Deep Learning for Tabular Data: An Empirical Study

by

Jan André Marais

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

Supervisor: Dr. S. Bierman

December 2018

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

Declaration

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

Date: .....

Abstract

Deep Learning for Tabular Data: An Empirical Study

J. A. Marais

Thesis: MCom (Mathematical Statistics)

December 2018

English abstract.

Uittreksel

Diepleer Tegnieke vir Gestruktrueerde Data: 'n Empiriese Studie

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

J. A. Marais

Tesis: MCom (Wiskundige Statistiek)

Desember 2018

Afrikaans abstract

Acknowledgements

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

- The UCI Machine Learning Repository (Dheeru and Karra Taniskidou, 2017) for hosting a platform to share datasets.

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
DL	Deep Learning
EHR	Electronic Health Records
kNN	k-Nearest Neighbour
mAP	Mean Average Precision
ML	Machine Learning
MLP	Multi-layer Perceptron
NLP	Natural Language Processing
NN	Neural Network
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
SotA	State-of-the-Art

Nomenclature

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 $\boldsymbol{x}$, $\hat{Y} \subseteq \mathcal{L}$, produced by $h(\cdot)$
$\boldsymbol{y}$	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 *Artifical Neural Networks* (NNs). Whereas until relatively recently, the neural network field was not a very 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/explore?date=all&q=deep%20learning
[3] https://www.semanticscholar.org/search?year%5B0%5D=2000&year%5B1%5D=

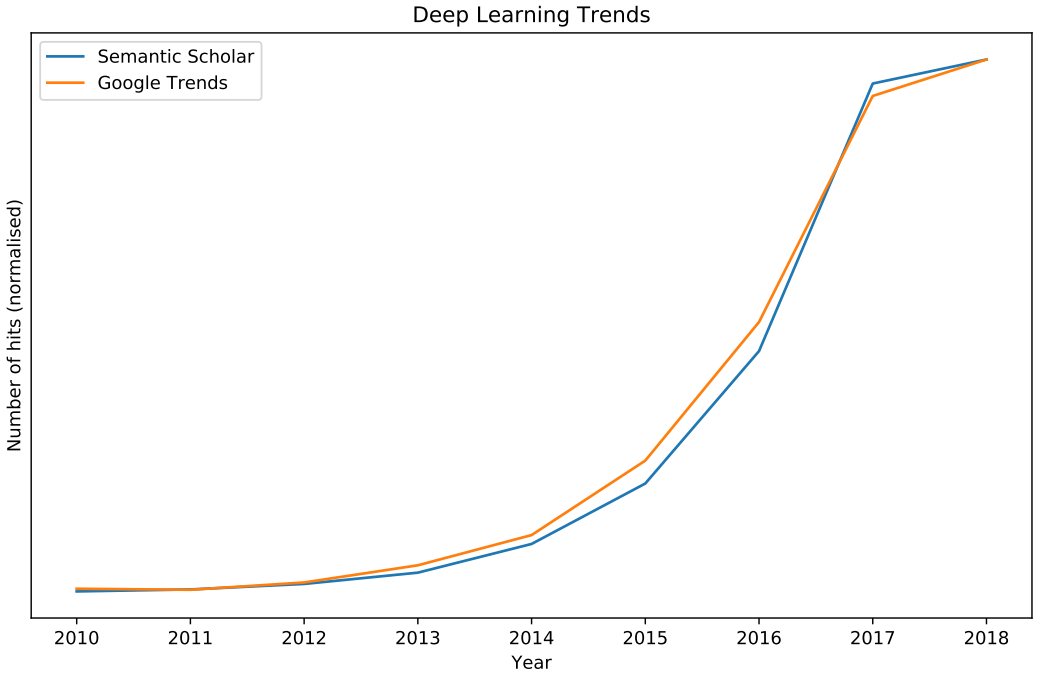


Figure 1.1: The exponential growth of published papers and Google search terms containing the term *Deep Learning*. Sources: Google Trends[2], Semantic Scholar[3]

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 are pixel values, in NLP the data are words and in audio processing the data represent sound waves. This is not a criterion for deep learning algorithms to be successful, but may be viewed as a driver for their sucess in these application domains. It is simpler to model data consisting of the same type of measurements, since each input feature can be treated the same. Furthermore in the above deep learning

2019&q=%22deep%20learning%22&sort=relevance

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 knowledge aquired by humans, and that learned by a deep learning model. For example, in computer vision, advancements in classifying pictures of pets will most likely also be 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 (see *Transfer Learning*).

A data domain in which deep learning does not flourish 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 (Shavitt and Segal, 2018, Song *et al.* (2018)). However, state-of-the-Art (SotA) results are reported only rarely (de Brébisson *et al.*, 2015) (and in this competition[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 extract of this dataset. The data were collected during an American census with the aim of predicting whether or not an indivdual earns more than $50,000 a year.

[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

	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 each column may contain different data types. Some columns may consist of continuous measurements, whereas other columns contain discrete or categorical measurements. Hence the columns are heterogeneous with respect to data types. 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-hopsital mortality rates, and 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).
- Drug discovery (Klambauer *et al.*, 2017).

Tabular datasets come in all shapes and sizes. The number of rows can range from hundreds to millions, and the number of columns also has no limits. It is not unusual for tabular datasets to be noisy. A proportion of the observations may have missing features and/or incorrect values. Continuous measurements may be based upon vastly different scales, some containing outliers, whereas categorical features may have high cardinality which leads to sparse data.

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

During the construction of models for tabular datasets, the most important step in terms of leading to improvements in the 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 in the face of 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 having different data types.

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

1.3 Challenges of Deep Learning for Tabular Data

Some of the challenges of deep learning for tabular data have been alluded to in earlier sections of this chapter. These will form the framework for the literature review which follow later on, therefore they are summarised below, in terms of relevant questions to ask when applyting deep learning in the context of tabular data:

- **How should input features be represented numerically?**: We have mentioned that tabular data consist of mixed feature types, *i.e.* a combination of categorical and continuous features. The question here

relates to how these 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 types, we will need a way to effectively learn the interactions between them and also a way to learn how they relate to the target. This 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 the datasets use 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 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 many applications.

Clearly there are plenty of considerations when it comes to using deep learning for tabular data. The main objective of this thesis is to find the best ways of overcoming the above challenges. 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 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 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

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

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 other 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. There all code, data and necessary documentation in order to reproduce the experiments that were done in this thesis will be 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 summary of the thesis.

1.4 Overview of Statistical Learning Theory

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

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

Many kinds of tasks can be solved using statistical learning. One of the most common learning tasks is that of *classification*, where it is expected of an

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

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, *i.e.* a dataset. 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, superivised 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 design 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 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 be a linear function of the input. Let $\hat{y}$ be the predicted response. We define the output to be

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

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

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

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

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

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

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

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

We can further generalise linear regression to the classification scenario. First, 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 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. Here we start by introducing 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, which we denote by $\hat{G}$. One approach would be to represent G by an indicator vector $\boldsymbol{Y}_G : K \times 1$, with elements all zero except in the G-th position, where it is assigned a 1, *i.e.* $Y_k = 1$ 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 can use a linear model $\hat{f}_k(\boldsymbol{X}) = \hat{\boldsymbol{w}}_k^T \boldsymbol{X}$ to approximate the true function. The linear model for classification divides the input space into a collection of regions labelled according to the classification, where the division is done 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) can be classified as follows:

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

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

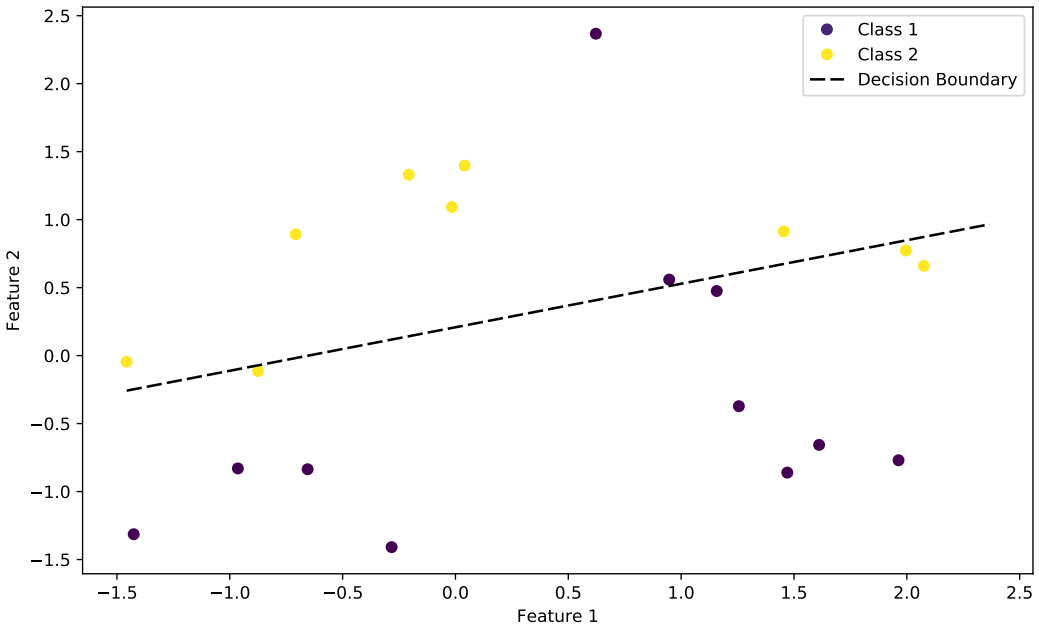


Figure 1.2: Linear model on simple binary classification dataset.

transform of $\hat{f}_k(\boldsymbol{x})$. That is,

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

Through this transformation, the estimates of the posterior probabilities both 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 with an estimate $\hat{f}$. 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, motivation, objectives and theoretical background of 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- and manifold learning, we then attempt to gain insight into what happens inside an 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 fighting 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 interpretting neural networks and their predictions.

Chapter 4 may be viewed as a core chapter of this 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 evaluate several claims made in the literature. The aim of the chapter is to evaluate and compare different approaches to 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 illustrate a way in which resulting neural networks may be interpreted by means of permutation importance and knowledge distillation.

The thesis concludes in Chapter 6, where we summarise the work, some highlights and the main take-home points. The limitations of this study are discussed, and promising future research directions are identified.

Chapter 2

Neural Networks

2.1 Introduction

Not unlike most supervised machine learning models, a neural network (NN) 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 a output signal to the next neuron. Figure 2.1 (a) provides a schematic of a biological neuron.

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

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

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

Although recently there has been plenty of excitement around 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 allowed neural networks to be more effective, leading to the renewed interest in the field.

The aim of this chapter is to provide an overview of neural networks, emphasising the 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. Finaly, the chapter concludes with a section on representation and manifold learning (§2.4) in an attempt to understand what a neural network is actually doing.

2.2 The Structure of a Neural Network

2.2.1 Neurons and Layers

In basic terms, a neural network processes an input $boldsymbolx$ 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, and follow with 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) (b)

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

In equation form, this function can 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 b 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 $\boldsymbol{x} = [x_1 \quad x_2 \quad \dots \quad x_p]^\intercal$ is the input, $\boldsymbol{w} = [w_1 \quad w_2 \quad \dots \quad w_p]^\intercal$ the weights and y the output. We may compress the above equation to $\boldsymbol{w}^\intercal \boldsymbol{x} = y$, where $\boldsymbol{x}$ includes the bias term and the weight vector $\boldsymbol{w}$ a unit element, *i.e.* $\boldsymbol{x} = [1 \quad x_1 \quad \dots \quad x_p]^\intercal$ is the input, $\boldsymbol{w} = [b \quad w_1 \quad \dots \quad w_p]^\intercal$.

The weights convey the importance of each input features in predicting the target. The larger $|w_k|$ is, the greater is the contribution of x_k towards the output. If $w_k = 0$, x_k has no influence on the target. However the weights are unknown and therefore we need to estimate them.

In linear regression this is done by means of the method of ordinary least squares. 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

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

use of two neurons, where the second neuron accepts the same input as the first neuron, 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.

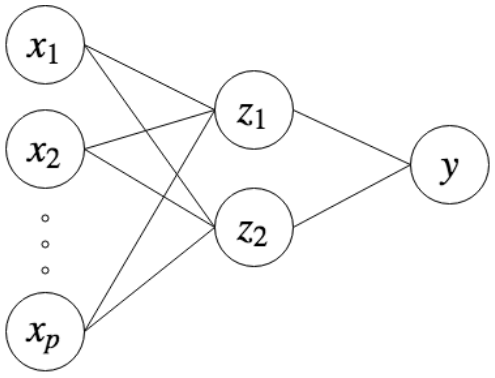


Figure 2.2: A simple neural network accepting p-sized inputs, with one hidden layer which has two neurons.

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 the first two neurons 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 these two neurons is what 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, W, and the third neuron, $\boldsymbol{w}_3$, can be collapsed into a single vector $\boldsymbol{w}$, effectively reducing all of the neuron operations back into a single neuron representation and thus the fitted model is still linear. In order to fit a non-linear model, a non-linear transformation

function is applied to the output of each layer. This function is called an *activation function*.

Subsequently the neural network equation can be written as

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

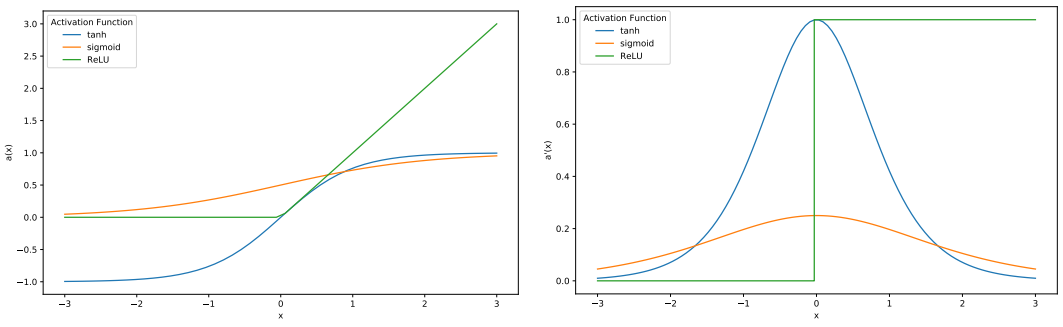
where a_1 denotes the activation function applied after the first (linear) layer, and where a_2 is the the activation function applied after the final layer.

The introduction of non-linear activation functions serves to enlarge the class of functions that can be approximated by the network *i.e.* making it possible for the network to learn complex, non-linear relationships between the inputs and the outputs. Next, we briefly discuss the various activation functions.

2.2.2 Activation Functions

There are plenty of activation functions to choose from, since any simple non-linear and differentiable function can be used. Originally, the *sigmoid* activation function was a common choice (Rumelhart *et al.*, 1988). It can be expressed as $\text{sigmoid}(x) = \frac{1}{1+e^{-x}}$. The shape of the sigmoid activation can be seen in Figure 2.3 (a). It has an S-shape and its values range between 0 and 1. The reason it fell out of favour is because of issues related to the gradient based optimisation procedure of neural networks (which we discuss in more detail in Section 2.3). One of the issues is that the values of sigmoid activations are not centered around zero. This may cause the gradient weight updates to veer too far in different directions.

The hyperbolic tangent or the *tanh* activation function, on the other hand, returns outputs centered around zero. It is expressed as $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ and its shape can be seen in Figure 2.3 (a). But the problem with both the sigmoid and tanh activation functions is that they can cause the gradients to saturate during training. By observing the tails of the sigmoid and tanh functions, it is clear that the gradient tends to zero as $|x| \to \infty$. This can cause the weight updates to be very close to zero and thus resulting in the network getting stuck at a certain point in the parameter space. Furthermore, the maximum gradient of the sigmoid activation function is 0.25 (at $x = 0.5$), and because of the nature of the chain rule this causes the lower layers to train much slower than the higher layers. The tanh activation function typically have larger derivatives



(a) (b)

Figure 2.3: Plots of various activation functions (a) and their local derivatives (b).

than the sigmoid and 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 here are plotted in Figure 2.3 (b). This will become more obvious after Section 2.3.

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

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

Again, its shape and derivative is plotted in Figure 2.3 (a) and (b), respectively. The ReLU limits the gradient vanishing problem as its derivative is always 1 when x is positive. 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 gets clamped to zero, then its weights will get zero gradient and may remain permanently "dead" during training. This sparsity of the activations is what some believe is the reason for the effectivenss of ReLUs (Sun *et al.*, 2014). If dead activations wants to be avoided, alternative activations functions are PReLUs (He *et al.*, 2015*a*) and Leaky ReLUs (Maas *et al.*), but they fall out of the scope of this work.

Typically, choosing the right activation function for a specific task comes down to trial and error, however, in most cases it would be sufficient to use ReLUs after each hidden layer. When doing classification, it is often useful to produce outputs between 0 and 1 for each category as estimates of the

conditional class probabilities. Therefore we may use a sigmoid activation function on the output layer. When doing (single label) multiclass classifcation it is also desirable for these ouputs to sum to 1, similar to prababilities. In that case we may use an activation function called the *softmax*, which is exactly the logit transformation introduced in §1.4: $\text{softmax}(\boldsymbol{x})_k = \frac{e^{x_k}}{\sum_{l=1}^{K} x_l}$. For regression tasks, we mostly omit an activation function on the output layer.

In our experiments we will show how the different activation functions perform on tabular data and compare it to the *Scaled Exponential Linear Units* (SELUs) (Klambauer *et al.*, 2017) activation, which is supposed to allow us to more effectively train deeper neural networks. The neural network depth refers to its number of hidden layers, whereas the width of a layer refers to the number of neurons it consists of. The choice in 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 capacity or the flexibility of a neural network; referring to its ability to model complex functions. The bigger the network, the more flexible the model is. Like with most such parameters, increasing the model size is usually benefical up to a certain point, after which its test performance starts to degrade (overfitting). In addition, more layers and more neurons adds to the time taken and hardware required to train the neural network.

Thus the challenge is to find a network size that is large enough to capture all the complexities in the data, but small enough to avoid overfitting. Currently the best way of finding the optimal size of a network for a given problem is by experimentation. Note, that this is true for many of the compoments of neural networks and deep learning. Later in this chapter, §2.3.5, we will see why tunining the network size is not necessarily the best way of controlling overfitting.

Theoretically, according to the universal approximation theorem (Cybenko, 1989), a neural network with a single hidden layer and a finite number of neurons can approximate any continuous function. Which begs the question as to why we need more hidden layers. Even though a neural network can represent any function, it does not mean that the available learning algorithms

can find these optimal weights (Ba and Caurana, 2013). It may also be that the number of neurons needed for a single hidden layer network to represent a specific function is infeasibly large. By choosing deeper networks we are assuming that the function we are trying to learn is composed of several simpler functions. Building in this prior proves to useful empirically (Goodfellow *et al.*, 2016, pp.197-198), especially for tasks of a hierarchical nature like in computer vision.

In our experiments we will see what effect the layer width and network depth have on the generalisation performance on different tabular datasets. The networks used for tabular data learning problems are typically much shallower than those used for say computer vision and NLP. We investigate why this is the case. Later in this chapter we view this problem from a representation learning perspective which will also give us some insight on the matter. But first, in the next section, we discuss the process of training a neural network.

2.3 Training a Neural Network

2.3.1 Optimisation

Statistical learning and optimisation are closely related. Optimisation refers to the task of either minimising or maximising some function $J(x)$ by altering x. The function we want to optimise is called the *objective function*. When we are minimising the objective function, we may also refer to the objective function as the *cost* or *loss function*. These terms will be used interchangeably throughout the thesis.

As mentioned in the previous chapter, 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 for the optimisation of neural networks.

Consider the MSE loss function:

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

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

To find the weights, $\boldsymbol{w}$, that minimise L, we can 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. In order 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. Therefore, once we have determined this direction, we can update the weights by a step in the opposite direction - thereby reaching a smaller value of L.

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

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

After obtaining the above N partial derivatives, an update at the $(r+1)$-th iteration may be obtained as follows:

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

where γ is called the *learning rate* and determines the size of the step taken towards the optimal direction. 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; we will discuss its significance in §2.3.6.

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

Note that a weight update is made by evaluating the gradient over a set of observations, $\{\boldsymbol{x}_i, i = 1, \ldots, n\}$. One of the advantages of gradient descent is that at an iteration, the gradient need not be computed over the complete training dataset, *i.e.* $n \leq N$. When updates are iteratively determined by using subsets of the data, the process is called *mini-batch gradient descent*. This is extremely helpful in large-scale applications, since it obviates computation of the full loss function over the entire dataset. This leads to faster convergence, because of more frequent parameter updates, and allows processing of data sets that are too large to fit into a computer's memory. The choice regarding batch size depends on the available computation power. 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. 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). Note at this point that the collection of iterations needed to make one sweep through the training data set is called an *epoch*.

The 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 compared to only using one example. However, note that it remains common to use the term SGD when actually referring to mini-batch gradient descent. Gradient descent in general has often been regarded as slow or unreliable but it works well for optimising neural networks. SGD will most probably not find even a local minimum of the objective function. It typically however finds a very low value of the cost function quickly enough to be useful.

2.3.2 Optimisation Example

To illustrate the SGD algorithm, consider the linear model in a classification context. Suppose we are given a training dataset with two-dimensional inputs and only two possible classes. Let the data be generated in the same way as described in (Hastie *et al.*, 2009, pp. 16-17).

We want to fit a linear regression model to the data such that we can classify an observation to the class with the highest predicted score. In the binary case it is only necessary to model one class probability and then assign an observation to that class if the score exceeds some threshold (usually 0.5), otherwise it is assigned to the other class. Therefore the decision boundary is given by $\{\boldsymbol{x} : \boldsymbol{x}^T\hat{\boldsymbol{w}} = 0.5\}$.

The example is illustrated in Figure 2.4. The colour shaded regions represent the parts of the input space classified to the respective classes, as determined by the decision boundary based upon OLS parameter estimates. Gradient descent was applied to the determine the optimal weights using a learning rate of 0.001. Since the total number of training observations are small, it is not necessary to use SGD. In Figure 2.4, the dashed lines represent the decision boundary defined by the gradient descent parameter estimates at different iterations. We observe that initially the estimated decision boundary is far from the OLS solution, but as the update iterations proceed, the decision boundary is rotated and translated until finally matching the OLS line. It took 29 iterations for the procedure to reach convergence.

2.3.3 Backpropogation

In Section 2.3.1 we discussed how to fit a linear model using the Stochastic Gradient Descent optimisation procedure. Currenlty, SGD is the most effective way of training deep networks. To recap, SGD optimises the parameters θ of a network to minimise the loss,

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

With SGD the training proceeds in steps and at each step we consider a mini-batch of size $n \leq N$ training samples. The mini-batch is used to approximate the gradient of the loss function with respect to the paramaters by computing,

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

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

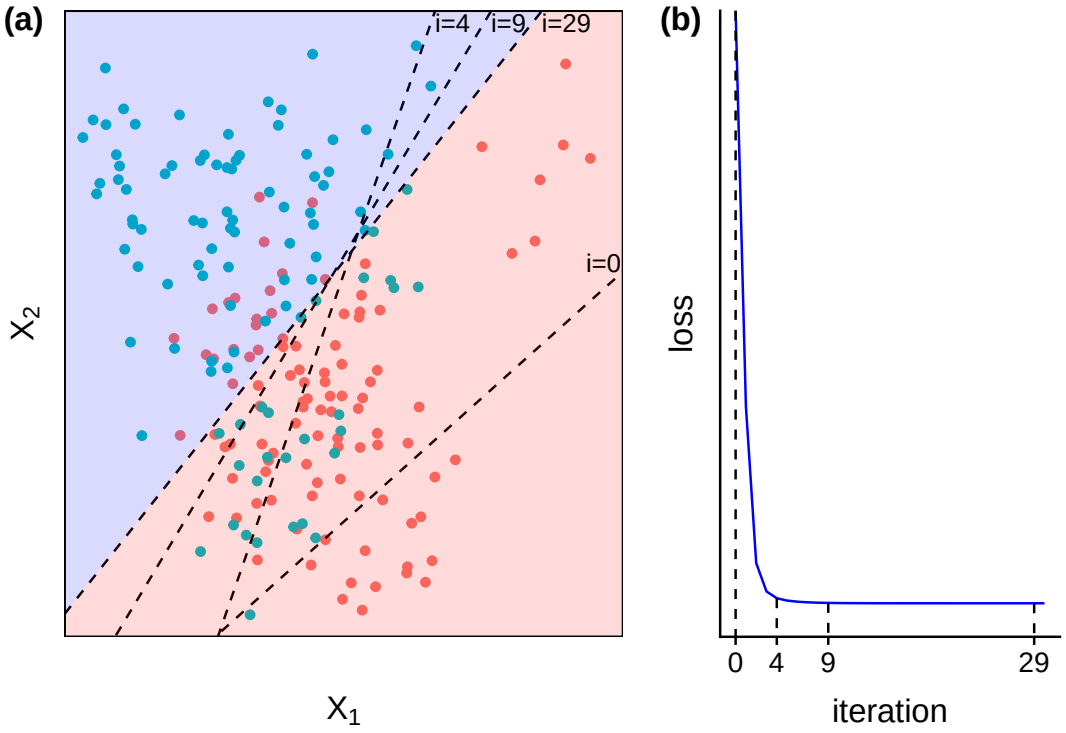


Figure 2.4: Plots of the gradient descent example. (a) The data points in input space. The shades in the background represent the class division in input space, with the decision boundary determined by linear least squares estimation. The dashed lines represent the decision boundaries learned at different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

This section discusses the same procedure, but applied to a simple single hidden layer neural network for multiclass classification, decomposed as:

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

where $\sigma(\cdot)$ is the sigmoid activation and $g(\cdot)$ the softmax activation. The neural network has a set of unknown adjustable weights that defines the input-output function of the network. They are the parameters of the linear function of the inputs, $\boldsymbol{\alpha}_m = (\alpha_{0m}, \alpha_{1m}, \ldots, \alpha_{pm})$, and the paramaters of the linear transformation of the derived features, $\boldsymbol{\beta}_k = (\beta_{0k}, \beta_{1k}, \ldots, \beta_{mk})$. Denote the complete set of parameters by θ. Then the objective function for regression can be chosen as the sum-of-squared-errors:

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

and for classification, the cross-entropy:

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

with corresponding classifier $G(\boldsymbol{x}) = \arg\max_k f_k(\boldsymbol{x})$. Since the neural network for classification is a linear logistic regression model in the hidden units, the paramaters can be estimated by maximum likelihood. According to Hastie *et al.* (2009, p. 395), the global minimiser of $L(\theta)$ is most likely an overfit solution and we instead require regularisation techniques when minimising $L(\theta)$. Furthermore, as the network becomes larger, MLE becomes intractable.

Therefore, one rather uses gradient descent and the *backpropogation* algortihm (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 by iterative application of the chain rule for differentiation. This is done by a forward and backward sweep over the network, keeping track only of quantities local to each unit.

In detail, the backpropogation algorithm for the sum-of-squared error objective function,

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

is as follows. Following the chain-rul, the relevant derivatives for gradient descent are:

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

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

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

Now write the gradients as

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

The quantities, δ_{ki} and s_{mi} are errors from the current model at the output and hidden layer units respectively. From their definitions, they satify the following,

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

which is known as the backpropogation equations. Using this, the weight updates can be made with 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 backpropogated via the backpropogation equations to obtain s_{mi}. These are then used to update the weights. This approach extends naturally to any size network and differentiable layers (functions).

Backpropogation is simple and its local nature (each hidden unit passes only information to and from its connected units) allows it to be implented efficiently in parallel. The other advantage is that the computation of the gradient can be done on a batch (subset of the training set) of observations. This allows the network to be trained on very large datasets.

In summary, training a neural network consists of these four steps:

1. Initialise the network weights: give random set of numbers to the network parameters.
2. Forward propogation: pass the input through the network layers to produce an output.
3. Calculate the error: compare the predicted output with the true output and measure the difference using an objective function.
4. Backward propogation: calculate the gradients of the objective function with respect to the weights and update the weights accordingly.

These four steps are typically repeated until convergence of the loss function. It can take many training epochs for the objective function to converge.

2.3.4 Weight initialisation

Before one begins training a neural network, one needs to initialise its weights. We expect a well trained neural network to have an equal number of weights smaller than zero and greater than zero. If we initialise all weights to be zero, every neuron will compute the same output and therefore prodcue the same gradient and undergo the same weight update. We still want to initialise the weights as small as possible but each one unique. Sampling from zero mean and unit variance Gaussian distribution is a natural fit, however, the variance of the outputs from a randomly initialised neuron grows with increasing number of inputs. Therefore we may scale the weight vector by the square root of its number of inputs to normalise the variance to 1. This ensures that all neurons in the network initially have approximately the same output distribution and empirically improves the rate of convergence. (He *et al.*, 2015*b*) derived a weight initialisation specifically for ReLU neurons.

2.3.5 Basic Regularisation

From §2.2.3 it seems that smaller neural networks can be preferred if the data is not complex enough to prevent overfitting. However, this is untrue and there are much better ways of regularising neural networks, which we discuss shortly and in the next chapter. 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. Whereas the local minima of larger networks are much better in terms of their actual loss (Choromanska *et al.*, 2014).

One of the preferred ways of fighting overfitting in neural networks is by using L1 or L2 regularisation, *i.e.* adding a penalty term proportional to the magnitude of the weights to the objective function. This is exactly what is done in Ridge Regression and the Lasso (Hastie *et al.*, 2009, Ch. 4). In L2 regularisation we add the term $\frac{1}{2}\lambda w^2$ to the objective and in L1 regularisation the penalty term $\lambda|w|$. These penalties forces the weights to be small and the λ parameter controls the strength of the regularisation. The "$\frac{1}{2}$" in front of the L2 penalty is added for convenience to make the derivative equal to λw. This makes it then equivalent to decaying each of the weights linearly towards zero: $w' = w - \lambda w$, which is also known as *weight decay*. In practice, L2 regularisation outperforms L1 regularisaion.

Another basic way of preventing the neural network to overfit is to stop the training process "early" and to not train until the training loss converges. A converged training loss is not equivalent to an optimal test loss and therefore the test or validation loss should be observerd during training. One approach is to stop training when 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.

2.3.6 Adaptive Learning Rates and Annealing

Although the right learning rate can reduce training time and improve performance, there is no silver bullet when it comes to setting the parameter. A small learning rate slows downs the training time, but is safer against overfitting and overshooting the optimal solution. With a large learning rate, convergence may be reached quicker, but the optimal solution may not have been found. One could do a line search of a range of possible values, but this usually takes too long for bigger networks.

However, one does not need to keep the learning rate fixed throughout the training process. A popular approach is to decrease the learning rate by a fraction after a certain number of epochs or as the validation loss starts to converge, as done in (He *et al.*, 2015*a*) for example. The intuition is that larger steps can be taken when we are still far away from an optimal position on the loss surface, and then gradually take smaller steps once we get closer to not overshoot it. Note, that this means that we would also need to tune the rate of decrease and the time steps of each decrease during training. Fortunately, it is believed that the learning algorithms are not that sensitive to this choice.

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

reader can refer to the cited publications as it falls out of the scope of this work.

2.4 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 will 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 the properties of the input given to it. Thus a large part of building machine learning models is to find the best way of representing the raw data to make it easier for the models to extract useful information. This is typically a laborious manual task of creating, analyzing, evaluating and selecting appropriate features[2] requires practitioner expertise and domain knowledge. This *feature engineering* process is more trial-and-error than a systematic recipe. Therefore if one can effectively automate this process, it will save a lot of time and raise the performance ceiling of models. Automatically learning representations of the data that make it easier to extract useful information for classifiers or other predictors is called representation learning (Bengio *et al.*, 2013).

A neural network can be viewed from the perspective of representation learning. 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 learns a representation of the data that is optimal for classification.

Each of the simple but non-linear modules of a neural network transforms the representation at one level (starting with the raw input) into a representation at a higher, slightly more abstract level. With the composition of enough such transformations, very complex functions can be learned (Lecun *et al.*, 2015).

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

These transformations can emphasise (and create) features that are important for discrimination and drop those which are redundant.

Let us work through a simple example to illustrate the representations learned by a neural network. Consider a dataset with two classes; the two curves on a plane shown in Figure 2.5. Clearly, the observations from the two classes are not linearly separable in their raw form. Thus if we fit a single layer neural network (*i.e.* only an output layer) to this data, we will get an unsatisfactory decision boundary, since the decision boundary can only be linear, as shown in Figure 2.6. However, if we fit a two-layer neural network, where the hidden layer has two neurons and a sigmoid acitvation to the same dataset, the decision boundary perfectly separates the two classes. This is shown in Figure 2.7.

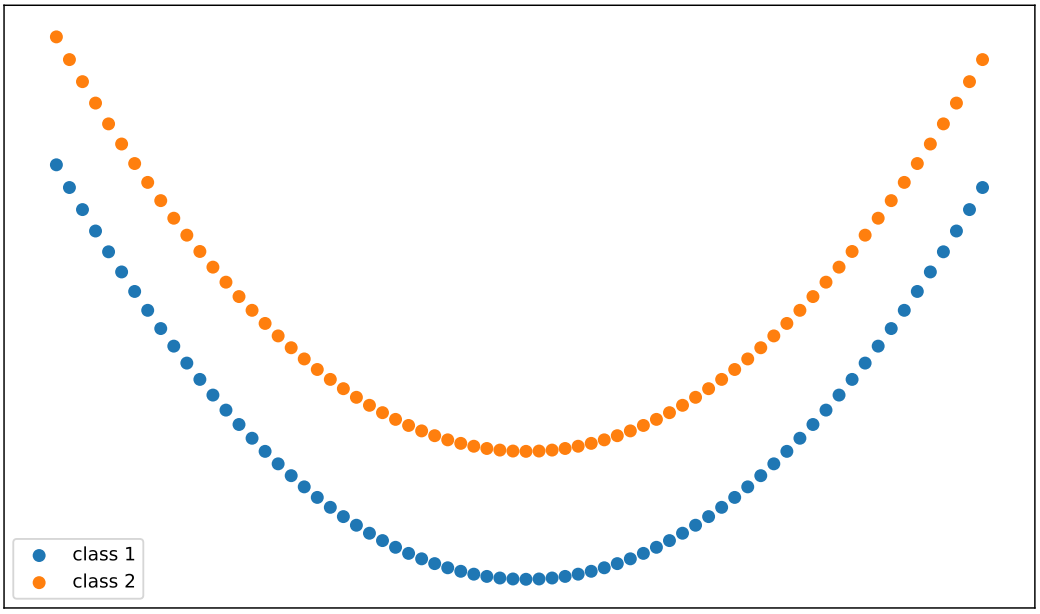


Figure 2.5: Simple dataset with two linearly inseparable classes.

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. This shows how the hidden layer projected the input data into a space where the observations from the two classes are linearly separable. Which leaves it to the final layer to find the best hyperplane between them.

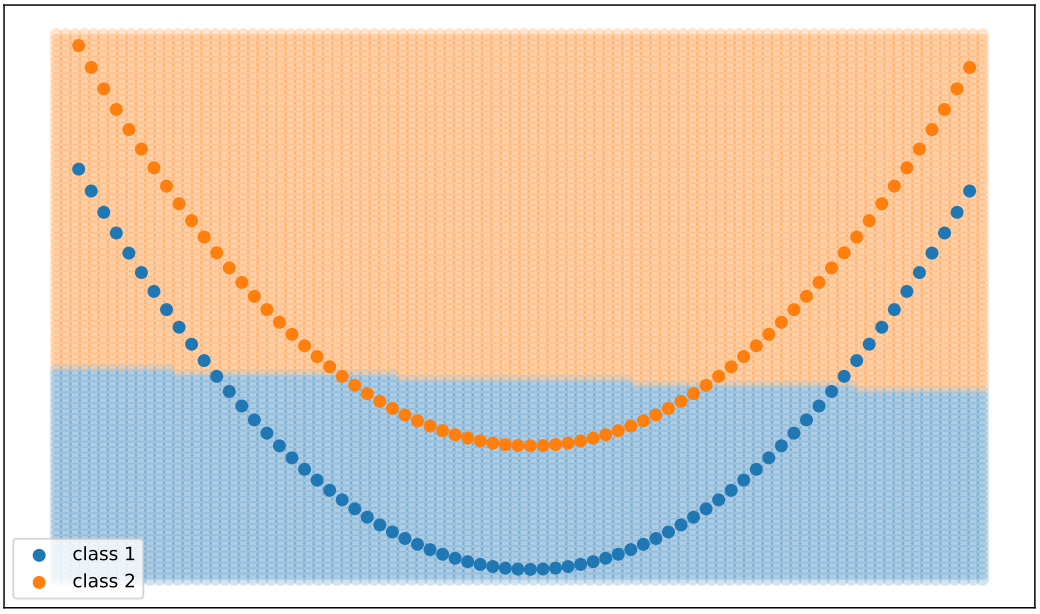


Figure 2.6: Decision boundary of 1-layer neural network.

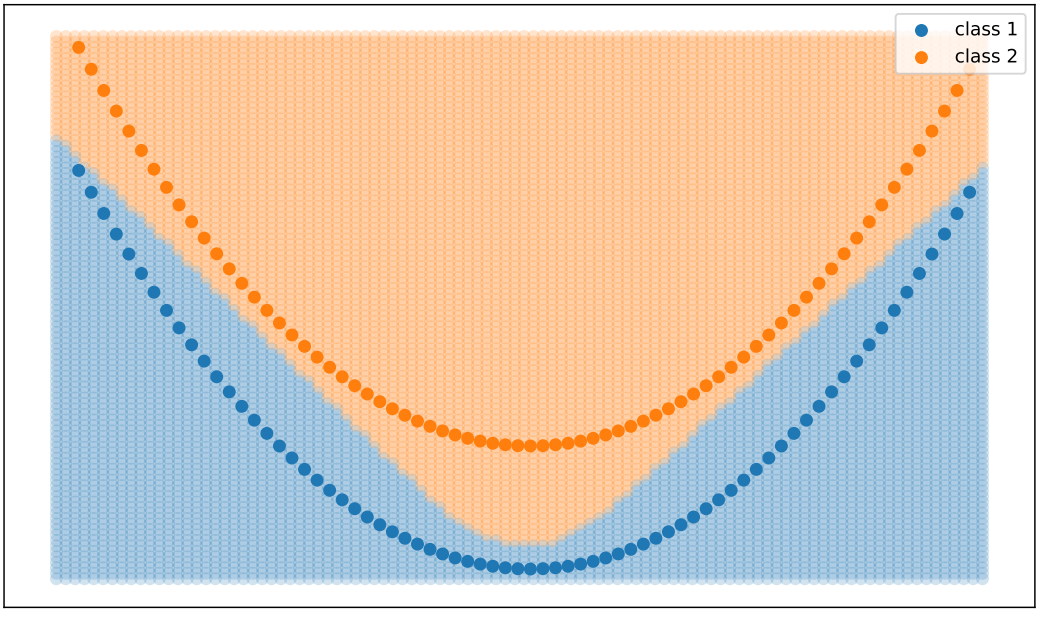


Figure 2.7: Decision boundary of 2-layer neural network.

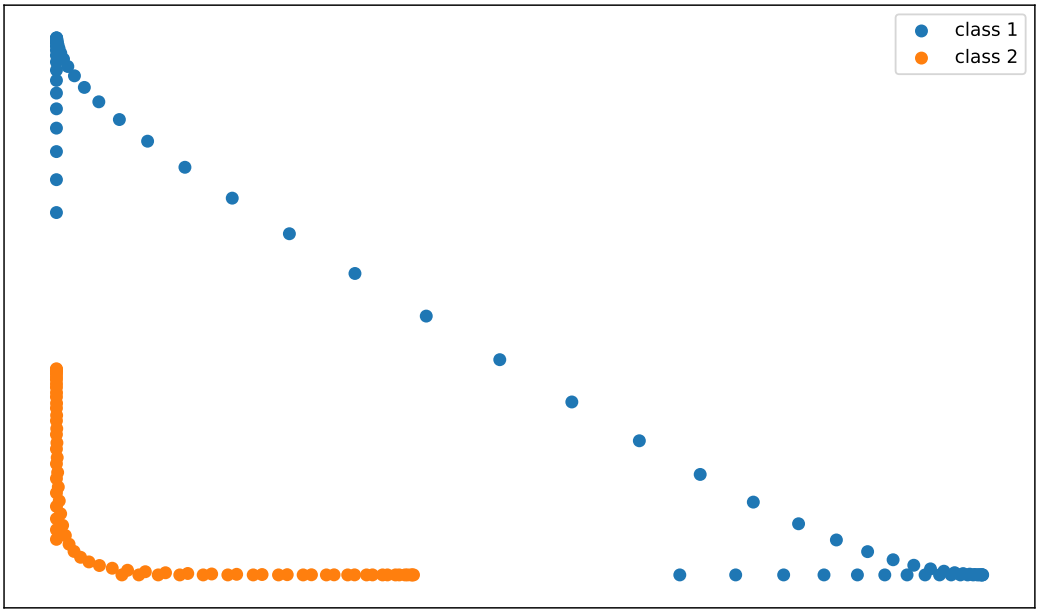


Figure 2.8: Hidden representation of 2-layer neural network.

Of course this is a very simple example, but the same concepts apply to more complicated datasets and models. However, even though 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 optimsation procedures and architecture design arises. Without the aforementioned, it is likely that the network will get stuck in a sub-optimal local minima, not being able to find the optimal representation of the data. In the following chapters we explore the approaches available to find optimal represenations of tabular data for classification and regression tasks.

[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. The difference between a classical neural network and a deep neural network, is merely its number of layers. Even a network with two hidden layers is sometimes referred to as deep.

In this work, we use the term deep learning to refer to modern developments in the space of neural networks. In the last decade, tons of contributions have been made to the field: new types of layers, different ways of training and novel approaches to fight overfitting. Reviewing these developments requires much more space than a chapter, therefore, for the purpose of this work, we only highlight the developments we feel are important for our next chapter on deep learning for tabular data.

Practically all deep learning developments are aimed at combatting overfitting, either explicitly or implicitly. Since neural networks can already approximate any function, but our learning algorithms cannot necassrily find these solutions, we need ways of learning more efficiently from the data. This chapter starts by introducing a different class of neural networks called *autoencoders* (§3.2). Autoencoders are mostly used as an unsupervised learning technique to help us learn more robust representations from data without labelled data, which we may then transfer to supervised learning algorithms. This process of transferring knowledge from one network to another is called transfer learning, which we discuss in §3.3.

Thereafter, we look at layers that help with regularisation: *dropout* (§3.4.1) and *batch normalisation* (§3.5.1). Data augmentation is a technique used to

artificially enlarge the training dataset and thus also helps fighting overfitting. We discuss this in §3.4.2. The other types of layers we look at in this chapter are, *skip-connections* (§3.5.2), *attention* modules (§3.5.4) and *embedding* layers (§3.5.3). They are designed to make deeper neural networks more effective, let them focus more on the important features and to process discrete input, respectively.

Then we devote a section to a training policy called the *1cycle* policy (§3.6). This method provides a way of automatically finding a good learning rate and drastically reduces the number of training iterations needed. We conclude this chapter with a discussion on the interpretability of neural networks (§4.5).

3.2 Denoising Autoeoncoders

The basic autoencoder is a neural network that is trained to attempt to reconstruct its inputs. The simplest form of an autoencoder is a neural network with with one hidden layer and an output layer the same size of the input layer (illustrated in Figure 3.1). The linear layer transforming the input to the hidden layer is referred to as the encoder, $\boldsymbol{z} = f(\boldsymbol{x})$, and the layer producing the output from the hidden layer is called the decoder, $\boldsymbol{x}' = g(\boldsymbol{z})$. The autoencoder can be trained in the same way as any other neural network, but by minimising a reconstruction loss. A reconstruction loss measures the distance between the reconstruction of the input and the actual input and thus the MSE loss function is common choice for continuous data.

Autoencoders technically belong to the self- (or semi-) supervised class of methods, although many still think of it as unsupervised. It is unsupervised in the sense that it does not require labelling, but it is stll supervised in the sense that it predicts an ouput.

In this setting, if the number of neurons in the hidden layer is greater than or equal to the number of input features, the autoencoder will be able to perfectly reconstruct $\boldsymbol{x}$ from $\boldsymbol{z}$, *i.e.* $\boldsymbol{x}' = \boldsymbol{x}$. However this is not a very useful model and instead autoencoders are usually built 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). When there are no non-linear activation functions

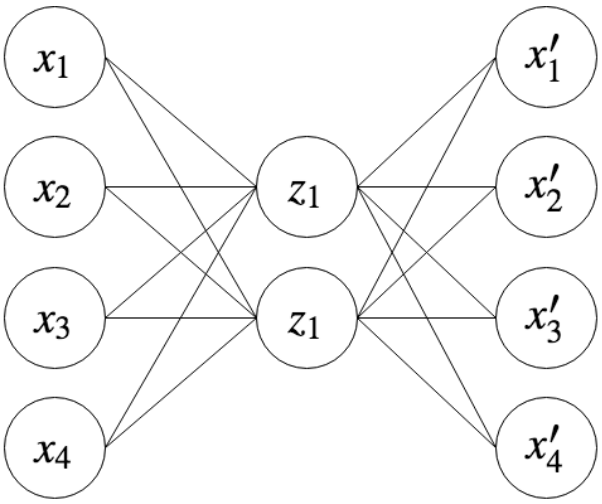


Figure 3.1: A simple single hidden layer autoencoder with 4-dimensional inputs and 2 neurons in the hidden layer.

after the linear layers, one can show that this autoencoder is equivalent to a *Principal Component Analysis* (PCA) of the inputs.

There is no restriction in the number or the size of the layers used for the encoder and decoder, and if activation functions are used, one can potentially learn a more powerful non-linear generalisation of PCA. However, if the encoder and decoder networks are allowed too much capacity, the autoencoder can learn to reconstuct the inputs without learning useful properties of the data. It is hard to tell if an autoencoder has learned a useful latent representation of the data. One way to evaluate it is to use the features extracted by the encoder for a supervised learning task and compare its performance with a model using the raw data as inputs. Autoencoders have also been used to initialise the weights of a supervised learning network, *i.e.* using the learned weight of the autoencoder as the initial weights for the supervised learning network of the same size (Larochelle *et al.*, 2009).

An autoencoder can learn to perfectly copy the input if the encoder and decoder have enough capacity, even though the latent representation is of a smaller diemsion than the inputs, and thus will not learn useful features of the data. Therefore we may want to consider other types of contraints to impose on the autoencoder.

A *denoising autoencoder* (DAE) (Vincent *et al.*, 2008) first adds noise to its inputs before it is passed to the encoder. Thus in order to minimise the reconstruction loss, the DAE should learn how to reconstruct the original input

from a corrupted version of itself. (Alain and Bengio, 2014) shows that DAEs do learn useful structures of the data.

The choice of the type of noise added to the inputs depends on the data types. One can 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. If there is too much noise in the corruption step, the DAE might not learn anything useful. The type and amount of noise to be use are things that should be experimented with.

Another type of autoencoder is a *Variational Autoencoder* (VAE) (Kingma and Welling, 2013). Instead of encoding the data into a latent vector representation, a VAE encodes the data as a Gaussian distribution. In the decoding step, observations are then sampled from the learned distribution before passing it to the fully connected layers. This is an interesting direction for future research but not in the scope of this work.

3.3 Pretraining and Transfer Learning

Unsupervised learning played a key part in the rise of deep learning (debatedly starting with (Hinton *et al.*, 2006)). It made it easier to train deeper neural networks with a process called *pretraining*. Pretraining is based on the assumption that representations learned from one task can sometimes be useful for accomplishing other tasks in the same input domain. For example, we can use a DAE to learn the structure of the data from unlabelled data and then use what it has learned as features or initialisation for a supervised learning task with the same type of inputs. We can also do supervised pretraining by first training the network to estimate one target variable and then use those weights or features when training to estimate a different target variable, for example.

Unsupervised pretraining for supervised learning is very common in NLP (Devlin *et al.*, 2018, Howard and Ruder (2018)) and supervised pretraining for supserivsed learning is very common in computer vision (Yosinski *et al.*, 2015, He *et al.* (2015*a*)). It is not yet clear theoretically why pretraining works. It may be that using the pretrained weights as initialisation to the supervised model provides a better starting position on the loss surface and thus have regulartory effects (Goodfellow *et al.*, 2016, Ch. 14). Pretraining is most effective when there is little data available for the supervised task but a lot of data available for the pretraining task.

The process of transferring what is learned doing one task (for example in pretraining) to another task (supervised learning for examples) is called *transfer learning*. When done effectively, pretraining and transfer learning can together dramatically reduce the number of training samples and computational power need to train a model.

(Zeiler and Fergus, 2014) provides an insigtful investigation of the what a CNN learns when trained on image data. From observing the types of features extracted for a trained image model (Figure 3.2) it becomes clear why they are also effective on other image datasets. The learned filters seemed to look for generic image features like edges and color gradients which are useful for most computer vision tasks. (Howard and Ruder, 2018) describes a good practice when doing transfer learning via gradual unfreezing and discriminative fine-tuning.

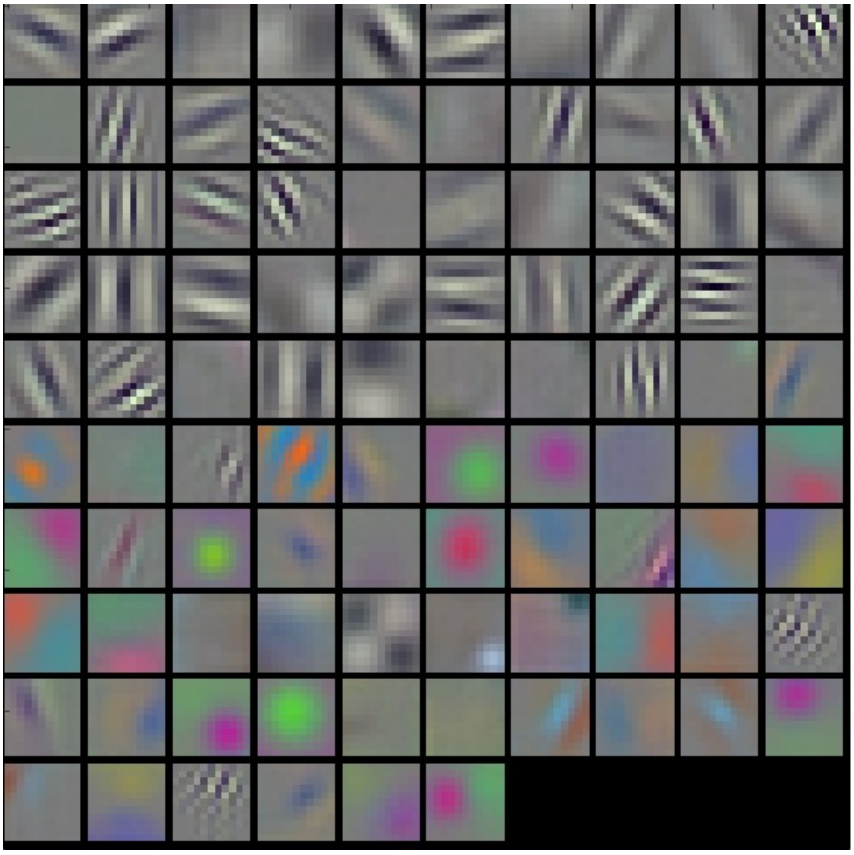


Figure 3.2: Visualising the first layer convolutional filters leared by a neural network in a large image dataset.

3.4 More Regularisation

In §2.3.5 we discussed basic regularisation methods for neural networks. There are many other ways of combatting overfitting. The following section discusses two regularisation techniques proven to be extremely powerful in almost every application of deep learning.

3.4.1 Dropout

Dropout consists of randomly setting the output of a hidden neuron to zero with probability p. See Figure 3.3 for an illustration of how dropout effects the connections between neurons. The neurons which are set to zero 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 technique ensures that a neuron does not rely on the signals of a particular set of other neurons, It is therefore forced to learn more robust features that are useful in conjunction with many different random subsets of the other units (Krizhevsky *et al.*, 2012).

In the original paper they used $p = 0.5$ but the optimal selection will depend on the use case and is typically found by experimentation. At test time, no units are dropped out and their output is multiplied by $1 - p$ to compensate for the fact that all of the units are now active. Dropout does tremendously well to combat overfitting, but it slows down the covergence time of training.

There are also parallels to be drawn between dropout and ensembling approaches (Hinton *et al.*, 2012). Since, at each training iteration a unique set of neurons are active which may then each be viewed as unique models. During training these models are combined - similar to the process of ensembling models.

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 to capture all interactions and relationships. This is not possible in the real world. Such a dataset is not available and would be infinitely large. And even if it were, machine learning will then be unecessary since all the

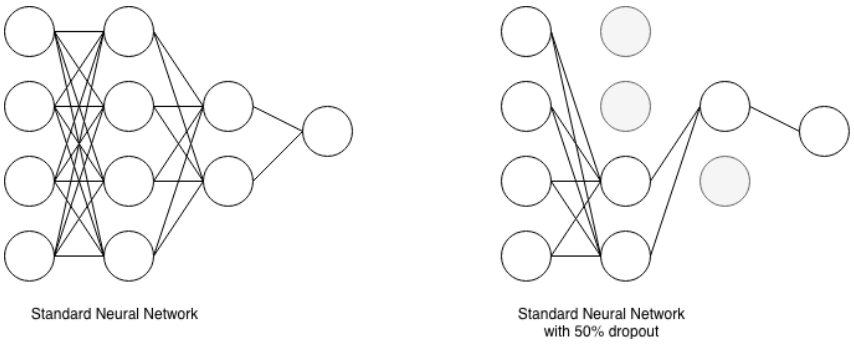


Figure 3.3: The effect dropout has on connections between neurons.

possible observations are observable and one could simply use a lookup or a search when predicting 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 models generalisablity. If the collection of new samples is not available, we can try to artificially create more through *data augmentation*.

This is a standard approach especially in computer vision applications. For example, from a single image, we can rotate it, flip it horizontally, shift it in any direction, crop it, and many other transformations without destroying the semantic content of the image. But by doing so we are artificially increasing the size of the training set to help with overfitting. In Figure 3.4 we see how a single image of a cat is turned into eight images by doing random transformations of the original image. In all of these images it is still possible to tell that it is an image of a cat. Of course this is not as effective as genuine new data samples, but it is a very effective and efficient substitute (Perez and Wang, 2017). Data augmentation consistently leads to improved generalisation.

Figure 3.4: Example data augmentations for images.

Data augmentation can be formalised by the *Vicinal Risk Minimisation*

principle (Chapelle *et al.*, 2001) where 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 to enlarge it. In image classification one can define the vicinity of an image as the set of its horizontal reflections and minor rotations, for example.

Note that data augmentation is dataset and domain dependent and that the augmentations should preserve the semantic content or the signal in the observation. For example, taking too small crops of an image will ignore the context and may make it impossible to tell what is inside. And augmentations that are suitable for image data does not necessarily make sense for text data.

3.5 Modern Architectures

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

3.5.1 Normalisation

One of the things that complicate the training of neural networks is the fact that hidden layers have to adapt to the continuously changing distribution of its inputs. The inputs to each layer are affected by the paramaters of all its preceding layers and a small change in a preceding layer can lead to a much bigger difference in output as the network becomes deeper. When the input distribution to a learning system changes, it is said to experience covariate shift (Shimodaira, 2000).

Using ReLUs, carefull weight initialisation and small learning rates can help a network to deal with the internal covariate shift. However, a more effective way would be to ensure that the distribution of non-linearity inputs remains more stable while training the network. (Ioffe and Szegedy, 2015) proposed *batch normalisation* to do just that.

A batch normalisation layer normalises its inputs to a fixed mean and variance (similar to how the inputs of the network is normalised) and therefore it can be applied before any hidden layer in a network to prevent internal covariate shift. The addition of this layer dramatically accelerates the training

of deep neural networks, also because it can be used with higher learning rates. It also helps with regularisation (Ioffe and Szegedy, 2015), therefore in some cases dropout is not necessary.

The batch normalising transform over a batch of univariate inputs, $x_1, \ldots, x_n$ is done by the following steps:

1. Calculate the mini-batch mean, μ, and variance, σ^2:

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

2. Normalise the inputs,

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

   where ϵ is a constant to ensure numerical stability.

3. Scale and shift the values,

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

   where γ and β are the only two learnable paramaters of a batch normalisation layer.

The reason for the scale and shift step is to allow the layer to represent the identity transform if the normalised inputs are not suitable for the following layer, *i.e.* the scale and shift step will reverse the normalisation step if $\gamma = \sqrt{\sigma^2 + \epsilon}$ and $\beta = \mu$. Batch normalisation has now become a standard when training deep CNNs.

3.5.2 Skip-connections

Residual Networks became very popular after it was used to win one of the ImageNet competiions (He *et al.*, 2015*a*). The residual connection layer can simply be formalised as

$$y = F(x) + x,$$

i.e. combining the input to the layer(s) with the output of the layer(s). Here, the combination is by addition, but other ways can also be used, like multiplication or concatenation. The idea is also illustrated in Figure 3.5.

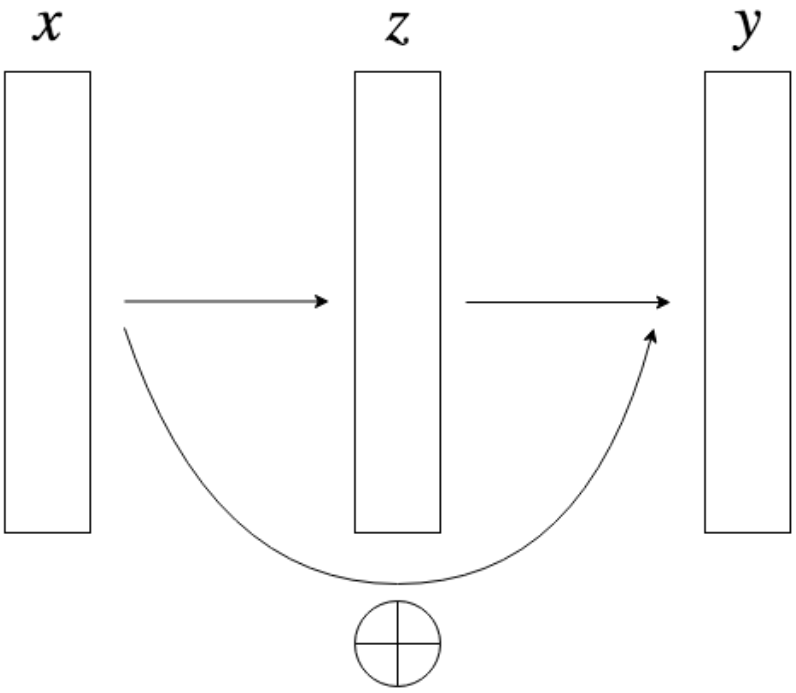


Figure 3.5: Diagram conceptualising a skip-connection.

The DenseNet (Huang *et al.*, 2016) is another well-known network making use of skip connections. In a DenseNet each layer is connected to every other layer in the network. There are multiple benefits to using skip-connections: they alleviate the vanishing-gradient problem, strengthen feature propagation, encourage feature reuse, and may reduce the number of parameters required (Huang *et al.*, 2016). These benefits show in the empirical evaluations. It is interesting to note that one can draw a parallel between ResNets and boosting methods since both are approaches to fitting a model to the residual (Huang *et al.*, 2017).

3.5.3 Embeddings

An embedding is a layer that maps a discrete input to a numeric vector representation. It was first used in NLP in order to represent words as numbers so that it can be processed by a numeric model. For instance the word “woman”

may be represented by the vector $[1, 3, 5]$ and the word "man" by $[2, 4, 6]$. The goal is to map discrete inputs to a meaningful vector space where items with similar meaning exist close to each other. In contrast to using a one-hot encoded representation of words, where all words equally far apart. See Figure 3.6 for an illustration of a such space[1].

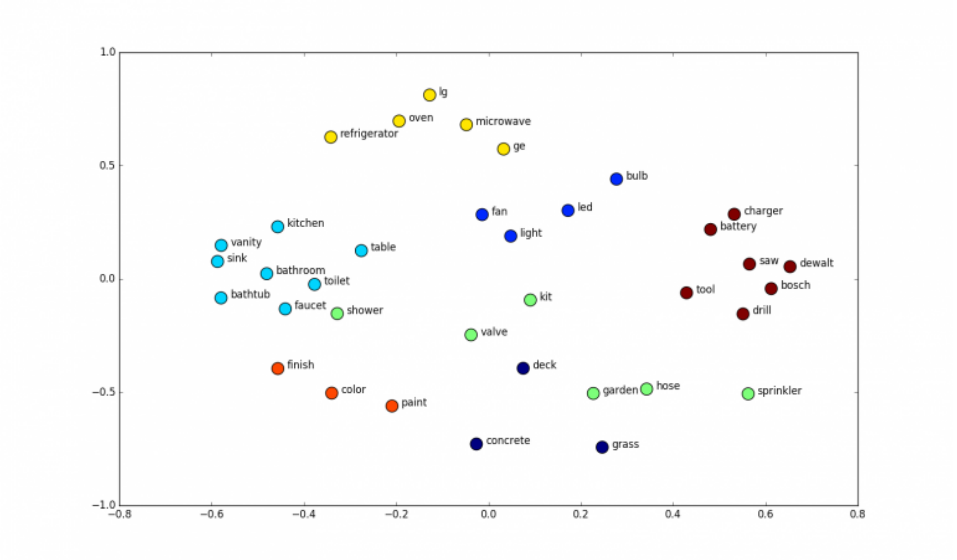


Figure 3.6: Learned word embeddings in a 2-dimensional space.

Initially the mappings were configured independenty of the neural network with approaches based on co-occurences (Mikolov *et al.*, 2013). The real breakthrough came when the mappings were defined as learnable layers in the network (as done in all of the latest NLP research (Howard and Ruder, 2018, Devlin *et al.* (2018))). Thus they can be tuned just like any other parameter in the network. The parameters of the embedding function (or layer) are first randomly initialised and then gets tuned along with the rest of the neural network during training.

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 $W : k \times p$ is the matrix containing the embedding where p is the number discrete categories and k is the embedding size. The embedding layer can be reused by all input features with the same input type, making it more efficient and reducing the memory footprint of the model.

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

3.5.4 Attention

Attention is one of the standout breakthroughs made in deep learning in recent times; especially playing an integral part to NLP's success and other sequence related tasks (Vaswani *et al.*, 2017, Devlin *et al.* (2018)). It was first popularised in neural machine translation (Bahdanau *et al.*, 2014) and now almost used ubiquitously in natural language processing applications. It was also found useful in computer vision applications, like image captioning (Xu *et al.*, 2015), and in audio processing (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 RNN to sequentially output words describing the image. With the use of an attention model, the network is restricted to only look at certain parts of an image at every step, avoiding having to look at the full image everytime. This is illustrated in Figure 3.7. Notice how the network focusses on the bird part of the image when predicting the word "bird" and "flying" and then focussing on the water part of the image when predicting words like "body" and "water". Similarly, when used in machine translation, we can visualise the attention weights to see how the network focusses on a different subset of words for predicting each word in the target language (Figure 3.8).

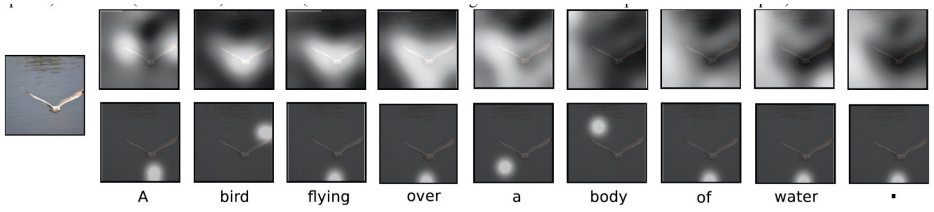


Figure 3.7: Attention applied to image captioning.

In an ideal world, an attention module should not be necessary and we would want the network to learn if certain parts of images are not required to look at. However, currently this built-in prior proves to be extremely helpful with current learning algorithms.

The core of an attention module can be summarised with these equations:

$$\boldsymbol{z} = f(\boldsymbol{x})$$
$$\boldsymbol{\alpha} = \text{softmax}(f_{att}(\boldsymbol{x}))$$
$$\boldsymbol{y} = \boldsymbol{z} \otimes \boldsymbol{\alpha}$$

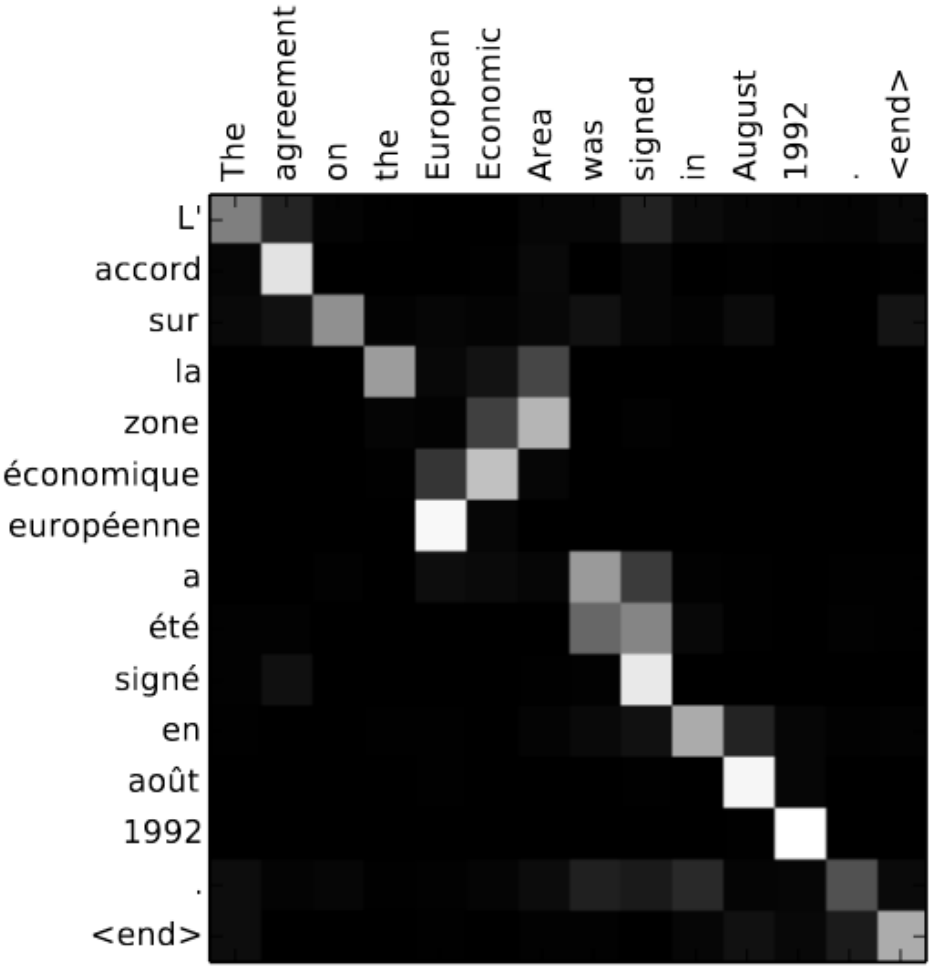


Figure 3.8: Attention applied to machine translation.

$\boldsymbol{z}$ it the ordinary activations produced by a layer f given input $\boldsymbol{x}$. $\boldsymbol{\alpha}$ are the weights produced by the attention layer, f_{att} after a logit transform so that the weights sum to one. The output of the attention module, $\boldsymbol{y}$, is then an elementwise multiplication, $\otimes$, of $\boldsymbol{z}$ and $\boldsymbol{\alpha}$, however other combinatorial operations can also be used. More detail can be found in (Xu *et al.*, 2015). We will 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

The process of setting the hyperparameters for learning algorithms are hard. It requires expertise, extensive trial and error and is more of an art form than a

science. There are many hyperparameters to consider: learning rate, batch size, momentum, and weight decay, to name a few. Although the right parameters make a huge difference in training time and performance, there are no simple and easy ways to find them. Doing a grid search or a random search in the parameter space is computationally very expensive.

In this section we summarise the work done in (Smith, 2018) to give guidance on finding and setting the learning rate, batch size and weight decay. The approach suggested is based on finding the right balance between overfitting and underfitting by using the right hyperparameters. By looking at the validation loss at early stages of training, one will find clues to which parameters are optimal. This eliminates the necessity of running complete grid or random searches.

By following the hyperparameter selection approach discussed next, one can achieve tremendous improvements in training time and accuracy. If done correctly one may exploit a property called *super-convergence* (Smith and Topin, 2017), where neural networks can be trained an order of magnitude faster than with standard training methods. Figure 3.9 plots the test loss over each training iteration of the proposed hyperparameter strategy compared to a fixed learning rate policy. We see that with the proposed method, the model achieves a better accuracy than the standard approach and in eigthth of the training iterations. Next we discuss how to select hyperparameters to achieve super-convergence.

Firstly note that the effects of the learning rate, batch size and other regularisation techinques, like weight decay, are tightly coupled and should be tuned together. We start with setting the learning rate. In Chapter 2 we showed that the learning rate is an important parameter and that one can either keep it fixed during training or decrease it by some factor when it looks necessary. (Smith, 2015) suggests using *cyclical learning rates* (CLR). A CLR cycle consists of two stages, one where the learning rate linearly increases during training up until a certain maximum and one where the learning rate decreases linearly back to a chosen minimum. One needs to specify the number of epochs in each stage, also known as the step size and the minimum and maximum learning rate. One reason why CLR is believed to work is that variable learning rate allows the model to better dodge saddle points on the loss surface.

In his most reason work, (Smith, 2018), the author suggest using only a single cycle of CLR, called the *1cycle* policy. The learning rate setting of the

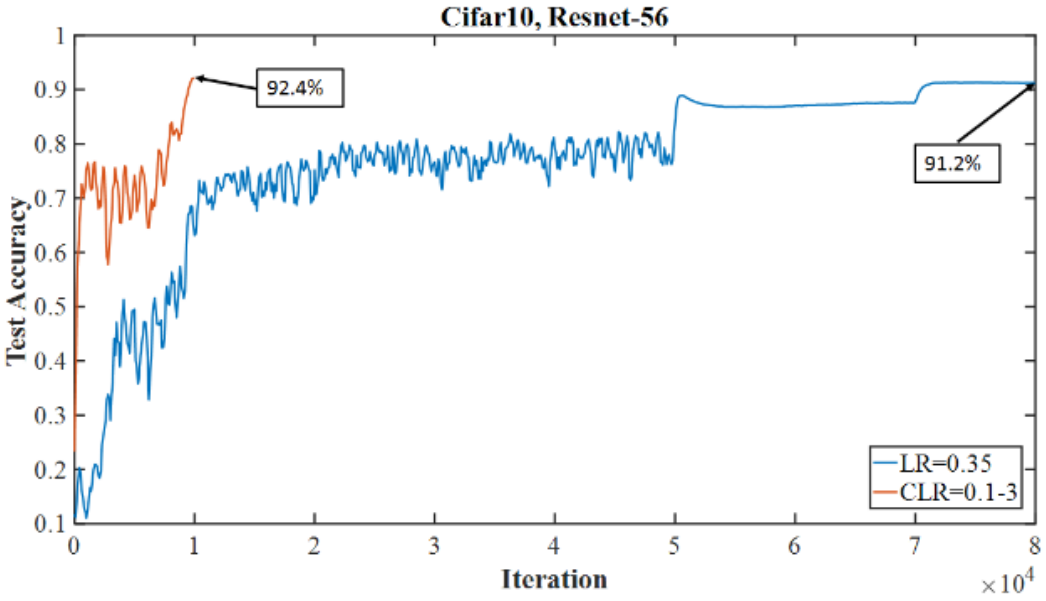


Figure 3.9: Reduced training iterations and improved performance with super-convergence principle.

1cycle policy is illustrated in Figure 3.10.

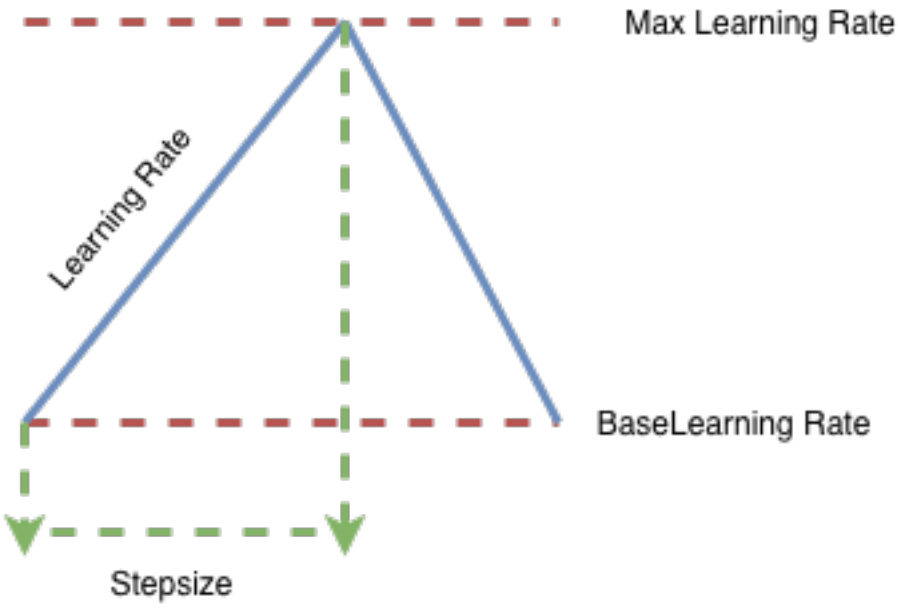


Figure 3.10: The learning rate schedule of the 1Cycle policy.

Clearly one also needs to set the bounds of the learning rate, however luckily, the author designed a learning rate range test to help one do just that. In the learning rate range test, training starts with a small learning rate which is

slowly increased linearly throughout a pre-training run. Typically the training loss will slowly start to decrease as the learning rate increases, up until a certain point at which the learning rate becomes to large and the loss starts to increase. The learning rate at this point is the largest learning rate one can use during training. The lower bound of the learning rate can be set to a factor of 10 times less than the max. See Figure 3.11 for example output of the range test and how to deterimin the learning rate bounds for the 1Cycle policy. One should be careful of too small a stepsize since that increases the rate of the learning rate increase which might make the training process unstable.

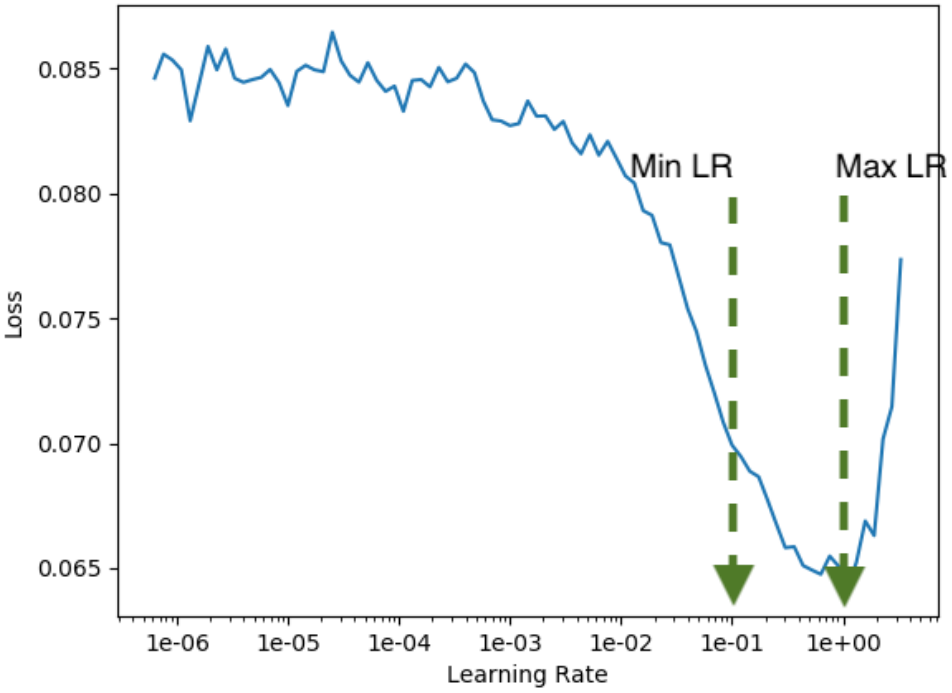


Figure 3.11: An example output of a learning rate range test.

Note that large learning rates also acts as a form regularisation (Smith, 2015). Thus if one uses large learning rate, one might want to reduce some of the other regularisation controls. Along with the learning rate (Smith, 2018) suggests and shows that larger batch sizes allow for training with larger learning rates and thus convergence can be reached quicker. So when setting the batch size one would choose the largest possible that fits into memory.

The weight decay is another important value to be set properly along with the 1Cycle policy. Since it is a regularisation parameter it is important that it is in balance with the learning rate. The proposed way of setting the weight decay is through a grid search. Once can do a few short training runs at differnt weight decays and see if the validation loss gives any indication of which one is better. Another option is to test different weight decays during the learning rate range test and see how they behave. Then we will look for the weigth decay that produces the most stable loss and allows the use of the largest learning rate. In our experiments we follow these suggestions to find the best hyperparamters for each model and dataset.

3.7 Model Interpretation

Although Deep Learning is now the state-of-the-art for many machine learning tasks, it is still trailing behind other algorithms in terms of model interpretability. But keep in mind this is not an unusual trade-off; between prediction performance and model interpretability. 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 deep learning's greatest criticisms and is a large reason why it cannot be used in some 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 to gain insights from neural networks. We discuss the topic briefly in the following sections.

3.7.1 Neural Network Specific

We have showed in §2.4 that it is possible to inspect activations and weights of layers at different levels of the network. If the network is small, one might gain insight to what the network has learned or why it is making certain decisions. However, most useful neural networks are at least three layers deep, making its activations and weights more complex to interpret.

When fully convolutional networks are used, there are ways to visualise which parts of the inputs were important in making a certain decision. These visualisations are called class activation maps (Zhou *et al.*, 2016) ((Selvaraju

et al., 2017) suggests a similar approach). But they cannot be used with fully connected layers and only with fully convolutional networks.

Another common interpretation tool, in order to gain insight into what specific neurons are looking for, is to rank the inputs by the magnitude of their activation at that neuron. Then, if one would be able to manually spot similarities between the highest ranked inputs, one would have a potential description of the pattern that triggers that neuron. This approach is called *Activation Maximisation* (Erhan *et al.*, 2009). This process is manual and not guarranteed to produce useful insights, especially when there are many neurons to investigate.

An interesting take on model interpretation is shown 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 is the process of transferring the knowledge from one model (or an ensemble of models) to another model by training the target model to estimate the predictions of the source model (or ensemble). (Frosst and Hinton, 2017) and (Che *et al.*, 2016) used a soft-decision tree and boosted trees respectively to learn the mapping between the input and the neural network predictions. Once the tree based methods are trained, the usual intepretation tools of tree based models, like feature importance of evaluating how a sample traverses the tree, can be used to attempt to understand what the neural network is doing. It is interesting that in both publications the authors note that the tree based models trained on the neural network predictions achieved a better performance that the ones trained on the actual targets - indicating the value knowledge distillation can add.

3.7.2 Model Agnostic

Besides interpretation tools specifically designed for neural networks, one can also make use of model agnostic tools, *i.e.* ones that can be used with any machine learning model. *The Permutation importance algorithm* computes feature importance scores for each input feature. The importance measure of a feature is deterimend by the model's sensitivity to a random permutation of the values of that feature. The expectation is that when a feature with a strong signal is shuffled before passed to the 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 have a major impact on the model's performance.

This technique was first introduced by (Breiman, 2001) for the Random Forest algorithm but can also be generalised to other models. The steps for calculating the feature importances by the permuation importance algorithm are as follows:

1. Train a neural network on a dataset with p input features.
2. Evaluate the network on a validation set to obtain a performance metric m_0.
3. For each of the p input features, do:
    - 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 to obtain m_j, $j = 1, \ldots, p$.
4. Rank the features based on $m_j - m_0$ (if a bigger m is better).

Permutation importance only produces sensical results on the assumption that the features are independent. Permuting the feature 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.* 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 is not dependent on it.

Partial Dependence Plots can be used to visaulise the relationship between input features and the target. This is another post-training interpretation tool and was first used in (Friedman, 2001). Once a neural network is trained, we can evaluate what effect a change in any input feature has on a single prediction, by observing the change in the prediction. Say for example we want to see how feature X_1 influences the model prediction. By taking a single observation from the data we can evaluate how the model prediction changes by changing the value of X_1 to other possible values of X_1. And since this behaviour will most like vary for different observations, this 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 then be calculated along with standard errors of these effects.

Other model agnostic interpretation tools, like SHAP values (Lundberg and Lee, 2017), are also available. Here we only discussed the basic approaches. In the next two chapters we will show examples of how these model interpretation techniques can be implemented.

Chapter 4

Deep Learning for Tabular Data

4.1 Introduction

Up to now we have covered the basics of neural networks, as well as the more recent advancements in the era of deep learning. The aim of this chapter is to explore how the approaches and techniques we discussed can be leveraged when doing deep learning with tabular data. Tabular data is vastly different to so-called unstructured data like images, text and speech and therefore we do not expect all of the methods to be as effective. Deep learning on tabular data is not as well researched compared to the aformentioned data domains and thus it is not always clear how to solve certain modelling challenges. There is only a handful of publications reporting sucessful implementations of deep learning for tabular data on important applications such as 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), to name a few.

The tabular data domain is still dominated by tree-based models like random forests and gradient boosted trees. This makes us curious as to why deep learning is not as effective here as it is in most other data domains. This chapter will help shed light on this issue and point to promising ways of improving the current state-of-the-art.

This chapter is structured based on the tabular dataset challenges introduced

[1]To predict the probability of a user clicking an item, critical to online applications.

in §1.3. For each of the challenges we provide a recap of the issue, review the literature to discuss how they are currently being treated, and provide suggestions for improving these approaches, (where applicable). In §4.2 we look at ways of representing the input features of a tabular dataset. §4.3 is abut approaches for leveraging feature interactions. A large part of this chapter is on how to be more sample efficient; this we discuss in detail in §4.4. Then we briefly investigate ways of interpreting deep neural networks for tabular data in §4.5. In the final section we discuss the 1Cycle policy and hyperparameter selection for tabular data neural networks, in addition to other miscellaneous topics that do not fit into the above categorisation.

4.2 Input Representation

One of the major design considerations when building a deep neural network for tabular data is the input representation, *i.e.* how should one represent each feature numerically? This choice may heavily influnce the model's ability to extract patterns from the input and the effectiveness of the optimisation algorithm. What makes this decision harder is that the features in a tabular dataset are highly heterogeneous (Shavitt and Segal, 2018). What we decide for one features might not be optimal for another and we want to ensure that no feature dominates another in the training process.

A tabular dataset typically has continuous features and categorical features. Different approaches are needed to process each. But the processing should be done in such a way that the one feature type does not dominate the other in terms of feature extraction and optimisation.

A tabular dataset can be high-dimensional and very sparse which makes the task more difficult but even more important to do effectively, as noted by many (Song *et al.*, 2018, Wang *et al.* (2017), Qu *et al.* (2016), Cheng *et al.* (2016*a*), Anonymous (2019), Covington *et al.* (2016)). An example of one such challenging tabular dataset is the Criteo dataset[2]. Its feature dimension is ~30 million, exhibiting sparsity of ~99%. Although not always as extreme, these properties are common in tabular datasets.

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

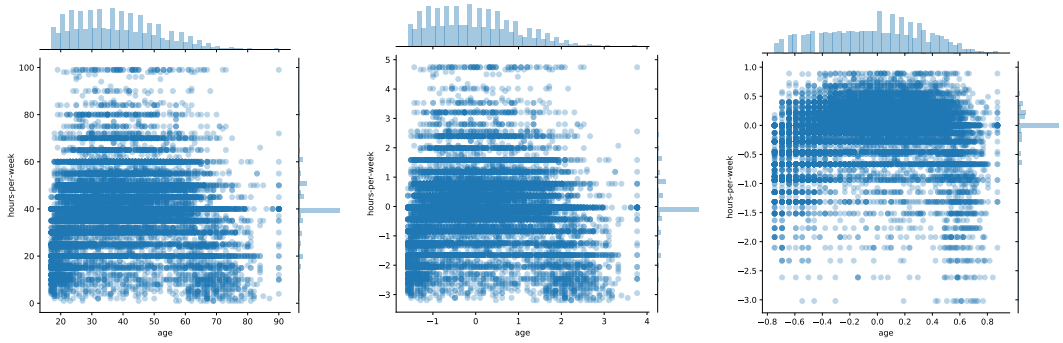
4.2.1 Numerical Features

One of the things that make tree-based methods so attractive is that the scale and the distribution of the features hardly matter, as long as their relative ordering is 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 the features are on different scales, one might dominate the weight updates, and if a feature constains a large value, it may throw of the optimisation procedure and cause exploding or vanishing gradients (Clevert *et al.*, 2015). This means that one should do a proper standardisation of all continuous features in a tabular dataset.

The typical standardisation approach for numerical features in deep learning is to do mean centering and variance scaling, *i.e.* $\tilde{x} = (x - \mu)/\sigma$, where $\mu$and σ is the mean and standard deviation of X respectively ($x \in X$). One would expect this transformation to be sufficient, but in practice, many have found otherwise.

In (Haldar *et al.*, 2018) they suggest to first inspect the distribution of each of the features. If a feature looks gaussian, do the standard normalisation, $(x-\mu)/\sigma$, but if the feature looks more like a power law distribution, transform it by $\log\left((1+x)/(1+\text{median})\right)$. This ensures that the bulk of the values lie between {-1,1} and that the median is close to zero. As an example, we illustrate what effect these transformations have on 2 continuous features in the Adult dataset, Age and Hours-per-week, in Figure 4.1. In Figure 4.1 (a) we see that the features are on a roughly similar scale but their distributions are totally different. By applying mean centering and unit variance scaling (b) we see the values of the Hours-per-week feature are mostly in {-1,1} but for the Age features there are still plenty of values outside that range. When doing the power distribution transformation of (Haldar *et al.*, 2018), we observe that the Age feature is now in the proposed value range. The downside to this approach is that it is not automatic and can become cumbersome with many variables.

To reduce the possibly high variance of numeric features (Song *et al.*, 2018) suggests to transform the a numeric feature to $\log^2(x)$ if $x > 2$. This worked in their use-case but it is hard to imagine that this work on all other cases. This transformation causes a discontinuity at $x = 2$ and a possible overlap between values the were originally less than 2 with those that were greater than 2. In addition, this transformation does not address the extreme values on the negative side. (Wang *et al.*, 2017) simply used a standard log transform,



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

Figure 4.1: Effect of normalisation on continuous variables.

$\log x$, to normalise continuous features.

In (Covington *et al.*, 2016) they also found that the proper normalisation of numeric features were critical for the model to converge. Their approach was to transform the numeric features to be equally distributed in [0,1) using the cumulative distribution $\tilde{x} = \int_{\infty}^{x} df$, where f is the distribution of x. They approximated the integral with linear interpolation on the quantiles of the feature values computed as a preprocessing step.

Another possibility for numeric feature normalisation that we have not seen used in the deep learning for tabular data literature, is to use a batch normalisation layer as the initial layer to numeric features. This will apply the same type of scaling as the zero mean and unit variance transformation, but with transformation parameters learned from each batch. Thus the need to preprocess numeric features is removed. The caveat is that the quality of the transformation depends on the batch statistics and thus if the batch is not representative of the full data distribution (which is likely if the batch size is small), then it might derail the training procedure.

The fact is that the only way to know for sure which normalisation to apply to numeric features is by experimations, since it varies between datasets. The standard normal scaling is a safe transformation to use if experimentation wants to be avoided.

4.2.2 Categorical Features

Most of the sparsity in tabular datasets come from 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 one-hot encode categorical features. That is, if we have a categorical feature with three possible categories, for example, the one-hot encoded form of the three categories will be: Category 1: $[1, 0, 0]$, Category 2: $[0, 1, 0]$ and $[0, 0, 1]$ for Category 3.

There are multiple inefficiencies when using one-hot encodings with neural networks. The obvious one we have already mentioned is that it introduces sparsity to the data, since the dimension of the one-hot encoded form is equal to the number or categories in a feature. Thus if we have plenty of high-cardinality features in our dataset, the data will be extremely sparse and difficult to model. This also increases the size needed of the first linear layer and thus we need machines with greater memory and processing power. Neural networks can easily overfit such sparse data (Covington *et al.*, 2016).

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 are equally far apart now matter how semantically similar or dissimilar they are. This makes it harder for the model to learn useful patterns.

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

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

The reason why it is used all-round is because it does not have the same issues as the one-hot encoded representations. A similar formulation to §3.5.3,

we define the embedding for the j-th categorical feature by:

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

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

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

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

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

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

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

Entity embedding not only reduces memory usage and speeds up neural networks compared to one-hot encoding, but more importantly, by mapping similar values close to each other in the embedding space it reveals the intrinsic

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

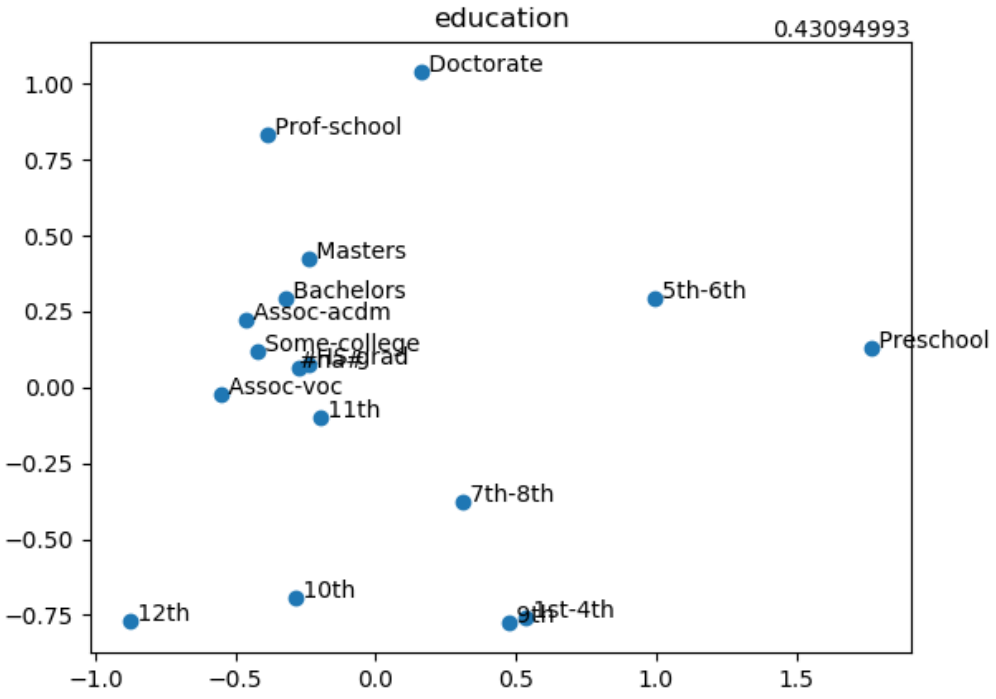


Figure 4.2: PCA of the Education entity embedding weight matrix.

To prove that these entity embeddings actually learns something useful, besides plotting the embedding matrix, one can also feed them along with the continuous features to other learning algorithms and see how it affects perform-

ance. (Guo and Berkhahn, 2016) found that the embeddings obtained from the trained neural network boosted the performance of all tested machine learning methods considerably when used as the input features. These embeddings can be reused on different machine learning tasks and do not have to be re-learned for each dataset. Instacart and Pinterest, referenced above, reported sucessful implementation of this approach.

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

4.2.3 Combining Features

Once the continuous and categorical features have been processed and embedded, we need a way to combine them before passing it to the rest of the network.

Standard: The embedding for each categorical variable gets concatenated to the continuous variables and then gets passed to the rest of the layers in the network.

Dealing with mixed input types. (Song *et al.*, 2018), (Wang *et al.*, 2017), (Qu *et al.*, 2016), (Cheng *et al.*, 2016*a*) Processing numerical and categorical features and their combinations. (Song *et al.*, 2018) embeds both the numerical and categorical features separately into a lower-dimensional representation. By mapping both types in the same feature space faciliates more effective learning of interactions between the mixed features.

The embedding for the j-th numerical features is obtained by:

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

where x_j is a scalar and $\boldsymbol{v}_j$ is the associated weight vector. Why does it help to increase the dimension of the numerical features? Is it only for balanced representation when combined with categorical features? The numerical and categorical embeddings are concatenated to form a single vector representation (Song *et al.*, 2018). (Wang *et al.*, 2017) does not embed numerical but just stack the normalised numerical features along with the categorical embeddings.

4.3 Learning Feature Interactions

Traditionally, input features to a machine learning algorithm must be handcrafted from raw data, relying on practitioner expertise and domain knowledge to determine explicit patterns of prior interest. The engineering process of creating, analyzing, selecting, and evaluating appropriate features can be laborious and time consuming, and is often thought of as a "black art" requiring creativity, trial-and-error, and oftentimes luck.
In contrast, deep learning techniques learn optimal features directly from the data itself, without any human guidance, allowing for the automatic discovery of latent data relationships that might otherwise be unknown or hidden.

That being said, preprocessing of data in deep learning is not totally free of human engineering. See in the Normalisation section what measures should be taken to ensure a NN can learn effectively from tabular data.

Feature engineering in general is hard and time consuming with no clear recipe to follow. But it is also very crucial to an effective learning system. The main aim is to find a low-dimensional representation of sparse and high-dimensional raw features and their meaningful combinations. Some of the challenges of feature engineering is listed below.

It is widely held that 80% of the effort in an analytic model is preprocessing, merging, customizing, and cleaning datasets, not analysing them for insights (Rajkomar *et al.*, 2018).

The success of predictive algorithms largely depends on feature selection and data representation. The feature selection process and finding the best data representation is largely a manual and painful process.

In most machine learning tasks the greatest performance gains can be achieved by feature engineering wheras better algorithms only result in incremental boosts. In feature egineering one strives to create new features from the original features based on some domain knowledge of the data or otherwise, that makes it easier for the model to estimate the target. Although a crucial step to make the most out of the data, this can be a very laborious process. There is no formal path to follow in this stage and thus usually consists of many a trial and error, benefitted by domain knowledge of the data, only accessible in some cases. A huge advantage of using NNs on tabular data (and other data structures) is that the feature engineering process gets automated to some extent. A NN learns these optimal feature transformations implicitly during

the training process. The hidden layers of a NN can be viewed as a feautre extractor that was optimised to map the inputs into the best possible features space for a model (the final layer of the network) to operate in.

Extracting high-order combinations of features. (Song *et al.*, 2018), (Wang *et al.*, 2017), (Qu *et al.*, 2016), (Guo *et al.*, 2017) They key question here is to determine which feautres to combine and how to form meaningful high-order features. Effective prediction usually relies on modelling high-order interactions between features. Majority of the time needs domain experts to help massage the data. Can follow a brute force approach but enumerating all the possible high-order features will exponentially increase the model search space which will just further increase the risk of overfitting. Can use multiple fully connected layers with non-linear activations of a NN. Fully-connected layers model all feature interactions impplicitly, but is not good enough to learn all types of interactions.

Standard approach (Covington *et al.*, 2016) MLP

Fully connected model structure leads to very complex optimization hyperplanes with a high risk of falling into local optimums.

4.3.1 Attention

These layers are inefficient in learning multiplicative feature interactions. Hard to explain which features and combinations were important. (Song *et al.*, 2018) uses a multi-head self-attention mechanism which they call the interacting layer. (Zhou *et al.*, 2017) also uses some form of attention but without the softmax layer to reserve intensity of activations. The idea comes from (Vaswani *et al.*, 2017) which itself stems from work done in (Bahdanau *et al.*, 2014). Within in the interacting layer each feature is allowed to interact with every other feature and automatically determine which of those interactions are relevant to the output. They also combine a residual connection between layers so that different orders of feature interactions can be combined. To explain the attention mechanism, consider feature j and the step to determine which high-order features involving feature j are meaningful. We first define the correlation between features j and k under attenion head h as:

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

where $\phi^{(h)}(.,.)$ is an attention function which defines the similarity between two features. It can be defined by a neural network or a simple inner product like in (Song *et al.*, 2018):

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

where $W^{(h)}_{\text{query}}$ and $W^{(h)}_{\text{key}}$ are transformation matrices which map the original embedding space into a new space. The representation of feature j in subspace h is then updated by combining all relevant features guided by coefficients $\alpha^{(h)}_{j,k}$:

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

$\tilde{\boldsymbol{e}}^{(h)}_j$ is a combination of feature j and its relevant features under attention head h. Therefore it is a learned combinatorial feature. Since a feature can be involved in various different combinations, we use multiple heads to extract combinations, *i.e.* $\{\tilde{\boldsymbol{e}}^{(h)}_j\}^{H}_{h=1}$. (Song *et al.*, 2018) used H=2. All of these combinatorial features are concatenated into a single vector, $\tilde{\boldsymbol{e}}_j$. Then finally the output is combined with its raw input (residual connection) and sent through a ReLU:

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

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

(Wang *et al.*, 2017) uses the cross-network which is an automated way of building cross-features. Each layer produces higher-order interactions based on existing ones, and keeps the interactions from previous layers. The cross-network is trained jointly with a DNN.

(Wang *et al.*, 2017) makes a case for finding a bounded-degree feature interactions, saying that all the Kaggle competions are won with feature engineering of low-degree interactions, whereas DNNs learn highly non-linear interactions implicitly. (Wang *et al.*, 2017) cross-network consists of cross-layers that can be formalised as:

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

where $\boldsymbol{x}_l$ is the output of the l-th cross layer; $\boldsymbol{x}_0$ is the input vector; $\boldsymbol{w}_l$ and $\boldsymbol{b}_l$ are its associate weight and bias paramters respectively. Each cross layer adds

back its input after feature crossing in a residual connection fashion. (Wang *et al.*, 2017) experimented with 1-6 cross layers. The degree of cross features grows with cross-network depth. The DNN trained in parallel is just a simple network with fully-connected layers and ReLUs. The output of the two streams are concatenated, send through a fully connected layer and a sigmoid layer. (Qu *et al.*, 2016) used something called a product layer, which takes pairwise inner or outer products of all feature combinations and concatenates it to all linear combinations. The output is then fed to 2 fully-connected layers. According to (Guo *et al.*, 2017) it is necessary to capture both low and high-order interactions (and wide&deep paper). They also have to parallel streams of networks, one the FM capturing the low order interactions and one the DNN captuting the high-order interactions. (Cheng *et al.*, 2016*a*) believes it is both important to learn to memorise and generalise. Where memorise refers to recalling from known observations and generalise to predict accurately on unseen samples. They attempt to achieve this again with two streams, one linear layer (wide) and one deep network (deep). The wide stream learns to memorise and the deep network learns to generalise. Combined by a weighted sum. (Anonymous, 2019) Fully connected model structure leads to very complex optimization hyper-planes with a high risk of falling into local optimums. Therefore it is necessary to explicitly leverage expressive feature combinations. Furthermore it help to limit the model size to make learning more efficient. To achieve this they use automatic feature grouping, feature group reduction and recursive endocder with share embeddings. These ideas seems a little ad-hoc and not end-to-end.

4.3.2 Skip-Connection

(Song *et al.*, 2018) shows that residual connection gives better results.

(Wang *et al.*, 2017) also used a residual connection.

4.4 Sample Effeciency

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

We suggest two techniques for overcoming this problem: data augmentaion and unsupervised pre-training. (Zhang *et al.*, 2016) also did pretraining with DAEs. DAEs enforce robustness to partially destroyed inputs. Can also be view from a manifold learning perspective (Vincent *et al.*, 2008). Should also consider VAE and GANS (Anonymous, 2019) uses output from GBDT to train an initial model and then to use it a initialisation of the actual model. They call it the transfer of structured knowledge. Data augmentation for tabular datasets is rarely studied. Can use corruption like DAEs or swap noise but then creates inputs that does not exist in the real data distribution.

4.4.1 Unsupervised Pretraining

Feature learning problem is also relevant here like in §4.3.

(Miotto *et al.*, 2016) used a stacked denoising autoencoder to learn patient representations from EHR data. They found that these representations were useful features in predicting future health states of patients. By using these learned representations as input significantly improved the performance of predictive models compared to those only using the raw inputs.

(Miotto *et al.*, 2016) presented a novel unsupervised deep feature learning method to derive a general-purpose patient representation for EHR data that facilitates clinical predictive modelling. A stacked denoising autoencoder was used. Unsupervised feature learning attempts to overcome limitations of supervised feature space definition by automatically identifying patterns and dependencies in the data to learn a compact and general representation that make it easier to automatically extract useful information when building classifiers or other predictors (Miotto *et al.*, 2016). These techniques are very familiar and effective in text, audio and image processing, but not with tabular data. (Geras and Sutton, 2014) on gradual increasing of corruption ratio; but applied to images.

4.4.2 Data Augmentation

- Blankout
- Swap Noise
- (Kosar and Scott, 2018) - hybrid bootstrap
- using GANS are interesting by out of scope.

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

When working with images, we can randomly perturb the pixel intensities by a small amount so that it is still possible to make sense of its content. By adding 1 for example to all pixels and all colors in an image, will only make it slightly brighter and we will still be able to make sense of it. Bu with tabular data we can just randomly add a small amount to any feature. The input features will probably not all be on the same scale and the addition of noise might result in a feature value that is out of the true distribution. In addition, it does not make sense to add anything to a discrete variables. Thus in order to inject random noise to a tabular data sample, the noise should be scaled relative to each input feature range and the results should be a valid value for that feature. This also helps the model to be more robust to small variations in the data.

(Van Der Maaten *et al.*, 2013) suggests an augmentation approach that does this called Marginalised Corrupted Features (MCF). The MCF approach adds noise to input from some known distribution.

In the original Denoising Autoencoding papaer (Vincent *et al.*, 2008), they used a blank-out corruption procedure. Which is randomly selecting a subset of the input features and mask their values with a zero. The only conceptual problem with this approach is that for some features a zero value actually carries some meaning, so a suggestion is to blank-out features with a unique value not already belonging to that feature distribution.

Another input corruption approach shown to work empirically here is what is called Swap Noise (Kosar and Scott, 2018). The swap noise procedure corrupts inputs by randomly swapping input values with those of other samples in the datasets. In this way you ensure that the corrupted input at least have valid feature values. But it still might produce combinations of features that are not actually possible.

All of these methods have hyperparameters that needs to be set. I haven't gone into detail as I still need to decide what is relevant to this thesis.

Mixup. The way mixup creates artifical samples is by the following original

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

formulation:

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

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

λ controls the strength of the interpolation between input-output pairs. The closer λ is to 0 or 1, the closer the artificial sample will be to an actual training sample. The authors suggest using $\lambda \sim \text{Beta}(\alpha, \alpha)$ for $\alpha \in (0, \infty)$. They observed best performance when $\alpha \in [0.1, 0.4]$ and if α is too high, they experience underfitting.

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

Typically data augmentation procedures are dataset dependent and therefore requires expert knowledge. It is hard to think of ways to augment tabular data, even more so a generic way of doing so. However, from this defnition it is clear that mixup can be used on any type of data, including tabular datasets.

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

terms of model predicions and gradien norms. This is because mixup leads to decision boundaries that transition linearly between classes, resulting in smoother predictions.

The authors (Zhang *et al.*, 2017) tested mixup data augmentation on tabular datasets. They tested it on 6 classification datasets from the UCI Machine Learning repository. They used a 2-layer MLP with 128 neuros each and a batch size of 16. They found that mixup improved the performance on 4 out of the 6 datasets.

See Figure 4.3.

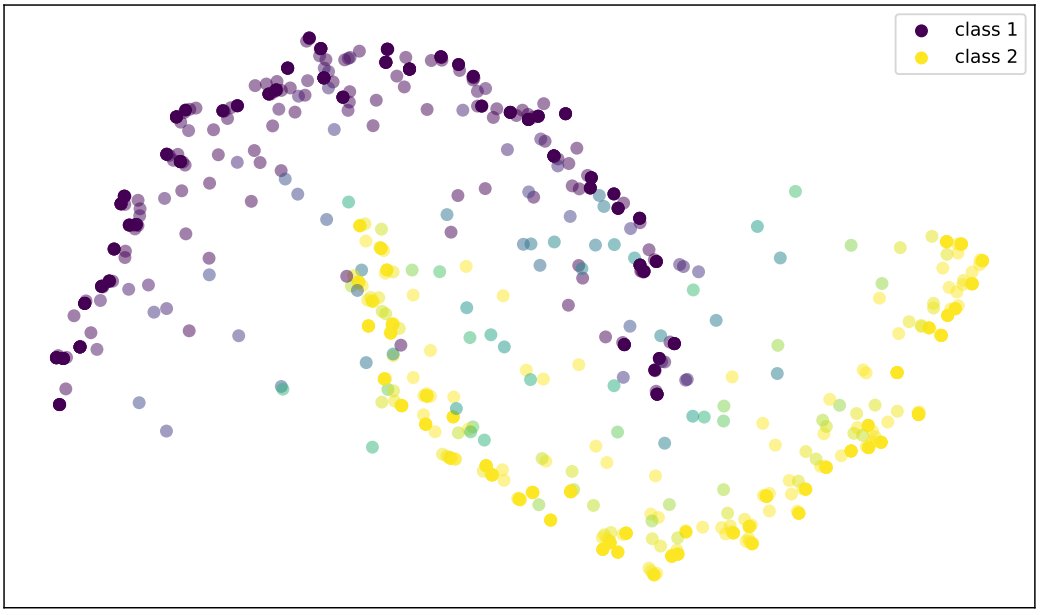


Figure 4.3: Illustration of points created by mixup augmentation.

4.4.3 Regularisation Learning

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

to see that this approach is totally intractable since the only way to train these hyperpararmeters are brute force and repetitive tweaking and validatig (derivative free methods). A workaround is to make these regularisation parameters trainable like all of the other points in the network. This is achieved by minimising the counterfactual loss, a novel loss function proposed by (Shavitt and Segal, 2018). They found that training NNs by optimising the counterfactual loss, outperform other regularisation approaches for NNs and results in NNs that are comparable to gradient boosted trees. The learned regularisation parameters can even help with interpretting feature importance.

4.4.4 Layer Normalisation

DNNs on tabular data also struggle to go deeper than 3 or 4 layers.

Fully connected model structure leads to very complex optimization hyperplanes with a high risk of falling into local optimums.

Batchnorm: (Wang *et al.*, 2017)

The batch normalisation layer attempts to normalise neuron activations to zero mean and unit variance (Ioffe and Szegedy, 2015). It has become the stanard when training deep CNNs. Training with normalisation techniques is perturbed by stochastic gradient descent, stochastic regularisation (like dropout) and the estimation of the normalisation parameters. Both RNNs and CNNs can stabilise learning via weight sharing, therefore they are less prone to perturbations. Fully-connected NNs do not have this luxury and shows high variance in the training error when trained with normalisation techniques.

Fully-connected DNNs with normalisation techniques are very sensitive to perturbations. DNNs exhibit a high variance in training error when trained using BatchNorm. This hinders the learning process. Combined with dropout just magnifies the effect. (Klambauer *et al.*, 2017) suggested the use of SeLUs which is an activation function which helps the network to maintain zero mean and unit variance activations. By using this activation, there is no need for a BatchNorm layer. SNNs do not suffer from exploding or vanishing gradients. They paper tested on 123 tabular datasets to show that on average SNNs are the best. But it is quite finicky to get the implementation right. It requires a very specific weight initialisation, one that does not really make sense for embedding matrices. And even when you get it right, the improvement is not necessarily significant. That said, it has not been tested by others on tabular

data, so it is worth a try.

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

Give more detail if it is proved to be useful. Also needs a specific type of dropout.

(Klambauer *et al.*, 2017) tested SELUs on 121 classification datasets from the UCI Machine Learning repository. They compared DNNs with SELU activations to other DNNs and other classifiers like Random Forests and SVMs. They found that on the datasets with less than 1000 obervations, random forests and SVMs performed the best. However, for the datasets with more than 1000 observations, DNNs with SELU activations performed the best overall. The classifiers were compared by ranking them by their accuracy for each prediction task and doing a pairwise Wilcoxon test.

Another thing the authors found when comparing SELUs with other activations is that the model selection approach for SELU DNNs resulted in much deeper networks than DNNs with other activations.

4.4.5 Dropout

(Zhang *et al.*, 2016) compared dropout with L2 and found dropout to be better.

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

Dropout: 0.5 (Song *et al.*, 2018), tuned (Zhang *et al.*, 2016) (Qu *et al.*, 2016) (Guo *et al.*, 2017) found dependent on dataset and model.

Other regularisation: L_2 penalty (Song *et al.*, 2018), (Wang *et al.*, 2017), (Zhang *et al.*, 2016), (Qu *et al.*, 2016) mini-batch aware l2 for large inputs (Zhou *et al.*, 2017),

4.5 Interpretation

Plot embeddings Plot attention Methods discussed in previous chapter.

Interpreting DNNs. (Song *et al.*, 2018) Model explainability is important for various reasons. Helps to know how to improve your model or where it

goes wrong. Like a sanity check. If you cannot explain how a prediction is made, you cannot know how good it is. (Song *et al.*, 2018) uses the multi-head self-attention mechanism to evaluate correlations between features - globally and locally . (Zhou *et al.*, 2017) plots attention and categorical embeddings.

(Haldar *et al.*, 2018) notes that the permutation test only produces sensical results on the assumption that the features are independent. Permuting the feature independently created examples that never occurred in real life, and the importance of features in that invalid space sent us in the wrong direction. The test however is somewhat useful in determining features that were not pulling their weight. If randomly permuting a feature did not affect the model performance at all, it was a good indication that the model is probably not dependent on it.

4.6 Other

4.6.1 Hyperparameter Selection

- 1cycle not used with tabular data before

Choosing DNN structural hyperparameters. Since there are no shared patterns among the diverse tabular datasets, it is hard to design a universal architecture that will fit all. Most of these parameters are very dependent on the dataset and other modeling choices and therefore the need to tune them. Structural hyperparameters are usually found using some brute search. (Song *et al.*, 2018) used a sigmoid layer for binary classification. Hidden layer size: 32 (Song *et al.*, 2018) Number of hidden layers: 32 (Song *et al.*, 2018) experimented to see how many layers they should choose [1,2,3,4]. (Wang *et al.*, 2017) tested number of layers [2-5] (Wang *et al.*, 2017) tested hidden layer sizes at [32-1024] (Zhang *et al.*, 2016), (Qu *et al.*, 2016), (Guo *et al.*, 2017), (Covington *et al.*, 2016) experimented for number and size of layers. Activation functions: tanh (Zhang *et al.*, 2016), tanh vs sigmoid vs relu (tanh and relu depending on dataset) (Qu *et al.*, 2016), relu vs tanh (relu better) (Guo *et al.*, 2017) Shapes: Diamond, constant, increasing, decreasing (Zhang *et al.*, 2016) found that Diamond shape works best. found constant to work the best (Guo *et al.*, 2017)

Choosing DNN learning hyperparameters. Loss function: logloss with L_2 penalty (Song *et al.*, 2018), (Wang *et al.*, 2017), (Zhang *et al.*, 2016),

(Qu *et al.*, 2016) mini-batch aware l2 for large inputs (Zhou *et al.*, 2017), Batch size: 1024 (Song *et al.*, 2018) 512 (Wang *et al.*, 2017) Optimiser: Adam (Song *et al.*, 2018), (Wang *et al.*, 2017) Gradient clipping: (Wang *et al.*, 2017) Learning rate: 0.001-0.0001 (Wang *et al.*, 2017), [1, 0.1, . . . , 0.0001] (Zhang *et al.*, 2016) Early stopping: (Song *et al.*, 2018), (Wang *et al.*, 2017), (Zhang *et al.*, 2016)

4.7 Still to categorise

(Zhou *et al.*, 2017) is very similar to the rest of these citations.

Chapter 5

Experiments

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

— Ali Rahimi and Ben Recht, *NIPS 2017*

5.1 Introduction

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

In this chapter we aim for good empiricism by evaluating the models on different types of prediction tasks and datasets, exploring many hyperparamters and doing cross-validation for unbiased performance measures along with standard errors. Our work is not necessarily about beating the benchmark and consist of simple experiments that aid in the understanding of how the techniques work, as used throughout the thesis.

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

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

to reproduce the experiments that were done in this thesis[2]. This is often an overlooked feature of experiments, but is however crucial for transparent and accountable reporting and making your work useful for others to build upon.

The main aim of this chapter is to better understand the behaviours of certain models and parameters and to cross-check the literature with empirical observations. We focus on the same main issues when it comes to DL on tabular as in the previous chapter, which are:

- how to represent the inputs,
- how to learn from feature interactions, and
- how to fight overfitting.

The more general hyperparameters, like learning rate, batch size, layer size and layer depth will not receive attention here since it has already been discussed at relevant parts previously. However, since these parameters are tightly linked with each other and other model parameters, we still do a hyperparamter search where we deem appropriate and report the findings in Appendix B. The rest of the chapter continues as follows: In §5.2 an overview of the datasets used for these experiments are given and why the were chosen. We discuss our evaluation procedure and metrics in §5.3. Thereafter we start with the main experiments.

5.2 Datasets

Our experiments are done on multiple datasets. Thus we can distinguish between findings that are only true for certain datasets and tasks and findings that hold more universallt.

The criteria for selecting the datasets were: - Strong model performance baselines exist; so that we can determine how far we are from the SoTA and that is actually a relevant problem. - Entirely open source; so that anyone can access it, reproduce it and build on it. - More than 20,000 observations; since NNs are data hungry. - Does not require too much preprocessing; so that most of the energy goes into the modelling phase. - Contain a mix of continuous and categorical features.

[2]All of these are shared publicly at https://github.com/jandremarais/tabularLearner

We chose two datasets for regression, one for binary classification and one for multi(class/label) classification from the UCI machine learning repository [Dua2017]. The chosen datasets are:

The Adult dataset[3]. This dataset was collected during a census. The task here is to predict whether or a not a certain person's income exceeds $50,000 per year. The features available are things like *age*, *education*, *sex* and *race*. In total there are 14 features and 48,842 observations.

Forest Cover Type[4]: Predicting forest cover type from cartographic variables. This is a multiclass clasification task. There are 581,012 observations.

Taxi Fare Prediction: Regression task (possibly) https://www.kaggle.com/c/new-york-city-taxi-fare-prediction

- Maybe https://www.kaggle.com/c/costa-rican-household-poverty-prediction/data
- Maybe https://www.kaggle.com/c/home-credit-default-risk

Look at the datasets used by (Anonymous, 2019), also (Zhang *et al.*, 2017)

- Criteo (Song *et al.*, 2018), (Wang *et al.*, 2017), (Qu *et al.*, 2016), (Guo *et al.*, 2017)
- Avazu (Song *et al.*, 2018)
- KDD12 (Song *et al.*, 2018)
- MovieLens-1M (Song *et al.*, 2018)
- iPinYou (Qu *et al.*, 2016)
- Forest Cover type (Wang *et al.*, 2017)
- Higgs (Wang *et al.*, 2017)
- Tox21 (Klambauer *et al.*, 2017)
- Yahoo (Anonymous, 2019)
- Letor (Anonymous, 2019)
- Protein (Anonymous, 2019)
- A9A (Anonymous, 2019)
- Flight (Anonymous, 2019)

Regularisation learning paper (Shavitt and Segal, 2018) only tested their approach on one dataset.

[3]http://archive.ics.uci.edu/ml/datasets/Adult
[4]https://archive.ics.uci.edu/ml/datasets/covertype

5.3 Evaluation

(Klambauer *et al.*, 2017) did a once-off three-way split of the data into training, validation and testing datasets. Hyperparemeter decisions were made based on the valdiation dataset performace and then the selected models are compared on the test datasets. The models were compared using the pairwise Wilcoxon rank test. The problem with doing a once-off split is that it does not account for the variance of the model and the performance of the model can in fact be very sensitive to the subset of data. By doing cross-validation, we can have more robust performance metrics, including the benefit of reporting on standard errors.

The other problem with (Klambauer *et al.*, 2017) is that they only tested on classification tasks and not regression. Models can behave quite differently on the two types of tasks. (Zhang *et al.*, 2017) also only tested mixup data augmentation on tabular datasets where the task was classification.

For the regression tasks we will compare the various models using the mean squared error and for classificaion we use cross-entropy. These are the metrics directly being optimised during the training process. When comparing the results to previous work, we base it on the metrics that are commmon for the specific dataset.

5.3.1 Metrics

- loss function
- task specific
- dataset specific
- time and memory

AUC, Logloss for binary classification (Song *et al.*, 2018), (Wang *et al.*, 2017), (Zhang *et al.*, 2016), (Qu *et al.*, 2016) No cross-validation (Song *et al.*, 2018), (Zhang *et al.*, 2016) Run time per epoch (Song *et al.*, 2018), (Guo *et al.*, 2017) Model size (Song *et al.*, 2018) Think (Wang *et al.*, 2017) used CV since the report se's for hyperparameter tunings.

5.3.2 Cross-validation

For most of the experiments we will do a 5-fold cross validation (Hastie *et al.*, 2009, p. 241) to estimate the performance of a model. That is, randomly

dividing the dataset in five equal parts and then in turn, hold out one of those parts for validation purposes and train the model using the remaining four parts. Figure 5.1 visually explains how the dataset is sub-divided. The performance of the model can then be evaluated on the held-out part. This process is repeated for every one of the five segmentations of the dataset and thus five measurements of the performance of model is obtained. We can then compute the average over these five measurements to obtain a less biased estimate of the model performance. Another advantage of this approach is that we can obtain standard error for the model performance.

Cross-validation is rarely done in Deep Learning, since the models typically take very long to train and any repitition is thus more costly. However, Deep Learning is also mostly applied to large datasets and if a large test set is available, the gains from cross-validation diminishes. Fortunately, the NNs applied to tabular data are much smaller than ones used for unstructured data and for this work we have access to sufficient computing power. And therefore cross-validation makes sense.

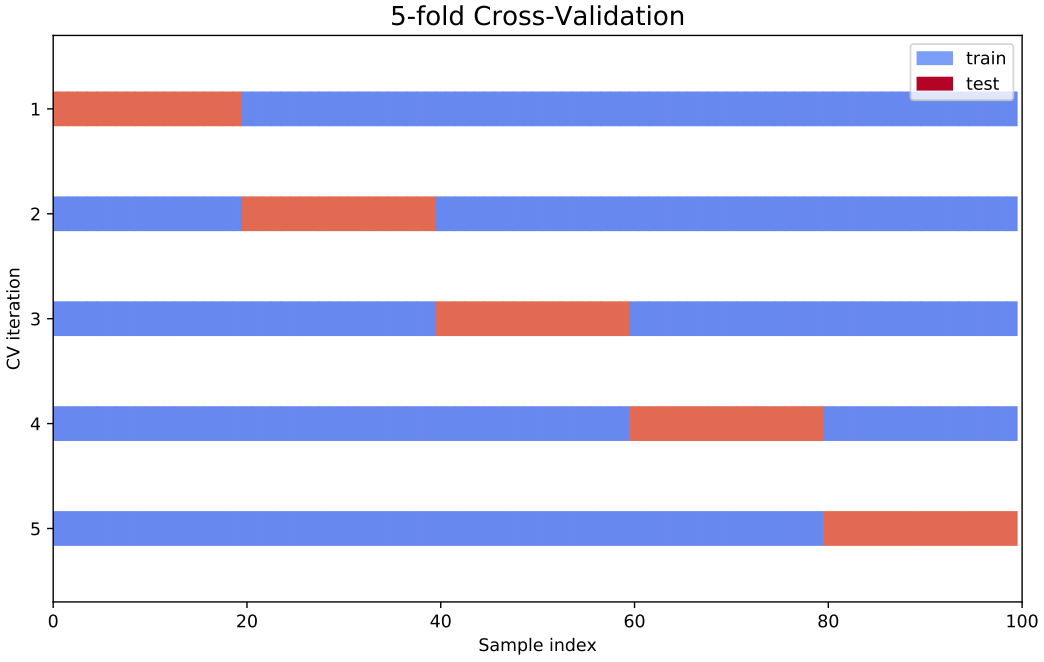


Figure 5.1: 5-Fold Cross-validation dataset split schematic.

5.4 General Approach

- which numeric normalisation
- should we embed numerics

Unfortunately, we cannot follow the hyperparameter selection process suggested by (Smith, 2018) for all the experiments. The process is too manual. Therefore we follow the approach once on each dataset to find a good selection of learning rate, number of epochs and weight decay and then use these paramters for the rest of the experiments on this dataset. If the model significantly changes over experiments, we might need to rerun the parameter selection process. Thus we will definitely not find the optimal model for each experiment but it should be sufficient to use as comparisons. According to (Smith, 2018) these parameters are also quite robust and the model is not too sensitive on these choices.

Might follow a bit of a greedy approach when selecting optimal paramters. All the hyperparameters are very dependent on each other but we cannot run experiments for every possible combination. Therefore we find optimal parameters for a certain experiment and then assume that these parameters are also good for other experiments.

Preprocessing. - Remove infrequent features (Song *et al.*, 2018) - numeric transform by $\log^2(z)$ if $Z > 2$ (Song *et al.*, 2018). - numeric features log transform (Wang *et al.*, 2017)

The idea is to do as little feature engineering as possible. So the steps we take here are generic steps that are applicable to any dataset. We do no feature selection since we would want the model to learn by itself which features are relevant.

5.5 Architectural Search

- Number of layers (vs with SeLU) (vs other activations) [1-10]
- Layer size [32-2048]
- Architecture shape [Constant, increasing, decreasingm diamond]
- Embedding sizes [proportional, fixed]
- dropout [0-1]

Recently found that the below experiments were already done by (Guo *et al.*, 2017, Qu *et al.* (2016), Zhang *et al.* (2016)). This was however only explored for Click-through rate prediction data. Thus the below experiments should be done in the light of these findings and can be compared to the their findings.

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 hope to find a rule of thumb that might act as a good starting point and guideline to choose the network size. We also want to get a feel for how important these hyperparameters are.

- Constant size Layer sizes: 32, 64, 128, 256, 512, 1024, 2048 Number of layers: 1,2,3,4,5,6 At a constant dropout.

Suppose we choose three layers, compare the following shape at approximately equal number of parameters. Shapes: Constant, decreasing, increasing, diamond, hourglass

The aim of these experiments are to evaluate performance at different embedding sizes. We explore embedding sizes at different ratios of the cardinality of categorical variables. The ratios we look at are: 10%, 20% 30%, 40%, 50%, 60%, 70%, 80% and 90% of the cardinality of each categorical feature. Possibly explore certain max embedding sizes.

As we increase the size we will also look at the effect it has by visually inspecting the embedding layers in a 2-dimensional space.

Again, we expect there to be optimal embedding size for each variable depedning on the cardinality of the variable and how complex its relationship is with the other variables and the target. We expect the ideal embedding size to be as small as possible but still being able to capture all of the information of the variable.

Look at wide and deep models from (Cheng *et al.*, 2016*a*). They restrics all embeddings to be of size 32.

5.6 Sample Size

- accuracy vs size of dataset

5.7 Mixup

- does it help the validation loss

5.8 Pretraining

- does it help the validation loss
- Are these features useful for tree based methods.

5.9 Attention

- with residual

5.10 Comparisons To Tree-based Methods

- Compare Neural Networks to Gradient Boosted Machines and Random Forests.

5.11 Example Interpretation

- plot embeddings
- plot attention matrices
- SHAP and permutation

Chapter 6

Conclusion

- What was done in the thesis?
- Is Deep Learning useful for tabular data?
- If it is, when?

6.1 Promising Future Directions

- Where should future work on the subject focus on?

Variational autoencoders for pretraining. Generative Adversarial Network for data augmentation (and other tab data specific augs). Feauture reuse per feature regularisation.

- more datasets
- review the code

Appendices

Appendix A

Datasets

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

Appendix B

Hyperparameter Search

Appendix C

Software and Code

- Deep Learning Library: Pytorch and Fastai
- Hardware: GTX1080Ti
- Python and Jupyter Notebooks for programming environemnt
- github for version control
- RMarkdown for writing and compiling the thesis document

C.1 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 [1]. More instructions on how to implement the code is contained in the file named `README.md`, in the repository.

[1] https://github.com/jandremarais/tabularLearner

Bibliography

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

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

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

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

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

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

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

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

Chapelle, O., Weston, J., Bottou, L. and Vapnik, V. (2001). Vicinal risk minimization. In: Leen, T.K., Dietterich, T.G. and Tresp, V. (eds.), *Advances in Neural*

Information Processing Systems 13, pp. 416–422. MIT Press.
Available at: http://papers.nips.cc/paper/1876-vicinal-risk-minimization.pdf

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

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

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

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

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

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

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

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

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

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.

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

Geras, K.J. and Sutton, C.A. (2014). Scheduled denoising autoencoders. *CoRR*, vol. abs/1406.3269. 1406.3269.
Available at: http://arxiv.org/abs/1406.3269

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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. (). Rectifier nonlinearities improve neural network acoustic models.

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

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

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

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

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

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

Rajkomar, A., Oren, E., Chen, K., Dai, A.M., Hajaj, N., Liu, P.J., Liu, X., Sun, M., Sundberg, P., Yee, H., Zhang, K., Duggan, G.E., Flores, G., Hardt, M., Irvine, J., Le, Q.V., Litsch, K., Marcus, J., Mossin, A., Tansuwan, J., Wang, D., Wexler, J.,

Wilson, J., Ludwig, D., Volchenboum, S.L., Chou, K., Pearson, M., Madabushi, S., Shah, N.H., Butte, A.J., Howell, M., Cui, C., Corrado, G. and Dean, J. (2018). Scalable and accurate deep learning for electronic health records. *CoRR*, vol. abs/1801.07860. 1801.07860.
Available at: http://arxiv.org/abs/1801.07860

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

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

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

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

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

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

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

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

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

Smith, L.N. (2015). No more pesky learning rate guessing games. *CoRR*, vol. abs/1506.01186. 1506.01186.
Available at: http://arxiv.org/abs/1506.01186

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

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

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

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

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

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

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

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

Wu, Y., Schuster, M., Chen, Z., Le, Q.V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., Kaiser, L., Gouws, S., Kato, Y., Kudo, T., Kazawa, H., Stevens, K., Kurian, G., Patil, N., Wang, W., Young, C., Smith, J., Riesa, J., Rudnick, A., Vinyals, O., Corrado, G., Hughes, M. and Dean, J. (2016). Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, vol. abs/1609.08144. 1609.08144.
Available at: http://arxiv.org/abs/1609.08144

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

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

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

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

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

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

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