenges.

Machine learning (ML) plays an important role in solving these problems by building deep neural networks (DNN) through deep learning (DL). Such networks mimic the function of biological brains and are extremely successful in solving specific problems because they modify neuronal parameters during learning on their own. However, the success of learning is mostly determined by the DNN architecture and the network parameters set by the architect. With these parameters, we limit the neural network (NN) design to the architect's experience rather than the algorithm selecting which settings are most appropriate for a given problem. To address this issue, computer scientists have begun to design biomimetic algorithms to generate NNs by neuroevolution. The neuroevolutionary method identifies and optimises the best NN design to solve a certain problem. Swarm intelligence (SI) algorithms, for example, seek the best possible solution to a given problem by simulating the behaviour of natural organisms (e.g., ants).

The solution rendered by neuroevolution represents the architecture of the NN model. Such a method is novel, particularly for challenges in the construction of NNs, for which we only know the input data (start) and the final state (target), but not the cognitive procedural steps that must be accomplished in between. This is known as unsupervised learning. An autoencoder is a concrete example of a NN model that receives input data, performs a process, and returns output data. Because we are only interested in the cognition, the goal of this model is to make the input and output data as similar as possible. We can indirectly observe that the model has learned to process

the data correctly, if the input and output data are similar despite the operations performed on them. However, if there is a lot of variation we classify it as anomalous (Anomaly detection).
With such a method for the discovery of novel NN designs, we can build computer systems which we cannot understand the operation of but know how they should behave. E.g. "How does a living system define changes in the environment as a source of fear or pleasure?"

Goals:

1. Implement a neural architecture search (NAS) for anomaly detection.
2. Use a swarm intelligence algorithm to optimise the search space when creating neural network (NN) models.
3. Allow an unsupervised machine learning algorithm to make decisions that mark the threshold between normal and anomalous data instances.
4. Compare automatically generated and manually created neural network models for anomaly detection.
5. Open-source project to engage further research activity in this field.

Research questions:

RQ1: Are automatically generated NN models comparable to manually created ones in terms of anomaly detection?

RQ2: What percentage of anomalies inserted into the dataset is enough for a NN model to learn from?

RQ3: Can swarm intelligence algorithms be used to effectively search for autoencoder architecture?

Hypotheses based on the research questions:

H1: The total number of metrics is greater in automated models than in manual ones.

H2: The 1% of anomalies in the dataset is enough during the learning phase to detect half of them in the 0.9 quantiles during testing.

H3: The ant colony optimisation (ACO) algorithm can be used to construct useful autoencoder architectures.

The thesis is derived from the above research questions and hypotheses.

The NAS technique with a swarm intelligence search strategy can design novel NN architectures for a single objective search, with little or no help from human experts.

1.1 Chapter contents

This Master's thesis comprises 8 chapters. We will learn about ML in the second chapter, which continues with an overview of artificial neural networks, the parameters required for their operation, and how evolutionary neural networks are built. The third chapter presents the knowledge that is necessary for anomaly detection, such as the definition of various types of anomaly and how to detect each of them with the help of ML. The main objective of this work is presented in the fourth chapter, as a computational system that is capable of performing the evolutionary construction of new NN models to detect anomalies in a dataset. In this chapter, we also cover the NN type of autoencoder, with the objective of learning how they operate and how we can use them for anomaly detection. We learn about the ant colony optimisation method, which serves as our primary architecture for construction of the autoencoder. In the fifth chapter, we use the acquired knowledge to develop a programme that incorporates an ACO-based NAS for anomaly detection. This chapter includes the programme overflow and a detailed explanation of the components. The sixth chapter is based on experiments and compares manual construction of a NN with our implementation of AutoDaedalus. We answer our research questions and confirm our hypotheses on the basis of experimental data. The seventh chapter presents in-depth debates, and the eight chapter summarises the thesis.

Chapter Ⅱ

2 BACKGROUND AND METHODS

2.1 Machine learning

Machine learning (ML) is a subset of artificial intelligence that focuses on teaching computers how to learn without the requirement for particular task programming. The key notion is that it is feasible to develop algorithms that can learn and predict from data by themselves [1]. ML is an expanding area of data science. Algorithms are taught to generate classifications or predictions using statistical approaches, allowing data mining projects to reveal important insights.
This kind of mined knowledge is important for the growth metrics that businesses employ while moving products to the market [2]. According to UC Berkeley [3] ML is composed of three parts.

- **A decision process**: Machine learning algorithms are used to produce predictions or classifications in general. The algorithm will provide an estimate of a pattern in the data based on some input data, which can be labelled or unlabelled.
- **An error function:** An error function is used to assess the model's prediction. If there are known instances, an error function may be used to compare the model's accuracy.
- **A model optimisation process:** Weights are adjusted to decrease the gap between the known example and the model's estimate if the model can fit better to the data points in the training set. This assessed and optimised procedure will be repeated by the algorithm, which will update weights on its own until a certain level of accuracy is reached.

2.1.1 Learning methods

Supervised learning (SL) – Is the most common learning approach in neural networks (NNs). It is learning via the teacher-student relationship, where the teacher possesses the environmental knowledge. The representation of an environment is expressed with a set of input-output pairs (features and labels). This learning method is applied in the field of classification or value prediction (regression). A learning algorithm is taught with examples, which represent input and expected

output, e.g. a correctly classified value or output numerical value. Weights are adjusted according to the difference between an actual vs. predicted network result with a loss function[4].

Unsupervised learning (UL) – Is the exact opposite of supervised learning. It does not require any pre-labelled or completely labelled dataset. Unsupervised learning is self-organised learning. Its main goal is to investigate underlying patterns and make predictions about the outcome. We provide the computer data and instruct it to search for hidden features and logically cluster the data. This learning method is used for clustering, anomaly detection, association, autoencoders. It is difficult to assess the accuracy of an algorithm that has trained with unsupervised learning since the data lacks a recognised "ground truth" element. However, labelled data is difficult to come by in many study areas, or it is prohibitively expensive. In some circumstances, allowing the deep learning model to discover patterns on its own can yield excellent results [4].

Reinforcement learning (RL) – Does not rely on either supervised or unsupervised learning. RL algorithms learn to react to their surroundings on their own in any given context. This field of study is expanding quickly and creating a wide range of learning algorithms which can be used in robotics, gaming, and other fields. There is always a start and an end state for a learning agent. However, there may be multiple ways to reach the end state. An agent tries to manipulate the environment in a reinforcement learning problem to its benefit. On success, the agent is rewarded and appreciated, while If the agent is rewarded and appreciated for displaying good behaviour, then it should be penalised and disappreciated in equal measure upon failure to display good behaviour. The agent learns from its environment in this way [4].

Differences are summed up in Table 1, as explained in the paper [5].

Criteria	Supervised ML	Unsupervised ML	Reinforcement ML
Learning	Trained with labelled data and guidance.	Self-training with unlabelled data without any guidance.	Works on interacting with the environment
Type of data	Labelled	Unlabelled	No predefined data
Type of problems	Regression and classification	Association and clustering	Exploitation or exploration
Algorithms	Linear regression, Logistic regression, SVN, KNN, etc…	K-Means, C-Means, Apriori	Q-Learning SARSA
Goal	Calculate outcomes	Discover underlying patterns	Learn a series of actions
Application	Forecasts trading, Risk evaluation	Recommendation systems, Anomaly detection	Gaming, Self-driving vehicles

Table 1 Types of machine learning

2.1.2 Overfitting and underfitting

A model that overfits the training data is referred to as overfitting. When a model learns the information and noise in the training data to the point where it degrades the model's performance on new data, this is known as overfitting. This means that the model picks up on noise or random fluctuations in the training data and learns them as concepts. The issue is that these concepts do not apply to new data, limiting the model's ability to generalise. Nonparametric and nonlinear models, which have more flexibility when learning a target function, are more prone to overfitting. As a result, many nonparametric machine learning algorithms feature parameters or strategies that limit and constrain the amount of detail learned by the model. The problem of overfitting can be solved in various ways, the most basic of which is adding more data to the dataset or reducing model complexity [6]. On the other hand, we have the underfitting problem. A model is defined as underfitting if it cannot model and generalise to new data. A machine learning model that is recognised as underfitting is unsuitable, as evidenced by its poor performance on the training data. Underfitting is rarely discussed since, given a decent performance metric, it is simple to discover.

The solution is to move on and experiment with different machine learning techniques. Nevertheless, underfitting serves as a good counterpoint to the issue of overfitting.

Since overfitting is more difficult to overcome, multiple methods are used to reduce it. The obvious first step that can be taken is to add more data to the learning process, which in some cases will not be possible as we already possess all the available data. The next step is data augmentation, a solution to the previous problem when we do not have more data for training. It is a process that makes minor changes to data such as flips, rotations, scaling, or translation. That kind of data transformation will make neural networks believe they are facing new instances. In Figure 1, we can see a demonstration of data augmentation for convolutional neural networks (CNN) training models [7], [8].

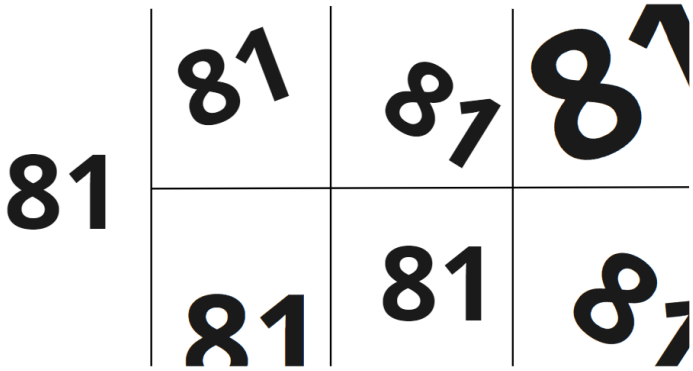

Figure 1 Data augmentation on a single image

When the previously mentioned steps do not provide a suitable solution, we need to add regularisation to our NN model. The four most popular options are dropout, L1 and L2 regularisation, and cross-validation. The dropout technique prevents interdependent learning by changing the outputs of randomly selected neurons to 0 during each training cycle. L1 regularisation estimates the median of the data, while L2 regularisation estimates the mean of the data to avoid overfitting. The last technique is cross-validation. The idea is to construct many tiny train-test splits using the initial training data. These divisions can be used to fine-tune the model. Data is partitioned into k subsets, or folds, in typical k-fold cross-validation. The method is then iteratively trained on k-1 folds, with the remaining fold serving as the test set (holdout fold). With cross-validation, one may fine-tune hyperparameters using only the data from the original training set. This allows us to keep the test set as a truly unseen dataset for the selection of a final model. The selection of techniques is dependent on the dataset.

2.1.3 Deep learning

Deep Learning (DL) is a subset of ML approaches that employs artificial neural networks (ANNs) that are inspired by the structure of neurons in organic brains. DL essentially consists of three or more layers in a neural network. The term “deep learning” originally referred to the presence of numerous layers in an artificial neural network, but its meaning has evolved over time. While 10 layers were sufficient a few years ago to account for network depth, today it is usual for a network to be deep if it has hundreds of layers. The way each algorithm learns is where DL and ML differ. These algorithms can take text, pictures, voice recordings, and learn important characteristics of the data, which can significantly reduce the need for human expertise (especially in feature extraction) and allows the usage of larger datasets [9]. As Jeff Dean mentioned in his slides [10], with more data plus bigger models plus more computations, results get better.

2.2 Neural networks

We could describe neural networks as a set of algorithms whose architecture is inspired by the human brain for recognising patterns in data. They use a sort of machine perception to categorise or cluster raw data. All real-world data, whether images, sounds, text, or time series, must be translated into numerical vectors which form patterns that the NN can recognise, optimise and even predict. This allows researchers from many scientific disciplines to design artificial neural networks to solve a variety of problems. In paper [4], the author asks ‘Why artificial neural networks?’, which is answered by the fact that at the time of writing, von Neumann's modern computer did not offer characteristics comparable to a human brain. A few of the characteristics mentioned by the author are:

- learning ability
- generalisation ability
- adaptivity
- inherent contextual information processing
- fault tolerance

As we know, modern digital computers outperform humans when it comes to numerical computation. However, humans are still much better at solving perceptual problems, such as recognising the same human face in different spacetimes. For example, recognising a known person on a group photo taken during childhood and a present picture. Or recognising a potentially dangerous pattern of human driving behaviour based on previous experience. Not only are humans better at those tasks, they are also very flexible in solving common problems without much effort.

Nevertheless, as more resources come into the field of artificial intelligence, applications based on neural networks will become better [11].

2.2.1 Artificial neural networks

Artificial neural networks (ANNs) are comprised of node layers containing an input layer, one or more hidden layers, and an output layer, they are used to simulate human neural networks. Each node, or artificial neuron, is connected to all others and has a weight and threshold linked with it. The nodes represent a space where computation happens, with similar characteristics as human neurons, which need a specific threshold of stimulation to be activated (can be controlled with bias manipulation). This activation represents a passage of data in our ANN from a given layer to the next one in a network. Otherwise, no data is passed along to the next layer of the network. A node combines data input with a set of coefficients and weights amplifying or dampening that input. It is important to choose the right input data concerning the task which the algorithm is trying to learn. For example, if we want to classify the data with a minimum error, we need to find the right input data. To determine whether and to what extent a signal should progress further through the neural network, the input and weight products are summed and passed through an activation neuron. When a signal passes through different layers of neurons, which consequently activate other activations nodes, we can determine the outcome of a neural network. The relationship representing the neuron output signal is given by the following equation [12]:

$$0 = f(net) = f\left(\sum_{j=1}^{n} w_j x_j\right) \tag{1}$$

Where w_j represents the weighting vector. Function $f(net)$ is referred to as an activation function, where the net variable is a scalar product of the input and weight vectors,

$$net = w^T x = w_1 x_1 + \cdots + w_n x_n \tag{2}$$

T is a transposition of a matrix. The final output value O is computed as

$$O = f(net) = \begin{cases} 1, if\ w^T x \geq \theta \\ 0,\ otherwise \end{cases} \tag{3}$$

where θ is the threshold level of a neuron. If the result is 1, data will flow through the network, if it is 0, then it is a dead end.

In Figure 2 we can see the components of a single node. A collection of nodes is called a node layer (Figure 3), representing a row of neurons in a neural network. Each of them turns on and off like an electrical switch as the input is fed through the network. Starting with an initial input layer that receives the user's data, each layer's output is the subsequent layer's input.

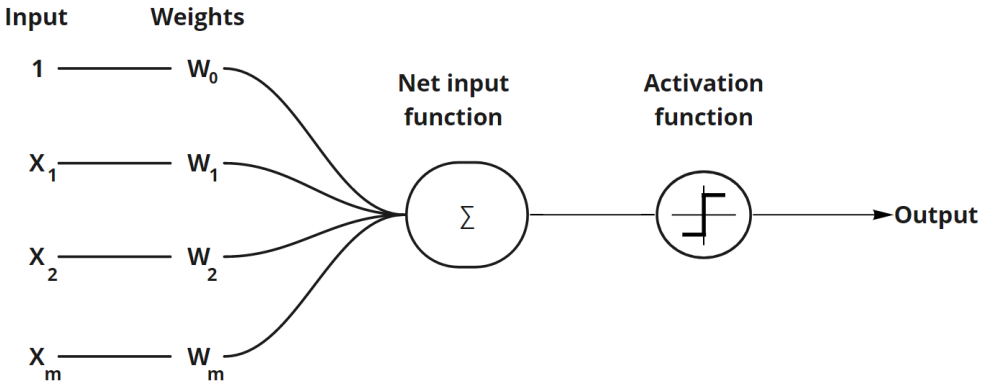

Figure 2 Neural network node components

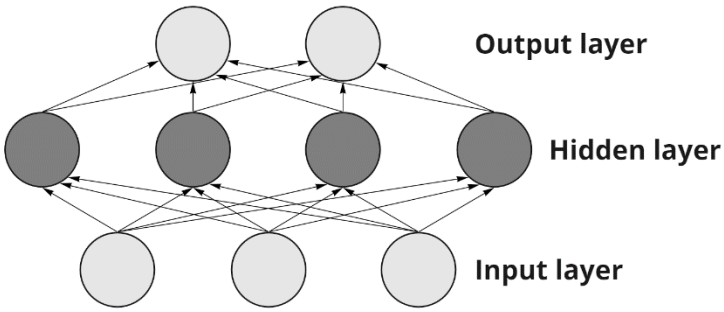

Figure 3 Artificial neural network scheme

The concept of forward feeding is usually the case, even though recurrent neural networks that allow feedback connections also exist.

2.2.2 Activation functions

When we want to shape the neuron's output, activation functions come to the rescue. They enable us to set the output boundary for a given task, consequently with this operation, we can determine the result of a neural network as well. An activation function defines how the weighted sum of an input in a node is transformed into an output in the neural network. Although networks are designed to use the same activation function for all nodes in a layer, the activation function is

applied within or after the internal processing of each node in the network. A neural network typically contains three sorts of layer. An input layer, which accepts initial data into the system for processing; a hidden layer that receives input from the previous layer, runs an algorithm, and then sends the calculated output to the next layer; and an output layer that makes a prediction. In this process, the hidden layers typically use the same activation function. The output layer on the other hand uses the activation function which fits the requirements for a prediction by the model. Various activation functions may be utilised in neural networks, however, only a few are utilised in practice for hidden and output layers.

The following are the most often utilised activation functions for hidden layers:

- **Sigmoid activation function**

This is a mathematical function, the plot of which has a characteristic "S" shape. It is used in machine learning mainly because at a certain value of X it gradually maps the value of Y, which is very practical in classifying data that we know to have only two meanings. The output is limited to values between 0 and 1. The equation is expressed as follows:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{4}$$

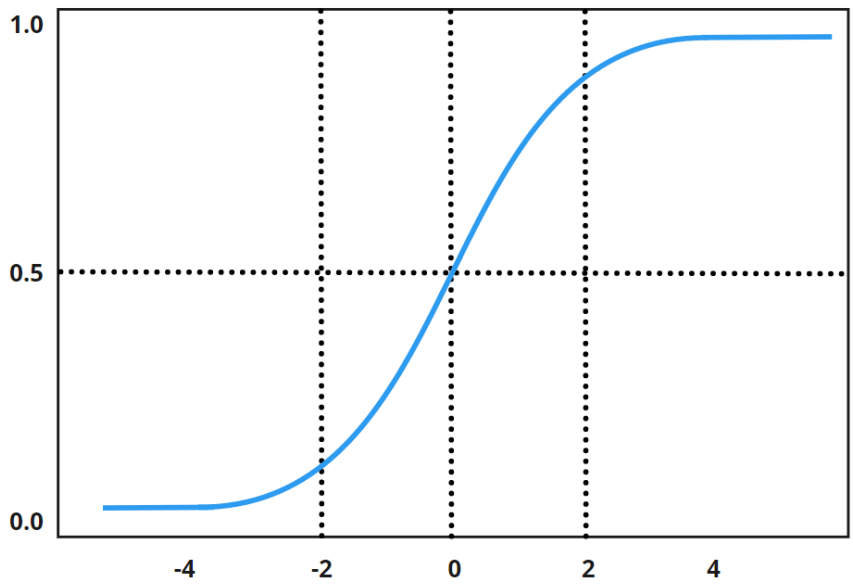

Figure 4 Sigmoid function graph

- **ReLU activation function**

This consists of the function F (z) = max (0, z), which means that if the result is positive, it will print the same value, otherwise the output is 0. It is popular because it is easy to use and effective in getting around the limits of other popular activation functions like Sigmoid and Tanh. It is less prone to vanishing gradients, which prohibit deep models from being trained, yet it can suffer from other issues such as saturated or "dead" units. In Figure 5 below, we can see the graph of the function. The equation is expressed as follows:

$$\begin{cases} 0 & if x \leq 0 \\ x & if\ x\ > 0 \end{cases} = \max\{0, x\} = x1_x > 0 \tag{5}$$

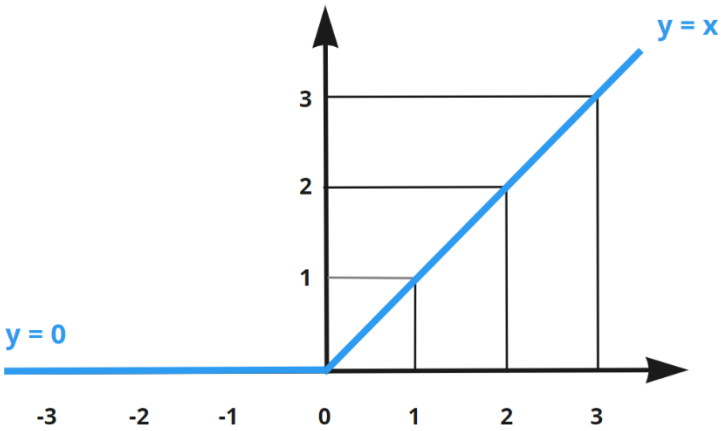

Figure 5 ReLU graph function

- **Tanh or hyperbolic tangent Activation Function**

This function accepts any real value as input and returns a value between -1 and 1. The larger the input (higher positive number), the closer the output is to 1.0, and the smaller the input (higher negative number), the closer the output is to -1.0. It is calculated as follows:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{6}$$

Where *e* is the base of the natural logarithm.

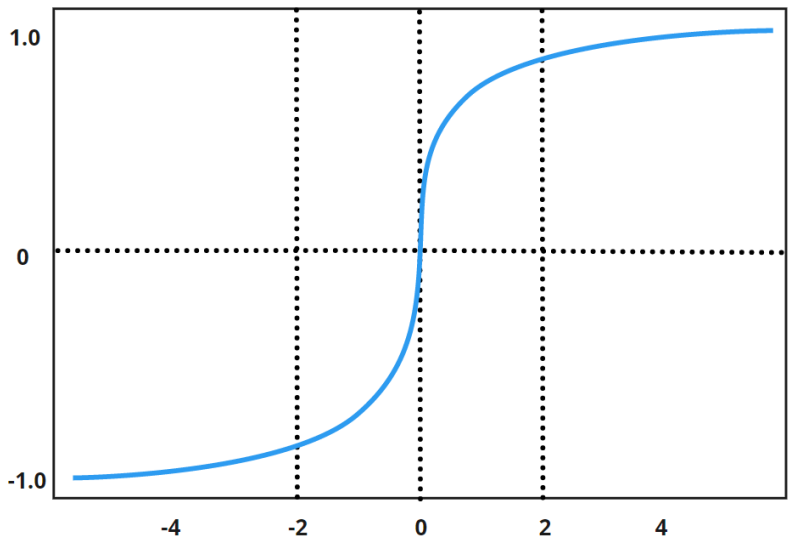

Figure 6 Tanh graph function

We can see that the 'S' shape of the Tanh function is similar to that of the Sigmoid function in Figure 6. When choosing an activation function, it often boils down to the architecture of the neural network used in a model. Common architectures in modern neural network models such as the multi-level perceptron and convolution neural networks will use the ReLU activation function or its extensions as (Leaky ReLU, GELU, ELU, …) [13]. Tanh or sigmoid activation functions, or perhaps both, are still extensively used in recurrent networks. The LSTM (long short-time memory), for example, frequently employs Sigmoid activation for recurrent connections and Tanh activation for output [14].

Output layers are used for a direct output prediction of a neural network model. It is important to note that all feedforward neural networks have an output layer. Activation for those layers can be done by Linear, Logistic (Sigmoid), and Softmax functions since they represent a list of most commonly used ones.

- **Linear output activation function**

This function is sometimes referred to as an identity function because it always returns the same value that was passed into it. This is due to the multiplication with 1.0, which does not make any change to the weighted sum of the input. Formula:

$$f(x) = x \tag{7}$$

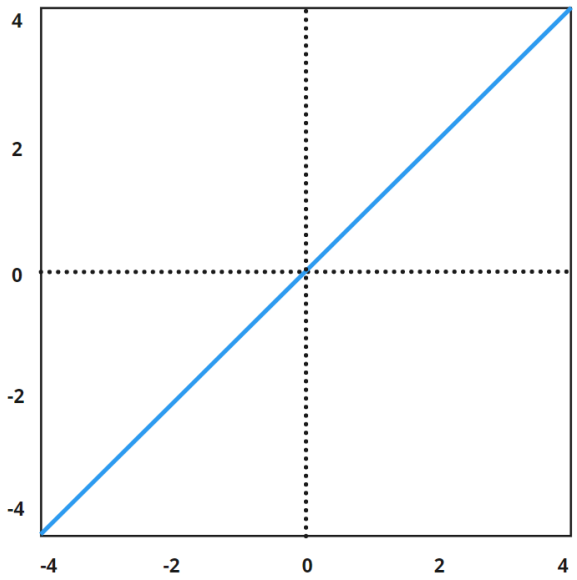

Figure 7 Identity function graph on the real numbers

- **Softmax activation function**

The Softmax function transforms a vector of integers into a vector of probabilities, with the probability of each value proportional to the vector's relative scale. Softmax is applied as the activation function for multi-class classification issues involving more than two class labels. In comparison with a Sigmoid function which is used to represent a probability distribution over a binary variable, Softmax is used to represent the probability distribution over a discrete variable with *n* possible values.

$$\frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}} \; for\; i = 1, \dots, J \tag{8}$$

Where *e* is a natural logarithm base and *x* is a vector of outputs.

2.2.3 Types of architectures

- **Feedforward networks**

The connections between neurons in this sort of NN do not form a cycle or loop. The information is simply flowing forward from the input to the subsequent levels. There may be several intermediary hidden layers depending on the network design. Despite being the oldest and most basic form of network design, the feedforward NN is still frequently employed in machine learning. Figure 8 shows the mentioned architecture with 6 neurons in the input layer, 3 neurons in a hidden layer, and a single neuron in the output layer [4].

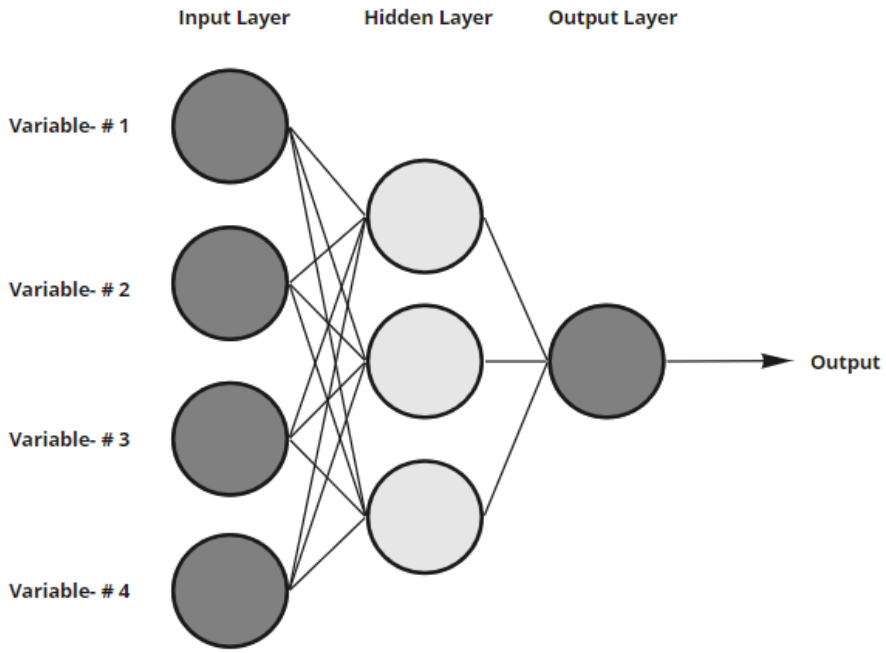

Figure 8 Example of a feedforward network with a single hidden layer

- **Recurrent networks**

The feedback neural network remembers the lessons learned from the past (previous iterations) state and applies it to the future (next iteration). Here we must remember that an ordinary DNN uses its learned state for the future, but this learned knowledge is forthcoming from the entire pre-completed training. Meanwhile, an RNN works the same way, but in addition remembers the state that has been learned from the previous input while the output is being generated. An RNN may have a single or multiple inputs and outputs. The hidden state vector, which contains context,

determines output. This is based on prior inputs and their results. As a result, the same input may produce a different output depending on the previous inputs in the series.

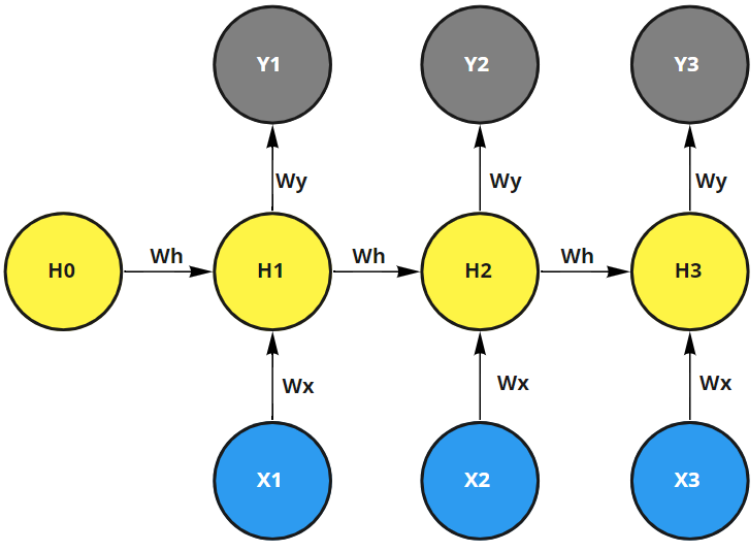

Figure 9 Example of feedback network with a hidden state that is meant to carry pertinent information from one input item in the series to others.

2.3 Evolutionary construction of neural networks

For conventional machine learning algorithms, the hyperparameter optimisation problem has been addressed with a variety of methods. For example, we can use techniques such as grid search, random search, Bayesian optimisation, meta-learning, and others. Those techniques try to find a set of optimal hyperparameters, which are used in an algorithm during the learning process. However, when it comes to a deep learning architecture, the problem becomes much more challenging to solve. Not only because more time and computational resources are required, but extensive knowledge and understanding of both NN and optimisation processes [15]. Deep learning engineers are expected to have a solid comprehension of what architecture will perform best in a specific scenario, and yet it is rarely the case. The various possible design architectures that can be created are endless. This is where neural architecture search (NAS) is used to automate NN architecture engineering. Its goal is to figure out a network topology that will give the best result on a given task. As presented in this article [16], NAS is a system with three primary components [16].

2.3.1 Search space

In principle, the search space determines which architectures can be designed and a set of rules on how layers can be connected and set up (e.g., convolutional, fully connected, pooling). Since engineers often set up this component to simplify the search, researchers [16] are concerned that it can bring human bias into a construction. Human intervention in a search can prevent finding novel NN architectures which are beyond human understanding. In Figure 10, it is shown how the cycle of human-based topology is crafted in comparison to NAS-based. According to another article [17], the main difference is human intervention in search space construction and trial-and-error spent resources.

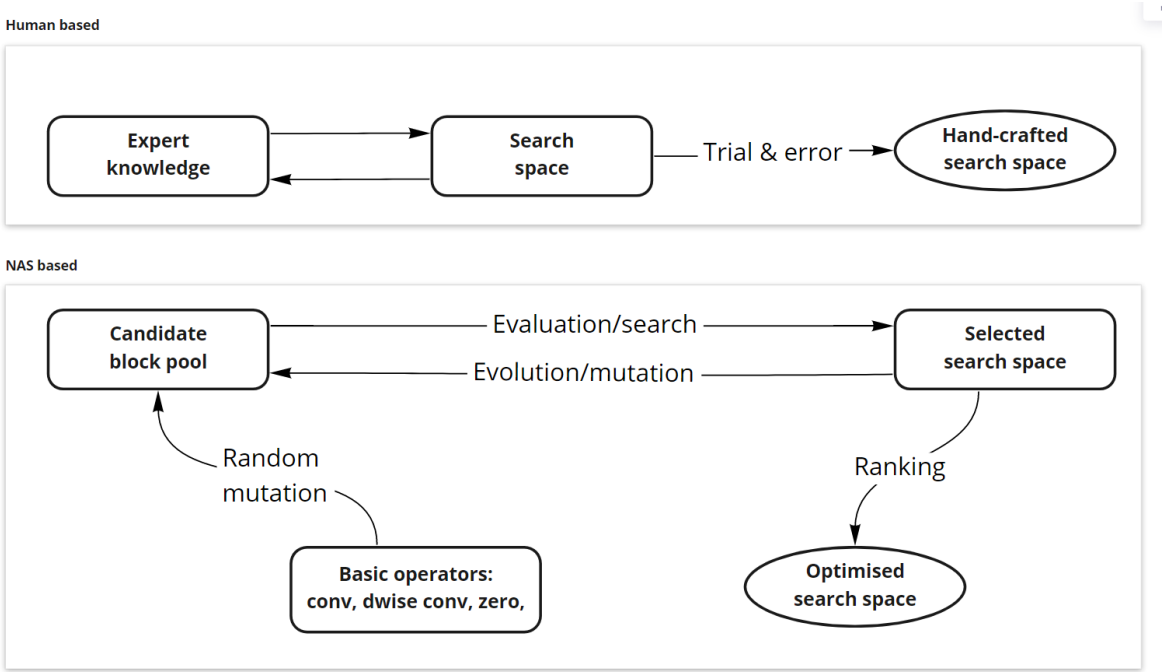

Figure 10 NAS search space

2.3.2 Search strategy

A network architecture candidate pool is generated using a NAS search algorithm. It strives to generate high-performance architecture candidates based on the child model performance parameters (e.g., high accuracy, low latency). Examples of those algorithms are based on Bayesian optimisation, Reinforcement Learning (RL), Genetic Algorithm (GA), Weight sharing, and One-shot [12].

2.3.3 Evaluation strategy

The goal of NAS is typicaly to discover an architecture that produces high prediction performance on data that has not been seen before. To get feedback for optimising the search algorithm, we need to measure, estimate, or anticipate the performance of each child model. Candidate evaluation can be quite costly, hence several innovative evaluation methods have been proposed to save time or calculation. When we evaluate a child model, we are generally interested in its accuracy on a validation set. Recent research has begun to look into other aspects of a model, such as model size and latency, because specific devices may have memory constraints or require quick response times. As presented in Figure 11, NAS can be visualised as a pipeline of components. Each of these components plays a vital role when building an effective NN model for a specific problem.

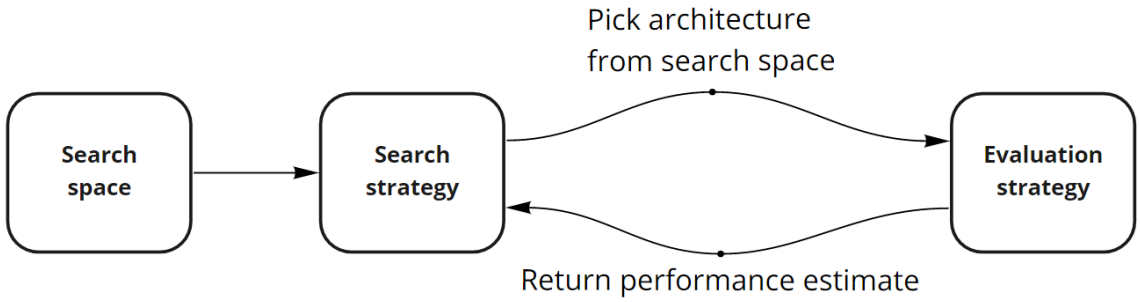

Figure 11 The general framework of NAS.

Chapter Ⅲ

3 ANOMALY DETECTION

Anomaly detection is the ability to find patterns in data that are not in line with expected content. The main goal of this process is to define a norm, technique, barrier which will separate outliers from normal data. Those data points are often referred to as anomalies, outliers, or unnatural [18]. An anomaly threshold cannot be generally set due to the fact that it can be used in a variety of domains. For that reason, it has become widely studied in statistics and machine learning, where it is also known as outlier detection, deviation detection, novelty detection, and exception mining [19]. Over time, a variety of anomaly detection techniques have been implemented for particular uses, such as the monitoring of sensor data on the international space station [20], credit card fraud detection where the system mines a database [21], network traffic analyser for UDP flooding [22], while others are designed to be more generic. Anomaly detections can be done based on available data labels that denote whether an instance is normal or anomalous. Since anomalous behaviour is often dynamic in nature, three methods of anomaly detection are commonly used. The supervised anomaly detection technique in which classes for normal and anomalous data instances are given. A semi-supervised technique in which classes are only assigned to normal and not to anomalous data instances. An unsupervised technique in which normal and anomalous data instances are presented, but no class labels are assigned [18]. Anomaly detection has high importance because its behaviour is often critical to a running system, this is also why we see a wider implementation of it in industry.

3.1 What is an anomaly?

According to Chandola et al. [18], anomalies are patterns in data which do not display characteristics that are similar to normal data, their instances are significantly different from the remaining data. This behaviour can be observed in any dimension of data if we can define a normal

subset to begin with. In Figure 12, we can see two-dimensional datasets, where the majority of observations lie in the region N_1 and N_2 which represent normal data. Points that are far enough away from these regions represent anomalies such as o_1, o_2 and points in the region O_3.

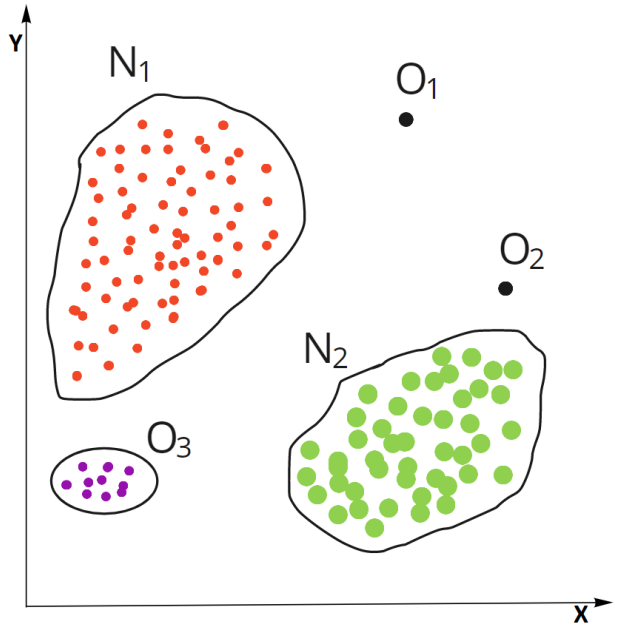

Figure 12 A simple example of datasets and anomalies

While anomalies and noise are related, they have distinct concepts. Some authors use the term "weak outliers" to describe an instance that is outside the interquartile range, but within minimum and maximum, and "strong outliers" to describe an instance that is beyond all borders [23]. Noise in data is usually random and originates for a variety of reasons. It may not be interesting unless it can rate the quality of the instrument generating the data. In Figure 13(a), a single point A seems to be very different from the rest of the data in aspects of features X and Y, therefore it is certainly an anomaly in our example. Meanwhile, the situation in Figure 13(b) is much more subjective, therefore it is much harder to state confidently if A is noise or an anomaly in the data. Point A in Figure 13(b) is relatively more likely to present a data point for noise since it seems its randomness shares similarities to other noise points. In addition, anomaly refers to an outlier type that is of interest to an analyst, where point A does not have any strong evidence to flag it as an anomaly [23].

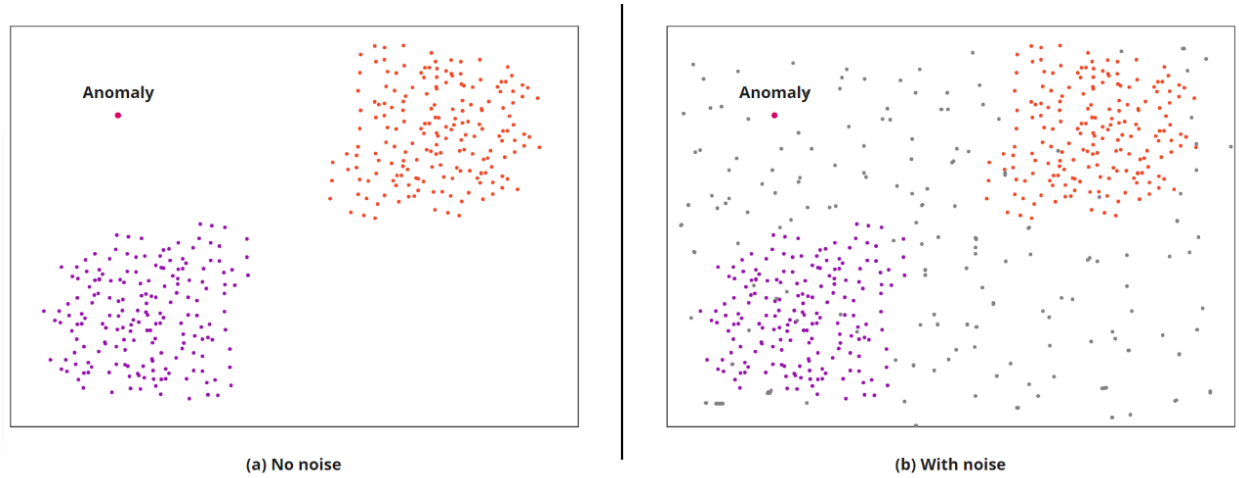

Figure 13 The difference between noise and anomalies

Within the unsupervised situation anomaly detection where previous samples of anomalies do not seem to be available, the distinction is due to the fact of the semantic boundary between normal data and true anomalies. Noise is commonly presented as a weak outlier of normal data, which does not meet a robust criterion for a data point to be interesting or anomalous enough to an analyst.

To further understand the difference between noise and anomalies, Figure 14 gives a good overview of different regions on a continuous spectrum from normal data to noise and to anomalies. The distinction between the regions of the spectrum is frequently not exactly defined and is made on an *ad-hoc* basis based on application-specific criteria. A noisy system is mostly the main factor why many data points do not have a clear separation between noise and anomaly. Regardless of that, the noise generated by a noisy system process will be deviant enough to have a lower outlier score compared to anomalies, which typically have a higher outlier score. After all, it comes to the interest of an analyst to regulate the separation of noise and anomalies [23].

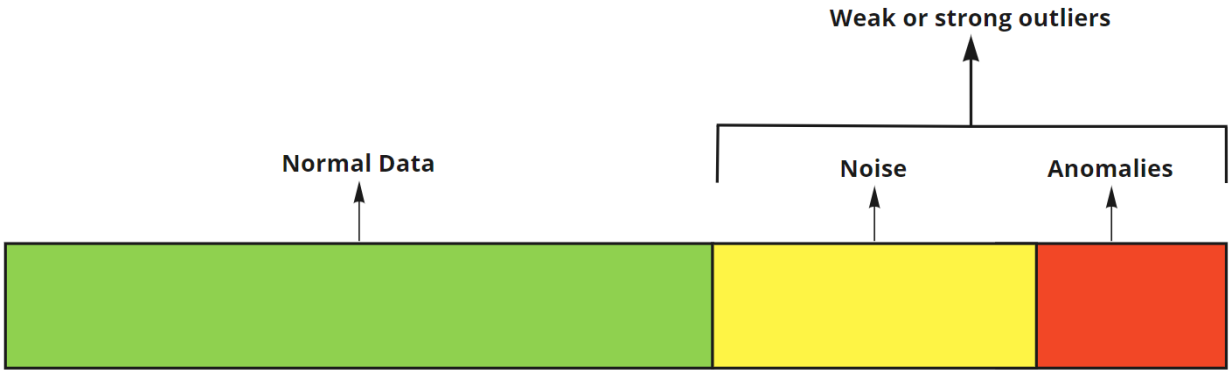

Figure 14 The spectrum from normal data to outliers

3.2 Types of anomalies

When performing anomaly detection, it is important to understand different types of anomalies, such as point, contextual or collective anomaly.

3.2.1 Point anomaly

When a particular data point in the dataset deviates from the normal pattern of behaviour, it can be termed a point anomaly. This is considered a simple type of anomaly, and is therefore the subject of many research and study communities. Taking a look at Figure 12, where points o_1, o_2 and subset O_3 lie outside the boundary of normal data, which marks them as point anomalies, these often represent an extreme deviation that happens randomly and has no particular meaning. An example in real life would be when a developer is committing a source code on average 4.5 times per day, but if it becomes 8 or more times on any random day, it is considered to be a point anomaly.

3.2.2 Contextual anomaly

When a data instance is anomalous in a specific context and not otherwise, it is termed a contextual anomaly. Even when observing the same point through different contexts, we will not always receive an indication of anomalous behaviour. To detect it we need to combine contextual and behavioural attributes.

1. *Contextual attributes* are used to determine the context (or neighbourhood) for a data instance. Time and space are most frequently used. For example, when a developer commits a source code during the final stages of release, they are very likely to make a greater number of commits per day, which is considered normal. On the other hand, making a lot of commits during non-busy days is considered unexpected, anomalous, and would therefore require a deeper analysis to be explained. We flag values based on different periods.
2. *Behavioural attributes* define the noncontextual characteristics of an instance. In our example, the number of commits would be correlated with the development team of which our developer is a member.

When observing values for behavioural attributes within a specific context, anomalous behaviour can be detected. A data instance may represent a contextual anomaly in one context, while in another situation, an identical data instance (in behavioural attributes) may be considered normal. When identifying contextual and behavioural attributes for a contextual anomaly, the previously mentioned property is a key [18]. An example of a contextual anomaly can be seen in Figure 15.

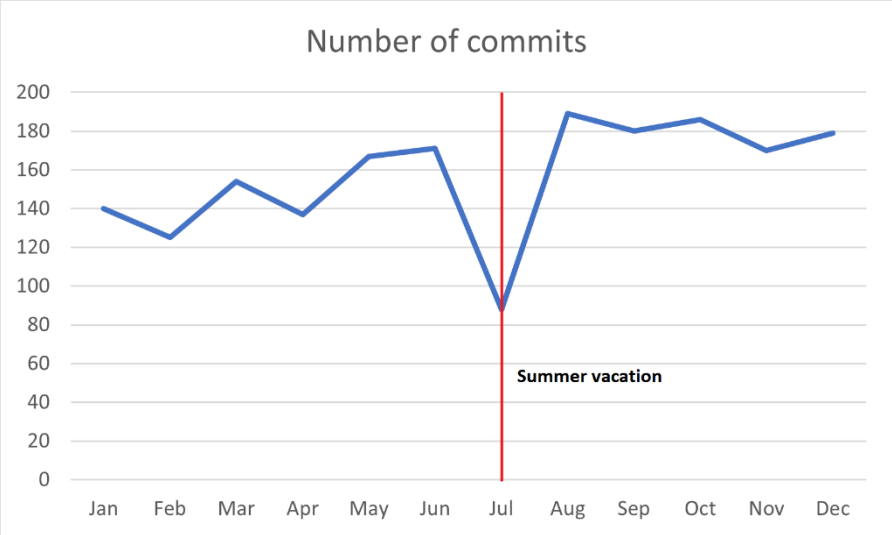

Figure 15 Number of commits per month

3.2.3 Collective anomaly

When a collection of related data instances behaves irregularly in relation to the overall dataset, it is referred to as a collective anomaly. It is possible that an individual data instance is not an anomaly in and of itself but is labelled as such because it is part of a collection. Some authors also refer to collective anomalies as contextual anomalies based on the idea that we can look at the whole collective pattern of the data stream with contextual incorporation [19]. In our example, we could try to find collective anomalies if we would look at the source code commits of the entire team each day, as seen in Figure 16.

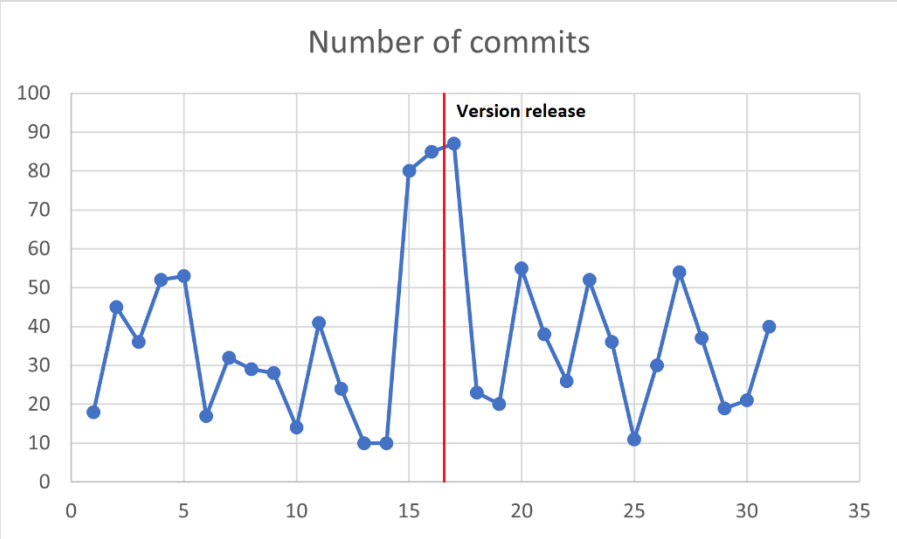

Figure 16 Number of commits per day

3.3 Detection in machine learning

Detection of anomalies when analysing deviations from normal behavioural patterns on different datasets is a non-trivial task. Based on available data labels which denote whether a data instance is normal or anomalous, we can conduct anomaly detection techniques with three different models.

3.3.1 Supervised anomaly detection

To train in supervised mode, a training dataset with labelled instances for both normal and anomalous classes is needed. Building a predictive model for normal vs. anomalous classes is a common strategy in these situations. Any data instance that has not been seen is compared to the model to identify which class it belongs to. In supervised anomaly detection, two fundamental

challenges arise. To begin, there are considerably fewer anomalous examples in the training data than there are normal cases. Ingredients for the performance of supervised anomaly detection are presented in Figure 17.

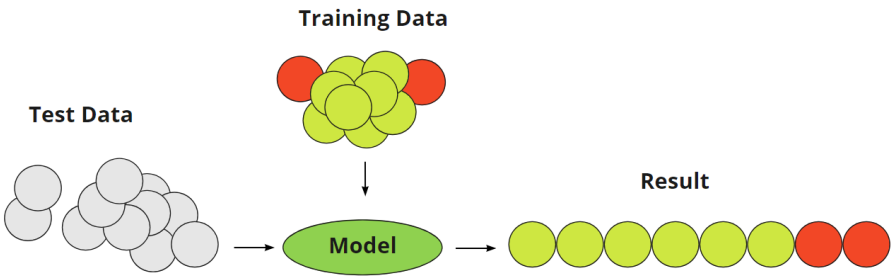

Figure 17 Supervised anomaly detection

Second, getting correct and representative labels, particularly for the anomaly class, can be difficult. In article [18], the author mentioned several strategies developed to inject false anomalies into a regular dataset to acquire a labelled training dataset. Apart from these two concerns, the problem of supervised anomaly detection is comparable to that of creating predictive models.

3.3.2 Semi-supervised anomaly detection

The basic assumption for the semi-supervised technique is that most of the data come from the same (unknown) distribution, which we refer to as the normal part of the data. A few observations, on the other hand, come from different distributions and are classified as anomalies. For example, a spacecraft's fault detection or network attacks can produce anomalies that cannot be sampled but can represent an accident or attack on a system. As computer systems become more sophisticated, relying on the availability of labelled datasets will be increasingly difficult. Ingredients for the performance of semi-supervised anomaly detection are presented in Figure 18.

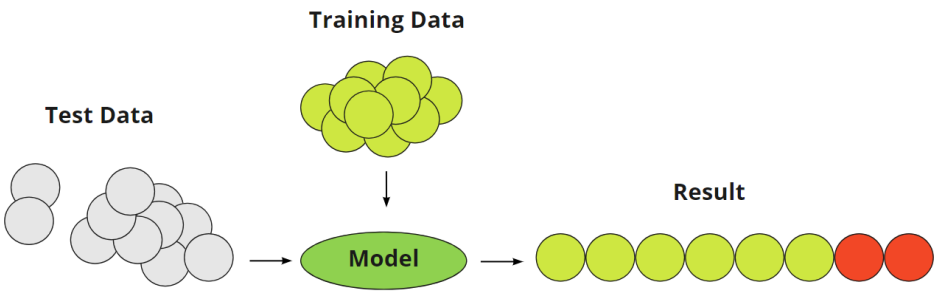

Figure 18 Semi-supervised anomaly detection

3.3.3 Unsupervised anomaly detection

This technique receives a large dataset with mostly normal elements, yet there are outliers buried inside the dataset. A distinction between a training and test dataset is not made. The concept is that an unsupervised anomaly detection system scores data only on the dataset's particular characteristics. Distances or densities are commonly used to determine what is normal and what is an outlier [24]. The ability to process a large amount of data is a major advantage of the unsupervised anomaly detection process. The unsupervised technique is the most flexible approach, which does not require any labels, which can semi-automate the manual inspection of data and help analysts to focus on the suspicious elements of data instead of determining the deviation boundary to separate normal from anomalous data [25]. Ingredients for the performance of unsupervised anomaly detection are presented in Figure 19.

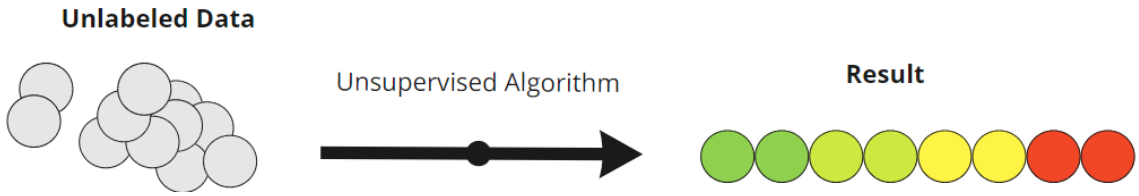

Figure 19 Unsupervised anomaly detection

3.4 Output of anomaly detection

How anomalies are reported is a key aspect of any anomaly detection technique. Anomaly detection systems typically provide one of the following two sorts of output [18]. Firstly, a label can be used to indicate whether an instance is anomalous or normal [24]. Secondly, a score or confidence value that indicates the extent of the anomaly can be more informative [24].

The sort of output is conditional on the technique used for the anomaly detection algorithm. Labels are often used for supervised anomaly detection since they are used together with classification algorithms, their value is often binary. Scores, on the other hand, are more common in semi-supervised and unsupervised anomaly detection algorithms. This is primarily due to practical concerns, as programmes frequently rank anomalies and only show the user the top abnormalities. Scores allow the analyst to determine thresholds in a specific domain, to select relevant anomalies.

Chapter IV

4 SOLUTION

Autoencoders are a type of NN which may be applied in unsupervised anomaly detection. We shall get to know them better in the following chapter.

4.1 Autoencoders for anomaly detection

An autoencoder is a specific type of NN in which the dimensions of input and output are the same, e.g., if we put an image of size 50x50 pixels into an autoencoder model, we will get an output with the same dimensions. We can say that an autoencoder is a replicator neural network since it replicates data from the input to the output in an unsupervised way. By sending the input through the NN, the autoencoder reconstructs each dimension of the input (Figure 20). It may appear trivial to use a neural network to replicate an input, however, the size of the input is reduced during the replication process, resulting in a smaller representation (latent space). In comparison to the input and output layers, the hidden layers of the NN have fewer units. As a result, the reduced representation of the input is stored in the hidden layers. This reduced representation of the input is used to generate the output [26].

4.1.1 Architecture of autoencoders

An autoencoder is made up of three parts:

- **Encoder**: Is a fully connected, feedforward neural network that compresses the input image into a latent space representation and encodes it as a compressed representation in a lower dimension. The deformed representation of the original input is the compressed data.
- **Latent space**: The reduced representation of the input that is supplied to the decoder is stored in this section of the network.
- **Decoder**: Like the encoder, the decoder is a feedforward network with a structure that mirrors the encoder. This network is in charge of reconstructing the input from the code to its original dimensions.

The encoder and decoder are defined as transitions ϕ and ψ, such that:

$$\phi: \mathrm{X} \to \Upsilon \text{ (encoder)}$$
$$\psi: \Upsilon \to \mathrm{X} \text{ (decoder)}$$
$$\phi, \psi = \underset{\Phi,\psi}{\arg\min} \|\chi - (\psi \circ \phi)\chi\|^2$$

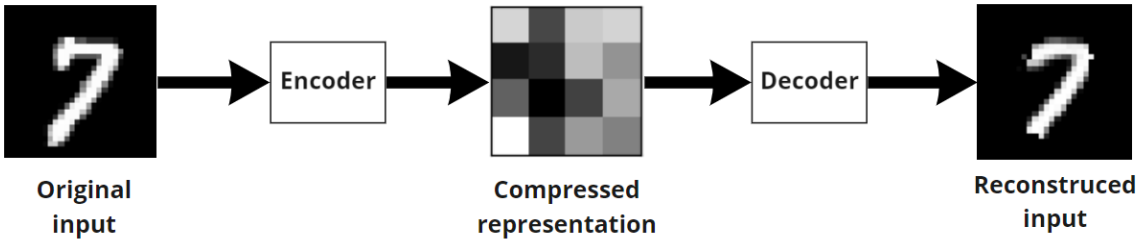

Figure 20 The input image is encoded to a compressed representation and then decoded

Compression and decompression functions have three main properties such as they are data-specific, which means that they can only compress data efficiently if it is similar to the data that they have been trained on, e.g. an autoencoder trained on pictures of cars would do a rather poor job of compressing pictures of flowers. This is due to the fact that the features it learnt are car-specific. Another property marks the autoencoder's functions as lossy, because their output degrades the original input, due to the fact during the learning phase, a model reduces the original input to a latent space, from which it later attempts to reconstruct the output. Since all original details of the data cannot be represented in the reduced dimensions of the latent space, a loss results during reconstruction. The distance function is used to compute the difference between the

input and reconstructed data to minimise the reconstruction loss. Weights are adjusted based on the result. We need to define a distance function between the information loss when building a compressed representation of the input data and the decompressed representation to reduce the reconstruction loss. The lower it is, the better the model is. Automatic learning from data examples is another property of the autoencoder's function. This can be very useful when considering that we just need appropriate training data with which to train a model which we want to perform well on a specific type of input without any new engineering.

4.1.2 Depth of model

Many autoencoders are trained with a single layer encoder and decoder, however, using deep (multiple hidden layers) encoders and decoders renders numerous benefits.

- Depth can exponentially reduce the required quantity of training data [13].
- Deep autoencoders produce superior compression than shallow or linear autoencoders in tests (e.g. memorisation in convolutional autoencoders [27]).
- Depth can exponentially reduce the computational cost [13].

4.1.3 Types of autoencoders

To avoid autoencoders from learning the identity function and to improve their ability to collect essential information and learn richer representations, a variety of approaches are available. A few examples are:

- Shallow Autoencoders
- Deep Autoencoders
- Stacked Autoencoders
- Sparse Autoencoders
- Denoising Autoencoders
- Variational Autoencoders
- Beta Variational Autoencoders
- Vector-Quantised Variational Autoencoders

Each of them has its unique use cases. A good article explaining the differences between them can be found here [28].

4.1.4 Applications of autoencoders

The autoencoder can be used to learn a representation for a variety of purposes.
Many new autoencoder architectures can be created by merging or modifying existing models for a variety of applications. Some autoencoder applications are listed below.

- **Anomaly detection**

The idea behind using autoencoders for these tasks is that a trained autoencoder will learn the latent subspace of normal samples. Once trained it would have a low reconstruction error for normal data and a high reconstruction error for anomalies. However, recent research has revealed that certain autoencoding models are not capable of reliably detecting anomalies, even though they can be very good at recreating anomalous samples [29].

- **Classification**

While autoencoders are trained in an unsupervised environment (without labels), they can also be utilised in a semi-supervised environment (with labels on part of the data) to improve classification results. The encoder is "plugged" into a classification network and used as a feature extractor in this situation. This is most commonly done in a semi-supervised learning environment, in which a big dataset is provided for a supervised learning task, but only a tiny fraction of it is labelled. The fundamental assumption is that samples with the same label should correspond to some latent presentation that the latent layer of autoencoders can approximate [26]. They share similar characteristics of patterns in data (similar to anomaly detection methods). To use this setup in practice, the autoencoder is first trained with the unsupervised technique. The next step is to set aside a decoder (or used in parallel) and use an encoder as the initial part of a classification model. The final result can be viewed in Figure 21.

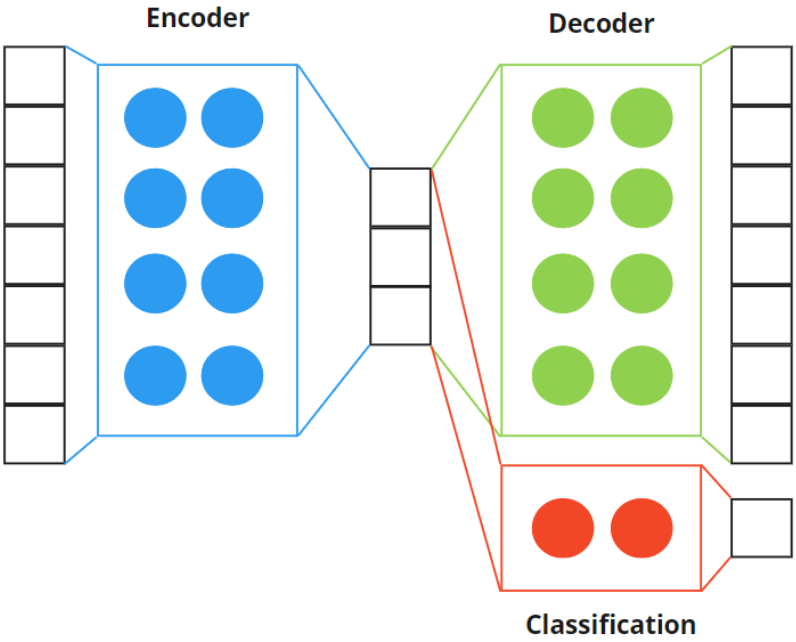

Figure 21 Example of autoencoder usage in semi-supervised technique

- **Clustering**

Clustering is an unsupervised task in which the goal is to divide data into groups with samples that are similar to each other but different from samples in other groups. Since the majority of clustering techniques are dimensionality-sensitive and suffer from the curse of dimensionality, the authors of the following paper [26] pointed out an example in which the data was assumed to have some low-dimensional latent representation, autoencoders can be used to calculate such representations for data with fewer characteristics. Similar to the classification approach, a model is built. Furthermore, each data point's latent representation (the encoder's output) is then saved and used as the input for any clustering method.

- **Popularity prediction**

A stacked autoencoder system recently showed promise in forecasting the popularity of social media posts, which can be useful for online advertising techniques. They used a stacked autoencoder followed by a multilayer perceptron network. Authors of research [30] used available metadata from the user's account and published posts. Even due to the complexity of such a NN model, they have shown that it can be utilised for commercial applications. One of the commercial applications would be predicting the popularity of the next sponsored articles, and therefore help with price-fixing for the post.

- **Image processing**

Autoencoders have characteristics that are beneficial in image processing. We can use them for lossy image compression, where they become competitive with other compression algorithms [31], or in more demanding applications, such as medical imaging. Autoencoders have been utilised for image denoising [32] as well as super-resolution [33]. The key information for such applications is usually found in the latent space of the autoencoder.

4.2 Evolutionary neural networks

Deep learning, in which neural network weights are taught via stochastic gradient descent versions, has received a lot of attention in recent machine learning. With the rise of computational capabilities (including the increased speed of GPUs) and large datasets, a different approach arises from the area of neuroevolution, which uses evolutionary algorithms to optimise neural networks, and is inspired by the idea that real brains are the result of evolution. Learning neural network building blocks (for example, activation functions, hyperparameters, designs), and even the methods for learning themselves are all possible with neuroevolution, while most neural learning techniques simply focus on changing the strength of neural connections (i.e., their connection weights). As mentioned in paper [34] deep learning and deep reinforcement learning differ from neuroevolution in that, these maintain a population of solutions during a search, allowing for extreme exploration and huge parallelisation. Evolutionary NNs are powerful, especially in applications of reinforcement learning, evolutionary robots, and attempts to create artificial life in a digital world [35].

4.2.1 Evolutionary computation

Evolutionary computation (EC) is a method of engineering and optimisation in which solutions are created through processes modelled after Darwinian evolution rather than being built from first principles. One of the main methodologies in what is known as "nature-inspired computing" is evolutionary computation. As described in the book [36], if we take a look at technical terms, evolutionary computation is an example of a heuristic search, or search by trial and error, where the (trials) in EC are potential solutions, and the (error) is the measurement of how distant a trial is from the desired outcome. When creating new trials, the error is used to help determine which trial will be used next. The general guideline is that the best way to further minimise error is to create new trials by modifying the prior trials with the lowest errors.

The first step in an EC algorithm is to create a population of individuals that represent possible solutions to the problem. The initial population could be generated at random or by feeding it into an algorithm. Individuals are assessed using a fitness function, with the outcome indicating how effectively they solve or come near to solving the task. Individuals are then subjected to operators inspired by natural evolution, such as crossover, mutation, selection, and reproduction. A new population is generated based on the fitness values of newly evolved individuals. Some individuals are culled to maintain the population size, as is the case in nature. This process continues until the criterion for termination is met. The most common criterion for stopping the algorithm is when it reaches the specified number of generations. As a result, the best individual with the greatest fitness value is chosen [36], [37].

The general steps of EC are as follows:

```
initialise population
evaluate the fitness value of each individual
while the optimal solution is not found and
 the number of generations defined is not reached
 select parents
 apply genetic operators to the selected individuals
 evaluate fitness values of new individuals
 select individuals for the next generation
end while
return the best individual
```

The problem to be solved usually determines in an obvious way what the search space is, and what the objective function is. The chapter Evolutionary Computation in book [36] gives us an example if one wants to discover the largest value of $f(x,y) = \sin(x^2 + 2x - 3)\cos(-2y + y^2 + 1)$ on the intervals $-1 \leq x \leq 1 \; and \; 0 \leq y \leq 1$, with intervals representing a search space and the objective function f(x,y) itself.

Suppose we translate this mathematical example to the field of EANN (evolutionary artificial neural networks) which map input to the desired output. In that case, we could say that the search space is a set of weights and topology of the network connections. Furthermore, the objective function represents how closely the candidate's map matches the desired map using the closeness of a test set of inputs such as a medium squared error.

4.2.2 Swarm intelligence

Swarm intelligence (SI) is a type of computational intelligence technique used to solve complex problems such as optimisation, routing or decision-making. Scientists once again looked to nature for inspiration when developing complex techniques and algorithms for problem-solving.

SI involves a collective study of how individuals in a population interact with one another at the local level. Agents follow simple rules, and there is no centralised control system in place to predict the behaviour of individual agents. The random iteration of a specific degree between the agents results in emergent “intelligent” behaviour that is unknown to individual agents. Algorithms based on these characteristics are members of the SI algorithm family. Many surveys in recent years have demonstrated how promising these algorithms are for solving issues in a variety of disciplines [38, p. 17], [39], [40]. Many swarm intelligence algorithms have been proposed as a result of the popularity of this research topic. Particle Swarm Optimisation (PSO), Artificial Bee Colony (ABC), and Ant Colony Optimisation (ACO) are a few examples. A comprehensive review of the majority of them was conducted in the following paper [41]. Let us take a closer look at ACO.

4.2.3 Ant colony optimisation

Ants are eusocial insects that live in colonies of up to hundreds of millions of workers. Due to the intricate activity that occurs in ant colonies, several studies have been undertaken to better understand the collective behaviour of ants. A French researcher named Grassé identified an indirect form of communication among ants. Individual communication, or stigmergy, as he called it [42] is pheromonal and only accessible locally. He noticed that the results of these reactions could operate as additional significant triggers for both the producing insect and the colony's other members.

Following are the two primary characteristics of stigmergy that set it apart from other forms of communication [42].

- Stigmergy is an indirect, non-symbolic method of communication mediated by pheromone traces deposited in the environment: insects share information by affecting their environment.
- Stigmergic information is local: it can only be retrieved by insects that visit the location where it was deposited (or its immediate neighbourhood).

Individual-to-individual and individual-to-environment interactions appear to be more complex. These complex behaviours are the result of the collective behaviour of very undemanding

individuals [43]. In the context of collective behaviour, eusocial insects are essentially incentive and response agents. The individual performs simple basic measures involving chance based on the information perceived in the local environment. Despite their individual simplicity, colonies of eusocial insects comprise a highly structured social superorganism. Deneubourg thoroughly researched ant pheromone deposition and the resulting behaviour. From that research also the famous double bridge experiment was conducted [42], [44].

- **Example for better understanding of ACO**

Demonstration of pheromone usage in the ant colony, when searching for food.

Let us look at an example from the following paper [45]. Consider the following scenario: there are two ways to return food back to colony. There is no pheromone on the ground at first. As a result, the likelihood of picking either of these two paths is equal, i.e. 50%. Consider two ants who pick two alternative paths to get the meal, each with a fifty-fifty chance of success.

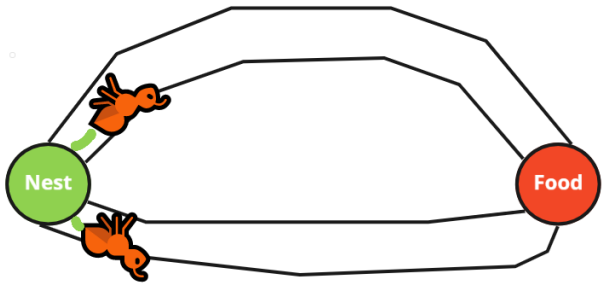

Figure 22 Two ants travel different paths

These two pathways are separated by a significant distance. The ant that takes the shorter path will arrive at the food source first.

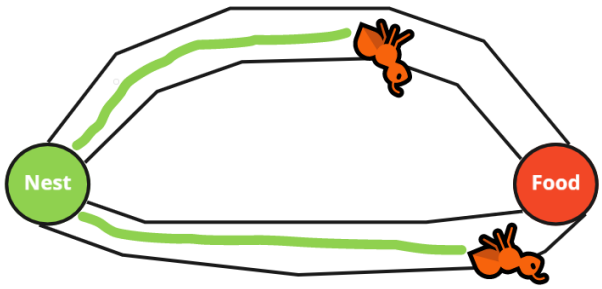

Figure 23 The ant which takes the shortest path, reaches food first

It returns to the colony after locating food and carrying some food with it. As it returns, following its own pheromone trail along its original path, it leaves more pheromone on the ground. The ant that takes the shortest route to the colony will arrive first.

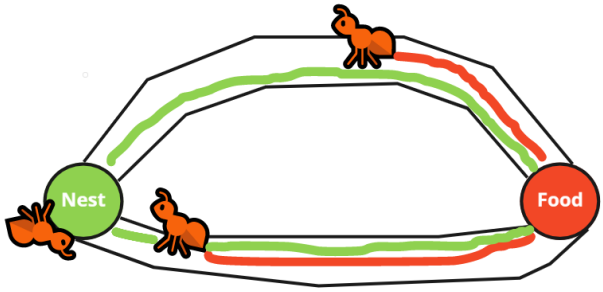

Figure 24 On the way back to the nest, the path is marked again by pheromone

The colony (superorganism) is exploring the phase space of food gathering possibilities. The deposited pheromone signal accumulates upon the shorter path more quickly than on the longer path, because more ants have travelled the shorter path in the same amount of time. At some point, a tipping point occurs and most of the following ants take the shorter (more densely pheromone-laden) path.

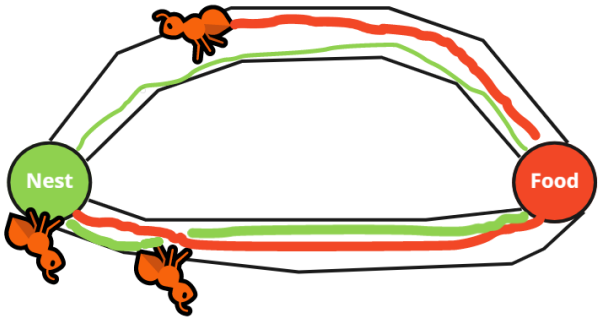

Figure 25 The third ant will travel along the path with the greatest loading of pheromone

When the ant that took the longer path returns to the colony, other ants had already taken the path with the greater pheromone load. When another ant attempts to reach the colony's target (food), it will discover that a shorter path has a greater loading of pheromone. As a result, it chooses

the path with the greatest load of pheromone, which is also the shortest path. Let us look at the options and pick the best one (in the picture below).

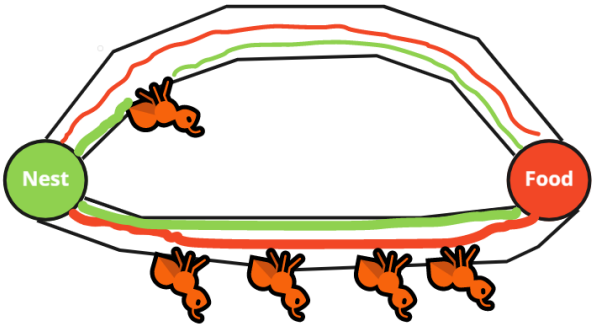

Figure 26 After both paths are marked with pheromone, ants will more likely choose the shortest path

After multiple repetitions of this process, the shorter path has a greater pheromone loading and an increased chance of being followed, and all ants will take the shorter path the next time.

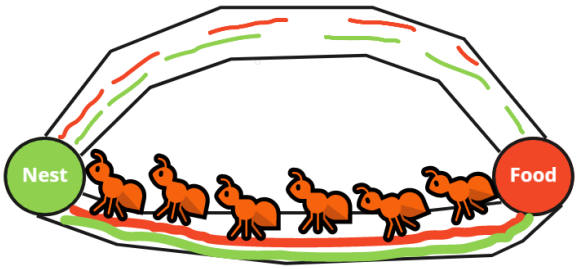

Figure 27 After multiple iterations, the most used path will have a greater pheromone loading

4.2.3.1 ACO algorithm

When a biological ant is converted into an artificial one, we may describe it as a basic computational agent the goal of which is to find the best solution to a given optimisation problem. When using the ant colony example, the optimisation problem must be transformed. Its artificial representation must be expressed on a weighted graph, by which agents can find the shortest path. Following are the steps of an algorithm as described in this paper [43].

When applying the first step of the algorithm, the *generateSolutions()* method is used to build a solution to the problem. The colony of ants is used to visit the edges of the graph and seeks solutions. When a solution is found, the next stage for the pheromone update is executed.

The *pheromoneUpdate()* method alters the pheromone trails during the pheromone update stage. While ants discover better solutions, the pheromone values on these paths increase, while the pheromone traces on worse pathways reduce to avoid local convergence. In practice, altering pheromone loadings improves the chance of paths that have been identified as suitable options being reused.

The last step is the *daemonActions()* method, which uses centralised measures that cannot be performed by a single ant. A daemonic mechanism is the activation of local optimisations or the selection of global information to determine if pheromone loadings need to be increased, and along which pathways. This phase is not necessary for all versions of the ACO algorithm.

The basic ACO algorithm is shown in pseudocode (below) and consists of the three main steps described above, with a loop which is run until the condition is met.

```
procedure ACO_MetaHeuristic is
  while not terminated do
    generateSolutions()
    pheromoneUpdate()
        daemonActions()
  repeat
end procedure
```

4.3 Evolutionary autoencoders

As mentioned in paper [46], autoencoders are a type of unsupervised deep learning approach that may be used for a variety of tasks, including information retrieval (e.g., image search), image denoising, machine translation, and feature selection. These applications are feasible because the autoencoder learns to condense key information about the environment. With the wider usage of autoencoders, a challenge arises. When the application's domain is changed, such as from image denoising to feature selection, it is frequently the case that it is difficult to determine which network design or network characteristics must be altered or changed for the new application usage.

The time necessary to train the network plus the lack of insight as to how the various layer types and hyper-parameters will interact with each other makes designing a neural network challenging.

A response to this issue has been addressed lately [47]–[49], with which computer scientists are trying to build an efficient NAS algorithm to find the optimum between search time (resources) for

NN architecture and reconstructed error of a generated NN. For example, one recently proposed evolutionary method is the evolutionary autoencoder (EvoAE) [47], the main objective of which is to speed up the training of autoencoders when constructing a DNN. EvoAE evolves a population of autoencoders by learning a characteristic of each model in the form of hidden nodes. The evaluation of autoencoders is measured by their reconstruction quality. Crossover and mutation are used to generate the new autoencoders, in which the chromosome represents a hidden node and associated weights and connections. With that technique, human intervention in the construction of NNs is reduced. The authors have condensed the entire algorithm into four steps which are executed in each generation:

a. Selection of autoencoder pairs by reconstruction error and fitness value.
b. Crossover to generate two new autoencoders; children inherit the characteristics of both parents such as hidden nodes and associated weights.
c. Mutation operator which under a given mutation rate adds or deletes a node from an autoencoder.
d. Usage of backpropagation to minimise reconstruction error for each child.

With this kind of method, we can significantly improve the architecture search for our domain-specific problem. As the authors have mentioned, there is much future work to be done in this field. We have used similar concepts when building a NAS algorithm in the following experimental chapter.

Chapter V

5 IMPLEMENTATION

In this section, we will present the practical implementation of the NAS system by the code name AutoDaedalus[1], which aims to discover the best performing autoencoder architecture, when identifying the anomalies in a dataset. We will start with an explanation of why such auto systems are becoming increasingly important when building a NN for a variety of problems in multiple domains. With our work, we want to predominantly move all the manual effort of setting the hyperparameters of a NN model, building a topology of a NN model from a human engineer to a NAS system, which is limited only by the configuration settings set by the human operator. We are one of the first to combine concepts such as the NAS system that builds models with ACO algorithm to identify anomalies with an autoencoder. Work was done as a fork of an existing open-source research project by the code name DeepSwarm [50], which we see as a great starting point when developing SI for NAS. Our implementation was limited by computational, and human resources.

5.1 Limitations

Our research project was limited by the following three factors:

- **Computational resources**

For development and training environment we used the Razer Blade 15 Advanced (Early 2021 model - RZ09-036) with Intel i7-10875H CPU, Nvidia GeForce RTX 3080 8 GB 6144 CUDA cores GPU, and 32 GB RAM.

[1] https://github.com/SasoPavlic/AutoDaedalus

- **Human resources and time scope**

Work was done during the course of 5 months, starting from an open-source DeepSwarm project to a final working prototype and experiment, by a single student software developer under the supervision of a professor mentor.

5.2 AutoDaedalus scope

The AutoDaedalus project is limited by the type of NN it can build. This is mainly because we are focusing on building an unsupervised model which should be able to create a logical border between normal data instances and anomalies. For this task we have chosen an autoencoder NN. Although autoencoder structures can be formed in a variety of ways, we have focused on shallow and deep autoencoders with fully connected layers. Furthermore, AutoDaedalus generates NN models based on the configuration file specified by a human operator. Once the model architecture is generated it trains and is evaluated on the MNIST dataset with parameters defined in a configuration file.

5.3 Tools and frameworks

For software development, we have used the following:

- **DL frameworks**

When it came to choosing DL frameworks, we began with the low-level, such as Tensorflow, which is intended primarily for professional and expert use in creating neural models, where things are based on lower-level implementation, which in practice means that we must be familiar with all of the components when using it. We used Tensorflow exclusively as our backend, with the high-level DL framework Keras on top, to simplify the process of constructing the NN models that can be consumed by the AutoDaedalus project functions. The key advantage of using Keras is how simple it is to construct and train a model, and then evaluate it.

Since we had at our disposal one of the best laptop GPUs available at the time, we needed to make sure our NN models could train and run on it. To make this possible we needed to install the Nvidia CUDA. The CUDA platform is a software layer that allows computing kernels to have direct access to the GPU's virtual instruction set and parallel computational components [51].

- Programming language

Python 3.8.X was used as a key tool for data processing, developing scripts, displaying plots, and deep learning. Together with the great set of packages and strong online community support, it is one of the preferred options in data science.

- Packages

Python packages used: Pyaml, Scikit-Learn, Matplotlib, Numpy, Tensorflow, Keras

Ubuntu packages used: Lambda stack, which provides a one-line installation and management of popular Linux AI software [52].

5.4 AutoDaedalus workflow

Since AutoDaedalus attempts to simulate the entire workflow of a human engineer, when it comes to designing a NN model, the entire workflow of a programme is divided into multiple parts. Each of them takes over the settings which are passed as parameters from the configuration file. This kind of process allows the human operator to control all the puzzles from a single-entry point, such as settings for the dataset, DeepSwarm object, and NN layers types to be used. Figure 28 shows the flowchart of the main AutoDaedalus components.

Figure 28 AutoDaedalus flowchart of main components

Following are explanations of the main components in the entire workflow of NAS, ACO, and usage of the autoencoder to detect anomalies.

5.4.1 Configuration file setup

It all starts with the configuration file which controls the workflow during run time. It aims to set the edge boundaries of NAS when generating architectures as well as defining the data instances that represent normal instances and anomalies. All these settings and others are grouped into three main sets.

```
DataConfig:
  valid_label: [1,7,8,9] # Values representing normal instances
  anomaly_label: [0] # Values representing anomalous instances
  contamination: 0.01 # Amount of anomalies in a dataset in %
  test_size: 0.2 # Ratio between train and test dataset size
  random_state: 42 # State of the random number generator

DeepSwarm: # DeepSwarm object responsible for providing a user-facing
interface
  save_folder:
  metrics: accuracy # Metrics to evaluate the models
  max_depth: 10 # Maximum and a minimum depth of hidden layers
  min_depth: 1 # on one side of the Autoencoder
  reuse_patience: 1 # Number of times weight can be reused

  aco: # Ant colony optimisation object
    pheromone:
      start: 0.1 # Starting pheromone value
      decay: 0.1 # Local pheromone decay
      evaporation: 0.1 # Global pheromone decay
      verbose: 1 # Logging components
    greediness: 0.50 # Greediness of ants
    ant_count: 10 # Maximum amount of ants (models)
    latent_dim: 16 # Dimension of compressed space in Autoencoder

  anomaly:
    quantile: 0.98 # Instance out of this quantile are anomalies

  backend:
    epochs: 75 # Number of epochs per ant (model)
    batch_size: 32 # Number of batches in epoch per ant (model)
    patience: 5 # Early stopping during the training
    verbose: 1 # Logging components
    optimiser: adam # Optimiser for training
    loss: binary_crossentropy # Loss function for training
```

```yaml
Nodes: # Layers types used when building the topology

  InputNode: # First layer in encoder model
    type: Input # Type of layer in Keras
    attributes:
      shape: [!!python/tuple [28, 28, 1]]# Shape of input
    transitions:
      DenseNode: 1.0 # Transition possibility for next layer

  InputDecoderNode: # First layer in decoder model
    type: Input
    attributes:
      shape: [ !!python/tuple [ 14 ] ] # Shape of output
    transitions:
      DenseNode: 1.0

  FlattenNode: # Flatten layer before latent space in Autoencoder
    type: Flatten
    attributes: { }
    transitions:
      DenseNode: 1.0

  ReShapeNode: # Layer used when decoding back from latent space
    type: Reshape
    attributes:
      target_shape: [ !!python/tuple [ 7, 7, 1 ] ]
    transitions:
      DenseNode: 1.0

  DenseNode: # Hidden layer in autoencoder
    type: Dense
    attributes:
      output_size: [128, 64, 32, 16, 8, 4, 2]
      activation: [ReLU, LeakyReLU,Tanh]
    transitions:
      DenseNode: 0.2
      DenseNode2: 0.3
      DenseNode3: 0.1
      DenseNode4: 0.1
      DenseNode5: 0.3

  DenseNode2: # Hidden layer in autoencoder
    type: Dense
    attributes:
      output_size: [128, 2]
      activation: [ReLU, LeakyReLU,Tanh]
    transitions:
      DenseNode: 0.2
      DenseNode2: 0.2
      DenseNode3: 0.2
      DenseNode4: 0.2
      DenseNode5: 0.2
```

```
  DenseNode3: # Hidden layer in autoencoder
    type: Dense
    attributes:
      output_size: [ 32, 16, 8, 4]
      activation: [ReLU, LeakyReLU]
    transitions:
      DenseNode: 0.1
      DenseNode2: 0.1
      DenseNode3: 0.4
      DenseNode4: 0.2
      DenseNode5: 0.2

  DenseNode4: # Hidden layer in autoencoder
    type: Dense
    attributes:
      output_size: [128]
      activation: [ReLU, LeakyReLU,Tanh]
    transitions:
      DenseNode: 0.4
      DenseNode2: 0.1
      DenseNode3: 0.1
      DenseNode4: 0.1
      DenseNode5: 0.3

  DenseNode5: # Hidden layer in autoencoder
    type: Dense
    attributes:
      output_size: [2]
      activation: [ReLU, LeakyReLU,Tanh]
    transitions:
      DenseNode: 0.2
      DenseNode2: 0.1
      DenseNode3: 0.3
      DenseNode4: 0.3
      DenseNode5: 0.2

  OutputNode: # Final output layer in decoder model
    type: Output
    attributes:
      output_size: [1]
      activation: [Sigmoid]
    transitions: {}
```

5.4.2 Preparation of the dataset

The dataset used in AutoDaedalus needs to be set up for unsupervised learning. The dataset needs to have both normal and anomalous data instances without any labels denoting the class of an instance. When it comes to selecting the right data instances, parameters from the configuration file are passed. The `valid_label` contains an array of labels that represent the normal ones, on the other hand, the `anomaly_label` represents an array of anomalous ones. Before choosing actual data instances, the `contamination` parameter is passed alongside as well. With it, we determinate the number of anomalous instances in the final trainable dataset. An example of a build dataset can be seen in Figure 29.

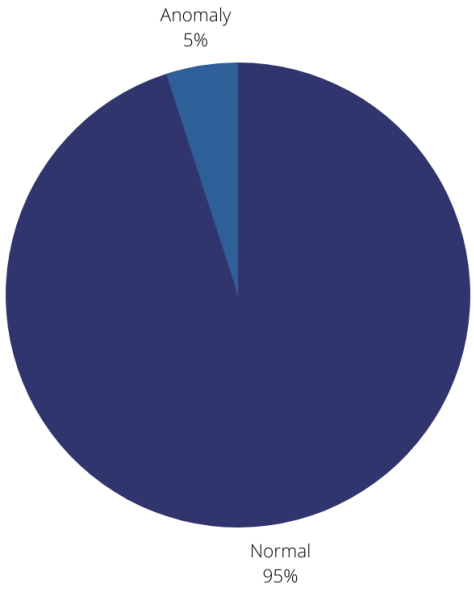

Figure 29 Dataset ratio between normal and anomalous data instances

Once we have both types of data instances in place, we shuffle the dataset and split it to train and test the dataset according to the `test_size` parameter as seen in Figure 30.

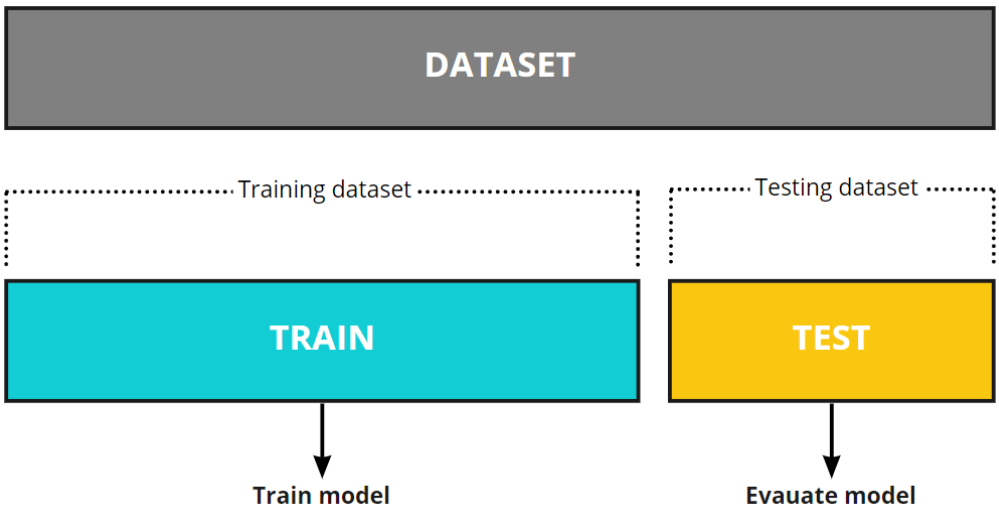

Figure 30 Split of training and testing dataset

5.4.3 Backend initialisation

Our development was done in the Keras framework, to run the application on our low-level Tensorflow framework which serves as a backend, we need to first initialise it. This is done by calling the superclass of Tensorflow Keras API. `BaseBackend` is an abstraction class, which ensures that all the needed properties are initialised together with methods for the DL process, such as `generate_model`, `train_model`, `evaluate_model`.

5.4.4 DeepSwarm with ACO

As mentioned before, DeepSwarm is an open-source research project conducted by Edvinas Byla and Wei Pang [50]. Their contribution to the field of NAS showed great achievements in comparison to previously published methods. Since their source code was designed to form colonies of ants to generate convolutional neural networks (CNN), we needed to redesign the majority of the source code, that is responsible for model topology formation. The reason for this is that the structure or sequence of the layers which form the CNN model are different from the ones of the autoencoder model.

Our modified version of the ACO algorithm is as follows. To generate the ant's path which represents connections between the NN model, we have implemented two methods. The first one is `generate_encoder_path`, which generates an internal graph that includes the `input_node` by default. By this, we ensure that whatsoever dataset is pushed into the model, the first layer will accept it accordingly. After that, a specified number of ants is generated. Whose starting point

begins in an `input_node`. As explained by the authors of DeepSwarm, the ant selects one of the available nodes in the next layer of the CNN using the ACS selection rule. In our autoencoder structure scenario, once the next layer is selected, the ant chooses the node's parameters based on the response of the selection rule. The selection rule is applied based on the possible transitions each node has and the amount of pheromone used in this process. Transitions represent the neighboured nodes to which ants can travel. Once an ant reaches the current maximum allowed depth, `Flatten_node` flattens the previous layer's output to a 1D vector. After that, another `Dense` layer is added to compress the 1D vector to the desired latent space. At this stage, the Keras model representation from the current graph would look like Figure 31.

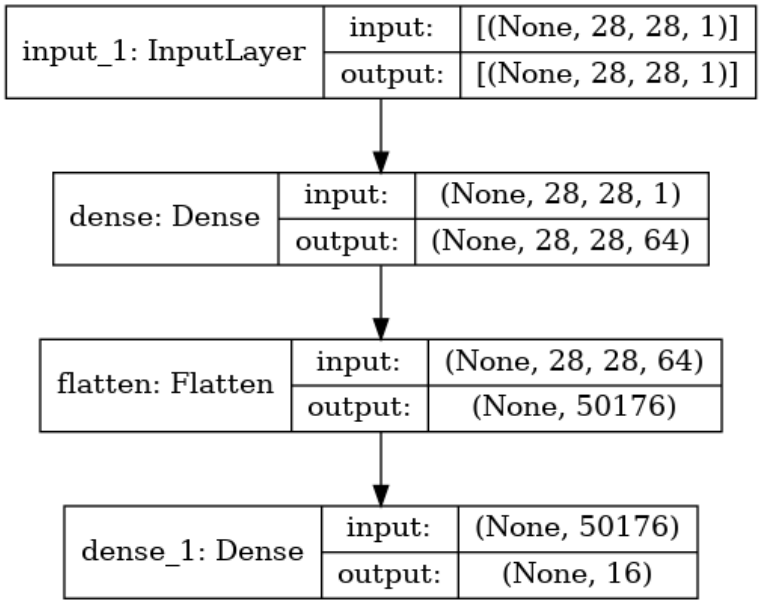

Figure 31 Structure of layers in an encoder

The second method is `generate_decoder_path`, whose structure is mirrored to that of the encoder. This time the ant's path needs to start from where the previous one end. As a result, the graph begins with the `input_layer`, whose input shape corresponds to the shape of the latent space. After the first layer, one `Dense` and `Reshape` layers are added. Their purpose is to rebuild the same vector shape as before it was compressed to the `latent_space`. Following layers are added with the same logic as they were in the encoder method. Lastly, we obtain the rebuilt data as an output. Decoder representation (Figure 32), which matches with the above encoder (Figure 31).

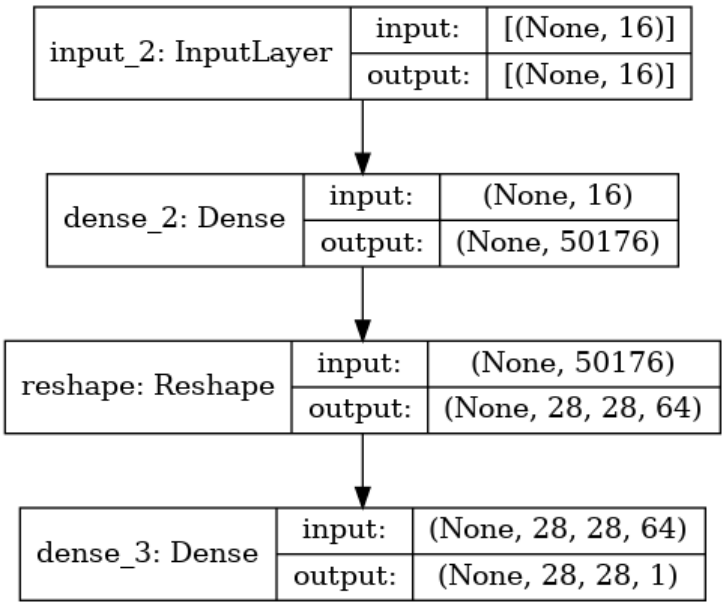

Figure 32 Structure of layers in a decoder

Once the encoder and decoder models are built, the next step is to use them as functional models as a whole to form an autoencoder model. This can be represented by the following equation:

$$y = \text{decoder}(encoder(x)) \tag{9}$$

where *x* is the input to the model and *y* is its reconstruction

The final structure of the autoencoder model with its `input_layer`, `encoder_layer`, and `decoder_layer` is shown in Figure 33.

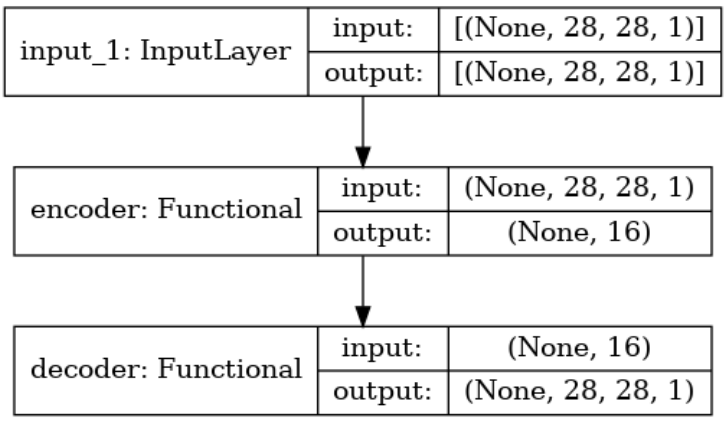

Figure 33 Structure of layers in the autoencoder

The next step is to apply the local update of used pheromone by ant depending on the specified layers in the generated model. This cycle continues until `ant_count` for `current_depth` is reached. Once this is done, the ants are sorted by the metrics score they achieve, and if a new best ant is discovered, this is also updated. The new best ant may then be used to apply a global pheromone update. At the end of the ACO search algorithm, the best ant is returned to the DeepSwarm.

5.4.5 Evaluation

The evaluation of an ant is done by running the model it produces. This is done with multiple matrices which help us to visualise the model's training and final predicted results of anomalies in a dataset. Therefore, we can see how the model performed through the training and validation dataset.

The first metric on our list is the `training_loss`, which shows the `train_loss` and `val_loss` during the fitting of our model to training data. The value of `train_loss` is the distance between the ground truth and the reconstruction. In this combination (training and validation loss) a portion of training data during each epoch is used as validation, expressed by `vall_loss`. With this graph, we can better understand how the model is learning during the learning steps. If the *val*_loss starts to increase or is stale, it is better to stop the training to prevent overfitting. Generated plots are shown in Figure 34.

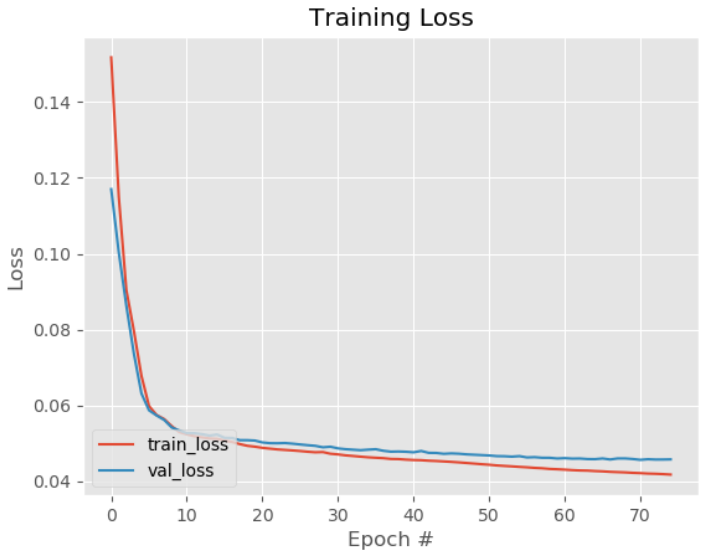

Figure 34 Training loss metrics

The second metric is similar to the previous one, except that it measures the accuracy of the model. Therefore, we have the combination of `train_acc` and `val_acc` to plot on our graph (Figure 35). At first sight, it might not be clear why we should be interested in accuracy when dealing with the unsupervised learning model, but our initial aim when training the model is its ability to reconstruct data as best as possible since this difference between original and reconstructed data will serve as a threshold for anomaly detection.

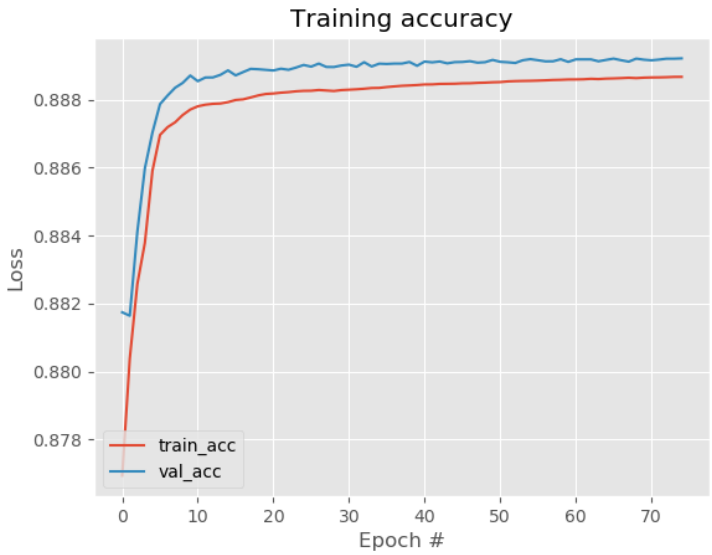

Figure 35 Training accuracy metrics

Continuing with the metric is helpful to a human operator who controls the training. It helps to visualise the results, especially when we are training the model with multiple labels, how the original image is compressed and decompressed through the NN model. As seen in Figure 36, we can assume that some data instances can be potentially identified as false anomalies, due to poor reconstruction by the model, and are therefore prepared for manual inspection. A potential candidate for inspection is number 7 (third example from the left).

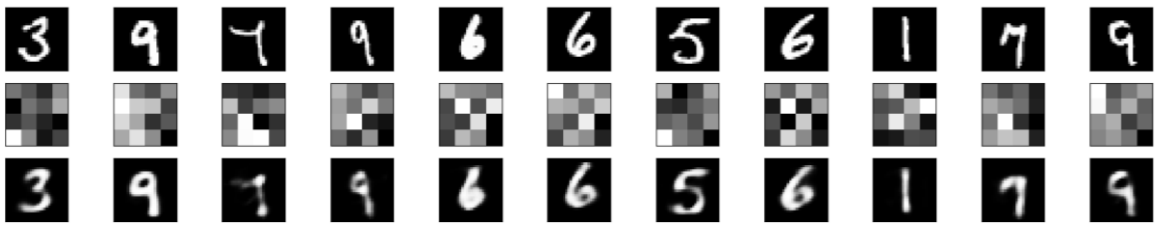

Figure 36 Orginal, compressed, and reconstructed image representation

Another metric method that is helpful during the training and final validation is the loss function *MAE*_loss which uses mean absolute error (MAE), to compute the squared error between the original image and the reconstructed image. When calculating it on all data instances we can see the bar chart showing us number_of_samples that fall into a specific MAE loss value in the following Figure 37.

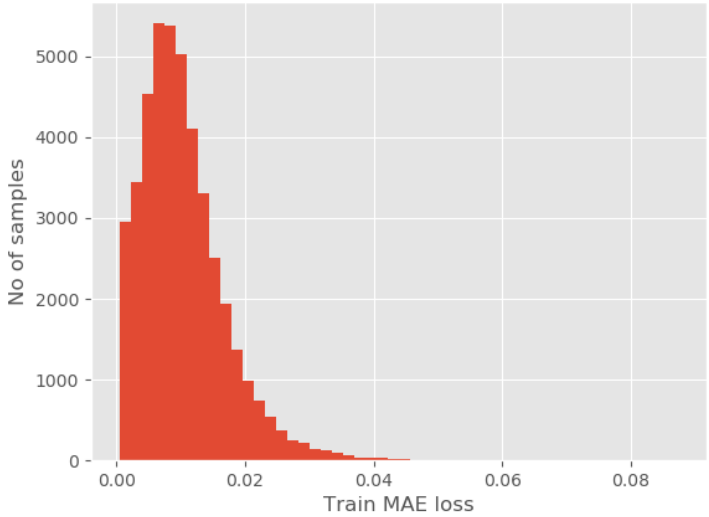

Figure 37 MAE loss in training samples

The next metrics are a direct evaluation of the model when it comes to the final goal of the whole workflow of AutoDaedalus. Starting with the receiver operating characteristic visualisation (ROC). With it, we can get a good overview of the trade-off between sensitivity (true positive rate (TPR)) and specificity (1 – false-positive rate (FPR)). To calculate the ROC curve, in our scenario we need to measure how good our model is when it comes to detecting anomalies within a specific quantile of the dataset. Since all data instances that have greater reconstructed error (measured by MSE) fall into a specified quantile (specified by anomaly_quantile parameter). With quantile and MSE values, calculation of which data instances represent anomalies and which ones are normal is possible. Secondly, the confusion matrix needs to be calculated from the previously determined normal and anomalous data instances. This is shown in the following Table 2.

Actual / Predicted	Anomaly	Normal
Anomaly	TP	FN
Normal	FP	TN

Table 2 Confusion matrix

Where true positive (TP), false negative (FN), false positive (FP), and true negative (TN) are calculated by the following rule:

$$TP = count\ of\ anomalies\ identified\ in\ given\ quantile \tag{10}$$

$$FN = all\ anomalies\ in\ dataset - TP \tag{11}$$

$$FP = all\ instances\ in\ quantile - TP \tag{12}$$

$$TN = all\ normal\ instances - FP \tag{13}$$

From that point calculation of precision, recall and F-score is calculated as well.

$$precision = \frac{TP}{TP + FP} \tag{14}$$

$$recall = \frac{TP}{TP + FN} \tag{15}$$

$$Fscore = 2 * \frac{precision * recall}{precision + recall} \tag{16}$$

Once all intermediate matrices for ROC are calculated, we can proceed to the next function which is to calculate TPR, FPR, and needed false negative rate (FNR) and true negative rate (TNR) values over the range of quantiles [0 … 1]. During quantile range iteration, the threshold for separation between normal and anomalous data is changed and therefore the values of the matrix table are changed accordingly. With values in the confusion matrix changed, the following metrics are re-calculated:

$$TPR = \frac{TP}{TP + FN} \tag{17}$$

$$FNR = \frac{FN}{TP + FN} \tag{18}$$

$$TNR = \frac{TN}{TN + FP} \tag{19}$$

$$FPR = 1 - TNR \tag{20}$$

Once all mentioned metrics are calculated over the different ranges of quantile, the line for the ROC curve can be plotted with the associated area under the curve (AUC) score. An example of the ROC curve is shown as a green-dotted line in Figure 38. From the plot, we can see the correlation between the TPR and FPR values in the range [0 … 1]. These values are calculated by moving quantile values on a range [0 … 1] by step 0.01.

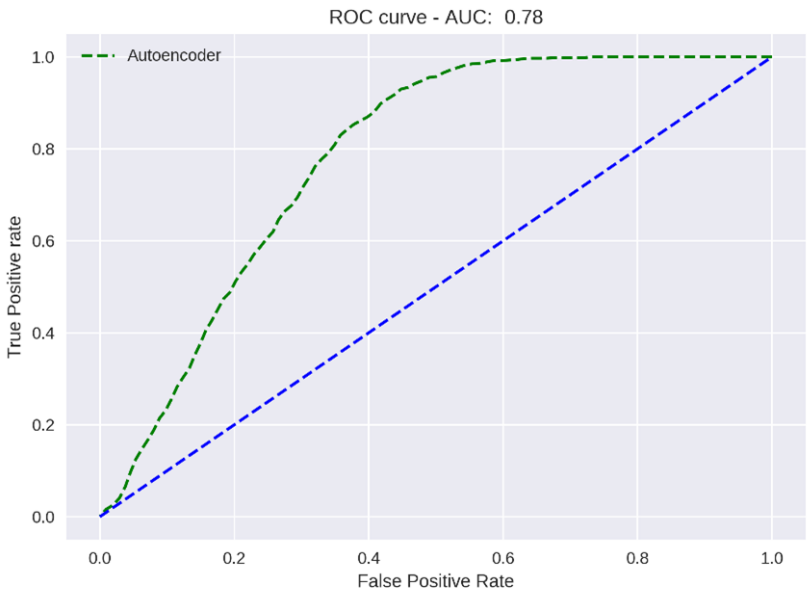

Figure 38 ROC curve for autoencoder model

5.4.6 Finding the best model

The best model can be automatically found by providing the desired metric to the configuration file. Since the term “best” can represent the different metrics for a given dataset and desired distinction between normal and anomalous data instances, the most appropriate method is still to manually handpick the model which does the job well in a specific scenario with the help of all available matrices.

Chapter VI

6 EXPERIMENT

With the experiment, we wanted to see how well a manually built autoencoder detects anomalies in a dataset compared to the best model generated by our open-source AutoDaedalus project. Since there are many ways to build an autoencoder architecture, we have set limits to the extent of the experiment. The same goes for anomaly detection which is specific to the given dataset. The experiment was an excellent testing technique in software development since it gave us a better understanding of the components that are required to build a system that is as sophisticated as AutoDaedalus. One of the main goals of the experiment was to allow other researchers to replicate the experiment, which is why we included a script in the project that is ready to build the manual autoencoder by changing the model layers, a separate script that can run a specific model on-demand, and the ability to save all the generated models and associated matrices in a specified location on a disk. All of this is beneficial for continuous testing and development.

6.1 Experimental environment

6.1.1 Dataset overview

The MNIST dataset was used for all conducted experiments. The dataset contains 60,000 training and 10,000 testing small square 28x28 pixel grayscale images. All digits are handwritten and are in the range [0 … 9]. The distribution of classes in a dataset is seen in Table 3:

Value	Train count	Train %	Test count	Test %
1	6742	11.24	1135	11.35
7	6265	10.44	1028	10.28
3	6131	10.22	1010	10.1
2	5958	9.93	1032	10.32
9	5949	9.92	1009	10.09
0	5923	9.87	980	9.8
6	5918	9.86	958	9.58
8	5851	9.75	974	9.74
4	5842	9.74	982	9.82
5	5421	9.04	892	8.92

Table 3 MNIST dataset class distribution

Parameter *test_size* from configuration file was for all experiments set to 0.2 (train : test) with *random_state* value set to 42.

6.1.2 Software components

For experimental purposes the following packages, programmes, and frameworks were used:

Library / Frameworks / IDE /	Version
Linux kernel	5.11.0-27-generic
Ubuntu	20.04.2 LTS
Tensorflow	2.4.1
Keras	2.3.1
CUDA	release 11.1
Python	3.8.10
DeepSwarm	0.09
PyCharm	2021.1.1 (Professional Edition)
Pyyaml	5.3.1
Scikit-learn	0.22.2
Numpy	1.21.2
Seaborn	0.11.1

Table 4 Used software components

6.1.3 Types of generated models

All models in the experiments are generated by one of two options. Either they are generated by the AutoDaedalus programme or they are manually created. Following are the allowed specifications during the experiment.

- An experimental model can have:
  - Allowed types of layers: `Input, Dense, Flatten, Reshape`
  - The dense layer allowed attributes
    - Output size : `[128, 64, 32, 16, 8, 4, 2]`
    - Activation function: `ReLU`, `LeakyReLU`, `Tanh`, `Sigmoid`
  - Optimiser: `Adam`
  - Loss: `binary_crossentropy`
  - Metrics: `accuracy`, `loss`

6.1.4 Available matrices

When a model is generated, trained, and evaluated, the following infographics are created for it:

Infographic name	**Explanation**
decoder_shape.png	Shape representing decoder model
Encoder_shape.png	Shape representing encoder model
Plt_acc.png	Plot showing training and validation accuracy
Plt_anomalies_095.png	Plot showing found anomalies in quantile = 0.95
Plt_anomalies_098.png	Plot showing found anomalies in quantile = 0.98
Plt_anomalies_0995.png	Plot showing found anomalies in quantile = 0.995
Plt_encoded_image.png	Plot with original, compressed, and decompress image
Plt_loss.png	Plot showing training and validation loss
Plt_MAE.png	Plot showing MAE loss over a count of samples
Plt_reconstructed_results.png	Plot showing an original and decoded image
Roc_curve.png	Plot with ROC curve
Autoencoder.yaml	The configuration file used to generate a model
Deepswarm.log	Entire log during the AutoDaedalus run time

Table 5 Generated infographic per NN model

List of information that is logged during the model evaluation:

Logged information for N model
Number of instances in the dataset
The actual number of all anomalies in a dataset
The actual number of all valid labels in a dataset
Number of TP anomalies found in X quantile
TP, FN, FP, TN values
Recall, Precision, F1-score
TPR, FPR, ROC, AUC values

Table 6 Logged information when a model is evaluated

6.2 Example of operational evolutionary NN

Before reviewing the experiment between manually and auto-constructed NN models, let us look at how AutoDeadalus generates new models. As mentioned in the implementation chapter, it all starts with the configuration file. For this example, we will set it up to make a NN model capable of finding anomalies for a single label. This means we set up `DataConfig` parameters as follows:

```
DataConfig:
  valid_label: [1]
  anomaly_label: [0]
```

All others parameters in the configuration file remain as represented in the implementation chapter. When AutoDaedalus is run, it starts to search for the best-performing model architecture within the allowed limitations set by the configuration file, such as `max_depth`, allowed layers, `ant_count` per depth. Bellow, we can see the information which is logged once a model is generated during the search phase.

```
2021-08-20 09:48:24,259

Current search depth: 1

Generating ant: 1

Ant: 0x7fde03e38580

 Loss: 0.047672

 Accuracy: 0.888302

 Path: InputNode(shape:(28, 28, 1)) ->

DenseNode(output_size:32,activation:LeakyReLU) ->

FlattenNode() ->

DenseNode(output_size:16, activation:Tanh) ->

InputDecoderNode(shape:16) ->

DenseNode(output_size:128, activation:Tanh) -> ReShapeNode(target_shape:(7, 7,
1)) ->

DenseNode(output_size:128, activation:Tanh) ->

OutputNode(output_size:1, activation:Sigmoid)

Hash: 5cacabbd674602951179f6482dabd8ed1cfd62b7f9738242d8e320b6d0fc5119
```

The hash string, which is the universal key identifier when it comes to identifying a generated model in folders on a disk, may also be found in logged information. Once the model is created, it is evaluated using all of the matrices listed in Table 5 and Table 6. The above model (ant: 0x7fde03e38580) is the best performing one at the moment (the best ant is calculated based on a parameter). This model will be the “best ant” until the evolutionary cycle continues and a new best ant is discovered.

```
2021-08-20 09:59:25,676
New best ant found
================================================================================
 Ant: 0x7fde0012edc0
 Loss: 0.046139
 Accuracy: 0.888457
 Path: InputNode(shape:(28, 28, 1)) ->
 DenseNode(output_size:128, activation:ReLU) ->
 FlattenNode() ->
 DenseNode(output_size:16, activation:Tanh) ->
 InputDecoderNode(shape:16) ->
 DenseNode(output_size:64, activation:Tanh) ->
 ReShapeNode(target_shape:(28, 28, 2)) ->
 DenseNode(output_size:64, activation:Tanh) ->
 OutputNode(output_size:1, activation:Sigmoid)
 Hash: 6917d7c05d4c5590cf3a537ee6ce1333019d61a5a3b1d98ec1117b1074e47ecd
```

Once this happens, the ACO algorithm will replace it with the new one and corresponding measures will be executed such as update of global pheromone. At the end of set search space, we will have a collection of all generated models.

6.3 Anomaly detection with the help of an evolutionary NN

When explaining anomaly detection, we need to first clearly understand what represents the anomalies in a given dataset. Anomalies are patterns in data which do not conform with the characteristics of normal data. Interpreting the previous sentence means that the majority of data instances must be from one class and the minority from a different class. This is valid only when training the NN model since at this stage it needs to learn the patterns in data in order to distinguish between class labels. When testing the model any ratio of normal and anomalous data instances can be presented, because at this stage the model has already learnt how to distinguish between the them. When data is pushed into a model with the objective of finding anomalies in the dataset the following happens:

1. Reconstruction of data instances
2. Calculation of MSE for each data instance
3. Anomaly threshold is calculated based on a list of MSE and quantile limit
4. Each reconstruction is checked to see whether it passes over the threshold or not
5. Detected anomalies are displayed
6. Model performance metrics are saved

An example of anomaly detection is shown in Figure 39.

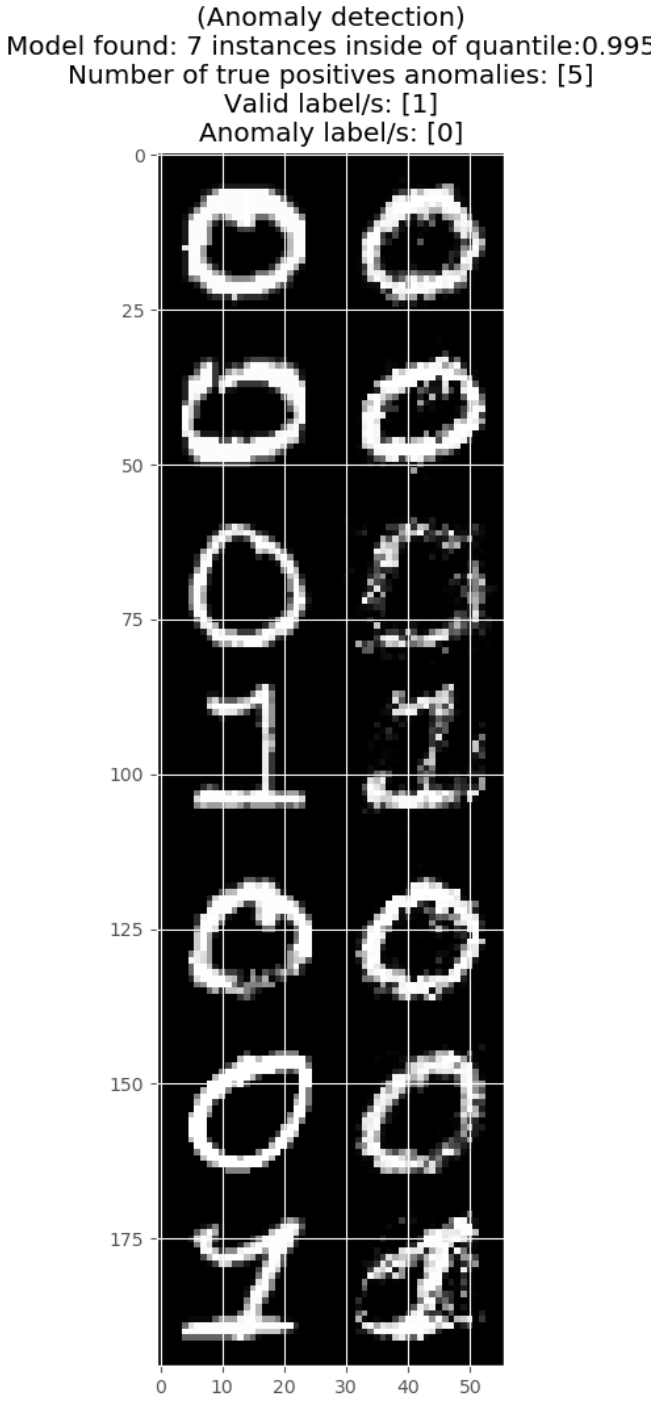

Figure 39 Anomaly detection example

6.4 Results

In this section, we present two experiments together with the evaluation and final results. Experiments are differentiated only by the number of valid labels that are presented in the testing dataset. Digit classes rules were as follows:

- Single-label experiment
  - Valid labels : `[1]`
  - Anomaly labels : `[0]`
- Multi label experiment
  - Valid labels : `[1,2,3,4,5,6,7,8,9]`
  - Anomaly labels : `[0]`
- Quantile value: 0.9

In both experiments, we wanted to empirically test the model architecture that was generated by the AutoDaedalus method versus a manually built model, based on our experience and examples found online. Firstly we tested both methods on a single label experiment and secondly on the multi label experiment. Created architectures for each tested NN model are presented in the tables below.

6.4.1 Single label experiment

- **Manual autoencoder implementation**

Settings used for a manually crafted model:

Maximum depth set to 1:

Depths / Parameters	Encoder	Latent space	Decoder	Activation
1st layer	128	16	128	(ReLU, ReLU)

Table 7 Manual model single label 1 layer

Maximum depth set to 2:

Depths / Parameters	Encoder	Latent space	Decoder	Activation
1st layer	128	16	64	(ReLU, ReLU)
2nd layer	64	16	128	(ReLU, ReLU)

Table 8 Manual model single label 2 layer

Maximum depth set to 3:

Depths / Parameters	Encoder	Latent space	Decoder	Activation
1st layer	128	16	32	(ReLU, ReLU)
2nd layer	64	16	64	(ReLU, ReLU)
3rd layer	32	16	128	(ReLU, ReLU)

Table 9 Manual model single label 3 layer

Maximum depth set to 4:

Depths / Parameters	Encoder	Latent space	Decoder	Activation
1st layer	128	16	16	(ReLU, ReLU)
2nd layer	64	16	32	(ReLU, ReLU)
3rd layer	32	16	64	(ReLU, ReLU)
4th layer	16	16	128	(ReLU, ReLU)

Table 10 Manual model single label 4 layer

Maximum depth set to 5:

Depths / Parameters	Encoder	Latent space	Decoder	Activation
1st layer	128	16	4	(ReLU, ReLU)
2nd layer	64	16	16	(ReLU, ReLU)
3rd layer	32	16	32	(ReLU, ReLU)
4th layer	16	16	64	(ReLU, ReLU)
5th layer	4	16	128	(ReLU, ReLU)

Table 11 Manual model single label 5 layer

- **AutoDaedalus autoencoder implementation**

Settings used for an automatically generated model:

Maximum depth set to 1:

Depths / Parameters	Encoder	Latent space	Decoder	Activation
1st layer	128	16	64	(ReLU, Tanh)

Table 12 AutoDaedalus model single label 1 layer

Maximum depth set to 2:

Depths / Parameters	Encoder	Latent space	Decoder	Activation
1st layer	8	16	2	(LeakyReLU, LeakyReLU)
2nd layer	128	16	2	(LeakyReLU, ReLU)

Table 13 AutoDaedalus model single label 2 layer

Maximum depth set to 3:

Depths / Parameters	Encoder	Latent space	Decoder	Activation
1st layer	128	16	16	(ReLU, Tanh)
2nd layer	2	16	16	(Tanh, Tanh)
3rd layer	2	16	8	(LeakyReLU, Tanh)

Table 14 AutoDaedalus model single label 3 layers

Maximum depth set to 4:

Depths / Parameters	Encoder	Latent space	Decoder	Activation
1st layer	8	16	64	(Tanh, ReLU)
2nd layer	32	16	32	(LeakyReLU, Tanh)
3rd layer	4	16	4	(ReLU, Tanh)
4th layer	128	16	128	(Tanh, LeakyReLU)

Table 15 AutoDaedalus model single label 4 layers

Maximum depth set to 5:

Depths / Parameters	Encoder	Latent space	Decoder	Activation
1st layer	64	16	128	(Tanh, Tanh)
2nd layer	4	16	128	(ReLU, ReLU)
3rd layer	4	16	64	(Tanh, ReLU)
4th layer	128	16	16	(ReLU, Tanh)
5th layer	8	16	128	(LeakyReLU, Tanh)

Table 16 AutoDaedalus model single label 5 layers

6.4.2 Multi label experiment

- **Manual autoencoder implementation**

Settings used for a manually crafted model:

Maximum depth set to 1:

Depths / Parameters	Encoder	Latent space	Decoder	Activation
1st layer	64	16	128	(ReLU, ReLU)

Table 17 Manual model multi label 1 layer

Maximum depth set to 2:

Depths / Parameters	Encoder	Latent space	Decoder	Activation
1st layer	128	16	64	(ReLU, ReLU)
2nd layer	64	16	128	(ReLU, ReLU)

Table 18 Manual model multi label 2 layer

Maximum depth set to 3:

Depths / Parameters	Encoder	Latent space	Decoder	Activation
1st layer	128	16	32	(ReLU, ReLU)
2nd layer	64	16	64	(ReLU, ReLU)
3rd layer	32	16	128	(ReLU, ReLU)

Table 19 Manual model multi label 3 layer

Maximum depth set to 4:

Depths / Parameters	Encoder	Latent space	Decoder	Activation
1st layer	128	16	16	(ReLU, ReLU)
2nd layer	64	16	32	(ReLU, ReLU)
3rd layer	32	16	64	(ReLU, ReLU)
4th layer	16	16	128	(ReLU, ReLU)

Table 20 Manual model multi label 4 layer

Maximum depth set to 5:

Depths / Parameters	Encoder	Latent space	Decoder	Activation
1st layer	128	16	4	(ReLU, ReLU)
2nd layer	64	16	16	(ReLU, ReLU)
3rd layer	32	16	32	(ReLU, ReLU)
4th layer	16	16	64	(ReLU, ReLU)
5th layer	4	16	128	(ReLU, ReLU)

Table 21 Manual model multi label 5 layer

- **AutoDaedalus autoencoder implementation**

Settings used for an automatically generated model:

Maximum depth set to 1:

Depths / Parameters	Encoder	Latent space	Decoder	Activation
1st layer	64	16	32	(Tanh, LeakyReLU)

Table 22 Figure 55 AutoDaedalus multi label 1 layer

Maximum depth set to 2:

Depths / Parameters	Encoder	Latent space	Decoder	Activation
1st layer	2	16	2	(Tanh, LeakyReLU)
2nd layer	64	16	2	(ReLU, LeakyReLU)

Table 23 Figure 55 AutoDaedalus multi label 2 layer

Maximum depth set to 3:

Depths / Parameters	Encoder	Latent space	Decoder	Activation
1st layer	4	16	2	(ReLU, Tanh)
2nd layer	4	16	8	(LeakyReLU, Tanh)
3rd layer	16	16	128	(LeakyReLU, ReLU)

Table 24 Figure 55 AutoDaedalus multi label 3 layer

Maximum depth set to 4:

Depths / Parameters	Encoder	Latent space	Decoder	Activation
1st layer	128	16	4	(Tanh, LeakyReLU)
2nd layer	128	16	128	(Tanh, Tanh)
3rd layer	2	16	16	(ReLU, Tanh)
4th layer	8	16	4	(LeakyReLU, ReLU)

Table 25 Figure 55 AutoDaedalus multi label 4 layer

Maximum depth set to 5:

Depths / Parameters	Encoder	Latent space	Decoder	Activation
1st layer	32	16	2	(LeakyReLU, Tanh)
2nd layer	8	16	8	(ReLU, LeakyReLU)
3rd layer	64	16	128	(ReLU, LeakyReLU)
4th layer	4	16	8	(Tanh, Tanh)
5th layer	8	16	2	(LeakyReLU, Tanh)

Table 26 Figure 55 AutoDaedalus multi label 5 layer

6.4.3 Comparison of methods

After both of the tested methods gave us results for each best NN model of a given depth, the next step was to compare different results based on available metrics. Because the majority of our metrics are based on anomaly detection outcomes, our primary goal was to discover a better performing method on our dataset for each experiment.

Let us begin with the single-label experiment. The given results are from the above NN models presented in tables: Table 7, Table 8, Table 9, Table 10, Table 11, Table 12, Table 13, Table 14, Table 15, Table 16.

NN Model	Depth on one side	Recall	Precision	F1-score	AUC
Manual method	1	0.215	0.995	0.354	0.997
Manual method	2	0.215	0.995	0.354	0.997
Manual method	3	0.246	1.000	0.356	0.997
Manual method	4	0.216	1.000	0.356	0.998
Manual method	5	0.214	0.991	0.352	0.997
Max		0.246	1.000	0.356	0.998
Total		1.106	4.981	1.772	4.986

Table 27 Single label experiments result for the manual method

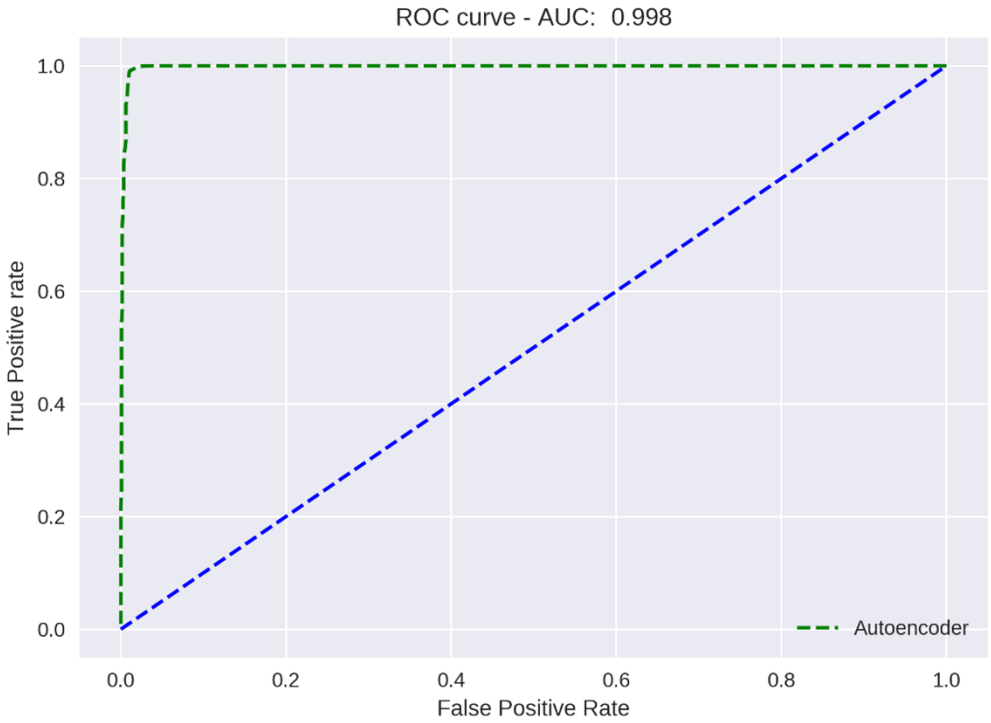

Figure 40 ROC curve of the best performing model produced by the manual method in the single label experiment

NN Model	Depth on one side	Recall	Precision	F1-score	AUC
AutoDaedalus method	1	0.213	0.986	0.351	0.991
AutoDaedalus method	2	0.212	0.981	0.349	0.991
AutoDaedalus method	3	0.215	0.995	0.354	0.992
AutoDaedalus method	4	0.213	0.986	0.351	0.992
AutoDaedalus method	5	0.213	0.986	0.351	0.990
Max		0.215	0.995	0.354	0.992
Total		1.066	4.934	1.756	4.956

Table 28 Single label experiment results of the AutoDaedalus method

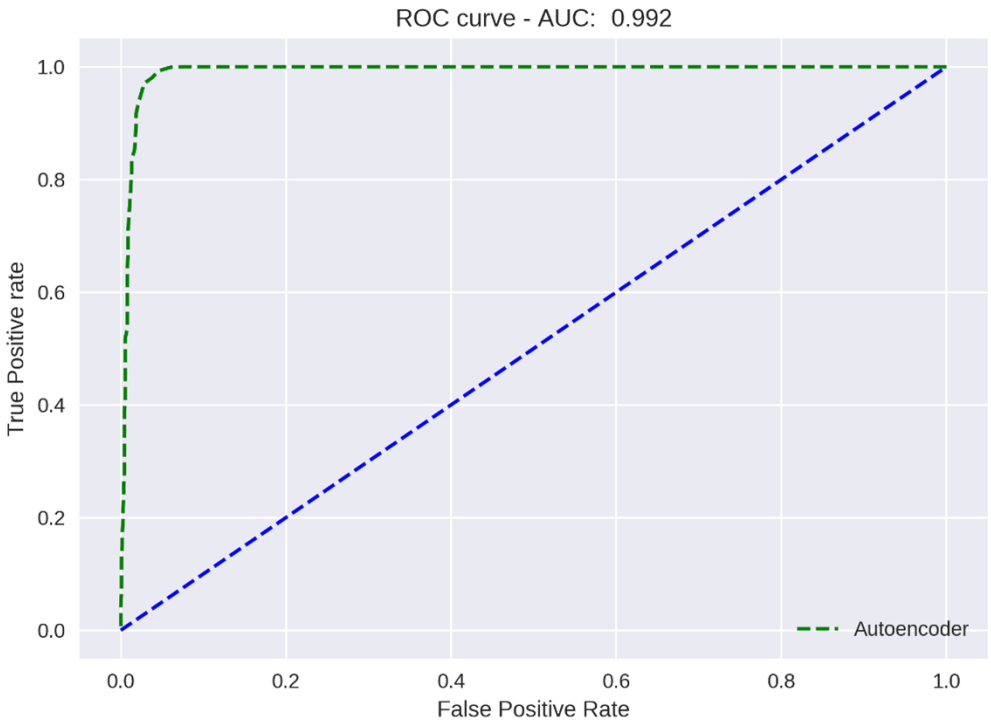

Figure 41 ROC curve of best performing model produced by the AutoDaedalus method in a single label experiment

When comparing the results of Table 27 and Table 28, we can see that both methods rendered very similar results in the single label experiment. Even the ROC curves in Figure 40 and Figure 41 show very similar results. This is mostly due to the fact, that both of them are close to perfection. This means that both experimental methods are able to distinguish between normal and anomalous

data instances with very high certainty. One of the reasons for such good results is the big difference between the normal and anomalous data instances that were compared, with 1 and 0 digits having a clear difference in shape and structure. Looking a little more critically at these tables, we see that the manual method performed a bit better in this experiment. The largest difference was in the recall metric and AUC score. Aside from that, we can say with strong confidence that both methods performed very well with our selected dataset.

The second experiment was designed to have a more complex dataset. The reason for this is that we wanted to ensure that the small difference between the two methods in the first experiment grew larger. Furthermore, datasets with multiple number labels in the valid class and a single number label in the anomaly class are thought to be more realistic. Since even human experts are unable to define rules that apply only to anomalies in real-world applications. Results from multi label experiments with the above NN models are presented in tables: Table 17, Table 18, Table 19, Table 20, Table 21, Table 22, Table 23, Table 24, Table 25, Table 26. Results from the multi label experiment are shown below.

NN Model	Depth on one side	Recall	Precision	F1-score	AUC
Manual method	1	0.223	0.219	0.221	0.748
Manual method	2	0.199	0.195	0.197	0.731
Manual method	3	0.191	0.187	0.189	0.729
Manual method	4	0.202	0.198	0.200	0.759
Manual method	5	0.218	0.214	0.216	0.776
Max		0.223	0.219	0.221	0.776
Total		1.033	1.013	1.023	3.743

Table 29 Multi label experiments result for a manual method

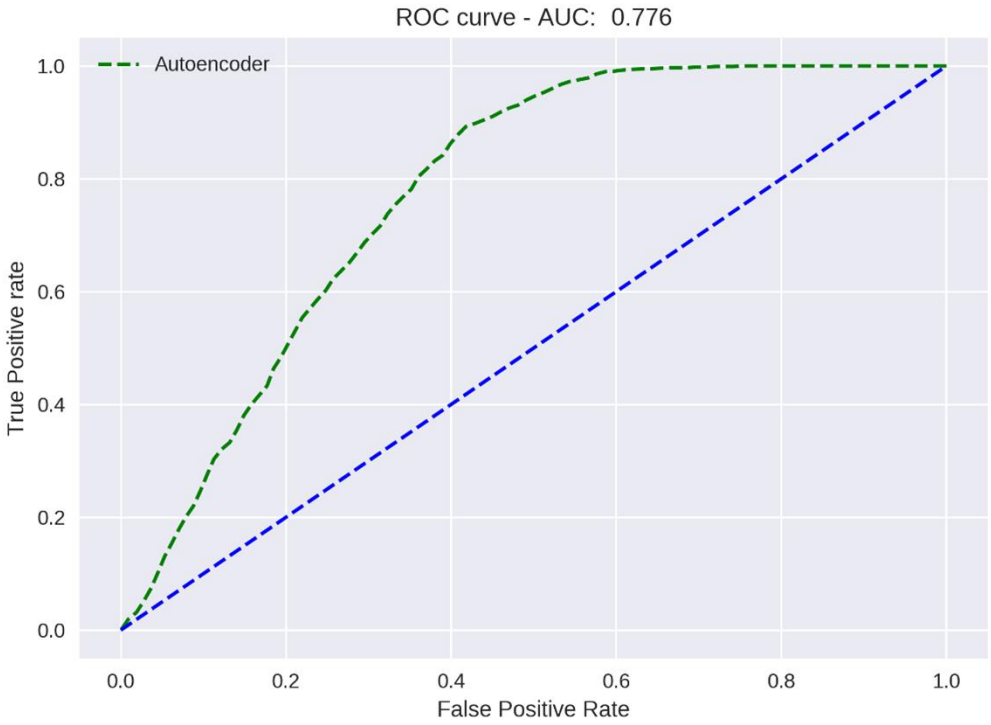

Figure 42 ROC curve of the best performing model produced by the manual method in the multi label experiment

NN Model	Depth on one side	Recall	Precision	F1-score	AUC
AutoDaedalus method	1	0.208	0.204	0.206	0.751
AutoDaedalus method	2	0.194	0.190	0.192	0.745
AutoDaedalus method	3	0.212	0.208	0.210	0.780
AutoDaedalus method	4	0.193	0.189	0.191	0.776
AutoDaedalus method	5	0.501	0.491	0.496	0.877
Max		0.501	0.491	0.496	0.877
Total		1.308	1.282	1.295	3.929

Table 30 Multi label experiments result for the AutoDaedalus method

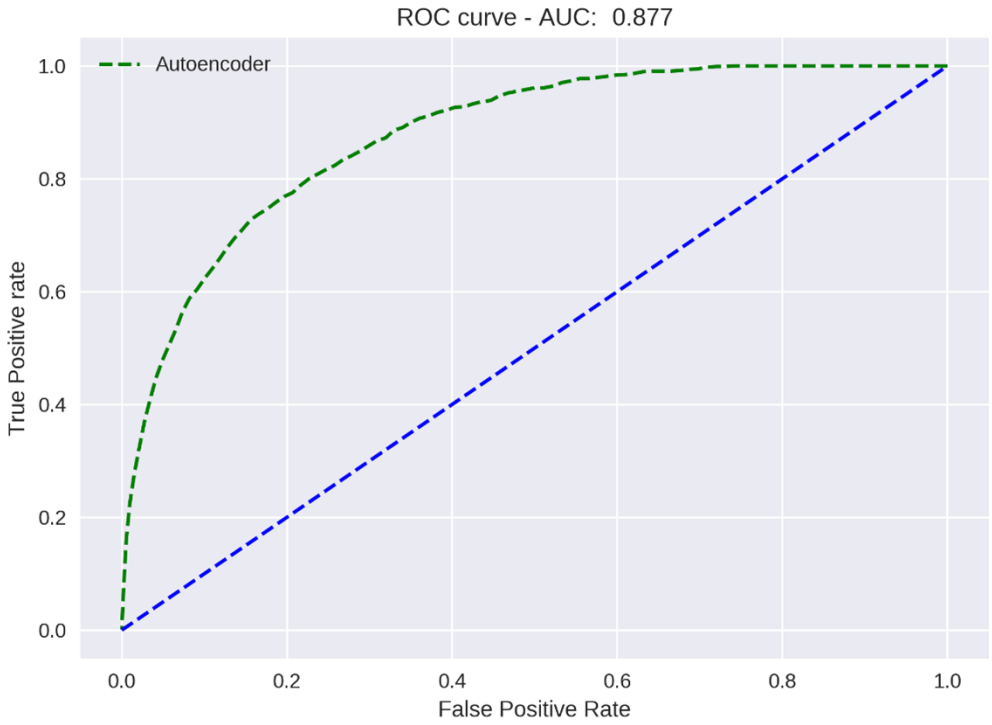

Figure 43 ROC curve of the best performing model produced by the AutoDaedalus method in the multi label experiment

This experiment was more computationally intensive and thus executed for a much longer period, especially when AutoDaedalus needed to find the best performing models out of 50 models (`max_depth=5, ant_count=10`). Nevertheless, the experiment furnished us with very interesting results. Looking at Table 29 and Table 30, we can say that both methods proved themselves with relatively good results, but in the end, AutoDaedalus rendered the better-performing NN model. As we can see all of the metrics were better when compared to the manual method. Also, the ROC curves in Figure 42 and Figure 43 show that the best AutoDaedalus model outperforms the best manually built model. When comparing the slopes of the curves, we can see that the one in Figure 43 is steeper than the one in Figure 42. As a result, TPR becomes bigger faster with fewer data instances compared to FPR, and therefore more correctly identified anomalies in the dataset.

The results from both experiments are presented in Table 31 and Figure 44.

NN Model	Experiment	Calculation	Recall	Precision	F1-score	AUC
Manual method	Single label	MAX	0.246	1.00	0.356	0.998
Manual method	Single label	TOTAL	1.106	4.981	1.772	4.986
Manual method	Multi label	MAX	0.223	0.219	0.221	0.776
Manual method	Multi label	TOTAL	1.033	1.013	1.023	3.743
AutoDaedalus method	Single label	MAX	0.215	0.995	0.354	0.992
AutoDaedalus method	Single label	TOTAL	1.066	4.934	1.756	4.956
AutoDaedalus method	Multi label	MAX	0.501	0.491	0.496	0.877
AutoDaedalus method	Multi label	TOTAL	1.308	1.282	1.295	3.929

Table 31 Comparison of experimental results (tabulated)

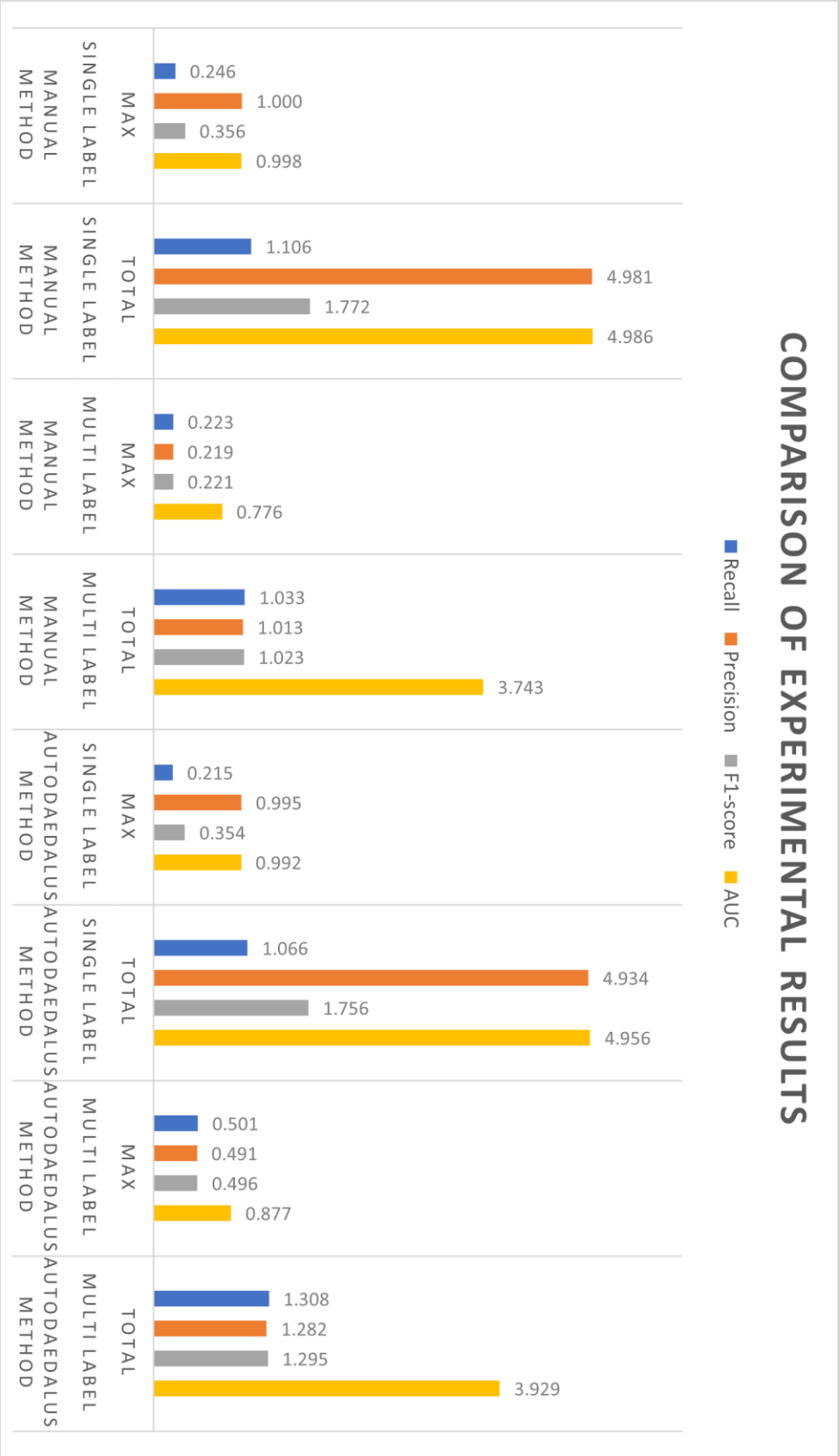

Figure 44 Comparison of experimental results (graphed)

Chapter VII

7 DISCUSSION

In this chapter, we will interoperate and discuss the experimental results. As we have seen in the previous chapter, we have split our testing into two experiments. First, we will discuss the first experiment where we had a dataset with one (1) valid class and one (0) anomaly class. Results have shown that both methods produced highly accurate anomaly detection in the MNIST dataset. This is also true for all model depths in both methods. It is interesting to see that AutoDaedalus generated NN models with a lot of different combinations of activation functions and output space dimensions, but was nevertheless able to render NN models with results that were almost identical to those rendered by the manual method. This is a good indication that not only the logically accepted architecture designed by human NN architects can perform well. Upon further inspection of the results of the first experiment we can say that the best NN models created by the manual models were at a depth of 3 and 4. In comparison, AutoDaedalus generated the best NN model with a depth of 3. To summarise, based on our measurements depth 3 is the peak value at which the NN model detects anomalies the best. Moving to the second experiment, we can easily see that the results were not as good as those of the first experiment. There are several reasons for this. The greater complexity of the dataset, with valid values in classes (1,2,3,4,5,6,7,8,9) and anomalous values in class 0, and the relativity simple type of layers (Dense) for such a task. If the precision metric which measures the ability of a model to identify only the relevant data points, reached a value of 1 (1 is the best) in the first experiment, then it had in the second experiment on average one-quarter of the value. Also, when looking into recall metrics we can see that both NN models from the manual and AutoDaedalus methods, had some difficulties identifying valid values as anomalies due to the large reconstruction losses of the NN models. The reason for this is that when the NN model was trained, it was unable to achieve better accuracy, mainly because of the simplicity of the architecture required for this task. When looking only at the F1-score we might

think that the results are not of much use, but we need to consider that a big factor is the acceptance of the value of the quantile that we chose for anomaly detection. The border between normal and anomaly in a dataset can be a relative term. Since there is no clear definition of what a true anomaly is in a specific dataset. An example in the MNIST dataset can be made of the numbers 1 and 7. To what extent will we claim that 1 does not look like 7, or vice versa? When considering this we must understand that the model found a specific data instance which it was unable to recreate with a small loss, and therefore it could still have been an anomaly even though its class represented the valid label. On the other hand, specific anomalous data instances can have a small reconstructed loss, and therefore would not be detected as an anomaly since it would not fall over the selected quantile. Nevertheless, looking back over the results of this experiment we can summarise that there was no particular outstanding NN model depth which was best among the manually created NN models, but on the AutoDaedalus side a clear winner was the NN model with a depth of 5. Not only was it the best in its class, but it was also the best in the whole second experiment. Its anomaly detection success was more than half. From the output in log files, we found that it identified 491 anomalies as true positives out of all 980 anomalies in the dataset. With this finding, we can safely conclude that our AutoDaedalus method offers competitive performance compared to our list of manually created NN models with a simple dataset and even better results than the manual method when faced with a more complex dataset. We assume that a reason for the better performing NN model could lie in its architecture since in a complex dataset a key to better performance can be a combination of different activation functions and a mixed distribution of output space dimensionality.

With all tests under the hood, we can conclude our hypothesis. When answering the first research question RQ1, we can accept the H1 hypothesis, since the total sum of the F1-score and AUC for all models grouped by method is 3.5% higher with the AutoDaedalus method than with the manual method. When finding the answer to the second research question RQ2, we need to limit the scope of the question. If we are looking for the best overall NN model, then the H2 hypothesis is true. But if we are looking at all constructed NN models with both methods, the result of the H2 hypothesis is negative. Therefore, we partly accept the H2 hypothesis. When it comes to acceptance of the H3 hypothesis, which answers the research question RQ3 we can strongly accept it, since swarm algorithms, in our case ACO, proved themselves when building autoencoder architectures. Hypothesis H3 is accepted. We end this discussion with the acceptance of the thesis as well since our experiments proved that novel NN models can be designed by the AutoDaedalus method.

Chapter VIII

8 CONCLUSION

At the beginning of our master's thesis, we set five goals to help us answer the research questions that interested us. Throughout the whole work, we have focused on providing the necessary theory and development of tools in order to obtain solid outcomes, and to confirm the hypotheses based on these research questions.

We have followed up the master's thesis by reviewing the necessary theory. With this, we gained additional knowledge on multiple topics. At the start, we briefly introduced machine learning concepts, which are needed to understand neural networks. We wanted to understand how various types of NN architectures are formed and how certain activation functions help during the DNN training. All this knowledge was a great help in understanding what possibilities there are for the construction of NNs. Since we wanted to build a model that would be capable of self-building NN architectures, we needed to understand the main parts of the NAS technique. We learned that the search space represents the edge boundaries, where the search strategy operates when it comes to the construction of NN architectures. In practice, the search space can include concepts such as layer types, how they are connected, and various parameters. Finally, we needed a performance estimation technique, which in our case would need to be based on a swarm intelligence system. Once we gathered the knowledge of NN construction, we moved on to understanding what anomalies are and how best to detect them. Before detecting anomalies we needed to understand different types, such as the point, contextual and collective anomaly. In addition, we wanted to understand what data noise is and how it differs from an anomaly. Until now, we have looked at the entire setup of the NAS method and its objective in our work. We next started to research how we may provide a solution to the aforementioned anomaly detection. We proposed a special type of NN autoencoder as a solution whose natural architecture is tailored to our needs. We explored autoencoder properties such as encoder, latent space, decoder, model depth, autoencoder types,

and applications in which it may be utilised. The final part of our research was to find out how we can automatically construct multiple autoencoder models and evolve them by use of the ACO algorithm. As we learned from the ACO algorithm, if one of the ants is not able to construct a NN model with good performance, others follow stigmergically and continue the exploration until the best NN model is constructed. With all of the theory in place, the practical phase could begin. We completed the goals of our master's thesis by building the open-source AutoDaedalus programme which is capable of constructing new autoencoder topologies using the ACO algorithm based on the search space limitations specified in the configuration file. We may use it to identify the best performing NN models for anomaly detection. During the implementation, we explained the workflow of AutoDaedalus with several examples and infographics. In the experimental chapter, we have reported two experiments and compared the manual and AutoDaedalus methods on the MNIST dataset to detect as many anomalies as possible. The purpose of the experiments was to obtain answers to our research questions and hypotheses. When we compared the two methods, we discovered that there is no significant difference between them for a simple dataset. Differences became noticeable when we began the second experiment, which used a considerably more complicated dataset. In this experiment, our proposed method proved its effectiveness by constructing a better performing NN model. At the end of our master’s thesis, we can conclude that the NAS technique using swarm intelligence as a search strategy can construct novel autoencoder architectures that can be deployed for anomaly detection.

9 CITATIONS AND BIBLIOGRAPHY

[1] "Machine Learning textbook." https://www.cs.cmu.edu/~tom/mlbook.html (accessed Jul. 24, 2021).

[2] "What is Machine Learning?," Jul. 01, 2021. https://www.ibm.com/cloud/learn/machine-learning (accessed Jul. 24, 2021).

[3] "What Is Machine Learning? - I School Online," *UCB-UMT*, Jun. 26, 2020. https://ischoolonline.berkeley.edu/blog/what-is-machine-learning/ (accessed Jul. 24, 2021).

[4] A. K. Jain, J. Mao, and K. M. Mohiuddin, "Artificial neural networks: a tutorial," *Computer*, vol. 29, no. 3, pp. 31–44, Mar. 1996, doi: 10.1109/2.485891.

[5] S. Arora, "Supervised vs Unsupervised vs Reinforcement," *AITUDE*, Jan. 29, 2020. https://www.aitude.com/supervised-vs-unsupervised-vs-reinforcement/ (accessed Jul. 04, 2021).

[6] T. Dietterich, "Overfitting and undercomputing in machine learning," *ACM Comput. Surv.*, vol. 27, no. 3, pp. 326–327, Sep. 1995, doi: 10.1145/212094.212114.

[7] A. Mikołajczyk and M. Grochowski, "Data augmentation for improving deep learning in image classification problem," in *2018 International Interdisciplinary PhD Workshop (IIPhDW)*, May 2018, pp. 117–122. doi: 10.1109/IIPHDW.2018.8388338.

[8] C. Shorten and T. M. Khoshgoftaar, "A survey on Image Data Augmentation for Deep Learning," *J. Big Data*, vol. 6, no. 1, p. 60, Jul. 2019, doi: 10.1186/s40537-019-0197-0.

[9] "What is Deep Learning?," Jun. 30, 2021. https://www.ibm.com/cloud/learn/deep-learning (accessed Jul. 24, 2021).

[10] J. Dean, "Large-Scale Deep Learning for Intelligent Computer Systems," p. 69.

[11] E. Brynjolfsson and A. Mcafee, "ARTIFICIAL INTELLIGENCE, FOR REAL," *Artif. Intell.*, p. 31.

[12] P. H. Sydenham and R. Thorn, Eds., *Handbook of measuring system design*. Chichester, England: Wiley, 2005.

[13] "Deep Learning." https://www.deeplearningbook.org/ (accessed Jul. 03, 2021).

[14] J. Brownlee, "How to Choose an Activation Function for Deep Learning," *Machine Learning Mastery*, Jan. 17, 2021. https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/ (accessed Jul. 03, 2021).

[15] G. Kyriakides and K. Margaritis, "NORD: A python framework for Neural Architecture Search," *Softw. Impacts*, vol. 6, p. 100042, Nov. 2020, doi: 10.1016/j.simpa.2020.100042.

[16] T. Elsken, J. H. Metzen, and F. Hutter, "Neural Architecture Search: A Survey," *ArXiv180805377 Cs Stat*, Apr. 2019, Accessed: Feb. 28, 2021. [Online]. Available: http://arxiv.org/abs/1808.05377

[17] D. Zhou *et al.*, "AutoSpace: Neural Architecture Search with Less Human Interference," *ArXiv210311833 Cs*, Mar. 2021, Accessed: Jun. 13, 2021. [Online]. Available: http://arxiv.org/abs/2103.11833

[18] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Comput. Surv.*, vol. 41, no. 3, p. 15:1-15:58, Jul. 2009, doi: 10.1145/1541880.1541882.

[19] M. Ahmed, A. N. Mahmood, and Md. R. Islam, "A survey of anomaly detection techniques in financial domain," *Future Gener. Comput. Syst.*, vol. 55, pp. 278–288, Feb. 2016, doi: 10.1016/j.future.2015.01.001.

[20] "How Airbus Detects Anomalies in ISS Telemetry Data Using TFX." https://blog.tensorflow.org/2020/04/how-airbus-detects-anomalies-iss-telemetry-data-tfx.html (accessed May 28, 2021).

[21] E. Aleskerov, B. Freisleben, and B. Rao, "CARDWATCH: a neural network based database mining system for credit card fraud detection," in *Proceedings of the IEEE/IAFE 1997 Computational*

Intelligence for Financial Engineering (CIFEr), Mar. 1997, pp. 220–226. doi: 10.1109/CIFER.1997.618940.
[22] T. Matek, "Anomaly detection in computer networks using higher-order dependencies," masters, Univerza v Ljubljani, 2017. Accessed: May 28, 2021. [Online]. Available: http://eprints.fri.uni-lj.si/3992/
[23] C. C. Aggarwal, *Outlier Analysis*. Cham: Springer International Publishing, 2017. doi: 10.1007/978-3-319-47578-3.
[24] M. Goldstein and S. Uchida, "A Comparative Evaluation of Unsupervised Anomaly Detection Algorithms for Multivariate Data," *PLOS ONE*, vol. 11, no. 4, p. e0152173, Apr. 2016, doi: 10.1371/journal.pone.0152173.
[25] E. Eskin, A. Arnold, M. Prerau, L. Portnoy, and S. Stolfo, "A Geometric Framework for Unsupervised Anomaly Detection," in *Applications of Data Mining in Computer Security*, D. Barbará and S. Jajodia, Eds. Boston, MA: Springer US, 2002, pp. 77–101. doi: 10.1007/978-1-4615-0953-0_4.
[26] D. Bank, N. Koenigstein, and R. Giryes, "Autoencoders," *ArXiv200305991 Cs Stat*, Apr. 2021, Accessed: Jul. 08, 2021. [Online]. Available: http://arxiv.org/abs/2003.05991
[27] A. Radhakrishnan, K. Yang, M. Belkin, and C. Uhler, "Memorization in Overparameterized Autoencoders," *ArXiv181010333 Cs Stat*, Sep. 2019, Accessed: Jul. 11, 2021. [Online]. Available: http://arxiv.org/abs/1810.10333
[28] "The Great Autoencoder Bake Off," *Don't Repeat Yourself*, Jan. 24, 2021. /autoencoders/2021/01/24/Autoencoder_Bake_Off.html (accessed Apr. 14, 2021).
[29] E. Nalisnick, A. Matsukawa, Y. W. Teh, D. Gorur, and B. Lakshminarayanan, "Do Deep Generative Models Know What They Don't Know?," *ArXiv181009136 Cs Stat*, Feb. 2019, Accessed: Jul. 11, 2021. [Online]. Available: http://arxiv.org/abs/1810.09136
[30] S. De, A. Maity, V. Goel, S. Shitole, and A. Bhattacharya, "Predicting the popularity of instagram posts for a lifestyle magazine using deep learning," in *2017 2nd International Conference on Communication Systems, Computing and IT Applications (CSCITA)*, Apr. 2017, pp. 174–177. doi: 10.1109/CSCITA.2017.8066548.
[31] L. Theis, W. Shi, A. Cunningham, and F. Huszár, "Lossy Image Compression with Compressive Autoencoders," *ArXiv170300395 Cs Stat*, Mar. 2017, Accessed: Jul. 11, 2021. [Online]. Available: http://arxiv.org/abs/1703.00395
[32] L. Gondara, "Medical image denoising using convolutional denoising autoencoders," *2016 IEEE 16th Int. Conf. Data Min. Workshop ICDMW*, pp. 241–246, Dec. 2016, doi: 10.1109/ICDMW.2016.0041.
[33] T.-H. Song, V. Sanchez, H. EIDaly, and N. Rajpoot, "Hybrid deep autoencoder with Curvature Gaussian for detection of various types of cells in bone marrow trephine biopsy images," *2017 IEEE 14th Int. Symp. Biomed. Imaging ISBI 2017*, 2017, doi: 10.1109/ISBI.2017.7950694.
[34] K. O. Stanley, J. Clune, J. Lehman, and R. Miikkulainen, "Designing neural networks through neuroevolution," *Nat. Mach. Intell.*, vol. 1, no. 1, pp. 24–35, Jan. 2019, doi: 10.1038/s42256-018-0006-z.
[35] J. Lehman and R. Miikkulainen, "Neuroevolution," *Scholarpedia*, vol. 8, no. 6, p. 30977, Jun. 2013, doi: 10.4249/scholarpedia.30977.
[36] "Encyclopedia of Evolutionary Biology | ScienceDirect." https://www.sciencedirect.com/referencework/9780128004265/encyclopedia-of-evolutionary-biology (accessed Jul. 15, 2021).
[37] S. Sen, "Chapter 4 - A Survey of Intrusion Detection Systems Using Evolutionary Computation," in *Bio-Inspired Computation in Telecommunications*, X.-S. Yang, S. F. Chien, and T. O. Ting, Eds. Boston: Morgan Kaufmann, 2015, pp. 73–94. doi: 10.1016/B978-0-12-801538-4.00004-5.
[38] D. Shrivastava, S. Sanyal, A. K. Maji, and D. Kandar, "Chapter 17 - Bone cancer detection using machine learning techniques," in *Smart Healthcare for Disease Diagnosis and Prevention*, S.

Paul and D. Bhatia, Eds. Academic Press, 2020, pp. 175–183. doi: 10.1016/B978-0-12-817913-0.00017-1.
[39] G. Vrbančič, I. Fister, and V. Podgorelec, "Swarm Intelligence Approaches for Parameter Setting of Deep Learning Neural Network: Case Study on Phishing Websites Classification," in *Proceedings of the 8th International Conference on Web Intelligence, Mining and Semantics*, New York, NY, USA, Jun. 2018, pp. 1–8. doi: 10.1145/3227609.3227655.
[40] "comparison - What is the difference between artificial intelligence and swarm intelligence?," *Artificial Intelligence Stack Exchange*. https://ai.stackexchange.com/questions/21142/what-is-the-difference-between-artificial-intelligence-and-swarm-intelligence (accessed Jul. 16, 2021).
[41] L. Brezočnik, I. Fister, and V. Podgorelec, "Swarm Intelligence Algorithms for Feature Selection: A Review," *Appl. Sci.*, vol. 8, no. 9, Art. no. 9, Sep. 2018, doi: 10.3390/app8091521.
[42] M. Dorigo, M. Birattari, and T. Stutzle, "Ant colony optimization," *IEEE Comput. Intell. Mag.*, vol. 1, no. 4, pp. 28–39, Nov. 2006, doi: 10.1109/MCI.2006.329691.
[43] M. Thuma, "OPTIMIZACIJA S KOLONIJAMI MRAVELJ," 2013. https://dk.um.si/IzpisGradiva.php?id=42762 (accessed Jul. 16, 2021).
[44] J.-L. Deneubourg, S. Aron, S. Goss, and J. M. Pasteels, "The self-organizing exploratory pattern of the argentine ant," *J. Insect Behav.*, vol. 3, no. 2, pp. 159–168, Mar. 1990, doi: 10.1007/BF01417909.
[45] C. Blum, "Ant colony optimization: Introduction and recent trends," *Phys. Life Rev.*, vol. 2, no. 4, pp. 353–373, Dec. 2005, doi: 10.1016/j.plrev.2005.10.001.
[46] J. Hajewski, S. Oliveira, and X. Xing, "Distributed Evolution of Deep Autoencoders," *ArXiv200407607 Cs*, Apr. 2020, Accessed: Jul. 24, 2021. [Online]. Available: http://arxiv.org/abs/2004.07607
[47] S. Lander and Y. Shang, "EvoAE – A New Evolutionary Method for Training Autoencoders for Deep Learning Networks," in *2015 IEEE 39th Annual Computer Software and Applications Conference*, Jul. 2015, vol. 2, pp. 790–795. doi: 10.1109/COMPSAC.2015.63.
[48] J. Hajewski and S. Oliveira, "An Evolutionary Approach to Variational Autoencoders," in *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*, Jan. 2020, pp. 0071–0077. doi: 10.1109/CCWC47524.2020.9031239.
[49] F. Charte, A. J. Rivera, F. Martínez, and M. J. del Jesus, "EvoAAA: An evolutionary methodology for automated neural autoencoder architecture search," *Integr. Comput.-Aided Eng.*, vol. 27, no. 3, pp. 211–231, Jan. 2020, doi: 10.3233/ICA-200619.
[50] E. Byla and W. Pang, "DeepSwarm: Optimising Convolutional Neural Networks using Swarm Intelligence," *ArXiv190507350 Cs Stat*, May 2019, Accessed: Mar. 12, 2021. [Online]. Available: http://arxiv.org/abs/1905.07350
[51] "CUDA," *Wikipedia*. Jul. 03, 2021. Accessed: Aug. 23, 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=CUDA&oldid=1031775035
[52] "Lambda Stack: an AI software stack that's always up-to-date." https://lambdalabs.com/lambda-stack-deep-learning-software (accessed Aug. 23, 2021).