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.

1.1 Problem Description

- Motivation
- Goal

1.2 Background

- (Un)Supervised Learning
- regression/classification

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.*

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

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 2.4.

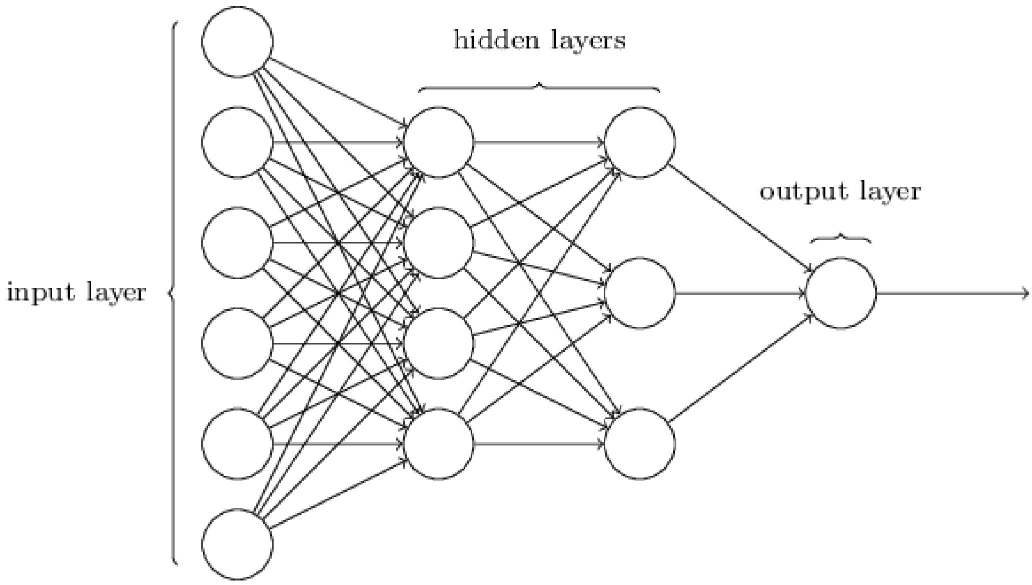

Figure 2.1: The structure of an artifical neural network (This is still a placeholder).

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:

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

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.*

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

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.*

$$\boldsymbol{x}^\intercal W = \boldsymbol{z}^\intercal,$$

where

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

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

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

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.

TBC

2.3 OLD STRUCTURE SECTION

A NN maps inputs to outputs according to a series of simple functions or transformations stacked on-top of each other. One of the main building blocks of an NN is the very common linear transformation. For an easy introduction to NNs we first discuss the linear model. Consider the linear model for the task of binary classification where the target is coded as zeros and ones and thus $f : \mathbb{R}^p \to \{0, 1\}$. The linear model assumes the the output, y, can be obtained from a weighted average of the input $\boldsymbol{x}$ and thus we can write:

$$f(\boldsymbol{x}) = w_0 + w_1x_1 + w_2x_2 + \cdots + w_px_p = w_0 + \boldsymbol{w}^\intercal\boldsymbol{x},$$

where w_j, $j = 1, 2, \ldots, p$, is the *weight* applied to the j-th feature, also referred to as *coefficients* in classical statistics, and w_0 a scalar added to the weighted combination, known as the *bias* term in machine learning or the *intercept* in statistics. For convenience we usually write

$$f(\boldsymbol{x}) = \boldsymbol{w}^\intercal\boldsymbol{x},$$

where we include the bias term in the weight vector $\boldsymbol{w}$ and add a constant feature to the inputs:

$$\boldsymbol{w} = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_p \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_p \end{bmatrix}.$$

The true values for $\boldsymbol{w}$ are unknown and therefore we need to estimate them from the data. The estimated weights are denoted as $\hat{\boldsymbol{w}}$ and the output obtained using the estimated weights (*i.e.* the predictions) is given by:

$$\hat{y} = \hat{f}(\boldsymbol{x}) = \hat{\boldsymbol{w}}^\intercal\boldsymbol{x}$$

Similar to the description in **??**, we want the prediction $\hat{y}$ to be as close as possible to the true value y, measured by a loss function, L, and therefore choose $\hat{\boldsymbol{w}}$ as:

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}^*} \sum_i L(y_i, \boldsymbol{w}^{*\intercal}\boldsymbol{x}),$$

which can be found using an optimisation algorithm, discussed shortly. A common loss function, typically used for regression but used here for illustration purposes, is the *squared error* loss:

$$L_{MSE}(y_i, \hat{y}_i) = (y_i - \hat{y}_i)^2$$

The simplest form (and also the origin) of DNNs is a *feedforward neural network*, also known as the *multilayer perceptron* (MLP). They are called *feedforward* because information flows through the function being evaluated from the inputs $\boldsymbol{X}$, through the intermediate computations used to define f, and finally to the output $\boldsymbol{Y}$ (Goodfellow *et al.*, 2016). The *network* in the name refers to the structure of this type of model which is most naturally visualised as a network of inter-connected nodes.

2.3.1 Single Layer Perceptron

Like most other supervised learning models, a neural network is a mapping from an input to an output. The central idea of a neural network is to extract linear combinations of the inputs as derived features, and then model the target as a non-linear function of these features (Hastie *et al.*, 2009, Ch. 11). This idea was developed separately in the fields of statistics and artificial intelligence. In statistics, the first methods built on this idea was called the Projection Pursuit Regression (PPR) model (see Hastie *et al.*, 2009, pp. 389-392). This model can be written as

$$f(\boldsymbol{X}) = \sum_{m=1}^{M} g_m(\boldsymbol{\omega}_m^T \boldsymbol{X}),$$

where $\boldsymbol{X}$ is the usual input vector of p components and $\boldsymbol{\omega}_m$, $m = 1, \ldots, M$, p-sized vectors with unknown parameters. Thus, the PPR model is an additive model in the derived features, $V_m = \boldsymbol{\omega}_m^T \boldsymbol{X}$. $g_m(\cdot)$ is called a ridge function and is to be estimated. V_m is the projection of $\boldsymbol{X}$ onto the unit vector $\boldsymbol{\omega}_m$, and we seek $\boldsymbol{\omega}_m$ such that the model fits well, hence the name, Projection Pursuit. The details of this method is beyond the scope of this thesis and can be found at the reference above.

The term neural networks is used for a large class of models and learning methods. First, consider the "vanilla" neural network, known as the single layer perceptron. It is a neural network with a single hidden layer and trained by backpropogation. It can be applied to both regression and classification. It takes an input, $\boldsymbol{X} : 1 \times p$, transforms it to a hidden layer $\boldsymbol{Z} : 1 \times M$ and then uses $\boldsymbol{Z}$ as input to model the target, $\boldsymbol{Y} : 1 \times K$. This structure can be represented as a network as shown in Figure 2.2.

The number of units in the final layer matches the dimensionality of the output, denoted by K. Thus for classic regression, $K = 1$, and for multiclass classification, K is the number of possible categories, where unit k, $k = 1, \ldots, K$, represents the score for class k. For this discussion we will describe neural

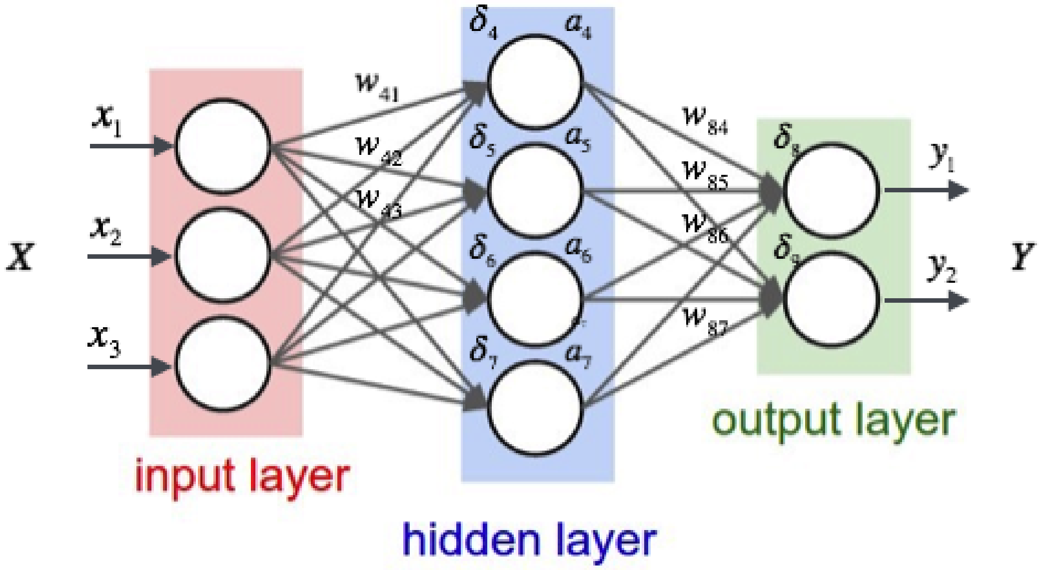

Figure 2.2: Graph structure of a vanilla neural network.

networks for multiclass classification. Thus there are K target measurements, $\boldsymbol{Y} = \{Y_1, Y_2, \ldots, Y_K\}$. Y_k is coded as 1 when class k is present and as 0 otherwise.

The hidden layer units, $\boldsymbol{Z} = \{Z_1, Z_2, \ldots, Z_M\}$, are a set of features derived from the input. They are created by first taking a linear combination of the inputs and then sending it through a non-linear *activation function*, $a(\cdot)$,

$$Z_m = a\left(\alpha_{0m} + \boldsymbol{\alpha}_m^T \boldsymbol{X}\right),$$

for $m = 1, \ldots, M$. α_{0m} and $\boldsymbol{\alpha}_m$ are the coefficients of the linear mapping. Note that a layer that outputs a linear transformation of its inputs in this fashion is also called a *fully-connected* or *dense* layer. The activation function, $a(\cdot)$, was usually chosen to be the sigmoid function, $a(v) = \frac{1}{1+e^{-v}}$. However these days, there are many, more effective activation functions used in deep neural networks which we discuss in Section 2.3.2.

The output units of the neural network can then be expressed as

$$f_k(\boldsymbol{X}) = g_k\left(\beta_{0k} + \boldsymbol{\beta}_k^T \boldsymbol{Z}\right),$$

for $k = 1, \ldots, K$. Here, the β's are the coefficients of the linear combination of the derived features, $\boldsymbol{Z}$, and $g_k(\cdot)$ is another activation function. Originally, for both regression and classification, $g_k(\cdot)$ was chosen to be the identity function, but they later found that the softmax function was better suited for multiclass classification, defined as

$$g_k(\boldsymbol{T}) = \frac{e^{T_k}}{\sum_k e^{T_k}}.$$

This function is exactly the transformation used in the multilogit model discussed in **??**. It produces output in the range [0,1], summing to 1, similar to the properties of conditional class probabilities.

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.

Note, that if the $a(\cdot)$ activation function was the identity function or another linear function, the whole network would collapse into a single linear mapping from inputs to outputs. By introducing the non-linear activations, it greatly enlarges the class of functions that can be approximated by the network (see universal approximator).

In a statistical learning sense, the hidden units can be thought of as a basis function expansion of the original inputs. The neural networks is then a standard linear (multilogit) model with the basis expansions as inputs. The only difference to the conventional basis function expansion techinique in Statistical Learning (Hastie *et al.*, 2009, Ch. 5) is that the parameters of the basis functions are learned from the data.

One can now also see the relationship between a neural network and the PPR model. If the neural network has one hidden layer, it can be written in the exact same form as the PPR model. The difference is that the PPR uses a nonparametric function $g_m(v)$, while the neural network uses far simpler non-linear activation functions, like $a(\cdot)$.

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.

2.3.2 Activation Functions

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 $\tan(x)$.

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.

2.4 Training a Neural Network

2.4.1 Backpropogation

In **??** we discussed how to fit a linear model using the Stochastics 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:

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

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. (I'm not sure if this is possible with deeper networks, and with the non-linear activations?). 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

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

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 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.

2.4.2 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.

2.4.3 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.

2.5 Representation Learning

- What is the Neural Network actually doing?

2.6 Summary

Chapter 3

Deep Learning

- Recent advancements in deep learning which could be useful to applying in tabular data

Chapter 4

Neural Networks for Tabular Data

- Considerations for applying DL to tabular data

4.1 Entity Embeddings

4.2 Normalising Continuous Variables

- how to normalize continuous variables
- mean subtract and error divide
- rankGauss
- scale to 0-1

4.3 Regularisation Learning

- https://arxiv.org/pdf/1805.06440.pdf

Chapter 5

Interpreting Neural Networks

5.1 Model Agnostic

- Permutation Importance
- Partial Dependece
- SHAP

5.2 Neural Network Specific

- Distilling Neural Networks, i.e. training a decision tree on train neural network generated data. https://arxiv.org/pdf/1711.09784.pdf
- Interpreting activations.
- Plotting embeddings in lower dimensional space with PCA or t-sne

Chapter 6

Experiments

6.1 Method

6.1.1 Datasets

- regression
- classification
- need multiple datasets for robust conclusions
- this project will not look at feature engineering so this part must be obtained from somewhere else if the data requires a lot of preprocessing.

6.1.2 Evalutation

- 5-fold CV for standard errors
- dataset specific metrics so that can compare to other work.
- training and inference times because sometimes it takes a lot of computing power and then not useful to everyone.

6.2 Structure

6.2.1 Number of Layers

- Evaluate training and performance as the number of layers increase

6.2.2 Size of Layers

- Evaluate training and performance as the the size of the layers increas

6.2.3 Size of Embeddings

- Evaluate training and performance at different embedding sizes.
- Inspect embedding matrices by plotting in lower dimensions.

6.2.4 Skip Connections

- ResNets and DenseNets
- See what it does to performance if every layers is connected to every other layer.

6.3 Training

6.3.1 One-cycle Policy

- Leslie Smith's 1 cycle and superconvergence work
- Is it better than standard training procedures w.r.t training time and performance

6.3.2 Batch Size

- how does batch size influence model metrics

6.3.3 Augmentation and Dropout

- How can we augment inputs
- Is dropout effective for regularising (and with above augmentations?)

6.4 Unsupervised Pre-training

6.4.1 Autoencoders

- How does initialising the net with autoencoder learned weights compare to random initialisation?

6.4.2 Feature Extraction

- Are these features useful for tree based methods.

6.5 Comparisons To Tree-based Methods

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

6.5.1 Sample Size

- Model perfomances at different number of samples

6.5.2 Number of Feature

- Model perfomances at increasing number of feature

6.5.3 Noise

- Model perfomances at different signal to noise ratios
- Shuffle columns of datasets before training

6.5.4 Feature Importance

- How does tree-based feature importance compare to permutation importance of neural net?

Chapter 7

Conclusion

- What was done in the thesis?
- Is Deep Learning useful for tabular data?
- If it is, when?
- Where should future work on the subject focus on?

Appendices

Appendix A

Appendix A

Description of each of the datasets used in Experiments.

Bibliography

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.

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.

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