A Deep Learning Toolkit for High-Dimensional, Sequential Data

James Patrick Robert O' Donoghue

Bachelor of Music (Hons)

Graduate Diploma in Information Technology

A Dissertation submitted in fulfilment of the

requirements for the award of

Doctor of Philosophy (Ph.D.)

to

Dublin City University

Faculty of Engineering and Computing, School of Computing

Supervisor: Mark Roantree

Janurary 2017

Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Doctor of Philosophy is entirely my own work, and that I have exercised reasonable care to ensure that the work is original, and does not to the best of my knowledge breach any law of copyright, and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: ____

ID No.: 11212487

Date: 6th January 2017

Acknowledgements

There was a long road to get to the end of this thesis, and I would like thank everyone who helped me along the way. First of all, I would like to thank my supervisor, Dr. Mark Roantree for taking a chance on someone whose primary degree was in music. I probably would not have had the opportunity to do a PhD otherwise. His guidance and perspective enabled the moulding of some very raw ideas into a piece of work and a dissertation I am proud to present.

Many thanks also to: Andrew, for helping my understanding of statistics; Michael, for proofing papers and discussing ideas; Cian, for a developers perspective; and Gavin, for sending that deep learning paper and the maths help.

In an endeavour such as this, emotional and technical support are equally as important. Therefore, I would like to thank my parents, sisters, brother, in-laws and my nieces and nephews for their love and support. Your company in Dublin, and the trips to see you in Kerry kept me sane. In particular, I would like to thank my dad, Finán, for taking time out on his Christmas holidays to proof this thesis, and my mom, Elizabeth, for always sending me back with steaks. They were a crucial energy source for my brain.

I would also like to thank my friends for their support, sitting through numerous practice presentations and trying not to let your eyes glaze over when I talked about multi-layered neural networks could not have been easy!

Last but certainly not least, I would like to thank my wonderful girlfriend Fiona. I think driving from Galway to Dublin, late on a rainy New Years eve to surprise me (with dinner), when I was on my own, in the final days of writing, is indicative of the love and support you gave me throughout. Thanks for being there.

Contents

1 Introduction **1**

1.1 Data Mining and Machine Learning 1

1.2 Deep Learning 3

1.3 Practical Machine Learning Problems 5

1.3.1 Deep Learning Application Goals 7

1.4 Problem Statement and Hypothesis 9

1.4.1 Hypothesis 10

1.4.2 Research Questions and Contribution 10

1.5 Summary and Dissertation Structure 12

2 Related Research **15**

2.1 Applied Learning: Contrasting Shallow and Deep 15

2.1.1 Anomaly Detection 16

2.1.2 Representation Learning 21

2.1.3 Sequential Data Mining 26

2.1.4 Hyper-Parameter Selection 29

2.1.5 Summary: Application Issues and Requirements 31

2.2 Machine Learning and Data Models 32

2.2.1 Summary: Data Model Issues and Requirements 35

2.3 Deep Learning Frameworks 36

2.3.1 Summary: Framework Issues and Requirements 38

2.4 Summary of Related Research 39

3 Neural Networks **41**
3.1 Basic Concepts 42
3.1.1 Dataset Concepts 42
3.1.2 Neural Network Concepts 43
3.2 Neural Network Components 46
3.2.1 Feed Forward Hidden Layer 48
3.2.2 Recurrent Layer 52
3.2.3 Output Regression Layer 54
3.3 Neural Network Types 64
3.3.1 Multi-Layer Perceptron 64
3.3.2 Recurrent Neural Network 69
3.3.3 Restricted Boltzmann Machine 70
3.3.4 Deep Belief Network 77
3.4 Summary 80

4 Methodology and Architecture **81**
4.1 An Approach to Deep Learning 81
4.1.1 Preparation Stage 85
4.1.2 Learning Stage 89
4.1.3 Post-Processing Stage 90
4.2 POL and the Deep Learning Architecture 91
4.2.1 Interoperable Storage Layer 91
4.2.2 Toolkit Layer 93
4.2.3 Experiment Layer 95
4.3 Summary 95

5 Configuring Deep Learners **97**
5.1 Configurable Stages 97
5.2 Architecture and Constructs 99
5.3 Deployment Architecture 103
5.4 Optimisation 106
5.5 Summary 110

6 Deep Learning Data Model **111**

6.1 The POL Data Model 111

6.1.1 Experiment type 113

6.1.2 Dataset type 114

6.1.3 Learner Type 117

6.2 Storing and Reloading Deep Learners 123

6.2.1 DLDB Technologies 125

6.2.2 The Deep Learning Experiment Database 127

6.3 Analysing Performance Data 128

6.3.1 Setting Up and Accessing Deep Learning Storage 128

6.3.2 Evaluating and Visualising Deep Learning Experiments . . . 130

6.4 Summary 135

7 Evaluation **136**

7.1 Evaluation Overview and Setup 136

7.2 Case Study 1: Predicting Heart Rates 139

7.2.1 Predictions and Hyper-Parameter Optimisation 141

7.2.2 Hyper-Parameter Interpretation 142

7.2.3 Summary for Case Study 1 145

7.3 Case Study 2: Predicting Patient Survival Rates 146

7.3.1 Hyper-Parameter Optimisation and Interpretation 150

7.3.2 Prediction Analysis 156

7.3.3 Summary for Case Study 2 160

7.4 Case Study 3: Detecting Unusual Chords 161

7.4.1 Hyper Parameter Optimisation 164

7.4.2 Anomaly Detection 165

7.4.3 Feature Representation and Model Parameter Analysis 169

7.4.4 Summary for Case Study 3 176

7.5 Summary 177

8 Conclusions **178**

8.1 Thesis Summary 178

8.2 Limitations 180
8.3 Future Research 180
8.3.1 A Standard Data Model for Deep Learning 181
8.3.2 Experiment Databases 181
8.3.3 Future Toolkit Applications 182
8.3.4 Future Toolkit Development 182
8.4 Closing Summary 183

Bibliography **184**

Glossary of Terms **201**

Appendices **203**

A Notation **204**

B Activations and Hypotheses **205**
B.1 Node Level Linear Activation Representations 205
B.2 Alternative Rectified Linear Units 206
B.3 Deriving Free Energy Functions 206

C Cost Functions **208**
C.1 Deriving Negative Log Likelihood from Likelihood 208

D Partial Derivatives **209**
D.1 Derivative Rules 209
D.1.1 Chain Rule 209
D.1.2 Quotient Rule 209
D.2 Activation Function Partial Derivatives 210
D.2.1 Derivative Linear Function wrt Theta 210
D.2.2 Derivative of Sigmoid Function wrt Input 210
D.2.3 Derivative of Softmax Function wrt Hypothesis Output . . . 210
D.3 Cost Function Partial Derivatives 212
D.3.1 Derivative of Mean Squared Error with respect to Parameters 212

D.3.2 Derivative of Cross Entropy Cost wrt a 213
D.3.3 Derivative of Negative Log Likelihood wrt a 213
D.3.4 Derivative of Negative Log Likelihood wrt Theta 214

E Storage Overview **216**

F System Screenshots **217**

List of Figures

3.1 Dataset 42
3.2 Neural Network Components 44
3.3 Calculating Activations 45
3.4 Hidden Layer: Hidden to Hidden 48
3.5 Different Node Routes 50
3.6 Recurrent Node Unfolded in Time 53
3.7 Single Output Regression Representation 54
3.8 Multiple Output Regression Representation 55
3.9 Single Class Multi-Layer Perceptron 65
3.10 Restricted Boltzmann Machine 72
3.11 Deep Belief Network 78
3.12 Pre-Training DBN Layers as RBMS 79

4.1 Data Mining Framework for Deep Learning Overview 83
4.2 Feature Categorisations and Transformations 86
4.3 Deep Learning System Architecture 92

5.1 Experiment Stages Overview 98
5.2 Overview of Configurable Deep Network Design 100
5.3 Configurable Deep Network 103

6.1 POL Conceptual Model 112
6.2 NoSQL Storage Model for Deep Learning 127

7.1 Sports Performance Star Schema 139

7.2 Activations 143
7.3 Hidden Nodes 143
7.4 Gradient Truncate 144
7.5 Learning Rate 144
7.6 Batch Size 145
7.7 Updates 145
7.8 Top 20 Hidden Layers 152
7.9 Top 20 Layer 1 Nodes 152
7.10 Top 20 Alpha 153
7.11 Top 20 Lambda 153
7.12 Top 20 Updates 154
7.13 Top 20 Epochs 154
7.14 Top 20 Batch Sizes 154
7.15 Trial 24 Learning Curves 158
7.16 Trial 32 Learning Curves 158
7.17 Free Energy Distribution of Top Level RBM for Trial 13 Run 9 . . . 166

F.1 Example of CDN Configuration 217
F.2 Example of Toolkit Hyper-Parameter Evaluation 218
F.3 Example of Toolkit Model Parameter Evaluation 218

List of Tables

2.1 Summary: Application Issues and Requirements 31
2.2 Deep Learning Data Issues and Requirements 35
2.3 Deep Learning Framework Issues and Requirements 38
2.4 Requirements Summary and Design Principals 40

4.1 Candidate Labelling Assignment 88

7.1 Evaluation Goals 137
7.2 gaa_rnn_05-04-16_11:48:02 Hyper-Parameter Search Space 140
7.3 Case Study 1: Top 10 Scores 142
7.4 Hyper-Parameter Summary Statistics 142
7.5 Features used for MAAS experiment 147
7.6 Survival Overview 148
7.7 Hyper-Parameter Search Space 150
7.8 MAAS DBN Top Architecture Hyper-Parameters 151
7.9 MAAS DBN Top Training Hyper-Parameters 153
7.10 MAAS DBN Top Performing Scores 157
7.11 Result of Query to Updates for Trial 24 159
7.12 Bach Chord Label Class Distribution 163
7.13 Bach Stacked RBM Hyper-Parameter Search Bounds 164
7.14 Bach Stacked RBM Scores and Architectures 165
7.15 Bach Anomalies 167
7.16 Frequency Count of Each Feature Occurrences 169
7.17 $L^{(1)}$ Significant Inputs 171

7.18 $L^{(2)}$ Significant Inputs 175
7.19 $L^{(3)}$ Significant Inputs 175

A.1 Mathematical Notation 204

E.1 Storage Size Statistics 216
E.2 Snapshot Counts 216

Abstract

Deep learning is a more recent form of machine learning based on a set of algorithms that attempt to learn using a deep graph with multiple processing layers, where layers are composed of multiple linear and non-linear transformational nodes. While research in this area has shown to improve the predictive accuracy in a number of domains, deep learning systems are highly complex and experiments can be hard to manage. In this dissertation, we present a deep learning system, built from scratch, which enables fully configurable deep learning experiments. By configurable, we mean selecting the overall learning algorithm, the number of layers within the deep network, the nodes within network layers and the propagation functions deployed at each node. We use a range of deep network configurations together with different datasets to illustrate the potential of this system but also to highlight the difficulties in tuning the model and hyper-parameters to maximise accuracy. Our research also provides a conceptual data model to capture all aspects of deep learning experiments. By specifying a conceptual model, it provides a platform for the storage and management of experimental snapshots, a key support for experiment and parameter optimisation and analysis. In addition, we developed a toolkit which supports the management and analysis of deep learning experiments and provides a new method for pausing and calibrating experiments. It also offers possibilities for interchanging experiment setup and results between deep learning researchers. Our validation takes the form of a series of case studies built from the requirements of end users and demonstrates the effectiveness of our toolkit in building deep learning algorithms.

Chapter 1

Introduction

This dissertation is about data mining with a specialised form of machine learning called *Deep Learning.* As an introduction, in §1.1 we provide a brief overview of *machine learning* and *data mining* and the relationship between the two, as both terms are often used in similar bodies of work. In §1.2, we then introduce the specialised form of machine learning known as *deep learning.* This is a state of the art machine learning technique which has great potential for traditional data mining tasks, but the complexity and scope of running and analysing deep learning experiments make its practical application is notoriously difficult. This motivates the research presented here, as traditional data-mining paradigms are ill-suited to the usage of deep networks in many applications areas. We discuss some of these real world applications and a context for our work in §1.3. We define our hypothesis and outline the contributions made during the course of this research in §1.4. Finally, we present a summary and outline the structure of the dissertation in §1.5.

1.1 Data Mining and Machine Learning

Data mining is a diverse, inter-disciplinary sub-field of computer science. Also known as knowledge discovery in databases or computational data analysis, data mining draws from areas such as database systems, data warehousing, statistics, machine learning, and often overlaps with data visualisation, information retrieval, and high-performance computing [57], [47]. It emerged in the 1980s as computational power, and data capture and storage tools vastly improved [57]. Data analysis was originally the remit of statisticians, who considered the lack of an apriori hypothesis bad practice. Data mining can be described as the process of discovering some previously unknown information from a dataset [57], or the extraction of interesting patterns, automatically or semi-automatically, from large datasets [137].

Machine learning is the method used to devise complex models and algorithms which lend themselves to various analytical tasks during the data mining process. Machine learning is used to accomplish tasks such as prediction, which in the commercial context is known as *predictive analytics*. A computer program is said to *learn* if its performance on a task improves with experience, according to some performance measure [93]. These learned analytical models allow researchers, data scientists, engineers, and analysts to 'produce reliable, repeatable decisions and results' and uncover 'hidden insights' through learning from historical relationships and trends in the data.

Both Machine learning and data mining involve the study of algorithms that can extract information automatically. There is a significant overlap between the two fields, with the use of very similar algorithms which can be traced back many years [36]. Furthermore, data mining is a research topic that has taken much of its inspiration and techniques from machine learning but with different goals. Whereas data mining looks to extract knowledge and actionable, meaningful insights for *human* stakeholders, machine learning looks to discover how *machines* can learn from data and improve their performance on a task [57].

Data mining therefore, encompasses a broader process which relates to a person carrying out an analysis of data, in a specific situation, on a particular data set, with a goal in mind. Typically, this person will also have to select and engineer data, leverage the power of the various pattern recognition techniques that have been developed in machine learning and finally, present the knowledge gleaned from the data in an easily consumable format for another person. Quite often, the dataset is large and complicated, containing special problems such as having more variables than observations. Usually, the goal is either to generate some preliminary insights in an area where there was little knowledge beforehand, or to predict future observations accurately. Moreover, analytical procedures can be either ‘unsupervised’ (we do not know the answer: *discovery*) or ‘supervised’ (we do know the answer: *prediction*). Common data mining techniques include cluster analyses, classification and regression trees, and neural networks.

Bordawekar et. al [22] present the main functional goals and problem-types of data analytics, found after a two year study of the area. The functional goal of this research is *prediction*, and its associated problem types are: *supervised* and *unsupervised learning*; as well as *descriptive* and *inferential statistics.* This research will also encounter *feature learning* (or dimensionality reduction) and *semi-supervised learning*, as some datasets used in this research are high-dimensional and not all outcomes are labelled. Explanations of the problem types are as follows:

- **Supervised learning**: predicts the class of an unlabelled sample based on previous labelled observations.
- **Unsupervised learning**: segments or finds patterns in unlabelled data where grouped samples have common traits.
- **Semi-supervised learning**: uses both labelled and unlabelled data during the training process.
- **Descriptive and inferential statistics**: employs statistical modelling to describe the dataset or infer information from it.
- **Feature learning/dimensionality reduction**: reduces the feature-space in which the data exists or learns those variable interactions most relevant to an outcome.

1.2 Deep Learning

There is a limit to what can be done with techniques traditionally used to approach the analytics problems described in §1.1. If more than one problem-type is encountered in a dataset, the use of many shallow algorithms in conjunction is required. For example, using a feature selection method and subsequently using a classification algorithm to make predictions. Shallow refers to the *depth* of the algorithm's architecture, specifically, the number of *levels* of learning function operations [12]. Any architecture with *less than three* layers of learning functions is considered shallow as they consist of one, or at most, two layers.

The desire to approach the problems outlined in §1.1 with a single *end-to-end* solution consisting of a *deep* architecture has existed for some time [12]. The realisation of this came with the advent of Deep Learning.

Deep learning refers to a recent breakthrough in machine learning, where *deep architectures*, made up of many levels or *layers* of non-linear operations are used to model data. A theory that the brain is organised in a deep architecture, abstracting input information into multiple levels of meaning, where each corresponds to a different location in the cortex [12], inspired the creation of these algorithms. Therefore, a central premise behind deep learning is that, like the brain, these algorithms can learn high-level, abstract features from data [13]. These high-level features better represent the outcome or dataset being modelled and correspond to *latent* variables in the dataset [13]. The lower layers in deep architectures correspond to localised, specific learned features and as the data progresses, or is *fed forward* through the architecture, it is transformed into even more abstract representations where the layers deeper in the architecture correspond to higher level representations.

We now give an example of one of many possible uses in a clinical setting. The input we provide to a deep neural network could be a wide-range of low-level biometric features, such as height and weight. The network could then transform and learn those most relevant to a latent concept like obesity, which could then be combined with other latent variables to represent an even higher level concept such as metabolic conditions. We will explain the exact processes behind how these high-level features are computed in Chapter 3.

The Deep Belief Network in [60] is often seen as the first deep learning algorithm and the paper that began a more serious focus on deep learning, although recent surveys [117] argue that deep learning has existed for much longer. The Deep Belief Network [60] solved the characteristic problem of *vanishing gradients* for deep, *feed-forward*, neural network architectures but, Long Short Term Memory [63] had already solved this problem for *recurrent* architectures. Recurrent architectures are used in time series prediction and are considered deep in *time*, as they do not have multiple *different* layers of representation but essentially, have a layer for every data-point in a time-series. In order to *train* neural network architectures and increase their accuracy, an error or *learning signal* must be back-propagated from the output to the input of the network. The *vanishing gradient* problem refers to previous algorithms where this learning signal disappeared in deeper architectures so it was not possible to learn multiple layers of features.

When the Deep Belief Network was first proposed and implemented, it achieved the state of the art results [60] on the MNIST (Mini National Institute of Standards and Technology) hand-written digit dataset classification task [80]. The MNIST classification task is a benchmark data set in machine learning, which is used to test new and existing algorithms, comparing their accuracy in determining what hand-written digit is contained in a picture. Since then, improvements were made and other deep algorithms such as the Deep Boltzmann Machine [115] continued to improve upon the state of the art. Today deep learning algorithms continue to break records on many other benchmark datasets [38], but computer vision, speech recognition and natural language processing remain the subject of most deep learning applications. However, the issues of complexity and management of deep learning experiments can be considered as the main obstacle to their more widespread use.

Neural networks and deep neural networks have been applied with great success to feature learning [77], [61], anomaly detection [48] and sequential prediction [78]. As neural network layers can be combined, these problems can be approached with one architectural algorithmic solution, but deep neural networks are notoriously hard to build, configure, optimise and interpret.

1.3 Practical Machine Learning Problems

As part of this research, we regularly worked with domain experts who compiled data sets, but needed assistance in performing certain analyses not easily possible with off the shelf software. Their goals were generally to gain insights and perform more in depth or more accurate analyses. In this dissertation, we use two of these data sets as part of our evaluation.

The first data set was The Maastricht Ageing Study [130], from the health and ageing domain, which arose from a collaboration on the INnovative Mid-life INtervention for Dementia Deterrence Project (In-Mindd) [35] project. The second, was from a collaboration with colleagues from the field of sport and human performance. The latter dataset consisted of sensor data gathered from Gaelic Football players during competitive games. To ensure a more rigorous evaluation of our approach, we introduced a third dataset, which exhibited similar data properties to the first two but consisted of data gathered from a completely different source and field. This dataset was the Johann Sebastian Bach Chorales MIDI data set [7]. The introduction of a multimedia music dataset ensures our methods are generic and not limited to a particular application domain. Furthermore, it is a dataset that is freely available on line which would allow practitioners to test and recreate our methods.

The Maastricht Ageing Study (MAAS) was a longitudinal cohort study, in which biometric and survey data relating to cognitive function and health was collected on ageing individuals in the Netherlands at fixed, 3 year intervals over the course of a 12 year period [130]. This study resulted in the MAAS data set, which consists of 3441 unique records and 1835 unique features spread throughout 86 'tests' or study subsections. The domain experts were interested in a subset of this data which consisted of *modifiable* dementia risk and protective factors, and how these related to the probability of a person surviving without dementia. This approach sought to determine the modifiable factors which influenced dementia and could enable a person to adjust their lifestyle accordingly to lower the risk that they would develop dementia. A additional focus was placed on how these variables *interact*. The discovery of multivariate interactions could provide not only a means of more accurate prediction, but also provide insights into the *mechanism* in which different physical attributes influence each other and the outcome of dementia.

The Gaelic Football Sensor data set consists of information gathered over a series of 17 competitive games of Gaelic Football. Gaelic Football is a native Irish

sport overseen by the Gaelic Athletic Association [4] which, like many team sports involves repeated, short duration, high intensity bouts of anaerobic exercise interspersed with sustained light to moderate aerobic activity. At each of 17 competitive games, 10 of 15 players in a team were fitted with sensor devices to record heart rate, speed, distance, latitude, longitude and acceleration. Measures were recorded multiple times in each second of the game. Other data, such as the teams opponent, warmup and match start and end times were also recorded. The resulting dataset contained in excess of 200 million values and 33 features. Due to the nature of contact sport, the devices incur a number of blows during each game, introducing many potential anomalies. In this case therefore, the analysis motivates anomaly detection in the first instance. Furthermore, the domain expert wanted to predict when players would reach exhaustion in a game. Predicting future heart rates and how player bio markers interact in these predictions is the first step for this task. These predictions could then be used to substitute players before they reached their peak and gain some insight into how various physical elements influenced performance, which could then form the basis for improved game and training strategies.
J.S. Bach Chorales data set consists of data on 60 Bach chorales (hymns), coded as 5665 MIDI events, which relate directly to the number of instances in the dataset. Each event is labelled as 1 of a possible 101 chords and has 14 attributes to describe the event along with the chorale ID, event number (a simple index) and a further 14 attributes to describe each event. 12 of these attributes are binary relating to the presence or absence of each of the simple 12 chromatic pitches possible in a chord (it does not contain octave information), a string representing the letter name of the bass note of the chord and finally, an attribute relating to the meter of the chord (for how long the chord is sustained). This is a prepared dataset with which we have domain expertise. Thus, it enabled a further evaluation with the development, testing and interpreting of result data as experiments progressed.

1.3.1 Deep Learning Application Goals

In examining each of these data sets, although originating from heterogeneous application domains, they exhibit similar data properties and problems. Therefore, in broad terms, three functions are desirable when building a deep learning solution for data mining: anomaly detection, (sequential) prediction and learning feature representations. We provide a brief description of each before defining our problem statement in this next section.

- **Anomaly Detection** is often a first step in data mining applications. *Anomalies* are generally defined as unusual events which occur within a dataset, where a subset of these events are *outliers*. Outliers are occurrences that make either no physical sense, or appear so extreme they are considered probabilistically infeasible. The detection of anomalies was an important requirement for the Gaelic Football dataset, but was also a step in the analysis of the MAAS data. Human error in data capture is often present in these studies, but aside from human error, anomalies could contain valuable information. For example, one individual in the MAAS dataset may develop dementia while adhering to the profile of a *not-at-risk* lifestyle. Our anomaly detection techniques are evaluated using the J.S. Bach dataset to identify unusual chords in the context of a chorale. This anomaly detection experiment is *completely* unsupervised as we do not know those records are anomalous and those that conform.

- **Sequential Prediction** was a required mining process in each of the three data sets, as time is a relevant factor in informing predictions. In shallow architectures, most time-series analyses work on the premise that a current time point is conditioned on the immediately preceding point or can only incorporate a short, specific window of time. Predicting a time-step based on all previous time steps is more desirable [65] as short time analyses cannot fully capture temporal dependencies in high-dimensional sequences.

- **Learning Feature Representations** is also a requirement common to researchers for all data sets. Either the data is *high-dimensional* meaning too many features are present, or there is a requirement to learn how the input features *interact* to provide greater insight. Learning a feature representation can overcome both of these problems. Too many features, especially with a low instance of records can cause a model to over-fit. Over-fitting is when a model classifies (or clusters, etc.) the data upon which it is built very well, but does not generalise or perform well on unseen data. Many traditional feature selection methods are sub-optimal [3], [13], [65] and dimensionality reduction techniques such as Principal Component Analysis remove some of the variance in the data. Furthermore, features are often hand-crafted - for example, a quotient of two features - for greater predictive power, which is an unsustainable and non-scalable practice. Discovering variable interactions is a very difficult problem due to the high number of possible permutations involved or the need to identify an apriori hypothesis in relation to latent classes. Furthermore, it is often necessary to combine these classes with a separate modelling technique when using traditional mining solutions. Neural networks provide a means to learn a feature representation [13], [61], but in practice, on non-image data, the features are notoriously difficult to interpret. Finally, testing whether these interactions are accurate in clinical research, remains a difficult task. Devising a method to accurately model and test these interactions could lead to far greater risk prediction in relation to disease.

1.4 Problem Statement and Hypothesis

Building a deep learning experiment is a complex and time consuming process. If we are to invest the time and computational resources to this approach to machine learning, it is imperative that we can measure a tangible benefit in terms of predictive capabilities. Deep learning experiments are big experiments as there are a very high number of data transformations and experiments tend to run for many hours. In reality, there are a high number of parameter settings and combinations of parameter values that can influence the quality of the learning algorithm. Furthermore, researchers may wait up to a number of days to discover that settings were incorrect or that the quality of learning had tailed off during the experiment. What is required for researchers in a deep learning setup is the ability to build their own deep learning configurations and thus, test using different deep learners for different problems across multiple domains. Additionally, this requires technology supports to be able to freeze experiments, retrace to previous decisions, analyse across result sets etc. Finally, researchers should be able to share their experiences (settings, results) in a well understood fashion for a collective better understanding of how deep learning machines work.

1.4.1 Hypothesis

The machine learning experiment has been described earlier in this dissertation as a continuous process of Configure, Build, Optimise and Interpret (results), and as presented in our problem statement, a single experiment can run for many hours due to the complexity of a Deep Learning machine. In order to adequately support and manage deep learning experiments, there is a need for a new level of supporting methodologies and tools.

The hypothesis we present is that in order to enable new levels of deep learning it requires: a novel approach to configuring and running deep learning experiments; a semantically powerful data model representation of all elements of the deep learning machine; and the development of a toolkit which is based on the data model approach to deliver functions that manage the experiment and better analyse the results.

By new levels of deep learning, we refer to: a greater degree of experiment automation; increased possibilities for the analysis and interpretation of experiments; and portable, detailed experiment data capture to aid experiment reproduction and reuse.

1.4.2 Research Questions and Contribution

There are a number of research questions that must be answered as part of this research and dissertation.

1. Can we design a *configurable* deep learner system which integrates the optimisation of both model parameters and hyper-parameters and provides a platform for the implementation and extension of new methods? There is little evidence in current state of the art where deep learning frameworks pro-

vide full, integrated optimisation. For example, Google TensorFlow [1] does not provide for integrated hyper-parameter optimisation within its software model. Nvidia Digits [100] does provision for a hyper-parameter sweep, but it is for some but not all hyper-parameters and does not provide a fully integrated configuration and parameter optimisation approach.

2. Is it possible to capture all aspects of a deep learning experiment to enable a robust interpretation of results at *any iteration* of the optimisation process? The benefit of this level of detail in the analysis phase enables the development of a set of operators that manipulate both setup parameters and results in the optimisation of deep learning experiments. It can also be used to exchange the output from deep learning results across the research community and thus, allow deep learning researchers to quickly build upon the work of previous efforts.

3. Can a set of analytical functions be developed that support the key elements of result interpretation: selection of the best performing model and selection of the best hyper-parameter configuration? Further analytical functions should include the narrowing of bounds for hyper-parameter search and interpreting how the *abstract* learnt features relate to the *original* dataset features.

The contribution provided in this dissertation is illustrated through the outputs achieved by answering the above 3 key research questions. We will present a novel methodology that integrates both data mining and deep learning. Its benefit to data mining practitioners is to enable them to incorporate deep learning into their data mining operations. Its benefit for deep learning researchers, is the suggestion of a standard process for experimentation for increased levels of reproducibility and understanding. This methodology has at its core, a data driven approach to capture all aspects of the deep learning experiment. This deep learning data model provides the foundation for both the deep learning software and the toolkit to support experiment configuration, management and interpretation. Our extensive evaluation shows that the toolkit developed in this research successfully solve the problems presented, providing increased levels of experiment automation and enabling new insights into, and interpretations of deep learning representations. In summary, this research provides key methods and tools to enable deep learning interpretations and experiment automation configurations that were not previously possible.

1.5 Summary and Dissertation Structure

As researchers, in order to satisfy the analytical needs outlined in the previous section, it is necessary to develop solutions which utilise appropriate techniques in the current state of the art. In machine learning, this is the area of deep learning. Deep learning is a complicated form of analysis. Not only do model parameters have to be optimised but the optimisation of hyper-parameters is also a fundamental requirement. These hyper-parameters are inputs into the model parameter training process. This could be the number of iterations that is performed in training or the number of layers in the deep learning architecture. Furthermore, although deep learning frameworks exist, implementing new algorithms is cumbersome and lacks a common methodology. Finally, no framework exists to analyse the results *and* parameters of experiments, and use an interpretation of intermediary results to feed into a new level of experiments. In this chapter, we highlighted the major requirements of the deep learning experiment: Build, Configure, Optimise and Interpret. Furthermore, we motivated our research by stating that the system and toolkit to achieve all these requirements of the deep learning experiment does not currently exist. The goal of this research is to deliver an overall framework by providing a configurable deep learner; a data driven methodology with fully specified deep learning data model; a toolkit whose functions are based upon and specified by the deep learning data model; and which interfaces with the configurable deep learner to deliver powerful new functionality for deep learning researchers. We provide a comprehensive evaluation using 3 diverse datasets whereby we tackle 3 major machine learning problems: anomaly detection, feature selection and learning, and sequential data mining. Thus, we consider our Toolkit to be successfully evaluated if we can demonstrate the achievement of each of the following major requirements:

1. **Application.** One or more of the deep learning application goals from §1.2 are

achieved on each of the 3 heterogeneous datasets presented. This demonstrates generic domain application and the possibilities to achieve multiple goals with a single experiment if experiment data is persisted.

2. **Configuration.** An arbitrary number of hyper-parameter configurations can be generated automatically and generically and the relevant number of deep learning algorithm instances, agnostic of the algorithm employed are constructed based on these parameters. This demonstrates the generic, simple and automatic configuration of experiments.

3. **Optimisation.** Model parameters of each deep learning algorithm instance are optimised and the best performing hyper-parameter configuration is returned. This demonstrates the integration of hyper- and model parameter optimisation.

4. **Interpretation.** Optimal model parameters can be explored to discern what the network is likely to have learnt; top hyper-parameter distributions can be explored to determine why they are optimal; and both parameters and performance at any part of the training process can be examined at a reasonable fidelity. This demonstrates increased levels of interpretability, reproducibility and empirical rigour.

Our approach to these research goals has led to a dissertation structure as follows. In Chapter 2, we examine the state of the art in deep learning and how the 3 major machine learning problems are addressed. In Chapter 3 and as an introduction to our own approach and system, we describe the basic concepts in neural networks and the pre-existing shallow and deep algorithms used throughout this dissertation, along with the relevant terminology, notation and mathematical functions. In Chapter 4, we present the methodology which underpins our data mining and deep learning experiment approach as well as the architecture of the system we have built to manage deep learning experiments. In Chapter 5, we provide a detailed description of our Configurable Deep Network design for building deep neural architectures and optimisation framework which incorporates this design. In Chapter 6, we present our conceptual data model for deep learning experiments and the Deep NoSQL Toolkit which was designed and built using the data model as a blueprint. The toolkit provides storage of results at any point in a deep learning experiment, interpretation and analysis of results, and sharing of experiment configuration and results with other deep learning researchers. In Chapter 7, we present our evaluation. Finally, in Chapter 8 we summarise the research presented here and outline possibilities for future work.

Chapter 2

Related Research

In the previous chapter, we introduced the fields of data mining and machine learning and briefly explored a new, specialised field of machine learning called deep learning. We then highlighted problems in its practical application, which motivates the research presented in this dissertation. Therefore, in §2.1 we explore practical applications of machine and deep learning, along with current approaches to hyper-parameter optimisation, a topic which is not well covered in deep learning research. Subsequently in §2.2, we examine data driven approaches to machine learning which facilitate sharing, reproducibility and standard description of learning experiments. In §2.3, we introduce currently available deep learning frameworks by exploring the most popular approaches. Finally, in §2.4, we summarise the chapter by highlighting those elements of deep learning research which remain open for continued research.

2.1 Applied Learning: Contrasting Shallow and Deep

This section explores the practical, applied aspects of machine and deep learning where topics examined relate directly to the requirements outlined in §1.3.1. Therefore, anomaly detection forms the focus of §2.1.1, feature learning and dimensionality reduction is dealt with in §2.1.2 and sequential prediction is covered in §2.1.3. For each application we examine traditional shallow approaches before exploring the deep learning equivalent, demonstrating the improvements offered by deep learning, either in terms of accuracy, where deep learning holds the state of the art on a benchmark in that area, or offers improvements above and beyond shallow learning such as greatly increased automation. In §2.1.4, we introduce state of the art methods of hyper-parameter optimisation, a process integral to any practical deep learning application.

2.1.1 Anomaly Detection

Perhaps one of the most well known (shallow) clustering algorithms also used for anomaly detection is DBSCAN [44]. This algorithm has undergone a number of extensions - such as adding a hierarchical component [26] or a spatio-temporal element [19] - since its inception in [44]. As it is a clustering algorithm DBSCAN works on unsupervised tasks. Local Outlier Factor (LOF) [25] is another unsupervised, clustering based, anomaly detection technique which shares a similar theoretical basis to DBSCAN.

Both DBSCAN and LOF algorithms are density based. Density based algorithm operate on a geometric concept for clustering, which assumes that non-anomalous data points will be clustered together in a similar region of space. Therefore, a distance metric is required for both. DBSCAN requires input parameters *minPts* and ϵ as hyper-parameters, where *minPts* is the minimum number of data points within a distance of ϵ from the data point being queried. If the current data point has *minPts* within the ϵ distance, then the point is added to a cluster as well as all those points within ϵ distance. DBSCAN, unlike k-means does not require the selection of the number of clusters beforehand but it does require the setting of the *minPts* and ϵ parameters, which can be difficult if the data is not well understood. We enable the optimisation of similar parameters by storing the results and comparing different settings performance on an objective function. DBSCAN can use any distance metric which is positive but perhaps the most used is Euclidean distance which suffers from the *curse of dimensionality* [10] in contrast to deep neural networks which automatically reduce dimensions.

In contrast to DBSCAN, Local Outlier Factor (LOF) examines each data point and determines the *degree* that the point is anomalous. It examines the point to see how isolated it is in relation to other points in the data set, or how many other data points are close in its locality. Unlike DBSCAN, a major advantage of LOF is that it does not approach anomaly detection in a binary fashion. Instead, it assigns a data point as anomalous or non-anomalous and also assigns a measure of how anomalous each point is. However, the value that is assigned, as with weights in a neural network, can be difficult to interpret. A further advantage LOF has over DBSCAN is that it can handle clusters of different densities, whereas DBSCAN requires the setting of the density parameter ϵ before execution which means the density of each cluster examined with DBSCAN is required to be homogeneous. The major disadvantage with LOF is interpreting the score given and deciding where the outlier threshold is, although recent efforts have been made to rectify this [74]. LOF also requires the setting of a hyper-parameter, that is a value for k - to determine the number of nearest neighbours to compare a data point. Although efforts have been made to lessen its effect [73], it requires advance setting and therefore, needs a hyper-parameter optimisation technique. We include hyper-parameter optimisation within our deep learning system.

In [81], the authors present a novel approach for anomaly detection incorporating both density and grid-based clustering algorithms. Their primary focus is high dimensional data and they test their algorithm on the KDD Cup 1999 network dataset [83]. The approach taken was to optimise the pMafia algorithm, using a Frequency-Pattern tree in an intermediate step in order to improve the detection rate. In their evaluation, it was shown that the improvement in detection rate had a negative side effect in generating a higher number of false positives. By their own admission, the algorithm works best for datasets with certain characteristics. This means that if there is an entire window of anomalous data, this may affect the performance of the detection method.

The authors of [122] developed the Robust Support Vector Machine, to accomplish a similar but slightly different goal to anomaly detection. They aim to demonstrate that Robust SVMs can still correctly identify images when unknown outliers exist in the data. This algorithm is an improvement on the standard support vector machine (SVM) algorithm as the incorporation of the averaging technique makes the decision function less susceptible to outliers through an adaptive margin and thus, avoids overfitting the learning algorithm. We also classify with data that contains anomalous points, but the identification of the anomalous data itself is important, as anomalies can sometimes contain important information in themselves.

In [108], the authors propose an extension to k-means algorithm called x-means to identify outliers in Gaussian datasets. This work is novel as it did not require the user setting of the k hyper-parameter. The algorithm performed exceptionally well with regard to identifying the exact number of clusters and compared well against the k-means algorithm. It is likely that x-means is faster when compared to deep learning, but it does not learn a hierarchical feature representation that can be then used to determine variable interactions and serve as input to classification procedures. For this reason, deep learning performs better than shallow learning approaches.

In [119], the authors propose a novel Principal Components Classifier (PCC) to detect anomalies on the KDD Cup 1999 network dataset. The dataset relates to network access data and it contains anomalies in the form of network intrusions. Therefore anomaly detection in this case is network intrusion identification. The PCC produced a false positive rate of only 1%, showing their approach was robust to false positives. Our approach also employs an energy paradigm but in a probabilistic context. Unfortunately, although they were able to keep their false hit rate static, all the other metrics degraded significantly in terms of quality with relation to false positives. In contrast to the KDD cup data set which contained completely clean, non-anomalous data training data, we worked with real-world datasets, containing unidentified anomalous examples. All newly created datasets, perhaps generated from online data or from sensor networks, will be of the same unclassified nature, which is addressed in our research.

The use of neural networks for anomaly detection has been ongoing since 1999 [50]. With the advent of deep learning, recent projects have returned to using both shallow [48], [87] and deep neural networks [116], [140], [138] for this purpose. There has been more recent interest in using neural networks for anomaly detection and particularly, the use of the free energy measure of a *shallow* Restricted Boltzmann Machine to identify these anomalies [48], [87]. Deep neural networks have not been exploited for unsupervised anomaly detection before 2016 [140], although there has been semi-supervised Deep Belief Networks [138] and fully supervised approaches such as [116].

In [48], the authors use the free energy of a *Discriminative* or *supervised* Restricted Boltzmann Machine to investigate whether data-points are anomalous or non anomalous. Here, they achieve good results, noting a difference in the free energy between anomalous and non anomalous data when they train and test on real network traffic data. However, when they use the KDD Cup 1999 network traffic dataset [83] for training and real data for testing, performance degrades significantly. Therefore, they draw the conclusion that simulated traffic data is *not* a good benchmark for intrusion detection tasks. The application of their work is unlike ours as our data is fully unsupervised, meaning we have no apriori knowledge as to whether a sample is anomalous or not. Furthermore, we take a multi-layered approach and use the free-energy of the top layer of a *deep* network to determine whether a sample is anomalous or not.

In [87], the authors also use the free energy of a *shallow* RBM to detect anomalies but also use a Spatio-Temporal Pattern Network to extract key features from time series data in order to identify features upon which to train the RBM. Their method shows that the probability distribution of a normal subsequence sample is noticeably different to an anomalous subsequence sample when the Kullback-Lieber divergence is measured between both samples. Unlike our research, this adopts a shallow machine learning approach but *is* a fully unsupervised example of anomaly detection with RBMs. This proves to be an interesting and effective approach for them, but as deep networks can be used as feature extractors in high-dimensional time series [24], we have adopted this approach in our research.

If we look specifically at applications of *deep* learning to anomaly detection, there has been little research [138], [140], [116] and only one study has investigated completely unsupervised energy based models [140]. In [116], the authors investigate using a hybrid Deep Belief Network and Support Vector Machine approach. The Deep Belief Network is used as a generic feature extractor and these high-level features are fed into a Support Vector Machine classification layer. They discovered that the combination of the DBN and SVM performed better than either the DBN or SVM used in isolation for classification. They also compare the DBN to PCA, Chi-Square and Gain Ratio feature selection methods and determined the DBN to outperform these. Our work differs in that we are investigating the free enrgy of a top-level RBM in a *completely unsupervised* DBN to determine anomalies in contrast to their supervised classification task.

The approach taken by the authors in [138] is semi-supervised. They pre-train a stack of RBMs before unrolling these stacked RBMs and further adjusting the parameters of all layers together, as unsupervised auto-encoders and performing a final tuning step where the parameters are adjusted with respect to class labels. This makes adjustments to make the final fine tuning step more sensitve to minority classes. Their hypothesis was that as the Deep Belief Network learns an internal configuration of what data is most probable, if it then receives an input vector and reconstructs it very closely, the input is not anomalous but if the reconstruction is very different, then the sample is anomalous. This differs from our work in that their anomaly detection element is semi-supervised, containing some class labels. They also use reconstruction error instead of the network's energy measure to identify anomalous samples.

In [140], the authors use both measures of energy and reconstruction error together to detect anomalies. They also provide different generalised Energy Based Models for static, spatial (image) and sequence (audio) in contrast to the anomaly detection aspect of our work which does not account for spatial and sequence information. In contrast to our method of directly optimising the free energy, they use score matching instead of Maximum Likelihood Estimation training which results in a simplified training process and easier generalisation to the spatial and sequence data. The results of [140], at least match and more often surpass, state of the art shallow anomaly detection methods on several benchmark datasets. Furthermore, it seems to be the first investigation of its kind and the only research to investigate deep, unsupervised, energy based models for anomaly detection. However, in [140] they do not provide a means to investigate and gain insight into what features are learned in order to detect these anomalies. Furthermore, there are no details as to how hyper-parameters were selected for their final results.

2.1.2 Representation Learning

We consider feature selection, dimensionality reduction and discovering variable interactions or latent classes as related under the subject of representation learning. The authors in [3] review the suite of traditional feature selection methodologies and their efficacy with various shallow learning algorithms. They formalise the process of feature selection into a two step operation which first evaluates each individual feature's predictive power, and subsequently applies a cutting criterion - a methodology for cutting all but those features evaluated to have the best descriptive power to an outcome. The number of features cut depends on the parameters and the cutting criterion used.

Feature evaluation methods used included information gain (the difference between the entropy of the class and the entropy of the class when conditioned on the feature being evaluated), gain ratio (the ratio between the information gain and entropy of a feature), gini index (the probability of two instances randomly chosen having a different class), relief-f (ranks and weights an instance based on the features of that instance and its Manhattan (L1) distance from the next closest feature) and relevance (measure for how relevant features are to the needs of a user). Once the relevance to the classification of each feature in the dataset was analysed, various cutting criteria were applied. The filtering methods used were: fixed number (select a fixed number of features), fraction (select a fraction of the total features), threshold (choose features whose evaluation is over a certain threshold), threshold given as a fraction (features that are over a certain threshold, where this threshold is a fraction of the range of the evaluation function), difference (selects features from the greatest evaluated until the difference between that and the subsequent feature is above a certain threshold) and slope (select features until the slope to the next feature is over a certain threshold). It was found that when a certain threshold was exceeded, greater reductions lead to the loss of relevant features, which lead to poor prediction accuracy. They found that it was not possible to determine which method performed best overall as each algorithm tended to work best with a particular feature selection method. From this project, it can also be seen that much time was spent analysing the best feature selection method and cutting criterion to use before analysing the data.

In summary, [3] motivates the need for the automation or an artificially “intelligent” method to perform feature evaluation and selection. Deep learning provides this automation. The authors could not learn a representation of the data but instead, a lot of effort is consumed by focusing on features to discard and the methods with which to do this. Deep learning inherently learns a representation of the data and thus, reduces considerable manual effort while learning more relevant features and not discarding data. It learns this representation while training a classifier, so only one algorithm needs to be used rather than a whole suite.

Feature selection is generally treated as a separate step to learning in most mining applications and this again can be seen in the efforts of [46], where a new confidence metric for medical data classifications is proposed. To select relevant features, they applied a single variable classifier to only those instances with no missing data and measured their performance using the area under the receiver operation characteristic curve (AUC) measure. The final score calculated the average AUC over multiple classifiers and the features were ranked according to this score. There were originally nine features and the top four ranked were selected to train upon. This number was chosen as the accuracy of the models learned improved up until a sixth feature was discarded. Subsequently, a number of multi-variate experiments were performed using different feature combinations: for example, all nine, top four, etc. The selection process described is manually intensive, building multiple models until performance degrades. Our deep learning approach together with the accompanying toolkit will be shown to provide far greater efficiency,

By *not* using deep learning methodologies in [46], an approach which would not discard any features but learn the most relevant abstraction, a significant manual overhead is incurred. On the other hand, completely discarding features can also be detrimental to the final model. Although a feature might appear irrelevant to an outcome on its own, when this feature is combined with one or more other features, the predictive power of this feature combination could be far greater than the predictive power of individual features. This manual crafting of features requires much time, effort and knowledge of the subject area and is unsustainable in large data sets. Crafting of features is also not possible for the feature selection methods here. Deep algorithms automatically learn a representation of the data, automating this process, as will be shown later in this dissertation.

In a similar fashion, Principal Component Analysis (PCA) [69], which is possibly the most widely used feature reduction technique, discards certain data. The aim of PCA is to transform the data via a *linear* orthogonal transformation so the components into which the data are transformed are linearly uncorrelated. These components can then be used as the features for a particular analysis. After transformation, the first principal component contains greatest variance, the second the next greatest variance and each component continues in this fashion with decreasing variance. There are an equal number of components to original features but normally only the first two or three *principal components* are selected as input into a separate analytical process, discarding the others. Therefore, some variance and as a result, information in the data, is lost before analysis. In contrast to the non-linear analysis possible with neural networks PCA is a *linear* transformation, which is not as expressive as its non-linear counterpart. Neural networks learn how variables *interact* and combine them into more abstract features based on *all* of the data instead of discarding it. However, the downside is that the weights and how they combine are very difficult to interpret and are often presented as 'black box' solutions [105], although some methods do exist to investigate what shallow networks learn [105], [49], [106], [52], [142]. As we use deep learning algorithms, our approach benefits from the more powerful aspects of multiple levels of non-linear transformations, but this also increases the interpretation complexity.

As one of our unsupervised datasets is taken from a longitudinal study on dementia, we also examined research [97] which sought to model *latent classes* relating to behaviour and its association with dementia analysis. This would be typical of a shallow learning approach to extracting latent classes or *abstract features* from data. The authors sought to identify distinct behavioural patterns across six domains: church-attendance; smoking; alcohol use; social interaction; and physical exercise.

The methodology used is Latent Class Analysis. Latent Class Analysis can only measure latent classes from dichotomous (binary categorical) variables and therefore, continuous values are not possible inputs. Furthermore, LCA requires a number of further steps after the latent class identification, exponentially increasing the number of peripheral models needed as the features increase. First LCA is applied and a number of possible latent class numbers (similar to our number of nodes) are tested, then multinomial regression is applied to assign a sample to the relevant class (behavioural sub-category), before finally running another regression model for each identified class to evaluate survival probabilities. Neural networks essentially incorporate all steps into a single end-to-end solution, where latent variables are identified, samples activate relevant latent variables (hidden nodes) and classification probabilities are identified all during the course of training. In our approach, a single algorithm replaces the multiple steps required in this approach, for dimensionality reduction, latent class analysis and classification. In addition, neural networks can model *continuous* data and continuous interactions between sub-categories in a *non-linear* paradigm in contrast to the linear LCA. The result is that more expressive data is captured using our system.

The power of deep algorithms for feature learning and unlabelled class detection can be seen by a relatively recent application of the Google research team [77]. They argued that most applications up to that point had only learnt lower level features with approaches such as k-means or shallow RBMs. They train a deep sparse autoencoder on a large dataset of *completely unlabelled* images, consisting of randomly sampled 200x200 frames from 10 million YouTube videos. They argue that time is a major prohibitive factor in training large deep neural networks, demonstrating their solution took 3 days on 1,000 parallelised computers consisting of 16,000 cores. As our datasets are not as large nor our networks as complex - in contrast to their 1 billion trainable parameters - we should not suffer from this issue of training time as much, but it is still a concern of these networks. An autoencoder is a deep architecture where an algorithm is tasked with learning the identity function of the input. That is, based on certain parameters the autoencoder tries to learn function to approximately reconstruct the input.

In [77], high level features relating to a human face detector are successfully learned, and through experiment the possibility of learning those which relate to a ‘cat face’ and a human body are also demonstrated. Here, high level features are easily interpreted, as they can simply be visualised and form a simplified or generic picture of the object they represent. It is much more difficult to do this with non-image data. They successfully test their method on the Labelled Faces in the Wild [64] and the ImageNet [40] datasets. They demonstrated a **70%** relative improvement on the state of the art, showing conclusively that abstract and relevant features can be learned not only from labelled but also *unlabelled* data with deep networks. In [77], they show how successful deep learning can be in application but do not give any insight into why the particular configuration of their deep architecture was chosen, nor are any of the other hyper-parameters presented. As part of our system, we include a means to interpret these learned features for non image-based datasets which shows a wider applicability of this approach.

Regularisation is a method of avoiding over-fitting a model to training data. A model over-fits if it correctly characterises the training data but cannot generalise well to unseen instances. Popular means of doing this are L1 and L2 regularisation, which is adding a mathematical term to a learning hypothesis cost function in order to penalise large parameter values and give the model greater generalisation power [137]. In the case of the L1 regularisation, this term is based on the Manhattan norm and in the L2 regularisation, the term is based on the Euclidean norm. Recent regularisation developments in Deep Learning are dropout [62], and dropconnect [136].

Dropout [61] randomly zeros a subset of activations within each layer, preventing the co-adaptation of features - where a feature detector is only helpful in the context of several other feature detectors. These results held the benchmark for accuracy in the CIFAR-10 tiny images dataset and the Mixed National Institute of Standards and Technology (MNIST) [80] database [75], but has been surpassed by an extension of this method called dropconnect [136]. Instead of zeroing or dropping a random subset of the activation functions, they instead drop a random subset of the *weights* in each layer. This form of regularisation provides a means of improving the accuracy of deep networks. We have implemented Dropout within our framework as this method had not been applied for anomaly detection with deep neural networks. Finally, in the medical context, DBNs have been used for medical text classification [139], as well as to aid in medical decision making with electronic health records [82]. Neither [139] or [82] provide a methodology on how to choose the initial hyper-parameter configuration of a deep learning architecture. Furthermore, they use third party implementations of a DBN which do not allow for the extension with further algorithms, activation functions or hyper-parameter configurations. In [139], the authors utilise a single hidden layer in their DBN, which arguably is not a deep architecture, although they do employ a unsupervised pre-training step. None of these approaches have the levels of configuration or control we describe for our system in Chapter 5.

2.1.3 Sequential Data Mining

The task of predicting disease spread through the mining of micro-blogs on a social network (Twitter) is tackled by Sadilek et al. [114]. They present a method of mining noisy, incomplete and temporal geo-located twitter data. An SVM is first trained to label a corpora of over 200 million tweets to evaluate what messages identify a person as 'sick' and what identify a person as 'normal' or 'other'. Then, using a conditional random field (CRF), they model the temporal aspect of the data as well as the geographic-location to predict whether a person will get sick based on time and co-location with sick individuals. They use Viterbi decoding and the forwards-backwards algorithm to infer missing data, an inference algorithm for the CRF that computes the posterior marginals of all hidden state variables where there are a sequence of observations. An approach such as this would be typical of shallow mining. While they demonstrate a degree of success, they must manually select the features to use and are limited with CRF, as performance degrades the further they predict into the future. Deep learning is less vulnerable to these issues. Recurrent Neural Net variants such as Long Short Term Memory (LSTM) as they have been shown to learn what they need to hold in-memory from the past for an arbitrarily long period of time [63], [53]. The problems encountered in longitudinal trials involve modelling over a much longer time period, as well as the prediction of much further into the future than [114]. Therefore, this type of application motivates the use of a deep network or at least, a Recurrent Neural Network.

Survival analysis is the term used for the most popular type of sequential health analysis, that is, for the prediction of patient survival rates in datasets like MAAS, which we introduced in §1.3. It is a prediction of, given the current time point, what is the probability that a participant would develop a disease at the next time-point and thus, is a form of supervised sequential prediction. Artificial neural networks (ANNs) exhibit great potential, but have not been widely applied in survival analyses. Deep neural networks exhibit even greater potential, but have been utilised to a significantly lower degree in this context. Shallow neural networks have been shown to be at least on par with logistic models developed specifically for survival analysis [86] but more often, can out-perform traditional logistic methods, even in the health context [103], [118]. There have been reviews of the health benefit to be found from ANNs in medical intervention where the authors show that although ANNs are not in widespread use for health applications, they have had a significant clinical impact when used. This was notably in areas such as cervical cytology and the early detection of acute myocardial infarction (heart attack) [85].

The majority of applications of ANNs in survival analysis has tried to predict patient mortality after surgery and sometimes with great success [2], [118], [86]. There has been very little work applying ANNs to dementia analysis [5], [89], [90] and only one of these deals with survival analysis or sequential health analytics. Furthermore, all previously mentioned studies compare ANN methods to traditional approaches like Cox's regression or other machine learning algorithms and unlike our research, do not test deep neural networks for the purpose. Furthermore, none of the studies provide an analysis of the features learned in hidden layers. This is highly relevant to clinical and health analytics as the interactions between input factors could generate knowledge on how a disease develops or how an athletes biomarker affects performance. There has been research into variable interactions in survival analysis using ANNs [37] but once again this, was specific to breast cancer surgery and not widely applicable eg. dementia survival or performance analytics in sport.

Exploring the work of [89] further, they compare two Neural Nets (MultiLayer Perceptron and radial basis function neural nets) and find they are outperformed by Support Vector Machines (highest), Random Forests and Linear Discriminant Analyses. As ANNs are complex to train, there are several possible reasons for these findings, some of which the authors of [89] mention. Primarily, ANNs are highly sensitive to the tuning (hyper) parameters used to initialise and inform the training procedure and can vastly affect the quality of models learned. In [89], the only hyper-parameters optimised are the number of hidden layer nodes, where all other settings were "commonly used in data mining applications" and not chosen specific to the data in question. Furthermore, [89] uses a grid-search optimisation procedure, in contrast to random search for *all* hyper-parameters, a methodology shown to outperform the traditional grid-search [16] technique. Finally, their data-split methodology is non-optimal for the purposes of choosing hyper-parameters. Their strategy essentially splits the data in two - a portion for training and evaluating hyper-parameters and a held-out cross-validation set for evaluating the accuracy of the overall classifier. When the data on which the model is trained is also used to evaluate hyper-parameter performance, this has the effect of over-fitting the hyper-parameters to the training data. Instead to properly evaluate performance, *at least* a three-way split is advised, one portion to build the model (training), one to evaluate the validity of the hyper-parameters (validation) and a final held-out portion to evaluate final performance on completely unseen data (test).

Finally, a review [65] of the current state of music information retrieval, proposes that deep learning would make it possible to select the relevant features or dimensions in a high-dimensional, sequential dataset. This was shown to be the case in [24] where they used [65] to motivate the use of deep architectures on music information retrieval. In [65], they successfully improved upon the previous state of the art for accuracy in many music information retrieval and learning datasets. They proposed a Recurrent Neural Network Restricted Boltzmann Machine (RNN-RBM) to model the temporal dependencies in high-dimensional sequences, as applied to polyphonic music transcription. The RNN is a type of neural network where the connections between units form a directed cycle. This allows the model to create an internal state which models the temporal and dynamic data. The RNN is then fed to an RBM and together, the conditional distribution of the next time step is modelled given all previous time-steps. Traditional methods use only the previous time-step to infer the next step. The RNN-RBM is considered deep because the output of the RNN is the training input of the RBM. An RNN unfolded in time is also equivalent to a deep architecture. They use the output of the RNN as input to a *conditional* RBM to predict the next time step given the previous time-step. To demonstrate the wide applicability of our approach, we also include a music dataset in our evaluation but unlike [65], we use a deep stack of RBMs and examine the feature representations learned.

2.1.4 Hyper-Parameter Selection

The authors of [88] assert that gradient free and model-based optimisation is the current gold standard in hyper-parameter selection and this is shown to be the case for deep networks in recent literature [18], [121], [16]. For this reason, we focused on leveraging a subset of these gradient free techniques.

In [16], the authors propose a random search method to find the best hyper-parameter configuration for deep architectures. They compare their results to previous work [76], which used a multi-resolution grid-search coupled with a manual optimisation intervention element. In [16], they also carry out a series of simulation experiments where random search is compared to both grid-search and low discrepancy sequential methods. Their main contribution is a large series of non-simulated experiments which search for the best hyper-parameters in both a one-layer neural network and Deep Belief Network. These are carried out on eight datasets in order to recreate and compare the experimental results with those obtained in [76].

Random search is found to outperform grid search on all datasets for single layer neural networks. For Deep Belief Network experiments, random and grid search perform comparitively on four datasets, grid search performs best on three datasets, while random search works best on the fourth. In [16], the authors offer many reasons as to why random search is a better option. Most of these reasons hinge on their demonstration that the hyper-parameter search space, although high-dimensional,

has a low effective dimensionality. This means that although there are many parameters to tune, only a particular subset of these have a great effect on training the model and this subset is different for every dataset. The property of low effective dimensionality leads to random search being more effective than grid search as it leaves fewer gaps in the search space and it does not require as many iterations in order to find the optimum hyper-parameter configuration. In our system, we include both automatic grid and random search but we allow for search at multiple resolutions. Both [16] and [76] chose to globally optimise the parameters of the entire network at once. We previously explored incrementally tuning the constituent parts of a deep network, although our evaluation showed this to require some element of manual intervention [101].
Other techniques include a Bayesian approach [121], where shallow learning approaches are optimised by modelling their performance as a sample from a Gaussian process. They demonstrate that their method offers improvements on previous approaches and can reach or even surpass human expert selection. In [88], the authors propose a gradient based approach to optimisation through reversible learning. They argue that a reversible learning gradient approach, allows for a much richer parametrisation of deep networks in their hyper-parameters, in contrast to previous approaches where the number of hyper-parameters that can be optimised is limited. A gradient approach allows for 100s of hyper-parameters to be optimised, where necessary. Both [121] and [88] involve *meta-iterations* which is what we refer to as hyper-parameter configuration *trials*, where complete runs of model parameter optimisations which are run for different hyper-parameter configuration trials. Our framework provides for this high-level approach and therefore, these methods can easily be integrated. Unlike our approach, none of these methods discussed offer a high level focus which examines those elements common to all, or exist in a framework that allows for the analysis of *why* certain hyper-parameters work for particular applications.

Table 2.1: Summary: Application Issues and Requirements

	Issues	Requirements
Configuration		
a1	Many heterogeneous DN types	Develop abstract DN config. model
a2	DL experiments complex to perform	Define steps to perform for DL exp.
a3	Very difficult to use DL for DM	Link DL to existing DM processes
Optimisation		
a4	Many approaches to HP opt.	Develop abstract model of HP opt.
a5	HP opt. often manual	Provide automatic HP opt.
a6	HP choice often opaque, arbitrary	Provide empirical (interim) results
Interpretation		
a7	No analyses why HPs optimal	Provide analysis why HPs optimal
a8	No means to generally analyse MPs	Provide generic MP analysis

2.1.5 Summary: Application Issues and Requirements

Table 2.1 summarises the issues we have discovered when deep learning has been applied to the practical goals for data mining outlined in §1.3.1. Each issue relates to current shortfalls in the Configuration, Optimisation or Interpretation of deep learning (DL) experiments for greater levels of experiment utilisation, automation and reproduction. First, there are a large number of deep network (DN) types, approached in heterogeneous ways (Issue a1). Thus, for automation, an abstract, homogeneous DN configuration approach is required. Next, there is no definition of the required steps and components in a DL experiment. This would greatly reduce application complexity (Issue a2). Furthermore, links to existing knowledge extraction processes would make the development of practical DL data mining solutions (Issue a3) far easier in business contexts. For optimisation, hyper-parameter (HP) choice is often manual (Issue a5), or opaque and arbitrary (Issue a6). Abstracting and automating the approach to the many possible HP optimisation schemes (Issue a4) and capturing interim models, would provide empirical evidence currently omitted from literature. Finally, hyper- and model parameters cannot or are not analysed in the literature (Issues a7, a8). Providing the means to analyse these parameters, generically, would encourage further applications of deep networks as we understand *why* hyper-parameters are optimal, and *what* a network learns.

2.2 Machine Learning and Data Models

In this section, we explore *data model* approaches to machine learning [54], [45], [133], [134], [79]. Much of this interest is recent and lead to the development of certain experimental supports to aid in performing, analysing and exchanging machine learning experiments [45], [133], [134], [79]. Tools such as Ontologies and Model Interchange Formats (MIFs) were designed for *shallow* machine learning paradigms. However, we believe a rethink was required to capture all aspects of deep learning experiments.

To accurately reproduce a machine or deep learning experiment, transparency and empirical reasoning in the choices made throughout the experiment are crucial [20]. Despite this, levels of research on data mining and machine learning experiment reproducibility, interoperability and exchange are low. Furthermore, final results presented in the literature, are often the product of multiple experiments or multiple learner optimisations [121], with interim results often omitted. When deep learner models and results *are* shared, they lack a defined data model and are generally presented in language or library specific serialisation formats which establishes a low level of interoperability. We believe that all aspects of a deep learning experiment should be capturable and stored, including intermediate results, if required.

There have been attempts to address the issues of learning function interoperability [109], [56], [54], [133], to capture the data properties of a machine learning experiment through the development of various ontologies [45], [72], [133], [79] and methodologies for the persistent and transparent storage of experiment data [21], [133], [135]. The first Model Interchange Format (MIF) defined for predictive data mining functions was the Predictive Model Markup Language (PMML) [54] by the Data Mining Group [56]. Their aim was to provide an *open* mechanism for working with different types of predictive models which arose in data-mining, by defining a convenient language for exporting and importing model descriptions between different systems. Their experience with data mining applications had shown the usefulness that a flexible interchange mechanism would provide and that previous interchange formats tended to be closed and proprietary.

Although some deep learning models can be represented by PMML, we found it to be ill-suited to the specific concerns of deep learning for a number of reasons. Primarily, PMML lacks a flexible, abstract, conceptual data model to describe certain elements relevant only to deep learning models and their context within a experiment [54]. As its schema is fixed, it cannot handle a rapidly changing environment such as deep learning. Finally, because PMML lacks such a conceptual model, it is inextricably linked to XML. PMML is essentially a pre-defined XML schema and for reasons which we will shortly present, XML is not an optimal mark-up representation for the data generated from deep learning experiments for a number of reasons.

The focus of PMML is on model deployment and interchange and as it does not describe many elements in a machine learning experiment, it cannot be used to implement a storage model for interim results. Specifically, it cannot capture crucial information relating to hyper-parameter optimisation. Furthermore, PMML is focused on predictive models and does not provide sufficient functionality for unsupervised methods [56]. It focuses on the interchange and deployment of predictive models whereas we capture information on how models are trained. Finally, PMML can only represent certain fixed model types, as any new model has to be added specifically by the data mining group. Our model provides flexible concepts that can be used to represent general deep learning functions and is extensible.

XML is ill-suited to representation for deep learning for a number of reasons. First, it is a document-oriented and not data-oriented which means it cannot capture many fine grained elements of deep learning. Syntactically, it is quite verbose when compared to more modern language agnostic serialisation formats, which makes it more expensive to store and query and harder to read by the human eye [98]. Lastly, and perhaps most importantly, because of its syntax, XML requires specialised parsing before it can be imported into programming language data-structures. We view these reasons as why JSON has surpassed XML in popularity as a web-native data-interchange format [98] and is therefore, a better interoperable format to deploy a data model of this type.

The Portable Format for Analytics, PFA [109], [127], [111] abstracts the description of a machine learning models allowing user-defined algorithms and models, which more closely aligns with our approach. It incorporates JSON for its implementation, which is a more suitable language for deep learning as it is a data representation language, less verbose and faster than XML [98]. Furthermore, PFA provides a mechanism to export and exchange models found through language-specific software implementations from the environment in which they were built. The aim of PFA is to provide a means for the deployment of such models in production environments rather than a mechanism to store and analyse the elements which made up the experiment which generated the learner model *as well* as a means to exchange and deploy the learner model. Furthermore it is a 'mini-language' and not a data model for concept representation and storage. It provides the capability to take in and score data according to the learner model that it has implemented. This means it is a complicated language, whereas we sought to deliver a light-weight data representation and storage format that is simple to use and contains a formal description of a deep learning *experiment and model.* Finally, it does not capture the concept of *intermediate results.*

The authors of [45] present the MEX vocabulary, a lightweight interchange format for Machine Learning experiments. It is presented as an extension to the PROV-O [79] ontology, which is a W3C recommended vocabulary for the representation and exchange of provenance information generated by different applications and systems. The aim of the vocabulary presented in [45] is similar to our own, but instead they take a linked-data, semantic web approach instead of a concrete data-modelling approach. Also similar to our own work, they provide a description of the core elements of a learning experiment instead of exhaustively defining all elements relating to the knowledge discovery process. Although they represent a generic learning experiment, they do not provision for deep learning experiments, which we contend encompasses elements unique to deep learning and therefore requires a specialised representation. Furthermore, they do not link their vocabulary to a persistent storage structure which would physically store all the experiment results. The DMOP ontology for data mining [72] is fundamentally different to our approach. They provide an abstract ontology which exhaustively describes the *entire* knowledge discovery or data mining process but do not provide a physical or implementation model for this ontology, which is crucial in pratical applications. We also argue that an RDF graph-store does not map as well to the hierarchical structures presented in deep learning. Finally, RDF stores and the aforementioned ontologies are more suited to metadata whereas an experiment encompasses both data *and* metadata.

2.2.1 Summary: Data Model Issues and Requirements

Table 2.2: Deep Learning Data Issues and Requirements

	Issues	Requirements
Configuration		
b1	Special DL structures not modelled	Create conceptual model of DL
b2	DL data requirements evolve rapidly	Make schema flexible, not fixed
Optimisation		
b3	No physical model for ML or DL exps.	Provide physical storage model
b4	ML models omit optimisation data	Capture optimisation (interim) data
Interpretation		
b5	DL uses language specific data formats	Use interoperable storage format
b6	Verbose data formats expensive	Use lightweight data formats
b7	Bulky data models cumbersome	Include minimal attributes possible

Table 2.2 summarises the current issues and requirements in relation to the capture and exchange of DL experiments, which again inhibit automation, application and reproduction. Primarily, DL utilises specialised structures - tensors, layers - which have no current representation (Issue b1). Instead, model parameters are typed as individual integers, limiting usefulness. As a rapidly evolving field, with new networks under constant development, current static data models quickly become irrelevant (Issue b2) for DL. For optimisation, existent paradigms do not provide physical models (Issue b3), or capture model *and* hyper-parameter optimisation (Issue b4), a paramount practical concern. Finally, DL experiments and learners should utilise lightweight, interoperable formats, with minimal required attributes. Language-specific (Issue b5), verbose (Issue b6) and overly-exhaustive paradigms (Issue b7) are less portable and require steep learning curves.

2.3 Deep Learning Frameworks

In this section, we examine research into frameworks for deep learning. We discovered that although some frameworks had a wide range of functionality in certain areas, for example, better model parameter optimisation techniques or a broader range of deep learning algorithms, none address *all* concerns identified in this research as being critical to managing deep learning experiments. At the time of writing, there are *at least* 30 deep learning libraries [126] at varying stages of maturity and range of functionality. Therefore, we cover only the most popular deep learning libraries that are in use today. To attempt to narrow the field of discussion, we discuss those libraries listed by Nvidia who are leaders in the field of deep learning as their Graphical Processing Units are the de facto engine of deep learning research [99].

Caffee (Convolutional Architecture for Fast Feature Embedding) [68], CNTK (Computational Network Toolkit) [42], Tensorflow [1] and MXNet [31] were all built using C++. Torch [34] is built on Lua and Deeplearning4j [39] on Java. Finally, frameworks such as Digits [100], Chainer [129] and Theano [9], [128], [17] are all Python based. We also include the Python library Neon [124] as this has been reported to achieve the state performance for training several deep architectures [8]. Furthermore, as Theano operates at quite a low level, tools like Keras [33], Lasagne [41] and Blocks [132], based upon Theano are better suited to comparison. Of all these frameworks Caffee, Theano, Torch and are the most popular and widely used by the deep learning community for research and development [71]. Four of the systems presented are the focus of most comparisons in research [8] and these will be discussed now.

Caffee is primarily focussed on computer vision and multimedia applications [68], [39], although recently it has been expanding its application aims. Caffee is highly useful, providing a series of pre-trained models and a means to serialise them, but they are provided as *caffemodel* binaries. These require the user of these models to also utilise Caffe as their deep learning library of choice. Hyper-parameter optimisation is not explicitly provided in this framework as separate libraries must be used in conjunction with Caffee. Furthermore, although models can be stored, there is no standard interoperable data model which links the learner generation to the hyper-parameter optimisation in an interoperable format. Their software model is layer-based like ours, but does not contain the higher level of abstraction common to all the deep networks that would allow users to easily implement new layers and architectures. Finally, Caffee does not allow abstract hyper-parameter optimisation processes to be included into new implementations from the outset.

Deeplearning4j offers a package with the most similar functionality to our analytical toolkit which is called Arbiter, but their configurations of hyper-parameter optimisation will be difficult to share. It is based on Java objects and types and has no published data model, interchange format or experiment database, so sharing is difficult. At the time of writing, Nvidias Digits platform is the only library that offers a hyper-parameter sweep, although their sweep is limited to just batch size and learning rate. In contrast, our data model captures all aspects of deep learning experiments.

The Theano library [17], [9], [128] was integrated with our own library and thus, we can extend and adapt its functionality. Neither Keras nor Lasagne have built-in hyper-parameter optimisation capabilities. Theano has a companion library called Jobman, which allows you to schedule and run experiments and store the results of different hyper-parameter trials in a flat relational database. However, it is not based on a published data model that describes each element of a deep learning experiment, nor does it allow for the storage of interim or final model parameters.

There are libraries which implement state of the art optimisation techniques such as *Hypergrad* [55], *Hyperopt* [15] and *Spearmint* [120] but these frameworks are not based on a flexible software model or published data model for the analysis of deep learning experiments. Instead, they must be integrated with the deep learning framework being utilised, or can serve as a segregated wrapper process for deep learning experiment scripts, which makes experiments hard to repeat and reuse.

Table 2.3: Deep Learning Framework Issues and Requirements

	Issues	Requirements
Configuration		
c1	Highly manual algorithm config.	Provide automatic algorithm config.
c2	Separation of concerns	Loosely couple toolkit components
c3	Often domain specific	Make tools domain agnostic
Optimisation		
c4	Provide none or limited HP opt.	Provide opt. for all HPs
c5	Separate libraries for MP and HP opt.	Integrate MP and HP opt.
c6	HP opt tools do not capture MP opt.	Capture MPs and HPs
c7	Cannot pause, reload, back-trace exps.	Capture interim results
Interpretation		
c8	No analysis functions for parameters	Provide parameter analysis functions

2.3.1 Summary: Framework Issues and Requirements

Table 2.3 presents the issues found from the review of current deep learning frameworks. Essentially, no framework covers all three elements of experiment Configuration, Optimisation *and* Interpretation and most only provide limited optimisation in terms of hyper-parameters (Issue c4) or require separate libraries (Issue c5). A major issue with wider application of deep learning for current frameworks is that a number are domain specific (Issue c3) and require the manual configuration of algorithms (Issue c1). Separation of concerns (Issue c2) is addressed by most frameworks but we believe that it is important to highlight as capture, configuration, optimisation and interpretation should be loosely coupled to provide the greatest degree of flexibility possible for an experimental framework. Issues c6 and c7 are related to concerns raised in the previous section, but again here we focus on the parameters themselves rather than the optimisation process as current HP optimisation frameworks do not capture model *and* hyper-parameters and DL frameworks do not provision for experiment pause, reload or back-tracing. Finally none currently provide hyper-parameter and model parameter analysis functionality, necessary to interpret an experiment.

2.4 Summary of Related Research

The related research in this chapter contrasted shallow learning with deep learning and highlighted where we make new contributions to the state of the art in deep learning. With regard to hyper-parameter optimisation, we explored gold standard methods and provided an overview of the state of the art. What we show to be lacking is a suitable experimental framework that can support *multiple* goals for a single experiment. There does not exist a framework or method where the multiple outcomes of: anomaly detection; representation learning for dimensionality reduction and interaction learning; sequential prediction; and hyper-parameter optimisation, can be *integrated* and *understood* in a single experiment. By covering these key areas, this served to highlight how our research makes a contribution to deep learning in an overall sense.

We also show that there have been attempts to specify a data model for machine learning. This has obvious benefits: a standard for describing these experiments; sharing and reusing experimental setup and results; extending systems which use this data model to include new analytical functions which become available to all researchers. However, we showed that the field of deep learning research does not currently have an adequate data model.

State of the art deep learning frameworks provide certain supports but lack other functionality present in a traditional data mining and machine learning contexts. Furthermore, there has been no end-to-end energy based deep learning solution that allows a user to detect anomalies, discover variable interactions and make predictions in time-series data, although recent literature points to the potential for the development of such a solution. We argue that this is because current frameworks do not have or are not based on a *abstract* software model and *method* generally applicable to the majority of deep learning paradigms which integrates hyper-parameter optimisation.

Table 2.4 therefore summarises the requirements we have presented in previous sections and the principals which follow from these requirements which we have used to develop our solution in Chapters 4, 5 and 6. In summary experiments

Table 2.4: Requirements Summary and Design Principals

	Tool Requirements	Principals
Configuration		
a1	Develop abstract DN config. model	Genericness, Extensibility
a2	Define steps to perform for DL exp.	Simplicity, Reusability
a3	Link DL to existing DM processes	Flexibility
b1	Create conceptual model of DL	Flexibility, Reproducibility
b2	Make schema flexible, not fixed	Flexibility, Extensibility
c1	Provide automatic algorithm config.	Automation
c2	Separation of concerns	Atomicity
c3	Make tools domain agnostic	Genericness, Reusability
Optimisation		
a4	Develop abstract model of HP opt.	Genericness, Extensibility
a5	Provide automatic HP opt.	Automation, Simplicity
a6/b4/c6	Provide/capture interim results	Reproducibility
b3	Provide physical storage model	Simplicity
c4	Provide opt. for all HPs	Automation, Simplicity
c6	Integrate MP and HP opt.	Automation, Simplicity
c7	Capture MPs and HPs	Flexibility
Interpretation		
a7	Provide means to analyse optimal HPs	Interpretability
a8	Provide generic MP analysis	Genericness, Interpretability
b5	Use interoperable storage format	Reusability, Portability
b6	Use lightweight data formats	Simplicity, Portability
b7	Include minimal attributes possible	Simplicity
c8	Provide parameter analysis functions	Interpretability

should be *configurable*, in that experiments and learner configurations should be generic, extensible, automated, simple, reproducible, reusable and flexible, with experiment framework components themselves loosely coupled and atomic. The experiment should allow for full *optimisation*, of model and hyper-parameters, where the processes involved are generic, extensible, automated, simple, reproducible and flexible. Finally, all parameters and elements of a DL experiment should be portable and have a means to be *interpreted* in a generic and simple way. Some of these requirements will be satisfied over the course of our solution design, whereas others can only be demonstrated to be satisfied at experiment run-time in Chapter 7.

Chapter 3

Neural Networks

In the previous chapter, we explored machine learning literature and identified the need for a framework that enables easy configuration, running, interpretation and sharing of deep learning experiments. In this chapter, we aim to explain the theory, terminology and mathematical notation of neural networks, specifically those used in this research. In §3.1 we provide an introduction to, and the definitions of, several concepts central to the chapter and to the theory. As deep learning is a substantial and complex topic, we endeavour to make this chapter as accessible as possible. We break deep algorithms and neural networks into simpler atomic components and explain the *building blocks* of these networks in §3.2. In §3.3, we stitch these components together and explain the shallow and deep neural networks used throughout this research. It is the most theoretical chapter in the dissertation, but necessary to give the reader an understanding of the utility and function of these algorithms. The notation used throughout the chapter we have synthesised from [96], [84], [125] and [53], but we have altered it in order to try and achieve a consistent style across heterogeneous neural architectures. The analysis presented in this chapter was necessary to satisfy the *requirement* of developing an abstract approach to the *configuration* and *optimisation* of deep networks in a generic and extensible manner.

$$V = [\ V_1 \quad V_2 \quad \dots \quad V_n\ ]$$

$$X = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \dots & x_{mn} \end{bmatrix} \quad Y = \begin{bmatrix} y_{11} & \dots & y_{1K} \\ y_{21} & \dots & y_{2K} \\ \vdots & \ddots & \vdots \\ y_{m1} & \dots & y_{mK} \end{bmatrix}$$

Figure 3.1: Dataset

3.1 Basic Concepts

Before discussing neural network algorithms, it is necessary to introduce the appropriate terminology and notation for the basic concepts that will be used throughout this and subsequent chapters.

3.1.1 Dataset Concepts

We begin with a definition of a *Dataset*, the input to a neural network or machine learning algorithm.

Definition 3.1. Dataset.
Let $\mathcal{D}$ be a dataset which we define as $\mathcal{D} := \{V, X, Y\}$, a triple with a set of feature variable labels $V := \{v_1, \dots, v_n\}$, an input data matrix $X \in \mathcal{X}^{m \times n}$ and a classifications matrix $Y \in \mathcal{Y}^{m \times K}$, where $\mathcal{X}$ is the input space, $\mathcal{Y}$ is the target space, m is the number of rows, n is the number of features and K is the number of classifications.

The input and target spaces can be binary $\mathcal{X} = \{0, 1\}^{m \times n}$; $\mathcal{Y} = \{0, 1\}^{m \times K}$; or in the domain of real numbers $\mathcal{X} = \mathbb{R}^{m \times n}$; $\mathcal{Y} = \mathbb{R}^{m \times K}$. For unsupervised learning there are no classifications, $\mathcal{Y} = \varnothing$. To refer to all values for a particular feature or sample, we use the subsequent definitions of column vectors and row vectors.

Definition 3.2. Data Matrix.
A matrix is a 2-dimensional array of numbers. Let $X = \{x_{11}, \dots, x_{mn}\}$ be the flattened data matrix, where each element $x_{ij} \in X$ refers to the value for the jth feature in the ith sample.

In Definitions 3.2 and 3.1, we show there are $m \times n$ values in a data matrix, the number of samples multiplied by the number of features. The data matrix can also be thought of containing m **horizontal** n-dimensional row vectors. These rows of the data matrix are also known as data row vectors, samples, instances or observations.

Definition 3.3. Data Row Vector.
Let $x_{i\bullet} = \{x_{i1}, \ldots, x_{in}\}$ be the data row vector, where $x_{i\bullet}$ is set the of values for all features in the ith sample of the data matrix X, in dataset $\mathcal{D}$.

In definition 3.3, a Data Row Vector $x_{i\bullet}$ contains all attribute (or variable) values $\{x_{i1}$ to $x_{in}\}$ for the ith instance of the data matrix $X \in \mathcal{D}$. Alternatively, a data matrix can also be said to contain n **vertical** m-dimensional column vectors. These columns can also be referred to as feature, attribute or variable vectors.

Definition 3.4. Data Column Vector.
Let $x_{\bullet j} := \{x_{1j}, \ldots x_{mj}\}$ be the data column vector $x_{\bullet j}$, which contains all sample values for the jth feature in $X \in \mathcal{D}$.

In definition 3.4, a Data Column Vector $x_{\bullet j}$, contains the set of all sample (or instance) values x_{1j} to x_{mj} for the jth feature of the data matrix $X \in \mathcal{D}$.
The Y classification matrix is addressed in the same way as the data matrix X, an element is referred to as $y_{ij} \in Y$. In the case of binary (yes/no; 1 or 0) or real-valued classifications, $K = 1$. This gives a column vector $Y = y_{\bullet}$, where y_i is the classification associated with ith sample, either 1 or 0 in the case of binary, or a numeric value in the case of real-valued outcomes. If there are one or more possible classifications from a discrete set, $Y = y_{\bullet\bullet}$, a binary matrix. The classification associated with the ith sample is then a row vector $y_{i\bullet}$, where $y_{ij} = 1$ if the jth classification is assigned to the ith sample. Conversely, $y_{\bullet j}$ is a column vector indicating to what samples a particular classification is assigned.

3.1.2 Neural Network Concepts

The aim here is to provide high-level introductory definitions and deal with technical aspects in the following sections. To describe a neural network, we first define a

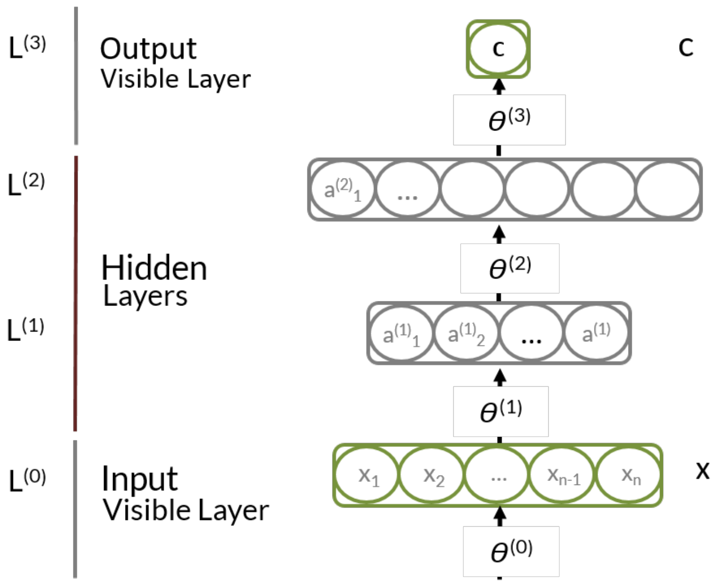


Figure 3.2: Neural Network Components

Neural Network, and subsequently the *Layer* and *Node* components.

Definition 3.5. Neural Network.

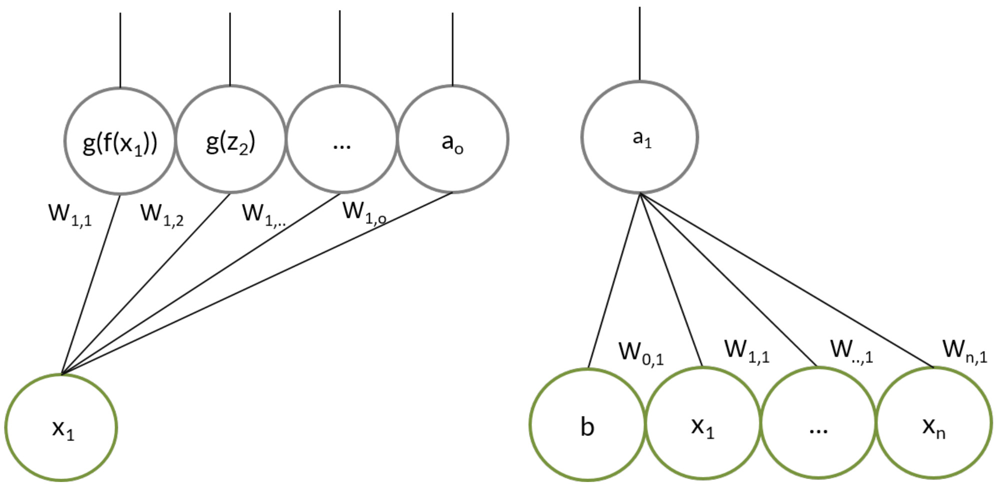
A neural network L, is a collection of layers, $L := \{L^{(0)}, \dots, L^{(|L|)}\}$, where $|L| >= 2$.

In definition 3.5, $|L|$ is a network's *cardinality* or number of layers, where $|L| \geqslant 2$. Each layer $L^{(l)}$ is referenced by a superscript index l, denoting the layer's position in the network architecture. Figure 3.2 shows there are three distinct layer-types possible: *visible-input*, *hidden* and *visible-output*.

$L^{(0)}$ is always the input layer. Generally, the output of each node in the bottom *visible input* layer represents a feature value x_{ij} in the dataset. For supervised models (predicts an outcome given an input) there is a *visible output* layer. In such cases, $L^{(|L|)}$ or C refers to the output layer. Layers $L^{(1)}$ to $L^{(|L|-1)}$ are then the hidden layers. In unsupervised networks, $L^{(1)}$ to $L^{(|L|)}$ are hidden. Inputs are combined and an abstract features or *variable interactions* are learnt in hidden layers. We now define the *Layer* construct.

Definition 3.6. Layer.

A Layer $L^{(l)} := \{N^{(l)}, \theta^{(l)}\}$, is a tuple containing nodes $N^{(l)}$ and node parameters



(a) Node Sending (b) Node Receiving

Figure 3.3: Calculating Activations

$\theta^{(l)}$ for those nodes in layer $L^{(l)}$ of a neural network.

Figure 3.2 shows we apply Definition 3.6 to all layers in a neural network, to give a homogeneous data-model. Generally the parameters for the input layer are null as the outputs of *input* layer equate to dataset variables, *except* in the case of some unsupervised networks, which we describe in §3.3.3. A Layer also combines each parameter with the appropriate node.

Definition 3.7. Node Parameters.
The parameters for the lth layer in a network are given by $\theta^{(l)} := \{b^{(l)}, W^{(l)}\}$, $W^{(l)}$ is the set of weights and $b^{(l)}$ is the set of biases for a layer.

In Definition 3.6 we show that a set of weights $W^{(l)}$ and biases $b^{(l)}$ are associated with each layer and together are referred to as $\theta^{(l)}$. The simplest component of a neural network is a *node*, which is the computational unit in a neural network.

Definition 3.8. Node.
A node $N_o^{(l)}$ can be defined with the properties $\{a^{(l')}, z_o^{(l)}, a_o^{(l)}, f_l, g_l\}$, where l is the layer identifier and o the node identifier within the layer; $a^{(l')}$ is the vector of inputs; and $z_o^{(l)}$ and $a_o^{(l)}$ are the linear and output activation energies, calculated by f_l and g_l, respectively.

In Definition 3.8, $a^{(l')}$ represents a vector of inputs received from another layer $L^{(l')}$. The single linear activation energy value is given by $z_o^{(l)}$ and generated by the node's linear activation function f_l; $a_o^{(l)}$ is the single output activation value; and g_l is the nodes output activation function used to calculate the output activation.
Activations are the values calculated by a node and refers to a node's *importance*. We show in Figure 3.3 how parameters are combined with inputs to calculate node activations. Taking a weight matrix $W^{(l)}$ as an example, weight $W_{io}^{(l)}$ is the coefficient combined with input x_i, as part of the calculation for oth node in the current layer $L^{(l)}$.

3.2 Neural Network Components

Neural networks consist of *visible* and *hidden* layers. Visible layers are composed of *observed* variables, which are present in the data input and classification output layers. Algorithms can learn to map visible inputs directly to visible classification outputs. More expressive algorithms, like neural networks, can learn an intermediate representation to better relate inputs to outputs. *Hidden layers* comprised of *unobserved* or *latent* variables, detailing input variable interactions, make up this intermediary representation.
We now introduce components which are common to, and reused in, several neural networks. These components act as building blocks which are combined in different ways to form more complex learning architectures.

- Feed-forward hidden layers. These are the most basic component of a neural network. A feed forward layer is *integral* to all other components as its functionality can be *extended* to realise other layers, like those in the descriptions that follow. Alternatively, it can be *combined* with other layers to make up complex networks. Furthermore, in understanding the hidden nodes of a feed-forward layer, we learn how a network combines inputs which amounts to the *interactions* a network learns between them.

- Recurrent hidden layers. These function similar to the basic feed-forward layer, taking input and extracting *learnt features.* In contrast, they can account for or sequential or time-series data. The recurrency allows the network to incorporate data from previous time-points into the classification process for a current time-point, allowing for the use of contextual information in order to make predictions.

- Regression visible output layers. These layers take input and *perform classifications* based on this input. There are three types: *linear*, *logistic*, and *softmax* regression layers. In the context of this dissertation, their main function is that of a *visible output* layer to supervised neural networks. Regression layers extend the concept of a hidden layer through cost functions, and as regression itself is a *shallow* learning algorithm, the ability to update parameters.

Note on Descriptions. We explore components and neural networks, via high level processes or key *functions* which we have defined for **Layer** and **Learner** in our Configurable Deep Network in Chapter 5. For a Layer these are: *initialisation* of parameters; *propagation* of values; and *sampling.* For a neural network they consist of: building a *hypothesis*; calculating a *cost*; building *updates* for parameters through the calculation of derivatives; and finally, optimisation, through a given *training* procedure.

Note on Equations. Where possible we describe vectorised equations. Vectorised calculations allow for a single sample *or* a batch (matrix) of samples to be processed in parallel through standard matrix operations providing greater efficiency. Where possible and for simplicity, we will provide equation descriptions in terms of a single input and output vector.

Note on Equation Inputs. The input to a hidden layer $L^{(l)}$ can come from another hidden layer in the architecture $L^{(l-1)}$ or $L^{(l+1)}$, or from a data matrix X. For simplicity, we show the equations of Sections 3.2.1 and 3.2.2 as layer $L^{(l)}$ receiving the activation energies $a^{(l-1)}$ from hidden layer $L^{(l-1)}$ as input. As regression is itself a learning algorithm, we show the input of the equations in §3.2.3 to be x, although when functioning as the output layer of a supervised network, the input would be $a^{(l-1)}$, the previous layer's activation energies.

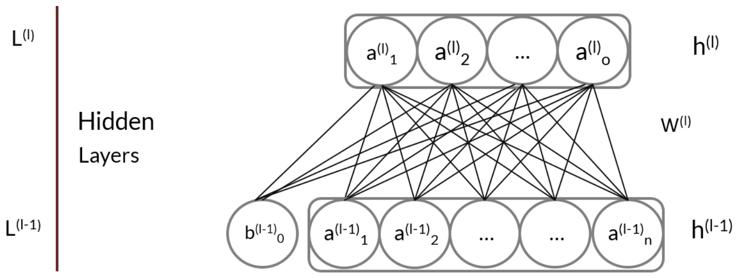


Figure 3.4: Hidden Layer: Hidden to Hidden

3.2.1 Feed Forward Hidden Layer

Figure 3.4 shows the make-up of a basic feed-forward hidden layer, receiving input from another hidden layer in a neural network. The hidden layer's function is to take values from an *input visible* layer (the data) or another *hidden*, *transform* these values and output transformed values to the next layer - either another *hidden* layer or an *output* layer. Figure 3.4, omits input layer weights for the sake of simplicity. The hidden layer learns *latent* features from the data through these transformation functions by weighting and combining inputs differently in each node.

3.2.1.1 Parameter Initialisation

Before any calculations are carried out, the *parameters* $\theta^{(l)}$, for a layer $L^{(l)}$ must be instantiated. These parameters consist of the **weight matrix** $W^{(l)}$ and **bias vector** $b^{(l)}$. We initialise bias vectors with values of 0 and a dimension $|N^{(l)}|$, equal to the number of nodes in $L^{(l)}$. The dimension of a weight matrix $W^{(l)}$ is $|N^{(l-1)}| \times |N^{(l)}|$, where $|N^{(l-1)}|$ is the number of nodes in the previous layer $L^{(l-1)}$. The initial values of the weight matrix depend on the non-linear activation function employed. In neural networks in general, there are two types of layers: those which output probabilities and those which output numerical values. Thus, we focus on two types of activation function and adopt standard initialisation procedures from the literature [51].

First, we employ the logistic sigmoid function (described shortly in Equation (3.4)), as its outputs can be interpreted as probabilities. Equation (3.1) shows the initialisation procedure for binary, logistic units. Weights are drawn randomly and uniformly between the bounds shown.

$$W \sim \mathcal{U}\left[-4\,\frac{\sqrt{6}}{\sqrt{|N^{(i-1)}|+|N^{(i)}|}}, 4\,\frac{\sqrt{6}}{\sqrt{|L^{(i-1)}|+|L^{(i)}|}}\right] \tag{3.1}$$

The second, rectified linear function (Equation (3.5)), is used for Rectified Linear Units (ReLU) as this function outputs numeric values. Equation (3.2) shows the parameter initialisation procedure for the ReLU function. Weights are drawn randomly from the normal distribution, with variance $\frac{2}{|N^{(i-1)}|}$.

$$W \sim \mathcal{N}\left(\mu = 0, \sigma^2 = \frac{2}{|N^{(i-1)}|}\right) \tag{3.2}$$

These random sampling initialisations are employed to break *symmetry* during learning. Symmetry is when each hidden node receives identical learning signals. The specific procedures outlined in Equation (3.1) and Equation (3.2) were used as they have been shown to enable the learning of *good* weights [51]. Unless expressly stated otherwise, the initialisation procedure outlined above is followed for *all hidden layers* in non-recurrent architectures in §3.3.

3.2.1.2 Feed-Forward Propagation

The process of transforming values through a *node*, *layer* or entire *neural network* is called *propagation*. When values are propagated *from* the input layer through the architecture, we call this *feed-forward propagation*. When values or gradients are propagated *towards* the input layer, we refer to this as *back-propagation*. In this section, we focus on **layer** feed-forward propagation.

Linear Function. First, the *linear activation* for a layer is calculated. This energy vector is represented by $z^{(l)} \in \mathbb{R}^{|N|}$, a vector of real numbers equal in dimension to the number of nodes in the layer, where each element $z_i^{(l)}$ is the linear activation for the ith node in the lth layer.

$$z^{(l)} = f_l(a^{(l-1)}) = a^{(l-1)} \cdot W^{(l)} + b^{(l)} \tag{3.3}$$

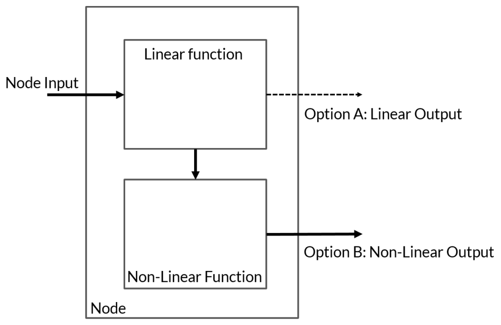


Figure 3.5: Different Node Routes

Equation (3.3) shows the calculation of a layer's *linear energy* $z^{(l)}$. The defined function $f_l \colon a^{(l-1)} \mapsto z^{(l)}$, represents a layer's *linear* activation function. The function receives $a^{(l-1)}$ as input, the vector of *activation energies* from the previous layer $L^{(l-1)}$. The *linear energy* is calculated as the *dot product* between $a^{(l-1)}$ and the weight matrix $W^{(l)}$ for the lth layer, before adding a bias vector, $b^{(l)}$.

Once $z^{(l)}$ is calculated, it can be output directly. Figure 3.5 shows this as Option A, where $a^{(l)} = z^{(l)}$. Alternatively, the process carries out a non-linear transform on the values in $z^{(l)}$. Figure 3.5 shows this as Option B. The most common option is B, as non-linear transformations can model more complex relationships between inputs and outputs.

$$a^{(l)} = g(z^{(l)}) = sigmoid(z^{(l)}) = \frac{1}{1 + e^{-z^{(l)}}} \tag{3.4}$$

Non-Linear Function. Equation (3.4) shows the *logistic sigmoid* function. The vectorised notation signifies the application of the function to each value $z_i^{(l)}$ in $z^{(l)}$. The function $g \colon z^{(l)} \mapsto a^{(l)}$ invokes a non-linear mapping from the result of the linear function to a layer's output. Here g is defined as the sigmoid function. The logistic sigmoid function takes each value in $z^{(l)}$ and returns a value between 0 and 1, which can be interpreted as a series of probabilities. The output tends towards 1 as a value $z_i^{(l)}$ approaches $+\infty$ and 0 as $z_i^{(l)}$ approaches $-\infty$. The symbol e is Euler's number, a constant that enables the compression between 0 and 1. In this case, $a^{(l)}$ is a range of values between 0 and 1 which can be interpreted as $p(N^{(l)} = 1 | a^{(l-1)}, W^{(l)}, b^{(l)})$, the *probability* of each node *firing* given the input and parameters.

$$a^{(l)} = g(z^{(l)}) = ReLU(z^{(l)}) = max(0, z^{(l)}) \tag{3.5}$$

Equation (3.5) shows the second activation function used in our experiments. The Rectified Linear Unit (ReLU) function outputs 0 if the linear energy $z^{(l)}$ is less than 0, otherwise it outputs the linear energy value.
The advantage of the sigmoid function is that values in $a^{(l)}$ never grow exponentially, as they are always constrained to be between 0 and 1. Thus, they can be interpreted as probabilities, but the gradient can vanish. This is not the case for the ReLUs. ReLUs have the advantage of learning a sparse representation - outputting a lot of 0's - and its gradient is never 0. Sparse representations generally lead to more accurate models [12]. The advantages for each function are different, leading to uses in different situations, but ultimately give a better learning signal.
Sampling Function. In certain cases, the *state* $s_o^{(l)}$ of each node in a layer is required. The state of a node describes whether a node *fires* or not, outputting 1 if it does, otherwise outputting a 0 if it does not. For our work, we dealt with sampling from binary probabilities only: Equation (3.4) generates binary outputs. There are two ways we generate states. Equation (3.6), shows the first, a threshold function. If the probability calculated for a node $a_i^{(l)}$ is above a given threshold (usually 0.5) it outputs 1, otherwise it outputs 0.

$$s^{(l)} = \begin{cases} 1 & \text{if } a^{(l)} > threshold \\ 0 & \text{otherwise} \end{cases} \tag{3.6}$$

The second method shown in Equation (3.7), draws $|a^{(l)}|$ samples from a Bernoulli distribution with a probability $p = a^{(l)}$. This sets each node to 1 with a probability of $a^{(l)}$ and to zero with a probability of $(1 - a^{(l)})$. A Bernoulli distribution is a discrete probability distribution with two possible outcomes: 1 or 0.

$$s^{(l)} = \sim B(|a^{(l)}|,\ a^{(l)}) \tag{3.7}$$

These functions are important as firstly, they indicate the presence or absence of a hidden feature for a particular data sample. Secondly, in binary classifications they are required to determine if a sample is positive or negative. Thirdly, they are used to generate hidden feature samples during training a Restricted Boltzmann Machine.

3.2.2 Recurrent Layer

A recurrent hidden layer functions similarly to a feed-forward hidden layer as it takes inputs, applies transformations and generates outputs. The important difference is that it can incorporate *time-series* and *sequential* data processing. We will refer to each data-point in a sequence as a *time-step* although the sequence may not necessarily be time based. For example, hand-writing analysis consists of analysing a *sequence* of letters.
Figure 3.6 shows a recurrent node **unfolded in time**. The node takes input data x for the *current time-step* along with the value output from the *node itself* at the *previous time-step* $a^{(l)(t-1)}$ (note the new index for time). This is the recurrent process, where layer outputs are passed to the same layer at the next time-step as well as to the next layer in the architecture.
The recurrent process begins by initialising the hidden nodes $h^{(l)}$ with a value $a^{(init)}$ (normally zero), for $t = 0$, the time-point before the start of the sequence. This value, set at $t = 0$ and shown as $a^{(1)(0)}$, is fed-forward to begin the sequence. The result of the first calculation $a^{(1)(1)}$ is sent to the next hidden layer $L^{(2)(1)}$ at the current time-point and to the layer itself, one time-step in the future $L^{(1)(2)}$. This process is repeated from the start time $t-t$ (time 0) up to the final time t. Therefore, $a^{(l)(t)}$ at time t is calculated from the output activations $a^{(l)(t-1)}$ at time $t-1$ and data input at the current time-step $a^{(l-1)(t)}$.

3.2.2.1 Parameter Initialisation

It is necessary to initialise a third parameter for recurrent layers in addition to those $W^{(l)}$ and $b^{(l)}$ outlined in §3.2.1.1. This parameter is the recurrent weight matrix. The recurrent, hidden to hidden weight matrix WR has a dimension $|N^{(l)}| \times |N^{(l)}|$, and is used to propagate values from one layer to the same layer a time-step forward

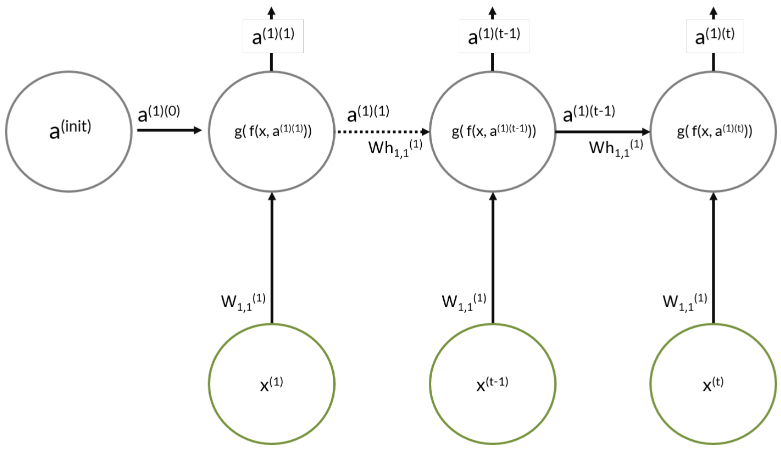


Figure 3.6: Recurrent Node Unfolded in Time

in the sequence. For logistic units, the initialisation procedure outlined in Equation (3.1) is used for $W^{(l)}$ and $WR^{(l)}$. For ReLUs, we initialise the input weights $W^{(l)}$ according to Equation (3.2), but for recurrent weights $WR^{(l)}$, we initialise an identity matrix $I_{|N^{(i)}|}$ in order to learn long term dependencies as outlined in [78].

3.2.2.2 Recurrrent Propagation

As the recurrent layer also takes in to account hidden states of the layer at the previous time-point, its propagation function takes on a different form.

$$z^{(l)(t)} = f(a^{(l-1)(t)}, a^{(l)(t-1)}) = a^{(l-1)(t)}W + a^{(l)(t-1)}WR + b^{(l)} \tag{3.8}$$

Linear Function. Equation (3.8) shows recurrent propagation. In this equation, $f\colon a^{(l-1)(t)} \mapsto z^{(l)(t)}$ is the *recurrent* linear function. Its inputs are the activation outputs from another hidden layer at the current *time point* $a^{(l-1)(t)}$, **and** $a^{(l)(t-1)}$, the activations that the layer itself computed at the previous time-point $t-1$. The parameter W is the weight matrix for the inputs at the current time-point, WR is the weight matrix for the recurrent inputs and b is the bias vector. The function first computes the *dot product* between $a^{(l-1)(t)}$ with the weight matrix $W^{(l)}$, at time t. This dot product is equivalent to the calculation for ordinary feed-forward propagation shown in Equation (3.3). Next, the recurrent input $a^{(l)(t-1)}$ is multiplied

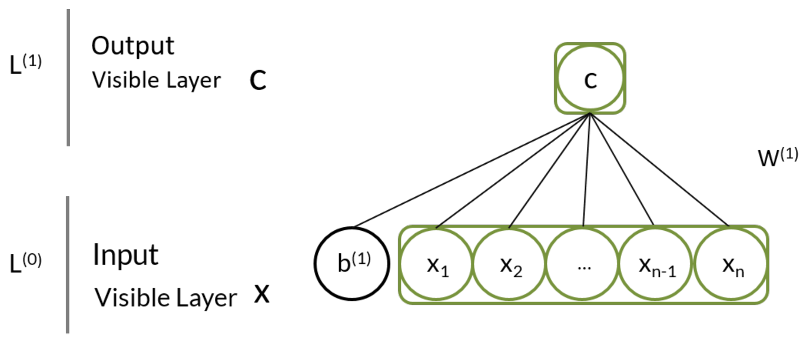


Figure 3.7: Single Output Regression Representation

with the recurrent weight matrix WR. Finally, the result of these two dot products, pertaining to the input and recurrent activation energies respectively, are combined with the bias vector $b^{(l)}$ to give the linear activation energy $z^{(l)(t)}$ of the lth layer at the current time-point t.

When we take the states of the hidden layer at previous time-points into account, the network not only learns an abstract representation of the input features but also has an arbitrarily long 'memory' in which it can reason about the current time-point based on the states of previous time-points. We will explore this notion further in §3.3.2. Once the linear activations are calculated, the same non-linearities outlined in §3.2.1.2 can be applied, as well as sampling processes, if required.

3.2.3 Output Regression Layer

There are various forms of regression functions which take input features and output *classifications* or *predictions*. Conceptually, the different forms of regression are similar to a neural network hidden layer. The prediction or *hypothesis* function of regression can be considered a combination of the *propagation* and *sampling* functionality of a hidden layer, but we use it as a neural network output layer. Regression extends the concept of a hidden layer with an ability to calculate a *cost* and apply *updates* to parameters in order to lower the computed cost.

We focus on four types of regression with different outputs.

1. **Linear regression**, corresponding to a **single**, **real-valued** output.

2. **Logistic regression**, corresponding to a **single**, **binary-class** output, for

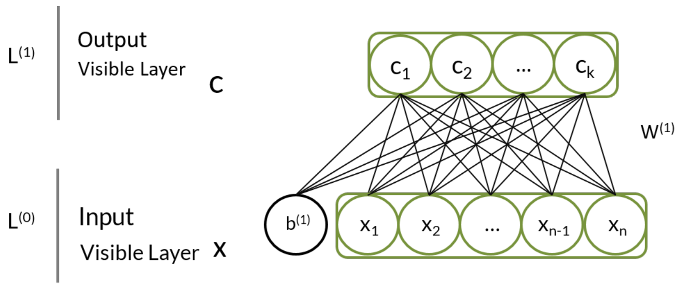


Figure 3.8: Multiple Output Regression Representation

example: yes/no; 1/0.

3. **Multiple logistic regression**, corresponding to **multi-class** output where a single input can have **multiple** classifications.

4. **Softmax regression**, corresponding to **multi-class** output where a single input has a **single** classification from a range of possibilities.

We break our discussion into six parts. The first, §3.2.3.1 describes the **hypothesis functions** used to produce the desired outputs. The *hypothesis function* takes input dataset variables and outputs *numeric predictions* or *probabilities* that are used for classification. The second, §3.2.3.2 describes the **cost functions** used for different classification types and the **regularisation functions** combined with these cost functions. Cost functions measure the difference between the *ground truth* and the *hypothesis* output. Regularisation gives a means to increase the cost if model parameters are overly large. The final three sections then cover the functions required and the mechanism by which we minimise the (possibly regularised) cost function. Minimising a cost function - using **derivative**, **parameter update** and **optimisation** functions - brings predictions closer to the ground truth. Minimising a regularised cost function results in simpler models with smaller weight values and thus, should have better classification performance on unseen instances. These cost and regularisation functions, as well as the procedure for calculating and applying updates, are also used in more complicated learning algorithms and will therefore, be referred to when discussing the neural networks in §3.3.

Figure 3.7 shows regression, consisting of an input vector and a single classification node output, as is the case with linear and logistic regression. Figure 3.8 shows multiple classification nodes, as is the case with softmax and multiple logistic regression. We **initialise parameters** for regression with zeros instead of the random procedure discussed in §3.2.1. This process is also adopted for neural network *output* layers. We adopt this initialisation procedure as we assume no collinearity between abstract hidden features in a neural network, which form the input to the regression layer. Collinear variables are highly correlated, where one input variable can be used to predict another.

3.2.3.1 Hypothesis Function

The regression *hypothesis function* $h_\theta(x)$, produces a continuous numeric value *or* probability, when given input x. This enables a *prediction* denoted as $\hat{y}$. *Linear*, *logistic* and *softmax* regression hypothesis functions each have a different form and the type of regression employed is dependent on the type of variable to be classified.
Linear Regression. Equation (3.9) shows the hypothesis function for *linear regression.* The prediction $\hat{y}$ is equal to the output of the hypothesis function $h_\theta(x)$. The hypothesis function for linear regression is directly equivalent to the linear propagation function f of a hidden layer with **one** node (corresponding to a single output prediction), given by Equation (3.3) in §3.2.1.2. Therefore the prediction $\hat{y}$ is equivalent to z in Equation (3.3). It predicts a real-valued, numeric output $\hat{y_i} \in \mathbb{R}^m$, for each data sample $x_{i\bullet} \in X^{m \times n}$.

$$\hat{y} = h_\theta(x) = f(x) \tag{3.9}$$

Logistic Regression. Equation (3.10) shows the logistic hypothesis function, where the function f is given by the linear function in Equation (3.3) and g is equivalent to the *sigmoid* function given by Equation (3.4) in §3.2.1.2. Therefore, the logistic regression hypothesis function is directly equivalent to the activation energy a of a hidden layer with one node and a sigmoid activation function. Instead of representing the probability of a node firing or not, the hypothesis function output represents a class membership probability $p(\hat{y} = 1|x, \theta)$. The node state s given by the sampling function in Equation (3.6) is equivalent to the actual prediction $\hat{y}$ of logistic regression, where $\hat{y}_i \in \{0,1\}^m$.

$$\hat{y} = h_\theta(x) = g(f(x)) \tag{3.10}$$

Multiple Logistic Regression. The process and calculations described for logistic regression are extended for multiple logistic regression. Where logistic regression calculates a single binary probability of class membership, multiple logistic regression calculates many, where for each classification, the same calculation described for logistic regression is repeated for every possible class. This is equivalent to a hidden layer with multiple hidden nodes and a *sigmoid* activation. The output $\hat{y}_i$ is a vector of ones and zeros, with ones indicating a positive prediction for the class referenced by that index.

Softmax Regression. As was the case with multiple logistic regression softmax regression outputs probabilities for multiple classes, but *unlike* multiple logistic regression the output predicts membership of a **single** class. As such, the output is a binary vector. If there are K possible classifications and m samples, we have a classification matrix $Y_{\bullet\bullet} \in \{0,1\}^{m\times K}$ where each classification vector for a sample $Y_{i\bullet}$ is of length K and has a (positive) bit in the position which represents the correct classification. This is known as *1-of-k* or *one-hot* encoding.

$$h_\theta(x) = p(\hat{y} = k|x,\theta) = \frac{e^{z_k}}{\sum_{k'=1}^{K} e^{z_{k'}}} \text{ for } k \in \{1,\ldots,K\} \tag{3.11}$$

$$\hat{y} = \underset{k\in\{1,\ldots,K\}}{\operatorname{argmax}} \frac{e^{z_k}}{\sum_{k'=1}^{K} e^{z_{k'}}} \tag{3.12}$$

Equation (3.11) shows the softmax hypothesis function. Similar to logistic regression, its input is a vector $z \in \mathbb{R}^K$, of real-valued numeric outputs from a linear function $f(x)$, where K is the number of possible classifications. The hypothesis function also has K outputs where each output $h_\theta(x)_k$ represents the probability that the classification at index k is positive, $p(\hat{y} = k|x,\theta)$.

The softmax function is a generalised version of the logistic sigmoid function. Each value z_k in z undergoes a transform e^{z_k} and is then normalised by the sum of all transformed elements $\sum_{k'=1}^{K} e^{z_{k'}}$. As all elements sum to 1, the output corresponds to a vector of mutually exclusive class probabilities $h_\theta(x) \in (0,1)^K$. Equation (3.12) then outputs an index $k \in \{1, \ldots, K\}$, of the element with the highest probability, corresponding to the predicted classification.

3.2.3.2 Cost Functions and Regularisation

Every hypothesis function requires a *cost function*, the measure which evaluates performance and is minimised to give more accurate predictions. The cost function, like the hypothesis function, is dependent on the type of classification required. For linear outputs, we employ the **mean squared error** cost; for binary probabilities as is the case in *logistic* and *multiple logistic regression*, we use the **cross entropy** cost; and for mutually exclusive multi-class classifications as with *softmax regression*, we employ the **negative log likelihood** cost. The procedure used to minimise the cost will be presented in §3.2.3.3.
Although we discuss cost functions in the context of regression, they are also used in more complex learning architectures and will therefore, be referred to in later sections. We represent the cost function as $J(\theta)$, the cost of all model parameters θ at a particular point in training.

$$J_{mse}(\theta) = \frac{1}{2} \cdot \frac{1}{|X|} \sum_{i=1}^{|X|} (h_\theta(x)_i - y_i)^2 \tag{3.13}$$

Mean Squared Error. Equation (3.13) shows the mean squared error cost function, used for evaluating real-valued numeric predictions. Its input is the given ground truth vector y and the predictions $\hat{y}$ output from the hypothesis function parameterised by θ. The output is a measure of how much error exists in the predictions made with model parameters θ. The error for a sample is given by the difference between the prediction and ground truth, $\hat{y} - y$. The difference is squared to ensure a positive value for the error of each prediction and finally the result is averaged over the number of predictions made with the current parameters, which is usually the number of samples in a data matrix or data batch $|X|$. The co-efficient of $\frac{1}{2}$ makes the derivative calculation easier as we will see in §3.2.3.3.

$$J_{ce}(\theta) = -p \log q = -\frac{1}{|X|} \sum_{i=1}^{|X|} y_i \log h_\theta(x_{i\bullet}) - (1 - y_i) \log(1 - h_\theta(x_{i\bullet})) \tag{3.14}$$

Cross Entropy Cost. Equation (3.14) measures the difference between two probability distributions: p, the 'true' distribution of *actual* target values and q the model distribution, or *predicted* probability outcomes. We measure the probability of an event occurring as well as the probability of the complementary situation, the event not occurring, therefore, $p \in \{y, 1 - y\}$ and $q \in \{h_\theta(x), 1 - h_\theta(x)\}$. When $y_i = 1$ we use the probability of the event occurring $h_\theta(x)$ and the next term is zero as $1 - 1$ is 0. Whereas when $y_i = 0$ we use the term $1 - h_\theta(x)$ as $1 - 0$ is 1. When minimised, cross entropy brings the model distribution closer to the actual distribution, improving the quality of predictions.

$$J_{nll}(\theta) = -\frac{1}{|X|} \sum_{i=1}^{|X|} \log h_\theta(x_{i\bullet})_c \tag{3.15}$$

Negative Log-Likelihood. In Equation (3.15), the hypothesis $h_\theta(x_{i\bullet})_c$, is equivalent to $P(\hat{y}_{ic}|x_{i\bullet}, \theta)$, the probability the model assigned to the *actual* class. Furthermore, when minimised, Equation (3.15) raises the probability of the actual class versus all other classes. By *minimising* the *negative* probability of the actual class, we are *raising* the probability the model assigns to the correct class. The likelihood of the model parameters, conditioned on the data matrix X (the classification conditioned on the input variables $(y|x)$ is equal to the probability of the data matrix conditioned on the model parameters, as shown in Equation (C.1) in Appendix C, which is the origin for this cost function. The cross-entropy cost is actually equivalent to negative log likelihood with only two classes.

Regularisation. A regularisation function computes a value which is added to a cost function, such as those introduced in Equations 3.13, 3.14 and 3.15. The purpose of a regularisation function is to incorporate *actual* model-parameter values into the cost and not just the result they give through a hypothesis function. The effect of regularisation is to encourage model parameters to take on smaller values by penalising those which are overly large. The result is a simpler model which is less likely to overfit the training data. Overfitting occurs when models perform very well on data they are trained on but do not generalise well with unseen instances.

$$R(\theta) = \sum_{i=1}^{n} |\theta_i|^p \tag{3.16}$$

Equation (3.16) shows the general form for a regularisation function. In effect, the regularisation function sums the absolute values for all model parameters, raised to the power p and adds the result to the cost function. The parameters p and λ are hyper-parameters where p refers to the type of regularisation employed and λ is the strength of the regularisation, or how much large parameters are penalised. Generally, the value chosen for p is 1 or 2, giving L1 or L2 regularisation, respectively. For regression, θ is a single instance of weights and biases, but for neural networks θ encompasses the model parameters for each layer $\{\theta^{(1)}, \ldots, \theta^{(|L|)}\}$

3.2.3.3 Parameter Updates

Once we have instantiated parameters for a model, defined a hypothesis function to make predictions and have a method to determine the cost of these predictions, a mechanism is still required to compute the *value* which we use to update model parameters and lower the cost. The **derivative function**, which computes the derivative of a cost function with respect to model parameters, gives this value.

For each cost function in §3.2.3.2, we explore the relevant derivative. For clarity and simplicity, we refer to model parameters here in their combined form θ. The derivatives explored here form the foundation, and will be used for the explanation of more complex derivatives required in the update functions of the neural networks in §3.3. We use derivatives in the regression optimisation process, which minimises the cost to achieve better accuracy. This update process outlined in Equation (3.21), can be generalised and is used for **all** learning algorithms employed in this dissertation.

Derivative of Mean Square Error Cost. Equation (3.17) shows the partial derivative of $J_{mse}(\theta)$ with respect to the model parameters. We provide a full derivation in Appendix D, Equation D.7. The derivative is essentially an error signal (the hypothesis output minus the ground truth) used to inform updates. This error signal is then combined with the input to update model parameters.

$$\begin{aligned}\frac{\partial J_{mse}(\theta)}{\partial \theta} &= \frac{\partial \frac{1}{2}(h_\theta(x) - y)^2}{\partial \theta} \\ &= (h_\theta(x) - y)x\end{aligned} \tag{3.17}$$

Derivative of Cross Entropy Cost. Equation (3.18) shows the expansion, via the chain rule, of the intermittent functions in the derivative of the cross entropy cost with respect to model parameters. The chain rule, described in Appendix D.1.1, allows us to expand complex derivatives required to improve hypothesis outputs and are crucial to complicated neural networks. We do not provide derivations of the individual terms here, but refer the reader to Appendix D. The term on the left of Equation (3.18) expands into three on the right. The first, derived in Equation (D.8), is the derivative of the cross entropy cost (Equation (3.14)) with respect to the output of the logistic function, a. This is multiplied by the second, which we derive in Equation (D.4), the derivative of the logistic function (Equation (3.4)) with respect to the output of the linear function, z. Finally, these are multiplied by the derivative of the linear function with respect to the model parameters, which we calculate in Equation (D.3).

$$\frac{\partial J_{ce}(\theta)}{\partial \theta} = \frac{\partial J(\theta)}{\partial a} \cdot \frac{\partial g(z)}{\partial z} \cdot \frac{\partial f(x)}{\partial \theta} \tag{3.18}$$

Equation (3.19) shows the substitution of the terms derived in Equations (D.8), (D.4) and (D.3) for those in Equation (3.19), before they are simplified and presented in the final line, which shows the result in an alternative formulation as $a = h_\theta(x)$, the hypothesis for logistic regression.

$$\begin{aligned}\frac{\partial J_{ce}(\theta)}{\partial \theta} &= \frac{(a-y)}{a(1-a)} \cdot a(1-a) \cdot x \\ &= (h_\theta(x) - y)x\end{aligned} \tag{3.19}$$

Derivative of Negative Log Likelihood. In the case of softmax regression, we take the same approach as with the cross entropy derivative, but the chain rule expansion takes a slightly more complicated form. There are multiple output probabilities, but the cost depends solely on the probability of the correct classification. Therefore, we have to calculate the derivative of the cost of the correct classification probability, but with respect to the parameters for *all* outputted probabilities. Equation (3.20) shows the correct classification as index k and all other parameters indexed as i, as well as the final result. As the result ends up the same as Equation (3.19), we will not perform a detailed derivation here but refer the reader to Equation (D.10), in Appendix D.

$$\begin{aligned}\frac{\partial J_{nll}(\theta)}{\partial \theta} &= \frac{\partial J(\theta)}{\partial a_k} \cdot \frac{\partial a_k}{\partial z_i} \cdot \frac{\partial z_i}{\partial \theta_i} \\ &= (h_\theta(x) - y)x\end{aligned} \tag{3.20}$$

The important thing to note here is that although there are different derivatives, with varying degrees of complexity for the different cost functions, the result always ends up the **same**, an important point for the explanations to come in §3.3.

Update Function. Given the calculated gradient, we can now define the parameter updates. Equation (3.21) shows how we update parameters for a regression model.

$$\theta := \theta - \alpha \frac{\partial J(\theta)}{\partial \theta} \tag{3.21}$$

The update procedure is the same for **every other** learning algorithm described in this dissertation. The procedure takes each model-parameter (weight or bias) and replaces each parameter simultaneously, with the value attained by subtracting the relevant cost function derivatives multiplied by a *learning rate* α. The learning rate is a hyper-parameter which defines the magnitude of weight updates, informing us how much to lower the cost function for each iteration.

3.2.3.4 Parameter Optimisation

There are many methods to optimise model parameters but in this dissertation we will focus on gradient descent. The optimisation process performs many iterations of model parameter updates to minimise a cost function and thus, improve the quality of predictions. The procedure for basic gradient descent can be seen in Algorithm 1 and is repeated until the model converges on the lowest cost possible.

Algorithm 1 Basic Gradient Descent Algorithm

```
1: while cost $\neq$ minimum do
2:     Compute the hypothesis given the input $h_\theta(x)$
3:     Calculate the cost of the current parameters $J(\theta)$
4:     for $\theta^{(i)}$ in $\theta$ do
5:         Determine the gradient $\frac{\partial J(\theta)}{\partial \theta}$
6:     end for
7:     Update all parameters in parallel $\theta := \theta - \alpha \frac{\partial J(\theta)}{\partial \theta}$
8: end while
```

There are three types of gradient descent: batch Gradient Descent (GD) uses the entire dataset for each update; Stochastic Gradient Descent (SGD) uses a single data sample for each parameter update; and Mini-batch Stochastic Gradient Descent (MSGD) uses a subset or *batch* of the data to update parameters. All procedures ultimately achieve a better model accuracy.

Throughout this dissertation, we focus on **mini-batch stochastic gradient descent** (MSGD) [23], with an additional process called **early-stopping** [112]. A data matrix is first split into training, validation and test sets X^t, X^v, X^s, which are disjoint subsets of the data matrix X. For each iteration of optimisation, we calculate the hypothesis for each sample in a small *batch* (subset of samples), get the mean cost for this batch and calculate the relevant derivative so the update defined in Equation (3.21) can be performed, until the lowest cost is achieved.

Early stopping introduces another hyper-parameter called **patience**, through which MSGD can terminate early in order to prevent over-fitting. It is the minimum number of iterations that will be performed before exiting the optimisation process.

Early-stopping also requires testing the model at regular intervals on a separate, 'held-out' validation set, X^v. If performance on the validation set keeps improving over a certain threshold, the patience is increased and the optimisation process continues.

MSGD is computationally more efficient than performing updates based on the entire dataset (batch GD). In non-convex, non-smooth error functions, it also enables the optimisation function to escape poor local minima (which leads to poor accuracy) and find better minima to enable improved accuracy. Early stopping helps to avoid over-fitting by adding regular evaluation of the model's performance on held-out validation data. It terminates the procedure once the validation performance stops improving. One cannot test the predictive power of a model using the data used to train the model as it will result in an upwardly biased estimate of accuracy. The patience parameter ensures the optimisation procedure exits when performance stops improving on the held-out validation set.

3.3 Neural Network Types

In this section, we describe the four types of shallow and deep neural networks used throughout the dissertation. The descriptions which follow build upon the components outlined in the previous section and will therefore, refer heavily to the content and equations of §3.2.

3.3.1 Multi-Layer Perceptron

A *Multi-Layer Perceptron* (MLP) is a form of *supervised* feed-forward neural network. Its purpose is to make classifications or predictions when given input data. A secondary purpose, is to learn a layer of *abstract*, higher level features in its hidden layer, to better represent the data, determine variable interactions and give more accurate predictions. Although its architecture can be extended to include multiple hidden layers, the learning signal tends to vanish in deeper architectures. In this dissertation, we define a Multi-Layer Perceptron as a feed-forward neural network with **one** hidden layer.

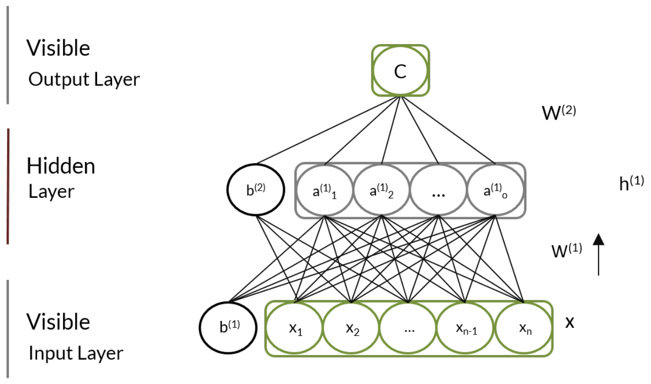


Figure 3.9: Single Class Multi-Layer Perceptron

Figure 3.9 shows a representation of a Multi-Layer Perceptron with a single output node, corresponding to a linear or binary classification task. It is also possible to have an MLP with multiple output nodes, which would be the case for multi-class classification. It is composed of a *hidden* layer and an output *regression* layer, combining the two and incorporating their functionality. Its functionality therefore, is essentially described as a combination of a feed-forward hidden layer from §3.2.1 and a regression output layer from §3.2.3. Parameter initialisation occurs as described in §3.2 for the visible and hidden layers.

3.3.1.1 Hypothesis and Cost Functions

Equation (3.22) shows the Multi-Layer Perceptron hypothesis function. The input x propagates first through the hidden layer transformation $g_1 \colon x \mapsto g_1(x) = a^{(1)}$ and is then input into an output regression layer, which holds the classification function $g_2 \colon a^{(1)} \mapsto g_2(a^{(1)}) = a^{(2)}$, where $a^{(2)}$ is the output classification.

$$\begin{aligned} h_\theta(x) &= a^{(2)} \\ &= g_2(f_2(a^{(1)})) \\ &= g_2(f_2(g_1(f_1(x)))) \end{aligned} \tag{3.22}$$

As described in §3.2.1, the hidden layer transformation can be solely linear $f_1 \colon x \mapsto f_1(x) = z^{(1)}$ using $\theta^{(1)}$, as defined in Equation (3.3), or combined with a non-linear function $g_1\colon z^{(1)} \mapsto g_1(z^{(1)}) = a^{(1)}$. In our case either the sigmoid function (from Equation (3.4)) or the ReLU (from Equation (3.5)) are employed. The purpose of these hidden layer functions is to learn the abstract feature representations to better model input and output relationships. The linear function allows input features to be weighted whereas the non-linear function allows more complex representations to be learnt. For fully connected hidden layers in general, each node in the hidden layer $L^{(1)}$, is connected to every node in layer $L^{(0)}$, and every node in $L^{(2)}$. Each activation energy a is passed through these connections.

The *activation energies* of the hidden layer $a^{(1)}$ are then sent to the output regression layer where values propagate through g_2 to output predictions, as described in §3.2.3. Classification dependent, this is a solely linear transform $f_2\colon a^{(1)} \mapsto f_2(a^{(1)}) = z^{(2)}$ (Equation (3.3)), using $\theta^{(2)}$, for real-valued classification tasks; or combined with the logistic (Equation (3.4)) or the softmax functions (Equation (3.11)) for binary and multi-class tasks respectively.

The **cost function** employed for a Multi-Layer Perceptron is dependent on the classification being performed, as described in §3.2.3. Furthermore, the *same* cost functions introduced in Equations 3.13, 3.14 and 3.15 are used and for the same classification tasks. The *difference* is the more complicated MLP hypothesis, defined in 3.22, replaces that defined for Regression when calculating relevant costs.

3.3.1.2 Updates and Optimisation - Error Backpropagation

In order to calculate the gradients which will be used to update the model parameters, a process called *error back-propagation* is required. The parameters of each layer must be updated with respect to the cost of the hypothesis output. This is performed by *back-propagating* what we call the *error gradient* from the top *output* layer through to the *hidden* and *input* layers. Essentially, we find the error between the hypothesis output and the ground truth, then back-propagate this *error* through the layers to determine the performance of the hidden features and update network parameters accordingly, in order to improve classification performance.

To update MLP parameters, we are required to compute two forms of gradient values. Equation 3.23 shows the first, the gradient of the cost with respect to the

output layer parameters; and Equation 3.24 shows the second, the gradient of the cost with respect to the **hidden layer parameters**. The chain rule (Appendix D.1.1) is again employed for easier gradient computation, as it was in §3.2.3.3.

$$\begin{aligned}\frac{\partial J(\theta)}{\partial \theta^{(2)}} &= \frac{\partial J(\theta)}{\partial z^{(2)}} \cdot \frac{\partial f_2(a^{(1)})}{\partial \theta^{(2)}} \\ &= \frac{\partial J(\theta)}{\partial a^{(2)}} \cdot \frac{\partial g_2(z^{(2)})}{\partial z^{(2)}} \cdot \frac{\partial f_2(a^{(1)})}{\partial \theta^{(2)}}\end{aligned} \tag{3.23}$$

Error Gradient. We can see the first two lines of Equations (3.23) and (3.24) are the same, save for the layer index. Thus we define the *error gradient.*

$$\begin{aligned}\frac{\partial J(\theta)}{\partial \theta^{(1)}} &= \frac{\partial J(\theta)}{\partial z^{(1)}} \cdot \frac{\partial f_1(x)}{\partial \theta^{(1)}} \\ &= \frac{\partial J(\theta)}{\partial z^{(2)}} \cdot \frac{\partial f_2(a^{(1)})}{\partial a^{(1)}} \cdot \frac{\partial g_1(z^{(1)})}{\partial z^{(1)}} \cdot \frac{\partial f_1(x)}{\partial \theta^{(1)}}\end{aligned} \tag{3.24}$$

Equation (3.25) defines error-gradient $\delta^{(l)}$, which represents the derivative of the cost with respect to the linear activations z of the lth layer. It carries the *error signal* we *back-propagate* through the layers, allowing us to update model-parameters to improve classifications. This term is important as it allows us to express the value for the layer gradient in a simple *general* form for this and more complicated algorithms.

$$\delta^{(l)} := \frac{\partial J(\theta)}{\partial z^{(l)}} \tag{3.25}$$

Layer Gradient General Form. If we substitute δ from Equation (3.25) into the first lines of Equations (3.27) and (3.28), it gives us a convenient, general form for a layer's gradient, shown in Equation (3.26). This formula allows us to calculate the values used to update *any* feed-forward neural network layer parameters through *error back-propagation.* This formula will be used again for algorithms in §3.3.2 and §3.3.4 and again in Chapter 7 for our experiments.

$$\frac{\partial J(\theta)}{\partial \theta^{(l)}} = \delta^{(l)} \cdot a^{(l-1)} \tag{3.26}$$

Output Layer Gradient. As we mentioned at the beginning of the section, the *error-gradient* has a **different** form for an output layer $L^{(|L|)}$ and an inner, hidden layer $L^{(l)}$. The output *error gradient* $\delta^{(|L|)}$ for a Multi-Layer Perceptron is shown in Equation (3.27). It is the difference between the hypothesis and the ground truth $h_\theta(x) - y$. This calculation was shown in Equations (3.17), (3.19), (3.20) where different cost functions were shown to produce the same gradient. The error gradient δ is then multiplied by the previous layer activations to get the *layer gradient.* This formula can be generalised to the output layer of *any* neural network which uses mean square error, cross entropy or negative log likelihood loss if we replace the output layer index (2) with ($|L|$) and the hidden layer index (1) with (l).

$$\frac{\partial J(\theta)}{\partial \theta^{(2)}} = \delta^{(2)} \cdot a^{(1)} = (h_\theta(x) - y) \cdot a^{(1)} \tag{3.27}$$

Hidden Layer Gradient. In Equation (3.28), we can see how errors are *back-propagated* through to hidden layers. To calculate the hidden layer $\delta^{(1)}$, we require the error gradient $\delta^{(l+1)}$ of the next layer, which in this case $\delta^{(2)}$, is the output gradient. As can be seen in Equation (3.28), $\delta^{(1)}$, is the dot product of $\delta^{(2)}$ and the transpose of the parameters $\theta^{T(2)}$, whose result is multiplied element-wise with the derivative of the layers activation function. This formula can be generalised to any feed-forward neural network hidden layer if we replace the layer index (1) with (l) and the layer index (2) with ($l + 1$), as shown in Equation (3.29).

$$\begin{aligned}\frac{\partial J(\theta)}{\partial \theta^{(1)}} &= \delta^{(1)} \cdot x \\ &= (\delta^{(2)} \theta^{T(2)}) * g_1'(z^{(1)}) x \end{aligned} \tag{3.28}$$

The power of these expression can be fully realised when presented with deep architectures such as those in §3.3.2 and 3.3.4 as they give a uniform way to approach the updates of each hidden layer in much deeper architectures.

$$\delta^{(l)} = \frac{\partial J(\theta)}{\partial z^{(l)}} = (\delta^{(l+1)} \cdot \theta^{T(l+1)}) \cdot g'(z^{(l)}) \tag{3.29}$$

The same update function, defined in Equation (3.21), is again used for the MLP. However, where there was only one parameter to update for the Regression learner, the process iterates through all parameters for each layer in the MLP, where the parameters are $\theta = \{\theta^{(1)}, \ldots, \theta^{|L|}\}$. The same optimisation process of MSGD, described in §3.2.3.4, is again used as each parameter must be updated in parallel.

3.3.2 Recurrent Neural Network

A Recurrent Neural Network [53] is a form of supervised neural network which can accommodate sequential data. Unlike a Multi-Layer Perceptron, which can be described as a combination of a feed-forward hidden layer and a regression layer, a Recurrent Neural Network is a combination of a *recurrent hidden layer*, outlined in §3.2.2 and an *output regression layer*, as shown in §3.2.3. A Recurrent Neural Network is used in place of a Multi-Layer Perceptron when the dataset is sequential in nature. By sequential, we mean time-based or where the context of a particular data sample has meaning. For example, the sports performance dataset used for evaluation purposes in Chapter 7 uses heart rate data which is time-based and prior heart rate values have an effect on current heart rate values. The Recurrent Neural Network is also the first *deep* neural network. Although it is not deep in the traditional sense of number of hidden layers, it is deep if unfolded over time, as described in §3.2.2. As all processes to describe a recurrent neural net have been described in sections 3.2.2, 3.2.3 and 3.3.1, we will only briefly cover these topics here and describe how they are used in the Recurrent Neural Network paradigm.

Parameter initialisation occurs in the same fashion as a Multi-Layer Perceptron where the hidden layer parameters are initialised using Equations (3.1) and (3.2) and regression parameters are initialised to zero as described in §3.2.3. The **hypothesis function** takes the same form as the Multi-Layer Perceptron, described in §3.3.1.1

except the linear function used in the Multi-Layer Perceptron is replaced with the recurrent linear function defined in Equation (3.8), for the hidden layer. For the non-linear functions, we employed both the sigmoid (defined in Equation (3.4) and the ReLU (defined in Equation (3.5)), in different experiments. The regression layer operates in the same way as described in §3.2.3 except instead of the regression layer receiving input from the dataset, it receives its input from the recurrent layer. The **cost functions** are again dependent on the type of classification being performed and regularisation can be added as per §3.2.3.2, with the difference being there are now multiple sets of model parameters $\theta = \{\theta^{(1)}, \ldots, \theta^{(|L|)}\}$.
Another difference between the Recurrent Neural Network and the Multi-Layer Perceptron is the **derivatives**. Instead of regular back-propagation, a process known as *back-propagation through time* is required. This is very similar to the regular back-propagation process, with all necessary formulae defined previously in §3.3.1.2. The difference is besides the *input-to-hidden* and *hidden-to-out* weight matrices, we must update the recurrent hidden layer parameters. Although these gradients are calculated as normal hidden layer gradients using Equation (3.26), the *error-gradient* must be passed back as to the point in time where we *truncate* the gradient, and possibly to the beginning of the sequence (t = 0). For example, if we are calculating the cost of the input at time $t = t$ we use the hidden states from $t = \{0, \ldots, t-1\}$. Therefore, the *error gradient* is passed back from time t through $t-1, t-2$ and possibly as far back as $t = 0$, in order to adjust the hidden-to-hidden parameters to better classify at current time-point t. Finally, the same process for **parameter updates** and **optimisation** is used. This process effectively adjusts what is learning in the 'memory' of the Recurrent Neural Network and allows it to better use past time points to reason about and classify current time-points.

3.3.3 Restricted Boltzmann Machine

A Restricted Boltzmann Machine (RBM) [58] is a two layer neural network, with one hidden and one visible layer. It is the only *unsupervised* algorithm discussed in this dissertation. Unsupervised algorithms do not learn models which classify data but which may, for example, extract patterns, like clusters, from the data. The aim is for nodes in the hidden layer to learn abstract features which better describe the data and allow us to perform analyses where classifications do not exist. For example, to determine how input features cluster and combine in the hidden nodes, input feature interactions can be extracted, which is an analysis we perform on the Bach dataset in our evaluation. The Restricted Boltzmann Machine also provides a means to learn the abstract features which further aid in prediction for deep learning applications.

Like Regression, a Restricted Boltzmann Machine extends the concept of a hidden layer. However, unlike Regression and other learning algorithms we have described, a Restricted Boltzmann Machine is an **energy-based** learning algorithm. *Energy-based models*, through an **energy function**, associate a real-valued, scalar energy value with each variable configuration (sample) in a dataset [12]. In this research, we cover two types of Restricted Boltzmann Machines:

- **Bernoulli Restricted Boltzmann Machines** (BRBM): for binary input nodes (variables).
- **Gaussian-Bernoulli Restricted Botlzmann Machines** (GRBM): for real-valued, Gaussian input nodes.

Input variables to both types of Restricted Boltzmann Machine must be homogeneous because of the requirements of the *energy function* which will be discussed in §3.3.3.1. Two types of RBM are required, to handle two possible types of input variables. The BRBM can only handle binary input features. This is sufficient for binary and categorical data but information is lost if we bin (make categorical), continuous data. Therefore, the use of the Gaussian RBM is also motivated, as it can handle continuous Gaussian input. Although the BRBM and the GBRBM have different types of visible nodes, corresponding to the different types of input variables, both have *binary hidden units.* **Parameter initialisation** is carried out according to the procedure outlined in §3.2.1.1.

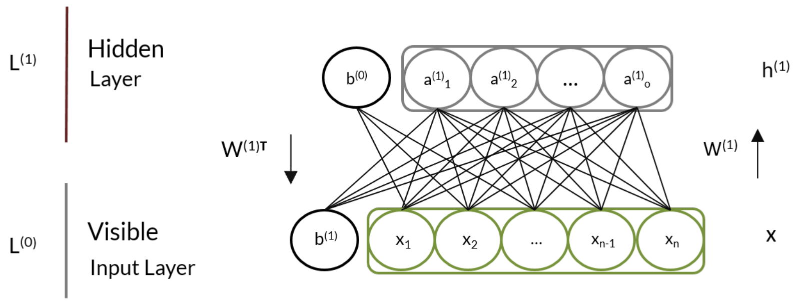


Figure 3.10: Restricted Boltzmann Machine

3.3.3.1 Hypothesis Function

The concept of an *energy function* is taken from particle physics, where stable and therefore, desirable configurations of particles occupy a *low energy* state and are more probable [12]. For example, liquid water is more stable than steam, occurs more often in nature (more probable) and has less energy (heat). Therefore, the energy function for water would output a higher value for steam than water.

Applying this to data, the aim is to learn an energy function which outputs a low energy for desirable (probable) configurations of the data and a high energy for unusual configurations. The following is a theoretical example in the context of the sports performance dataset introduced in Chapter 1, which we will refer to here as the GAA dataset. Assume there is a player who is moderately active throughout a match but is only in contact with the ball once, for a very short period. During this period, they increase their athletic work-rate dramatically. The energy function for the player, at this point in the match, *should* output a much *higher* energy than at other points, as the biometric sensors receive values which are far *less probable* for the player in the context of this match.

In both the BRBM and GBRBM, we express the energy of the models in terms of their **free energy**. The *free energy* allows us to express the energy of a configuration in terms of the visible nodes alone, leading to an easier expression of the cost function as we marginalise the hidden variables. The *total energy* functions $E(x, h)$ for both algorithms and the relationship between the total energy and free energy is shown in Appendix B.3. We define the **free energy** function of a Restricted Boltzmann Machine to be its *hypothesis function*.

$$\mathcal{F}_\theta(x) = -\sum_{i=1}^{|x|} x_i b_i^{(0)} - \sum_{j=1}^{|h^{(1)}|} \log(1 + e^{z_j^{(1)}}) \tag{3.30}$$

$$\mathcal{F}_\theta(x) = \sum_{i=1}^{|x|} \frac{(x_i - b_i^{(0)})^2}{2} - \sum_{j=1}^{|h^{(1)}|} \log(1 + e^{z_j^{(1)}}) \tag{3.31}$$

Equations (3.30) and (3.31) show the *free energy* function $\mathcal{F}\colon x \mapsto \mathcal{F}(x)$ of the **Bernoulli** and **Gaussian-Bernoulli** Restricted Boltzmann Machines, respectively. The input x to these function represent a data sample $x_{i\bullet}$, where the row index is omitted for simplicity. Therefore, in both equations x_i is the value for the ith feature in a visible input layer. The variable $z_j^{(1)}$ represents the linear activation of the jth hidden node. The bias term $b^{(0)}$ represents the *visible bias*, used when propagating values from the hidden to the visible layer. This parameter will be further utilised in the context of *Gibbs Sampling* in §3.3.3.3. Finally, the number of nodes in the hidden and visible layers are given by $|h^{(1)}|$ and $|x|$, respectively. There is no variance parameter σ in our formulation of the *Gaussian-Bernoulli* free energy, as we standardise input data to have zero mean and unit variance. Furthermore, we do not add Gaussian noise to the *reconstructions* (discussed later in §3.3.3.3). The *free energy* is the complementary value to the systems *total energy.* It is the amount of energy that is still available to the system for use. To revisit our earlier example, the player who only has one very short period of hard athletic activity will have a lot of free energy throughout the game and little during their period of increased work.

3.3.3.2 Cost Function

As before, the cost function computes a value which is used to evaluate the hypothesis function output. For the RBM monitoring cost function, we again use the negative log likelihood, but unlike the cost function presented previously in (3.15), we do not have a classification. Therefore, we must use an *apriori* probability $p(x)$, in place of a *conditional* probability $p(y|x)$. The *apriori* is a measure of the probability of observing an event x, out of all possible events, rather than the probability of obtaining a classification y, given an input sample x.

$$P(x_{i\bullet}) = \frac{e^{-\mathcal{F}(x_{i\bullet})}}{\sum\limits_{x} e^{-\mathcal{F}(x_{i\bullet})}} \tag{3.32}$$

Equation (3.32) shows the data sample *apriori* calculation for an RBM. The free energy for a sample $x_{i\bullet}$ is computed and divided by sum of all possible free energies. In practice, only the sum of free energies in the training set is computed. To again use the GAA analogy, we receive a data-sample $x_{i\bullet}$ for a particular point in the match and calculate the free energy. Equation 3.32, calculates the probability of observing this particular sample out of all other samples measured throughout the match.

$$J_{nll}(\theta) = -\frac{1}{|\mathcal{D}|}\sum_{i=1}^{|\mathcal{D}|} P(x_{i\bullet}) \tag{3.33}$$

Equation (3.33) shows the form of the negative log likelihood for the RBM. Minimising this measurement, raises the probability $P(x_{i\bullet})$ of a training sample $x_{i\bullet}$ compared to all other samples $x'_{i\bullet} \in X$. As the probability of the sample is raised, from Equation 3.32 we can see that the free energy will also be raised. Furthermore, as the free energy depends on the values in the visible layers and the marginalised hidden layer values, the joint probability distribution $P(x, h)$ is also learned, maximising the probability of the training data in the visible and hidden layers. Therefore, the more often the network sees a variable combination *or* a similar combination of variables, the more it will raise the free energy and lower the total energy of that type of configuration.

In relation to our GAA example, most measurements for a player with a low general work rate will be similar, showing low values for heart rate, acceleration, etc. Therefore, the RBM will receive inputs and calculate free energies resulting from these types of measurements often, raising their free energy. Abstract, hidden layer variables will also be learnt to reflect this. Therefore, if similar configurations are encountered, the result will be a high-free energy, whereas unusual, high-work rate configurations will produce a low free energy or high total energy. This satisfies the aim of the RBM where desirable configurations of the data produce a low total energy.

3.3.3.3 Updates and Optimisation - Contrastive Divergence

To train both types of Restricted Boltzmann Machine, we use n step **Contrastive Divergence**, CD_n. Contrastive divergence amounts to *gradient descent* with n steps of **Gibbs sampling** at each iteration of the gradient descent procedure. We use CD_1, *Contrastive Divergence* with 1 step of *Gibbs Sampling* at each iteration, instead of running the Gibbs chain to convergence. We use CD_1, as this is a standard approach in the literature and shown to have good performance [59]. To keep the gradient calculation *homogeneous* for both RBMs, we standardise continuous data to zero mean and unit variance and do not add Gaussian noise when sampling the visible nodes in the GRBM. Otherwise, we would require a different gradient function for the GRBM.

As we use gradient descent, we are again required to calculate the *gradient* - the partial derivative of the cost function (3.33) with respect to model parameters θ. This will be substituted into the update function presented in Equation (3.21) and used to optimise model parameters in gradient descent. Equation (3.34) shows the function to calculate the gradient for an RBM. *Contrastive Divergence* and *Gibbs sampling* are required to calculate this gradient.

$$\frac{\partial J_{nll}(\theta)}{\theta_{ij}} = \langle x_i h_j \rangle_{data} - \langle x'_i h'_j \rangle_{model} \tag{3.34}$$

Contrastive Divergence consists of two phases: the *positive phase* $\langle x_i h_j \rangle_{data}$ and the *negative phase* $\langle x'_i h'_j \rangle_{model}$. The angle brackets in Equation (3.34) denote expectations (means) of the distribution specified by the subscript that follows. Therefore, the positive phase refers to values in the visible and hidden layers generated by the *data* and the negative phase refers to values which are generated by the *model*, through Gibbs sampling.

In the **positive phase**, data is forward-propagated to calculate the probabilities of the hidden layer to give h. The linear function 3.3 and the sigmoid function 3.4 are used for this. The binary *states* are then sampled with Equation (3.7). This gives the values necessary for $\langle x_i h_j \rangle_{data}$ and is the mean of the outer product of x and h, the input data and the hidden states calculated from the input data, respectively. The positive phase relates to the *data itself* and the abstract features extracted *from the data.* For example, the positive phase gets a data sample from the GAA dataset and might extract a hidden feature like 'high work rate': '(yes/no)'.

In the **negative phase**, binary states are calculated for the hidden layer from the data x. Values (*not* gradients) are then back-propagated to the visible layer, with the functions (3.3) and (3.4). This gives a *reconstruction* x' of the data and is equivalent to *one* step of Gibbs sampling. The hidden h' states are then calculated by propagating the reconstruction x' forward. The visible reconstruction and the hidden states generated from it, give the variables for $\langle x'_i h'_j \rangle_{model}$. The negative phase is necessary to extract the *internal representation* the model holds about the data. For example, if the algorithm is trained on the player with a low general work rate, then the model will make the 'high work rate' abstract feature *fire* with a low probability and output low values for visible features such as *heart rate.* This becomes the probabilistic characteristics the model learns about a player.

The purpose of the two phases of Contrastive Divergence is to bring the models internal representation $P(x, h)$ closer to the actual distribution of variables. Therefore, the visible and hidden states generated from the data, as well as the visible and hidden states generated from the model, provide a means to update the parameters in gradient descent and better model the data in the visible and hidden layers. For example, in our experiments in Chapter 7, we have a set of input variables that are transformed into a completely new set of abstract variables in the hidden layer. The updating process carried out by the algorithm determines how inputs are combined in the hidden nodes by assigning and updating weights that combine inputs into the abstract features and thus learns the joint distribution.

It is important to note, that for the back-propagation process, the **transpose** of the weight matrix is used. When we use the transpose, $W_{ij} = W_{ji}^{\mathsf{T}}$, the same weight propagates the value from the ith visible node to the jth hidden node and back-propagates the value from jth hidden node to the ith visible node. The transpose ties the feed-forward and back-propagation weights, which is necessary to use the RBM learning algorithm. Furthermore, less parameters need to be learned and a form of regularisation occurs because the weights are constrained. Weights are tied for both types of Restricted Boltzmann Machines, but for value back-propagation we use different *reconstructed* visible states for each. This keeps the reconstructed *visible states* consistent with the variables that were inputted to each RBM. In the case of the GRBM, linear activation energies are used to give Gaussian visible unit reconstructions. For BRBMs, a Bernoulli state sample is taken for the visible units, from the logistic activation to give a reconstruction of Bernoulli input variables as was shown in Equation (3.7).

3.3.4 Deep Belief Network

A Deep Belief Network [60], [14] differs from the RBM, in that it contains *multiple* hidden layers. Furthermore, unlike the Multi-Layer Perceptron and recurrent neural network, it can be successfully trained with more than one hidden layer. This means that each subsequent layer learns a more abstract feature representation and increases the accuracy of the model. The accuracy does not increase indefinitely, but like the number of nodes in a hidden layer, the number of hidden layers becomes another hyper-parameter. A Deep Belief Network can be used for unsupervised, supervised problems, or a combination of the two for *semi-supervised* applications. For example there may be a class imbalance or not all samples are classified. All necessary mathematical functions were defined in previous sections and no new mathematical functions are required here. Parameter initialisation occurs according to the methodology outlined in §3.2.1.1, with cost functions already outlined in §3.2.3.2.

3.3.4.1 Optimisation

Training procedures and deep architectures are what characterise Deep Belief Networks. We have already mentioned that an arbitrary number of hidden layers are possible in the architecture, given enough data. Unsupervised **pre-training**, coupled with a supervised or unsupervised **fine-tuning** training procedure [60], [14]

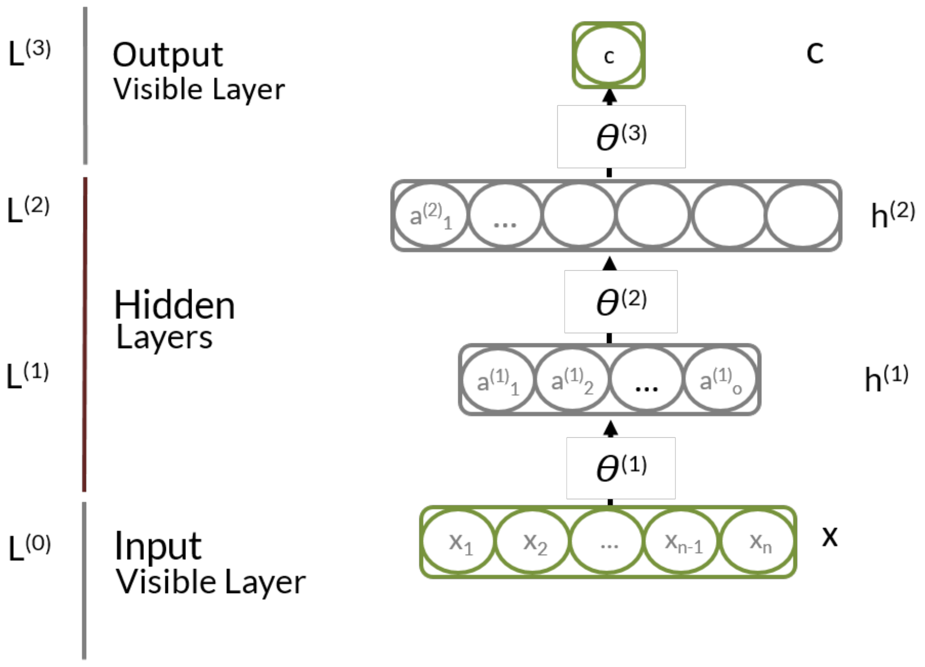


Figure 3.11: Deep Belief Network

ushered in the modern paradigm of deep learning.

We will first discuss *unsupervised-pre-training* which is the prerequisite for both *supervised fine-tuning* and *unsupervised fine-tuning*. We will give examples of unsupervised pre-training with the assumption that we will next be performing supervised fine-tuning, as this is a more intuitive way of explaining the optimisation process.

Unsupervised Layer-Wise Pre-Training. Unsupervised pre-training greedily optimises each hidden layer of a deep neural network, individually, as a Restricted Boltzmann Machine. The aim is to first fit the model parameters to the data alone, *without* respect to a specific classification or task. Therefore, when we then want to *fine-tune*, with respect to, for example, a classification, the parameters are already partially optimised. Thus, those parameters which optimise the network for a particular classification task are easier to find. Pre-training in effect, narrows the search space for optimising the weights for an outcome of interest and thus, provides an improvement to the learning process. It is similar to first finding clusters which describe the data very well and subsequently, using these clusters to train a classifier for increased predictive accuracy.

Figure 3.12 shows this process. First, $L^{(0)}$ and $L^{(1)}$ are treated as the visible and

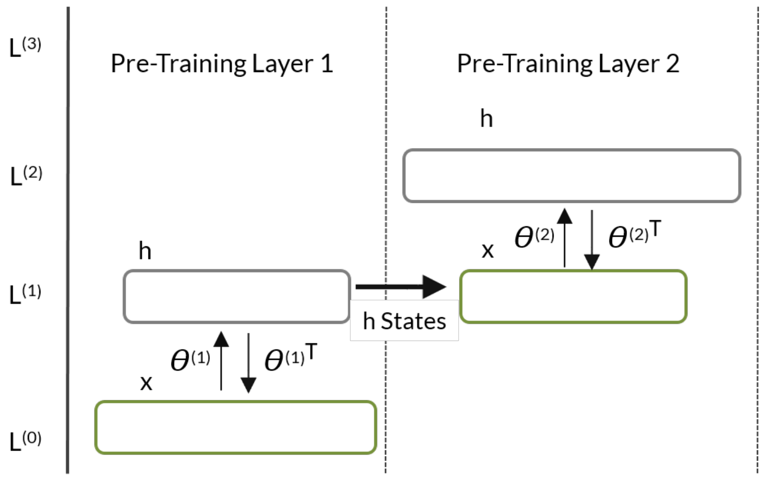


Figure 3.12: Pre-Training DBN Layers as RBMS

hidden layers respectively of an RBM and trained as such, according to the procedure outlined in §3.3.3. Once the first layer has been optimised as an RBM, the process proceeds to $L^{(2)}$, where $L^{(1)}$ is now treated as the *visible layer* and $L^{(2)}$ as the hidden. The *visible* states for $L^{(1)}$ are found by propagating dataset values from $L^{(0)}$ to $L^{(1)}$ with the already optimised $\theta^{(1)}$. This process is repeated as many times as there are hidden layers in the architecture. It is important to note that for each layer the RBM weights are *tied*, where for example, $\theta^{(1)}$ is used to calculate $L^{(1)}$ and $\theta^{(1)\mathsf{T}}$ is used to calculate $L^{(0)}$.

Fine-Tuning With Supervised Back-Propagation. When unsupervised pre-training has completed, *fine-tuning* with supervised back-propagation tunes the entire network with respect to a classification outcome. Once pre-training is complete, fine-tuning with back-propagation proceeds exactly as if we were optimising a Multi-Layer Perceptron, as described in §3.3.1. The parameters optimised in pre-training are used to instantiate the hidden layers of the MLP, then the relevant output *classification* layer is added to the top of the architecture and *all* weights and biases are then adjusted with standard back-propagation.

3.4 Summary

In this chapter, we presented a detailed discussion of neural networks. As part of that description, it was necessary to begin with basic concepts common to most machine learning approaches, before a full description of all of the components required to build a deep learning system. At that point, we were able to discuss the different forms of neural networks which led to a description of what comprises a deep learner. Our next step is to develop an architecture in which deep learners can be deployed. In the following chapter, we describe our methodology for building deep learning experiments and the architecture we designed to enable these experiments.

Chapter 4

Methodology and Architecture

Chapter 3 illustrated the wide range of components that comprise a deep learning experiment. It also demonstrated the significant complexity involved in both the construction of deep learning experiments and the analysis of results. The main goal of this chapter is to present a high level overview of an architecture developed to manage the deep learning experiment. Thus, this chapter provides a landscape for the more detailed research which follows in subsequent chapters. In §4.1, we present an overall methodology for deep learning. This draws on elements from standard and pre-existing data mining approaches but also highlights the significant efforts required in the preparation of both input data and the experiment itself. Thus, this methodology satisfies requirements a2 and a3 presented in §2.4 as we have defined the *steps to perform a deep learning experiment* which is *linked to existing data mining processes* and was designed to be *simple*, *reusable* and *flexible.* In §4.2, we describe our system architecture in terms of the functionality of application layers and system components, wherein we satisfy requirement c3, as we design *atomic* components, each dedicated to a *separate concern* within the deep learning experiment process.

4.1 An Approach to Deep Learning

Deep learning is a relatively new field, rooted in artificial intelligence and neural network research. As such, deep learning experiments presented in the literature often lack a common approach, underlying data model, or basis for standard knowledge-extraction process models. The application of pre-existing data mining processes to the specialised field of deep learning could greatly benefit the presentation and understanding of experiments, as well as allowing easier integration of deep learning into business processes.

In this section, we present our methodology and data pipeline for data-mining with deep learning. Before detailing our extended version, we will briefly introduce some widely used and pre-validated methodologies, some of which we integrated and extended in our own architecture.

Generally, a data mining or machine learning experiment follows a certain procedure to extract *optimal models.* As such, industry and academia have gone to great lengths to agree upon a standard process for the extraction of knowledge from data. The CRoss Industry Standard Process for Data Mining (CRISP-DM), defined in 1999, is the current industry standard for data mining [30]. It is the most widely used framework [110] consisting of: *business understanding*, *data understanding*, *preparation*, *modelling*, *evaluation* and *deployment.* The SEMMA [6] process - Sample Explore Modify Model Assess - defined by the SAS Institute Ltd. for their enterprise mining software, is the next most used standardised model [110]. Both give a good general overview of the data mining process but are better suited to business applications. The problems presented here motivates both a more technical and explicit approach.

Two processes which go into greater technical detail are those presented in [57], and the data mining curriculum of the ACM SIGKDD (Association of Computing Machinery Special Interest Group on Knowledge Discovery in Databases) curriculum committee [29]. The method outlined in [57] builds on the *knowledge discovery in databases* process, first presented in [47]. These approaches mainly focus on: *data cleaning*, *integration*, *selection*, *transformation* and *mining*, followed by *pattern evaluation* and *knowledge presentation* [57]. The data mining curriculum is focused on: *database* and *data management* concerns, *data preprocessing*, *choice of model* and *statistical inference considerations*, *interestingness metrics*, *algorithmic complexity considerations*, *post-processing of discovered structure*, *visualisation* and *understandability*, *maintenance*, *updates*, and *model life cycle considerations* [29].

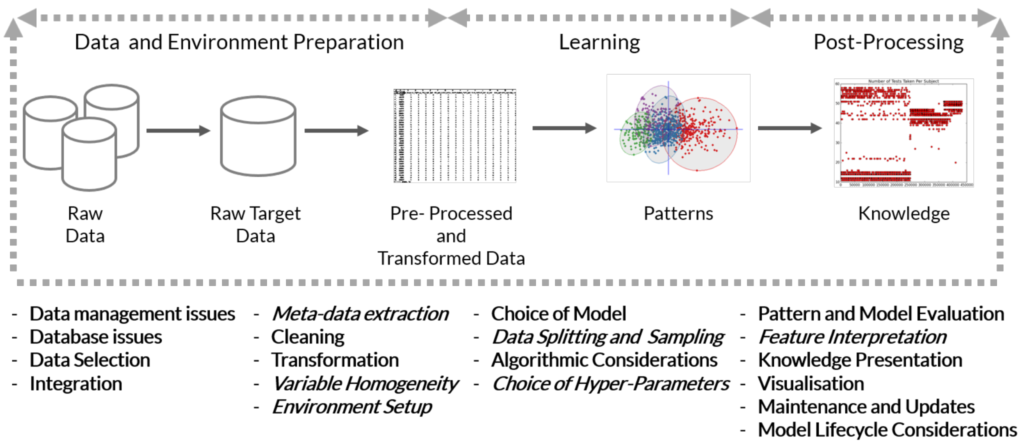


Figure 4.1: Data Mining Framework for Deep Learning Overview

Although more fine-grained, the concepts dealt with in these frameworks are still quite abstract in terms of their specification. This has led to the development of ontologies to elaborate upon more fine-grained issues. These ontologies include those that describe data mining and machine learning experiments [45] or learning algorithms and data mining concepts in more general terms [72].

High-level process models such as CRISP-DM are too general to account for the critical and more granular elements of deep learning experiments. Ontologies, although more fine-grained, are not fine-grained enough in some respects, as they often focus on *concepts* and their relations, rather than describing *data attributes*, which is at the level one needs to specify. Furthermore, ontologies are expensive to construct and require a significant learning curve for researchers. Consequently, we aim to address both high *and* low level issues with our approach, defining easy to understand processes and artifacts, which aid in performing and understanding a deep learning experiment.

Figure 4.1 shows our methodology, which first breaks data mining with deep learning into three, high-level processes. The first, *Preparation*, of the data and environment, will be detailed in §4.1.1. *Learning*, the next step, will be focus of §4.1.2. Finally, *Post-Processing* will be elaborated upon in §4.1.3. Figure 4.1 also shows that each of these high-level steps contain a series of lower-level elements, resulting in a hierarchy of experimental concerns. Each lower level concern will be addressed in the appropriate section.

Our core contributions can be found in the second and third steps, where we have developed two artifacts to address a deep learning experiment with even greater granularity: a software model to address architectural and procedural concerns; and a lightweight, flexible and interoperable data model to describe its data properties and their relations. These are the focus of Chapters 5 and 6, respectively. We have also designed an artifact to semi-automate the data transformation workflow for the *Preparation* step. The workflow does not form the core of our contribution and will be dealt with in §4.1.1 with a brief discussion.

The higher level steps of Preparation, Learning and Post-Processing were chosen to map almost directly to Data Preparation, Modelling and Evaluation steps of the CRISP-DM solution. As this is essentially an industry standard for data mining [30], [110], it allows us to integrate deep learning experiments seamlessly into existing workflows. The lower level concerns, shown below the graphic in Figure 4.1, were selected in order to form a representative superset of the elements outlined in [57] and [29]. The functions in italics are those we have added to extend the original frameworks. These were added as they are important aspects of deep learning experiments not explicitly represented in the original methodologies. Thus, we consider it necessary to highlight these components.

The concept behind each process is simple. In *Preparation* the knowledge worker selects, transforms and prepares the data so it can be directly input to a learning algorithm. They also configure the experimental environment, consisting of any software libraries and experiment management tools. In *Learning*, the knowledge worker takes input data and using the configured environment, builds the relevant learner model, which can be used to satisfy some pre-defined analytical goal. Finally in *Post-Processing*, they take the models and data generated during the Learning process and perform the required evaluations, interpretations and visualisations. The process is iterative, with each step repeated as required. Often, the result of one procedure can lead to a discovery which informs a subsequent step, leading to the possible repetition of particular steps, each time with greater knowledge.

For example, in the Post-Processing step, when learner models are analysed, this information can lead to the discovery of information, like better hyper-parameter settings, which could then be fed back into the *Learning* step in order to improve upon classification or description accuracy.

4.1.1 Preparation Stage

We now explore the first of three processes in our methodology, concerned with preparation of the experimental environment *and* the data. The outlined methodology first addresses **Database** and **Data Management Issues**, as well as **Data Selection** and **Integration** which converts *raw data* to *target data*, an example of which would be the extraction of data from an operational data store, to a data warehouse fact table or comma separated value file. These elements are extremely important when data mining. In fact, when combined with other preparatory concerns, this process has been shown to consume up to 80% of the time on a data mining task [141].

In deep learning literature experiments generally begin with *target data.* Therefore, as this research is mainly focused on the deep learning *experiment* aspect of data mining, we assume *target data* will be input and we explore those elements of our methodology concerned with extracting **knowledge** from this target data.

During this research, we identified a number of concerns when converting target data into a format suited to a deep learning algorithm. These elements are *crucial* in order to ensure the framework is domain agnostic (requirement C2). Principally, **Homogeneity of Input Variables** is often required, particularly with energy based models such as the Restricted Boltzmann Machine (RBM). In the context of the RBM, this refers to homogeneity of the data type - binary or continuous. However, with variants such as Auto-Encoders and Recurrent nets, this can refer to the data being on a homogeneous *scale*, where the data values of one feature do not span a significantly larger range than other features.

Variable homogeneity is achieve through *cleaning* missing data from the data set and *transforming* either the scale or type of each feature. Once features are transformed to the relevant scale or type, they can also be transformed, at a language data-

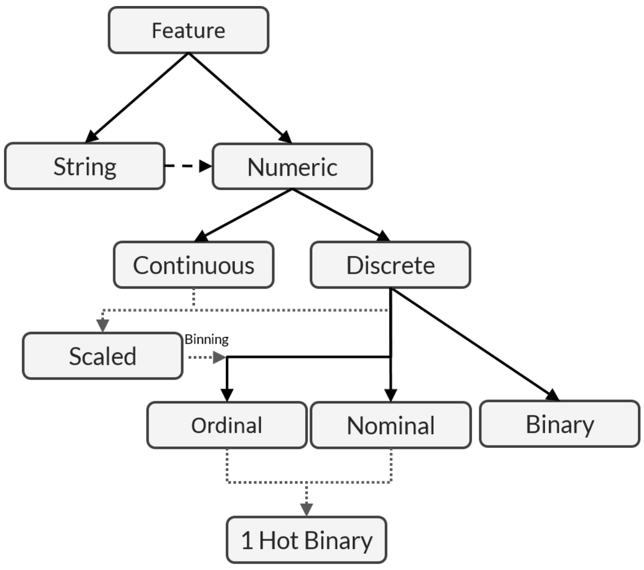


Figure 4.2: Feature Categorisations and Transformations

type level, in a float representation, to take advantage of the speed-ups graphical processing units accrue. It is not possible to clean or transform the data without first *extracting metadata*, which identifies missing values and extracts feature data types and scales.

For the reasons outlined above, metadata extraction is a core concern of deep learning experiments, as is determining the degree of variable homogeneity required. Thus, we added both elements to our methodology to highlight their significance.

Metadata Extraction. To detect missing values, we cycle through the data row-wise and column-wise, storing the indices containing missing, n/a or null values. Subsequently, for those rows and columns with missing values, we determine the percentage missing from both views. This metadata is then used to inform that imputation strategy to be used to impute the generate values. The imputation strategies employed will be described in the context of the experiments in Chapter 7.

Once missing data has been identified, we then extract metadata relating to the *type* of each feature. For this process we need to disambiguate **feature type** from **data type**, where feature type refers to whether a variable is continuous or discrete and *data type* as to its representation, whether it is a string, integer or float and so on.

Feature Transformation. Figure 4.2 shows the workflow we have developed to aid feature *type identification* and *transformation.* The workflow allows for a highly manual process to be semi-automated and further enables the process of making the framework *domain agnostic*

Feature transformation cannot be fully automated as different algorithms require differently formatted data and therefore, different transformations. This necessitates input from the knowledge worker at various points. Solid black arrows refer to categorisation steps and dashed arrows signify conversions, where arrows with large dashes are mandatory and arrows with small dashes are optional.

Figure 4.2 shows the input to the process is a *feature.* In practice, this is a *feature column* - all values the feature takes on in a data set. The first step shown in Figure 4.2 identifies and labels the *data-type* of each feature: string or numeric. We group floats and integers into a single category but do not discard this information. As the *Learning* process requires all inputs to be numeric, we convert categorical string variables to a numeric encoding and store this mapping or *encoding* as metadata. At this level, we also extract the number of *unique* values for each feature.

Proceeding down to the second and third levels in Figure 4.2, after determining the number of unique values contained in each feature column, all recorded metadata is used to assign a *candidate feature type* to the variable.

Table 4.1 shows the rule set for candidate label assignment. If a feature's data type is a float and the number of unique values contained in the column are greater than 50% of its total cardinality, then a candidate label of **Float** is assigned. There are two cases when *only* a label of **Discrete** is assigned. The first case is when the distinct cardinality of a feature is less than 50% and its data type is a float. The second case is when the variable's representation is an integer and its distinct cardinality is greater than 50% of its total size. If possible, more specific discrete variable categorisations are applied in certain cases. If the distinct cardinality is less than 50% of overall feature cardinality and the data value discovered is an integer *or* a string, then a candidate label of **Nominal/Ordinal** is applied. Finally, when the distinct feature cardinality is 2, then a label of **Binary** is applied.

Once the candidate labels have been assigned, input is required from the knowledge

Table 4.1: Candidate Labelling Assignment

Data Type	Number Unique Values	Candidate Label
Float	>50%	Continuous
Float	<50%	Discrete
Integer	>50%	Discrete
Integer	<50%	Nominal/Ordinal
String	<50%	Nominal/Ordinal
Any	2	Binary

worker. At this point, they need to decide, dependent on the algorithms and data, whether transformations like scaling is required for continuous and discrete variables or conversions like *binning*, which converts discrete variables into ordinal/nominal variables or further conversion such as *one-hot encoding.* The binning process divides the range (difference between minimum and maximum) of a variable into a number of separate *bins* or sub ranges. These bins are numbered and the value of each variable is assigned a number depending on which *bin* its value is contained in. These numbers are used as the new values, resulting in a ordinal variable. *One-hot* or *one-of-k* encoding is identical to the process described for converting classifications into binary vectors are converted for softmax regression layers in §3.2.3.1.

If we are using a Multilayer Perceptron, simple regression layer or Recurrent Neural Network, inputs can be mixed. For example, a mixture of (scaled) nominal, continuous and discrete variables can be used for these algorithms. For the Restricted Boltzmann Machine (RBM) and Deep Belief Network (DBN), inputs are forced to be homogeneous. Two distinct homogeneous types are possible for the RBM and DBN: **binary**, or **real-valued Gaussian** (continuous). In order to use a Gaussian-Binary RBM, *all* inputs must be continuous. If not, all inputs must be encoded as binary and the Binary RBM used. Therefore, continuous or discrete features must first be *binned*, to convert them into nominal variables and then *one-hot encoded* to result in binary inputs.

4.1.2 Learning Stage

The primary goals of this process are the choice of algorithm to fit the required analytical goal(s) and the optimisation of the algorithm.

Choice of Model refers first to utilising an algorithm that will satisfactorily complete your identified analytical requirements. This includes the identification of the optimal configuration of model parameters using multiple runs of algorithm training.

Data Splitting and Sampling is part of our extension to the Deep Learning process as it is significant in identifying the optimal run to inform the choice of model. A successful regime of data splitting, sampling, or a combination of the two, ensures that in identifying the optimal run, a parameter configuration which is *biased*, or *overfits* the data, is not chosen. Bias occurs when the weights form *too simple* a solution and therefore, cannot accurately classify data upon which it was trained *or* identify new instances. Overfitting, conversely, is when *too complicated* a solution is formed, which performs extremely well on the data upon which it was trained but does not generalise well to unseen data. A simple process such as splitting the data into training, validation and test sets, or sampling methods such as bootstrapping can vastly improve the quality of the models [27].

Algorithmic Considerations includes algorithmic parameter optimisation strategies, as well as run time and resource requirements encountered when deploying these algorithms. Within parameter optimisation strategies, there are two types of parameters for optimisation: hyper-parameters and model parameters. Hyper-parameters or meta parameters are those which are input into the training process whereas model parameters are those weights and biases resulting from the process. Within this consideration is the choice for model optimisation, for example, mini-batch stochastic gradient descent, which is the process used in our approach.

Choice of Hyper-Parameters is another extension included in our approach. Although it is a subset of *Algorithmic Considerations*, it is often overlooked in the literature and can *hugely* affect the quality of the models found [27].

These components of a deep learning methodology will be discussed in detail in Chapter 5, where we present our Configurable Deep Network. This component of our system was created to address each of the concerns outlined above, within the context of the *Learning* process for a particular deep learning algorithm.

4.1.3 Post-Processing Stage

The goal of this process is to *interpret*, *evaluate* and *present* those patterns, models and data generated during the *Learning* process, described in §4.1.2. Specifically, there are three inter-related evaluations and interpretations required: the analysis and interpretation of hyper-parameters; analysis and interpretation of model-parameters; and analysis of errors.

Pattern and Model Evaluation includes the analysis of the *error* or *cost* of the patterns found. For supervised tasks this amounts to the accuracy of classifications and predictions. For unsupervised tasks, it relates to how well the data is described, for example, the extent that clusters fit the data or how well inputs are reconstructed. This can also include the evaluation of weights and their stability in our extended approach.

Feature Interpretation is our only custom concern in Post-Processing. It relates to interpreting how, in deep architectures, the inputs combine and interact in the hidden nodes and how hidden nodes themselves combine and interact to form more abstract features in higher layers.

Knowledge Presentation and **Visualisation** refer to addressing how the insights generated from the evaluations and interpretations performed in *Post-Processing* of the patterns generated during *Learning* are presented. This is one of the most important concerns of the method, as without a clear, concise and interpretable presentation of what has been discovered, the information found is difficult to understand.

Model Life Cycle Considerations and **Maintenance and Updates** are the domain of the knowledge worker. These issues refer to ensuring that the knowledge and patterns which result from *Learning* and *Post-Processing*, stay relevant and do not suffer from concept drift. Concept drift is when the statistical properties of the data set or the concept being modelled change and the models or knowledge is wrong or no longer relevant.

Within our system *Post-Processing* is enabled by our deep learning experiment data-model and toolkit. Therefore, just as *Learning* will be explored further in Chapter 5 the above concerns of *Post-Processing* will be explored further in Chapter 6.

4.2 POL and the Deep Learning Architecture

We adopted a holistic approach covering both software and data engineering aspects to deep learning. Our system deploys a deep learning experiment in a modular fashion, where multiple architectural layers separate experiment and modelling functionality from data manipulation, storage and access logic. Figure 4.3 shows our system architecture where the design of each of these processes was driven by the data model which will be described in detail in Chapter 6. The architecture comprises **Interoperable Storage**, **Toolkit** and **Experiment** layers. Each component in the *Toolkit* layer, is linked to one or more of the goals highlighted in section §4.1. The *Interoperable Storage* layer supports these processes, providing experiment data persistence as well as a means for interoperable data and model exchange. The *Experiment Layer* then represents each of the high level processes the knowledge worker engages with throughout a deep learning experiment. Architectures such as that presented in Figure 4.3 are well-defined and common in areas such as enterprise computing and many business processes but now, we bring this approach to the design of deep learning experiments.

4.2.1 Interoperable Storage Layer

The *Interoperable Storage Layer* has multiple goals. The first goal is to capture *all* elements required to fully *build and reproduce* a deep learning experiment. Storing this data facilitates greater flexibility in the learning process and gives empirical support and transparency into certain experimental choices, such as hyper-parameter selection, a process that is often performed heuristically without capturing decisions or why they were made. The second goal is to capture these data with an *interoperable* and *lightweight* format which is *extensible* and *simple* to understand and use. This facilitates easy sharing of results for comparison, analysis and reuse, as well

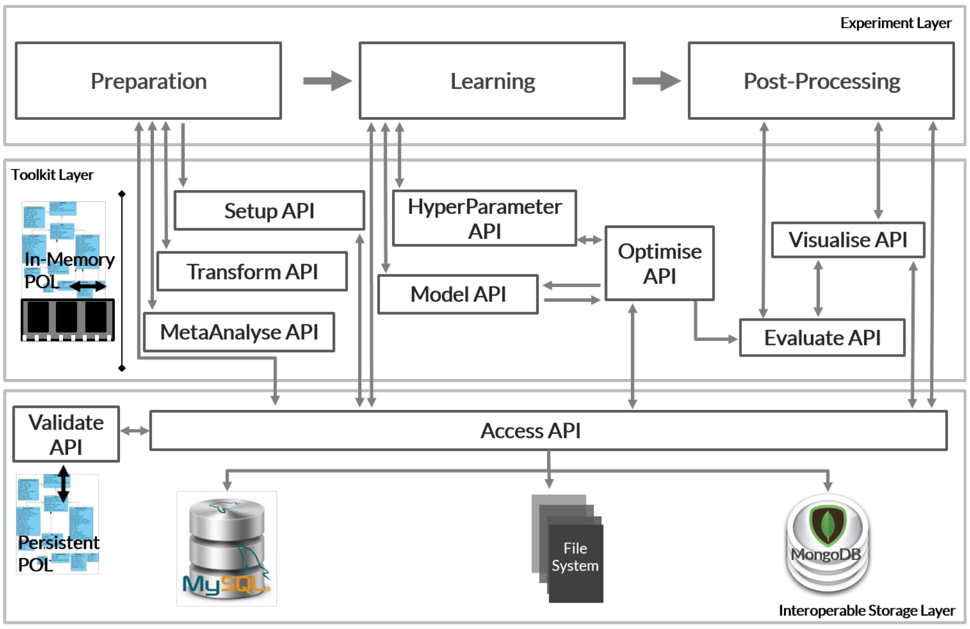


Figure 4.3: Deep Learning System Architecture

as allowing user customisation, but still providing a standard description of common and important elements of a deep learning experiment. Finally, each element or entity should be *atomic*, meaning if an experiment component or iteration was decoupled from the overall experiment, it would still have meaning.

In order to develop an interoperable API and storage mechanism to deliver the goals specified, we first required a conceptual data model to describe the constituent parts. This should enable a faithful repetition of the experiment but this time, allow the user to add or remove different components as needed. The underlying data model is called the *Parameter Optimisation for Learning* (POL) data model.

Once the POL data model was specified, a data description language and storage mechanism were required to provide a physical realisation of the POL. To achieve the goals of a simple, interoperable and extensible format, the POL data model was mapped to JavaScript Object Notation (JSON), with some of the higher level concepts pushed to the data storage mechanism. The *Access API* in the Interoperable Storage Layer abstracts the POL storage details from the higher level libraries and implementation of the POL data model, using JSON. This level contains all necessary functionality to read and write attribute values before, during and after a deep learning experiment.

The **Validate API** component ensures all data constructs conform to the POL model. Both the AccessAPI and ValidateAPI are required to read and write to the storage layer.

MongoDB was chosen for this implementation of our system as it provides a native JSON store and facilitates a direct mapping of POL data model elements to the NoSQL model (database). The architecture also shows MySQL in the system as the abstraction of storage through the API requires wrappers for any database selected for storage. The physical elements of the NoSQL model are replicated in memory, as Python dictionaries, which again provides a direct mapping to the JSON POL, allowing experiment data to be written to the database at intermittent and final stages. In Chapter 6, we provide a comprehensive description of the POL data model and the advantages of our design decisions.

4.2.2 Toolkit Layer

The *Toolkit Layer*, as presented in Figure 4.3 contains all necessary functionality to setup, run and analyse a deep learning experiment and is controlled by the Experiment Layer. All components use the Access API to read and write to Mongo. Eight component libraries were developed to cover different aspects of deep learning. Three components are required to prepare the experiment environment and data: *MetaAnalyse*, *Transform* and *Setup.* Three components enable construction and optimisation of learning algorithms: *HyperParameter*, *Model* and *Optimise.* Finally, two components: *Evaluate* and *Visualise*; are concerned with evaluation, interpreting and presenting results. They also aid in the optimisation process as these libraries contain a number of built-in evaluatation functions. The Setup, Access, Visualise, Validate and Evaluate components along with MongoDB are all part of the *Deep NoSQL Toolkit* which is described in Chapter 6. The remaining libraries comprise the *Configurable Deep Network*, which generates experiment data through building and optimising learning functions, and is covered in detail in Chapter 5.

The **MetaAnalyse API** and **Transform API** provide for the tasks of *meta-data extraction*, *cleaning* and*transformation* in our methodology. Using these components, we implemented the workflow shown in Figure 4.2 for variable type extraction and conversion. All rules shown in 4.1 are encoded in these libraries and allow a user to ensure *variable homogeneity*, where necessary.

The **Setup API** contains all necessary functions to configure and instantiate necessary POL constructs before an experiment is run, providing the *environment setup* aspect of our methodology. It first takes pre-processed and transformed data and converts it to an in-memory POL data model format by way of a shared Theano variable. It also generates the relevant hyper-parameter search space, basing the ranges of certain hyper-parameters on the metadata extracted from the input data. It then generates an experiment ID and using the Access API, it constructs an instance of the full POL data model in JSON and in Mongo to facilitate writing the hyper-parameter search space to the database.

The **HyperParameter API** enables hyper-parameter search and optimisation to provide *hyper-parameter choice.* Currently it can perform grid search [76] or random search [16]. As input, it takes the number of hyper-parameter trials to be performed and generates the appropriate number of hyper-parameter configurations.

The **Optimise API** interacts with both the Model and HyperParameter components. If called from the HyperParameterAPI, it iterates through each of the hyper-parameter configurations and for each iteration, splits and samples the data; makes a call to the ModelAPI for the appropriate algorithm; and finally, trains the model. Thus, this component relates to *data sampling and splitting* along with some aspects of *algorithmic considerations* and *hyper-parameter choice* of our methodology. It also separates the generation of hyper-parameters and learning models from their optimisation, allowing for different optimisation techniques to be added for both, without interfering with their construction. The OptimiseAPI uses the AccessAPI to store experiment *snapshots* after a number of *adjustable* stages of model parameter optimisation. It also stores snapshots once a set of model parameters have converged - reached the optimal solution possible - for a particular hyper-parameter configuration.

The **Model API** can be called directly from the experiment layer or from the OptimiseAPI, as part of the hyper-parameter optimisation process. It contains all appropriate modules (outlined in Chapter 3), to enable construction the of different deep networks with specified hyper-parameter configurations. This again calls the OptimiseAPI in order to optimise the model parameters with a procedure such as gradient descent.

The **Evaluation API** can be called from the OptimiseAPI, the VisualiseAPI or from the Experiment Layer. It contains functions to analyse and rank the performance of different trial-runs in the learning process, as well as the ability to analyse learned features. These functions refer to the *pattern and model evaluation*, *feature interpretation* and partially to the *knowledge presentation* concerns of our approach. If called from the OptimiseAPI, it automatically returns the top ranking hyper-parameter configuration or performs a multi-resolution search.

Finally the **Visualise API** interacts with the AccessAPI, EvaluateAPI and Experiment Layer, addressing the *knowledge presentation* and *visualisation* aspects of our approach and allows for visualisations such as learning curves, relating hidden features to inputs, and error graphs.

4.2.3 Experiment Layer

The *Experiment Layer* is the *application* layer in our architecture with a goal to provide the high level processes which an application performs in a deep learning experiment. The processes consist of experimental setup (Preparation); exploration and modelling (Learning) and evaluation (Post-Processing). It is possible to interact with POL data model functions from this layer by using the AccessAPI, thereby making use of the interoperability, storage and formal definition of deep learning experiments. If directly interacting with the JSON Access API, custom functions must be generated in place of the high-level component Python APIs provided in the Toolkit Layer for configuration and analysis.

4.3 Summary

The purpose of this chapter was to provide a high level outline of the process for deep learning experiments. By doing so, we describe the overall landscape for deep learning research and highlight those aspects where this dissertation provides a contribution. In §4.1, we presented our overall approach which included a discussion of key activities in deep learning research together with a description of the tasks involved in preparing data fur usage in experiments. Section §4.2 introduced our system architecture, which is built upon our data and software models and forms the core of our contribution. All toolkit libraries are developed using Python and Theano [9], [17] and are currently accessible using Python APIs. The design of these libraries is driven by the POL data model to provide higher level functionality for deep learning. Furthermore, they are designed to work seamlessly with the POL AccessAPI. The POL data model captures all elements of the deep learning experiment and will be covered in detail in Chapter 6. However, there are important functional aspects of the Learner that cannot be captured by a data model. Before our discussion of the POL data model, it is necessary to describe those functional aspects of deep learning which we argue must include both model and hyper parameter optimisation.

Chapter 5

Configuring Deep Learners

The previous chapter presented our system architecture and methodology for data mining with deep learning. This chapter presents the Configurable Deep Network (CDN), which relates to the *Learning* stage of our methodology. The Configurable Deep Network addresses requirement a1: the development of an *abstract (software) configuration approach* for deep network architectures, which is designed to be *generic* and *extensible*, thus enabling *simple* algorithm configuration. The CDN integrates model parameter *and* hyper-parameter optimisation (requirement c6) and is unlike other configurable approaches in that it explicitly provisions for and integrates a generic approach to hyper-parameter optimisation (requirement c6). In §5.1, we provide a definition of the generic *stages* of a deep learning experiment; in §5.3, we provide a deployment based description of the Configurable Deep Network, describing it in greater technical detail and relating it to the algorithms introduced in Chapter 3; and finally, in §5.4, we present a generic configuration process for optimising deep learners. The CDN is designed to be generic, simple, extensible and enable greater automation through automatically configuring learners according to a standardised hyper-parameter approach (requirement c1).

5.1 Configurable Stages

The aim of any learning algorithm is to find those parameters which achieve lowest cost for a hypothesis function. Updating *model parameters* alone is not sufficient.

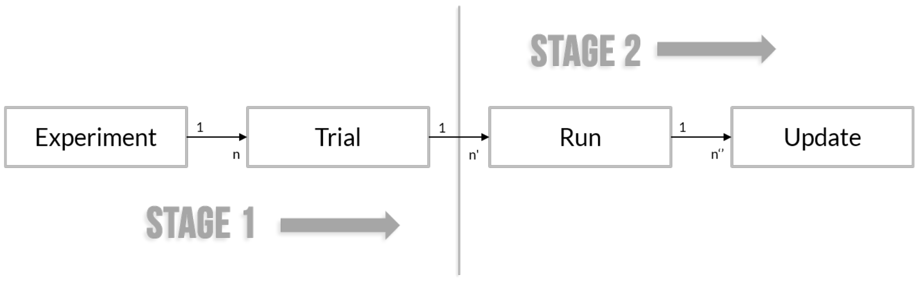


Figure 5.1: Experiment Stages Overview

To achieve optimum levels, *hyper-parameters* (also known as meta parameters) also require optimisation.
Hyper-parameters are those parameters not learnt during algorithm training but are inputs to the model parameter learning process. They influence how effectively model parameters are updated and ultimately, learnt. Hyper-parameter optimisation is often approached as a "black art" [16], meaning the optimisation of these parameters is not generally addressed in the literature, except in cases where hyper-parameter optimisation strategies are specifically explored [18], [121]. We aim to highlight hyper-parameter optimisation in our Configurable Deep Network approach, defining in this section two hierarchical, conceptual *stages* which are necessary in a deep learning experiment. In the remainder of this discussion, we will refer to the Configurable Deep Network as CDN.
Figure 5.1 gives an overview of the two hierarchical stages involved in a deep learning experiment. These stages form the basis of our CDN and its configuration process for optimisation. Figure 5.1 shows that in the first stage and at the highest level, we have an *Experiment*, for which there are n hyper-parameter configuration *Trials*. A hyper-parameter configuration is a single set of hyper-parameters, used for the instantiation and optimisation of a single deep learning algorithm instance. Proceeding to stage two, for each Trial there are then n' complete model optimisation *Runs* and for each run there are n'' *Updates*. A Run involves the complete process of instantiating a particular deep learning algorithm, with a particular set of hyper-parameters and *running* the required number of *updates* to these model parameters in order to achieve the lowest achievable cost for that particular hyper-parameter configuration.

In summary, the upper or *Trial* stage is concerned with **hyper-parameter** optimisation and Stage 2, or the *Run* stage addresses the lower level concerns pertaining to *Updates* for **model-parameter** optimisation.

5.2 Architecture and Constructs

This section gives an overview of the Configurable Deep Network and comprising components. It is a systems based software approach to deep learning experiments. Together with the generic process we will present in §5.4, it gives a modular framework which provides a means to easily construct deep architectures and couple them with *any* model parameter or hyper-parameter optimisation process.

The basis of this approach is an analysis and explicit definition of the high level functions and components common to deep neural networks. The definition of this functionality, coupled with a generic software approach, by means of a well established *deign pattern*, allowed us to develop a *learner factory*, a core component in our novel construct, the CDN. The adoption of the *factory design pattern* in our CDN was enabled by abstracting our approach to construction of deep learning algorithms, through these function definitions. Therefore, it is not specific to the type of network being optimised or the parameter optimisation processes involved, as long as the functions discussed here are present. The CDN encompasses the *Model*, *HyperParameter* and *Optimise* components outlined in the architecture in Chapter 4.

Figure 5.2 shows the conceptual overview for our CDN. There is a further *Optimise* component, but it is necessary to discuss this in the context of the *application* of the CDN. Therefore, *Optimise* will be explored in §5.3 and §5.4 as part of our application of the CDN and its configuration process. We now discuss each construct and define the common functions and concepts necessary to run an experiment with a deep learning algorithm.

Experiment. At the highest level of *Experiment*, general settings such as the desired algorithm, number of hyper-parameter trials, model runs and the hyper-parameter search space are entered by the knowledge worker. This is a conceptual

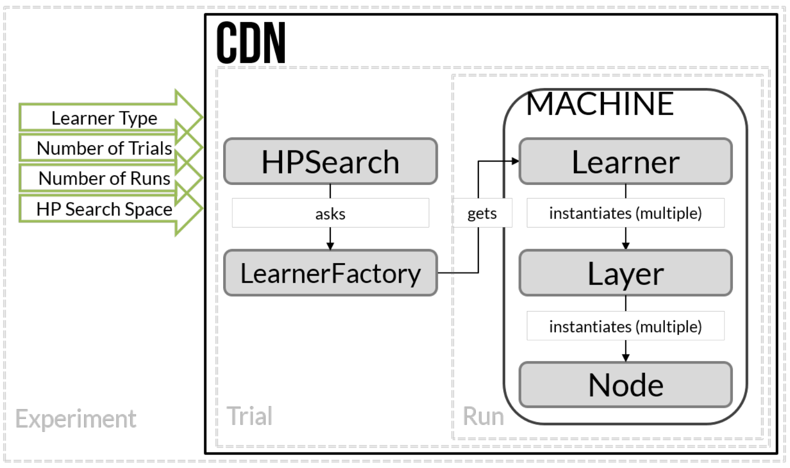


Figure 5.2: Overview of Configurable Deep Network Design

construct that in reality is used to describe the process of calling *HPSearch*.

HPSearch. This is a construct at the *Trial* level, it is the master process in the CDN and contains two functions:

- `GenerateConfigurations.` In the case of search algorithms this function generates *n_trial* hyper-parameter configurations where are then used by the `search_optimise` function.
- `SearchOptimise.` Takes the hyper-parameter configurations generated, and performs the configuration and optimisation process which we will describe in §5.4. In simple terms, it selects the appropriate data and subsequently makes a call to the `LearnerFactory` for each particular hyper-parameter configuration. It then analyses all learner models built within the CDN and returns the best performing model.

LearnerFactory. The `LearnerFactory` takes as input the desired algorithm type and hyper-parameter and returns the appropriate `Learner`, through its `get_learner` function. This concept is explored further in §5.3.

Learner. The `Learner` construct, at the Run level, we define as a single instance of a machine learning algorithm. Learners then contain an important sub-construct called a **Layer**. Traditionally, the concept of a layer has only been applied to neural networks, but shallow learning algorithms can be thought of as having two layers: an input and output layer. For example, linear regression is equivalent to a two-layer network with a visible input layer equating to features and a visible output layer with one node equating to the classification. By extension we can also define a **DeepLearner** as a `Learner` with more than three `Layers` [12].

For a single `Learner` optimisation *Run*, all learning algorithms have a similar procedure for training. First, model parameters are **initialised**, giving a starting point from which they can be tuned. Keeping with the conventions outlined in Chapter 3, we denote model parameters θ which is a combination of the *weights* and *biases* for an entire Learner. Model parameters allow for prediction in the case of supervised models, or data reconstructions or clustering in the case of unsupervised models. The data then goes through a **hypothesis function** $h_\theta(x)$, bounded by these model parameters in order to, for example, predict an outcome. The **cost**, $J(\theta)$, of these parameters is then calculated, with a function that measures the difference between the *hypothesis* output and the *actual* outcome or *ground truth.* The optimal model is then found by **updating** θ to minimise the cost of the hypothesis, as calculated by $J(\theta)$ during a process called **training**.

We now describe our generic conceptual framework for Learner construction and training. This functional framework allows us to approach the design of each Learner in a homogeneous fashion. The **Learner functions** are as follows:

1. `Initialise`. Equivalent to a learner's *constructor*, this associates appropriate input hyper-parameters with the learner instance; configures the architecture by instantiating relevant layers and nodes according to the hyper-parameters; and initialises model parameters (weights and biases).

2. `BuildHypothesis`. Dependent on the analytic functional goal and the algorithm in question, this component takes the instantiated model-parameters and constructs the relevant hypothesis. By combining input variables with model parameters in the necessary format, this generates the desired output.

3. `BuildCost`. Before updating model-parameters, we need a measure of how well the current parameters perform the required task. This method constructs

the function which takes the output of the hypothesis function, compares the hypothesis conditioned on the model-parameters with the ground truth to give a measure of how well the algorithm is currently performing.

4. `BuildUpdates`. Here we construct all functions necessary to enable accurate updates of model-parameters in order to achieve a *hypothesis* which gives a lower *cost*. This improves the performance of the algorithm with respect to the analytic goal.

5. `Train`. This method runs the optimisation procedure using the above three functions where for each data sample or batch, the hypothesis for the input data is generated; the cost calculated; and the model parameters updated; in order to improve their performance against the cost. This function also evaluates performance on held-out data for reduction of over-fitting. In the case of a *DeepLearner*, this function calls two sub-functions of `pre_train` and `fine_tune`. First *pre_train* is an unsupervised procedure which optimises the model-parameters to fit the dataset, then *fine_tune* adjusts these parameters with respect to a classification output or further fine-tunes with respect to a specific unsupervised task.

Layer. Like a Learner, all *Layers* have a similar procedure for calculating output. First layer parameters are **initialised** by the Learner. Input then **propagates** through various activation functions to calculate the activation energy. If the state is required, a **sampling** function is employed and finally if using **dropout** [123] to regularise the network, a function randomly zeros a subset of output activations or states of a layer. Therefore, we define the generic **Layer** methods as follows:

- `Initialise`. Instantiates the parameters for a layer, similar to the parent function in a learner.

- `Propagate`. Takes the input for each node in a layer - a layer can have single or multiple nodes - and performs whatever calculations are necessary to transform in the input to the output *activation energy*. This usually consists of a linear and non-linear transformation.

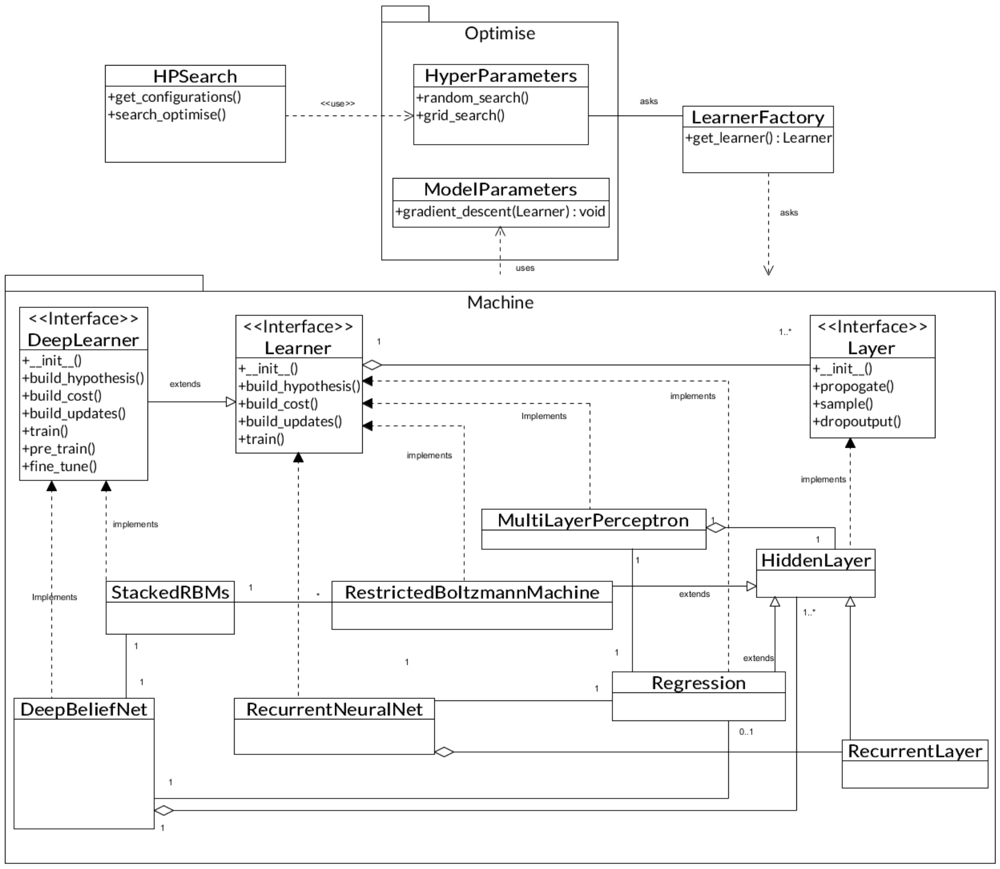


Figure 5.3: Configurable Deep Network

- `Sample`. Takes the calculated activation energy for a layer and generates the *states* for each node.

- `Dropout`. Randomly zeroes output activations or states with a probability p representing the proportions of activations zeroed.

5.3 Deployment Architecture

In this section, we provide a deployment based description of the CDN, presenting its detailed design which shows a generic approach to the design of deep learning systems. It allows for easy configuration of existing algorithms, a framework for the development of new ones and enables the optimisation of an arbitrary number of hyper-parameters within the development model.

As can be seen in Figure 5.3, the *Machine* element of the CDN has three high-level constructs: *Learners* which contain the logic for the over-arching algorithm, *DeepLearners* which is the deep extension of a Learner, and *Layers* which contain and execute activation functions as well as handle connections with other layers. Figure 5.3 also shows the deployment of the CDN to learning machines and core elements of neural networks introduced in Chapter 3. There are shallow learners, deep learners and supporting layer constructs. The supporting constructs, necessary for more complex constructs are:

- `HiddenLayer` and
- `RecurrentLayer`.

These reusable, core components enable the development of our shallow learners, which are themselves components of more complex architectures. The shallow learners are:

- `Regression,`
- `RestrictedBoltzmannMachine,` (RBM)
- `MultiLayerPerceptron` (MLP) and
- `RecurrentNeuralNetwork` (RNN).

These shallow Learners are necessary pre-requisites to make the deployment of Deep Learners easier. The Deep Learners are then:

- `StackedRBMs` and
- `DeepBeliefNetwork` (DBN).

Figure 5.3 shows that the CDN framework separates model construction from model optimisation. This separation is performed by the *Optimise* container construct, for both hyper-parameter *and* model parameter optimisation. The hyper-parameter optimisation process makes a call to the `LearnerFactory` and when `Train` is run on a particular learner instance, a call is sent to the `ModelParameters` in `Optimise` and the chosen optimisation process is then performed. This separation of concerns allows for easy addition and testing or 'plug and play' of different and new optimisation techniques for experimentation. There are further *utility* constructs, for example those which are used to initialise layer parameters, but for simplicity, we omit these from Figure 5.3. This design enables the easy configuration and construction of complex deep neural architectures, using a combination of modular components and well established design patterns. Thus, the design promotes reuse of components by analysing elements common to deep architectures.

Each learner and component relates directly to a section in Chapter 3, except for the `StackedRBMs`. The `StackedRBMs` relate to the unsupervised pre-training element of §3.3.4.1. Keeping with the formal approach described in Chapter 3, our design takes a vectorised approach to algorithm design for faster code-completion. As such, the concept of a *node* is encapsulated by a layer where nodes are represented by vector objects and node weights are combined into a matrix data-structure where each column vector of the matrix represents the weights for a particular node. For reasons of optimisation, we did not represent the *node* construct within the CDN design.

The most basic construct in the overall process is a `HiddenLayer`. It implements the Layer interface and `RecurrentLayer` is a generalisation of `HiddenLayer`, as the linear function is the only conceptual difference between a feed forward hidden layer and recurrent hidden layer.

Next, the `Regression` and `RestrictedBoltzmannMachine` are the first implementations of the *Learner* interface. These learners also *extend* the core functionality of the *HiddenLayer*. Both function as stand-alone *learners* but can be combined with other constructs to implement more complex machines. Thus, their relationships with other entities are modelled through associations as opposed to aggregations. For clarity, we omit the various forms of regression and model them as a single entity.

The `MultiLayerPerceptron` and `RecurrentNeuralNetwork`, because they are *shallow* learners. form a one-to-one *aggregation* relationship with `HiddenLayers` and `RecurrentLayers` respectively. They also have an *association* with `Regression` be-

cause this component acts as their output layer and is also a stand-alone learner. These algorithms are therefore constructed as a combination of pre-existing components with additional logic to tie these components into a single cohesive entity.
Finally, the *DeepLearner* interface is implemented by the `StackedRBM`, as well as the `DeepBeliefNetwork` construct. The `StackedRBMs` has a one-to-many *association* with the `RestrictedBoltzmannMachine`.
The `DeepBeliefNetwork` has a one-to-one association with the `StackedRBMs` to manage pre-training in deep networks where each hidden layer is first trained as a RBM as described in §3.3.4.1. Each `RestrictedBoltzmannMachine` parameter set is then used to initialise a `HiddenLayer`. Therefore, the `DeepBeliefNetwork` has a one-to-one relationship with `Regression` and a one-to-many aggregation relationship with `HiddenLayer`.
The technologies chosen to implement our CDN are Python and Theano [17], [9]. Theano is a specialised Python library designed for constructing optimised deep learning algorithms. It provides the capability to declare *symbolic* mathematical expressions and easy manipulation of *tensors* for vectorised implementations. Tensors are abstract mathematical concepts to describe geometric objects in n dimensional space, for example scalars, vectors or matrices of any dimension. The symbolic functions which are defined to operate on tensors can then be added to numeric functions to perform computation. Theano also provides automatic differentiation, which calculates derivatives in order to update model parameters for complex models. This library optimises the massively computationally expensive mathematical components of the architecture by compiling them into C and Cuda (GPU) code to achieve faster code completion.

5.4 Optimisation

The optimisation of deep learning algorithms involves two stages. The first or upper stage is concerned with hyper-parameters and the second, *nested* stage is concerned with model parameters. We focus on the process for optimising parameters within the CDN. The aim is to provide a generic optimisation process for both model *and* hyper-parameters.

Although designed for deep neural networks, this process also applies to shallow algorithms as we include both in our research and evaluation. The process is a *meta-algorithm* for building distributions of hyper parameters and model parameters. A meta-algorithm can be described as an algorithm involved in running other algorithms. This process allows for the ranking and analysis of parameter distributions, which in turn leads to robust hyper-parameter choice and insight into model parameter weightings, to give solutions with empirical and statistical rigour.

Algorithm 2 Configurable Deep Network Optimisation

```
    $D$ : input dataset
    $algorithm$ : string name of learning algorithm to be optimised
    $search\_space$ : object with search bound for each hyper-parameter
    $trials$: number of hyper-parameter configurations to trial
    $runs$: number of models to optimise for each hyper-parameter trial

 1: procedure SearchOptimise($D$, $algorithm$, $search\_space$, $trials$, $runs$)
 2:     $hp\_configs \leftarrow trials$ configurations generated from $hp\_search\_space$
 3:     $i \leftarrow 0$
 4:     while $i < num\_trials$ do
 5:         $j \leftarrow 0$
 6:         while $j < num\_runs$ do
 7:             $D' \leftarrow$ bootstrapped sample from $D$
 8:             $learner \leftarrow hp\_configs[i]$
 9:             optimise $learner$ on $D'$ w.r.t. a pre-defined cost $J$
10:             store $learner$ parameters
11:             store $learner$ performance
12:         end while
13:     end while
14:     analyse parameters
15:     if $optimalreached$ then
16:         exit
17:     else
18:         narrow search space and run again
19:     end if
20:     return best hyper-parameters and associated model
21: end procedure
```

Algorithm 2 shows our generic two-stage meta-algorithm for building learning parameter distributions. It enables robust optimisation and analysis, through the analysis of the generated distributions. This enables insight into why certain hyper-parameters configurations work well and how *variables interact* through the interpretation of feature representations learnt. The meta-algorithm utilises the Learner functions and the construction methodology outlined in §5.2. In our implementation, for the optimisation of a single learning algorithm instance (Line 9), mini batch stochastic gradient descent, described in §3.2.3.4, is used. However, the optimisation function is unimportant as the `build_updates` and `train` functions are abstract and make calls to external constructs outside the *machine* component. As the approach is generic, it can be used with other optimisation methodologies. Our meta-algorithm design was based on hyper-parameter search paradigms. However, all hyper-parameter optimisations begin with a number of hyper-parameter configurations, optimise single Learner instances for chosen configurations and analyse these choices before proceeding with a more focused hyper-parameter optimisation. Algorithm 2 can be described by three broad steps, the first and third steps relate to the Trial stage of an experiment whereas the second step relates to the Run stage. The steps can be described as follows:

1. **Determine Hyper-Parameter Configurations**. The broad purpose of this **Trial** stage step is to determine all testing hyper-parameter configurations. The input to this step is the number of trials (*trials*) and runs (*runs*) to perform as well as the *hyper-parameter search space* which contains the upper and lower bound to search for each hyper-parameter. To determine these configurations, in our implementation, either a regular search-grid is constructed and every combination of hyper-parameters in this grid becomes a configuration, or configurations are drawn randomly and uniformly between the bounds outlined in the search-space. The output of this step is a series of hyper-parameter configurations equal to *trials* in Algorithm 2.

2. **Build Parameter Distributions**. Once the hyper-parameter configurations have been determined, we proceed to the **Run** stage concerns, which involves generating the model-parameter distributions. There can be multiple Runs or a single Run for a hyper-parmater configuration, depending on the needs of the experiment. The input to this step is the number of runs to perform

($runs$), the dataset ($\mathcal{D}$) the algorithm is to be trained on, the desired learning $algorithm$ and the hyper-parameter configurations ($hp\_configs$) determined in Step 1. For each hyper-parameter trial configuration, instances of $runs$ for the required learning algorithm are constructed and optimised. To do this, a bootstrap sample - random sample *with replacement* from original full dataset - is taken and a learning algorithm is built and optimised with respect to a predefined cost. This involves performing numerous *Updates* on the model parameters with an optimisation procedure of choice.

A single, optimised learning algorithm instance for a single hyper-parameter configuration is referred to as a *Trial-Run*. The outputs of a particular Trial-Run are optimised model parameters along with the performance measures they achieve for the hyper-parameter trial configuration.

The output of the entire step is then $trials \times runs$ distributions of hyper-parameters and performance measures and $trials$ sets of $runs$ model parameter distributions.

3. **Analyse Parameter Distributions**. The distributions generated in Step 2 are then inputted to this process and have two purposes. The first is the assessment of hyper-parameter performance and the second is the assessment of model parameters. Model parameter assessment allows us to determine if model parameters are stable across multiple runs. We rank the distributions by the cost achieved on a held out validation subset of the overall dataset (X^v - defined in Chapter 3).

   For hyper-parameters the user can, through an analytical framework parameter, either select the single best performing configuration of hyper-parameters and related model-parameters, or generate a reduced search space for further optimisation. Regarding model parameters, those inputs significant to a hidden node in a layer and their relative importance compared to other inputs, are presented. The mechanisms used to perform these assessments are part of the post-processing step described in detail in Chapter 6.

   The output of this step is the best performing hyper-parameters and related model parameters, or a narrowed bounds in which to continue searching for hyper-parameters. This search includes a map which shows how input features combine in the hidden layers.

5.5 Summary

In many deep learning research projects, hyper-parameter optimisation is either overlooked or multiple frameworks must be utilised in parallel for their optimisation. The aim in designing a configurable deep learner was to provide modular, loosely-coupled, reusable software components which contain core functionality, and provide a *framework* in which deep learning researchers can easily configure and run automated deep learning experiments. These experiments include both hyper-parameter *and* model parameter optimisation. Without going into too much implementational detail, this chapter served to motivate and highlight our approach to a design which allows for a building-block approach to building these complex experiments. At this point, we can see that a deep learning experiment is both large and complex with many working parts. The next step, and the focus of Chapter 6, is the specification of a data model to capture all of the components of a deep learner. This is crucial in providing both data management and analytical capabilities for the system described here.

Chapter 6

Deep Learning Data Model

The previous chapter described a parameter optimisation and feature interpretation process as part of the Configurable Deep Network software architecture. In this chapter, we present our deep learning data model, which was designed to capture all aspects of the deep learning experiment, from configuration to experimental runs and optimisations and interpretation of results. In §6.1 we introduce our Deep Learning Data Model which is referred to as the POL data model: Parameter Optimisation for Learning. This was designed to satisfy requirement b1 - *a conceptual model for deep learning*, designed with *flexibility* in mind and to enable easier reproduction of experiments. In §6.2, we describe our *physical data model* and Deep Learning DataBase (DLDB), which implements the POL conceptual data model and satisfies requirement b3, enabling *simple* storage of results. Finally, in §6.3, we present the POL API and Evaluation toolkit, which interacts with our physical data model and provides high-level exchange, visualisation and evaluation functions for the deep learning experiments, within which requirement c8 is satisfied as we provide *parameter analysis functions* for *interpretable* results.

6.1 The POL Data Model

The Parameter Optimisation for Learning (POL) model is a conceptual model designed specifically for Deep Learning. Its goal is first to be *simple* where the minimal amount of attributes (requirement b7) are included but still contains enough infor-

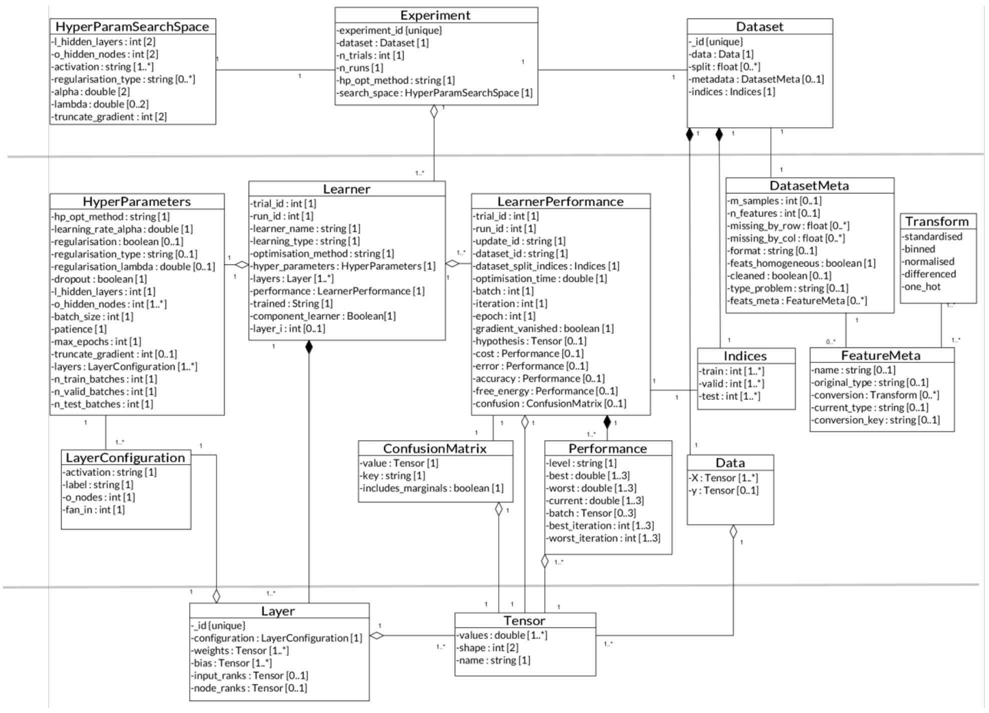

HyperParamSearchSpace
-l_hidden_layers : int [2]
-o_hidden_nodes : int [2]
-activation : string [1..*]
-regularisation_type : string [0..*]
-alpha : double [2]
-lambda : double [0..2]
-truncate_gradient : int [2]
Experiment
-experiment_id {unique}
-dataset : Dataset [1]
-n_trials : int [1]
-n_runs [1]
-hp_opt_method : string [1]
-search_space : HyperParamSearchSpace [1]
Dataset
-_id {unique}
-data : Data [1]
-split : float [0..*]
-metadata : DatasetMeta [0..1]
-indices : Indices [1]
HyperParameters
-hp_opt_method : string [1]
-learning_rate_alpha : double [1]
-regularisation : boolean [0..1]
-regularisation_type : string [0..1]
-regularisation_lambda : double [0..1]
-dropout : boolean [1]
-l_hidden_layers : int [1]
-o_hidden_nodes : int [1..*]
-batch_size : int [1]
-patience [1]
-max_epochs : int [1]
-truncate_gradient : int [0..1]
-layers : LayerConfiguration [1..*]
-n_train_batches : int [1]
-n_valid_batches : int [1]
-n_test_batches : int [1]
Learner
-trial_id : int [1]
-run_id : int [1]
-learner_name : string [1]
-learning_type : string [1]
-optimisation_method : string [1]
-hyper_parameters : HyperParameters [1]
-layers : Layer [1..*]
-performance : LearnerPerformance [1]
-trained : String [1]
-component_learner : Boolean[1]
-layer_i : int [0..1]
LearnerPerformance
-trial_id : int [1]
-run_id : int [1]
-update_id : string [1]
-dataset_id : string [1]
-dataset_split_indices : Indices [1]
-optimisation_time : double [1]
-batch : int [1]
-iteration : int [1]
-epoch : int [1]
-gradient_vanished : boolean [1]
-hypothesis : Tensor [0..1]
-cost : Performance [0..1]
-error : Performance [0..1]
-accuracy : Performance [0..1]
-free_energy : Performance [0..1]
-confusion : ConfusionMatrix [0..1]
DatasetMeta
-m_samples : int [0..1]
-n_features : int [0..1]
-missing_by_row : float [0..*]
-missing_by_col : float [0..*]
-format : string [0..1]
-feats_homogeneous : boolean [1]
-cleaned : boolean [0..1]
-type_problem : string [0..1]
-feats_meta : FeatureMeta [0..*]
Transform
-standardised
-binned
-normalised
-differenced
-one_hot
Indices
-train : int [1..*]
-valid : int [1..*]
-test : int [1..*]
FeatureMeta
-name : string [0..1]
-original_type : string [0..1]
-conversion : Transform [0..*]
-current_type : string [0..1]
-conversion_key : string [0..1]
LayerConfiguration
-activation : string [1]
-label : string [1]
-o_nodes : int [1]
-fan_in : int [1]
ConfusionMatrix
-value : Tensor [1]
-key : string [1]
-includes_marginals : boolean [1]
Performance
-level : string [1]
-best : double [1..3]
-worst : double [1..3]
-current : double [1..3]
-batch : Tensor [0..3]
-best_iteration : int [1..3]
-worst_iteration : int [1..3]
Data
-X : Tensor [1..*]
-y : Tensor [0..1]
Layer
-_id {unique}
-configuration : LayerConfiguration [1]
-weights : Tensor [1..*]
-bias : Tensor [1..*]
-input_ranks : Tensor [0..1]
-node_ranks : Tensor [0..1]
Tensor
-values : double [1..*]
-shape : int [2]
-name : string [1]

Figure 6.1: POL Conceptual Model

mation to *reproduce* and represent an abstract deep learning *experiment* and *learning algorithm*. Secondly, it is designed so the *schema is flexible* and *user-extensible*. The field of deep learning is highly variable and fast-evolving so user-customisation is paramount when designing such a model. There are four broad aspects to the POL model: input, model-parameter, hyper-parameter and (interim and final) performance data management (req. c7). The highest level of abstraction is an *experiment*, which is run upon a *dataset* and results in a number of trained *learners*. We focus on these concepts in sections §6.1.1, §6.1.2 and §6.1.3. Within the experiment entity, are all types and attributes required to describe parameter and result data and the settings used to achieve these results. The data model was given the term Parameter Optimisation for Learning (POL) model, as it is used as a platform to deliver both Hyper-parameter optimisation and Model parameter optimisation. Thus, the POL model provisions for the capture of optimisation and *interim result data*, therefore also satisfying requirement a6/b4/c6. Furthermore we design the POL model to be *flexible* and *extensible*

Figure 6.1 shows all types present in the POL. The highest level types *Experiment* and *HyperParameterSearchSpace* are covered in §6.1.1. The types *Dataset*, *DatasetMeta*, *Transform*, *FeatureMeta*, *Indices* and *Data*, describing the dataset are covered in §6.1.2. The learning and performance types, *Learner*, *HyperParameters*, *LayerConfiguration*, *LearnerPerformance*, *ConfusionMatrix*, *Performance*, *Layer* and *Tensor* are covered in §6.1.3.

Terminology. For research which encompasses both data modelling and machine learning, it is necessary to clarify the usage of the term *model*. It is used in both data management and machine learning but with different meanings. The *conceptual data model* presented in this research captures all of the *data properties* of a deep learning experiment. For this reason, we will use the term **data model** to refer to our representation of these data concepts. When discussing the machine learning aspects of our work, we will use the term **learner** to refer to the generic concept of a deep learning *algorithm* and **learner model** for the predictive or descriptive *model* which can be associated with a learner to refer to the model (parameter) instances learned over the course of the training process.

6.1.1 Experiment type

The **Experiment** type represents the first major concept in our model. In this section, we describe the *Experiment* type and its associated *HyperParamSearchSpace* type which together form the highest level in a deep learning experiment. An *Experiment* encapsulates the entire process of hyper-parameter and model parameter optimisation in a deep learning experiment on the target dataset. To reiterate the process introduced in Chapter 5, an experiment has multiple trials and each trial consists of multiple runs and updates, where a trial tests a single hyper-parameter configuration. A single hyper-parameter configuration is used for multiple, complete optimisation runs and each run consists of multiple updates. Experiment has the following attributes:

- `experiment_id`: A unique string ID for the experiment.
- `dataset`: This is a complex dataset object, representing the dataset.
- `n_trials`: The number of hyper-parameter configurations to trial.

- `n_runs`: The number of complete learner optimisation runs to perform on each hyper-parameter trial configuration.
- `hp_opt_method`: The name of the procedure which is used to optimise the hyper-parameters.
- `search_space`: The hyper-parameter search space.

The **HyperParameterSearchSpace** is the space in which we search for each of these parameters. Hyper-parameters are those parameters that are not learned by the model but are instead input to the learning process in order to influence model training. Like a dataset, it is an input into a deep learning experiment with some attribute values determined by the size and properties of the dataset. For the search space, we define an upper and lower bound in which we search for the best setting for each hyper-parameter. The experiment should instantiate n_trial hyper-parameter *configurations* within these bounds. As all hyper-parameter concepts in **HyperParameterSearchSpace** are contained in **HyperParameters**, we will provide a detailed description of **HyperParameters** in §6.1.3, keeping in mind that this relates to a single configuration of hyper-parameters generated within bounds that are set out in the search-space.

6.1.2 Dataset type

The **Dataset** type is the second major concept in the POL data model. In this section, we describe the Dataset type and the five associated types which together are used to capture all characteristics of a dataset needed for direct input into a deep learning experiment. It encapsulates the data upon which the experiment is run and leads to the generation of all learners and performance achieved from these learners. Dataset has the following attributes:

- `_id`: A string ID to provide unique identification for the dataset.
- `data`: An instance of the complex **Data** object representing the actual data stream. The **Data** object has the attribute `x` to represent data which will be

input into the hypothesis function and an optional `y` attribute, if the dataset has classifiers.

- `split`: A list containing the proportional splits for the training, validation and test sets respectively. The proportions take the form of percentages, where, for example, (0.6, 0.3, 0.1) would represent 60% of the data to be used for training, 30% for validation and 10% for testing.

- `metadata`: A complex object relating to the metadata collected and generated to describe the dataset.

- `Indices`: A complex object relating to the configuration of the dataset for a particular trial run. The **Indices** class has three attributes: `train`, `valid` and `test`, each relating to how the dataset should be split for training. Each of these attributes contains a list which stores the original position of a row in the dataset at the index at which it should be utilised. For example, if `train` holds the value 0 at index 100, the first row of the dataset should be the 100th input into the hypothesis function during training. This is a strategy which allows for easy partitioning, bootstrapping and shuffling of data by manipulating index arrays rather than partitioning and manipulating the dataset and re-generating multiple redundant data tensors for each trial-run.

The **DatasetMeta** type captures the metadata properties of a dataset through the following attributes:

- `m_samples`: the number of samples in the dataset.

- `n_features`: the number of features in the dataset.

- `missing_by_row`: Equal in length to `m_samples`, it captures the percentage *missing* of each sample.

- `missing_by_col`: Equal in length to `n_features`, it captures the percentage *missing* of each feature.

- `format`: Represents the most complex feature type in the dataset eg. continuous, discrete or one-hot.

- `properties` Indicates if the data is high-dimensional, sparse, sequential, time series, etc.
- `feats_homogeneous`: A boolean representing if the features are all of the same type.
- `cleaned`: A boolean to indicate if missing data has been dealt with.
- `type_problem`: Captures whether the dataset is supervised, unsupervised, semi-supervised, etc.
- `feats_meta`: A list equal in size to `n_features`, which holds a **FeatureMeta** instance entry.

FeatureMeta exists to represent the meta-data collected and generated about each feature and has the attributes listed below. **FeatureMeta** has a zero-or-many-to-one relationship with **DatasetMeta** because feature meta-data can be collected on none, all or some features.

- `name`: short descriptive identifier of the feature.
- `original_type`: captures the type of feature before any transformation: continuous, binary, discrete.
- `conversion`: which holds a **Transform** instance to describe which transformations have been carried out on a feature.
- `current_type`: holds the current type of the feature.
- `conversion_key`: holds the mapping from the original value to the current feature.

Transform currently has the following Boolean attributes to signify if a particular conversion has been applied to a feature: `standardised`, which gives continuous features zero mean and unit variance; `differenced`, which takes the difference between two subsequent points in a time-series to remove non-stationarity; `normalised`, to signify a feature is scaled to be between 0 and 1; `binned`, indicating a continuous feature has been discretised; and `one-hot` is the process which enables continuous and discrete features to be input into learners which require binary input. These transformations were covered in Chapter 4.

6.1.3 Learner Type

In this section, we describe the *Learner* type and the 7 associated types which together fully describe all aspects of the deep learning algorithm. This includes the *LearnerPerformance* type which captures the performance of the algorithm, at the required snapshot in time.

The **Learner** type is the largest *core* concept of the POL data model and encapsulates the concept of a deep learning algorithm and the learner model instance associated with it. The *Learner* type contains all necessary function parameters to allow a user to utilise the learner instance according to its unsupervised or supervised purpose. It also contains necessary input parameter settings to find and exactly recreate a training process. The POL data model is designed specifically for deep and shallow neural networks although the extension to other machine learning algorithms should be straightforward as most can be described as a neural network with no hidden layers. The *Learner* class has the following attributes:

- `_id`: A unique identifier for the snapshot.
- `trial_id`: The trial number which is instantiating the learner.
- `run_id`: The run number which is instantiating the learner.
- `update_id` An ID to identify the update which generated the snapshot. It is a combination of `trial_id` and `run_id` together with the `epoch` and `iteration` which produced the snapshot, that are attributes of **LearnerPerformance**.
- `learner_name`: Name of the learning algorithm, for example: Restricted Boltzmann Machine or Recurrent Neural Network (RNN).
- `learning_type`: Analytical goal or learning task, for example the learning carried out was supervised, unsupervised or reinforcement.

- `optimisation_method`: Training algorithm for the learning function, for example mini-batch stochastic gradient-descent (MSGD).
- `hyper_parameters`: An instance of the **HyperParameters** class referring to input and fixed parameters used by the optimisation process and initialised within the search-space bounds.
- `layers`: A list of **Layer** objects, which transform features into more abstract features or classifications and predictions. The `layers` attribute contains all model-parameters: the weights and biases, for each layer of the `Learner`.
- `performance`: Instance of the `Learner_Performance` object, containing a result snapshot for an update, or the final result if training has finished.
- `trained`: String attribute which indicates if the **Learner** instance is the final result of a model-parameter optimisation run, a pre-train run, or if the instance is a snapshot of a particular update.
- `component_learner`: Only relevant in the case of deep learning experiments. It is a Boolean attribute to indicate if the update being stored is actually part of a larger process, for example an RBM being pre-trained as part of a DBN.
- `layer_i`: Again only relevant for deep learning. An integer to indicate a component learner's position in the overall architecture.

The **Hyper-Parameters** class holds the hyper-parameter configuration for a single learner and each attribute contains a single element generated from the bounds of the search-space, along with some derived attributes which are generated when the training process starts. Ultimately, it is used to instantiate the relevant *Learner* for training. Its attributes are:

- `hp_opt_method`: The name of the algorithm which was used to optimise the hyper-parameters.
- `learning_rate_alpha`: The co-efficient used in the model parameter update function which determines the magnitude of the updates for one iteration of optimisation (usually a form of gradient descent).

- `dropout`: A Boolean indicating if dropout was applied to the learner or not.
- `regularisation`: A Boolean indicating if regularisation was applied to the learner.
- `regularisation_type`: The form, if any, of model-parameter regularisation, for example *L1, L2* or *none*.
- `regularisation_lambda`: The co-efficient in the regularisation function which determines the penalty placed on very large or small weights and biases. This can be null if *dropout* or *no regularisation* is applied.
- `l_hidden_layers`: Number of hidden layers in the **Learner**; 1 or less is considered shallow otherwise it is considered deep.
- `o_hidden_nodes`: List where each element describes the number of hidden nodes in each hidden layer.
- `batch_size`: Number of dataset rows to use for mini batch stochastic gradient descent (MSGD), which affects an algorithm's learning ability. A size of 1 is synonymous with stochastic gradient descent, whereas a size equal to the number of training samples denotes batch gradient descent.
- `patience`: *Minimum* number of *updates* to apply during optimisation. It is used by the early stopping process and is increased during optimisation if performance on the validation set continues to improve above a certain threshold.
- `max_epochs`: *Maximum* number of *epochs* to iterate while applying updates to the learning function. The bounds depend on the number of model-parameters and rows in the dataset.
- `truncate_gradient`: This only applies for recurrent architectures optimised with back-propagation through time (BPTT). This parameter is used to determine how far in the past to pass errors in back-propagation as the process suffers from vanishing gradients if too many time-steps are iterated.

- `layers`: A list of `LayerConfiguration` objects which detail the setup and label of each layer in the architecture.
- `n_train_batches, n_valid_batches, n_test_batches`: Number of batches in training, validation and test sets, respectively.

The **LayerConfiguration** type captures settings which are specific to each layer in the learning algorithm. The number of **LayerConfiguration** objects are generated equal to `l_hidden_layers` and can be used to generate the appropriate layer configuration. A **LayerConfiguration** instance contains: a string `label` which concatenates *input*, *output* or *hidden* with its position in the architecture; an `activation` string, to specify the type of *activation function* to be used; `o_nodes` to designate the number of nodes in the layer; and `fan_in`: the number of nodes in the previous layer.

The **Layer** type is an individual input, hidden or output component of a shallow or deep neural network. It takes input from other layers (or features if the Layer instance is the input layer) and transforms this input into a series of learned features which can result in classifications, predictions or data reconstructions. A learner must have at least one layer. The structure of the Layer type is as follows:

- `id`: position of layer in the architecture.
- `configuration`: An instance of the **LayerConfiguration** class.
- `weights`: Parameters of a layer that allow for the calculation of the weighted sum from the input provided from previous layers, represented by a **Tensor** instance.
- `bias`: The bias tensor allows the activation function to more accurately represent the feature being learned or the classification function by moving the function as opposed to simply growing or shrinking it. It also represents the output of a learner model if all the parameters were set to zero.
- `node_ranks`: A simple one-dimensional tensor which contains the rank of each learned feature according to its impact on the model.

- `input_ranks`: A tensor similar to feature ranking but containing information on how much each element in the previous layer contributes to a particular node in the current layer.

The **Tensor** class represents a geometric mathematical object, an instance of which is scalar, vector, or matrix. Its `values` are always captured as a numeric array of values with one dimension. The attribute `shape` contains the original shape of the object so the one-dimensional array can be restored to its original configuration. Finally, `name` is unique for the object. Although we explain **Tensor** in the context of the **Learner** type, this type is used by any object which requires the storage of numeric values in the form of a matrix, vector or scalar.

The **LearnerPerformance** type captures the performance measures and output of the learner instance at a particular iteration in learning. Its flexibility allows it to capture the final performance of the model, the performance at the epoch level, or the performance at the update level. Its structure is comprised of the following:

- `trial_id` Inherited from **Learner**.
- `run_id`: Also inherited from **Learner**.
- `update_id`: Also inherited from **Learner**.
- `dataset_id`: The dataset identifier, inherited from **Dataset**.
- `dataset_split_indices`: a complex **Indices** object, described in §6.1.2. If stored with the results, it gives a user the *exact* means to recreate the results found at this snapshot of the experiment.
- `optimisation_time`: Wall-clock time of how long it took to get to this particular update of the model.
- `batch`: The batch of the dataset on which these performance metrics were generated.
- `iteration`: The number of iterations of updates that have elapsed up to and including this *snapshot*.

- `epoch`: The number of complete iterations through the dataset which have occurred.
- `gradient_vanished`: A Boolean indicating whether or not the gradient had vanished for this snapshot.
- `hypothesis`: The output of the learners hypothesis function, represented by a **Tensor** object, this is stored at the end of a complete *run*.
- `cost`, `error`, `accuracy`: A complex **Performance** object which relates to the objective function used to measure the efficacy of the hypothesis.
- `free_energy`: Although not strictly a performance measure it is represented by the **Performance** object. The free energy of a system is only applicable in the case of energy based algorithms such as Restricted Boltzmann Machines.
- `confusion_matrix`: an instance of a **ConfusionMatrix** object.

The **ConfusionMatrix** type is used only in supervised learning experiments. Like a contingency table, it compares the predictions against the ground truth for each class in the label-set and is usually only captured at the end of a *trial-run*. It contains: a `key`, which describes the format of the rows and columns - which represent the ground truth predictions; `value`, a tensor containing the actual values for the true and false positives and negatives; and finally, `includes_marginals` which indicates if the matrix includes extra rows containing aggregates.

The **Performance** type captures the score computed by an objective function. Each attribute contains a list with at least one value, which is the training set score. The array indexes refer to the portion of the data from which they were obtained: the value at index 0 refers to a training score, 1 refers a validation score; and 2 refers to a test score and so on. As a snapshot can be captured at various points, **Performance** will not necessarily contain scores for *every* portion of the split dataset, as an experiment might only validate after a set number of epochs or test once in an entire trial.

- `level`: Represents the performance level of capture and can be one of three options: trial-run, epoch, or update.
- `current`: An array of values with a cardinality equal to the number of dataset splits. Each value relates to the performance of the hypothesis for the current iteration of model training on the relevant portion of the dataset.
- `best` and `worst`: The same cardinality as `current`, these attributes hold the value of the best and worst scores found so far over the course of learning.
- `best_iteration` and `worst_iteration`: Capture the number of epochs iterated to find the best and worst scores respectively.
- `batch`: If the level is epoch and scores are aggregated through averaging, this can be used to hold the performance in a number of tensor arrays for the individual batches or samples.

6.2 Storing and Reloading Deep Learners

The goal of the Deep Learner Database (DLDB) is to capture a snapshot from any point in a deep learning experiment. It should be possible to analyse the experiment at that moment in time and: restart the experiment at any time, from that point; retrace the experiment to the point where a specific decision was made and move forward in a different direction; or export the experiment results for sharing. The DLDB schema is a physical representation of POL (conceptual) data model. It also addresses concerns that are present only during experiment run-time which cannot form part of the POL data model. This section begins with a motivation as to why the data storage layer is critical to deep learning and the benefits that are accrued by using persistent storage for experiments. In §6.2.1, we discuss the technologies used in the creation of the first deep learning experiment database. This database, combined with our published POL data model provides a platform for sharing experiment configurations and results. Finally, in §6.2.2, we discuss critical aspects of our storage mechanism.

The DLDB, relates to the data storage and access layer in the architecture outlined in Chapter 4. The storage and access layer is critical in allowing researchers to persist, query and if required, share data resulting from deep learning experiments. Persistent storage of deep learning experiment data relates to writing to disk all *data properties* described in the POL data model, which are generated over the course of numerous hyper-parameter *trials*, consisting of multiple model parameter optimisation *runs* which are in turn the result of many thousands of *updates*, as outlined in the optimisation process presented in §5.4.

Often in deep learning experiments, hyper-parameter configuration *trials* are performed manually and only the performance measure in the form of the score achieved on some objective function is kept and *not* the model-parameters which generate the scores. Furthermore, during optimisation *runs*, it is usual for data and model-parameters to be discarded with only the final optimised learner recorded. At the end of an experiment, the researcher is left with only a series of performance numbers for different hyper-parameter trials, together with the final optimised learner.

The advantages highlighted in [133] for machine learning databases also accrue from deep learning experiment databases. If all experiment settings are stored and easily exported and imported in an interoperable format, it is possible to *reproduce* experiment results and **reuse** and build upon those results. Another advantage is that new queries can be run on historic experiment data. The POL data model is the first step towards the **standardisation** of deep learning experiment approaches and data capture. However, the next step is the provision of a mapped storage model which enables persistence at every step in the experiment.

Our work is similar to [133] but focuses on deep learning. Furthermore, instead of describing *only* model optimisation *runs*, we also distinguish hyper-parameter configuration *trials*, as these are a major component of deep learning experiments. We conceptualise their combintation as a *Trial-Run*, for example, Trial 50, Run 2. We also define the more granular *Update* stage of an experiment, where we capture model parameters. Having *both* more granular and higher-level stages of capture allows us to debug and interrogate the hyper-parameter *and* model parameter optimisation processes. We will leverage our DLDB in the evaluation to demonstrate this depth of experiment analysis, which otherwise is not generally possible.

6.2.1 DLDB Technologies

The decision was made to deploy our POL data model using an *interoperable* data representation language so that experiments can be easily shared, thus, satisfying requirement b5 and adhering to the *portability* design principal. This representation language should be coupled with a suitable storage mechanism so results can *persist* after the lifetime of an experiment. We chose JavaScript Object Notation (JSON) as the physical data representation and MongoDB's NoSQL as the storage model. Therefore, our work is the first machine learning experiment database to adopt a completely NoSQL paradigm and the first experiment database with a published data model specifically for deep learning. Given that we have proposed a common data model for deep learning, the first requirement was for the data representation to be **language independent** and **easily parsed** by most languages. Querying, coding and writing to these structures should be simple and fast. Finally, both the representation and storage should be sufficiently *expressive*, accounting for the tree-like structures outlined in our data model which contain complex, nested objects and few joins.

JSON's primary purpose is as a *lightweight* language for *data interchange* (requirement b6). It is a subset of JavaScript, but also C and YAML V1.2 [70], [67], [11]. Although it is a subset of these languages and JSON uses JavaScript style object declaration syntax, it is *language-independent*, as it is stored as human-readable text. Finally its *data oriented* markup is in contrast to a document oriented paradigm.

JSON is based on just two data structures. The first is an **object**, which is a collection of name-value pairs, and the second is an ordered list of values or **array** [67]. Both structures can be directly mapped to many languages' native data types, adding to JSON's interoperable nature. These structures allow a JSON object to be *easily parsed* and exchanged between many heterogeneous languages, especially those which are dynamically typed. *Values* can be strings, numbers, Booleans, arrays or *other objects* and arrays can scale dynamically with any of these types [70]. Thus, JSON can easily represent complex, nested, tree-like structures and although the structures are *simple*, they are also *expressive* and *flexible*.

For the reasons presented, JSON is easily interpreted by machines. Furthermore, its lightweight, self-describing and text-based syntax means it is also easily read and interpreted manually. In this, it is similar to XML, but as a result of the reduced syntax and omission of closing tags, it is easier to read when compared to other interoperable formats [98].

MongoDB has a direct mapping to JSON and extends JSON as a structured data management system with add-on functionality. Mongo stores data as Binary JSON (BSON) [66], a serialised binary format of JSON text objects. BSON further compresses and increases the speed and access of JSON objects [66]. A record in MongoDB is a direct mapping to a top-level JSON object. MongoDB adds two more levels of abstraction: a **collection** which is an aggregation of JSON objects; and a *database*, which is a grouping of collections. MongoDB's rich query language provides the ability to *shard* and distribute JSON databases and like regular JSON, allows for flexible and dynamic schema, enabling fluent polymorphism [66]. The flexibility and speed of Mongo for non-normalised data is important when the data model of a particular instance of a deep learner can change dramatically in configuration and scale, over the course of a learning experiment.

As there is a direct mapping between Mongo and JSON, there is no need for expensive model transformation when querying or storing. If Python or another dynamically typed language is used to manipulate the JSON data, there is very little parsing involved. A JSON object can be passed *natively* as a Python dictionary. The JSON ordered list is an important structure when creating a storage model for the POL model. An ordered list allows us to represent mathematical objects such as bias vectors directly, or weight matrices and other tensors easily, in a flattened format. These can also be represented as Python lists or arrays in other languages. Finally, storage such as MongoDB allows for easy export of an *entire* experiment, via a database dump, which can be uploaded to an online environment for simple collaboration and sharing and thus, other researchers can download, import and analyse the same results.

6.2.2 The Deep Learning Experiment Database

We have already provided a detailed discussion of each POL data model concept in section §6.1 and here, we limit our description to those elements specific to *storage and persistence*. As such, figure 6.2 illustrates only those constructs appropriate to this discussion.

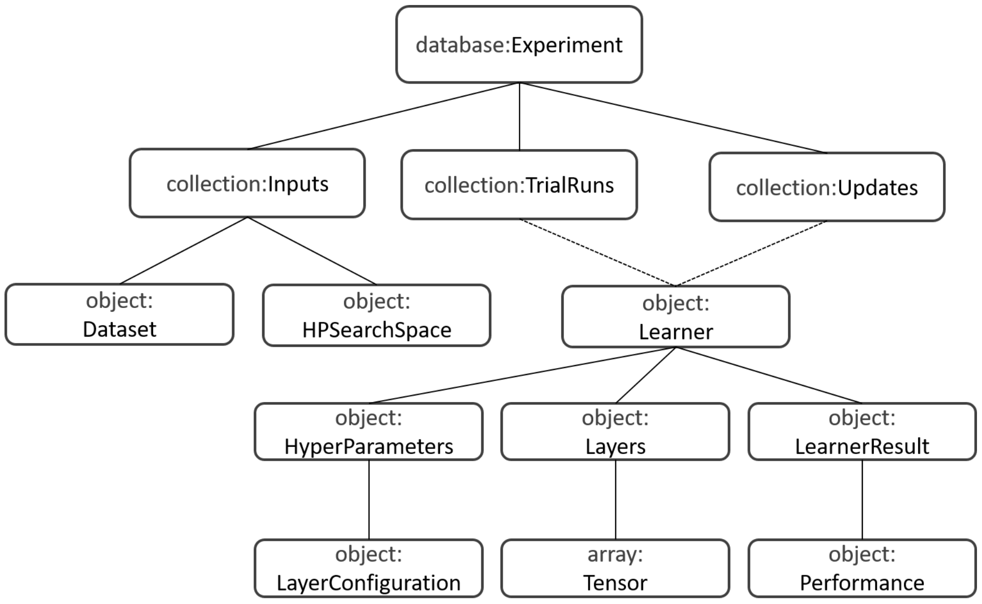


Figure 6.2: NoSQL Storage Model for Deep Learning

Recall that an *experiment* represents the optimisation of *all* parameters. Within an experiment, a number of hyper-parameter configuration *trials* are performed, which consist of one or many model optimisation *runs*, which themselves are the result of many model parameter *updates*. We regard a *snapshot* as the capture of all model and hyper-parameters, and appropriate performance indicators for any learner instance, at any stage of the experiment.

If we are to potentially store thousands of *learner snapshots* over the course of the experiment, we need to augment our conceptual model with certain structures for its physical implementation. Similar to Mongo extending the utility of JSON, we extended the POL model to take advantage in Mongo using the *collection* structures, which are aggregations of objects, and the *database* itself, which is an aggregation of collections.

In figure 6.2, we represent the *Experiment* as a MongoDB *database*, as it is at the highest level of abstraction in the POL model. The *Experiment* stores the results of all trials, runs and updates and these entities are linked in the MongoDB schema. The new high-level concepts which extend the POL model are: *TrialRun*, *Updates* and *Inputs*. These constructs are suited to the strengths of NoSQL, which better manages non-normalised and expressive structures. Inputs contains the *HyperParameterSearchSpace* and *Dataset*. *Trial-Run* then contains *only* optimised learners whereas *Updates* contains non-optimised *interim* learner performance *snapshots* for every trial-run. The update *snapshots* are captured at a predefined stages throughout training. It is possible but unwise to store *every* update as this would result in unmanageable volumes of data. Furthermore, rather than store every component of a snapshot, researchers can decide on those aspects of the snapshot to be stored. It is worth noting that for a *deep* experiment, once a learner has been *pre-trained* this result is stored in *TrialRuns* as well as the final *fine-tuned* snapshot.

6.3 Analysing Performance Data

Storing result data is of little use without the means to query, analyse, present and share it. These elements are enabled by our physical model but it is only through the **Access**, **Validate**, **Visualise**, and **Evaluate** APIs that we can effectively achieve these goals. Together with the **Setup** API and **DLDB**, these components collectively make up our Deep NoSQL Toolkit (DNT).

6.3.1 Setting Up and Accessing Deep Learning Storage

We discuss the `Setup`, `Access` and `Validate` APIs together as their functionality is closely related. The `Access` API contains the necessary functionality to read and write POL model data using the *Schema* and *Entities* components. Both contain a definition file for each element in the POL model, save for *Experiment*, which is encapsulated by the `Setup` API.

Schema and Entities. The `Schema` component contains *JSON* schema definitions and `Entities` contains Python schema definitions. These are the Persistent and the

In-Memory POL components respectively. The Configurable Deep Network interacts with `Entities` while an experiment is running and the `Validate` API interacts with the `Schema` component before anything is written to disk. The `Validate` API contains only one function: `validate`. This function is to ensure each object written to the persistent store conforms to the POL data model.

Setup. *Experiment* is not contained in the `Validate` or `Entities` components, nor is *Trial-Runs*, *Updates* or *Inputs*. These components are implemented as MongoDB *databases* and *collections* and can only be instantiated at experiment run-time by the `Setup` API. The `Setup` API has two functions. The first, `read_data`, takes the cleaned, transformed dataset as input, queries the *Entities* component and populates the in-memory POL dataset type. The second function, `generate_environment`, takes as input a POL in-memory dataset and an optional POL hyper-parameter search space. If no search space is given, the *Entities* component is queried and a default hyper-parameter search space is generated with information from the dataset. The function `generate_environment`, then generates an experiment ID with information from the dataset and the current time stamp, before instantiating the *Experiment* type in Mongo and writing the search via the `Access` API. The *TrialRun* and *Updates* structures are also instantiated at this point in Mongo. During experiment run-time, we ensure *TrialRun* and *Updates* contain the right structures by only writing to *TrialRun* at the end of a model optimisation (pre-train or fine-tune) run and writing to *Updates* at all other times.

Access. The `Access` component, contains two elements: `Write`, which contains the functionality to create and update persistent POL objects; and `Read`, which reads POL objects. The functions in `Read` and `Write` communicate with either a MongoDB persistent store or the file-system, in serialised binary or non-serialised formats. The two elements can be described as follows:

- **Write**. The `Write` component contains the database functions: `set_entity`, `update_entity`, `snapshot`, and `dump_experiment`. Its file system functions, `serialise` and `to_file`, take the desired output path and the relevant POL entity, query `Validate` and write these entities to disk. The `set_entity` function, takes as input the experiment ID, the relevant POL object and the collection, which is either *TrialRun* or *Updates.* The `update_entity` function, takes the same parameters as *set_entity as well as* a *key.* It first searches for the object with the relevant high level ID and then, for the *key* that was passed. If the key exists, it then updates the current value with the new value. If the key does not exist, the function creates the key and adds the value to this key. This is an important feature when using the evaluation functions in order to associate, for example, node ranks within a layer to a particular learner. The function `snapshot`, takes as input a *Learner* instance and its associated *LearnerPerformance* object along with a Boolean to indicate whether to write to the database or to the file-system. It combines *Learner* with *LearnerPerformance* to form a *snapshot.* This *snapshot* reflects the result of a particular Update in the Experiment. Subsequently, it queries `Validate` to ensure conformation to the POL. Finally, it uses `set_entity` to write to MongoDB, or `serialise` or `to_file` to output snapshots to the file-system. If *snapshots* are written to the file system, *TrialRuns* and *Updates* are realised as directories. The `dump_experiment` function outputs the entire data for an experiment to the file system, when given an output path.

- **Read.** The functions in `Read` are: `get_entity`, `deserialise` and `from_file`. The function `get_entity` reads solely from the database. It takes the experiment ID, the collection name, the name of the entity (this can be an sub-entity of a learner) and either the Learner ID or the Update ID, to return the correct entity. If it does not receive a collection to query, the function defaults to querying the *TrialRun* collection. The functions `deserialise` and `get_entity` take as input, a file system path and return the object found at that path.

6.3.2 Evaluating and Visualising Deep Learning Experiments

The components crucial to processing results and progressing the experiment are the *Visualise* and *Evaluate* APIs. The role of *Evaluate* is to analyse the data and learner models generated over the course of an experiment while the role of *Visualise* is to present elements or analyses found in the experiment to the user.

Evaluate. All aspects of data captured in the DLDB can be examined with our toolkit using **Evaluate** which contains two sub-components: `Performance` and `Parameters`. The data can be examined at four different stages of *roll-up* or *drill-down* and therefore, we can interact with data at the *Experiment* aggregation level, *Trial*, *Trial-Run*, or *Update* level. The *Performance* component in `Evaluate` contains 7 functions:

- `get_performance` is the lowest level function, used by all other functions in `Performance`. It queries the database, returns a list of POL objects in JSON, where the key is the *snapshot _id* and the value is the *LearnerPerformance* object for that snapshot. The input parameters are as follows: *exp_id*, *collection*, *obj_f*, *obj_adj*, *trial*, *run*, *portion*, *lowest* and *limit.* The *exp_id* and *collection* parameters refer to the name of the database, and the appropriate collection, either the *TrialRun* or *Updates.* The third parameter *obj_f* relates to the objective function used in the query: cost, error or accuracy. The next parameter *obj_adj* or *objective adjective* determines whether the query is run on the *best*, *worst* or *current* fields of the objective function. *Trial* and *Run* are important parameters as they decide the level of granularity for queries. If both IDs passed are null, then the query executes at the *Experiment* level. If only the *Trial* ID is passed, then the query executes at the *Trial* level. Finally, if the *Run* ID is passed along with the *Trial* ID, the query executes at the update level. The *portion* parameter refers to the dataset portion, whether the query is to order results by the performance found on the training, validation or test set. *Lowest* is a Boolean, which defaults to true. This determines how the results are ordered, either ascending or descending. This means queries can be run for gradient ascent and descent, or for those parameters that generated the best *or* worst results. Finally *limit*, determines the amount of documents which will be returned. The *k* best or worst performing results can be returned. Input parameters have default values. The input *obj_f* defaults to ‘cost’, *obj_adj* defaults to ‘current’ and *portion* defaults to 2 which signifies the validation set.

- `get_top_k_ids` is built upon `get_performance`. This takes as input the experiment ID, the type of objective you want to query, *limit* and *lowest*. It returns a list of relevant Learner IDs aggregated at the *Experiment* level.

- `get_top_k_error_scores` requires the experiment ID, the *obj_f* - the type of objective function you want to query - and the result of `get_top_k_ids` as input. It returns a matrix of the relevant performance scores found for the top-k *Learners*, again at the *Experiment* level. These scores are: performance on training, validation and test sets, as well as the training to validation score ratio and the number of *actual* iterations and updates performed for each *TrialRun*.

- `get_all_errors` takes as input the *exp_id*, the *collection* name, the *trial*, the *run*, which can be null, and *type_objective*. If the *run* is null, it queries the trial-runs collection and returns a list of all the final errors for each run in a *Trial*. If it is not null, it returns a list of all the errors found over the course of a series of *Updates* for a particular trial-run.

- `get_sample_level_performance` requires a learner instance, a dataset and makes predictions on the dataset with the learner. It then returns a triple of (*train_errors*, *valid_errors*, *test_errors*) where each is a list with a Boolean indicating if the classification was correct or differences if predicting a continuous value.

- `get_sample_level_time_stats` can only be used for time series predictions. The result of `get_sample_level_performance` is required to compute the Durbin-Watson and Dickie-Fuller statistics, which are returned from this function. These statistics measure if all stationary elements of a time series are accounted for in the error.

- `get_anomalous_indexes` takes as input a vector of free energies and a critical value to multiply against its mean. It returns those indexes that are deemed anomalous through examination of the free energies. It does this by returning sample indexes whose free energies are different to the average energy by more than the critical value times the sample standard deviation, according to Chauvnet's method [113].

The *Parameters* element in `Evaluate` has 4 functions:

- `coalesce_hyper_parameters` receives a list of all the hyper-parameter objects for the top-k *TrialRuns* and combines them in a form for analysis. This function can also generate descriptive statistics for these combinations. These statistics are: the mean, median, standard deviation and percentiles for each hyper-parameter.

- `reduce_search_space` selects those values for each hyper-parameter that are in the top 3 percentiles. The result of `coalesce_hyper_parameters` is taken as input. We demonstrate the reasoning for this method through the experiments presented in Chapter 7.

- `coalesce_model_parameters` takes as input all final *Learners* for a *Trial* and combines all weights found for each *Run* in a *Trial* as a multi-dimensional tensor, where the first dimension is the layer, the second is a node, the third is a node input and the fourth is the weight found at each *Run*.

- `feature_analyse` takes the result of `coalesce_model_parameters` and performs a statistical *t-test* to determine those weights which are significantly different from zero over all runs. This determines which inputs significantly combine in a node, amounting to what an abstract feature in a hidden layer *learns*.

Visualise. Similar to evaluate, the `Visualise` API has two sub-components. These elements are again named `Performance` and `Parameters`. The *Performance* element in `Visualise` has 3 functions:

- `visualise_top_k_learning_curves` takes the *top k ids* found and for each of these k IDs uses `get_all_errors`. It then subsequently calls the next function `visualise_learning_curves` for each.

- `visualise_learning_curves` takes the *train_scores* and *valid_scores* for a particular learner in a *Trial-Run*, generated by its hypothesis function and outputs a plot of mean scores for each epoch iterated over all updates for that particular epoch.

- `visualise_error_distribution` takes the training, validation and test scores at the sample level, found with `get_sample_level_performance` and plots these, to investigate if the error is random.

- `visualise_confusion_matrix` takes the confusion matrix generated for the relevant learner as input and outputs a heat map of the true and false positives.

The *Parameters* component in `Visualise` has 3 functions:

- `visualise_hyper_parameter_distributions` takes hyper-parameter statistics generated with `coalesce_hyper_parameters` and outputs multiple graphs. The graphs are visualisations of the frequency distributions of each hyper-parameter for the the top-k Learners, in the form of histograms, except for the `activations` and `dropout` parameters which are represented with bar charts.

- `visualise_layer_distributions` takes the desired layer, node and input ID, along with the *coalesced* model parameters for a particular trial. It then visualises the distribution of input weights for a particular node in a layer over all runs in a trial. The output is a scatter plot for the distribution of input weights found which link a particular input to a hidden node.

- `visualise_layer_weights` takes a trial ID, run ID and layer ID, retrieves the appropriate Learner and Layer objects and outputs a heat map showing the strength of the weights which connect one layer to another for a single run in a single trial for the relevant layer.

6.4 Summary

While this dissertation motivates the use of deep learning for data mining purposes, Chapters 3, 4 and 5 have shown that these experiments are complex to build, run and ultimately, analyse in order to improve performance and understanding. This motivated a need to create a data model which can capture all aspects of a deep learning experiment. To date, this crucial aspect of deep learning has not been addressed. While this work is ongoing to some degree [94], we believe we have made a significant contribution to this issue by specifying and publishing [102] the first deep learning data model.

In Section §6.1, we presented the POL data model and described its three major components: Experiment, Dataset and Learner. Within Learner, all of the constructs on which to build performance analyses and instantiate a deep learning model with relevant parameters are captured by the data model. There are 2 aspects to experiment performance data. The first is the ability to capture a *snapshot* of the experiment at the required level of granularity. This is presented in Section 6.2.2 where the Deep Learning Database is discussed. As this is a direct image of the POL data model, it can capture a precise point in the deep learning experiment. Essentially, this functionality allows us to pause, store, reload and share deep learning experiments. Thus, we can analyse which hyper-parameter configurations and learning functions worked best, in order to determine optimal configurations. These investigations are only possible through the querying and assessment of interim learners. The second aspect requires the analysis of the deep learner's performance. In Section §6.3, we describe the different collections of functions that use the Deep Learning Database to analyse performance data.

This ability to analyse experiments at different levels of granularity provide us with our toolkit for deep learning. The final stage in this dissertation is to evaluate how the toolkit performs using a variety of data mining tasks and different types of datasets.

Chapter 7

Evaluation

In this chapter, we evaluate our approach to deep learning. Our extensive analysis in previous chapters has shown that utilising deep learning for data mining is majorly complex, involving many intricate components and a steep learning curve. Therefore, the overall goal of this chapter is to demonstrate that deep learning *can* be incorporated into the major data mining activities outlined in the introductory chapter. We demonstrate this through the realisation of the 3 deep learning *application* requirements presented in §1.3.1, our first evaluation criteria. The second component of our evaluation is to demonstrate that the Configurable Deep Network and data model driven Toolkit together provide the supports to achieve the 3 major *framework* requirements: complete *configuration*, *optimisation*, and *interpretation* of the learners and experiment. In §7.1, we provide an overview of the evaluation. The main parts of the evaluation are provided via case studies in §7.2, §7.3 and §7.4, with data mining requirements across multiple datasets, involving different learner types. A final summary is provided at the end of the chapter.

7.1 Evaluation Overview and Setup

Our validation successfully addresses the 4 evaluation criteria outlined in Chapter 1 over 3 case studies. The first case study involves the sports performance (GAA) dataset; the second involves the MAAS dataset, which contains data from a longitudinal study; and the third dataset is part of a large musical dataset.

Table 7.1: Evaluation Goals

Evaluation Criteria		Study 1	Study 2	Study 3
Application				
	Anomaly Detection			✓
	Prediction	✓	✓	
	Representation Learning			✓
Configuration				
	MLP		✓	
	RNN	✓		
	RBM		✓	✓
	StackedRBM		✓	✓
	DBN		✓	
Optimisation				
	Model Parameters	✓	✓	✓
	Hyper-Parameters	✓	✓	✓
Interpretation				
	Model Parameters			✓
	Hyper-Parameters	✓	✓	

Table 7.1 shows an overview of the major evaluation criteria and the components tested in each study. In each case study, we address real world data mining requirements. The requirements for the first 2 datasets were provided by end user collaborators and the third dataset uses an ongoing requirement for musicians. Each requirement relates to one or more of the deep learning *application* goals presented in §1.3.1. The *domain agnostic* (requirement c2) nature of our tool set is demonstrated through the achievement of these 3 *application* requirements in 3 heterogeneous areas. This also demonstrates the generic and wide application of our research.

We next evaluate the *configurable* and *optimisable* nature of the CDN. It is applied to different combinations of the 5 neural networks presented in Chapter 3. It uses a homogeneous interface to automatically generate heterogeneous network configurations according to *all* hyper-parameters. It then optimises the resultant model parameters accordingly. Thus, we also successfully validate *automated optimisation of all hyper-parameters* (requirements a5 and c4). Figure F.1 in Appendix F provides an example of this. The described functionality demonstrates the simple, flexible and generic principals which guided the systems design.

The final aspect, is to evaluate the DNT's utility for *generic* experiment *interpreta-*

tion. §7.2 focuses on the sports performance (GAA) dataset and hyper-parameter optimisation - via model selection - and interpretation (Appendix F Figure F.2). These are generated from a Recurrent Neural Network experiment for time-series predictions. §7.3 explores survival analysis predictions for the MAAS dataset. In this section we also investigate hyper-parameter optimisation and interpretation. For this, we utilise Multi-Layer Perceptrons, Restricted Boltzmann Machines (RBMs) and Deep Belief Networks (DBNs). We then explore anomaly detection, model-parameter (feature representation) interpretation (Appendix F Figure F.3) and briefly, hyper-parameter optimisation, for the Bach music dataset in §7.4. The Bach experiment utilises RBMs and Stacked RBMs.

All experiments and learners are easily *reproducible* as they are stored according to a well defined data model in an interoperable format. Furthermore the are easily *portable* as the largest experiment is just over 1.5GB, uncompressed.

Experimental Environment. Experiments were run on Ubuntu 14.04.4 LTS and a NVIDIA GeForce GT 620 810MhZ Graphical Processing Unit. The latest code was developed using 64-bit Python 2.7.6 in PyCharm 2016.2.3. IDE. Experiments use NumPy 1.9.2 [131], Theano 0.7.0 [17], Pandas 0.18.0 [91], PyMongo 2.6.3 [95] and their dependencies. Mongo 3.2.11 stored JSON models and for ad-hoc querying Robomongo 0.9.0 was used.

Validation and Sampling Procedures. Shallow experiments use nested cross-validation [27]. We also adopted the nested cross-validation protocol for time-series experiments in order to preserve temporal dependencies and contextual information when performing back-propagation through time. This gives a more accurate measure of error, as testing on data used to train a Learner gives an upwardly biased estimate of performance. For deep experiments on non-sequential data and in order to get a robust estimate of parameter and performance variance, bootstrap resampling *in conjunction* with nested cross-validation was used. For each experiment *Trial-Run*, the indices which relate to the samples of the dataset used, are fixed. Bootstrap resampling consists of selecting a random sample *with replacement* from the dataset to contain the same number of samples as in the original dataset.

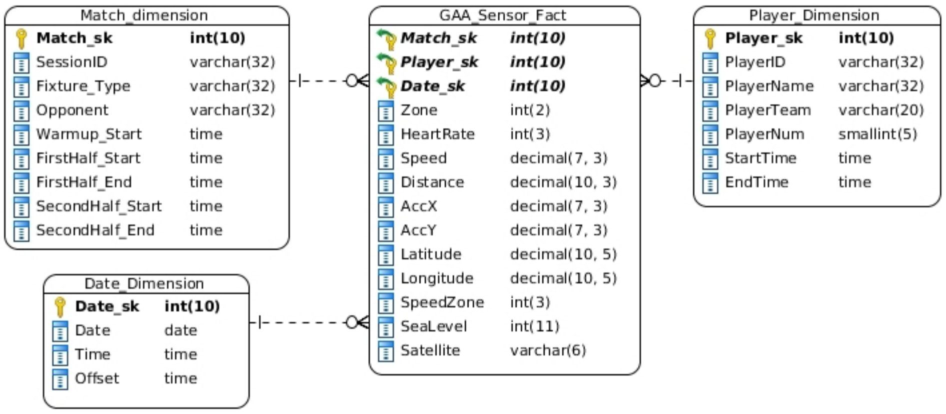


Figure 7.1: Sports Performance Star Schema

7.2 Case Study 1: Predicting Heart Rates

The user requirement in the first case study uses the sports performance (GAA) dataset to *predict* future heart rates using past biometric markers. In order to meet this evaluation, it is necessary to select optimal hyper-parameter configurations. As a result, the analysis presented here examines the distribution and tendency of top performing hyper-parameters when a Recurrent Neural Network was trained for heart rate prediction 5 seconds in the future. In Figure 7.1, the features for the Sports Performance (GAA) dataset are displayed in their native schema.

Sensor values for 1 player in 1 match were selected, giving 81,165 instances. As sensors generate at least 10 sets of measures per second, we average these to obtain the desired level of granularity of 1 measure per second, resulting in almost 8,000 instances. Only relevant performance data was selected, beginning at pre-match warm-up and ending 4 minutes and 30 seconds after the match termination, leaving a final tally of 6,211 instances. For these experiments 4 out of the 28 features presented in Figure 7.1 were selected for the player. These input attributes were x and y accelerometer values, *distance covered* and *heart rate* - all continuous measures. The target prediction variable is the *heart rate* value 5 seconds into the future.

Table 7.2 shows the *hyper-parameter search space* for experiment *gaa_rnn_05-04-16_11:48:02*, or upper and lower bound searched for each hyper-parameter. This experiment optimised an RNN on the subset of the sports performance data pre-

Table 7.2: gaa_rnn_05-04-16_11:48:02 Hyper-Parameter Search Space

Hyper-Parameter	Description	Bounds (low, high)
b_size	samples in a mini batch	(60, 600)
l_hidden	number of hidden layers	(1, 1)
o_nodes	number of hidden layer nodes	(1, 10)
learning_rate α	co-efficient for weight updates	(0.0001, 0.9)
max_updates	max possible updates to perform	(10, 10000)
truncate_gradient	number of time steps to BP errors	(5, 100)
activation	hidden layer activation	(relu, logistic)

viously described. 90 hyper-parameter configuration *Trials* were performed with 2 *Runs* for each trial. Model-parameters were optimised with MSGD and Early Stopping [112], [23].

We now examine the bounds presented in Table 7.2. For *batch size*, we set the bound to between roughly 1% and 10% of dataset instances. Too small a batch would not take advantage of matrix multiplication speed-ups and too large a batch, could not benefit from MSGD's ability to escape poor local minima. The number of *hidden nodes* ranges from 1 to $2n + 1$ where n is the number of input features. Using more nodes is likely to over-fit the data. *Learning rate* boundaries are in line with those in the literature [16]. The boundaries for *max updates* were again chosen according to dataset size. This was an earlier experiment that used updates instead of epochs. It was estimated the lower bound would very likely under-fit the data whereas the higher end would over-fit. For the number of seconds for which errors are back-propagated, *truncate gradient* was initialised to be between 5 and 100. We estimated that this should be enough relevant contextual information for prediction and anything more would likely result in a vanishing gradient. There is a choice of two *activation functions*: logistic or Rectified Linear, corresponding to probabilistic and numeric outputs, respectively. Similar heuristics were used for choosing hyper-parameter bounds across all experiments. The bounds for *l_hidden* were both set to 1 as this particular network has 1 hidden layer. Dropout and regularisation were not used for this experiment.

The *configurable* nature of our framework is demonstrated by Table 7.2. There is significant *variety* in terms of different neural networks that can be generated within these search bounds. Using the POL data model, the only inputs required are: *HyperParamSearchSpace* type, the algorithm type and the number of *Trials* and *Runs* desired. The system easily configures and trains *180* total RNN models, allowing us to optimise model parameters *and* hyper-parameters at the same time. We now use our Toolkit for hyper-parameter analysis and optimisation, determining how top-k hyper-parameters are *distributed*.

7.2.1 Predictions and Hyper-Parameter Optimisation

To perform the analyses, we utilise four functions from our APIs: `get_top_k_ids` and `get_top_k_error_scores`, from the *Performance* component in the *Evaluate* API, `get_entity` from *Read* in our *Access* API and `coalesce_hyper_parameters` from *Parameters* in our *Evaluate* API. For 10 out of the top 20 configurations, we now explore the results of `get_top_k_error_scores`. This enables us to better understand how each learner model performed.

Table 7.3 shows the training (*t_score*), validation (*v_score*) and test (*tst_scores*) performance associated with the top 10 out of the top 20 hyper-parameter configurations analysed in this section, at the *Experiment* level of aggregation. It also shows the ratio of training to validation scores in the *t_v_ratio* column. All scores were returned with `get_top_k_error_scores` and rounded to two decimal places for presentation. For the *t_score*, *v_score* and *tst_score*, values closer to zero are better. For *t_v_ratio*, scores closer to 1 are better, as this means the training and validation scores are close. The scores in this case are the result of the mean squared error cost function.

The top 10 ranked hyper-parameter configurations shown in Table 7.3 are identified by the *Trial-Run* combinations. Trial-Run 25-0 was top ranked for the validation score and the training to validation ratio. In the top 10, the 8th ranked Trial-Run 18-1 had the best training score and 7th ranked Trial-Run 24-0 had the best test score. Multiple validation scores are 1.60 due to rounding. Although the top ranked 25-0 does not have the best training score, the *ratio* of training to validation is also the best of the Experiment. As the values for both datasets are close, it suggests the model has not overfit the data. The test score of 7th ranked 24-0 is the best

Table 7.3: Case Study 1: Top 10 Scores

rank	trial	run	t_scores	v_scores	tst_scores	t_v_ratio
1	25	0	1.32	**1.53**	1.79	**0.86**
2	73	0	1.25	1.57	1.61	0.80
3	80	0	1.13	1.58	1.68	0.71
4	53	0	1.21	1.59	1.71	0.76
5	18	0	1.15	1.59	1.59	0.72
6	73	1	1.24	1.60	1.67	0.77
7	24	0	1.15	1.60	**0.99**	0.72
8	18	1	**1.13**	1.60	1.59	0.70
9	89	1	1.17	1.60	1.31	0.73
10	80	1	1.14	1.60	1.61	0.71

but we cannot select hyper-parameters according to the test set as this would give an upwardly biased estimate of performance. This motivates the need for another measure for model selection, rather than just the validation score alone.

7.2.2 Hyper-Parameter Interpretation

Table 7.4: Hyper-Parameter Summary Statistics

ID	**alpha**	**b_size**	**nodes**	**max_updates**	**updates**	**truncate**
mean	0.404	289.240	4.360	3966.870	3520.900	53.980
std	0.284	178.533	2.765	2704.307	3569.484	26.251
min	00.5	68	1	132	30	6
25%	0.139	118	2	1394	326	33
50%	0.373	242	4	3729	2230	55
75%	0.655	451.750	6.250	6332.000	7332.000	74.250
max	0.889	619	10	9899	9932	100

Table 7.4 shows summary statistics for a sample of the Top 20 performing hyper-parameter configurations of 180 *Trial-Runs*, namely the mean, standard deviation, minimum and maximum, at the 25th, 50th (median) and 75th percentiles. This again is at the *Experiment* level of granularity. The statistics presented in Table 7.4 provide valuable information which can be used in subsequent functions but in order to understand the meaning of each of these values and to *understand* optimal hyper-parameter configurations, visualisations are more interpretable. Hence, in Figures 7.2 to 7.7 we present histograms of the frequency distributions for each hyper-

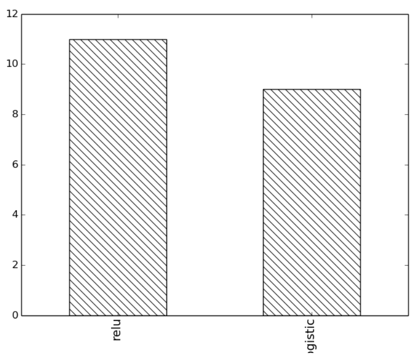


Figure 7.2: Activations

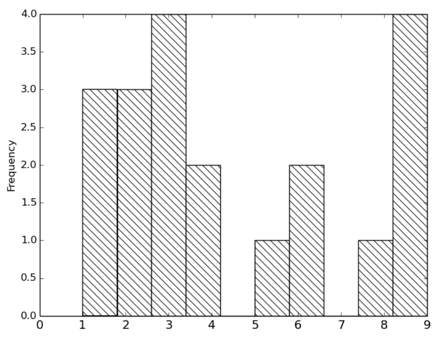


Figure 7.3: Hidden Nodes

parameter. To generate these visualisations the `visualise_hp_distributions` function in the `Parameters` component of our `Visualise` API was required. We present and examine the results for each hyper-parameter individually for ease of presentation and to avoid confusion.

Activations. Figure 7.2 is a simplistic graph included here only for completeness as it shows the *activations* of the Top 20 returned learner models. Activations are categorical strings and therefore require different analysis to numeric hyper-parameters. As can be seen from Figure 7.2, the count of models with Rectified Linear Units is close to those with Logistic activations. This result suggests both activations have similar performance but as Relu outweighs logistic 11:9, Relus are deemed to be the higher performing activation function.

Hidden Nodes. Figure 7.3 shows the frequency distribution for the top 20 hidden node counts. From Table 7.4 can see the average number of hidden nodes is 4.5 and the median is 3.5. A median value less than the average signifies a right-skewed distribution. Therefore, in distributions such as this, the median is a better measure for the central tendency of the values being measured. We found all hyper-parameter distributions to contain such a skew. Given a median of 3.5, over 50% of the hidden node counts for the top 20 Learners is less than 4. This is shown in the frequency distribution plot in Figure 7.3 and would suggest that there are less than 4 abstract classes which, in this case, *relate* the input to the output.

Gradient Truncate. Examining Figure 7.4, we can see the best performing values for *truncate gradient.* Table 7.4 shows the mean and median to be 65.7 and 64.5

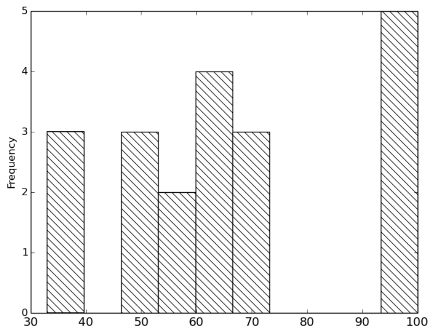


Figure 7.4: Gradient Truncate

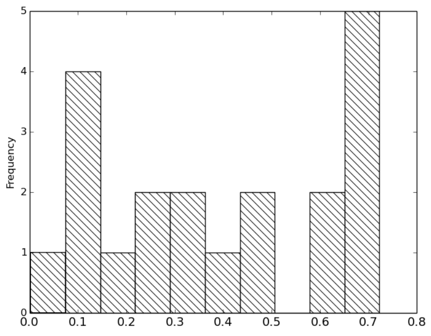


Figure 7.5: Learning Rate

respectively. These values again signify a right-skewed distribution. There are two possible analyses for why the distribution is centred at these values. The first is that time-points $t_{i-65} : t_i$ have the greatest impact on t_{i+10}, suggesting all activity for a minute leading up to a particular time likely has the greatest effect on future heart rate. The second is that for time-points > 65 the gradient disappears but this is unlikely as there was also good performance found for this hyper-parameter in the range of 90 to 100.

Learning Rate Alpha. For learning rate, only one configuration had a value below 0.1 in the Top 20, all others were greater than 0.1 and centred around 0.3932, as Table 7.4 shows. The median again was below this at 0.3453. Both figures are quite large for a learning rate and suggest that the cost function is quite steep and minima are approached fast.

Batch Size. Table 7.4 and Figure 7.6 shows the maximum batch size which occurs in the top 20 is 558 and the minimum is 73. This is a larger spread than other hyper-parameter values. By again examining the mean and median of 233.85 and 145.5 respectively, we see that the distribution is again skewed. Figure 7.6 shows that the most frequent batch sizes in the top 20 are at or around the lowest bound of the search space, suggesting that smaller batches allow gradient descent to better model the cost and escape local minima easier.

Updates. An analysis of the *actual updates performed* (Figure 7.7) shows that when *early stopping exited* is more informative than examining the *max updates* parameter. This is due to the fact that it gives a greater impression on how iterations

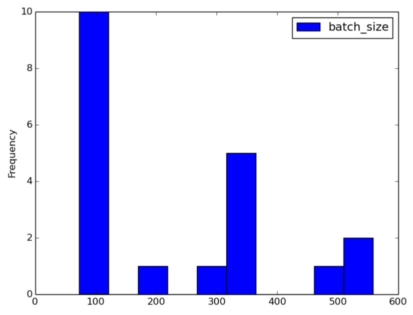


Figure 7.6: Batch Size

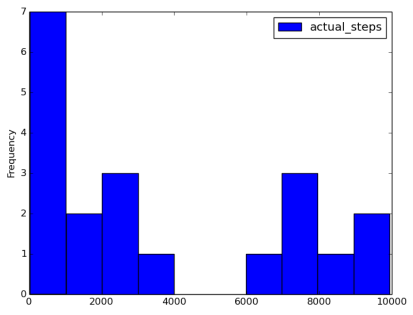


Figure 7.7: Updates

effect training, showing when the training process actually exits. This analysis is verified by the exiting epochs and iterations, as they exit early, compared to the number of iterations possible. For example, the median and mean of actual iterations are 2230 and 3520.9 respectively, much lower than the upper bound. This means that gradient descent was steep and fast, as was previously evidenced by large learning rates.

The above analyses show that the median better represents the central tendency of all parameters. Skewed distributions do not lend themselves to a parametric analysis. Therefore, our selection methodology for reducing the bounds of a *HyperParamSearchSpace*, consisted of taking the median to represent the central tendency and taking those values within the first three quartiles to be the new search bounds. In effect, the minimum becomes the new lower bound and the value for the 75th percentile becomes the new upper bound.

7.2.3 Summary for Case Study 1

In summary, what does all of this mean for our requirement to predict future heart rates? The answer is that we successfully *optimised* hyper-parameters and *analysed why* these results are optimal for predicting future heart rates. This demonstrated the effectiveness of the DNT for helping to interpret hyper-parameters and furthermore, we show that it gives us an *insight* into how the hyper-parameters effect the training process and thus, lead to further optimisation. We also demonstrated that an RNN can be successfully trained on the sports performance dataset and use sequential, time-based information to make contextual predictions.

In this case, these analyses are still possible without the toolkit but much more difficult and time consuming. Our configurable system together with the toolkit allow for the selection of the top performing hyper-parameter configuration and associated learner instance for deployment to predictions and for reducing the search space to run further experiments. We will analyse learner models found for prediction and query the updates of *past* learners in §7.3.

7.3 Case Study 2: Predicting Patient Survival Rates

Clinical research colleagues through their research, identified a number of *modifiable* dementia risk factors. Once identified, they wanted to explore the possibility of predicting dementia risk a number of years in the future, based on a baseline measurement, for middle aged individuals. This would validate an approach where an intervention in middle-age to *modify* peoples behaviour could lower their associated dementia risk in later life.

In effect, there are 2 case studies used in the second evaluation but as there is an element of overlap and both use the MAAS dataset, we present the evaluation and findings together. The first part of the evaluation (Case Study 2A, presented in 7.3.1) seeks to construct accurate learner *models* that provide learner models for *prediction* in the second case study. In effect, this represents *hyper-parameter optimisation* and uses MLPs and DBNs. The second part of the evaluation (Case Study 2B, presented in 7.3.2) is a time-based prediction which concerns survival analysis. Here, we use measurements at a single time-point, to predict survival probabilities at various time points in the future. The output of the first case study, where optimal hyper-parameter configurations are identified, meaning whether a DBN or MLP performs best is then used in the second case study for survival analysis, a sequential prediction task.

Features. Our dataset was a subset of the full MAAS data constrained to middle aged individuals (those who were 40+ at baseline), and measurements relating to relevant *modifiable* risk factors. This resulted in *955 instances*. At the time of

Table 7.5: Features used for MAAS experiment

Feature	Data Type	Description	n_missing	%missing
p_id	num.	Anonymised ID	n/a	n/a
age_risk	cont.	Risk age and gender	0	0
ed_risk	cont.	yrs_ed. dem. risk	14	1.47
cog_activ50	bin.	Cognitively active @50+	19	1.99
hrs_phys_act	disc.	Hours physically active/day	24	2.51
bmi	cont.	Body mass index	2	0.21
diabetes	bin.	Participant has diabetes	0	0.0
mod_alcohol	bin.	Moderate alcohol intake	44	4.61
smokes	bin.	Smokes cigarettes	6	0.63
depress_scl	disc.	SCL depression score	41	4.29
hypertense	bin.	High blood pressure	15	1.57
cholesterol	bin.	High cholesterol	0	0.0
cvd	bin.	Cardiovascular disease	0	0.0
kidney	bin.	Kidney disorder	0	0.0

writing, we are not aware of other studies that have used a Deep Belief Network to perform *deep* survival analysis on a longitudinal ageing study.

Table 7.5 shows the features for the case studies in this domain, relating to the *modifiable* dementia risk factors. It contains the name (*Feature*) and *Data Type* of each feature. The data types are: numeric (num.), binary (bin.), discrete (disc.), or continuous (cont.). In most cases we were only provided with binary features, as measurements to be performed on middle aged individuals were only clinically validated for dichotomous cut-offs. Where possible, continuous and discrete features were used as previous work has shown their importance in this context [104]. We pre-processed and normalised those features marked as discrete (disc.) and continuous (cont.). This ensured values between 0 and 1, enabling their input into the RBM and DBN training processes. The RBMs treat these values as binary probabilities, which works well in practice [14]. Table 7.5 also shows the count (*n_missing*) and percentage missing (*%missing*) for each feature. As missing values were low in number, they were filled with the feature average. The workflow and processes described in Chapter 4 were used to identify all data types, ensure homogeneity of variables and identify missing data. The *age_risk* and *educ_risk* features are actual dementia relative risks for age, sex and education level. Although non-modifiable,

Table 7.6: Survival Overview

Time	At Risk	Hazard	Censored	p_t	S_t
t=0	955	8	159	0.991623	0.991623
t=6	788	17	224	0.978426	0.97023
t=12	547	37	na	0.932358	0.912244

these elements were provided due to their importance in predicting dementia in later life.

Survival Target Variable. To perform survival analysis, we require the number of individuals at each study time point (baseline, 6 years and 12 years) who were at risk, developed dementia (encountered *hazard*) or dropped out of the study or died without developing dementia (were *censored*). Table 7.6 shows the result of this analysis in the *At Risk*, *Hazard* and *Censored* columns. Utilising the methodology from [32], our output or *target* variable was then a vector of 1s and 0s where a 1 is found until the hazard - dementia - occurs. For, example if a participant encountered the hazard at the third time-point or beyond, the target vector would be: $\{1, 1, 0\}$. For censored individuals - those who dropped out of the study or died without getting dementia - we required the use of the Kaplan-Meier estimate of survival [32], a non-parametric estimate of the probability of survival to a particular time-point and beyond.

$$S(t) = p(t) \cdot S(t-1) \tag{7.1}$$

$$p(t) = \frac{n_t - h_t}{n_t} \tag{7.2}$$

$$n_t = n_{t-1} - c_{t-1} - h_{t-1} \tag{7.3}$$

The calculation for the cumulative Kaplin-Meier statistic is shown in Equation (7.1), where t is the point under investigation, $p(t)$ is the probability of survival at the current time-point and $S(t-1)$ is the cumulative survival probability up to the previous time point $t-1$. The calculation of $p(t)$ is shown in Equation 7.2. The number of people who encountered the hazard h_t is subtracted from the number of people at risk n_t for the current time t which is then divided by n_t. Those at risk, n_t for time-point t, is calculated with Equation (7.3). It is the result of subtracting the number who encountered the hazard h_{t-1} and the number who were censored c_{t-1} from those at risk n_{t-1} in the previous time-point $t-1$. Table 7.6 shows the result of these calculations in the p_t and S_t columns. An example of a target vector for someone who survived up to the first and second time-points but was censored for the third is: $\{1, 1, 0.912244\}$.

Search Space. Table 7.7 shows the hyper-parameter search space for these case studies. As the number of instances in the dataset is quite low, we chose a batch size (b_size) to reflect this, although still between roughly 1% and 10% of the total dataset size. In terms of hidden layer nodes, we chose a minimum of 5, which would compress the inputs and a maximum of 50, more than 3 times the number of input features, which would likely overfit, but is still in a sensible range. This gives the search optimisation the opportunity to extract the best configuration. The bounds for the regularisation parameter λ are taken from those values commonly cited in the literature. Dropout has only two possible options, true or false. We use an upper bound of 0.1 for the learning rate α, as in deep architectures the gradient tends to vanish for large values of α, we adjusted the lower bound to reflect this decrease in the upper bound. *Patience* is an eary stopping parameter, which directly relates to the *minimum* number of epochs to perform. For deep experiments we will constrain the type of activation function to be logistic as this is necessary to run our RBM and DBN training process. The number of hidden layers was set to between 1 and 3. This allows us, due to the highly *configurable* nature of our CDN, to *compare* deep and shallow learning as part of a single experiment optimisation. 1 hidden layer is shallow and more than this is deep. Given the number of instances in the dataset, more than 3 layers would likely have too many model parameters and overfit the data.

Our system was used to generate 128 *Trial* configurations within the search space for hyper-parameters presented in Table 7.7. For each configuration, 1 model was optimised. Therefore, there are 128 Trials and 1 Run for each Trial. Both L1

Table 7.7: Hyper-Parameter Search Space

Hyper-Parameter	Description	Bounds (low, high)
b_size	samples in a mini batch	(2, 100)
l_hidden	number of hidden layers	(1, 3)
o_nodes	number of hidden layer nodes	(5, 50)
regularisation λ	co-efficient for regularisation	(0.0001, 1)
dropout	whether to perform dropout	(True, False)
learning_rate α	co-efficient for weight updates	(0.00001, 0.1)
patience	minimum number of epochs	(10, 50)
activation	hidden layer activation	(logistic)
max_epochs	maximum epochs to iterate	(150, 1000)

and L2 regularisation was added to the cost function. For this experiment, interim *snapshots* were written to the database every *10th* validation. The dataset was split into 70% for the training, 15% for validation and 15% for testing. As we previously discussed because of the hidden layer search space our approach allows us to test both DBN and MLP configurations, shallow and deep over the course of a single experiment. Storing interim learner instances generated at the *pre-training* stage along with testing 1 or more hidden layers also allows for, if required, the analysis of single layer and stacked RBMs, all in a single experiment. This further emphasises the natural flexibility and benefits of performing deep learning experiments with our configurable system (the CDN) and the toolkit (DNT).

7.3.1 Hyper-Parameter Optimisation and Interpretation

The toolkit functions used to generate the results are `get_top_k_ids`, `get_entity` and `coalesce_hyper_parameters`. These analyses are at the *Experiment* level of granularity. For presentation purposes, we split the top performing hyper-parameter configurations - the result of `coalesce_hyper_parameters` - into two tables. The first, Table 7.8 shows the top 10 performing *architectural* hyper-parameter configurations. The second, Table 7.9, shows associated training hyper-parameters. The associated top-k error scores for these configurations are examined in §7.3.2.

Deep Learner Architectural Results. Table 7.8 shows the *trial_id* of the configuration, the number of hidden layers (*l_hidden*) and the number of nodes in the

Table 7.8: MAAS DBN Top Architecture Hyper-Parameters

trial_id	l_hidden	o_nodes1	o_nodes2	o_nodes3
24	1	29	na	na
54	3	25	23	10
2	3	24	44	40
20	2	34	28	na
84	3	22	48	45
56	1	31	na	na
32	2	36	22	na
97	1	20	na	na
74	3	14	34	26
51	3	32	27	29

first (*o_nodes1*), second (*o_nodes2*) and third layers (*o_nodes3*) where applicable, for each of the top performing configurations as measured on the validation set. Table 7.8 shows that the top performing learner configuration had 1 layer, a *shallow* learner. The next four were *deep* with 3, 3, 2 and 3 layers, respectively. Of the final five, 2 were shallow with 1 layer and 4 were deep with 3, 2, 3 and 3 layers. The top performing Trial configuration, Trial 24, had 29 hidden nodes, over 2 times greater than the number of input features. All the top performing configurations had node counts in the first hidden layer *greater* than the number of input nodes. All hidden node counts were found to be large, for example, Trial 2 had 24, 44 and 40 nodes in its hidden layers.

Deep Learner Architectural Analysis. From Table 7.8, we see that a shallow architecture performs *best* overall. However, when we aggregate scores across the top 10 performing configurations, 7 are deep with layer counts of 2 or above and 3 are shallow. If we extend the analysis beyond the top 10 to the top 20 configurations, as shown in Figure 7.8 - generated with `visualise_hyper_parameter_distributions` - there are 14 deep configurations and 6 shallow. This could mean that deep are better overall and could warrant a second round of experimentation, but for this analysis we determine shallow to be the most optimal. Given the size of the dataset, this makes sense as deep networks could perform better for the unsupervised aspect, in describing the data, but it is likely in a dataset of this size that there are too many parameters in deep configurations not to overfit, when *relating* the input to

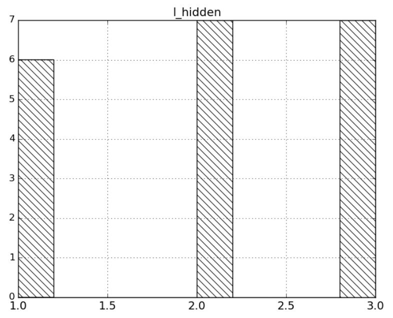


Figure 7.8: Top 20 Hidden Layers

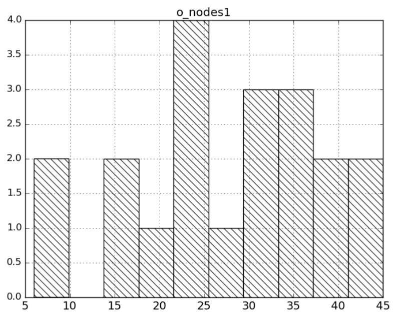


Figure 7.9: Top 20 Layer 1 Nodes

the output predictions.

Examining the hidden node counts, it is surprising that a higher number of hidden nodes outperforms smaller numbers, as it was expected a smaller node count would perform better. However, this could be evidence that the risk factors have a very large number of ways in which they can combine, interact and cluster when attempting to predict a complex outcome like dementia, and the probability of surviving without developing the condition, especially 12 years into the future. Figure 7.9 shows that when we extend our analysis to the top 20 configurations, the optimal layer 1 node configurations cluster from roughly 22 up to 45, with a spike in values around 24. This is interesting because it is twice the number of nodes in the dataset and suggests that there are at least twice as many latent classes which relate the input to the output. In the next part of the case study, we show learner models with dropout perform better than those without dropout. This could be another reason for the high instances of hidden nodes, ensuring a sparse representation is learned for prediction.

Deep Learner Training Hyper-Parameter Results. Table 7.9 shows top performing training specific hyper-parameters. Values are rounded to 3 decimal places. The learning rate is given in the *alpha* columns, whether or not dropout was used by the *dropout* column and the strength of regularisation given by the *lmbda* columns. The batch size is shown in *b_size*, the patience parameter for the minimum number of epochs to iterate by the *pat.* column. Finally, the *updates* and *epochs* columns show the actual number of updates and epochs iterated, respectively, whereas *max_e* shows

Table 7.9: MAAS DBN Top Training Hyper-Parameters

rank	trial	alpha	dropout	lmbda	b_size	pat.	updates	epochs	max_e
1	24	0.006	TRUE	0.467	83	50	6248	781	781
2	54	0.099	FALSE	0.661	95	31	1624	232	947
3	42	0.069	TRUE	0.490	69	38	4590	510	735
4	20	0.031	TRUE	0.475	9	18	6660	90	564
5	84	0.033	TRUE	0.322	83	50	2896	362	858
6	56	0.076	TRUE	0.332	74	30	2700	300	801
7	32	0.063	TRUE	0.935	19	14	4410	126	544
8	97	0.082	FALSE	0.066	74	41	1719	191	191
9	74	0.024	FALSE	0.803	83	31	2016	252	974
10	51	0.061	TRUE	0.520	22	12	4980	166	383

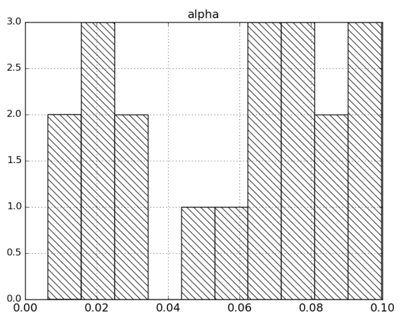


Figure 7.10: Top 20 Alpha

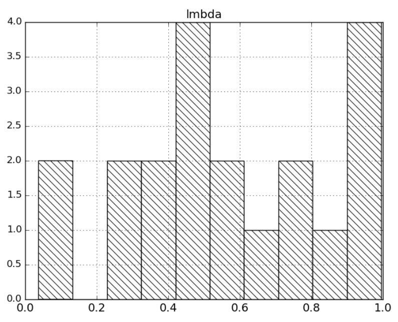


Figure 7.11: Top 20 Lambda

the input hyper-parameter of the maximum allowed epochs. Examining certain visualisations, generated by `visualise_hyper_parameters`, while referring back to Table 7.9 provides for more interpretable results. For presentation purposes, we only include the top 10 results in Tables 7.8 and 7.9. *Visualisations* presented extend to the *top 20*, thus providing a larger distribution of hyper-parameter results for analysis.

Table 7.9 shows that 7 of the top 10 configurations performed best *with* dropout. Figure 7.10 shows that there are two clusters of high performing settings for *alpha*. Table 7.9 shows that although the allowable range was from 0.0001 to 0.1, of the top 10 values, 9 are between 0.02 and 0.09. However, the top performing learning rate is much lower at 0.006. This is reflected in Figure 7.10 where the highest cluster of values is between 0.06 and 0.09, but there is also a cluster at 0.02. Figure 7.11 shows that the distribution of regularisation coefficients *lamda* is relatively uniform

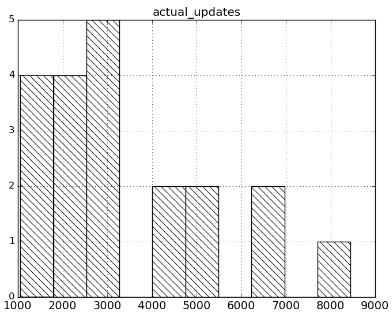


Figure 7.12: Top 20 Updates

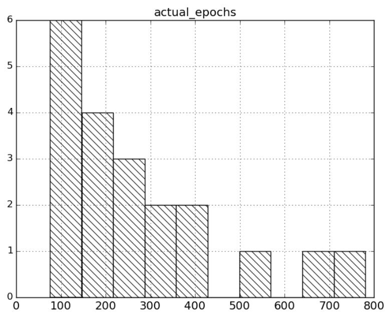


Figure 7.13: Top 20 Epochs

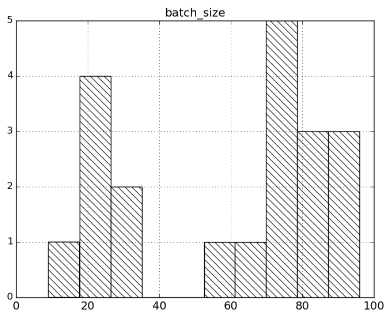


Figure 7.14: Top 20 Batch Sizes

but has peaks at roughly 0.5 and close to 1. This is also seen in Table 7.9, where the top performing *lambda* is 0.467. Furthermore, 9 of the top 10 values for lambda are above 0.3 with only one smaller parameter of 0.06, although the lower bound for this parameter was *0.0001*.

Figures 7.12 and 7.13 show the frequency distributions of *updates* executed and *epochs* iterated. Both distributions centre a the lower end, where 13 out of the 20 *updates* are roughly below 3000 and 17 of the 20 *epochs* are roughly below 400. This is also reflected in Table 7.9, in aggregate. In contrast, these parameters for the top performing Trial 24, are at the upper end of the distribution, iterating up to the maximum number of epochs of 781 and executing 6248 updates. Figure 7.14 shows the top 20 batch sizes. These results cluster at two ranges, between 10 and 30 and between 50 and roughly 95. Trial 24 in Table 7.9 has a batch size at the upper cluster of 83.

Deep Learner Training Hyper-Parameter Analysis. It is interesting that in many ways the top performing configuration - Trial 24 - has some parameters which highly contrast the general and aggregate trend of all top 20 configurations. It has a *lower* learning rate, is *shallow* and executes a much *higher* number of updates. Other parameters for this Trial are commensurate with the general trend. For example, it utilised dropout, has high levels of regularisation and employs a large batch size. Our analysis is that at some point in the training process, the lower learning rate and particular configuration of its other hyper-parameters, allowed it to enter into a deeper cost function 'valley' or 'hole' that was overshot by other configurations, to achieve higher levels of predictive accuracy. This would also explain the much higher number of updates executed. Once a particular local minima has passed, a higher number of updates were necessary to proceed down to a deeper point in the landscape of the cost function, ultimately achieving a lower cost and increased accuracy. Generally, in the top 20, high λ *regularisation co-efficients* worked better. High values for this parameter suggest that lower values for weights and biases improve predictive accuracy. As the dataset is relatively small and the architectural configurations of the DBN and MLP networks are complex, containing many parameters, high-values for regularisation make sense. Constraining model parameters to lower values, gives a simpler learner model, despite complex architectural configurations. This allows the learner model to continue to generalise well and predict unseen data, despite low instance counts. Although simpler in terms of smaller model parameter weights, we still benefit from the gains of learning layers of abstract feature representations. These abstract features better relate the input data to the outcome to be predicted. Generally, the actual *epochs* and *updates*, centre at lower values. Considered with the higher values of the α *learning rate* suggests that for these configurations, MSGD was steeper and faster, where early stopping caused the learning process to terminate before the max epochs parameter was reached. This means that these models could likely overfit if the training procedure continued. For these configurations, lower cost values could not be found but training still exited when their optimal predictive accuracy was found. Finally, 13 of the top 20 *batch sizes* clustered at higher values. This suggests that including higher levels of information in each update, from larger swathes of data, allows the training process to better model the cost and find lower costs, improving predictive accuracy.

To test and optimise the high volume of hyper-parameters used in this evaluation is a complex process and highly cumbersome if the means to automatically configure and optimise many learner model instances does not exist. Our CDN easily enabled this. Furthermore, the means to analyse the resulting top performing hyper-parameter configurations is enabled automatically and in a standardised form with the storage model and analytical capabilities of the DNT. The analyses presented here are possible without our toolkit but are far more difficult. Furthermore, we have successfully identified that in this case, a *shallow* architecture (Trial 24) performs best. This quickly informs the data miner that employing the highly complex deep learner will not yeld better results. We will now explore predictive accuracy at the *Experiment* level to confirm this, along with the *Trial-Run* and *Update* levels of aggregation to determine if the final learner model found, produces the most accurate predictions achieved over the course of all updates for that learner instance.

7.3.2 Prediction Analysis

Experiment Level Predictions. Table 7.10 shows the training (*t_score*), validation (*v_score*) and test scores (*tst_scores*) associated with the hyper-parameters presented in Tables 7.8 and 7.9. It also shows the ratio of training to validation scores in the *t_v_ratio* column. These results were returned with `get_top_k_error_scores`. All values are rounded to 4 decimal places for presentation purposes, so some values appear the same but differences are found at higher levels of precision. The best training to validation ratio is that with the score closer to one, whereas for the other scores, values closer to zero are better. As with hyper-parameter results, the configuration with the lowest score on the validation set is considered the best. The scores quoted here are found with mean squared error cost as presented in Chapter 3.

Deep Learner Experiment Level Prediction Results. Table 7.10 shows the the best validation score 0.0286, found in Trial 24. The best test result is 0.0286, also found at Trial 24. The best training score is 0.0598, found in Trial 32, which

Table 7.10: MAAS DBN Top Performing Scores

rank	trial_id	t_scores	v_scores	tst_scores	t_v_ratio
1	24	0.0802	**0.0286**	**0.0286**	2.8036
2	54	0.0791	0.0355	0.0355	2.2282
3	2	0.0791	0.0382	0.0382	2.0726
4	20	0.0818	0.0400	0.0380	2.0467
5	84	0.0776	0.0436	0.0435	1.7797
6	56	0.1093	0.0447	0.0448	2.4451
7	32	**0.0598**	0.0460	0.0461	**1.2989**
8	97	0.1227	0.0469	0.0469	2.6126
9	74	0.0875	0.0526	0.0526	1.6640
10	51	0.1004	0.0535	0.0535	1.8760

also resulted in the best training to validation ratio of 1.2989.

Deep Learner Experiment Level Prediction Analysis. The training to validation ratio for Trial 24 is 2.8036. This identifies what appears to be a larger gap between these scores, compared to other configurations. This could evidence a certain level of overfitting although this Trial configuration also generalises best to test data. In contrast, Trial 32 has a validation score and test score almost double that of Trial 24, but its training to validation ratio is much closer to 1. Thus, through this analysis we identify Trials 24 and 32 to warrant furhter investigation. To do this, we can use the DNT to analyse interim learners and interrogate which is in fact the best. Therefore, subsequent examinations will focus on Trial 24 and 32 at the Trial-Run level of aggregation. These further investigations are *not* possible without the storage of interim learner instances, to which only our DNT provides access.

Deep Learner Trial-Run Level Prediction Results. The results presented here are found with `get_all_errors` which queries *Updates* of Trial-Runs 24 and 32. In reality, these are Trial-Run 24-0 and 32-0, but as there is one Run per trial in this case, we omit the Run ID. Once the result of all model parameter updates is returned for both Trial-Runs, we employ `visualise_learning_curves` to enable easier analysis. Figures 7.15 and 7.16 show the learning curves returned for all *snapshots* captured in Trials 24 and 32. These are *fine-tuning* snapshots as this is the procedure relevant to analyse the predictive aspect of the DBN. The

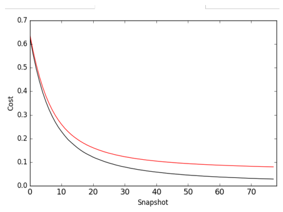


Figure 7.15: Trial 24 Learning Curves

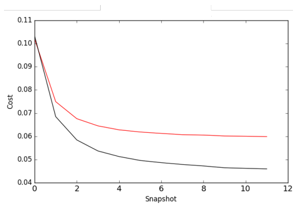


Figure 7.16: Trial 32 Learning Curves

learning curves appear smooth as *snapshots* were captured every 10th validation. If snapshots were taken at every validation, the curves would likely appear jagged. The y-axis shows the cost achieved for the captured *snapshot* and the x-axis relates to the snapshot index. Multiplying the x-axis by 10 gives the epoch at which that snapshot was captured. The red line relates to training error found and the black line refers to validation error.

Figure 7.15 shows an initial steep descent from snapshot 0 to roughly snapshot 15 (epoch 150). From snapshot 15 (epoch 150) to roughly snapshot 40 (epoch 400), the descent is still relatively steep. From this point on, the learning process is more gradual, where the slope of the learning curves begin to level off but still proceed down, until roughly snapshot 50 (epoch 500). From this snapshot on the error still proceeds down, but at a more gradual degree and appears almost flat. The distance between the two curves is low throughout. Figure 7.16 also shows an initial steep descent from snapshot 0 to snapshot 1 (epoch 1). The curves are still close from snapshot 1 to 2 and proceed down at a slightly more gradual level but from this snapshot onwards, the curves begin to diverge and the distance between them gradually increases.

Deep Learner Trial-Run Level Prediction Analysis. For both learning curves, we initially observe that the validation error is lower than training error. This is due to the much higher number of instances in the training set and thus, there is more opportunity for error in the training predictions. What is also immediately clear is that the learning curves for Trial 24 are much *closer* and *lower* than those for Trial 32. This shows Trial 24 has neither underfit or overfit and as a result, there is sufficient evidence to select Trial 24 as the best performing. Low levels of underfitting and overfitting mean the learner model has learned well from the

Table 7.11: Result of Query to Updates for Trial 24

	Final	Past
Train Cost	**0.0802236**	0.0802515
Validation Cost	0.0286141	**0.0286098**
Epoch	781	780

training data *and* can generalises well to unseen instances, thus providing better predictions for survival analysis. Trial 32 on the other hand exhibits evidence of overfitting, which leads to poorer predictions. Also evident is that at no point does the validation error progress upwards, in either of the curves. This is due to early stopping exiting the training process before high levels of overfitting are reached, evidence that early stopping worked. Finally, in Figure 7.15, at the very end, it looks like the curves are slightly diverging, which could be evidence of the learner model moving away from the optimal configuration. We can investigate this by examining the *interim* learners at the level for this Trial-Run.

Deep Learner Updates Performance Results. To query the *updates* for this Trial-Run in greater detail, we use `get_best_performance`, from our Toolkit, at the *Trial-Run* level of aggregation. This will enable us to determine if at some point in the process of training, a more optimal configuration was found, which is superior to the configuration presented at the end of the learning process.

Table 7.11 shows the most informative elements of the POL *Performance* instance which result from the `get_best_performance` function. These are the training and validation costs and the epoch at which these costs were found. The *Final* column refers to the final performance found for Trial 24 and stored in the *Trial-Runs* collection. The *Past* column refers to result of the performance query to the *Updates* store. We show the cost results rounded to 7 decimal places for presentation purposes. We can see that the final learner model has the lowest training cost of 0.0802236, *but* the *interim* learner achieves the best validation cost of 0.0286098, stored 1 epoch before training exited.

Deep Learner Updates Performance Analysis. The increased training score for the final model at epoch 781 and the decreased validation score compared to that achieved at epoch 780 is evidence of the training process experiencing low levels of overfitting towards the end of the training process. Although slight, there is an improvement by the past learner model. Thus, we can query the learner model stored at epoch 780 with `get_entity` and deploy this for use.

If *interim* learner models were not captured, we could *not* generate learning curves after an experiment terminates. Nor is it possible to query *past updates* to determine if previous iterations of training achieved better predictive accuracy than the point at which training exited.

7.3.3 Summary for Case Study 2

The first part of this second case study identified the most accurate learner configuration for use in predictions. In this case, a shallow architecture was optimal. In terms of usability, we easily enabled the optimisation and the analysis of the results. First, we configured and optimised 128 combinations of shallow *and* deep architectures using the CDN. The comparison of shallow to deep was enabled through the setting of a single input parameter. Furthermore, our system enabled this successful optimisation and the generation of an optimal learner instance configuration in a manner where the results are easily reproducible. All parameters and configurations required to reproduce the experiment along with 6585 update snapshots detailing the steps of the learning process, are captured and automatically recorded by the toolkit. This level of reproducibility is *not* possible without the storage of all parameters and interim results. Furthermore, the DNT enabled an analysis which lead to the *understanding* of why this configuration was optimal for survival predictions on a small dataset. While optimisation of hyper-parameters is still possible without our toolkit, the degree of result interrogation and reproducibility is not available in any of the existing literature.

The second part of the case study successfully met the requirement of generating survival analysis predictions. We confirmed the results presented in §7.3.1, and determined did the final result achieve optimal predication accuracy of the entire learning process. We found this not to be the case, with training proceeding for 1 epoch longer than necessary. Although we could generate survival analyses without the toolkit, the analyses presented at multiple levels of experiment aggregation, from *Update*, to *Trial-Run*, to *Trial* and *Experiment* are *not* possible without it. Furthermore, it would not be possible to query updates of multiple learner instances after an experiment had completed, or restore more accurate learner instances found during past updates. In summary, this is the first attempt to utilise deep learning for survival analysis.

7.4 Case Study 3: Detecting Unusual Chords

Musical harmony analysis involves identifying musical notes, timing, keys and chords in new pieces of music and is a time-consuming process. Furthermore, musical pieces often contain unusual chord configurations, which are difficult to detect for non-expert human users. A means to automate and understand the process of extracting feature representations and unusual events from musical pieces could greatly aid musicians in understanding unseen musical pieces and provide a means to understand these for quicker interpretation and use. Furthermore, new musical pieces are generally *unlabelled*, unless sheet music is provided and can be regarded as *unsupervised* data. The third case study consists of 3 parts utilising *unsupervised* analysis techniques of the Bach music dataset. Although the Bach dataset contains chord labels, these are not used for training purposes and instead, single layer and deep Stacked RBMs are used in the evaluation. As there is an overlap in terms of learners used and all are based on the Bach dataset, these case studies and findings are presented together. The first part of the evaluation (Case Study 3A, presented in 7.4.1) seeks to find accurate *unsupervised* learner models and compare shallow to deep representations and acts as a platform to the second part of the case study. This second part (Case Study 3B, presented in 7.4.2) utilises the optimal learner identified in the first case study to investigate if RBM free energies can be used in anomaly detection for the identification of unusual chord configurations. The final part of the case study (Case Study 3C, presented in 7.4.3) aims to analyse the feature representation learned by the optimal RBM configuration and extract what features are learned in the hidden nodes. At the time of writing, there is only one other study into the use of deep stacked RBM free energies for anomaly detection and they do not use greedy, unsupervised layer-wise training. Furthermore, work in interpreting the hidden features of these types of learner models, outside of computer vision, has not been undertaken to this degree.

Features. The Bach dataset originally contained 17 variables. The first, *Choral ID* is an ID corresponding to the file names from the following website [28]. The next is the *Event number*, which is simply the assignment of a cumulative, 1-indexed number for each event - chord change - inside a chorale. The next 12 are binary *Pitch Classes*, indicating whether a particular pitch is present in an event with a "YES"/"NO" string. The first pitch is C, so for example, the 3rd feature would be a binary indicator if C was present, the 4th would indicate the presence or absence of C♯/D♭, the 5th relates to D, and so on. The last two variables *Bass* and *Meter*, relate to the pitch of the chord's bass note and how long the chord was sustained for respectively. *Bass* is a categorical String and *Meter* is an ordinal variable from 1 to 5. Each pitch indicator had to be converted to numeric binary and the *Bass* and *Meter* variables were converted into one-hot encoded vectors, using the workflow described in §4.1.1. A sharpened lower note is equivalent to a flattened upper note. Finally, the last feature is the *Chord Label*, which amounts to the chord that occurs in the event given by the Event ID. This was omitted from training, but used in the evaluations that follow. There are 29 input features after 1 hot encoding the 14 original features.

Table 7.12 shows the distribution of classes for the Bach [7] dataset. The chords D♯d6 and F_d7 are omitted from Table 7.12 for presentation purposes and occur only once. The hash symbol or ♯, refers to a sharpened note; b to a flattened; lower case m to a minor chord; upper case M to a major chord; and d to a diminished chord. Table 7.12 shows some chord labels occur with much less frequency than others, therefore our aim is to identify these in Case Study 7.4.2 as anomalous samples in the data. For Case Study 7.4.2 we consider the top 10 chord frequencies to be non-anomalous. The frequencies of these chords are all above 200.

Table 7.13 shows the hyper-parameter search space for the Bach dataset in the evaluation that follows. The batch size (*b_size*) bound was set to roughly between 1%

Table 7.12: Bach Chord Label Class Distribution

Num. >55		55 >Num.		Num. >6		6 >= Num.		3 >= Num.	
Class	Num.	Class	Num.	Class	Num.	class	Num.	Class	Num.
D_M	503	G_M7	52	D_M4	16	G♯d7	6	G_m6	3
G_M	489	B_M7	46	A_M4	16	C_M6	6	F_m6	3
C_M	488	E_M7	43	C♯d7	15	E_d	6	D♭d7	2
F_M	389	F_m	42	F_M4	14	G♯m	6	C_d6	2
A_M	352	C♯M	39	F♯d	14	B♭m6	6	B_m6	2
B♭M	312	F_M7	38	E_m6	14	A♯d	5	C_d7	2
E_M	295	D♭M	37	E_M4	14	B♭d	5	A_M6	2
A_m	258	F♯M7	34	F♯M4	12	A_d	5	C♯d6	2
E_m	241	D_m7	33	D_m6	12	F_M6	4	A_m4	2
B_m	217	B♭m	26	G♯d	11	A♯d7	4	D♯m	2
G_m	179	C♯m	24	A_m7	11	D_d7	4	D♭d	2
D_m	165	E_m7	24	A_m6	10	D♭m	4	D♯M	2
E♭M	146	D♭M	21	C♯d	10	D♯d7	4	C♯M4	2
C_m	144	C_m7	20	C♯m7	9	D_M6	3	D♭m	2
F♯m	143	F♯m7	19	G_M4	8	G_d	3	E♭d	1
B_M	143	B_m7	19	B_d7	8	B_M4	3	F♯d7	1
F♯M	90	G_m7	18	C♯M7	7	B♭M7	3	E♭M7	1
C_M7	66	B_d	17	F♯m6	7	D♭m7	3	A♭d	1
D_M7	58	C_m6	17	F_m7	7	F_d	3	G♯M	1
A_M7	56	C_M4	16	D♯d	7	G_M6	3	D♭M7	1

and 10% of the dataset size at 50 and 500. The number of hidden layers (*l_hidden*) tested was from 1 (shallow) to 3 (deep). The number of hidden layer nodes was constrained to being between 2 and 20 to make it easier to enable easier analysis of the feature representation. Regularisation is not applicable for training of stand-alone RBMS. *Dropout* again has only two options, True or False. The bounds for the learning rate, are set low as it was determined the gradient was less likely to vanish in deeper architectures. The *patience* parameter (minimum nuber of epochs to iterate) was set to between 50 and 100. Finally *max epochs* was set to between 100 and 1000 as we determined that the lower number would be sufficient for training and 1000 epochs would likely overfit the data. The CDN was used to perform random search in this evaluation for 30 Trials and 10 Runs for each Trial, resulting in *300* learner optimisations, again demonstrating the flexibility of the CDN. This was required to get a distribution of model parameter weights for feature representation analysis.

Table 7.13: Bach Stacked RBM Hyper-Parameter Search Bounds

Hyper-Parameter	Description	Bounds (low, high)
b_size	samples in a mini batch	(50, 500)
l_hidden	number of hidden layers	(1, 3)
o_nodes	number of hidden layer nodes	(2, 20)
dropout	whether to perform dropout	(True, False)
learning_rate α	co-efficient for weight updates	(0.0001, 0.005)
patience	min epochs to iterate	(10, 50)
activation	hidden layer activation	(logistic)
max_epochs	maximum epochs to iterate	(100, 1000)

We also test shallow and deep configurations as the hidden layer search space is betewen 1 and 3. The data was split with 70% for training 15% for validation and 15% for testing.

7.4.1 Hyper Parameter Optimisation

The functions used for this analysis are `get_top_k_ids`, `get_entity`, `coalesce_hps` and `get_top_k_error_scores`. As we explored the result of these functions in depth in §7.3, we only explore hyper-parameter results in this section at a higher level. Furthermore, we select the *top 5* performing configurations for analysis to determine the overall top performing. Table 7.14 shows the most informative hyper-parameters and the training and validation scores as returned from these functions. The *hidden* column signifies how many hidden layers are in the architecture. The *o1*, *o2*, and *o3* columns refer to how many nodes are in the 1st, 2nd and 3rd layers. The *alpha* column shows the learning rates. The associated training and validation *monitoring cost* scores are shown in the *t_score* and *v_score* columns. Table 7.14 shows cost values rounded to four decimal places. Depending on the configuration of the visual samples, this score can go to negative values if more features are positive in the input sample or the reconstruction.

Hyper-Parameter Results. Table 7.14 shows that Trial 13, Run 9 achieved the lowest cost of 0.0071. It is a *deep* architecture with 3 layers and 13, 9 and 2 nodes, respectively, in these layers. Furthermore, all top 5 architectural configurations in Table 7.14 are deep. Node counts found are low, where of the 14 hidden layer

Table 7.14: Bach Stacked RBM Scores and Architectures

rank	trial_id	run_id	t_scores	v_scores	alpha	hidden	o1	o2	o3
1	13	9	0.0508	**0.0071**	0.00056	3	13	9	2
2	5	1	0.0625	0.0096	0.0006	2	8	2	na
3	12	5	-0.1354	-0.0106	0.0022	3	9	15	5
4	9	2	-0.0070	-0.0144	0.00186	3	12	15	2
5	14	6	0.0280	0.0221	0.00058	3	8	2	11

configurations presented, 9 had values below 10 and 4 had values between 10 and 15. Learning rates are low with 3 values at or below 0.0006 and two values in the range 0.001 and 0.0025.

Hyper-Parameter Analysis. The results in Table 7.14 shows that `get_top_k_ids` is robust to negative values and can therefore, account for both gradient ascent and descent. For this task, a *deep* network was shown to outperform its shallow equivalent, achieving a lower cost. The deep network with 13, 9 and 2 nodes in its hidden layers was found to be the best, suggesting that multiple levels of data abstractions or *feature representation*, which compress the data in each subsequent layer, better *describes* the input data. Therefore, feature representations learned are compressed representations of the data. Low learning rates suggest that for the complex deep architectures, lower learning rates enable more optimal descriptions and feature representations to be reached through the attainment of lower costs. Trial-Run 13-9 will therefore, be the focus of the anomaly detection analysis in the next section as this has the lowest score on the validation set of 0.0071.

7.4.2 Anomaly Detection

In order to utilise the top learner model found in the first part of this case study, we query the Trial-Run store with `get_entity` to return the relevant learner instance found at Trial-Run 13-9. We then use the *LearnerFactory* of the CDN to instantiate a Learner with pre-trained parameters and relevant hyper-parameters, again demonstrating the highly configurable nature of the CDN, even upon Experiment completion. We also require the *dataset_split_indices* object from the POL *Learner-Performance* instance in order to use the CDN `Preprocess` API to split the data

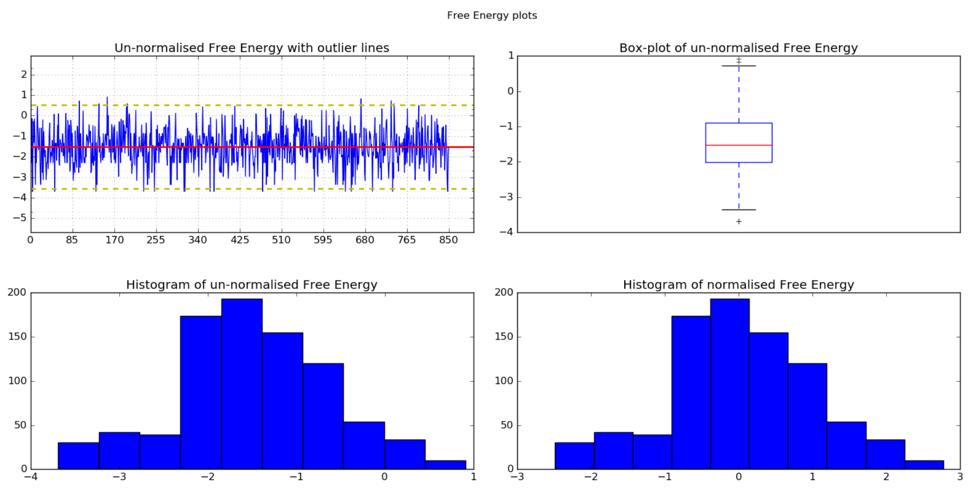


Figure 7.17: Free Energy Distribution of Top Level RBM for Trial 13 Run 9

and *recreate* the exact dataset configuration which generated the learner instance for Trial-Run 13-9. This process is crucial because if we performed tests with the generic configuration of the dataset, data samples which were used to train the learner instance could inadvertently be used to test it and get an upwardly biased representation of the learner's performance. This is an important aspect in the reproducibility of an experiment: determining the *precise* configuration of the dataset used to train the learner model. Once instantiated, we call `build_hypothesis` and use the `hypothesis` function of the learner to generate the appropriate free energies. To interpret the free energy of the top level RBM generated with pre-trained parameters of Trial-Run 13-9, we require two more functions. The first is from the `Visualise` component, `visualise_free_energy_distribution`. The second, `get_anomalous_indexes` from the `Evaluate` component, is used to examine the free energies and extract anomalous examples.

Deep Anomaly Results. Figure 7.17 shows the result of the visualisation function. The plot on the top left of Figure 7.17 charts the free energy for each sample in the test set of the Bach Dataset. The y-axis denotes the free energy value and the x-axis denotes the test dataset sample index. The dashed lines represent values calculated for values within 2.33 standard deviations of the mean. This corresponds to where 98% of the data should appear in a normal distribution and is called the critical

Table 7.15: Bach Anomalies

chord	frequency	dataset index
B♭M	312	2626
B♭M	312	1072
G_M7	**52**	1082
G_m	**179**	5272
G_M	489	5472
F_M	389	3066
G_M	489	2572

value. The plot on the top right is a box plot of un-normalised energy values. In this plot, the y-axis, represents the free energy values. The central rectangle corresponds to the inter-quartile range of the data, the bottom of the rectangle represents the value for quartile 1, that value which 25% of the data occurs behind, the red line in the centre denotes the median and the top of the box plot denotes the third quartile or value that 75% of the sorted free energies occur behind. The whiskers beyond the central rectangle denote values at 1.5 times the inter-quartile range above the third quartile and 1.5 times below the 1st quartile respectively, with any values outside these, very likely outliers. The two plots on the bottom left and bottom right respectively show histograms of the frequency distributions of free energies. These plots allow us to *understand* how the free energy is distributed for each sample in the test set. Furthermore, Figure 7.17 shows that for this dataset, energies are normally distributed. The dashed lines on the the plot on the top left plot shows us what samples our system determines to be anomalous. In addition, the box plot on the top right shows there are some very extreme values of energies within the dataset. Table 7.15 shows the result of `get_anomalous_indexes`, corresponding to those samples that appear outside the dashed lines in the top left plot of Figure 7.17. We used a critical value of 2.33 so only a small number of points with extreme energy values are returned. For *interpretable* analysis, we only consider points beyond the upper limits. Table 7.15 shows the anomalous indexes returned in the *dataset index* column, along with the chord labels corresponding to this index in the *chord* column and the count of that particular chord frequency within the dataset in the *frequency* column. Table 7.15 shows that the method used to investigate anomaly detection with deep RBM free energies, only 2 out of the 7 are anomalous, which is less than 50% accuracy. The result does return two anomalous indexes, 52 and 179.

Deep Anomaly Analysis. The identification of anomalies presented here is significant, due mainly to the high number of classes (101 different chord labels) many of which are anomalous. Furthermore, anomalous examples occur throughout the training data *and* the problem is unsupervised. Each of these problems increase the difficulty of anomaly detection. If we lowered the critical value, we could identify more anomalous instances but this would also raise the false positive rate. Furthermore, there could be other factors at work within the data which mean the samples identified are not unusual within the context of the dataset, but the configuration *is* unusual in other ways. For example, the inversion of the chord (note in the bass) or the length the chord is sustained for could be unusual. Furthermore, the chord itself could be out of place (from another key) in the context of a single sequence (piece), but not in context of the dataset as a whole. We do not currently have this information, therefore further investigation into the data at finer-grained levels of granularity is required. It is also possible that our interpretation of the free energy requires refinement as it is a highly complex concept, but has been shown to work in other contexts [140], [48].

Deep Anomaly Summary. Although we have demonstrated that this configuration is optimal in §7.4.1, these initial experiments in anomaly detection were not particularly accurate. We have highlighted key problems with the conditions of this element of the evaluation, in terms of the properties of the dataset used. As it is the first deep analysis on this highly complex unsupervised dataset, problems are to be expected with preliminary results. There has been successful studies of deep energy based networks for anomaly detection, but score matching is used [140] in contrast to greedy layer-wise training employed here. Ultimately, we have shown the utility of the DNT and CDN for analysing the energies of top level deep RBMs. Therefore, further investigations of its use for anomaly detection are easier as a result. To successfully develop and refine this technique, we require either a synthetic dataset or one where the distribution of the data is well understood and can be manipulated. This would allow us to control the exact conditions of the data and determine the

Table 7.16: Frequency Count of Each Feature Occurrences

Chord Notes		Chord Bass Notes		Chord Meter	
Feature	Frequency	Feature	Frequency	Feature	Frequency
C	1790	bassC	600	meter1	126
C♯/D♭	954	bassC♯/D♭	218	meter2	1820
D	2365	bassD	**689**	meter3	**1842**
D♯/E♭	709	bassD♯/E♭	251	meter4	918
E	2125	bassE/Fb	678	meter5	959
F	1284	bassF/E♯	462		
F♯/G♭	1412	bassF♯/G♭	456		
G	2142	bassG	681		
G♯/A♭	659	bassG♯/A♭	181		
A	**2375**	bassA	688		
A♯/B♭	1021	bassA♯/B♭	312		
B/Cb	1791	bassB	449		

best way to utilise the free energy for anomaly detection. Our toolkit easily imports and processes new datasets for input into deep learning algorithms, train these algorithms with a range of different hyper-parameters and analyse the results. We now explore the utility of the DNT for analysing the feature representations of the learner model found in Trial 13 across *all* runs.

7.4.3 Feature Representation and Model Parameter Analysis

The final part of this case study is the analysis of the hidden feature representation of the deep Stacked RBM found to be the best performing in §7.4.1. This is at the *Trial* level of granualarity. Neural networks are often considered uninterpretable 'black boxes', unless visualising the filters learned for image processing. An important aspect of data analysis is how *interpretable* the model learned is, especially for applications as in this case study and in health analytics. This amounts to how input features interact and can provide a great deal of information. Otherwise, the learned representation in deep neural networks is underutilised. To our knowledge we are the first to provide a means to investigate these interactions for deep networks in a *generally applicable* way. The Toolkit functions required here are `coalesce_model_parameters` and `feature_analyse`. For this analysis we also require counts of how many times each feature appears in the dataset. Table 7.16

shows these counts. Each cell refers to each *Feature* in the dataset and contains a *Count* of how many times it appears in the dataset. We have highlighted in bold the most frequent feature for each feature type. A flattened (♭) upper note is equivalent to a sharpened (♯) lower note. This is why some notes are presented as, for example, G♯/A♭. The notes G♯/A♭ are equivalent notes and hence were encoded equivalently and can be interpreted as either.

Tables 7.17, 7.18 and 7.19 show the result of `feature_analyse` - those inputs to each layer considered statistically significant from zero via a one-sample t-test on 10 sets of optimised parameters from 10 Runs in the top evaluated hyper-parameter Trial 13 configuration. The column $N_j^{(i)}$ contains the index of the node in the layer being examined. For example, 1 in Table 7.17 refers to node $N_1^{(1)}$. The column *Sig.* $L^{(i-1)}$ *Inputs* refers to the node index of those input(s) considered significant for that node from the previous layer. The *Combining* column shows the name of the feature or the type of node the index in the *Sig.* $L^{(i-1)}$ *Inputs* column refers to.

Layer 1 Feature Representations Results. Table 7.17 shows those inputs from $L^{(0)}$ for each node in $L^{(1)}$ considered significantly different from zero with a significance level of 95%. Of the 13 nodes, 5 nodes were found to have one significant input, 2 had two significant inputs, 1 had three significant inputs, 1 had four significant inputs, 1 had five significant inputs and 3 found six inputs to be significant. We now provide a more detailed analysis. To avoid confusion, each node is examined in the order of their input cardinality.

1 Input Results. Examining those nodes with 1 input significant input, Nodes $N_1^{(1)}$ and $N_6^{(1)}$ had *bassA*. This feature relates to an A note being in the bass of the chord for an instance or *event* in the dataset. $N_{14}^{(1)}$ had *bassD* (a D note in the bass) as its only significant input, $N_{22}^{(1)}$ had *bassA♯/B♭* and $N_{26}^{(1)}$ had *meter3* as its only significant input.

1 Input Analysis. Referring back to Table 7.12, one can see that D major and A major are in the top 5 most frequently occurring chords. The bass notes of these chords, when not inverted (the root of the chord is in the bass) are D and A, respectively. These notes combine significantly by themselves in 3 nodes. Two of the other top 5 most frequently occurring chords have these notes in the bass if an

Table 7.17: $L^{(1)}$ Significant Inputs

$N_i^{(1)}$	Sig. $L^{(0)}$ Nodes	Combines
1	21	{bassA}
2	26	{meter3}
3	15, 22	{bassD♯/E♭} {bassA♯/B♭}
4	6, 12, 24, 27	{F♯/G♭} {bassC} {meter1} {meter4}
5	1, 11, 16, 18, 21	{C♯/D♭} {B} {bassE} {bassF♯/G♭} {bassA}
6	21	{bassA}
7	8, 11	{G♯/A♭} {B}
8	22	{bassA♯/B♭}
9	18, 19, 24	{bassF♯/G♭} {bassG} {meter1}
10	4, 8, 12, 19, 21, 26	{E} {G♯/A♭} {bassC} {bassG} {bassA} {meter3}
11	14	{bassD}
12	2, 10, 11, 12, 18, 24	{D} {A♯/B♭} {B} {bassC bassF♯/G♭} {meter1}
13	0, 3, 13, 14, 17, 23	{C} {D♯/E♭} {bassC♯/D♭} {bassD} {bassF} {bassB}

inversion of the chord is played. For example, G major, the *second* most frequent chord has a D in the bass for its 2nd inversion and F major, the third most occurring chord has an A in the bass in its first inversion. Furthermore, *bassA* and *bassD* are the first and second most occurring bass notes throughout the dataset at 689 and 688 respectively as can be seen in Table 7.16. Therefore, the network correctly identified these as major factors in describing the data, assigning complete nodes (1, 3 and 11) to identify these features. $N_{22}^{(1)}$ more likely refers to a B♭ in the bass note than A♯. We can see from Table 7.12 that B♭ major is the 6th most occurring chord in the dataset with 312 instances. It is the 9th most frequent bass note in terms of frequency count in Table 7.16. As it is the sole significant input to a single node, it shows the learner model learnt to highlight its importance within the dataset. With regard to meter3, this is the most occurring meter for a chord in the dataset (length a chord is sustained). Node 2 having this as its sole significant input shows the learner model successfully learnt to also highlight this features importance within the dataset, in order to identify that most chords are sustained for three beats. These nodes therefore can be seen to be bass note identification features.

2 Input Results. We will now examine the 2 nodes where 2 inputs were considered significant. $N_3^{(1)}$ learned to combine *bassD♯/E♭* and *bassA♯/B♭*. $N_7^{(1)}$ combined *G♯/A♭* and *B*. For $N_3^{(1)}$ we will consider the flattened notes E♭ and B♭.

2 Input Analysis. The Bach chorales dataset is a set of Choral church music. As we have already discussed B♭ major is a very frequent *chord* in the dataset, so this would suggest that the *key* of B♭ major often occurs in the dataset. Church music is characterised with a particular cadence (chord progression) known as a *plagal* cadence. This progression is from chord IV to chord I. E♭ is the the fourth note of B♭ Major. Therefore the learner model has learned that there is a strong association between those notes. If we examine the dataset further we are likely to find that a bass B♭ often goes to E♭ and E♭ would often go to B♭. Further interrogation of the data is required to identify why this combination is favoured over others. For $N_7^{(1)}$ G♯ is a note in the key of B major and these tones are only a semi-tone (half a note) apart. These nodes can therefore be thought of key identifiers, or those which provide contextual information.

3 Input Result and Analysis. $N_9^{(1)}$ combines *bassF♯/G♭*, *bassG* and *meter1*. A common musical technique is just before the final chord (chord I) is reached at the end of a musical piece the last note of the key - the 7th note - is played. This gives an impression of finality. The note F♯ is the 7th note of G major, the second most frequent chord in the dataset, so it is likely that this progression in the bass occurs often. Furthermore, a progression such as this is often less accented as the 7th note of a key is considered 'unstable', which is likely as to why *meter1* is also combined with the notes in this node.

4 Input Result and Analysis. Nodes with more than 3 significant inputs get more difficult to analyse and interpret. For example $N_4^{(1)}$ combines *F♯/G♭*, *bassC*, *meter1* and *meter4*. This does not make as much sense as F♯ or G♭ are not in the key of C and *meter1* and *meter4* refer to a very unaccented event and a very accented event. C would be the bass note and G♭ would be in the chord of C diminished and this chord occurs four times in the dataset but it does not make sense why a node would learn this combination. Although if we examine this node along with $N_9^{(1)}$, this input combination could begin to make sense. F♯ *is* in the key of G. If our analysis of $N_9^{(1)}$ is correct and it relates to a passing note in a sequence for a cadence (final chord progression), C is the fourth note of G so these two nodes taken together could have learned to identify the combination of notes that progress from the fourth chord of G to the first chord of G with a passing 7th note in between. This would also explain why the node has a combination of *meter4* and *meter1*, a long sustained chord before a short final chord or vice versa. We can examine if these nodes are in fact combined by rerunning the `feature_analyse` function but this time passing the second layer index. What is unclear is why, if these nodes relate to a chord progression, was only this progression learned.

5+ Input Result and Analysis. We will finally analyse $N_5^{(1)}$ which learned a combination of *bassA*, *C♯/D♭*, *bassE*, *bassF♯/G♭* and *B*; $N_{10}^{(1)}$ which has a combination of $N_{22}^{(1)}$, *bassC*, *E*, *G♯/A♭*, *bassG* and *meter3*; $N_{12}^{(1)}$ with a combination of *D*, *bassF♯/G♭*, *A♯/B♭*, *bassC*, *B* and *meter1*; and $N_{13}^{(1)}$ with *bassB*, *D♯/E♭*, *bassC♯/D♭*, *bassD*, *bassF* and *C*. Interestingly, if we first take $N_5^{(1)}$ and disregard whether or not the note was in the bass, all the notes in this combination are a third apart and suggest a chord namely: A, C, E, G, B. When we consider the symbols and the alternative representations of the note a huge variety of chords could be identified, especially withing the context of key identification and bass identification chords. If for example bassA, C♯, bassE and B was input, this node could identify an A major and minor and the associated 9th chords of these and their inversions. *A major* is also the fifth most frequent chord in the dataset. Taking another possible combination from $N_5^{(1)}$: bassE and note B could be used to identify *E major*. If we look at $N_{10}^{(1)}$, $N_{12}^{(1)}$ and $N_{13}^{(1)}$ in the same way, these notes also are combined in thirds. For $N_{10}^{(1)}$: A, C, E, G, G and meter; for $N_{12}^{(1)}$: D, F, A, C and B; and for $N_{13}^{(1)}$: B, D, D, D, F and C. The different symbols on repeated notes would allow it to distinguish between major, minor or diminished chords. Therefore where other nodes learned to identify specific, key, note and meter elements, these nodes identify chords in aggregate. Before we leave this layer, it is worth noting that some input features are completely absent from from Table 7.17, a point we will come back to later. We now can investigate deeper layers by passing relevant layer indices to `feature_analyse`.

Layer 2 Feature Representations Results. For the analysis of L^2 nodes, the significance level was set to 90% for the t-test in `feature_analyse`, as all nodes showed no significant inputs at the 95% probability level. Table 7.18 shows $N_1^{(2)}$

has one significant input: *meter3*. $N_2^{(2)}$ has significant weights on the connections *from* $N_7^{(1)}$ and $N_9^{(1)}$ in the previous layer, combining what we determined to be a *key node* and a bass progression or *cadence* node. $N_3^{(2)}$ significantly relates to $N_5^{(1)}$ in the previous layer, which was identified as a chord node. $N_4^{(2)}$ was found to have no significant inputs. $N_5^{(2)}$ in $L^{(2)}$ significantly weights $N_2^{(1)}$ in the previous layer. $N_6^{(2)}$ significantly weights $N_{11}^{(1)}$ from the previous layer and $N_7^{(2)}$ has a significant connection with the previous layer's $N_2^{(1)}$. Node $N_8^{(2)}$ significantly combines $N_1^{(1)}$ and $N_5^{(1)}$ from the previous layer, both relating to the *bassA* input feature and $N_9^{(2)}$ combines the 6th and 12th nodes from the previous layer. These related to a *bassA* and a chord node.

Layer 2 Feature Representation Analysis. From Table 7.18 we can see that the network is learning more abstract representations of the data, or feature *interactions*. It is parsing out those input features that have the biggest effect in the dataset. For example Nodes $N_1^{(2)}$, $N_5^{(2)}$ and $N_1^{(2)}$ all relate to *meter3*, the most frequent timing feature. $N_2^{(2)}$ combines a key with what we interpret as a cadence chord and in doing so, likely now relates to a chord feature. $N_3^{(2)}$ directly relates to $N_5^{(1)}$ in the previous layer which we have already analysed. $N_6^{(2)}$ then has *bassD* as its only significant input, which is the bass note of the most frequent chord in the dataset. Interestingly the network learned that the two *bassA* are the same at this level, combining them into one node. This again, is the root note of one of the top five most frequent chords and is the second most occurring bass note (as shown in Table 7.16). Finally, the last node combines the *bassA* note with another chord node we analysed in the previous layer, also making this a chord node. $N_4^{(2)}$ was found to have no significant inputs. Our evaluation, leading on from the missing inputs from the first layer, is not only does the network learn nodes relevant to significant features but also those which relate to *minor* features. This allows for a greater degree of description for unsupervised data.

Layer 3 Results and Analysis. Table 7.19 shows those inputs found to be significant with a probability of 90% for the final layer. The first node combines major features we have already discussed but the second node has no significant inputs. Our hypothesis is that one node relates to an abstraction of the major

Table 7.18: $L^{(2)}$ Significant Inputs

$N_i^{(2)}$	Sig. $L^{(1)}$ Nodes	Combines
1	2	meter3
2	7, 9	key and cadence
3	5	chord node
4	n/a	n/a
5	2	meter3
6	11	bassD
7	2	meter3
8	1, 5	bassA and bassA
9	6, 12	bassA and chord

Table 7.19: $L^{(3)}$ Significant Inputs

$N_i^{(3)}$	Sig. $L^{(2)}$ Nodes	Combining
1	5, 7, 9	meter3, meter3, bassA and chord
2	n/a	n/a

inputs - as shown - and the other node relates to abstractions of the *minor* input features. This would account for the *minor* variance or unusual aspects of the data. This is a very interesting finding if this is indeed the case, as we successfully, reduced the dimensionality of the data, down to two features but in contrast to techniques such as PCA, we *do not* discard any of the dataset variance.

Feature Representation Analysis Summary. In our analysis, we identified the first layer to learn lower level features relating to bass note and specific meters, as well as combination or chord nodes. In subsequent layers, we identified further abstractions of the data which combines major input features and found that certain features have no significant inputs, but as the network uses these weights when describing the data we believe that it also learns a representation of the *minor* dataset features, this motivate the need for a second means of analysis which can identify the minor inputs to each node. We have shown here that our Toolkit gives us a means to look inside the 'black box' of deep networks, for a generic application and trace the representations learned from the first hidden layer, through to the deepest point in the architecture. Thus, providing a means to begin to *understand* the machinations of these complex networks. Furthermore, if interim learner models were not stored, these results would not be possible.

7.4.4 Summary for Case Study 3

This first part of this case study again demonstrated how using the toolkit successfully optimised hyper-parameters for the deep learner configuration to be used in the subsequent analyses. We were easily able to configure and optimise *300* different combinations of shallow and deep architectures. In this case, our system evaluated a *deep architecture* to be the best performing, identified the correct hyper-parameter configuration and associated learner model for deployment in the subsequent analyses.

The second part of the case study investigated the use of the free energy of the top level RBM in a deep network for *unsupervised* anomaly detection. We again demonstrated the utility of our CDN system and toolkit, querying the best performing learner instance and configuring a new deep learner with pre-trained parameters. Although our attempts in anomaly detection were not as successful as we had initially hoped, we are the first to approach anomaly detection with deep networks with this training procedure and identified a number of ways in which we can improve performance. Furthermore, we were successful in demonstrating that we could easily analyse the energy of the top RBM in a deep stack. Both of these factors will allow us to continue our investigation in deep learning for anomaly detection, which others have shown to be successful [140].

Finally, in the last part of the evaluation, we successfully analysed the feature representation learned by a deep neural network. Our approach in this regard is not possible without storing interim learners. We demonstrated how to look inside the 'black box' of deep feature representations, tracing values from the input through *all* hidden layers to the deepest representation. Our means to do this is generic and unlike other approaches, this technique can be applied to many different types of datasets. Through our analysis, we identified nodes that were likely to have learned representations of *minor* inputs, an important finding that motivates the extension of our techniques to also examine minor contributing features. This also shows that unlike shallow dimensionality reduction techniques, like PCA, deep networks also make use of minor features in unsupervised tasks and do not discard any data variance, while still learning good models. Finally, we demonstrated that a single learner instance could be deployed to *two* data mining tasks.

7.5 Summary

The goal of this chapter was to validate our hypothesis, that deep learning experiments require supports not available before this research was undertaken. To provide depth and demonstrate general applicability of our research, we used datasets from 3 separate domains, data mining requirements specified by domain experts (sports performance and dementia datasets) and tackled an ongoing problem from the discipline of music.

The evaluation was spread across 3 separate case studies, achieved all 3 *application* goals and was sufficiently exhaustive to include all five shallow and deep networks presented in Chapter 3. We demonstrated the ease with which it was possible to *configure* and *optimise* large deep learning experiments with the Configurable Deep Network (CDN) system developed for this research. We stored results with our Deep NoSQL Toolkit (DNT) and harnessed it to perform analyses and *interpretations* which are either difficult and time consuming, or not possible with current approaches. The three evaluation case studies demonstrate the way in which our system and supports can be used and highlight the power in application and ease at which we could achieve results. We also showed their utility in understanding and interpreting the results found. The results were not always as accurate as we would have liked but were more than sufficient to validate our system and approach. It is our hope that future research projects can use the platform provided in this research to generate improved levels of predictive accuracy that deep learning can offer.

Chapter 8

Conclusions

In this final chapter, we present an overview of the research presented in the thesis and highlight potential areas where this research maybe reused and extended. In section 8.1, we review the dissertation in terms of its overall goals and examine how we have addressed and validated these goals. In section 8.3, we discuss a number of areas where the work presented in this dissertation can be developed into new areas of research.

8.1 Thesis Summary

This dissertation is about deep learning. At the start of the journey, it was felt that a working hypothesis would state that deep learning algorithms were superior to those used in shallow learning approaches and our experiments would validate this hypothesis. However, it quickly became evident that the process of constructing deep learning experiments so as to yield meaningful conclusions and develop necessary optimisations was only possible through a very narrow focus in terms of requirements and problem domains (datasets). Our goal was broader: we sought to show that deep learning could be included in a variety of data mining operations in a broad problem space. We determined that this motivated a new approach to deep learning which required a formal description of the landscape and a set of supports to help the data mining practitioner that wishes to incorporate deep learning into their research. We can now restate the hypothesis made in Chapter 1 that: *in order to* *enable new levels of deep learning, a novel approach for configuring and running deep learning experiments; a semantically powerful data model representation of all elements of the deep learning machine; and the development of a toolkit which is based on the data model approach, to deliver functions that manage the experiment and better analyse the results*, is required. We will now review this dissertation and examine our hypothesis in the light of our research results and validation.

In Chapter 1, we motivated our research into deep learning. Specifically, we identified 3 key areas of data mining into which we planned to incorporate deep learning: *anomaly detection*, *sequential prediction* and *learning feature representations.* We also presented our 3 domains which offered different challenges through different datasets and requirements. In Chapter 2, we motivated our approach by analysing the state of the art in deep learning research.

Our contribution begins in Chapter 3, where to the best of our knowledge, we present a complete description of the constituent components required across a variety of deep learner configurations. The main learners discussed were *Multi-Layer Perceptrons*, *Recurrent Neural Networks*, *Restricted Boltzmann Machines* and *Deep Belief Networks.* The constituent components, feed forward hidden layer, recurrent layer and regression algorithms are also described in depth along with various cost functions. This chapter was crucial in understanding how to build and configure a deep learning experiment for the appropriate deep learner. Our approach and overall system architecture was presented in Chapter 4. In Chapter 5, we presented a more focused discussion on how to build and configure a deep learner using the constructs described in Chapter 3 together with the supports we provided in Chapter 5.

The main contribution in our research is presented in Chapter 6. The POL data model was specified in an attempt to capture all components, to the level of granularity required, necessary to capture the *state* of a deep learning experiment, at *any* iteration. It was felt that this was the best approach to optimising the deep learner as it enabled the development of functions to optimise and analyse the learner, from any point in the experiment. A further benefit of publishing the POL data model [102], is that it is now possible to share a snapshot of the results and settings of a deep learning experiment at any iteration, restart the experiment from that snapshot, or exchange the current hyper-parameter and model parameter settings at a required iteration snapshot.

We presented an extensive evaluation in Chapter 7 where we validated the hypothesis put forward in this dissertation. There were 3 case studies presented, which through a series of end-user requirements, across heterogeneous datasets, included the major data mining goals outlined in Chapter 1. The five principal learner algorithms were included in the evaluation and by stepping through the experimentation process, the evaluation showed how the Toolkit supported the decision making process (configuring and optimising the learner) and enabled a better understanding and interpretation of the results.

8.2 Limitations

Although we have demonstrated a comprehensive study in the field of deep learning, there are some limitations to the current work. First, although it is implicitly interoperable, we have not concretely tested the portability of result data stored in the physical POL by exchanging it with other researchers. Second, we have not employed our framework to analyse benchmark datasets, which would give a reliable measure of the quantitative performance of the toolkit. Finally although we have tested the toolkit for computational graph and probabilistic deep learning architectures, we have not yet extended it to convolutional architectures or tested its extensible nature by deploying it to the wider community.

8.3 Future Research

The field of deep learning is still relatively new and there are many important research topics that remain open. We focus on the approach that we have taken and discuss how the work presented in this dissertation can provide a platform to further extend deep learning research.

8.3.1 A Standard Data Model for Deep Learning

Our goal in enabling parameter optimisation in experiments was made possible by capturing the state of the experiment at very fine levels of granularity and providing tools for analysing iteration snapshots. Thus, the Parameter Optimisation for Learning (POL) data model was specified and implemented using JSON with a NoSQL storage model. However, this part of our research has rich possibilities for further research as such a data model does not currently exist. After publishing the data model in [102], we were invited to be part of the Machine Learning Schema Community Group [94]. Their target is to define a community agreed schema for data mining and machine learning that will inform, for example, the development of markup languages and data exchange standards. The POL data model provides a strong platform for the development of a standard data model for machine and deep learning. By continuing this research topic, it will be possible to develop a suite of reusable tools, specify a generic API to read and write experiment databases, and increase levels of sharing and cooperation among the data mining community.

8.3.2 Experiment Databases

Experiment databases for machine learning are a relatively new concept. The concept was introduced in [20] and expanded upon in [21], [133] and [134], where the aim was to formalise the elements of a machine learning experiment, enable their storage and in doing so make experiment reproduction and reuse trivial. Although the concept has important repercussions as a means of machine learning experiment reproduction, it has received limited research. However, it has found some success with tools such as OpenML [134], which at the time of writing has has 1705699 [107] uploaded experimental runs. Our specification of the POL data model together with the development of a JSON-based access library for MongoDB provides a new platform for the continuation of research into experiment databases, in an area that will benefit both machine learning and deep learning. Our DNT, while containing a number of useful functions for storing, reloading and analysing interim and final experiment results can be expanded with features, such as those which perform differentially private queries [43], which allow more accurate means of selecting top performing model configurations. This problem was highlighted in Chapter 7. Benefits such as increased options for hyper-parameter optimisation could also be leveraged using our system. Persisting experiment data and decoupling it from the experiment process allows for new methods of hyper-parameter optimisation to be tested and developed. For example, evolutionary algorithms could easily be enabled by the functions in our Toolkit.

8.3.3 Future Toolkit Applications

One could consider the applications presented in the evaluation as preliminary studies in new domains for deep learning. Future work in sequential prediction can harness the CDN to easily *combine* the RNN and DBN to test if improvements can be made in survival analyses and heart-rate predictions. One could use the methodology shown in §7.4 to analyse the feature interactions learnt in these contexts. For the analysis of feature representations, future research should seek to identify a methodology for the extraction of the *minor* contributors to the representations learnt in a deep representation. In this research, we successfully analysed the major features in the Bach dataset but methods for a deeper evaluation could examine an individual sample with known properties, to determine if those nodes which *fire* relate to the abstract features present in that particular sample.

8.3.4 Future Toolkit Development

We have developed the requirements for this toolkit based upon literature and our own deep learning experiment experience. To further validate and explore possible developments for the tools we have developed, explicit definitions of different user groups and requirements gathering and testing with the wider deep learning and data mining community would be beneficial. Questions relating to the completeness of the current operator set and the extensibility and completeness of the model schema could be answered, as well as discovering if more advanced functionality or experiment analytics are required. In addition the utility and need for the current visualisation and analysis tools could be evaluated. Furthermore, greater automation could be incorporated into the algorithm selection process by suggesting or automatically utilising the relevant deep network, dependent on the type of data that was input.

8.4 Closing Summary

At the start of this chapter, the point was made that this dissertation would originally focus on shallow versus deep learning but our discovery was that while this may well be the topic of future research dissertations, we felt that the more important focus was to *better understand* how to configure a deep learning experiment, determine why configuration decisions are made and better interpret the results. A simple Google search (eg. [92]) shows that there remains a lack of understanding as to precisely when to choose a deep learning setup. As the primary goal of this research dissertation, it is our belief that we have advanced this understanding to a significant degree and have lowered the entry requirements for data mining practitioners who wish to incorporate deep learning into their research.

Bibliography

[1] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016. 11, 36

[2] Daniel Ansari, Johan Nilsson, Roland Andersson, Sara Regnér, Bobby Tingstedt, and Bodil Andersson. Artificial neural networks predict survival from pancreatic cancer after radical surgery. *The American Journal of Surgery*, 205(1):1–7, 2013. 27

[3] Antonio Arauzo-Azofra, Jos Luis Aznarte, and Jos M. Bentez. Empirical study of feature selection methods based on individual feature evaluation for classification problems. *Expert Systems with Applications*, 38(7):8170 – 8177, 2011. 9, 21, 22

[4] Gaelic Athletic Association. Rules of gaa football. `www.gaa.ie/about-the-gaa/our-games/football/rules`, 2015. [Online; Accessed 09/07/2015]. 7

[5] Mohammad Ali Azadeh, Abbas Keramati, Hazhir Tolouei, Reza Parvari, and Shima Pashapour. Estimation and optimisation of right-censored data in survival analysis by neural network. *International Journal of Business Information Systems*, 14(3):322–334, 2013. 27

[6] Ana Azevedo and Manuel Filipe Santos. KDD, SEMMA and CRISP-DM: a parallel overview. In *IADIS European Conference on Data Mining 2008,*

Amsterdam, The Netherlands, July 24-26, 2008. Proceedings, pages 182–185, 2008. 82

[7] JSB Chorales. `http://www-etud.iro.umontreal.ca/~boulanni/icml2012`. [Dataset; Online; Accessed 18-September-2015]. 6, 162

[8] Soheil Bahrampour, Naveen Ramakrishnan, Lukas Schott, and Mohak Shah. Comparative study of caffe, neon, theano, and torch for deep learning. *arXiv preprint arXiv:1511.06435*, 2015. 36

[9] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, and Yoshua Bengio. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012. 36, 37, 96, 106

[10] Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1 edition, 1957. 16

[11] Oren Ben-Kiki, Clark Evans, and Ingy dt Net. YAML aint markup language (yaml) version 1.2. `http://yaml.org/spec/1.2/spec.html`, 2016. [Online; Accessed 02-November-2016]. 125

[12] Yoshua Bengio. Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009. 4, 51, 71, 72, 101

[13] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 35(8):1798–1828, 2013. 4, 9

[14] Yoshua Bengio, Pascal Lamblin, Dan Popovici, Hugo Larochelle, et al. Greedy layer-wise training of deep networks. *Advances in neural information processing systems*, 19:153–160, 2007. 77, 147

[15] James Bergstra. Hyperopt. `https://github.com/hyperopt/hyperopt`, 2016. [Online; Accessed 10-November-2016]. 37

[16] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13:281–305, February 2012. 28, 29, 30, 94, 98, 140

[17] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation. 36, 37, 96, 106, 138

[18] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems*, pages 2546–2554, 2011. 29, 98

[19] Derya Birant and Alp Kut. St-dbscan: An algorithm for clustering spatial–temporal data. *Data & Knowledge Engineering*, 60(1):208–221, 2007. 16

[20] Hendrik Blockeel. Experiment databases: A novel methodology for experimental research. In *Knowledge discovery in inductive databases*, pages 72–85. Springer, 2005. 32, 181

[21] Hendrik Blockeel and Joaquin Vanschoren. Experiment databases: Towards an improved experimental methodology in machine learning. In *Knowledge Discovery in Databases: PKDD 2007*, pages 6–17. Springer, 2007. 32, 181

[22] Rajesh Bordawekar, Bob Blainey, and Chidanand Apte. Analyzing analytics. *SIGMOD Record*, 42(4):17–28, 2013. 3

[23] Léon Bottou. Stochastic gradient learning in neural networks. *Proceedings of Neuro-Nımes*, 91(8), 1991. 63, 140

[24] Nicolas Boulanger-Lewandowski, Yoshua Bengio, and Pascal Vincent. Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription. In *ICML*, 2012. 19, 28

[25] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jörg Sander. LOF: identifying density-based local outliers. In *Proceedings of the 2000*

ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA., pages 93–104, 2000. 16

[26] Ricardo J. G. B. Campello, Davoud Moulavi, and Jörg Sander. Density-based clustering based on hierarchical density estimates. In *Advances in Knowledge Discovery and Data Mining, 17th Pacific-Asia Conference, PAKDD 2013, Gold Coast, Australia, April 14-17, 2013, Proceedings, Part II*, pages 160–172, 2013. 16

[27] Gavin C Cawley and Nicola LC Talbot. On over-fitting in model selection and subsequent selection bias in performance evaluation. *Journal of Machine Learning Research*, 11(Jul):2079–2107, 2010. 89, 138

[28] Bach Central. Bach central midi website. `http://www.bachcentral.com/`. [Online; Accessed 18-December-2016]. 162

[29] Soumen Chakrabarti, Martin Ester, Usama Fayyad, Johannes Gehrke, Jiawei Han, Shinichi Morishita, Gregory Piatetsky-Shapiro, and Wei Wang. Data mining curriculum: A proposal (version 0.91). Technical report, Intensive Working Group of ACM SIGKDD Curriculum Committee, 2006. 82, 84

[30] Pete Chapman, Julian Clinton, Randy Kerber, Thomas Khabaza, Thomas Reinartz, Colin Shearer, and Rudiger Wirth. Crisp-dm 1.0 step-by-step data mining guide, 2000. 82, 84

[31] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015. 36

[32] Chih-Lin Chi, W. Nick Street, and William H. Wolberg. Application of artificial neural network-based survival analysis on two breast cancer datasets. In *AMIA 2007, American Medical Informatics Association Annual Symposium, Chicago, IL, USA, November 10-14, 2007*, 2007. 148

[33] Franois Chollet. Keras. `https://github.com/fchollet/keras`, 2016. [Online; Accessed 02-November-2016]. 36

[34] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011. 36

[35] In-MINDD Consortium. In-mindd website. `www.inmindd.eu`, 2014. [Online; Accessed 31/01/14]. 6

[36] Mark W Craven and Jude W Shavlik. Using neural networks for data mining. *Future generation computer systems*, 13(2):211–229, 1997. 2

[37] Michele De Laurentiis and PeterM. Ravdin. Survival analysis of censored data: Neural network analysis detection of complex interactions between variables. *Breast Cancer Research and Treatment*, 32(1):113–118, 1994. 27

[38] Deeplearning4j. Deep learning's accuracy. `https://deeplearning4j.org/accuracy.html`, 2016. [Online; Accessed 02-November-2016]. 5

[39] Deeplearning4j. Dl4j vs. torch vs. theano vs. caffee vs. tensorflow. `https://deeplearning4j.org/compare-dl4j-torch7-pylearn`, 2016. [Online; Accessed 02-November-2016]. 36

[40] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009. 25

[41] Sander Dieleman, Jan Schlter, Colin Raffel, and Others. Lasagne: First release., August 2015. 36

[42] Mike Seltzer Kaisheng Yao Oleksii Kuchaiev Yu Zhang Frank Seide Zhiheng Huang Brian Guenter Huaming Wang Jasha Droppo Geoffrey Zweig Chris Rossbach Jie Gao Andreas Stolcke Jon Currey Malcolm Slaney Guoguo Chen Amit Agarwal Chris Basoglu Marko Padmilac Alexey Kamenev Vladimir

Ivanov Scott Cypher Hari Parthasarathi Bhaskar Mitra Baolin Peng Xuedong Huang Dong Yu, Adam Eversole. An introduction to computational networks and the computational network toolkit. Technical report, Microsoft, October 2014. 36

[43] Cynthia Dwork, Vitaly Feldman, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Aaron Roth. The reusable holdout: Preserving validity in adaptive data analysis. *Science*, 349(6248):636–638, 2015. 182

[44] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96), Portland, Oregon, USA*, pages 226–231, 1996. 16

[45] Diego Esteves, Diego Moussallem, Ciro Baron Neto, Tommaso Soru, Ricardo Usbeck, Markus Ackermann, and Jens Lehmann. Mex vocabulary: a lightweight interchange format for machine learning experiments. In *Proceedings of the 11th International Conference on Semantic Systems*, pages 169–176. ACM, 2015. 32, 34, 83

[46] Shobeir Fakhraei, Hamid Soltanian-Zadeh, Farshad Fotouhi, and Kost Elisevich. Confidence in medical decision making: application in temporal lobe epilepsy data mining. In *Proceedings of the 2011 workshop on Data mining for medicine and healthcare*, pages 60–63. ACM, 2011. 22

[47] Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. From data mining to knowledge discovery in databases. *AI magazine*, 17(3):37, 1996. 1, 82

[48] Ugo Fiore, Francesco Palmieri, Aniello Castiglione, and Alfredo De Santis. Network anomaly detection with the restricted boltzmann machine. *Neurocomputing*, 122(0):13 – 23, 2013. Advances in cognitive and ubiquitous computing Selected papers from the Sixth International Conference on Innovative

Mobile and Internet Services in Ubiquitous Computing (IMIS-2012). 5, 18, 19, 168

[49] G. David Garson. Interpreting neural-network connection weights. *AI Expert*, 6(4):46–51, April 1991. 23

[50] Anup K Ghosh and Aaron Schwartzbard. A study in using neural networks for anomaly and misuse detection. In *USENIX Security*, 1999. 18

[51] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *International conference on artificial intelligence and statistics*, pages 249–256, 2010. 48, 49

[52] A.T.C. Goh. Back-propagation neural networks for modeling complex systems. *Artificial Intelligence in Engineering*, 9(3):143 – 151, 1995. 23

[53] Alex Graves. *Supervised sequence labelling.* Springer, 2012. 26, 41, 69

[54] Robert Grossman, Stuart Bailey, Ashok Ramu, Balinder Malhi, Philip Hallstrom, Ivan Pulleyn, and Xiao Qin. The management and mining of multiple predictive models using the predictive modeling markup language. *Information and Software Technology*, 41(9):589–595, 1999. 32, 33

[55] Harvard Intelligent Probabilistic Systems Group. Hypergrad. `https://github.com/HIPS/hypergrad`, 2016. [Online; Accessed 10-November-2016]. 37

[56] The Data Mining Group. Pmml online specification. `http://dmg.org/pmml/v4-2-1/GeneralStructure.html`. [Online; Last accessed 30/03/2016]. 32, 33

[57] Jiawei Han, Micheline Kamber, and Jian Pei. *Data mining: concepts and techniques.* Morgan kaufmann, 2006. 1, 2, 82, 84

[58] Geoffrey Hinton. A practical guide to training restricted boltzmann machines. *Momentum*, 9(1):926, 2010. 70

[59] Geoffrey E Hinton. Training products of experts by minimizing contrastive divergence. *Neural computation*, 14(8):1771–1800, 2002. 75

[60] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006. 4, 5, 77

[61] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006. 5, 9, 25

[62] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012. 25

[63] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997. 4, 26

[64] Gary B Huang, Manu Ramesh, Tamara Berg, and Erik Learned-Miller. Labeled faces in the wild: A database for studying face recognition in unconstrained environments. Technical report, Technical Report 07-49, University of Massachusetts, Amherst, 2007. 25

[65] Eric J Humphrey, Juan P Bello, and Yann LeCun. Feature learning and deep architectures: new directions for music informatics. *Journal of Intelligent Information Systems*, 41(3):461–481, 2013. 8, 9, 28, 29

[66] MongoDB Inc. The MongoDB 3.2 manual. `https://docs.mongodb.com/manual`, 2016. [Online; Accessed 02-November-2016]. 126

[67] ECMA International. The JSON data interchange format. `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf`, 2013. [Online; Accessed 02-November-2016]. 125

[68] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014. 36

[69] Ian Jolliffe. *Principal component analysis*. Wiley Online Library, 2002. 23

[70] JSON. JSON.org. `http://www.json.org/`, 2016. [Online; Accessed 02-November-2016]. 125

[71] KDnuggets. Popular deep learning tools - a review. `http://www.kdnuggets.com/2015/06/popular-deep-learning-tools.html`, 2016. [Online; Accessed 09-November-2016]. 36

[72] C Keet, Claudia dAmato, Zubeida Khan, and Agnieszka Lawrynowicz. Exploring reasoning with the DMOP ontology. In *3rd Workshop on Ontology Reasoner Evaluation (ORE14)*, volume 1207, pages 64–70, 2014. 32, 34, 83

[73] Hans-Peter Kriegel, Peer Kröger, Erich Schubert, and Arthur Zimek. Loop: local outlier probabilities. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 1649–1652. ACM, 2009. 17

[74] Hans-Peter Kriegel, Peer Kröger, Erich Schubert, and Arthur Zimek. Interpreting and unifying outlier scores. In *SDM*, pages 13–24. SIAM, 2011. 17

[75] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009. 25

[76] Hugo Larochelle, Dumitru Erhan, Aaron Courville, James Bergstra, and Yoshua Bengio. An empirical evaluation of deep architectures on problems with many factors of variation. In *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, pages 473–480, New York, NY, USA, 2007. ACM. 29, 30, 94

[77] Quoc V Le. Building high-level features using large scale unsupervised learning. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8595–8598. IEEE, 2013. 5, 24, 25

[78] Quoc V Le, Navdeep Jaitly, and Geoffrey E Hinton. A simple way to initialize recurrent networks of rectified linear units. *arXiv preprint arXiv:1504.00941*, 2015. 5, 53

[79] Timothy Lebo, Satya Sahoo, Deborah McGuinness, K Belhajjame, J Cheney, et al. Prov-o: The prov ontology. w3c recommendation, 30 april 2013. *World Wide Web Consortium*, 2013. 32, 34

[80] Yann Lecun and Corinna Cortes. The MNIST database of handwritten digits. `http://yann.lecun.com/exdb/mnist/`. [Dataset; Online; Accessed 16-November-2016]. 5, 25

[81] Kingsly Leung and Christopher Leckie. Unsupervised anomaly detection in network intrusion detection using clusters. In *Proceedings of the Twenty-eighth Australasian conference on Computer Science-Volume 38*, pages 333–342. Australian Computer Society, Inc., 2005. 17

[82] Znaonui Liang, Gang Zhang, Jimmy Xiangji Huang, and Qmming Vivian Hu. Deep learning for healthcare decision making with emrs. In *Bioinformatics and Biomedicine (BIBM), 2014 IEEE International Conference on*, pages 556–559. IEEE, 2014. 26

[83] Moshe Lichman. 1999 kdd cup dataset, UCI machine learning repository, 2013. Dataset; Available on: `https://archive.ics.uci.edu/ml/datasets/KDD+Cup+1999+Data`. 17, 19

[84] Seymour Lipschutz. *Schaum's outline of essential computer mathematics.* McGraw-Hill, 1987. 41

[85] Paulo JG Lisboa. A review of evidence of health benefit from artificial neural networks in medical intervention. *Neural networks*, 15(1):11–39, 2002. 27

[86] P.J.G. Lisboa, H. Wong, A. Vellido, S.P.J. Kirby, P. Harris, and R. Swindeil. Survival of breast cancer patients following surgery: a detailed assessment of the multi-layer perceptron and cox's proportional hazard model. In *Neural Networks Proceedings, 1998. The 1998 IEEE International Joint Conference on*, volume 1, pages 112–116 vol.1, May 1998. 27

[87] Chao Liu, Sambuddha Ghosal, Zhanhong Jiang, and Soumik Sarkar. An unsupervised spatiotemporal graphical modeling approach to anomaly detection in

distributed cps. In *2016 ACM/IEEE 7th International Conference on Cyber-Physical Systems (ICCPS)*, pages 1–10. IEEE, 2016. 18, 19

[88] Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Gradient-based hyperparameter optimization through reversible learning. In *Proceedings of the 32nd International Conference on Machine Learning*, 2015. 29, 30

[89] João Maroco, Dina Silva, Ana Rodrigues, Manuela Guerreiro, Isabel Santana, and Alexandre de Mendonça. Data mining methods in the prediction of dementia: A real-data comparison of the accuracy, sensitivity and specificity of linear discriminant analysis, logistic regression, neural networks, support vector machines, classification trees and random forests. *BMC research notes*, 4(1):299, 2011. 27, 28

[90] Joo Maroco, Dina Silva, Manuela Guerreiro, Alexandre de Mendona, and Isabel Santana. Prediction of dementia patients: A comparative approach using parametric versus nonparametric classifiers. In *Advances in Regression, Survival Analysis, Extreme Values, Markov Processes and Other Statistical Applications*, Studies in Theoretical and Applied Statistics, pages 269–280. Springer Berlin Heidelberg, 2013. 27

[91] Wes McKinney. Data structures for statistical computing in python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56, 2010. 138

[92] Clay McLeod. The truth about deep learning. `http://www.kdnuggets.com/2016/06/truth-deep-learning.html`, 2016. [Online; Accessed 06-January-2017]. 183

[93] Tom M Mitchell. *Machine learning. WCB.* McGraw-Hill Boston, MA:, 1997. 2

[94] ML-Schema Group. W3C machine learning schema community group. `https://www.w3.org/community/ml-schema/`, 2016. [Online; Accessed 09-November-2016]. 135, 181

[95] MongoDB. Pymongo. `https://api.mongodb.com/python/current/`, 2016. [Online; Accessed 12-December-2016]. 138

[96] Andrew Ng. Coursera and Stanford: machine learning. `https://www.coursera.org/course/ml`. [Online; Accessed 10-July-14]. 41

[97] Maria C Norton, Jeffrey Dew, Heeyoung Smith, Elizabeth Fauth, Kathleen W Piercy, John Breitner, JoAnn Tschanz, Heidi Wengreen, and Kathleen Welsh-Bohmer. Lifestyle behavior pattern is associated with different levels of risk for incident dementia and alzheimer's disease: the cache county study. *Journal of the American Geriatrics Society*, 60(3):405–412, 2012. 23

[98] Nurzhan Nurseitov, Michael Paulson, Randall Reynolds, and Clemente Izurieta. Comparison of json and xml data interchange formats: A case study. *Caine*, 2009:157–162, 2009. 33, 34, 126

[99] Nvidia. Deep learning frameworks. `https://developer.nvidia.com/deep-learning-frameworks`, 2016. [Online; Accessed 09-November-2016]. 36

[100] Nvidia. Digits. `https://github.com/NVIDIA/DIGITS`, 2016. [Online; Accessed 02-November-2016]. 11, 36

[101] Jim O'Donoghue and Mark Roantree. A framework for selecting deep learning hyper-parameters. In Sebastian Maneth, editor, *Data Science - 30th British International Conference on Databases, BICOD 2015, Edinburgh, UK, July 6-8, 2015, Proceedings*, volume 9147 of *Lecture Notes in Computer Science*, pages 120–132. Springer International Publishing, 2015. 30

[102] Jim O'Donoghue and Mark Roantree. A toolkit for analysis of deep learning experiments. In *Advances in Intelligent Data Analysis XV - 15th International Symposium, IDA 2016, Stockholm, Sweden, October 13-15, 2016, Proceedings*, pages 134–145, 2016. 135, 179, 181

[103] Jim O'Donoghue, Mark Roantree, and Martin Van Boxtel. A configurable deep network for high-dimensional clinical trial data. In *Neural Networks (IJCNN), 2014 International Joint Conference on*. IEEE, July 2015. 27

[104] Jim O'Donoghue, Mark Roantree, and Andrew McCarren. Variable interactions in risk factors for dementia. In *Research Challenges in Information Science (RCIS), 2016 IEEE Tenth International Conference on*, pages 1–10. IEEE, 2016. 147

[105] Julian D Olden and Donald A Jackson. Illuminating the black box: a randomization approach for understanding variable contributions in artificial neural networks. *Ecological modelling*, 154(1):135–150, 2002. 23

[106] Julian D Olden, Michael K Joy, and Russell G Death. An accurate comparison of methods for quantifying variable importance in artificial neural networks using simulated data. *Ecological Modelling*, 178(34):389 – 397, 2004. 23

[107] OpenML. Openml.org website. `http://www.openml.org/`, 2016. [Online; Accessed 06-January-2017]. 181

[108] Dan Pelleg, Andrew W Moore, et al. X-means: Extending k-means with efficient estimation of the number of clusters. In *ICML*, pages 727–734, 2000. 18

[109] PFA online specification. `http://dmg.org/pfa/index.html`. [Online; Accessed 30-March-2016]. 32, 33

[110] Gregory Piatetsky-Shapiro. KDnuggets Polls: Data Mining Methodology. `http://www.kdnuggets.com/polls/2014/analytics-data-mining-data-science-methodology.html`, 2014. [Online; Last accessed 19/11/2016]. 82, 84

[111] Jim Pivarski, Collin Bennett, and Robert L. Grossman. Deploying analytics with the portable format for analytics (PFA). In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 579–588, 2016. 33

[112] Lutz Prechelt. Early stopping-but when? In *Neural Networks: Tricks of the trade*, pages 55–69. Springer, 1998. 63, 140

[113] Stephen M Ross. Peirce's criterion for the elimination of suspect experimental data. *Journal of Engineering Technology*, 20(2):38–41, 2003. 133

[114] Adam Sadilek, Henry A. Kautz, and Vincent Silenzio. Predicting disease transmission from geo-tagged micro-blog data. In *AAAI*, 2012. 26, 27

[115] Ruslan Salakhutdinov and Geoffrey E Hinton. Deep boltzmann machines. In *International Conference on Artificial Intelligence and Statistics*, pages 448–455, 2009. 5

[116] Mostafa A Salama, Heba F Eid, Rabie A Ramadan, Ashraf Darwish, and Aboul Ella Hassanien. Hybrid intelligent intrusion detection scheme. In *Soft computing in industrial applications*, pages 293–303. Springer, 2011. 18, 19

[117] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015. 4

[118] Hon-Yi Shi, King-Teh Lee, Hao-Hsien Lee, Wen-Hsien Ho, Ding-Ping Sun, Jhi-Joung Wang, and Chong-Chi Chiu. Comparison of artificial neural network and logistic regression models for predicting in-hospital mortality after primary liver cancer surgery. *Plos One*, 7, 2012. 27

[119] Mei-Ling Shyu, Shu-Ching Chen, Kanoksri Sarinnapakorn, and LiWu Chang. A novel anomaly detection scheme based on principal component classifier. Technical report, DTIC Document, 2003. 18

[120] Jasper Snoek. Spearmint. `https://github.com/JasperSnoek/spearmint`, 2016. [Online; Accessed 10-November-2016]. 37

[121] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems*, pages 2951–2959, 2012. 29, 30, 32, 98

[122] Qing Song, Wenjie Hu, and Wenfang Xie. Robust support vector machine with bullet hole image classification. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 32(4):440–448, Nov 2002. 17

[123] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014. 102

[124] Nervana Systems. Neon. `https://github.com/NervanaSystems/neon`, 2016. [Online; Accessed 02-November-2016]. 36

[125] Theano Team. Getting started: Notation. `http://deeplearning.net/tutorial/gettingstarted.html`. [Online; Accessed 02-December-2016]. 41

[126] Teglor. Deep learning libraries by language. `http://www.teglor.com/b/deep-learning-libraries-language-cm569/`, 2016. [Online; Accessed 09-November-2016. 36

[127] The Data Mining Group. PFA: Portable format for analytics (version 0.8.1) specification. Technical report, Data Mining Group - PFA Working Group, 2015. 33

[128] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016. 36, 37

[129] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*, 2015. 36

[130] Martin PJ van Boxtel, Frank Buntinx, Peter J Houx, Job FM Metsemakers, André Knottnerus, and Jellemer Jolles. The relation between morbidity and cognitive performance in a normal aging population. *The Journals of Gerontology Series A: Biological Sciences and Medical Sciences*, 53(2):M147–M154, 1998. The Maastricht Ageing Study, Determinants of Cognitive Ageing - Check. 6

[131] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011. 138

[132] Bart Van Merriënboer, Dzmitry Bahdanau, Vincent Dumoulin, Dmitriy Serdyuk, David Warde-Farley, Jan Chorowski, and Yoshua Bengio. Blocks and fuel: Frameworks for deep learning. *arXiv preprint arXiv:1506.00619*, 2015. 36

[133] Joaquin Vanschoren, Hendrik Blockeel, Bernhard Pfahringer, and Geoffrey Holmes. Experiment databases. *Machine Learning*, 87(2):127–158, 2012. 32, 124, 181

[134] Joaquin Vanschoren, Jan N van Rijn, and Bernd Bischl. Taking machine learning research online with OpenML. In *Proceedings of the 4th International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*, pages 1–4, 2015. 32, 181

[135] Joaquin Vanschoren, Jan N Van Rijn, Bernd Bischl, and Luis Torgo. Openml: networked science in machine learning. *ACM SIGKDD Explorations Newsletter*, 15(2):49–60, 2014. 32

[136] Li Wan, Matthew Zeiler, Sixin Zhang, Yann L Cun, and Rob Fergus. Regularization of neural networks using dropconnect. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1058–1066, 2013. 25

[137] Ian H Witten and Eibe Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005. 2, 25

[138] DF Wulsin, JR Gupta, R Mani, JA Blanco, and B Litt. Modeling electroencephalography waveforms with semi-supervised deep belief nets: fast classification and anomaly measurement. *Journal of neural engineering*, 8(3):036015, 2011. 18, 19, 20

[139] Antonio Jimeno Yepes, Andrew MacKinlay, Justin Bedo, Rahil Garnavi, and Qiang Chen. Deep belief networks and biomedical text categorisation. In *Australasian Language Technology Association Workshop 2014*, page 123, 2014. 26

[140] Shuangfei Zhai, Yu Cheng, Weining Lu, and Zhongfei Zhang. Deep structured energy based models for anomaly detection. *arXiv preprint arXiv:1605.07717*, 2016. 18, 19, 20, 168, 176

[141] Shichao Zhang, Chengqi Zhang, and Qiang Yang. Data preparation for data mining. *Applied Artificial Intelligence*, 17(5-6):375–381, 2003. 85

[142] Stacy L zesmi and Uygar zesmi. An artificial neural network approach to spatial habitat modelling with interspecific interaction. *Ecological Modelling*, 116(1):15 – 31, 1999. 23

Glossary

bootstrap resampling Selecting a number of random samples with replacement from a dataset. 138

collinearity Input variables which are highly correlated, where one input variable can be used to predict another. 56

cost function Gives a measure of model performance, which we aim to minimise in order to improve model accuracy and for example, make better predictions. 58

firing When the state of a node is equal to 1. 50

hyper-parameter A parameter which is input to the model parameter learning process which affects how effectively model parameters are updated and ultimately, learnt. 98, 114

hyper-parameter configuration A single set of hyper-parameters, used for the instantiation and optimisation of a single deep learning algorithm instance. 98

hypothesis function Takes data as input and outputs the learner hypothesis, for example a classification or reconstruction of the data. 56, 97

learner Our conceptual construct in the Configurable Deep Network, which refers to a single instance of a machine learning algorithm. 100

learning rate Determines the magnitude of gradient descent updates, the size of the 'step' down the slope of the cost function. 62

meta-algorithm An algorithm which consists of other nested algorithms deployed as one of the parent process' functional steps. 107

overfitting When models perform very well on data they are trained on but do not generalise satisfactorily to unseen instances.. 60

patience Hyper-parameter which determines the minimum number of iterations of a gradient descent procedure to perform. 63

propagation The process of transforming values through a node, layer or entire neural network. 49

run The process of taking a learning algorithm from instantiation to optimisation through some sort of training functions. 98

state Whether a node fires or not, outputting 1 if it does, otherwise outputting a 0 if it does not. 51

trial A concept which captures *trialling* a particular hyper-parameter configuration by instantiating and optimising either a single or multiple learning algorithm instances. 98

trial-Run A single optimised learning algorithm for a single particular hyper-parameter configuration. 109

update A single amendment to model parameters which incrementally improves a models performance in relation to a cost function. 98

Appendices

Appendix A

Notation

Table A.1: Mathematical Notation

Symbol	Meaning
$\mathcal{D}$	Dataset
$\mathcal{X}$	Input space
X	Data matrix
x_{ij}	Data element, value for ith feature in jth sample
$\mathcal{Y}$	Target Space
Y	Classification Matrix
y	Target element
m	Number of data samples
n	Number of data features
$L^{(l)}$	Layer l layer in neural network
$W^{(l)}$	Weight Matrix for lth layer
$W_{ij}^{(l)}$	Weight between ith input and jth node in layer l
$b^{(l)}$	Bias Vector for lth layer
$b_j^{(l)}$	Bias term for the jth node in lth layer
f	Linear function
$z^{(l)}$	Vector of linear activations for lth layer
$z_i^{(l)}$	Value for linear activation for ith node in lth layer
g	Non-linear activation function
$a^{(l)}$	Vector of non-linear activations for lth layer
$a_i^{(l)}$	Value for non-linear activation for ith node in lth layer
h_θ	Hypothesis function, parameterised by θ
$J(h_\theta(x))$ or $J(\theta)$	Cost of hypothesis output
$N_i^{(l)}$	Node i in layer l

Appendix B

Activations and Hypotheses

B.1 Node Level Linear Activation Representations

Feed-Forward Node. In Equation (B.1), the linear activation $z_o^{(l)}$, for the oth node in the lth layer is calculated by taking each input variable $a_1^{(l-1)}, \dots, a_n^{(l-1)}$ multiplying each $a_i^{(l)}$ by the relevant weight $w_{1o}^{(l)}, \dots, w_{no}^{(l)}$ and adding the relevant scalar bias term b_o to the final weighted sum.

$$z_o^{(l)} = f(a^{(l-1)}) = \sum_{i=1}^{n} a_i^{(l-1)} w_{io}^{(l)} + b_o \tag{B.1}$$

Recurrent Node. Equation B.2 shows the calculation which occurs at each recurrent node. The linear activation $z_j^{(l)(t)}$, for the jth node, in the lth layer at the t time-point is calculated by taking each input variable $a_1^{(l-1)(t)}, \dots, a_n^{(l-1)(t)}$ in the previous $l-1$ layer at the current time point t and multiplying each by the relevant input weight $w_{1o}^{(l)}, \dots, w_{no}^{(l)}$. Then we multiply the activations calculated for the current lth layer at the previous time-point $t-1$: $a_1^{(l)(t-1)}, \dots, a_n^{(l)(t-1)}$ by the relevant recurrent weight $w_{1o}^{(l)}, \dots, w_{no}^{(l)}$. Finally, we add the scalar bias term b_o to the final weighted sum.

$$z_j^{(l)(t)} = f(a^{(l-1)(t)}, a^{(l)(t-1)}) = \sum_{i=1}^{n} a_i^{(l-1)(t)} W_{ij} + \sum_{k=1}^{o^{(t-1)}} a^{(l)(t-1)} WR_{kj} + b_j \tag{B.2}$$

B.2 Alternative Rectified Linear Units

The noisy ReLU

$$a^{(l)} = g(z^{(l)}) = NoisyReLU(z^{(l)}) = max(0, z^{(l)} + n); \quad n{\sim}\mathcal{N}\left(\mu, \sigma(z^{(l)})\right) \tag{B.3}$$

where Gaussian noise is added to the linear output when greater than 0. The learner then learns a more robust model. The input to the leaky ReLU

$$a^{(l)} = g(z^{(l)}) = LeakyReLU(z^{(l)}) \ = \begin{cases} z^{(l)} & \text{if } z^{(l)} > 0 \\ 0.01z^{(l)} & \text{otherwise} \end{cases} \tag{B.4}$$

is multiplied by an (arbitrary) coefficient of 0.01 when the input is less than zero, giving a small non-zero gradient during learning. The parametric ReLU

$$a^{(l)} = g(z^{(l)}) = ParamReLU(z^{(l)}) = \begin{cases} z^{(l)} & \text{if } z^{(l)} > 0 \\ cz^{(l)} & \text{otherwise} \end{cases} \tag{B.5}$$

learns the co-efficient c in order establish a more robust leakage parameter.

B.3 Deriving Free Energy Functions

In the follwing equations x represents a vector of visible units, this is normally represented with v but to keep the notation used throughout the dissertation homogeneous we use x.

The probability function where there are only visible units, x is a vector of visible units and E is the energy function:

$$P(x) = \frac{e^{-E(x)}}{\sum\limits_{x} e^{-E(x)}} \tag{B.6}$$

The probability function where there are also hidden units:

$$P(x) = \sum_{h} P(x,h) = \frac{e^{-E(x,h)}}{\sum\limits_{x}\sum\limits_{h} e^{-E(x,h)}} \tag{B.7}$$

And the free energy is related to the energy function by:

$$\mathcal{F}(x) = -ln \sum_{h} e^{-E(x,h)} \tag{B.8}$$

The energy of a **Bernoulli RBM** is:

$$E(x,h) = -xb^{(0)\mathsf{T}} - hb^{(1)\mathsf{T}} - xWh^{\mathsf{T}} \tag{B.9}$$

and the energy function of a **Gaussian Bernoulli RBM** is:

$$E(x,h) = -\sum_{i \in visible} \frac{(x_i - b_i^{(0)})^2}{2\sigma_i^2} - \sum_{j \in hidden} b_j^{(1)} h_j - \sum_{i,j} \frac{x_i}{\sigma_i} W_{ij} h_j. \tag{B.10}$$

Appendix C

Cost Functions

C.1 Deriving Negative Log Likelihood from Likelihood

$$
\begin{aligned}
\mathcal{L}(\theta;\mathcal{D}) &= P(\mathcal{D};\theta) && \text{(C.1a)}\\
&= \prod_{i=0}^{|\mathcal{D}|} P(y_{ic}|x_{i\bullet};\theta) && \text{(C.1b)}\\
-\log\mathcal{L}(\theta;\mathcal{D}) &= -\log\prod_{i=0}^{|\mathcal{D}|} P(y_{ic}|x_{i\bullet};\theta) && \text{(C.1c)}\\
&= -\sum_{i=0}^{|\mathcal{D}|} \log P(y_{ic}|x_{i\bullet};\theta) && \text{(C.1d)}
\end{aligned}
$$

Appendix D

Partial Derivatives

D.1 Derivative Rules

D.1.1 Chain Rule

If a function $g(z)$ takes the output of another function as input in the form $g(f(x))$ where $z = f(x)$ the chain rule states that:

$$\frac{\partial g(f(x))}{\partial x} = \frac{\partial g(z)}{\partial z}\frac{\partial f(x)}{\partial x}, \tag{D.1}$$

the derivative of the outer function with respect to (wrt) x is equivalent to the derivative of the outer function wrt z times the derivative of the inner function wrt the x.

D.1.2 Quotient Rule

If a function $h(z)$ can be expressed as the quotient of two other functions $h(z) = \frac{f(z)}{g(z)}$ than the quotient rule states that:

$$\frac{\partial h(z)}{\partial z} = \frac{f'(z)g(z) - g'(z)f(z)}{[g(z)]^2} \quad \text{where } g'(z) = \frac{\partial g(z)}{\partial z} \text{ and } f'(z) = \frac{\partial f(z)}{\partial z} \tag{D.2}$$

D.2 Activation Function Partial Derivatives

D.2.1 Derivative Linear Function wrt Theta

$$f'(x) = \frac{\partial f(x)}{\partial \theta} = \frac{\partial x\theta}{\partial \theta} = x \tag{D.3}$$

D.2.2 Derivative of Sigmoid Function wrt Input

$$\begin{aligned}
g'(z) &= \frac{\partial g(z)}{\partial z} && && \text{(D.4a)}\\
&= \frac{\partial(\frac{1}{1+e^{-z}})}{\partial z} && && \text{(D.4b)}\\
&= \frac{\partial(1+e^{-z})^{-1}}{\partial z} && && \text{(D.4c)}\\
&= \frac{\partial u^{-1}}{\partial u}\frac{\partial u}{\partial z} && \text{letting } (1+e^{-z}) = u && \text{(D.4d)}\\
&= -u^{-2}(-e^{-z}) && && \text{(D.4e)}\\
&= -(1+e^{-z})^{-2}(-e^{-z}) && \text{substituing back for } u && \text{(D.4f)}\\
&= \frac{-1}{(1+e^{-z})^2}(-e^{-z}) && && \text{(D.4g)}\\
&= \frac{1}{1+e^{-z}}\frac{e^{-z}}{1+e^{-z}} && && \text{(D.4h)}\\
&= \frac{1}{1+e^{-z}}\left(1 - \frac{1}{1+e^{-z}}\right) && \text{as} \quad \frac{1}{1+e^{-z}} + \frac{e^{-z}}{1+e^{-z}} = 1 && \text{(D.4i)}\\
&= g(z)(1-g(z)) \quad \text{or} \quad a(1-a) && && \text{(D.4j)}
\end{aligned}$$

D.2.3 Derivative of Softmax Function wrt Hypothesis Output

When calculating the point-wise derivative for the softmax function there are in fact two different derivatives that we have to calculate. The cost function used only calculates the negative log probability for the kth value of a, a_k, that corresponds to the correct classification. The reason for this is we have a binary class vector where only the kth value is equal to one. Therefore, we have to calculate the derivatives of the output a_k wrt to the input value z_k. Importantly, we also have to calculate the derivative of a_k with respect to the other inputs to the softmax function z_i where $k \neq i$ and k corresponds to the index of the correct class in the target vector y. Therefore, $a_k = p(a_k = y_k|z)$. In the case of $i = k$ the derivative is

$$
\begin{aligned}
\frac{\partial a_i}{\partial z_i} &= \frac{\partial(\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}})}{\partial z_i} && && \text{(D.5a)}\\
&= \frac{\partial \frac{e^{z_i}}{u}}{\partial z_i} && \text{let } \sum_{j=1}^{K} e^{z_j} = u && \text{(D.5b)}\\
&= \frac{e^{z_i} u - e^{z_k} e^{z_i}}{u^2} && \text{quotient rule and } \frac{\partial u}{\partial z_i} = e^{z_i} && \text{(D.5c)}\\
&= \frac{e^{z_i}}{u}\left(\frac{u - e^{z_i}}{u}\right) && && \text{(D.5d)}\\
&= \frac{e^{z_i}}{u}\left(1 - \frac{e^{z_i}}{u}\right) && && \text{(D.5e)}\\
&= \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}\left(1 - \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}\right) && \text{substituting back in for } u && \text{(D.5f)}\\
&= a_i(1 - a_i) && \text{note equivalence to sigmoid derivative} && \text{(D.5g)}
\end{aligned}
$$

and when $k \neq i$ the derivative is:

$$
\begin{aligned}
\frac{\partial a_k}{\partial z_i} &= \frac{\partial(\frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}})}{\partial z_i} && && \text{(D.6a)}\\
&= \frac{\partial \frac{e^{z_k}}{u}}{\partial z_i} && \text{let } \sum_{k=1}^{K} e^{z_k} = u && \text{(D.6b)}\\
&= \frac{0(u) - e^{z_i} e^{z_k}}{u^2} && \text{quotient rule and } \frac{\partial u}{\partial z_i} = e^{z_i} && \text{(D.6c)}\\
&= -\frac{e^{z_i}}{u}\frac{e^{z_k}}{u} && && \text{(D.6d)}\\
&= -\frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}}\frac{e^{z_k}}{\sum_{k=1}^{K} e^{z_k}} && && \text{(D.6e)}\\
&= -a_i a_k. && && \text{(D.6f)}
\end{aligned}
$$

D.3 Cost Function Partial Derivatives

D.3.1 Derivative of Mean Squared Error with respect to Parameters

In Line D.7b, we substitute the mean square error cost function from Equation (3.13). In Line D.7c, as $h_\theta(x) = f(x) = z$ for linear regression as shown in Equation (3.9), we substitute in z for $h_\theta(x)$ for clarity. Furthermore, we extract the constant $\frac{1}{2}$ and get the derivative of $(z-y)^2$, giving $2(z-y)$. Finally, as $z = x\theta$, from Equation (3.3), we substitute this in Line D.7d. The derivative of $x\theta$ is simply x which gives us the result in the second last line, and finally, in the last line we show another formulation where we substitute back in $h_\theta(x)$ for z.

$$\frac{\partial J_{mse}(\theta)}{\partial \theta} = \frac{\partial \frac{1}{2}(h_\theta(x) - y)^2}{\partial \theta} \tag{D.7a}$$

$$= 2 \cdot \frac{1}{2}(z - y) \cdot \frac{\partial (z - y)}{\partial \theta} \tag{D.7b}$$

$$= (z - y) \cdot \frac{\partial (x\theta - y)}{\partial \theta} \tag{D.7c}$$

$$= (z - y)x \tag{D.7d}$$

$$= (h_\theta(x) - y)x \tag{D.7e}$$

D.3.2 Derivative of Cross Entropy Cost wrt a

$$
\begin{aligned}
\frac{\partial J(\theta)}{\partial a} &= \frac{\partial - (y \log(h_\theta(x)) - (1-y)\log(1-h_\theta(x))}{\partial a} && \text{(D.8a)}\\
&= \frac{\partial - (y \log a - (1-y)\log(1-a))}{\partial a} && \text{(D.8b)}\\
&= -\left[\frac{\partial y \log a}{\partial a} - \frac{\partial (1-y)\log(1-a)}{\partial a}\right] && \text{(D.8c)}\\
&= -\left[y\frac{\partial \log a}{\partial a} - (1-y)\frac{\partial \log(1-a)}{\partial a}\right] && \text{(D.8d)}\\
&= -\left[y\frac{1}{a} - (1-y)\frac{1}{(1-a)}\right] && \text{(D.8e)}\\
&= -\left[\frac{y}{a} - \frac{(1-y)}{(1-a)}\right] && \text{(D.8f)}\\
&= -\left[\frac{(1-a)y - (a(1-y))}{a(1-a)}\right] && \text{(D.8g)}\\
&= -\left[\frac{(y - ay - a + ay))}{a(1-a)}\right] && \text{(D.8h)}\\
&= \frac{a-y}{a(1-a)} && \text{(D.8i)}
\end{aligned}
$$

D.3.3 Derivative of Negative Log Likelihood wrt a

The difference between softmax and logistic regression is that there are multiple different linear inputs to the non-linear activation function whereas in logistic regression there is only one. Therefore, when computing the derivatives you have to get the derivative of the output with respect to each input as there is only one non-zero term in the cost function. That means when getting the derivative for each linear z_i where $i \in 1, \ldots o^{(l)}$, the number of nodes in the output layer, you have to get its derivative wrt to one output a_j where $j \in 1, \ldots K$ where K is the number of possible classifications. That means there are two types of gradients, one when you are getting the gradient of the node whose output is the classification and the other is when you are getting the gradients of the nodes whose output isn't the classification.

$$\frac{\partial J(\theta)}{\partial a_k} = \frac{\partial - \sum_{k=1}^{K} y_k \log h_\theta(x)}{\partial a_k} \tag{D.9a}$$

$$= \frac{\partial - \sum_{k=1}^{K} y_k \log a_k}{\partial a_k} \tag{D.9b}$$

$$= -\sum_{k=1}^{K} y_k \frac{\partial log a_k}{\partial a_k} \tag{D.9c}$$

$$= -\sum_{k=1}^{K} y_k \frac{1}{a_k} \tag{D.9d}$$

$$\tag{D.9e}$$

D.3.4 Derivative of Negative Log Likelihood wrt Theta

Line D.10b shows the substitution of the first term, that is the derivative of the negative log likelihood cost with respect to the output probability of the correct classification derived in Equation (D.9), as well as the last term which is the derivative of the linear function with respect to to all parameters. Line D.10c shows the substitution for the middle term of the Equation with that shown in Equation (D.5) for when $k = i$, that from Equation (D.6) when $k \neq i$. The following lines then show a simplification of these terms. Note that even though the derivative is much more complicated we end up with the same derivative as outlined in Equation (3.19) for logistic regression with only one output node.

$$
\begin{aligned}
\frac{\partial J_{nll}(\theta)}{\partial z_i} &= -\sum_{k=1}^{K} \frac{y_k}{a_k} \cdot \frac{\partial a_k}{\partial z_i} \cdot x && \text{(D.10a)}\\
&= \left[-\frac{y_i}{a_i} a_i(1-a_i) - \sum_{i \neq k}^{K} \frac{y_k}{a_k}(-a_i a_k) \right] x && \text{(D.10b)}\\
&= \left[-y_i(1-a_i) - \sum_{i \neq k}^{K} y_k(-a_i) \right] x && \text{(D.10c)}\\
&= \left[-y_i + y_i a_i + \sum_{i \neq k}^{K} y_k a_i \right] x && \text{(D.10d)}\\
&= \left[-y_i + \sum_{i=1}^{K} y_k a_i \right] x && \text{(D.10e)}\\
&= \left[-y_i + \sum_{i=1}^{K} y_k (a_i) \right] x && \text{(D.10f)}\\
&= -(y_i + a_i)x \qquad \text{as} \sum_{i=1}^{K} y_k = 1 && \text{(D.10g)}\\
&= (a_i - y_i)x && \text{(D.10h)}\\
&= (h_\theta(x) - y)x && \text{(D.10i)}
\end{aligned}
$$

Appendix E

Storage Overview

Storage Size. Table E.1 shows the average size for each *snapshot* stored in bytes and the total size of each experiment stored in giga-abytes.

Table E.1: Storage Size Statistics

Experiment ID	Avg. Snap.(B)	Exp.(GB)
gaa_rnn_05-04-16_11:48:02	62,177.92	1.546
maas_dbn_03-01-17_12:32:09	19,578.12	0.087
bach_srbm_31-12-16_13:53:53	57,747.88	0.402

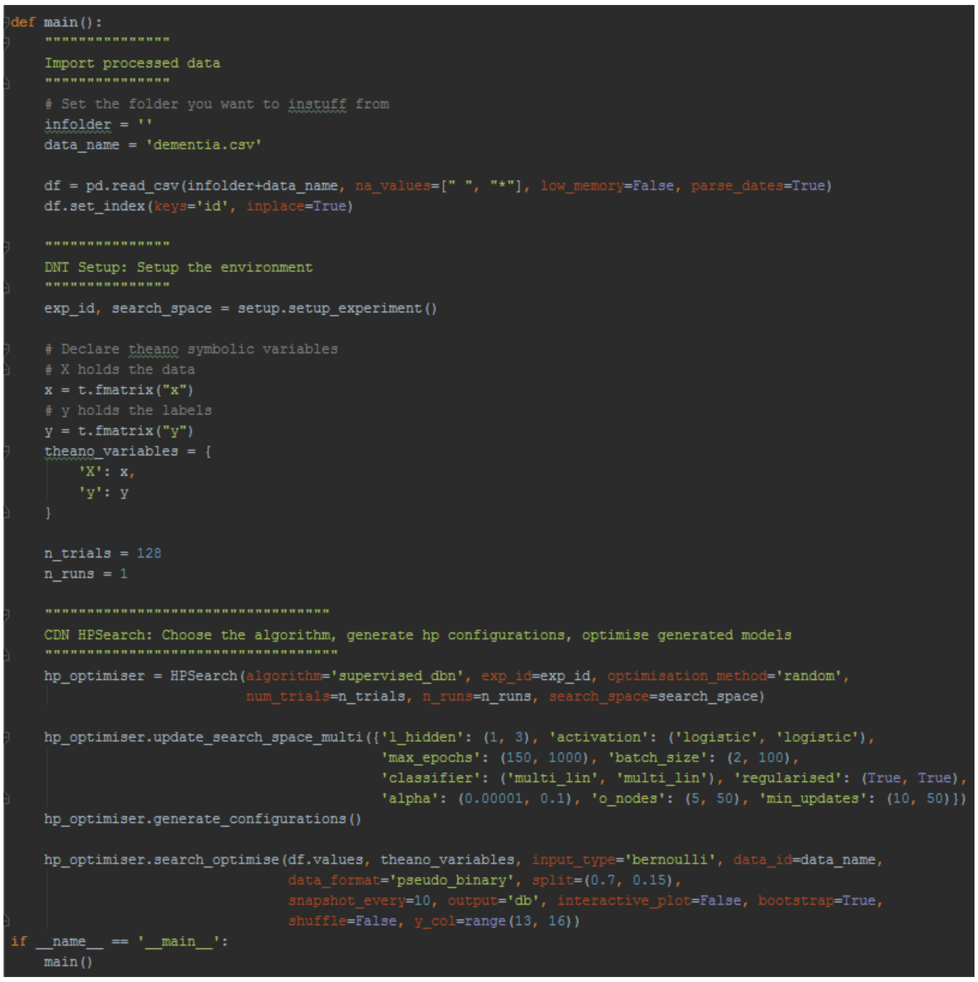
Snapshot Counts. Table E.2 shows the count of snapshots stored over all experiments. For deep experiments we stored both the result of the pre-training and fine-tuning procedures in *TrialRuns*. We did not store *snapshots* at every validation, to take advantage the benefits of results storage and decreased querying times.

Table E.2: Snapshot Counts

Experiment ID	Trial-Runs		Updates	Total
	Unsuper.	Super.		
gaa_rnn_05-04-16_11:48:02	n/a	180	40,089	40,269
maas_dbn_03-01-17_12:32:09	128	128	6,585	6,841
bach_srbm_31-12-16_13:53:53	n/a	300	8,516	8817

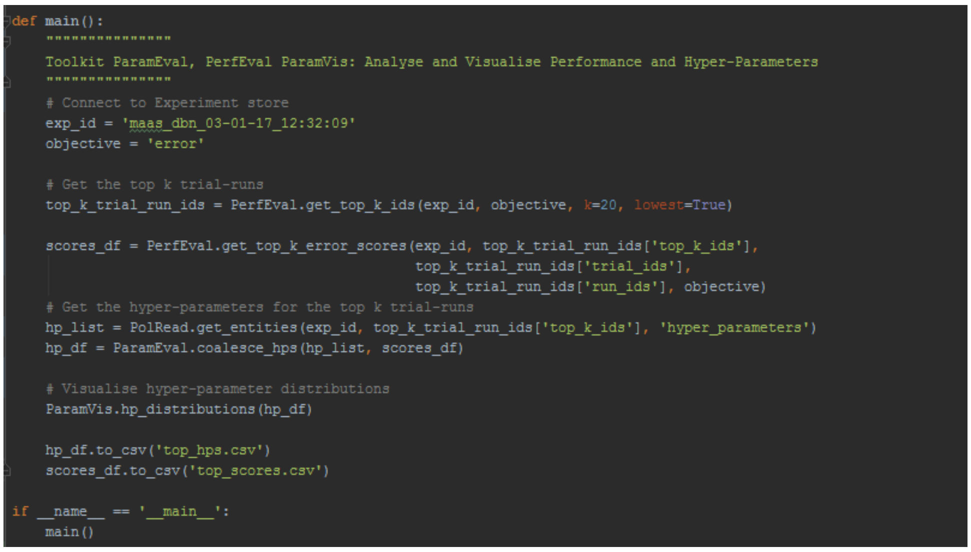
Appendix F

System Screenshots



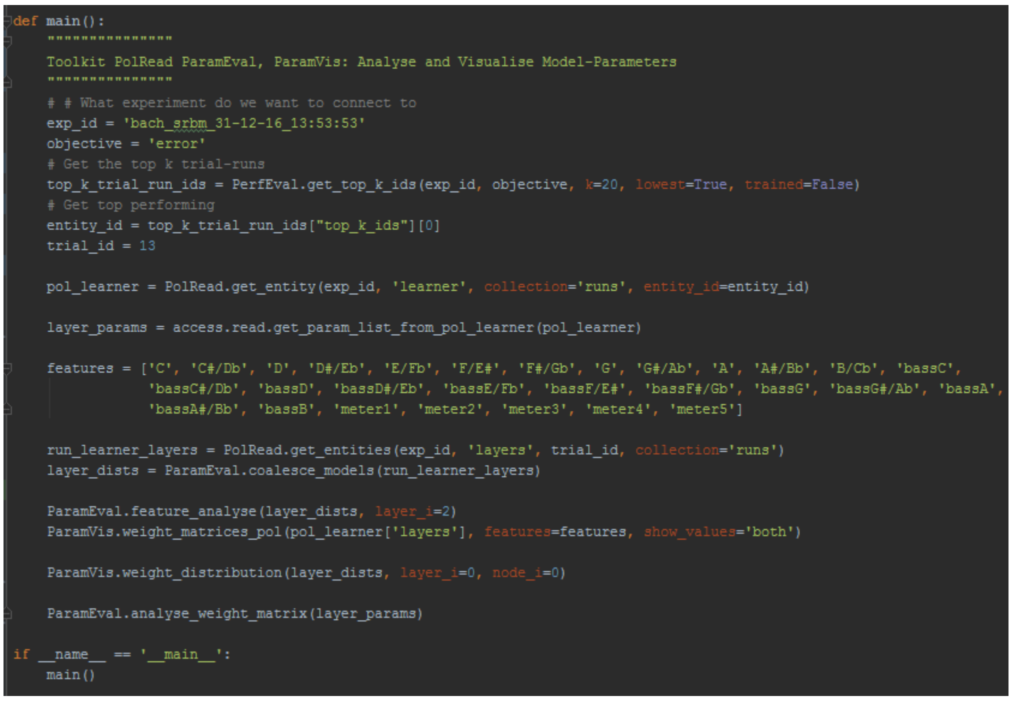
```python
def main():
    """""""""""""""
    Import processed data
    """""""""""""""
    # Set the folder you want to instuff from
    infolder = ''
    data_name = 'dementia.csv'

    df = pd.read_csv(infolder+data_name, na_values=[" ", "*"], low_memory=False, parse_dates=True)
    df.set_index(keys='id', inplace=True)

    """""""""""""""
    DNT Setup: Setup the environment
    """""""""""""""
    exp_id, search_space = setup.setup_experiment()

    # Declare theano symbolic variables
    # X holds the data
    x = t.fmatrix("x")
    # y holds the labels
    y = t.fmatrix("y")
    theano_variables = {
        'X': x,
        'y': y
    }

    n_trials = 128
    n_runs = 1

    """"""""""""""""""""""""""""""""""
    CDN HPSearch: Choose the algorithm, generate hp configurations, optimise generated models
    """""""""""""""""""""""""""""""""""
    hp_optimiser = HPSearch(algorithm='supervised_dbn', exp_id=exp_id, optimisation_method='random',
                            num_trials=n_trials, n_runs=n_runs, search_space=search_space)

    hp_optimiser.update_search_space_multi({'l_hidden': (1, 3), 'activation': ('logistic', 'logistic'),
                                            'max_epochs': (150, 1000), 'batch_size': (2, 100),
                                            'classifier': ('multi_lin', 'multi_lin'), 'regularised': (True, True),
                                            'alpha': (0.00001, 0.1), 'o_nodes': (5, 50), 'min_updates': (10, 50)})
    hp_optimiser.generate_configurations()

    hp_optimiser.search_optimise(df.values, theano_variables, input_type='bernoulli', data_id=data_name,
                                 data_format='pseudo_binary', split=(0.7, 0.15),
                                 snapshot_every=10, output='db', interactive_plot=False, bootstrap=True,
                                 shuffle=False, y_col=range(13, 16))
if __name__ == '__main__':
    main()
```

Figure F.1: Example of CDN Configuration



```python
def main():
    """""""""""""""
    Toolkit ParamEval, PerfEval ParamVis: Analyse and Visualise Performance and Hyper-Parameters
    """""""""""""""
    # Connect to Experiment store
    exp_id = 'maas_dbn_03-01-17_12:32:09'
    objective = 'error'

    # Get the top k trial-runs
    top_k_trial_run_ids = PerfEval.get_top_k_ids(exp_id, objective, k=20, lowest=True)

    scores_df = PerfEval.get_top_k_error_scores(exp_id, top_k_trial_run_ids['top_k_ids'],
                                                top_k_trial_run_ids['trial_ids'],
                                                top_k_trial_run_ids['run_ids'], objective)
    # Get the hyper-parameters for the top k trial-runs
    hp_list = PolRead.get_entities(exp_id, top_k_trial_run_ids['top_k_ids'], 'hyper_parameters')
    hp_df = ParamEval.coalesce_hps(hp_list, scores_df)

    # Visualise hyper-parameter distributions
    ParamVis.hp_distributions(hp_df)

    hp_df.to_csv('top_hps.csv')
    scores_df.to_csv('top_scores.csv')

if __name__ == '__main__':
    main()
```

Figure F.2: Example of Toolkit Hyper-Parameter Evaluation



```python
def main():
    """""""""""""""
    Toolkit PolRead ParamEval, ParamVis: Analyse and Visualise Model-Parameters
    """""""""""""""
    # # What experiment do we want to connect to
    exp_id = 'bach_srbm_31-12-16_13:53:53'
    objective = 'error'
    # Get the top k trial-runs
    top_k_trial_run_ids = PerfEval.get_top_k_ids(exp_id, objective, k=20, lowest=True, trained=False)
    # Get top performing
    entity_id = top_k_trial_run_ids["top_k_ids"][0]
    trial_id = 13

    pol_learner = PolRead.get_entity(exp_id, 'learner', collection='runs', entity_id=entity_id)

    layer_params = access.read.get_param_list_from_pol_learner(pol_learner)

    features = ['C', 'C#/Db', 'D', 'D#/Eb', 'E/Fb', 'F/E#', 'F#/Gb', 'G', 'G#/Ab', 'A', 'A#/Bb', 'B/Cb', 'bassC',
                'bassC#/Db', 'bassD', 'bassD#/Eb', 'bassE/Fb', 'bassF/E#', 'bassF#/Gb', 'bassG', 'bassG#/Ab', 'bassA',
                'bassA#/Bb', 'bassB', 'meter1', 'meter2', 'meter3', 'meter4', 'meter5']

    run_learner_layers = PolRead.get_entities(exp_id, 'layers', trial_id, collection='runs')
    layer_dists = ParamEval.coalesce_models(run_learner_layers)

    ParamEval.feature_analyse(layer_dists, layer_i=2)
    ParamVis.weight_matrices_pol(pol_learner['layers'], features=features, show_values='both')

    ParamVis.weight_distribution(layer_dists, layer_i=0, node_i=0)

    ParamEval.analyse_weight_matrix(layer_params)

if __name__ == '__main__':
    main()
```

Figure F.3: Example of Toolkit Model Parameter Evaluation