Stockholms
universitet

Using autoencoders to initialize neural networks for claims prediction

Jonathan Bengtsson

Masteruppsats i försäkringsmatematik
Master Thesis in Actuarial Mathematics

Masteruppsats 2025:4
Försäkringsmatematik
Februari 2025

www.math.su.se

Matematisk statistik
Matematiska institutionen
Stockholms universitet
106 91 Stockholm

Mathematical Statistics
Stockholm University
Master Thesis **2025:4**
http://www.math.su.se

Using autoencoders to initialize neural networks for claims prediction

Jonathan Bengtsson*

February 2025

Abstract

In this thesis we discuss predicting number of insurance claims under a Poisson model assumption, using fully connected artificial neural networks. We initialize the weights of these networks using autoencoders, with special attention to the handling of categorical features in the data. More specifically we primarily use a joint embedding of categorical features to learn numerical representation of categories, instead of using the somewhat older and more established way of handling this - separate entity embeddings. We then use these representations together with numerical features to learn representations of all features - representations which we then use to initialize hidden layers in fully connected feed-forward networks. We use denoising autoencoders and undercomplete autoencoders. We evaluate prediction power on a real car insurance data set and find evidence of improvement in comparison to standard methods.

*Postal address: Mathematical Statistics, Stockholm University, SE-106 91, Sweden. E-mail: jonathan.e.bengtsson@gmail.com. Supervisor: Mathias Millberg Lindholm.

Acknowledgements

I would like to thank my supervisor Mathias Millberg Lindholm for guidance, encouragement and interesting discussions. I also would like to thank my family for their support.

Contents

Chapter 1

Introduction

In the last decades computing power has increased significantly, so consequently machine learning techniques have become more and more feasible. This has surged research in the area and machine learning techniques have found increasingly more applications. One field where applications have been proven useful is actuarial science.

In this thesis we focus on using artificial neural networks, which is a subset of machine learning, for predicting number of claims for a set of insurance policy holders given a set of characteristics for each policy holder.

In this chapter we present an overview of the thesis, the particular actuarial problem it concerns, the neural network methods that we try for solving the problem, and the main takeaways of the thesis.

In chapter 2 we give a description of the different types of neural networks that we will use for prediction and chapter 3 concerns usage of neural networks for insurance pricing. We restrict ourselves to predicting the number of claims, we do not try to predict claim sizes.

In chapter 4 we present methods for evaluating the quality of predictors.

In chapter 5 we first present a data set and then apply our neural network predictors discussed in earlier chapters and evaluate them on the data set.

We conclude the thesis with a discussion of our results in chapter 6.

The implementation of neural networks is done mostly in Python using the Tensorflow and Keras libraries, but we also try some implementation from scratch using C. As a reference we use a GBM (gradient boosting machine) model, a well-established method, from the GBM library in R. The code used for this thesis can be found at *https://github.com/JonathanBengtsson.*

1.1 Insurance pricing

A common assumption when modeling insurance portfolios is that the number of claims and the sizes of claims are independent. This justifies the modeling of number of claims and claim sizes as separate processes. This report focuses on predicting the number of claims N for a contract, given duration v and a set of characteristics for the policyholder $\mathbf{X}$. The model used throughout this thesis is

$$N|\mathbf{X}, v \sim \text{Pois}\left(v\mu(\mathbf{X})\right), \tag{1.1}$$

which is a standard choice for modeling number of claims. Common traditional ways to model μ are generalized linear models (GLM) and gradient boosting machines (GBM). In this thesis we focus on using neural network based methods for modeling μ. For comparison, we predict the number of claims for the same data set using a GBM model.

1.2 Artificial neural networks

The particular type of neural network we will use for predicting N is a feed-forward network, which will be further explained in Chapter 2.

The network takes the duration v and characteristics of a policy holder $\mathbf{X}$ as input, and returns a predicted number of claims N as output. In order to achieve this the neural network is trained using empirical sets of characteristics and durations $(\mathbf{X}_i, v_i)_{i=1}^{m}$ and corresponding observed number of claims $(N_i)_{i=1}^{m}$. The performance of the network can then be evaluated using an other set of empirical characteristics and durations $(\mathbf{X}_i,v_i)_{i=m}^{m+n}$ and compare these predictions by the network with the observed number of claims $(N_i)_{i=m}^{m+n}$. This type of training is called supervised learning and the training process consists in fact of solving an optimization problem by tuning a large number of parameters of the network - weights w and biases θ.

1.2.1 Handling categorical features

The characteristics of the policyholders $\mathbf{X}$ are generally referred to as features in machine learning applications. These features can be of three different types: numerical, categorical and binary. Numerical features are straightforward to handle, and binary features can often be implemented by replacing them with a numerical feature taking one of two values, for example $\{0, 1\}$ or $\{-1, 1\}$. How to handle categorical features is less evident. It can be especially hard when the number of labels for one or several categories is large (large cardinality). Since categorical features are common in insurance applications, we especially pay attention to this.

1.2.2 Autoencoders for weight initialization

Before training a neural network it is necessary to decide the initial values of the weights w and biases θ of the network. This is often done by some type of randomization. Inspired by earlier work [1], we try to first train an other type of network - an autoencoder - and then use some of its weights to initialize some of the weights in our original feedforward network, in an attempt to improve performance. There is evidence [1] that this approach may be particularly useful for weights connecting categorical feature inputs to the network.

1.3 Conclusions from experiments on data

A main inspiration for the methods we try in this thesis is an article by Delong & Kozak (2020)[1]. Our conclusions are roughly in accordance with theirs.

We use a data set called freMTPL2freq (French Motor Claims Datasets) to test our prediction methods, and we find that feedforward neural networks perform better than the reference method, GBM. We also find that using autoencoders to initialize weights in feedforward networks seems to increase performance. The particular types of autoencoders called *denoising autoencoders* seem to be better than (non-denoising, ordinary) autoencoders, but this conclusion is less certain than the other ones.

Chapter 2

Artificial Neural Networks

The underlying ideas for the class of machine learning techniques called artificial neural networks emerged from using networks of neurons in mammalian brains as inspiration [5, p. 2-3]. Most of these techniques were developed in the later half of the 20th century, and the main purpose initially was to explain neuro-physiological mechanisms. Here we present some of the main milestones.

- McCulloch and Pitts introduced in 1943 an abstract model for a neuron using Carnap's logical syntax, and demonstrated how such units could be coupled together in order to represent logical functions. Today algebraic notions are used, but in principle the model of the neurons are the same as the model McCulloch and Pitts used – these neurons are often referred to as McCulloch-Pitts neurons.

- In 1949 Hebb described how neural networks learn by strengthening connections between neurons that are active at the same time. This is now referred to as Hebb's learning principle.

- The notion of *perceptron* was introduced in 1958 by Rosenblatt for layered networks of McCulloch-Pitts neuron. Rosenblatt showed that such networks could solve problems that a single McCulloch-Pitts neuron could not.

- In 1986 Rumelhart *et al.* showed that perceptrons can be trained by gradient descent. This provides a general method for efficiently training perceptrons with many layers (multi-layer perceptrons) using backward propagation.

Since then both theoretical descriptions and methods as well as applications of artificial neural networks have been further explored and refined. However, this is still a very active area of research.

2.1 Neurons

We use the model introduced by McCulloch and Pitts. This models neurons as binary threshold units with only two possible outputs, or states: active and inactive [5, p. 7-9].

In practice, a neuron takes a number of input signals x_1,...,x_n and which are weighted and summed, possibly with a bias term θ (also called threshold): $a = \sum_{i=1}^{n} w_i x_i - \theta$. An activation function $g(\cdot)$ (also called transfer function) is applied to this sum and the result becomes the output signal $y = g(a)$. In total

$$y = g\left(\sum_{i=1}^{n} w_i x_i - \theta\right). \tag{2.1}$$

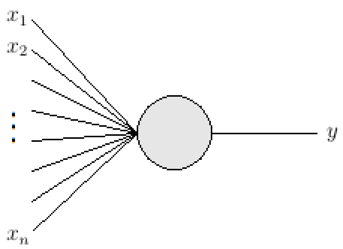


Figure 2.1: A single McCulloch-Pitts neuron with n inputs, as described in equation (2.1).

Since we have many neurons in a network we use indices to keep track of them. For neuron j:

$$y_j = g_j\left(\sum_{i=1}^{n} w_{ij} x_i - \theta_j\right).$$

2.1.1 Activation functions

The activation function(s) g_j should model the biological behavior of the output signal being *active* if the input signals are strong enough, and *inactive* if the input signal is not strong enough [5, p. 9-10]. A simple choice of activation function is the step function

$$g(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0. \end{cases}$$

Other common choices are sigmoid function such as the logistic function

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

and the hyperbolic tangent function

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

It should be noted that there is in general no constraint on the weights w_{ij} to be positive, so the interpretation of input signals needing to be strong enough to activate the neuron is not necessarily straightforward.

2.2 Supervised learning

There are several applications of neural networks. One common class of problems that neural networks are applied to is supervised learning problems where we have a set of patterns that we want our algorithm to classify [5, p. 72-75], [6, p. 103]. If we feed a pattern to the algorithm as input (in these applications: a neural network, historically e.g. a GLM) we want the network to return a certain output, called target. For each input pattern we have a desired target output, the task consists of making the network recognize each of these patterns. In many applications we not only want our network to recognize the patterns we use to train (the process of tuning) it, the network should also recognize similar patterns.

Let each pattern be a vector of values (sometimes called a feature vector). We label the p input patterns with the index $\kappa = 1, ..., p$. Now denote pattern κ: $x^{(\kappa)}$, and the corresponding target: $t^{(\kappa)}$.

2.2.1 Feedforward networks (multilayer perceptrons)

For multilayer perceptrons, also called feedforward neural networks or deep feedforward networks [6, p. 164], we group neurons (in this application also called perceptrons) into layers [5, p. 3].

It is common to depict the input layer(s) to the left, and output layer(s) to the right. Between these there can be one or more hidden layers.

Within each of these layers there are no connections between the neurons. There are only one-way connections from neurons in a layer to the next layer; this next layer is generally depicted as the layer immediately to the right. There are no connections from right to left, and no connections that skip a layer.

In a fully connected feedforward network each neuron in a layer is connected to all neurons in the next layer. By a neuron (neuron 1) in a layer being connected to an other neuron (neuron 2) in the next layer we mean that the output signal of neuron 1 acts as one of the input signals of neuron 2.

An example of a fully connected feedforward network can be seen in figure 2.2.

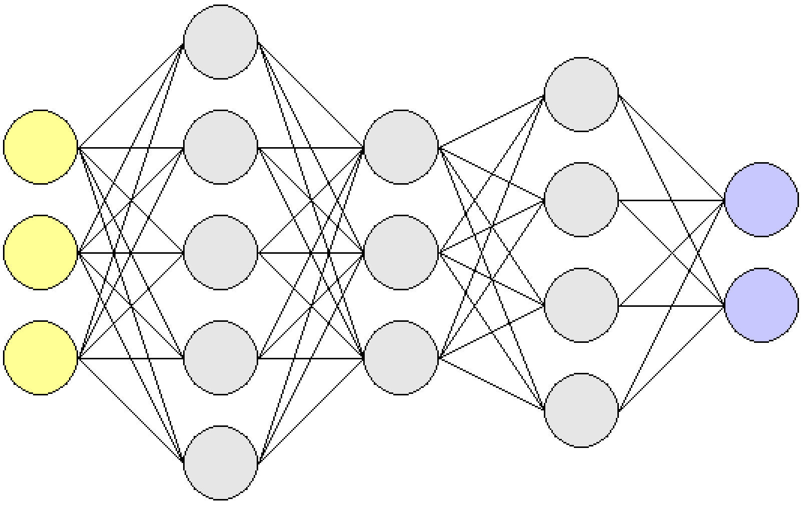


Figure 2.2: A fully connected feedforward network (multilayer perceptron) with 3 neurons in the input layer (yellow) and 2 neurons in the output layer (blue). The network has 3 hidden layers with 5 neurons in the first hidden layer, 3 neurons in the second hidden layer and 4 neurons in the third hidden layer.

When training a network, we consider its design to be fixed, i.e. the number of layers is fixed and the number of neurons in each layer is fixed. The weights between connected neurons are not fixed and subject to change by training algorithms. The bias terms are not fixed either and may also be changed by training algorithms.

Notation and forward propagation

We start by introducing some notation. Let the index l denote the layer. Let l take the values $l = 0, ..., L$ where $l = 0$ is the input layer and $l = L$ is the output layer. Let j refer to a neuron in layer l, let i refer to a neuron in layer $l-1$, and let k refer to a neuron in layer $l+1$. Let $n^{(l)}$ be the number of neurons in layer l, meaning that i, j, k takes values $i = 0, ..., n^{(l-1)}$, $j = 0, ..., n^{(l)}$ and $k = 0, ..., n^{(l+1)}$.

Denote the output of a neuron j in layer l: $V_j = V_j^{(l)}$, the activation function $g_j = g_j^{(l)}$ and the bias $\theta_j = \theta_j^{(l)}$. The weights between layer l and layer $l-1$ are denoted $w_{ij} = w_{ij}^{(l)}$.

Denote the weighted sums of inputs plus bias $a_j = a_j^{(l)} = \theta_j + \sum_{i=0}^{n^{(l)}} w_{ij}^{(l)} V_i^{(l-1)}$. We now turn our attention to describing the process of forward propagation.

A pattern, represented by $n^{(0)}$ values, is fed by setting the output value of neurons in the input layer $V_j^{(0)}$ to the values of the pattern. These values are used as input signals to the next layer, we can then evaluate the output signals in this next layer

$$V_j^{(1)} = g_j^{(1)} \left(\sum_{i=0}^{n^{(0)}} w_{ij}^{(1)} V_i^{(0)} - \theta_j^{(1)} \right).$$

This is repeated with outputs of layer 1 acting as inputs of layer 2, and so on

$$V_j^{(l)} = g_j^{(l)} \left(\sum_{i=0}^{n^{(l-1)}} w_{ij}^{(l)} V_i^{(l-1)} - \theta_j^{(l)} \right)$$

until we reach the output layer

$$V_j^{(L)} = g_j^{(L)} \left(\sum_{i=0}^{n^{(L-1)}} w_{ij}^{(L)} V_i^{(L-1)} - \theta_j^{(L)} \right).$$

We define vectors denoting the values of the neurons in layer l

$$\mathbf{V}^{(l)} = \begin{bmatrix} V_1^{(l)} \\ \\ \vdots \\ \\ V_{n^{(l)}}^{(l)} \end{bmatrix},$$

weights matrices and bias vectors for layer l

$$\mathbf{W}^{(l)} = \begin{bmatrix} w_{11}^{(l)} & w_{12}^{(l)} & \cdots & w_{1,n^{(l-1)}}^{(l)} \\ w_{21}^{(l)} & & & \\ \vdots & & \ddots & \vdots \\ & & & \\ w_{n^{(l)},1}^{(l)} & & \cdots & w_{n^{(l)},n^{(l-1)}}^{(l)} \end{bmatrix}, \qquad \boldsymbol{\theta}^{(l)} = \begin{bmatrix} \theta_1^{(l)} \\ \\ \vdots \\ \\ \theta_{n^{(l)}}^{(l)} \end{bmatrix}.$$

If we use the same activation function $g^{(l)}$ within each layer, which is often the case, we can write

$$\mathbf{V}^{(l)} = g^{(l)}(\mathbf{W}^{(l)} \mathbf{V}^{(l-1)} - \boldsymbol{\theta}^{(l)}).$$

We now introduce a notation for the layers in the network. Layer l we denote $f^{(l)}$ and we define it as

$$f^{(l)}(\mathbf{z}) = g^{(l)}(\mathbf{W}^{(l)}\mathbf{z} - \boldsymbol{\theta}^{(l)}).$$

We see that

$$f^{(l)}(\mathbf{V}^{(l)}) = g^{(l)}(\mathbf{W}^{(l)}\mathbf{V}^{(l-1)} - \boldsymbol{\theta}^{(l)})$$

holds and using the notation $O_j := V_j^{(L)}$ for each output neuron and

$$\mathbf{O} = \begin{bmatrix} O_1 \\ \vdots \\ O_{n^{(L)}} \end{bmatrix}$$

for all the output layer neurons, if we denote the input $\mathbf{x}$ we now have a compact way of describing an entire network,

$$\mathbf{O} = f^{(L-1)} \circ f^{(L-2)} \circ ... \circ f^{(2)} \circ f^{(1)}(\mathbf{x}). \tag{2.2}$$

Using a special notation for the output layer gives us the advantage that we have an easy way to express the output when a specific pattern is fed to the network (no need to specify layer with an index). When pattern $\mathbf{x}^{(\kappa)}$ is fed we denote the outputs

$$\mathbf{O}^{(\kappa)} = \begin{bmatrix} O_1^{(\kappa)} \\ \vdots \\ O_{n^{(L)}}^{(\kappa)} \end{bmatrix} = f^{(L-1)} \circ ... \circ f^{(1)}(\mathbf{x}^{(\kappa)}).$$

Initializing the network

Once we have decided the number of layers and the number of neurons in each of these layers it remains to decide the values of the weights and the biases. As we have already mentioned, deciding the values of these variables is the very meaning of training the network. However we need to decide the initial values. This is generally done by initializing them to random numbers, for example normally distributed $w_{ij}^{(l)}(t = 0) \sim N(0,1)$ or uniformly distributed on some interval, for example $[-1,1]$. For bias values an other common option is to initialize them to zero.

The way to initialize weights and biases is not always clear. In fact one of the main purposes with this report is to investigate a method based on autoencoders for an initialization of a network that we hope is better than random, which then can be fine tuned using conventional training methods.

2.2.2 Defining training of a network as an optimization problem

In order to be able to tell if the network performs well on some set of patterns with corresponding targets, we need a measure for this purpose. This measure is called a loss (or cost) function and can be defined in different ways, but should be designed such that it has its global minimum when the output of the network for each pattern equals the correct target pattern. One such loss function is

$$L(y, \hat{y}) = \sum_j (y_j - \hat{y}_j)^2.$$

Setting y to be the output we get when we feed a certain pattern $x^{(\kappa)}$ and $\hat{y}$ to be the corresponding correct output $t^{(\kappa)}$ we can use this to measure the performance of our network

$$\sum_\kappa L(V^{(L)}|_{V^{(0)}=x^{(\kappa)}}, t^{(\kappa)}) = \sum_\kappa \sum_j \left(V_j^{(L)}|_{V^{(0)}=x^{(\kappa)}} - t_j^{(\kappa)} \right)^2. \tag{2.3}$$

Having defined a loss function L we can define the task of training the network as an optimization problem. If we define the set of all weights $\boldsymbol{W}$ and the set of all biases $\boldsymbol{\theta}$ we see that given patterns $x^{(\kappa)}$ and corresponding targets $t^{(\kappa)}$ we want to find weights and biases such that the total loss is minimized:

$$\min_{\boldsymbol{W},\boldsymbol{\theta}} \sum_\kappa L\left(\boldsymbol{W}, \boldsymbol{\theta} | x^{(\kappa)}, t^{(\kappa)}\right).$$

Many methods for training neural networks in a supervised learning setting consist of algorithms for solving this optimization problem, although the loss function L may vary. It is very common to use gradient based methods, for example stochastic gradient descent, which is an adaptation of the ordinary gradient descent optimization algorithm.

2.2.3 Computing the gradient using backwards propagation

Computing the gradient of the loss function w.r.t. weights and biases can be done in several ways, but one of the most common ways of doing this is by the process of backwards propagation. Backwards propagation makes use of the fact that in a feedforward network the neurons are only connected to neurons in adjacent layers, yielding a computationally less expensive algorithm [6, p. 200].

Keeping in mind our compact formulation of a feedforward network 2.2, we note that backwards propagation essentially consists of applying the chain rule to the entire network, where the intermediate derivatives are interpreted as values of the neurons in the hidden layers. The nested structure of the feedforward network makes it possible to compute the value of the neurons in layer l by using already known values of the neurons in layer $l+1$. We only need to compute the last unknown partial derivatives between layer l and $l+1$. For a network with three layers $\mathbf{O} = f^{(3)} \circ f^{(2)} \circ f^{(1)}(\mathbf{x})$, with individual layers $\mathbf{y} = f^{(1)}(\mathbf{x}), \mathbf{z} = f^{(2)}(\mathbf{y}), \mathbf{O} = f^{(3)}(\mathbf{z})$:

$$\begin{aligned}
\frac{\partial \mathbf{O}}{\partial \mathbf{z}} &= \frac{\partial f^{(3)}}{\partial \mathbf{z}} \\
\frac{\partial \mathbf{O}}{\partial \mathbf{y}} &= \frac{\partial \mathbf{O}}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial \mathbf{y}} = \frac{\partial \mathbf{O}}{\partial \mathbf{z}} \cdot \frac{\partial f^{(2)}}{\partial \mathbf{y}} \\
\frac{\partial \mathbf{O}}{\partial \mathbf{x}} &= \frac{\partial \mathbf{O}}{\partial \mathbf{y}} \cdot \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \frac{\partial \mathbf{O}}{\partial \mathbf{y}} \cdot \frac{\partial f^{(1)}}{\partial \mathbf{x}}
\end{aligned}$$

So, for each layer when we compute the partial derivatives we can use results from previous layers.

Square distance loss function

One of the most commonly used loss functions (also called energy function) is the squared distance loss function

$$L = \frac{1}{2} \sum_{\kappa} \sum_{i} (t_i^{(\kappa)} - O_i^{(\kappa)})^2$$

Here we let $t_i^{(\kappa)}$ denote the i:th component of the target pattern κ. For clarity we call the output of the network the i:th output neuron of the network that we get when we forward propagate the input pattern κ. Hence for input pattern $x^{(\kappa)}$ we have the output neurons $O_i^{(\kappa)} = V_i^{(L)}(x^{(\kappa)})$.

To simplify notation we consider the case with one input pattern. This is easy to generalize to cases with more input patterns since we only need to sum up the contributions from each pattern in order to obtain the total loss.

We want to compute the gradient, meaning we want to differentiate the loss function with respect to all weights and all bias terms. The derivative w.r.t. the bias term in the output layer and the derivative w.r.t. the weights connected to the output layer is easiest to compute $(\theta_n^{(L)}, w_{nm}^{(L)})$. We apply the chain rule as described in previously, with some small deviations since we differentiate a loss function not the output, and we do not differentiate w.r.t. the values of the neurons but w.r.t. weights and biases.

After differentiating w.r.t. weights and bias connected to the output layer (layer L) we proceed by compute derivatives w.r.t. bias term and weights in the next layer $\theta_n^{(L-1)}, w_{nm}^{(L-1)}$ using results from computing derivatives w.r.t. weights and bias in layer L. Then when computing w.r.t. bias term and weights in layer $L-2$ we use results from when we computed derivatives w.r.t. layer $L-1$. In this way we proceed recursively until we have computed all derivatives [5, p. 91-107].

The details of the derivations are given in appendix A, here we just present the final formulas for recursively computing the gradient:

$$\begin{aligned}
\frac{\partial L}{\partial w_{mn}^{(l)}} &= d_n^{(l)} V_m^{(l-1)}, && 1 \leq l < L \\
\frac{\partial L}{\partial \theta_n^{(l)}} &= d_n^{(l)}, && 1 \leq l < L \\
\frac{\partial L}{\partial w_{nm}^{(l)}} &= (t_n - O_n) g'(a_n^{(l)}) V_m^{(l-1)}, && l = L \\
\frac{\partial L}{\partial \theta_n^{(l)}} &= (t_n - O_n) g'(a_n^{(l)}), && l = L
\end{aligned}$$

where the partial results $d_\cdot^{(\cdot)}$ are computed recursively starting from $l = L$.

$$\begin{aligned}
d_i^{(L)} &= (t_i - O_i) g'(a_i^{(L)}) \\
d_i^{(L-1)} &= \sum_j d_j^{(L)} w_{ij}^{(L)} g'(a_i^{(L-1)}) \\
d_i^{(L-2)} &= \sum_j d_j^{(L-1)} w_{ij}^{(L-1)} g'(a_i^{(L-2)}) \\
&\dots \\
d_i^{(l)} &= \sum_j d_j^{(l+1)} w_{ij}^{(l+1)} g'(a_i^{(l)}) \text{ for } 1 \leq l < L.
\end{aligned}$$

Softmax output and Cross entropy loss function

The softmax output is suitable to use when we deal with a classification problem and the output neurons signifies different classes. The Softmax output has the property that all outputs sum to 1. Hence, the outputs may be interpreted as probabilities.

Softmax output for output neuron i is defined as follows:

$$O_i = \frac{e^{a_i^{(L)}}}{\sum_k e^{a_k^{(L)}}} \tag{2.4}$$

When we use Softmax outputs it is appropriate to use another loss function, namely the Cross entropy loss function

$$L = \sum_{\kappa} \sum_{i} t_i^{(\kappa)} \log(O_i^{(\kappa)}). \tag{2.5}$$

Here we just present the formulas for computing the gradient, the details of the derivations can be found in appendix A.

$$\begin{aligned} \frac{\partial L}{\partial w_{mn}^{(l)}} &= d_n^{(l)} V_m^{(l-1)} \\ \frac{\partial L}{\partial \theta_n^{(l)}} &= d_n^{(l)} \end{aligned}$$

for $1 \leq l \leq L$, where the partial results $d_\cdot^{(\cdot)}$ are computed recursively starting from $l = L$:

$$\begin{aligned} d_i^{(L)} &= \sum_j (t_i - t_j O_i) \\ d_i^{(L-1)} &= \sum_j (t_i - t_j O_i) w_{ij}^{(L)} g'(a_i^{(L-1)}) \\ d_i^{(L-2)} &= \sum_j d_j^{(L-1)} w_{ij}^{(L-1)} g'(a_i^{(L-2)}) \\ &\dots \\ d_i^{(l)} &= \sum_j d_j^{(l+1)} w_{ij}^{(l+1)} g'(a_i^{(l)}) \text{ for } 1 \leq l < L-1. \end{aligned}$$

Prediction and Poisson deviance loss function

For the application of prediction, in our case predicting number of claims N (it would be possible to do this also for frequency N/v, where v as before denotes duration), it is appropriate to use one single Poisson output neuron

$$\xi = e^{R+\theta_1^{(L)}+\sum_j w_{j1}^{(L)} V_j^{(L-1)}}. \tag{2.6}$$

where R denotes the exposure (for our applications this is duration v). Although μ is usually used to denote the Poisson output, we will use ξ to avoid confusion with the function $\mu(\cdot)$ we introduced previously for an other purpose.

When this type of output is used, i.e. the outputs are assumed to follow a Poisson model, the appropriate loss function to use is the Poisson deviance loss function [7]. Using N as target (the observed number of claims), and once again assuming that we only have one input pattern we write it

$$L = 2N\left(\frac{\xi}{N} - 1 - \log(\frac{\xi}{N})\right)$$

The details of the derivations can be found in appendix A, here we just present the final formulas. The components of the gradient are given by

$$\begin{aligned}\frac{\partial L}{\partial w_{mn}^{(l)}} &= d_n^{(l)} V_m^{(l-1)} \\ \frac{\partial L}{\partial \theta_n^{(l)}} &= d_n^{(l)},\end{aligned}$$

where the partial results $d_\cdot^{(\cdot)}$ are computed recursively starting from $l = L$:

$$\begin{aligned}d_i^{(L)} &= 2(\xi - N) \\ d_i^{(L-1)} &= d_1^{(L)} w_{i1}^{(L)} g'(a_i^{(L-1)}) \\ d_i^{(L-2)} &= \sum_j d_j^{(L-1)} w_{ij}^{(L-1)} g'(a_i^{(L-2)}) \\ &\dots \\ d_i^{(l)} &= \sum_j d_j^{(l+1)} w_{ij}^{(l+1)} g'(a_i^{(l)}) \text{ for } 1 \leq l < L-1.\end{aligned}$$

2.2.4 Training the network: optimizing the loss function

Knowing how to compute the gradient we are now in a position to use a gradient based optimization algorithm to optimize the loss function (2.3).

Stochastic gradient descent and different versions of it are among the most commonly used optimization algorithms for deep learning [6, p. 149], and the algorithm we use in this thesis – Nesterov-accelerated adaptive moment estimation (NADAM) is one of these [2]. In appendix B we present the algorithms on which it is based.

Nesterov-accelerated adaptive moment estimation (NADAM)

NADAM, shown in algorithm 1, improves the ADAM algorithm (adaptive moments) by incorporating insights from an other algorithm known as Nesterov's accelerated gradient (NAG) [2]. The ADAM improves SGD in two ways. First, it uses moments – taking into account gradients from previous iterations. And second, it uses an adaptive learning rate.

The use of moments allows the algorithm to move faster when the update direction in subsequent iterations are similar, and slower when the direction oscillates more significantly.

NAG improves SGD with moments by, for each update, move in the direction of the momentum, and then in this point compute the gradient and move in the direction of the negative gradient. In SGD with momentum (classical momentum) we first compute the gradient and then move in the direction of momentum plus gradient.

Note that with $\mathbf{g}_k^2$ we mean power of 2 applied elementwise.

Algorithm 1 Nesterov-accelerated adaptive moment estimation

```
Require: Starting point (initial parameters) $\mathbf{z}_0$.
Require: Step sizes (learning rate) $\epsilon$
Require: Exponential decay rates for moment estimates $\rho_1, \rho_2 \in [0,1)$
Require: Small constant $\delta$ for numerical stabilization.
Require: Stopping condition $C$
  $k \leftarrow 1$
  $\mathbf{r}_0 \leftarrow \mathbf{0}$ (first order moment)
  $\mathbf{s}_0 \leftarrow \mathbf{0}$ (second order moment)
  while $C$ not fulfilled do
      Sample a set of $m$ indexes $M_k$ for specifying the minibatch to use in this iteration
      Compute gradient $\mathbf{g}_k \leftarrow \frac{1}{m}\nabla_{\mathbf{z}_{k-1}} \sum_{i \in M_k} L(f(x^{(i)}; \mathbf{z}_{k-1}), t^{(i)})$
      First order moment $\mathbf{s}_k \leftarrow \rho_1 \mathbf{s}_{k-1} + (1 - \rho_1)\mathbf{g}_k$
      Second order moment $\mathbf{r}_k \leftarrow \rho_2 \mathbf{r}_{k-1} + (1 - \rho_2)\mathbf{g}_k^2$
      Bias correction in first moment $\hat{\mathbf{s}}_k \leftarrow \frac{\rho_1}{1-\rho_1^{k+1}}\mathbf{s}_k + \frac{(1-\rho_1)}{1-\rho_1^k}\mathbf{g}_k$
      Bias correction in second moment $\hat{\mathbf{r}}_k \leftarrow \frac{\rho_2}{1-\rho_2^k}\mathbf{r}_k$
      Update $\mathbf{z}_k \leftarrow \mathbf{z}_{k-1} - \frac{\epsilon}{\sqrt{\hat{\mathbf{r}}_k + \delta}}\hat{\mathbf{s}}_k$ (operations elementwise)
      $k \leftarrow k + 1$
  end while
  return $\mathbf{z}_k$
```

2.3 Representing categorical data

A common way of handling categorical variables, not only for neural networks specifically, is to represent it using so-called one-hot encoding [1].

Suppose that we have a variable consisting of c categorical features. We call the different values that a categorical feature can take labels. Since categorical features in general have different number of labels we introduce notation for this, category $j \in \{1, ..., c\}$ have m_j labels. In a one-hot encoding setting each such category is represented by m_j binary variables, where all binary variables are 0 except the one that represents the value of the categorical feature.

For example, if we have a categorical feature $\tilde{z}$ that represents color (category) with four possible colors, say: blue, green, red and yellow (labels), a one hot encoding of such a variable is $(1_{\tilde{z}="blue"}, 1_{\tilde{z}="green"}, 1_{\tilde{z}="red"}, 1_{\tilde{z}="yellow"})'$. So, if $\tilde{z} = "green"$ then the one-hot encoded representation is $\tilde{z} = (0, 1, 0, 0)'$. If in addition to color (category 1) we have other categorical features, for example month (category 2) and finger on a hand (category 3), then the number of categories is $c = 3$ and numbers of labels are $m_1 = 4, m_2 = 12$ and $m_3 = 5$.

The point of using a set of binary variables as representation instead of just having one integer-valued variable where different integers represent different labels, is that in the former way there is no order between the labels which on the other hand we would get if we used an integer-valued variable to represent it. Often it is the case that the labels of a categorical feature do not have an internal order, for example different types of cars, although there are cases where such an order exists, for example when stratifying an age span. However, for our applications we will use one-hot encoding handling for all categorical features.

If we denote a categorical feature x_j with m_j different labels $\{b_1^j, ..., b_{m_j}^j\}$, in the process of one-hot encoding we transform this one-dimensional feature variable into an m_j dimensional vector

$$x_j \mapsto \mathbf{x}_j^{\text{cat}} = (x_{j_1}, ...x_{j_{m_j}})' = (\mathbf{1}_{x_j=b_1^j}, ..., \mathbf{1}_{x_j=b_{m_j}^j})'$$

Doing this for all features we get our one-hot encoded categorical feature vector $\mathbf{x} = ((\mathbf{x}_1^{\text{cat}})', ..., (\mathbf{x}_c^{\text{cat}})')'$.

If we start with data in a d-dimensional vector, representing d features, and of these features c are categorical and $d - c$ are numerical, we can do the same one-hot encoding process by just ignoring the numerical features. Supposing that the order of the variables is such that the c first variables are categorical and the remaining ones are numerical, the process of one-hot encoding would give us the variable

$$\mathbf{x} = ((\mathbf{x}^{\text{cat}})', (\mathbf{x}^{\text{num}})')' = ((\mathbf{x}_1^{\text{cat}})', ..., (\mathbf{x}_c^{\text{cat}})', x_{c+1}, x_{c+2}, ...x_d)'$$

which is a variable of dimension $D = \sum_{j=1}^{c} m_j + d - c$.

2.4 Autoencoders

An autoencoder is a neural network that is characterized by the task it tries to perform, namely copying the input and returning it as output [6, p. 499]. It consists of an encoder part that encodes the input $f_{\text{enc}}(\mathbf{x}) = \mathbf{h}$ and a decoder part that tries to reconstruct the input from the code that the encoder outputs, $f_{\text{dec}}(\mathbf{h}) = \mathbf{r}$. Typically, an autoencoder is a feedforward network with one hidden layer. In that case the neurons in the hidden layer describe the code $\mathbf{h}$.

Since the autoencoder then can be seen as just a special case of a feedforward network, one can train it in the same ways as other feedforward networks.

The usefulness of the autoencoder comes from imposing limitations on it. If the autoencoder cannot copy perfectly it is forced to prioritize which properties of the input data that should be encoded, so that the decoder is able to make sufficiently good reconstructions from it. Often these prioritized properties of the data are useful.

To sum up, the usefulness of the autoencoder comes from the fact that the training process is trying to find a compromise between two different objectives:

- To learn a representation $\mathbf{h}$ of the input $\mathbf{x}$ in the encoder part, so that $\mathbf{x}$ can be approximately recovered by the decoder part from $\mathbf{h}$.
- Satisfying the limitations imposed on the autoencoder. Typically, this can be constraints that comes from the design of the autoencoder or from some type of regularization penalty term added to the reconstruction cost.

2.4.1 Constraining the autoencoder

A common way to impose limitations on the autoencoder is to make it **undercomplete** [6, p. 500-501]. An undercomplete autoencoder is a feedforward network with one hidden layer where the number of neurons in the hidden layer is lower than the number of input/output neurons.

As usual with feedforward networks the training process consists of minimizing a loss function $L(\mathbf{x}, f_{\text{dec}}(f_{\text{enc}}(\mathbf{x})))$. In in our previously introduced terminology for feedforward networks (2.2), the undercomplete autoencoder can be written as

$$\mathbf{O} = f^{(2)} \circ f^{(1)}(\mathbf{x})$$

with $f^{(1)} = f_{\text{enc}}$ and $f^{(2)} = f_{\text{dec}}$. The fact that it is an autoencoder means that $n^{(2)} = \dim(\mathbf{x})$, and if the autoencoder is undercomplete we have that the number of neurons in the hidden layer is smaller than the dimension of the input data, meaning that we require $n^{(1)} < n^{(2)}$. The choice of activation functions $g^{(1)}, g^{(2)}$ depends on the specific application, but can for example be hyperbolic tangent functions, or just linear functions.

If we use the mean squared error as loss function, and we use linear activation functions for the decoder layer, the autoencoder learns the same subspace as PCA (principal component analysis). Undercomplete autoencoders with nonlinear encoder functions f_{enc} and nonlinear decoder functions f_{dec} can therefore be thought of as generalizations of PCA.

2.4.2 Denoising autoencoders

A denoising autoencoder is a type of autoencoder that is not defined by the way it is designed, but how it is trained. In the training step for denoising autoencoders we corrupt (perturb) the input data $\mathbf{x}$ by some process $p(\cdot)$ but not the output data. Hence, the loss function we minimize in the training step is $L(\mathbf{x}, f_{\text{dec}}(f_{\text{enc}}(p(\mathbf{x}))))$ [6, p. 504-505, 507-512].

The point of adding noise to the input data is to increase the autoencoder's ability to recognize data that is similar but not identical to the data represented in the training set. This is useful under the assumption that small changes in input data should yield small changes in output data (i.e. if we have a well-conditioned problem).

We present two methods of adding noise here, one for numerical data $p_1(\cdot)$ and one for categorical data $p_2(\cdot)$. The reason we use these types of noise is that they are used by Delong&Kozak (2020)[1] in their experiment for predicting number of claims in the model with the best result.

When we add noise, we apply $p_1(\cdot)$ and $p_2(\cdot)$ to each sample.

Gaussian noise for numerical features

This type of perturbation consists of adding a small normally distributed noise to each numerical feature, $p(\mathbf{z}) = [N(z_1, \sigma^2), N(z_2, \sigma^2), ...]^T$. The size of the noise can be varied by choosing different values on σ.

Sample noise for categorical features

For each category $j = 1, ..., c$ we compute the empirical distribution F_j using the training data. The noise consists of replacing a fraction of the categories (randomly chosen) with a random value drawn from the corresponding empirical distribution.

Let $\tilde{z}_j \sim F_j$ for $j = 1, ..., c$ and denote the set of indices of the categories to replace by S, where the elements of S are drawn from a uniform distribution from 1 to c. The number of elements (indices) that are drawn, q, is used as a hyperparameter to set the amount of noise. Then we can write $p_2(\mathbf{z}) = [(z_1 \cdot \mathbf{1}_{1 \notin S} + \tilde{z}_1 \cdot \mathbf{1}_{1 \in S}), (z_2 \cdot \mathbf{1}_{2 \notin S} + \tilde{z}_2 \cdot \mathbf{1}_{2 \in S}), ...]^T$.

2.4.3 Autoencoders for numerical features

Autoencoders for numerical features can be done in different ways, but we follow Delong&Kozak (2020)[1], who were inspired by an approach investigated by Rentzmann&Wütrich (2019)[9]. We use an undercomplete autoencoder with one hidden layer, where the exact number of neurons $n^{(1)}$ in the hidden layer will vary. For activation functions we use the hyperbolic tangent function for the encoder and linear activation function for the decoder. In figure 2.3 we show an example of this type of autoencoder.

Like Delong&Kozak (2020), but unlike Rentzmann&Wütrich (2019) we train bias terms, since we follow Delong&Kozak (2020) in using the min-max scaler transformation for input data instead of scaling the data to mean zero and unit variance. To measure the reconstruction error between prediction and input we use the mean squared error loss function.

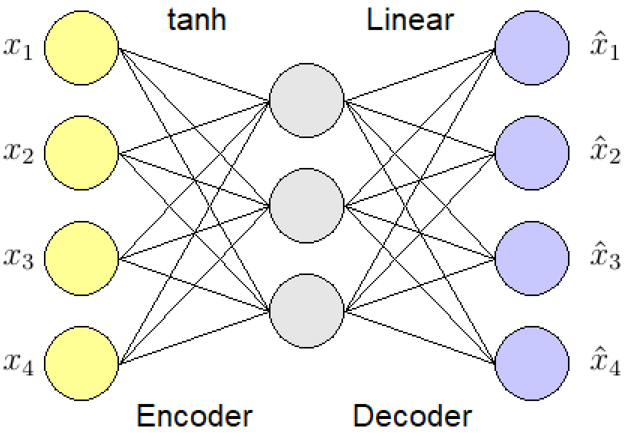


Figure 2.3: Autoencoder for numerical features with 4 inputs and 4 outputs and a hidden layer with 3 neurons. The neurons in the hidden layer use the hyperbolic tangent function as activation function, while the output layer uses linear activation functions.

2.4.4 Autoencoders for categorical features

Autoencoders for categorical features is a much less explored subject than autoencoders for numerical or binary features. For insurance applications categorical features are often relevant, since customers can almost always be divided into groups that are best represented as categorical features for quantitative analysis purposes. Just as we do for numerical features, we follow (parts of) the approach used by Delong&Kozak(2020)[1].

We will only consider the case where the categorical data is one-hot encoded, since this is a standard way of representing categorical data where the labels do not have an internal order. When one-hot encoding is used there is no need to use biases between the input layer and the hidden layer, since the sum of inputs for any given neuron in the hidden layer is always equal to the number of categories c. A bias term b could always be replaced by adding b/c to every weight connected to the neuron in the hidden layer.

In the first layer (the encoder) we use linear activation functions, since for one-hot encoded input we get unique constants for each label in each category, so there is no need to use any transformations that are more complex.

We want to do a classification so we will use the cross-entropy loss function (2.5) and softmax (2.4) for the output neurons. If we had data with only one feature the hot one-encoded data could be handled in a straightforward manner. We could use one softmax function over all outputs with the interpretation of the values of the output neurons as probability for the categorical feature to be a specific label.

Since we often do not have only one feature but several features, and for each one of these features the network is supposed to pick the correct label, there is no one clear way in which to proceed. Here, we consider three possible ways to handle classification with several features.

Separate autoencoders for each feature

A straightforward way to do classification with several features is to simply do it with different networks for each feature. This is possible for our purpose of training autoencoders. For each network we then have let the number of inputs $n^{(0)}$ and output neurons $n^{(2)}$ are be $n^{(0)} = n^{(2)} = m_j$ – the number of labels for that categorical feature. The number of neurons in the hidden layer $n^{(1)}$ can vary, but since we use undercomplete autoencoders we restrict ourselves to cases with $n^{(1)} < m_j$.

The softmax for the output neuron i for these networks are

$$O_i = \frac{e^{a_i^{(L)}}}{\sum_{k=1}^{m_j} e^{a_k^{(L)}}},$$

where the total number of layers are $L = 2$, m_j is the number of labels for feature j. As before, $a_k^{(L)}$ is the input signal of neuron k in layer L. We show an example of such an autoencoder in figure 2.4.

There are however some disadvantages to this approach. First, we cannot make use of possible cross-term correlations in the data if such correlations exist. Second, we need to train one network for each feature which takes more time and computing power to do. And third, we need to investigate what number of neurons to use in the hidden layer $n^{(1)}$ for each categorical feature, which requires more analysis than if we only used one network.

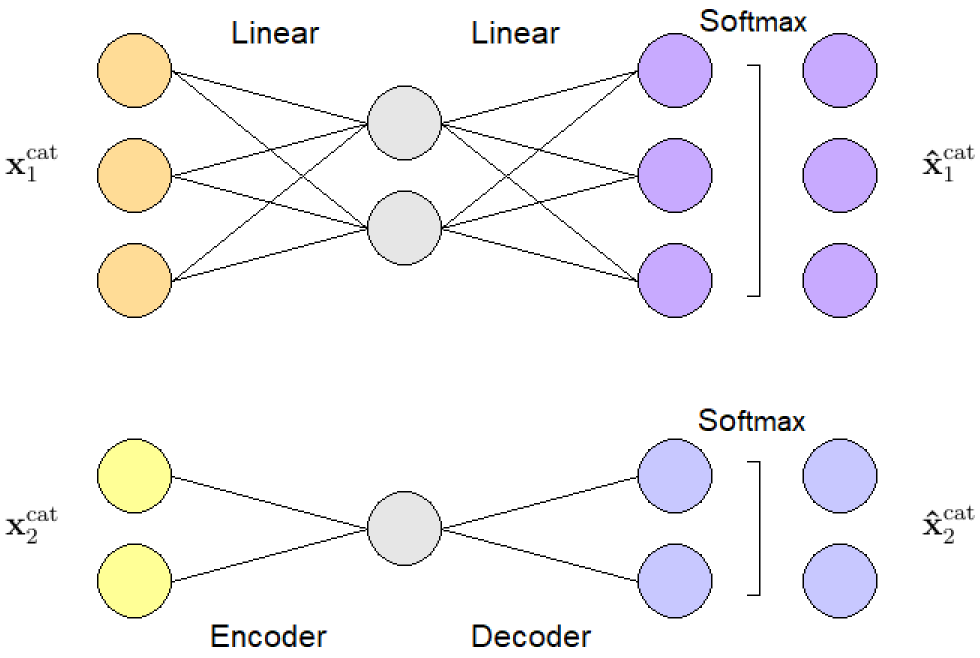

Linear
Linear
Softmax
$\mathbf{x}_1^{\text{cat}}$
$\hat{\mathbf{x}}_1^{\text{cat}}$
Softmax
$\mathbf{x}_2^{\text{cat}}$
$\hat{\mathbf{x}}_2^{\text{cat}}$
Encoder
Decoder

Figure 2.4: Two categorical features (3 labels for category 1, and 2 labels for category 2) with separate autoencoders for each categorical feature. In the encoder part of the autoencoders linear activation functions are used, and for the decoder part the output neurons are linearly connected to the hidden layer but with softmax taken over all output neurons.

One autoencoder in total, with one softmax for all outputs

We could also use just one network for all features, with one softmax for all output neurons. Compared to the method above, the network can capture correlative effects in the date. The number of input neurons and output neurons then are equal to the sum of the number of labels in each categorical feature $n^{(0)} = n^{(2)} = \sum_{j=1}^{c} m_j$

The softmax is designed to for each neuron output a number between 0 and 1 so that all outputs sum to one, which can be interpreted as probabilities of the categorical feature to belong to the label that the output neuron represents. Here the output is several features, but we only use one softmax. An example of this type of autoencoder can be seen in figure 2.5.

$$O_i = \frac{e^{a_i^{(L)}}}{\sum_{k=1}^{n^{(2)}} e^{a_k^{(L)}}},$$

Hence the probabilities of the labels within each categorical feature, will not sum up to one, but the probabilities for labels from all categories will. The way we interpret this is to, for each feature, say that the label that is picked by the network is the one with the output neuron with the highest probability. A problem with this is that it could be the case that for some of the features all labels may have very low probability although the correct one have the highest probability. In the process of training the network this correct classification would not be recognized as well as if the probability were significantly higher than the probabilities of the other labels.

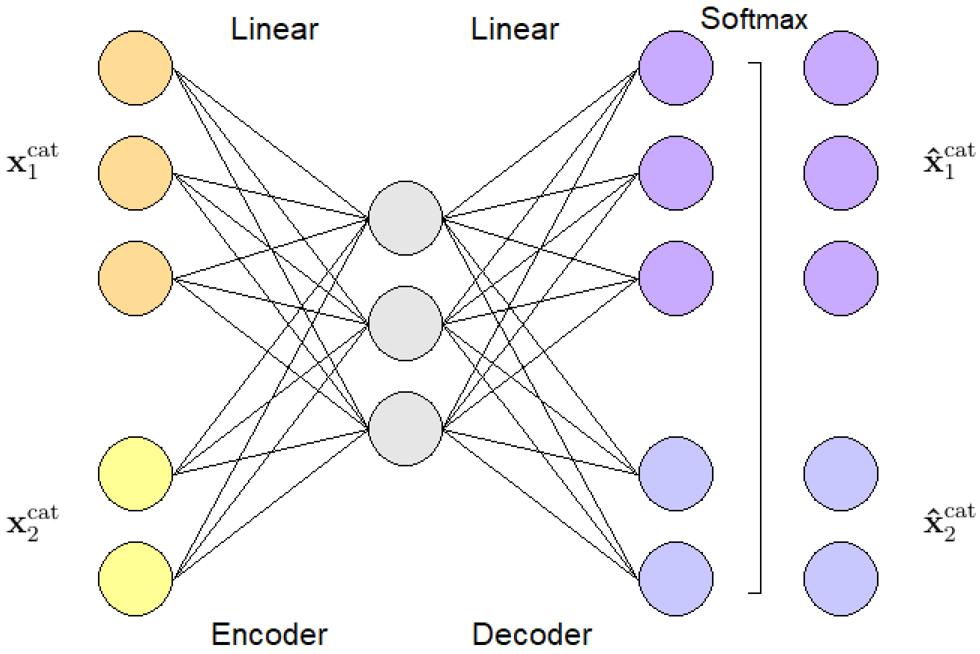

Linear
Linear
Softmax
$\mathbf{x}_1^{\text{cat}}$
$\mathbf{x}_2^{\text{cat}}$
$\hat{\mathbf{x}}_1^{\text{cat}}$
$\hat{\mathbf{x}}_2^{\text{cat}}$
Encoder
Decoder

Figure 2.5: Two categorical features (3 labels for category 1, and 2 labels for category 2) in one autoencoder with one softmax for categorical features. In the encoder part of the autoencoders linear activation functions are used, and for the decoder part the signals from the hidden layer (linearly connected) are taken as input to one softmax function.

One autoencoder in total, with one softmax per feature among the outputs

A third way of handling classification with several categorical features is to use one network, but with one softmax function over each set of output neurons that represent labels of the same feature. As before we have the number of inputs and outputs $n^{(0)} = n^{(2)} = \sum_{j=1}^{c} m_j$, and since we use undercomplete autoencoders we chose the number of neurons in the hidden layer $n^{(1)} < n^{(2)}$. Defining $m_0 = 0$ we can express these softmax functions as

$$O_i = \frac{e^{a_i^{(L)}}}{\sum_{k=m_{j-1}+1}^{m_j} e^{a_k^{(L)}}} \text{ with } j \text{ such that } m_{j-1} < i \leq m_j.$$

This way the output neurons can for each categorical feature, be interpreted as probabilities of the feature belonging to the label that the neurons represent, and we do not get the same problem as with the case where only one softmax function is used. The main disadvantage compared to the both previous methods is that it is a bit less straight forward to implement. We show an example of this type of autoencoder in figure 2.6.

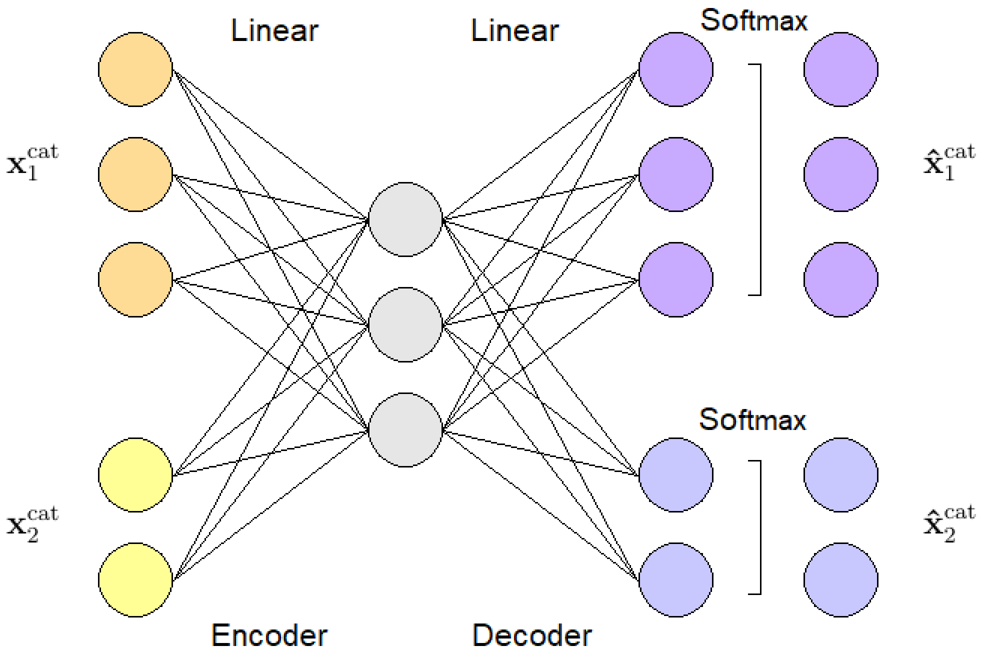

Linear
Linear
Softmax
$\mathbf{x}_1^{\text{cat}}$
$\hat{\mathbf{x}}_1^{\text{cat}}$
Softmax
$\mathbf{x}_2^{\text{cat}}$
$\hat{\mathbf{x}}_2^{\text{cat}}$
Encoder
Decoder

Figure 2.6: Two categorical features (3 labels for category 1, and 2 labels for category 2) in one autoencoder with a softmax for each categorical feature (c softmax functions). In the encoder part of the autoencoders linear activation functions are used. For the decoder part the signals from the hidden layer (linearly connected) that are connected to output neurons that represents labels of the same categorical feature are taken as input into the same softmax function, i.e. one softmax per categorical feature. So, here with $c = 2$ categories, two softmax functions are used.

Chapter 3

Insurance pricing

In order for the insurer to set a fair price for the premium, given a set of features characterizing a policyholder, we want to know the expected amount of money that the insurer needs to pay this type of policyholder. It is often appropriate to assume that the risk of a claim to occur is independent of the size of the claim. This motivates us to model the size of each claim given a set of features $\mathbf{X}$ separately from modeling the number of claims given a group of policyholder characterized by features $\mathbf{X}$. In this thesis we focus solely on predicting the number of claims.

We assume that claims occur according to an unknown distribution $\mu(\mathbf{X})$, which we want to model in order to predict future number of claims. We denote the distributions of these models $\pi(\mathbf{X})$. We focus on models where $\pi(\mathbf{X})$ is a neural network, more specifically a feedforward neural network. We also try gradient boosting machines (GBM) for reference, since this is a well-established method for predicting number of claims. An overview of GBM and of the particular version we use, is given in Appendix C.

3.1 Predicting number of claims using feedforward networks

A common type of artificial neural network that is used for insurance pricing, and can be used to predict number of claims, is the feedforward network with one output neuron and where the features $\mathbf{X}$ are fed to the input neurons [7]. Categorical features are generally one-hot encoded so assuming that we have d features, of which c are categorical features with m_j categories for feature j (see section 2.4 for more details on representing categorical data) the number of input neurons in our network is $D = \sum_{j=1}^{c} m_j + d - c$. There are a couple of design choices that need to be made on how to concatenate the categorical feature input neurons and the numerical feature input neurons [1] which will be discussed in the next section.

We chose to use the Poisson deviance loss function to measure the error between the observed number of claims $N^{(\kappa)}$ for a set of features (pattern) κ, and the network's prediction of number of claims $\pi(\mathbf{X}^{(\kappa)}, v^{(\kappa)})$ (were we write durations v explicitly) for the same features.

$$L(N^{(\kappa)}, \pi(\mathbf{X}^{(\kappa)}, v^{(\kappa)})) = 2N^{(\kappa)} \left(\frac{\pi(\mathbf{X}^{(\kappa)}, v^{(\kappa)})}{N^{(\kappa)}} - 1 - \log \left(\frac{\pi(\mathbf{X}^{(\kappa)}, v^{(\kappa)})}{N^{(\kappa)}} \right) \right),$$

which is a common choice for this type of application of feedforward networks [7]. More specifically the reason that Poisson deviance loss is suitable, is the underlying Poisson model assumption 1.1. For details on the implementation of these types of networks see chapter 2.

In our model we will assume that claims for different policyholders occur independently from each other, which means that we do not implement exposure v with an input neuron in the same way as numerical features. We implement it as a multiplication of the networks output with the exposure,

$$\pi(\mathbf{X}, v) = v \cdot \pi^*(\mathbf{X}), \tag{3.1}$$

where π denotes the entire predictor, and π^* is the feedforward network in the proper sense of the term, which predicts probability of a claim to occur given unit exposure, $\pi^*(\mathbf{X}) = \pi(\mathbf{X}, 1)$.

3.2 Architectures

The goal of our network is to predict the number of claims, this means that as for output neurons it is appropriate to use one output neuron with Poisson activation function (2.6). In section 2.3.3 we discussed this in more detail.

For input neurons, since we have both one-hot encoded categorical features and numerical features the choice of design is less evident. The naive choice of having all features fed to the network in the same input layer is a possibility but since one-hot encoded categories, where several input neurons represents one feature and numerical features are represented only by one neuron there is reason to believe that this is not optimal [1].

Instead we will use the first layer for categorical feature inputs, which will be connected to a number of neurons in the second layer. These neurons will constitute a numerical representation of the categorical features. Since the inputs can only have values one and zero we can use simple linear activation functions and there is no need to use bias terms (for the same reasons as in the encoder part of autoencoders, discussed in section 2.4.4). These representations will be concatenated with the numerical inputs and be connected to the next layer which will be a hidden layer with no inputs or outputs.

Next, we can choose whether we want to use only one hidden layer or several hidden layers before the output layer, consisting of a single neuron. For the hidden layers it is suitable to use a more complex activation function than the linear activation function. One common choice of activation function for this or these layers is the hyperbolic tangent function. We also note that in contrast to the first layer, we use bias terms in all connections.

For the first input layer, since the inputs are one-hot encoded one could argue that this grouping of inputs should be reflected in the network architecture by dividing the second layer neurons into subsets representing a feature so that inputs representing the same features are only connected to this one subset in the second layer. Then, for each categorical feature one would have to choose the number of neurons in the second layer that should represent the feature.

The other obvious design choice to make is to just connect all input neurons in the first layer to the second layer, see figure 3.1. The advantage of using a full connection between the inputs in the first layer and their representation in the second layer is that the network can then take into account potential correlations between different categorical features already in the second layer. This discussion is analogous to the one in section 2.4.4 on whether one should use separate autoencoders for each feature or just one autoencoder for all features. This latter design is the one that we will use, since attempts by Delong&Kozak(2020)[1] indicate that this design is the more promising one.

3.3 Improving prediction using autoencoders

The purpose of training autoencoders is to use representations given by the hidden layer in the autoencoders as initial values for some of the weights in our network. The architecture of the network for predicting number of claims to which we will use autoencoders to initialize weights, is of the type discussed in the previous section, with input neurons for the one-hot encoded categorical features in the first layer and input neurons for the numerical features in the second layer. It is for these two first layers that using initial weights obtained from autoencoders may turn out to be useful for improving the performance of the network in predicting the number of claims.

We train the autoencoder for categorical features, see section 2.4.4 for details. We use the hidden layer's neuron's values as representations of the

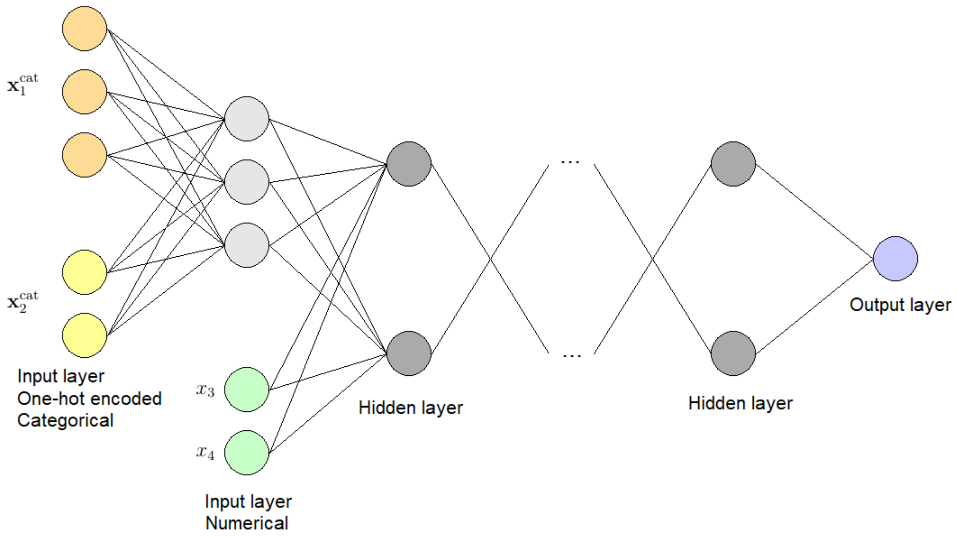


Figure 3.1: Architecture for net predicting number of claims with both one-hot encoded categorical variables and numerical variables as inputs.

categorical input. After that we re-scale the weights of this autoencoder so that for all inputs the output remains the same, but the neuron values that constitute the representation given by the hidden layer is in the interval $[-1,1]$. This rescaling process will be outlined in more detail in the next section.

Then we use these representations together with the numerical input values to train a second numerical autoencoder. To be more precise, the representations are obtained by feeding the data to inputs in the first layer, and for each sample fed to the network, the values of the neurons in the next layer concatenated with numerical inputs, forms a new sample in the new training set that we use to train the numerical autoencoder.

We use the weights between the input layer and the hidden layer in the categorical autoencoder as initial values of the weights in the first layer in the neural network. The weights in the numerical autoencoder between the input and the hidden layer we use as initial values for the weights in the second layer in our network.

After that we can train our network in the same way as if we were not using autoencoders.

3.3.1 Re-scaling the weights for the categorical autoencoder

We want to re-scale the weights in the autoencoder so that the neurons in the hidden layer only take values between 1 and -1. The reason for this is that the numerical features are scaled to $[-1{,}1]$, and when we use the encoder weights for initializing weights between the categorical input layer and the concatenation layer, we want (before training the network) all the neurons in the concatenation layer to take values on the same scale for all (properly scaled) input patterns.

Re-scaling of the weights $\mathbf{w}^{\text{enc}}$ in the encoder part of the categorical autoencoder, i.e. between the input layer and the hidden layer, can be done[1] by the transformation

$$\begin{aligned} w_{i,j}^{\text{enc}} \mapsto w_{i,j}^{*,\text{enc}} &= \frac{2}{\max_p\{x_i^{(p),\text{enc}}\} - \min_p\{x_i^{(p),\text{enc}}\}} w_{i,j}^{\text{enc}} \\ &\quad - 2\frac{\min_p\{x_i^{(p),\text{enc}}\}}{c(\max_p\{x_i^{(p),\text{enc}}\} - \min_p\{x_i^{(p),\text{enc}}\})} + \frac{1}{c}, \end{aligned}$$

so that for pattern $\lambda = \text{argmax}_p\{x_i^{(p),\text{enc}}\}$ and $\langle \mathbf{w}_i, \mathbf{x}^{(\lambda),\text{cat}} \rangle = \sum_{k=1}^{\bar{m}_c} x_k^{(\lambda),\text{cat}} w_{i,k}$ we get

$$\begin{aligned} \langle \mathbf{w}_i^*, \mathbf{x}^{(\lambda),\text{cat}} \rangle &= \frac{2}{\max_p\{x_i^{(p)}\} - \min_p\{x_i^{(p)}\}} \sum_{k=1}^{\bar{m}_c} x_k^{(\lambda),\text{cat}} w_{i,j} \\ &\quad - \frac{1}{c}\left(\frac{2\min_p\{x_i^{(p)}\}}{\max_p\{x_i^{(p)}\} - \min_p\{x_i^{(p)}\}} - 1 \right) \sum_{k=1}^{\bar{m}_c} x_k^{(\lambda),\text{cat}} \\ &= \frac{2}{\langle \mathbf{w}_i, \mathbf{x}^{(\lambda),\text{cat}} \rangle - \min_p\{x_i^{(p)}\}} \langle \mathbf{w}_i, \mathbf{x}^{(\lambda),\text{cat}} \rangle \\ &\quad - \frac{1}{c}\left(\frac{2\min_p\{x_i^{(p)}\}}{\langle \mathbf{w}_i, \mathbf{x}^{(\lambda),\text{cat}} \rangle - \min_p\{x_i^{(p)}\}} - 1 \right) c \\ &= \frac{2\langle \mathbf{w}_i, \mathbf{x}^{(\lambda),\text{cat}} \rangle}{\langle \mathbf{w}_i, \mathbf{x}^{(\lambda),\text{cat}} \rangle - \min_p\{x_i^{(p)}\}} - \frac{2\min_p\{x_i^{(p)}\}}{\langle \mathbf{w}_i, \mathbf{x}^{(\lambda),\text{cat}} \rangle - \min_p\{x_i^{(p)}\}} + 1 \\ &= 1, \end{aligned}$$

where we used that $\sum_{k=1}^{\bar{m}_c} x_k^{(\lambda),\text{cat}} = c$ since our categorical inputs consists of one-hot encoded features there are always c elements with value 1 and the other elements have value 0.

With analogous computations one can verify that with $\eta = \operatorname{argmin}_p\{x_i^{(p),\text{enc}}\}$ one gets

$$\langle \mathbf{w}_i^*, \mathbf{x}^{(\eta),\text{cat}} \rangle = ... = -1.$$

We also need to re-scale the decoder part of the autoencoder[1]:

$$\begin{aligned} w_{i,j}^{\text{dec}} \mapsto w_{i,j}^{*,\text{dec}} &= \frac{\max_p\{x_j^{(p),\text{enc}}\} - \min_p\{x_j^{(p),\text{enc}}\}}{2} w_{i,j}^{\text{dec}} \\ \theta_i^{\text{dec}} \mapsto \theta_i^{*,\text{dec}} &= \theta_i^{\text{dec}} + \sum_{k=1}^{l} (w_{i,j}^{*,\text{dec}} + \min_p\{x_j^{(p),\text{enc}}\} w_{i,j}^{\text{dec}}). \end{aligned}$$

Since we only use the encoder part of the autoencoder to initialize weights and we do the re-scaling after training the network, there is no need to actually compute the re-scaling of the decoder weights in practice.

Chapter 4

Quality assessment of predictors

In this chapter we describe two methods for evaluating the quality of predictors. The notation we use when describing them is adapted for our particular application of them, although the methods are general statistical evaluation tools and not restricted to actuarial applications.

We assume that we have a set of features $X_1, ..., X_p$ which we denote $\mathbf{X}$ for short and a response, which in our case is the number of claims N. We are interested in modeling the relation

$$\mu(\mathbf{X}) = E[N|\mathbf{X}] \tag{4.1}$$

In actuarial applications generally we often think of $\mathbf{X}$ as information about policyholders and $\mu(\mathbf{X})$ as the pure premium. However, in this thesis we focus on modeling just the number of claims N, so we consider cases where the response is the number of claims N, and hence we think of $\mu(\mathbf{X})$ not as the pure premium but as the true expected number of claims.

In practice the function $\mathbf{X} \mapsto \mu(\mathbf{X})$ is unknown in general, and we try to model it with a simpler function $\pi(\mathbf{X})$. The quality of a method for prediction can thus be assessed using the pairs $(\mu(\mathbf{X}), \pi(\mathbf{X}))$. For insurance applications one factor which is almost always relevant for predictions is the time policyholders are exposed to risk - duration v. If v is assumed to be known, we could treat it is as one of the features in $\mathbf{X}$, but because of its importance we write it explicitly as $\mu(\mathbf{X}, v)$ and $\pi(\mathbf{X}, v)$. We note that one often assumes that the risk is linearly related to duration and it is common to use models where this relation holds, i.e. $\pi(\mathbf{X}, v) = v\pi(\mathbf{X}, 1)$.

In this chapter we assume that we have n observations $(N_i, \mathbf{X}_i, v_i)_{i=1}^n$ and a method for prediction $\pi(\mathbf{X}, v)$ that we want to assess.

4.1 Risk-ordered predictions compared with empirical responses

The following method is similar to the binned response plots described and used by Delong et al. [14] and Lindholm et al. [4].

First we want to obtain a risk order from the predictions. This is done by ordering the predictions for all observations, but with duration v set to 1, from smallest to largest. We define the set of integers $I = \{1, 2, ..., n\}$ representing the indices. Next, we define an integer-valued function $\rho(i) : I \to I$ to keep track of the risk order, where $\rho(i)$ if such that

$$\pi(\mathbf{X}_{\rho(i)}, 1) \leq \pi(\mathbf{X}_{\rho(i+1)}, 1) \text{ for } i = 1, ..., n-1 \tag{4.2}$$

holds. This risk order we will use to compare $(\pi(\mathbf{X}_i, v_i))_{i=1}^n$ with $(N_i)_{i=1}^n$. In order to do that we will order $(\pi(\mathbf{X}_i, v_i))_{i=1}^n$ according to the risk order $\rho(i)$, and put them into b equally large bins of size n/b, where b (and n) is chosen such that n/b is an integer, where we compute the mean for each bin. We apply the same procedure to $(N_i)_{i=1}^n$. We define bin functions

$$\beta_\pi(k) = \frac{1}{n/b} \sum_{i=\frac{n}{b}(k-1)+1}^{\frac{n}{b}k} \pi(\mathbf{X}_{\rho(i)}, N_{\rho(i)}) \qquad \text{for } k = 1,...,b$$

$$\beta_N(k) = \frac{1}{n/b} \sum_{i=\frac{n}{b}(k-1)+1}^{\frac{n}{b}k} N_{\rho(i)} \qquad \text{for } k = 1,...,b$$

Plotting $\beta_\pi(k)$ as a curve and $\beta_N(k)$ as dots we can see how well our prediction method π works depending on the size of the risk. Examples of this can be seen in figures 5.2 and 5.3.

4.2 Concentration curves

Another method for assessing the quality of methods for prediction is to use concentration curves. This is particularly useful when we want to compare performance of several prediction methods $\pi_1, \pi_2, ...$, since we can easily plot the concentration curve for each one of them in the same plot and compare their performances for different risk sizes. For example we could observe that one method, π_1, better predicts events for low-risk areas in the feature space than π_2, but π_2 may yield better predictions than π_1 for high-risk areas in the feature space.

We consider a method π with distribution function $F_\pi(z) = \mathrm{P}[\pi(\mathbf{X}, v) \leq z]$ for $z \geq 0$. We define the concentration curve [3] of $\mu(\mathbf{X}, v)$ with respect to π based on the information contained in $\mathbf{X}$ and v as

$$\alpha \mapsto \mathrm{CC}[\mu(\mathbf{X}, v), \pi(\mathbf{X}, v); \alpha] = \frac{\mathrm{E}[\mu(\mathbf{X}, v)\mathrm{I}(\pi(\mathbf{X}, v) \leq F_\pi^{-1}(\alpha))]}{\mathrm{E}[\mu(\mathbf{X}, v)]} \tag{4.3}$$

Using our definition (4.1) and the law of total expectation we see that

$$\begin{aligned}\mathrm{E}[\mu(\mathbf{X}, v)\mathrm{I}(\pi(\mathbf{X}, v) \leq F_\pi^{-1}(\alpha))] &= \mathrm{E}[\mathrm{E}[N|\mathbf{X}, v]\mathrm{I}(\pi(\mathbf{X}, v) \leq F_\pi^{-1}(\alpha))]\\ &= \mathrm{E}[\mathrm{E}[N\mathrm{I}(\pi(\mathbf{X}, v) \leq F_\pi^{-1}(\alpha))|\mathbf{X}, v]]\\ &= \mathrm{E}[N\mathrm{I}(\pi(\mathbf{X}, v) \leq F_\pi^{-1}(\alpha))]\end{aligned}$$

and hence it is possible to use N instead of $\mu(\mathbf{X}, v)$,

$$\mathrm{CC}[\mu(\mathbf{X}, v), \pi(\mathbf{X}, v); \alpha] = \mathrm{CC}[N, \pi(\mathbf{X}, v); \alpha] = \frac{\mathrm{E}[N\mathrm{I}(\pi(\mathbf{X}, v) \leq F_\pi^{-1}(\alpha))]}{\mathrm{E}[\mu(\mathbf{X}, v)]}.$$

If we assume the our samples $(N_i, \mathbf{X}_i, v_i)_{i=1}^n$ to be independent and identically distributed, we can estimate the concentration curve:

$$\begin{aligned}\widehat{\mathrm{CC}}[\mu(\mathbf{X}, v), \pi(\mathbf{X}, v); \alpha] &= \widehat{\mathrm{CC}}[N, \pi(\mathbf{X}, v); \alpha]\\ &= \frac{n}{\sum_{i=1}^n N_i} \frac{1}{n} \sum_{i|\hat{\pi}(\mathbf{X}_i, v_i) \leq \hat{F}_\pi^{-1}(\alpha)} N_i\\ &= \frac{\sum_{i|\hat{\pi}(\mathbf{X}_i, v_i) \leq \hat{F}_\pi^{-1}(\alpha)} N_i}{\sum_{i=1}^n N_i}\\ &= \left\{\hat{F}_\pi^{-1}(z) = \frac{1}{n}\sum_{i=1}^n \mathrm{I}(\hat{\pi}(\mathbf{X}_i, v_i) \leq z)\right\}\\ &= \frac{1}{\sum_{i=1}^n N_i} \sum_{i|\hat{F}_\pi(\hat{\pi}(\mathbf{X}_i, v_i)) \leq \alpha} N_i\\ &= \frac{1}{\sum_{i=1}^n N_i} \sum_{i|[\frac{1}{n}\sum_j^n \mathrm{I}(\hat{\pi}(\mathbf{X}_j, v_i) \leq \hat{\pi}(\mathbf{X}_i, v_i))] \leq \alpha} N_i,\end{aligned} \tag{4.4}$$

where $\hat{\pi}$ denotes the estimated predictor.

In our applications we focus on the frequency of claims N/v (number of claims divided by duration for the corresponding contracts). Thus we compare frequency of claims $\mu(\mathbf{X}, v)/v$ with predictions where we set $v = 1$: $\pi(\mathbf{X}, v) = \pi(\mathbf{X}, 1)$. Inserting this into (4.4), we get

$$\widehat{\mathrm{CC}}[\mu(\mathbf{X}, 1), \pi(\mathbf{X}, 1); \alpha] = \frac{1}{\sum_{i=1}^n \frac{N_i}{v_i}} \sum_{i|[\frac{1}{n}\sum_j^n \mathrm{I}(\hat{\pi}(\mathbf{X}_j, 1) \leq \hat{\pi}(\mathbf{X}_i, 1))] \leq \alpha} \frac{N_i}{v_i}. \tag{4.5}$$

Note that the risk order in (4.5), $i|\hat{F}_\pi(\hat{\pi}(\mathbf{X}_i, 1)) \leq \alpha$, is the risk order (4.2) expressed cumulatively (corresponding to the set of indices $i|\rho(i)/n \leq \alpha$).

Since we have different durations for different contracts in our data set, by setting duration $v = 1$ we compare the just predictive power of the method μ. Otherwise the distribution of durations in the data set (which are just given in the data, and thus completely unrelated to the methods for prediction) would influence the concentration curve. However we note that the difference in duration for different data points influences our methods of prediction μ in that points $\mathbf{X}$ with more duration v in the data set are better estimated than points $\mathbf{X}$ with less duration v.

Furthermore we note that our prediction methods work by computing a probability of claim per time unit and then multiplying by duration (3.1), so setting $v = 1$ makes sense for risk ordering.

Examples of usage of concentration curves can be seen in figure 5.4.

4.2.1 Modified concentration curves

Although risk-ordering with durations set $v = 1$ is reasonable, using $\mu(\mathbf{X}, v)/v$ relies on the assumption (1.1). Since this assumption most likely only holds approximately at best, using $(N_i/v_i)_{i=1}^n$ in (4.5) probably introduces noise into the evaluation. We therefore also try a modified version of concentration curves, where we risk order as in (4.5) but use $(N_i)_{i=1}^n$ instead of $(N_i/v_i)_{i=1}^n$,

$$\begin{aligned}\widehat{\mathrm{CC}}_{\mathrm{mod}}[\mu(\mathbf{X}, v), \pi(\mathbf{X}, v); \alpha] &= \widehat{\mathrm{CC}}[\mu(\mathbf{X}, v), \pi(\mathbf{X}, 1); \alpha] \\ &= \frac{1}{\sum_{i=1}^n N_i} \sum_{i|[\frac{1}{n}\sum_j^n \mathrm{I}(\hat{\pi}(\mathbf{X}_j,1) \leq \hat{\pi}(\mathbf{X}_i,1))] \leq \alpha} N_i.\end{aligned}$$

Chapter 5

Application

5.1 The data set

To investigate the methods discussed in previous sections we apply the methods to a data set called freMTPL2freq (French Motor Claims Datasets), available via the R package CASdatasets, consisting of 678 013 observations. The set contains data about car insurance policies. For each contract we have data about various features of the policyholders, exposure period and number of claims during the exposure period.

Following Delong&Kozak(2020)[1] we use 100 000 randomly sampled observations (contracts), in order to keep computations less time consuming.

A common practice in machine learning is to divide the sample into three types of subsets:

- A **training set** which is used to train the network.
- A **validation set** which is used to evaluate the network's performance as we vary different parameters.
- A **test set** is then used to make the final evaluation of the performance, using the parameters that performed the best on the validation set.

We divide our set of 100 000 samples into five subsets consisting of 20 000 samples each.

For the preliminary experiment, where we compare performance of different types of autoencoders, we perform training and evaluation on each of our five subsets. This is the same approach as Delong&Kozak(2020)[1]. Since the purpose of training autoencoders is to obtain a low dimensional numerical representaion of categorical features, assuming that each combination of levels of the categories are represented to a similar extent in each of the five subsets, there is no advantage performing the evaluation on an "out-of-sample" validation set compared to the "in-sample" approach we use.

For predicting the number of claims, we divide our 100 000 samples into a set of 80 000 samples which we use as training set and validation set alternately in a 4-fold cross validation process. The validation sets thus consist of 20 000 samples and training sets of 60 000 samples. The remaining 20 000 samples we use as a test set.

The features we use are described in 5.1.

Feature	Feature name in data set	Type	No. levels
Area code.	Area	Categorical	6
The power of the car.	VehPower	Categorical	6
Age of the car (years).	VehAge	Categorical	3
Driver's age (years).	DrivAge	Categorical	7
Brand of the car.	VehBrand	Categorical	11
Region (areas are subdivided into regions).	Region	Categorical	21
Bonus malus (bonussystem depending on environmental impact of the vehicle):50-350 <100 means bonus, >100 means malus.	BonusMalus	Numerical	-
The density of inhabitants where the driver of the car lives.	log-Density	Numerical	-
Gas: diesel or regular.	VehGas	Binary	2

Table 5.1: Features in the data set freMTPL2freq that we use, description, type and for categorical features, number of levels.

5.1.1 A note on reliability

Since our data is generated by purchases of insurance contracts it is likely there are several combinations of features where we do not have any samples. Hence we do not know anything about the so-called true probability of these contracts resulting in claims. It is reasonable to require our model to handle these cases in a smoothing way, so that features in the data that are unrepresented in the training set are predicted to result in a number of claims close to the number of resulting claims to input data that is similar but represented in the training data.

For example, for a specific value on Area, VehPower, VehAge, DrivAge and Region, we may only have data for 2 vehicle brands. Then for contracts with the other 9 vehicle brands, and all other data the same, we want the model to predict a number of claims similar to the number of claims predicted for the data where the categories, except vehicle brand, are the same.

We also note that combinations of the categorical features that we do have contracts for (input that it is represented in our training data) have different amounts of exposure. The more exposure we have for a certain input the more we can trust in the observed number of claims to be a good representation of the underlying generating distribution $\mu(\mathbf{X})$ for that input. We want our model $\pi(\mathbf{X})$ to take this into account.

5.1.2 Overview of the data

In table 5.2 we present some summary statistics for each of our five 20 000 sample subsets of the total 100 000 samples.

We check exposures for each subset and category (we do not present the numbers here since it would take up a large amount of space but is only of peripheral interest), and we note that for each level in each category there seems to be a fairly similar amount of exposure. Thus we can assume that dividing the data into these subsets will cause no bias related to misrepresentation.

	Set 1	Set 2	Set 3	Set 4	Set 5	All sets
No. claims:						
Total	1071	1103	1002	1117	1050	5343
Mean	0.05355	0.05515	0.05010	0.05585	0.05250	0.05343
Variance	0.05539	0.05871	0.05330	0.06553	0.05565	0.05772
Observed:						
0 claims	18976	18956	19053	18965	19007	94957
1 claim	977	989	894	963	938	4761
2 claims	47	53	51	70	53	274
3 claims		1	2	1	2	6
4 claims						0
5 claims		1				1
6-10 claims						0
11 claims				1		1
>11 claims						0

Table 5.2: Summary statistics for the data set and its subsets. Note that the samples have different exposure times.

5.1.3 Preparation of the data

Before we start modeling, we transform the data using min-max scaling for the numerical data and binary data. For each numerical feature it maps the largest value represented in the data to 1 and the smallest value to -1. The other values in the data are mapped linearly onto $(-1{,}1)$. For binary variables the value $True$ is mapped to 1 and $False$ is mapped to -1. This transformation is done to the entire data set consisting of 100 000 samples, not to the training set, validation set and test set independently. Bonus-malus is capped at 150.

For categorical features we use one-hot encoding as described in section 2.3.

The preparation and cleaning of the data is the same as Delong&Kozak(2020)[1], which in turn follows Wüthrich (2019) [11] where the process is described in detail.

5.2 Types of autoencoders

We start by trying different types of autoencoders for categorical data and evaluate them by two measures which we call marginal precision and joint precision. These are the same measures as Delong&Kozak(2020)[1] use, so we can easily compare our results with theirs. Marginal precission is the percentage of correctly classified features (each correctly classified feature is weighted by dividing by the number of levels of the feature). Joint precision is the percentage of correctly classified samples – all features for a correctly classified sample are correctly reproduced by the autoencoder.

We try different numbers of neurons in the hidden layer of the autoencoder, two different numbers of epochs in the training process (15 epochs and 500 epochs), and the three different architectures outlined in section 2.4.4. namely:

1. Separate autoencoders for each feature (**separate**),
2. One autoencoder in total, with one softmax for all output (**softmax all**),
3. One autoencoder in total, with one softmax per feature among the outputs (**softmax each**).

Here we use all 100 000 samples and compute marginal and joint precision on each of our five subsets of the total 100 000 samples.

5.2.1 Implementation from scratch in C

We start by implementing the networks and training algorithms – backwards propagation for computing the gradient and NADAM for optimizing the loss function, as described in section 2.2 – from scratch in C without external libraries. In table 5.3 we show the results.

Although implementation from scratch is useful for understanding the underlying concepts of artificial neural networks, it turned out to be relatively slow to run. Since optimizing program code is note the focus on this thesis, we chose only to try out a few combinations of hyperparameters using this C-code and then proceed to use Python with Keras (runned on top of Tensorflow), which are well established libraries for both industrial and academic applications.

The results from our C implementation of autoencoders (table 5.3) seem to correspond well with the results that Delong&Kozak(2020)[1] obtained, which is an indication that the implementation is correct, but not optimal from a programming perspective.

Neurons in hidden layer	Epochs	Learning rate	Network type	Joint precision	Marginal precision
6	15	0.001	separate	0.114%	25.99%
6	15	0.001	softmax all	0.44%	33.65%
6	15	0.001	softmax each	0.264%	37.75%
6	500	0.001	softmax all	18.053%	68.95%
6	500	0.001	softmax each	76.64%*	91.58%*

Table 5.3: Performance of autoencoders using code implemented from scratch in C. On each of our 5 data sets we train an autoencoder and evaluate the joint precision and marginal precision on the same training set. The performance measures in this table are the averages of the results on the five data sets. For the run marked with an asterisk the autoencoder was only trained and evaluated on one data set.

5.2.2 Implementation using Python with Keras

The type of architecture that seems to yield best results from our limited trials with autoencoders implemented in C was "softmax each". Since this is in accordance with Delong&Kozak(2020)[1], for the Python/Keras implementation we restrict ourselves to only considering this type of network.

We compute marginal and joint precision on each of our five subsets of the total 100 000 samples, for 15 epochs and 500 epochs. For the learning rate hyperparameter, we use values 0.01 and 0.001. As for the number of neurons, we try autoencoders with 6, 8, 10, 12, 15, 20 and 30 neurons in the hidden layer. The result can be seen in figure 5.1.

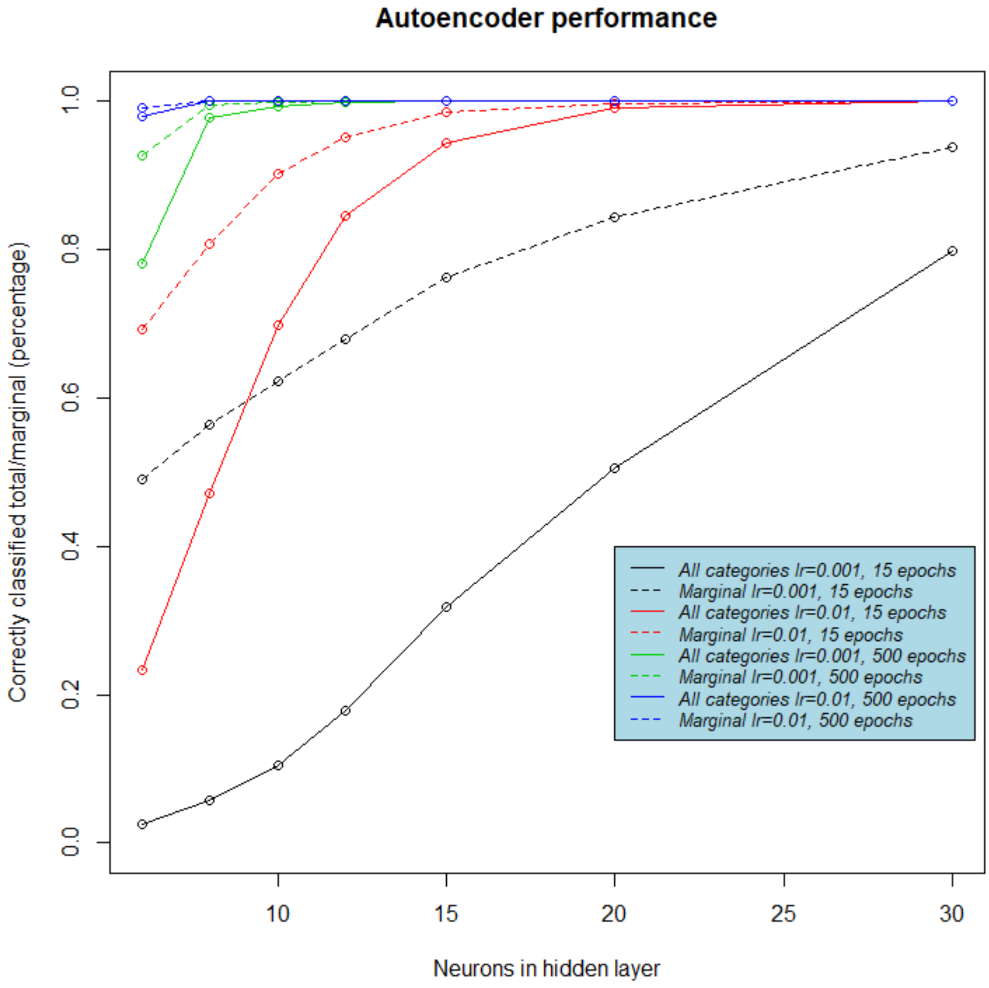


Figure 5.1: Performance of autoencoders with 6, 8, 10, 12, 15, 20 and 30 neurons in the hidden layer, using Python and Keras. On each of our 5 data sets we train an autoencoder and evaluate the joint precision and marginal precision on the same set. The performance measures in this table are the averages of the results on the five data sets. Here we only use the autoencoders of type "softmax each".

More neurons in the hidden layer yields better performance, and longer training times yields better performance – this seems reasonable. We also note that our results are very similar to the results obtained by Delong&Kozak(2020)[1], so we agree with their conclusion to use 8 neurons in the hidden layer in further experiments.

5.3 Predicting number of claims

For predicting the number of claims using empirical data we try three different methods:

1. Gradient boosting machines
2. Feedforward networks
3. Feedforward networks with (some) weights initialized using autoencoders

Since each of these methods contains several hyperparameters, we use 4-fold cross validation on the first 80 000 samples in our set in order to find as good hyperparameters as possible. Using the hyperparameters we found, we compare the three on an independent test set – the last 20 000 samples.

In addition to these three methods we first try an intercept model to predict future number of claims, that is we take the mean on the training set and use this as a predictor for every data point in the validation set.

We use Poisson deviance loss to measure performance of the methods, since it is not so evident what constitutes a good value of Poisson deviance loss, using a simple intercept model is useful to better understand the predictive power of the more complex models.

We present the results in table 5.4.

	Used as validation set				
	Set 1	Set 2	Set 3	Set 4	Mean
Intercept only	0.3200126	0.3281709	0.3081252	0.3350932	0.3228505

Table 5.4: Poisson deviance loss for predicting number of claims with training set intercept. 4-fold cross validation using the first 80 000 observations.

5.3.1 Gradient boosting machines

Gradient boosting machines (GBM), explained in more detail in appendix C, is not our main focus in this report but we use it as a reference since it is a fairly commonly used method for predicting number of claims.

We start by using Gaussian loss function and exposure as weights, see table 5.5. Then we try to use the Poisson loss function with exposure as log-offset, see table 5.6.

The parameter number of trees constitutes the number of iterations computed (number of trees), but we use the best iteration from a 10-fold cross validation process (this is a feature of the library, we do not do it manually) for prediction. When we use Gaussian loss function the best iteration is obtained quite early, so there is no point in trying large numbers of trees. However when we use Poisson loss function, sometimes the best iteration is obtained comparatively late (at about between iteration 10 000 and 15 000), which is why we in this case mainly use 15 000, but also include one attempt with 1000 trees for comparison. The downside of using larger numbers of trees is longer computation times.

	Used as validation set				
	Set 1	Set 2	Set 3	Set 4	Mean
intDepth=1	0.3191205	0.3283910	0.3115208	0.3350613	0.3235234
intDepth=2	0.3179736	0.3316878	0.3130488	0.3361251	0.3247088
intDepth=3	0.3281063	0.3361417	0.3099411	0.3359109	0.3275250

Table 5.5: Poisson deviance loss for predicting number of claims with GBM using Gaussian loss function and exposure as weights. Shrinkage = 0.001. 4-fold cross validation for the first 80 000 observations.

When we use Gaussian loss function, increasing the interaction depth does not seem to improve the performance, whereas in the case when we use Poisson loss function, using interaction depth = 3 gives us the best result. GBM with Gaussian loss function performs worse than the simple intercept model (table 5.4), so clearly it is not useful for this application. Just having a quick glance at the data as in section 5.1.2, the data seems to be closer to Poisson distributed rather than normally distributed, so the fact that GBM with Poisson loss function performs better is not very surprising.

The fact that increasing the interaction depth yields better results, as we observe when using Poisson loss function, is an indication that there are dependencies between the features.

	Used as validation set				
	Set 1	Set 2	Set 3	Set 4	Mean
intDepth=1, 1000 trees	0.3212994	0.3303933	0.3131945	0.3370526	0.3254849
intDepth=1, 15000 trees	0.3118076	0.3219771	0.3079024	0.3286407	0.3175820
intDepth=2, 15000 trees	0.3100158	0.3181216	0.3052694	0.3299227	0.3158324
intDepth=3, 15000 trees	0.3087078	0.3171972	0.3040827	0.3288496	0.3147093

Table 5.6: Poisson deviance loss for predicting number of claims with GBM using Poisson loss function and log-exposure as offset. Shrinkage = 0.001. 4-fold cross-validation for the first 80 000 observations.

Although there are stochastic elements of GBM, since we use GBM mainly for reference and each evaluation of a model with a certain set of hyperparameters is fairly time consuming, we choose not to evaluate them more than once each.

5.3.2 Feedforward networks

The networks we try here have the design presented in section 3.2 and are depicted in figure 3.1.

Since there are a lot of different hyperparameters we can vary, and training models is time consuming we have to make restrictions on what hyperparameters we try.

We start by trying models with three hidden layers, since this number of layers yielded good results for Delong&Kozak (2020)[1] and also autoencoders have been shown to be unsuitable for shallow and small neural networks [12] [13] - three layers is hopefully a good compromise, since we also want to keep the model small if possible.

We decide to start by trying networks with 60 neurons in total, distributed differently in the three hidden layers.

Our previous experiment regarding autoencoders for categorical features shows that 8 neurons in the layer connected to the categorical input seems suitable. From this experiment we also conclude that when we use autoencoders for categorical features, we use learning rate 0.001 and train the autoencoder for 500 epochs.

Since we have not performed any experiments regarding the choice of learning rate and number of epochs for the numerical autoencoder, we use inspiration from Delong&Kozak (2020)[1]. For three layers in the model with autoencoders without noise added that performs best for them the numerical autoencoder is trained for 200 epochs and uses learning rate 0.005. When they use denoising autoencoders in the model that performs best the numerical autoencoder is trained for 15 epochs and uses learning rate 0.005.

We decide to try models both using autoencoders and denoising autoencoders, and decide to keep the number of epochs trained and learning rates constant. The values we decide to use are learning rate 0.001 and 500 epochs for categorical autoencoders. For numerical autoencoders we use learning rate 0.005 and 200 epochs.

As discussed in section 2.4.4. on categorical autoencoders, we do not use bias terms in the layer with the neurons that constitutes the representation of the categories (the layer where we use 8 neurons). Except for this, bias terms are used in all neurons in the network, and we initialize both biases and weights for all neurons using Xavier initialization, unless the neurons are initialized with weights from autoencoders.

For the feedforward network we use the learning rate 0.0001, since this learning rate was used for most of the successful networks tried by Delong&Kozak (2020)[1].

When we use denoising autoencoders we use sample noise for categories (for $q = 2$ features per sample) and Guassian noise for numerical features (with $\sigma = 0.1$) since this was the type of noise used by Delong&Kozak (2020)[1] for their model that performed the best.

Initial attempts

A first attempt using three layers with 10, 20 and 30 neurons and autoencoders (not denoising) for 4-fold cross validation, average computed from 5 runs, and parameters as described above yields the result **0.3134550** in terms of Poisson deviance. The same network but without autoencoders gives the result **0.3138441**, and if we instead use denoising autoencoders (as described in the previous section) we get the result **0.3137780**. Although we get increased performance when we use autoencoders, training them for 500 and 200 epochs is time consuming, so it may be worth considering possibilities of reducing the number of epochs that we train the autoencoders for.

The model which gives the best performance for Delong&Kozak (2020)[1] is a model where denoising autoencoders are used – the numerical autoencoder is trained for 15 epochs with learning rate 0.005, and autoencoder for categorical features is also trained for 15 epochs with learning rate 0.005. We try these values for three layers with number of neurons 10, 20 and 30 as before, and with (non-denoising) autoencoders and get the result **0.3137979** and using denoising autoencoders we get **0.3137256**, so for non-denoising autoencoders we clearly get a worse result, but for denoising autoencoders we actually get a somewhat better result.

We also mention that we made an attempt at using early stopping with 15 epochs of patience when training autoencoders, but since autoencoders both for numerical and categorical features mostly did not stop before the 500 or 200 epochs it is not clear that stopping before 500 and 200 epochs is the best option.

Since we want to have these hyperparameters fixed (in order to limit the scope of this thesis), we decide to use 500 and 200 epochs, although we note that for denoising autoencoders it is probably possible to improve the results somewhat by using fewer epochs when training the autoencoders.

It is also possible that we could find a useful early-stopping condition if we for example use fewer epochs of patience or allow the training process to stop if improvements in Poisson deviance are small enough (we now only allow stopping if Poisson deviance ceases to decrease).

Three layers with 60 neurons

We now train different permutations of networks with 10, 20 and 30 neurons in three layers, without autoencoders, with autoencoders, and with denoising autoencoders using previously mentioned hyperparameters. We use 4-fold cross validation and 5 runs - in total 20 runs for each model, with 60 000 samples as training set and 20 000 samples as validation set.

For the feedforward network (but not for autoencoders, if used) we also use an early stopping condition. We stop if we see no improvement (on the validation set) for 15 epochs. Note that the set we use for early stopping is the same validation set we use for prediction. These 80 000 samples correspond to the sets 1-4 in table 5.2.

The 20 000 samples corresponding to set 5 in table 5.2 are used in the next step, as an independent test set to compare some of the best models with each other and also the best GBM model and the intercept model for reference.

The result of runs with three layers and permutations of 10, 20 and 30 neurons can be seen in table 5.7. We note that the best model is the model with autoencoders and 20-30-10 neurons. For models using denoising autoencoders the setting 30-20-10, which is also the setting that performs best if we use no autoencoders.

Conf. neurons	Poisson deviance ($\cdot 10^2$)		
	No AE	With AE	With denoising AE
10-20-30	31.38441 (0.86701)	31.34550 (0.86737)	31.37780 (0.84886)
10-30-20	31.40698 (0.86972)	31.33979 (0.85685)	31.35682 (0.84909)
20-10-30	31.40325 (0.88503)	31.32712 (0.86557)	31.36300 (0.85859)
20-30-10	31.36925 (0.86003)	31.31859 (0.87391)	31.34188 (0.87538)
30-10-20	31.36017 (0.86955)	31.32668 (0.86540)	31.36644 (0.85364)
30-20-10	31.36008 (0.88026)	31.32016 (0.88478)	31.33366 (0.87864)

Table 5.7: Predictions made with 4-fold cross validation, mean taken over five runs (i.e. 20 in total) and standard deviation in parentheses. The set used for checking the early stopping condition is also used for prediction here. The networks tried here all have 60 neurons in three hidden layers, we try different configurations of them (10-20-30 means 10 neurons in the first layer and 30 in the last layer). For each configuration we try using: 1. no autoencoders, 2. with autoencoders but without adding noise when training them, 3. denoising autoencoders. For denoising autoencoders we use sample noise for categories (for $q = 2$ features per sample) and Gaussian noise for numerical features (with $\sigma = 0.1$). The feedforward network is trained for a maximum of 1000 epochs (early stopping condition with 15 epochs of patience), and when autoencoders are used AE for categorical features are trained for 500 epochs with learning rate 0.001 (no early stopping), and AE for numerical features are trained for 200 epochs with learning rate 0.005 (no early stopping).

In general models using autoencoders perform better than models that use denoising autoencoders. But models that use denoising autoencoders perform better than models that do not use autoencoders.

We decide to pick three of these models for the next step, where we compare them using the independent test set: 30-20-10 without autoencoders, 20-30-10 with autoencoders, and 30-20-10 with denoising autoencoders.

Three layers: 50-35-20

Since we to a large degree are inspired by Delong&Kozak (2020)[1], it is interesting to test the the hyperparameters and models with denoising autoencoders that performed best for them: three layers with 50-35-20 neurons; for autoencoders for categories we use sample noise ($q = 2$) features per sample and train them for 15 epochs with learning rate 0.005, and for autoencoders for numerical features we use Gaussian noise $\sigma = 0.1$ and train them for 15 epochs with learning rate 0.005.

In addition to this we also try training and evaluating the model in the same way as Delong&Kozak (2020)[1]: we use 60 000 samples as training set, 20 000 samples as validation set used for determining early stopping with 15 epochs of patience (for a maximum of 1000 epochs), and Poisson deviance computed on an independent test set consisting of 20 000 samples.

In this way we train the model 10 times and compute the average Poisson deviance.

For comparison we also try the same model with modifications: 1. without using autoencoders, 2. using non-denoising autoencoders and train the autoencoders for categorical features for 500 epochs with learning rate 0.001, and for numerical features 200 epochs with learning rate 0.005, i.e. same hyperparameters as in previous experiment, but models are trained and evaluated in the same way as we do for the model with 50-35-20 neurons and denoising autoencoders.

When we use denoising autoencoders we get the result **0.3080082**, for the model without any autoencoder we get the result **0.3088068** and for the model using non-denoising autoencoders we get **0.3081206**. The setting where we use denoising autoencoders performs best, which is consistent with the result of Delong&Kozak (2020)[1].

One layer, without using autoencoders

For Delong&Kozak (2020)[1] there were some well-performing models using only one hidden layer, especially so for models where autoencoders were not used. We decide to try models with one layer without using autoencoders where the networks have 3, 4, 6, 8, 10, 15, 20, 30, 50 and 80 neurons in a single hidden layer. Except for this, the models are trained and evaluated in the same way as models with 10, 20, and 30 neurons in three layers.

We see in table 5.8 that all models have similar performance although models with only 3 and 4 neurons in the hidden layer seem to perform a bit worse. This is also true for the largest models with 50 and 80 neurons. The model using 20 neurons performs the best, **0.3137784**, and we decide to pick this model from here to include in further comparisons with the other well-performing models from previous experiments.

Number of. neurons	Poisson deviance ($\cdot 10^2$) Mean	 Std. deviation
3	31.41211	0.88134
4	31.40179	0.88170
6	31.37958	0.88205
8	31.42447	0.86552
10	31.38509	0.89553
15	31.39181	0.88311
20	31.37784	0.86005
30	31.38671	0.87705
50	31.40366	0.85271
80	31.46292	0.81731

Table 5.8: Predictions made with 4-fold cross validation, mean taken over five runs (i.e. 20 in total). The set used for checking the early stopping condition is also used for prediction here. The networks tried here all have one hidden layer and we try 10 different numbers of neurons between 3 and 80. The feedforward network is trained for a maximum of 1000 epochs (early stopping condition with 15 epochs of patience).

We also note for all runs for models with 6 or more neurons, the early stopping is activated, while for networks with 3 and 4 in a few cases the networks are trained for the maximum number of epochs (1000). So the reason that networks with more neurons, for example 50 or 80, in the hidden layer, performs worse than networks with 20 neurons in the hidden layer seems not to be because of insufficient training time for the larger networks.

5.3.3 Comparing methods

We train each of our three predictors once using the 80 000 samples previously used for 4-fold cross validation as training set (sets 1-4 in table 5.2), and then we evaluate them using our 20 000 last samples not previously used as test set (set 5 in table 5.2). For each predictor we make a risk ordering evaluation as described in section 4.1, and we compare the methods using concentration curves as described in section 4.2. As hyperparameters we use the best ones from our previous results, see table 5.9 for a summation, we also present the Poisson deviance for the runs used to create risk-ordering plots and concentration curves. The number of epochs we train networks are the average numbers of epochs based on the early stopping condition in the runs in section 5.3.2.

Predictor	Hyperparameters	Poisson deviance
Intercept only	-	0.3228505

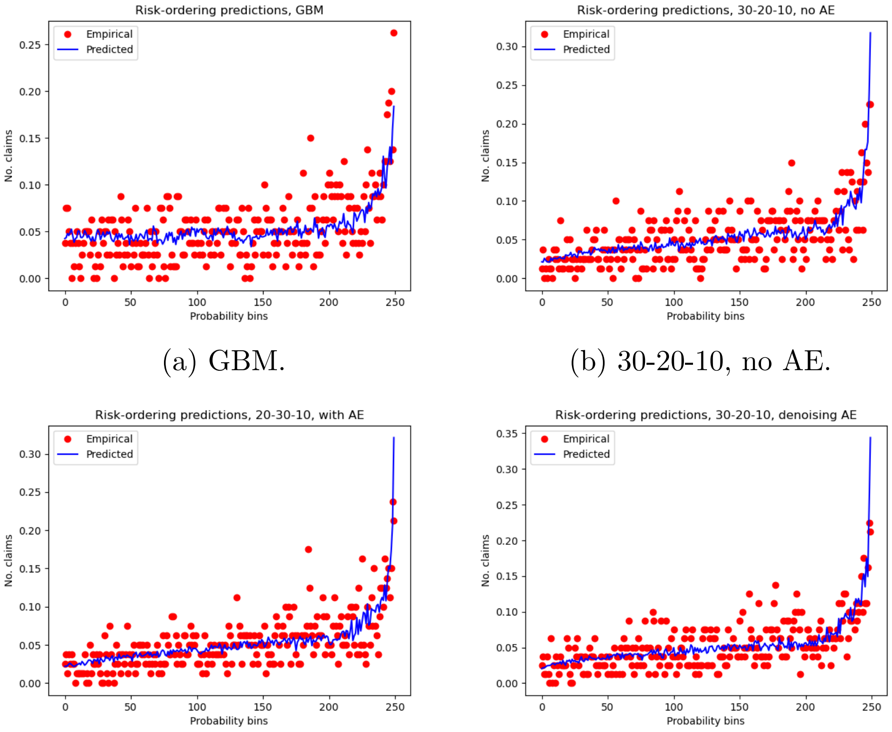
GBM	intDepth=3,
	15000 trees,
	Shrinkage = 0.001.
		0.3089879

Feedforward network	3 layers: 30-20-10, 243 epochs.	0.3083416
Feedforward network with autoencoders	3 layers: 20-30-10, 237 epochs. CatAE: 500 epochs, learning rate: 0.001. NumAE: 200 epochs, learning rate: 0.005.	0.3074430
Feedforward network with denoising autoencoders	3 layers: 30-20-10, 243 epochs. CatAE: 500 epochs, learning rate: 0.001. NumAE: 200 epochs, learning rate: 0.005.	0.3088095
Feedforward network	3 layers: 50-35-20, 183 epochs.	0.3080738
Feedforward network with autoencoders	3 layers: 50-35-20, 155 epochs. CatAE: 500 epochs, learning rate: 0.001. NumAE: 200 epochs, learning rate: 0.005.	0.3074336
Feedforward network with denoising autoencoders	3 layers: 50-35-20, 198 epochs. CatAE: 15 epochs, learning rate: 0.005. NumAE: 15 epochs, learning rate: 0.005.	0.3078815
Feedforward network	1 layer: 20, 463 epochs.	0.3083020

Table 5.9: Some of the better performing networks based on previous experiments in section 5.3.2, GBM and intercept-only models. Hyperparameters not listed in the table are described in section 5.3.2. The Poisson deviances for the networks are for the run that is used to compute risk-ordering and concentration curves, with prediction made on the independent test set.

Risk-ordering

Risk-ordering, as described in section 4.1, can be seen in plots 5.2 and 5.3.



(a) GBM.

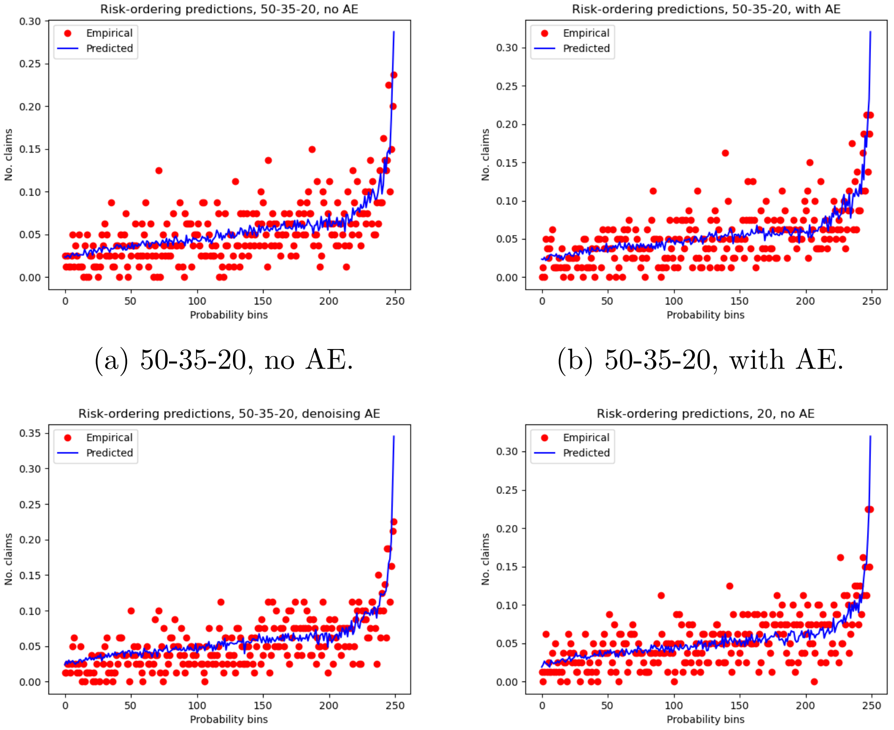
(b) 30-20-10, no AE.

(c) 20-30-10, with AE.

(d) 30-20-10, with denoising AE.

Figure 5.2: Risk-ordering for predictors listed in table 5.9 part 1/2.

In figures 5.2 and 5.3, we see no clear difference between different models. Network models compared to GBM seem to perform similarly well or slightly better. The models using denoising autoencoders, especially the model with 50-35-20 neurons, seem to perform slightly better than the other models.



(a) 50-35-20, no AE.

(b) 50-35-20, with AE.

(c) 50-35-20, with denoising AE.

(d) One layer: 20, no AE.

Figure 5.3: Risk-ordering for predictors listed in table 5.9 part 2/2.

Concentration curves for frequency of claims

To compare the performance of different predictors we also use concentration curves for frequency of claims, as described in section 4.2. The concentration curves are plotted in figure 5.4. Since the curves are pretty close to each other at many times, we present the values of the concentration curves, used to create figure 5.4, in table 5.10.

Pr.	Predictor ($\cdot 10^2$)							
	GBM	30-20-10	20-30-10 AE	30-20-10 den. AE	50-35-20	50-35-20 AE	50-35-20 den. AE	20
0.1	2.64	2.54	1.87	1.74	1.90	2.13	**1.53**	2.97
0.2	6.84	7.44	**6.18**	8.72	9.29	6.95	6.23	12.92
0.3	16.29	17.23	11.33	11.83	17.44	**10.41**	16.80	18.38
0.4	20.72	21.21	21.11	**14.53**	22.71	15.17	20.99	24.10
0.5	24.27	27.45	24.43	**18.30**	26.25	24.57	24.74	27.93
0.6	29.89	31.60	28.16	**27.18**	32.54	28.50	28.71	32.30
0.7	34.46	35.24	34.90	**34.41**	36.61	35.34	36.40	36.96
0.8	43.83	**43.02**	43.09	49.32	49.60	44.06	49.28	44.54
0.9	56.99	56.66	57.25	57.00	56.60	56.86	56.76	**50.99**
	Poisson deviance ($\cdot 10^2$)							
	30.899	30.834	30.744	30.881	30.807	**30.743**	30.788	30.830

Table 5.10: Values used for the concentration curves (frequency) in figure 5.4. Best values in bold font, and worst values underlined. We also repeat Poisson deviances from table 5.9.

Judging by figure 5.4 and table 5.10 it seems like the models using autoencoders, especially denoising autoencoders, performs best, while the network model with only one layer performs worst at most probabilities.

Compared to network models, GBM have an average performance. If we compare the model 30-20-10 with denoising autoencoders with GBM we see that GBM performs better at probabilities 0.2, 0.8 and 0.9, while for all other probabilities 30-20-10 with denoising autoencoders performs better (for 0.4, 0.5, 0.6 and 0.7 it performs better than the other network models too).

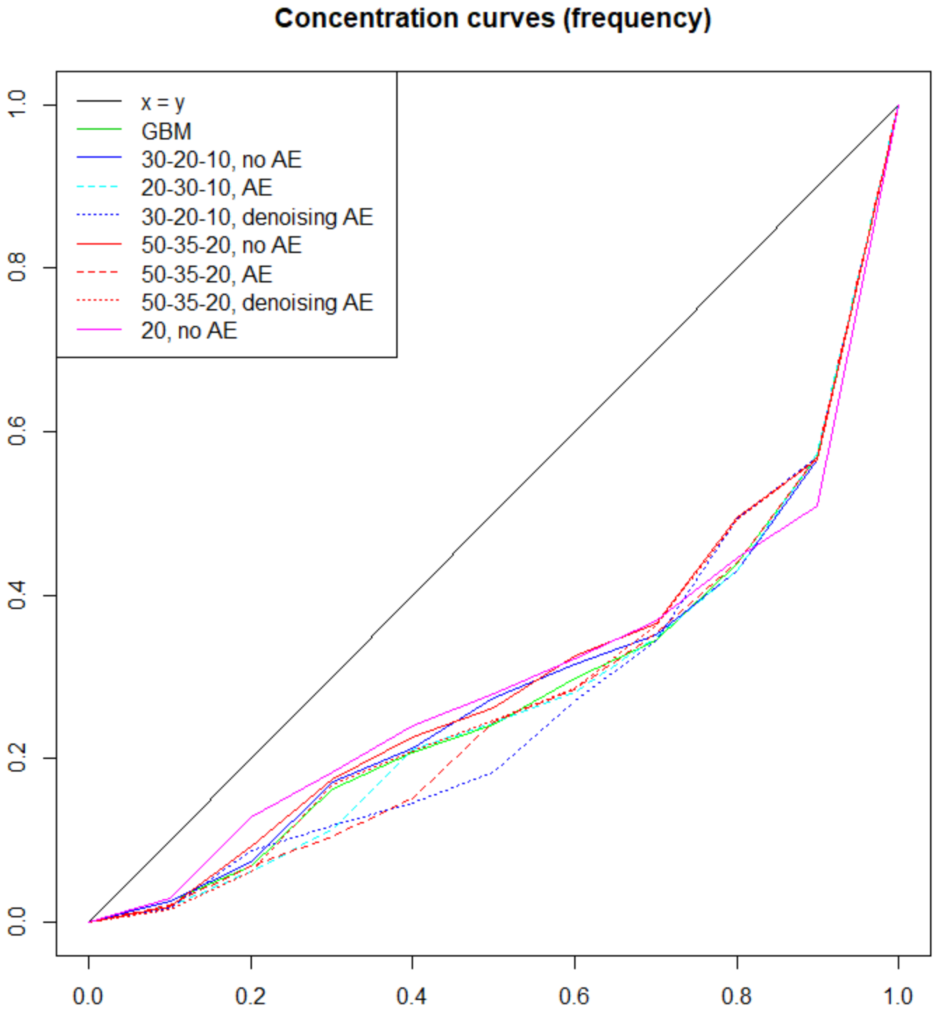


Figure 5.4: Concentration curves comparing predictions using the different predictors (frequency of claims, see section 4.2) listed in table 5.9. Since values of the curves often are close to each other, we also present these values in table 5.10.

Modified concentration curves

We note that the concentration curves for frequency of claims in figure 5.4 are not very smooth, which is an indication that our suspicion discussed in sections 4.2-4.2.1 that our assumption (1.1) about distribution of claims N only holding approximately, is probably right.

In figure 5.5 we present modified concentration curves (as discussed in section 4.2.1). Here concentration curves are done for the number of claims, but with risk-ordering by frequency of claims. As before, since values of concentration curves often are very close to each other we also present them in table 5.11.

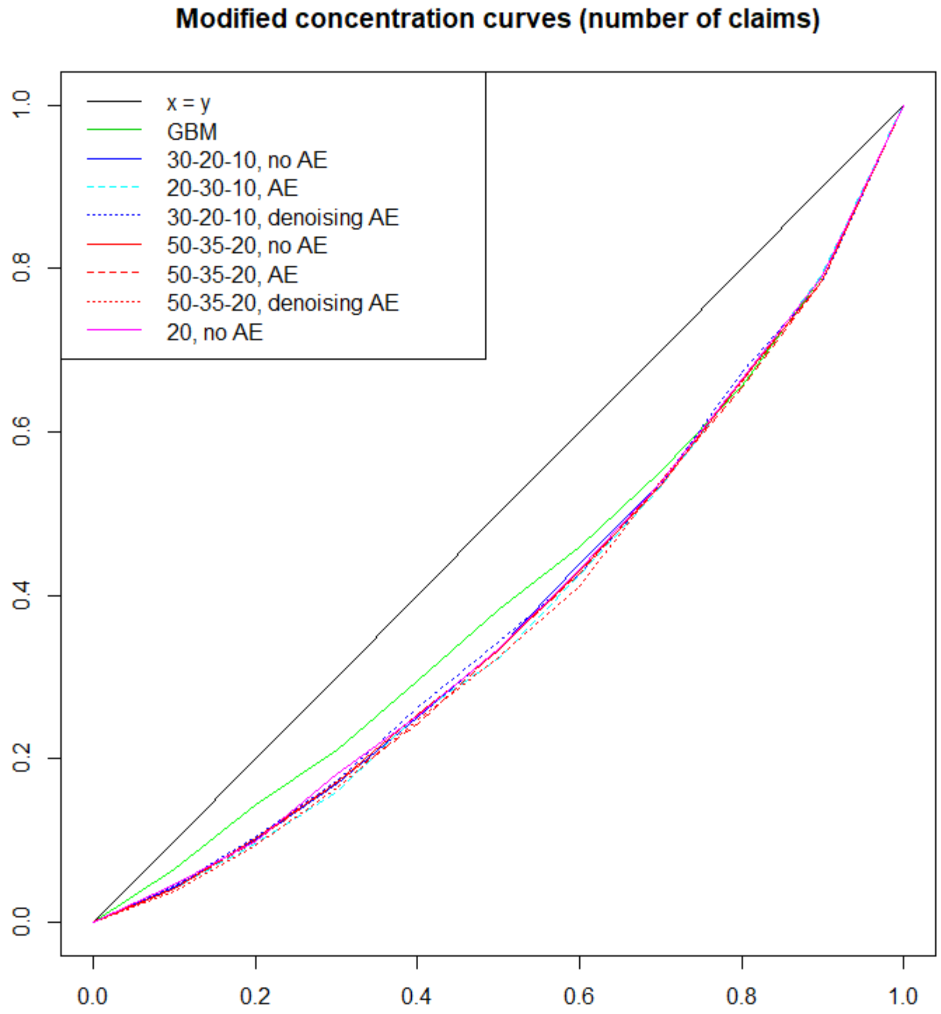


Figure 5.5: Modified concentration curves comparing predictions using the different predictors (number of claims, risk order by frequency of claims, see section 4.2.1) listed in table 5.9. Since values of the curves often are close to each other, we also present these values in table 5.11.

Pr.	Predictor ($\cdot 10^2$)							
	GBM	30-20-10	20-30-10 AE	30-20-10 den. AE	50-35-20	50-35-20 AE	50-35-20 den. AE	20
0.1	6.57	4.38	4.10	4.48	4.10	4.10	**3.81**	4.67
0.2	14.38	10.19	9.62	10.57	10.29	9.90	**9.43**	9.90
0.3	21.05	16.95	**16.00**	17.43	17.24	17.24	16.57	18.19
0.4	29.43	25.05	25.33	26.29	25.43	**24.19**	24.67	25.14
0.5	38.29	33.43	**32.29**	34.29	33.24	33.43	32.48	33.43
0.6	45.90	43.90	42.48	42.67	43.05	42.76	**41.24**	43.24
0.7	55.14	53.90	**53.33**	53.62	53.52	53.81	54.00	53.90
0.8	65.62	66.57	66.38	67.43	66.29	**65.52**	66.10	66.57
0.9	78.95	78.95	79.14	**78.38**	78.48	78.48	78.48	79.05
	Poisson deviance ($\cdot 10^2$)							
	30.899	30.834	30.744	30.881	30.807	**30.743**	30.788	30.830

Table 5.11: Values used for the modified concentration curves (number of claims, risk ordered by frequency) in figure 5.5. Best values in bold font, and worst values underlined. We also repeat Poisson deviances from table 5.9.

In figure 5.5 and table 5.11, we see that all neural networks clearly perform better than GBM, except at some high probabilities (0.8 and 0.9). Moreover we see that neural networks using autoencoders (denoising or non-denoising) mostly perform better than networks without autoencoders. It is less clear which of the networks using autoencoders that perform best, but we point out that the network with 50-35-20 neurons that uses denoising autoencoders is a good candidate. It performs best at probability levels 0.1, 0.2 and 0.6, and when it does not perform best it mostly performs better than median (of the models we tried here).

The overall impression, judging by assessments by risk-ordering and concentration curves, is that it is probably possible to find neural network models that perform better than standard methods such as GBM, if one chooses network design and hyperparameters carefully.

Predictions

We use the models previously evaluated on a validation set to measure performance on the independent test set (set 5 in 5.2), but we do not try all models - we restrict ourselves to using the seven network settings presented in table 5.9. For the three model settings with 50-35-20 neurons, we retrain them (since they were not previously trained in the same way as the other models, i.e. using 4-fold cross-validation and average of five runs – 20 models in total, and with the validation set used for determining early stopping with patience of 15 epochs).

In figure 5.6 we plot the Poisson deviances for these 7·20 models computed on the test set and the training sets. In addition we plot the average Poisson deviances for each of the seven parameter settings. The averages plotted in the figure we also present in table 5.12.

It seems like networks with (non-denoising) autoencoders performs best on the test set, but not on the training set, although the models using denoising autoencoders are pretty close to models using (non-denoising) autoencoders in terms of performance on the test set.

Over all, we se a larger variance for all models on the training set compared to the test set. This however is no doubt at least partially due to the fact that the single models (each point) for a set of hyperparameters are trained using 4-fold cross validation, and therefore only partially use the same training data. We also observe that models using autoencoders, both non-denoising and denoising autoencoders, have a smaller variance than networks were autoencoders were not used.

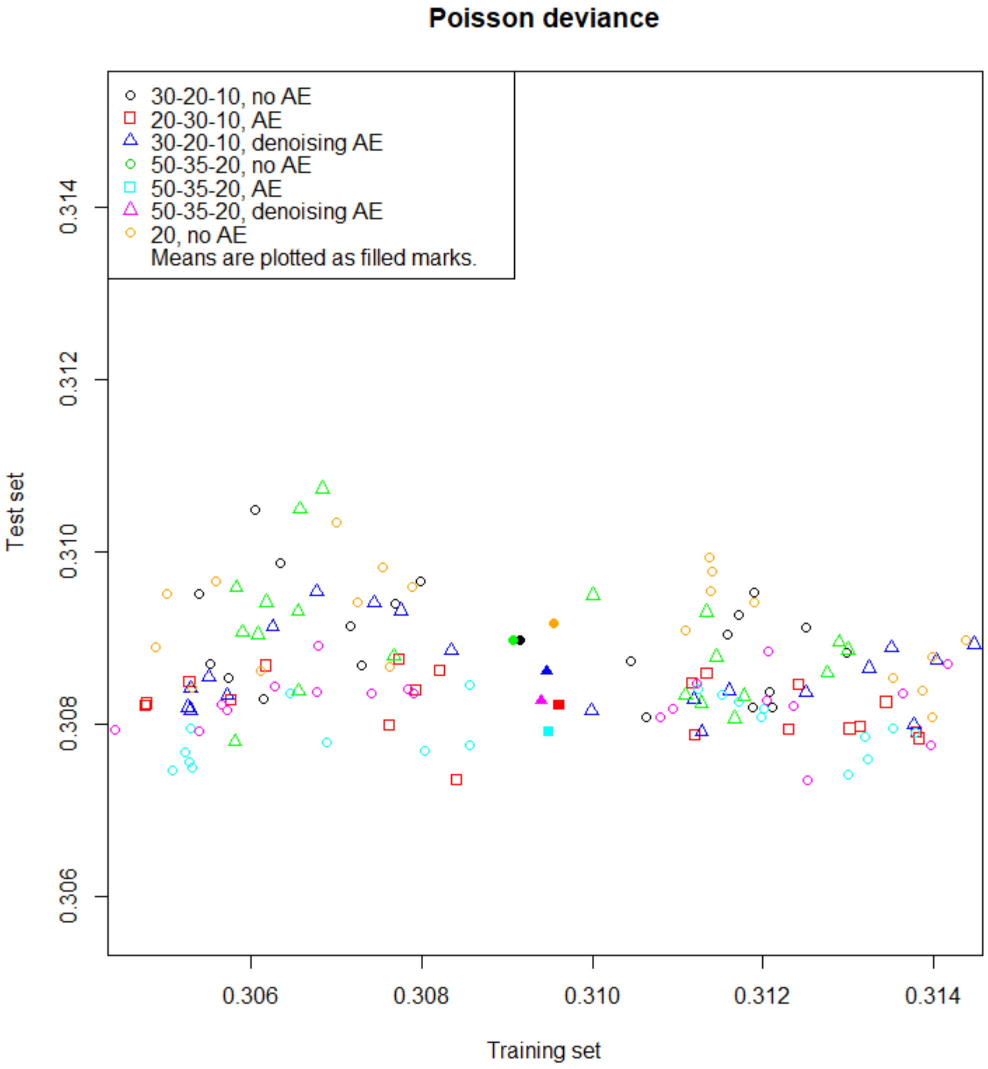


Figure 5.6: We use networks for predicting using training data and test data, and plot Poisson deviance. The models used are the same ones as in section 5.3.2, except for the three settings with 50-35-20 neurons, since they were not trained in the same way as the other models - these are trained again, but in the same way as the other models with three layers. The same scale is used on both axes.

Predictor	Poisson deviance	
	Training set	Test set
Intercept only	-	0.3228505
GBM	-	0.3089879
30-20-10	0.3091587 (0.002820922)	0.3089812 (0.0006392708)
20-30-10 with autoencoder	0.3096118 (0.003271673)	0.3082169 (0.0003565637)
30-20-10 with denoising autoencoder	0.3094618 (0.003433416)	0.3086091 (0.0004761066)
50-35-20	0.3090609 (0.002836947)	0.3089794 (0.0007491169)
50-35-20 with autoencoder	0.3094906 (0.003323964)	0.3079088 (0.0003369356)
50-35-20 with denoising autoencoder	0.3093917 (0.003268549)	0.3082644 (0.0003591402)
20, one layer	0.309553 (0.003446142)	0.3091737) (0.0006105103)

Table 5.12: The averages of predictions plotted in figure 5.6, standard deviations in parentheses. For the intercept only and the GBM models, we repeat results from table 5.9.

Chapter 6

Discussion

In this thesis we have used feedforward neural networks with both numerical and categorical features and known exposure, to predict the number of claims.

We have trained autoencoders to learn representations (numerical) of the categorical features ("one-hot encoded"), in order to use these representations as initial values for weights and biases before training the feedforward network. We have also used autoencoders to learn representations of the concatenation of representations of the categorical features with numerical features. These representations we then used as initial values for weight and biases in the next layer in the feedforward network.

In addition to this we used a well-established method, gradient boosting machines, as a reference.

Global performance

We clearly see that all models we tried clearly perform better than a simple intercept model. The results presented in table 5.12 are the most relevant when comparing models, since here we trained the networks in the same way (4-fold cross validation five times, i.e. each model was trained 20 times), and GBM was trained on the sets used for 4-fold cross validation and evaluated on the same independent test set as network models.

Here we see that all network models perform better than GBM except the one with only one single layer. Furthermore the model with 30-20-10 neurons without autoencoders only very slightly performs better than GBM. However all four models using autoencoders clearly perform better than the other network models as well as GBM. We also observe that the standard deviations between instances of models are clearly smaller for the models using autoencoders, compared to network models without autoencoders. In terms of average Poisson deviance on the independent test set, we do not observe any advantage from using denoising autoencoders compared to using only non-denoising autoencoders.

When evaluating the models on the test set we do not observe any advantage from using autoencoders, the three-layer neural networks without autoencoders perform best. This, however, is much less relevant than the evaluation on the test set and only illustrates the importance of using independent test sets for evaluation.

Another general pattern we observe is that models using more neurons (with other parameters equal such as use/non-use of autoencoders) improves performance. Models with 10+20+30 = 60 neurons perform better than the model using only one layer with 20 neurons, and models using 50+35+20 = 105 neurons perform better than models using 60 neurons. But we also see that the use of autoencoders may yield better performance than adding more neurons, for example 20-30-10 with autoencoders performs better than 50-35-20 without autoencoders.

Local performance

Although in terms of global performance as presented in table 5.12, non-denoising autoencoders performs better than denoising autoencoders, there may be local (in terms of probability level of events) advantages from using denoising autoencoders.

For example if, say, model A predicts events with probabilities between 0.1 and 0.2 badly, but very well at all other probability levels, while model B predicts reasonably well at all probability levels, model A may outperform model B in terms of global measures such as Poisson deviance. But if events at probability levels 0.1-0.2 are important (for example claim sizes may be large), it may be preferable to use model B, or to use model A in combination with some other model.

In figures 5.2 and 5.3 models using denoising autoencoders seem to have a somewhat more even performance over different probabilities compared to models, we especially note this in relation to models using only non-denoising autoencoders, since these seem to have better global performance.

Assessing the methods using concentration curves for frequency of claims, see figure 5.4 and table 5.10, we also notice this pattern to some degree. Most obvious is the fact that the network using only one layer clearly performs the worst, except at large probability levels 0.8-0.9, interestingly. For probability level 0.9 it even performs better than all other models. GBM and network models not using autoencoders seem to perform similarly well, worse than networks using autoencoders, but better than the one-layer network. As for networks using autoencoders we see that the model using 30-20-10 neurons and denoising autoencoders performs better than the model using 20-30-10 neurons and non-denoising autoencoders at most probability levels. For levels 0.4-0.7 the model with 30-20-10 neurons and denoising autoencoders even performs best of all models tried here. However for the models using 50-35-20 neurons the use of denoising autoencoders or non-denoising autoencoders seem to yield similar performance.

Since denoising autoencoders for the 50-35-20 network model were trained only for 15 epochs (but for 500 and 200 epochs for the other networks using autoencoders), there is a risk of representations not being learned properly. In section 5.2.2 where we tried performance (ability to properly re-generate the input) of different autoencoders, we saw that autoencoders with 8 neurons (which is what we used in subsequent experiments) trained for 15 epochs (with learning rate 0.01 – larger than what we later used: 0.005) only classified about 50% correctly, although in terms of the marginal measure we used about 80% were correctly classified.

This could explain why for the model using 30-20-10 neurons and denoising autoencoders outperforms the model with 20-30-10 neurons and non-denoising autoencoders, but for 50-35-20 the models using denoising autoencoders and non-denoising autoencoders they perform similarly.

On the other hand, when we investigate local performance using modified concentration curves (number of claims, but risk-ordering by frequency of claims, see figure 5.5 and table 5.11), we see somewhat different results than when we used concentration curves for frequency. As before we see clearly (even more clearly) that neural network methods outperform GBM, and we also see that networks using autoencoders perform better than networks not using autoencoders in general. However the four models using autoencoders perform similarly well. If we were to point out one model that performs best, a good candidate would be the model using 50-35-20 neurons and denoising autoencoders. It performs best at the probability levels (0.1, 0.2 and 0.6, no model performs best at 4 probability levels), it never performs worst and even when it is not the best model it mostly performs better than the median model. When we compare methods using concentration curves for frequency of claims this model did not stand out, while the model with 30-20-10 neurons and denoising autoencoders seems to be best. From results using modified concentration curves, the model with 30-20-10 neurons and denoising autoencoders seems not to be better than other models using autoencoders, so results are to some degree conflicting.

Using concentration curves for frequency of claims probably introduces noise (see section 4.2.1), but on the other hand the modified version of concentration curves used here is not a standard method so it is not clear which method is more reliable. However, with results from risk-ordering and both types of concentration curves taken together, we can draw some conclusions. Neural networks seem to be better than GBM, and networks using autoencoders seem to perform better than networks not using autoencoders. We also see some evidence that using denoising autoencoders seems to be better than using non-denoising autoencoders, but this is much less certain.

Permutations of layers with 10, 20 and 30 neurons

In the thesis we tried only a limited set of hyperparameters, mostly by varying use of autoencoders, numbers of neurons and distribution of neurons in the hidden layer(s). An interesting pattern we saw when trying permutations of 10, 20 and 30 neurons in three hidden layers (see table 5.7) was that having large numbers of neurons in early hidden layers seems to generally yield better results than networks with fewer neurons in early hidden layers . One could have theorized that having 10 neurons in the second layer would have caused worse performance due to a bottle-neck effect, this was however not always observed. For example when not using autoencoders, the setting 30-10-20 outperformed the setting 20-30-10 (which corresponds to a more smooth network design: 11-20-30-10-1, with 8 (representation of categories) + 3 (numerical and binary input) neurons in the concatenation layer and 1 neuron in the output layer). In other cases we do observe a bottle-neck effect; for example, using denoising autoencoders, 20-10-30 performs worse than the smother design 10-30-20.

The theorized rule of thumb of more neurons in early hidden layers yielding better results seem to mostly hold, but not always. For example when using denoising autoencoders, 30-10-20 performs worse than 20-10-30.

Conclusions

In conclusion autoencoders seem to increase local performance as well as global performance. Regarding whether non-denoising autoencoders or denoising autoencoders are preferable, results are conflicting. There seem to be possible advantages from using denoising autoencoders if one can find appropriate parameter settings. We recall our experiments with three layers with neuron setting 50-35-20 were networks were trained in an other way (same as Delong&Kozak(2020)[1]) where Poisson deviance were lowest for the setting using denoising autoencoders, so there seem to be possible to get improvement even on global measures by using denoising autoencoders compared to non-denoising autoencoders.

Not all combinations of categories are represented in the data set, but it is reasonable that combinations of categories that are similar to each other should have similar probabilities of accidents to occur. This well-conditioned property – small changes in input yields small changes in output – is the type of setting that denoising of autoencoders should be well suited for.

Possible improvement

There are a lot of routes not investigated in this thesis, for example one could vary hyperparameters: learning rate, the other hyperparameters related to NADAM (there are more hyperparameters than just learning rate), numbers of epochs trained, patience for early stopping, learning rates and epochs trained for autoencoders, different types of noise (and related hyperparameters) for denoising autoencoders, initializing deeper layers using autoencoders (we only initialized two layers this way).

As for the problem of how to handle categorical data, and the related problem that may arise where cardinality is large, one could also consider other techniques for learning representations of categories. Essentially any type of dimension-reducing neural network may be relevant here.

We also acknowledge a weakness in our comparison between models in section 5.3.3. For risk-ordering, and concentration curves we only trained each model once. It is possible that we could get more reliable results by training the models several times and using averages instead, however the results we obtained still seem consistent enough to be useful.

Choice of optimization algorithm

In addition to running experiments on data, this thesis includes a study of the underlying algorithms for training feedforward neural networks. Training a feedforward neural network essentially consists of solving an optimization problem. It is well-known that the objective function of an optimization problem affects what optimization algorithm to use. We would like to point out that the starting point may also affect what optimization algorithm that is suitable.

By initializing weights and biases to pre-trained values changes the starting point in the optimization, hopefully to something closer to an optimum. ADAM and NADAM algorithms for example use exponentially weighted moving averages instead of just accumulated gradients because it improves performance when applied to nonconvex objective functions, but with pre-initialized weights and biases the starting point may already be within an area where the objective is locally convex where the best, or at least a good optimum is located. In that case it may be better to use an algorithm adapted for convex problems.

For the settings we have tried where we initialize two layers using autoencoders, this is probably not very relevant since weights in all other layers, including hidden layers are randomly initialized. But since it is possible to generalize the method we use for training our numerical autoencoders, to pre-train weights of deeper layers using autoencoders, one could theoretically initialize all weights with values from pre-trained autoencoders in this way. This would be a situation where this discussion on choice of optimization algorithm could be worth considering.

Appendix A

Gradient computation

The topic of gradient computation by backwards propagation was introduced in section 2.2.3. Here we present the details.

A.1 Square distance loss function

We start by computing derivatives w.r.t. bias and weights in the last layer L.

$$\frac{\partial L}{\partial w_{nm}^{(L)}} = \frac{\partial}{\partial w_{nm}^{(L)}} \left(\sum_i \frac{1}{2}(t_i - O_i)^2 \right) = \sum_i (t_i - O_i) \cdot \frac{\partial O_i}{\partial w_{nm}^{(L)}} \tag{A.1}$$

We compute the derivative of the output w.r.t. weights.

$$\begin{aligned}\frac{\partial O_i}{\partial w_{nm}^{(L)}} &= \frac{\partial}{\partial w_{nm}^{(L)}} \left(g(a_i^{(L)}) \right) \\ &= \frac{\partial}{\partial w_{nm}^{(L)}} \left(g \left(\theta_i + \sum_j w_{ji}^{(L)} V_j^{(L-1)} \right) \right) \\ &= g'(a_i^{(L)}) \cdot \frac{\partial}{\partial w_{nm}^{(L)}} \left(\theta_i + \sum_j w_{ji}^{(L)} V_j^{(L-1)} \right) \\ &= g'(a_i^{(L)}) V_m^{(L)} \delta_{in} \end{aligned} \tag{A.2}$$

Similarly when we differentiate w.r.t. to the bias term in the output layer we get

$$\frac{\partial O_i}{\partial \theta_n^{(L)}} = g'(a_i^{(L)}) \delta_{in}.$$

We now insert (A.2) into (A.1)

$$\frac{\partial L}{\partial w_{nm}^{(L)}} = \sum_i (t_i - O_i) \cdot g'(a_i^{(L)}) V_m^{(L)} \delta_{in} = (t_n - O_n) g'(a_n^{(L)}) V_m^{(L-1)} \tag{A.3}$$

and similarly when differentiating w.r.t. the bias term we get

$$\frac{\partial L}{\partial \theta_n^{(L)}} = \sum_i (t_i - O_i) \cdot g'(a_i^{(L)}) \delta_{in} = (t_n - O_n) g'(a_n^{(L)}) \tag{A.4}$$

When computing the gradient for weights and biases in the next layer we want to make use of these results for the first layer so we denote a partial result from these first computations as follows:

$$d_i^{(L)} = (t_i - O_i) g'(a_i^{(L)}).$$

The use of this will be clear in subsequent steps.

We now turn our attention to the bias term in layer $L-1$ ($\theta_n^{(L-1)}$) and weights between layers $L-2$ and $L-1$ ($w_{nm}^{(L-1)}$). As we did before we differentiate the loss function w.r.t. these weights and bias term.

$$\frac{\partial L}{\partial w_{nm}^{(L-1)}} = \frac{\partial}{\partial w_{nm}^{(L-1)}} \left(\sum_i \frac{1}{2} (t_i - O_i)^2 \right) = \sum_i (t_i - O_i) \cdot \frac{\partial O_i}{\partial w_{nm}^{(L-1)}}$$

Now when differentiating the output neurons, we make use of the chain rule in order to be able to use the partial result from the L:th layer computations.

$$\frac{\partial O_i}{\partial w_{nm}^{(L-1)}} = \sum_{k_1} \frac{\partial O_i}{\partial V_{k_1}^{(L-1)}} \cdot \frac{\partial V_{k_1}^{(L-1)}}{\partial w_{nm}^{(L-1)}} \tag{A.5}$$

We compute the first factor:

$$\frac{\partial O_i}{\partial V_{k_1}^{(L-1)}} = \frac{\partial}{\partial V_{k_1}^{(L-1)}} \left(g \left(\theta_i^{(L)} + \sum_j w_{ji}^{(L)} V_j^{(L)} \right) \right) = g'(a_i^{(L)}) \cdot w_{k_1 i}^{(L)}, \tag{A.6}$$

and the second factor:

$$\begin{aligned} \frac{\partial V_{k_1}^{(L-1)}}{\partial w_{nm}^{(L-1)}} &= \frac{\partial}{\partial w_{nm}^{(L-1)}} \left(g \left(\theta_{k_1}^{(L-1)} + \sum_j w_{jk_1}^{(L-1)} \cdot V_j^{(L-2)} \right) \right) \\ &= g'(a_{k_1}^{(L-1)}) \cdot V_m^{(L-2)} \cdot \delta_{nk_1} \end{aligned} \tag{A.7}$$

Inserting (A.6) and (A.7) into (A.5) we obtain

$$\begin{aligned}\frac{\partial O_i}{\partial w_{nm}^{(L-1)}} &= \sum_{k_1} \frac{\partial O_i}{\partial V_{k_1}^{(L-1)}} \cdot \frac{\partial V_{k_1}^{(L-1)}}{\partial w_{nm}^{(L-1)}} \\ &= \sum_{k_1} g'(a_i^{(L)}) w_{k_1 i}^{(L)} \cdot g'(a_{k_1}^{(L-1)}) V_m^{(L-2)} \delta_{nk_1} \\ &= g'(a_i^{(L)}) w_{ni}^{(L)} g'(a_n^{(L-1)}) V_m^{(L-2)} \end{aligned} \tag{A.8}$$

We now use this result from (A.5) to compute the derivative of the loss function

$$\begin{aligned}\frac{\partial L}{\partial w_{nm}^{(L-1)}} &= \sum_i (t_i - O_i) \cdot \frac{\partial O_i}{\partial w_{nm}^{(L-1)}} \\ &= \sum_i (t_i - O_i) g'(a_i^{(L)}) w_{ni}^{(L)} g'(a_n^{(L-1)}) V_m^{(L-2)}.\end{aligned}$$

We note that we can use the partial result from earlier computations to evaluate this as

$$\frac{\partial L}{\partial w_{nm}^{(L-1)}} = \sum_i d_i^{(L)} w_{ni}^{(L)} g'(a_n^{(L-1)}) V_m^{(L-2)}$$

In order to use this result for simplifying computations when we differentiate w.r.t. to biases in layer $L-2$ and weights between layers $L-2$ and $L-3$ we denote a partial result:

$$d_i^{(L-1)} = \sum_j d_j^{(L)} w_{ij}^{(L)} g'(a_i^{(L-1)}) \tag{A.9}$$

We begin to see a pattern in how we can recursively use previous results to compute derivatives w.r.t. weights and biases in the other layers. But to clarify, we explicitly compute the derivatives w.r.t. weights and bias for one more layer.

$$\frac{\partial L}{\partial w_{nm}^{(L-2)}} = \frac{\partial}{\partial w_{nm}^{(L-2)}} \left(\sum_i \frac{1}{2} (t_i - O_i)^2 \right) = \sum_i (t_i - O_i) \cdot \frac{\partial O_i}{\partial w_{nm}^{(L-2)}} \tag{A.10}$$

Here we use the chain rule twice when computing the output neuron derivative:

$$\frac{\partial O_i}{\partial w_{nm}^{(L-2)}} = \sum_{k_1} \sum_{k_2} \frac{\partial O_i}{\partial V_{k_1}^{(L-1)}} \cdot \frac{\partial V_{k_1}^{(L-1)}}{\partial V_{k_2}^{(L-2)}} \cdot \frac{\partial V_{k_2}^{(L-2)}}{\partial w_{nm}^{(L-2)}} \tag{A.11}$$

As before we have for the first factor

$$\frac{\partial O_i}{\partial V_{k_1}^{(L-1)}} = g'(a_i^{(L)}) \cdot w_{k_1 i}. \tag{A.12}$$

And analogously to previous calculations we get for the third factor

$$\frac{\partial V_{k_2}^{(L-2)}}{\partial w_{nm}^{(L-2)}} = g'(a_{k_2}^{(L-2)}) \cdot V_m^{(L-3)} \delta_{nk_2}. \tag{A.13}$$

It remains to compute the second factor:

$$\begin{aligned}\frac{\partial V_{k_1}^{(L-1)}}{\partial V_{k_2}^{(L-2)}} &= \frac{\partial}{\partial V_{k_2}^{(L-2)}} \left(g \left(\theta_{k_1}^{(L-1)} + \sum_j w_{jk_1}^{(L-1)} V_j^{(L-2)} \right) \right) \\ &= g'(a_{k_1}^{(L-1)}) \cdot w_{k_2 k_1}^{(L-1)} \end{aligned} \tag{A.14}$$

Inserting (A.12), (A.14) and (A.13) into (A.11) yields

$$\begin{aligned}&\frac{\partial O_i}{\partial w_{nm}^{(L-2)}} \\ &= \sum_{k_1} \sum_{k_2} \left(g'(a_i^{(L)}) w_{k_1 i} \right) \left(g'(a_{k_1}^{(L-1)}) w_{k_2 k_1}^{(L-1)} \right) \left(g'(a_{k_2}^{(L-2)}) V_m^{(L-3)} \delta_{nk_2} \right) \\ &= \sum_{k_1} g'(a_i^{(L)}) w_{k_1 i}^{(L)} \cdot g'(a_{k_1}^{(L-1)}) w_{nk_1}^{(L-1)} \cdot g'(a_n^{(L-2)}) V_m^{(L-3)} \end{aligned} \tag{A.15}$$

Inserting (A.15) into (A.10) gives us the derivatives w.r.t. the weights of the loss function:

$$\begin{aligned}\frac{\partial L}{\partial w_{nm}^{(L-2)}} &= \sum_i (t_i - O_i) \cdot \frac{\partial O_i}{\partial w_{nm}^{(L-2)}} \\ &= \sum_{k_1} \sum_i (t_i - O_i) g'(a_i^{(L)}) w_{k_1 i}^{(L)} g'(a_{k_1}^{(L-1)}) w_{nk_1}^{(L-1)} g'(a_n^{(L-2)}) V_m^{(L-3)} \\ &= \sum_{k_1} d_{k_1}^{(L-1)} w_{nk_1}^{(L-1)} g'(a_n^{(L-2)}) V_m^{(L-3)} \end{aligned}$$

In the last step we used the previous result (A.9).

We see that we once again can define a new partial result to use for computing derivatives in the next layer:

$$d_i^{(L-2)} = \sum_j d_j^{(L-1)} w_{ij}^{(L-1)} g'(a_i^{(L-2)})$$

Now we are in a position to observe a general pattern:

$$\begin{aligned}
d_i^{(L)} &= (t_i - O_i) g'(a_i^{(L)}) \\
d_i^{(L-1)} &= \sum_j d_j^{(L)} w_{ij}^{(L)} g'(a_i^{(L-1)}) \\
d_i^{(L-2)} &= \sum_j d_j^{(L-1)} w_{ij}^{(L-1)} g'(a_i^{(L-2)}) \\
&\dots \\
d_i^{(l)} &= \sum_j d_j^{(l+1)} w_{ij}^{(l+1)} g'(a_i^{(l)}) \text{ for } 1 \leq l < L.
\end{aligned}$$

From which we easily compute the derivatives that the gradient consists of (except output layer):

$$\begin{aligned}
\frac{\partial L}{\partial w_{mn}^{(l)}} &= d_n^{(l)} V_m^{(l-1)}, & 1 \leq l < L \\
\frac{\partial L}{\partial \theta_n^{(l)}} &= d_n^{(l)}, & 1 \leq l < L
\end{aligned}$$

And for the weights and bias in the output layer we recall our previous results (A.3) and (A.4)

$$\begin{aligned}
\frac{\partial L}{\partial w_{nm}^{(L)}} &= (t_n - O_n) g'(a_n^{(L)}) V_m^{(L-1)} \\
\frac{\partial L}{\partial \theta_n^{(L)}} &= (t_n - O_n) g'(a_n^{(L)})
\end{aligned}$$

A.2 Softmax output and Cross entropy loss function

As for the case with the square distance loss function we assume we only have one pattern and omit the pattern index κ. As before we want to compute the gradient w.r.t. weights and biases, so we start with differentiating w.r.t. weights connecting to the output layer L.

$$\frac{\partial L}{\partial w_{nm}^{(L)}} = \sum_i \frac{t_i}{O_i} \frac{\partial O_i}{\partial w_{nm}^{(L)}} \tag{A.16}$$

We use the chain rule for $a_i^{(L)}$

$$\frac{\partial O_i}{\partial w_{nm}^{(L)}} = \sum_s \frac{\partial O_i}{\partial a_s^{(L)}} \cdot \frac{\partial a_s^{(L)}}{\partial w_{nm}^{(L)}}. \tag{A.17}$$

We compute the factors

$$\frac{\partial O_i}{\partial a_s^{(L)}} = O_i(\delta_{is} - O_s) \tag{A.18}$$

$$\frac{\partial a_s^{(L)}}{\partial w_{nm}^{(L)}} = \delta_{sm} V_n \tag{A.19}$$

and insert them, (A.18) and (A.19), into (A.17).

$$\frac{\partial O_i}{\partial w_{nm}^{(L)}} = O_i(\delta_{im} - O_m)V_n$$

We can now use this in (A.16), so we get

$$\begin{aligned}\frac{\partial L}{\partial w_{nm}^{(L)}} &= \sum_i \frac{t_i}{O_i} \frac{\partial O_i}{\partial w_{nm}^{(L)}} = \sum_i \frac{t_i}{O_i} O_i(\delta_{im} - O_m)V_n = \\ &= \left(\sum_i (t_m - t_i O_m) \right) V_n\end{aligned}$$

Now we can, as before, define a help variable to keep track of the partial results that we will use for the next layer.

$$d_i^{(L)} = \sum_j (t_i - t_j O_i)$$

Note that if $\sum_i t_i = 1$ this can be reduced to $d_i^{(L)} = t_i - O_i$. However for some of our applications, when we use one Softmax for outputs that signifies more than one class, this will not be the case.

Differentiating w.r.t. the bias variable can be treated in a similar way. We obtain

$$\frac{\partial L}{\partial \theta_n^{(L)}} = ... = d_i^{(L)}.$$

We now compute the derivatives w.r.t. weights in the next layer, $L-1$.

$$\frac{\partial L}{\partial w_{nm}^{(L-1)}} = \sum_i \frac{t_i}{O_i} \frac{\partial O_i}{\partial w_{nm}^{(L-1)}} \tag{A.20}$$

When computing the derivative of the output i we use the chain rule w.r.t. the neurons in layer $L-1$.

$$\frac{\partial O_i}{\partial w_{nm}^{(L-1)}} = \sum_l \frac{\partial O_i}{V_l^{(L-1)}} \cdot \frac{V_l^{(L-1)}}{\partial w_{nm}^{(L-1)}} \tag{A.21}$$

We see that the second factor is the same as in the case with square distance loss function (A.7)

$$\frac{V_l^{(L-1)}}{\partial w_{nm}^{(L-1)}} = g'(a_l^{(L-1)})V_m^{(L-2)}\delta_{nl} \tag{A.22}$$

and for the first factor we use the chain rule w.r.t. $a_i^{(L)}$ and get

$$\begin{aligned}
\frac{\partial O_i}{\partial V_l^{(L-1)}} &= \frac{\partial O_i}{\partial a_l^{(L)}} \cdot \frac{\partial a_l^{(L)}}{\partial V_l^{(L-1)}}, \\
\frac{\partial O_i}{\partial a_l^{(L)}} &= O_i(\delta_{il} - O_l), \\
\frac{\partial a_l^{(L)}}{\partial V_l^{(L-1)}} &= w_{li}^{(L)} \\
\implies \frac{\partial O_i}{\partial V_l^{(L-1)}} &= O_i(\delta_{il} - O_l)w_{li}^{(L)}. \qquad \text{(A.23)}
\end{aligned}$$

We now insert (A.22) and (A.23) into (A.21) and get

$$\frac{\partial O_i}{\partial w_{nm}^{(L-1)}} = O_i(\delta_{in} - O_n)w_{ni}^{(L)}g'(a_n^{(L-1)})V_m^{(L-2)} \tag{A.24}$$

We use this, (A.24), with (A.20) to obtain the derivative

$$\begin{aligned}
\frac{\partial L}{\partial w_{nm}^{(L-1)}} &= \sum_i t_i(\delta_{in} - O_n)w_{ni}^{(L)}g'(a_n^{(L-1)})V_m^{(L-2)} = \\
&= \left(\sum_i (t_n - t_iO_n)w_{ni}^{(L)}g'(a_n^{(L-1)})\right)V_m^{(L-2)}
\end{aligned}$$

We see here that actually we can not use the partial result from layer L, but defining this partial result for layer $L-1$

$$d_i^{(L-1)} = \sum_j (t_i - t_jO_i)w_{ij}^{(L)}g'(a_i^{(L-1)}),$$

this result can be used in derivatives w.r.t. weights and biases in further layers. We now realize that for further layers the computations will be the same as in the squared distance loss function case, so we get the results:

$$\frac{\partial L}{\partial w_{mn}^{(l)}} = d_n^{(l)} V_m^{(l-1)}$$
$$\frac{\partial L}{\partial \theta_n^{(l)}} = d_n^{(l)}$$

for $1 \leq l \leq L$, where the partial results $d_\cdot^{(\cdot)}$ are computed recursively starting from $l = L$:

$$\begin{aligned} d_i^{(L)} &= \sum_j (t_i - t_j O_i) \\ d_i^{(L-1)} &= \sum_j (t_i - t_j O_i) w_{ij}^{(L)} g'(a_i^{(L-1)}) \\ d_i^{(L-2)} &= \sum_j d_j^{(L-1)} w_{ij}^{(L-1)} g'(a_i^{(L-2)}) \\ &\dots \\ d_i^{(l)} &= \sum_j d_j^{(l+1)} w_{ij}^{(l+1)} g'(a_i^{(l)}) \text{ for } 1 \leq l < L-1. \end{aligned}$$

A.3 Frequency prediction and Poisson deviance loss function

We start by differentiating w.r.t. weights connected to the output neuron

$$\begin{aligned} \frac{\partial L}{\partial w_{m1}^{(L)}} &= 2\frac{\partial \xi}{\partial w_{m1}^{(L)}} - 2N \frac{\partial (R + \theta_1^{(L)} + \sum_j w_{j1}^{(L)} V_j^{(L-1)})}{\partial w_{m1}^{(L)}} \\ &= 2\frac{\partial \xi}{\partial w_{m1}^{(L)}} - 2N V_m^{(L-1)} = 2\xi V_m^{(L-1)} - 2N V_m^{(L-1)} \\ &= 2(\xi - N) V_m^{(L-1)} \end{aligned}$$

and similarly

$$\frac{\partial L}{\partial \theta_1^{(L)}} = 2(\xi - N).$$

We define a partial result

$$d^{(L)} = 2(\xi - N).$$

For the weights in the next layer $L-1$ we get

$$\frac{\partial L}{\partial w_{mn}^{(L-1)}} = 2\frac{\partial \xi}{\partial w_{mn}^{(L-1)}} - 2N\frac{\partial(\log \xi)}{\partial w_{nm}^{(L-1)}}. \tag{A.25}$$

We start with the second term, and use the chain rule using the layer $L-1$ neurons as the middle variables in the chain.

$$\begin{aligned}\frac{\partial(\log \xi)}{\partial w_{nm}^{(L-1)}} = \frac{\partial(\sum_j w_{m1}^{L)} V_j^{(L-1)})}{\partial w_{nm}^{(L-1)}} &= \sum_l \frac{\partial(\sum_j w_{m1}^{L)} V_j^{(L-1)})}{\partial V_l^{(L-1)}} \cdot \frac{\partial V_l^{(L-1)}}{\partial w_{nm}^{(L-1)}} \\ &= \sum_l w_{l1}^{(L)} \cdot \frac{\partial V_l^{(L-1)}}{\partial w_{nm}^{(L-1)}}. \end{aligned} \tag{A.26}$$

For the first term we also use the chain rule with the layer $L-1$ neurons as the middle variables in the chain

$$\frac{\partial \xi}{\partial w_{mn}^{(L-1)}} = \sum_l \frac{\partial \xi}{V_l^{(L-1)}} \frac{V_l^{(L-1)}}{\partial w_{mn}^{(L-1)}} = \sum_l \xi w_{l1}^{(L)} \frac{V_l^{(L-1)}}{\partial w_{mn}^{(L-1)}} \tag{A.27}$$

From previous computations in the squared distance loss function case (A.7), we remember that

$$\frac{V_l^{(L-1)}}{\partial w_{mn}^{(L-1)}} = g'(a_l^{(L-1)}) V_m^{(L-2)} \delta_{nl}.$$

Inserting (A.27) and (A.26) and (A.7) into (A.25) yields

$$\begin{aligned}\frac{\partial L}{\partial w_{mn}^{(L-1)}} &= \sum_l 2 w_{m1}^{(L)} (\xi - N) \frac{V_l^{(L-1)}}{\partial w_{mn}^{(L-1)}} \\ &= \sum_l 2 w_{l1}^{(L)} (\xi - N) g'(a_l^{(L-1)} V_m^{(L-2)} \delta_{nl} \\ &= 2 w_{n1}^{(L)} (\xi - N) g'(a_n^{(L-1)}) V_m^{(L-2)}.\end{aligned}$$

Using our previous partial result we see that

$$\frac{\partial L}{\partial w_{mn}^{(L-1)}} = d_1^{(L)} w_{n1}^{(L)} g'(a_n^{(L-1)}) V_m^{(L-2)}.$$

We define a new partial result

$$d_i^{(L-1)} = d_1^{(L)} w_{i1}^{(L)} g'(a_n^{(L-1)}).$$

Now we realize that we can compute derivatives recursively in the same way as for the previous cases. We summarize it here

$$\begin{aligned}
d_i^{(L)} &= 2(\xi - N) \\
d_i^{(L-1)} &= d_1^{(L)} w_{i1}^{(L)} g'(a_i^{(L-1)}) \\
d_i^{(L-2)} &= \sum_j d_j^{(L-1)} w_{ij}^{(L-1)} g'(a_i^{(L-2)}) \\
&\text{...} \\
d_i^{(l)} &= \sum_j d_j^{(l+1)} w_{ij}^{(l+1)} g'(a_i^{(l)}) \text{ for } 1 \leq l < L-1.
\end{aligned}$$

Using these partial results, we can compute all the components of the gradient:

$$\begin{aligned}
\frac{\partial L}{\partial w_{mn}^{(l)}} &= d_n^{(l)} V_m^{(l-1)} \\
\frac{\partial L}{\partial \theta_n^{(l)}} &= d_n^{(l)}.
\end{aligned}$$

Appendix B

Optimization algorithms

In this thesis we use Nesterov-accelerated adaptive moment estimation (NADAM) for optimizing the loss function when training neural networks. This algorithm is presented in chapter 2. In this appendix we present the main algorithms on which it is based.

NADAM was created by combining ideas from the ADAM algorithm (adaptive moments) and Nesterov's accelerated gradient algorithm (NAG), which both are versions of the Stochastic gradient descent algorithm (SGD) [2]. SGD itself is a variation of the optimization algorithm gradient descent (also known as steepest descent), so we start by giving a short description of the gradient descent algorithm.

B.1 Gradient descent

Suppose we have a multivariate differentiable function $F(\mathbf{z})$ that we want to minimize. The gradient descent algorithm we present here is a variation of a broader class of descent algorithms [8, p. 282-283], which we describe in algorithm2.

The basic idea of the gradient descent algorithm is to move in the direction in which the function F decreases the fastest, i.e. in the opposite direction of the gradient. This specifies the method D in the descent algorithm 2. In the version of the gradient descent we present here in algorithm 3, we also specify the method of choosing step sizes S.

Step sizes ϵ_k in algorithm 3 can be chosen to be constant, $\epsilon_k = \epsilon$ for all k. If different sizes of step sizes are used one typically chooses these to be increasingly smaller $\epsilon_{k+1} < \epsilon_k$ since the earlier estimates in the optimization process are more rough than the later ones.

There are several options for the stopping condition C, for example stop-

Algorithm 2 Descent algorithm

```
Require: $F(\mathbf{z})$ multivariate differentiable function
Require: Starting point $\mathbf{z}_0 \in \mathbb{R}^n$
Require: Stopping condition $C$
Require: A method $D$ of determining a descent direction $\mathbf{p}_k \in R^n$
Require: A method $S$ of determining step length $\alpha_k > 0$, such that $F(\mathbf{z}_k + \alpha_k \mathbf{p}_k) < F(\mathbf{z}_k)$ holds.
  $k \leftarrow 0$
  while $C$ not fulfilled do
    (descent direction) Determine descent direction $\mathbf{p}_k \in R^n$ by $D$
    (line search) Determine a step length $\alpha_k$ by $S$
    (update) $\mathbf{z}_{k+1} \leftarrow \mathbf{z}_k + \alpha_k \mathbf{p}_k$
    $k \leftarrow k + 1$
  end while
  return $\mathbf{z}_k$
```

Algorithm 3 Gradient descent (steepest descent)

```
Require: $F(\mathbf{z})$ multivariate differentiable function
Require: Starting point $\mathbf{z}_0 \in \mathbb{R}^n$
Require: Sequence of step sizes $\epsilon_k, k = 0, 1, 2, ...$
Require: Stopping condition $C$
  $k \leftarrow 0$
  while $C$ not fulfilled do
    $\mathbf{z}_{k+1} \leftarrow \mathbf{z}_k - \epsilon_k \nabla F(\mathbf{z}_k)$
    $k \leftarrow k + 1$
  end while
  return $\mathbf{z}_k$
```

ping if $|F(\mathbf{z}_{k+1}) - F(\mathbf{z}_k)| < \delta$ for some fixed small δ.

B.2 SGD - Stochastic gradient descent

In machine learning problems the function we want to optimize is generally of a specific type $F(\mathbf{z}) = \sum_i L(f(x^{(i)}; \mathbf{z}_k), t^{(i)})$, where L is a loss function which evaluates a distance between targets $t^{(i)}$ and output of the machine learning algorithm f given parameters $\mathbf{z}$ when fed input patterns $x^{(i)}$. In our case $\mathbf{z}$ consists of the weight and the threshold parameters of the neural network.

The idea for the SGD method is that instead of computing the gradient for F where we summarize over all patterns and targets, in each iteration k we sample a mini batch of m samples from the training set with corresponding targets and let M_k be the set of indices specifying these samples. We compute the gradient in the average of the sum of the loss function in these mini batch samples and update the parameters [6, p. 290-292]. We present SGD in algorithm 4. We note that just as for the gradient descent algorithm, step sizes ϵ_k in algorithm 4 can be chosen to be constant, $\epsilon_k = \epsilon$ for all k. If different sizes of step sizes are used one typically chooses these to be increasingly smaller $\epsilon_{k+1} < \epsilon_k$ since the earlier estimates in the optimization process are more rough than the later ones. We note that there are convergence results on how to chose learning rate schedules but in practice it is common to linearly decrease the learning rate up to the point of reaching some iteration $k = \tau$, and after that $k > \tau$ use a constant convergence rate [6, p. 291].

Algorithm 4 Stochastic gradient descent

```
Require: Starting point (initial parameters) $\mathbf{z}_0$.
Require: Sequence of learning rates (step sizes) $\epsilon_k, k = 0, 1, 2, ...$
Require: Stopping condition $C$
  $k \leftarrow 0$
  while $C$ not fulfilled do
      Sample a minibatch $M_k$.
      Compute gradient estimate $\hat{\mathbf{g}}_k \leftarrow \frac{1}{m}\nabla_{\mathbf{z}_k} \sum_{i \in M_k} L(f(x^{(i)}; \mathbf{z}_k), t^{(i)})$.
      Update $\mathbf{z}_{k+1} \leftarrow \mathbf{z}_k - \epsilon_k \hat{\mathbf{g}}_k$.
      $k \leftarrow k + 1$
  end while
  return $\mathbf{z}_k$
```

For the stopping condition C just as for the gradient descent algorithm there are several options, for example we can chose to stop if $|F(\mathbf{z}_{k+1}) -$

$F(\mathbf{z}_k)| < \delta$ for some fixed small δ.

B.3 Momentum and Nesterov's accelerated gradient

B.3.1 Stochastic gradient descent with momentum (classical momentum)

This extension of SGD consists of replacing the negative gradient as descent direction with a decaying sum of previous updates plus negative gradient (a momentum vector) [2]. Compared to SGD this algorithm (see algorithm 5) moves slower when the direction of the update significantly oscillates, and faster when it does not.

Algorithm 5 Stochastic gradient descent with momentum

```
Require: Starting point (initial parameters) $\mathbf{z}_0$.
Require: Sequence of learning rates (step sizes) $\epsilon_k, k = 0, 1, 2, ...$
Require: Stopping condition $C$
Require: Decay factor $\mu_d$
  $k \leftarrow 0$
  while $C$ not fulfilled do
      Sample a minibatch $M_k$.
      Compute gradient estimate $\hat{\mathbf{g}}_k \leftarrow \frac{1}{m}\nabla_{\mathbf{z}_k} \sum_{i \in M_k} L(f(x^{(i)}; \mathbf{z}_k), t^{(i)})$.
      Update momentum vector $\mathbf{m}_{k+1} \leftarrow \mu_d \mathbf{m}_k + \epsilon_k \hat{\mathbf{g}}_k$
      Update $\mathbf{z}_{k+1} \leftarrow \mathbf{z}_k - \mathbf{m}_{k+1}$.
      $k \leftarrow k + 1$
  end while
  return $\mathbf{z}_k$
```

B.3.2 Nesterov's accelerated gradient (NAG)

In SGD with momentum we take momentum into account in the update of $\mathbf{z}_k$. NAG extends SGD with momentum by also taking momentum into account in the computation of the gradient. The update in SGD with momentum can be seen to consist of first moving in the direction of the previous momentum and then in the direction of the gradient (computed in the point before update), NAG improves this by first moving in the direction of the previous momentum, then computing the gradient in this new point, and after that move in the direction of the gradient [2]. For difficult optimization objective functions there is evidence that NAG is superior to both SGD and SGD with momentum. We present the algorithm in algorithm 6.

Algorithm 6 Nesterov's accelerated gradient

```
Require: Starting point (initial parameters) $\mathbf{z}_0$.
Require: Sequence of learning rates (step sizes) $\epsilon_k, k = 0, 1, 2, ...$
Require: Stopping condition $C$
Require: Decay factor $\mu_d$
  $k \leftarrow 0$
  while $C$ not fulfilled do
      Sample a minibatch $M_k$.
      Compute gradient estimate $\hat{\mathbf{g}}_k \leftarrow \frac{1}{m}\nabla_{\mathbf{z}_k} \sum_{i \in M_k} L(f(x^{(i)}; \mathbf{z}_k - \mu_d \mathbf{m}_k), t^{(i)})$.
      Update momentum vector $\mathbf{m}_{k+1} \leftarrow \mu_d \mathbf{m}_k + \epsilon_k \hat{\mathbf{g}}_k$
      Update $\mathbf{z}_{k+1} \leftarrow \mathbf{z}_k - \mathbf{m}_{k+1}$.
      $k \leftarrow k + 1$
  end while
  return $\mathbf{z}_k$
```

B.4 Adaptive moments estimation (ADAM)

The previously discussed optimization algorithms are all algorithms that use a set learning rate (or a set scheme of learning rates). ADAM uses an adaptive learning rate, with only one global learning rate as a parameter. The algorithm [6, p. 306] is shown in algorithm 7.

Conceptually, ADAM can be thought of as a combination of RMSProp – a modification of the AdaGrad algorithms (we will not go into these algorithms here), and momentum with some additional modifications [6, p. 303-306]. Compared to previously mentioned algorithms ADAM uses a decaying mean over previous gradients rather than a decaying sum over previous updates [2].

The update is based on the idea of moving in the direction of an exponentially decaying moving average of the first order moment of the gradient. The learning rate is adapted by scaling these updates with an accumulated squared gradient (scaled – an exponential moving average), a second moment estimate. Using exponentially weighted moving averages rather than just accumulated gradients improves performance when applied to nonconvex objective functions, since when the objective is not convex the gradients close to the initial point z_0 may not be relevant.

To account for the initialization at zero for both first and second order moments, ADAM includes a bias correction for this.

Algorithm 7 Adaptive moments estimation

```
Require: Starting point (initial parameters) $\mathbf{z}_0$.
Require: Global learning rate (step size) $\epsilon$ (suggested default: 0.001)
Require: Exponential decay rates $\rho_1, \rho_2 \in [0,1)$ (suggested defaults: 0.9 and
  0.999)
Require: Small constant $\delta$ for numerical stabilization (suggested default:
  $10^{-8}$)
Require: Stopping condition $C$
  $k \leftarrow 0$
  $\mathbf{s}_0 \leftarrow \mathbf{0}$ (first moment)
  $\mathbf{r}_0 \leftarrow \mathbf{0}$ (second moment)
  while $C$ not fulfilled do
      Sample a minibatch $M_k$.
      Compute gradient estimate $\hat{\mathbf{g}}_k \leftarrow \frac{1}{m}\nabla_{\mathbf{z}_k} \sum_{i \in M_k} L(f(x^{(i)}; \mathbf{z}_k), t^{(i)})$.
      Biased first moment estimate $\mathbf{s}_{k+1} \leftarrow \rho_1 \mathbf{s}_k + (1-\rho_1)\hat{\mathbf{g}}_k$
      Biased second moment estimate $\mathbf{r}_{k+1} \leftarrow \rho_2 \mathbf{r}_k + (1-\rho_2)\hat{\mathbf{g}}_k \odot \hat{\mathbf{g}}_k$
      Correct bias in first moment $\hat{\mathbf{s}}_{k+1} \leftarrow \frac{\mathbf{s}_{k+1}}{1-\rho_1^k}$
      Correct bias in second moment $\hat{\mathbf{r}}_{k+1} \leftarrow \frac{\mathbf{r}_{k+1}}{1-\rho_2^k}$
      Update $\mathbf{z}_{k+1} \leftarrow \mathbf{z}_k - \epsilon \frac{\hat{\mathbf{s}}_{k+1}}{\sqrt{\hat{\mathbf{r}}_{k+1}}+\delta}$
      $k \leftarrow k+1$
  end while
  return $\mathbf{z}_k$
```

Appendix C

Gradient boosting machines

We use the GBM package in R, the algorithm described here is based on the description in the GBM package vignette [10]. This implementation uses the AdaBoost algorithm's exponential loss function, but uses Friedman's gradient descent algorithm – more specifically, the stochastic gradient boosting version of it. This version of the algorithm achieves variance reduction by using subsampling.

We want to find an estimate $\hat{f}(\mathbf{x})$ of the function $f(\mathbf{x})$ that minimizes a loss function $\Psi(y, f)$:

$$\begin{aligned}\hat{f}(\mathbf{x}) &= \underset{f(\mathbf{x})}{\arg\min}\, \mathrm{E}_{y,\mathbf{x}}\Psi(y, f(\mathbf{x})) \\ &= \underset{f(\mathbf{x})}{\arg\min}\, \mathrm{E}_{\mathbf{x}}[\mathrm{E}_{y|\mathbf{x}}\Psi(y, f(\mathbf{x}))|\mathbf{x}]. \end{aligned} \tag{C.1}$$

The algorithm we use solves this for cases when $\mathbf{x}$ is assumed to be known. This reduces equation C.1 to equation C.2,

$$\hat{f}(\mathbf{x}) = \underset{f(\mathbf{x})}{\arg\min}\, \mathrm{E}_{y|\mathbf{x}}[\Psi(y, f(\mathbf{x}))|\mathbf{x}]. \tag{C.2}$$

In practice this estimate is computed using a given set of observed covariates (features) and responses $(\mathbf{x}_i, y_i)_{i=1}^{N}$.

The general idea of the algorithm is to stepwise change the estimate $\hat{f}(\mathbf{x})$ by adding (weighted) estimates of the negative gradient of the loss function. The negative gradient of the loss function is in each step estimated by first analytically computing it and then estimating it by fitting a regression tree using only a randomly selected subset of the observations, with the analytically computed negative gradient as response. This fitted regression tree is used as the estimate of the negative gradient that is added to $\hat{f}(\mathbf{x})$.

Closely following the algorithm as described in the vignette[10], we present the algorithm in algorithm 8.

Algorithm 8 Stochastic gradient boosting (following the implementation in the gbm package in R)

```
Require: Set of observed covariates and responses, $(\mathbf{x}_i, y_i)_{i=1}^N$.
Require: Loss function, $\Psi$.
Require: Number of iterations, $T$.
Require: Depth of each regression tree, $K$.
Require: Shrinkage (learning rate) parameter, $\lambda$.
Require: Subsampling rate (a fraction of $N$), $p$.
  Initialize $\hat{f}(\mathbf{x})$ to be a constant, $\hat{f}(\mathbf{x}) \leftarrow \arg\min_\rho \sum_{i=1}^N \Psi(y_i, \rho)$.
  Set $t \leftarrow 1$
  for $t = 1,..,T$ do
      Compute the negative gradient as the working response
          $$z_i \leftarrow -\frac{\partial}{\partial f(\mathbf{x}_i)} \Psi(y_i, f(\mathbf{x}_i)) \bigg|_{f(\mathbf{x}_i)=\hat{f}(\mathbf{x}_i)}$$

      Select $p \times N$ observations from the data set randomly.

      Fit a regression tree with $K$ terminal nodes, that estimates the gradient
      $g(\mathbf{x}) = \mathrm{E}(z|\mathbf{x})$, using only the randomly selected observations.

      Compute the optimal terminal node predictions $\rho_1, ..., \rho_K$ (using only
      the randomly selected observations), as
          $$\rho_k = \arg\min_\rho \sum_{\mathbf{x}_i \in S_k} \Psi(y_i, \hat{f}(\mathbf{x}_i) + \rho)$$
      where $S_k$ is the set of observations that define the terminal node $k$.

      Update $\hat{f}(\mathbf{x})$ as
          $$\hat{f}(\mathbf{x}) \leftarrow \hat{f}(\mathbf{x}) + \lambda \rho_{k(\mathbf{x})}$$
      where $\rho_{k(\mathbf{x})}$ is the index of the terminal node into which an observation
      with features $\mathbf{x}$ would fall.
  end for
  return $\hat{f}(\mathbf{x})$
```

Bibliography

[1] Delong, Ł., Kozak, A. (2023). The use of autoencoders for training neural networks with mixed categorical and numerical features. *ASTIN Bulletin: The Journal of the IAA*, Volume 53 , Issue 2 , May 2023 , pp. 213 - 232. Cambridge University Press. DOI: 10.1017/asb.2023.15

[2] Dozat, T. (2016). Incorporating Nesterov Momentum into Adam. *Proceedings of the 4th International Conference on Learning Representations*, Workshop Track, San Juan, Puerto Rico, 2-4 May 2016, 1-4. https://openreview.net/forum?id=OM0jvwB8jIp57ZJjtNEZ

[3] Denuit, M., Sznajder D., Trufin J.(2019). Model selection based on Lorenz and concentration curves, Gini indices and convex order. *Insurance: Mathematics and Economics*, Volume 89, November 2019, Pages 128-139. Elsevier B.V. DOI: 10.1016/j.insmatheco.2019.09.001

[4] Lindholm, M., Lindskog, F., Palmquist, J. (2023). Local bias adjustment, duration-weighted probabilities, and automatic construction of tariff cells. *Scandinavian Actuarial Journal*, Volume 2023, 2023 - Issue 10, Pages 946-973. DOI: 10.1080/03461238.2023.2176251

[5] Mehlig, B. (2021), Machine learning with neural networks. ISBN: 9781108494939, Cambridge University Press. DOI: 10.1017/9781108860604

[6] Goodfellow, I., Bengio, Y., Courville, A. (2016), Deep Learning. ISBN: 9780262035613, The MIT Press. https://www.deeplearningbook.org/

[7] Ferrario, A., Noll, A., Wüthrich, M. V. (2020), Insights from Inside Neural Networks. *SSRN Manuscript* https://ssrn.com/abstract=3226852

[8] Andréasson, N., Evgrafov, A., Patriksson, M., Gustavsson, E., Önnheim, M. (2013). An Introduction to Continuos Optimization (Second edition). ISBN 978-91-44-06077-4, Studentlitteratur AB, Lund.

[9] Renzmann, S., Wüthrich, M. V. (2019), Unsupervised learning: What is a sports car? https://ssrn.com/abstract=3439358

[10] Ridgeway, G. (2024), Generalized boosted models: A guide to the gbm package. *Vignette to the gbm package in R* https://cran.r-project.org/web/packages/gbm/

[11] Wüthrich, M (2019), From generalized linear models to neural networks, and back. *SSRN Electronic Journal.* doi 10.2139/ssrn.3491790.

[12] Erhan, D., Bengio, Y., Courville, A., Manzagol, P.-A., Vincent, P., and Bengio, S. (2010), Why does unsupervised pre-training help deep learning? *Journal of Machine Learning Research*, 11(19):625–660. http://jmlr.org/papers/v11/erhan10a.html

[13] Erhan, D., Manzagol, P.-A., Bengio, Y., Bengio, S., and Vincent, P. (2009). The difficulty of training deep architectures and the effect of unsupervised pre-training. In van Dyk, D. and Welling, M., editors, *Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics, volume 5 of Proceedings of Machine Learning Research*, pages 153–160, Hilton Clearwater Beach Resort, Clearwater Beach, Florida USA. PMLR. https://proceedings.mlr.press/v5/erhan09a.html.

[14] Delong, Ł., Lindholm, M., Zakrisson, H. (2023), On Cyclic Gradient Boosting Machines https://ssrn.com/abstract=4352505