Deep Learning for Tabular Data: An Exploratory Study

by

Jan André Marais

Thesis presented in partial fulfilment of the requirements for the degree of Master of Commerce (Mathematical Statistics) in the Faculty of Economic and Management Sciences at Stellenbosch University

Supervisor: Dr. S. Bierman

December 2018

The financial assistance of the National Research Foundation (NRF) towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at, are those of the author and are not necessarily to be attributed to the NRF.

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: .....

Abstract

Deep Learning for Tabular Data: An Exploratory Study

J. A. Marais

Thesis: MCom (Mathematical Statistics)

December 2018

English abstract.

Uittreksel

Diepleer Tegnieke vir Gestruktrueerde Data: 'n Verkennende Studie

("Deep Learning for Tabular Data: An Exploratory Study")

J. A. Marais

Tesis: MCom (Wiskundige Statistiek)

Desember 2018

Afrikaans abstract

Acknowledgements

I would like to express my sincere gratitude to the following people and organisations:

- Dr. S. Bierman for her guidance and patience as a supervisor and for allowing me freedom in the choice of research directions.
- The National Research Foundation (NRF) for financial support. *Opinions expressed and conclusions arrived at, are those of the author and are not necessarily to be attributed to the NRF.*
- The UCI Machine Learning Repository (Dheeru and Karra Taniskidou, 2017) for hosting a platform to share datasets.
- My parents and close family for believing in me and motivating me.
- Most importantly, my partner, for her never-ending support, love and understanding.

Contents

Declaration i

Abstract ii

Uittreksel iii

Acknowledgements iv

Contents v

List of Figures ix

List of Tables xii

List of Abbreviations and/or Acronyms xiii

Notation xiv

1 Introduction 1

1.1 Deep Learning 1

1.2 Tabular Data 3

1.3 Challenges of Deep Learning for Tabular Data 5

1.4 Overview of Statistical Learning Theory 7

1.5 Outline 13

2 Neural Networks 15

2.1 Introduction 15

2.2 The Structure of a Neural Network 16

2.2.1 Neurons and Layers 16

2.2.2 Activation Functions 19

2.2.3 Size of the Network . . . 21
2.3 Training a Neural Network . . . 22
2.3.1 Weight Initialisation . . . 23
2.3.2 Optimisation . . . 23
2.3.3 Optimisation Example . . . 25
2.3.4 Backpropagation . . . 26
2.4 Basic Regularisation . . . 29
2.5 Adaptive Learning Rates . . . 31
2.6 Representation Learning . . . 31

3 Deep Learning 36
3.1 Introduction . . . 36
3.2 Autoencoders . . . 37
3.3 Transfer Learning . . . 40
3.4 More Regularisation . . . 41
3.4.1 Dropout . . . 41
3.4.2 Data Augmentation . . . 44
3.5 Modern Architectures . . . 46
3.5.1 Normalisation . . . 46
3.5.2 Skip Connections . . . 47
3.5.3 Embeddings . . . 49
3.5.4 Attention . . . 50
3.6 Super-Convergence . . . 52
3.7 Model Interpretation . . . 56
3.7.1 Neural Network Specific . . . 57
3.7.2 Model Agnostic . . . 58

4 Deep Learning for Tabular Data 60
4.1 Introduction . . . 60
4.2 Input Representation . . . 61
4.2.1 Numerical Features . . . 62
4.2.2 Categorical Features . . . 63
4.2.3 Combining Features . . . 67
4.3 Learning Feature Interactions . . . 67
4.3.1 Attention . . . 70
4.3.2 Self-Normalising Neural Networks . . . 71
4.4 Sample Efficiency . . . 72

4.4.1 Data Augmentation . . . 72
4.4.2 Unsupervised Pretraining . . . 75
4.4.3 Regularisation . . . 76
4.5 Interpretation . . . 77
4.6 Hyperparameter Selection . . . 81

5 Experiments **84**
5.1 Introduction . . . 84
5.2 Datasets . . . 85
5.3 General Methodology . . . 86
5.3.1 Loss Function and Evaluation Metric . . . 86
5.3.2 Cross-validation . . . 86
5.3.3 Preprocessing . . . 88
5.3.4 General Hyperparameters . . . 88
5.4 Input Representation . . . 89
5.4.1 Embeddings Sizes . . . 89
5.5 Feature Interactions . . . 90
5.5.1 Attention . . . 90
5.5.2 SeLU . . . 91
5.5.3 Skip-Connections . . . 91
5.6 Sample Efficiency . . . 92
5.6.1 Data Augmentation . . . 93
5.6.2 Unsupervised Pretraining . . . 94
5.7 General Thoughts . . . 96

6 Conclusion **97**
6.1 Summary . . . 97
6.2 Limitations . . . 99
6.3 Promising Future Directions . . . 100

Appendices **102**

A Datasets **103**
A.1 Adult Dataset . . . 103

B Hyperparameter Search **107**
B.1 Width and Depth of Network . . . 107
B.2 Dropout . . . 107

C Software and Code **109**
C.1 Development Environment 109
C.2 Code and Reproducibility 109

Bibliography **110**

List of Figures

1.1 Linear model on simple binary classification dataset. 12

2.1 Comparison of a biological (a) and an artificial (b) neuron. 17
2.2 A simple neural network accepting p-sized inputs, with one hidden layer consisting of two neurons. 18
2.3 Activations. 20
(a) Function 20
(b) Local Derivative 20
2.4 Plots of the gradient descent example. (a) The training data points in input space. The shades in the background represent the class division in input space, with the decision boundary determined by linear least squares estimation. The dashed lines represent the gradient descent decision boundaries at different iterations. (b) The loss function at each iteration. 27
2.5 Simple dataset with two linearly inseparable classes. 33
2.6 Decision boundary of a single-layer neural network. 34
2.7 Decision boundary of a two-layer neural network. 34
2.8 Hidden representation of a two-layer neural network. 35

3.1 A simple single hidden layer autoencoder with four-dimensional inputs and with two neurons in the hidden layer. 39
3.2 Visualising the first layer convolutional filters leared by a neural network in a large image dataset. 42
3.3 The effect that dropout has on connections between neurons. 44
3.4 An example of data augmentation for images. 45
3.5 Diagram conceptualising a skip connection. 48
3.6 Learned word embeddings in a two-dimensional space. 49
3.7 Attention applied to image captioning. 51

3.8 Attention applied to machine translation. 51
3.9 An example output of a learning rate range test. 55
3.10 Reduced training iterations and improved performance facilitated by the super-convergence principle. 55

4.1 The effect of normalisation on continuous variables. 63
(a) Original 63
(b) Gaussian Norm 63
(c) Power Norm 63
4.2 PCA of the Education entity embedding weight matrix. 66
4.3 Combined representation of continuous and categorical features. . . 68
4.4 Illustration of points created by mixup augmentation. 74
4.5 The attention weights visualised for a singe observation in the dataset. 78
4.6 Permutation importance plot of a neural network trained on the Adult dataset. 79
4.7 Feature importance obtained from a boosted trees model trained on the neural network predictions. 80
4.8 Constant learning rate vs the 1Cycle schedule. 82
4.9 A learning rate range test with different weight decays. 83
4.10 A full training run with different weight decays. 83

5.1 5-Fold Cross-validation dataset split schematic. 87
5.2 Effect of the embedding size if all categorical features are mapped to the same number of dimensions. 89
5.3 Effect of variable sizes on the performance of the model. 90
5.4 Comparing the attention mechanism with a simple MLP. 91
5.5 The average performance of ReLU and SeLU activation functions for shallow and deep networks over every epoch. 92
5.6 The average performance of ReLU and SeLU activation functions for shallow and deep networks. 92
5.7 Average performance at each epoch for shallow and deep neural networks, with and without skip connections. 93
5.8 Overall performance of the skip-connections used in a shallow and deep neural network. 93
5.9 Effect of the number of training samples on the performance of neural networks. 94

5.10 Average performance of models with various mixup and weight decay parameters. 94
5.11 Performance per epoch for models with different weight decays and mixup ratios. 95
5.12 The effect of pretraining on the classifier's performance. 96

A.1 Histograms for each of the continuous features in the Adult dataset. 105
A.2 Bar plot for each of the categorical features in the Adult dataset. . 106

B.1 Effect of the layer width and network depth on the performance on the Adult dataset. 108
B.2 The effect of dropout on wide and narrow neural networks. 108

List of Tables

1.1 Preview of the Adult dataset. 4

4.1 Swap Noise Example. 73

List of Abbreviations and/or Acronyms

ANN	Artificial Neural Network
CNN	Convolutional Neural Network
CTR	Click-through Rate
CV	Computer Vision
DL	Deep Learning
EHR	Electronic Health Records
GAN	Generative Adversarial Network
kNN	k-Nearest Neighbour
mAP	Mean Average Precision
ML	Machine Learning
MLP	Multi-layer Perceptron
NLP	Natural Language Processing
NN	Neural Network
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
SotA	State-of-the-Art
VAE	Variational Autoencoder

Notation

N	number of observations in a dataset
p	input dimension or the number of features for an observation
K	number of labels in a dataset
$\boldsymbol{x}$	p-dimensional input vector $(x_1, x_2, \ldots, x_p)^\intercal$
λ	label
$\mathcal{L}$	complete set of labels in a dataset $\mathcal{L} = \{\lambda_1, \lambda_2, \ldots, \lambda_K\}$
Y	labelset associated with $\boldsymbol{x}$, $Y \subseteq \mathcal{L}$
$\hat{Y}$	predicted labelset associated with $\boldsymbol{x}$, $\hat{Y} \subseteq \mathcal{L}$
$\boldsymbol{y}$	K-dimensional label indicator vector, $(y_1, y_2, \ldots, y_K)^\intercal$, associated with observation $\boldsymbol{x}$
$(\boldsymbol{x}_i, Y_i)_{i=1}^N$	multi-label dataset with N observations
D	dataset
$h(\cdot)$	multi-label classifier $h : \mathbb{R}^p \to 2^{\mathcal{L}}$, where $h(\boldsymbol{x})$ returns the set of labels for $\boldsymbol{x}$
θ	set of parameters for $h(\cdot)$
$\hat{\theta}$	set of parameters for $h(\cdot)$ that optimise the loss function
$L(\cdot, \cdot)$	loss function between predicted and true labels
$f(\cdot)$	label prediction module, $f : \mathbb{R}^p \to \mathbb{R}^K$
$t(\cdot)$	thresholding function, $t : \mathbb{R}^K \to \{0, 1\}^K$
$\mathcal{N}(\boldsymbol{x})$	points in the input space neighbourhood of $\boldsymbol{x}$

Chapter 1

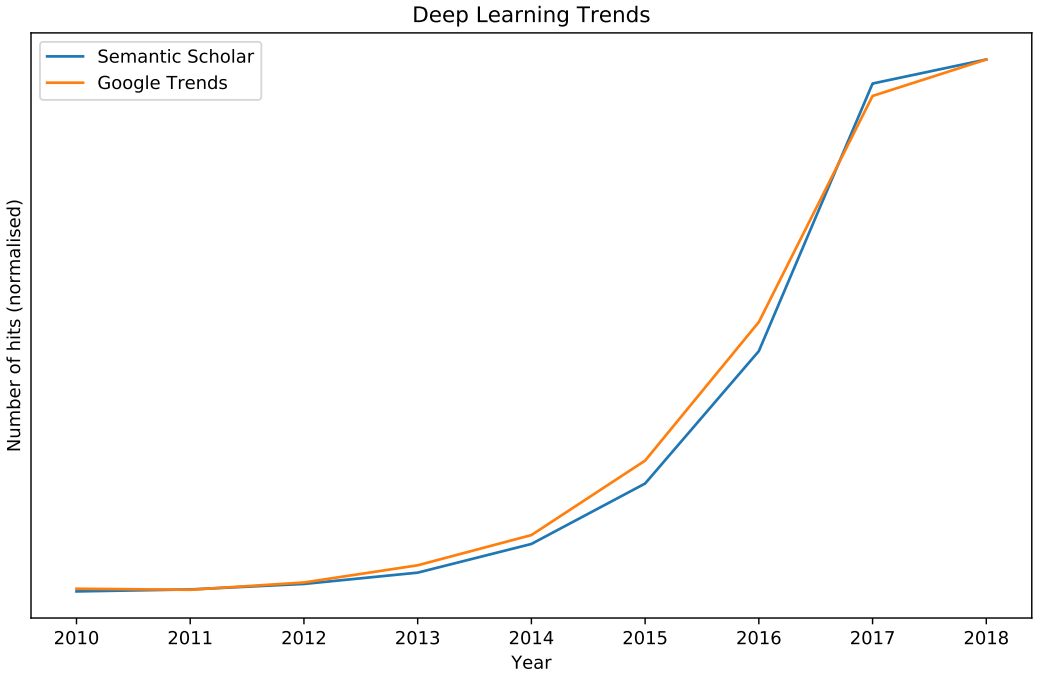
Introduction

1.1 Deep Learning

This thesis is concerned with the study of *deep learning* approaches to solve *machine learning* (ML) tasks. More specifically, our interest lies in machine learning tasks that may be solved using tabular data inputs. The deep learning field is an extention of the class of machine learning algorithms called *Artificial Neural Networks* (NNs). Whereas until relatively recently, the neural network field was not an over-active research field, rapid development in computing power and the growing abundance of data lead to advances in neural network optimisation and architecture. These advances constitutes the deep learning field as we know it today (Lecun *et al.*, 2015).

Currently, deep learning is receiving a remarkable amount of attention, both in research and in practice (see **??**). Much of the deep learning hype stems from the tremendous value neural networks have shown in application areas such as *computer vision* (Hu *et al.*, 2017), audio processing (Battenberg *et al.*, 2017), and *natural language processing* (NLP) (Devlin *et al.*, 2018). In these application areas, deep learning methods have reached a maturity level sufficient to be able to run these systems in a production or commercial environment. Examples of the application of deep learning in commercial applications include voice assistants like Amazon Alexa (Sarikaya, 2017), face recognition with Apple iPhones [1], and language translation with Google (Wu *et al.*, 2016).

[1] https://www.apple.com/business/site/docs/FaceID_Security_Guide.pdf



One of the most attractive attributes of deep learning is its ability to model almost any input-output relationship. This has lead to the use of deep learning in a very wide array of applications.

For example, deep learning has been used to generate art (Gatys *et al.*, 2015) and music (Mogren, 2016), to control various modules in autonomous cars (Fridman *et al.*, 2017), to play video games (Mnih *et al.*, 2013), to recommend movies (Covington *et al.*, 2016), to improve the quality of images (Shi *et al.*, 2016), and to beat the world's best Go player (Silver *et al.*, 2017).

A common characteristic of all of the above deep learning applications is that the data used to construct them contain the same type of values or measurements. That is, in computer vision the data represent pixel values, whereas in NLP and in audio processing the data represent words and sound waves. This is not a criterion for deep learning algorithms to be successful, but may be viewed as a driver for their success in these application domains. It is simpler to model data consisting of the same type of measurements, since each input feature may be treated the same. Furthermore in the above deep learning applications, it is found that in each of these domains, universal patterns exist. This allows for knowledge to be transferred between tasks belonging to the same domain. The knowledge to be transferred is both the knowledge aquired by humans, and the knowledge acquired by a deep learning model. For example, in computer vision, advances in classifying pictures of pets will most likely also

facilitate improved identification of tumors in X-rays. That is, patterns learned by a deep learning model when attempting one task, may also be useful in a different, but related task. This phenomenon constitutes a second reason for the successful application of deep learning methods, and is studied in the field of *transfer learning*.

A data domain in which deep learning have not yet been very successful, is that of tabular data. A *tabular dataset* can be represented by a two-dimensional table, where each of the rows of the table corresponds to one observation and where each column denotes an individual meaningful feature. We further explain the use of tabular data in Section 1.2 below.

Some research have recently been done on the use of deep learning models for tabular data. See for example Shavitt and Segal (2018) and Song *et al.* (2018). However, state-of-the-Art (SotA) results are reported only rarely (de Brébisson *et al.*, 2015), and in the Kaggle competition found at the following website[2]). Therefore it can be said that the area is nowhere near as mature or receiving as much attention as is the case with deep learning for computer vision or for NLP. In a comprehensive study in the paper by Fernández-Delgado *et al.* (2014), it was found that ML tasks that make use of tabular data are typically more effectively solved using tree-based methods. This is also evident when one considers the winning solutions of relevant Kaggle competitions[3]. A possible explanation for the superior performance of tree-based metods, is the heterogeneity of tabular data (Shavitt and Segal, 2018), which forms part of the discussion in the next section.

1.2 Tabular Data

In this section we make use of the so-called Adult[4] dataset in order to discuss the use of tabular data. The reader may refer to Table 1.1 for an excerpt of this dataset. Note that the data were collected during an American census where the aim was to predict whether or not an indivdual earns more than $50,000 a year.

Table 1.1 represents a typical tabular dataset, where the columns contain measurements on different features. Therefore different columns may contain

[2]https://www.kaggle.com/c/porto-seguro-safe-driver-prediction/discussion/44629
[3]https://www.kaggle.com
[4]http://archive.ics.uci.edu/ml/datasets/Adult

	age	occupation	education	race	sex	>=50k
1	49		Assoc-acdm	White	Female	1
2	44	Exec-managerial	Masters	White	Male	1
3	38		HS-grad	Black	Female	0
4	38	Prof-specialty	Prof-school	Asian-Pac-Islander	Male	1
5	42	Other-service	7th-8th	Black	Female	0
6	20	Handlers-cleaners	HS-grad	White	Male	0

Table 1.1: Preview of the Adult dataset.

different data types: some columns may consist of continuous measurements, whereas other columns may contain discrete or categorical measurements. Furthermore, in tabular data, the rows and columns occur in no particular order. This of course stands in contrast to image or text data.

Many important ML applications make use of tabular data. Some of these applications are listed below:

- Various tasks that make use of Electronic Health Records. These include the prediction of in-hospital mortality rates, and of prolonged length of stay (Rajkomar *et al.*, 2018);
- Recommender systems for items like videos (Covington *et al.*, 2016) or property listings (Haldar *et al.*, 2018);
- Click-through rate (CTR) prediction in web applications, *i.e.* predicting which item a user will click on next (Song *et al.*, 2018);
- Predicting which clients are at risk of defaulting on their accounts[5];
- Predicting store sales (Guo and Berkhahn, 2016); and
- Drug discovery (Klambauer *et al.*, 2017).

Tabular datasets take on various shapes and sizes: the number of rows may range from hundreds to millions, and the number of columns also has no limits. Other complicating characteristics of tabular datasets include: - That it is not unusual for tabular datasets to be noisy; - That a proportion of the observations may have missing features and/or incorrect values; and - That continuous measurements may be based upon vastly different scales, some even containing outliers, whereas categorical features may have high cardinality which in turn leads to sparse data.

During the construction of models for tabular datasets, the most important step in terms seeking improvements in model performance, is pre-processing

[5]https://www.kaggle.com/c/loan-default-prediction

and manipulation of the input features (Rajkomar *et al.*, 2018). This includes data merging, customising, filtering and cleaning. In a process called feature engineering, one strives to create new features from the original features based on some domain knowledge. The idea is that such engineered features enables a model to learn interactions between features, thereby facilitating more accurate prediction. Feature engineering is an extremely laborious process with no clear recipe to follow and therefore typically cannot succesfully be implemented without some domain expertise.

Ensemble methods based upon trees are currently viewed as the most effective machine learning models for tabular datasets. As mentioned above, a possible reason for this may be their robustness to different feature scales and data types, linked with their ability to effectively model interactions among features with different data types.

Indeed, in the context of tabular data, classical neural network approaches are no match for tree ensembles. Although the deep learning field has advanced and matured a lot in recent years, it is not yet clear how to leverage these modern techniques to effectively build and train deep neural networks on tabular datasets. In this thesis we explore ways of doing so. By reviewing the most recent literature on the topic, and through empirical work, we aim to summarise best practices when using deep learning for tabular data.

1.3 Challenges of Deep Learning for Tabular Data

Some of the challenges of deep learning for tabular data have been alluded to in earlier sections of this chapter. These will form the framework for our literature review later on. Therefore some of the important questions to ask when applying deep learning for tabular data (which relates to these challenges), are summarised below.

- **How should input features be represented numerically?** We have mentioned that tabular data consist of mixed feature data types, *i.e.* a combination of categorical and continuous features. The question here relates to how these heterogeneous features should be processed and presented to the model during training.

- **How can we exploit feature interactions?** Once we have found the optimal feature representation for all feature data types, we will need a way to effectively learn the interactions among them, and also a way to learn how they relate to the target. This is a crucial step towards the effective application of deep learning models to tabular data.
- **How can we be more sample efficient?** Tabular datasets are typically smaller than datasets used in computer vision and in NLP. Moreover, no general large dataset with universal properties exists to be used by a model to learn from (as is the case in for example in transfer learning for image classification). Thus a key challenge is to facilitate learning from less data.
- **How do we interpret model decisions?** The use of deep learning is often restricted by its perceived lack of interpretability. Therefore we need ways of explaining the model output in order for it to be useful in a wider array of applications.

Clearly there are several considerations when it comes to using deep learning for tabular data. The main objective of this thesis is to find the best ways of answering the above questions. Towards this objective, the study should lead to a thorough understanding of the *status quo* of the field, and of the necessary factors in order to ensure deep learning to be as effective in other data domains as it currently is in fields such as computer vision and NLP.

The study is divided in two parts. We start by first providing an overview of the relevant literature. Subsequently, we make use of experimental work in order to compare various deep learning algorithms (and possible improvements) on relevant datasets. Here an important aim will be to ensure our experiments to be *rigorous*. The importance of rigorous research has relatively recently again been emphasised during an NIPS talk[6], during which researchers in the deep learning field have been criticised for the growing gap between the understanding of its techniques, and practical successes. Currently much more emphasis is placed on the latter. The speakers urged the deep learning community to be more rigorous in their experiments where, for them, the most important part of rigor is better empiricism, not more mathematical theories. Better empiricism in classification may include, for example, practices such as using cross-validation to estimate the generalisation ability of a model,

[6]Talk given at NIPS2017 - https://www.youtube.com/watch?v=Qi1Yry33TQE

and reporting standard errors. Empirical studies should involve more than simply attempting to beat the benchmark. For example, where possible, they should also involve simple experiments that facilitate understanding why some algorithms are successful, while others are not.

In addition, we want the empirical work in this study to be as reproducible as possible. This aspect is often overlooked. However it is a crucial aspect, ensuring transparent and accountable reporting of results. Reproducibility add to the value of research, since without it, researchers are not able to build on each other's work. Hence all code, data and necessary documentation in order to reproduce the experiments done in this study are available [7].

Having stated the objectives of this study, we now turn to a discussion of the fundamental concepts of Statistical Learning Theory. This is followed by a more detailed overview of the thesis.

1.4 Overview of Statistical Learning Theory

Machine- or statistical learning algorithms (here used interchangeably) are used to perform certain tasks that are too difficult to solve with fixed rule-based programs. Hence statistical learning algorithms are able to use data in order to learn how to perform difficult tasks. For an algorithm to learn from data means that it can improve its ability to perform an assigned *task* with respect to some *performance measure*, by processing *data*. In this section we discuss some of the important types of tasks, data and performance measures in the statistical learning field.

A learning task describes the way in which an algorithm should process an observation. An observation is a collection of features that have been measured, corresponding to some object or event that we want the system to process, for example an image. We will represent an observation by a vector $\boldsymbol{x} \in \mathbb{R}^p$, where each element x_j of the vector is an observed value of the j-th feature, $j = 1, \ldots, p$. For example, the features of an image are usually the color intensity values of the pixels in the image.

Many kinds of tasks can be solved using statistical learning. One of the most common learning tasks is that of *classification*, where it is expected of an algorithm to determine which of K categories an input belongs to. In order to complete the classification task, the learning algorithm is usually asked to

[7] Shared publicly at https://github.com/jandremarais/tabularLearner

produce a function $f : \mathbb{R}^p \to \{1, \ldots, K\}$. When $y = f(\boldsymbol{x})$, the model assigns an input described by the vector $\boldsymbol{x}$ to a category identified by the numeric code y, called the *output* or *response*. In other variants of the classification task, f may output a probability distribution over the possible classes.

Regression is another main learning task and requires the algorithm to predict a continuous value given some input. This task requires a function $f : \mathbb{R}^p \to \mathbb{R}$, where the only difference between regression and classification is the format of the output.

Learning algorithms learns such tasks by observing a relevant set of data points. A dataset containing N observations of p features is commonly denoted by a data matrix $X : N \times p$, where each row represents a different observation and where each column corresponds to a different feature of the observations, *i.e.*

$$X = \begin{bmatrix} x_{11} & x_{12} & \ldots & x_{1p} \\ x_{21} & x_{22} & \ldots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N1} & x_{N2} & \ldots & x_{Np} \end{bmatrix}.$$

Often the dataset includes annotations for each observation in the form of a label (*i.e.* in classification) or in the form of a target value (*i.e.* in regression). These N annotations are represented by the vector $\boldsymbol{y}$, where the element y_i is associated with the i-th row of X. Therefore the response vector may be denoted by

$$\boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}.$$

Note that in the case of multiple labels or targets, a matrix representation $Y : N \times K$ is required.

Statistical learning algorithms can be divided into two main categories, *viz. supervised* and *unsupervised* algorithms. This categorisation is determined by the presence (or absence) of annotations in the dataset to be analysed. Unsupervised learning algorithms learn from data consisting only of features, X, and are used to find useful properties and structure in the dataset (see Hastie *et al.*, 2009, Ch. 14). On the other hand, supervised learning algorithms

learn from datasets which consist of both features and annotations, (X, Y), with the aim to model the relationship between them. Therefore, both classification and regression are considered to be supervised learning tasks.

In order to evaluate the ability of a learning algorithm to perform its assigned task, we have to construct a quantitative performance measure. For example, in a classification task we are usually interested in the accuracy of the algorithm, *i.e.* the percentage of times that the algorithm assigns the correct classification. We are mostly interested in how well the learning algorithm performs on data that it has not seen before, since this demonstrates how well it will perform in real-world situations. Thus we typically evaluate the algorithm on a *test set* of data points. This dataset is independent of the *training set* of data points that was used during the learning process.

For a more concrete example of supervised learning, and keeping in mind that the linear model is one of the main building blocks of neural networks, consider the learning task underlying *linear regression*. The objective here is to construct a system which takes a vector $\boldsymbol{x} \in \mathbb{R}^p$ as input and which predicts the value of a scalar $y \in \mathbb{R}$ as response. In the case of linear regression, we assume the output to be a linear function of the input. Let $\hat{y}$ be the predicted response. We define the output to be

$$\hat{y} = \hat{\boldsymbol{w}}^T \boldsymbol{x},$$

where $\hat{\boldsymbol{w}} = [w_0, w_1, \ldots, w_p]$ denotes a vector of parameters and where $\boldsymbol{x} = [1, x_1, x_2, \ldots, x_p]$. Note that an intercept is included in the model (also known as a *bias* in machine learning). The parameters are values that control the behaviour of the system. We can think of them as a set of *weights* that determine how each feature affects the prediction. Hence the learning task can be defined as predicting y from $\boldsymbol{x}$ through $\hat{y} = \hat{\boldsymbol{w}}^T \boldsymbol{x}$.

We of course need to define a performance measure to evaluate the linear predictions. For a set of observations, an evaluation metric tells us how (dis)similar the predicted output is to the actual response values. A very common measure of performance in regression is the *mean squared error* (MSE), given by

$$MSE = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2.$$

The process of learning from data (or fitting a model to a dataset) can be reduced to the following optimisation problem: find the set of weights, $\hat{\boldsymbol{w}}$, which produces a $\hat{\boldsymbol{y}}$ that minimises the MSE. Of course this problem has a closed form solution and can quite trivially be found by means of *ordinary least squares* (OLS) (see Hastie *et al.*, 2009, p. 12). However, we have mentioned that we are more interested in the algorithm's performance evaluated on a test set. Unfortunately the least squares solution does not guarantee the solution to be optimal in terms of the MSE on a test set, rendering statistical learning to be much more than a pure optimisation problem.

The ability of a model to perform well on previously unobserved inputs is referred to as its *generalisation* ability. To be able to fit a model that generalises well to new unseen data cases is the key challenge of statistical learning. One way of improving the generalisation ability of a linear regression model is to modify the optimisation criterion J, to include a *weight decay* (or *regularisation*) term. That is, we want to minimise

$$J(\boldsymbol{w}) = MSE_{\text{train}} + \lambda \boldsymbol{w}^T \boldsymbol{w},$$

where $J(\boldsymbol{w})$ now expresses preference for smaller weights. The parameter λ is non-negative and needs to be specified ahead of time. It controls the strength of the preference by determining how much influence the penalty term, $\boldsymbol{w}^T\boldsymbol{w}$, has on the optimisation criterion. If $\lambda = 0$, no preference is imposed, and the solution is equivalent to the OLS solution. Larger values of λ force the weights to decrease, and thus referred to as a so-called *shrinkage* method ((*cf.* for example Hastie *et al.*, 2009, pp. 61-79) and (Goodfellow *et al.*, 2016).

We may further generalise linear regression to the classification scenario. First, it is important to note the different types of classification schemes. Consider $\mathcal{G}$, the discrete set of values which may be assumed by G, where G is used to denote a categorical output variable (instead of Y). Let $|\mathcal{G}| = K$ denote the number of discrete categories in the set $\mathcal{G}$. The simplest form of classification is known as binary classification and refers to scenarios where the input is associated with only one of two possible classes, *i.e.* $K = 2$. When $K > 2$, the task is known as multiclass classification. In contrast, in *multi-label* classification an input may be associated with multiple classes (out of K available classes), where the number of classes that each observation belongs to, is unknown. In the remainder of this section, we introduce the two single label classification setups, *viz.* binary and multiclass classification.

In multiclass classification, given the input values $\boldsymbol{X}$, we would like to accurately predict the output, G, where our prediction is denoted by $\hat{G}$. One approach would be to represent G by an indicator vector $\boldsymbol{Y}_G : K \times 1$, with all elements zero except in the G-th position, where it is assigned a 1. That is, $Y_k = 1$ for $k = G$ and $Y_k = 0$ for $k \neq G$, $k = 1, 2, ..., K$. We may then treat each of the elements in $\boldsymbol{Y}_G$ as quantitative outputs, and predict values for them, denoted by $\hat{\boldsymbol{Y}} = [\hat{Y}_1, \ldots, \hat{Y}_K]$. The class with the highest predicted value will then be the final categorical prediction of the classifer, *i.e.* $\hat{G} = \arg\max_{k \in \{1,\ldots,K\}} \hat{Y}_k$.

Within the above framework we therefore seek a function of the inputs which is able to produce accurate predictions of the class scores, *i.e.*

$$\hat{Y}_k = \hat{f}_k(\boldsymbol{X}),$$

for $k = 1, \ldots, K$. Here $\hat{f}_k$ is an estimate of the true function, f_k, which is meant to capture the relationship between the inputs and output of class k. As with the linear regression case described above, we may use a linear model $\hat{f}_k(\boldsymbol{X}) = \hat{\boldsymbol{w}}_k^T \boldsymbol{X}$ to approximate the true function. The linear model for classification partitions the input space into a collection of regions labelled according to the predicted classification, where regions are created by linear *decision boundaries* (see Figure 1.1 for an illustration). The decision boundary between classes k and l is the set of points for which $\hat{f}_k(\boldsymbol{x}) = \hat{f}_l(\boldsymbol{x})$. These set of points form an affine set or hyperplane in the input space.

After the weights are estimated from the data, an observation represented by $\boldsymbol{x}$ (including the unit element) may be classified as follows:

- Compute $\hat{f}_k(\boldsymbol{x}) = \hat{\boldsymbol{w}}_k^T \boldsymbol{x}$ for $k = 1, \ldots, K$.
- Identify the largest component and classify to the corresponding class, *i.e.* $\hat{G} = \arg\max_{k \in \{1,\ldots,K\}} \hat{f}_k(\boldsymbol{x})$.

One may view the predicted class scores as estimates of the conditional class probabilities (or posterior probabilities), *i.e.* $P(G = k | \boldsymbol{X} = \boldsymbol{x}) \approx \hat{f}_k(\boldsymbol{x})$. However, these values are not the best estimates of posterior probabilities. Although the values sum to 1, they do not lie in the interval [0,1]. A way to overcome this problem is to estimate posterior probabilities using the *logit transform* of $\hat{f}_k(\boldsymbol{x})$. That is,

$$P(G = k | \boldsymbol{X} = \boldsymbol{x}) \approx \frac{e^{\hat{f}_k(\boldsymbol{x})}}{\sum_{l=1} e^{\hat{f}_l(\boldsymbol{x})}}.$$

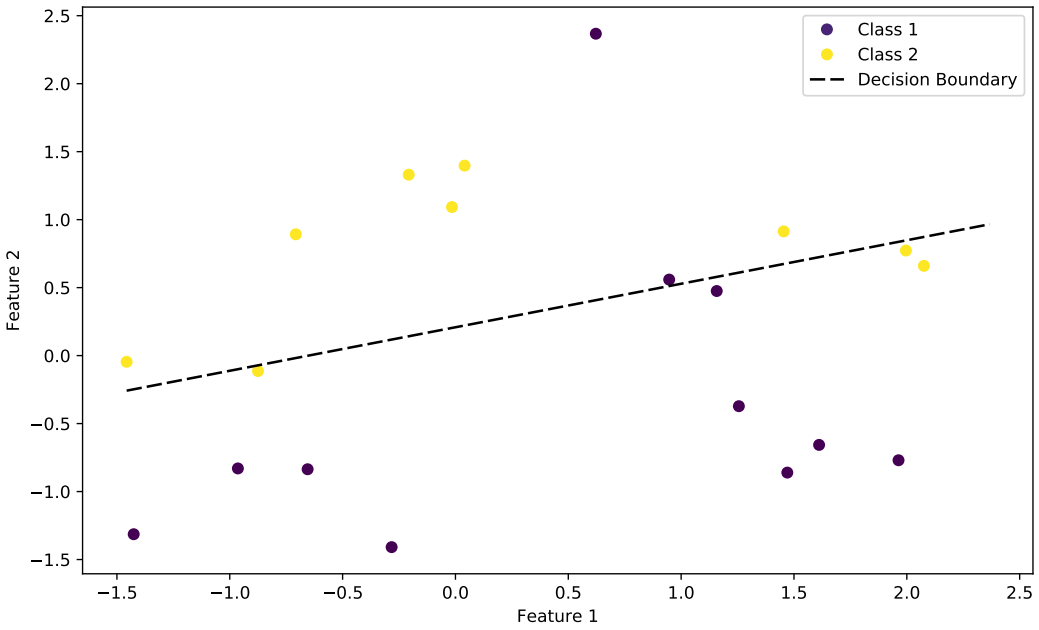


Figure 1.1: Linear model on simple binary classification dataset.

Through this transformation, the estimates of the posterior probabilities sum to 1 and are contained in [0,1]. The above model is the wellknown *logistic regression* model (Hastie *et al.*, 2009, p. 119). With this formulation there is no closed form solution for the weights. Instead, the weight estimates may be searched for by maximising the log-likelihood function. One way of doing this is by minimising the negative log-likelihood using gradient descent, which will be discussed in the next chapter.

Finally in this section, note that any supervised learning problem can also be viewed as a function approximation problem. Suppose we are trying to predict a variable Y given an input vector $\boldsymbol{X}$, where we assume the true relationship between them to be given by

$$Y = f(\boldsymbol{X}) + \epsilon,$$

where ϵ represents the part of Y that is not predictable from $\boldsymbol{X}$, because of, for example, incomplete features or noise present in the labels. Then in function approximation we are estimating f by $\hat{f}$. In parametric function approximation, for example in linear regression, estimation of $f(\boldsymbol{X}, \theta)$ is equivalent to estimating the optimal set of weights, $\hat{\theta}$. In the remainder of the thesis, we refer to $\hat{f}$ as the *model*, *classifier* or *learner*.

1.5 Outline

This chapter provided the context and some theoretical background for this study. An outline of the remainder of the thesis follows below:

In Chapter 2, the theory underlying neural networks is described. The building blocks of neural networks are discussed, thereby introducing neurons, basic layers and the way in which neural networks are trained. The important concept of regularisation is also discussed. Using the perspective of representation learning, we then attempt to gain insight into what happens inside a neural network.

Chapter 3 continues the discussion by focusing on the key advances in neural networks in recent times. The idea is that all concepts introduced in this chapter should potentially be able to facilitate the construction of improved deep neural networks on tabular data. Improved ways of preventing overfitting, such as data augmentation, the use of dropout and transfer learning, as well as the SotA training policy called *1Cycle* are analysed here. New developments in architectural design are also highlighted. The chapter concludes with approaches towards interpreting neural networks and their predictions.

Chapter 4 may be viewed as a core chapter of the thesis. It mainly serves as a literature review of all research with regard to deep learning for tabular data. The chapter is organised according to the modelling challenges faced when using deep learning for tabular data, investigating and comparing what other researchers have done in order to overcome these challenges. It will be seen that the key concept involves finding the right representation for tabular data. This may be done through embeddings, and by means of designing architectures that can efficiently learn feature interactions. This is for example done with attention models, possibly with the help of unsupervised pretraining.

In Chapter 5 we empirically investigate several claims made in the literature. The aim of the chapter is to evaluate and compare different approaches towards tackling the various challenges. Hence the main experiments involve evaluating neural networks at various samples sizes, evaluating potential gains from doing unsupervised pretraining and using data augmentation, and comparing attention modules with classic fully-connected layers. We also make use of permutation importance and knowledge distillation in order to illustrate a way in which neural networks may be interpreted.

The thesis concludes in Chapter 6, where we summarise our work, some

highlights and the main take-home points. The limitations of this study are discussed, and promising future research directions identified.

Chapter 2

Neural Networks

2.1 Introduction

Not unlike most supervised machine learning models, an artificial neural network is a function which maps inputs to outputs, *i.e.* $f : \boldsymbol{x} \rightarrow y$. The structure of f is often loosely compared to the structure of the human brain. Oversimplified, the brain consists of a collection of interconnected neurons. Each neuron can generate and receive signals. A received signal may be described as an input to a neuron, whereas a sent signal may be described as an output from that neuron. If two neurons are connected, it means that the output from the one neuron serves as input to the other. In a very simple model of the brain, one may argue that a neuron receives several signals, which it weighs and combines, and if the combined value of the inputs is higher than a certain threshold, the neuron sends an output signal to the next neuron. Figure 2.1 (a) provides a schematic of a biological neuron.

An artifical neural network tries to mimic this model of the human brain: it is set up to consist of several layers of connected units (or neurons). With exception of units in the first and final layers, each unit outputs a weighted combination of its inputs, combined with a simple non-linear transformation. In each layer of the neural network, the input is passed through each of the neurons. In turn, their output is passed to the next layer.

The transformation at each neuron is controlled by a set of parameters, also known as weights. Training a neural network involves tuning these weights in order to obtain some desired output. During training, the neural network receives as input a set of training data. The neural network weights are then

learned in such a way that, when given a new set of inputs, the output predicted by the neural network matches the corresponding response of interest as closely as possible. The process of using the training data to tweak the weights is done by means of an optimisation algorithm called Stochastic Gradient Descent (SGD).

Although recently there has been plenty of excitement around advances in neural networks, it is well known that they were invented many years ago. The development of neural networks dates back at least as far as the invention of perceptrons in (Rosenblatt, 1962). It is also interesting to compare modern neural networks with the Projection Pursuit Regression algorithm in statistics (Friedman and Stuetzle, 1981). Only recently a series of breakthroughs caused neural networks to become more effective, leading to the renewed interest in the field.

The aim of this chapter is to provide an overview of neural networks, emphasising their basic structure (§2.2) and the way in which they are trained (§2.3). This is done with a view to discuss modern neural network structures and training policies in Chapter 3, which in turn will help us shed light on Deep Learning for tabular data. In (§2.4) and (§2.5), regularisation for neural networks and the use of adaptive learning rates are discussed. These are necessary components for regulating the generalisation performance of NNs, as well as for keeping the required training time at bay. The chapter concludes with a section on representation learning (§2.6), which is an important topic toward understanding the inner workings of neural networks.

2.2 The Structure of a Neural Network

2.2.1 Neurons and Layers

In basic terms, a neural network processes an input $\boldsymbol{x}$ by sending it through a series of layers. The neurons in each layer apply some transformation to their inputs, resulting in a set of outputs which are again passed on to the next layer of neurons. Eventually, the final layer produces the neural network output. In this section we provide more detail regarding the neural network structure. We start with a description of the operations inside each neuron, followed by a discussion of the way in which the neurons may be connected in layers in order to form a complete neural network structure. Our discussion is based upon a

simple regression example.

Suppose we are in pursuit of a function which is able to estimate some continuous target, y, given a p-dimensional input $\boldsymbol{x}$, *e.g.* estimating the taxi fare from features such as distance travelled, time elapsed and number of passengers. A single neuron may act as such a function. It models y by computing a weighted average of the input features. This operation is illustrated in Figure 2.1 (b).

Figure 2.1: Comparison of a biological (a) and an artificial (b) neuron[1].

In equation form, this function may be written as

$$w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_p \cdot x_p + b = y,$$

where $\{w_k\}_{k=1}^p$, are the weights applied to each of the inputs $\{x_k\}_{k=1}^p$ and where b denotes the constant bias term. Clearly, this equation is simply the very common linear model and thus also can be written as

$$\boldsymbol{w}^\intercal \boldsymbol{x} + b = y,$$

where $\boldsymbol{x} = [x_1 \quad x_2 \quad \dots \quad x_p]^\intercal$ represents the input, and where resepctively $\boldsymbol{w} = [w_1 \quad w_2 \quad \dots \quad w_p]^\intercal$ and y denote the weights and output. We may of course compress the above equation to $\boldsymbol{w}^\intercal \boldsymbol{x} = y$, where $\boldsymbol{w}$ is amended to include the bias term, and where $\boldsymbol{x}$ is amended to include a 1, *i.e.* $\boldsymbol{x} = [1 \quad x_1 \quad \dots \quad x_p]^\intercal$ is the input, and $\boldsymbol{w} = [b \quad w_1 \quad \dots \quad w_p]^\intercal$ is the weight vector.

The weights convey the importance of each input feature in predicting the target. Larger values of $|w_k|$ indicate greater contributions of x_k toward the output. If $w_k = 0$, x_k has no influence on the target. However the weights are unknown and need to be estimated.

As discussed in Chapter 1, in linear regression this is done by means of OLS. However, since a neural network consists of many inter-conncected neurons,

[1]Image credit: https://www.jeremyjordan.me/intro-to-neural-networks/

an alternative estimation procedure is required. This is the topic of the next section.

Often a linear model will be too rigid to model a certain response of interest. In order to fit a more flexible model, we may add more neurons. Consider the use of two neurons, $z1$ and $z2$, where the second neuron ($z2$) accepts the same input as the first neuron ($z1$), but uses a different set of weights. Thus we have two different outputs produced by the two neurons, *i.e.* $z_1 = \boldsymbol{w}_1^\intercal \boldsymbol{x}$ and $z_2 = \boldsymbol{w}_2^\intercal \boldsymbol{x}$. In order to produce a final estimate from the initial two estimates, *viz.* z_1 and z_2, they are passed to a third neuron. That is, $y = \boldsymbol{w}_3^\intercal \boldsymbol{z}$, where $\boldsymbol{z} = [z_1 \quad z_2]^\intercal$. Figure 2.2 illustrates this pipepline in network form.

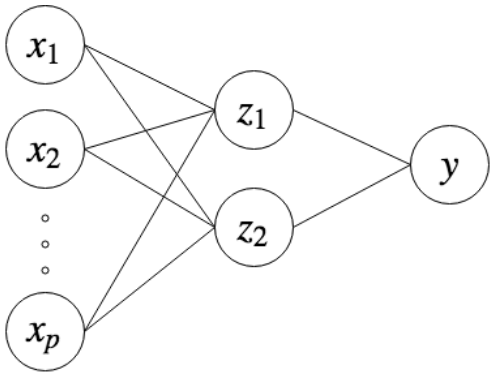


Figure 2.2: A simple neural network accepting p-sized inputs, with one hidden layer consisting of two neurons.

The first two neurons each received all p inputs and each produced a single output. These two outputs were received by the third neuron, and combined in order to produce the final output, *viz.* y. The operations performed by $z1$ and $z2$ may be expressed as $\boldsymbol{z} = W\boldsymbol{x}^\intercal$, where

$$W = \begin{bmatrix} \boldsymbol{w}_1^\intercal \\ \boldsymbol{w}_2^\intercal \end{bmatrix} = \begin{bmatrix} w_{10} & w_{11} & w_{12} & \dots & w_{1p} \\ w_{20} & w_{21} & w_{22} & \dots & w_{2p} \end{bmatrix} \quad \text{and} \quad \boldsymbol{z} = [z_1 \quad z_2]^\intercal.$$

The collection of $z1$ and $z2$ is called a layer. Since our third neuron (which is also a layer but with a single neuron) receives the output of this layer as input, it is possible to express the complete input-output relationship in one equation, *i.e.*

$$y = \boldsymbol{w}_3^\intercal \boldsymbol{z} = \boldsymbol{w}_3^\intercal W \boldsymbol{x}.$$

Note here that the weights from the first layer, *viz.* W, and the weights from the third neuron, *viz.* $\boldsymbol{w}_3$, may be collapsed into a single vector $\boldsymbol{w}$,

effectively reducing all of the neuron operations to a single neuron representation. Therefore the fitted model is still linear. In order to fit a non-linear model, a non-linear transformation function has to be applied to the output of each layer. This function is called an *activation function*.

Incorporating an activation function, the neural network equation may be written as

$$y = a_2\left(\boldsymbol{w}_3^\intercal a_1(W\boldsymbol{x})\right),$$

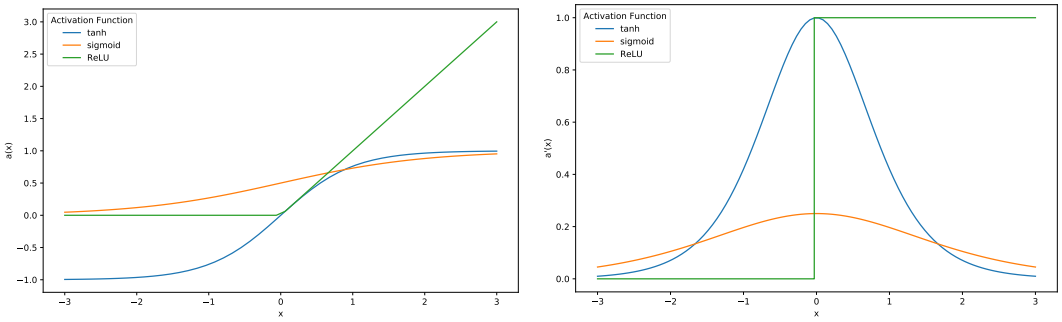
where a_1 denotes the activation function applied after the first (linear) layer, and where a_2 denotes the activation function applied after the final layer.

The introduction of non-linear activation functions serves to enlarge the class of functions that can be approximated by the network. That is, activiation functions enable the network to learn complex non-linear relationships between inputs and outputs. Next, we briefly discuss various activation functions.

2.2.2 Activation Functions

Since any simple non-linear and differentiable function can be used as activation function, there are plenty of activation functions to choose from.
Originally, the *sigmoid* activation function, *viz.* $\text{sigmoid}(x) = \frac{1}{1+e^{-x}}$, was a common choice (Rumelhart *et al.*, 1988). The S-shape of the sigmoid activation function, and its range between 0 an 1 is illustrated in Figure 2.3 (a). Note that the reason why the sigmoid function fell out of favour in terms of its use as activation function in neural networks is because of issues related to the gradient based optimisation procedure of NNs. For example, gradient weight updates that veer to far in different directions are caused by the values of sigmoid activations that are not centered around zero. Some other issues with the sigmoid activiation function are discussed in more detail in Section 2.3.

The hyperbolic tangent or *tanh* activation function, on the other hand, does return outputs centered around zero. It takes the form $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ and its shape is illustrated in Figure 2.3 (a). However, the problem with both the sigmoid and tanh activation functions is that they may lead to saturated gradients during training. To see this, consider the tails of the sigmoid and tanh functions, which indicate that the gradients of both these functions tend to zero as $|x| \to \infty$. During training, this may cause weight updates to be nearly zero, resulting in the network getting stuck at a certain point in the parameter space. Furthermore, the maximum gradient of the sigmoid activation function turns



(a) Function (b) Local Derivative

Figure 2.3: Activations.

out to be only 0.25 (at $x = 0.5$). The nature of the chain rule therefore causes lower layers in the network to train much slower than higher layers. The tanh activation function typically have larger derivatives than the sigmoid function. Thus it is not as susceptible to this vanishing gradient problem. However, it is still not immune to it. The local derivatives of the activation functions discussed in this section are illustrated in Figure 2.3 (b). The form of these functions will become more apparent in Section 2.3.

To date, the most popular choice in activation function is the *Rectified Linear Units* (ReLU) non-linearity. It is defined as:

$$\text{relu}(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} .$$

The shape of the ReLU activiation and of its derivative are illustrated in Figure 2.3 (a) and (b), respectively. Use of the ReLU alleviates the gradient vanishing problem as the derivative of this function is always 1 (for positive x values). This results in significantly shorter steps to convergence, as found by the authors in (Krizhevsky *et al.*, 2012). However, ReLUs may suffer from the "dead ReLU" problem, which is, if a ReLU neuron is assigned a zero weight, its weights receive zero gradients. In this way, neurons may be "clamped to zero" and may remain permanently "dead" during training. This sparsity of the activations is what some believe is the reason for the effectiveness of ReLUs (Sun *et al.*, 2014). If dead activations needs to be avoided, alternative activations functions to apply include PReLUs (He *et al.*, 2015*a*) and Leaky ReLUs (Maas *et al.*).

Selecting an appropriate activation function for a specific task typically

involves a trial-and-error process. For general tasks, the use of ReLUs after each hidden layer may suffice and has to some extent become standard practice. In the context of classification, for each class, it is often useful to produce outputs between 0 and 1 as estimates of conditional class probabilities. Therefore one often uses a sigmoid activation function on the output layer. In the context of (single label) multiclass classifcation, it is also desirable for these ouputs to sum to 1, therefore in this case, an activation function called *softmax* is typically used. Note that the softmax activation, *viz.* $\text{softmax}(\boldsymbol{x})_k = \frac{e^{x}k}{\sum_{l=1}^{K} x_l}$ is simply the logit transformation introduced in §1.4. In a regression context, we mostly omit the use of an activation function on the output layer.

In our empirical work, we will experiment with the use of the different activation functions discussed in this section, and compare their performance to that of *Scaled Exponential Linear Units* (SELUs) (Klambauer *et al.*, 2017), where the latter is supposed to facilitate more effective training of deeper neural networks. Note that the depth of a neural network refers to its number of hidden layers, whereas the width of a layer refers to the number of neurons it consists of. Selecting network depth and layer width is the topic of the next section.

2.2.3 Size of the Network

The network depth and the width of its hidden layers (*i.e.* the size of the network) are hyperparameters of the model. They control the ability of a neural network to model complex functions, which is also often referred to as its flexibility. In statistical learning it is well known that increasing the flexibility of a model is typically only beneficial up to a certain point, whereafter a further increase in flexibility will be detrimental to its prediction performance on new unseen data cases. Suboptimal test performance due to a too flexible model is known as *overfitting*. Appropriate selection of the flexibility of the model is also important in the case of neural networks. The challenge is to find a network size which is large enough to capture all the complexities in the data, but small enough to avoid overfitting. In addition, whereas more layers facilitate a more flexible fit, larger networks require more time and more hardware capacity in order to train them.

Currently the best way of finding the optimal size of a network for a given problem is by means of experimentation. Note therefore that for many of the

components of neural networks in deep learning, hyperparameter values are selected through a process of trial-and-error. Whereas appropriate specification of the size of a neural network is certainly important, in §2.4, we will see why tuning the size of a network is not necessarily the best way to control overfitting.

Theoretically, according to the universal approximation theorem (Cybenko, 1989), a neural network with a single hidden layer and with a finite number of neurons can approximate any continuous function. This begs the question: why are additional hidden layers required? As stated in (Ba and Caurana, 2013), although a neural network can represent any function, it does not mean that the available learning algorithm is able to find these optimal weights. Moreover, it may be the case that the number of neurons needed in order for a single hidden layer network to represent a specific function of interest, is infeasibly large. By choosing deeper networks we are assuming that the function we are trying to learn is composed of several simpler functions. Incorporating this prior belief has empirically been shown to be useful (Goodfellow *et al.*, 2016, pp.197-198). This is especially true in the case of tasks that may be partitioned into smaller subtasks, for example in computer vision.

In our empirical work we investigate the effect of network depth and layer width on the generalisation performance of neural networks. We also analyse why networks used in the case of tabular data are typically much shallower than in the case of computer vision or NLP applications. For some additional insight in the matter, later on in this chapter we view the problem of the specification of the network size from a representation learning perspective.

2.3 Training a Neural Network

Briefly, basic training of a neural network entails the following four steps:

1. Initialisation: random numbers are assigned to the network parameters.
2. Forward propagation: the input is passed through the network layers in order to produce an output.
3. Error calculation: the predicted output is compared to the true output, and the difference measures by means of an appropriate objective function.
4. Backward propagation: the gradients of the objective function with respect to the weights are obtained, and the network weights are updated

accordingly.

The above steps are typically repeated until the loss function is found to converge. Note however that convergence may require many training epochs. The following four subsections are each devoted to a discussion of one of the aforementioned steps.

2.3.1 Weight Initialisation

Taining a neural network starts with a weight initialisation step. In order to think about sensible initialisation, first note that we expect the number of positive and negative weights of a well trained neural network to be equal. If we initialise all weights to be zero, each neuron will compute the same output. Consequently, each neuron will produce the same gradient and undergo the same weight update. Hence we want to initialise the weights as small as possible, but each weight should be unique. Sampling the initial weights from the standard normal distribution seems to be a natural choice. However it turns out that the variance of the outputs from randomly initialised neurons grow as the number of inputs increase. Therefore an option is scale the weight vector generated from the standard norm distribution by the square root of its number of inputs. Such a scaling step will normalise the variance of the output, ensuring that all neurons in the network initially have approximately the same output distribution. In addition, it serves to improve the rate of convergence when training the network.

Another option for weight initialisation is the proposal by (**?**), which was made specifically in the context of using ReLU activation functions.

2.3.2 Optimisation

We have briefly seen in Chapter 1 that there is a connection between statistical learning and optimisation. Optimisation refers to the task of altering x in order to either minimise or maximise some *objective function* $J(x)$. When we are minimising the objective function, the latter is often also referred to as the *loss function*, or the *cost*. In the remainder of the thesis, note that these different terms for the loss function will be used interchangeably.

As mentioned in Chapter 1, parameter estimation (or optimisation) of a linear (or logistic regression) model is usually done using OLS or maximum

likelihood estimation (MLE). In this section, however, we discuss an alternative parameter estimation method which is also relevant in the optimisation of neural networks.

Therefore consider the MSE loss function:

$$\begin{aligned} L &= \sum_{i=1}^{N} L_i \\ &= \sum_{i=1}^{N} \sum_{k=1}^{K} (y_{ik} - f_k(\boldsymbol{x}_i))^2 \\ &= \sum_{i=1}^{N} \sum_{k=1}^{K} (y_{ik} - \boldsymbol{w}_k^T \boldsymbol{x}_i)^2, \end{aligned}$$

where in this case $f_k(\cdot)$ denotes the linear model used to predict the k-th class posterior probability. Note that although the MSE loss is mostly used in regression and not really well suited for classification, we make use of it here for illustration purposes.

In order to find the weights $\boldsymbol{w}$ that minimise L, we follow a process of iterative refinement. That is, starting with a random initialisation of $\boldsymbol{w}$, one iteratively updates the values such that L decreases. The updating steps are repeated until the loss converges. To minimise L with respect to $\boldsymbol{w}$, we calculate the gradient of the loss function at the point $L(\boldsymbol{x}; \boldsymbol{w})$. The gradient (or slope) of the loss function indicates the direction in which the function has the steepest rate of increase. Once we have determined this direction, we can update the weights by a step in the opposite direction - thereby reaching a smaller value of L.

The gradient of L_i is computed by obtaining the partial derivative of L_i with respect to $\boldsymbol{w}_k$, *i.e.*:

$$\frac{\partial L_i}{\partial \boldsymbol{w_k}} = -2(y_{ik} - \boldsymbol{w}_k^T \boldsymbol{x}_i)\boldsymbol{x}_i.$$

The above gradient is obtained for the loss at each data point, whereafter an update of the weight vector at the $(r+1)$-th iteration may be obtained as

$$\boldsymbol{w}_k^{(r+1)} = \boldsymbol{w}_k^{(r)} - \gamma \sum_{i=1}^{n} \frac{\partial L_i}{\partial \boldsymbol{w}_k^{(r)}},$$

where γ determines the size of the step taken towards the optimal direction and is called the *learning rate*. Of course γ needs to be specified by the user. One typically would like to set the learning rate small enough so that one does

not overshoot the minimum, but large enough to limit the number of iterations before convergence. The learning rate is a crucial parameter when training neural networks. Its significance is discussed in §2.5.

The procedure of repeatedly evaluating the gradient of the objective function, followed by a parameter update, forms the basis of the optimisation procedure for neural networks and is called *gradient descent* (Cauchy, 1847).

Note that a weight update is made by evaluating the gradient over a subset of the training observations, *viz.* $\{\boldsymbol{x}_i, i = 1, \ldots, n\}$. One of the advantages of gradient descent is that during each iteration, the gradient need not be computed over the complete training dataset, *i.e.* $n \leq N$. When updates are iteratively determined using subsets of the training data, the process is called *mini-batch gradient descent*. Of course the gradient obtained using mini-batches is only an approximation of the gradient of the full loss but it seems to be sufficient in practice (Li *et al.*, 2014). The option of using mini-batch gradient descent is extremely helpful in large-scale applications, since it obviates computation of the loss function over the entire training dataset. This leads to faster convergence, because of more frequent parameter updates, and allows processing of data sets that are too large to fit in a computer's memory. A choice regarding batch size depends on the computation power available. Typically a batch consists of 64, 128 or 256 data points, since in practice many vectorised operation implementations work faster when their inputs are sized in powers of 2. Note at this point that the collection of iterations needed to make one sweep through the training dataset is called an *epoch*.

An extreme case of mini-batch gradient descent is when the batch size is selected to be 1. This is called *Stochastic Gradient Descent* (SGD). Recently SGD has been used much less, since it is more efficient to calculate the gradient in larger batches of training data cases. However, note that it remains common to use the term SGD when actually referring to mini-batch gradient descent. The use of gradient descent in general has often been regarded as slow or unreliable. SGD will most probably not even find a local minimum of the objective function, however it typically finds a very low value of the cost function quickly enough to be useful. Thus gradient descent has been proven to be efficient for optimising neural networks.

2.3.3 Optimisation Example

In order to illustrate the SGD algorithm, we consider the linear model in a binary classification context, *i.e.* $K = 2$. Also in our example, suppose the training data are generated in the way described in (Hastie *et al.*, 2009, pp. 16-17), where the inputs are two-dimensional, *i.e.* $p = 2$. Suppose we want to fit a linear regression model to the training data and classify an observation to the class with the highest predicted score. Of course in the binary classification it is only necessary to model one class probability: an observation is then assigned to the corresponding class if the score exceeds some threshold (usually 0.5). Therefore the decision boundary is given by $\{\boldsymbol{x} : \boldsymbol{x}^T\hat{\boldsymbol{w}} = 0.5\}$.

Optimisation of the regression weights by means of gradient descent is illustrated in Figure 2.4. The colour-shaded regions represent the regions of the input space classified to the respective classes, as determined by the decision boundary based upon the OLS parameter estimates. Since the number of training observations are small, it was not necessary to make use of mini-batch gradient descent. Note that the learning rate was set equal to 0.001. The decision boundaries defined by the gradient descent parameter estimates at different iterations are represented by the dashed lines in Figure 2.4. Initially the estimated decision boundary lies far from the OLS solution. However, after convergence (29 iterations later), the gradient descent line matches the OLS line.

2.3.4 Backpropagation

Currently, SGD is the most effective way of training deep neural networks. Recall that in Section 2.3.2 we described how to fit a linear model using the SGD optimisation procedure. We have seen that SGD optimises the parameters θ of a network to minimise the loss function. That is,

$$\theta = \arg\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} l(\boldsymbol{x}_i, \theta).$$

Using SGD, training a neural network involves several iterations. During each iteration we consider a mini-batch consisting of $n \leq N$ training examples. The mini-batch is used to approximate the gradient of the loss function with respect to the parameters via the following derivative:

$$\frac{1}{n} \frac{\partial l(\boldsymbol{x}_i, \theta)}{\partial \theta}.$$

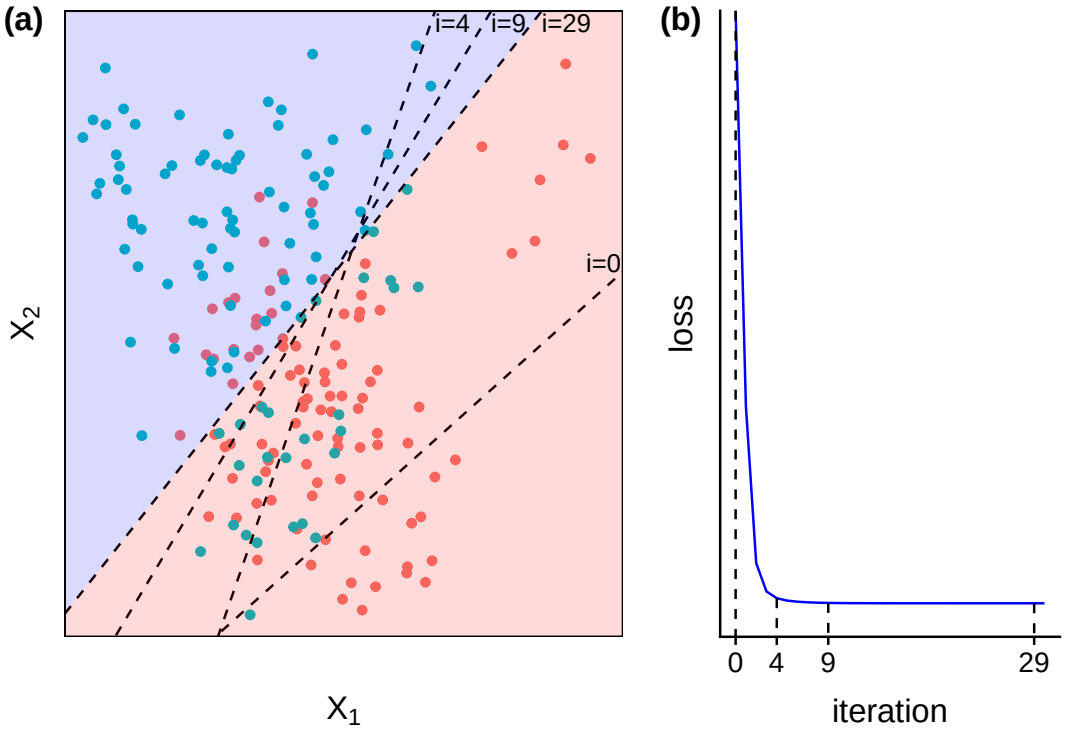


Figure 2.4: Plots of the gradient descent example. (a) The training data points in input space. The shades in the background represent the class division in input space, with the decision boundary determined by linear least squares estimation. The dashed lines represent the gradient descent decision boundaries at different iterations. (b) The loss function at each iteration.

Using a mini-batch of training examples instead of one training example at a time produces a better estimate of the gradient over the full training set, and is computationally much more efficient.

In this section we discuss the same procedure, but applied to a simple single hidden layer NN for multiclass classification, which may be decomposed as follows:

$$f_k(\boldsymbol{x}) = g_k(\boldsymbol{\beta}_k^\intercal \boldsymbol{z}), \quad k = 1, \ldots, K$$
$$z_m = \sigma(\boldsymbol{\alpha}_m^\intercal \boldsymbol{x}), \quad m = 1, \ldots, M$$

where $\sigma(\cdot)$ is the sigmoid activation and where $g(\cdot)$ is the softmax activation. Here there are two sets of unknown adjustable weights that defines the input-output function of the network. They are the parameters of the linear function of the inputs, *viz.* $\boldsymbol{\alpha}_m = (\alpha_{0m}, \alpha_{1m}, \ldots, \alpha_{pm})$, and also the parameters of the linear transformation of the derived features, *viz.* $\boldsymbol{\beta}_k = (\beta_{0k}, \beta_{1k}, \ldots, \beta_{mk})$. If we denote the complete set of parameters by θ, then recall that the objective function for regression may be chosen to be the sum of squared errors, *i.e.* we

have

$$L(\theta) = \sum_{k=1}^{K} \sum_{i=1}^{N} \left(y_{ik} - f_k(\boldsymbol{x}_i)\right)^2,$$

whereas in the context of classification, the loss function may be specified as the so-called *cross-entropy*. The latter loss function is defined as follows:

$$L(\theta) = -\sum_{i=1}^{N} \sum_{k=1}^{K} y_{ik} \log f_k(\boldsymbol{x}_i),$$

with the corresponding classifier denoted by $G(\boldsymbol{x}) = \arg\max_k f_k(\boldsymbol{x})$. Since a neural network for classification takes the form of a linear logistic regression model in the hidden units, note that the parameters may be estimated using maximum likelihood. According to Hastie *et al.* (2009, p. 395) however, the global minimiser of $L(\theta)$ is most likely an overfit solution. Therefore we require regularisation techniques when minimising $L(\theta)$. Furthermore, as the size of the network increases, MLE soon becomes intractable.

Therefore, one rather uses gradient descent and the *backpropagation* algorithm (Rumelhart *et al.*, 1988) to minimise $L(\theta)$. This is possible because of the modular nature of a neural network, allowing the gradients to be derived through iteration of the chain rule for differentiation. In broad terms, the iterative calculation of derivatives occur during a forward and backward sweep over the network, keeping track only of quantities local to each unit.

In more detail, the backpropogation algorithm for the sum-of-squared error objective function, previously given as

$$\begin{aligned} L(\theta) &= \sum_{i=1}^{N} L_i \\ &= \sum_{i=1}^{N} \sum_{k=1}^{K} (y_{ik} - f_k(\boldsymbol{x}_i))^2, \end{aligned}$$

is as follows. We start by obtaining the derivatives required in order to implement gradient descent. For this example, following the chain rule, the relevant derivatives are

$$\begin{aligned} \frac{\partial L_i}{\partial \beta_{km}} &= -2(y_{ik} - f_k(\boldsymbol{x}_i)) g_k'(\boldsymbol{\beta}_k^T \boldsymbol{z}_i) z_{mi}, \\ \frac{\partial L_i}{\partial \alpha_{ml}} &= -\sum_{k=1}^{K} 2(y_{ik} - f_k(\boldsymbol{x}_i)) g_k'(\boldsymbol{\beta}_k^T \boldsymbol{z}_i) \beta_{km} \sigma'(\boldsymbol{\alpha}_m^T \boldsymbol{x}_i) x_{il}. \end{aligned}$$

Given these derivatives, a gradient descent update at the $(r+1)$-th iteration takes the form

$$\beta_{km}^{(r+1)} = \beta_{km}^{(r)} - \gamma_r \sum_{i=1}^{N} \frac{\partial L_i}{\partial \beta_{km}^{(r)}},$$
$$\alpha_{ml}^{(r+1)} = \alpha_{ml}^{(r)} - \gamma_r \sum_{i=1}^{N} \frac{\partial L_i}{\partial \alpha_{ml}^{(r)}}.$$

We may now rewrite the gradients as follows:

$$\frac{\partial L_i}{\partial \beta_{km}} = \delta_{ki} z_{mi},$$
$$\frac{\partial L_i}{\partial \alpha_{ml}} = s_{mi} x_{il}.$$

Note that the quantities δ_{ki} and s_{mi} are errors from the current model at the output and hidden layer units respectively. From their definitions it can be seen that

$$s_{mi} = \sigma'(\boldsymbol{\alpha}_m^T \boldsymbol{x}_i) \sum_{k=1}^{K} \beta_{km} \delta_{ki},$$

which is known as the so-called *backpropagation equations*. Using the above set of equations, weight updates proceed by means of an algortihm consisting of a forward and a backward pass over the network. In the forward pass, the current weights are fixed and the predicted values $\hat{f}_k(\boldsymbol{x}_i)$ are computed. In the backward pass, the errors δ_{ki} are computed, and then channelled via the backpropogation equations in order to specify s_{mi}. These values are then used to update the weights.

We have seen that backpropagation is a simple algorithm. It can easily be implemented on any size network and differentiable layers. Its local nature (each hidden unit only passes information to and from its connected units) allows it to be implented efficiently in parallel. Another advantage is that the computation of the gradient can be done on a batch of training observations. This allows the network to be trained on very large datasets.

Having discussed basic NN training, we now turn to a discussion of two components that are very important in ensuring neural networks to be useful. Basic ways to regularise a neural network, and the use of adaptive learning rates are discussed in the following two sections.

2.4 Basic Regularisation

In §2.2.3 the importance of selecting an appropriate size for a neural network was emphasised. It may seem that smaller networks should generally be preferred in order to prevent an overfit. However smaller networks are harder to train with local methods such as gradient descent because their loss functions have poor local minima that are easy to converge to. In contrast, local minima of larger networks are typically smaller (Choromanska *et al.*, 2014).

There are indeed more effective ways of regularising neural networks. We briefly discuss these methods in the remainder of this section, and in Chapter 3. One of the preferred ways of preventing overfitting in neural networks is by using L1 or L2 regularisation, *i.e.* by adding a penalty term to the objective function, where the penalty term is proportional to the magnitude of the NN weights. The role of the penalty term is to encourage the weight estimates to be small. This is of course the same strategy as the one followed when regularising the least squares linear model by means of Ridge Regression or the Lasso (Hastie *et al.*, 2009, Ch. 4). The difference between L1 and L2 regularisation lies in the form of the penalty term, which is $\lambda|w|$ in L1 regularisation, and $\frac{1}{2}\lambda w^2$ in the case of L2 regularisation. Specification of the λ parameter is important since it determines the severity of the penalty of large NN weight values. Note that the "$\frac{1}{2}$" in front of the L2 penalty is added for the sake of convenience since it renders the derivative of the penalty term equal to λw. The latter form implies a linear decay of the weights towards zero, *i.e.* $w' = w - \lambda w$, which is also known as *weight decay*. In neural network applications it has been shown that L2 regularisation typically outperforms L1 regularisation and is therefore preferred (Reference?)

An alternative way of preventing a neural network to overfit is so-called *early termination* of the training process. That is, one refrains from training the network until the training loss converges. Since the training loss after convergence is not equivalent to the test loss, the loss function on a validation set should be monitored during training. Early termination involves stopping the training process as soon as the validation loss stops decreasing.

> As mentioned before, the learning rate also plays a big part in finding the optimal weights. Next we discuss how we can tune the learning rate to train faster and to find better local minima. More advanced regularisation techniques are discussed in Chapter 3.

2.5 Adaptive Learning Rates

Although in the neural network literature it is known that a more optimal learning rate may reduce the time to train the network and improve its test performance, optimal specification of this parameter is not such an easy task. A small learning rate slows downs the training time, but is safer against overfitting and overshooting the optimal solution. With a large learning rate, convergence may be reached quicker, but the optimal solution may not have been found. Toward appropriate specification of the learning rate one may consider doing a line search over a range of possible values. However in the case of large networks such an approach is typically too time consuming.

In contrast to a once off specification of the learning rate, an alternative approach is to allow the learning rate to be adapted during the training process. A popular approach is to decrease the learning rate by a fraction after a fixed number of epochs, or as soon as the validation loss starts to converge (*cf.* for example He *et al.*, 2015a). The intuition is that larger steps may be afforded while the weight estimates are still far away from an optimal position on the loss surface, while smaller steps need to gradually be taken once closer to the optimal weight vector value in order to take care not to overshoot it. Note that an adaptive learning rate requires one to also tune the rate of decrease and the time steps of each decrease during training. Fortunately it is believed that neural network learning algorithms are not very sensitive to these choices.

In addition, there are ways of manipulating the learning rates at a local level, as opposed to the aforementioned global methods. Adagrad (Duchi *et al.*, 2011) is an adaptive learning rate method which increases the learning rate at neurons with small gradients and *vice versa*. Adam (Kingma and Ba, 2014) is the most commonly used weight update approach. It also uses the magnitude of the gradient to control each weight update, in addition to the previous iteration's gradients and it combines them in a smooth fashion. This resembles the physical property of momentum (Bengio *et al.*, 2012). For more detail, the reader can refer to the cited publications as it falls out of the scope of this work.

2.6 Representation Learning

We are now familiar with the mathematical operations of basic layers, how they are connected and how their weights are tweaked to minimise a loss function. In this section we discuss why this works and what the neural network is actually doing to model the data. The central idea is that of a *data representation* (Bengio *et al.*, 2013) and that at each layer of the network the data is transformed into a higher-level abstraction of itself. Understanding and interpreting neural networks remains a challenge (Frosst and Hinton, 2017), but the notion of learning an optimal data representation allows us to gain a deeper intuition of the inner mechanics of neural networks.

Machine learning models are very sensitive to the form and properties of the input it is presented with. Thus a large part of constructing machine learning models is to find the best way of representing the raw data in order to simplify the extraction of useful information. This *feature engineering* process typically is a laborious manual task which entails creating, analyzing, evaluating and selecting appropriate features[2]. There is therefore unfortunately no systematic recipe for feature engineering. Instead, it is a trial-and-error process which requires practitioner expertise and domain-specific knowledge. In representation learning, the idea is to find a way of effectively automating the feature engineering process. That is, the goal is to automatically learn representations of the data that make it easier to extract useful information for classifiers or other predictors (Bengio *et al.*, 2013). Such automation seems to have the potential of saving a lot of time and raising the performance ceiling of machine learning models.

Importantly, a neural network may be viewed from the perspective of representation learning. To see this, consider a classification task. Since the final layer of a neural network is a linear model, in order for the network to produce accuracte predictions, the previous layers should be able to project the data into a space where the classes are linearly separable. Thus the network needs to learn a representation of the data that is optimal for classification.

Starting with the raw input, each of the simple (but non-linear) modules of a neural network transforms the data representation at one level into a representation at a higher, slightly more abstract level. Each transformation may create and/or emphasise features that are important for discrimination,

[2]http://blog.kaggle.com/2014/08/01/learning-from-the-best/

and drop those which are redundant. When a sufficient number of such transformations are combined, very complex functions may be learned by a neural network (Lecun *et al.*, 2015).

In order to illustrate the data representations learned by a neural network, consider the following simple example. Suppose we have available a dataset with two classes, *viz.* the two curves on a plane as displayed in Figure 2.5. In their original form, clearly the observations from the two classes are not linearly separable. If we fit a single layer neural network to these data (*i.e.* a network with only an output layer), the resulting decision boundary can only be linear (as shown in Figure 2.6) and will thus be unsatisfactory. However, if we fit a two-layer neural network to the same dataset (where the hidden layer has two neurons and a sigmoid activation), the resulting decision boundary perfectly separates the two classes (as shown in Figure 2.7).

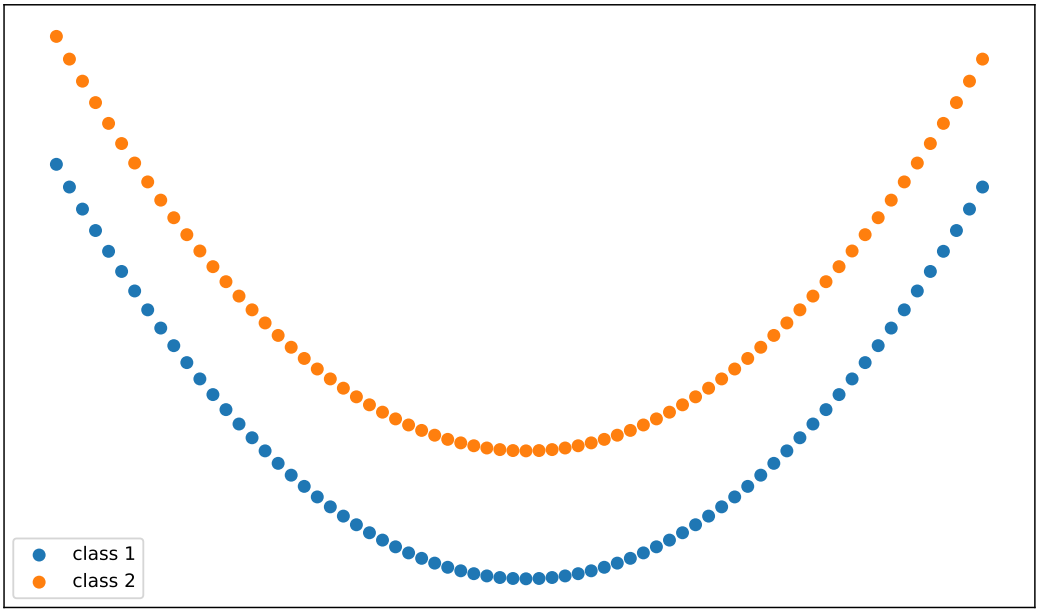


Figure 2.5: Simple dataset with two linearly inseparable classes.

Since the hidden layer consists of only two neurons, we are able to plot the output from the hidden layer after the raw data has passed through it. This is depicted in Figure 2.8. From \autoref{fig:simple_dataset_complexNN_rep it can be seen how the hidden layer projects the input data into a space where the observations from the two classes are linearly separable, which then leaves it to the final layer to find the best hyperplane between the two classes.

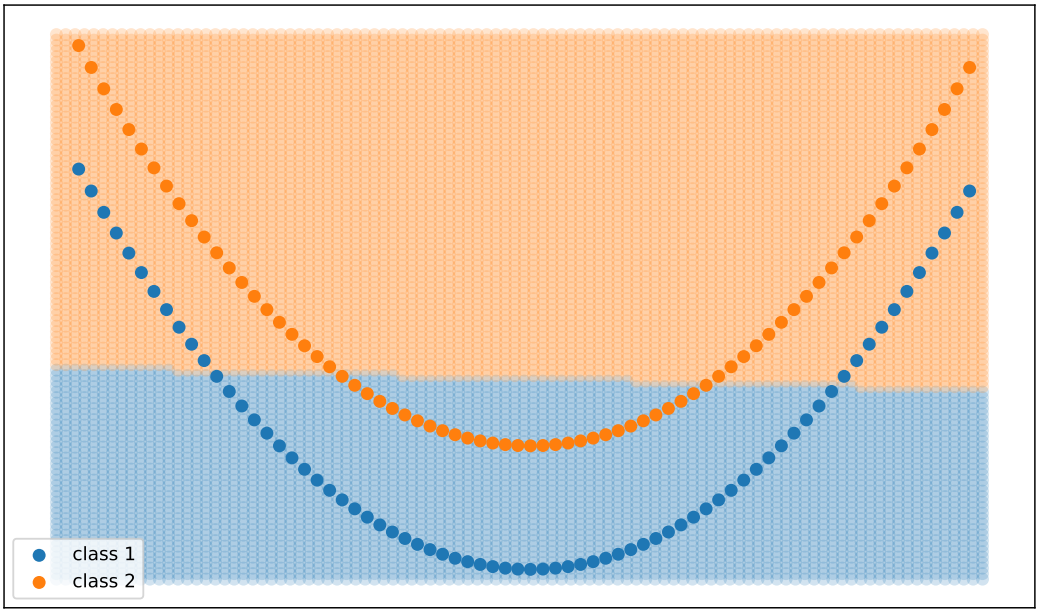


Figure 2.6: Decision boundary of a single-layer neural network.

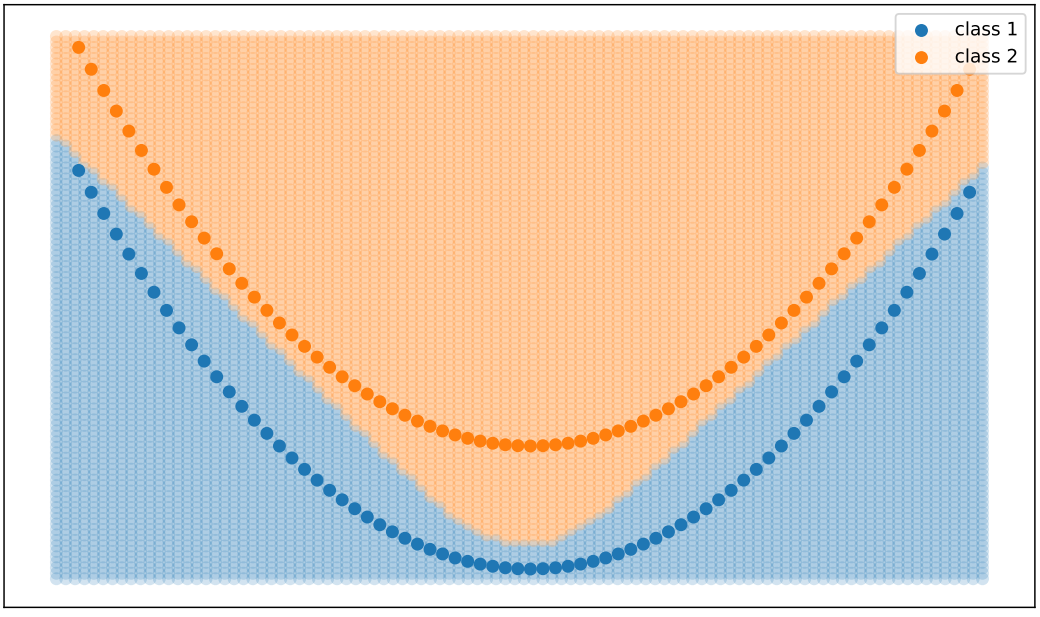


Figure 2.7: Decision boundary of a two-layer neural network.

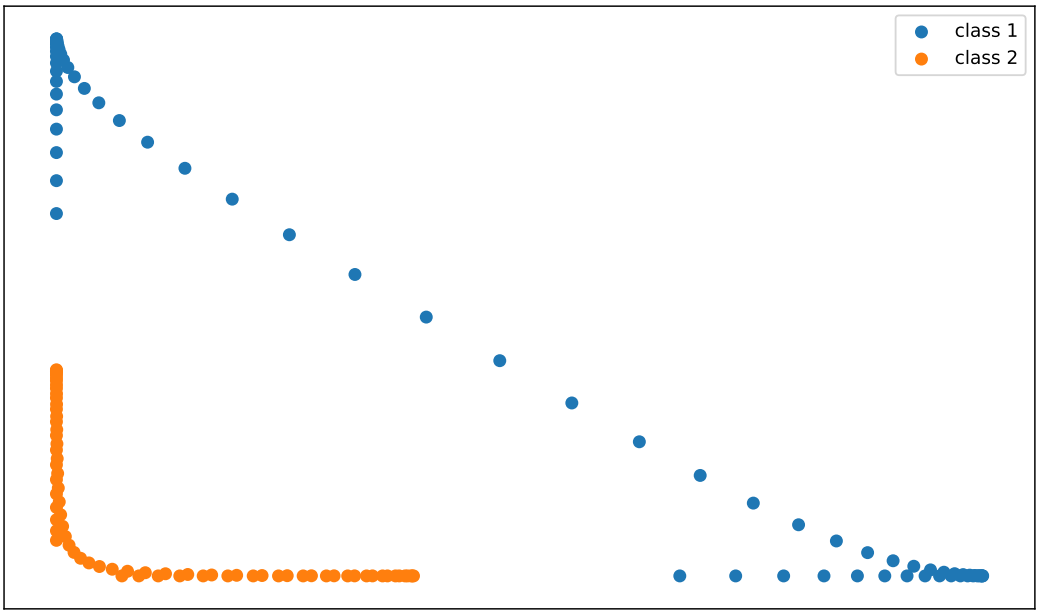


Figure 2.8: Hidden representation of a two-layer neural network.

Although the above example represents a very simple data setup and neural network architechture, the same concepts apply to more complicated datasets and models. It should however be noted that although it is technically possible to separate any arrangement of points with a sufficiently large network[3], in reality it can become quite challenging to find such representations. This is where the need for more data, regularisation, smarter optimisation procedures and architecture design arises. Without the aforementioned, it is likely that the network will get stuck in a sub-optimal local minimum, unable to find the optimal representation of the data. In the chapters to follow we explore the approaches available to find optimal representations for tabular data in regression and classification contexts.

[3]http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

Chapter 3

Deep Learning

3.1 Introduction

Deep learning is a broadly used term. To many, the difference between a classical neural network and a deep neural network lies merely in the number of network layers. Even a network with two hidden layers is sometimes referred to as 'deep'. From this perspective, deep (multilayer) neural networks were already proposed by Alexei Ivakhnenko and colleagues more than half a century ago (Ivakhnenko and Lapa, 1965). In the paper by Vincent *et al.* (2008) it is stated that "the belief that additional levels of functional decomposition will yield increased representational and modeling power is not new". The authors then cite McClelland *et al.* (1986), Hinton (1989), and Utgoff and Stracuzzi (2002).

Still the boom in deep learning research may be said to have started only a bit more than a decade ago, since effective training of a multi-layer network was only made possible after a breakthrough presented in the papers by Hinton *et al.* (2006), **?**, **?**, Bengio *et al.* (2007), and Lee *et al.* (2008). From around 2006 onwards, tons of contributions have been made to the neural network field. New types of layers, more effective training strategies, and novel approaches to guard against overfitting have since been proposed. A more general definition of *deep learning* therefore encompasses all modern developments in the neural network space. Because of the large number of deep learning contributions during the past decade, note that in this chapter we restrict attention to developments that form a basis for deep learning in the context of tabular data.

Conceptual categorisation of modern contributions to the deep learning

field is a difficult task. Perhaps one way to order some of the more important developments, is to use the aim or effect of their implementation as criterion. Since representation learning is such an important characteristic of NNs, many deep learning developments focus on this aspect. As mentioned before, neural networks are able to approximate any function. However, learning algorithms are not necessarily able to find these solutions, therefore we need more efficient ways to learn from the available data. An example is the class of neural networks called *autoencoders* (§3.2). These neural networks are mostly used as unsupervised learning methods with the aim of aiding us to learn more robust data representations, which may subsequently be transferred to supervised learning algorithms. We have mentioned in Chapter 1 that the process of transferring knowledge from one network to another is called transfer learning. Note that more detail regarding transfer learning is provided in §4.4.2.

A large number of deep learning developments are explicitly geared toward regularisation in order to prevent overfitting. We discuss two important modern proposals that fall in this category, *viz.* the use of the so-called *dropout* method in (§3.4.1) and the use of *data augmentation* in (§3.4.2).

A miscellaneous group of modern deep learning proposals that succeed in hitting several targets, are discussed in (§3.5.1), (§3.5.2), (§3.5.3) and (§3.5.4). Amongst others, the successes of this group of developments include regularisation, increasing speed, and reducing sensitivity to starting weights (batch normalisation); improving accuracy (skip connections); representing and visualising categorical variables, and representation learning (embeddings); and visualising and interpreting NNs (attention). The use of these modern architecture have become standard practice in deep learning applications.

With regard to optimisation of neural networks, a method has been proposed which provides a way of almost automatically finding a good learning rate. This leads to a drastic reduction in the number of training iterations needed. The aforementioned training method (referred to as the *1cycle policy*) is discussed in (§3.6).

We conclude the chapter with a discussion on the interpretability of neural networks in (§4.5).

3.2 Autoencoders

Autoencoders play a pertinent role in deep learning. According to **?**, the idea of autoencoders have been around since the late 1980's - see for example **?**, **?** and **?**. In those years, autoencoders were used for dimensionality reduction and for learning important features in a more general domain than neural networks. From 2006 onwards, their application in neural networks facilitated efficient training of deep neural nets, which in turn caused an upsurge in deep learning research (Hinton *et al.*, 2006; **?**; Bengio *et al.*, 2007). Autoencoders are also used to facilitate transfer learning, which we will see is an important way of ensuring efficient training in deep neural networks.

Some of the most popular types of autoencoders are so-called *denoising autoencoders* (Vincent *et al.*, 2008), *stacked denoising autoencoders* (**?**), *sparse autoencoders* (Makhzani and Frey, 2013), *contractive autoencoders* (**?**), *semi-supervised recursive autoencoders* (**?**), and *variational autoencoders* (Kingma and Welling, 2013). In this section, we start with an explanation of a plain vanilla autoencoder. This is followed with a brief description of denoising, sparse, variational and contractive autoencoders.

A basic autoencoder is a neural network which is trained to attempt to reconstruct its inputs. The simplest form of an autoencoder is a neural network with one hidden layer and with an output layer having the same size as the input layer. This architecture is illustrated in Figure 3.1). The linear layer transforming the input to the hidden layer is referred to as the *encoder*, *viz.* $\boldsymbol{z} = f(\boldsymbol{x})$, and the layer producing the output from the hidden layer is called the *decoder*, *viz.* $\boldsymbol{x}' = g(\boldsymbol{z})$. The autoencoder can be trained in the same way as any other neural network, however the loss function to be minimised is called *reconstruction loss*. The reconstruction loss function measures the difference between the reconstructed and actual input, hence the MSE loss function is a common choice in the case of continuous data. Hence technically, note that autoencoders belong to the self- (or semi-) supervised class of methods, although many still think of it as unsupervised. It is unsupervised in the sense that it does not require labelling, but it is supervised in the sense that it does predict an ouput.

If the number of neurons in the hidden layer of an autoencoder is greater than or equal to the number of input features, it is possible to perfectly reconstruct $\boldsymbol{x}$ from $\boldsymbol{z}$, *i.e.* $\boldsymbol{x}' = \boldsymbol{x}$. This is however not a very useful model. A

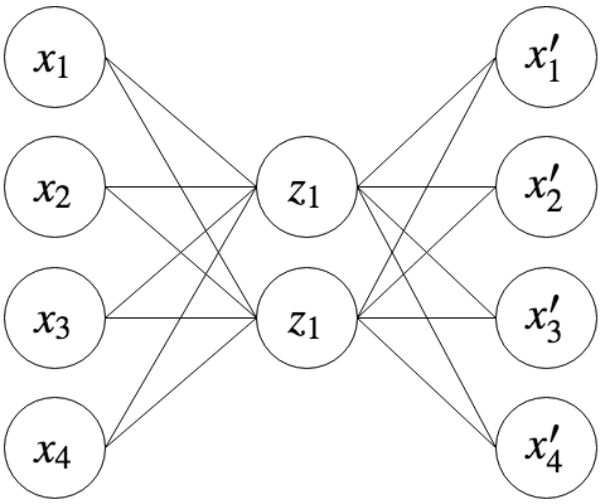


Figure 3.1: A simple single hidden layer autoencoder with four-dimensional inputs and with two neurons in the hidden layer.

useful autoencoder is an NN that succeeds in finding latent representations of the data that are of a smaller dimension than that of the input domain, thereby learning useful hidden data features. These data features may then be carried over to neural networks for supervised learning tasks. In this way, autoencoders are frequently use to initialise the weights of a supervised learning network. That is, the learned weights of the autoencoder are use to initialise the weights of the supervised learning network (of the same size) (Larochelle *et al.*, 2009).

Note that in order to encourage representation learning by means of autoencoders, they are typically constructed with some type of constraint imposed. A common option is to restrict the number of neurons in the hidden layer to be smaller than the number of input features. This forces the autoencoder to capture only the most useful properties of the data in the hidden representation and can thus effectively be used as a way of dimensionality reduction (Hinton and Salakhutdinov, 2006). Indeed, if there are no non-linear activation functions after the linear layers, one can show that this autoencoder is equivalent to application of *Principal Component Analysis* (PCA) to the inputs. Of course there is no restriction to the number or the size of the layers used in the encoder and decoder, and if activation functions are used, one can potentially learn a more powerful non-linear generalisation of PCA. It is difficult to verify whether or not an autoencoder has learned a useful latent representation of the data. One way to evaluate the representation is to use the features extracted by the encoder in a supervised learning task and to then compare its performance to

that of a model using only the raw data as inputs.

An alternative way of imposing constraints in autoencoders, is to add noise to the inputs before passing it to the encoder. This is the strategy implemented by *denoising autoencoders* (DAEs) (Vincent *et al.*, 2008). Thus in order to minimise the reconstruction loss, DAEs are required to learn how to reconstruct the original inputs from a corrupted version of themselves. The choice of the type of noise added to the inputs depends on the available data types. One may block out inputs with zeros if zeros have no other meaning in the data, or one might want to add Gaussian noise to continuous outputs as long as it falls within the true range of the features. The type and amount of noise to be used are factors that practitioners typically experiment with. If too much noise is added in the corruption step, the DAE might not learn anything useful. The interested reader is referred to the paper by Alain and Bengio (2014), where it is shown how DAEs are able to learn useful data structures.

Instead of limiting the number of neurons in the hidden layer, one may instead use a hidden layer with more neurons than the number of inputs. However, the number of hidden neurons that may be active at the same time, is restricted. The restriction is enforced by means of the addition of a regularisation term in the reconstruction loss, or one may manually set all but k of the weights with largest absolute value equal to zero. In this way we may fit a so-called *sparse autoencoder*.

Another type of autoencoder is a *variational autoencoder* (VAE). In VAEs, it is assumed that the input is generated by a directed graphical model $p(\boldsymbol{x}|\boldsymbol{z})$. The aim is then to learn the posterior distribution $p(\boldsymbol{z}|\boldsymbol{x})$. In the decoding step, observations are sampled from the learned distribution before passing it to the set of fully connected layers.

Finally in this section, in a *contractive autoencoder* (CAE), one encourages learning useful features by means of the use of a regularised reconstruction loss function.

3.3 Transfer Learning

Semi-supervised (or 'unsupervised) learning exploited for example in autoencoders played a key part in the rise of deep learning. This is also stated in the paper by **?**: "Training a deep network to directly optimise only the supervised objective function of interest by gradient descent, starting from random

initialised parameters, does not work very well. What works *much* better is to use a local unsupervised criterion to (pre)train each layer in turn, with the goal of learning to produce a useful higher-level representation from the lower-level presentation output by the previous layer. From this starting point on, gradient descent on the supervised objective leads to much better solutions in terms of generalisation performance." Indeed, autoencoders and related methods facilitated successful deep neural networks using transfer learning or *pretraining*. For example, it is possible to use a DAE to learn the latent features from unlabelled data, and then to use these features to initialise a deep neural net for a related supervised learning task based on the same type of inputs.

Note that instead of using autoencoders in pretraining, one may also do supervised pretraining. This is for example done by first training the network to estimate a certain target variable (say y), and by then using those weights or features when trying to predict a different target variable (say y').

Unsupervised pretraining for supervised learning is very common in NLP (Devlin *et al.*, 2018, Howard and Ruder (2018)) , whereas supervised pretraining for supervised learning is widely used in computer vision (Yosinski *et al.*, 2015, He *et al.* (2015*a*)). To our knowledge, theoretical proofs towards understanding why pretraining works, cannot yet be found in the literature. It is postulated that using the pretrained weights as initialisation to the supervised model provides a better starting position on the loss surface, thereby inducing regulatory effects (Goodfellow *et al.*, 2016, Ch. 14). Pretraining is most effective in scenarios where a relatively small dataset is available for the supervised task, but where a lot of data are available for the pretraining task.

We conclude this section with the following illustration of pretraining and transfer learning, taken from Zeiler and Fergus (2014). By observing the types of features extracted from a trained image model (Figure 3.2), one gains insight into why they are also effective in other image analysis tasks. The learned filters seem to identify generic image features such as edges and color gradients which should prove useful in most computer vision tasks.

3.4 More Regularisation

In §2.4 we discussed basic regularisation methods for neural networks. There are however many alternative ways of combatting overfitting. Here we discuss

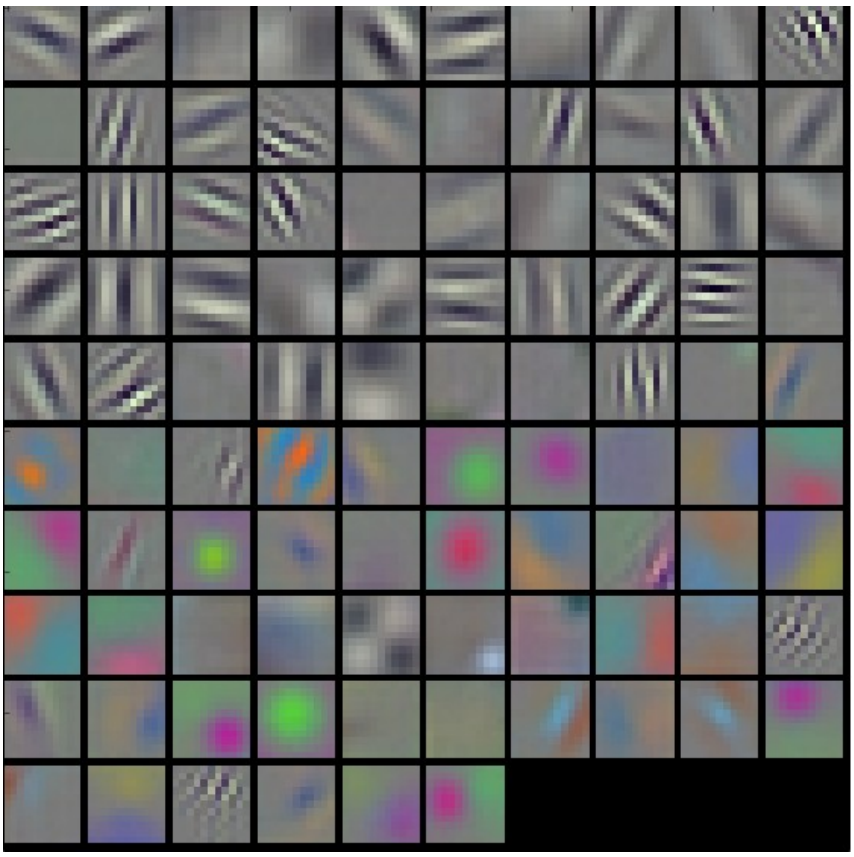


Figure 3.2: Visualising the first layer convolutional filters leared by a neural network in a large image dataset.

two regularisation techniques proven to be extremely powerful in almost every application of deep learning.

3.4.1 Dropout

Regularisation by means of dropout was proposed in a paper by Srivastava and co-authors (2014). The term refers to the process of temporarily removing sets of neurons and their connections during NN training. In Srivastava *et al.* (2014), the authors explain that the dropout invention started with the challenge of attempting to optimise neural networks by means of averaging multiple neural networks. This pursuit was motivated by the following two notions. First, it may be viewed as a gold standard for 'regularising' the network across all possible parameter values. Here the idea is to train a huge number of fixed size neural networks, where each network is optimised using a different parameter value. This is done for all possible settings of the parameter values.

Conceptually, one would then obtain the final NN by averaging the weights of all these neural networks, where the output corresponding to each parameter setting should be weighted by its posterior probability given the training data. Of course obtaining this oracle is impracticable, but in dropout Srivastava *et al.* (2014) found a way to approximate it. Second, it is well known in statistical learning and in machine learning that model averaging typically leads to an improvement in generalisation performance. Construction of an ensemble neural network using the classical ensemble framework is however infeasible. For ensemble models to be successful, predictions resturned by base learners should be as uncorrelated as possible. In order to obtain uncorrelated outputs, neural networks should either be trained using different training datasets, or different architectures. The former idea requires huge amounts of data during training, which may not be available, whereas the latter idea implies the daunting task of experimenting with and finding the optimal set of parameters for a large number of networks. Dropout solves both of the above problems.

The following explanation of dropout follows the description in Srivastava *et al.* (2014). Consider a neural network with r nodes. Note that this network can be seen to consist of 2^r *thinned neural networks*, where each thinned network consists of the neurons that survived dropout. Importantly, the weights in each thinned network are shared by the other thinned networks, therefore the number of weights remain in the order of r^2 at the most. During training, with each input presented to the network, dropout reoccurs. That is, each neuron is temporarily omitted from the network with probability p. This implies that with each input presented, a new thinned network is trained. Fortunately during testing, only a single neural network needs to be fitted. No nodes are dropped and the weights of the network are obtained by reducing the weights from the thinned models by a factor p, thereby obtaining an approximate estimate of the average weights of the thinned models. The reader may refer to Figure 3.3 for an illustration of how dropout affects the connections between neurons.

In Srivastava *et al.* (2014), note that $p = 0.5$ is suggested to work well for hidden nodes, whereas a p value closer to 1 is recommended for visible nodes. In practice the exact optimal value of p depends on the use case and is typically found via experimentation.

In mathematical notation, note that the dropout model may be described

as follows.

...

In summary of this section, dropout removes a neuron from a network with probability p. The neurons that are omitted do not contribute to the forward pass and do not participate in backpropogation. Every time an input is presented, the neural network samples a different set of neurons to be dropped out. This ensures that a neuron does not rely on the signals of a particular set of other neurons, and discourages neurons to co-adapt. Thereby the neural network is forced to learn more robust features that are useful in conjunction with many different random subsets of the other neurons (Krizhevsky *et al.*, 2012).

Furthermore, we have seen that there are parallels to be drawn between dropout and ensembling approaches (Hinton *et al.*, 2012). In each training iteration a unique set of neurons are active, and a unique network is fitted. During testing these different models are combined. This is of course the same paradigm as in ensemble learning.

It has been shown that dropout does tremendously well in guarding against overfitting. Unfortunately however, it slows down the convergence time of training.

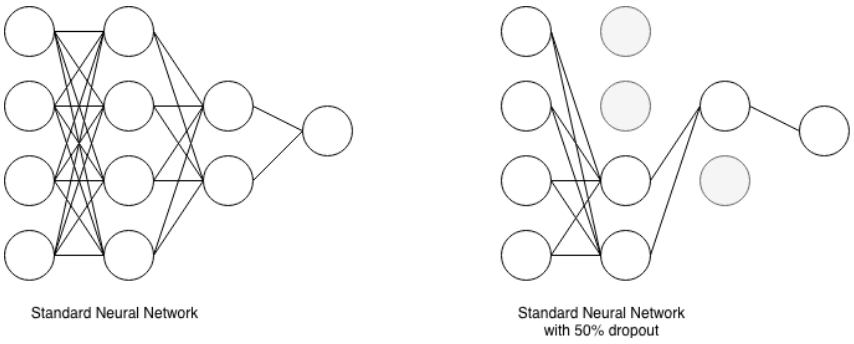


Figure 3.3: The effect that dropout has on connections between neurons.

3.4.2 Data Augmentation

Recall that our aim with predictive models is to generalise well to an unseen test set. In an ideal world we would train a model on all possible variations of the data in order to capture all interactions and relationships. This is of course not possible in the real world. Such a dataset is not available and would

be infinitely large. Of course if it were available, machine learning would be unnecessary, since all possible observations would be available and one could simply use a lookup in order to predict outcomes.

In reality we have a finite subset of the full data distribution to train on. Any new samples with unique feature combinations will likely improve the model's generalisablity. If the collection of new samples is not available, we can try to artificially create these through data augmentation. This is a standard approach followed especially in computer vision applications. For example, a single image can be rotated, flipped horizontally or vertically, shifted in any direction, and cropped. Many other transformations are also possible without destroying the semantic content of the image. By means of such transformations we are able to artificially increase the size of the training set in order to avoid overfitting.

Consider for example Figure 3.4, used to illustrate the way in which a single image of a cat may be converted into eight images through random transformations of the original image. In all of these images, a cat is still recognisable. Of course data augmentation cannot be as effective as observing genuine new data samples. Still it is a very effective and efficient substitute (Perez and Wang, 2017). In fact, it is well known that data augmentation consistently leads to improved generalisation.

Figure 3.4: An example of data augmentation for images.

Data augmentation may be formalised by means of the *Vicinal Risk Minimisation* principle (Chapelle *et al.*, 2001). According to this principle, human knowledge is required to describe a vicinity around each observation in the training data so that artificial examples can be drawn from the vicinity distribution of the training sample in order to enlarge it. In image classification one

may define the vicinity of an image as, for example, the set of its horizontal reflections and minor rotations.

With respect to the types of data augmentations that may be applied, one is typically guided by the dataset and application domain, since all augmentations should preserve the semantic content or the signal in the original observations. For example, making too small crops of an image will ignore the context and may make it impossible to recognise its objects. Moreover, augmentations that are suitable for image data are not necessarily sensible for text data.

3.5 Modern Architectures

In the following sections we highlight some of the recently developed neural network layers and modules. Amongst others, these operations were designed to make training more robust and efficient, to help learn more robust representations, and to be able to model more useful features.

3.5.1 Normalisation

A factor which complicates the training of neural networks is the fact that hidden layers have to adapt to the continuously changing distribution of their inputs. The inputs to each layer are affected by the parameters of all preceding layers, and a small change in a preceding layer can lead to a much bigger difference in output as the depth of the network increases. When the input distribution to a learning system changes, it is said to experience covariate shift (Shimodaira, 2000).

The use of ReLUs, careful weight initialisation, and small learning rates should all help a network to deal with internal covariate shifts. However, a more effective approach is to ensure that the distribution of inputs remains more stable while training the network. For this purpose Ioffe and Szegedy (2015) proposed *batch normalisation* (BN).

A batch normalisation layer normalises its inputs to a fixed mean and variance (similar to the way in which the inputs of the network are normalised). Therefore BN can be applied before any hidden layer in a network in order to prevent internal covariate shift. The addition of BN layers facilitates the use of higher learning rates, thereby dramatically accelerating the training process of deep neural networks. Moreover, implementing batch normalisation assists

with regularisation (Ioffe and Szegedy, 2015), so much so that in some cases its use obviates the need for dropout.

In more detail, the normalising transform over a batch of univariate inputs, $x_1, \ldots, x_n$, where $n < N$ is performed as follows:

1. The mini-batch mean, μ, and variance, σ^2 are obtained:

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$
$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - u)^2$$

2. The inputs are normalised, *i.e.*

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}},$$

   where ϵ is a constant to ensure numerical stability.

3. The output values are scaled and shifted, *i.e.*

$$y_i = \gamma\hat{x}_i + \beta,$$

   where γ and β are the only two learnable parameters in a batch normalisation layer.

The motivation for Step 3 is to allow the layer to represent the identity transform in cases where the normalised inputs are not suitable for the following layer. That is, the scale-and-shift step will reverse the normalisation step if $\gamma = \sqrt{\sigma^2 + \epsilon}$ and $\beta = \mu$. Note that the application of batch normalisation has become standard practice when training deep convolutional NNs.

3.5.2 Skip Connections

A further modern architecture which speeds up training of deep neural networks considerably, is the use of so-called *skip connections* or *shortcut connections*. The invention of skip connections is attributed to He *et al.* (2015*b*), He *et al.* (2016), Huang *et al.* (2016) and Srivastava *et al.* (2015). Note that our discussion is largely based on the exposition given in He *et al.* (2015*b*). The inspiration for skip connections was the occurrence of the *degradation problem*, whereby from a certain point, the addition of hidden layers in deep neural networks

leads to a decrease in their training accuracy. Since any deep neural network can be made shallower by means of setting the transformations in some hidden layers equal to the identity function, intuitively deeper neural networks should always be able to achieve higher training accuracy than shallower networks. Therefore the occurrence of the degradation problem indicates that multiple non-linear transformations in deep neural networks are unable to learn identity mappings.

In He *et al.* (2016), the degradation problem is addressed by rephrasing the learning objective of a deep neural network. Instead of requiring a group of layers to learn some undefined non-linear mapping, say $H(\boldsymbol{x})$, the objective is to learn a *residual function*, *viz.* $F(\boldsymbol{x}) = H(\boldsymbol{x}) - \boldsymbol{x}$.
In their paper He *et al.* (2016) postulate that it is more tractable to find $F(\boldsymbol{x})$ than $H(\boldsymbol{x})$. For example, in the problematic scenario where $H(\boldsymbol{x})$ should be $H(\boldsymbol{x}) = \boldsymbol{x}$, it is easier to truncate $F(\boldsymbol{x}) to zero than for multiple non-linear layers to learn the identity mapping. This is indeed supported by the results of empirical work in
$F(\boldsymbol{x}) + \boldsymbol{x}$.

The above framework is referred to as *deep residual learning*. Residual Networks (ResNets) became very popular after they were used in the winning model of one of the ImageNet competitions (He *et al.*, 2015*a*). It is in these models that skip connections come into play. Very simply, skip connections are additional connections between different layers in NNs that bypass one or more layers of non-linear transformations (Emin and Xaq, 2018). This idea is illustrated in Figure 3.5.

In feedforward neural networks, if skip connections output identity mappings, they can efficiently be used to obtain $H(\boldsymbol{x})$ via $F(\boldsymbol{x}) + \boldsymbol{x}$: the skip connection output $\boldsymbol{x}$ is just added to the $F(\boldsymbol{x})$ output obtained from a set of stacked layers. Note that although here, the skip connection and stacked layer output are combined using addition, other ways such as multiplication or concatenation may also be used.

Multiple benefits to using skip connections have been reported in the literature. They have been shown to alleviate the vanishing-gradient problem, strengthen feature propagation, encourage feature reuse, and may also reduce the number of parameters required (Huang *et al.*, 2016). Furthermore, it is interesting to note that one can draw a parallel between ResNets and boosting methods since both are approaches for fitting models to residuals (Huang *et al.*, 2017).

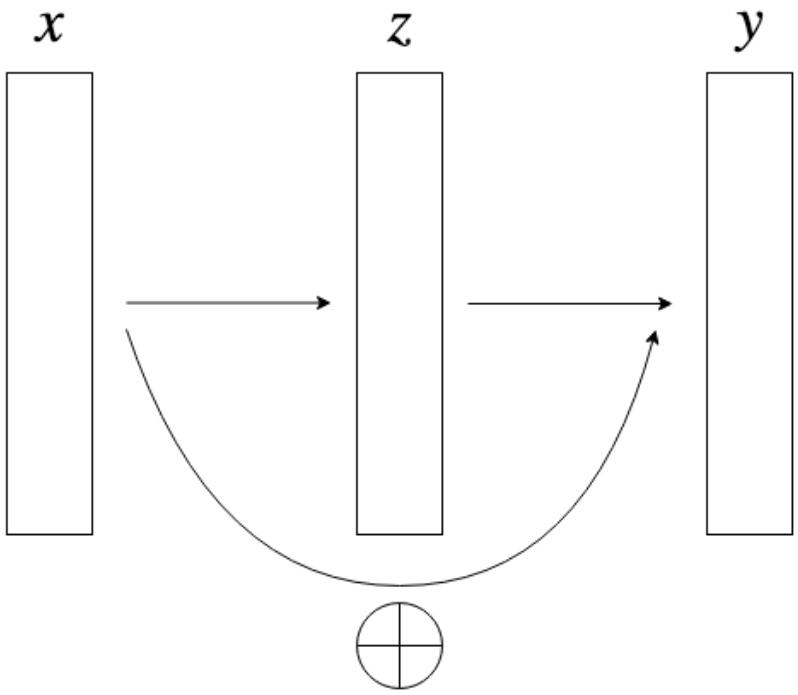


Figure 3.5: Diagram conceptualising a skip connection.

For a final word on skip connections in this chapter, we refer to the remark in Emin and Xaq (2018), *viz.* that a completely satisfactory explanation for the success of skip connections remains elusive. The authors then proceed by proposing a novel explanation for being able to substantially improve the performance of deep neural networks.

3.5.3 Embeddings

An embedding is a layer which maps a discrete input to a numeric vector representation. It was first used in NLP in order to represent words as numbers so that they may be processed by numeric models. For instance the word 'woman' may be represented by the vector $[1, 3, 5]$, and the word 'man' by $[2, 4, 6]$. In finding appropriate embeddings, the objective is to map discrete inputs to a meaningful vector space wherein items with similar meanings are found in close proximity to each other. This is in contrast to using a so-called *one-hot encoded* representation of words, where all words lie equally far apart. The reader may refer to Figure 3.6 for an illustration of such a space[1].

[1]https://www.shanelynn.ie/get-busy-with-word-embeddings-introduction/

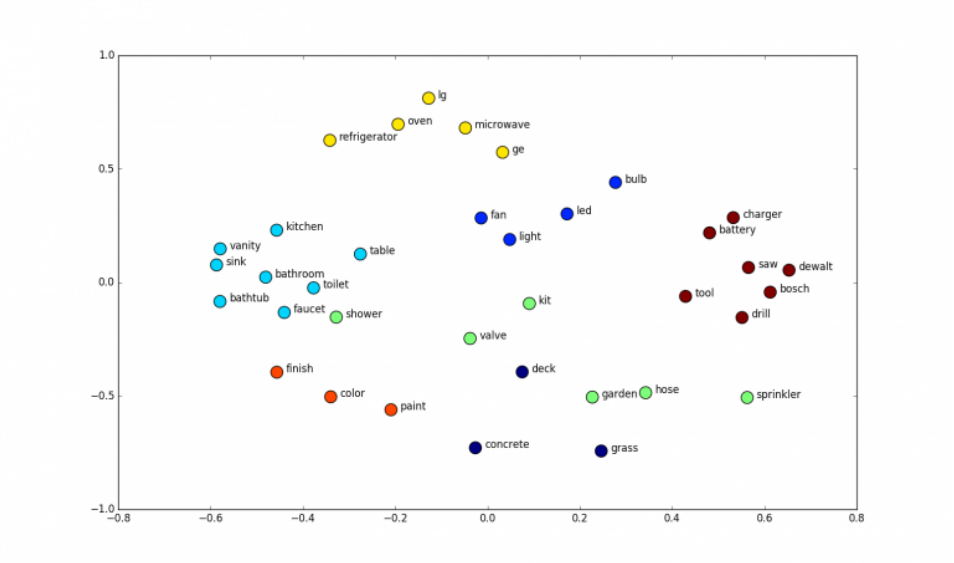


Figure 3.6: Learned word embeddings in a two-dimensional space.

Initially, mappings were configured independenty of neural networks using approaches based upon co-occurences (Mikolov *et al.*, 2013). The real breakthrough came when mappings were defined as learnable layers in the network (Howard and Ruder, 2018, Devlin *et al.* (2018)). Thus embeddings may now be tuned just like any other parameter in the network. Of course the parameters of the embedding function (or layer) first needs to be (randomly) initialised. Thereafter during training, they are tuned along with the rest of the neural network weights.

An embedding operation can either be viewed as a table lookup or a matrix multiplication of the discrete input in a one-hot encoded form, *i.e.* $\boldsymbol{e} = W\boldsymbol{x}$, where $\boldsymbol{x}$ is a discrete input in one-hot encoded form, and where $W : k \times p$ is the matrix containing the embedding, with p the number of discrete categories and k the embedding size. An embedding layer may be re-used by all input features having the same input type, thereby improving efficiency and reducing the memory footprint of the model.

3.5.4 Attention

The incorporation of *attention* modules in networks is one of the standout breakthroughs made in deep learning in recent times. They especially played an integral part in advances in NLP and other sequence related tasks (Vaswani *et al.*, 2017, Devlin *et al.* (2018)). Attention was first popularised in the neural

machine translation field (Bahdanau *et al.*, 2014). Currently it is almost used ubiquitously in NLP applications. Also in computer vision applications, the use of attention modules have been found extremely useful. Examples include image captioning (Xu *et al.*, 2015) and audio processing tasks (Duong *et al.*, 2016).

The main idea of an attention module is to force a layer to only focus on a certain subset of its inputs at different stages of computation. For example, in image captioning, one may use a recurrent neural network (RNN) to sequentially output words describing the image. With the use of an attention module, the network is at each step restricted to only consider certain parts of an image, thereby avoiding to have to consider the full image every time. This is illustrated in Figure 3.7. Notice how the network focuses on the bird part of the image when predicting the word "bird" and "flying", whereas it focuses on the water part of the image when predicting the words "body" and "water". Similarly, when used in machine translation, we may visualise attention weights to capture the way in which a network focuses on a different subset of words when predicting each word in the target language (Figure 3.8).

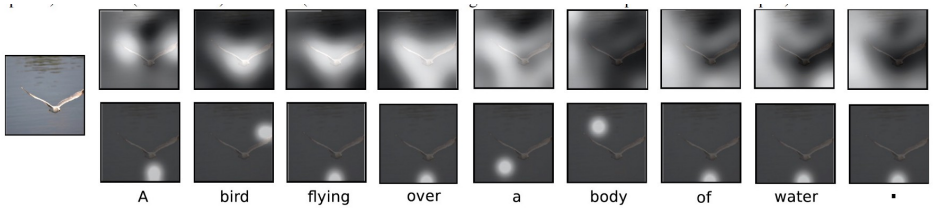


Figure 3.7: Attention applied to image captioning.

An insightful discussion of attention modules may be found in Xu *et al.* (2015). In this section however, a summary of the core of an attention module through the following three equations suffices:

$$\begin{aligned} \boldsymbol{z} &= f(\boldsymbol{x}) \\ \boldsymbol{\alpha} &= \text{softmax}(f_{att}(\boldsymbol{x})) \\ \boldsymbol{y} &= \boldsymbol{z} \otimes \boldsymbol{\alpha}, \end{aligned}$$

where $\boldsymbol{z}$ denotes the ordinary activations produced by a layer f given input $\boldsymbol{x}$. The symbol $\boldsymbol{\alpha}$ represent the weights produced by the attention layer, f_{att}, after a logit transform to ensure the weights to sum to 1. The output of the attention module, $\boldsymbol{y}$, is then obtained through an elementwise multiplication $\otimes$ of $\boldsymbol{z}$ and

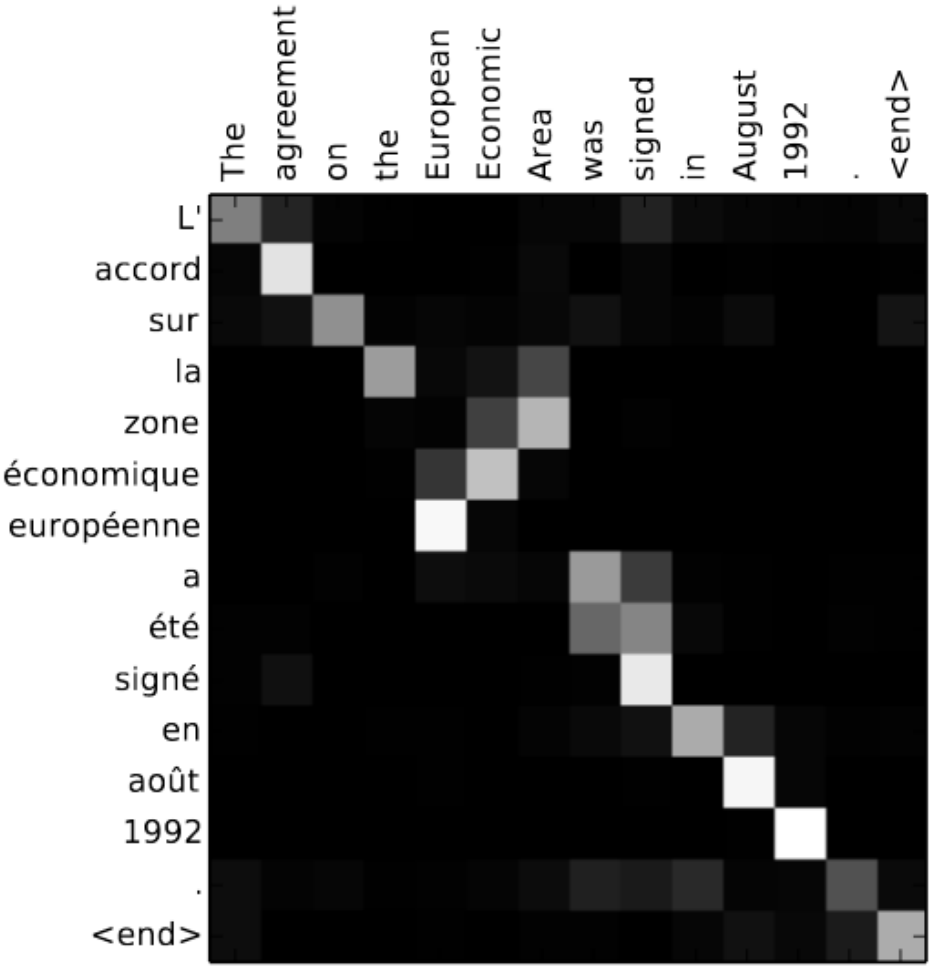


Figure 3.8: Attention applied to machine translation.

$\boldsymbol{\alpha}$, although it should be noted that alternative combinatorial operations may also be used.

Finally in this section, note that we elaborate on the concepts of *self-attention* (Cheng *et al.*, 2016*b*) and *multi-head attention* (Vaswani *et al.*, 2017) in Section 4.3.1.

3.6 Super-Convergence

Specifying the hyperparameters to be used in learning algorithms is a difficult process. It requires expertise, typically involves extensive trial-and-error and is often described as more of an art form than a science. Moreover, in NNs there are many hyperparameters that need to be considered: thus far in the

thesis we have encountered learning rate, batch size, momentum and weight decay. Although there are no easy ways to find them, appropriate parameter values do have a huge effect on training time and on the performance of neural networks. A grid search or random search in the parameter space seems not to be an option because of its computational expense.

In this section we summarise the work on structuring specification of NN hyperparameters presented in the papers by Smith in 2015 and 2018, and by Smith and Topin in 2017. Through these approaches, the necessity of running complete grid or random searches is eliminated, rendering tremendous improvements in training time and accuracy.

In **?**, the author considers specification of the learning rate parameter. This is a worthwhile enterprise: in Chapter 2 the importance of the appropriately specifying the learning rate became evident. To re-emphasise, the learning rate is viewed as the hyperparameter whose appropriate specification is most important compared to all other neural network hyperparameter values. We saw that one may choose to keep its value fixed throughout training, or to allow it to decrease from a certain point in training onwards. The latter was the preferred approach until *cyclical learning rates* (CLR) were proposed in Smith (2015).

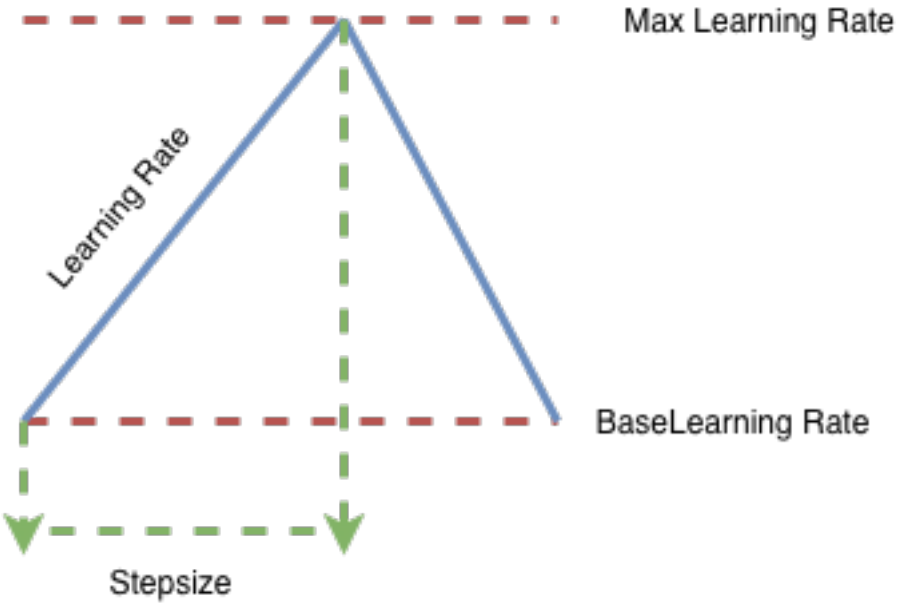
In CLR, as training of a network progresses, the learning rate is varied cyclically between two boundary values. A cycle consists of two steps, *viz.* a stage when the learning rate increases to a pre-specified maximum value, and a stage where the learning rate decreases to a chosen minimum value. Different cyclical functions of the learning rate were experimented with. Since they performed very similarly, the simplest function, *viz.* the so-called *triangular learning rate policy* was adapted. In a triangular function, the learning rate increases linearly up to the maximum, and then decreases linearly back to the minimum. Note that during training, many cycles of learning rates are traversed.

In order to implement cyclical learning rates, one needs to specify the number of epochs during each stage (also known as the step size), as well as the minimum and maximum learning rates to be attained. In Smith (2015), a simple method for finding reasonable learning rate boundaries is provided. In this learning rate range test, training starts with a small learning rate which is then slowly increased in a linearly fashion throughout a pre-training run. Typically with an increase in the learning rate, the training loss decreases until

a point where the network converges. After this point the learning rate becomes too large, causing the loss to start increasing. Hence this is the largest learning rate one should consider using during training. In CLR it is suggested that the lower bound of the learning rate be set to a factor of 3 or 4 times less than the maximum learning rate. For an update of the CLR proposal, the reader is referred to Smith and Topin (2017).

Importantly, during application of the learning rate range test towards fitting ResNets to specific image datasets, **?** discovered that the test loss remains constant up to very large values for the learning rate. This surprising phenomenon is called *super-convergence*.

The implication of super-convergence is that in some setups one may simply use a single learning rate cycle together with unusually large learning rate values in order to train neural networks an order of magnitude faster than when using standard training methods. Whether or not super-convergence may occur in a specific network architecture may be verified using the learning rate range test, similar to the way in which the super-convergence discovery was made. If super-convergence is possible, Smith and Topin (2017) suggest a slight modification of CLR, *viz.* to "use a single cycle that is smaller than the total number of iterations/epochs and to allow the learning rate to decrease several orders of magnitude less than the initial learning rate for the remaining iterations". Note that the initial learning rate is an abnormally large value. The unusually large learning rate used leads to an additional benefit, *viz.* the facilitation of regularisation. The above learning rate methodology is called *1cycle*, and its learning rate setting is illustrated in **??**.



Clearly also in the 1cycle regime, bounds for the learning rate need to be set. For this purpose the learning rate range test may be used, but with the lower bound of the learning rate set to a factor of about 10 times less than the maximum learning rate. Figure 3.9 illustrates an example of output obtained from the learning rate range test and of how to determine the learning rate bounds to be used in the 1cycle policy.

Note that one should be careful of specifying a too small step size since this increases the rate by which the learning rate parameter increases, which in turn might render the training process unstable.

Next consider Figure 3.10, taken from Smith and Topin (2017). Here the test loss over each training iteration in the 1cycle methodology is compared to that of a fixed learning rate policy.

From Figure 3.10 it is clear that using the proposed method, the model achieves improved accuracy compared to the standard approach. This is done in an eigth of the number of training iterations required in the original framework.

In his most recent work, Smith (2018) considers a more comprehensive methodology which recognises the interdependence of specification of the learning rate, batch size and other regularisation techniques such as weight decay. An important remark in this paper is that contributions to the amount of regularisation on a network should be balanced across the various hyperparameter values. For example, since large learning rates also act as a form of regularisation (Smith, 2015), if one uses a large learning rate in the 1cycle

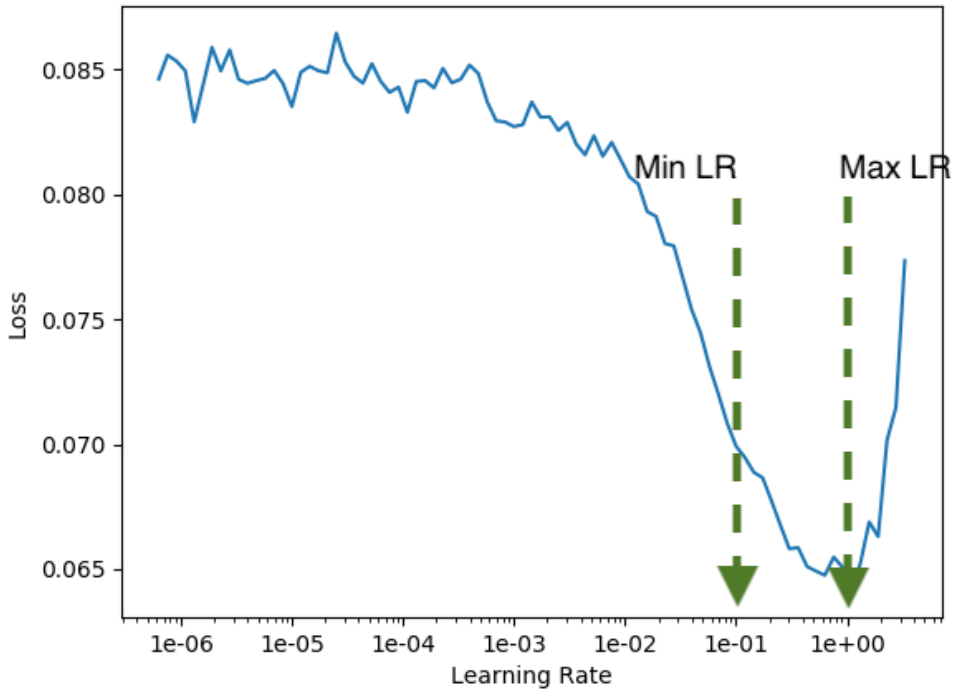


Figure 3.9: An example output of a learning rate range test.

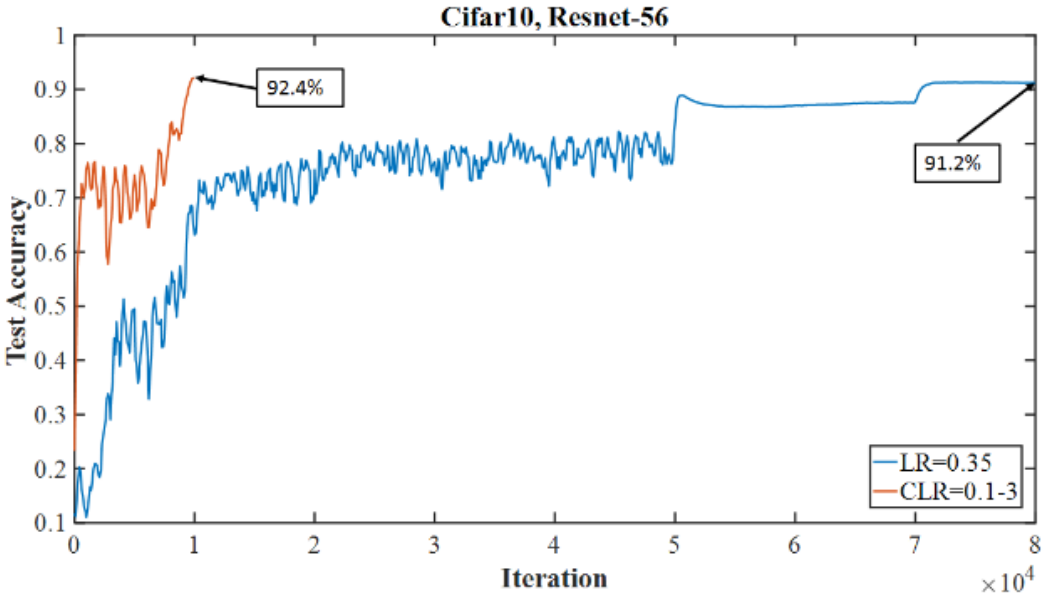


Figure 3.10: Reduced training iterations and improved performance facilitated by the super-convergence principle.

regime, one would have to reduce some of the other regularisation controls.

In the 2018 paper, specification of batch size and of weight decay is also considered. It is shown that larger batch sizes allow for training with larger learning rates, thus convergence can be reached quicker. Therefore when setting the batch size, the recommendation is to select the largest possible size that fits into memory.

The weight decay parameter is another important value that needs to be properly specified during the 1cycle policy. Since it is a regularisation parameter it should be in balance with the learning rate. The proposed way of setting the weight decay is by means of a grid search. A few short training runs are performed at different weight decay values. The validation loss function is then inspected for any hints with regard to a preferred choice. Alternatively, various weight decay values may be used during the learning rate range test and their behaviour compared. The preferred weight decay value is the one which produces the most stable loss and which allows the use of the largest learning rate. Note that in our experiments we follow the abobve suggestions in order to tune the hyperparameter values specific to each different model.

3.7 Model Interpretation

Although deep learning has become the state-of-the-art approach in many machine learning tasks, it is still trailing behind other algorithms in terms of model interpretability. This is however is not an unusual occurrence: in statistical learning the trade-off between prediction performance and interpretability is well known. Deep neural networks are occasionally referred to as "black boxes" since it is very difficult to interpret what is going on inside the stacks of linear and non-linear layers. This is one of the biggest criticisms against the deep learning field, and often a stumbling block for deployment in production environments. For example, in the clinical domain, model transparency is of utmost importance, given that predictions might be used to affect real-world medical decision-making and patient treatments (Shickel *et al.*, 2017). Fortunately some work has been done in order to enable one to gain insight from neural networks. This topic is briefly discussed in the following two sections.

3.7.1 Neural Network Specific

We have showed in §2.6 that it is possible to inspect activations and weights of layers at different levels of a neural network. If the network is small, one might gain insight into what the network has learned, or into why it is making certain decisions. However, most useful neural networks are at least three layers deep, rendering its activations and weights more complex to interpret.

When fully convolutional networks are used, there are ways to visualise which parts of the inputs are important in making certain decisions. These visualisations are called *class activation maps* (Zhou *et al.*, 2016; Selvaraju *et al.*, 2017). They can unfortunately only be used with fully convolutional networks, and not with fully connected layers.

Another common interpretation tool in order to gain insight into what specific neurons are looking for, is to rank inputs by the magnitude of their activations. Then, if it is possible to spot similarities between the highest ranked inputs, this provides a potential description of patterns triggering neurons. The above approach is called *activation maximisation* (Erhan *et al.*, 2009). Since it relies on human inspection, activation maximisation is typically infeasible in the face of many neurons to investigate.

An interesting take on model interpretation is described in Frosst and Hinton (2017) and Che *et al.* (2016). Their ideas are based on the process of *knowledge distillation* (Hinton *et al.*, 2015) and leverages the fact that decision trees are easier to interpret. Knowledge distillation entails the process of transferring knowledge from one model (or an ensemble of models) to another model by means of training the target model to estimate the predictions of the source model (or ensemble). Frosst and Hinton (2017) and Che *et al.* (2016) respectively use a soft-decision tree and boosted trees to learn the mapping between the inputs and the neural network predictions. Once the tree based methods have been trained, the usual intepretation tools of tree based models, like feature importance or evaluating the way in which a sample traverses the tree, may be used in an attempt to understand the neural network. It is interesting that in both publications the authors note that tree based models trained on the neural network predictions achieved improved performances over the ones trained using the actual targets. This is indicative of the value knowledge distillation is able to add.

3.7.2 Model Agnostic

Besides interpretation tools specifically designed for neural networks, one may of course also make use of model agnostic tools, *i.e.* methods that may be used in combination with any machine learning model. The so-called *permutation importance algorithm* is such a tool, computing feature importance scores corresponding to each input feature. The importance measure of a feature is determined by the sensitivity of that feature to random permutations of its values. The expectation is that when a feature with a strong signal is shuffled before input to a model, the performance of the model will drop significantly. On the other hand, if a feature has little effect on the target predictions, shuffling its values will not exhibit a major effect on its performance.

Permutation importance was first introduced by Breiman (2001) in the random forest algorithm, but may be generalised to other models. The steps for calculating feature importances are provided below:

1. Train a neural network on a dataset with p input features.
2. Evaluate the network on a validation set in order to obtain a performance metric m_0.
3. For each of the p input features:
    - Create a copy of the validation set and randomly shuffle the feature in this copy.
    - Evaluate the neural network on this version of the validation set, thereby obtaining m_j, $j = 1, \ldots, p$.
4. Rank the features based upon $m_j - m_0$ (if a bigger m is preferred).

Unfortunately note that permutation importance only produces sensical results if the assumption of independent features holds. Permuting features independently creates examples that never occurr in real life and the importance of features in that invalid space may be misleading. The test however can still be useful to identify inputs that are not important, *i.e.* features that are not used by the model. If randomly permuting a feature does not affect the model performance at all, it may be a good indication that the model does not depend on that feature.

A second example of a interpretation tool that is independent of the particular model fitted, is the use of *partial dependence plots*. These graphs may be used to visualise relationships between a target and input features (Friedman,

2001). Once a neural network is trained, we may evaluate the effect of a change in any input feature on a single prediction by observing the change in the prediction. Suppose for example we evaluate the way in which a feature X_1 influences predictions. By taking a single observation from the data we can evaluate how the model prediction changes by means of changing the value of X_1 to other possible values of X_1. And since this behaviour will most likely vary for different observations, the above process should be repeated for a subset of observations from the dataset. The average effect on the predictions at different values of X_1 can subsequently be obtained, along with standard errors of these effects.

For more model agnostic interpretation tools (for instance the use of so-called *SHAP values*), the reader is referred to **?**. Note also that examples of implementation of these model interpretation techniques may be found in the next two chapters.

Chapter 4

Deep Learning for Tabular Data

4.1 Introduction

In Chapters 2 and 3 we covered the basics of neural networks, as well as the more recent advances in deep learning. The aim of this chapter is to explore ways in which the modern deep learning approaches in Chapter 3 may be leveraged in the application of deep learning for tabular data (DLTD). In Chapter 1 we alluded to the differences of tabular data compared to unstructured data (such as images, and data used in text and speech applications). The widely acclaimed successes of deep learning typically occur in areas such as image classification, machine translation and speech recognition. However, in the literature only a handful of publications report successful implementation of deep learning for tabular data. In these papers the applications include recommender systems (Haldar *et al.*, 2018); click-through rate prediction[1] (Song *et al.*, 2018); analysis of electronic health records (Rajkomar *et al.*, 2018); and transport related problems (de Brébisson *et al.*, 2015). Not much research has been done in the area of deep learning for tabular data, therefore it is often unclear how to solve certain modelling challenges. Hence the tabular data domain is still dominated by tree-based models such as random forests and gradient boosted trees. This begs the question as to why deep learning is not nearly as effective here as it is in most other data domains. The aim of this chapter is to help

[1]To predict the probability of a user clicking an an item, which is of critical interest in online applications.

illuminate this issue, and to indicate promising directions towards improving current state-of-the-art performances.

The structure of the chapter is based upon the challenges that occur when using deep learning for tabular datasets, as identified and described in §1.3. For each challenge, we reconsider the issue, review the literature to discuss current methodology, and (where possible) provide suggestions towards improving these approaches. We start in §4.2 by considering ways to represent input features in tabular data. §4.3 is devoted to approaches that are used to leverage feature interactions. A large part of this chapter is devoted to methodology which facilitates sample efficiency; we discuss in detail in §4.4. This is followed by a brief discussion of ways to interpret deep neural networks for tabular data in §4.5. In the final section, the 1cycle policy and hyperparameter selection in DLTD are discussed, in addition to a selection of miscellaneous topics that do not fit into the above framework.

4.2 Input Representation

One of the key design considerations when constructing a deep neural network for tabular data is the input representation, *i.e.* the way in which one should numerically represent each feature. This choice may heavily influence the ability of the neural net to extract patterns from the input, as well as optimisation efficiency during training. This is a more difficult decision in the case of tabular data, since here features are typically highly heterogeneous (Shavitt and Segal, 2018). A representation may be optimal for some features, but not for others, and we want to ensure that no feature dominates the others during training. Moreover, a tabular dataset typically contains both continuous and categorical features, where different approaches are needed to process each. Tabular dataset are often high-dimensional and very sparse. This is a scenario in which the adverse effects of improper input representations is magnified, as noted by many (Song *et al.*, 2018, Wang *et al.* (2017), Qu *et al.* (2016), Cheng *et al.* (2016*a*), Anonymous (2019), Covington *et al.* (2016)). An example of an extremely high-dimensional and sparse tabular dataset is the so-called Criteo dataset[2]. Its feature dimension is ~30 million, with a sparsity of ~99%.

[2]https://www.kaggle.com/c/criteo-display-ad-challenge

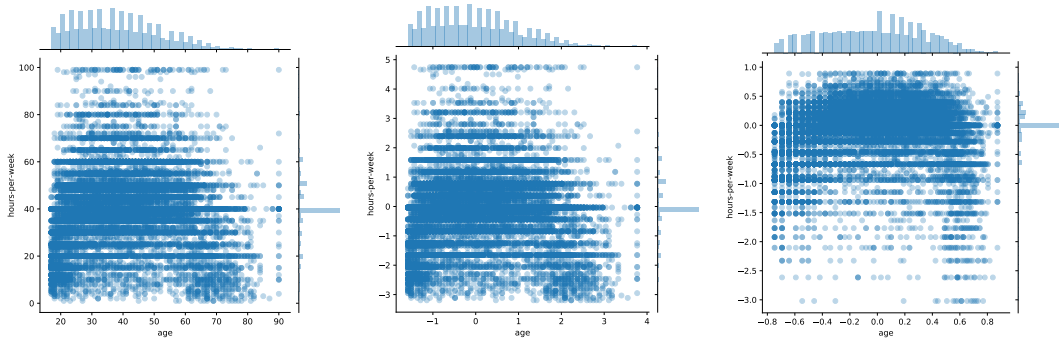
4.2.1 Numerical Features

A major advantage of tree-based methods is that the scale and distribution of features hardly matter. The only requirement is that their relative ordering should be meaningful. With neural networks we are not that fortunate. Neural networks are very sensitive to the scale and distribution of its inputs (Ioffe and Szegedy, 2015). If features are measure on different scales, a single feature might dominate the weight updates. In this case, if a feature contains a large value, it may throw off the optimisation procedure, thereby causing gradients to 'explode' or to 'vanish' (Clevert *et al.*, 2015). The implication is that proper standardisation of all continuous features in tabular data is mandatory.

The optimal standardisation in DLTD varies between datasets. Hence the only way to know for sure which normalisation to apply to numeric features is by means of experimentation. The typical standardisation approach for numerical features in deep learning is to do mean centering and variance scaling, *i.e.* $\tilde{x} = (x - \mu)/\sigma$, where $\mu$and σ are the mean and standard deviation of X respectively ($x \in X$). One would expect this transformation to also be sufficient for tabular data, but in practice it has often been found otherwise.

In (Haldar *et al.*, 2018) the authors propose first inspecting the distribution of each feature. If a feature seems to follow a normal distribution, standard normalisation, *viz.* $(x - \mu)/\sigma$, may be performed. However if a feature seems to approximately follow a power law distribution, it should rather be transformed via $\log\left((1 + x)/(1 + \text{median})\right)$. The above mapping ensures that the bulk of the values lie within {-1,1}, and that the median feature value is close to zero. Consider for example the effect of this transformation on two continuous features (*viz.* Age and Hours-per-week) in the Adult dataset. This is illustrated in Figure 4.1. We see in (a) that the features have roughly the same scale, but their distributions are totally different; and in (b) that after applying mean centering and unit variance scaling, the values of the Hours-per-week feature mostly lie in {-1,1}, however many Age values remain outside {-1,1}. Consequently, we apply the power distribution transformation in (Haldar *et al.*, 2018), and observe that all Age values now lie within {-1,1}. Of course the downside of the above approach is that it involves a manual process and very cumbersome in high-dimensional data setups.

With a view to reduce potential high variances exhibited by numeric features, (Song *et al.*, 2018) suggests transforming a numeric feature via $\log^2(x)$ if $x > 2$. This was successful in their use-case, but it is hard to imagine that this solution



(a) Original (b) Gaussian Norm (c) Power Norm

Figure 4.1: The effect of normalisation on continuous variables.

will generalise to many other applications. Note that the Song *et al.* (2018) transformation causes a discontinuity at $x = 2$, as well as a possible overlap between values that were originally less than 2 with those that were greater than 2. In addition, it does not take care of extreme values on the negative side. (Wang *et al.*, 2017) simply use the standard log transform, *i.e.* $\log x$, to normalise continuous features.

Covington *et al.* (2016)] also report appropriate normalisation of numeric features critical for DLTD to converge. Their approach is to transform numeric features to be equally distributed in [0,1). This is done using the cumulative distribution $\tilde{x} = \int_{\infty}^{x} df$, where f is the distribution of x. The integral is approximated using linear interpolation on the quantiles of the feature values computed as a preprocessing step.

An option for numeric feature normalisation which we have not yet encountered in the DLTD literature, is to specify the initial layer to numeric features to be a batch normalisation layer. This will affect the same type of scaling as in the case of zero mean and unit variance transformation, but the transformation parameters are learned from each batch. Hereby the need to preprocess numeric features is removed. The caveat is that the transformation quality depends on the batch statistics. That is, in cases where the batch is not representative of the full data distribution (which is likely if the batch size is small), the training procedure might be negatively affected.

4.2.2 Categorical Features

Most of the sparsity in tabular datasets is induced by categorical features. Since neural networks cannot process discrete categories or objects, we need to find a numeric representation for each class. The standard approach is use to one-hot encoded categorical features. That is, the one-hot encoded form corresponding to a categorical feature with (say) three levels, is $[1, 0, 0]$ for Level 1; $[0, 1, 0]$ for Level 2; and $[0, 0, 1]$ for Category 3.

Multiple inefficiencies occur when using one-hot encodings in neural networks. Clearly its use introduces sparsity to the data: the dimension of the one-hot encoded form is equal to the number or categories in a feature. Thus if a dataset consist of many high-cardinality features, the data will be extremely sparse and difficult to model. If not handled properly, sparse data may easily cause neural networks to overfit (Covington *et al.*, 2016). The presence of categorical features with many levels also increases size requirements of the first linear NN layer, which in turn creates a need for more computing power and memory.

The other problem with one-hot encodings of categorical features is that there is no notion of similarity and distances between categories. In this representation, all categories lies equally far apart, now matter how semantically similar or dissimilar they are. This makes it harder for the model to learn useful patterns.

An alternative to one-hot encodings as representations of categorical features for neural networks are *entity embeddings*. An entity embedding is the exact same operation as (word) embeddings we have discussed in §3.5.3, but applied to categories instead of words. Therefore an entity embedding assigns a numeric vector representation to each category in a categorical feature, for example: Category 1: $[0.05, -0.1, 0.2]$, Category 2: $[0.2, 0.01, 0.3]$ and $[-0.1, -0.2, 0.05]$ for Category 3. Once all of the categorical features have been embedded, their representations can be concatenated and passed to the rest of the network.

The first published work in modern times on entity embeddings was in the taxi destination prediction challenge (de Brébisson *et al.*, 2015). In another Kaggle sucess story, (Guo and Berkhahn, 2016) succesfully used entity embeddings for predicting the total sales of a store. Companies like Instacart and Pinterest have reported the effective use of entity embeddings on their internal datasets. Currently, all research on deep learning for tabular data makes use of entity embeddings - see for example (Song *et al.*, 2018), (Wang *et al.*, 2017),

(Covington *et al.*, 2016) and (Zhou *et al.*, 2017).

The reason why it is used all-round is because it does not have the same issues as the one-hot encoded representations. A similar formulation to §3.5.3, we define the embedding for the j-th categorical feature by:

$$\boldsymbol{e}_j = V_j \boldsymbol{x}_j$$

where $\boldsymbol{x}_j$ is the one-hot encoded vector representation of the j-th categorical variable and V_j is the associated embedding/weight matrix. The weights in V_j are learned along with all of the other parameters in the network.

The first advantage of entity embeddings is that it speeds up training and reduces the memory footprint which further improves the generalisation ability of the network (Covington *et al.*, 2016), (Guo and Berkhahn, 2016). This especially useful when working with dimensional and sparse inputs. Suppose we have a dataset with two categorical features, X_1 and X_2, with cardinality of C_1 and C_2, respectively. Furthermore, suppose that the first hidden layer in the neural network accepts inputs of size q and thus we need project an observation with these two features into a vector representation of the same size. If we were to use one-hot encoded representations of X_1 and X_2 we would need a weight matrix of size $(C_1 + C_2) \times q$. However, if we use entity embeddings of X_1 and X_2 we may have two weight matrices of sizes $C_1 \times q/2$ and $C_2 \times q/2$, which is in total half the number of parameters needed compared to the pure one-hot encoded representation.

The size of the embeddings is a hyperparamter of the model and again there is no way to tell what this value should be beforehand. Most publications rely on a grid search to find the optimal sizes. For example (Song *et al.*, 2018) experimented with embedding sizes [8,16,24,32] and found 16 to work the best, and (Cheng *et al.*, 2016*a*) found that an embedding size of 32 was optimal for their use-case. The selection complete depends on the data and the network used.

(Wang *et al.*, 2017) and (de Brébisson *et al.*, 2015) used different embedding sizes for each categorical features and suggested these rules-of-thumb:

- $6 \times (\text{cardinality})^{\frac{1}{4}}$ (Wang *et al.*, 2017)
- $\max(2, \min(\text{cardinality}/2, 50))$ (de Brébisson *et al.*, 2015)

It makes sense to have different embedding sizes for categorical features with different complexities.

Entity embedding not only reduces memory usage and speeds up neural networks compared to one-hot encoding, but more importantly, by mapping similar values close to each other in the embedding space it reveals the intrinsic properties of the categorical variables, which one cannot obtain with one-hot encoding. This allows us to interpret the classes of the categorical features. The embeddings can be visualised to gain further insight into the data and model decision making. The weights associated with each category's projection onto the embedding space can be plotted with any dimension reduction technique like t-sne or PCA. Then we can compare the categories based on their relative distances and positions in this reduced space. In Figure 4.2 we plot a 2 component PCA of the embedding matrix of the Education categorical feature in the adult dataset. We see that the school categories all lie in the bottom-right corner of the space, with some notion of ranking from grade 5 (top-right) to grade 12 (bottom-left). The tertiary education classes are in a separate cluster and their levels of education correspond conveniently with the vertical axis.

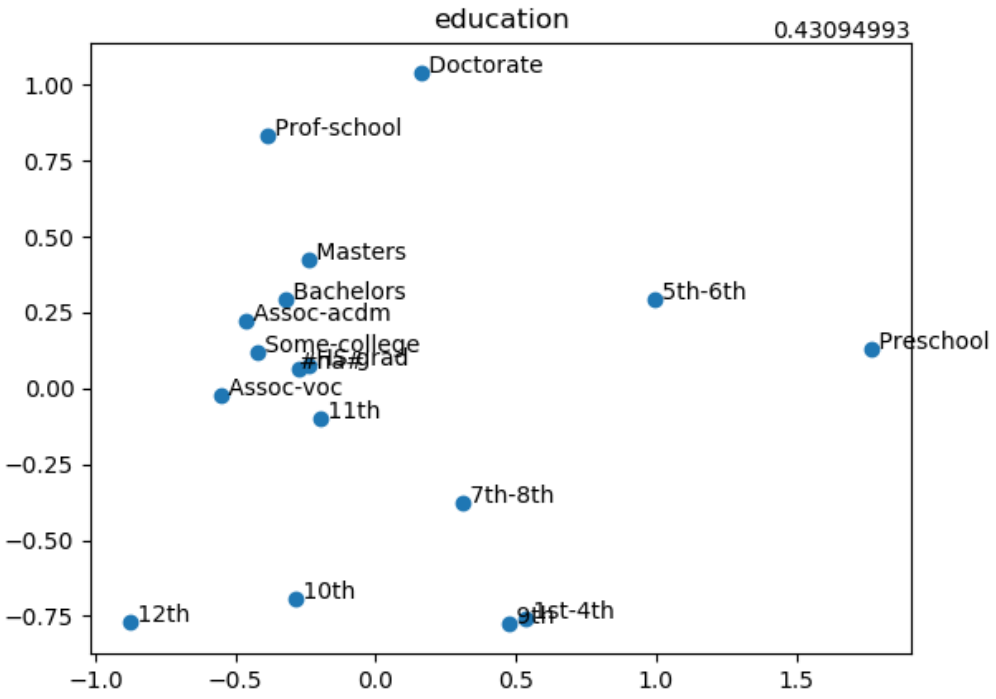


Figure 4.2: PCA of the Education entity embedding weight matrix.

To prove that these entity embeddings actually learns something useful,

besides plotting the embedding matrix, one can also feed them along with the continuous features to other learning algorithms and see how it affects performance. (Guo and Berkhahn, 2016) found that the embeddings obtained from the trained neural network boosted the performance of all tested machine learning methods considerably when used as the input features. These embeddings can be reused on different machine learning tasks and do not have to be re-learned for each dataset. Instacart and Pinterest, referenced above, reported sucessful implementation of this approach.

The entity embedding approach is very flexible. One can reuse an embedding over different categorical features if the features have overlapping categories. (Zhou *et al.*, 2017) has an intersting take on multi-hot categorical features; where a feature can have more than category associated with it. The embedding layer for that instance then outputs a list of embeddings with length the same as the number of categories associated with that instance and feature. The list of embeddings then gets projected back into a fixed-length representation by doing a pooling operation.

4.2.3 Combining Features

Once the continuous and categorical features have been processed and embedded, we need a way to combine them before passing it to the rest of the network. The standard approach is to concatenate each categorical variable embedding to the continuous variables, as was done in (Haldar *et al.*, 2018) and (Wang *et al.*, 2017) for example, illustrated in Figure 4.3. The potential problem with this approach is that some features might be over-represented in this vector. For example, one of the continuous features may be very important for prediction, but it gets lost when concatenated with all the entity embeddings each taking up more space in the combined representation.

In (Song *et al.*, 2018) they embed both the numerical and categorical features into the same sized embedding. By mapping both types in the same feature space facilitates more effective learning of interactions between the mixed features. The embedding for the j-th numerical features is obtained by:

$$\boldsymbol{e}_j = \boldsymbol{v}_j x_j$$

where x_j is a scalar and $\boldsymbol{v}_j$ is the associated weight vector.

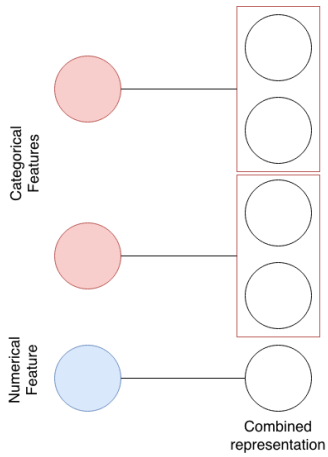


Figure 4.3: Combined representation of continuous and categorical features.

4.3 Learning Feature Interactions

In most machine learning tasks the greatest performance gains can be achieved by feature engineering wheras better algorithms may only result in incremental boosts. In feature egineering one strives to create new features from the original features based on some domain knowledge of the data or otherwise, that makes it easier for the model to estimate the target and help capture high-order interactions between features. Although a crucial step to make the most out of the data, this can be a very laborious process. It is widely held that 80% of the effort in an analytic model is preprocessing, merging, customizing, and cleaning datasets, not analysing them for insights (Rajkomar *et al.*, 2018). There is no formal path to follow in this stage and thus usually consists of many a trial and error, benefitted by domain knowledge of the data, which is not always accessible. A huge advantage of using neural networks on tabular data (and other data structures) is that the feature engineering process gets automated to some extent (with the caveat of numerical feature preprocessing.). A neural network learns these optimal feature transformations and interactions implicitly during the training process. The hidden layers of a neural network can be viewed as a feauture extractor that was optimised to map the inputs into the best possible feature space for a model (the final layer of the network) to operate in.

The standard approach is to stack a few fully-connected layers one after the other to map the input representation to the output, as was done in (Covington *et al.*, 2016). Fully-connected layers model all feature interactions implicitly and

in ideal world, we would expect this architecture to be sufficient. In practice, with out current learning algorithms, a simple MLP is not good enough to learn all types of interactions. A fully connected model structure leads to very complex optimisation hyperplanes with a high risk of falling into local optimums.

Therefore it is necessary to explicitly leverage expressive feature combinations or encourage the network to learn better high-order feature interactions. This issue receives attention in publications such as (Song *et al.*, 2018), (Wang *et al.*, 2017), (Qu *et al.*, 2016) and (Guo *et al.*, 2017). The restrictions we impose on the fully-connected structure may further help to limit the model size to make learning more efficient. They key question we asked in this section is how do we help the network to determine which features to combine to form meaningful high-order features. We first briefly review some of the suggestions in the literature, after which we focus on the attention mechanism which we believe is the most powerful method of learning feature interactions.

The authors of (Wang *et al.*, 2017) makes a case for finding a bounded-degree of feature interactions, arguing that all the Kaggle competions are won with feature engineering of low-degree interactions, whereas deep neural networks learn highly non-linear interactions implicitly. They suggest using an automated way of building cross-features, called the cross-network. In the cross-network each layer produces higher-order interactions based on existing ones, and keeps the interactions from previous layers. The cross-network consists of cross-layers that can be formalised as:

$$\boldsymbol{x}_{l+1} = \boldsymbol{x}_0 \boldsymbol{x}_l^\intercal \boldsymbol{w}_l + \boldsymbol{b}_l + \boldsymbol{x}_l$$

where $\boldsymbol{x}_l$ is the output of the l-th cross layer and the input to the $(l+1)$-th cross-layer; $\boldsymbol{x}_0$ is the combined input representation; $\boldsymbol{w}_l$ and $\boldsymbol{b}_l$ are its associate weight and bias paramters respectively. Each cross-layer adds back its input after feature in the same fashion as a skip-connection. The degree of the cross-features grows with cross-network depth. The authors experimented with 1-6 cross-layers and found that a depth of 6 gave the best results.

In parallel to the cross-network, they also used a standard deep neural network to learn the highly non-linear feature interactions. The DNN accepts the same input and its output is then concatenated with that of the cross-network. These two networks can then be trained simultaneously.

(Qu *et al.*, 2016) used something called a product layer, which takes pairwise inner or outer products of all feature combinations and concatenates it to all linear combinations. The output is then fed to 2 fully-connected layers.

According to (Guo *et al.*, 2017) and (Cheng *et al.*, 2016*a*) it is necessary to capture both low and high-order interactions. They both achieve this by having two parallel networks, similar to (Wang *et al.*, 2017), where one learns high-order interactions and the other low-order interactions. Both use a deep neural network to learn the high-order interatcions. (Cheng *et al.*, 2016*a*) uses a shallow but wide neural network to capture the low order interactions and where (Guo *et al.*, 2017) uses a learnable factorisation machine to do that. Similar to the other parallel stream networks, the output of the two streams gets concatenated before passing it to the classification layer.

4.3.1 Attention

Based on the results in the paper (Song *et al.*, 2018) and by our findings in general deep learning research §3.5.4, we deem attention to be the most promsing mechanism to model feature interactions. (Song *et al.*, 2018) uses a multi-head self-attention mechanism which they call the interacting layer. Within in the interacting layer each feature is allowed to interact with every other feature and automatically determine which of those interactions are relevant to the output.

To explain the attention mechanism, consider feature j and suppose we want to determine which high-order features involving feature j are meaningful. We first define the correlation between features j and k under attenion head h as:

$$\alpha_{j,k}^{(h)} = \frac{\exp\left(\phi^{(h)}(\boldsymbol{e}_j, \boldsymbol{e}_k)\right)}{\sum_{l=1}^{L} \exp\left(\phi^{(h)}(\boldsymbol{e}_j, \boldsymbol{e}_l)\right)}$$

where $\boldsymbol{e}_l$,$l = 1, \ldots, L$ is the embedding of the l-th features. $\phi^{(h)}(.,.)$ is an attention function which defines the similarity between two features. It can be defined by a trainable layer or a simple inner product as in:

$$\phi^{(h)}(\boldsymbol{e}_j, \boldsymbol{e}_k) = \left\langle W_{\text{query}}^{(h)} \boldsymbol{e}_j, W_{\text{key}}^{(h)} \boldsymbol{e}_k \right\rangle$$

where $W_{\text{query}}^{(h)}$ and $W_{\text{key}}^{(h)}$ are transformation matrices which map the original embedding space into new spaces of the same dimension. The representation of feature j in subspace h is then updated by combining all relevant features

guided by the coefficients $\alpha_{j,k}^{(h)}$:

$$\tilde{\boldsymbol{e}}_j^{(h)} = \sum_{k=1}^{K} \alpha_{j,k}^{(h)} W_{\text{value}}^{(h)} \boldsymbol{e}_k$$

$\tilde{\boldsymbol{e}}_j^{(h)}$ is a combination of feature j and its relevant features under attention head h. Note that $\alpha_{j,k}^{(h)}$ for $k = 1, \ldots, K$ sum to 1 since they went to a logit transform/softmax operation.

$\tilde{\boldsymbol{e}}_j^{(h)}$ is thus a learned combinatorial feature. Since a feature can be involved in various different combinations, we use multiple heads to extract the multiple combinations, *i.e.* $\{\tilde{\boldsymbol{e}}_j^{(h)}\}_{h=1}^{H}$. In the original paper (Song *et al.*, 2018) used $H = 2$, but this is typically a hyperparemeter one needs to tune. All of these combinatorial features are concatenated into a single vector, $\tilde{\boldsymbol{e}}_j$. Then finally the output is combined with its raw input and sent through a ReLU activation:

$$\boldsymbol{e}_j^{\text{res}} = \text{ReLU}\left(\tilde{\boldsymbol{e}}_j + W_{\text{res}} \boldsymbol{e}_j\right)$$

This mapping from $\boldsymbol{e}_j$ to $\boldsymbol{e}_j^{\text{res}}$ is done for each feature to form the interacting layer. The activations of the interacting layer is thus a representation of the high-order features of its inputs. These interacting layers can be stacked on-top of each other to form arbitrary order combinatorial features.

(Zhou *et al.*, 2017) follows a vaguely similar idea to attention by learning the weights to apply to the hidden representations. Interestingly, they removed the softmax layer as a way to mimick probabilities to reserve the intensity of activations.

In terms of the skip-connection already mentioned a couple of times above - both (Song *et al.*, 2018) and (Wang *et al.*, 2017) found by experimentation that by having skip-connections to connect lower-level features with higher-level ones improves the performance of the network.

4.3.2 Self-Normalising Neural Networks

By studying the literature of deep neural networks for tabular data we rarely see the optimal network depth go beyond three of four layers. The reason is that a fully connected model have very complex optimisation hyperplanes which increases the risk of falling into bad local optimums. A proposed way of training deeper neural networks is to make use of *Self-Normalising Neural Networks* (Klambauer *et al.*, 2017). They were developed as an alternative to

batchnorm layers since batchnorm layers often become unstable when using SGD or stochastic regularisation techniques like dropout, especially when using fully-connected neural networks. This is exhibited by a high-variance in the training error.

The self-normalising neural network is simply a neural network with a novel activation function called the *SeLU*. The SeLU helps the network to maintain zero mean and unit variance for the activations at all network levels. The SeLU activation eliminates the need for a BatchNorm layer and is also much safer against exploding or vanishing gradients. It is defined as:

$$\text{selu}(x) = \lambda \begin{cases} x & \text{if} \quad x > 0 \\ \alpha e^x - \alpha & \text{if} \quad x \leq 0 \end{cases}$$

where $\lambda = 1.0507$ and $\alpha = 1.6733$ are special constants derived in the paper. When using the SeLU activation function, one should be aware of the special weight initialisation and dropout technique that should be used in tandem.

(Klambauer *et al.*, 2017) tested SeLUs on 121 classification datasets from the UCI Machine Learning repository. They compared DNNs with SeLU activations to other DNNs and other classifiers like Random Forests and SVMs. They found that on the datasets with less than 1000 obervations, random forests and SVMs performed the best. However, for the datasets with more than 1000 observations, DNNs with SeLU activations performed the best overall. The classifiers were compared by ranking them by their accuracy for each prediction task and doing a pairwise Wilcoxon test. Another thing the authors found when comparing SeLUs with other activations is that the model selection approach for DNNs with SeLUs resulted in much deeper networks than DNNs with other activation functions.

These are all promising results but we have not seen SeLUs been used in other applications or papers on tabular datasets.

4.4 Sample Efficiency

It is well know that deep neural networks require a large amout of data to generalise well. Typically, tabular datasets are not as large as unstructured datasets like images and texts. There is also no large tabular dataset from which knowledge can be transferred, like an ImageNet for computer vision

or a Wikipedia corpora for NLP equivalent. We suggest two techniques for overcoming this problem: data augmentaion and unsupervised pre-training.

4.4.1 Data Augmentation

Data augmentation for tabular datasets is rarely studied. Tabular data is very different to image data and the standard augmentations used in computer vision does not make sense with tabular data. You cannot rotate or scale an observation from a tabular data without losing its meaning. One transformation that does make sense for tabular input is the injection of random noise.

When working with images, we can randomly perturb the pixel intensities by a small amount so that it is still possible to make sense of its content. By adding 1 for example to all pixels and all colors in an image, will only make it slightly brighter and we will still be able to make sense of it. But with tabular data we can just randomly add a small amount to any feature. The input features will probably not all be on the same scale and the addition of noise might result in a feature value that is not within the true distribution. In addition, it does not make sense to add anything to a discrete variables. Thus in order to inject random noise to a tabular data sample, the noise should be scaled relative to each input feature range and the results should be a valid value for that feature. This also helps the model to be more robust to small variations in the data. (Van Der Maaten *et al.*, 2013) suggests an augmentation approach that does this called *Marginalised Corrupted Features* (MCF). The MCF approach adds noise to input from some known distribution. The process is manual and it cannot be applied to discrete variables.

In the original DAE paper (Vincent *et al.*, 2008), they used a blank-out corruption procedure. Which is randomly selecting a subset of the input features and masking their values with a zero. This similar to dropout regularisation but instead applied to the inputs. The only conceptual problem with this approach is that for some features a zero value actually carries some meaning, so a suggestion is to blank-out features with a unique value not already belonging to that feature distribution.

Another input corruption approach shown to work empirically here is what is a technique called *Swap Noise* (Kosar and Scott, 2018). The swap noise procedure corrupts inputs by randomly swapping input values with those of other samples in the datasets. We have illustrated the approach in Table 4.1.

Table 4.1: Swap Noise Example.

age	occupation	education	race	sex	>=50k
Original Dataset					
49	NA	Assoc-acdm	White	Female	1
44	Exec-managerial	Masters	White	Male	1
38	NA	HS-grad	Black	Female	0
38	Prof-specialty	Prof-school	Asian-Pac-Islander	Male	1
42	Other-service	7th-8th	Black	Female	0
20	Handlers-cleaners	HS-grad	White	Male	0
Sample with swap noise					
49	Prof-specialty	Prof-school	Asian-Pac-Islander	Female	1

In this way you ensure that the corrupted input have at least valid feature values. But it still might produce combinations of features that are not actually possible according to the true data distribution.

The final data augmentation we look at is *mixup augmentation* (Zhang *et al.*, 2017). The way mixup creates artifical samples is by the following formulation:

$$\tilde{\boldsymbol{x}} = \lambda \boldsymbol{x}_i + (1-\lambda)\boldsymbol{x}_j \tilde{\boldsymbol{y}} = \lambda \boldsymbol{y}_i + (1-\lambda)\boldsymbol{y}_j$$

where $\boldsymbol{x}$ is a input vector,$\boldsymbol{y}$ a one-hot encoded output vector and $\lambda \in [0, 1]$. $(\boldsymbol{x}_i, \boldsymbol{y}_i)$ and $(\boldsymbol{x}_j, \boldsymbol{y}_j)$ are two samples drawn at random from the training data. Thus mixup assumes that linear interpolations of input vectors lead to linear interpolations of corresponding targets. We visualise the creation of artificial samples through mixup on a toy dataset in Figure 4.4.

λ controls the strength of the interpolation between the input-output pairs. The closer λ is to 0 or 1, the closer the artificial sample will be to an actual training sample. The authors suggest using $\lambda \sim \text{Beta}(\alpha, \alpha)$ for $\alpha \in (0, \infty)$, since this makes it more likely that the mixed-up sample lie closer to one of the original samples than in the middle of the two. They observed best performance when $\alpha \in [0.1, 0.4]$ and if α is too high, they experience underfitting.

Other ablation studies they did was to find at which stages of the network the interpolation should happen, *e.g.* raw input, after embedding, hidden layer, *etc.* But the experiments are not extremely clear and therefore warrants further discussion here.

Typically data augmentation procedures are dataset dependent and therefore requires expert knowledge. It is hard to think of ways to augment tabular data,

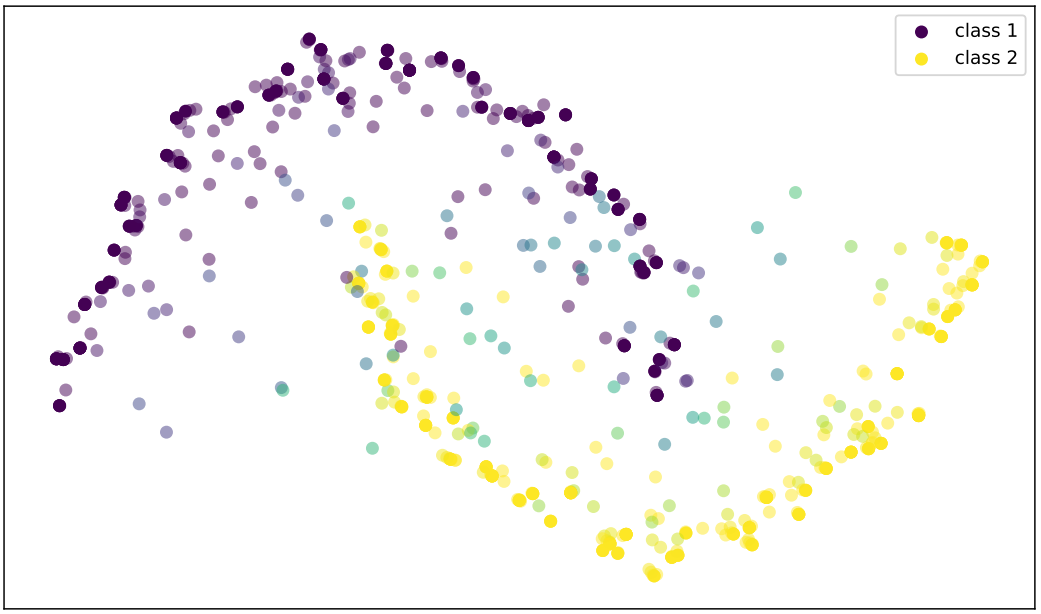


Figure 4.4: Illustration of points created by mixup augmentation.

even more so a generic way of doing so. However, from this defnition it is clear that mixup can be used on any type of data, including tabular datasets.

Mixup data augmentation can be understood as a mechanism to encourage the model to behave linearly in-between training samples. (Zhang *et al.*, 2017) shows that this linear behaviour reduces the amount of undesirable variation when predicting new samples further away from the training samples. They also argue and show empirically how training with mixup is more stable in terms of model predicions and gradient norms. This is because mixup leads to decision boundaries that transition linearly between classes, resulting in smoother predictions.

The authors tested mixup data augmentation on tabular datasets. They tested it on six classification datasets from the UCI Machine Learning repository. They used a 2-layer MLP with 128 neurons each and a batch size of 16. They found that mixup improved the performance on four out of the six datasets.

All of these methods have hyperparameters that needs to be set. For example, with swap noise and blank-out we must set the proportion of feature values to be corrupted and with mixup we need to set the mixup weight. These parameters control the strength of the regularisation and thus should be tuned accordingly.

4.4.2 Unsupervised Pretraining

We described unsupervised pretraining in . It is a very effective way of training deep neural networks. It also addressed the feature engineering issue we face with tabular data by automatically learning useful feature combinations in an unsupervised fashion. However, we only found two published papers using this approach for tabular data, (Zhang *et al.*, 2016) and (Miotto *et al.*, 2016). (Miotto *et al.*, 2016) used a stacked denoising autoencoder as a feature learning method to derive a general-purpose patient representation for EHR data that facilitates clinical predictive modelling. The trained DAE is the used as a fixed feature extractor to obtain representations of the data that can be passed to supervised learning algorithms. (Zhang *et al.*, 2016) also did pretraining with DAEs.

Feature learniattempts to overcome limitations of supervised feature space definition by automatically identifying patterns and dependencies in the data to learn a compact and general representation that make it easier to automatically extract useful information when building predictive models (Miotto *et al.*, 2016). These techniques are very familiar and effective in text, audio and image processing, but not with tabular data.

The DAEs enforce the representations to be robust to partially destroyed inputs. By using these learned representations as input, significantly improved the performance of predictive models compared to those only using the raw inputs.

There are a lot of unknowns when it comes to using DAEs for pretraining. First we need to decide on the structural hyperparameters of the network (number of layers, sizes, etc.) and then we need to decide what learning paramters to choose. This is not as thoroughly explored as hyperparamter selection for supervised neural networks. We also need to decide which loss function we need to used for the DAE. This choice depends on the type of the input. If all the inputs are numeric we can use the common MSE loss, but if the input has categorical features, this decision needs more consideration. (Miotto *et al.*, 2016) transformed all numeric features to be in the range of [0,1] and one-hot encoded the categorical features so that they can use the binary cross-entropy as the loss function. But we cannot follow this approach if we wish to use entity embeddings for the categorical features. A suggestion is to have a separate multiclass cross-entropy loss for each categorical feature and use the MSE loss for the continuous features. These lossed should then

be combined to obtain a total loss. How to weight the different losses is also something that is not yet researched.

We suggest treating the unsupervised pretraining rather as a transfer learning problem than a feature extraction method like in (Zhang *et al.*, 2016) and (Miotto *et al.*, 2016). That is, slicing the head of the trained DAE and replacing it with the required classification layers. Then first train only the new layers keeping the lower level DAE layers fixed and then train all of the layers simultaneously. By doing it this way, the "feature extractor" part of the DAE can still be tweaked to be optimal for the supervised learning task. We experiment with this approach in the next chapter.

4.4.3 Regularisation

Since regularisation can also help one to be more sample efficient we have a brief discussion about it here. A popular form of regularisation in the papers we have studied is dropout (Song *et al.*, 2018), (Zhang *et al.*, 2016), (Qu *et al.*, 2016), (Guo *et al.*, 2017). The dropout percentage was mostly determined using a grid search and the results showed the optimal percentage depends on the data and the model being used.

Interesetingly, (Haldar *et al.*, 2018) found that dropout was not effective in their application. They pinned it down to dropout producing invalid input scenarious that distracted the model. Therefore they opted for hand crafted noise shapes taking into account the distribution of the relevant feature.

The other form of regularisation used for tabular data is weight decay (Song *et al.*, 2018), (Wang *et al.*, 2017), (Zhang *et al.*, 2016), (Qu *et al.*, 2016) and (Zhou *et al.*, 2017), (Zhang *et al.*, 2016) compared dropout with L2 and found dropout to be better.

For a different approach to regularisation, recally the difference between tabular data and unstructured data. A difference between the two data types that stand out is the relative importance of each of the important features with respect to the target. In computer vision a large amount of pixels should change before an image is of something else. Whereas in tabular data a very small change in a single feature may have totally different behaviour with respect to the target (Shavitt and Segal, 2018). The same authors mention that this can be addressed by including a separate requlatisation term for each of the weights in the network. These regularisation terms are seen as

additional model hyperparameters. It is easy to see that this approach is totally intractable since the only way to train these hyperpararmeters are brute force and repetitive tweaking and validatig which cannot be done in by a gradient based method. A workaround is to make these regularisation parameters trainable like all of the other points in the network. This is achieved by minimising the counterfactual loss, a novel loss function proposed by (Shavitt and Segal, 2018). They found that training neural networks by optimising the counterfactual loss, outperform other regularisation approaches for neural networks and results in neural networkss that are comparable in performance to gradient boosted trees.

4.5 Interpretation

We have mentioned that model explainability is important for various reasons. It helps one to know how to improve one's model or where it goes wrong. One can even go so far as to say if you cannot explain how a prediction is made, you cannot know how good it is. In the literature we focussed on for this work, not much attention was given to model interpretability. Although there are a few extra interpretation options avaible when using some of the abovementioned approaches to tabular data.

Firstly, if one uses entity embeddings, the embeddings can be visualised once projected into a low dimensional space, as done in Figure 4.2. (Zhou *et al.*, 2017) is an example in the literature that did this. The attention layer can also provide insight into which feature combinations are important. (Song *et al.*, 2018) and (Zhou *et al.*, 2017) made use of this information. The information is obtained from the attentional weights α which is based on feature similarity and can indicate which features are comined to make predicions for certain observations. Example output in (Song *et al.*, 2018) from this approach can be found in Figure 4.5. Also see (Shavitt and Segal, 2018) for how the learned regularisation parameters were used to evaluate feature importance.

(Haldar *et al.*, 2018) used one of the model agnostic model interpretation algorithms. They used the permuation importance algorithm to evaluate the importance of features. However, they came to the same conclusion as what we stated in §3.7.2. That is that the permutation test did not produce sensical results because the features were not independent. Permuting the feature independently created examples that never occurred in real life, and

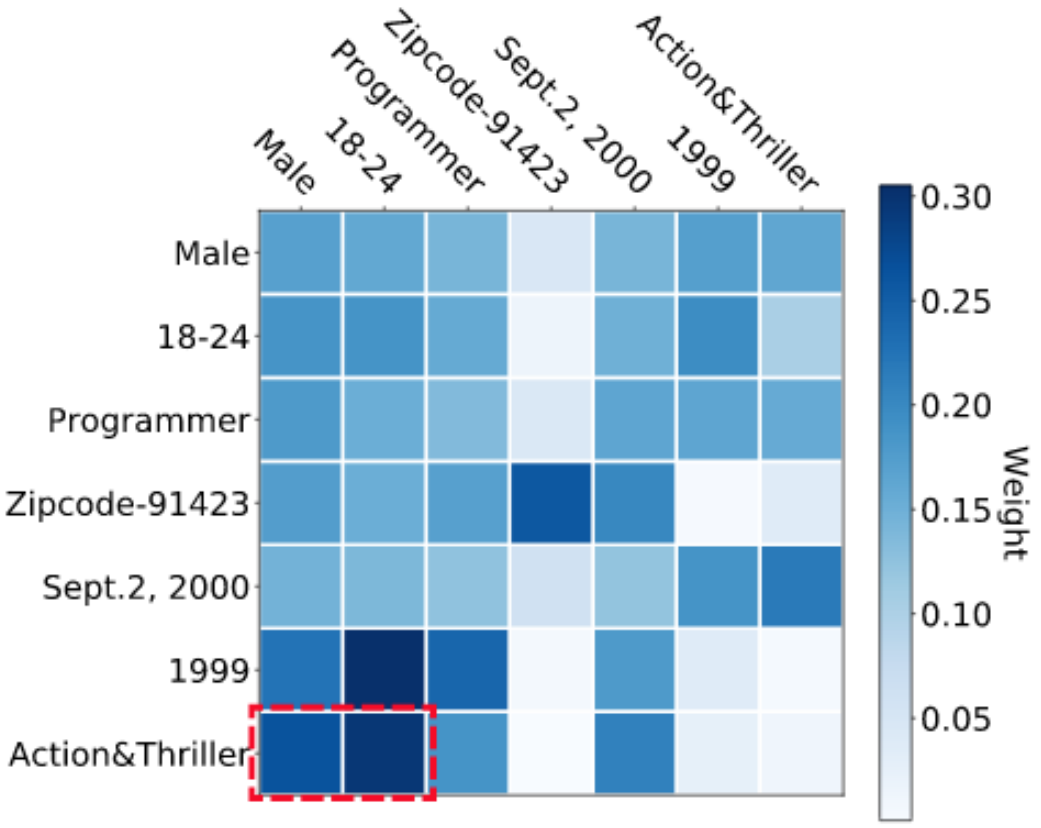


Figure 4.5: The attention weights visualised for a singe observation in the dataset.

the importance of features in that invalid space misled them. They did find it somewhat useful to identify redundant features.

We implemented an example of a permutation importance plot for a trained neural network on the Adult dataset. We shuffled each feature 5 times to get a more robust estimate and to obtain a standard error and we measured the performance in terms of the log loss. The resulting plot is in Figure 4.6. The standard error are too small to show on the figure. The horizontal axis measures the average drop in the log loss when the feature is shuffled before prediction. The ranking of the features can be justified intuitively. It is reassuring to see that sex, race and native-country are some of the less important features when it comes to predicting the income of an indivdual.

Next we compare the permutation importance with the feature importance obtained from knowledge distillation to a boosted trees algorithm (see §3.7.1). To recap the process, we use the predictions of the neural network trained

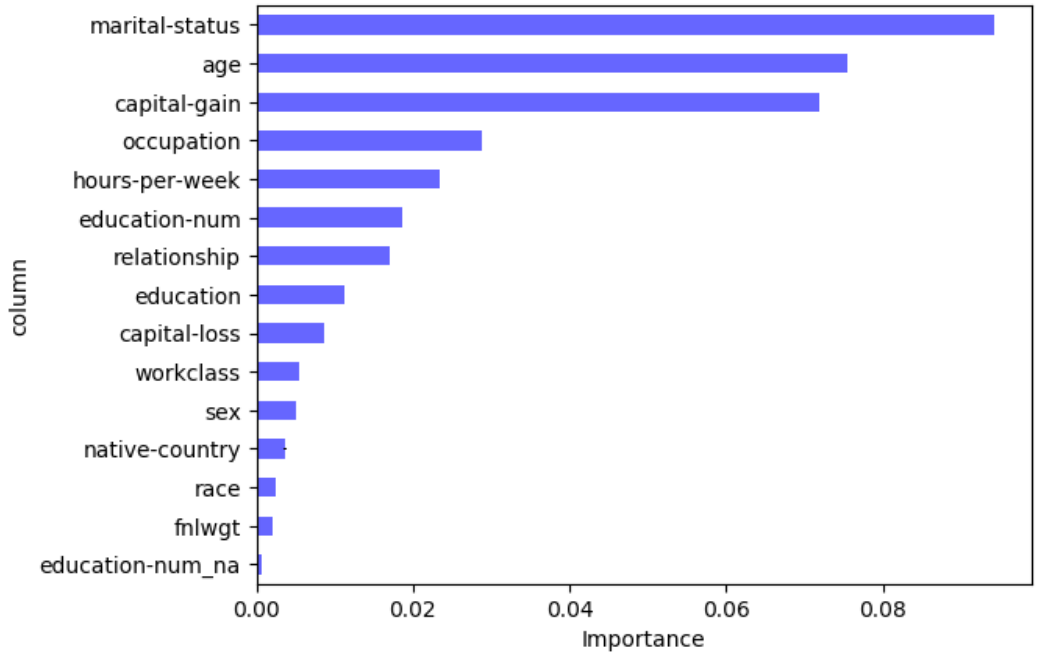


Figure 4.6: Permutation importance plot of a neural network trained on the Adult dataset.

on the Adult dataset as the target for a boosting model to predict using the same input as that given to the neural network. Once the boosting model is trained on these targets, we can extract the feature importance based on the number of times the base trees used a certain feature to split on. The results are displayed in Figure 4.7.

The ranking of the variables between the two approaches are somewhat the same, except for the marital-status and fnlwgt features. This difference might firstly be caused by the permutation importance method not picking up on multivariate feature interactions and thus incorrectly thinks that marital-status is an important feature, but in reality the performance drop was actually caused by another feature dependent on marital-status. The other explanation we can give is that the decision trees are more likely to split on features with more splitting points and since the fnlwgt feature is the feature with most unique values in the dataset, it contributed to the reason why it was considered so important for the boosting model. These two examples show that one should take care when interpretting models and should use more than one source of information when possible.

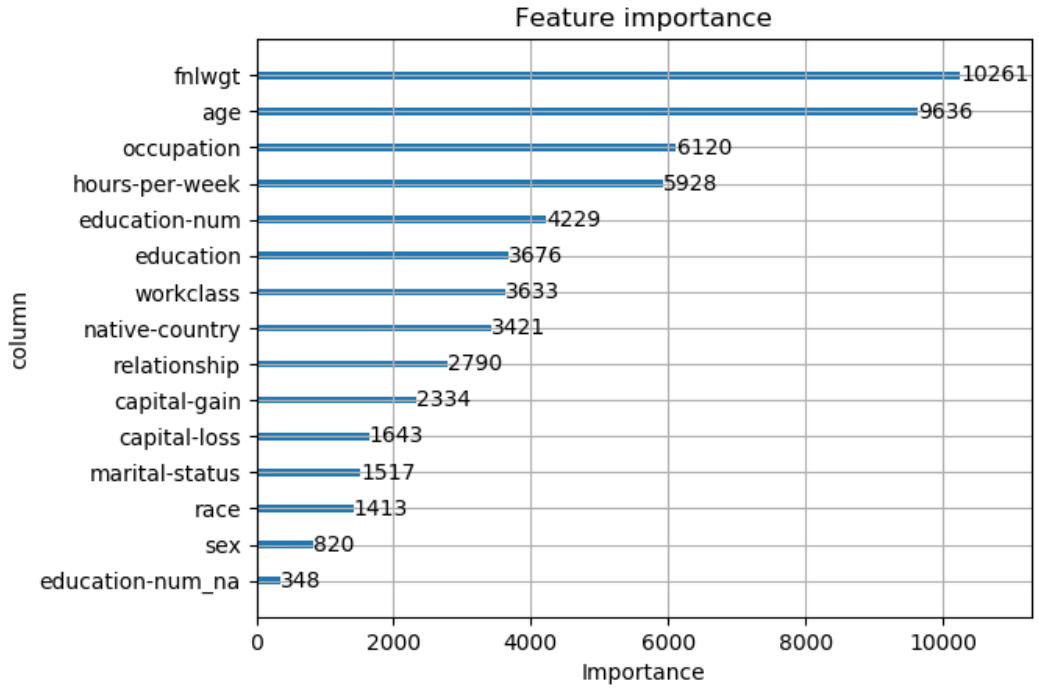


Figure 4.7: Feature importance obtained from a boosted trees model trained on the neural network predictions.

4.6 Hyperparameter Selection

We have already identified a plethora of hyperparameters to be tuned if we want to sucessfully fit a deep neural network to tabular data. Broadly speaking, there are structural hyperparamters to select, like the number of layers and the type of activation function, but then there are also learning algorithm hyperparameters, like the learning rate or the weight decay. In term of the structural parameters, since there are no shared patterns among the diverse tabular datasets, it is hard to design a universal architecture that will fit all. In most of the literature we have investigated the researcher performs a grid search over many combinations of of hyperparamters (Song *et al.*, 2018), (Wang *et al.*, 2017), (Zhang *et al.*, 2016), (Qu *et al.*, 2016), (Guo *et al.*, 2017), (Covington *et al.*, 2016). Most of these parameters are very dependent on the dataset and other modeling choices and therefore the need to tune them. The main structural hyperparameters tuned in the majority of the papers are: hidden layer size, number of hidden layers, activation functions and the shape of the fully-connected layers (*i.e.* is it a constant shape, increasing, decreasing or a diamond shape). The values found varied accross publications, showing again

that the step is necessary when working with custom datasets and models. We include some similar experiments in Appendix B

In terms of the learning algorithm hyperparamters, not much experimentation was shown in the publication itself. The only choice that were shown was selecting the learning rate over a grid of values, as in (Zhang *et al.*, 2016) and (Wang *et al.*, 2017), and kept the selection constant during training. Most of the work used the Adam optimiser and early stopping to prevent overfitting (Song *et al.*, 2018), (Wang *et al.*, 2017), (Zhang *et al.*, 2016). Large batch sizes were chosen in (Song *et al.*, 2018), 1024 and (Wang *et al.*, 2017), 512.

Clearly, the 1Cycle policy and the concept of superconvergence has not yet been tested in the tabular data setting. Therefore we will test its effectiveness here using the Adult dataset. We compare it against a constant learning rate. The evaluation method is described in chapter 5. We first do a learning rate range test and find that the optimal learning rate bounds for the one-cycle policy is 0.01 and 0.1. In our experiment we compare training a neural network with a fixed learning rate at the lower bound, 0.01, a neural network with a fixed learning rate at the upper bound, 0.1 and a neural network with a 1Cycle learning rate schedule in these bounds. The results are displayed in Figure 4.8. From the green line, we see that when training with too large of a (constant) learning rate, the validation loss struggles to converge. Training with the smaller constant learning rate (blue line) works better in this case, but the losses plateau quite early on and shows no sign of improving. The 1Cycle learning rate update policy (orange line) shows the best validation loss and accuracy of the three methods. This proves that the 1Cycle policy is also effective when working with tabular data.

Since the weight decay plays and important part in the super-convergence phenomena, we do an experiment to find the optimal value and to see how it influences the performance. By the suggestion of (Smith, 2018) we try weight decay values for 10^{-3}, 10^{-5} and 0. First we do a learning rate range test with the different weight decays to identify the best value. The results are displayed in Figure 4.9. From the range test it seems that a weight decay of 10^{-5} gives the best performance. To check that this result holds during a full training run, we do an experiment to compare the full training runs with the different weight decays. The results are displayed in Figure 4.10. From the results it shows that the learning rate range test is a good indicator of the optimal weight decay, since in the full training run, the model with weight decay of 10^{-5} performed

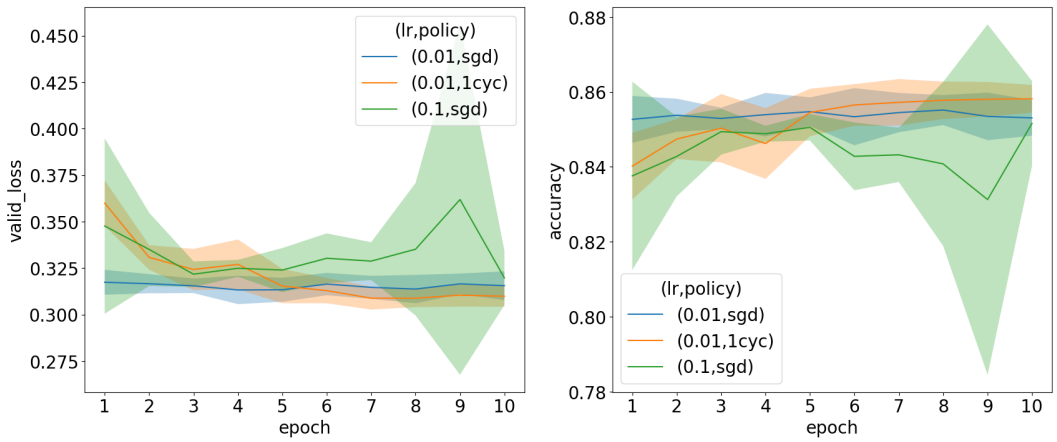


Figure 4.8: Constant learning rate vs the 1Cycle schedule.

the best. Note, that the differences between the training runs are small, and thus if we were to choose one of the other weight decays for this task, we would have only performed slightly worse.

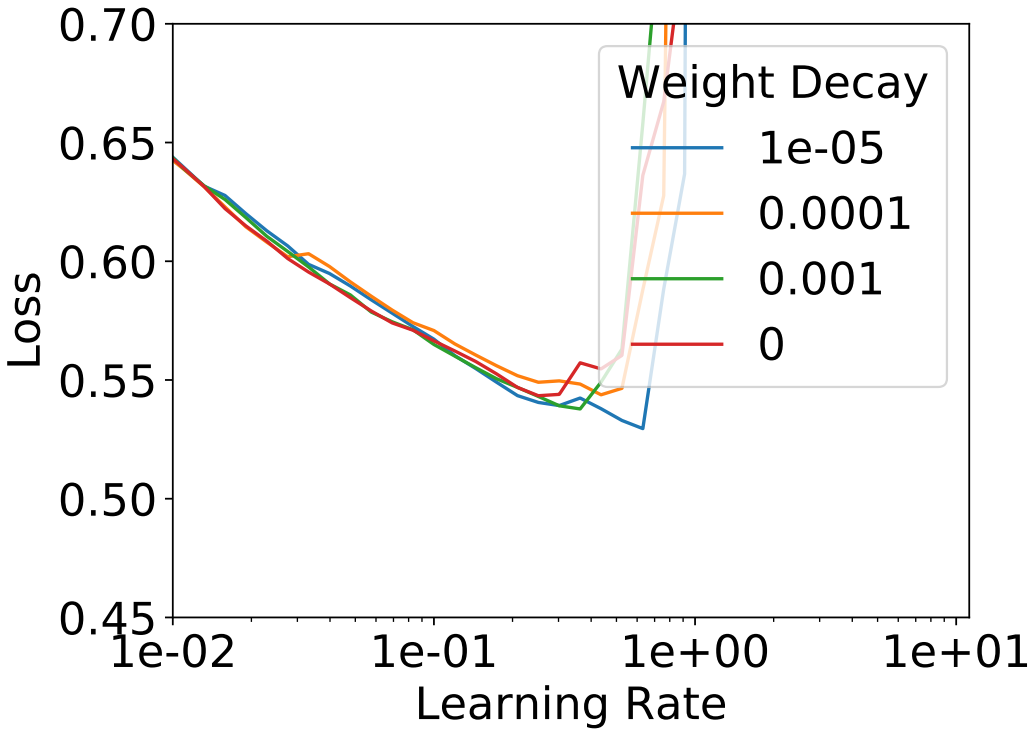


Figure 4.9: A learning rate range test with different weight decays.

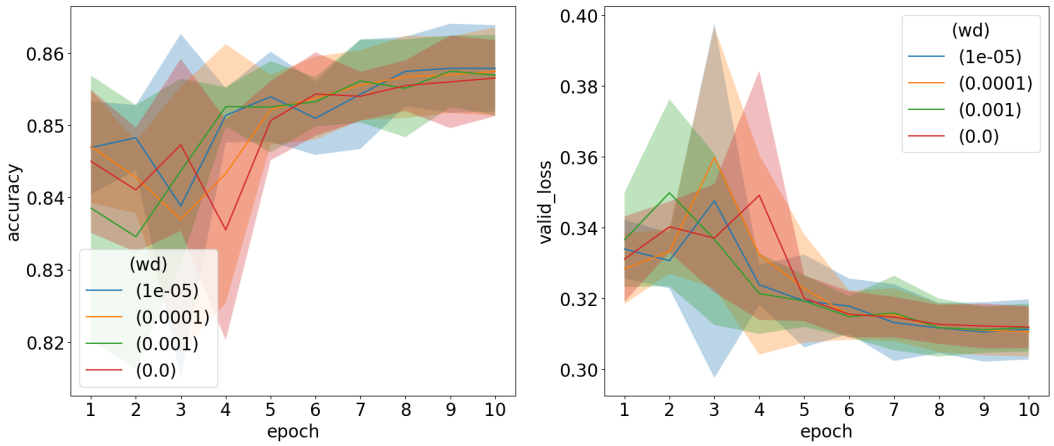


Figure 4.10: A full training run with different weight decays.

Chapter 5

Experiments

"For us, the most important part of rigor is better empiricism, not more mathematical theories."

— Ali Rahimi and Ben Recht, *NIPS 2017*

5.1 Introduction

Since theory and practice does not always go hand-in-hand, it is usually advantageous to compliment a theoretical study or literature review with empirical results. Another motivation for empirical study is that we regard the ability to implement an approach equally as important as understanding the theory behind it. We characterise a good empirical experiment as one that is *rigorous* and *reproducible*. Recently the field of DL has been criticised for the growing gap between the understanding of its techniques and its practical successes[1] where most of the recent focus was on the latter. The speakers urged the deep learning community to be more rigorous in their experiments where, for them, the most important part of rigor is better empiricism, not more mathematical theories.

In this chapter we aim for good empiricism by exploring many hyper-paramters in various data scenarios and doing cross-validation for unbiased performance measures along with standard errors. Our work is not necessarily about beating the benchmark and instead consists of simple experiments that aid in the understanding of how the techniques work and what effect they have.

Furthermore, we want all our experiments to be as reproducible as possible. Therefore we provide all the code, data and necessary documentation

[1]How do I cite the talk given at NIPS2017 - https://www.youtube.com/watch?v=Qi1Yry33TQE

to reproduce the experiments that were done in this thesis (see Appendix C). This is often an overlooked feature of experiments, but is however crucial for transparent and accountable reporting and making your work useful for others to build upon.

The main aim of this chapter is to better understand the behaviours of certain models and parameters and to cross-check the literature with empirical observations. We focus on the same main modelling challenges for deep learning on tabular datasets that we discussed in chapter 4, which are:

- how to represent the inputs,
- how to learn from feature interactions, and
- how to be more sample efficient.

The model interpretation challenge was covered in section 4.5.

The more general hyperparameters, like learning rate, batch size, layer size and layer depth is not the focus of these experiments and some we have already experimented with in previous chapters. However, since these parameters are tightly linked with each other and other model parameters, we still do a hyperparamter search where we deem appropriate and report the findings in Appendix B. The rest of the chapter continues as follows: In §5.2 we discuss the dataset used for these experiments and why it was chosen. Thereafter, in §5.3, we discuss the general methodology of the experiments and how they are evaluated. Then the main experiments follow. §5.4 compares the performance of various entity embedding sizes. §5.5 looks at the proposed approaches to model high-order feature interactions. §5.6 evaluates the different approaches to avoid overfitting in constrained data environments.

5.2 Datasets

We chose to do our experiments on the Adult dataset for the following reasons:

- **Simplicity**: We wanted a dataset that is representative of a real-world case but one that does not have specific modelling challenges like plenty of missing values or highly imbalanced classes. Our goal is to evaluate the models on a generic tabular dataset and not one that requires special attention or the skills of a domain expert. Thus when we fall short we know it is because of the model and not something that is in the data.

- **Minimal preprocessing**: We want to focus our time on training the algorithms and not preprocessing the data. The Adult dataset is relatively clean.
- **Open access**: Since we want our work to be reproducible, we want the dataset we use to be accessible by all.
- **Good size**: Neural networks are data hungry and therefore for optimal performance we prefer a medium size dataset. If we want to test the performance of the models on smaller datasets, we can just run the experiment on a subset of the data.
- **Strong baselines**: In order to know how well we are doing we need to be able to compare our performance with those of others.

For more specifics on the dataset, one can refer to Appendix A

5.3 General Methodology

5.3.1 Loss Function and Evaluation Metric

Since this is a binary classification task we train the neural network to optimise the binary cross-entropy loss as defined in §2.3.4, with $K = 2$. This is the standard loss function to optimise for binary classification (Song *et al.*, 2018, Wang *et al.* (2017), Zhang *et al.* (2016), Qu *et al.* (2016)). We also monitor the accuracy of the model during training since this is often more interpretable, but note that the accuracy is not directly being optimised and that the binary cross-entropy only acts as an differentialable proxy of the accuracy.

5.3.2 Cross-validation

For most of the experiments we will do a 5-fold cross validation (Hastie *et al.*, 2009, p. 241) to estimate the performance of a model. That is, randomly dividing the dataset in five equal parts and then in turn, hold out one of those parts for validation purposes and train the model using the remaining four parts. Figure 5.1 visually explains how the dataset is sub-divided. The performance of the model can then be evaluated on the held-out part. This process is repeated for every one of the five segmentations of the dataset and thus five measurements of the performance of model is obtained. We can then compute the average over these five measurements to obtain a less biased estimate of

the model performance. Another advantage of this approach is that we can obtain standard error for the model performance. These standard errors will be displayed on the figures illustrating the results of the experiments as confidence intervals, *i.e.* $\mu \pm \sigma$. This helps us to evaluate the significance of the results.

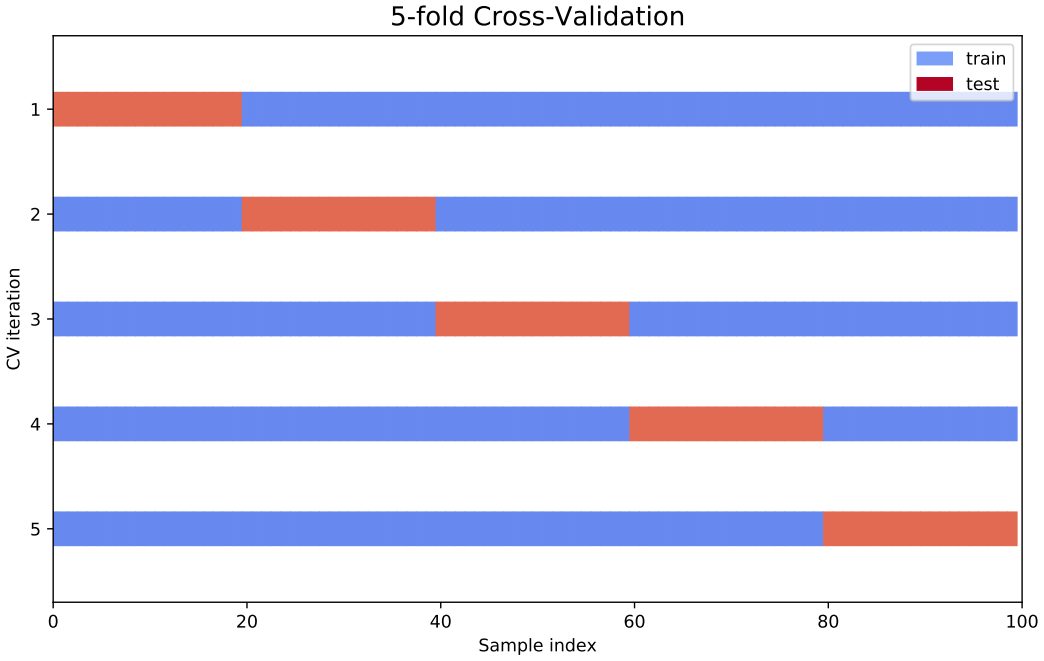


Figure 5.1: 5-Fold Cross-validation dataset split schematic.

It is common in deep learning research to do a once-off split of the data dividing it into training, validation and (sometimes) testing datasets, for example in (Klambauer *et al.*, 2017, Song *et al.* (2018), Zhang *et al.* (2016)). Cross-validation is rarely done in Deep Learning, since the models typically take very long to train and any repitition is thus more costly. The problem with doing a once-off split is that it does not account for the variance of the model and the performance of the model can in fact be very sensitive to the subset of data. By doing cross-validation, we can have more robust performance metrics, including the benefit of reporting on standard errors. However, Deep Learning is also mostly applied to large datasets and if a large test set is available, the gains from cross-validation diminishes. Fortunately, the neural networks applied to tabular data are much smaller than ones used for unstructured data and therefore cross-validation makes sense.

5.3.3 Preprocessing

For our experiments we aimed to do as little feature engineering and preprocessing as possible, in the spirit of automated analysis. We do not create any new feature combinations. The only necessary preprocessing steps are to convert the categorical features into integers (necessary for entity embeddings), standard normal scaling for the continuous features (necessary for optimisation algorithms), mean impute the missing values in the continuous features and assign a "null" category to missing values in the categorical features. We do not do feature selection since we would want the model to learn by itself which features are relevant.

5.3.4 General Hyperparameters

Based on our findings in §4.6 we decide to train all models using the 1Cycle policy and the Adam optimiser. Unfortunately, we cannot follow the hyperparameter selection process suggested by (Smith, 2018) for all the experiments. The process is too manual. Therefore we follow the approach once to find a good selection of the learning rate, number of epochs and weight decay and then use these paramters for the rest of the experiments on this dataset. Some of this is reported in Appendix B. If the model significantly changes over experiments or we experience instable training, we rerun the parameter selection process. Thus we will definitely not find the optimal model for each experiment but it should be sufficient to use for comparison purposes. According to (Smith, 2018) these parameters are also quite robust and the model is not too sensitive on these choices.

We follow a bit of a greedy approach when selecting optimal parameters. All the hyperparameters are very dependent on each other but we cannot run experiments for every possible combination. Therefore we find optimal parameters for a certain experiment and then assume that these parameters are also good for other experiments. If not specified otherwise, we used a simple MLP as the network mapping the input representation to the output. The basice structural hyperparameters we tested can be found in Appendix B. There we experimented with things like the dropout ratio, width and depth of the network, amongst other.

5.4 Input Representation

5.4.1 Embeddings Sizes

The aim of these experiments is to evaluate performance at different embedding sizes. We first explore fixed embedding sizes for all categorical features. The sizes we experiment with are [2,4,8]. Again, we expect there to be optimal embedding size for each variable depedning on the cardinality of the variable and how complex its relationship is with the other variables and the target. We expect the ideal embedding size to be as small as possible but still being able to capture all of the information of the variable. But the only way to tell is by trial and error. We would also like to see if the rules-of-thumb mentioned in §4.2.2 provide reasonable performance. A benefit that these schemes can add is that they depend on the cardinality of the categorical feature.

The results of the one sized embeddings are displayed in Figure 5.2. We see that if all embeddings are fixed at size 2, the model showed greater performance than at 4 and 8. One reason for this may be that the categorical features are not that important for the prediction of the target and that a smaller representation helps to reduce noise.

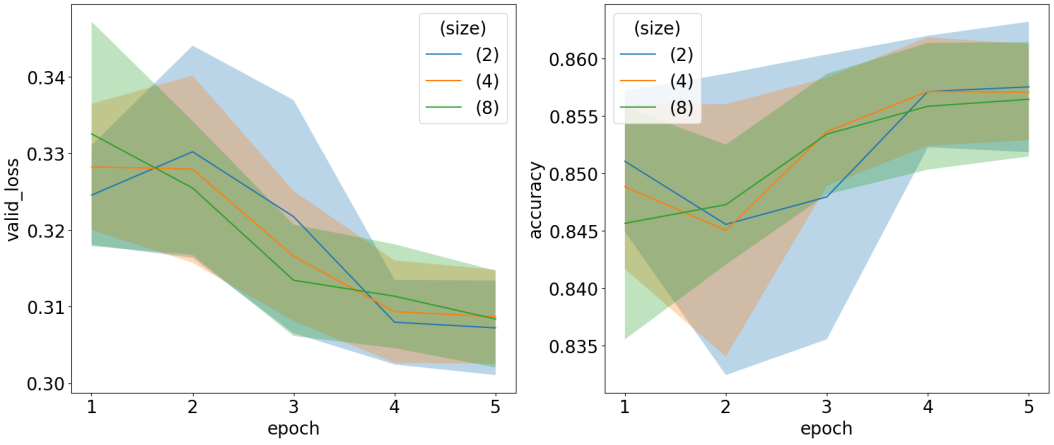


Figure 5.2: Effect of the embedding size if all categorical features are mapped to the same number of dimensions.

We now compare the constant size 2 embeddings with the rules from (Wang *et al.*, 2017) and (de Brébisson *et al.*, 2015) to see how the cardinality dependency effects the performance. The results are shown in Figure 5.3. There is almost no difference between the approaches. Brebisson's methos performs

sightly better on the validation loss and the fixed size of 2 performs the best on accuracy. On other datasets this result will most likely be different if the categorical features contain more rich information.

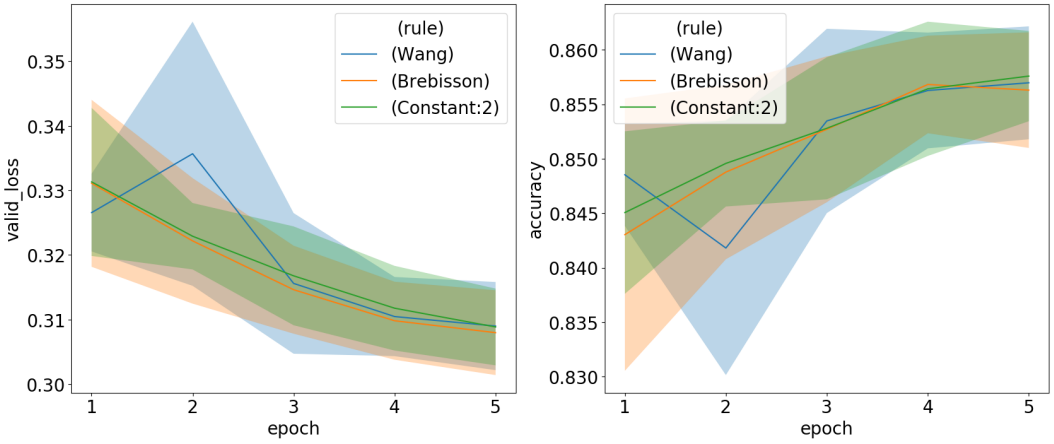


Figure 5.3: Effect of variable sizes on the performance of the model.

5.5 Feature Interactions

5.5.1 Attention

We believe attention is the most promising approach to modelling high-order interactions between features for tabular data, since it has been so effective in other data domains. To test this assuptiom we implemented a multi-head attention module as described in (Song *et al.*, 2018) so that we can compare it to a standard neural network. We did not have the computing power to try different configurations for this module and thus we went with the biggest configuration that could be run in reasonable time. We chose $H = 3$ (Number of heads) and use embedding sizes of 3 all-round. We only used on multi-head attention layer and connected its output to a single hidden layer of size 200, which in turn is connected to the output layer. We compared it to a simple MLP with a similar number of total parameters. The results of this experiment is given in Figure 5.4. The model with the attention mechanism performs worse than the one without and exhibits higher variance. In the future we would like to experiment with more of the hyperparameters of the multi-head attention

module and try stacked attention layers. It will be interesting to see how this results would change if the dataset had more features.

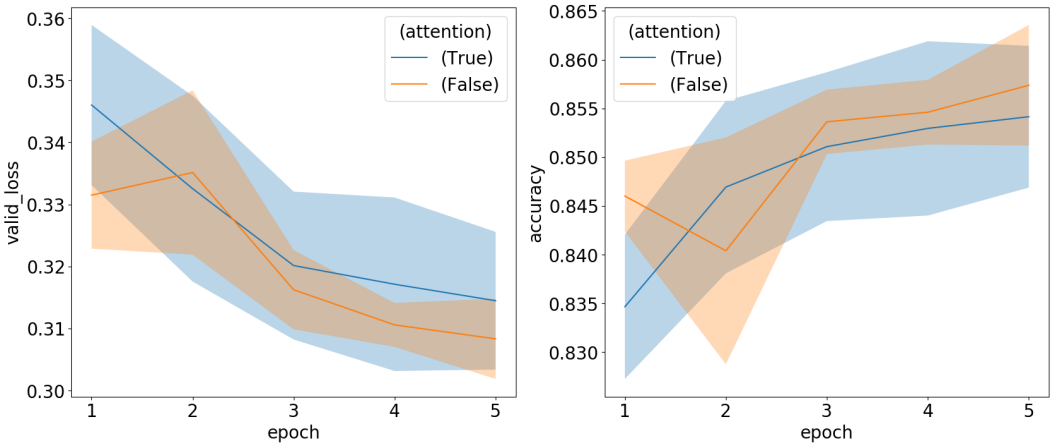


Figure 5.4: Comparing the attention mechanism with a simple MLP.

5.5.2 SeLU

Deeper networks can help us learn higher-order feature interactions. The SeLU activation function is supposed to help us train deeper neural networks. Therefore we compare it to the ReLU activation function over two models, one with 2 layers and the other with 8, to investigate its impact on performance. The SeLU activation function was implemented with its special weight intitialisation, but no dropout was used. In Figure 5.5 and Figure 5.6 we observe that the model with 8 layers and SeLU activations, perform the best overall in terms of the validation loss, but in terms of accuracy the ReLU model with 2 layers showed the best results. The differences observed are minor which leads us to believe that the choice is not that important and that sticking with shallow ReLU networks are sufficient until more experiments are done.

5.5.3 Skip-Connections

Another mechanism we can use to make deeper networks possible is a skip-connection. Skip-connections may also help to combine different orders of feature interactions. The skip connection combines the activations before and afer a linear layer through an elementwise addition. We tested networks with skip-connections for a shallow and a deep neural network. The results

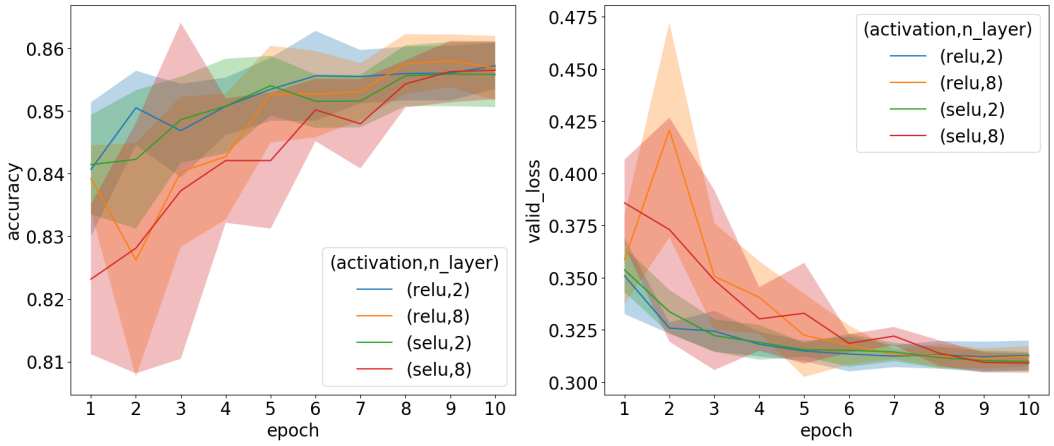


Figure 5.5: The average performance of ReLU and SeLU activation functions for shallow and deep networks over every epoch.

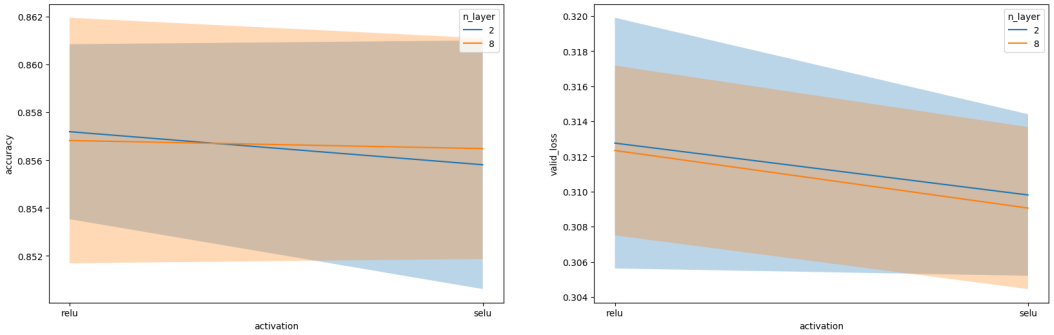


Figure 5.6: The average performance of ReLU and SeLU activation functions for shallow and deep networks.

are summarised in Figure 5.7 and Figure 5.8. On average did the deeper neural network perform better and in terms of the validation loss did the skip-connections slightly improve the performance. Again, there is very little between the two types and thus we cannot say one is better thant the other. For simplicity sake, we would suggest not using skip-connections, until more experiments have been done.

5.6 Sample Efficiency

We did an experiment to see how the number of samples available to the network for training influences its learning ability. We trained the network at training set sizes of 1000, 2000, 4000, 8000, 16000 and 32000. The results are reported in Figure 5.9. As expected, the network performs better as the training set

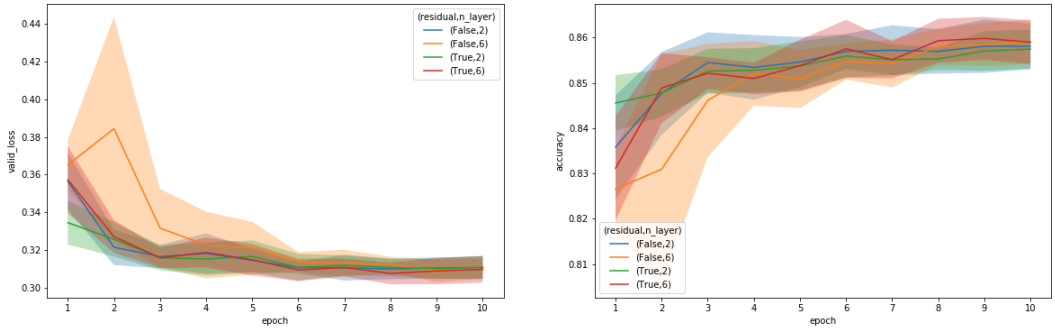


Figure 5.7: Average performance at each epoch for shallow and deep neural networks, with and without skip connections.

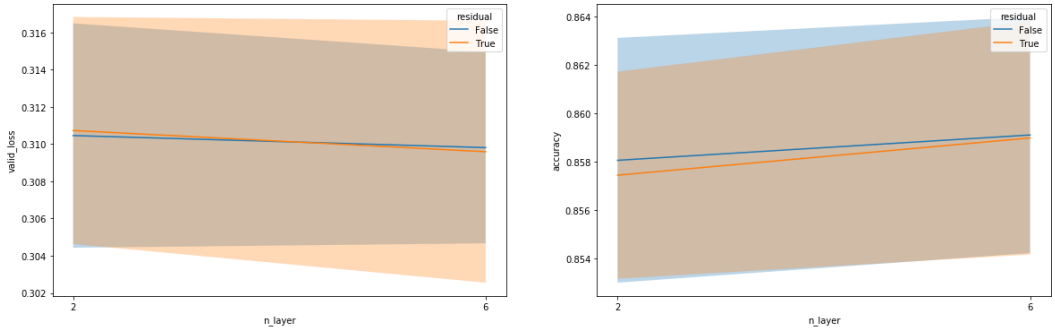


Figure 5.8: Overall performance of the skip-connections used in a shallow and deep neural network.

increases, but the increase diminish as the sample size gets higher. We do observe a strange results where the model trained only using 4000 samples exhibit one of the best accuracies.

5.6.1 Data Augmentation

Mixup was the most promising form of data augmentation from our literature review. In this experiment we want to investigate how mixup with different mixup ratios influence the performance of our neural networks on the adult dataset. Since data augmentation also acts as a form of regularisation, we want to see the interaction between weight decay and mixup ratios. We experimented with mixup α-ratios of [0,0.4] and with weight decays of $[10^{-5}, 10^{-3}]$ on a neural network with 3 hidden layers of 200 units each. The results of the experiment are captured in Figure 5.10 and Figure 5.11.

The results indicate that mixup is not improving the validation loss or the accuracy of the models. It is also interesting to see the interaction between

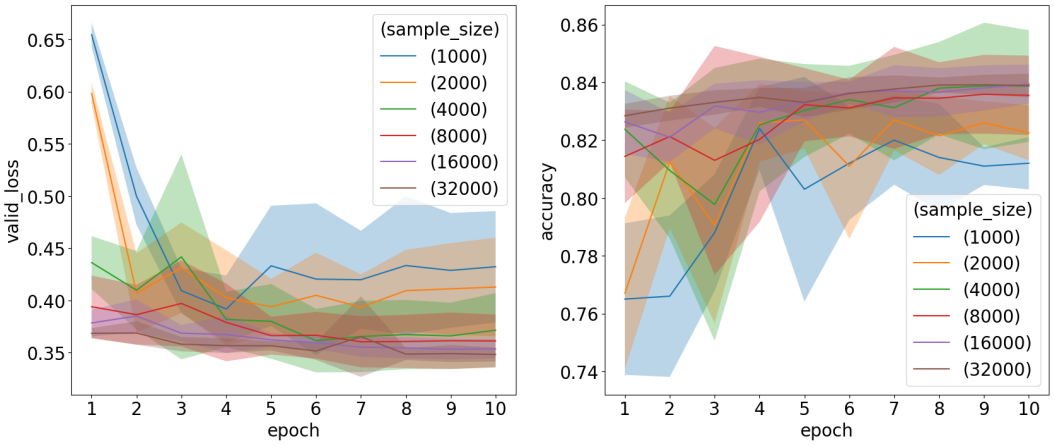


Figure 5.9: Effect of the number of training samples on the performance of neural networks.

mixup and weight decay. Since both are forms of regualrisation we would expect one to reduce the need of the other, but this is not what the results suggest. Although a promising technique from the literature, we think the value of mixup depends on the dataset and whether or not interpolating between samples makes sense.

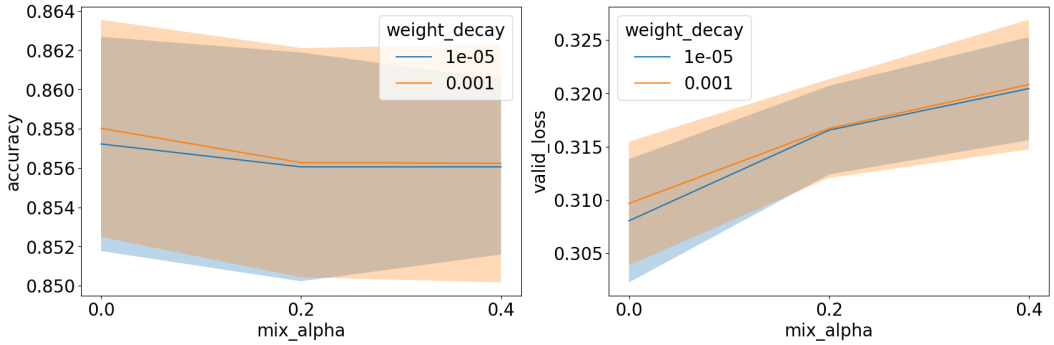


Figure 5.10: Average performance of models with various mixup and weight decay parameters.

5.6.2 Unsupervised Pretraining

Here we investigate whether or not DAEs as a method of unsupervised pretraining is beneficial to a supervised learning neural network. We trained a DAE with swap noise for 15 epochs on the Adult dataset, with the swap noise proportion set at 15%. The DAE had 3 hidden layers of 500 units each.

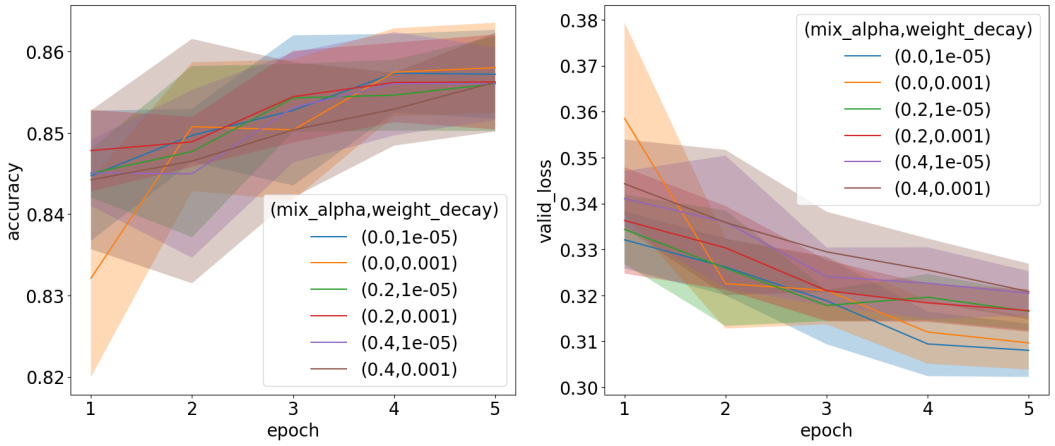


Figure 5.11: Performance per epoch for models with different weight decays and mixup ratios.

Then we transfer the learned weights to a supervised learning network, also with 3 hidden layers of 500 units. But the supervised network has a different output layer to the DAE. Therefore we first keep the transferred weights fixed and train only the last layer of the supervised learning network, as a way of initialising its weights. Otherwise if we would have trained all the weights together from the start, the random weights of the output layer might have interfered with the learned weights in the hidden layers. Once the output layer is initialised with this process, we can train all of the network simultaneously. This final stage of training is showed in Figure 5.12 where we compare it to a model without pretraining. We see that in terms of both the accuracy and the validation loss that the pretrained model has an advantage over the trained from scratch classifier. This makes sense since the pretrained model does not have to start from a random initialisation. But as training continues this gaps becomes smaller. In terms of the validation loss the two models are equivalent when training completed, but in terms of the accuracy the trained from scratch classifier outperformed the pretrained model. We do not believe these results are conclusive since there are still so many avenues to explore for unsupervised pretraining with DAEs. We do not know yet how to design the DAE in terms of its network architecture, amount of noise injected, lenght of training cycle, etc, and then what the best way is of transferring this knowledge to a classifier. As a test to validate if the DAE learned something useful one can feed the activations from one of the DAE hidden layers to another machine learning algorithm and see whether these representations perform better than with the

raw data. From the results of this basic implementation, we do still believe that there is value in this approach and that it warrants further exploration.

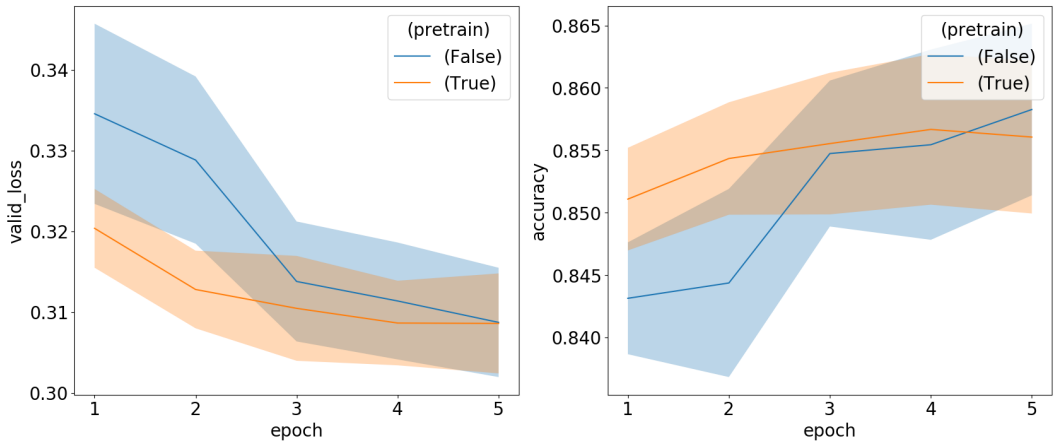


Figure 5.12: The effect of pretraining on the classifier's performance.

5.7 General Thoughts

We ran thorough experiments for the most promising techniques in deep learning for tabular data. In most of the experiments there were not clear differences between the performances of the different techniques. Our one hypothesis is that the Adult dataset does not lend itself to more advanced modelling techniques and that the basic neural network is close to the best one can get. Our other concern is that the greedy fashion of our hyperparameter selection has limited the various approaches. Since it was not feasible to tune the required paramters for all models in all folds, we selected a set of paramters from initial experiments and used them for most of the experiments. But since these parameters are so inter-connected, chaning one already means that the others are not optimal anymore. We were aware of this limitation from the start, but we still do not know however to what extent it influenced our results.

Chapter 6

Conclusion

In this work we set out to investigate a relatively under-explored area of deep learning; applying deep learning approaches to tabular datasets. The goal was to find and understand the best approaches for this task by reviewing the literature and doing an empirical study. To assist in this endeavour we also aimed to review the core concepts of neural networks and discuss the modern advancements in deep learning, with the hope of identifying approaches that are transferrable to our domain.

In this chapter we summarise the work of this thesis and evaluate how each of the chapters contributed to the set out objectives (§6.1). The limitations of this work is discussed in §6.2, how they affected the study and how it could be improved upon. We conclude this work with a section on promising future directions in this area of research (§6.3).

6.1 Summary

In **Chapter 1** the motivation and the objectives of this work were introduced. It was stated that deep learning for tabular data is an important topic but that it has not received enough attention in the literature. The main issues that (we feel) need to be addressed in order to make progress in the field were highlighted. The chapter also provided an overview of the fundamentals of Statistical Leaning Theory, including different learning paradigms, loss functions, optimisation and overfitting, to serve as background for the problems we were trying to solve.

Chapter 2 was about neural networks and how to train them. Neurons,

layers, activation functions and network architectures were described in detail to gain a complete understanding of the mechanics of neural networks. To describe how a neural network is trained, the backpropogation and stochastic gradient descent algorithms were introduced, with the help of examples. The chapter included a brief look at basic regularisation methods for neural networks. This was followed by a section on representation learning to gain insight to what a neural network is actually learning and how it does so. This chapter provided us with the fundamentals of neural networks to enhance our capacity to understand the more advanced developments in deep learning that follows.

Chapter 3 was about deep learning in general. The major developments contributing to the recent success of deep learning were investigated. The chapter introduced autoencoders and how they can be used for unsupervised pre-training, along with the concept of transfer leaning. Very effective regularisation techniques, data augmentation and dropout were described. This was followed by a review of the more modern layers and architecture designs in deep learning, which included normalisation layers, skip-connections, embeddings and the attention mechanism. The chapter included a section on the concept of superconvergence and how it can be achieved by using the 1Cycle policy and more effective hyperparameter selection, with the help of an example and experiment. A brief discussion on the neural network specific and model agnostic tools available for interpretting deep neural networks was also provided and examples were shown. The topics that were focussed on in this chapter gave insight into how deep learning achieved its success in other data domains, which provided a good starting point in an attempt to improve deep learning for tabular data.

Chapter 4 was about deep learning for tabular data. In this chapter a review was done on all the latest literature on applying deep neural networks to tabular data. The main challenges to deep learning for tabular data, as stated in Chapter 1, guided this review. The chapter explores the various ways of preprocessing continuous features and embedding categorical features in a way that is optimal for a neural network. Then it investigated how a neural network can be encouraged to learn better feature interactions, by using attention modules and cross-features. The SeLU activation was proposed, along with skip-connections, as ways to build deeper neural networks for more complex feature learning. This was followed by evaluating approaches that make deep neural networks more sample efficient. It focussed on denoising autoencoders

as a way of pretraining a network and the data augmentation techniques, swap noise and mixup. Reference was made to per-feature regularisation. Then there was a discussion on more ways of intepretting neural networks for tabular data, considering the approaches disscussed earlier in the chapter. An example of interpretting a network with knowledge distillation was also given. The chapter concluded with an empirical investigation of superconvergence on tabular data. This chapter provided us with a solid technical understanding of the approaches available to overcome the challenges in the field. It also gave sufficient detail to be able to implement these promising approaches.

Chapter 5 contains the empirical experiments. The experiments served to assist the exploratory study of deep learning for tabular data. The experiments were categorised by three main themes: input representation, feature interaction and sample efficiency. In the input representation experiments we evaluated the effect the entity embedding sizes had on the performance of the model. The attention mechanism, along with skip-connections and the SeLU activation were experimented with to see how they do in modeling feature interactions. Finally, in the sample efficiency section, we investigated how sensitive neural networks are to the number of observations in a dataset and tested unsupervised pretraining as an approach to alleviate this sensitivity. We also looked at the mixup and swap noise data augmentation approaches as means to avoid overfitting.

6.2 Limitations

There were various aspects that limited the impact of this study. Two technical limitations were experienced:

- **Access to large compute:** Deep learning techniques are notorious for the amount of computing power they require. There was limited access to cloud computing providers on which some of the experiments were done. However, in most cases, we only had access to a small personal machine without a graphical processing unit (GPU). This significantly increased the running time of the experiments and hindered rapid iteration of different approaches. In future work we would make sure that sufficient computing power is available for experimenting in this field.

- **Access to quality code:** Many of the recent developments discussed in this work had no official implementation at the time of writing. This forced us to rewrite many of the code in order to validate the results claimed in the literature and to be able to use it in our data and models. Sometimes technical details were left out of the original papers and we had to improvise to get a working example. By doing it this way we risked using code for our experiments that might not have been exactly what the authors intendend.

The other limitations to this study were:

- **Experiments on a single dataset:** Due to the technical limitations described above, we only had the capacity to process a single dataset for our experiments. This made our results inconclusive, but still adds value as an exploratory study. There is also a possibility that this dataset was in a sense to easy to solve and that we could not demonstrate the full power of the deep learning approaches. This study would have benefitted from experiments on datasets with different properties and with different tasks. We published all of our code online if anyone wants to build on this work.

- **Based on pre-prints:** Deep learning is such a fast developing area of research and in an attempt to keep this work relevant, pre-prints of publications were cited. Pre-prints are not peer-reviewed and subject to change. We did our best to critically evaluate the work cited and to confirm findings with experiments. Although we tried to keep up with the deluge of publications, there remains a possibility that new publications arised during the post-research phase of this work.

6.3 Promising Future Directions

Throughout this thesis we have identified promising research directions which, however, did not fit into the scope of this work. Besides repeating the empirical work on a wider variety of datasets, we propose the following to be a valuable research path to follow:

Generative models for tabular data. Here we specifically refer to *Variational Autoencoders* (VAE) (Kingma and Welling, 2013) and *Generative*

Adversarial Networks (GAN) (Goodfellow *et al.*, 2014). A VAE provides a probabilistic manner for describing an observation in a latent space. That is, instead of using a single value to describe an attribute in the latent space, like the standard autoencoder, a VAE uses a probability distribution as a description of a latent space attribute. The greatest value we think VAEs can add is as a method of unsupervised pretraining. When using DAEs for unsupervised pretraining, it needs to be injected with some noise. However, the best noise schemes we found, were swap noise and blank-out, but as mentioned before, neither makes complete sense in the tabular dataset environment. A VAE can provide a more robust way of doing unsupervised learning because it decoding function needs to learning to process probabilistic output and thus it is not reliant on noise injection. In addition, once trained, a VAE can be used to generate new samples by sampling from the latent probabilistic distribution, which may serve as a way to generate more training samples.

We also believe that GANs offer a good alternative to generating new training samples. A GAN consists of two neural networks, a generator (G) and a discriminator (D). The task of G is: given random noise as input, generate artificial samples of the data that are indistinguishable from the genuine samples. The task D is to attempt to discriminate between the two. GANs have shown tremendous value in data synthesis, especial in the domains of computer vision and speech synthesis, producing life-like faces (Karras *et al.*, 2017) and voices (Donahue *et al.*, 2018). We think GANs can achieve similar success in data synthesis for tabular data and can thus also be used to artificially enlarge training datasets for supervised learning.

Appendices

Appendix A

Datasets

Details of each of the datasets used in Chapter 5 and elsewhere.

A.1 Adult Dataset

The Adult dataset, originally used in (Kohavi, 1996), is data that was extracted from the census bureau database[1] and can be accessed from this link[2]. The task here is to predict whether or a not a certain person's income exceeds $50,000 per year. Thus it is posed as a binary classifcation problem.

In total there are 14 features and 48,842 observations. Two-thirds of the observations were randomly selected to form the training set and the rest allocated to the test set. Note, that we will not use any of the observations in the test set during our experiments. Hyperparemeter and model selection decisions are made based on the valdiation dataset performace and then if we wish we can evaluate the selected models on the test set for the most accuracte estimation of the generalisation ability of the models.

The details for each of the features are listed below. We indicate the continuous features and the classes for the categorical features.

- *age*: continuous
- *workclass*: Private, Self-emp-not-inc, Self-emp-inc, Federal-gov, Local-gov, State-gov, Without-pay, Never-worked
- *fnlwgt*: continuous

[1] http://www.census.gov/en.html
[2] http://archive.ics.uci.edu/ml/datasets/Adult

- *education*: Bachelors, Some-college, 11th, HS-grad, Prof-school, Assoc-acdm, Assoc-voc, 9th, 7th-8th, 12th, Masters, 1st-4th, 10th, Doctorate, 5th-6th, Preschool
- *education-num*: continuous
- *marital-status*: Married-civ-spouse, Divorced, Never-married, Separated, Widowed, Married-spouse-absent, Married-AF-spouse
- *occupation*: Tech-support, Craft-repair, Other-service, Sales, Exec-managerial, Prof-specialty, Handlers-cleaners, Machine-op-inspct, Adm-clerical, Farming-fishing, Transport-moving, Priv-house-serv, Protective-serv, Armed-Forces
- *relationship*: Wife, Own-child, Husband, Not-in-family, Other-relative, Unmarried
- *race*: White, Asian-Pac-Islander, Amer-Indian-Eskimo, Other, Black
- *sex*: Female, Male
- *capital-gain*: continuous
- *capital-loss*: continuous
- *hours-per-week*: continuous
- *native-country*: United-States, Cambodia, England, Puerto-Rico, Canada, Germany, Outlying-US(Guam-USVI-etc), India, Japan, Greece, South, China, Cuba, Iran, Honduras, Philippines, Italy, Poland, Jamaica, Vietnam, Mexico, Portugal, Ireland, France, Dominican-Republic, Laos, Ecuador, Taiwan, Haiti, Columbia, Hungary, Guatemala, Nicaragua, Scotland, Thailand, Yugoslavia, El-Salvador, Trinadad&Tobago, Peru, Hong, Holand-Netherlands

The distributions of the continuous features are plotted in Figure A.1 and the counts of the categories for each of the categorical features can be found in Figure A.2. The highest reported accuracy in the original paper (Kohavi, 1996) was 85.9% by the NBTree algorithm and in an unpublished experiment[3] one researcher achieve ~88% accuracy with boosted trees.

[3]https://www.kaggle.com/kanav0183/catboost-and-other-class-algos-with-88-accuracy

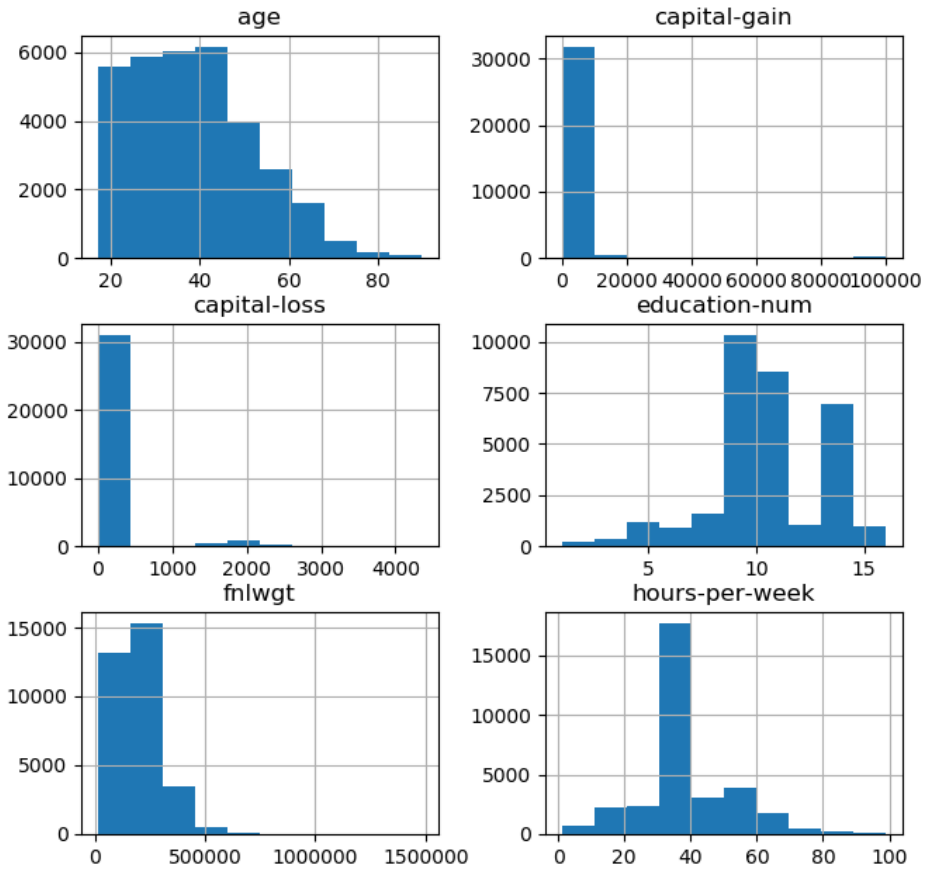


Figure A.1: Histograms for each of the continuous features in the Adult dataset.

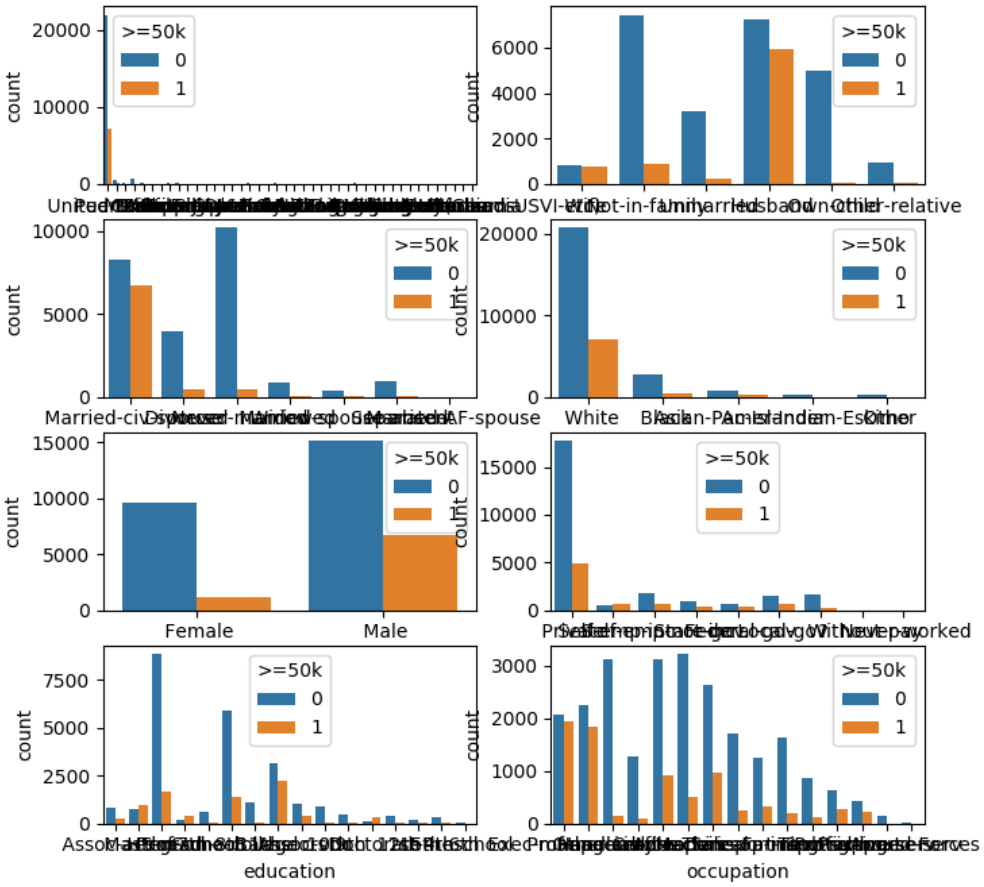


Figure A.2: Bar plot for each of the categorical features in the Adult dataset.

Appendix B

Hyperparameter Search

B.1 Width and Depth of Network

This is a very common hyperparamter to train, for exmaple, done by (Guo *et al.*, 2017, Qu *et al.* (2016), Zhang *et al.* (2016)). Here we investigate the effect of the size of the network on the different datasets. We compare the performance of the models at different numbers and sizes of layers. Larger networks are more flexible and therefore we expect it to act similarly to any learning model flexibility parameter. Increasing the network size will be beneficial up until a certrain point until it becomes too big and be more prone to overfitting. We also want to get a feel for how important these hyperparameters are. On the Adult dataset we experimented with layer depths of 1, 3, 6 and 12, and layer widths of 32, 128, 512, 2048. Note give all layers the same width as this is found to work good enough in practice (Guo *et al.*, 2017, Qu *et al.* (2016), Zhang *et al.* (2016)). The results of the experiment are displayed in Figure B.1. The results are mixed between the accuracy and the validation loss and there is not much separating any of the models. Except for the very wide network that had a diverging validation loss. For the sake of simplicity we would recommend to use a 3-layer network with 128 units.

B.2 Dropout

Dropout is almost always used in deep learning and it is a typical parameter to tune before modelling. We wanted to find out what the best dropout ratio is for our models and how the performance varies over different values. We

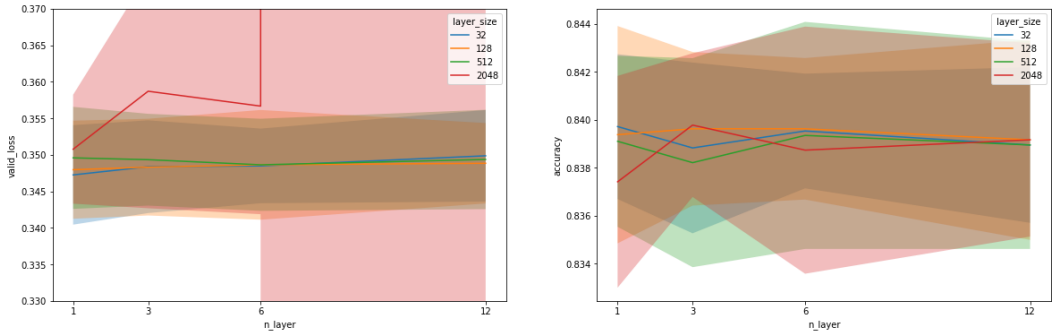


Figure B.1: Effect of the layer width and network depth on the performance on the Adult dataset.

tested dropout proportions of [0.1,0.2,...,0.8] on a wide and narrow network respectively. The results are given in Figure B.2. From these results we see that any dropout ratio between 0.2 and 0.5 will suffice for both types of networks.

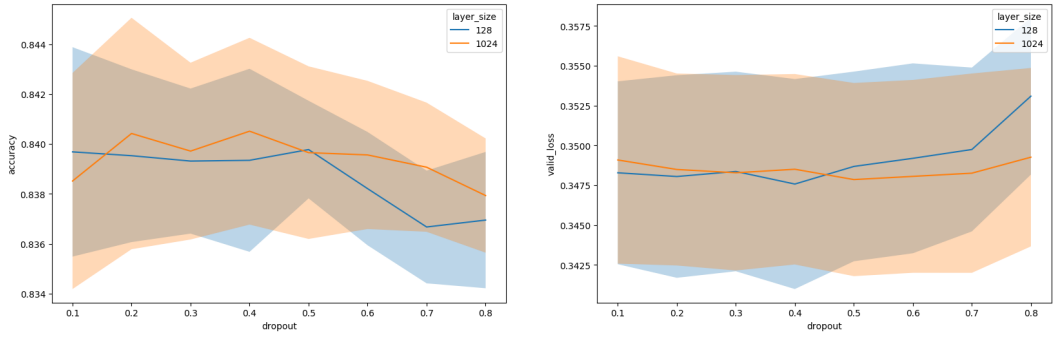


Figure B.2: The effect of dropout on wide and narrow neural networks.

Appendix C

Software and Code

C.1 Development Environment

- Deep Learning Libraries: Pytorch and Fastai
- Hardware:
  - MacBook Air (2017)
  - Cloud Providers: AWS EC2[1], SalamanderAI[2], Google Compute Engine[3].
- Programming Language: Python
- github for version control
- RMarkdown for writing and compiling the thesis document

C.2 Code and Reproducibility

Note that all of the code used in the thesis, including the source documents, is made available in the tabularLearner Github repository [4]. More instructions on how to implement the code is contained in the file named `README.md`, in the repository.

[1]https://aws.amazon.com/ec2/
[2]https://salamander.ai/
[3]https://cloud.google.com/compute/
[4]https://github.com/jandremarais/tabularLearner

Bibliography

Alain, G. and Bengio, Y. (2014). What regularized auto-encoders learn from the data-generating distribution. *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 3563–3593.

Anonymous (2019). Tabnn: A universal neural network solution for tabular data. In: *Submitted to International Conference on Learning Representations*. Under review.
Available at: https://openreview.net/forum?id=r1eJssCqY7

Ba, L.J. and Caurana, R. (2013). Do deep nets really need to be deep? *CoRR*, vol. abs/1312.6184. 1312.6184.
Available at: http://arxiv.org/abs/1312.6184

Bahdanau, D., Cho, K. and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *CoRR*, vol. abs/1409.0473. 1409.0473.
Available at: http://arxiv.org/abs/1409.0473

Battenberg, E., Chen, J., Child, R., Coates, A., Gaur, Y., Li, Y., Liu, H., Satheesh, S., Seetapun, D., Sriram, A. and Zhu, Z. (2017). Exploring neural transducers for end-to-end speech recognition. *CoRR*, vol. abs/1707.07413. 1707.07413.
Available at: http://arxiv.org/abs/1707.07413

Bengio, Y., Boulanger-Lewandowski, N. and Pascanu, R. (2012). Advances in optimizing recurrent networks. *CoRR*, vol. abs/1212.0901. 1212.0901.
Available at: http://arxiv.org/abs/1212.0901

Bengio, Y., Courville, A. and Vincent, P. (2013 Aug). Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 8, pp. 1798–1828. ISSN 0162-8828.

Bengio, Y., Lamblin, P., Popovici, D. and Larochelle, H. (2007). Greedy layer-wise training of deep networks. In: *Advances in Neural Information Processing Systems 19*, pp. 153–160. MIT Press.

Breiman, L. (2001). Random forests. *Machine learning*, vol. 45, no. 1, pp. 5–32.

Chapelle, O., Weston, J., Bottou, L. and Vapnik, V. (2001). Vicinal risk minimization. In: Leen, T.K., Dietterich, T.G. and Tresp, V. (eds.), *Advances in Neural Information Processing Systems 13*, pp. 416–422. MIT Press.
Available at: http://papers.nips.cc/paper/1876-vicinal-risk-minimization.pdf

Che, Z., Purushotham, S., Khemani, R.G. and Liu, Y. (2016). Interpretable deep models for icu outcome prediction. *AMIA ... Annual Symposium proceedings. AMIA Symposium*, vol. 2016, pp. 371–380.

Cheng, H., Koc, L., Harmsen, J., Shaked, T., Chandra, T., Aradhye, H., Anderson, G., Corrado, G., Chai, W., Ispir, M., Anil, R., Haque, Z., Hong, L., Jain, V., Liu, X. and Shah, H. (2016*a*). Wide & deep learning for recommender systems. *CoRR*, vol. abs/1606.07792. 1606.07792.
Available at: http://arxiv.org/abs/1606.07792

Cheng, J., Dong, L. and Lapata, M. (2016*b*). Long short-term memory-networks for machine reading. *CoRR*, vol. abs/1601.06733. 1601.06733.
Available at: http://arxiv.org/abs/1601.06733

Choromanska, A., Henaff, M., Mathieu, M., Arous, G.B. and LeCun, Y. (2014). The loss surface of multilayer networks. *CoRR*, vol. abs/1412.0233. 1412.0233.
Available at: http://arxiv.org/abs/1412.0233

Clevert, D.-A., Unterthiner, T. and Hochreiter, S. (2015 November). Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs). *ArXiv e-prints*. 1511.07289.

Covington, P., Adams, J. and Sargin, E. (2016). Deep neural networks for youtube recommendations. In: *Proceedings of the 10th ACM Conference on Recommender Systems*. New York, NY, USA.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314.

de Brébisson, A., Simon, É., Auvolat, A., Vincent, P. and Bengio, Y. (2015). Artificial neural networks applied to taxi destination prediction. *CoRR*, vol. abs/1508.00021. 1508.00021.
Available at: http://arxiv.org/abs/1508.00021

Devlin, J., Chang, M.-W., Lee, K. and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Donahue, C., McAuley, J. and Puckette, M. (2018). Synthesizing audio with generative adversarial networks. *arXiv preprint arXiv:1802.04208*.

Duchi, J., Hazan, E. and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159.

Duong, L., Anastasopoulos, A., Chiang, D., Bird, S. and Cohn, T. (2016). An attentional model for speech translation without transcription. In: *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 949–959.

Emin, O. and Xaq, P. (2018). Skip connections eliminate singularities. In: *International Conference on Learning Representations*.

Erhan, D., Bengio, Y., Courville, A. and Vincent, P. (2009). Visualizing higher-layer features of a deep network.

Fernández-Delgado, M., Cernadas, E., Barro, S. and Amorim, D. (2014). Do we need hundreds of classifiers to solve real world classification problems? *Journal of Machine Learning Research*, vol. 15, pp. 3133–3181.
Available at: http://jmlr.org/papers/v15/delgado14a.html

Fridman, L., Brown, D.E., Glazer, M., Angell, W., Dodd, S., Jenik, B., Terwilliger, J., Kindelsberger, J., Ding, L., Seaman, S., Abraham, H., Mehler, A., Sipperley, A., Pettinato, A., Seppelt, B., Angell, L., Mehler, B. and Reimer, B. (2017). MIT autonomous vehicle technology study: Large-scale deep learning based analysis of driver behavior and interaction with automation. *CoRR*, vol. abs/1711.06976. 1711.06976.
Available at: http://arxiv.org/abs/1711.06976

Friedman, J.H. (2001). Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pp. 1189–1232.

Friedman, J.H. and Stuetzle, W. (1981). Projection pursuit regression. *Journal of the American statistical Association*, vol. 76, no. 376, pp. 817–823.

Frosst, N. and Hinton, G.E. (2017). Distilling a neural network into a soft decision tree. *CoRR*, vol. abs/1711.09784. 1711.09784.
Available at: http://arxiv.org/abs/1711.09784

Gatys, L.A., Ecker, A.S. and Bethge, M. (2015). A neural algorithm of artistic style. *CoRR*, vol. abs/1508.06576. 1508.06576.
Available at: http://arxiv.org/abs/1508.06576

Goodfellow, I., Bengio, Y. and Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A. and Bengio, Y. (2014). Generative adversarial nets. In: *Advances in neural information processing systems*, pp. 2672–2680.

Guo, C. and Berkhahn, F. (2016). Entity embeddings of categorical variables. *CoRR*, vol. abs/1604.06737. 1604.06737.
Available at: http://arxiv.org/abs/1604.06737

Guo, H., Tang, R., Ye, Y., Li, Z. and He, X. (2017). Deepfm: A factorization-machine based neural network for CTR prediction. *CoRR*, vol. abs/1703.04247. 1703.04247.
Available at: http://arxiv.org/abs/1703.04247

Haldar, M., Abdool, M., Ramanathan, P., Xu, T., Yang, S., Duan, H., Zhang, Q., Barrow-Williams, N., Turnbull, B.C., Collins, B.M. and Legrand, T. (2018 October). Applying Deep Learning To Airbnb Search. *ArXiv e-prints*. 1810.09591.

Hastie, T., Tibshirani, R. and Friedman, J. (2009). *The elements of statistical learning: data mining, inference and prediction*. 2nd edn. Springer.
Available at: http://www-stat.stanford.edu/~tibs/ElemStatLearn/

He, K., Zhang, X., Ren, S. and Sun, J. (2015*a*). Deep residual learning for image recognition. *CoRR*, vol. abs/1512.03385. 1512.03385.
Available at: http://arxiv.org/abs/1512.03385

He, K., Zhang, X., Ren, S. and Sun, J. (2015*b*). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, vol. abs/1502.01852. 1502.01852.
Available at: http://arxiv.org/abs/1502.01852

He, K., Zhang, X., Ren, S. and Sun, J. (2016 10). Identity mappings in deep residual networks. vol. 9908, pp. 630–645.

Hinton, G. (1989). Connectionist learning procedures.

Hinton, G., Vinyals, O. and Dean, J. (2015). Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*.

Hinton, G.E., Osindero, S. and Teh, Y.-W. (2006 July). A fast learning algorithm for deep belief nets. *Neural Comput.*, vol. 18, no. 7, pp. 1527–1554. ISSN 0899-7667.
Available at: http://dx.doi.org/10.1162/neco.2006.18.7.1527

Hinton, G.E. and Salakhutdinov, R.R. (2006). Reducing the dimensionality of data with neural networks. *science*, vol. 313, no. 5786, pp. 504–507.

Hinton, G.E., Srivastava, N., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, vol. abs/1207.0580.
Available at: http://arxiv.org/abs/1207.0580

Howard, J. and Ruder, S. (2018). Fine-tuned language models for text classification. *arXiv preprint arXiv:1801.06146*.

Hu, J., Shen, L. and Sun, G. (2017). Squeeze-and-excitation networks. *CoRR*, vol. abs/1709.01507. 1709.01507.
Available at: http://arxiv.org/abs/1709.01507

Huang, F., Ash, J., Langford, J. and Schapire, R. (2017). Learning deep resnet blocks sequentially using boosting theory. *arXiv preprint arXiv:1706.04964*.

Huang, G., Liu, Z. and Weinberger, K.Q. (2016). Densely connected convolutional networks. *CoRR*, vol. abs/1608.06993. 1608.06993.
Available at: http://arxiv.org/abs/1608.06993

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, vol. abs/1502.03167.
Available at: http://arxiv.org/abs/1502.03167

Ivakhnenko, A. and Lapa, V. (1965 04). Cybernetic predicting devices. p. 250.

Karras, T., Aila, T., Laine, S. and Lehtinen, J. (2017). Progressive growing of gans for improved quality, stability, and variation. *CoRR*, vol. abs/1710.10196. 1710.10196.
Available at: http://arxiv.org/abs/1710.10196

Kingma, D.P. and Ba, J. (2014). Adam: A method for stochastic optimization. *CoRR*, vol. abs/1412.6980. 1412.6980.
Available at: http://arxiv.org/abs/1412.6980

Kingma, D.P. and Welling, M. (2013). Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*.

Klambauer, G., Unterthiner, T., Mayr, A. and Hochreiter, S. (2017). Self-normalizing neural networks. *CoRR*, vol. abs/1706.02515. 1706.02515.
Available at: http://arxiv.org/abs/1706.02515

Kohavi, R. (1996). Scaling up the accuracy of naive-bayes classifiers: a decision-tree hybrid. Citeseer.

Kosar, R. and Scott, D.W. (2018 January). The Hybrid Bootstrap: A Drop-in Replacement for Dropout. *ArXiv e-prints*. 1801.07316.

Krizhevsky, A., Sutskever, I. and Hinton, G.E. (2012). Imagenet classification with deep convolutional neural networks. In: *Proceedings of the 25th International Conference on Neural Information Processing Systems*, NIPS'12, pp. 1097–1105. Curran Associates Inc., USA.
Available at: http://dl.acm.org/citation.cfm?id=2999134.2999257

Larochelle, H., Bengio, Y., Louradour, J. and Lamblin, P. (2009). Exploring strategies for training deep neural networks. *Journal of machine learning research*, vol. 10, no. Jan, pp. 1–40.

Lecun, Y., Bengio, Y. and Hinton, G. (2015 5). Deep learning. *Nature*, vol. 521, no. 7553, pp. 436–444. ISSN 0028-0836.

Lee, H., Chaitanya, E. and Andrew, N. (2008). Sparse deep belief net model for visual area v2. In: Platt, J.C., Koller, D., Singer, Y. and Roweis, S.T. (eds.), *Advances in Neural Information Processing Systems 20*, pp. 873–880. Curran Associates, Inc.

Li, M., Zhang, T., Chen, Y. and Smola, A.J. (2014). Efficient mini-batch training for stochastic optimization. In: *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pp. 661–670. ACM, New York, NY, USA. ISBN 978-1-4503-2956-9.
Available at: http://doi.acm.org/10.1145/2623330.2623612

Maas, A.L., Hannun, A.Y. and Ng, A.Y. (). Rectifier nonlinearities improve neural network acoustic models.

Makhzani, A. and Frey, B. (2013). k-sparse autoencoders. *CoRR*, vol. abs/1312.5663.

McClelland, J., Rumelhart, D. and the PDP Research Group (eds.) (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*. MIT Press.

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

Miotto, R., Li, L., Kidd, B.A. and Dudley, J.T. (2016). Deep patient: An unsupervised representation to predict the future of patients from the electronic health records. In: *Scientific reports*.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M.A. (2013). Playing atari with deep reinforcement learning. *CoRR*, vol. abs/1312.5602. 1312.5602.
Available at: http://arxiv.org/abs/1312.5602

Mogren, O. (2016). C-RNN-GAN: continuous recurrent neural networks with adversarial training. *CoRR*, vol. abs/1611.09904. 1611.09904.
Available at: http://arxiv.org/abs/1611.09904

Perez, L. and Wang, J. (2017). The effectiveness of data augmentation in image classification using deep learning. *CoRR*, vol. abs/1712.04621. 1712.04621.
Available at: http://arxiv.org/abs/1712.04621

Qu, Y., Cai, H., Ren, K., Zhang, W., Yu, Y., Wen, Y. and Wang, J. (2016). Product-based neural networks for user response prediction. *CoRR*, vol. abs/1611.00144. 1611.00144.
Available at: http://arxiv.org/abs/1611.00144

Rajkomar, A., Oren, E., Chen, K., Dai, A.M., Hajaj, N., Liu, P.J., Liu, X., Sun, M., Sundberg, P., Yee, H., Zhang, K., Duggan, G.E., Flores, G., Hardt, M., Irvine, J., Le, Q.V., Litsch, K., Marcus, J., Mossin, A., Tansuwan, J., Wang, D., Wexler, J., Wilson, J., Ludwig, D., Volchenboum, S.L., Chou, K., Pearson, M., Madabushi, S., Shah, N.H., Butte, A.J., Howell, M., Cui, C., Corrado, G. and Dean, J. (2018). Scalable and accurate deep learning for electronic health records. *CoRR*, vol. abs/1801.07860. 1801.07860.
Available at: http://arxiv.org/abs/1801.07860

Rosenblatt, F. (1962). *Principles of neurodynamics: perceptrons and the theory of brain mechanisms*. Report (Cornell Aeronautical Laboratory). Spartan Books.
Available at: https://books.google.ca/books?id=7FhRAAAAMAAJ

Rumelhart, D.E., Hinton, G.E. and Williams, R.J. (1988). Neurocomputing: Foundations of research. chap. Learning Representations by Back-propagating Errors, pp. 696–699. MIT Press, Cambridge, MA, USA. ISBN 0-262-01097-6.
Available at: http://dl.acm.org/citation.cfm?id=65669.104451

Sarikaya, R. (2017 Jan). The technology behind personal digital assistants: An overview of the system architecture and key components. *IEEE Signal Processing Magazine*, vol. 34, no. 1, pp. 67–81. ISSN 1053-5888.

Selvaraju, R.R., Cogswell, M., Das, A., Vedantam, R., Parikh, D., Batra, D. *et al.* (2017). Grad-cam: Visual explanations from deep networks via gradient-based localization. In: *ICCV*, pp. 618–626.

Shavitt, I. and Segal, E. (2018 May). Regularization Learning Networks: Deep Learning for Tabular Datasets. *ArXiv e-prints*. 1805.06440.

Shi, W., Caballero, J., Huszár, F., Totz, J., Aitken, A.P., Bishop, R., Rueckert, D. and Wang, Z. (2016). Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1874–1883.

Shickel, B., Tighe, P., Bihorac, A. and Rashidi, P. (2017). Deep EHR: A survey of recent advances on deep learning techniques for electronic health record (EHR) analysis. *CoRR*, vol. abs/1706.03446. 1706.03446.
Available at: http://arxiv.org/abs/1706.03446

Shimodaira, H. (2000). Improving predictive inference under covariate shift by weighting the log-likelihood function. *Journal of Statistical Planning and Inference*, vol. 90, no. 2, pp. 227 – 244. ISSN 0378-3758.
Available at: http://www.sciencedirect.com/science/article/pii/S0378375800001154

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A. *et al.* (2017). Mastering the game of go without human knowledge. *Nature*, vol. 550, no. 7676, p. 354.

Smith, L.N. (2018). A disciplined approach to neural network hyper-parameters: Part 1 - learning rate, batch size, momentum, and weight decay. *CoRR*, vol. abs/1803.09820. 1803.09820.
Available at: http://arxiv.org/abs/1803.09820

Smith, L.N. and Topin, N. (2017). Super-convergence: Very fast training of residual networks using large learning rates. *CoRR*, vol. abs/1708.07120. 1708.07120.
Available at: http://arxiv.org/abs/1708.07120

Song, W., Shi, C., Xiao, Z., Duan, Z., Xu, Y., Zhang, M. and Tang, J. (2018). Autoint: Automatic feature interaction learning via self-attentive neural networks. *CoRR*, vol. abs/1810.11921. 1810.11921.
Available at: http://arxiv.org/abs/1810.11921

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958.

Srivastava, R.K., Greff, K. and Schmidhuber, J. (2015). Training very deep networks. In: *Advances in Neural Information Processing Systems 28*, pp. 2377–2385.

Sun, Y., Wang, X. and Tang, X. (2014). Deeply learned face representations are sparse, selective, and robust. *CoRR*, vol. abs/1412.1265. 1412.1265.
Available at: http://arxiv.org/abs/1412.1265

Utgoff, P. and Stracuzzi, D. (2002). Many-layered learning. *Neural Computation*, vol. 14, no. 10, pp. 2497–2529.

Van Der Maaten, L., Chen, M., Tyree, S. and Weinberger, K.Q. (2013). Learning with marginalized corrupted features. In: *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pp. I–410–I–418. JMLR.org.
Available at: http://dl.acm.org/citation.cfm?id=3042817.3042865

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L. and Polosukhin, I. (2017). Attention is all you need. *CoRR*, vol. abs/1706.03762. 1706.03762.
Available at: http://arxiv.org/abs/1706.03762

Vincent, P., Larochelle, H., Bengio, Y. and Manzagol, P.-A. (2008). Extracting and composing robust features with denoising autoencoders. In: *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pp. 1096–1103. ACM, New York, NY, USA. ISBN 978-1-60558-205-4.
Available at: http://doi.acm.org/10.1145/1390156.1390294

Wang, R., Fu, B., Fu, G. and Wang, M. (2017). Deep & cross network for ad click predictions. *CoRR*, vol. abs/1708.05123. 1708.05123.
Available at: http://arxiv.org/abs/1708.05123

Wu, Y., Schuster, M., Chen, Z., Le, Q.V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., Kaiser, L., Gouws, S., Kato, Y., Kudo, T., Kazawa, H., Stevens, K., Kurian, G., Patil, N., Wang, W., Young, C., Smith, J., Riesa, J., Rudnick, A., Vinyals, O., Corrado, G., Hughes, M. and Dean, J. (2016). Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, vol.

abs/1609.08144. 1609.08144.
Available at: http://arxiv.org/abs/1609.08144

Xu, K., Ba, J., Kiros, R., Cho, K., Courville, A.C., Salakhutdinov, R., Zemel, R.S. and Bengio, Y. (2015). Show, attend and tell: Neural image caption generation with visual attention. *CoRR*, vol. abs/1502.03044. 1502.03044.
Available at: http://arxiv.org/abs/1502.03044

Yosinski, J., Clune, J., Nguyen, A., Fuchs, T. and Lipson, H. (2015). Understanding neural networks through deep visualization. *arXiv preprint arXiv:1506.06579*.

Zeiler, M.D. and Fergus, R. (2014). Visualizing and understanding convolutional networks. In: *European conference on computer vision*, pp. 818–833. Springer.

Zhang, H., Cissé, M., Dauphin, Y.N. and Lopez-Paz, D. (2017). mixup: Beyond empirical risk minimization. *CoRR*, vol. abs/1710.09412. 1710.09412.
Available at: http://arxiv.org/abs/1710.09412

Zhang, W., Du, T. and Wang, J. (2016). Deep learning over multi-field categorical data: A case study on user response prediction. *CoRR*, vol. abs/1601.02376. 1601.02376.
Available at: http://arxiv.org/abs/1601.02376

Zhou, B., Khosla, A., Lapedriza, A., Oliva, A. and Torralba, A. (2016). Learning deep features for discriminative localization. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2921–2929.

Zhou, G., Song, C., Zhu, X., Fan, Y., Zhu, H., Ma, X., Yan, Y., Jin, J., Li, H. and Gai, K. (2017 June). Deep Interest Network for Click-Through Rate Prediction. *ArXiv e-prints*. 1706.06978.