TECHNISCHE
UNIVERSITÄT
WIEN

Diplomarbeit

Deep Learning in Life Insurance Risk Prediction

ausgeführt zum Zwecke der Erlangung des akademischen Grades einer

Diplom-Ingenieurin

eingereicht an der Technischen Universität Wien, Fakultät für Mathematik und Geoinformation

von

Caroline Gerharter, BSc

(Matr.Nr.: 01225897)

unter der Anleitung von

Univ.Prof. Dipl.-Math. Dr.rer.nat. **Thorsten Rheinländer**

Institut für Stochastik und Wirtschaftsmathematik
Technische Universität Wien
Wiedner Hauptstaße 8, 1040 Wien, Österreich

Wien, 20. Oktober 2019 ______________________ ______________________
(Betreuer) (Verfasser)

Abstract

This diploma thesis deals with the implementation of an artificial neural network to predict life insurance risks. At first, general terms of deep learning are declared and defined. A brief insight into life insurance risk, specifically into the crucial parameter, the probability of dying, is given. Consequently, the structure of a deep neural network, the different activation functions and optimisers are explained in detail. This also includes a precise explanation of the training algorithm of a deep neural network. Finally, the calculation of the probability of dying is performed. Several experiments are carried out to test different scenarios for the neural network and the simulations are thoroughly analysed. In conclusion, the calculation of the probability of dying via an artificial neural network worked exceptionally well. A model with four hidden layers, overall 640 neurons, the Adam optimiser and either the ELU, TanH or Softplus activation function yielded by far the best results for this problem.

Key words: Life insurance risk / Approximation Probability Of Dying / Life Table / Artificial Neural Network / Training Algorithm / Activation Function / Optimiser

Kurzfassung

Diese Diplomarbeit beschäftigt sich mit der Implementierung eines künstlichen neuronalen Netzes, um Lebensversicherungsrisiken vorauszusagen. Zu Beginn werden allgemeine Begriffe von maschinellem Lernen erklärt und definiert. Außerdem wird ein kurzer Einblick in die Lebensversicherung und vorallem in den gesuchten Parameter - die Sterbewahrscheinlichkeit - gegeben. Die Struktur eines tiefen neuronalen Netzes, die verschiedenen Aktivierungsfunktionen und Optimierer werden detailliert veranschaulicht, inklusive einer genauen Darlegung des Trainingsalgorithmus. Zuletzt wird die Berechnung durchgeführt. Mithilfe von einigen Experimenten werden verschiedene Szenarien getestet und anschließend ausführlich analysiert. Insgesamt funktioniert die Approximation der Sterbewahrscheinlichkeit mithilfe eines künstlichen neuronalen Netzes außerordentlich gut. Ein Model mit vier verdeckten Schichten, insgesamt 640 Neuronen, dem Adam Optimierer und entweder der ELU, TanH oder Sofplus Aktivierungsfunktion liefert die weitaus besten Resultate für das Problem.

Schlagworte: Lebensversicherungsrisiko / Berechnung Sterbewahrscheinlichkeit / Sterbetafel / Künstliches neuronales Netz / Trainingsalgorithmus / Aktivierungsfunktion / Optimierer

Danksagung

Hiermit möchte ich mich bei allen bedanken, die mich während meines Studiums unterstützt haben und deren Beistand zur Fertigstellung dieser Diplomarbeit beigetragen hat.

Besonders bedanken möchte ich mich bei Prof. Dipl.-Math. Dr.rer.nat.Thorsten Rheinländer, welcher jederzeit für meine Anliegen da war. Durch seine ausgezeichnete Betreuung, konstruktive Kritik und zahlreichen Tipps hat er mich beim Verfassen dieser Arbeit tatkräftig unterstützt.

Ferner möchte ich meiner Familie meinen außerordentlich Dank aussprechen. Meine Eltern, meine Geschwister und mein Großvater haben mich durch das gesamte Studium begleitet und standen mir stets mit gutem Rat zur Seite.

Mein allergrößter Dank jedoch gebührt meiner Mutter, Michaela, die mich stets ermutigt hat und zu jeder Zeit für mich da war. Sie hat mir das Studium ermöglicht und mir dabei geholfen, das Ziel im Auge zu behalten. Besonders in schwierigeren Zeiten war sie für mich da, und ich werde ihr immer dafür dankbar sein.

Eidesstattliche Erklärung

Ich, Caroline Gerharter, BSc, erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Wien, 20. Oktober 2019

(Caroline Gerharter, BSc)

Contents

Chapter 1

Introduction

Over the last years, machine learning has become an immensely prominent subject. It is a significant part of artificial intelligence and deals with the analysis of data followed by prediction of future outcomes. In other words, it learns from given data, even though it is not explicitly programmed to do so.

This can be applied to various areas. Amongst many other topics it includes classification of objects, recognizing handwriting, email spam filtering, face detection or predicting tomorrow's stock market price by knowing current and historic market conditions and other side information [Mur12].

The idea behind machine learning was inspired by the activities of the human brain to solve quite complex problems.

Machine learning has been a research topic for many years. Nevertheless, it only recently became THE topic for researchers and companies, because of its requirement of a great deal of computational power. Previously computers were not nearly as powerful as they are now, so compiling a machine learning program was a major issue. However, computational power has vastly improved in the last few years, eliminating this shortcoming.

A kind of machine learning application is the use of multilayer perceptrons, which is a class of feedforward artificial neural networks. These perceptrons were introduced many years ago and important statements were discovered in 1989 by both Kurt Hornik and George Cybenko [HSW89, Cyb89]. Again, due to a lack of high computing capacity, the use of these multilayer perceptrons was not feasible until more recently.

Although machine learning can deliver great results, the calculation process is not easily comprehensible which might be an uncertainty issue, e.g., for auditors. Nevertheless, it is a great method to achieve results in a fast and efficient way. Many 'regular' statistical methods use up significantly more time to compute than machine learning and can thus not compete in today's environment.

Currently machine learning is not widely used by life insurance companies. Traditionally, they rely on actuarial calculations for prediction of premiums and mortality rates. Researchers do try to embed it in future calculations in order to improve accuracy, efficiency and speed of calculations. A growing number of insurers has also started to include predictive analysis in their portfolio. Still, more research is required for the field to mature.

In order to analyse risk profiles, insurers have to go through an underwriting process supported by specialists in this field. The customer, or potential policy holder, must go through various medical check ups and has to provide several documents to the insurance agent to be considered for an application. Afterwards, the underwriting process begins, where risks and premiums are calculated for the policy holder. For better determination, policy holders are divided into different risk groups, commonly known as risk classification. This underwriting process often takes over a month, involves various specialists and is very expensive [BJ18].

This process should be reduced by using machine learning for better and faster calculations.

1.1 Machine learning - definition of terms

Machine learning can be divided into different approaches, namely the predictive or supervised learning approach and the descriptive or unsupervised learning approach. There also is a less frequent approach - reinforcement learning.

In simple terms, supervised learning deals with pattern recognition between inputs and outputs. By contrast unsupervised learning only has input data and tries to find interesting patterns merely by analysing these inputs [Mur12].

1.1.1 Unsupervised learning

Unsupervised learning deals with the observation only of a random vector $\boldsymbol{x} \in \mathbb{R}^n$ without a target value $y \in \mathbb{R}$ and tries to learn the unconditional probability distribution $p(\boldsymbol{x})$ or find notable properties of the distribution. In this case the vectors $\boldsymbol{x}_i$ are features [GBC16].

A widely used application of unsupervised learning is clustering. Hereby, the goal is to divide data into different groups by identifying interesting patterns in the data [Mur12].

1.1.2 Supervised learning

In supervised learning several examples of a random vector $\boldsymbol{x} \in \mathbb{R}^n$ with its associated target value $y \in \mathbb{R}$ are observed and then learn to predict y from $\boldsymbol{x}$ generally through calculation of the conditional probability distribution $p(y|\boldsymbol{x})$.

Classification is one of the most used applications of supervised learning whereas the target value $y \in 1, \ldots, K$. K depicts the number of classes in the network. A famous and widely used classification problem is the Iris flower classification introduced by statistician Ronald Fisher. It includes four characteristics of the flower defining the vector $\boldsymbol{x}$ and classifies into three different flowers y via machine learning. To estimate the distribution $p(y|\boldsymbol{x})$ we can use maximum likelihood estimation to find the best parameter vector $\boldsymbol{\phi}$ for its family of distributions $p(y|\boldsymbol{x}; \boldsymbol{\phi})$. This concept of calculation is called logistic regression [GBC16].

The other main application is linear regression whereby, distinguished from classification, the target variable is continuous. The prediction of the life insurance risk parameter in this thesis forms a linear regression problem which is why we will focus on this problem from now on. To train the neural network it will be fed with a training sample, followed by testing on existing data. The training works through back propagation, which is a gradient descent method which we will illustrate in more detail later on.

Chapter 2

Mortality risk prediction

Predictive modelling has already existed for a while. It deals with analysing big data sets, followed by recognizing patterns in the data and using this information to predict future outcomes. In life insurance predictive modelling often entails usage of applications by means of life tables, which were first introduced by John Graunt and Edmund Halley already in the 17th century [BTK+10].

By now, the use of, e.g., Generalized Linear Models and Credibility Theory, also known as Empirical Bayes, which are forms of predictive modelling, are standardly used by insurance companies. However, the relatively new goal for life insurance companies is to expand predictive modelling, particularly with the assistance of machine learning techniques, in order to solve various problems and reform business processes.

The main goal of this thesis is to simulate the parameter q_x, which is the probability of someone dying between the ages x and $x+1$. My focus on the prediction lies on employees and workers with pension insurance. To predict this probability I have requested data described in the documentation *The Austrian Pension Insurance Table 2018-P (In german: AVÖ 2018-P: Rechnungsgrundlagen für die Pensionsversicherung)* [KHS18], from now on referred to only as 'AVÖ 2018-P'. This data package includes a life table which has been calculated with the help of 'non-machine learning' predictive modelling. I will use these calculations to compare the results and consequently minimize the error of the machine learning model I will later on describe in detail to calculate q_x.

I will explain the structure of the AVÖ 2018-P life table and calculations in the following sections.

2.1 Structure of life table

The variable q_x depicts an essential element of a life table and is used to calculate premiums for life insurances.

The outputs in the AVÖ 2018-P result from calculations based on data from the legal Austrian pension fund during the time period 2000-2017. The table depicts the probability of dying of employees in Austria. It includes a decomposition of data into white-collar workers and mixed stands. The mixed stands table contains both white-collar and blue-collar workers, which actually reflects the whole population of Austria overall quite well. The life table for white-collar workers shows a much lower probability of dying than the composed one.

There are also different tables for male, female and unisex population. Each table includes probabilities of dying regarding active population, invalidity pension, age pension and widow pension. I chose to include differentiations between active and invalidity pension whilst training and testing my model.

2.2 Base table

The first step for construction of the AVÖ 2018-P were calculations to generate a base table to later use as a reference calculator for other years.

The year chosen for the base table is 2008. To create the white-collar and blue-collar workers table, data from the 'Allgemeines Sozialversicherungsgesetz (ASVG) (engl.: General social security act) compulsory insured' was used.

2.3 Generation life tables

Since there is a still ongoing trend in mortality improvement, it is vital to work with generation life tables.

The probabilities in the AVÖ 2018-P table result from base year $t_0 = 2008$ with a mortality improvement trend ν_x. Therefore, the probability of dying of an x aged person in year t is given as

$$q_x(t) = q_x(t_0) \cdot exp(-\nu_x \cdot F(t - t_0)) \tag{2.1}$$

whereby the support function is given as

$$F(t) = \frac{1}{\lambda} \cdot arctan(\lambda \cdot t) \tag{2.2}$$

with trend reduction parameter $\lambda = 0.005$.

The half-life $t_{1/2}$ of the reduction, so the time in which the trend will reduce to a quantity half of its initial value, is equal to $\frac{1}{\nu}$ years. With λ being 0.005 the half-life $t_{1/2} = 200$. Therefore, the trend will reach half of its value after 200 years.

Chapter 3

Deep neural network

The model we are using to calculate the probability of dying q_x is an artificial neural network. This system was inspired by the functions of the human brain. It can detect outputs that are dependent on many different inputs.

The feed forward neural network we are using is a multilayer perceptron. This is one of the many applications of machine learning and notably one of the more popular ones.

There is also a single-layer perceptron, which only has an input and an output layer. An input node sends a weighted linear function directly to the output. Therefore, the possibilities of the perceptron are limited which is why we will use the superior multiple-layer perceptron.

It consists of three or more layers, namely input, hidden and output layer. The hidden layer is the intriguing part of a neural network and makes it a deep neural network. There often are multiple hidden layers in between input and output. Each of the hidden layer nodes uses a non-linear activation function. More precisely, those non-linear functions are logistic regression models [NT16].

The output layer uses either another logistic regression function or a linear regression function. Since the aim of this neural network is not to classify data which outputs a logistic regression function, we will focus on the regression problem with a linear regression function as the final outcome.

A deep neural network can be described as following [BGTW19]:

Definition 3.1 *Let $L, N_0, N_1, \dots, N_L \in \mathbb{N}$, activation function $\rho : \mathbb{R} \to \mathbb{R}$ and affine linear functions $W_\ell : \mathbb{R}^{N_{\ell-1}} \to \mathbb{R}^{N_\ell}$ for $\ell = 1, \dots, L$. Then a feed forward neural network $F : \mathbb{R}^{N_0} \to \mathbb{R}^{N_L}$ is given by*

$$F = W_L \circ F_{L-1} \circ \cdots \circ F_1 \text{ with } F_\ell = \rho \circ W_\ell \text{ for } \ell = 1, \dots, L-1$$

The activation function ρ is acting component-wise. $L+1$ stands for the number of layers. N_0 denotes the dimension of the input layer, whereas N_L that of the output layer and $N_1, \dots, N_{L-1}$ are the dimensions of the $L-1$ hidden layers.

The neural network is a weighted acyclic directed graph with L layers that are filled with nodes whereby edges only exist between adjacent layers, depicted in Figure 3.1. The nodes in the network are called neurons.
The affine function W_ℓ is defined by a matrix $A \in \mathbb{R}^{N_\ell \times N_{\ell-1}}$ and an affine part $t \in \mathbb{R}^n$ by the function $W_\ell(x) = A_\ell x + t_\ell$ for any $\ell = 1, \dots, L$. For any $i = 1, \dots, N_{\ell-1}$ and

$j = 1, \ldots, N_\ell$ the matrix $(A_\ell)_{i,j}$ depicts the weight of the edge which connects node i in layer $\ell - 1$ with node j in layer ℓ. The sum of N_ℓ given by $N := \sum_{i=1}^{L} N_i$ makes up the total number of neurons.

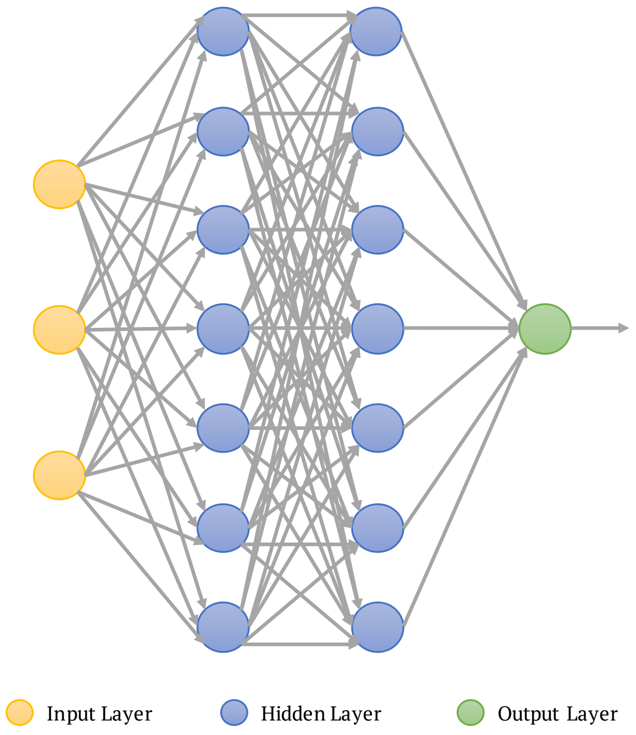

Figure 3.1: Structure of a fully connected multilayer perceptron

A neural network can be fully connected or only partially. Partial connection means that not all neurons from layer ℓ are connected with all nodes in layer $\ell + 1$. Then the network has so called sparse connectivity [BGKP19], shown in figure 3.2.

The set of neural networks from $\mathbb{R}^{d_0} \to \mathbb{R}^{d_1}$ with activation function ρ is denoted by $\mathcal{N}^{\rho}_{\infty,d_0,d_1}$. If we only choose a sequence of subset of $\mathcal{N}^{\rho}_{\infty,d_0,d_1}$, then this sequence is denoted by $\{\mathcal{N}^{\rho}_{K,d_0,d_1}\}_{K\in\mathbb{N}}$ whereby the following properties hold:

- $\bigcup\limits_{K\in\mathbb{N}} \mathcal{N}^{\rho}_{K,d_0,d_1} = \mathcal{N}^{\rho}_{\infty,d_0,d_1}$
- $\mathcal{N}^{\rho}_{K-1,d_0,d_1} \subset \mathcal{N}^{\rho}_{K,d_0,d_1} \forall K \in \mathbb{N}.$

Next, we will introduce an important theorem about $\mathcal{N}^{\rho}_{\infty,d_0,d_1}$.

Theorem 3.2 *Let ρ be bounded and non-constant, then for any finite measure μ on $(\mathbb{R}^{d_0}, \mathcal{B}(\mathbb{R}^{d_0}))$ and $1 \le p < \infty$, the set $\mathcal{N}^{\rho}_{\infty,d_0,1}$ is dense in $L^p(\mathbb{R}^{d_0}, \mu)$.*

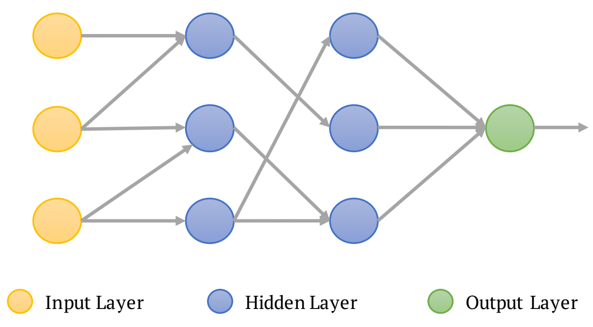

Figure 3.2: Structure of a sparsely connected multilayer perceptron

Theorem 3.3 *If the conditions from the previous theorem hold and additionally $\rho \in C(\mathbb{R})$, then $\mathcal{N}^{\rho}_{\infty,d_0,1}$ is dense in $C(\mathbb{R}^{d_0})$ for the topology of uniform convergence on compact sets.*

Since each component of an $\mathbb{R}^{d_1}$-valued neural network is an $\mathbb{R}$-valued neural network, Theorem 3.2 and 3.3 even hold for $\mathcal{N}^{\rho}_{\infty,d_0,d_1}$ with $d_1 > 1$ [BGTW19, Hor91].

3.1 Backpropagation

A multilayer perceptron is assigned random weights in its first layer. Then it calculates the error. In the following, it propagates through all hidden layers until the first layer. There it will adapt the weights to reduce the error. The process will be repeated several times. This propagation algorithm is called backpropagation [GP17].

For better understanding of the backpropagation algorithm we will first consider a network with no hidden units and one output unit. Therefore, the described network is a single layer perceptron which uses a more special rule for gradient descent than the backpropagation algorithm, namely the delta rule, often also known as Least-Mean-Squares (LMS) algorithm [AZ14].

3.1.1 Delta Rule

We will limit the network to having a linear activation function. Then the output of the network is the weighted sum of its input

$$\hat{y} = \sum_i w_i x_i + \theta$$

with bias term θ. The error function of this problem with predicted output $\hat{y}$ and target output y is given by

$$E = \frac{1}{2} \sum_r (y_r - \hat{y}_r)^2 \tag{3.1}$$

with $r \in \{1, \dots, n\}$ describing n different training patterns. Therefore, this problem can be seen as an optimisation problem, whereby the goal is to find the weights which reduce the error function. To solve it the method of gradient descent is used. Gradient descent means adjustment of the weight w_i by a value Δw_i which is proportional to the negative value of the partial derivative of the error corresponding to each weight. Thus, for weight i and learning pattern r it is given as

$$\Delta_r w_i = -\nu \frac{\partial E_r}{\partial w_i} \tag{3.2}$$

whereby ν is the learning rate. We will now substitute E_r by inserting formula 3.1 for the rth learning pattern and then applying the chain rule to split the derivative

$$\frac{\partial E_r}{\partial w_i} = \frac{\partial(\frac{1}{2}(y_r - \hat{y}_r)^2)}{\partial \hat{y}_r} \frac{\partial \hat{y}_r}{\partial w_i}$$

The left derivative is

$$\frac{\partial(\frac{1}{2}(y_r - \hat{y}_r)^2)}{\partial \hat{y}_r} = -(y_r - \hat{y}_r) = \sigma_r \tag{3.3}$$

whereby σ_r is the difference between the target output and the predicted output for learning pattern r. Since we have a linear output unit the right derivative results in

$$\frac{\partial \hat{y}_r}{\partial w_i} = x_{ri}$$

which is the corresponding input to the rth weight and the jth training pattern. Assembling both derivatives we obtain

$$\frac{\partial E_r}{\partial w_i} = \sigma_r x_{ri} \tag{3.4}$$

Finally inserting 3.4 into 3.2 we obtain

$$\Delta_r w_i = -\nu \sigma_r x_{ri} \tag{3.5}$$

which defines the delta rule. Since this method is not very complex it is widely used. However, it is not applicable on a multilayer perceptron due to not knowing each neurons' output exactly. Therefore, we will focus on the more general backpropagation algortihm.

3.1.2 Backpropagation for multilayer perceptrons

Backpropagation [AZ14] for a feedforward neural network with at least one hidden layer is just a more complex process of applying the delta rule recursively. At first, the error of the output layer is calculated in the same way as the delta rule. Following, it is used to calculate the error from the last hidden layer and then repeating the process recursively for each hidden layer until all errors are estimated. In the next step the weights are altered with a process similar to the delta rule. This procedure is done several times until a predefined value for the error is reached.

Applying the delta rule to a single layer perceptron, we can be certain to find a global minimum, since the error surface has a convex shape. However, for the multilayer perceptron this statement does not hold, since there is no unique minimum and it might happen to only obtain a local minimum.

Illustrating the more general backpropagation algorithm we will now consider a multilayer perceptron with a non-linear activation function ρ. The output function b_{rk} for each neuron k is defined as

$$b_{rk} = \rho(a_{rk}) = \rho\left(\sum_l w_{lk} b_{rl}\right) \tag{3.6}$$

The activation function ρ is non-linear and differentiable. Often the sigmoid function serves as the activation function. It is given by

$$\rho(x) = \frac{1}{1+e^{-x}} \tag{3.7}$$

It is quite practical, since the derivative is

$$\frac{d\rho(x)}{dx} = \rho(x)(1-\rho(x)) \tag{3.8}$$

To apply the gradient descent method we will first use the chain rule to calculate the partial derivative

$$\frac{\partial E_r}{\partial w_{ik}} = \frac{\partial E_r}{\partial a_{rk}} \frac{\partial a_{rk}}{\partial w_{ik}} \tag{3.9}$$

We can easily solve the right derivative

$$\frac{\partial a_{rk}}{\partial w_{ik}} = \frac{\partial}{\partial w_{ik}}\left(\sum_l w_{lk} b_{rl}\right) = \frac{\partial w_{ik} b_{ri}}{\partial w_{ik}} = b_{ri} \tag{3.10}$$

For the left derivative we apply the chain rule again to obtain

$$\frac{\partial E_r}{\partial a_{rk}} = \frac{\partial E_r}{\partial b_{rk}} \frac{\partial b_{rk}}{\partial a_{rk}} \tag{3.11}$$

The right derivative is

$$\frac{\partial b_{rk}}{\partial a_{rk}} = \frac{\partial \rho(a_{rk})}{\partial a_{rk}} = \rho'(a_{rk}) \tag{3.12}$$

straightforward corresponding to the derivative of the activation function. When calculating the left derivative we have to differentiate between two cases. Either b_{rk} corresponds to a neuron in the output layer, or to one in a hidden layer. If we use the squared error as the error measurement, then, for the first case, the derivative is equivalent to formula 3.3 ($\sigma_r = -(y_r - \hat{y}_r)$), since $b_{rk} = \hat{y}$. The other case is a little more complex. We apply the chain rule to the derivative

$$\frac{\partial E_r}{\partial b_r} = \sum_l \frac{\partial E_r}{\partial a_{rl}} \frac{\partial a_{rl}}{\partial b_r}$$

When substituting 3.6 in the right derivative we obtain

$$\frac{\partial E_r}{\partial b_r} = \sum_l \frac{\partial E_r}{\partial a_{rl}} \frac{\partial}{\partial b_r} \left(\sum_k w_{kl} b_{rk} \right) = \sum_l \frac{\partial E_r}{\partial a_{rl}} w_{kl} \tag{3.13}$$

where the derivative $\partial E_r \setminus \partial a_{rl}$ is the same as in 3.11. We will now substitute 3.10, 3.11, 3.12 and 3.13 in formula 3.9, so we obtain

$$\frac{\partial E_r}{\partial w_{ik}} = b_{ri} \cdot \sigma_{rk}$$

with

$$\sigma_{rk} = \frac{\partial E_r}{\partial b_{rk}} \frac{\partial b_{rk}}{\partial a_{rk}} = \begin{cases} (b_{rk} - y_{rk}) \rho'(a_{rk}) & \text{if } k \text{ is an output neuron} \\ \left(\sum_l \sigma_{rl} w_{kl}\right) \rho'(a_{rk}) & \text{if } k \text{ is an inner neuron.} \end{cases}$$

Finally, we will use the before proposed sigmoid activation function to show what the derivative looks like in a practical case. We already defined the function and its derivative in 3.7 and 3.8. So we obtain

$$\sigma_{rk} = \begin{cases} (b_{rk} - y_{rk}) b_{rk}(1 - b_{rk}) & \text{if } k \text{ is an output neuron} \\ \left(\sum_l \sigma_{rl} w_{kl}\right) b_{rk}(1 - b_{rk}) & \text{if } k \text{ is an inner neuron.} \end{cases}$$

Since we want to minimize the error with gradient descent, we have to change the weights with a learning rate ν. In order to find a better global minimum the algorithm changes the weights during each iteration. The equation for the change of each weight, similarly to equation 3.5, has the form

$$\Delta_r w_{rik} = -\nu \frac{\partial E_r}{\partial w_{rik}} = -\nu \cdot b_{ri} \cdot \sigma_{rk}$$

Therefore, the learning rule is

$$w = w - \nu \frac{\partial E}{\partial w} = w - \nu \nabla E(w) \tag{3.14}$$

3.1.3 Learning rate

Having a constant learning rate ν is generally not advisable. Usually, it poses a problem for both low and high learning rates. If it is low, the process of finding the optimal solution takes a long time. If it is high, it might work well in the beginning, but could very well oscillate around a point for a while or diverge unstably [Agg18]. Therefore, many optimizers use a variable learning rate. We will focus on different optimizers in section 3.3.

Having already introduced the sigmoid activation function we will now illustrate some other ones.

3.2 Activation

The activation function ρ maps from a node in layer $\ell - 1$ to a node in layer ℓ. Hereby, the resulting values will be in a desired range, depending on the chosen activation function.

There are several activation functions now used in neural networks. The one currently most widely used is the rectified linear unit (ReLU) [GP17]. It is important to use the right activation function for the multilayer perceptron so I will offer a brief insight on some popular ones and ultimately those I tested in my model.

The first activation function I want to introduce is the sigmoid, which is referring to the special case of the logistic function.

Definition 3.4 ***(Sigmoid)*** *(Figure 3.3) Let D be the set of input data and $\rho : D \to (0,1)$ a C^∞ function, then the sigmoid is given as*

$$\rho(x) = \sigma(x) = \frac{1}{1+e^{-x}}$$

The derivative is given as

$$\rho'(x) = \rho(x)(1 - \rho(x))$$

The hard sigmoid [ten19] is a piecewise linear approximation of the logistic function.

Definition 3.5 ***(Hard Sigmoid)*** *(Figure 3.4) Let D be the set of input data and $\rho : D \to (0,1)$ a C^∞ function, then the hard sigmoid is given as*

$$\rho(x) = \begin{cases} 0 & x < -2.5 \\ 0.2x + 0.5 & -2.5 \leq x \leq 2.5 \\ 1 & x > 2.5. \end{cases}$$

The derivative is given as

$$\rho'(x) = \begin{cases} 0.2 & x < -2.5 \leq x \leq 2.5 \\ 0 & else. \end{cases}$$

Another sigmoidal function is the Hyperbolic tangent (TanH) [KO11] which is also continuously differentiable.

Definition 3.6 ***(Hyperbolic tangent (TanH))*** *(Figure 3.5) Let D be the set of input data and $\rho : D \to (-1,1)$ a C^∞ function, then TanH is given as*

$$\rho(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

The derivative is given as

$$\rho'(x) = 1 - \rho(x)^2$$

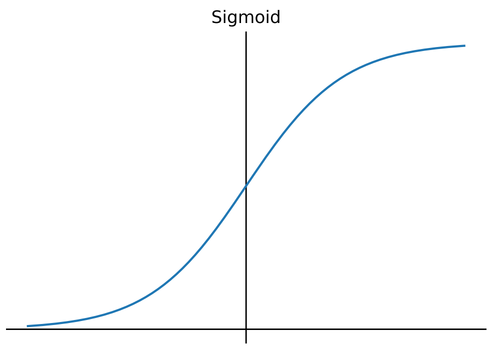

Figure 3.3: Sigmoid

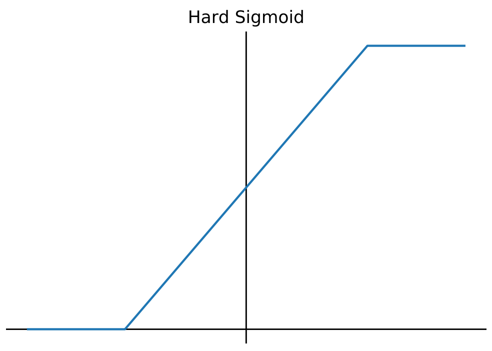

Figure 3.4: Hard Sigmoid

Definition 3.7 ***(Softsign)*** *(Figure 3.6) Let D be the set of input data and $\rho : D \to (-1, 1)$ a C^1 function, then Softsign is given as*

$$\rho(x) = \frac{x}{1 + |x|}$$

The derivative is given as

$$\rho'(x) = \frac{1}{(1 + |x|)^2}$$

Definition 3.8 ***(Rectified Linear Unit (ReLU))*** *(Figure 3.7) Let D be the set of input data and $\rho : D \to [0, \infty)$ a C^0 function, then the ReLU is given as*

$$\rho(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0. \end{cases}$$

The ReLU [LY17] is very popular despite its shortcoming that it is continuous, but not continuously differentiable. The derivative is given as

$$\rho'(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0. \end{cases}$$

The next activation function we will consider is the exponential linear unit (ELU) [RZL17].

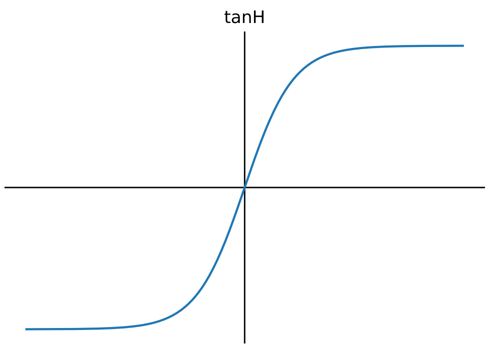

Figure 3.5: Hyperbolic tangent

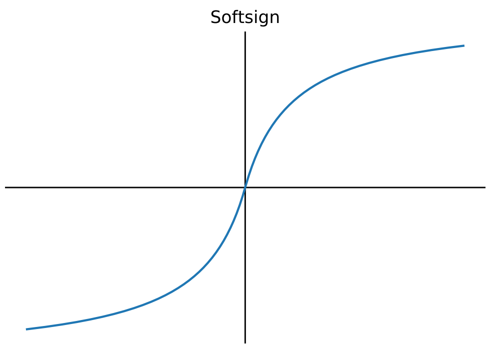

Figure 3.6: Softsign

Definition 3.9 ***(Exponential Linear Unit (ELU))*** *(Figure 3.8) Let D be the set of input data and $\rho : D \to (-a, \infty)$, then the ELU is given as*

$$\rho(a, x) = \begin{cases} a(e^x - 1) & x \leq 0 \\ x & x > 0. \end{cases}$$

For $a = 1$ ρ is in C^1 otherwise in C^0.

The derivative is given as

$$\rho'(a, x) = \begin{cases} \rho(a, x) + a & x \leq 0 \\ 1 & x > 0. \end{cases}$$

Definition 3.10 ***(Scaled exponential linear unit (SELU))*** *Let D be the set of input data and $\rho : D \to (-\lambda a, \infty)$ a C^0 function, then the SELU is given as*

$$\rho(a, x) = \lambda \begin{cases} a(e^x - 1) & x < 0 \\ x & x \geq 0. \end{cases}$$

with $\lambda = 1.0507$ and $\alpha = 1.67326$.

The derivative is given as

$$\rho'(a, x) = \lambda \begin{cases} ae^x & x < 0 \\ 1 & x \geq 0. \end{cases}$$

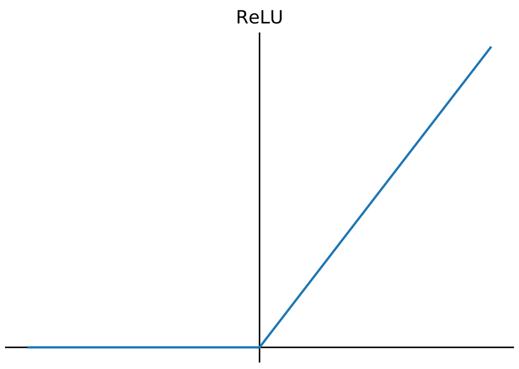

Figure 3.7: Rectified linear unit

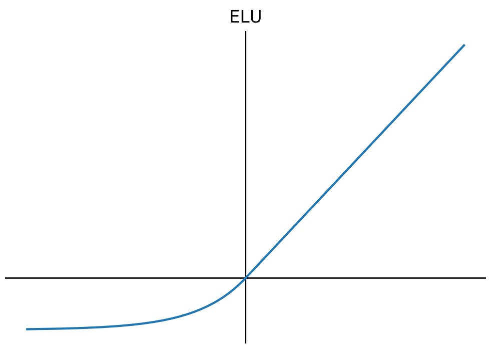

Figure 3.8: Exponential linear unit

The SELU is evidently simply a scaled modification of the ELU.

Definition 3.11 ***(Softplus)*** *(Figure 3.9) Let D be the set of input data and $\rho : D \to (0, \infty)$ a C^∞ function, then the softplus is given as*

$$\rho(x) = ln(1 + e^x)$$

The derivative is given as

$$\rho'(x) = \frac{1}{1 + e^{-x}}$$

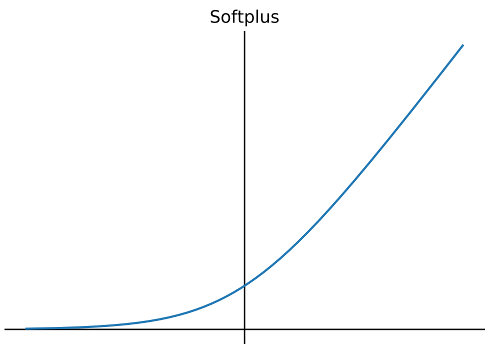

Figure 3.9: Softplus

3.3 Optimisation strategies

As explained in section 3.1 optimisation is the process of changing weights in the neural network in order to find strategies with lower errors after each iteration.

Since the process of finding the optimal solution might end in oscillation, it is often advisable to use optimisation other than stochastic gradient descent (SGD). There are many optimisers inlcuding slight modifications of the basic SGD which for example include momentum [Agg18], that yield far better results. Hence, I want to describe Momentum before introducing the different optimisers.

3.3.1 Momentum

When using momentum the learning rule differs from equation 3.14, namely with a different update rule for its weights w

$$w = w + \Delta w$$

while Δw is given as

$$\Delta w = \mu \Delta w - \nu \nabla E(w)$$

with error function $E(w)$, learning rate ν and smoothing parameter $\mu \in (0, 1)$, often also called the momentum parameter.

This process accelerates the learning, since it helps the algorithm to go in the correct direction. It is supposed to make the algorithm rather go in the consistent direction over multiple steps during the gradient descent, than to unnecessarily oscillate around a point. Therefore, gradient descent with momentum will reach the optimal solution sooner.

A slight adjustment to the regular momentum presents the Nesterov momentum. The difference between the two lies in where the gradient is computed.

$$\Delta w = \mu \Delta w - \nu \frac{\partial E(w + \mu \Delta w)}{\partial w}$$

Thus, the gradient already includes the update parameter Δw with its smoothing parameter μ. Those parameters have better comprehension of the change of gradients, so this incorporation might lead to even faster convergence.

Now, we will introduce a few optimizers and the differences among them.

3.3.2 Adagrad

The first optimiser, besides the basic SGD, we want to introduce is AdaGrad (adaptive gradient algorithm) [DHS11]. The purpose of the AdaGrad optimiser is to reduce the learning rate, if the gradient change is small and increase the learning rate if the gradient change is large. The small steps prevent the algorithm to jump over the optimum. All things considered, this algorithm makes the neural network find its optimum faster.

To define the learning rule for the AdaGrad optimiser we have to define a few auxiliary variables first. Lets say the gradient of the error at iteration t with respect to a parameter

w_i is given as $g_i^{(t)} = \nabla E^{(t)}(w_i)$. The outer product matrix is then given as

$$G^{(t)} = \sum_{\tau=1}^{t} g^{(\tau)} g^{(\tau)\top}$$

We need the diagonal of the outer product matrix $G^{(t)}$ to define the learning rule. The diagonal elements $G_{ii}^{(t)}$ of the matrix represent the sum of the squares of the gradients with respect to w_i until time step t. The learning rule then is defined as following

$$w_i^{(t+1)} = w_i^{(t)} - \frac{\nu}{\sqrt{G_{ii}^{(t)}} - \epsilon} \cdot g_i^{(t)} \tag{3.15}$$

where ϵ is simply a term to prevent from division by zero.

The major problem that the AdaGrad optimiser has is the way the learning rule is defined. Since $G_{ii}^{(t)}$ in the denominator in formula 3.15 denotes the squared gradients, the value is always positive. Hence, each iteration $G_{ii}^{(t)}$ increases, which leads to a smaller and smaller learning rule. A critical point will be reached when the learning rate is so small, that it will stop learning overall.

An improved version of the Adagrad optimiser is the Adadelta [Zei12].

3.3.3 Adadelta

The Adadelta fixes the AdaGrad's main issue, which reaches an infinitesimally small learning rate, by limiting the number of accumulated past gradients to a fixed size m, instead of summing up until iteration t.

For the method to be practical, Adadelta does not just accumulate the m previous squared gradients, but uses exponentially decaying average of the squared gradients. With a decay constant γ we declare this moving average as $M^{(t)}[g^2]$ which can be calculated as following

$$M^{(t)}[g^2] = \gamma M^{(t)}[g^2] + (1-\gamma)(g^2)^{(t)}$$

To now update the learning rule of the AdaGrad optimiser (formula 3.15) we need the square root of $M^{(t)}[g^2]$ which becomes

$$RMS^{(t)}[g] = \sqrt{M^{(t)}[g^2] + \epsilon}$$

The term ϵ serves as a parameter to prevent from division by zero. Collectively the parameter update is

$$\Delta w^{(t)} = -\frac{\nu}{RMS^{(t)}[g]} \cdot g^{(t)}$$

Next, we need to define another exponentially decaying average, since the units of $w^{(t)}$ and $\Delta w^{(t)}$ do not match. This moving average is then defined as

$$M^{(t)}[\Delta w^2] = \gamma M^{(t)}[\Delta w^2] + (1-\gamma)(\Delta w^{(t)})^2$$

The square root of $M^{(t)}[\Delta w^2]$ is given as

$$RMS^{(t)}[\Delta w] = \sqrt{M^{(t)}[\Delta w^2] + \epsilon}$$

Under assumption of local smoothness of the curvature of $\Delta w^{(t)}$ and, since we do not know its value at time step t, we approximate it by computing the RMS until the previous time step. This yields the parameter update

$$\Delta w^{(t)} = -\frac{RMS^{(t-1)}[\Delta w]}{RMS^{(t)}[g]} \cdot g^{(t)}$$

with lerning rule

$$w^{(t+1)} = w^{(t)} + \Delta w^{(t)}$$

Another optimiser which had the aim to eliminate AdaGrad's flaws is the RMSProp.

3.3.4 RMSProp

The RMSProp (Root Mean Square Propagation) [HSS12, Rud16] is a hitherto unpublished optimiser by Geoffrey Hinton.

It uses exponential averaging to update its weights. This means, the learning rate is adapted for each iteration by dividing the learning rate with the square root of the exponentially averaged value of w. Therefore the moving average is the same as for the Adadelta optimiser. We introduce an $m^{(t)} := M^{(t)}[g^2]$, so we obtain

$$m^{(t)} := \gamma m^{(t-1)} + (1-\gamma)(g^2)^{(t)} \tag{3.16}$$

with a decay factor $\gamma \in (0, 1)$. This is evidently the same update vector as for Adadelta.

Hinton suggests to set $\gamma = 0.9$ and the learning rate $\nu = 0.001$. Therefore, we obtain the learning rule

$$w^{(t+1)} = w^{(t)} - \nu \frac{g^{(t)}}{\sqrt{m^{(t)} + \epsilon}}.$$

Combining RMSProp and Momentum yields the Adam optimizer.

3.3.5 Adam

Adam (Adaptive Moment Estimation) [KB14] is currently the most used optimizer for neural networks.

It not only uses the exponentially decaying average of the past squared gradients, but also of the past gradients. Hence, as for Adadelta and RMSprop we have

$$m^{(t)} := \gamma m^{(t-1)} + (1-\gamma)(g^2)^{(t)}$$

In addition to this we introduce a parameter n, which is an exponentially smoothed value of the gradient. The smoothing of n is a modification of the momentum method and is given as

$$n^{(t)} := \delta n^{(t-1)} + (1-\delta)(g^2)^{(t)}$$

The bias-corrected second raw moment estimate is

$$\hat{m}^{(t)} = \frac{m^{(t)}}{1-\gamma^t}$$

and the bias-corrected first moment estimate is

$$\hat{n}^{(t)} = \frac{n^{(t)}}{1-\delta^t}$$

Then the tth iteration of the update with learning rate ν is given as

$$w^{(t+1)} = w^{(t)} - \nu \frac{n}{\sqrt{m} + \epsilon}$$

where ϵ is a small scalar to prevent division by zero.

The default setting of the Adam parameters for machine learning methods is

- $\nu = 0.001$
- $\gamma = 0.9$
- $\delta = 0.999$
- $\epsilon = 10^{-8}$

Another optimizer closely related to Adam and RMSProp is Nadam.

3.3.6 Nadam

Nadam (Nesterov ac-cellarated Adaptive Moment Estimation) [Her16, Doz16] operates as the name already reveals like the Adam optimizer, but with Nesterov momentum. Therefore, the moments are given as

$$\hat{m}^{(t)} = \frac{\gamma m^{(t)}}{1-\prod_{i=1}^{t+1}\delta^{(i)}} + \frac{(1-\delta^{(t)})\nabla E^{(t)}(w)}{1-\prod_{i=1}^{t}\delta^{(i)}}$$

and

$$\hat{n}^{(t)} = \frac{\delta^{(t+1)} n^{(t)}}{1-\delta^t}$$

The default settings of the Nadam optimizer are usually the same as for Adam, but with learning rate

- $\nu = 0.002$

which anyway is an adaptable factor.

3.4 Error

When performing a linear regression problem a common measurement of error is the mean squared error (MSE) [GBC16]. Suppose we have a train set of size m with the target vector y_{train} and a test set of size n with target vector y_{test}. Then the machine learning algorithm will be trained and try to predict a vector $\hat{y}_{test}$ for the real vector y_{test}. The MSE is then given by

$$MSE = \frac{1}{n}\sum_{j=1}^{n}(\hat{y}_{test} - y_{test})_j^2.$$

Chapter 4

Preparation of model and experiments

To create a deep neural network I used *Python* with the Keras tool.

To train my model I had to prepare the data derived from AVÖ 2018-P properly. Since the data I requested from the AVÖ 2018-P table is only for scientific use, including calculations for my model, and I am not permitted to publish it, I did not release any values for the parameter q_x.

4.1 Data preparation

At first I calculated probabilities of dying for several years and after various tests chose to focus on training data from the years 1986 to 2000 and testing data from year 2016. Since the probability of dying for a 120 year old person will be set to 1 by default I decided to leave out age 120, while training my model.

In the pension insurance data table AVÖ 2018-P there are calculations for people beginning with age 14, since it does not make sense to calculate probabilities of dying for pensions for younger children than that. Therefore, the starting age in my model is 14.

The final matrix that I used for training the machine learning algorithm looked as following, whereby the q_x^R represent the values I calculated with the AVÖ 2018-P table.

Beginning of Table

Age	**Year of Death**	**State**	$\boldsymbol{q_x^R}$
14	1986	1	q_{14}^{1986}
15	1986	⋮	q_{15}^{1986}
⋮	⋮		⋮
119	1986		q_{119}^{1986}
14	1987		q_{14}^{1987}
15	1987		q_{15}^{1987}
⋮	⋮		⋮
119	1987		q_{119}^{1987}
⋮	⋮		⋮
⋮	⋮		⋮
14	2000		q_{14}^{2000}

Continuation of Table 4.1

Age	Year of Death	State	q_x^R
15	2000		q_{15}^{2000}
$\vdots$	$\vdots$		$\vdots$
119	2000		q_{119}^{2000}
14	1986	2	q_{14}^{1986}
15	1986	$\vdots$	q_{15}^{1986}
$\vdots$	$\vdots$		$\vdots$
119	1986		q_{119}^{1986}
14	1987		q_{14}^{1987}
15	1987		q_{15}^{1987}
$\vdots$	$\vdots$		$\vdots$
119	1987		q_{119}^{1987}
$\vdots$	$\vdots$		$\vdots$
$\vdots$	$\vdots$		$\vdots$
14	2000		q_{14}^{2000}
15	2000		q_{15}^{2000}
$\vdots$	$\vdots$		$\vdots$
119	2000		q_{119}^{2000}

End of Table

Table 4.1: Data Preparation for Python

The variables in column *State* are 1 for active population, which is the same as old-age pension and 2 for invalidity pension. I made the same data preparation for year 2016 to compare the predicted data with the expected output.

Since the values from the input data vary widely, I normalized it. The normalization helps the network to operate faster and produce better results.

Ideally, the model should have a low mean squared error and be fast. To receive best results I tested various scenarios, which I will describe in the following sections.

4.2 Experiment 1 - number of layers vs. number of nodes

In my first experiment I focused on figuring out how a low amount of layers with a high amount of nodes and vice versa affected the neural network.

To receive meaningful results, I solely used the Adam optimiser and fixed the activation function. Further, I let the algorithm run through 800 epochs each time.

At first I focused on finding out how many neurons altogether where needed for the network to work properly. If there are too little neurons in the network the predicted values are just an average of what the values should be and do not represent the data properly. On the other hand, if there is a very high amount of neurons, the network takes a lot of time and might even overfit the data. After balancing those shortcomings I came to the conclusion to train my network with overall 640 neurons.

With all these initial conditions I finally examined the effects the number of nodes and number of layers had on the network. After several tests I came to the conclusion that it is best to have at least four hidden layers, since the mean squared error was rather high with less layers. Moreover, q_x for invalidity pension is, after slightly decreasing in the beginning, not strictly increasing over age like the q_x for active population, but has a local maximum at age 54 and then strictly decreases until reaching a local minimum at age 61. This drop in q_x also has to be represented by the trained data which usually only is the case when using at least four hidden layers. Otherwise the network might not identify the curve and the result might look like graph 4.1.

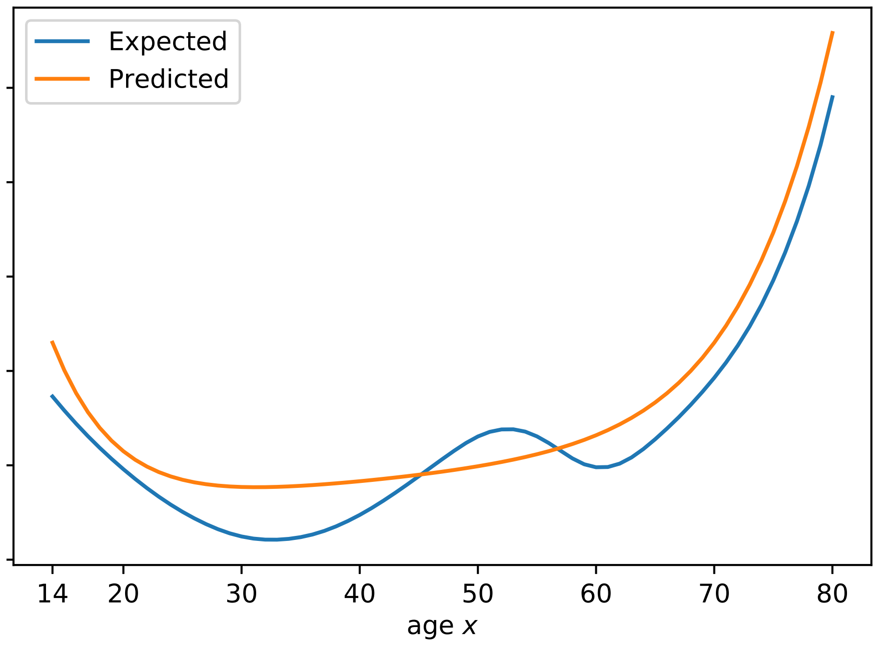

Figure 4.1: NN invalidity pension, ages 14 to 80, two hidden layers

After testing various scenarios I came to the conclusion that it is the best choice to use four hidden layers. This has two reasons, on the one hand the MSE does not really improve for more layers and on the other hand the network needs much more time to compile, while using more layers and less nodes.

When training the model with 512 nodes in its first layer and four hidden layers with each 32 nodes I received excellent results, shown in Figures 4.2, 4.3 and 4.4.

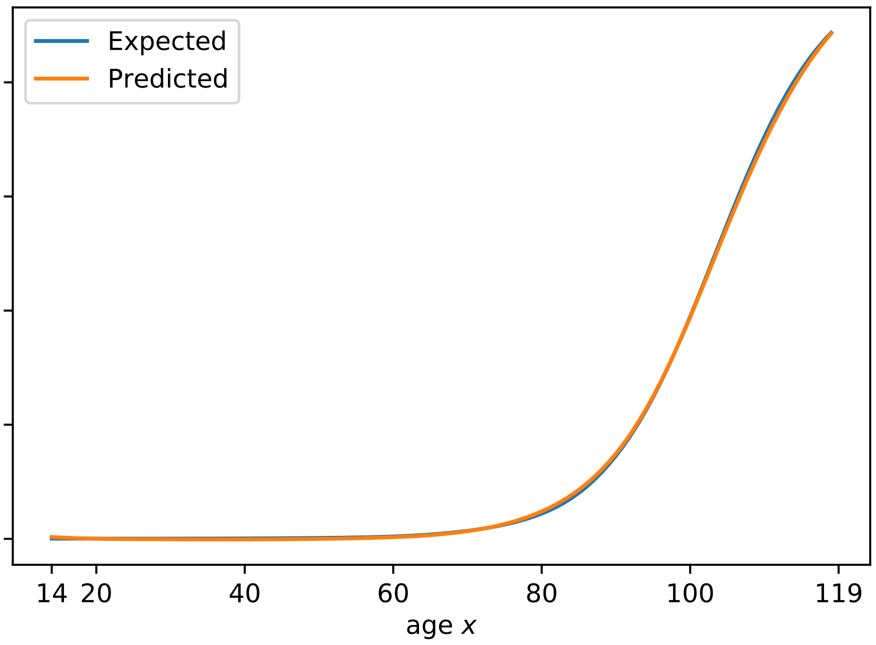

Figure 4.2: NN old-age pension, four hidden layers

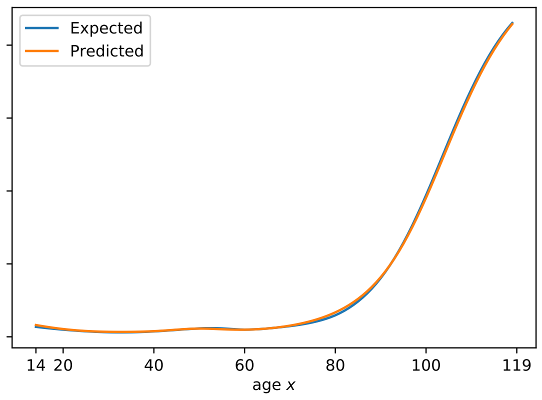

Figure 4.3: NN invalidity pension, four hidden layers

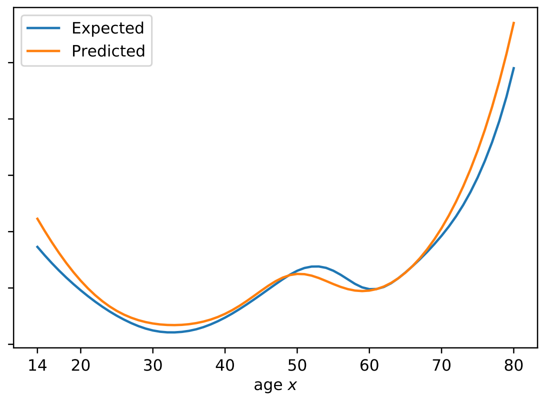

Figure 4.4: NN invalidity pension, ages 14 to 80, four hidden layers

4.3 Experiment 2 - activation functions

Experiment 2 focuses on testing various activation functions. I created a network containing four hidden layers and used the Adam optimiser. I also fixed the number of nodes in each layer to obtain comparable results. After testing each activation function five times and taking an average over the mean squared error I obtained results shown in table 4.2.

Table 4.2: Comparison of different activation functions with MSE value

Activation Function	**Mean Squared Error**
Sigmoid	5.55×10^{-5}
Hard Sigmoid	2.72×10^{-2}
Hyperbolic Tangent (TanH)	2.85×10^{-5}
Softsign	1.42×10^{-4}
Rectified Linear Unit (ReLU)	1.48×10^{-4}
Exponential Linear Unit (ELU)	3.30×10^{-5}
Scaled Exponential Linear Unit (SELU)	7.52×10^{-5}
Softplus	2.59×10^{-5}

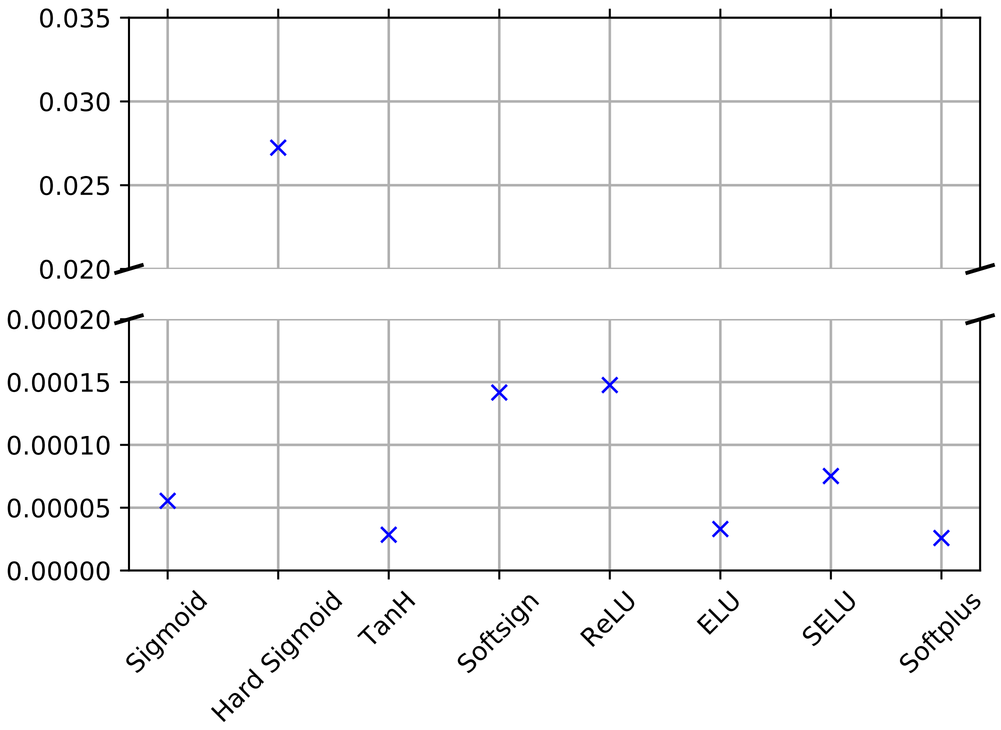

Figure 4.5: Comparison of different activation functions with MSE value

4.3.1 Sigmoidal functions

To analyse the results from table 4.2 the activation functions should be split up into two categories. On the one hand we consider sigmoidal functions. In that category belongs the sigmoid, the hard sigmoid, the hyperbolic tangent and the softsign. The other type of activation functions are rectifiers. Those include the rectified linear unit, the exponential linear unit, the scaled exponential linear unit and the softplus.

At first we consider the sigmoid activation function. It is very popular for simple problems, but suffers from the vanishing gradient problem [GHV17] which leads to poorer results. The vanishing gradient problem concerns the backpropagation algorithm and how it operates. Since it uses gradients to change its weights to optimize the algorithm it is quite impractical that the gradients for the sigmoidal function can be vanishingly small. Due to the fact, that the range of the sigmoid function is between 0 and 1, and the derivation uses the chain rule, it leads to multiplying very small numbers with each other, while following the backpropagation algorithm. As a result this might lead to obtaining vanishingly small numbers, so that the weight will not be changed at all and, therefore, might stop the backpropagation algorithm altogether.

The same problem does apply to all of the sigmoidal functions. Nevertheless, it has less impact on the hyperbolic tangent, than on the sigmoid. Hence, the hyperbolic tangent is a far better choice of sigmoidal activation functions, than the sigmoid function. As we can see in figure 4.5 the sigmoid activation function did not yield terrible results, but does not belong with the three activation functions that yielded best results either.

Then again, the rectifiers do not have the shortcoming of suffering under vanishing gradient problem at all, which is why their overall performance was better than of the sigmoidal activation functions.

The fact that the network did not work at all with the hard sigmoid activation function was to be expected, since it is only an approximation of the sigmoid. This means it works faster, but at the same time it is less precise and the error is bound to be higher. Therefore, it is not a good choice for regression problems, but might be fine to be used for classification tasks due to the calculation speed.

An example of a result of the neural network with a hard sigmoid activation function with a MSE of 1.49×10^{-04} is shown in Figures 4.6, 4.7 and 4.8. Evidently the predicted outputs are just piecewise linear functions, which shows the calculation for q_x did not work well at all.

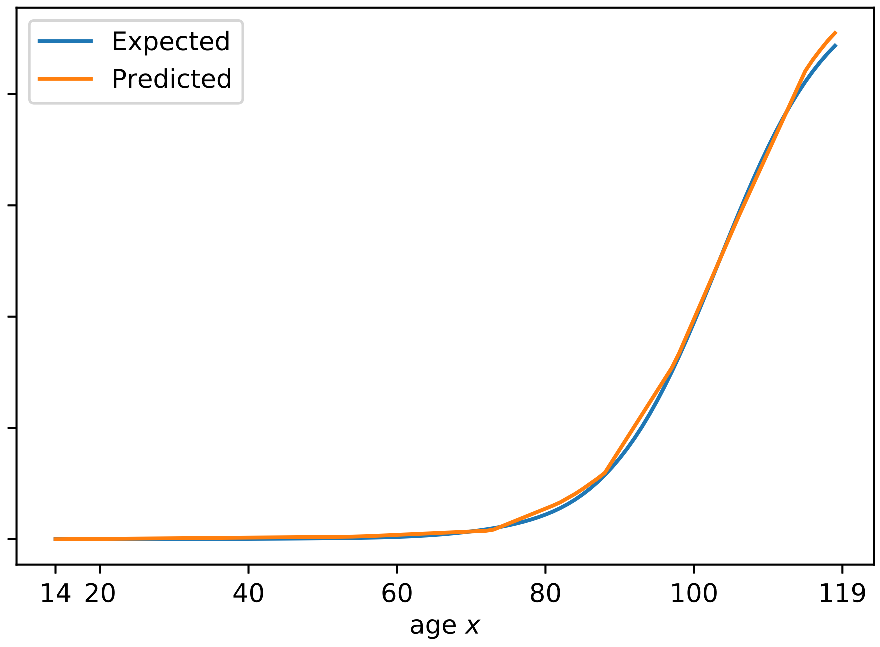

Figure 4.6: NN old-age pension, hard sigmoid activation function

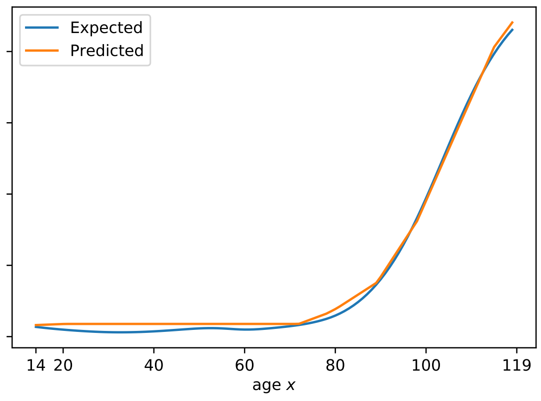

Figure 4.7: NN invalidity pension, hard sigmoid activation function

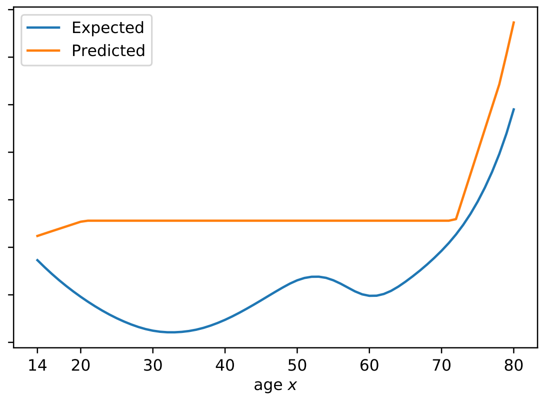

Figure 4.8: NN invalidity pension, ages 14 to 80, hard sigmoid activation function

Many times, while compiling the neural network with the hard sigmoid activation function, the network output would even only be a constant line over the whole data (Figure 4.9). Therefore, the hard sigmoid should not be chosen as the activation function.

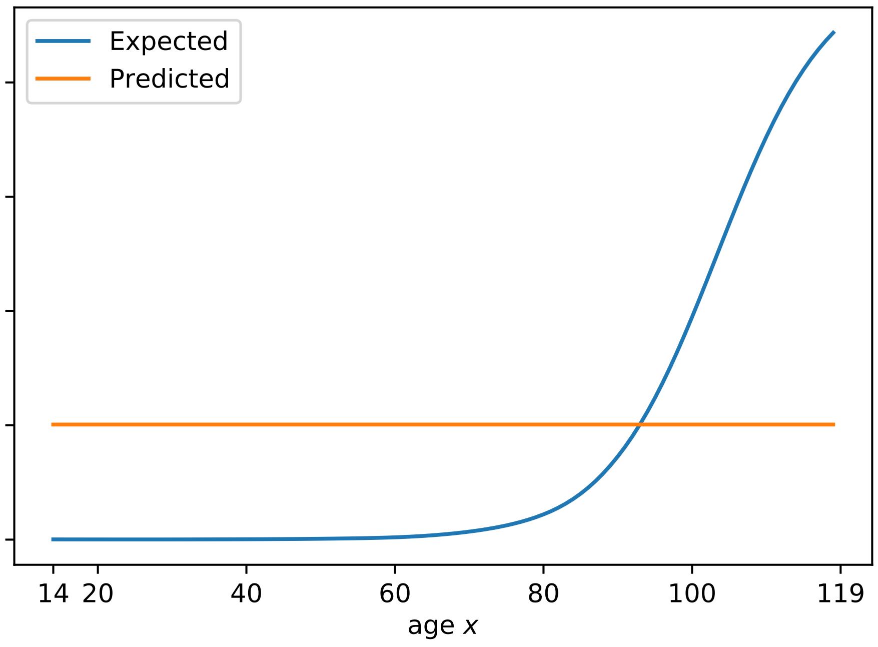

Figure 4.9: NN old-age pension, hard sigmoid activation function 2

The MSE for the hyperbolic tangent is very low. One reason for such a good result is that the vanishing gradient problem is a much smaller issue for the function, since its range is between -1 and 1. The fact that the data is centred around 0 leads to higher derivatives. The other benefit of the range being $(-1, 1)$ and not just $(0, 1)$ is that with the inclusion of $(-1, 0)$ the outputs are rather unbiased, meaning that the average is close to zero [LBOM12].

The MSE for using softsign as the activation functions is a bit higher. It is a continuous approximation of the sign function.

Overall, the hyperbolic tangent is definitely the best choice of a sigmoidal activation function (Figures 4.10, 4.11 and 4.12).

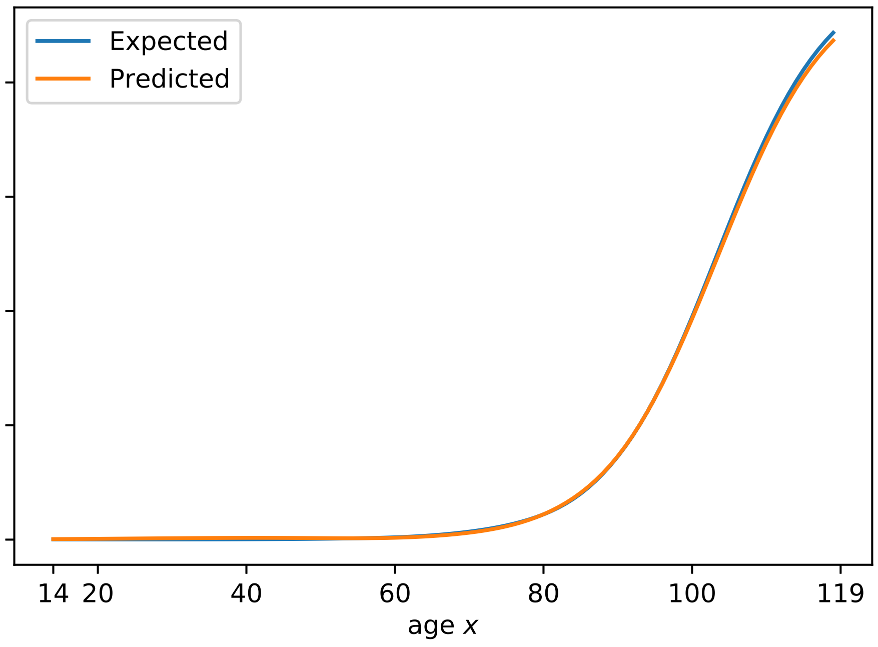

Figure 4.10: NN old-age pension, TanH activation function

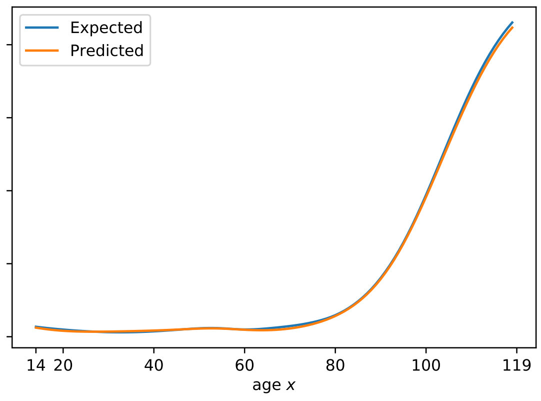

Figure 4.11: NN invalidity pension, TanH activation function

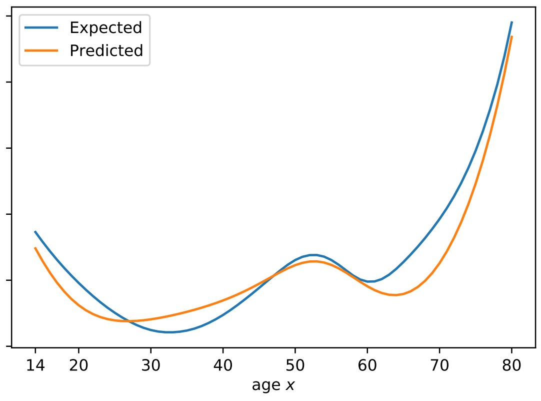

Figure 4.12: NN invalidity pension, ages 14 to 80, TanH activation function

4.3.2 Rectifiers

The rectified linear unit is probably the most used activation function. Nevertheless, in my model it yielded good, but not the best results. It might be the lack of continuous differentiability that causes it to yield worse results, which is why I also tested the softplus. The softplus is a smooth version of the ReLU. As you can see in figure 4.5, the softplus did perform better than the ReLU.

The exponential linear unit is also a modification of the ReLU. The neural network using the ReLU runs through all epochs faster, than with using the ELU, because it tries to make its mean activations closer to zero. Its mean squared error is really low and belongs to the three best activation functions for my neural network.

The scaled exponential linear unit did not perform quite as well as the exponential linear unit, but is still a good activation function. This was foreseeable, since it only is a slight modification of the ELU.

At last, I tested the softplus. As already mentioned, the softplus is just a modification of the ReLU and the smoothness caused better performance than when using the ReLU. With a mean squared error of 2.59×10^{-5} it has the best performance of all activation functions.

4.4 Experiment 3 - optimisers

In this section I am trying to find the best optimizer for my neural network. This time, I fixed the number of layers and nodes and the activation function.

After testing the various optimisers several times, I took an average over the results for each optimiser and want to present these results in table 4.3.

Table 4.3: Comparison of different optimisers with MSE value

Activation function	**Mean squared error**
Stochastic gradient descent (SGD)	1.04×10^{-3}
Adagrad	2.32×10^{-4}
Adadelta	5.12×10^{-4}
RMSProp	4.19×10^{-4}
Adam	2.32×10^{-5}
Nadam	4.34×10^{-4}

The neural network yielded abysmal results for basic stochastic gradient descent. It was expected that the MSE would be quite high, since it does not include any adaptations for the learning rate.

That AdaGrad performs surprisingly well. The MSE for the AdaGrad is the second best despite the fact that its learning rule is quite flawed as outlined in subsection 3.3.2. Since the denominator increases at each iteration the learning rule becomes vanishingly small. Therefore, running the network with AdaGrad was expected to yield bad results.

This decrease of the learning rate is generally fixed with the Adadelta optimiser, which predicted the data quite badly nonetheless. The same accounts for the RMSProp, which also is often described as an 'improved AdaGrad' and still has a higher mean squared error than the Adagrad.

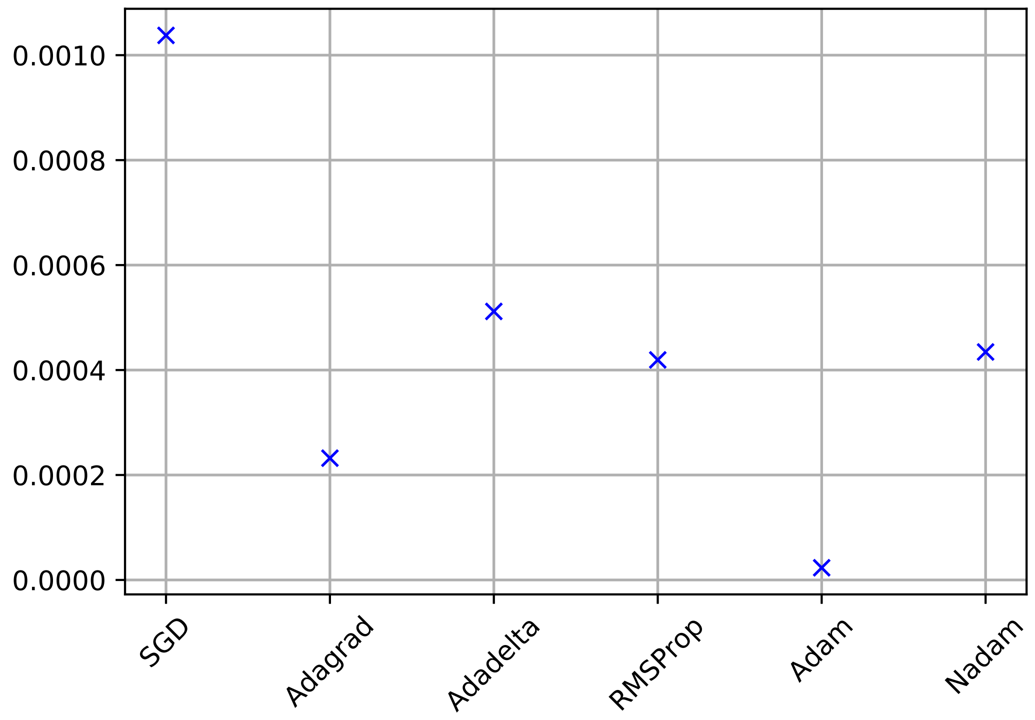

Figure 4.13: Comparison of different optimisers with MSE value

The by far best performance yielded the Adam optimiser. It is also by far the most popular and most used one.

The Nadam did not perform so well, which shows that the Nesterov momentum might not be better than 'normal' momentum.

Another factor for choosing an optimiser might be the time it takes the algorithm to compile. SGD and RMSProp are the fastest optimisers. Adagrad, Adadelta and Adam take a bit longer and Nadam takes almost double the time compared to SGD and RMSProp to compile the neural network.

4.5 Experiment 4 - number of epochs

For deciding on how many epochs my neural network should use I analysed the loss over time. I set the number of epochs to 800. Often the network's loss converged early on, seen in figure 4.14. Although sometimes the error converged slower and there was still improvement at later epochs, which is why I decided to generally let it run through 800 epochs.

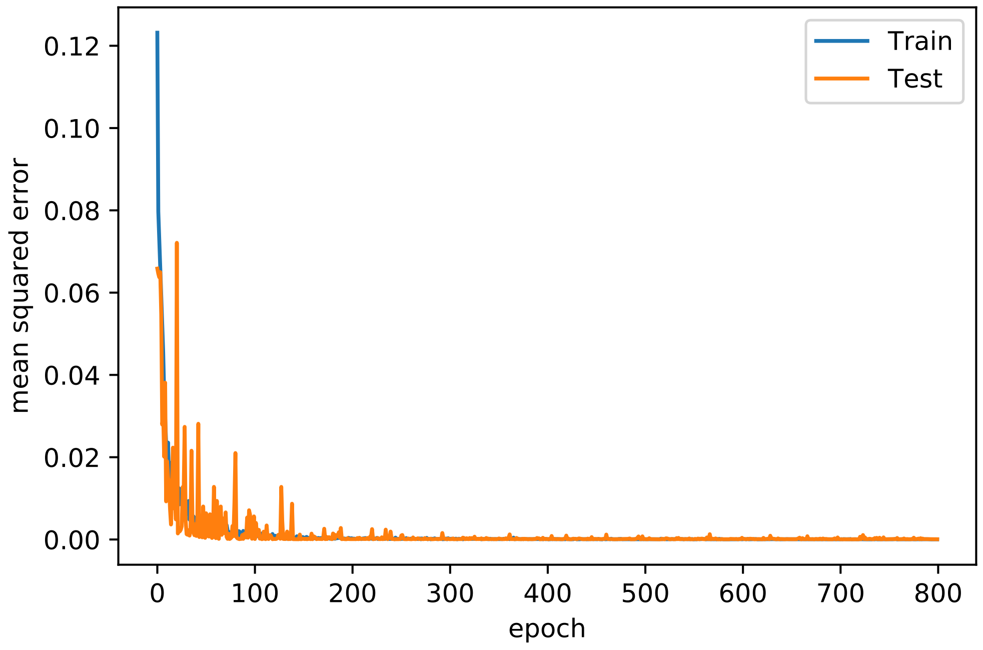

Figure 4.14: MSE over 800 epochs

Chapter 5

Conclusion

In this diploma thesis we investigated the use of a neural network in the insurance business. The approach of using deep learning in insurance is currently not widely spread. Hence, this thesis should act as a stepping stone for implementing this method in actuarial mathematics and contribute to this nascent research field.

It is crucial to evaluate a proper setting for a neural network to yield the best performance. Therefore, I tested between varying amount of nodes and layers, different activation functions and different optimisers.

For this problem I established it was best to use 640 neurons in the neural network. When using less neurons the results were not satisfying. With more than 640 neurons the network would still return good results, but the running time of the model extended drastically. I also found, that it was best to train the data with four layers.

All of the activation functions showed promising results, except for Hard Sigmoid, which definitely should not be used for a linear regression problem. Nevertheless, TanH, ELU and Softplus are the best activation functions to be used for the problem.

Although there are quite a few optimisers, the Adam optimiser is definitely the best choice for this problem. This is in line with much of the literature that finds the Adam optimiser to be the ideal optimiser for a variety of problems. It was quite unexpected that also the AdaGrad optimiser performed very well, since it has its flaw of having a constantly decreasing learning rule until it becomes vanishingly small.

It should also be noted that the convergence of the error, while running through epochs, should be tested and then a judgement call should be made of how many epochs make most sense to use.

Consequently, I could eliminate various scenarios due to bad results and found that the calculation of the parameter q_x via an ideal neural network shows exceptionally promising results. The process of finding the right model did take some time, but once it is built, it is an immensely fast and precise approach to calculate q_x.

All in all, solving this problem with an artificial neural network is a great solution and could affect the whole insurance business. Not only the insurance itself would profit from a fast and straightforward approach to calculate the probability of dying, but also insurers would benefit, since it would lead to better calculations for premiums. The insurers would benefit from reduced cost and can improve their bottom line and the customer receives a better and more affordable product. Hence, the results achieved in this thesis show a high potential to further the research in this field and apply the method of calculation in the insurance business.

Bibliography

[Agg18] Charu C Aggarwal. *Neural networks and deep learning*. Springer, 2018.

[AZ14] Antonios K Alexandridis and Achilleas D Zapranis. *Wavelet neural networks: with applications in financial engineering, chaos, and classification*. John Wiley & Sons, 2014.

[BGKP19] Helmut Bölcskei, Philipp Grohs, Gitta Kutyniok, and Philipp Petersen. Optimal approximation with sparsely connected deep neural networks. *SIAM Journal on Mathematics of Data Science*, 1(1):8–45, 2019.

[BGTW19] Hans Buehler, Lukas Gonon, Josef Teichmann, and Ben Wood. Deep hedging. *Quantitative Finance*, pages 1–21, 2019.

[BJ18] Noorhannah Boodhun and Manoj Jayabalan. Risk prediction in life insurance industry using supervised learning algorithms. *Complex & Intelligent Systems*, 4(2):145–154, 2018.

[BTK$^+$10] Mike Batty, Arun Tripathi, Alice Kroll, Cheng-sheng Peter Wu, David Moore, Chris Stehno, Lucas Lau, Jim Guszcza, and Mitch Katcher. Predictive Modeling for Life Insurance, Ways Life Insurers Can Participate in the Business Analytics Revolution. *Deloitte Consulting LLP*, 2010.

[Cyb89] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[Doz16] Timothy Dozat. Incorporating nesterov momentum into adam. 2016.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[GHV17] Garrett B Goh, Nathan O Hodas, and Abhinav Vishnu. Deep learning for computational chemistry. *Journal of computational chemistry*, 38(16):1291–1307, 2017.

[GP17] Antonio Gulli and Sujit Pal. *Deep Learning with Keras*. Packt Publishing Ltd, 2017.

[Her16] Andres Hernandez. Model calibration with neural networks. *Available at SSRN 2812140*, 2016.

[Hor91] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

[HSS12] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. *Cited on*, 14:8, 2012.

[HSW89] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[KB14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[KHS18] Reinhold Kainhofer, Jonas Hirz, and Alexander Schubert. AVÖ 2018-P: Rechnungsgrundlagen für die Pensionsversicherung. 2018.

[KO11] Bekir Karlik and A Vehbi Olgac. Performance analysis of various activation functions in generalized MLP architectures of neural networks. *International Journal of Artificial Intelligence and Expert Systems*, 1(4):111–122, 2011.

[LBOM12] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

[LY17] Yuanzhi Li and Yang Yuan. Convergence analysis of two-layer neural networks with relu activation. In *Advances in Neural Information Processing Systems*, pages 597–607, 2017.

[Mur12] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[NT16] Thembinkosi Nkonyana and Bhekisipho Twala. An empirical evaluation of machine learning algorithms for image classification. In *International Conference on Swarm Intelligence*, pages 79–88. Springer, 2016.

[Rud16] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[RZL17] Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.

[ten19] tf.keras.backend.hard_sigmoid Tensorflow, 2019. URL: `https://www.tensorflow.org/api_docs/python/tf/keras/backend/hard_sigmoid`.

[Zei12] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

List of Figures

List of Tables