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 ...

Contents

List of Figures

List of Tables

List of Abbreviations and/or Acronyms

AA	Algorithm Adaptation
ANN	Artificial Neural Network
BR	Binary Relevance
CAD	Computer Aided Diagnosis
CC	Classifier Chains
CNN	Convolutional Neural Network
CV	Computer Vision
ECC	Ensemble Classifier Chains
kNN	k-Nearest Neighbour
LP	Label Powerset
mAP	Mean Average Precision
ML-kNN	Multi-Label k-Nearest Neighbour
MLC	Multi-Label Classification
MLIC	Multi-Label Image Classification
PT	Problem Transformation
RAkEL	Random k-Labelsets
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

Deep learning resulted in tremendous improvements in many machine learning applications, especially in the domains of image, text and audio processing. The datasets in these domains are what some call unstructured data. Why is it called unstructured? In a sense the data is homogeneous. Cite reviews of deep learning in these domains. Show the growth of deep learning papers, conference applications and deep learning software. But where we haven't seen much exploration of deep learning is applying it to structure data also referred to as tabular data. Tabular data is also important. But each column is different and thus in a way more difficult to learn representations. At the moment methods on tabular data are dominated by tree based boosting methods. See kaggle competitions. In some cases where there was enough data deep learning got a slight upperhand. But it is still not clear when a tabular dataset is best suited for dl and neither how then to apply dl to such a dataset. This thesis acts as an tutorial on applying dl to tabular data. We will look at existing work on the matter, see that it is lacking, see what we can borrow from the other domains, do an empirical study to look for clues. Especially layers, embeddings, pretraining, augementation, modern training policies, batch size. The use of dl is often restricted by its perceived lack of interpretability and the here we will explore ways that we can interpret them with model agnostic and nn specific methods.

Deep learning is a revitalization of artifical neural networks or multilayer perceptrons. Nns have been use on tabular data but old techniques and very few of the moden techniques have been tested on tabular data.

Deep learning has already created .significant improvements in computer vision, speech recognition, and natural language processing

One of the first sucessful implementations of modern NNs for tabular data was in predicting the destination of a taxi ride based on its initial trajectory (de Brébisson *et al.*, 2015). It was hosted as a Kaggle competition and this solution outperformed all other entries by a significant margin.

Many tabular data sets are challenging to represent and model due to its high dimensionality, noise, heterogeneity, sparseness, incompleteness, random errors, and systematic biases (Miotto *et al.*, 2016).

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 algotihms only result inincremental 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 not 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.

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.

(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.

1.1 Problem Description

- Motivation
- Goal
- want to see if a machine can learn useful features for predictive modelling on unlabelled tabular data.

1.2 Background

- (Un)Supervised Learning
- regression/classification

1.2.1 Statistical Learning

Machine or statistical learning algorithms (used interchangably) are used to perform certain task that are too difficult or inefficient to solve with fixed rule-based programs. These algorithms are able to learn how to perform a task from data. For an algorithm to learn from data means that it can improve its ability in performing an assigned *task*, with respect to some *performance measure*, by processing *data*. This section gives a brief look at some of the important types of tasks, data and performance measures in the field of statistical learning.

A learning task describes the way an algorithm should process an observation. An observation is a collection of features that have been measured from 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, $j = 1, \dots, p$. 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 with statistical learning. One of the most common learning tasks is that of *classification*, where it is expected of an algorithm to determine which of K categories an input belongs to. To solve the classification task, the learning algorithm is usually asked to produce a function $f : \mathbb{R}^p \to \{1, \dots, K\}$. 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 the other 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 to classification is the format of its output.

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

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

Often the dataset includes annotations for each observation in the form of a label (classification) or a target value (regression). The N annotations are represented by the vector $\boldsymbol{y}$, where 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, *supervised* and *unsupervised* algorithms, 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 makes 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 evaluate the algorithm on a *test set* of data points, independent of the *training set* of data points 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 predicts the value of a scalar $y \in \mathbb{R}$ in 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 $\hat{\boldsymbol{w}} = [w_0, w_1, \ldots, w_p]$ is a vector of parameters and $\boldsymbol{x} = [1, x_1, x_2, \ldots, x_p]$. 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. Generalisation 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. A thorough discussion of MLC methods is given in **??**. 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 **??** 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 within [0,1]. A way to overcome this problem is to estimate the posterior probabilities using the *logit transform* of $\hat{f}_k(\boldsymbol{x})$. That is,

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

Through this transformation, the estimates of the posterior probabilities both sum to 1 and are squeezed into [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 following section.

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*.

1.3 Outline

Chapter 2

Neural Networks

2.1 Introduction

A Neural Network (NN), like any other machine learning model, is a function that maps inputs to outputs, *i.e.*

$$f : \boldsymbol{x} \to y.$$

The NN, f, receives input, $\boldsymbol{x}$, and produces output, y. What happens inside of f is loosely based on biological neural systems, or the brain. The brain consists of a collection of interconnected neurons, each sending and receiving signals between each other. An artifical NN tries to copy this stucture by modelling what happens inside of a single neuron by outputting a weighted combination of its inputs, combined with a simple non-linear transformation. The output of a neuron is referred to as activations. These neurons are grouped in so-called layers. At each layer the input is passed through each of the neurons and their activations, then in turn, gets passed to the next layer. See Figure 2.1 for an illustration of this structure. A more detailed explanation of the structure of a NN is given in section 2.2

The transformation at each neuron is controlled by a set of parameters, also known as weights. These weights can be tuned to obtain a desired output. When training a NN to perform a certain machine learning task, for instance classification, the NN is fed a bunch of data and tweaks its weights so that the resulting output matches the true target as close as possible. This process of tweaking the weights according to the data is done by an optimisation algorithm called Stochastic Gradient Descent (SGD). SGD and NN training is covered in detail in section 2.3.

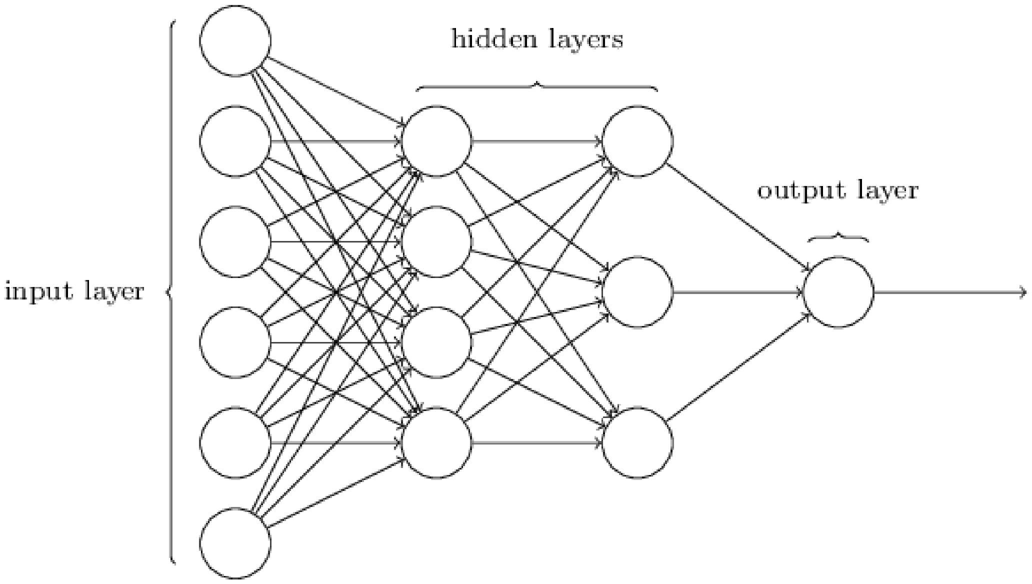

Figure 2.1: The structure of an artifical neural network (This is still a placeholder).

There has been plenty of excitement around NNs recently, but in fact NNs have quite a bit of history. The development of NNs dates at least as far back as Perceptrons in (Rosenblatt, 1962). It is also interesting to compare modern NNs with the Projection Pursuit Regression algorithm (Friedman and Stuetzle, 1981) developed in statistics. Only recently a series of breakthroughs allowed NNs to be more efficient and effective and therefore the revitalisation of the field.

The modular nature of a NN allows it to accept inputs and produce outputs of various shapes and sizes. Therefore NNs can be used for just about any machine learning task, from doing simple binary classification on tabular data, to generating full color images from black and white sketches. Modern structures like the Convolutional Neural Network and the Recurrent Neural Network are all based on the vanilla NN structure and training procedure explored in the rest of this chapter.

2.2 The Structure of a Neural Network

Recall, a NN processes an input by sending it through a series of layers, each applying some transformation to its input, to eventually produce an output and each layers consists of smaller computational units, called neurons. To understand and formulate the NN structure, we will start by describing the operation inside a single neuron and then gradually put the pieces together to form layers and then a complete NN. Suppose we want a function that estimates a taxi fare given the distance travelled, duration of the trip and number of passengers. A single neuron can act as such a function by taking a weighted average of these three inputs to produce an estimate of the taxi fare. **??** is a graphical representation of this function. In equation form, this function can be written as:

$$w_1 \cdot \text{distance} + w_2 \cdot \text{time} + w_3 \cdot \text{passengers} + b = \text{fare},$$

where w_i, $i = \{1, 2, 3\}$, are the weights applied to each of the inputs and b a constant added to the equation, better known as the bias term in machine learning. Clearly, this equation is simply the very common linear model and thus also can be written as:

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

where $\boldsymbol{x} = [\text{distance} \quad \text{time} \quad \text{passengers}]^\intercal$ is the input, $\boldsymbol{w} = [w_1 \quad w_2 \quad w_3]^\intercal$ the weights and z the output, *i.e.* the taxi fare. For convenience, we sometimes compress the above equation to $\boldsymbol{x}^\intercal \boldsymbol{w} = z$, where $\boldsymbol{x}$ includes the bias term and the weight vector $\boldsymbol{w}$ a unit element, *i.e.* $\boldsymbol{x} = [b \quad \text{distance} \quad \text{time} \quad \text{passengers}]^\intercal$ is the input, $\boldsymbol{w} = [1 \quad w_1 \quad w_2 \quad w_3]^\intercal$.

The weights determine how much each of the inputs contribute to the fare. For example, the distance (in km's) may be the most important driver of the taxi fare but the duration of the trip (in minutes) has little influence and the number passemgers has no effect. Then the weights may look something like this:

$$w_1 = 10, \quad w_2 = 0.5 \quad \text{and} \quad w_3 = 0.$$

But we do not know what these weights are before hand and therefore need to estimate them. With the classical linear model, these weights (or coefficients) are estimated using the ordinary least squares (OLS) method. Since a NN consists of many inter-conncected neurons, the OLS methods will not suffice. This is the topic of the next section.

Suppose a single neuron (or a linear model if you like) is not flexible enough to model the taxi fare given the distance, time and number of passengers. Now we decide to add another neuron. This neuron also accepts the same inputs as the first, but uses a different set of weights to estimate the fare. Now we have two neurons, each producing a different output:

$$\boldsymbol{x}^{\mathsf{T}}\boldsymbol{w}_1 = z_1 \quad \text{and} \quad \boldsymbol{x}^{\mathsf{T}}\boldsymbol{w}_2 = z_2.$$

So how do we get a final estimate of the fare from these two initial estimate? We feed it to another neuron of course, *i.e.*

$$\boldsymbol{z}^{\mathsf{T}}\boldsymbol{w}_3 = y$$

See **??** for a graphical representation.

The first two neurons both took in the distance, time and passengers as input and produced a single output. These operations can be expressed as a single equation, *i.e.*

$$\boldsymbol{x}^{\mathsf{T}}W = \boldsymbol{z}^{\mathsf{T}},$$

where

$$W = \begin{bmatrix}\boldsymbol{w}_1 & \boldsymbol{w}_2\end{bmatrix} = \begin{bmatrix} 1 & 1 \\ w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{33} & w_{32} \end{bmatrix} \quad \text{and} \quad \boldsymbol{z} = [z_1 \quad z_2]^{\mathsf{T}}.$$

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) takes the output of this layer as input, it is possible to express the complete input-output relationship in one equation, *i.e.*

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

Note here that the weights from the first layer, W, and the third neuron, $\boldsymbol{w}_3$, can collapse into a single vector $\boldsymbol{w}$, effectively reducing all of the neuron operations back into a single neuron representation and thus is clearly not a good way to model a network

However, a NN has a way to prevent this collapsing from happening and to allow for non-linear relationships between the inputs and outputs. It does this through the use of an activation function, a simple non-linear transformation. An activation is applied after each linear layer. So now the NN equation can be represented as:

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

Where a_1 is the activation function after the first linear layer and a_2 the activation after the final layer.

By introducing the non-linear activations, it greatly enlarges the class of functions that can be approximated by the network.

TBC

The activation function, $a(\cdot)$, was usually chosen to be the sigmoid function, $a(v) = \frac{1}{1+e^{-v}}$

In the previous section, we introduced activation functions, which are simple non-linear functions of its input. These are usually applied after a fully connected layer (linear transformation) and are crucial for the flexibility of a deep neural network. We also mentioned that the sigmoid activation, which was originally the go-to activation, is currently not the most popular choice. Another activation function originally thought to work well was, $a(x) = \tanh(x)$. However, by far the most common activation function used at the time of writing is the Rectified Linear Units (ReLU) non-linearity. Its definition is much simpler than its name and is defined as $a(x) = \max(0, x)$. It was introduced in (Krizhevsky *et al.*, 2012) and they showed that using ReLUs in their CNNs reduced the number of training iterations to reach the same point by a factor of 6 compared to using $\tan(x)$. The ReLU limits the gradient vanishing problem as its derivative is always one when x is positive. Gradient vanishing problem?

There are a plethora of proposals for activation functions, since any simple non-linear (differentiable?) function can be used. Some of the recent most popular choices are exponential linear units (ELUs) (Clevert *et al.*, 2015) and scaled exponential linear units (SELUs) (Klambauer *et al.*, 2017). The choice of activation function usually influences the convergence time and some might protect the training procedure from overfitting in some cases. The different activation functions can be experimented with, however it would be sufficient in most cases to use ReLUs. The other mentioned proposals have inconsistent gains over ReLUs and therefore it remains the standard choice.

However, very recently (Ramachandran *et al.*, 2017) used automated search techniques to discover novel activation functions. The exhaustive and reinforcement learning based searched identified a few promising novel activation functions on which the authors then did further empirical evaluations. They found that the so-called *Swish* activation function,

$$a(x) = x \cdot \sigma(\beta x),$$

where β is a constant (can also be a trainable parameter), gave the best empirical results. It consistently matched or outperformed ReLU's on deep networks applied to the domains of image classification and machine translation.

The number of units in the hidden layer, M, is also a value to be decided on. Too few units will not allow the network enough flexibility to model complex relationships and too many takes longer to train and increases the chance of overfitting. M is mostly chosen by experimentation. A good starting point would be to choose a large value and training the network with regularisation (discussed shortly).

The difference between the above discussed neural networks and current state-of-the-art deep learning methods, is the number and type of hidden layers. The following section discusse the popular activation functions used in DNNs.

The units in $\boldsymbol{Z}$ are called hidden since they are not directly observed. The aim of this transformation is to derive features, $\boldsymbol{Z}$, so that the classes become linearly separable in the derived feature space (Lecun *et al.*, 2015). Many more of these hidden layers (combination of linear and non-linear transformations) can be used to derive features to input into the final classifier. This is what we refer to as deep neural networks (DNNs) or deep learning methods.

- comment on number and size of layers
- lead into modern architectures
- lead into parameter optimisation

2.3 Training a Neural Network

2.3.1 Optimisation

As mentioned before, fitting a linear regression model can be reduced to finding the optimal weights to minimise the MSE function (with or without weight decay). In fact, typically model training procedures can be described as the search for its internal parameters that minimises or maximises some *objective function*. Therefore 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 remainder of the thesis.

As mentioned in the previous section, 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 toward 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. This value can be determined via a line search but is not always ideal since this may render the training time of DNNs too long. Another option is to reduce the learning rate after every fixed number of iterations. More detail regarding the implication of the learning rate will be given in **??**.

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 DNNs. 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 data set 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 **??**. 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 **??**, 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 networks 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.

This section discusses the same procedure, but applied to a simple single hidden layer neural network. This is made possible by the *backpropogation* algorithm. Note, this process extends naturally to the training of deeper networks.

The neural network described in the previous section has a set of unknown adjustable weights that defines the input-output function of the network. They are the $\alpha_{0m}, \boldsymbol{\alpha}_m$ paramters of the linear function of the inputs, $\boldsymbol{X}$, and the $\beta_{0k}, \boldsymbol{\beta}_k$ paramaters of the linear transformation of the derived features, $\boldsymbol{Z}$. Denote the complete set of parameters by θ. Then the objective function for regression can be chosen as the sum-of-squared-errors:

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

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 $L(\theta)$.

Therefore, one rather uses gradient descent and backpropogation 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. The relevant derivatives for the algortihm 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,

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

where γ_r is called the learning rate. Now write the gradients as

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

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 give obtain s_{mi}. These are then used to update the weights.

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. One sweep of the batch learning through the entire training set is known as an epoch. It can take many training epochs for the objective function to converge.

2.3.4 Learning Rate

The convergence times also depends on the learning rate, γ_r. There are no easy ways for determining γ_r. 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 will 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. One possible strategy for effective training is to decrease the learning rate every time after a certain amount of iterations.

Recently, in (https://arxiv.org/abs/1711.00489) (no bibtex entry), the authors found that, instead of learning rate decay, one can alternatively increase the batch size during training. They found that this method reaches equivalent test acccuracies compared to learning rate decay after the same amount of epochs. But their method requires fewer parameter updates.

2.3.5 Basic Regularisation

There are many ways to prevent overfitting in deep neural networks. The simplest strategies for single hidden layer networks are by early stopping and weight decay. Stopping the training process early can prevent overfitting. When to stop can be determined by a validation set approach. Weight decay is the addition of a penalty term, $\lambda J(\theta)$, to the objective function, where,

$$J(\theta) = \sum_{km} \beta_{km}^2 + \sum_{ml} \alpha_{ml}^2.$$

This is exactly what is done in ridge regression (Hastie *et al.*, 2009, Ch. 4). $\lambda \geq 0$ and larger values of λ tends to shrink the weights towards zero. This helps with the generalisation ability of a neural network, but recently more effective techniques to combat overfitting in DNNs have been developed. These are dicussed in **??**.

It is common to standardise all inputs to have mean zero and standard deviation of one. This ensures that all input features are treated equally. Now we have covered all of the basics for simple (1-layer) neural networks.

- move regularisation to next chapter
- lead into modern learning policies
- lead into what it is learning

2.4 Representation Learning

- What is the Neural Network actually doing?
- See (Bengio *et al.*, 2013)

Each layer of the network is trained to produce a higher-level representation of the observed patterns, based on the data it receives as input from the layer below, by optimizing an objective function. Every level produces a representation of the input pattern that is more abstract than the previous level because it is obtained by composing more non-linear operations.

2.5 Summary

Appendices

Appendix A

Appendix A

Description of each of the datasets used in Experiments.

Bibliography

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.

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.

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

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

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

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/

Klambauer, G., Unterthiner, T., Mayr, A. and Hochreiter, S. (2017 June). Self-Normalizing Neural Networks. *ArXiv e-prints.* 1706.02515.

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

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

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*.

Ramachandran, P., Zoph, B. and Le, Q.V. (2017). Searching for activation functions. *CoRR*, vol. abs/1710.05941. 1710.05941.
Available at: http://arxiv.org/abs/1710.05941

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