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

March 2019

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)

March 2019

From about 2006, deep learning has proven to be very successul in application areas such as computer vision, natural language processing, speech and audio recognition, machine translation, bioinformatics, and social network filtering. These successes were undoubtedly facilitated by many advances in neural network architectures. In contrast, deep learning has not yet been found to excel in the context of tabular datasets.

Many key machine learning tasks make use of tabular data, where currently the best machine learning models for tabular data use classification or regression trees as base learners. Therefore, the objective of this study is to identify, discuss and explore recent developments in deep learning which may be used to enhance the accuracy of deep neural networks in the tabular data domain. All major developments in the deep learning field are discussed and critically considered, with a view to improving deep learning in the context of tabular data. The challenges of applying deep learning to tabular data are identified, and on each of these fronts, potential improvements are proposed.

The most promising modern deep learning architectures are further explored by means of empirical work. We also evaluate the validity of findings reported in the literature, and comment on the effectiveness of recent proposals. A useful byproduct of the study is the development of a code base that may be used to implement the latest deep learning techniques, as well as for comparative model selection experiments.

Uittreksel

Diepleer Tegnieke vir Gestruktrueerde Data: 'n Verkennende Studie

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

J. A. Marais

Tesis: MCom (Wiskundige Statistiek)

Maart 2019

Vanaf ongeveer 2006 is die sukses van diepleer-tegnieke in toespassings-areas soos rekenaarvisie, taalprosessering, spraak- en klankherkenning, masjienvertaling, bio-informatika, en om sosiale netwerk te filtreer, alombekend. Die sukses van diepleer-metodes is ongetwyfeld aangehelp deur baie ontwikkelings rondom die argitektuur van neurale netwerke. Nogtans is bevind dat diep neural netwerke tot dusver nie goed vaar in die konteks van die gebruik van gewone matriksvorm data nie.

Verskeie belangrike masjienleer take maak gebruik van matriksvorm data, waar die beste masjienleer modelle in hierdie konteks klassifikasie- of regressiebome gebruik as basis. Derhalwe is die doelwit van hierdie studie om onlangse ontwikkelings in diepleer (wat gebruik kan word om die akkuraatheid van diep neural netwerke te verbeter in die konteks van matriksvorm-data), te identifiseer, te bespreek, en empiries te ondersoek. Alle belangrike ontwikkelings in die diepleer veld word bespreek, en krities beskou, ten einde diepleer te verbeter in die konteks van matriksvorm data. Die uitdagings wat die toepassing van diepleer op matriksvorm data bied, word geidentifiseer, en op elkeen van hierdie fronte word potensiële verbeterings voorgestel.

Die belowendste moderne diepleer argitekture word deur middel van empiriese werk verder verken. Ons evalueer ook die geldigheid van bevindings wat in die literatuur rapporteer word, en lewer kommentaar op die effektiwiteit

van onlangse voorstelle. 'n Nuttige byproduk van die studie is die ontwikkeling van 'n kodebasis wat gebruik kan word vir die implementering van die nuutste diepleer-tegnieke, asook vir vergelykende eksperimente rondom modelseleksie.

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

List of Figures

List of Tables

List of Abbreviations and/or Acronyms

ANN	Artificial Neural Network
CNN	Convolutional Neural Network
CTR	Click-through Rate
CV	Computer Vision
DNN	Deep Neural Network
DL	Deep Learning
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
OLS	Ordinary Least Squares
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
SLT	Statistical Learning Theory

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 Figure 1.1). 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
[2] https://trends.google.com/trends/
[3] https://www.semanticscholar.org/

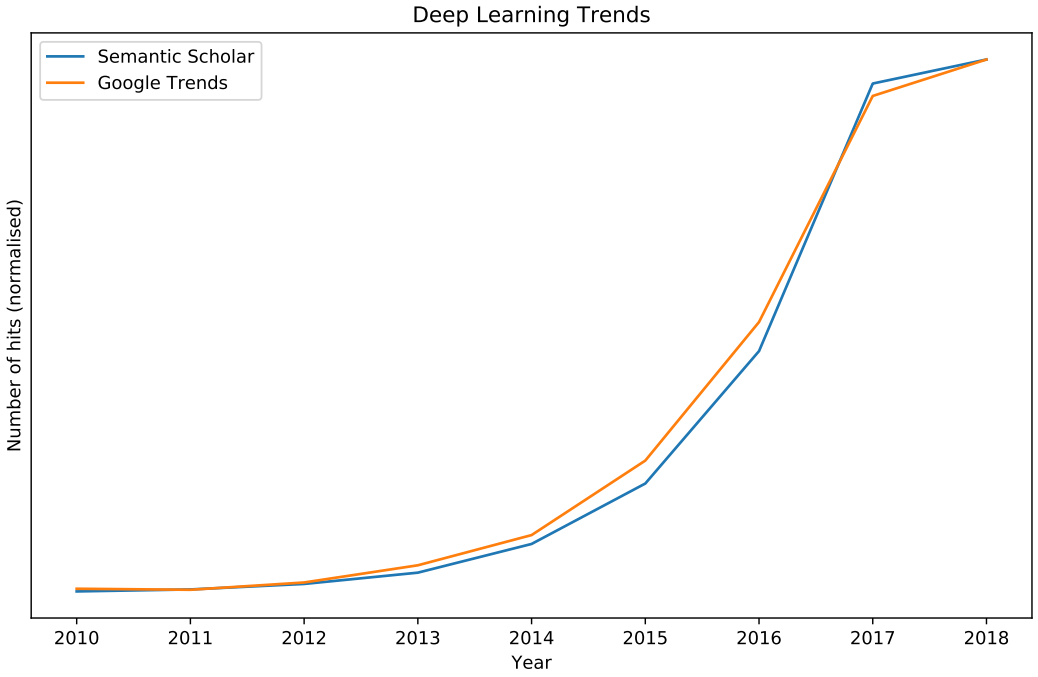


Figure 1.1: The exponential growth of published papers and Google search terms containing the term *Deep Learning*. Sources: Google Trends[2], Semantic Scholar[3]

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[4]). 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[5]. 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[6] 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

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

where the aim was to predict whether or not an indivdual earns more than $50,000 a year.

	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.

Table 1.1 represents a typical tabular dataset, where the columns contain measurements on different features. Therefore different columns may contain 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[7];
- 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

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

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 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 (DNNs) 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[8], 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

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

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, 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 [9].

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,

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

$j = 1, \ldots, p$. 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 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

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

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 $\hat{y} = \hat{\boldsymbol{w}}^\intercal \boldsymbol{x}$.

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

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

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, $\boldsymbol{w}^\intercal \boldsymbol{w}$, 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 $\hat{f}_k(\boldsymbol{X}) = \hat{\boldsymbol{w}}_k^{\intercal}\boldsymbol{X}$ 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 1.2 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 $\hat{f}_k(\boldsymbol{x}) = \hat{\boldsymbol{w}}_k^{\intercal}\boldsymbol{x}$ 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.

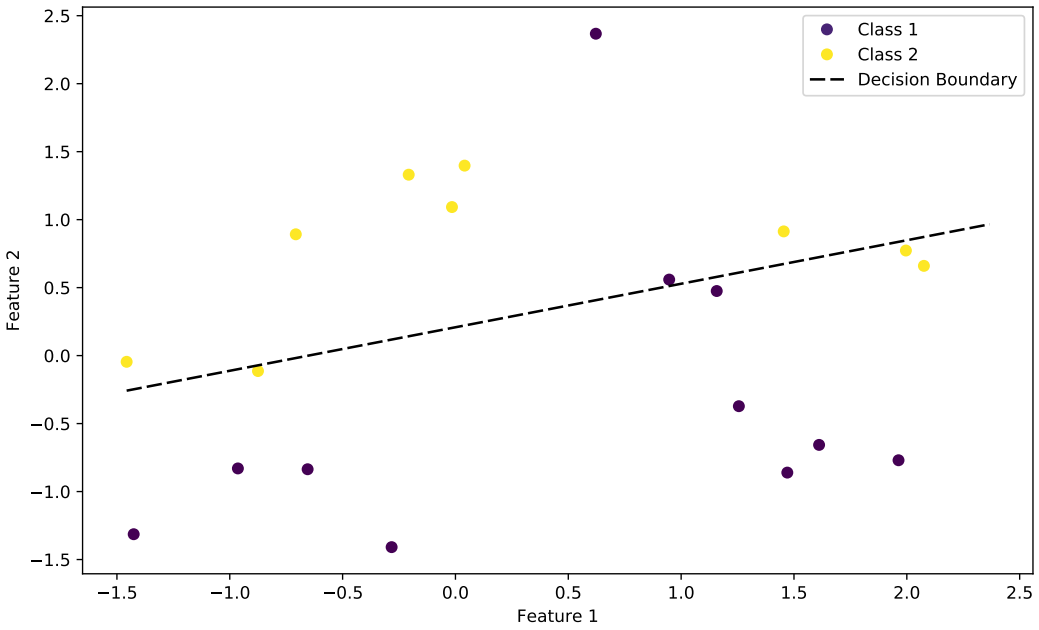


Figure 1.2: Linear model on simple binary classification dataset.

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,

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

Through this transformation, the estimates of the posterior probabilities sum to 1 and are contained in [0,1]. The above model is the well-known *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[1].

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

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

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.

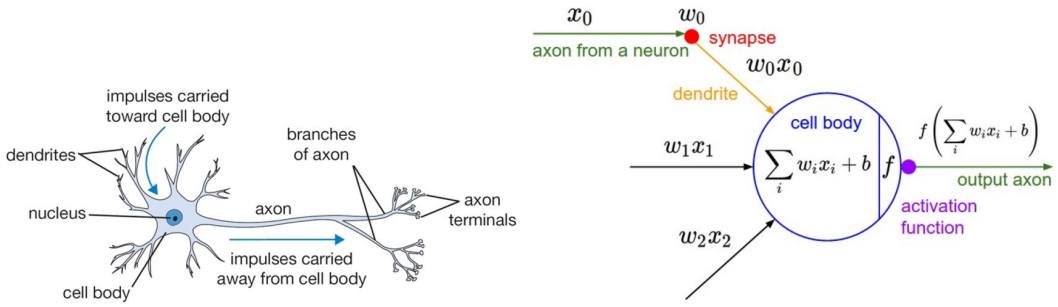
2.2 The Structure of a Neural Network

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).



(a) Biological (b) Artificial

Figure 2.1: Neuron comparison.

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, 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.

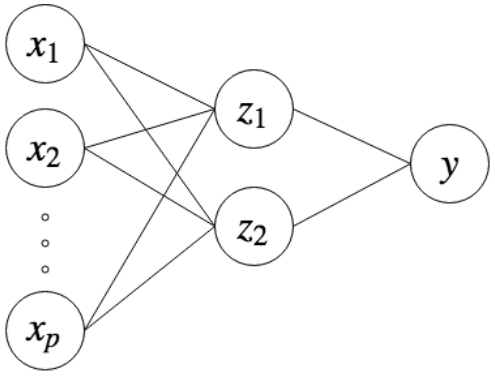


Figure 2.2: A simple neural network accepting p-sized inputs, with one hidden layer consisting of two neurons.

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

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

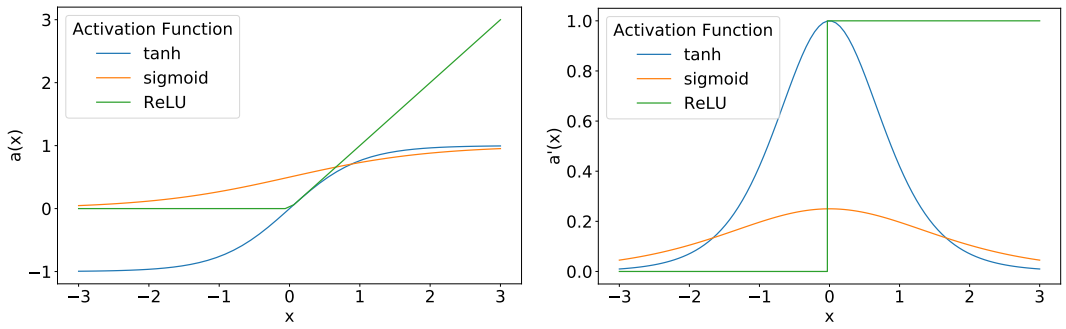
where a_1 denotes the activation function applied after the first (linear) layer, and where a_2 denotes the activation function applied after the final layer.

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



(a) Function (b) Local Derivative

Figure 2.3: Activation functions.

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 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 activation functions to apply include PReLUs (He *et al.*, 2015*a*) and Leaky ReLUs (Maas

et al., 2013).

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 linear 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 for a binary classification task. 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.

Note, that there also exists other classes of neural networks besides the basic feed-forward network focussed on thus far. The *convolutional neural network* (CNN) is a very common choice in computer vision (Krizhevsky *et al.*, 2012) and only recently in NLP applications (Devlin *et al.*, 2018). Its main difference to standard NNs is the use of a convolutional layer that performs a cross-correlation operation on its inputs with a set of learnable filters. The *recurrent neural network* (RNN) (Mikolov *et al.*, 2010) is a class of neural networks that are helpful for modelling variable length inputs or outputs, for example in machine translation or video classification. A RNN has loops in

them, allowing information to persist. It can also be thought of as multiple copies of the same network, each passing a message to its successor, if one were to "unfold" the loops. These classes of neural networks are not within the scope of this work since they have no clear benefit for using with tabular data.

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 measured 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. A poorly initialised network hampers the training procedure: it increases the number of iterations needed, reduces the quality of the local optima found and makes convergence more difficult. 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 to scale the weight vector generated from the standard normal 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.

Very deep neural networks have difficulties to converge when using the above mentioned initialisation (Simonyan and Zisserman, 2014). He *et al.* (2015*b*) proposes an alternative initialisation, which was made specifically in the context of using ReLU activation functions and making it easier to train deep neural networks.

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:

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

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.*:

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

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

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

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 case 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 $\{\boldsymbol{x} : \boldsymbol{x}^\intercal \hat{\boldsymbol{w}} = 0.5\}$.

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.

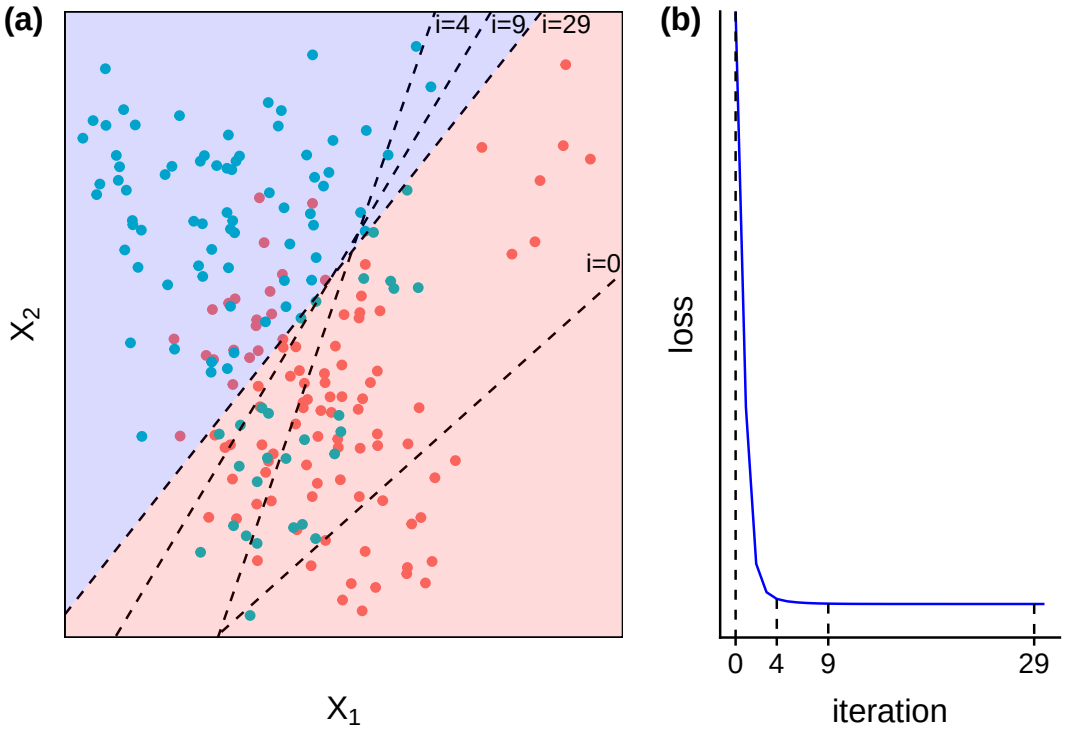


Figure 2.4: Plots of the gradient descent example. (a) The training data points in input space. The shades in the background represent the class division in input space, with the decision boundary determined by least squares estimation. The dashed lines represent the gradient descent decision boundaries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

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:

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

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:

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

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

$$\begin{aligned} \frac{\partial L_i}{\partial \beta_{km}} &= -2(y_{ik} - f_k(\boldsymbol{x}_i)) g_k'(\boldsymbol{\beta}_k^\intercal \boldsymbol{z}_i) z_{mi}, \\ \frac{\partial L_i}{\partial \alpha_{ml}} &= -\sum_{k=1}^{K} 2(y_{ik} - f_k(\boldsymbol{x}_i)) g_k'(\boldsymbol{\beta}_k^\intercal \boldsymbol{z}_i) \beta_{km} \sigma'(\boldsymbol{\alpha}_m^\intercal \boldsymbol{x}_i) x_{il}. \end{aligned}$$

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

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

We may now rewrite the gradients as follows:

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

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

$$s_{mi} = \sigma'(\boldsymbol{\alpha}_m^\intercal \boldsymbol{x}_i) \sum_{k=1}^{K} \beta_{km} \delta_{ki},$$

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 practice, L2 regularisation typically outperforms L1 regularisation in neural network applications.

An alternative way of preventing a neural network to overfit is so-called *early stopping* 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 stopping involves terminating 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 down 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 method. SGD maintains a single learning rate for all weight updates in the network during training (global). However, one may wish to maintain individual learning rates for different weights dependending on some behaviour or property exhibited by the weight during training (local) in order to speed up training and reach better local optima. *Adagrad* (Duchi *et al.*, 2011) is an adaptive learning rate method which magnifies the learning rate at neurons with small gradients and shrinks the learning rate at neurons with large gradients. *Adam* (Kingma and Ba, 2014) is the most commonly used weight update approach and builds on Adagrad by incorporating momentum (Bengio *et al.*, 2012). 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. The algorithm updates exponential moving averages of the gradient, m_t, and the squared gradient, v_t, where the exponential decay rates of these moving averages are controlled by the hyperparameters $\beta_1, \beta_2 \in [0, 1)$. Since the moving averages are initialised as zero, the gradient and the squared gradient moving averages are biased corrected ($\hat{m}_t, \hat{v}_t$). The weight update for Adam is given by the following equations:

$$\theta_t \leftarrow \theta_{t-1} - \gamma \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

where

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$$

and

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}, \quad v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2,$$

where g_t is the vector of partial derivatives of f_t w.r.t. θ evaluated at time step t. Adam was found to empirically outperform all other optimisation procedures in Ruder (2016).

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).

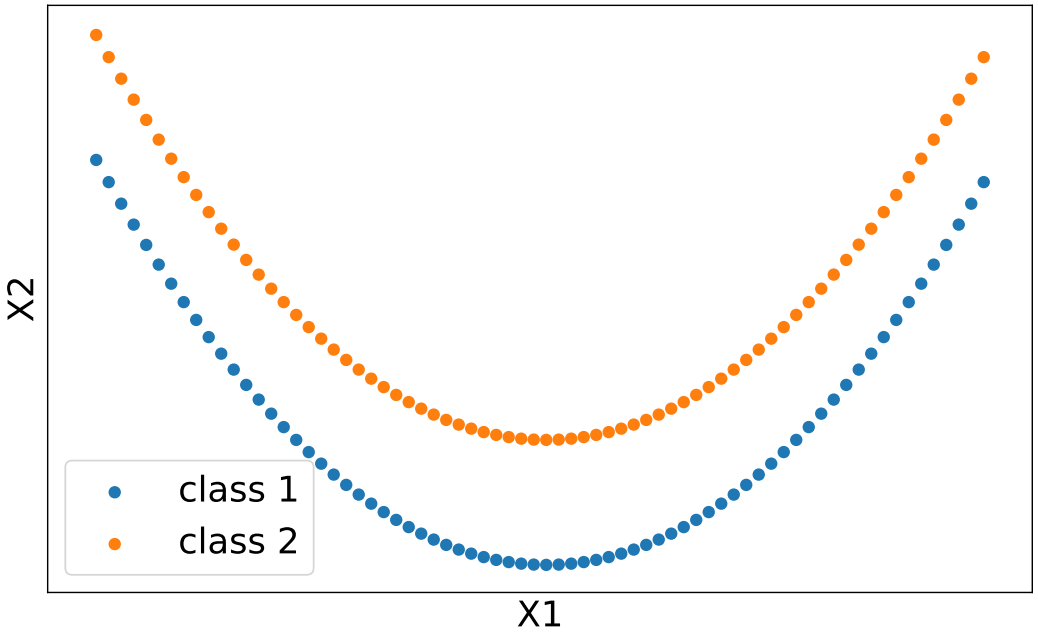


Figure 2.5: Simple dataset with two linearly inseparable classes.

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 Figure 2.8 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.

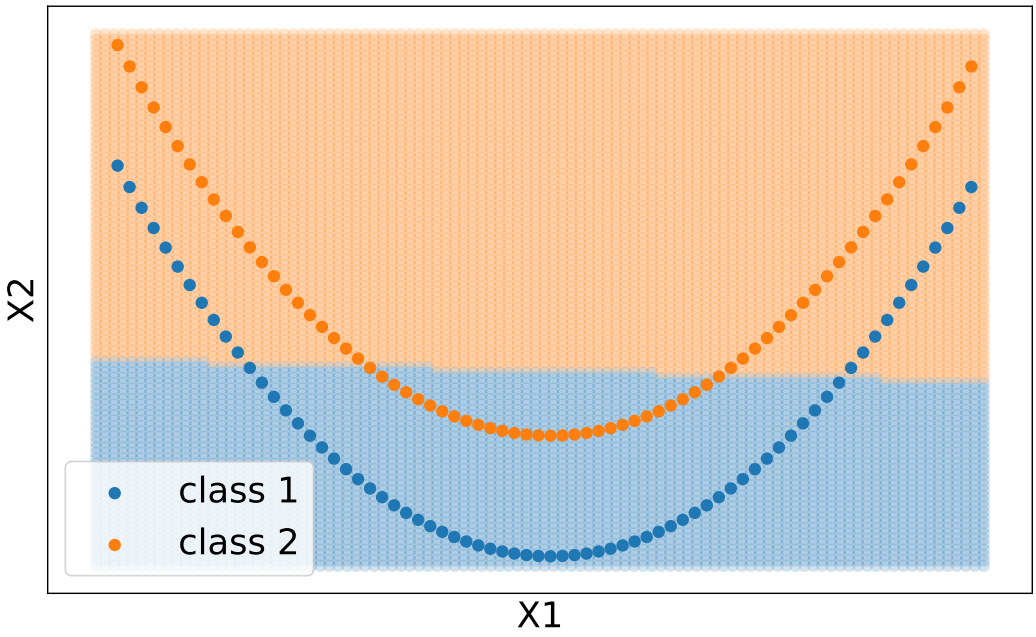


Figure 2.6: Decision boundary of a single-layer neural network.

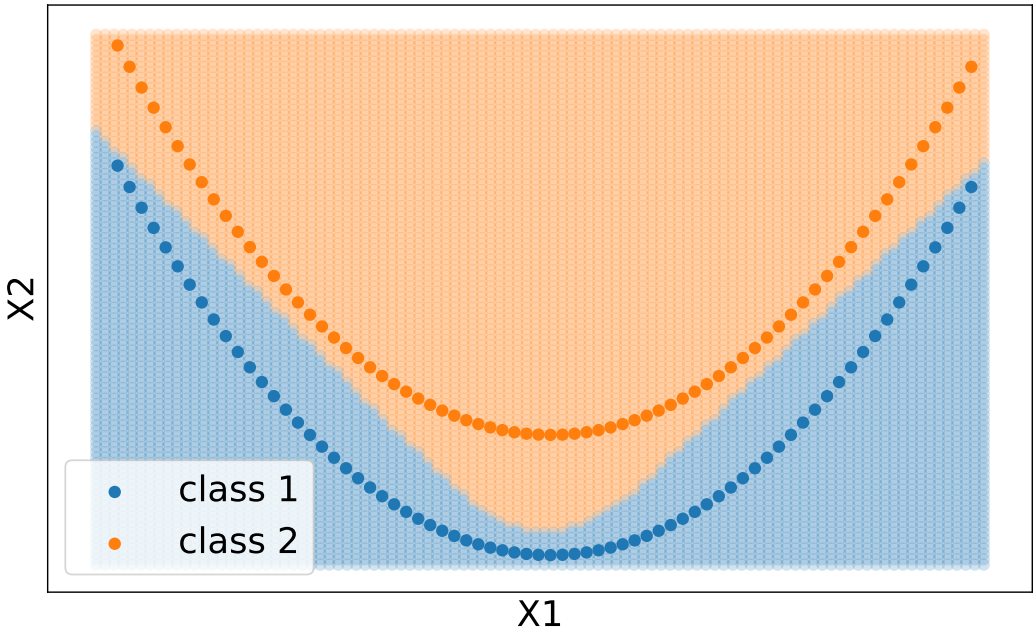


Figure 2.7: Decision boundary of a two-layer neural network.

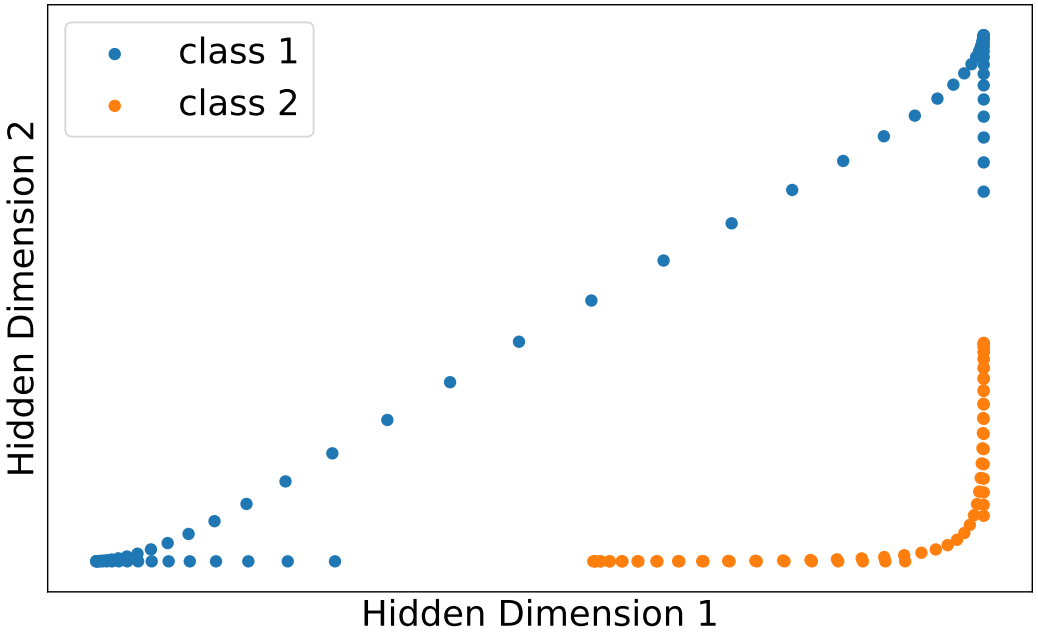


Figure 2.8: Hidden representation of a two-layer neural network.

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, 1966). 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 Rumelhart and McClelland (1986), Hinton (1990), 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), Hinton and Salakhutdinov (2006), Ranzato *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 improving deep learning in the context of tabular data. The developments

chosen to be focussed on are a result of our own extensive research and practical experience in the field deep learning.

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. More detail regarding transfer learning is provided in §3.3.

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 group of modern deep learning architectural 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 better regularisation, decreased training time, reduced sensitivity to starting weights, improved accuracy, representing and visualising categorical variables, more robust representation learning and more useful interpretability of NNs. The use of these modern architectures are present in almost all of the SotA deep learning models in the computer vision and NLP domains.

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 Goodfellow *et al.* (2014), the idea of autoencoders have been around since the late 1980's - see for example Lecun (1987), Bourlard and Kamp (1988) and Hinton and Zemel (1994). 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), *sparse autoencoders* (Makhzani and Frey, 2013), *contractive autoencoders* (Rifai *et al.*, 2011), *semi-supervised recursive autoencoders* (Socher *et al.*, 2011), 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 for reference.

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, with a slight change in the type of loss function to be minimised. The loss function used for a common autoencoder is called the *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 labelled data, 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

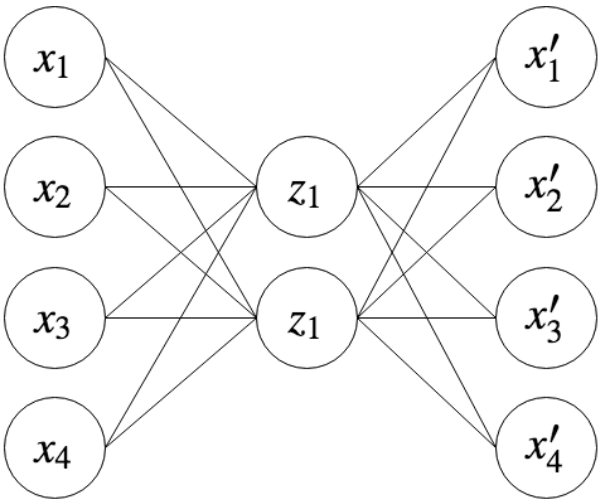


Figure 3.1: A simple single hidden layer autoencoder with four-dimensional inputs and with two neurons in the hidden layer.

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 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 the 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 Vincent *et al.* (2010): "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) of dataset D, and by then using those weights of the trained network as an initialisation when trying to predict a different target variable (say y') of dataset D'. This is most effective when there is overlap between the structure, semantics and properties of D and D'.

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.

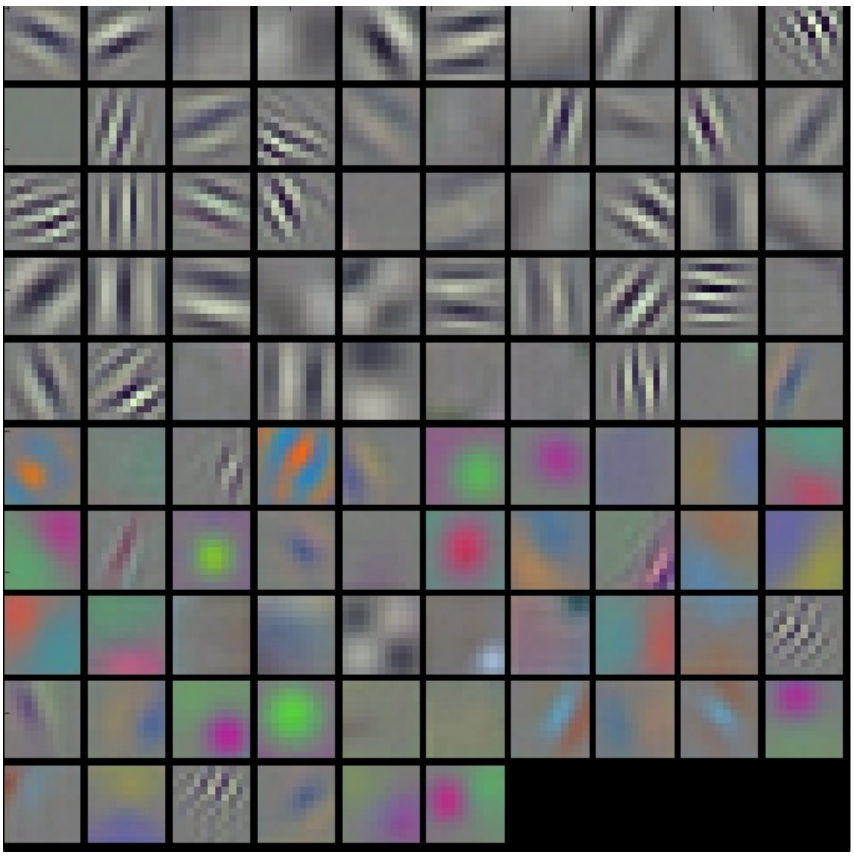


Figure 3.2: Visualising the first layer convolutional filters leared by a neural network in a large image dataset.

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 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 (Srivastava *et al.*, 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. Consider a NN with L hidden layers, each indexed with $l \in \{1, \ldots, L\}$. Let $\boldsymbol{z}^{(l)}$ and $\boldsymbol{z}^{(l+1)}$ denote the vector of inputs and outputs to layer l respectively. $W^{(l)}$ and $\boldsymbol{b}^{(l)}$ are the weights and biases at layer l. Then the feed-forward operation for any hidden unit i can be expressed as

$$z_i^{(l+1)} = a\left(\boldsymbol{w}_i^{(l+1)}\boldsymbol{z}^{(l)} + b_i^{(l+1)}\right),$$

where $a(\cdot)$ is any activation function. With dropout, this operation becomes

$$\begin{aligned} r_j^{(l)} &\sim \text{Bernoulli}(p) \\ \tilde{\boldsymbol{z}}^{(l)} &= \boldsymbol{r}^{(l)} * \boldsymbol{z}^{(l)} \\ z_i^{(l+1)} &= a\left(\boldsymbol{w}_i^{(l+1)}\tilde{\boldsymbol{z}}^{(l)} + b_i^{(l+1)}\right) \end{aligned}$$

$\boldsymbol{r}^{(l)}$ is a vector of independent Bernoulli random variables each of which has probability p of being 1. This vector is sampled and multiplied element-wise $(*)$ with the outputs of that layer to create the thinned outputs $\tilde{\boldsymbol{z}}^{(l)}$. The thinned outputs are then used as inputs to the next layer. At test time the weights are scaled as $W_{test}^{(l)} = pW^{(l)}$.

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.

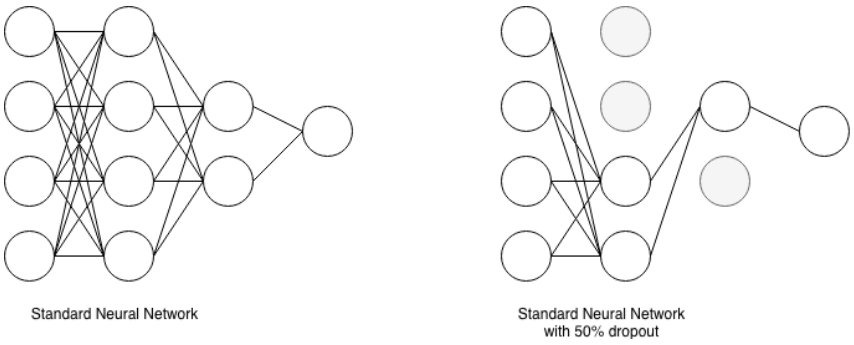


Figure 3.3: The effect that dropout has on connections between neurons.

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 ability to generalise. 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.

Data augmentation may be formalised by means of the *Vicinal Risk Min-*

Figure 3.4: An example of data augmentation for images.

imisation 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:

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

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 their paper. The original sought after function $H(\boldsymbol{x})$ may subsequently be found via the mapping $H(\boldsymbol{x}) = F(\boldsymbol{x}) + \boldsymbol{x}$.

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.

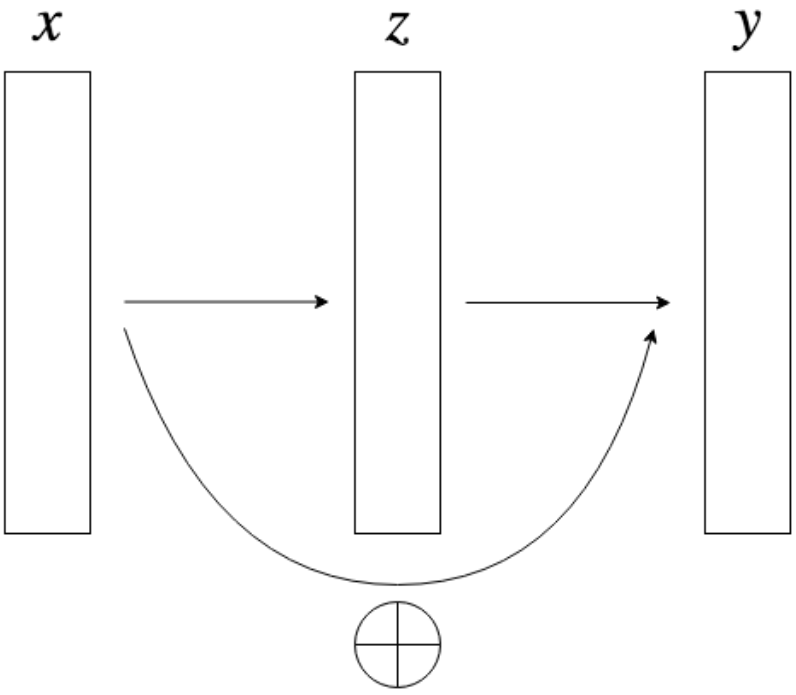


Figure 3.5: Diagram conceptualising a skip connection.

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).

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].

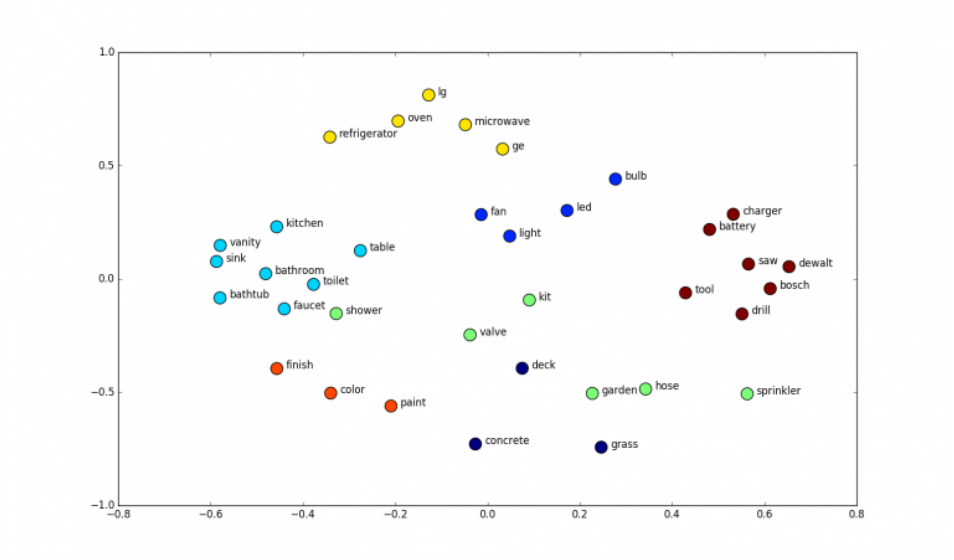


Figure 3.6: Learned word embeddings in a two-dimensional space.

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.

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

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 (Source: Xu *et al.* (2015)). 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 (Source: Bahdanau *et al.* (2014))).

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

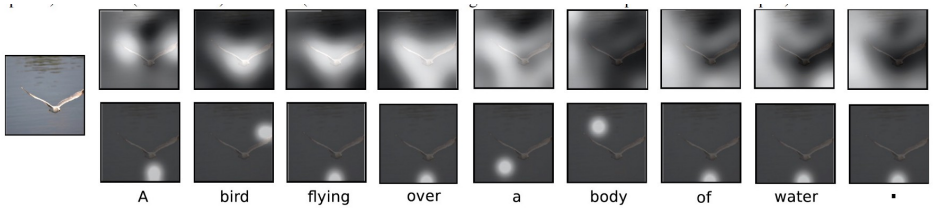


Figure 3.7: Attention applied to image captioning.

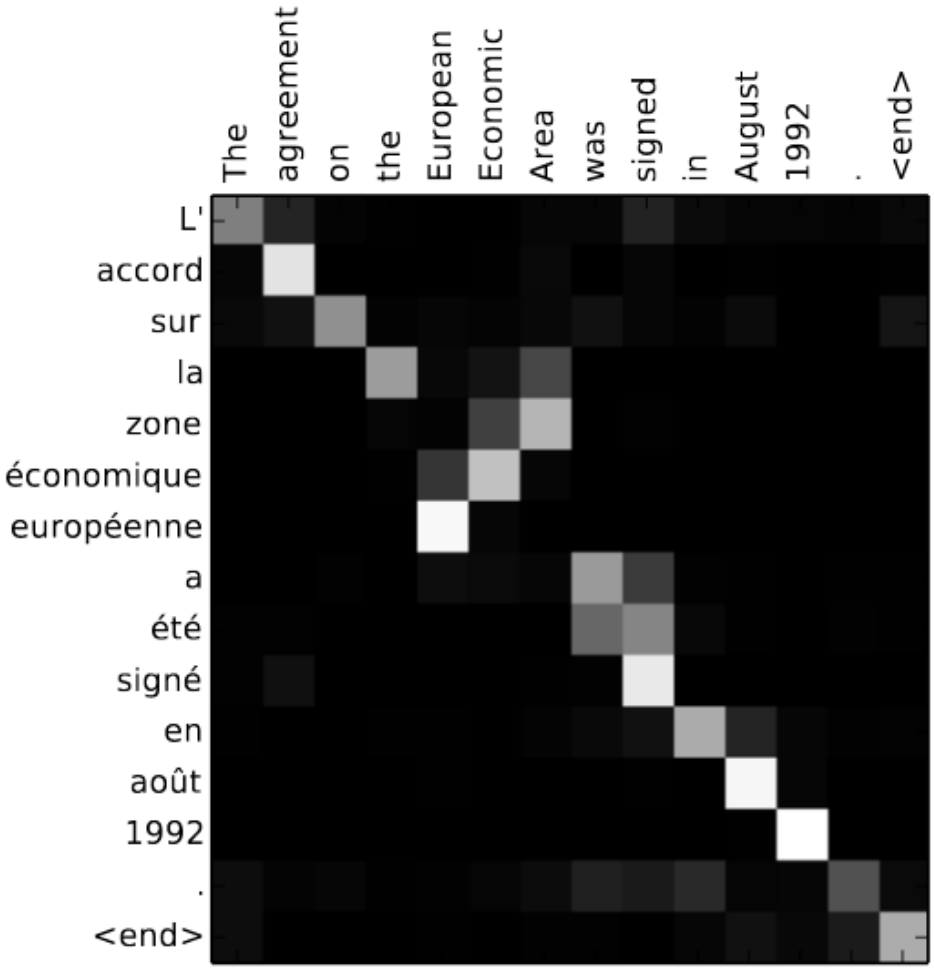


Figure 3.8: Attention applied to machine translation.

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 $\boldsymbol{\alpha}$, 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 Smith (2017), 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, Smith and Topin (2017) 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 Figure 3.9.

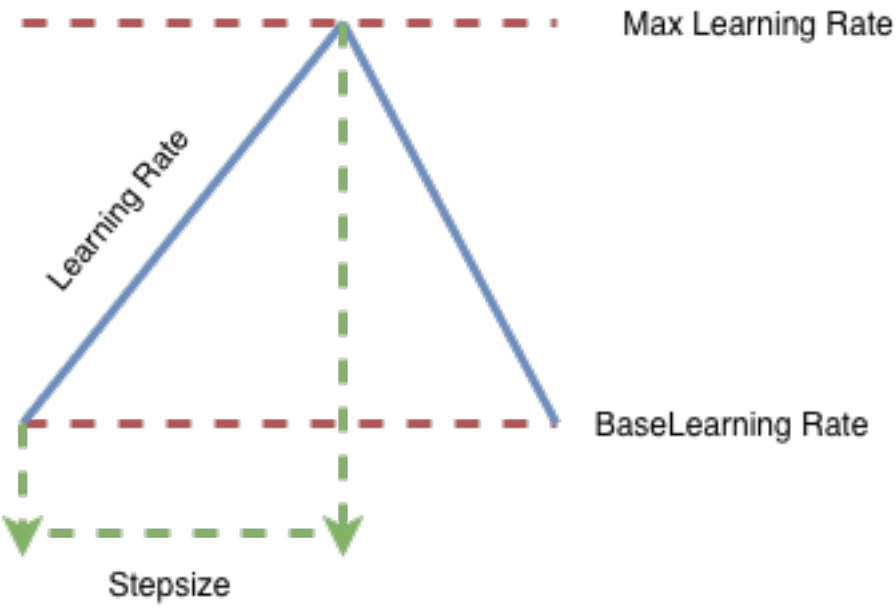


Figure 3.9: The learning rate schedule of the 1cycle policy.

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 3.10 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.

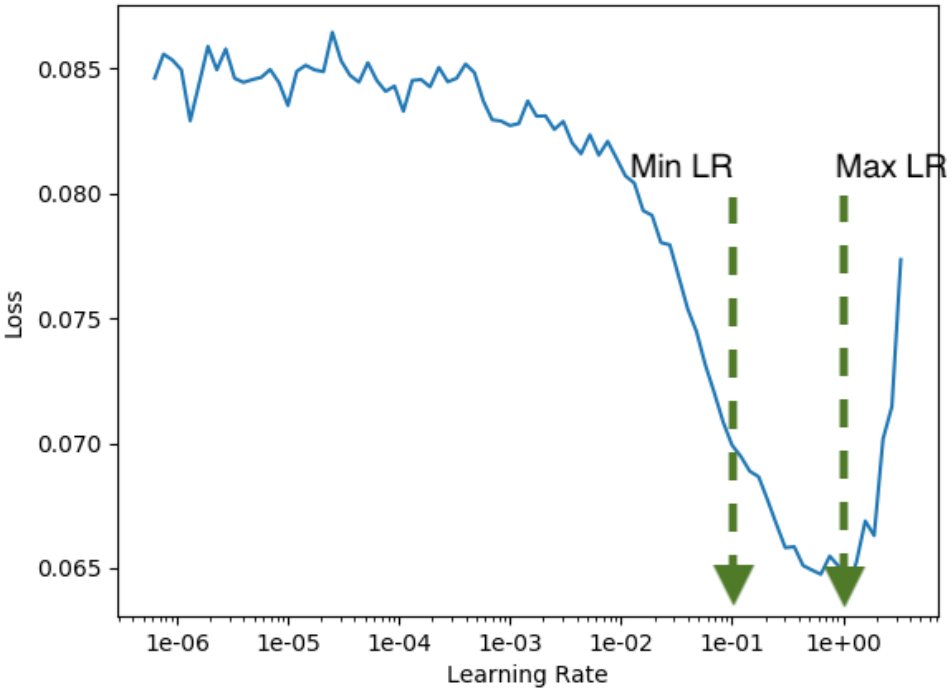


Figure 3.10: An example output of a learning rate range test.

Next consider Figure 3.11, 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 3.11 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 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

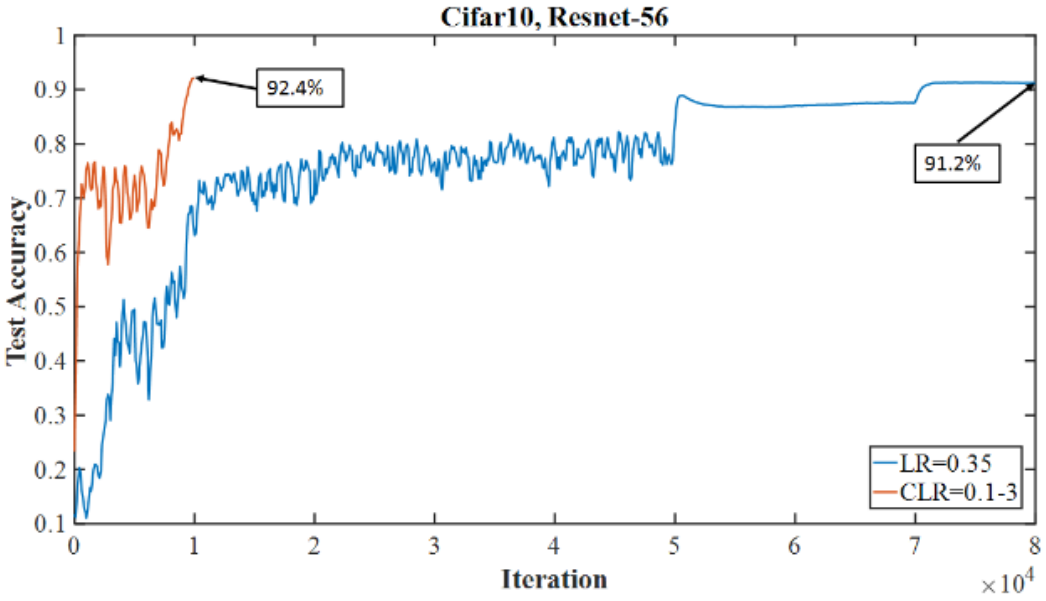


Figure 3.11: Reduced training iterations and improved performance facilitated by the super-convergence principle.

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 above 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. 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 Lundberg and Lee (2017). 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 computer vision, NLP and audio processing. However, in the literature only a handful of publications report successful implementation of deep learning for tabular data. In these papers, 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 illuminate this issue,

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

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*), 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

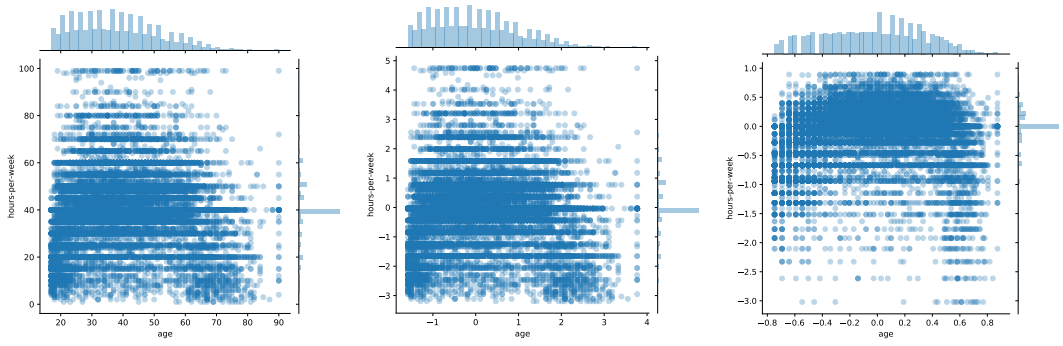
4.2.1 Numerical Features

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 measured 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 μ 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 to use 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 (*i.e.* features with a large number of categories), 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, no 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 is the use of *entity embeddings*. The first publication on entity embeddings in the context of deep learning followed as a result of a taxi destination prediction challenge (de Brébisson *et al.*, 2015). This was followed by a paper by Guo and Berkhahn (2016), wherein the authors succesfully used entity embeddings in order to predict the total sales of a store. Furthermore, companies such as Instacart and Pinterest report the effective use of entity embeddings on their internal datasets. Currently all research on deep learning for tabular data facilitate entity embeddings - see for example Song *et al.* (2018), Wang *et al.* (2017), Covington *et al.* (2016) and Zhou *et al.* (2017). The reason for their all-round use is the way in which they are able to ameliorate the issues of one-hot encoded representations.

An entity embedding entails exactly the same operations as with (word) embeddings (discussed in §3.5.3), only they are applied to categories instead of words. Therefore, an entity embedding assigns a numeric vector representation

to each level in a categorical feature. For example, Level 1 is embedded as $[0.05, -0.1, 0.2]$; whereas Levels 2 and 3 are embedded as $[0.2, 0.01, 0.3]$ and $[-0.1, -0.2, 0.05]$ respectively. In a formulation similar to the one in §3.5.3, note that the embedding for the j-th categorical feature is defined by

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

where $\boldsymbol{x}_j$ denotes the one-hot encoded vector representation of the j-th categorical input, where 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. Once all categorical features have been embedded, their representations may be concatenated and passed on to the network layers that follow.

Entity embeddings have been found to significantly speed up the training process and to reduce the memory footprint of a neural network. This in turn serves to further improve the generalisation ability of a network as stated in Covington *et al.* (2016) and Guo and Berkhahn (2016). The above advantages are especially useful when working with high-dimensional and sparse inputs (in tabular data). Suppose we have available a dataset with two categorical features, X_1 and X_2, with cardinality C_1 and C_2, respectively. Furthermore, suppose that the first hidden layer in the neural network is presented with inputs of size q. Hence, we need to 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 create two weight matrices of sizes $C_1 \times q/2$ and $C_2 \times q/2$, which is half the number of parameters needed in the case of one-hot encoded representation.

The size of an embedding (*i.e.* the number of features that represent similarities between inputs) is a hyperparameter of the model and again there is no way to specify this value *a priori*. Therefore, most publications rely on a grid search in order to find the optimal size. For example, Song *et al.* (2018) experiment with embedding sizes 8, 16, 24 and 32; and found the value 16 to work the best, whereas Cheng *et al.* (2016*a*) reported an embedding size of 32 to be optimal in their use-case. Thus, it is clear that appropriate specification of the size hyperparameter depends on the data at hand, and on the network architecture to be used.

Wang *et al.* (2017) and de Brébisson *et al.* (2015) made use of different embedding sizes for each categorical feature. Intuitively it seems to make sense

to incorporate different embedding sizes for categorical features with different levels of complexity. In the above papers, the following sizes are proposed:

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

In addition to advantages in terms of speed and memory usage, the use of entity embeddings as opposed to one-hot encoding maps similar values to lie close to each other in the embedding space. Hence it reveals the intrinsic properties of categorical variables, which cannot be obtained by using one-hot encoding. This allows us to interpret the classes of the categorical features. Embeddings may also easily be visualised, thereby facilitating interpretation of the data and of the decision making process of the network. In more detail, the weights associated with the projection of each category onto the embedded space may be plotted using any dimension reduction technique (such as t-sne or PCA). We may subsequently compare categories based on relative distances and positions in the reduced space. To illustrate, reconsider the 'Education' categorical feature in the Adult dataset (introduced in Chapter 2). In Figure 4.2 a two-component PCA of its embedding matrix is plotted. From this figure it is clear 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 lie in a separate cluster and very conveniently, their levels of education coincide with the vertical axis.

In order to verify whether entity embeddings are able to learn useful information, besides plotting the embedding matrix, one may also feed them (along with the continuous features) to other learning algorithms and observe the change in model performance. Along these lines Guo and Berkhahn (2016) tested the use of embeddings from trained NNs as inputs to a set of machine learning methods. They report that the use of embeddings improved the performance of all tested ML procedures by a significant margin. Moreover, these embeddings may be re-used in different machine learning tasks, obviating the need for them to be re-learned in other data scenarios.

The entity embedding approach is very flexible. If, for example, two features have overlapping categories, one may re-use the embedding for the one feature on the second feature. Zhou *et al.* (2017) holds and interesting view of the case of multi-hot categorical features. That is, where an observation of a categorical feature may simultaneously be assigned more than one level. The embedding

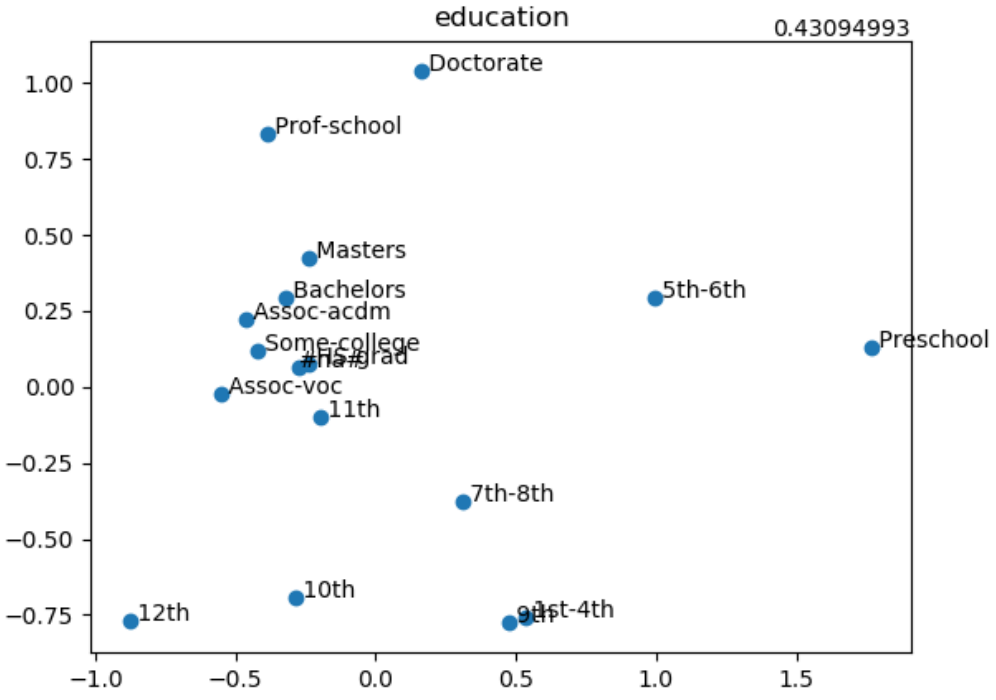


Figure 4.2: PCA of the 'Education' entity embedding weight matrix.

layer for that instance then outputs a list of embeddings having the same length as the number of categories associated with that instance and feature. Using some pooling operation, the list of embeddings are subsequently projected back to a fixed-length representation.

4.2.3 Combining Features

Once all continuous and categorical features have been processed and embedded, we need a way to combine them before their input may be transmitted 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. This is illustrated in Figure 4.3. The potential problem with this approach, however, is that some features might be over-represented in the resulting vector. Suppose for example that one of the continuous features are very important for prediction. When the continuous feature is concatenated with the entity embeddings, since all entity embeddings each take up more space in the combined representation, it may happen that

the contribution of the continuous predictor may be diluted.

In Song *et al.* (2018) the authors embed both numerical and categorical features into the same embedded space. Mapping both types of features into the same feature space facilitates more effective learning of interactions between the mixed set of features. Here 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 where $\boldsymbol{v}_j$ denotes the associated weight vector.

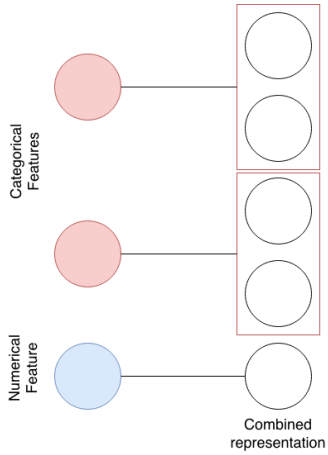


Figure 4.3: Combined representation of continuous and categorical features.

4.3 Learning Feature Interactions

In most machine learning tasks it is known that the greatest performance gains are typically achieved by means of feature engineering, whereas improved algorithms often only result in incremental performance boosts. In feature egineering one strives to use the original features to create a new set of features. This is done using domain expertise or by exploiting *a priori* knowledge of the data at hand. Using more informative features obtained in this way facilitates simpler estimation of the target, and capturing high-order interactions between features. Hence feature engineering is viewed as a crucial step in order to learn as much as possible from a training dataset. Unfortunately it entails a very laborious process.

Indeed, it is widely stated that in predictive modelling, typically 80% of the effort is devoted to steps involved during preprocessing, merging, customising,

and cleaning of datasets (Rajkomar *et al.*, 2018). Perhaps this is party due to the fact that feature engineering is not a structured process and usually consists of many unsuccesful trials and experimentation before more useful features are found. The effort associated with feature engineering could have been alleviated by more domain knowledge, but this is not always readily accessible.

A huge advantage of using neural networks on tabular data (and other data structures) is that the feature engineering process can be automated to some extent. That is, a neural network can learn these optimal feature transformations and interactions implicitly during the training process. In this way, the hidden layers of a neural network may be viewed as a feature extractor which is optimised to map the network inputs into the best possible feature space for the final layer of the network to operate in.

The traditional approach towards automatic feature engineering in NNs is to stack a few fully-connected layers in order to map the input representation to the output, as was done in Covington *et al.* (2016). The set of fully connected layers are used to implicitly model all feature interactions and in ideal world, we would expect this architecture to be sufficient. In practice however, without 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, thereby increasing the risk of falling into local optimums.

Therefore, it is necessary to explicitly leverage expressive feature combinations, or to encourage the network to learn better high-order feature interactions. The above objective 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 that we impose on a fully connected structure may further assist in limiting the size of the network, in turn causing learning to be more efficient.

Having motivated the importance of feature engineering, and emphasised the importance of further developing feature extraction methodology for neural networks, in the remainder of this section we focus on ways to assist a network to determine which features to combine in order to form meaningful high-order features. We briefly review some of the suggestions in the literature, whereafter we discuss the methodology which we believe to be the most powerful in terms of learning feature interactions.

In Wang *et al.* (2017), the authors makes a case for finding a bounded-degree of feature interactions. They argue that all Kaggle competions are

won when feature engineering comprises low-degree interactions. This stands in contrast to automatic feature engineering in deep neural networks, where highly non-linear interactions are typically learned. Therefore, in Wang *et al.* (2017) an automated way of constructing cross-features is proposed, and called a *cross-network*. A feature cross is a synthetic feature that encodes nonlinearity in the feature space by multiplying two or more input features together. In a cross-network, each layer produces higher-order interactions based on existing ones, and keeps the interactions from previous layers. More specifically, a cross-network consists of cross-layers which may be formalised as follows:

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

where $\boldsymbol{x}_l$ denotes the output from the l-th cross-layer and the input to the $(l+1)$-th cross-layer, and where $\boldsymbol{x}_0$ is the combined input representation. Furthermore, $\boldsymbol{w}_l$ and $\boldsymbol{b}_l$ represent the associated weight and bias parameters respectively. Each cross-layer adds back its input after a feature crossing in the same fashion as a skip-connection. Hence the degree of cross-features grows with increasing cross-network depth. The authors experimented with 1-6 cross-layers and found a depth of six to yield the best results. In parallel with the cross-networks in Wang *et al.* (2017), the authors also train a standard deep neural network in order to learn highly non-linear feature interactions. The DNN accepts the same input, and its output is then concatenated with that of the cross-network.

In Qu *et al.* (2016), the idea of a so-called *product layer* is proposed. A product layer calculates pairwise inner or outer products of all feature combinations and concatenate them to all linear combinations. The output is then sent to two fully connected layers.

In contrast to the approach in Wang *et al.* (2017), in Guo *et al.* (2017) and Cheng *et al.* (2016*a*) the authors aim to capture both low and high-order interactions. In both papers this is achieved by means of introducing two parallel networks, and learn the high-order interactions by means of a deep neural network. For the low order interactions, Cheng *et al.* (2016*a*) uses a shallow but wide neural network, and Guo *et al.* (2017) use a separate supervised learning model called a *factorisation machine* (Rendle, 2010). Again the output of the two streams are concatenated, and the resulting output is transmitted to the classification layer.

4.3.1 Attention

Based on the results presented in the paper by Song *et al.* (2018) and on findings reported in deep learning research in general, we deem attention to be the most promising mechanism to model feature interactions. In this section, note that we focus on a recent contribution by Song *et al.* (2018). In this paper a so-called *multi-head self-attention* mechanism is used. This mechanism is referred to as the *interacting layer*. Broadly, within the interacting layer each feature is allowed to interact with every other feature. During training, the interacting layer then automatically determines which of those interactions are relevant to the output. Multiple identical self-attention modules ("heads") are used, but each allowed to have different weights since a feature is likely involved in different combinatorial features.

In order to further explain the attention mechanism, consider a feature j and suppose that we want to determine which high-order interactions (which involve feature j) are meaningful. We start by first defining the correlation between features j and k under attention head h as follows:

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

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

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

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

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

where $\tilde{\boldsymbol{e}}_j^{(h)}$ denotes the 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 after undergoing a logit transform (or softmax operation). From the above, $\tilde{\boldsymbol{e}}_j^{(h)}$ thus denotes

a learned combinatorial feature. Since a feature may be involved in various different combinations, we use multiple heads to extract multiple combinations, *i.e.* $\{\tilde{\boldsymbol{e}}_j^{(h)}\}_{h=1}^{H}$, where H is the number of self-attention heads chosen. In Song *et al.* (2018) the value of H is set equal to 2, but this is typically a hyperparemeter that needs to be fine-tuned. After finding the combinatorial features, they are all concatenated into a single vector, *viz.* $\tilde{\boldsymbol{e}}_j$. The output is then finally combined with the original raw input, and sent through a ReLU activation:

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

Note that the above mapping from $\boldsymbol{e}_j$ to $\boldsymbol{e}_j^{\text{res}}$ is performed for each feature. In this way the interacting layer is formed. Clearly therefore, activations of the interacting layer forms a representation of the high-order features of its inputs. Multiple interacting layers may be stacked on-top of each other in order to form arbitrary order combinatorial features.

In a similar idea to attention, in Zhou *et al.* (2017) the attention weights to apply to the input of an attention layer is produced by a linear layer and learned along the rest of the network. Interestingly in this paper, the authors remove the softmax layer as a way to mimic probabilities and to reserve the intensity of activations.

Skip connections are also used in both Song *et al.* (2018) and Wang *et al.* (2017) to connect features modelling low level interactions with features modelling high order interactions. Their conclusion is that the use of skip connections improves the performance of a neural network.

4.3.2 Self-Normalising Neural Networks

In the DLTD literature, one rarely encounters an optimal network depth higher than three of four layers. The reason is that a fully connected model result in very complex optimisation hyperplanes which increases the risk of falling into local optima. A proposed way of training deeper neural networks is to make use of *self-normalising neural networks* (Klambauer *et al.*, 2017). These networks were developed as alternatives to the use of batchnorm layers, since the latter often become unstable when using SGD or stochastic regularisation techniques such as dropout. This is especially the case in fully connected neural networks, where the resulting instability manifests in terms of high-variance training errors.

The self-normalising neural network is simply a neural network which implements a novel activation function called the *SeLU*. The SeLU activation assists the network in maintaining zero mean and unit variance for activations at all network levels. In addition to obviating the need for a BatchNorm layer, SeLUs are much more immune to exploding or vanishing gradients. The definition of the SeLU activation function follows below:

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

where $\lambda = 1.0507$ and $\alpha = 1.6733$ are special constants derived in the paper.

In Klambauer *et al.* (2017), the application of SeLUs were tested using 121 classification datasets from the UCI Machine Learning repository. Here the performance of DNNs with SeLU activations were compared to the performance of other DNNs, and to that of completely different classifiers (including random forests and SVMs). The various models were compared using a pairwise Wilcoxon test on their ranked accuracies. It was found that in the case of datasets with less than 1000 obervations, random forests and SVMs performed the best. However, in the case of datasets with more than 1000 observations, DNNs with SeLU activations outperformed all other models. Despite the above promising results, we have not yet seen SeLUs used in any tabular dataset applications.

4.4 Sample Efficiency

Application of deep learning neural networks to tabular data is hindered by the typical size of these datasets. Tabular datasets are usually not nearly as large as unstructured datasets used to train neural networks for image and text classification. Moreover, in contrast to the ImagNet dataset for computer vision, or the Wikipedia corpora for NLP, there exists no large tabular dataset which may facilitate transfer learning. In this section we propose two techniques towards overcoming the above problem, *viz.* data augmentaion and unsupervised pre-training.

4.4.1 Data Augmentation

In the context of neural networks for tabular data, the topic of data augmentation is rarely studied. This is in part due to the fact that the augmentations

used in standard deep learning applications (such as computer vision and NLP) do not make sense in the case of tabular data. For example, one cannot rotate or scale an observation in a tabular data set without contorting its true nature and/or meaning. There are however some transformations that do make sense for tabular input. In this section we discuss is the injection of random noise by means of so-called *marginalised corrupted features* (MCF), *blank-out corruption*, *swap noise* and *mixup augmentation*.

When working with images, we may randomly perturb pixel intensities by a small amount so that it is still possible to make sense of the image content. For example, adding the value 1 to all pixels and colors in an image, will simply brighten the image. Such a translation can however not be used in tabular data application, since the inputs are mostly not measured on the same scale and therefore the addition of noise might result in feature values that lie outside the distribution that generated them. Thus, in order to inject random noise to a tabular data sample, the values to be added should be scaled in relation to the range of each input feature. Subsequently also, the validity of the resulting value for a particular feature should be verified. Incorporating some noise may potentially also cause a model to be more robust to small variations in the data. Along these lines, Van Der Maaten *et al.* (2013) suggest an augmentation approach called marginalised corrupted features. In the MCF approach, noise are generated from some chosen distribution and used to transform the input values. In Van Der Maaten *et al.* (2013), they experimented with Gaussian, Laplace and Poisson distributions for generating noise. But it is typically unclear beforehand which corruption distributions are useful for what types of data. In addition, this corruption procedure is only applicable to numerical features.

In the original 'denoising autoencoders' paper (Vincent *et al.*, 2008), note that a so-called *blank-out corruption* procedure is used. Blank-out corruption entails the random selection of a subset of the input features, and masking their values with zeros. This resembles dropout regularisation, but applied to the inputs. The only problem with this approach is that for some features a zero value actually carries some meaning. Hence, features should be 'blanked out' using some unique value that does not belong to the range of their distributions.

Another input corruption approach that has empirically been shown to work well[3], is the technique called *swap noise* (Kosar and Scott, 2018). The

[3]https://www.kaggle.com/c/porto-seguro-safe-driver-prediction/discussion/44629

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

swap noise procedure corrupts inputs by randomly swapping input values with those of other samples in the datasets, thereby ensuring that the corrupted input at least assume valid feature values. The use of swap noise is illustrated in Table 4.1.

Finally in this section, we consider the mixup augmentation approach (Zhang *et al.*, 2017). Here artifical samples are created by means of 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 $\boldsymbol{x}$ denotes an input vector, where $\boldsymbol{y}$ represents a one-hot encoded output vector, and where $\lambda \in [0, 1]$. Furthermore, $(\boldsymbol{x}_i, \boldsymbol{y}_i)$ 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 by means of mixup augmentation on a toy dataset in Figure 4.4.

The parameter λ governs the strength of the interpolation between the input-output pairs. The closer λ lies 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 for the mixed-up sample to lie closer to one of the original samples (as opposed to midway inbetween two samples). Best performances were obtained for $\alpha \in [0.1, 0.4]$. In cases where the α value was set too high, underfitting occur.

Typically data augmentation procedures depend on the dataset at hand and requires expert knowledge. It is therefore difficult to invent novel, generic ways to augment tabular data. Still, from the way in which mixup augmentation is

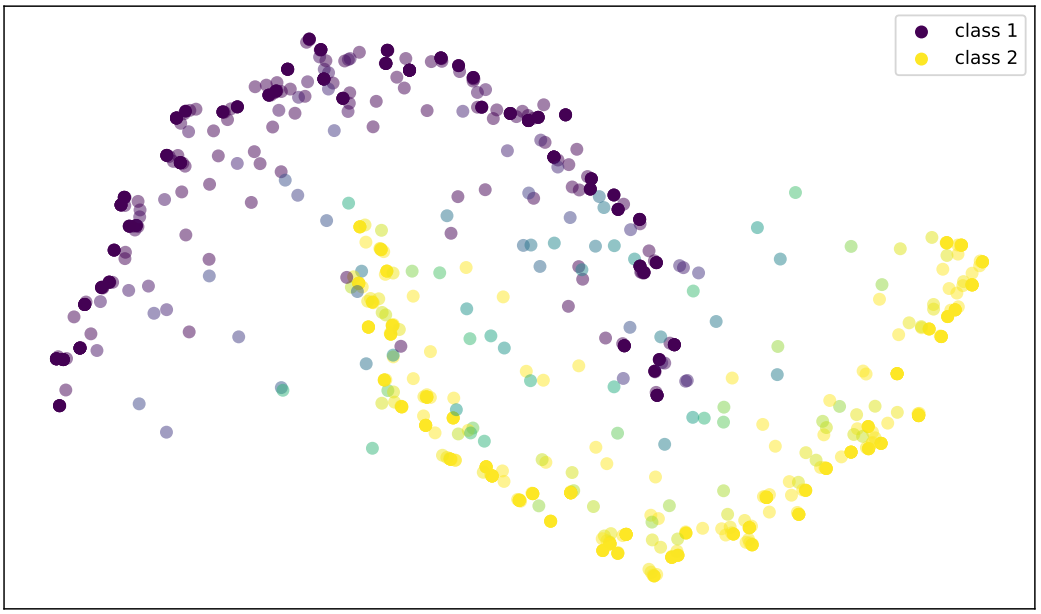


Figure 4.4: Illustration of the data points created by mixup augmentation.

defined, it is clear that this procedure may be used in the context of any type of data (including tabular datasets).

Mixup data augmentation may be understood in terms of a mechanism encouraging a model to behave linearly inbetween training samples. Zhang *et al.* (2017) show that this linear behaviour reduces the amount of undesirable variation when predicting new samples lying further away from the set of training samples. The authors also show empirically that training using mixup augmentation is more stable in terms of both model predictions and gradient norms. This is because mixup leads to decision boundaries that transition linearly between classes, resulting in smoother predictions.

In Zhang *et al.* (2017), the use of mixup data augmentation was tested on six tabular classification datasets from the UCI Machine Learning repository. The network architecture that was used for this purpose involves a two-layer MLP with 128 neurons in each layer, and a batch size of 16. It was found that mixup improved prediction performance in the case of four out of the six datasets considered.

In conclusion, it is important to note that all data augmentation metods presented here require hyperparameters to be tuned. When using swap noise and blank-out for example, the proportion of feature values to be corrupted needs to be specified. In mixup augmentation, the mixup weight needs to be

set. Since these parameters control the strength of regularisation, their values are important and should be carefully tuned.

4.4.2 Unsupervised Pretraining

Recall that unsupervised pretraining was described in §3.3, and presented as a very effective way of training deep neural networks. There we have seen that, by automatically learning useful feature combinations in an unsupervised fashion, pretraining may also be used to address the feature engineering issue. However, in the literature we only found two papers that make use of pretraining in the context of tabular data. These are Zhang *et al.* (2016) and Miotto *et al.* (2016). Both papers made use of pretraining with DAEs. In the latter paper, a stacked denoising autoencoder is used as feature learning method. The representations derived from the DAE are subsequently passed on to supervised learning algorithms, thereby enabling improved predictions. This approach is used in a medical application, where general-purpose patient representations facilitate clinical predictive modelling.

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 (for example the number and sizes of layers). Thereafter we need to decide which learning parameters to choose. Currently, parameter selection is an aspect which have not yet been as thoroughly explored as hyperparameter selection for supervised neural networks.

In addition we of course also need to decide which DAE loss function to use, where in this regard, a choice depends on the type of inputs. If all inputs are numeric, we may use the common MSE loss, but if the inputs include categorical features, more consideration is needed. Miotto *et al.* (2016) transformed all continuous features to lie within [0,1], whereas all categorical features were one-hot encoded. This enabled the use of binary cross-entropy as loss function. If we wish to use entity embeddings for categorical features, we can however not use their approach. A suggestion is to, for each categorical feature, import a separate multiclass cross-entropy loss function, and to use MSE loss for continuous features. The losses for categorical and continuous features may then be combined in order to obtain a total loss. The weights to be assigned to the two losses is an aspect which has not yet been researched.

Instead of the feature extraction method followed in Zhang *et al.* (2016)

and Miotto *et al.* (2016), we propose treating unsupervised pretraining as a transfer learning problem. That is, we suggest replacing the output layer of the DAE with the required output layers for the supervised learning task. Then one may first train the newly added layers, keeping the lower level DAE layers fixed. Subsequently, all layers may be trained simultaneously. Rather using this approach the feature extractor part of the DAE may still be tuned in order to be optimal for the supervised learning task. We experiment with the above approach in the following chapter.

4.4.3 Regularisation

Since regularisation may also be used to facilitate sample efficiency, we provide a brief discussion of the topic in this section. In Chapter 3, dropout was discussed as a popular form of regularisation, which is also true for DLTD and found in a lot of work such as Song *et al.* (2018), Zhang *et al.* (2016), Qu *et al.* (2016), Guo *et al.* (2017). In the original paper by Srivastava *et al.* (2014), it was advised that the dropout parameter p (which specifies the percentage of neurons to discard), should lie between 0.5 and 1. The exact value for p is typically determined using a grid search, and results in the literature indicate the optimal percentage to be data- and model dependent (compare for example the optimal p in Song *et al.* (2018), Zhang *et al.* (2016), Qu *et al.* (2016), Guo *et al.* (2017)).

Interestingly, Haldar *et al.* (2018) reported that dropout was not effective in their application. In their analysis of this result, they hypothesise that dropout produces invalid input scenarious instead of mimicking valid scenarios that may be missing in the training data. This may distract the model during training causing it to be suboptimal for 'real' examples. Therefore, the authors rather opt for hand-crafted noise shapes which takes into account the distribution of a relevant feature, similar to the MCF approach.

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

A different angle to regularisation makes use of the fact that the importance of each feature with respect the target is different for tabular and unstructured data. In computer vision for example, a large number of pixels have to change

before an image resembles an entirely different object. In tabular data, on the other hand, a very small change in a single feature may result in large changes with respect to predictions (Shavitt and Segal, 2018). In the latter paper, it is mentioned that this aspect may be addressed by means of including a separate regularisation term for each of the weights in the network. The learning loss is thus:

$$L(D, W, \Lambda) = L(D, W) + \sum_{i=1}^{M} \exp(\lambda_i) \cdot \|w_i\| ,$$

where $D = \{(x_i, y_i)\}_{i=1}^{N}$ are the training samples, W the weights of the model and $\Lambda = \{\lambda_i\}_{i=1}^{M}$ are the regularisation coefficients for each of the M parameters in the network. These regularisation terms are treated as additional model hyperparameters. Unfortunately it is clear that the above approach is totally impracticable. The only way to train these hyperpararmeters is through repeated tweaking and validating and this is not possible in a gradient-based method. A workaround is to make these regularisation parameters trainable, as is the case with all other modules in the network. This is achieved by minimising the so-called counterfactual loss function, a novel objective proposed by Shavitt and Segal (2018). The counterfactual loss is the emprical loss $L(D, W)$, where the weights have already been updated using the regularised loss $L(D, W, \Lambda)$. Results obtained from empirical work in the paper by Shavitt and Segal (2018) are promising. It was found that training neural networks by means of optimising the counterfactual loss leads to the NNs outperforming all other NN regularisation approaches.

4.5 Interpretation

The importance of model interpretation was mentioned in Chapter 3. A good understanding of a model helps one to improve the model in the areas where it is weak. Stated in another way, if one cannot explain the way in which a prediction is arrived at, one cannot be sure about its accuracy. In most of the literature for deep learning, thus far relatively little attention has been given to model interpretability. Luckily in the context of tabular data, some of the modern deep learning architecture provide more options in terms of simpler interpretation. In terms of modern NN architectures, in this section, we briefly discuss neural network interpretation as facilitated by the use of entity embeddings and attention layers. We also discuss model independent

interpretation by means of permutation test, and illustrate this approach on the Adult dataset introduced in Chapter 1.

With regard to the use of entity embeddings, note that these may easily be visualised once projected into a low-dimensional space. This was done Figure 4.2. For a second example, the interested reader may refer to an illustration in the paper by Zhou *et al.* (2017).

An attention layer can also be used to provide insight into the relative importance of certain feature combinations, as may be seen in the work by Song *et al.* (2018) and Zhou *et al.* (2017). Relative importance measures are obtained from the attention weights α, which in turn are based upon feature similarities. From these we may infer which features are combined to make predictions for certain observations. As an example, the output after using the above approach, as obtained in Song *et al.* (2018), is displayed in Figure 4.5. Also see Shavitt and Segal (2018) for how feature importance were evaluated using the regularisation strength of the learned per weight regularisation parameters λ_i, $i = 1, \ldots, M$.

Haldar *et al.* (2018) applied a model agnostic model interpretation algorithm in the context of DLTD. The authors made use of the permutation importance algorithm in order to evaluate the importance of features. However, they reached the same conclusion as stated in §3.7.2, *viz.* that the permutation test did not produce sensical results because the features were not independent. Independent permutation of the features created examples that never occur in real life, and the deduced importance of features in this invalid space is misleading. Subsequently the authors attempted to identify redundant features, which to some extent alleviated the above issue.

For illustration purposes, the remainder of this section is devoted to an implementation of a permutation importance plot on an NN fitted to the Adult dataset. In order to obtain a more robust estimate and standard errors, we perturbed each feature five times and measure the NN performance in terms of the log-likelihood loss function. Our results are displayed in Figure 4.6 (note that the standard errors were too small to be incorporated in the figure). The horizontal axis measures the average drop in the log loss function when a feature is shuffled before prediction. The feature ranking makes intuitive sense. It is reassuring to see that the features ‘gender’, ‘race’ and ‘native-country’ are some of the less important features when predicting income.

Next we compare permutation importance with feature rankings obtained

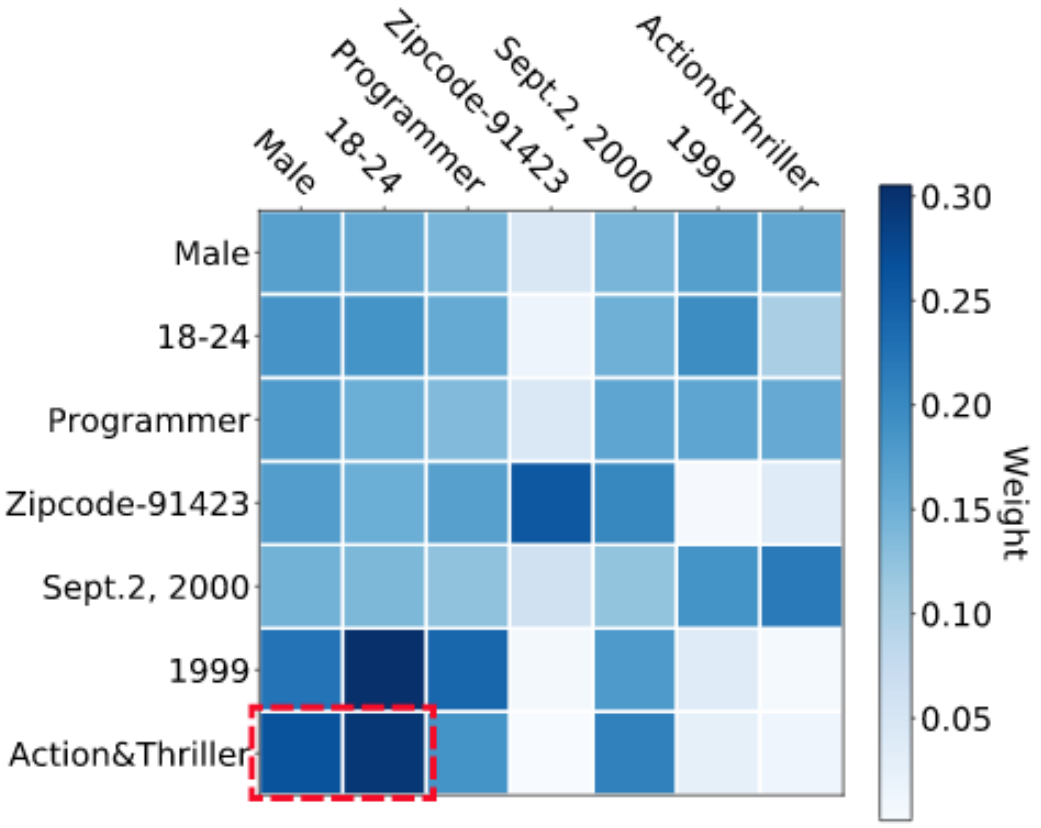


Figure 4.5: A display of the attention weights for a single observation in the dataset.

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 on the Adult dataset as the target for a boosted model to predict. The inputs to the neural network and boosted model are of course the same.
Once the boosted model has been trained on these targets, we may extract feature importances based upon the number of times that the individual trees used a certain feature to split on. The obtained feature importance values are displayed in Figure 4.7.

In the two approaches, the variable rankings are more or less the same. The only exceptions are the rankings obtained in the case of the 'marital-status' and 'fnlwgt' features. This difference might be caused by the permutation importance method not picking up on multivariate feature interactions. Therefore, it incorrectly perceives 'marital-status' to be an important feature, although in reality the performance drop is actually caused by another feature dependent

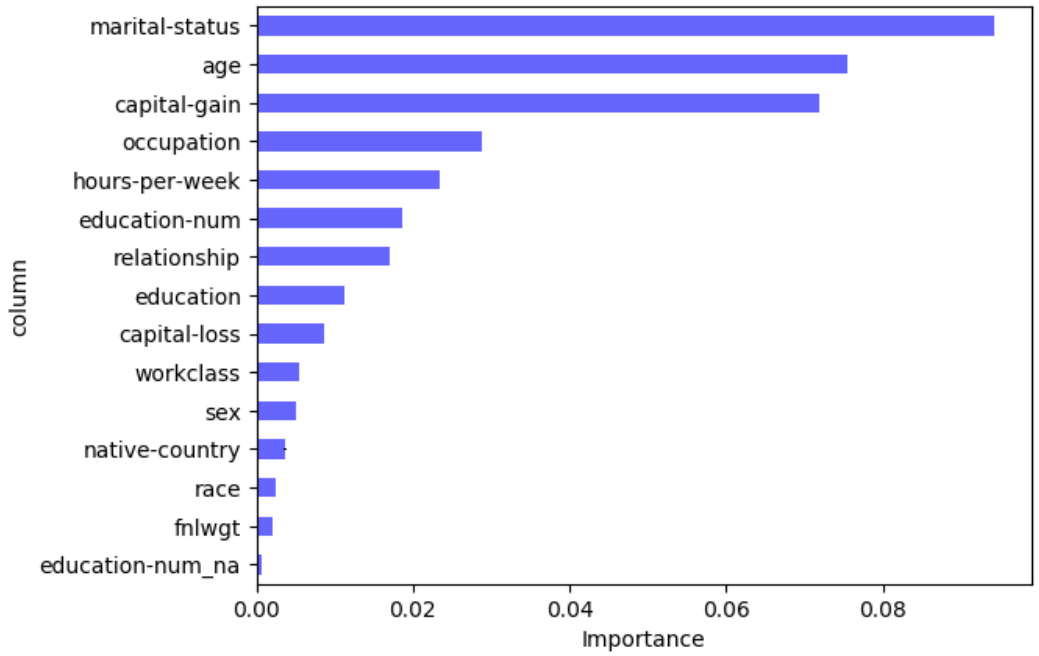


Figure 4.6: A permutation importance plot of an NN trained on the Adult dataset.

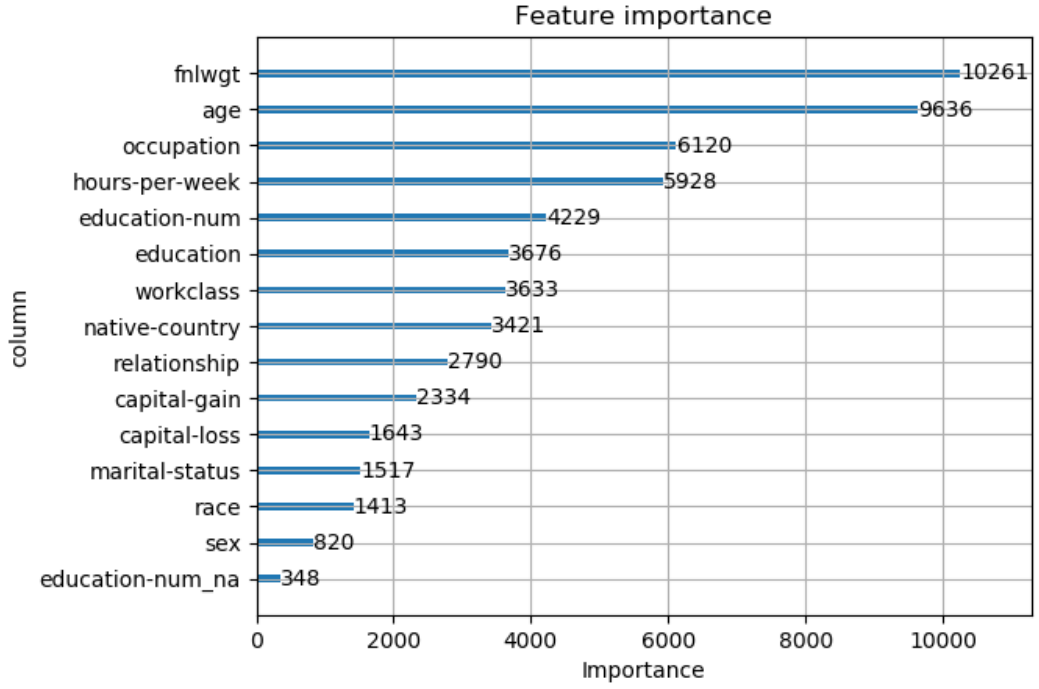


Figure 4.7: Feature importance values obtained from a boosted model trained on NN predictions.

on 'marital-status'. The other explanation we can give is that 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, this might have contributed to the reason 'fnlwgt' was considered to be so important in the case of the boosted model. The above differences in interpretation illustrates that one should take care when interpreting models. It seems to be a good idea to make use of more than one source of information when possible.

4.6 Hyperparameter Selection

Thus far, we have come across 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 hyperparameters to select, such as the number of layers and the type of activation functions to use; and there are also learning algorithm hyperparameters to specify, for example the learning rate or weight decay to be used. In term of structural parameters, since there are no shared patterns among diverse tabular datasets, it is very difficult to design a universal architecture to use in all setups. In most of the literature, typically a grid search over many combinations of hyperparameter values is performed (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 parameter values are very dependent on the dataset and other modeling choices; hence the need to tune them. The main structural hyperparameters tuned in the majority of the papers may be listed as follows: hidden layer size, number of hidden layers, activation functions, and the shape of fully-connected layers (*i.e.* a constant-, increasing-, decreasing- or diamond shape). The optimal hyperparameter values found varied accross publications, indicating once again the necessity of a hyperparameter tuning step in the case of uncustomary datasets and models. With regard to hyperparameter selection, we include some experiments of our own (similar to the experiments conducted in the aforementioned papers) in Appendix A.

In terms of hyperparameters specific to the optimisation algorithm (like learning rate, batch size, learning rate decay, *etc.*), not much experimentation was done in the researched papers. 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) and Wang *et al.* (2017), 1024 and 512 respectively.

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 1cycle 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.

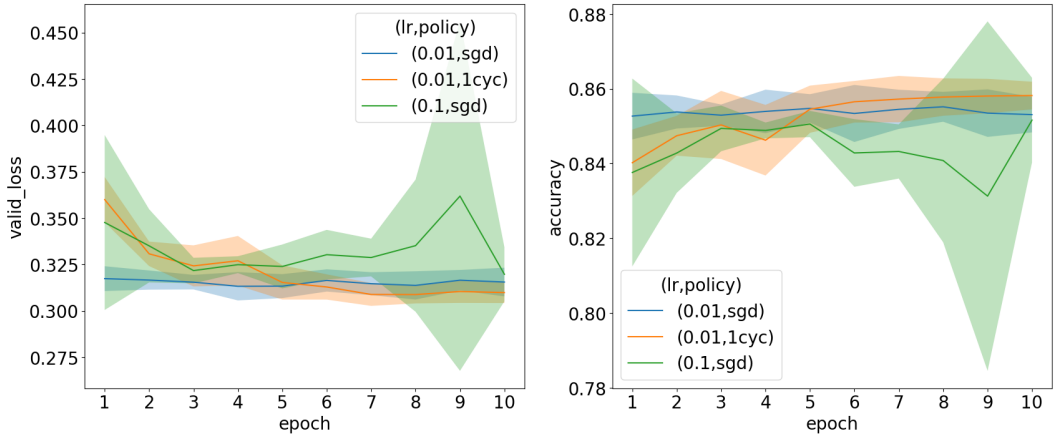


Figure 4.8: Constant learning rate vs the 1cycle schedule.

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 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.

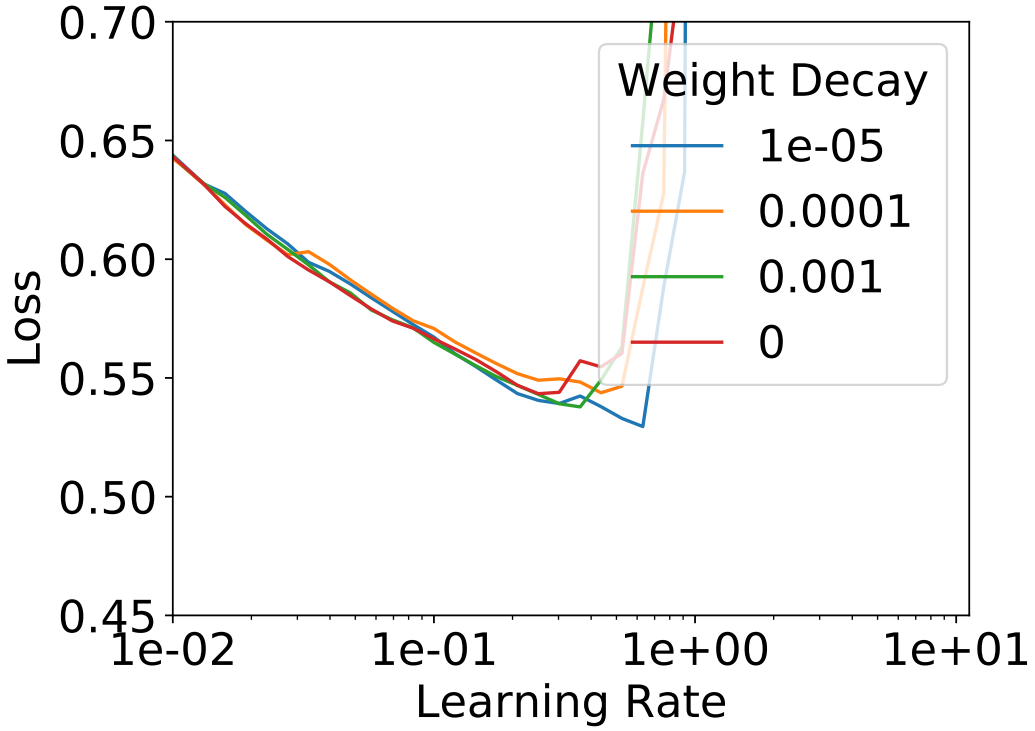


Figure 4.9: A learning rate range test with different weight decays.

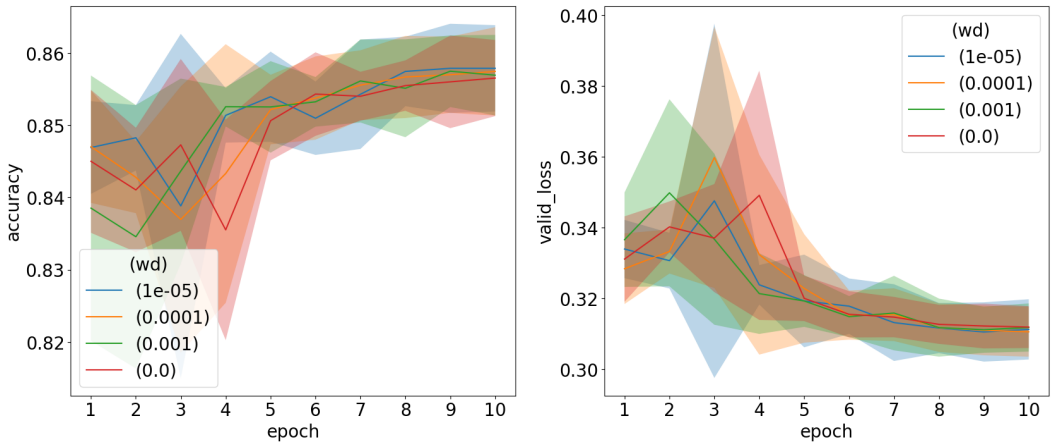


Figure 4.10: A full training run with different weight decays.

Chapter 5

Experiments

5.1 Introduction

The main objective in this chapter is to better understand the behaviours of certain neural network model architectures (and their parameters). The empirical work presented here also serves as verification in terms of the observations reported in the literature. Our focus is on the same deep learning modelling challenges as discussed in Chapter 4. These are:

- appropriate input representation;
- learning from feature interactions; and
- enhancing sample efficiency.

The reader will note that the aspect of model interpretation is not listed above, since this challenge was covered in §4.5.

Appropriate specification of more general hyperparameters such as learning rate, batch size, layer size and layer depth does not form part of the focus in our experimentations. The reason is that these do not form part of the main challenges presented by the use of neural networks for tabular data, and also, with some of these parameters we have already experimented in Chapter 4. Nevertheless, since these parameters are tightly linked with each other and with other model parameters, where appropriate, their specification is still done using a hyperparameter grid search. Our findings with regard to these grid searches may be found in Appendix A.

An outline of the remainder of the chapter is as follows: In §5.2 we discuss the dataset that was used in the empirical study, and motivate the reason

for having selected this specific dataset. Thereafter, in §5.3, we discuss the general methodology followed, as well as the measures of evaluation for each method. The experiments may be found in §5.4, §5.5 and §5.6. In §5.4 we compare the performance of various entity embedding sizes. §5.5 is devoted to an investigation of the use of several approaches towards modelling high-order feature interactions, whereas in §5.6 we evaluate the different approaches to avoid overfitting in constrained data environments.

In this chapter we aim for good empiricism by exploring many hyperparameters in various data scenarios and by performing cross-validation in order to obtain unbiased performance measures along with standard errors. Our goal is not necessarily to beat the benchmark, but instead consists of simple experiments designed to aid in understanding techniques.

The code for developing the models and running the experiments were written in Python. The deep learning library, Pytorch, was used to build the different neural networks. Many of the common layers, like the linear layer or the ReLU activation, for example, are provided by the library and we are left with combining these layers to form the models we required. However, some of the more modern layers/modules, like the attention mechanism or the SELU activation for example, are not yet provided by the library and therefore we wrote our own implementations thereof.

The optimisation of the neural networks were done with the help of the Fastai library which provides convenient utility functions for training with the 1cycle policy. We wrote code that builds on this library to perform various tasks required for these experiments, for example: cross-validation, hyperparameter selection, pretraining, data augmentation, transfer learning and visualising results. One may refer to Appendix B for more detail on the code and where to access it.

5.2 The Dataset

We chose to conduct our empricial work on the Adult dataset. This was done for the following reasons:

- **Simplicity**: The Adult dataset is representative of a real-world scenario, while not having specific modelling challenges (such as plenty of missing values or highly imbalanced classes). This fits in with our goal of evaluat-

ing models on a generic tabular dataset (as opposed to one that requires special attention or the skills of a domain expert). Thus, when in terms of prediction performance, a specific network architecture is found to fall short, we may know that this may be attributed to the model, and not to some peculiarity inherent in the data.

- **Minimal preprocessing**: We want to focus our time on training the algorithms and not on preprocessing the data. The Adult dataset is a relatively 'clean' dataset.
- **Open access**: Since we want our work to be reproducible, we want the dataset that we use to be accessible to everyone.
- **Good size**: We have seen that fitting deep neural networks requires large datasets. Hence, in order to yield optimal performance we prefer at least a medium sized dataset. If one wants to test models using smaller datasets, one may simply run the experiments on a subset of the data.
- **Strong baselines**: The Adult dataset has been used in other deep neural network studies, and therefore the performance of several methods on this dataset have been reported in the literature. In order to know how well we are doing, we need to be able to compare our performances with those of others.

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 validation dataset performance 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

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

- *workclass*: Private, Self-emp-not-inc, Self-emp-inc, Federal-gov, Local-gov, State-gov, Without-pay, Never-worked
- *fnlwgt*: continuous
- *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 5.1 and the counts of the categories for each of the categorical features can be found in

Figure 5.2. 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.

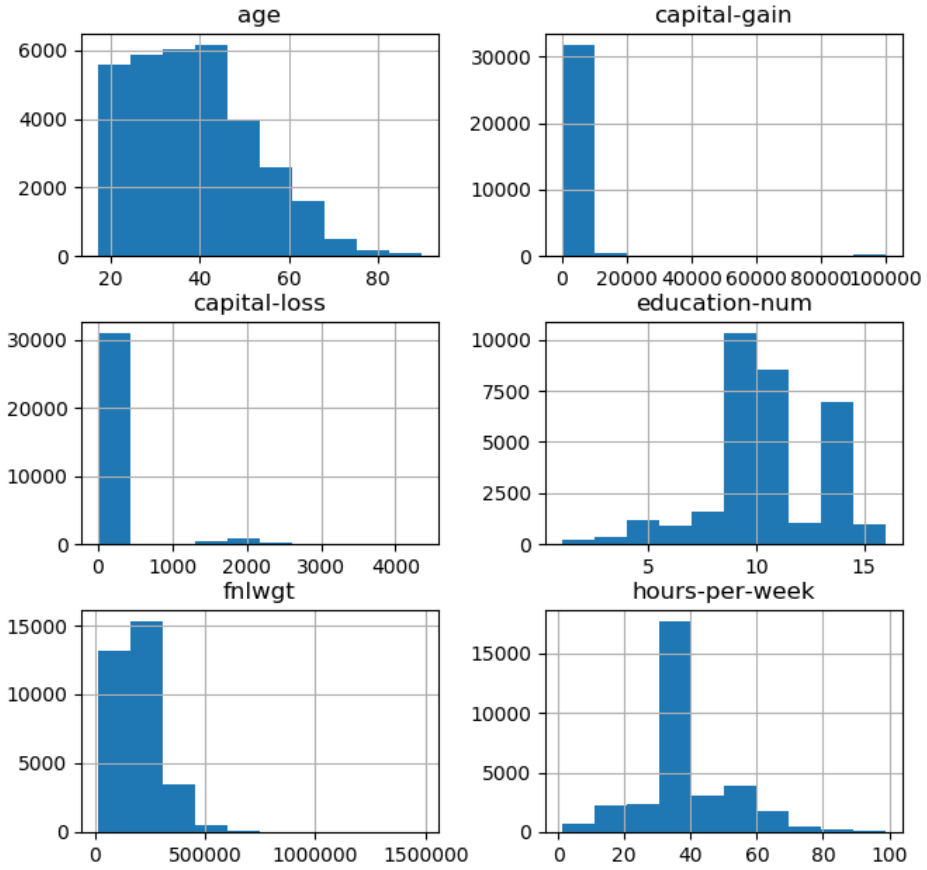


Figure 5.1: Histograms for each of the continuous features in the Adult dataset.

5.3 General Methodology

5.3.1 Loss Function and Evaluation Metric

Recall the learning problem associated with the Adult dataset, *viz.* to predict whether or not an indivdual earns more than $50,000 a year. Since this is a

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

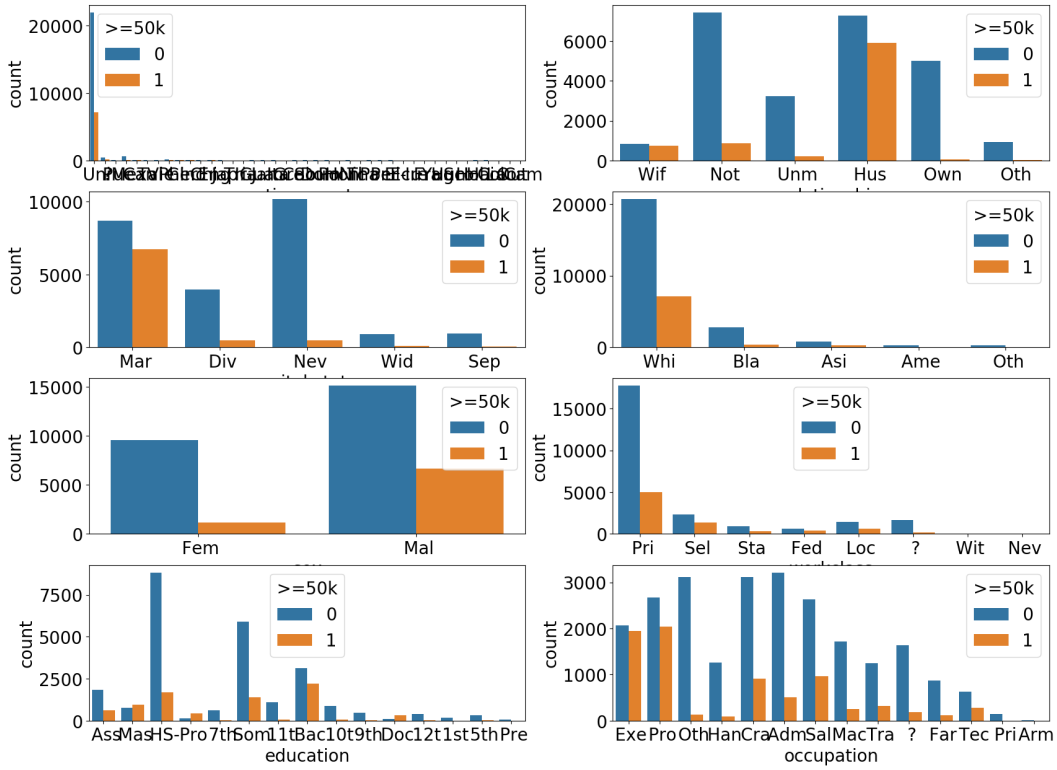


Figure 5.2: Bar plot for each of the categorical features in the Adult dataset.

binary classification task, we train a neural network to optimise the binary cross-entropy loss function, as defined in §2.3.4 (where of course we set $K = 2$). This is the standard loss function typically optimised when fitting NNs for binary classification (Song *et al.* (2018), Wang *et al.* (2017), Zhang *et al.* (2016), Qu *et al.* (2016)). Note that during training, the accuracy of the neural network will also be monitored. Accuracy is used when monitoring the training process since this measure is often simpler to interpret. Note however that the neural network is not trained to optimise its accuracy. In this sense the binary cross-entropy may be seen to act as differentiable proxy to the accuracy.

5.3.2 Cross-validation

We made use of five-fold cross validation (Hastie *et al.*, 2009, p. 241) to estimate the performance of a model. Cross-validation is very often used to estimate performance measures in tabular data applications. The approach is needed since in small to medium sized datasets, there are not enough data points to be able to split off a test dataset that may not be used during training. Cross-validation involves randomly partitioning the dataset into five equal

parts. Each data part takes turns to be used for validation, while the remaining four parts are then used to train the model.
Note that in this way, five model performance measurements are obtained. We may subsequently compute the average over these five measurements, thereby obtaining a less biased estimate of the performance of a model. The partitioning and training steps in five-fold cross-validation are depicted in Figure 5.3.

Figure 5.3: 5-Fold Cross-validation dataset split schematic.

Another advantage of cross-validation is that it yields a standard error associated with the performance estimate. In our results, standard errors will be displayed on figures, thereby providing confidence interval estimates of performances via *i.e.* $\mu \pm \sigma$. This should be helpful in terms of determining the significance of reported differences in performance.

One may ask why we make use of cross-validation while in deep learning research it is common practice to estimate performances using the validation set approach. See for example Klambauer *et al.* (2017), Song *et al.* (2018) and Zhang *et al.* (2016). In the validation set approach, a dataset is partitioned into a training-, validation- and (sometimes) also into a test dataset. In contrast to cross-validation, partitioning is performed only once. The problem with a once-off split is that it provides only a single estimate of the performance of a model.

Standard errors cannot be obtained, therefore the variance of a performance estimate cannot be gaged. Also in terms of bias, cross-validation performs better than the validation set approach, since the size of the training sets in cross-validation is closer to the size of the original training set. Cross-validation is therefore typically preferred to the validation set approach. However, bearing the computation intensity of training neural networks in mind, the use of cross-validation in deep learning seems to be very impractical. Repetitive partitioning and training steps are much more costly than in the case of tabular data. Fortunately, deep learning is mostly applied to large datasets. In these cases, large test sets are available, thereby obviating the need for cross-validation. Therefore in summary, we make use of cross-validation since our tabular dataset allows (and calls for) its use. In deep learning, the validation set method is practical and sufficient, since test datasets are large enough.

5.3.3 Preprocessing

As mentioned in §5.2, in our empirical analyses we aim to avoid feature engineering and preprocessing. We therefore did not create any new feature combinations. The only mandatory preprocessing steps involved: - mapping the text in categorical features to integer values (this was of course necessary to be able to apply entity embeddings); - standard normal scaling of the continuous features (which was necessary for optimisation algorithms), - mean imputation of missing values in the case of continuous features, and assign a "null" category to missing values in the case of categorical features. Note that we did not perform any *a priori* feature selection steps since we wanted the model to learn which features are relevant.

5.3.4 Hyperparameter Specification

If not stated otherwise, we made us of simple MLPs to map input representations to income predictions. Based on our findings in §4.6, we decided to train all networks using the 1cycle policy (and the Adam optimiser). Note that the hyperparameter selection process suggested by Smith (2018) in this context is a manual process. Therefore, we make use of the Smith (2018) approach only during the first experiment, thereby finding appropriate values for the learning rate, for the number of epochs, and for weight decay (called 'training hyperparameters' in this section). These parameter values are then used

in all empirical work to follow. This initial search for appropriate training hyperparameter values is reported in Appendix A. Note that in the second and third experiments, the parameter selection process was rerun in cases where the fitted model changed significantly from the previous experiment to the next. Also, whenever we encountered instability of the loss function during training, the parameter selection process was redone.

Following the above hyperparameter specification methodology, note that in each experiment we definitely do not expect to find the optimal model. The hyperparameter selection approach should however prove sufficient for comparing performances. Luckily also, according to Smith (2018), NNs have been found to be quite robust with respect to the specification of training hyperparameter values.

With regard to the specification of structural hyperparameter values, we experimented with the dropout percentage, and with the width and depth of the network. Again we found optimal values for these parameters once off, *viz.* in the first experiment. Thereafter, these parameter values were kept fixed. More details regarding the specification of structural hyperparameter values may be found in Appendix A.

5.4 Input Representation

5.4.1 Embedding Size

In this section we report on our experimentation with respect to the effect of embedding size on the prediction performance of a neural network. Specifically, we consider the use of 2, 4 and 8 neurons in the embedding layers. These values are chosen since in our setup, we expect the ideal embedding size to be a small value, but not so small that it is unable to capture all of the signal conveyed by the available features.

As stated in §5.4, we expect there to be an optimal embedding size for each variable, depending on the cardinality of the variable, and on how complex its relationship is with the other variables and with the target. Recall that in §4.2.2 a set of rules for representation of categorical variables is proposed, where these rules depend on the cardinality of each categorical feature. In addition to experimenting with embedding sizes, we would therefore also like to see whether these rules are able to, also in our data setup, provide reasonable

performances.

The results of the first experiment (in which we compared three pre-specified embedding sizes) are displayed in Figure 5.4. In the left panel, note that the loss function value, obtained for the validation data is plotted against the number of epochs; whereas in the right panel the validation accuracy is plotted on the y-axis. Shaded areas indicate standard errors. From this figure, since at epoch 5, the blue line lies below the red and green lines, it is clear that an embedding size of 2 is preferred. One reason for this may be that for these data, categorical features are not that important in terms of enhancing prediction accuracy. Therefore, a smaller representation may assist in reducing noise.

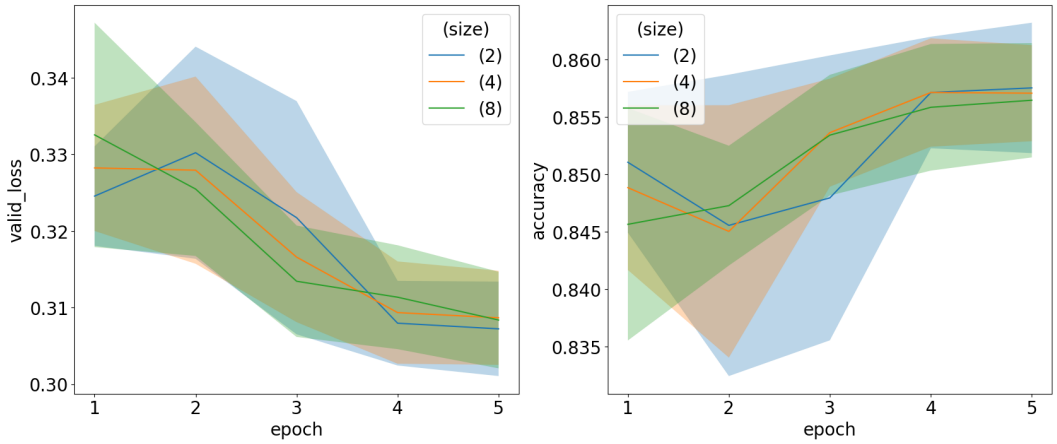


Figure 5.4: Effect of the embedding size if all categorical features are mapped to the same number of dimensions.

In order to investigate the effect of cardinality dependence, we compared the constant size 2 embeddings with the rules from Wang *et al.* (2017) and de Brébisson *et al.* (2015). The results from this experiment are shown in Figure 5.5. From the figure we infer there to be almost no difference between the three approaches. With respect to validation loss, de Brebisson's method performed sightly better, whereas in terms of accuracy, the fixed size of 2 performed the best. In datasets where categorical features contain more information, one may however expect this result to be different.

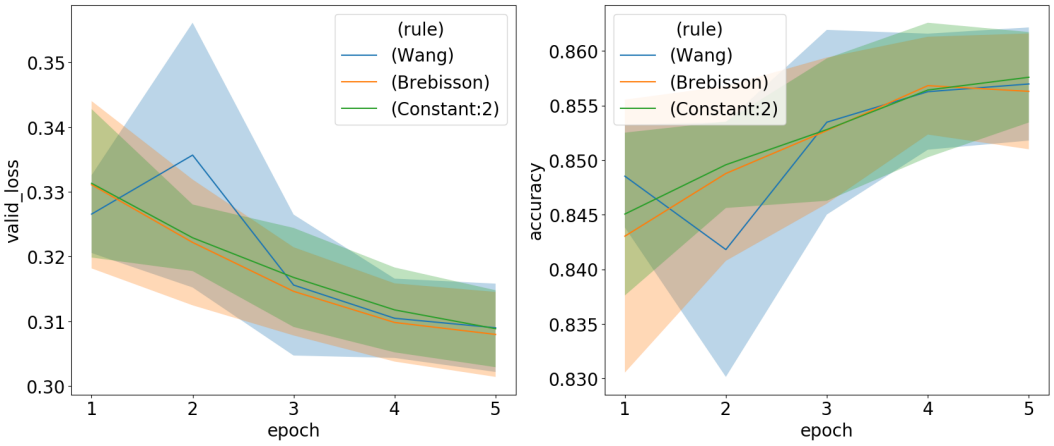


Figure 5.5: Effect of variable sizes on the performance of the NN model.

5.5 Feature Interactions

5.5.1 Attention

Since attention is so effectively used in other data domains, we believe it to be the most promising approach towards modelling high-order interactions between features for tabular data. In order to test this hypothesis, we implemented our own multi-head attention module, as described in Song *et al.* (2018). In this way, our results are comparable to those produced by a standard neural network. We did not have the computing power to try different configurations for this module and thus we went with the largest possible configuration that could be run in reasonable time. We chose the number of head, H, to be 3, and made use of embedding sizes of 3 throughout. Moreover, we only made use of a single multi-head attention layer, and connected its output to a single hidden layer of size 200, which in turn was connected to the output layer. The performance of the above architecture was then compared to the performance of simple MLP with a similar number of total parameters. The results of this experiment are reported in Figure 5.6. The left panel plots the average loss on the validation set over each of the five epochs and the right panel, the average accuracy. Surprisingly, the model with the attention mechanism (blue line, "*attention*=True") performed worse than the MLP (orange line, "*attention*=False") on its own; its predictions exhibiting higher variance than the MLP. A possible explanation is that the hyperparameter specification in the attention layers was inadequate, or that the single attention layer is not complex enough. In the future therefore, we would like to experiment with

more of the hyperparameters of the multi-head attention module and also try out stacked attention layers. Moreover, one wonders in which way the above results would have changed if a dataset with more features were used.

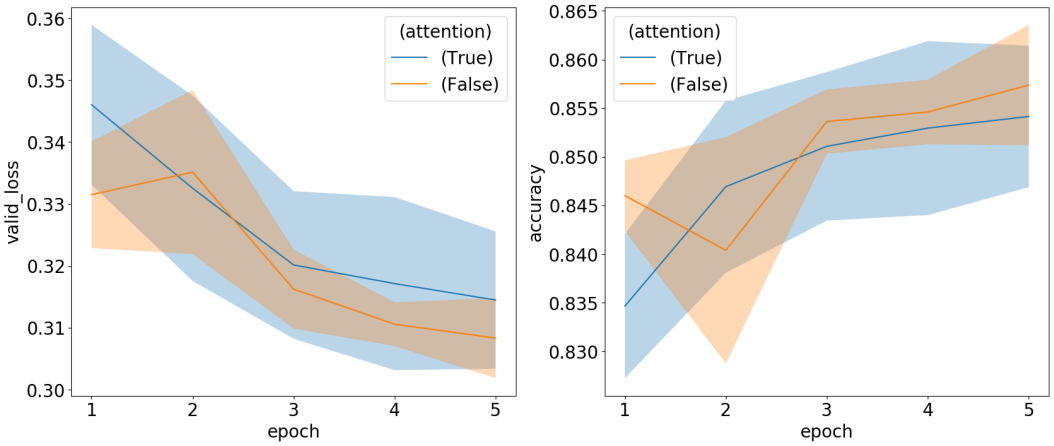


Figure 5.6: Comparing the attention mechanism with a simple MLP.

5.5.2 SeLU Activations

In Chapter 3 we have seen that 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 use of SeLUs to use of the ReLU activation using two networks. The first network was specified to have two layers, whereas the second network had eight layers in total. The SeLU activation function was implemented using its customised weight initialisation method, but no dropout was used. In Figure 5.7 we plot the average (over each fold) loss and accuracy on the validation set for each of the epochs for the different models and in Figure 5.8 we plot the average loss and accuracy on the validation set of the final models. We observe that in terms of validation loss, the model with eight layers using SeLU activations performed the best overall; whereas in terms of accuracy, the model with two layers using ReLU activations emerged as the winner. However, the differences are so minor that, in the absence of more evidence, we tend to still recommend the use of shallow ReLU networks.

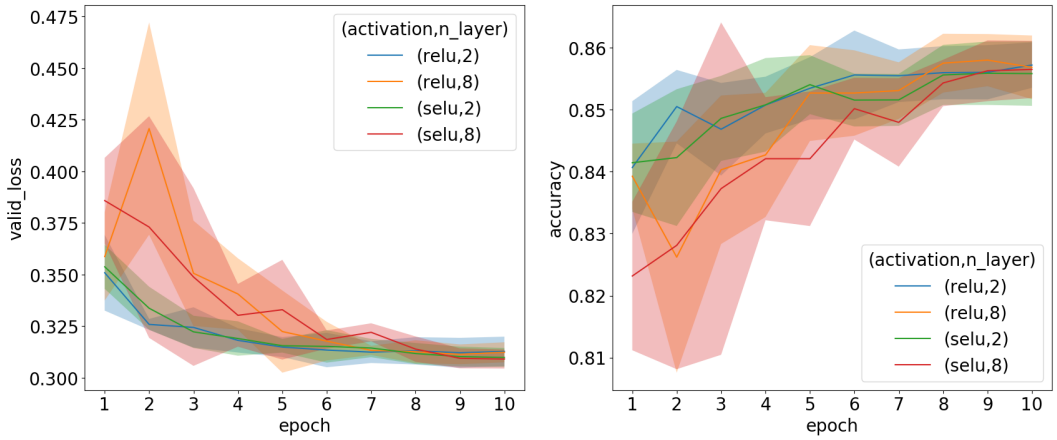


Figure 5.7: The average performance of ReLU and SeLU activation functions for shallow and deep networks as a function of the number of training epochs.

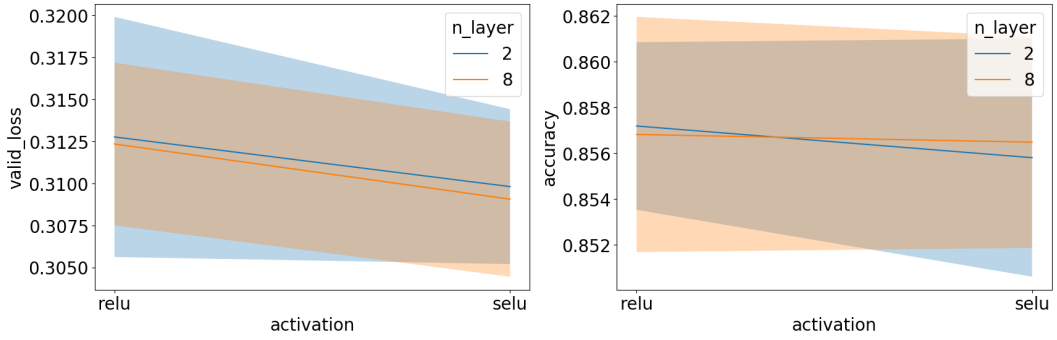


Figure 5.8: The average performance of ReLU and SeLU activation functions for shallow and deep networks.

5.5.3 Skip Connections

Another mechanism which have contributed to the successful implementation of deeper networks is skip connection. Apart from facilitating more efficient neural network training, skip connections have been found to alleviate the degradation problem, thereby improving the prediction performance of deep neural networks. Skip connections are often also used to combine different orders of feature interactions. Using this mechanism, elementwise addition (or other concatenation functions) may be used in order to combine input and output before and after a linear layer. In our empirical work, we investigate the performances of both a shallow (with 2 hidden layers) and a deep network (with 6 hidden layers) using skip connections. The results are summarised in Figure 5.9 and in Figure 5.10, where the average (over each fold) loss and accuracy is plotted for each epoch and for the final model, respectively. The

models with skip connections are indicated by "*residual*=True" in the figure legend and "*residual*=False" otherwise. On average, the deeper neural network with skip connections performed slightly better on the validation data, but once again, performances of the different networks are very similar. At this stage we therefore recommend the simpler design of not using skip connections.

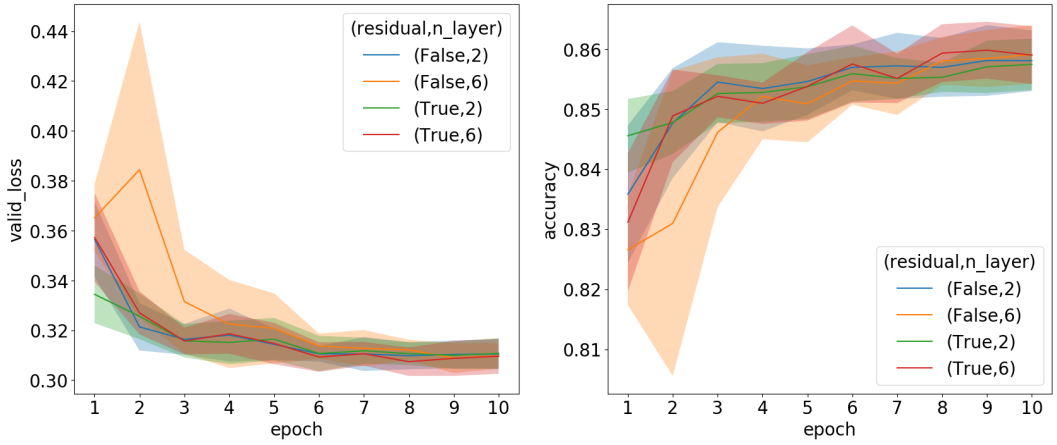


Figure 5.9: Average performance at each epoch for shallow and deep neural networks, with and without skip connections.

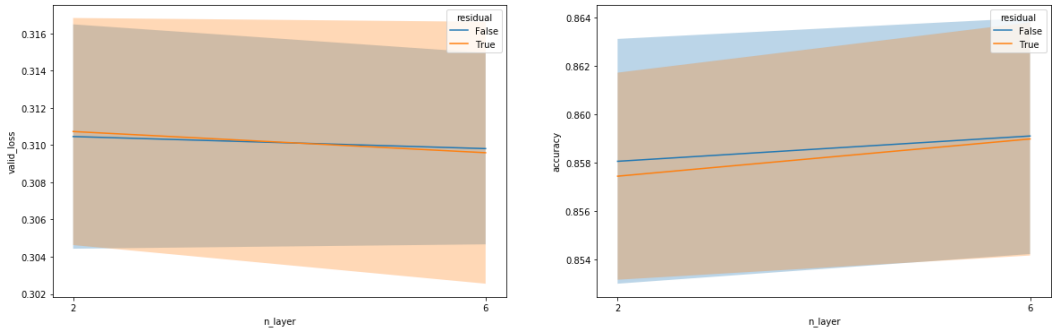


Figure 5.10: Overall performance of the skip connections used in a shallow and deep neural network.

5.6 Sample Efficiency

In the next experiment we investigate the effect of the number of samples on the prediction performance of a neural network. Therefore, the network was optimised using training dataset of sizes of 1000, 2000, 4000, 8000, 16000 and

32000. The results are reported in Figure 5.11. As expected, the network performs better with an increase in the training set size, where the enhancements in performance diminishes in the case of higher sample size values.

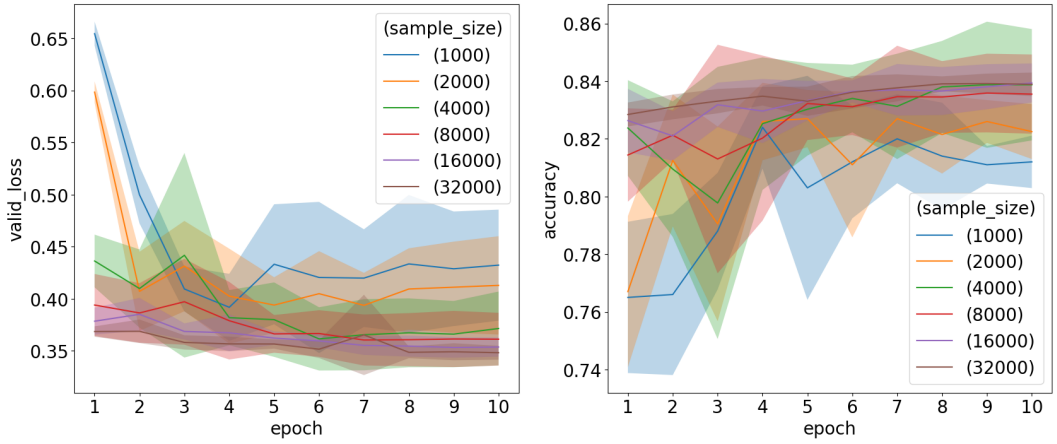


Figure 5.11: Effect of the number of training samples on the performance of neural networks.

5.6.1 Data Augmentation

From the literature review in Chapter 3, mixup augmentation emerged as seemingly the most promising form of data augmentation. In this experiment we wanted to investigate the way in which the application of mixup (own implementation), using various mixup ratios, influences the performance of neural networks on the Adult dataset. Since data augmentation also acts as a form of regularisation, we also wanted to investigate the interaction between weight decay and mixup ratios. We experimented with mixup α-ratios of 0 and 0.4, and with weight decays of $[10^{-5}, 10^{-3}]$ in a neural network with 3 hidden layers, consisting of 200 units each. The results of the experiment are displayed in Figure 5.12 and in Figure 5.13.

The results indicate that mixup is not improving either the validation loss or the accuracy of the models. The observed interaction between mixup and weight decay is an interesting aspect of the experiment. Whereas one might expect the use of mixup to reduce the need for weight decay and vice versa, this notion is not supported by our results. Although from the literature mixup seems to be a promising technique, its success seems to be domain specific.

Our suggestion would be that for tabular data, mixup should only be used when interpolating between samples makes sense.

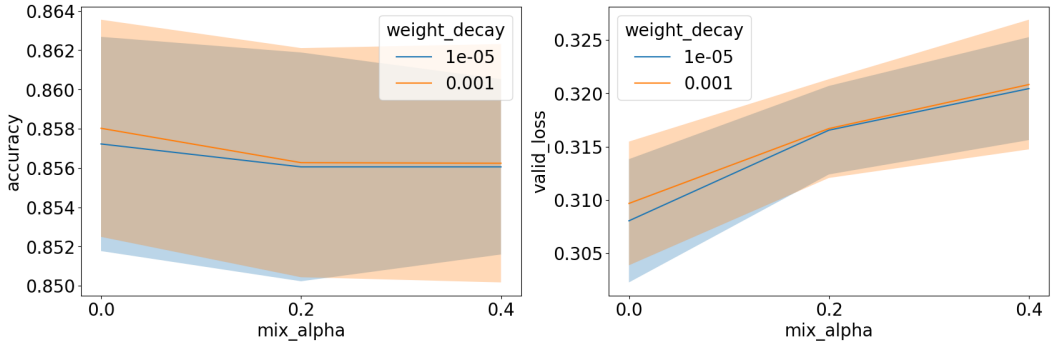


Figure 5.12: Average performance of models using various mixup and weight decay parameters.

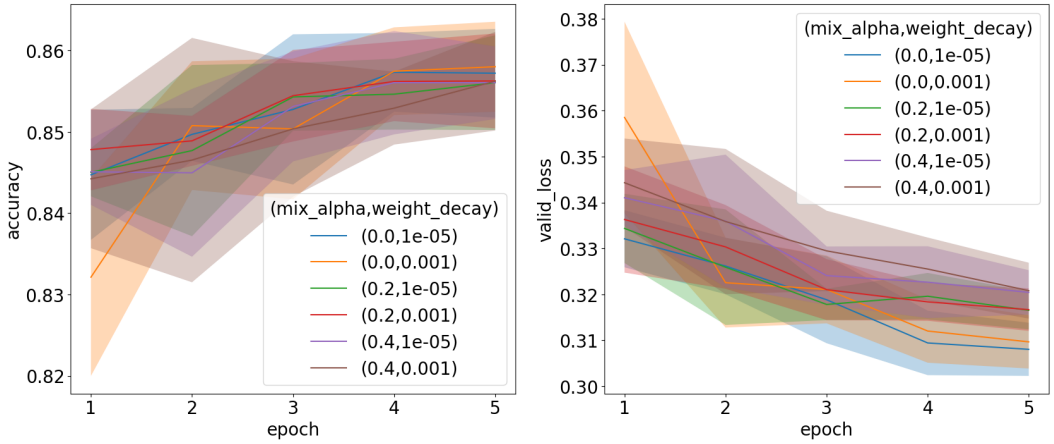


Figure 5.13: Performance per epoch for models with different weight decays and mixup ratios.

5.6.2 Unsupervised Pretraining

In our final experiment, we investigated whether or not the use of DAEs in unsupervised pretraining yield output which may be beneficial to a (supervised) neural network for tabular data. Therefore, we trained a DAE with swap noise (own implementation) for 15 epochs on the Adult dataset. We set the swap noise proportion equal to 15%, and the DAE had 3 hidden layers consisting of 500 units each. The learned weights were subsequently carried over to a supervised

learning network, where the latter also comprised 3 hidden layers consisting of 500 units each. Note that the supervised network had a different output layer than that of the DAE. Therefore, we first kept the transferred weights fixed, and trained only the last layer of the supervised learning network. This is done in order to initialise the network weights. Otherwise, if we simultaneously trained all the weights from the start, random weights to the output layer might have interfered with learned weights in the hidden layers. Once the output layer is initialised with this process, we may train all layers of the network simultaneously. This final stage of training is showed in Figure 5.14, where we compare it to the final stage of training of a model without pretraining. In terms of both accuracy and validation loss, we see that the pretrained model has an advantage over the classifier that was trained-from-scratch. The above result makes sense, since the pretrained model does not have to start from a random initialisation. However as expected, as training continues, this gaps gradually decreases.

Since there are still so many avenues to explore for unsupervised pretraining with DAEs, we do not believe these results to be conclusive. We do not yet know the best way to design DAEs in terms of their network architectures, the amount of noise injected, and the lengths of training cycles. Furthermore, investigation into the preferred way of transferring the learned knowledge from a DAE to the supervised classifier, remains outstanding. The results of this basic implementation seems to indicate that the use of unsupervised pretraining in the context of tabular data warrants further exploration.

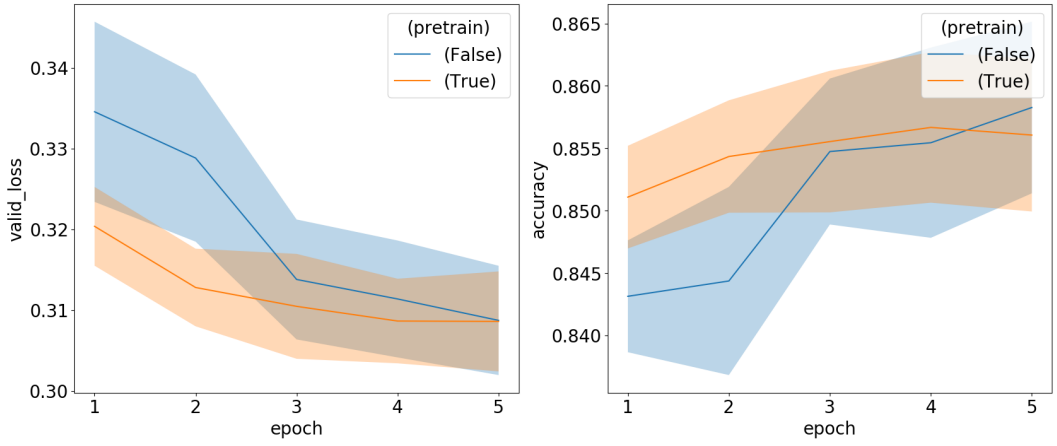


Figure 5.14: The effect of unsupervised pretraining on supervised classification for tabular data.

5.7 Summary

This chapter was devoted to a discussion of the empirical work that was done in order to test some of the latest successful mechanisms in the field of deep learning. Specifically, application of these methodologies were investigated in the context of tabular data.

Fairly disappointingly, in all 9 experiments, we did not find significant differences between the performances of the techniques considered. Therefore, regarding preferences with respect to methodologies for appropriate input representation, for learning from feature interactions and for leveraging sample efficiency, no clear recommendations could unfortunately be made.

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 parameters for all models in all folds, we selected a set of parameters from initial experiments and used them for most of the experiments. But since these parameters are so inter-connected, changing 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 study we set out to investigate a relatively under-explored area of deep learning, *viz.* the application of deep learning approaches to tabular datasets. We reviewed the best approaches for this task, and through empirical work, aimed to gain a better understanding of each of the proposed methodologies. In order to explore deep learning for tabular data, we provided an overview of neural networks and discussed modern advancements in the deep learning field. Some of the modern proposals could then be identified as potentially useful in the context of tabular data.

This chapter provides a summary of our work. In §6.1 the contribution of each chapter is discussed, and in §6.2, some limitations and avenues towards possible improvements are indicated.

We conclude the thesis with a section on promising future directions for research in the field of deep learning for tabular data §6.3.

6.1 Summary

In Chapter 1, the motivation and the objectives of the study were described. It was stated that deep learning for tabular data is an important topic, but in our opinion, also one that has not yet received sufficient attention in the literature. Hence the main issues that 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 (SLT), including various learning paradigms and loss functions; as well as aspects of optimisation and overfitting. The SLT framework served as background for the problems we aimed to solve.

The focus in Chapter 2 included the core concepts in neural networks, and neural network training. Neurons, layers, activation functions and network architectures were described in order to gain an understanding of the mechanics of neural networks. With regard to neural network training, the backpropogation and stochastic gradient descent algorithms were introduced, and illustrated by way of a few examples. The chapter included a brief look at basic regularisation methods for neural networks, as well as a section on representation learning. The aim of the latter section was to facilitate insight into the way that a neural network learns from data, and into what concepts are actually learned. In summary therefore, the contribution of the chapter was to equip us with the fundamentals of neural networks in order to grasp the ideas underlying modern developments in deep learning.

The topic in Chapter 3 was deep learning. The aim of the chapter was to provide insight into the methodologies enabling deep learning to be successful in the NLP and computer vision fields. This provided a good starting point for contemplating ways of improving deep learning for tabular data. The chapter started with an introduction to autoencoders and the concept of using them for unsupervised pretraining in transfer learning. We also discussed the use of data augmentation and dropout as highly effective regularisation techniques. 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. Here we discussed the 1cycle policy and more effective hyperparameter selection as ways in which superconvergence may be achieved. We concluded with a brief discussion (and examples) of both neural network specific and model agnostic tools that may be used to interpret deep neural networks.

Chapter 4 was devoted to the topic of deep learning for tabular data. This chapter entailed a review of recent contributions with regard to the application of deep neural networks to tabular data. As stated in Chapter 1, the review was guided by the main challenges posed by tabular data in this context. We explored various ways of preprocessing continuous features, and of optimally embedding and presenting categorical features. We also investigated approaches towards encouraging networks to learn better feature interactions. These included the use of attention modules and cross-features. In terms of training deeper neural networks for more complext feature learning, the SeLU activation

function was discussed, along with the use of skip connections. Subsequently, we described several approaches towards making deep neural networks more sample efficient. In this section we focused on denoising autoencoders, and on data augmentation techniques such as swap noise and mixup augmentation. The penultimate section described ways of interpreting neural networks for tabular data, and provided an illustration of interpretation by means of knowledge distillation. We concluded with an empirical investigation of superconvergence in the context of tabular data. Hence, Chapter 4 contributed to the technical understanding necessary for tackling challenges in deep learning for tabular data.

Our empirical work is summarised in Chapter 5. The experiments reported on, complement the exploratory study of deep learning for tabular data. We attempted to answer three main questions, *viz.* which approach to use for input representation, for inferring feature interactions, and for enhancing sample efficiency. In the input representation experiments, we evaluated the effect of entity embedding sizes on the performance of neural networks. With regard to feature interactions, we experimented with the attention mechanism, along with use of the SeLU activation function, and of skip connections. Finally, in the sample efficiency section, we attempted to gage the sensitivity of neural networks to the number of training samples. We tested the use of unsupervised pretraining towards alleviating this sensitivity, and investigated the use of mixup- and swap noise data augmentation as a means to prevent overfitting.

6.2 Limitations

There are various aspects that limited the impact of this study. In this regard, there were two main obstacles which we needed to overcome:

- **Access to large compute**: Deep learning techniques are notorious for the computing power they require. We had limited access to cloud computing providers on which some of the experiments were done. Therefore, most experiments were conducted on a small personal machine without a graphical processing unit (GPU). This significantly increased the running time of the experiments, and hindered rapid execution of the various approaches. In future one would want to ensure sufficient computing power to be available, enabling higher quality empirical work.

- **Access to quality code**: At the time of writing, many of the recent developments discussed in this thesis were not accompanied by any official implementation. This forced us to rewrite much of the code that was used to validate the results reported in the literature in order to be able to apply it to our data and models. Sometimes important technical details were omitted from the original papers. This called for some improvisation in order to obtain working examples. Unfortunately this approach brings with it the risk of unintentionally departing from the original intention of some of the proposed methodologies.

Further limitations were:

- **Experiments on a single dataset**: Due to the technical limitations mentioned above, we only had capacity to process a single dataset in our empirical study. This rendered our findings to be inconclusive, although we believe there still to be value in our exploratory study. The possibility remains that the strength of the signal in the Adult data does not make the use of this particular dataset amenable to demonstrating the full power of deep learning approaches. Naturally, repeating experimentation on datasets with different properties, and to applications with different tasks, would have facilitated clearer recommendations with regard to the questions that arise during implementation of deep learning for tabular data. Such an extention is possible, since we have made all or our code available on-line.

- **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 of course not peer-reviewed and subject to change. We did our best to critically evaluate the work cited, and to confirm findings with our own experiments. Although we tried to keep up with the deluge of publications that are currently available, there remains a possibility that new publications arised during the post-review phase of the study.

6.3 Future Directions

With a view to future research directions, in this final section we would like to point the reader to the potential of using *generative models* in the context of deep learning for tabular data. More specifically, it might be worthwhile to study variational autoencoders (Kingma and Welling, 2013) and *Generative Adversarial Networks* (GANs) (Goodfellow *et al.*, 2014). We propose studying VAEs as a means to perform more effective unsupervised pretraining, and we believe GANs to offer a good alternative to generating new training samples. In conclusion, therefore, a brief overview of VAEs and GANs follows below.

VAEs provide a probabilistic manner of describing observations in a latent space. That is, instead of using a single value to describe an attribute in the latent space, as is the cases with standard autoencoders, VAEs use probability distributions to describe a latent space attribute. When using DAEs for unsupervised pretraining, we have seen that they need to be injected with some noise. However, we have seen that neither of the two best noise schemes that we could find (*viz.* swap noise and blank-out), makes complete sense in the tabular dataset environment. We postulate that VAEs may provide a more robust way of doing unsupervised pretraining, since its decoding function learns probabilistic output. Thus, it is not reliant on noise injection. In addition, once trained, the latent probabilistic distribution of VAE can be used to generate additional training samples.

GANs consist of two neural networks, *viz.* a generator (G) and a discriminator (D). Given random noise as input, the task given to G is to generate artificial samples of the data that are indistinguishable from a set of genuine training samples. The task assigned to D is to attempt to discriminate between the artificial and genuine samples. GANs have shown tremendous value in data synthesis, especially in the domains of computer vision and speech synthesis, producing lifelike faces (Karras *et al.*, 2017) and voices (Donahue *et al.*, 2018). We postulate that GANs may be able to achieve similar successes in data synthesis for tabular data, which may be used to artificially enlarge training datasets for supervised learning.

Appendices

Appendix A

Hyperparameter Search

A.1 Width and Depth of Network

This is a very common hyperparameter 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, we 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 A.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.

A.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

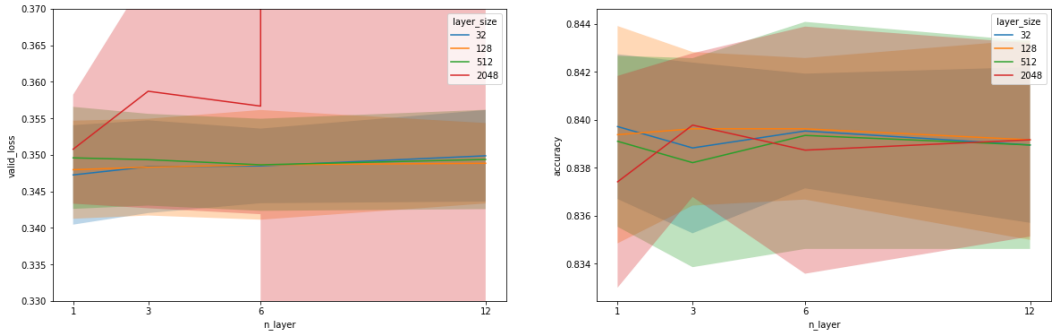


Figure A.1: Effect of the layer width and network depth on the performance on the Adult dataset.

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

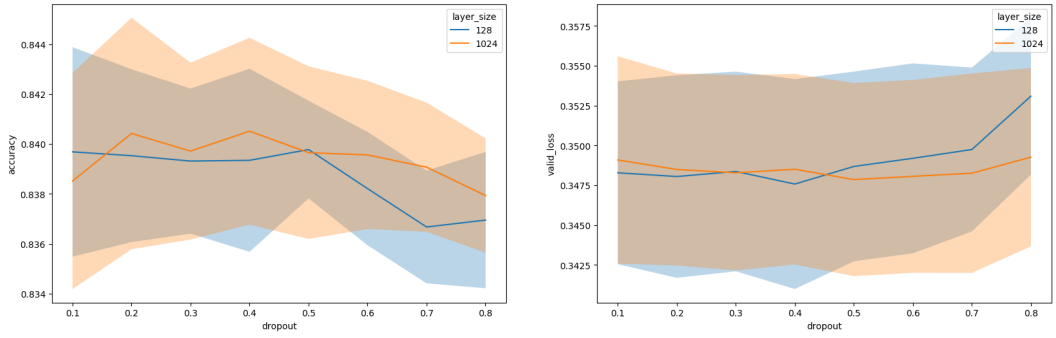


Figure A.2: The effect of dropout on wide and narrow neural networks.

Appendix B

Software and Code

B.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

B.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.

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.

Bourlard, H. and Kamp, Y. (1988). Auto-association by multilayer perceptrons and singular value decomposition. *Biological cybernetics*, vol. 59, no. 4-5, pp. 291–294.

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

Cauchy, A. (1847). Méthode générale pour la résolution des systèmes d'équations simultanées. pp. 536–538.

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

Hinton, G.E. (1990). Connectionist learning procedures. In: *Machine learning*, pp. 555–610. Elsevier.

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

Hinton, G.E. and Zemel, R.S. (1994). Autoencoders, minimum description length and helmholtz free energy. In: *Advances in neural information processing systems*, pp. 3–10.

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. (1966). *Cybernetic Predicting Devices*. JPRS 37, 803. Purdue University School of Electrical Engineering.
Available at: https://books.google.co.za/books?id=l38DHQAACAAJ

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. (1987 6). *PhD thesis: Modeles connexionnistes de l'apprentissage (connectionist learning models)*. Universite P. et M. Curie (Paris 6).

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

Lundberg, S. and Lee, S. (2017). A unified approach to interpreting model predictions. *CoRR*, vol. abs/1705.07874. 1705.07874.
Available at: http://arxiv.org/abs/1705.07874

Maas, A.L., Hannun, A.Y. and Ng, A.Y. (2013). Rectifier nonlinearities improve neural network acoustic models.

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

Mikolov, T., Karafiát, M., Burget, L., Černockỳ, J. and Khudanpur, S. (2010). Recurrent neural network based language model. In: *Eleventh Annual Conference of the International Speech Communication Association*.

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.
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.
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

Ranzato, M., Poultney, C., Chopra, S. and LeCun, Y. (2006). Efficient learning of sparse representations with an energy-based model. In: *Proceedings of the 19th International Conference on Neural Information Processing Systems*, pp. 1137–1144. MIT Press, Cambridge.

Rendle, S. (2010). Factorization machines. In: *Data Mining (ICDM), 2010 IEEE 10th International Conference on*, pp. 995–1000. IEEE.

Rifai, S., Vincent, P., Muller, X., Glorot, X. and Bengio, Y. (2011). Contractive auto-encoders: Explicit invariance during feature extraction. In: *Proceedings of the 28th International Conference on International Conference on Machine Learning*, pp. 833–840. Omnipress.

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

Ruder, S. (2016). An overview of gradient descent optimization algorithms. *CoRR*, vol. abs/1609.04747. 1609.04747.
Available at: http://arxiv.org/abs/1609.04747

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

Rumelhart, D.E. and McClelland, J.L. (1986). Parallel distributed processing: explorations in the microstructure of cognition.

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.

Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *CoRR*, vol. abs/1409.1556. 1409.1556.
Available at: http://arxiv.org/abs/1409.1556

Smith, L.N. (2017). Cyclical learning rates for training neural networks. In: *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pp. 464–472. IEEE.

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

Socher, R., Pennington, J., Huang, E.H., Ng, A.Y. and Manning, C.D. (2011). Semi-supervised recursive autoencoders for predicting sentiment distributions. In:

Proceedings of the conference on empirical methods in natural language processing, pp. 151–161. Association for Computational Linguistics.

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

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

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.