Insights from Inside Neural Networks

Andrea Ferrario* Alexander Noll† Mario V. Wüthrich‡

Prepared for:
Fachgruppe "Data Science"
Swiss Association of Actuaries SAV

Version of November 14, 2018

Abstract

We provide a tutorial that illuminates the aspects which need to be considered when fitting neural network regression models to claims frequency data in insurance. We discuss feature pre-processing, choice of loss function, choice of neural network architecture, class imbalance problem, as well as over-fitting. This discussion is based on a publicly available real car insurance data set.

Keywords. neural networks, architecture, over-fitting, loss function, dropout, regularization, LASSO, ridge, gradient descent, class imbalance, car insurance, claims frequency, Poisson regression model, machine learning, deep learning.

0 Introduction and overview

This data analytics tutorial has been written for the working group "Data Science" of the Swiss Association of Actuaries SAV, see

`https://www.actuarialdatascience.org`

The main purpose of this tutorial is to illuminate the aspects which need to be considered when fitting neural network regression models to claims frequency data in insurance. This tutorial is based on the introductory tutorial of Noll et al. [18]. We use the same French motor third-party liability insurance data set as in the latter tutorial, and we provide an in-depth analysis of neural network calibrations on that data set. As a result, we see that the models of Noll et al. [18] may substantially be improved by a careful model choice.

*Mobiliar Lab for Analytics, ETH Zurich, aferrario@ethz.ch
†PartnerRe Holdings Europe Limited, alexander.noll@partnerre.com
‡RiskLab, Department of Mathematics, ETH Zurich, mario.wuethrich@math.ethz.ch

1 The data and a warming-up exercise

1.1 French motor third-party liability insurance data

We revisit the data `freMTPL2freq` which is included in the R package `CASdatasets`, see Charpentier [4].[1] This data comprises a French motor third-party liability (MTPL) insurance portfolio with corresponding claim counts observed within one accounting year. This data has been illustrated and studied in the tutorial of Noll et al. [18]. Listing 1 provides a short summary of the data.

Listing 1: output of command `str(freMTPL2freq)`

```
1  > str(freMTPL2freq)
2  'data.frame':   678013 obs. of  12 variables:
3   $ IDpol     : num  1 3 5 10 11 13 15 17 18 21 ...
4   $ ClaimNb   : num [1:678013(1d)] 1 1 1 1 1 1 1 1 1 1 ...
5    ..- attr(*, "dimnames")=List of 1
6    .. ..$ : chr  "139" "414" "463" "975" ...
7   $ Exposure  : num  0.1 0.77 0.75 0.09 0.84 0.52 0.45 0.27 0.71 0.15 ...
8   $ Area      : Factor w/ 6 levels "A","B","C","D",..: 4 4 2 2 2 5 5 3 3 2 ...
9   $ VehPower  : int  5 5 6 7 7 6 6 7 7 7 ...
10  $ VehAge    : int  0 0 2 0 0 2 2 0 0 0 ...
11  $ DrivAge   : int  55 55 52 46 46 38 38 33 33 41 ...
12  $ BonusMalus: int  50 50 50 50 50 50 50 68 68 50 ...
13  $ VehBrand  : Factor w/ 11 levels "B1","B10","B11",..: 4 4 4 4 4 4 4 4 4 4 ...
14  $ VehGas    : Factor w/ 2 levels "Diesel","Regular": 2 2 1 1 1 2 2 1 1 1 ...
15  $ Density   : int  1217 1217 54 76 76 3003 3003 137 137 60 ...
16  $ Region    : Factor w/ 22 levels "R11","R21","R22",..: 18 18 3 15 15 8 8 20 20 12 ...
```

A detailed descriptive analysis of this data is provided in the tutorial of Noll et al. [18]. The analysis in that reference also includes a (minor) data cleaning part on the original data, which is used but not further discussed in the present manuscript.

1.2 Descriptive statistics of the exposure measure

We start by providing descriptive statistics about the exposure measure in this claims frequency example. The original model in Noll et al. [18] assumed that all insurance policies $i = 1, \ldots, 678'013$ have independent Poisson claims counts N_i with

$$N_i \overset{\text{ind.}}{\sim} \text{Poi}\left(\lambda(\boldsymbol{x}_i) v_i\right), \tag{1.1}$$

for given volumes $v_i > 0$ (time `Exposure` in years on line 7 in Listing 1) and a given claims frequency function $\boldsymbol{x}_i \mapsto \lambda(\boldsymbol{x}_i)$, where $\boldsymbol{x}_i$ describes the feature information of policy i, see Assumptions 2.1 in Noll et al. [18] and lines 8-16 in Listing 1. Since all policies were active within one accounting year, the volumes were considered pro-rata temporis $v_i \in (0, 1]$ for the corresponding time exposures. A time exposure as a volume measure has been criticized in the literature, see e.g. Verbelen et al. [25], and other exposure measures have been proposed. In the descriptive analysis in this section we analyze the linearity of the expected number of claims in this time exposure, i.e. we analyze empirically the linearity $v_i \mapsto \mathbb{E}[N_i] = \lambda(\boldsymbol{x}_i) v_i$.[2] We

[1] CASdatasets website `http://cas.uqam.ca`; the data is described on page 55 of the reference manual [3].

[2] Linearity in the volume means that it is considered as an offset in the regression function.

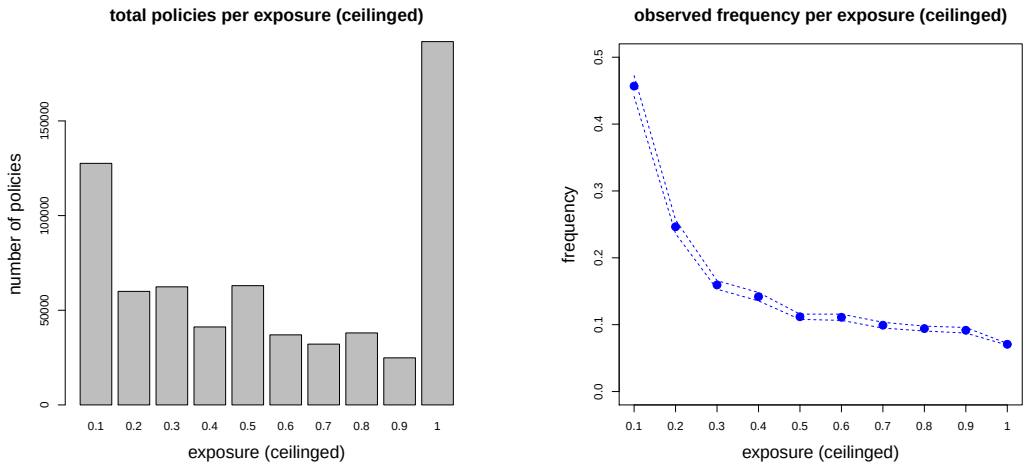

Figure 1: (lhs) histogram of the number of policies in each exposure group I_k, (rhs) claims frequency $\overline{f}_k$ per exposure group I_k, $k = 1, \ldots, 10$.

therefore build 10 exposure groups $I_k = \left(\frac{k-1}{10}, \frac{k}{10}\right]$, for $k = 1, \ldots, 10$, and we study the empirical frequencies on these exposure groups given by

$$\overline{f}_k = \frac{\sum_{i=1}^n N_i \, \mathbb{1}_{\{v_i \in I_k\}}}{\sum_{i=1}^n v_i \, \mathbb{1}_{\{v_i \in I_k\}}}.$$

In Figure 1 (lhs) and on the second line of Table 1 we provide the (relative) numbers of policies

exposure group I_k	1	2	3	4	5	6	7	8	9	10
rel. number of policies	18.8%	8.8%	9.2%	6.1%	9.3%	5.5%	4.7%	5.6%	3.7%	28.3%
empirical frequency $\overline{f}_k$	45.7%	24.6%	15.9%	14.2%	11.2%	11.1%	9.9%	9.4%	9.2%	7.1%

Table 1: relative number of policies in each exposure group and corresponding empirical frequencies $\overline{f}_k$, for $k = 1, \ldots, 10$.

in each exposure group $I_1, \ldots, I_{10}$. We observe that roughly 70% of the policies are exposed less than one accounting year. This seems to be a rather high percentage of policies that are only partially exposed during the accounting year, and this may be a peculiarity of the present insurance portfolio. In Figure 1 (rhs) and on the third line of Table 1 we provide the resulting empirical frequencies $\overline{f}_k$ on each exposure group $I_1, \ldots, I_{10}$. These empirical frequencies show a substantial decrease in increasing exposure. This clearly indicates that the expected number of claims may not be proportional to the time exposure $v_i > 0$.

Of course, the latter conclusion is not fully justified, because $\overline{f}_k$ only considers a *marginal* frequency, which may depend on the underlying portfolio structure. For this reason we also analyze interactions between the exposures v_i and the other feature components of $\boldsymbol{x}_i$. In Figure 2 we provide the bar plots of the relative numbers of policies in each exposure group $I_1, \ldots, I_{10}$ (red, orange, yellow, $\ldots$, blue, violet, magenta colors) for each label of each feature component of $\boldsymbol{x}_i$. We see non-homogeneity and rather strong interactions between the exposures v_i and some feature components of $\boldsymbol{x}_i$. For instance, the exposures are clearly increasing in driver's

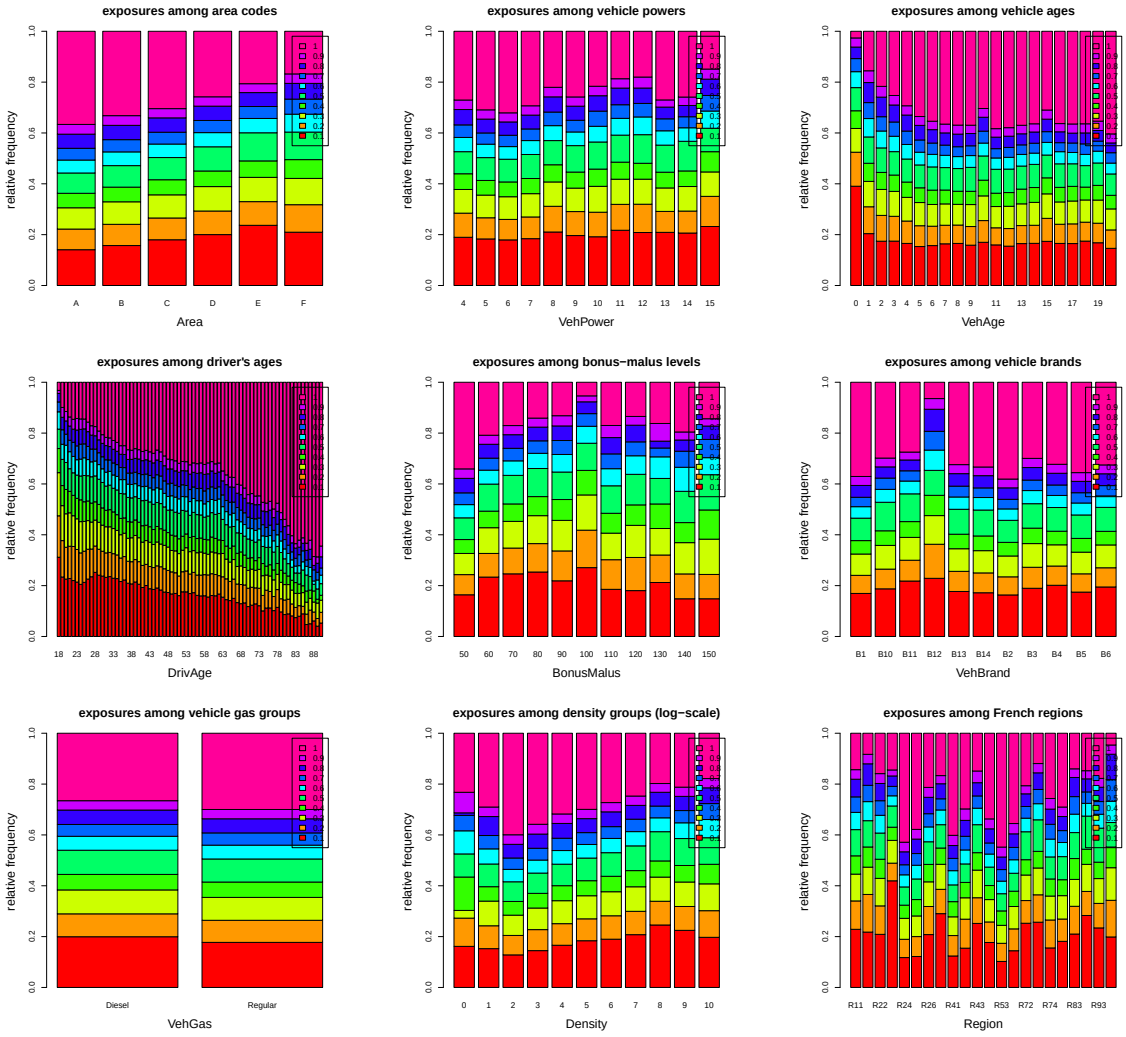

Figure 2: bar plots of the relative numbers of policies in each exposure group $I_1, \ldots, I_{10}$ (red, orange, yellow, ..., blue, violet, magenta) for all labels of all feature components of $\boldsymbol{x}_i$.

age, drivers with age 18 have hardly a full year of exposure, whereas 70% of the drivers with an age above 90 are exposed during the whole accounting year. Most of the other feature components show similar pictures, for instance, vehicle brand B12 is quite different compared to the other vehicle brands; a bonus-malus level of 100% has the shortest exposures (because this label contains new insurance contracts); vehicle ages 10 and 15 seem special (this may indicate newly bought second-hand cars where the age of the car is unknown). Such non-homogeneity may partially explain that $\overline{f}_k$ is non-constant in k. To get a clear answer to the question of the (non-)linear exposure measure we extend model assumption (1.1). This is done in the next section.

1.3 Warming-up exercise in neural network modeling

As a warming-up exercise we resume the neural network modeling approach of Section 6 of Noll et al. [18]. But we replace model assumption (1.1) by the following Poisson regression model

$$N_i \overset{\text{ind.}}{\sim} \text{Poi}\left(\mu(\boldsymbol{x}_i, v_i)\right), \tag{1.2}$$

where we a have regression function on the extended feature space $\mathcal{X}^+ = \mathcal{X} \times (0,1]$ given by

$$\mu : \mathcal{X}^+ \to \mathbb{R}_+, \qquad \text{with} \qquad (\boldsymbol{x}, v) \mapsto \mu(\boldsymbol{x}, v), \tag{1.3}$$

and where $\mathcal{X}$ is the feature space introduced in Section 6.1 of Noll et al. [18]. Thus, model (1.1) is obtained as a special case of model (1.2) by choosing the regression function in (1.3) as $\mu(\boldsymbol{x}, v) = \lambda(\boldsymbol{x})v$. Next, we introduce a fully-connected single hidden layer feed-forward neural network with $q_1 = 20$ hidden neurons for the modeling of the regression function (1.3). This is exactly the same model as in Section 6 of Noll et al. [18], except that we extend the input layer by the additional volume component $v \in (0,1]$. We introduce this neural network in a formal way because the corresponding notation will be used throughout this tutorial.
We start by defining a general feed-forward neural network, subsequently abbreviated as *network*. Choose $k \geq 1$ and hyperparameters $q_{k-1}, q_k \in \mathbb{N}$. A network *layer* is a mapping

$$\boldsymbol{z}^{(k)} : \mathbb{R}^{q_{k-1}} \to \mathbb{R}^{q_k}, \qquad \boldsymbol{z} \mapsto \boldsymbol{z}^{(k)}(\boldsymbol{z}) = \left(z_1^{(k)}(\boldsymbol{z}), \ldots, z_{q_k}^{(k)}(\boldsymbol{z})\right)', \tag{1.4}$$

with q_k *hidden neurons* in the (k*-th hidden*) layer given by

$$z_j^{(k)}(\boldsymbol{z}) = \phi\left(w_{j,0}^{(k)} + \sum_{l=1}^{q_{k-1}} w_{j,l}^{(k)} z_l\right) \overset{\text{def.}}{=} \phi\langle \boldsymbol{w}_j^{(k)}, \boldsymbol{z}\rangle, \qquad \text{for } j = 1, \ldots, q_k, \tag{1.5}$$

with weights $\boldsymbol{w}^{(k)} = (\boldsymbol{w}_1^{(k)}, \ldots, \boldsymbol{w}_{q_k}^{(k)})' = (w_{1,0}^{(k)}, \ldots, w_{q_k,q_{k-1}}^{(k)})' \in \mathbb{R}^{q_k(1+q_{k-1})}$ and activation function $\phi : \mathbb{R} \to \mathbb{R}$. We give some remarks:

- The k-th hidden layer is obtained by (feed-forward) mapping the q_{k-1} neurons $\boldsymbol{z}^{(k-1)} = (z_1^{(k-1)}, \ldots, z_{q_{k-1}}^{(k-1)})' \in \mathbb{R}^{q_{k-1}}$ to the q_k neurons $\boldsymbol{z}^{(k)} = (z_1^{(k)}, \ldots, z_{q_k}^{(k)})' \in \mathbb{R}^{q_k}$, see (1.4). Thus, layer $k-1$ has q_{k-1} neurons and layer k has q_k neurons. q_{k-1} and q_k are hyperparameters determining the architecture of the network. Moreover, we initialize q_0 to be the dimension of $\mathcal{X}^+$ with initial neurons $\boldsymbol{z}^{(0)} = (\boldsymbol{x}, v) \in \mathcal{X}^+$.

- The k-th hidden layer has $q_k(1+q_{k-1})$ parameters $\boldsymbol{w}^{(k)} \in \mathbb{R}^{q_k(1+q_{k-1})}$ called *weights*. They describe the scalar products $\langle \boldsymbol{w}_j^{(k)}, \boldsymbol{z}\rangle$ in the neurons, providing a reduction of dimension (in a first step) from q_{k-1} to 1, for indexes $j = 1, \ldots, q_k$. The intercepts $w_{j,0}^{(k)}$ are also called *biases* in the neural network literature.

- $\phi : \mathbb{R} \to \mathbb{R}$ is a (non-linear) activation function, which models the strengths of the activations in the neurons. Often, one of the following four choices is made

$$\phi(x) = \begin{cases} \frac{1}{1+e^{-x}} & \text{sigmoid activation function,} \\ \tanh(x) & \text{hyperbolic tangent activation function,} \\ \mathbb{1}_{\{x \geq 0\}} & \text{step function activation,} \\ x\mathbb{1}_{\{x \geq 0\}} & \text{rectified linear unit (ReLU) activation function.} \end{cases} \tag{1.6}$$

- For more comments we refer to Remarks 6.2 in Noll et al. [18].

A general network architecture with K hidden layers for our Poisson regression problem is obtained as follows: choose hyperparameters $q_1, \ldots, q_K \in \mathbb{N}$ and initialize q_0 to be the dimension of the feature space $\mathcal{X}^+ \subset \mathbb{R}^{q_0}$ (which provides the *input layer*). The network regression function is defined by the composition

$$\mu : \mathcal{X}^+ \to \mathbb{R}_+, \qquad (\boldsymbol{x}, v) \mapsto \mu(\boldsymbol{x}, v) = \left(g \circ \boldsymbol{z}^{(K)} \circ \cdots \circ \boldsymbol{z}^{(1)}\right)(\boldsymbol{x}, v),$$

with log-linear regression $g : \mathbb{R}^{q_K} \to \mathbb{R}_+$ defined by

$$\boldsymbol{z}^{(K)} \;\mapsto\; g(\boldsymbol{z}^{(K)}) = \exp\left(w_0^{(K+1)} + \sum_{j=1}^{q_K} w_j^{(K+1)} z_j^{(K)}\right) = \exp\langle \boldsymbol{w}^{(K+1)}, \boldsymbol{z}^{(K)} \rangle.$$

That is, this last layer (called *output layer*) only receives $q_{K+1} = 1$ neuron, and as activation function we choose the exponential function because claims frequencies should be strictly positive. This network architecture has *depth* K and receives a *network parameter* $\theta \in \mathbb{R}^r$ of dimension $r = \sum_{k=1}^{K+1} q_k(1 + q_{k-1})$ collecting all network weights $\boldsymbol{w}^{(k)}$, $k = 1, \ldots, K+1$. Two

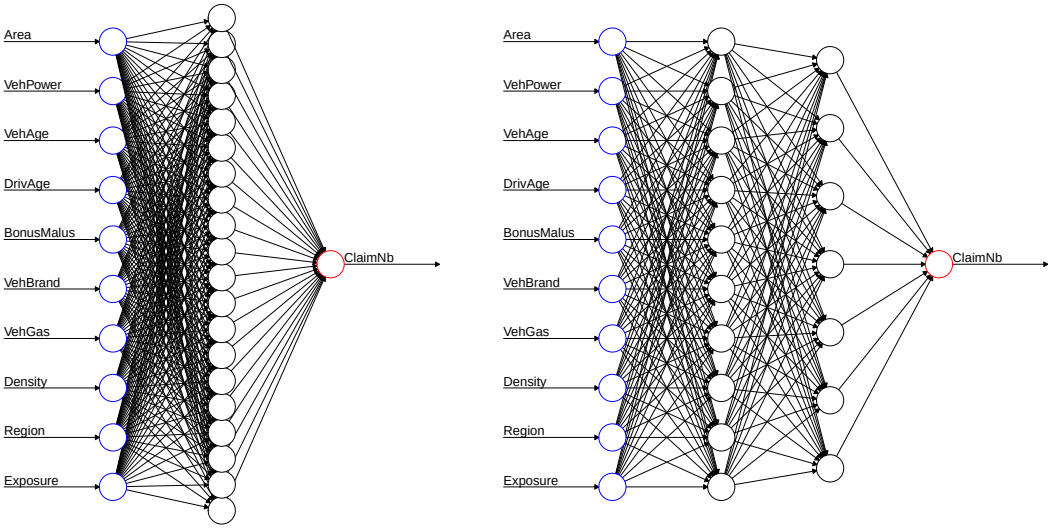

Figure 3: (lhs) (shallow) network with $K = 1$ hidden layer having $q_1 = 20$ neurons; (rhs) (deep) network with $K = 2$ hidden layers having $q_1 = 10$ and $q_2 = 7$ hidden neurons in the first and second hidden layer, respectively; input layers have dimension $q_0 = 10$ here; the input layer has blue color and the output layer has red color.

examples with $K = 1, 2$ hidden layers are given in Figure 3. These two examples have network parameters of dimensions $r = 241$ and $r = 195$, respectively.

For our warming-up example we choose exactly the same set-up as in Section 6 of Noll et al. [18], the only difference is that we now consider input $(\boldsymbol{x}, v) \in \mathcal{X}^+$ for modeling the expected number of claims $\mu(\boldsymbol{x}, v)$, whereas in [18] we have been considering input $\boldsymbol{x} \in \mathcal{X}$ for modeling the special case of $\lambda(\boldsymbol{x})v$. In particular, this modeling includes (i) feature pre-processing as in [18], (ii)

choice of a network with $K = 1$ hidden layer, (iii) choice of $q_1 = 20$ hidden neurons, and (iv) choice of hyperbolic tangent activation function $\phi(x) = \tanh(x)$. This corresponds to Figure 3 (lhs), and the network regression function reads as

$$(\boldsymbol{x}, v) \;\mapsto\; \mu(\boldsymbol{x}, v) = \exp\left(w_0^{(2)} + \sum_{j=1}^{q_1} w_j^{(2)} \tanh\left(w_{j,0}^{(1)} + \sum_{l=1}^{q_0-1} w_{j,l}^{(1)} x_l + w_{j,q_0}^{(1)} v \right) \right), \tag{1.7}$$

where the last terms $w_{j,q_0}^{(1)} v$, $j = 1, \ldots, q_1$, are the main difference to the model in [18]. These

	in-sample loss	out-of-sample loss
network with $q_1 = 20$ and $(\boldsymbol{x}, v) \mapsto \lambda(\boldsymbol{x})v$: model (1.1)	30.45048	31.58770
network with $q_1 = 20$ and $(\boldsymbol{x}, v) \mapsto \mu(\boldsymbol{x}, v)$: model (1.2)	29.40960	30.52430

Table 2: in-sample and out-of-samples losses of the network predictions for the two regression functions $(\boldsymbol{x}, v) \mapsto \lambda(\boldsymbol{x})v$ and $(\boldsymbol{x}, v) \mapsto \mu(\boldsymbol{x}, v)$; losses are in 10^{-2}.

modeling choices will be discussed in much more detail in the sections below.
For the moment, we just fit this model as described in Section 6 of [18], and we perform exactly the same in-sample and out-of-sample analysis as in [18]. The results are presented in Table 2. We observe a clear decrease in the resulting losses from the former model (1.1) to the latter model (1.2). This suggests that the time exposure $v \in (0, 1]$ should not be considered in a linear fashion, but the general approach with regression function μ as in (1.7) is more appropriate.

1.4 Designing a network

The choice of a particular network architecture and its calibration involve many steps which we are going to discuss in detail in this tutorial. This involves:

(a) data pre-processing

(b) choice of loss function (objective function) and performance measure for model calibration;

(c) number of hidden layers K;

(d) number of neurons $q_1, \ldots, q_K$ in the hidden layers;

(e) choice of activation function ϕ;

(f) optimization algorithm used for calibration which may include further choices of

  (i) initialization of algorithm,

  (ii) random (mini-)batches of data,

  (iii) stopping, number of iterations, number of epochs, etc.,

  (iv) parameters like learning rates, momentum parameters, etc.;

(g) normalization layers, dropout rates;

(h) regularization like LASSO or ridge regression, etc.

2 Data pre-processing: item (a)

In neural network modeling the choice of the scale of the feature components may substantially influence the predictive model. Therefore, data pre-processing requires careful consideration. We treat unordered categorical feature components (nominal) and continuous feature components (ordinal) separately. Ordered categorical feature components are treated like continuous ones, where we simply replace the ordered categorical labels by integers. Binary categorical feature components are coded by 0's and 1's for the two binary labels (for binary labels we do not distinguish between ordered and unordered components). Remark that if we choose an anti-symmetric activation function, i.e. $-\phi(x) = \phi(-x)$, we may also set binary categorical feature components to $\pm 1/2$, which may simplify initialization of optimization algorithms.

2.1 Unordered categorical feature components

We need to transform categorical (nominal) feature components to numerical values. The most commonly used transformations are the so-called *dummy coding* and the *one-hot encoding*. Both methods construct a binary representation for categorical labels. For dummy coding one label is chosen as reference label. Dummy coding then uses binary variables to indicate which label a particular policy possesses *if it differs* from the reference label. In our example we have two unordered categorical feature components, namely `VehBrand` and `Region`.[3] We use `VehBrand` as illustration. It has 11 different labels $\{\texttt{B1}, \texttt{B10}, \texttt{B11}, \texttt{B12}, \texttt{B13}, \texttt{B14}, \texttt{B2}, \texttt{B3}, \texttt{B4}, \texttt{B5}, \texttt{B6}\}$. We choose `B1` as reference label. Dummy coding then provides the coding scheme given in Table 3. We

label	feature components $\boldsymbol{x}^\star \in \{0,1\}^{10}$									
B1	0	0	0	0	0	0	0	0	0	0
B10	1	0	0	0	0	0	0	0	0	0
B11	0	1	0	0	0	0	0	0	0	0
B12	0	0	1	0	0	0	0	0	0	0
B13	0	0	0	1	0	0	0	0	0	0
B14	0	0	0	0	1	0	0	0	0	0
B2	0	0	0	0	0	1	0	0	0	0
B3	0	0	0	0	0	0	1	0	0	0
B4	0	0	0	0	0	0	0	1	0	0
B5	0	0	0	0	0	0	0	0	1	0
B6	0	0	0	0	0	0	0	0	0	1

Table 3: example dummy coding for categorical `VehBrand` labels.

observe that the 11 labels are replaced by 10-dimensional feature vectors $\boldsymbol{x}^\star$ in $\{0,1\}^{10}$, with components summing up to either 0 or 1 (row sums in Table 3).
In contrast to dummy coding, one-hot encoding does not choose a reference label, but uses an indicator for each label. In this way the 11 labels of `VehBrand` are replaced by the 11 unit vectors in $\mathbb{R}^{11}$.
Remark that other coding schemes could be used for categorical feature components such as Helmert's contrast coding. In classical generalized linear models (GLMs) the choice of the

[3] We treat the feature component `Area` as ordered categorical, as indicated in Noll et al. [18].

coding scheme typically does not influence the prediction (because one works with full rank matrices in a maximum likelihood estimation (MLE) framework), however, the interpretation of the results may change by considering a different contrast. In network modeling the choice of the coding scheme may influence the prediction: typically, we exercise an early stopping rule in network calibrations. This early stopping rule and the corresponding result may depend on any chosen modeling strategy, such as the encoding scheme of categorical feature components.
Remark that dummy coding and one-hot encoding may lead to very high-dimensional input layers in networks. For this reason, it is sometimes advantageous to use so-called embedding layers for categorical feature components. These embedding layers are, in some sense, close to categorical classes in a GLMs because each label receives its own parameters, see [22] for an example.

2.2 Continuous feature components

In theory, continuous feature components do not need pre-processing if we choose a sufficiently rich network, because the network may take care of feature components living on different scales. This statement is of purely theoretical value. In practice, continuous feature components need pre-processing such that they all live on a similar scale and such that they are sufficiently equally distributed across this scale. The reason for this requirement is that the calibration algorithms mostly use gradient descent methods (GDMs). These GDMs only work properly, if all components live on a similar scale and, thus, all directions contribute equally to the gradient. Otherwise, the optimization algorithms may get trapped in saddle points or in regions where the gradients are flat (also known as vanishing gradient problem). Often, one uses $[-1,1]$ as the common scale because the/our choice of activation function is concentrated on that scale, see (1.6); we also refer to Section 4.5, below.
A popular transformation is the so-called *MinMaxScaler*. For this transformation we fix each continuous feature component of $\boldsymbol{x}$, say x_l, at a time. Denote the minimum and the maximum of the domain of x_l by m_l and M_l, respectively. The MinMaxScaler then replaces

$$x_l \;\mapsto\; x_l^\star = \frac{2(x_l - m_l)}{M_l - m_l} - 1 \;\in\; [-1,1].$$

In practice, it may happen that the minimum m_l or the maximum M_l are not known. In this case one chooses the corresponding minimum and/or maximum of the features in the observed data. For prediction under new features one then needs to keep the original scaling of the initially observed data, i.e. the one which has been used for model calibration.
Another popular transformation considers the empirical residuals of each continuous feature component over the entire portfolio. For this we denote by $\overline{x}_l$ and s_l the empirical mean and standard deviation of the continuous feature component $x_{i,l}$ over the portfolio $i = 1, \ldots, n$. We then replace

$$x_l \;\mapsto\; x_l^\star = \frac{x_l - \overline{x}_l}{s_l}.$$

The empirical mean and standard deviation should only be based on the learning data $\mathcal{D}$, because the test data $\mathcal{T}$ should "truly" be unseen during learning; see Section 3.4 below for the definition of the learning data $\mathcal{D}$ and the test data $\mathcal{T}$. However, exactly the same scaling constants should be applied to *all* features.

Remark that if we have outliers, the above transformations may lead to very concentrated transformed feature components $x_{i,l}^\star$, $i = 1, \ldots, n$, because the outliers may, for instance, dominate the maximum in the MinMaxScaler. In this case, feature components should be transformed first by a log-transformation or by a quantile transformation so that they become more equally spaced (and robust).

Conclusion.
In our example we use dummy coding for the feature components `VehBrand` and `Region`. We use the MinMaxScaler for `Area` (after transforming $\{\mathrm{A}, \ldots, \mathrm{F}\}$ to $\{1, \ldots, 6\}$), `VehPower`, `VehAge` (after capping at age 20), `DrivAge` (after capping at age 90), `BonusMalus` (after capping at level 150) and `Density` (after first taking the log-transform). `VehGas` we transform to $\pm 1/2$ and the volume `Exposure` $\in (0, 1]$ we keep untransformed. The resulting feature space $\mathcal{X}^+$ has dimension $q_0 = 39$ (note that this differs from Figure 3 because we now use dummy coding for categorical feature components which turns `VehBrand` into a 10-dimensional dummy vector and `Region` into a 21-dimensional dummy vector).

3 Choice of loss function (objective function): item (b)

3.1 Poisson deviance loss function

For claims frequency modeling we make the Poisson model assumption (1.2). The natural choice of an objective function under this model assumption is the Poisson deviance loss. For a given estimator $\widehat{\mu}(\cdot)$ of $\mu(\cdot)$, the Poisson deviance loss on the data $\mathcal{D} = \{(N_i, \boldsymbol{x}_i, v_i) : i = 1, \ldots, n\}$ is given by

$$\mathcal{L}(\mathcal{D}, \widehat{\mu}(\cdot)) \;=\; \frac{1}{n} \sum_{i=1}^{n} 2N_i \left[\frac{\widehat{\mu}(\boldsymbol{x}_i, v_i)}{N_i} - 1 - \log\left(\frac{\widehat{\mu}(\boldsymbol{x}_i, v_i)}{N_i} \right) \right]. \tag{3.1}$$

If we consider a homogeneous expected frequency model $\mu(\boldsymbol{x}, v) = \lambda v$, the minimizer $\widehat{\mu}(\boldsymbol{x}, v) = \widehat{\lambda} v$ of the deviance loss (3.1) is obtained by the MLE

$$\widehat{\lambda}^{\mathrm{MLE}} = \frac{\sum_{i=1}^n N_i}{\sum_{i=1}^n v_i}.$$

In this homogeneous model the MLE is unbiased and its uncertainty can be quantified:

$$\mathbb{E}\left[\widehat{\lambda}^{\mathrm{MLE}}\right] = \lambda \qquad \text{and} \qquad \mathrm{Var}\left(\widehat{\lambda}^{\mathrm{MLE}}\right) = \frac{\lambda}{\sum_{i=1}^n v_i}.$$

We remark that on the set of distribution functions with finite first moment, the Poisson deviance scoring/loss function is strictly consistent for the expected value, see Definition 2.1 in [8].

3.2 Square loss functions

A second option that is often considered for the objective function are square losses and weighted square losses. These are given by, respectively,

$$\mathcal{L}(\mathcal{D}, \widehat{\mu}(\cdot)) \;=\; \frac{1}{n} \sum_{i=1}^{n} \left(N_i - \widehat{\mu}(\boldsymbol{x}_i, v_i)\right)^2,$$

and

$$\mathcal{L}(\mathcal{D}, \widehat{\mu}(\cdot)) \;=\; \frac{1}{n}\sum_{i=1}^{n} \frac{1}{\widehat{\mu}(\boldsymbol{x}_i, v_i)} \left(N_i - \widehat{\mu}(\boldsymbol{x}_i, v_i)\right)^2 .$$

These choices are motivated by the properties, under model assumption (1.2),

$$\mathbb{E}[N_i] = \mu(\boldsymbol{x}_i, v_i) \qquad \text{and} \qquad \mathrm{Var}(N_i) = \mu(\boldsymbol{x}_i, v_i).$$

In general, we do not recommend these latter two loss functions: the square loss does not consider the underlying volumes appropriately, and the weighted square loss is not robust because the estimated frequency $\widehat{\mu}(\cdot)$ appears in the denominator of the loss function.

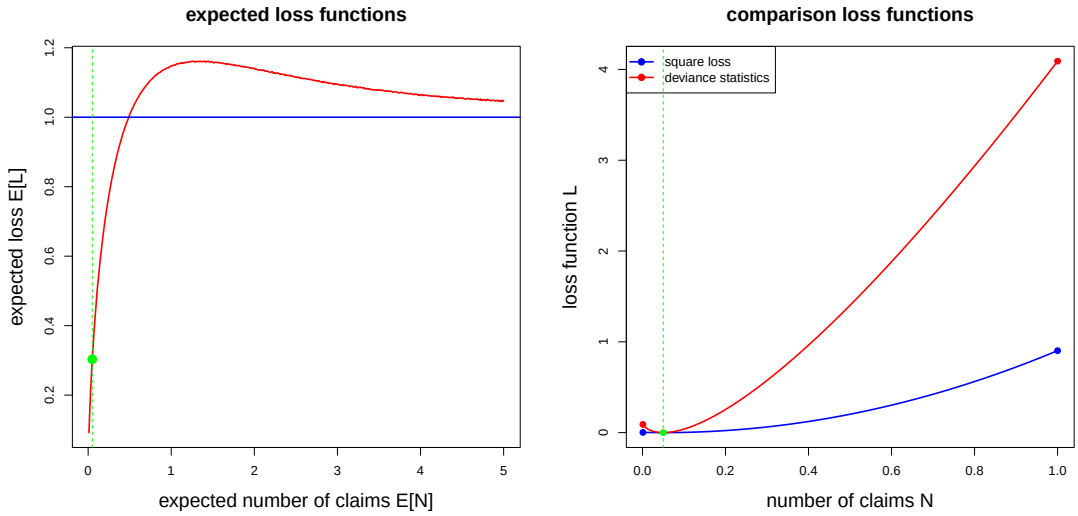

Figure 4: (lhs) expected deviance loss $\mathbb{E}[N] \mapsto \mathbb{E}[D^\star]$ (red color) and expected weighted square loss $\mathbb{E}[N] \mapsto \mathbb{E}[\varepsilon^2]$ (blue color) both as a function of $\mathbb{E}[N] = \mu(\boldsymbol{x}, v)$; (rhs) deviance loss and square loss as a function of the observed number of claims $N \in \{0, 1\}$ (red and blue dots) for $\mathbb{E}[N] = \mu(\boldsymbol{x}, v) = 5\%$.

In Figure 4 (lhs) we plot in red color the expected deviance loss

$$\begin{aligned} \mathbb{E}[D^\star] &= \mathbb{E}\left[2\left(\mu(\boldsymbol{x}, v) - N - N\log(\mu(\boldsymbol{x}, v)/N)\right)\right] \\ &= -2\mu(\boldsymbol{x}, v)\log\mu(\boldsymbol{x}, v) + 2\mathbb{E}[N\log N], \end{aligned} \tag{3.2}$$

as a function of the expected number of claims $\mathbb{E}[N] = \mu(\boldsymbol{x}, v)$ of a Poisson random variable. This expected deviance loss is around $30.3 \cdot 10^{-2}$ for an expected value of $\mathbb{E}[N] = 5\%$ (see green illustration in Figure 4 (lhs)). This order of magnitude is in line with Table 2. The blue line in Figure 4 (lhs) illustrates the expected loss of the weighted square loss function given by

$$\mathbb{E}[\varepsilon^2] = \mathbb{E}\left[(N - \mu(\boldsymbol{x}, v))^2/\mu(\boldsymbol{x}, v)\right] = 1,$$

which of course is equal to 1 under the corresponding Poisson assumption.[4]

[4]Remark that Figure 4 (lhs) exactly illustrates the expected dispersion estimates (deviance (red) and Pearson's (blue)) of the Poisson model, see Section 7.3.3 in [26].

In Figure 4 (rhs) we plot the deviance loss and the weighted square loss function

$$N \mapsto D^\star = 2\left[\mu(\boldsymbol{x},v) - N - N\log\left(\frac{\mu(\boldsymbol{x},v)}{N}\right)\right] \qquad \text{and} \qquad N \mapsto \varepsilon^2 = \frac{(N-\mu(\boldsymbol{x},v))^2}{\mu(\boldsymbol{x},v)},$$

for a given expected number of claims of $\mathbb{E}[N] = \mu(\boldsymbol{x},v) = 5\%$. Note that $N \in \{0,1,2,\ldots\}$ can only take integer values (whereas the red and blue lines in Figure 4 (rhs) illustrate $N \in [0,1]$). The green line in Figure 4 (rhs) gives the expected number of claims. A realization $N = 0$ only contributes a small amount to the loss, and $N = 1$ contributes hugely to the loss (more pronounced for the deviance loss). On the other hand, the sensitivity in a slight change of $\mu(\boldsymbol{x},v)$ has a comparably small influence, as can also be seen from Figure 4 (rhs). This shows that the loss functions will be largely dominated by the pure randomness in the realizations of N, and slight modifications in the model $\mu(\boldsymbol{x},v)$ can only hardly be detected. This is a common problem in low frequency problems (and also relates to the class imbalance problem in machine learning, see also next section).

Remark. In view of (3.2) we can minimize the expected deviance loss w.r.t. to the unknown parameter μ of N, i.e. we may consider

$$\widehat{\mu}^* = \arg\min_{\widehat{\mu}} \mathcal{R}(\mu|\widehat{\mu}) = \arg\min_{\widehat{\mu}} \mathbb{E}\left[2\left(\widehat{\mu} - N - N\log(\widehat{\mu}/N)\right)\right].$$

In [8] this minimizer is called the optimal point forecast for N under the Poisson deviance scoring function. This minimizer is given by $\widehat{\mu}^* = \mathbb{E}[N]$ which implies that this scoring function is (strictly) consistent for the expected value, see Theorem 2.2 in [8].

3.3 Binary loss functions

Often, claims frequencies are very small in insurance. In Table 4 we provide the policies with the

number of claims N_i	0	1	2	3	4	5	6	8	9	11	16
number of policies	643'953	32'178	1'784	82	7	2	1	1	1	3	1
total exposures v_i	336'616	20'671	1'153	53	3	1	0.3	0.4	0.1	1.1	0.3

Table 4: split of the portfolio w.r.t. number of claims.

corresponding numbers of claims. We observe that more than 90% of the policies do not suffer a claim. For this reason, one often speaks about a class imbalance because the outcome $N_i = 0$ is by far the most common one. Since the event $\{N_i > 1\}$ is even much less likely than the event $\{N_i = 1\}$, one is tempted to replace the Poisson problem by a binomial problem (binary classification problem). This may come at the *loss* of *some* information. We replace N_i by

$$Y_i = \mathbb{1}_{\{N_i \geq 1\}}. \tag{3.3}$$

Y_i has a binomial distribution under model assumption (1.2) with success probability

$$p(\boldsymbol{x}_i, v_i) = \mathbb{P}[Y_i = 1] = 1 - \mathbb{P}[Y_i = 0] = 1 - \mathbb{P}[N_i = 0] = 1 - \exp\{-\mu(\boldsymbol{x}_i, v_i)\}.$$

We can now apply binary classification to estimate $\widehat{p}(\boldsymbol{x}_i, v_i)$ which in turn provides estimator

$$\widehat{\mu}(\boldsymbol{x}_i, v_i) = -\log\left(1 - \widehat{p}(\boldsymbol{x}_i, v_i)\right). \tag{3.4}$$

The binomial model proposes the deviance loss function

$$\begin{aligned}\mathcal{L}(\mathcal{D}, \widehat{p}(\cdot)) &= -\frac{2}{n}\sum_{i=1}^{n} Y_i \log\left(\widehat{p}(\boldsymbol{x}_i, v_i)\right) + (1-Y_i)\log\left(1-\widehat{p}(\boldsymbol{x}_i, v_i)\right) \\ &= -\frac{2}{n}\sum_{i=1}^{n} Y_i \log\left(\frac{\widehat{p}(\boldsymbol{x}_i, v_i)}{1-\widehat{p}(\boldsymbol{x}_i, v_i)}\right) + \log\left(1-\widehat{p}(\boldsymbol{x}_i, v_i)\right).\end{aligned}$$

Observe that the first term under the sum on the second line is the logit transform of $\widehat{p}(\boldsymbol{x}_i, v_i)$. The first line gives the cross-entropy or Kullback-Leibler loss if we calculate its expected value w.r.t. Y_i, i.e. the Kullback-Leibler loss of $\widehat{p}(\boldsymbol{x}_i, v_i)$ w.r.t. $p(\boldsymbol{x}_i, v_i)$ is given by

$$\mathcal{R}(p|\widehat{p}) = \frac{1}{2}\mathbb{E}_Y\left[\mathcal{L}(\mathcal{D}, \widehat{p}(\cdot))\right] = -\frac{1}{n}\sum_{i=1}^{n} p(\boldsymbol{x}_i, v_i)\log\left(\widehat{p}(\boldsymbol{x}_i, v_i)\right) + (1-p(\boldsymbol{x}_i, v_i))\log\left(1-\widehat{p}(\boldsymbol{x}_i, v_i)\right),$$

where the expected values $\mathbb{E}_Y$ are applied to Y_i. The square loss function in the binomial model reads as

$$\mathcal{L}(\mathcal{D}, \widehat{p}(\cdot)) = \frac{1}{n}\sum_{i=1}^{n}\left(\widehat{p}(\boldsymbol{x}_i, v_i) - Y_i\right)^2 = \frac{1}{n}\sum_{i=1}^{n} Y_i\left(1-\widehat{p}(\boldsymbol{x}_i, v_i)\right)^2 + (1-Y_i)\widehat{p}(\boldsymbol{x}_i, v_i)^2,$$

where for the last identity we use $Y_i \in \{0, 1\}$. This provides expected square loss w.r.t. Y_i

$$\mathcal{R}(p|\widehat{p}) = \mathbb{E}_Y\left[\mathcal{L}(\mathcal{D}, \widehat{p}(\cdot))\right] = \frac{1}{n}\sum_{i=1}^{n} p(\boldsymbol{x}_i, v_i)\left(1-\widehat{p}(\boldsymbol{x}_i, v_i)\right)^2 + (1-p(\boldsymbol{x}_i, v_i))\widehat{p}(\boldsymbol{x}_i, v_i)^2.$$

If we minimize these expected losses, say for one risk only, we obtain

$$\mathcal{I}(p) = \min_{\widehat{p}} \mathcal{R}(p|\widehat{p}) = \begin{cases} -p\log p - (1-p)\log(1-p) & \text{entropy impurity function,} \\ p(1-p) & \text{Gini impurity function.} \end{cases}$$

Thus, the deviance loss and the square loss are directly related to the entropy and the Gini impurity functions, respectively, we also refer to Figure 6.7 in [27] on binary classification trees.

3.4 Conclusion of Sections 3.1-3.3 on the choice of the loss function

The detour via binary classification for an imbalanced Poisson prediction problem may have another disadvantage besides a potential loss of information due to (3.3). Namely, it may induce a bias because, using Jensen's inequality, we obtain from identity (3.4)

$$\mathbb{E}\left[\widehat{\mu}(\boldsymbol{x}_i, v_i)\right] \geq -\log\left(1-\mathbb{E}\left[\widehat{p}(\boldsymbol{x}_i, v_i)\right]\right).$$

For these reasons we work with Poisson deviance loss (3.1) as objective function. Model calibration is then done by making this deviance loss function small w.r.t. the estimator $\widehat{\mu}(\cdot)$ on the given data $\mathcal{D}$ (*learning data*). Since the resulting *in-sample loss* (3.1) is prone to over-fitting, we typically evaluate the quality of the fit by calculating an *out-of-sample loss* on *test data* $\mathcal{T} = \{(N_t, \boldsymbol{x}_t, v_t) : t = 1, \ldots, n_{\mathcal{T}}\}$ that is disjoint from $\mathcal{D}$ (which has been used to fit $\widehat{\mu}(\cdot)$):

$$\mathcal{L}(\mathcal{T}, \widehat{\mu}(\cdot)) = \frac{1}{n_{\mathcal{T}}}\sum_{t=1}^{n_{\mathcal{T}}} 2N_t\left[\frac{\widehat{\mu}(\boldsymbol{x}_t, v_t)}{N_t} - 1 - \log\left(\frac{\widehat{\mu}(\boldsymbol{x}_t, v_t)}{N_t}\right)\right]. \tag{3.5}$$

In all our analysis we use exactly the two data sets constructed in Section 2 of Noll et al. [18]. The learning data $\mathcal{D}$ has $n = 610'212$ policies and the test data $\mathcal{T}$ has $n_{\mathcal{T}} = 67'801$ policies, we also refer to Table 3 in [18]. A first illustrative example has already been provided in Table 2.

4 Optimization algorithms: item (f)

In the remainder of this manuscript we consider different network models (network architectures). We fit these models to the learning data $\mathcal{D}$ by making the in-sample losses $\mathcal{L}(\mathcal{D}, \widehat{\mu}(\cdot))$, given in (3.1), small. We assess the quality of these fits by considering the corresponding out-of-sample losses $\mathcal{L}(\mathcal{T}, \widehat{\mu}(\cdot))$, given in (3.5), on the test data $\mathcal{T}$.
A first naïve approach would aim at minimizing $\mathcal{L}(\mathcal{D}, \widehat{\mu}(\cdot))$ in $\widehat{\mu}(\cdot)$, this would provide the MLE for the corresponding Poisson network regression model. This approach is naïve because, typically, networks are over-parametrized and the MLE would heavily over-fit to the data $\mathcal{D}$. This would lead to a poor out-of-sample performance on $\mathcal{T}$, because an over-fitted model does not generalize to other data. Therefore, in network calibrations we are *not* interested in finding the MLE, but we would like to find a sufficiently good parametrization, which also has a good out-of-sample performance (generalization). Having this said, it is clear that typically in network calibrations there is a lot of redundancy[5] which results in many competing predictive models of similar quality. We briefly explain this based on the toy example given in Figure 5.

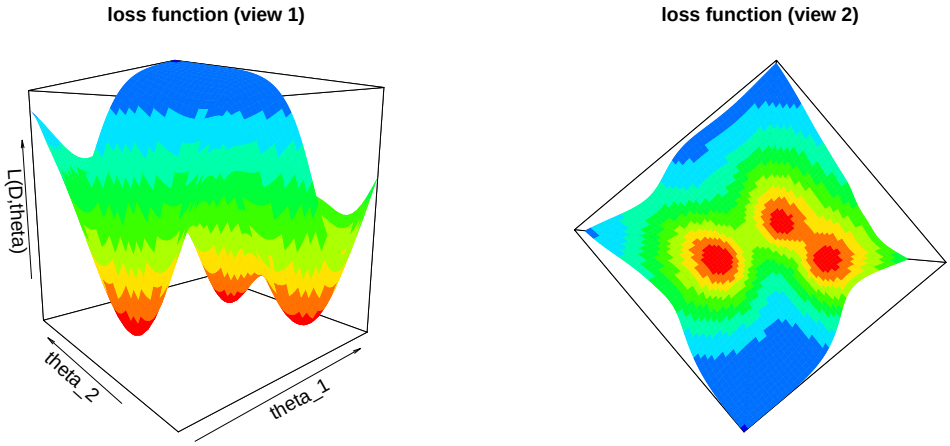

Figure 5: surface of the loss function $\theta \mapsto \mathcal{L}(\mathcal{D}, \theta)$ as a function of a two-dimensional network parameter $\theta = (\theta_1, \theta_2)'$: the two plots provide the loss surface from two different angles.

In Figure 5 we plot the surface of a loss function (objective function) $\theta \mapsto \mathcal{L}(\mathcal{D}, \theta)$ as a function of a two-dimensional parameter $\theta = (\theta_1, \theta_2)'$. The two graphs illustrate the same surface from two different angles. This objective function is assumed to have 3 different local minima (red color in plots), i.e. three extremal points for the choice of the parameter θ. If we assume that the red and yellow colors indicate parameter choices θ that over-fit to the data $\mathcal{D}$ (very small in-sample loss), we would choose a network parameter θ in the green part of the surface. Having continuity in θ immediately proves that there are infinitely many competing models with a similar performance (green area in the plots). Thus, choosing a sufficiently good parametrization means that we

[5] Recall that we try to fit a regression model to "noisy" observations which may lead to over-fitting if the regression function follows these noisy observations too closely. This is different compared to network approximations to (given) deterministic functions where we do not have the problem (and notion) of over-fitting.

choose a network parameter θ providing a loss in the green area of Figure 5, and there does not exist a unique best choice. A similar consideration holds true if we compare different network architectures.

4.1 Stochastic gradient descent method

Terminology: shallow and deep network.
Networks with one hidden layer $K = 1$ are called *shallow networks*, networks with multiple hidden layers $K > 1$ are called *deep networks*. A shallow and a deep network example are provided in Figure 3.

For the moment we choose a fixed network architecture consisting of a shallow network having $q_1 = 20$ hidden neurons in the single hidden layer, see Figure 3 (lhs). As activation function we choose the hyperbolic tangent. These choices provide exactly the regression function introduced in (1.7). The choice of the network architecture is often called "selection of hyperparameters". For the moment, we assume that these hyperparameters are given, and we discuss model calibration for such a given architecture. The choice of the hyperparameters is discussed in detail in Section 6, below.

The plain-vanilla method for network calibration is the gradient descent method (GDM), additionally using back-propagation for an efficient evaluation of the gradients. In GDMs, we study the in-sample loss $\mathcal{L}(\mathcal{D}, \mu(\cdot))$ as the objective function in the network parameter θ, that is, we consider

$$\theta \;\mapsto\; \mathcal{L}(\mathcal{D}, \mu_\theta(\cdot)), \tag{4.1}$$

where $\mu(\cdot) = \mu_\theta(\cdot)$ is the network regression function (1.7), and where the network parameter θ collects all the weights $\boldsymbol{w}^{(k)}$, $k = 1, \ldots, K+1$, see Section 1.3. The GDM looks iteratively for the direction of the maximal local decrease of the loss function (4.1) which is given by the negative gradient of $\mathcal{L}(\mathcal{D}, \mu_\theta(\cdot))$ w.r.t. θ, i.e. for the Poisson deviance loss function we have

$$-\nabla_\theta \mathcal{L}(\mathcal{D}, \mu_\theta(\cdot)) \;=\; -\frac{1}{n}\sum_{i=1}^n 2\left[\mu_\theta(\boldsymbol{x}_i, v_i) - N_i\right] \nabla_\theta \log(\mu_\theta(\boldsymbol{x}_i, v_i)).$$

A small step $\varrho > 0$ (called *learning rate*) into the direction of the negative gradient, that is, an update $\theta \mapsto \widetilde{\theta} = \theta - \varrho\, \nabla_\theta \mathcal{L}(\mathcal{D}, \mu_\theta(\cdot))$, will lead to a (maximal local) decrease in loss of size

$$\mathcal{L}(\mathcal{D}, \mu_{\widetilde{\theta}}(\cdot)) \;=\; \mathcal{L}(\mathcal{D}, \mu_\theta(\cdot)) - \varrho\, \|\nabla_\theta \mathcal{L}(\mathcal{D}, \mu_\theta(\cdot))\|^2 + o(\varrho), \qquad \text{as } \varrho \downarrow 0.$$

This is the locally optimal move from θ to $\widetilde{\theta}$ in the network parameter. Back-propagation is then used to efficiently calculate these (negative) gradients, for more theory and explanation on the back-propagation method we refer to Nielsen [17]. There are different versions of this GDM that explore optimal learning rates $\varrho > 0$, momentum-based improvements, Nesterov acceleration, etc. All this variants of the GDM aim at speeding up the convergence of the algorithm by not only considering the last optimal move. We briefly mention some of them which are available in the library Keras[6], for a more detailed description we refer to Sections 8.3 and 8.5 in Goodfellow et al. [9].

[6]Keras is a user-friendly API to TensorFlow, see https://tensorflow.rstudio.com/keras/

Listing 2: optimizer 'sgd'

```
optimizer_sgd(lr = 0.01, momentum = 0, decay = 0, nesterov = FALSE,
                                                  clipnorm = NULL, clipvalue = NULL)
```

Predefined gradient descent methods.

- The stochastic gradient descent method, called 'sgd',[7] can be fine-tuned for the speed of convergence by using optimal learning rates, momentum-based improvements, the Nesterov acceleration and optimal batches, see Listing 2;
- 'adagrad' chooses learning rates that differ in all directions of the gradient and that consider the directional sizes of the gradients ('ada' stands for adapted);
- 'adadelta' is a modified version of 'adagrad' that overcomes some deficiencies of the latter, for instance, the sensitivity to hyperparameters;
- 'rmsprop' is another method to overcome the deficiencies of 'adagrad' ('rmsprop' stands for root mean square propagation);
- 'adam' stands for adaptive moment estimation, similar to 'adagrad' it searches for directionally optimal learning rates based on the momentum induced by past gradients measured by an ℓ^2-norm;
- 'adamax' considers optimal learning rates as 'adam' but based on the ℓ^∞-norm;
- 'nadam' is a Nesterov accelerated version of 'adam'.

Listing 3: R script for fitting networks in Keras

```
1  library(keras)
2
3  model <- keras_model_sequential()
4  model %>%
5    layer_dense(units = q1, activation = 'tanh', input_shape = c(ncol(Xlearn))) %>%
6    layer_dense(units = 1, activation = k_exp)
7
8  summary(model)
9
10 model %>% compile(
11   loss = 'poisson',
12   optimizer ='sgd'
13 )
14
15 fit <- model %>% fit(Xlearn, learn$ClaimNb, epochs=100, batch_size=10000)
```

In order to perform the network calibration we use the R interface to Keras. The corresponding code is provided in Listing 3. On line 3 we initialize our `model` which uses TensorFlow backend.[8] On lines 4-6 we define a (fully-connected, feed-forward) shallow network with q_1 hidden neurons

[7] The letter 's' in 'sgd' stands for the stochastic choice of subsamples (batches) of a given size in the GDM.
[8] see https://keras.io/backend/

and hyperbolic tangent activation function, the output layer has exponential activation function, see also (1.7). Since the dimension of the feature space is $q_0 = 39$ we receive a $r = 821$-dimensional network parameter θ (this can be displayed by the command on line 8). Lines 10-13 compile the `model`, using the Poisson deviance loss function as objective function, finally on line 15 the `model` is fitted. For this fitting we only use the learning data $\mathcal{D}$ (encoded in the design matrix `Xlearn` and the responses `learn$ClaimNb`). `epochs` provides the number of times the entire learning data is run through the gradient descent algorithm. Since, typically, it is too costly to handle the entire learning data at once, the data is partitioned (randomly) into batches of size `batch_size`. Note that this partitioning of the data is of particular interest if we work with big data because it allows us to explore the data sequentially.

optimizer	epochs	batch size	run time	in-sample loss	out-of-sample loss
'sgd'	100	10'000	88.09s	30.15042	31.36468
'adagrad'	100	10'000	87.50s	29.47481	30.66941
'adadelta'	100	10'000	90.32s	29.44988	30.60121
'rmsprop'	100	10'000	89.51s	29.33483	30.51893
'adam'	100	10'000	90.28s	29.33336	30.50379
'adamax'	100	10'000	90.05s	29.45216	30.63665
'nadam'	100	10'000	90.25s	29.27684	30.42072

Table 5: results of the GDM illustrated in Listing 3 for different optimizers, always using the same initial parameter for θ; shallow network with $q_1 = 20$; losses are in 10^{-2}.

We run the GDM provided in Listing 3 for the different optimizers introduced above. For each optimizer we use the same initial parameter for θ (which provides a reasonable starting value for the algorithm).[9] The results are given in Table 5.[10] We note that the improved versions of the GDM, like 'rmsprop' or 'nadam' provide clearly better convergence results than the (plain-vanilla) 'sgd' of Listing 2. Since fine-tuning the 'sgd' for learning rates, etc., is too time-consuming we continue with the pre-specified optimizers 'rmsprop' and 'nadam'. From this table we see that the results of Table 2 can be obtained in roughly 90 seconds on a personal laptop with CPU @ 2.50GHz (4 CPUs) with 16GB RAM (which has a comparably modest computational power).

4.2 Comparison of network calibrations

We analyze the network calibrations obtained from the different optimizers presented in Table 5. They all have similar in-sample and out-of-sample performances and we would like to compare the resulting network parameters $\theta \in \mathbb{R}^r$. Recall that we have used the same hyperparameters and the same initial values for θ in all these calibrations.
In order to compare the resulting network parameters we first need to transform them because, in general, network parametrizations are not uniquely identifiable. For instance, if we have an

[9] The initial parameter is available from: https://github.com/JSchelldorfer/ActuarialDataScience/tree/master/2 - Insights from Inside Neural Networks

[10] We note that a deficiency in our use of the R interface to Keras is that we cannot reproduce the results because we cannot initialize a seed in the 'sgd' calibration. In fact, this is a deficiency discussed in the community that setting a seed for these algorithms sometimes works and sometimes it does not.

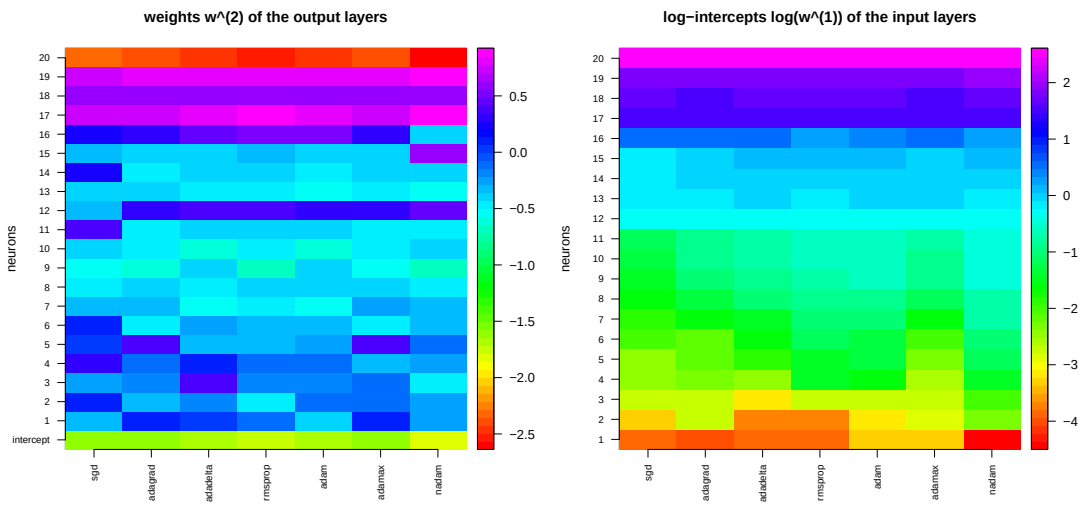

Figure 6: comparison of the calibrations obtained by the optimizers 'sgd', 'adagrad', 'adadelta', 'rmsprop', 'adam', adamax' and 'nadam': (lhs) output weights $\boldsymbol{w}^{(2)} = (w_j^{(2)})_{j=0,\ldots,q_1}$; (rhs) intercepts $0 < w_{1,0}^{(1)} < \ldots < w_{q_1,0}^{(1)}$ of the inputs (on the log-scale).

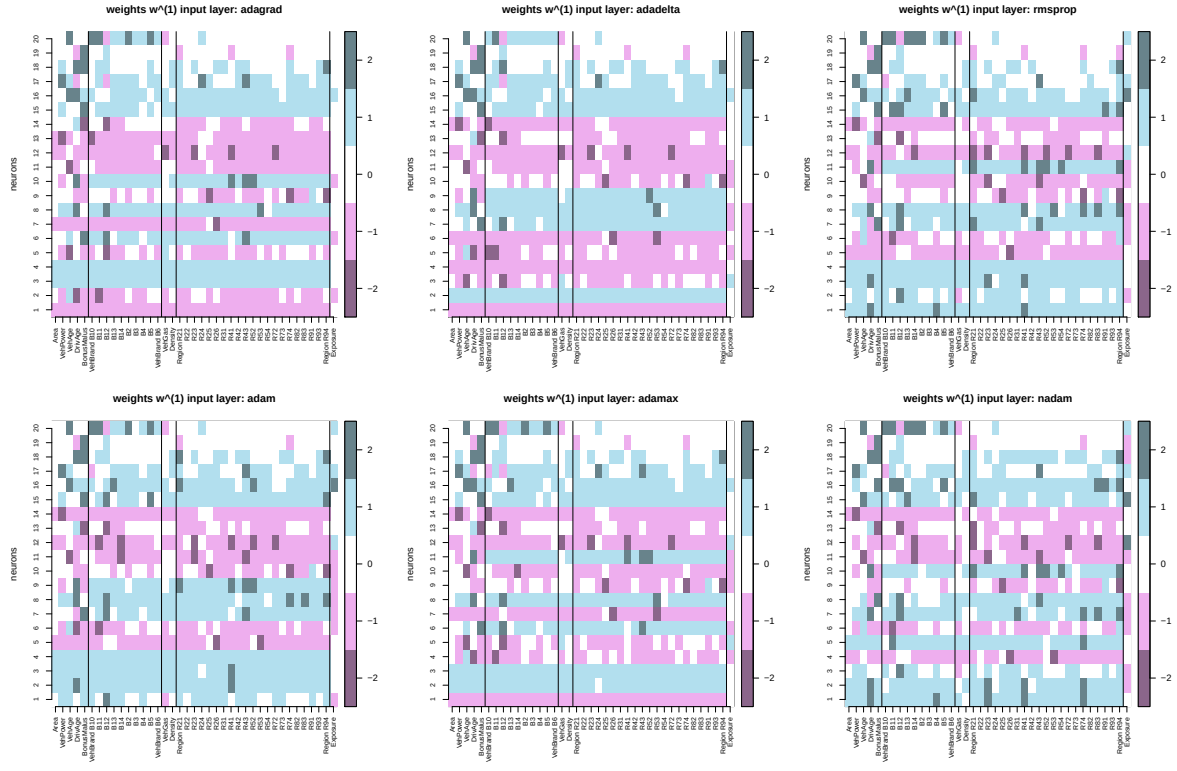

Figure 7: comparison of the calibrations obtained by the optimizers 'adagrad', 'adadelta', 'rmsprop', 'adam', adamax' and 'nadam': input weights $(w_{j,l}^{(1)})_{j=1,\ldots,q_1;l=1,\ldots,q_0}$.

anti-symmetric activation function $-\phi(x) = \phi(-x)$ we can perform the following sign switch operations in regression function (1.7):

$$\begin{aligned}\log \mu(\boldsymbol{x}, v) &= w_0^{(2)} + \sum_{j=1}^{q_1} w_j^{(2)} \phi\left(w_{j,0}^{(1)} + \sum_{l=1}^{q_0-1} w_{j,l}^{(1)} x_l + w_{j,q_0}^{(1)} v \right) \qquad (4.2)\\ &= w_0^{(2)} + \sum_{j\neq k} w_j^{(2)} \phi\left(w_{j,0}^{(1)} + \sum_{l=1}^{q_0-1} w_{j,l}^{(1)} x_l + w_{j,q_0}^{(1)} v \right) - w_k^{(2)} \phi\left(-w_{k,0}^{(1)} - \sum_{l=1}^{q_0-1} w_{k,l}^{(1)} x_l - w_{k,q_0}^{(1)} v \right).\end{aligned}$$

Thus, the two network parameters

$$\begin{aligned}\theta &= (w_{1,0}^{(1)}, \ldots, w_{k,l}^{(1)}, \ldots, w_{q_1,q_0}^{(1)}, w_0^{(2)}, \ldots, w_k^{(2)}, \ldots, w_{q_1}^{(2)})' \qquad \text{and}\\ \theta^- &= (w_{1,0}^{(1)}, \ldots, -w_{k,l}^{(1)}, \ldots, w_{q_1,q_0}^{(1)}, w_0^{(2)}, \ldots, -w_k^{(2)}, \ldots, w_{q_1}^{(2)})'\end{aligned}$$

give the same prediction (where we switch all signs that belong to index k). Secondly, enumeration of the hidden neurons may be permuted, still providing the same prediction. We solve this identifiability issue for anti-symmetric activation functions by considering a fundamental domain as introduced by Rüger–Ossen [23]. For a general network parameter θ, its fundamental version is constructed by the algorithm presented in [23], after Theorem 2: We consider the weights $\boldsymbol{w}^{(1)}$ from the input $(\boldsymbol{x}, v)$ to the first hidden layer $\boldsymbol{z}^{(1)}(\boldsymbol{x}, v)$ and we apply a sign switch operation (similar to (4.2)) so that all intercepts $w_{1,0}^{(1)}, \ldots, w_{q_1,0}^{(1)}$ are positive while letting the regression function $(\boldsymbol{x}, v) \mapsto \mu(\boldsymbol{x}, v)$ unchanged. Then, we apply a permutation operation to the indexes $j = 1, \ldots, q_1$ so that we arrive at (an order statistics)

$$0 < w_{1,0}^{(1)} < \ldots < w_{q_1,0}^{(1)}, \qquad (4.3)$$

for unchanged regression function $(\boldsymbol{x}, v) \mapsto \mu(\boldsymbol{x}, v)$. Note that this requires that all intercepts are non-zero and different from each other (which we assume for the moment). Then, we move iteratively through the hidden layers $k = 2, \ldots, K$, see Section 1.3. That is, we apply the above sign switch operation and the above permutation so that the regression function $(\boldsymbol{x}, v) \mapsto \mu(\boldsymbol{x}, v)$ does not change and such that for all hidden layers $k = 2, \ldots, K$ we have ordered intercepts

$$0 < w_{1,0}^{(k)} < \ldots < w_{q_k,0}^{(k)}.$$

Thus, every network parameter $\theta \in \mathbb{R}^r$ (satisfying the previous properties) has a unique representative (obtained by the algorithm above) in the fundamental domain

$$\boldsymbol{\Theta}^+ = \left\{ \theta \in \mathbb{R}^r;\ 0 < w_{1,0}^{(k)} < \ldots < w_{q_k,0}^{(k)}, \quad \text{for all } k = 1, \ldots, K \right\} \ \subset\ \mathbb{R}^r. \qquad (4.4)$$

There may still exist some redundancies in special cases, for instance, if the outgoing weights of a given neuron are equal to 0. However, as mentioned in [23], Section 2.2, these symmetries are of zero Lebesgue measure (for hyperbolic tangent activation), and will therefore be neglected for our purposes. Moreover, working on (reasonable) real observations will also imply that intercepts are different from 0 and different from each other. Thus, w.l.o.g., we may and will assume (after transformation) that the calibrated network parameter θ lies in the fundamental domain $\boldsymbol{\Theta}^+$. We then transform all network parameters obtained in Table 5 such that they all lie in the fundamental domain $\boldsymbol{\Theta}^+$. In Figure 6 (rhs) we illustrated the intercepts of the input weights

$(w^{(1)}_{j,0})_{j=1,\ldots,q_1}$ which are positive and ordered by construction, see (4.3). Note that in Figure 6 (rhs) we plot these intercepts on the log-scale. A short inspection of the figure proves that there are substantial differences between the calibrations. This exactly reflects our statement at the beginning of Section 4, namely, that we have several competing models of similar quality. These differences are also reflected in the output weights $\boldsymbol{w}^{(2)} = (w^{(2)}_j)_{j=0,\ldots,q_1}$, see Figure 6 (lhs), and in the input weights $(w^{(1)}_{j,l})_{j=1,\ldots,q_1,l=1,\ldots,q_0}$, see Figure 7.
Figures 6 and 7 now allow us to start interpreting the network calibrations. Since we work on the fundamental domain $\boldsymbol{\Theta}^+$, we know that the last hidden neuron $z^{(1)}_{q_1}(\boldsymbol{x}, w)$ belongs to the largest intercept $w^{(1)}_{q_1,0}$, see (4.3). The weights of this hidden neuron are illustrated on the first row of the plots in Figure 7. Since colors on this first row resemble each other between the different calibrations, we conclude that this last hidden neuron plays a similar role in all calibrations, taking care of `VehBrand` and `VehAge`. In particular, car brand `B12` seems to play a special role here (because the weight receives an opposite sign). The roles of other hidden neurons differ between the calibrations, especially, we can see that hidden neurons $z^{(1)}_{12}$ and $z^{(1)}_{15}$ play pronounced roles for the `Exposure` in the 'nadam' optimizer calibration, whereas in the 'adagrad' optimizer we receive generally smaller weights for `Exposure`. We could now do a more in-depth analysis of Figures 6 and 7, but we refrain from doing so here. The important takeaways are that we have several competing models of a similar quality, moreover, Figures 6 and 7 may play a crucial role in interpreting the neural network models.

4.3 Epochs, batches and computational time

In this subsection we analyze the precision and the computational time of the GDM in terms of the choice of the batch size, see line 15 of Listing 3. `epochs` indicates how many times we go through the entire learning data $\mathcal{D}$, and `batch_size` indicates the size of the subsamples considered in each GDM step. Thus, if the batch size is equal to the number of observations n we do exactly one GDM step in one epoch, if the batch size is equal to 1 then we do n GDM steps in one epoch until we have seen the entire learning data $\mathcal{D}$. Note that smaller batches are needed for big data because it is not feasible to simultaneously calculate the gradient on all data efficiently if we have many observations. Therefore, we partition the entire data at random into (mini-) batches in the application of the GDM.

optimizer	epochs	batch size	GDM steps per epoch	run time	av. time per GDM step	in-sample loss	out-of-sample loss
'rmsprop'	10	610'212	1	8.96s	0.8960s	31.82270	32.86120
'rmsprop'	10	122'043	5	8.86s	0.1772s	30.41564	31.61107
'rmsprop'	10	61'022	10	8.41s	0.0841s	30.12949	31.31887
'rmsprop'	10	12'205	50	8.69s	0.0174s	29.57422	30.77322
'rmsprop'	10	6'103	100	11.15s	0.0112s	29.51039	30.70593
'rmsprop'	10	1'221	500	20.83s	0.0042s	29.48395	30.63886
'rmsprop'	10	611	1'000	30.31s	0.0030s	29.57284	30.70585
'rmsprop'	10	123	5'000	93.84s	0.0019s	30.50588	31.35041

Table 6: batch sizes, run times, GDM steps and losses; shallow network with $q_1 = 20$, see also Figure 8 (lhs); losses are in 10^{-2}.

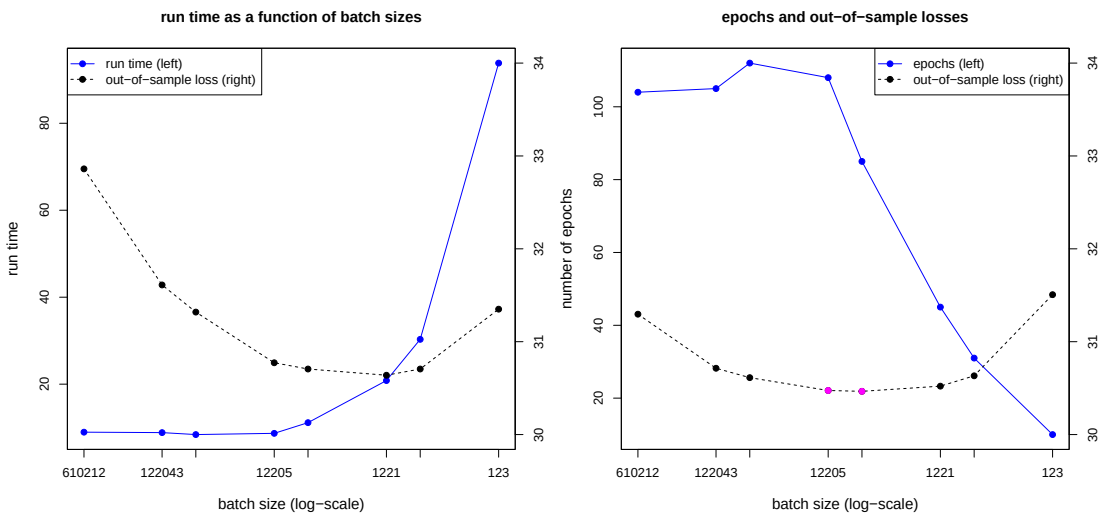

Figure 8: (lhs) illustration of run time and out-of-sample losses of Table 6; (rhs) illustration of epochs and out-of-sample losses of Table 7 for constant run time; losses are in 10^{-2}.

In Table 6 and Figure 8 (lhs) we compare the results for different batch sizes (using the optimizer 'rmsprop' and always having the same initial parameter for θ). For the maximal batch size $n = 610'212$ we can do exactly one GDM step in one epoch, for batch size 123 we can do 5'000 GDM steps in one epoch. For the maximal batch size we need to calculate the gradient on the entire data $\mathcal{D}$ which takes 0.8960s, for batch size 123 we calculate the gradient on 123 policies taking in average 0.0019s. Thus, the latter is, of course, much faster but on the other hand we need to calculate 5'000 gradients to run through the entire data (in an epoch). This results in 9.3840s per epoch, thus, 10 times longer than for the maximal batch size, see Figure 8 (lhs). That is, for the total run time we obtain a trade-off between the batch size and the number of GDM steps we need to perform in order to screen the whole data in an epoch. In our setup, the minimal run time of 8.41s (for 10 epochs) is achieved for a batch size of 61'022.
These run times should be contrasted with the quality of fit, in all set-ups of Table 6 we screen the entire data 10 times (number of epochs). The best out-of-sample performance for 10 epochs is achieved by a batch size of 1'221 which corresponds to 500 GDM steps per epoch (and a total run time of 20.83s), see also Figure 8 (lhs). Here, we have a trade-off between the number of GDM steps (smaller batch sizes) and the law of large numbers (bigger batch sizes). If the batch size is too small, then we consider too many batches that are not well-balanced, and hence too often we move into a direction that is not sufficiently relevant for the entire data. The art of calibration then is to fine-tune batch size, run time and performance.
In Table 7 and Figure 8 (rhs) we provide a similar analysis as in the previous table, but this time we try to keep the total run time constant (roughly 90s) by adjusting the number of epochs accordingly. From the table and the figure we see that the optimal batch size for our network architecture (in terms of run time versus out-of-sample loss) is roughly 10'000. Note that this is exactly the batch size used in the analysis of Table 5. For a homogeneous portfolio with $N_i \overset{\text{i.i.d.}}{\sim} \text{Poi}(\mu = 5\%)$ we obtain for batch size 10'000 confidence bounds of two standard

optimizer	epochs	batch size	GDM steps per epoch	run time	av. time per GDM step	in-sample loss	out-of-sample loss
'rmsprop'	104	610'212	1	86.27s	0.8295s	30.07556	31.29621
'rmsprop'	105	122'043	5	85.10s	0.1620s	29.49710	30.71401
'rmsprop'	112	61'022	10	87.14s	0.0778s	29.42203	30.61296
'rmsprop'	108	12'205	50	92.69s	0.0171s	29.30271	30.47451
'rmsprop'	85	6'103	100	87.27s	0.0102s	29.28959	30.46551
'rmsprop'	45	1'221	500	97.99s	0.0043s	29.38794	30.52137
'rmsprop'	31	611	1'000	98.85s	0.0031s	29.57330	30.63273
'rmsprop'	10	123	5'000	106.33s	0.0021s	30.60819	31.50641

Table 7: batch sizes, run times, GDM steps and losses; shallow network with $q_1 = 20$, see also Figure 8 (rhs); losses are in 10^{-2}.

deviations given by

$$\mu \pm 2 \cdot \sqrt{\frac{\mu}{10'000}} \;=\; 5\% \pm 0.4\%, \tag{4.5}$$

i.e. a precision of roughly 10% relative to the parameter μ to be estimated. This precision seems to be good in this situation, but of course, in general, this will depend on the complexity of the network architecture, on the heterogeneity of the underlying portfolio and on the level of μ (class imbalances in 0 will require higher batch sizes).

4.4 Stratified batches and under/over-sampling

4.4.1 Stratified batches

On line 15 of Listing 3 we specify the `batch_size` and the number of `epochs`: for the stochastic gradient descent steps we partition (at random) all observations $\mathcal{D}$ into batches of size `batch_size`. In one `epoch` we consider all batches once, and hence each case $(N_i, \boldsymbol{x}_i, v_i)$ in the observations is seen exactly once in an epoch. The partitioning of the data $\mathcal{D}$ into batches is done at random, and it may happen that several potential outliers lie in the same batch. This may imply that some steps of the 'sgd' algorithm deteriorate too much into a wrong direction (because the chosen batch is not a typical observation). This happens especially if the chosen batch size is small and the expected frequency is low (class imbalance problem), the latter providing rates of convergence of magnitude $\sqrt{\mu/\texttt{batch\_size}}$, see also (4.5). In statistics and in machine learning (especially in cross-validation) this difficulty is taken care of by choosing a stratified partition of the data $\mathcal{D}$, see also Section 2.6.4 in [27]. In our case, the partition may be chosen such that the cases $(N_i, \boldsymbol{x}_i, v_i)$ with $N_i = 0$ are equally distributed among the batches, and likewise for the observations $N_i > 0$ among the batches. This is exactly the idea behind considering a *stratified* partition of the data $\mathcal{D}$. Often this leads to better rates of convergence (because batches resemble more typical observations). In Figure 9 we compare the stratified case (red dots) to the non-stratified version (blue line) for batches of sizes 10'000 (lhs) and 1'000 (rhs), respectively. In both cases we see that the stratified version has better convergence properties. This is also illustrated in the in-sample losses provided in Table 8.[11]

[11] Note that the run times in Tables 5 and 8 differ because for the former we have used the R interface to Keras back-end version and the latter is a self-coded programme in R.

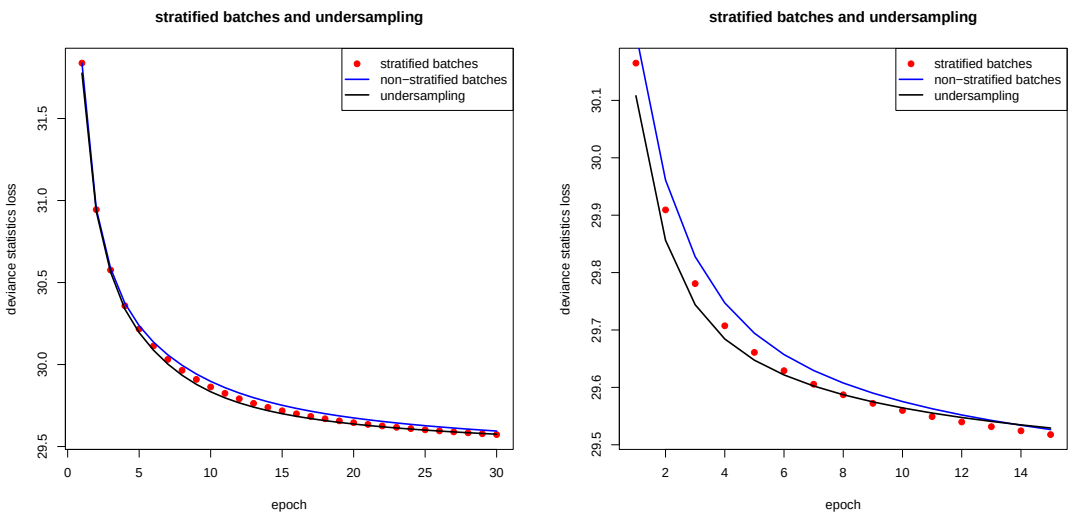

Figure 9: stratified batches and under/over-sampling: (lhs) `batch_size` $= 10'000$ with 30 epochs, and (rhs) `batch_size` $= 1'000$ with 15 epochs; for under/over-sampling we use factor 2.

batch type	epochs	batch size	GDM steps per epoch	run time	in-sample loss	out-of-sample loss
non-stratified	30	10'000	62	88.36s	29.59458	30.76872
stratified	30	10'000	62	88.49s	29.57311	30.75321
under/over-sampling	30	10'000	65	92.44s	29.57528	30.76261
non-stratified	15	1'000	611	53.43s	29.52651	30.69125
stratified	15	1'000	611	53.62s	29.51779	30.69621
under/over-sampling	15	1'000	641	55.57s	29.52924	30.71172

Table 8: momentum-based stochastic gradient descent 'sgd' method for stratified batches and with under/over-sampling; for under/over-sampling we use factor 2; shallow network with $q_1 = 20$; losses are in 10^{-2}.

4.4.2 Under/over-sampling

A second method to improve rates of convergence in low frequency problems is to consider so-called under-sampling (or over-sampling). A batch of size, say, 1'000 should roughly have 100 cases $(N_i, \boldsymbol{x}_i, v_i)$ with $N_i > 0$ and 900 cases with $N_i = 0$ for an empirical frequency of 10% (for volumes $v \equiv 1$). The idea behind under/over-sampling is to take too many cases with $N_i > 0$ into a batch, but compensating this over-weighting of claims by increasing the weight of the cases with $N_i = 0$. For instance, if we double the cases with $N_i > 0$ in a batch, then we only receive roughly 800 cases with $N_i = 0$ in that batch, and we compensate this *under-sampling* of zero-claims by attaching $2\mu(\boldsymbol{x}_i, v_i)$ to the latter cases.[12] This idea is, of course, related to importance sampling. This under/over-sampling with a factor 2 is illustrated in Figure 9 (black line) and in Table 8. In particular, in the early stage of the calibration we see a faster convergence using under/over-sampling, but for fine-tuning one should switch to the observations on the original

[12]This factor 2 has the interpretation of an offset in Poisson regression modeling.

scale. Doubling the positive cases $N_i > 0$ also implies that we have to perform more GDM steps per epoch, see Table 8.

Since the resulting differences in Table 8 are rather small, we will not further follow up stratified batches and under/over-sampling.

4.5 Initialization of gradient descent method

As described in Section 4.1, the main problem (for a given network architecture) is to find a good network parameter $\theta \in \mathbb{R}^r$. If we use the GDM, goodness of fit and convergence properties of the algorithm will depend on a good choice of an initial value for $\theta \in \mathbb{R}^r$ in the GDM algorithm. Some optimizers, like the ones in Keras, try to choose internally optimal initial values (the default initializer in Keras is "glorot_uniform", for details see Glorot-Bengio [7]).
We remind of Section 2 on feature pre-processing: in the activation functions ϕ introduced in (1.6), the change in activation mainly takes place in the neighborhood of the origin (except for ReLU). For this reason, all feature components were scaled to this domain (because the GDM is only sensitive around the origin w.r.t. the chosen activation functions). Far off the origin, the gradients will almost be zero and the GDM algorithm will not work (or convergence will be very slow). This implies that also initial weights θ of the GDM should be chosen such that the scalar products $\langle \boldsymbol{w}_j^{(k)}, \boldsymbol{z}^{(k-1)} \rangle$ are in the neighborhood of zero, see (1.5). This will ensure that we do not face the so-called *vanishing gradient problem*. Moreover, the initial network parameter should be randomized to make sure that it does not have unwanted symmetries. Symmetries in initializations may imply that the GDM gets trapped in saddle points. In deep networks with many hidden layers, one often inserts normalization layers to avoid the vanishing gradient problem. This normalization layers map the hidden neurons back to a reasonable scale around the origin. This normalization layers are part of the network architecture and will be further discussed in Section 6, below.

Remark on Poisson boosting machine. Another powerful machine learning method is the boosting machine. In Section 5 of Noll et al. [18] the boosting machine is based on Poisson regression trees. If we consider regression trees of depth 1, i.e. with 2 leaves, then the feature space $\mathcal{X}^+$ receives exactly one partition, see (5.1) in [18]. Thus, the Poisson boosting machine with regression trees of depth 1 leads to a partition of the feature space (by scalar products that are all orthogonal to the main coordinate axes). This partition and the corresponding regression parameters could be used to initialize a network parameter in the GDM in a network architecture with one hidden layer. If this network has q_1 hidden neurons, then we need q_1 iterations of the Poisson boosting machine. The resulting partition is then in line with the step function activation, see (1.5). The sigmoid activation approximates the step function activation by choosing large weights in the former, and the hyperbolic tangent activation is received as a scaled version of the sigmoid activation function. This provides another, though a bit cumbersome, method to initialize parameters in the GDM for shallow networks.

Remark on embedding/nesting. Another way of initializing a neural network is to embedding (nest) a classical actuarial model into the network. This can, for instance, be done by using a skip connection. Using this approach we can initialize the network such that the initial cali-

bration of the GDM exactly provides the classical actuarial model, and the network is boosting this classical actuarial model. This idea is described in more detail in [28].

5 Over-fitting, dropout and regularization: items (g)-(h)

Throughout this section we work with a shallow network having $q_1 = 20$ hidden neurons (and hyperbolic tangent activation).

5.1 Over-fitting and early stopping

In this section we discuss potential over-fitting of network regression functions to the learning data $\mathcal{D}$. As described in Section 4, most network architectures are over-parametrized.[13] This

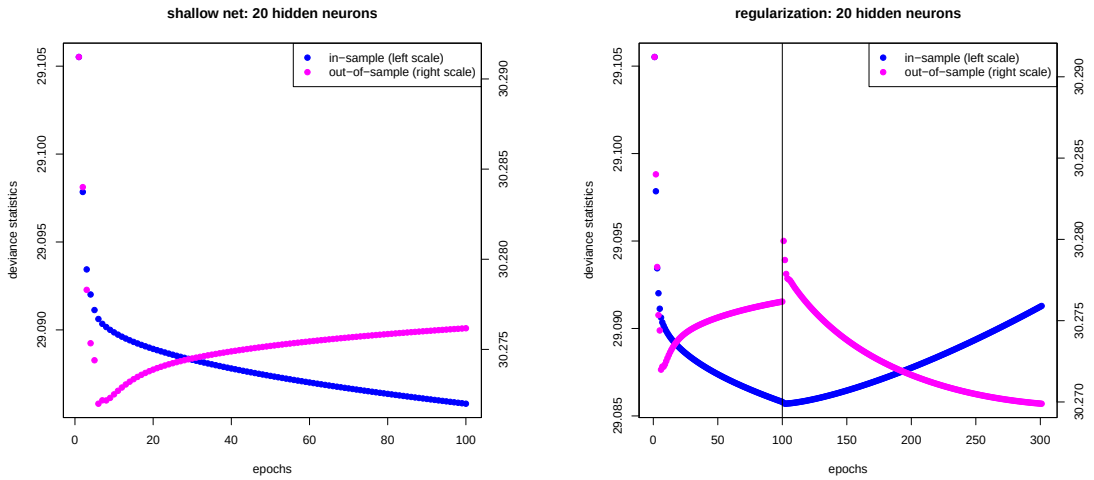

Figure 10: (lhs) over-fitting: decrease in in-sample loss $\mathcal{L}(\mathcal{D}, \widehat{\mu}(\cdot))$ (blue) and resulting out-of-sample loss $\mathcal{L}(\mathcal{T}, \widehat{\mu}(\cdot))$ (magenta); (rhs) ridge regularized regression after epoch 100.

implies that if we run the GDM for too long it will result in over-fitting to the learning data $\mathcal{D}$. This over-fitting implies that we will have a small in-sample loss but a poor out-of-sample performance (a big generalization error). In Figure 10 (lhs) we show the decrease in in-sample loss $\mathcal{L}(\mathcal{D}, \widehat{\mu}(\cdot))$ of the GDM on the learning data $\mathcal{D}$ in blue color, and the resulting out-of-sample losses $\mathcal{L}(\mathcal{T}, \widehat{\mu}(\cdot))$ for each epoch on the test data $\mathcal{T}$ are shown in magenta color. We see a typical picture of over-fitting here: the in-sample loss is monotonically decreasing, but the out-of-sample loss increases in later epochs. Thus, in view of Figure 10 (lhs), the GDM should be *early stopped* after roughly 10 epochs. This early stopping is more art than science, often it is done by partitioning the learning data $\mathcal{D}$ into a *training set* and a *validation set*. The training set is used for fitting the model and the validation set is used for out-of-sample back-testing (validation). We emphasize that we choose the validation set disjointly from the test data $\mathcal{T}$ because the early stopping rule should not have seen the test data $\mathcal{T}$, as this latter data may

[13] In Section 6.1 we discuss universality theorems which state that (noisy) observations can be approximated arbitrarily well if we allow for arbitrarily many hidden neurons (and if the noisy observations are non-conflicting).

still be used later for the choice of the optimal model (if, for instance, we need to decide between networks and boosting machines).

5.2 Regularization

In this section we study more sophisticated methods that prevent from over-fitting. A common choice in statistics is to introduce a regularization term (penalty function). This regularization term prevents regression functions from taking too wild shapes because more excessive network parameters receive a higher penalty term. Choose $p \geq 1$ and $\eta > 0$. We defined the regularized in-sample (Poisson deviance) loss by

$$\mathcal{L}(\mathcal{D}, \widehat{\mu}(\cdot); \eta) \;=\; \frac{1}{n} \sum_{i=1}^{n} 2N_i \left[\frac{\widehat{\mu}(\boldsymbol{x}_i, v_i)}{N_i} - 1 - \log \left(\frac{\widehat{\mu}(\boldsymbol{x}_i, v_i)}{N_i} \right) \right] + \eta \, \|\theta_-\|_p^p \,, \tag{5.1}$$

we also refer to (3.1), and θ_- denotes the network parameter excluding the intercept in the output layer. Note that the regression function $\widehat{\mu}(\cdot) = \widehat{\mu}_\theta(\cdot)$ depends on the network parameter θ. Thus, we penalize the in-sample loss for network parameters θ that have a bigger ℓ^p-norm for θ_-, and network parameters will be regularized towards 0. The bigger the regularization parameter η, the less the regression function will follow the (in-sample) observations, because for $\theta_- \equiv 0$ we have a homogeneous (regression) model only considering an intercept.
In statistics, one usually considers $p \in \{1, 2\}$: the ℓ^1-norm gives the so-called LASSO regression (least absolute shrinkage and selection operator regression), and the ℓ^2-norm regularization is called ridge regression, see Section 3.4 in Hastie et al. [10]. An ℓ^2-norm penalizes large values of the network parameter more than small ones, whereas the ℓ^1-norm applies the same scale for the entire range of the parameters.[14] We could also apply other regularization functions, for instance, we could use a different regularization parameter in each layer, we could exclude intercepts from regularizations, etc., but this is beyond the scope of this tutorial.
Minimizing the regularized in-sample loss (5.1) has a nice Bayesian interpretation. Observe that

$$-\frac{n}{2} \mathcal{L}(\mathcal{D}, \widehat{\mu}(\cdot); \eta) \;=\; \sum_{i=1}^{n} \left(-\widehat{\mu}(\boldsymbol{x}_i, v_i) + N_i \log \widehat{\mu}(\boldsymbol{x}_i, v_i) \right) \;-\; \frac{\eta n}{2} \, \|\theta\|_p^p \;+\; \text{const.}$$

That is, minimizing the regularized in-sample loss function provides the same result as maximizing the Bayesian posterior distribution of the network parameter on the observations $\mathcal{D}$, under the assumption that θ has a prior distribution $\pi(\theta) \propto \exp\{-\eta n \, \|\theta\|_p^p / 2\}$ (we may still have to discuss whether we want treat the intercept in the output layer separately). For ridge regression this corresponds to a Gaussian prior distribution and for LASSO regression we have a Laplace prior distribution. The regularized optimal network parameter is the Bayesian maximal a posteriori (MAP) estimator, for given observations $\mathcal{D}$.
We consider ridge regression in Figure 10 (rhs). The first 100 epochs exactly correspond to the GDM given on the lhs of the figure, and we continue the GDM in epoch 101, but adding a ridge penalty term to the loss function in the GDM, see (5.1). We choose regularization parameter $\eta = 100/\texttt{batch\_size}$. This implies that the network parameter is pushed towards 0, and that

[14] Ridge regression mainly leads to shrinkage of large parameter values, whereas LASSO regression also helps for parameter selection because certain parameters may be shrinkaged to zero for (sufficiently) large regularization parameter, i.e. LASSO regression leads to sparsity.

the resulting in-sample loss *without regularization* $\mathcal{L}(\mathcal{D}, \widehat{\mu}(\cdot)) = \mathcal{L}(\mathcal{D}, \widehat{\mu}(\cdot); \eta = 0)$ is increasing in the GDM steps (blue color), because we walk slightly into the wrong direction caused by regularization.[15] However, if properly fine-tuned the out-of-sample loss $\mathcal{L}(\mathcal{T}, \widehat{\mu}(\cdot))$ is decreasing because of reducing (or even eliminating) over-fitting. This is exactly what we can observe in Figure 10 (rhs) for epochs 101 to 300 (magenta color). The corresponding results are provided in Table 9. Observe that Figure 10 (rhs) shows that this fit may still be improved because it

regularization/dropout	in-sample loss	out-of-sample loss
un-penalized GDM version	29.08575	30.27990
ridge regression with $\eta = 100/$`batch_size`	29.09129	30.26989
dropout rate $p = 1\%$	29.15690	30.39025
dropout rate $p = 5\%$	29.41352	30.56195
dropout rate $p = 10\%$	29.58974	30.69311

Table 9: over-fitting, regularization and dropout; shallow network with $q_1 = 20$; losses are in 10^{-2}.

may not have sufficiently converged, yet. However, we should not judge this question about convergence on the basis of the test data $\mathcal{T}$, but rather on the basis of training and validation sets, we also refer to the discussion at the end of Section 5.1.
The crucial parameter to be chosen is the regularization parameter $\eta > 0$. On the one hand, it should be sufficiently big to prevent from over-fitting. On the other hand, it should not be too large, because a large regularization parameter may also induce a bias towards zero that may be too strong. Typically, optimal regularization parameters are determined by cross-validation. We refrain from doing such an analysis here because this would result in a time-consuming grid search. We come back to cross-validation in Section 7.1.

Remark. If we use LASSO regression with a large regularization parameter it will lead to sparsity because several weights are shrunk to (exactly) zero. As mentioned above, this (strong) regularization may imply a bias and, thus, it is often recommended that the sparse model is fitted again without penalization and using the penalized sparse calibration as initial value (in this second calibration). This will diminish the (potential) bias.

5.3 Dropout

Another way to prevent from over-fitting is to introduce dropout rates for individual neurons. That is, we choose a fixed so-called dropout rate which is a probability $p \in [0, 1)$. In each step of the GDM every neuron (in a specified layer) is removed (independently from the others) with probability p. On line 2 of Listing 4 a shallow network with q_1 hidden neurons is defined. Line 3 specifies the dropout layer that drops any of the q_1 neurons independently from each other with probability $p = 5\%$ in each learning step. These dropouts prevent from over-fitting and make the calibration more robust because individual neurons cannot be over-trained to a specific task, but the whole composite of neurons has the provide a good prediction, even in the case of some of them dropping out. We present the results for dropout probabilities $p \in \{1\%, 5\%, 10\%\}$ in Table 9. For each calibration we have chosen the same initial weights which have been obtained

[15] Of course, the *regularized* in-sample loss (5.1), not shown in Figure 10, decreases.

Listing 4: R script for a shallow network with dropout in Keras

```
model %>%
  layer_dense(units = q1, activation = 'tanh', input_shape = c(q0)) %>%
  layer_dropout(rate=0.05) %>%
  layer_dense(units = 1, activation = k_exp)
```

from the over-fitted model stated on the first line of Table 9, see also Figure 10 (lhs). We observe that the in-sample loss is increasing which, of course, is expected. Also the out-of-sample loss is increasing, which shows that this shallow network model is not fully competitive with the ridge regression results, but it is still in a comparable range with all previous calibrations. If the out-of-sample loss is too large, this may be a sign of a too big dropout probability p, or it may also be a sign of not having sufficiently many degrees of freedom to compensate dropouts which would speak for a more complex network architecture.

5.4 Comparison of the results of Table 9

In this section we compare the results of the different models provided in Table 9.

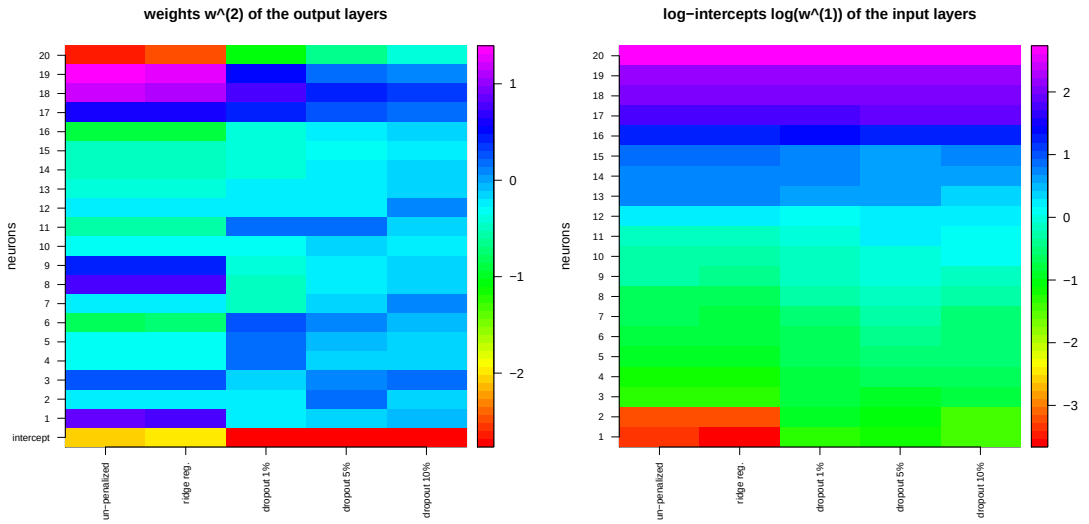

Figure 11: comparison of the calibrations obtained by the models of Table 9: (lhs) output weights $\boldsymbol{w}^{(2)} = (w_j^{(2)})_{j=0,\ldots,q_1}$; (rhs) intercepts $0 < w_{1,0}^{(1)} < \ldots < w_{q_1,0}^{(1)}$ of the inputs (on the log-scale).

In Figure 11 we provide the weights of the output layer $\boldsymbol{w}^{(2)} = (w_j^{(2)})_{j=0,\ldots,q_1}$ (rhs) as well as the intercepts $0 < w_{1,0}^{(1)} < \ldots < w_{q_1,0}^{(1)}$ (lhs, on log-scale) of the calibrations: un-penalized (which is over-fitting), with ridge regularization and with dropouts for $p \in \{1\%, 5\%, 10\%\}$. In Figure 12 we provide the corresponding input weights $(w_{j,l}^{(1)})_{j=1,\ldots,q_1;l=1,\ldots,q_0}$ (excluding intercepts). From these plots we see that the dropout calibrations substantially differ from the first two models. Comparing the first two models 'un-penalized' and 'ridge regression', we see that the network parameters look very similar, but the ridge regression version is slightly less displaced from zero due to regularization, this can especially be seen in Figure 11 (lhs). Note that the similarity is

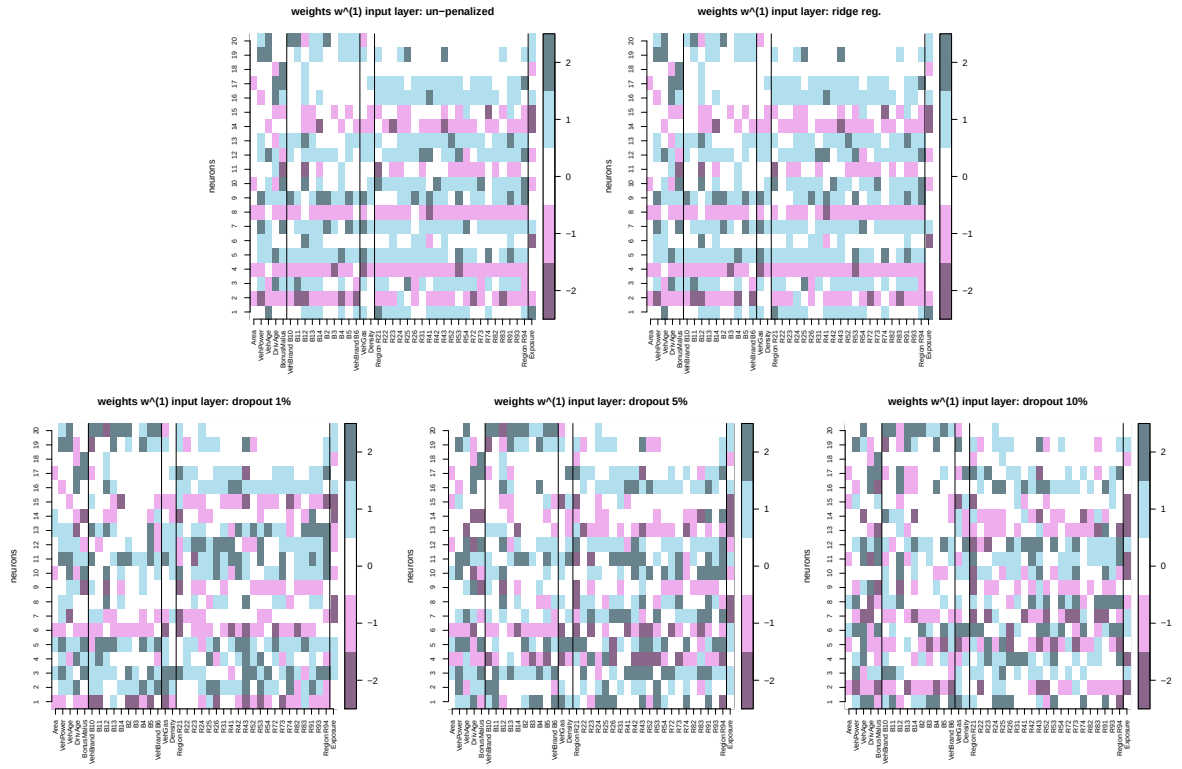

Figure 12: comparison of the calibrations obtained by the models of Table 9: input weights $(w^{(1)}_{j,l})_{j=1,\ldots,q_1;l=1,\ldots,q_0}$.

due to the fact that the over-fitted model is used as initial value for the regularized calibration, see Figure 10. From the colors in Figure 11 (lhs) we can also observe that the shrinkage is bigger for bigger parameter values, whereas smaller ones are not much affected by the ridge regularization here. If this is not appropriate one should use LASSO regression, which may result in dropping some parameters (sparsity).

The dropout versions look rather different, displacements in Figure 11 (lhs) are less extreme (and more scattered), and the input weights in Figure 12 look more scattered too compared to the ridge regularized version. This is explained by the property that a neuron may need to be able to take over the role of another one, in case the other neuron drops out, i.e. we have less over-training of individual neurons to particular tasks.

In Figure 13 we analyze the activated neurons $\boldsymbol{z}^{(1)}(\boldsymbol{x}_i) \in [-1,1]^{q_1}$, $i = 1,\ldots,n$, in the hidden layer received from the different calibrations given in Table 9. The first plot on the top row provides the singular values of a principal component analysis (PCA). We therefore construct the design matrix $Z = (\boldsymbol{z}^{(1)}(\boldsymbol{x}_1),\ldots,\boldsymbol{z}^{(1)}(\boldsymbol{x}_n))' \in \mathbb{R}^{n\times q_1}$. A singular value decomposition (SVD) provides

$$Z = U\Lambda V,$$

with orthogonal matrix $U \in \mathbb{R}^{n\times q_1}$, orthogonal matrix $V \in \mathbb{R}^{q_1\times q_1}$ and diagonal matrix $\Lambda = \text{diag}(\lambda_1,\ldots,\lambda_{q_1})$ having singular values $\lambda_1 \geq \ldots \geq \lambda_{q_1} \geq 0$, see Section 14.5 in Hastie et al. [10] for the SVD. Typically, small singular values indicate that the dimensionality of the problem can be reduced because there are some directions which do not sufficiently contribute to the

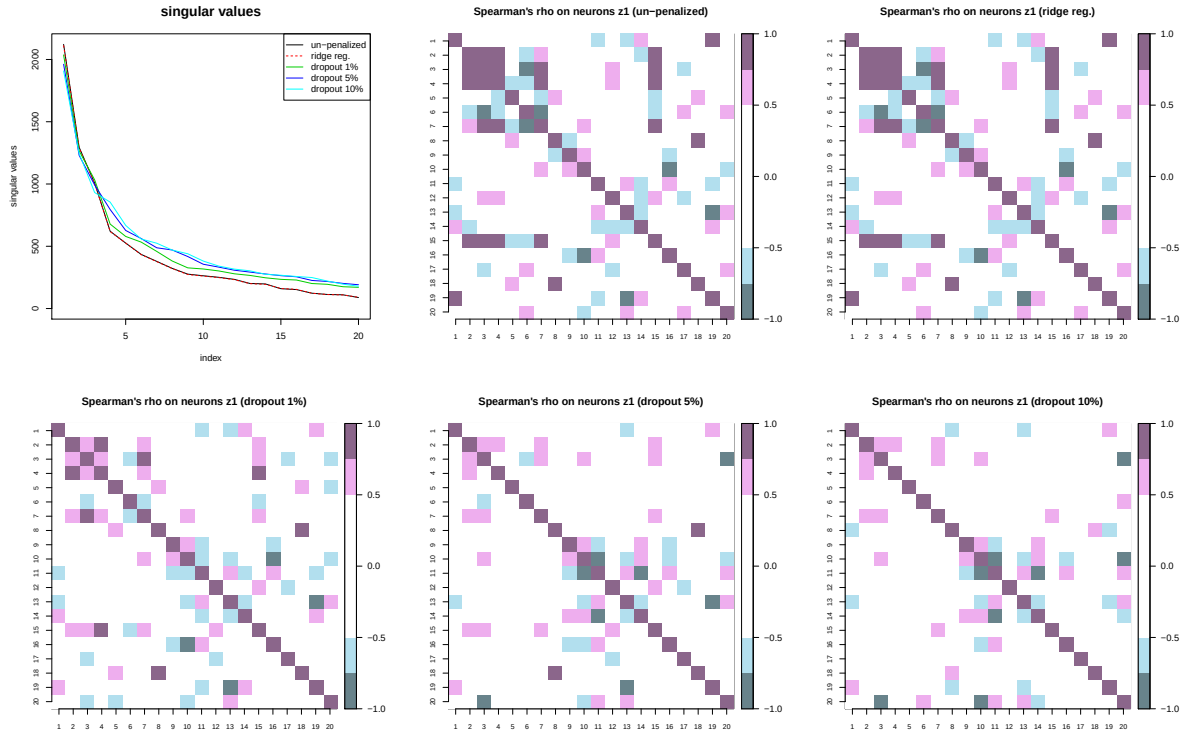

Figure 13: comparison of the calibrations obtained by the models of Table 9: (1st row, lhs) singular values $\lambda_1 \geq \ldots \geq \lambda_{q_1} \geq 0$ of a PCA of the activated neurons $\boldsymbol{z}^{(1)}(\boldsymbol{x}_i)$, $i = 1, \ldots, n$, for the different calibrations; other plots provide Spearman's rho correlations of the activated neurons $\boldsymbol{z}^{(1)}(\boldsymbol{x}_i)$, $i = 1, \ldots, n$.

explanation of the design matrix Z (on a linear scale). From Figure 13 we see, as expected, that the dropout versions provide bigger singular values (which says that more model complexity is needed under dropout). Moreover, the singular values do not immediately indicate that we should reduce the model complexity.

The other plots in Figure 13 show Spearman's rho correlations of the activated neurons $\boldsymbol{z}^{(1)}(\boldsymbol{x}_i)$, $i = 1, \ldots, n$, for the different calibrations provided in Table 9. On average dropout calibrations provide lower correlations in our example compared to ridge regularization. The reason is that in the ridge regression certain neurons play a very specific role, for instance, sorting out a particular `VehBrand` label for interacting feature components. Dropout prevents from extreme behavior. Note that even if these activated neurons show high correlations they are needed: Figure 14 shows the highly correlated neurons $z_2^{(1)}(\boldsymbol{x}_i), \ldots, z_4^{(1)}(\boldsymbol{x}_i)$ of the regularized version for the first 10'000 observations. Though we have high correlations, there are still many neurons that are far from being perfectly correlated. This exactly explains that these neurons have a special task of splitting the feature space according to some splitting criterion, i.e. separating certain labels which play a particular role in interacting feature components.

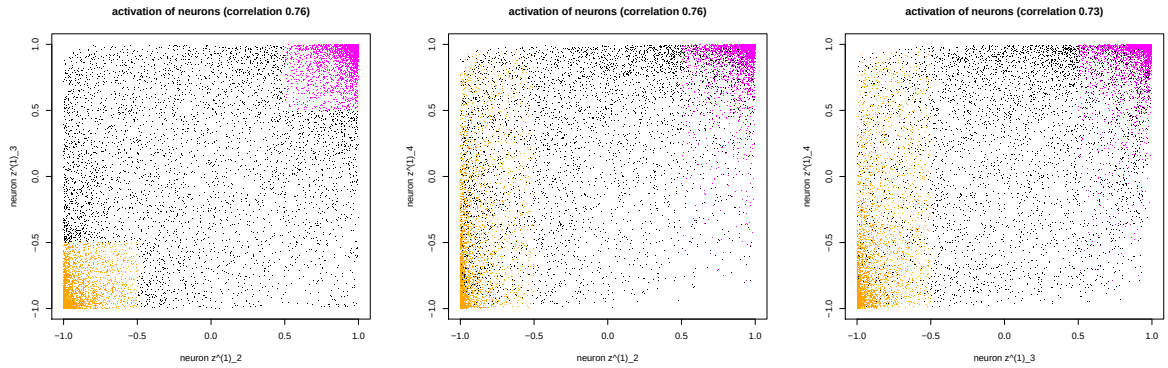

Figure 14: neurons $z_j^{(1)}(\boldsymbol{x}_i)$ for $j = 2, \ldots, 4$ and $i = 1, \ldots, 10'000$ of the regularized network calibration: orange color shows neurons with $\max\{z_2^{(1)}(\boldsymbol{x}_i), z_3^{(1)}(\boldsymbol{x}_i)\} \leq -0.5$, magenta color shows neurons with $\min\{z_2^{(1)}(\boldsymbol{x}_i), z_3^{(1)}(\boldsymbol{x}_i)\} \geq 0.5$.

6 Choice of the network architecture: items (c)-(e)

In the previous sections we have been working with a shallow network having one hidden layer with $q_1 = 20$ hidden neurons. We have treated this network architecture as indefeasible but, of course, we also have to question this choice. This is going to be done in the present chapter.

6.1 Mathematical results: universality theorems and complexity

We start to analyze the question of the network architecture from a theoretical point of view. In Section 1.3 we have introduced a network architecture that may consist of $K \in \mathbb{N}$ hidden layers each having $q_k \in \mathbb{N}$ hidden neurons, for $k = 1, \ldots, K$. An example with $K = 2$ and $(q_1, q_2) = (10, 7)$ is illustrated in Figure 3 (rhs).

A main theoretical question is: what is the minimal required network complexity so that we are able to model a fairly general class of regression functions? This question is related to Hilbert's 13th problem, and it was Kolmogorov [13] and his student Arnold [1] who gave a first answer to this kind of question.[16] These early results, however, have not been restricted to a single type of activation function. It was Cybenko [5] and Hornik et al. [11] who proved a *universality theorem* for the sigmoid activation function, saying that *shallow* networks are dense in the set of compactly supported continuous functions (either in supremum or in L^p-norm, $p \geq 1$). Thus, shallow networks with arbitrarily many hidden neurons are sufficient for approximating any compactly supported continuous function arbitrarily well.

Similar universality theorems (for non-polynomial activation functions) have been proved by Leshno et al. [14], Park–Sandberg [19, 20], Petrushev [21] and Isenbeck–Rüschendorf [12]. There are two types of proofs for these universality theorems, on the one hand there are algebraic methods of Stone-Weierstrass type and on the other hand Wiener-Tauberian denseness type proofs. A second stream of literature are Barron [2], Yukich et al. [29], Makavoz [15] and Döhler–Rüschendorf [6], these authors consider approximation results. There is more literature on applications to functional estimations using complexity regularization.

[16] see https://en.wikipedia.org/wiki/Kolmogorov-Arnold_representation_theorem

In view of this literature it seems to be sufficient to work with shallow networks. Why may we want to consider more complex network architectures? We start by discussing networks with one hidden layer. Assume for the moment that ϕ is the step function activation and $\mathcal{X}^+ = \mathbb{R}^{q_0}$. In this case we receive neurons in the single hidden layer given by

$$z_j^{(1)}((\boldsymbol{x}, v)) = \phi\langle \boldsymbol{w}_j^{(1)}, (\boldsymbol{x}, v)\rangle = \mathbb{1}_{\left\{\sum_{l=1}^{q_0-1} w_{j,l}^{(1)} x_l + w_{j,q_0}^{(1)} v \geq -w_{j,0}^{(1)}\right\}} \in \{0,1\}, \qquad \text{for } j = 1, \ldots, q_1.$$

Thus, neuron $z_j^{(1)}$ separates the feature space $\mathcal{X}^+$ with a hyperplane

$$H_j^{(1)} = \left\{ \boldsymbol{z} \in \mathbb{R}^{q_0};\ \sum_{l=1}^{q_0} w_{j,l}^{(1)} z_l = -w_{j,0}^{(1)} \right\}, \qquad \text{for } j = 1, \ldots, q_1.$$

This implies that a shallow network with q_1 hidden neurons provides a partition of the feature space $\mathcal{X}^+ = \mathbb{R}^{q_0}$ given by the hyperplanes $H_1^{(1)}, \ldots, H_{q_1}^{(1)}$ (for the step function activation). Zaslavsky [30] has proved that this partition can have a maximal complexity of

$$\#(q_1, q_0) = \sum_{j=0}^{\min\{q_0, q_1\}} \binom{q_1}{j} \quad \text{disjoint sets.} \tag{6.1}$$

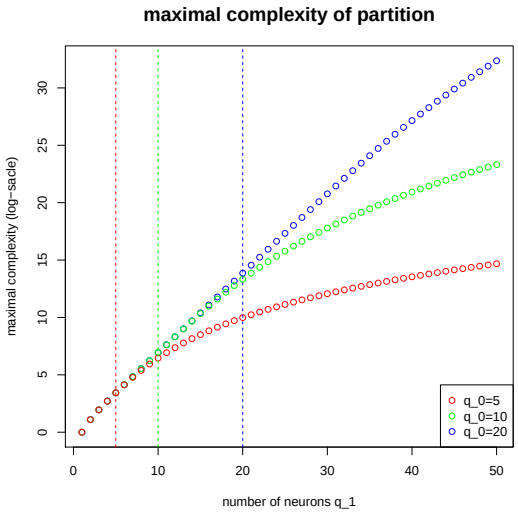

Figure 15: maximal complexity $\#(q_1, q_0)$ of partition (on log-scale) for $q_0 = 5, 10, 20$ and $q_1 = 1, \ldots, 50$ neurons.

In Figure 15 we plot the maximal complexity $\#(q_1, q_0)$ which can be achieved by a shallow network with $q_1 = 1, \ldots, 50$ hidden neurons and having feature space $\mathcal{X}^+ = \mathbb{R}^{q_0}$ of dimensions $q_0 = 5, 10, 20$ (red, green blue). Note that the graph is on the log-scale. We observe exponential growth for $q_1 \leq q_0$ and a slow down to a polynomial growth after q_0 (illustrated by the vertical lines in Figure 15). If we have a feature space of dimension $q_0 = 10$ and if we choose a shallow network with $q_1 = 20$ hidden neurons (and step function activation) we obtain a maximal complexity of $\#(q_1 = 20, q_0 = 10) = 616'665$. Thus, we obtain a fairly complex (multivariate)

step function to approximate the unknown regression function $\lambda : \mathcal{X} \to \mathbb{R}_+$. At the first sight, this seems sufficient for the data of Listing 1.[17]
For deep networks with more than one hidden layer, i.e. $K \geq 2$, the situation is more complicated. Montúfar et al. [16] provide a lower bound for the complexity that can be obtained by such a deep network. Assume that we have the ReLU activation function and that $q_k \geq q_0$ for all $k = 1, \ldots, K$. Theorem 4 in Montúfar et al. [16] states that the maximal complexity is bounded below by

$$\#(q_K, \ldots, q_0) \;\geq\; \left(\prod_{k=1}^{K-1} \left\lfloor \frac{q_k}{q_0} \right\rfloor^{q_0} \right) \; \sum_{j=0}^{q_0} \binom{q_K}{j} \qquad \text{(disjoint) linear regions.} \tag{6.2}$$

Example 6.1 (Shallow vs. deep networks: complexity)

We provide an example for the resulting complexity: choose $q_0 = 2$ and $q_k = 4$ for $k \geq 1$. From (6.2) we obtain

$$\#(q_K, \ldots, q_0) \;\geq\; 4^{K-1} \left(\binom{4}{0} + \binom{4}{1} + \binom{4}{2} \right) = \frac{11}{4}\, 4^K. \tag{6.3}$$

If we consider a shallow network with the same number of hidden neurons we obtain (6.1)

$$\#(Kq_1, q_0) = \binom{Kq_1}{0} + \binom{Kq_1}{1} + \binom{Kq_1}{2} = 8K^2 + 2K + 1. \tag{6.4}$$

From this we see that the complexity of the deep network grows faster (exponentially) than the complexity of a shallow network (polynomially) with the same number of hidden neurons, see (6.3) and (6.4). The deep network has $r = 20K - 3$ parameters and the shallow network $r = 16K + 1$ parameters, thus, the number of parameters grows linearly in both cases but at different speeds. This finishes the example. ∎

The previous example shows that the complexity of the resulting regression function may grow faster for deep networks than for shallow networks if the number of hidden neurons exceeds the dimension q_0 of the feature space. For this reason, it is often recommended to work with deep networks for more complex regression functions. Shallow networks typically have a lower approximation capacity than deep ones, and the size of the hidden layer in the shallow network often grows very fast in order to get good approximations. The following example shows, that a rather simple regression functions can be reconstructed easily by a deep network but not so easily by a shallow one.

Example 6.2 (Shallow vs. deep networks: partitions)

We present a simple example that can easily be reconstructed by a deep network, but not by a shallow one. Choose a two-dimensional feature space $\mathcal{X} = [0, 1]^2$ and define the regression function $\lambda : \mathcal{X} \to \mathbb{R}_+$ by

$$\boldsymbol{x} \;\mapsto\; \lambda(\boldsymbol{x}) = 1 + \mathbb{1}_{\{x_2 \geq 1/2\}} + \mathbb{1}_{\{x_1 \geq 1/2,\; x_2 \geq 1/2\}} \;\in\; \{1, 2, 3\}. \tag{6.5}$$

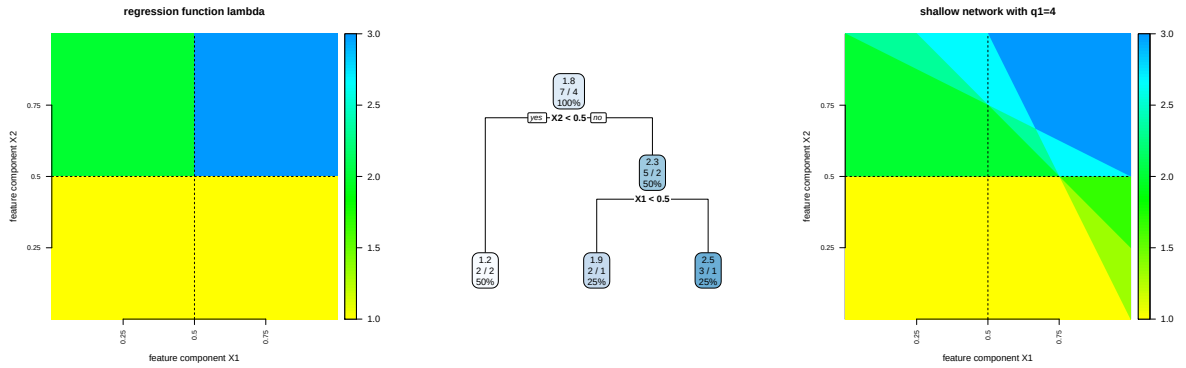

Figure 16: (lhs) regression function (6.5); (middle) regression tree with 2 splits and 3 leaves; (rhs) shallow network approximation with $q_1 = 4$ hidden neurons.

This regression function is illustrated in Figure 16 (lhs).
This regression function (6.5) can easily be modeled by a regression tree[18] with 2 splits and 3 leaves, see Figure 16 (middle). Namely, we need a first split for $\mathbb{1}_{\{x_2 \geq 1/2\}}$ and a second one for $\mathbb{1}_{\{x_1 \geq 1/2\}}$ *on the set* $\{x_2 \geq 1/2\}$. That is, we obtain the required interaction of the feature components x_1 and x_2 by this regression tree.
If we choose a shallow network with $q_1 = 4$ hidden neurons and step function activation we can construct the regression function

$$\boldsymbol{x} \;\mapsto\; \widehat{\lambda}(\boldsymbol{x}) = w_0^{(2)} + \sum_{j=1}^{4} w_j^{(2)} \mathbb{1}_{\{\langle \boldsymbol{w}_j^{(1)}, \boldsymbol{x} \rangle \geq 0\}}.$$

Figure 16 (rhs) provides such a regression function $\widehat{\lambda}$ for $w_0^{(2)} = w_1^{(2)} = 1$, $w_2^{(2)} = w_3^{(2)} = w_4^{(2)} = 1/3$, $\langle \boldsymbol{w}_1^{(1)}, \boldsymbol{x} \rangle = x_2 - 1/2$ (horizontal split), $\langle \boldsymbol{w}_2^{(1)}, \boldsymbol{x} \rangle = x_1 + x_2 - 5/4$ (split with slope -1), $\langle \boldsymbol{w}_3^{(1)}, \boldsymbol{x} \rangle = 2x_1 + x_2 - 2$ (split with slope -2), and $\langle \boldsymbol{w}_4^{(1)}, \boldsymbol{x} \rangle = x_1 + 2x_2 - 2$ (split with slope $-1/2$). We observe that this shallow network cannot perfectly replicate the true regression function (6.5), and more hidden neurons are needed for getting a better approximation (which is possible due to the universality theorems).
We rewrite (6.5) choosing step function activations: for $\boldsymbol{x} \in \mathcal{X}$ we define the first hidden layer

$$\boldsymbol{z}^{(1)}(\boldsymbol{x}) = \left(z_1^{(1)}(\boldsymbol{x}), z_2^{(1)}(\boldsymbol{x}) \right)' = \left(\mathbb{1}_{\{x_1 \geq 1/2\}}, \mathbb{1}_{\{x_2 \geq 1/2\}} \right)' \in \{0,1\}^2.$$

This provides

$$\begin{aligned} \lambda(\boldsymbol{x}) &= 1 + \mathbb{1}_{\{x_2 \geq 1/2\}} + \mathbb{1}_{\{x_1 \geq 1/2\}} \mathbb{1}_{\{x_2 \geq 1/2\}} \;=\; 1 + z_2^{(1)}(\boldsymbol{x}) + z_1^{(1)}(\boldsymbol{x})\; z_2^{(1)}(\boldsymbol{x}) \\ &= 1 + \mathbb{1}_{\{z_2^{(1)}(\boldsymbol{x}) \geq 1/2\}} + \mathbb{1}_{\{z_1^{(1)}(\boldsymbol{x}) + z_2^{(1)}(\boldsymbol{x}) \geq 3/2\}} \;=\; 1 + z_1^{(2)}\left(\boldsymbol{z}^{(1)}(\boldsymbol{x}) \right) + z_2^{(2)}\left(\boldsymbol{z}^{(1)}(\boldsymbol{x}) \right), \end{aligned}$$

with neurons in the second hidden layer given by

$$\boldsymbol{z}^{(2)}(\boldsymbol{z}) = \left(z_1^{(2)}(\boldsymbol{z}), z_2^{(2)}(\boldsymbol{z}) \right)' = \left(\mathbb{1}_{\{z_2 \geq 1/2\}}, \mathbb{1}_{\{z_1 + z_2 \geq 3/2\}} \right)', \qquad \text{for } \boldsymbol{z} \in [0,1]^2.$$

[17] Note that our situation is fairly more sophisticated because of categorical feature components.
[18] For an introduction to regression trees we refer to Section 4 of Noll et al. [18].

Thus, we obtain deep network regression function

$$\boldsymbol{x} \;\mapsto\; \lambda(\boldsymbol{x}) = \left\langle \boldsymbol{w}^{(3)}, (\boldsymbol{z}^{(2)} \circ \boldsymbol{z}^{(1)})(\boldsymbol{x}) \right\rangle,$$

with output weights $\boldsymbol{w}^{(3)} = (1,1,1)' \in \mathbb{R}^3$. We conclude that a deep network with $K = 2$ hidden layers and $q_1 = q_2 = 2$ hidden neurons, i.e. with totally 4 hidden neurons, can perfectly replicate example (6.5). ■

Conclusion.
The findings from the previous example are that if we have different types of interactions involving different numbers of feature components, we often receive better modeling properties from deep networks. This may have a positive effect on the rate of convergence in network calibrations. Unfortunately, there do not exist general results about a good choice of the depth of a network. Most of the results need additional structure and assumptions, for instance, Shaham et al. [24] prove that under certain assumptions on the feature space a sparsely connected neural network of depth 4 is sufficient to appropriately approximate nice regression functions.

6.2 Shallow vs. deep networks

In this section we study different network architectures. In particular, we compare shallow networks with $q_1 = 10, 20, 30, 40, 50$ hidden neurons to deep networks (with multiple hidden layers). In all models we choose the same batch size of 10'000 and we use the same optimizer

architecture	parameters	epochs	run time	in-sample loss	out-of-sample loss
shallow network $q_1 = 10$	411	300	231.05s	29.72569	30.81615
shallow network $q_1 = 20$	821	300	272.64s	29.51171	30.65739
shallow network $q_1 = 30$	1'231	300	324.64s	29.44280	30.68640
shallow network $q_1 = 40$	1'641	300	367.33s	29.48014	30.70085
shallow network $q_1 = 50$	2'051	300	428.07s	29.72033	30.92324
deep net $(q_1, q_2) = (10, 10)$	521	300	296.00s	29.52742	30.59831
deep net $(q_1, q_2) = (10, 20)$	611	300	335.10s	29.42255	30.46509
deep net $(q_1, q_2) = (20, 10)$	1'021	300	349.38s	29.33112	30.43660
deep net $(q_1, q_2) = (20, 20)$	1'241	300	386.25s	29.32964	30.52626
$(q_1, q_2, q_3) = (10, 10, 5)$	571	300	349.87s	29.48246	30.58802
$(q_1, q_2, q_3) = (10, 20, 5)$	731	300	390.54s	29.70383	30.90749
$(q_1, q_2, q_3) = (20, 10, 5)$	1'071	300	395.83s	29.32386	30.55984
$(q_1, q_2, q_3) = (20, 20, 5)$	1'331	300	455.80s	29.13141	30.38834
$(q_1, q_2, q_3) = (10, 10, 10)$	631	300	360.36s	29.40919	30.42307
$(q_1, q_2, q_3) = (10, 20, 10)$	841	300	402.76s	29.61282	30.59637
$(q_1, q_2, q_3) = (20, 10, 10)$	1'131	300	414.56s	29.22234	30.45742
$(q_1, q_2, q_3) = (20, 20, 10)$	1'441	300	469.89s	30.00502	31.03757
$(q_1, q_2, q_3, q_4) = (20, 15, 10, 5)$	1'336	300	481.24s	29.16868	30.45878
$(q_1, q_2, q_3, q_4) = (20, 20, 10, 5)$	1'491	300	501.53s	29.18276	30.48016

Table 10: comparison of different network architectures; losses are in 10^{-2}.

'nadam'. Since all architectures have different network parameters (of different dimensions r) we cannot use identical initial configurations for the network parameters. Therefore, we just use

the standard initialization provided in Keras, this is different from Table 5 (and therefore also the results differ). The results are given in Table 10, and we are going to analyze and discuss them in the next two subsections.

6.2.1 Shallow networks

We start by analyzing the shallow networks of Table 10. We observe an almost linear increase of run time in the number of hidden neurons. For 300 epochs and a batch size of 10'000, the network with $q_1 = 20$ seems to have the best out-of-sample behavior. But, of course, this heavily depends on the choice of the initial network parameter, for instance, we obtain better results in Table 5 with less run time.

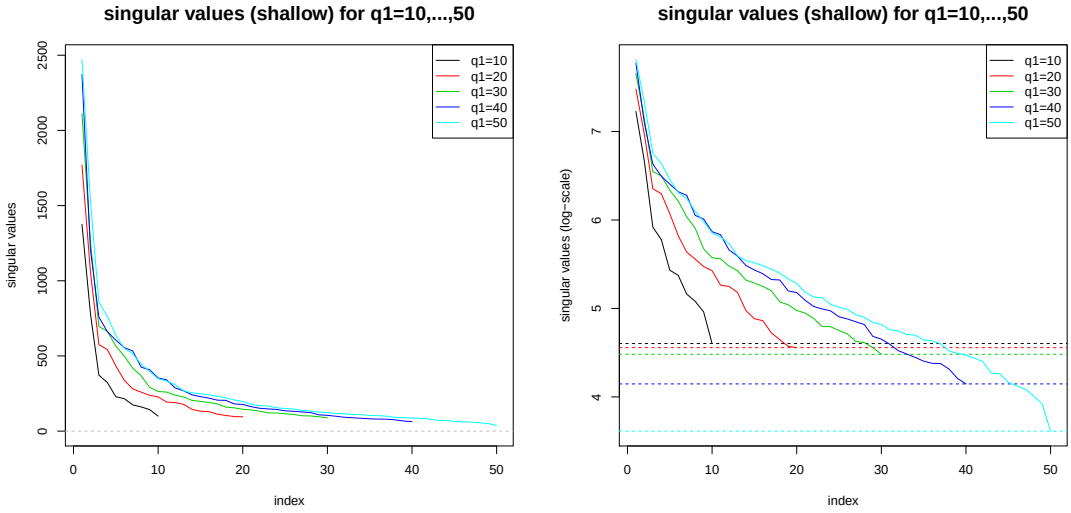

Figure 17: singular values $\lambda_1^{(q_1)} \geq \ldots \geq \lambda_{q_1}^{(q_1)} \geq 0$ of the neuron activations $\boldsymbol{z}^{(1)}(\boldsymbol{x}_i)$, $i = 1, \ldots, n$, of a shallow network with $q_1 = 10, \ldots, 50$ hidden neurons: (lhs) original scale, (rhs) log-scale.

In Figure 17 we provide the singular values $\lambda_1^{(q_1)} \geq \ldots \geq \lambda_{q_1}^{(q_1)} \geq 0$ of the PCAs of the neuron activations $(\boldsymbol{z}^{(1)}(\boldsymbol{x}_1), \ldots, \boldsymbol{z}^{(1)}(\boldsymbol{x}_n))' \in \mathbb{R}^{n \times q_1}$ of these shallow networks for $q_1 = 10, \ldots, 50$. The first observation is that the smallest singular values $\lambda_{q_1}^{(q_1)}$ are decreasing in the number of hidden neurons q_1, see Figure 17 (rhs) where the horizontal dotted lines illustrate the smallest singular values. We interpret this as follows: the more hidden neurons, the more fine-tuning of in-sample losses is done, and the less relevant additional new directions in the PCA are. The other observation is that we have $\lambda_j^{(10)} \leq \lambda_j^{(20)} \leq \lambda_j^{(30)} \leq \lambda_j^{(40)} \leq \lambda_j^{(50)}$ for many fixed j's (where defined). This may be interpreted as follows: the more hidden neurons, the more individual tasks they perform, and the more (orthogonal) heterogeneity in the hidden neurons is induced. In Table 6 of Noll et al. [18] we have concluded (based on cross-validation) that the optimal number of leaves in a regression tree for this claims frequency modeling problem is 33. On the one hand, shallow networks are more flexible,[19] and on the other hand they have some deficiencies, as shown in Example 6.2. Therefore, it is sometimes a good choice to take slightly

[19] Regression trees restrict to *standardized* binary splits which are rectangular in nature, shallow networks also allow for other angles.

less hidden neurons than the optimal number of leaves. In our case this would mean that a good choice is $q_1 \leq 33$. Observe that for such choices the smallest singular values $\lambda_{q_1}^{(q_1)}$, $q_1 \leq 30$, in Figure 17 are almost identical.
As for a lower bound for q_1 we need to study the minimal required complexity of the regression problem considered. Note that if we choose one hidden neuron only in a shallow network ($q_1 = 1$), then the hidden layer compresses all information to one dimension, i.e.,

$$(\boldsymbol{x}, v) \in \mathcal{X} \;\mapsto\; \boldsymbol{z}^{(1)}(\boldsymbol{x}, v) = \phi\langle \boldsymbol{w}_1^{(1)}, (\boldsymbol{x}, v)\rangle \in \mathbb{R}.$$

Any two policies i and i' with feature differences $(\boldsymbol{x}_i - \boldsymbol{x}_{i'}, v_i - v_{i'})$ being orthogonal to $\boldsymbol{w}_1^{(1)}$ have the same neuron activations $\boldsymbol{z}^{(1)}(\boldsymbol{x}_i, v_i) = \boldsymbol{z}^{(1)}(\boldsymbol{x}_{i'}, v_{i'})$. Thus, on hyperplanes orthogonal to $\boldsymbol{w}_1^{(1)}$ we receive identical neuron activations. This shows that if the number q_1 is too small, then too much information gets lost in the (first) hidden layer. Often a good choice is $q_1 > q_0$,[20] which is also a crucial assumption in complexity consideration (6.2). Table 10 indicates that $q_1 = 10$ is too small (and too much information gets lost).

Conclusion.
We choose dimension $q_1 = 20$ or 30 for our shallow network modeling approach. We remind of the (important) Example 6.2 which shows that shallow networks may have some deficiencies in modeling interactions in feature components. For this reason we turn our discussion to deep networks next.

6.2.2 Deep networks

If we consider deep networks with $K = 2$ hidden layers and $q_1, q_2 \in \{10, 20\}$, we obtain results that are better than the shallow network ones. They are better in the sense that we obtain smaller losses if we use the entire data on exactly 300 epochs. However, deep networks need more run time because more gradients have to be evaluated. Here, the best deep network with $K = 2$ hidden layers is $(q_1, q_2) = (20, 10)$: firstly, we separate all effects in the first hidden layer with $q_1 > q_0$ neurons (see also previous subsection), and secondly, we compress this information to $q_2 < q_1$ neurons allowing for interactions. This compression argument is often used in deep networks, but it does not entirely hold: if we have many complex interactions, we may need $q_2 > q_1$ neurons for good (better) convergence properties in calibration (the argument goes along the same lines as in Example 6.2).
Considering $K = 3$ hidden layers, there is still some improvement. The optimal strategy seems to firstly separate the effects by choosing $q_1 > q_0$ neurons, to secondly model the interactions by $q_2 \approx q_1$ neurons and to thirdly compress the information by choosing $q_3 < q_2$ neurons. This provides the smallest in-sample and out-of-sample losses. We also note that the models with $q_1 \leq q_0$ neurons and $q_2 \geq q_1$ neurons have a worse performance, there seems to be too much information getting lost if the first hidden layer is too small.
Finally, we consider $K = 4$ hidden layers with the same philosophy. After separating the effects by choosing $q_1 > q_0$ neurons we seem to receive fairly good results by using the compression argument, i.e. $q_k \leq q_{k-1}$. However, the results are not as good as for $K = 2, 3$ hidden layers.

[20] For this advice we account dimension 1 for a categorical feature component, and not the resulting dimension of the corresponding dummy coding.

This may indicate that 300 epochs are not sufficient for more than 3 hidden layers, because convergence may be slower due to increasingly more first order approximations.

Conclusion.
We conclude that in theory shallow networks are sufficient. However, deep networks allow for more/different modeling flexibility with less neurons, and, depending on the underlying problem, this results in better convergence behavior, see Example 6.2. Another idea that we are going to discuss below is to "grow" a network more systematically.

6.3 Activation functions

Next we discuss the choice of the activation functions. Some activation functions have been introduced in (1.6). Our first remark is that the sigmoid activation function $\varsigma(x) = (1 + e^{-x})^{-1}$ can easily be obtained from the hyperbolic tangent activation function, using the transformation $\tanh(x/2) = 2\varsigma(x) - 1$. For this reason, it is sufficient in theory to work with one version of the two activation functions. In practice, however, the particular choice of the activation function may matter: calibration of deep networks may be slightly simpler if we choose the hyperbolic tangent activation because this will guarantee that all hidden neurons $z_j^{(k)} \in [-1, 1]$, which is the domain of the main activation of the neurons in the next (hidden) layer $k + 1$.

activation functions	epochs	run time	in-sample loss	out-of-sample loss
hyperbolic tangent	300	324.64s	29.44280	30.68640
ReLU	300	314.11s	29.27721	30.49067
hard sigmoid	300	418.72s	30.31087	31.32395

Table 11: shallow network with $q_1 = 30$ hidden neurons, batch size 10'000 and 'nadam' optimizer; losses are in 10^{-2}.

The step function activation $\phi(x) = \mathbb{1}_{\{x \geq 0\}}$ is useful for theoretical considerations. It has, for instance, been used in Section 6.1. From a practical point of view it is less useful because it is not differentiable and difficult to calibrate. Moreover, the discontinuity also implies that neighboring feature components may have rather different regression function values: if the step function jumps, say, between driver's ages 48 and 49, then these two driver's ages may have a rather different insurance premium. For these reasons we do not pursue with the step function activation here. Note, however, that the step function activation can be received as limit of the sigmoid activation function, i.e.

$$\lim_{b \to \infty} (1 + e^{-bx})^{-1} = \mathbb{1}_{\{x \geq 0\}}, \qquad \text{for all } x \in \mathbb{R}.$$

Another activation that is sometimes used is the so-called hard sigmoid activation function. It is defined by

$$\phi(x) \;=\; \frac{(x \wedge 2.5) \vee (-2.5)}{5} + \frac{1}{2} \;=\; \begin{cases} 0 & \text{if } x \leq -2.5, \\ x/5 + 0.5 & \text{if } -2.5 < x < 2.5, \\ 1 & \text{if } x \geq 2.5. \end{cases}$$

This is a linear interpolation between -2.5 and 2.5 (as an approximation to the sigmoid activation function). Finally, we consider the ReLU activation function $\phi(x) = x\mathbb{1}_{\{x \geq 0\}}$ which has

received considerable attention because of its good properties.
In Table 11 we present the results for the three considered activation functions. For all three cases we consider a shallow network with $q_1 = 30$ hidden neurons, batch size 10'000, 300 epochs and the 'nadam' optimizer. The ReLU activation function provides slightly better (convergence) results here than the hyperbolic tangent activation function. The hard sigmoid activation function does not seem to be competitive and will therefore not further be considered.
We remark that the ReLU activation function often leads to sparsity in deep neural networks because some neurons remain unactivated for the entire input. Such an effect may be wanted because it reduces the complexity of the regression model, but it can also be an undesired side effect because it may increase the difficulty in model calibration because of more vanishing gradients.

6.4 Normalization layers

Calibration of deep networks often suffers the so-called vanishing gradient problem. This is caused by the fact that the features may lay in regions where the gradient becomes almost zero because the activation function is flat. This happens, for instance, for $\phi(x) = \tanh(x)$ if x is very large (and it may be an even more acute problem for the ReLU activation function). This problem becomes even more pronounced in deep networks because, using the chain rule for differentiation, one considers the product of potentially small values. To circumvent these difficulties, one often adds so-called normalization layers between the hidden layers. These normalization layers map the neurons $(z_j^{(k)}(\cdot))_{j=1,\ldots,q_k}$ back to be centered around the origin having unit variance. In Listing 5 we provide the corresponding code for a deep network with 4

Listing 5: R script for deep network with normalization layers in Keras

```
model %>%
  layer_dense(units = 20, activation = 'tanh', input_shape = c(q0)) %>%
  layer_batch_normalization() %>%
  layer_dense(units = 15, activation = 'tanh') %>%
  layer_batch_normalization() %>%
  layer_dense(units = 10, activation = 'tanh') %>%
  layer_batch_normalization() %>%
  layer_dense(units = 5, activation = 'tanh') %>%
  layer_dense(units = 1, activation = k_exp)
```

hidden layers having $(q_1, q_2, q_3, q_4) = (20, 15, 10, 5)$ hidden neurons.
In Table 12, lines 'normalization layers', we present the result for the hyperbolic tangent activation and the ReLU activation. We note that in the present situation it is not clear whether these normalization layers lead to an improvement in our regression problem. For the hyperbolic tangent activation it leads to a slightly smaller in-sample loss (29.09103 versus 29.16868), for the ReLU activation just the opposite happens (29.0954 versus 29.02128). In general, we would expect that normalization layers are more effective for the ReLU activation because it is unbounded from above, whereas the hyperbolic tangent activation always leads to neurons $z_j^{(k)} \in [-1, 1]$.
We may also consider over-fitting in the previous examples by adding dropout layers after each normalization layer having dropout probabilities of $p \in \{1\%, 5\%, 10\%\}$ (these are the same values as in Table 9). On the remaining lines of Table 12 we provide the corresponding results, i.e. deep

$(q_1, q_2, q_3, q_4) = (20, 15, 10, 5)$	parameters	epochs	run time	in-sample loss	out-of-sample loss
hyperbolic tangent activation function:					
no normalization	1'336	300	481.24s	29.16868	30.45878
normalization layers (Listing 5)	1'426	300	637.89s	29.09103	30.45007
normalization & dropout $p = 1\%$	1'426	300	708.39s	29.04267	30.44643
normalization & dropout $p = 5\%$	1'426	300	735.20s	29.02275	30.29956
normalization & dropout $p = 10\%$	1'426	300	710.71s	29.22315	30.39477
rectified linear unit (ReLU) activation function:					
no normalization	1'336	300	473.15s	29.02128	30.30155
normalization layers	1'426	300	584.81s	29.09954	30.59820
normalization & dropout $p = 1\%$	1'426	300	722.79s	29.00352	30.45296
normalization & dropout $p = 5\%$	1'426	300	727.39s	29.10925	30.39030
normalization & dropout $p = 10\%$	1'426	300	754.45s	29.56844	30.59977

Table 12: comparison of different network architectures considering normalization and dropout layers; losses are in 10^{-2}.

networks $(q_1, q_2, q_3, q_4) = (20, 15, 10, 5)$ with normalization layers and dropout layers (after each normalization layer). In view of these results we prefer a slightly higher dropout rate than in the shallow networks of Table 9. An obvious explanation for this is that in the model of Table 12 we have much more degrees of freedom and therefore dropouts should also be more likely in order to prevent from over-fitting.
A main conclusion is that we get better fits (for the same number of epochs) in deep networks (with dropout) at the expense of much more run time. Therefore, it is not clear whether we should really go for a more complex deep network in our example. At least we cannot answer this question on the basis of our loss function, and other measures may also be relevant.

6.5 Averaging networks

In the previous section we have been analyzing different network architectures. Unfortunately, there is not a clear preference for one architecture. This is commonly the case because we have several competing models which provide roughly the same quality in prediction. One could also claim that the model extensions have been done in a rather unstructured way. For illustrative purposes, we focus on shallow networks, and we discuss averaging (or blending) of models with $q_1 = 10, 20, 30$. Similar considerations can be done for any other network architectures.
Assume we have two models of type (1.7) having $q_1^a = 10$ and $q_1^b = 20$ hidden neurons:

$$\begin{aligned}
\log \mu^a(\boldsymbol{x}, v) &= w_0^{(2,a)} + \sum_{j=1}^{q_1^a} w_j^{(2,a)} \tanh\left(w_{j,0}^{(1,a)} + \sum_{l=1}^{q_0-1} w_{j,l}^{(1,a)} x_l + w_{j,q_0}^{(1,a)} v \right), \\
\log \mu^b(\boldsymbol{x}, v) &= w_0^{(2,b)} + \sum_{j=1}^{q_1^b} w_j^{(2,b)} \tanh\left(w_{j,0}^{(1,b)} + \sum_{l=1}^{q_0-1} w_{j,l}^{(1,b)} x_l + w_{j,q_0}^{(1,b)} v \right).
\end{aligned}$$

Averaging these two models means that we consider a new model with $q_1 = q_1^a + q_2^b = 30$ hidden

neurons given by

$$\begin{aligned}\log \mu(\boldsymbol{x}, v) &= \frac{1}{2}\log \mu^a(\boldsymbol{x}, v) + \frac{1}{2}\log \mu^b(\boldsymbol{x}, v) \\ &= \sum_{i\in\{a,b\}} \left[\frac{1}{2}w_0^{(2,i)} + \sum_{j=1}^{q_1^i} \frac{1}{2}w_j^{(2,i)} \tanh\left(w_{j,0}^{(1,i)} + \sum_{l=1}^{q_0-1} w_{j,l}^{(1,i)} x_l + w_{j,q_0}^{(1,i)} v\right)\right].\end{aligned} \tag{6.6}$$

This is the idea behind averaging (or blending) models. We take the first two models of Table

architecture	parameters	epochs	run time	in-sample loss	out-of-sample loss
shallow network $q_1 = q_1^a = 10$	411	300	231.05s	29.72569	30.81615
shallow network $q_1 = q_1^b = 20$	821	300	272.64s	29.51171	30.65739
shallow network $q_1 = 30$	1'231	300	324.64s	29.44280	30.68640
averaging $q_1^a = 10$ & $q_1^b = 20$	1'231	–	–	29.55012	30.66832
shallow network $q_1 = 30$	1'231	300	323.44s	29.22012	30.55035

Table 13: blending models; the first three models are taken from Table 10; losses are in 10^{-2}.

10 and average these two models to a shallow network with $q_1 = q_1^a + q_1^b = 30$ hidden neurons. The result of this aggregated model is presented on the second last line of Table 13. Of course, the network parameter of this averaged model can now be used as initial network parameter in the GDM for the model with $q_1 = 30$ hidden neurons. If we perform 300 epochs with this new initial network parameter, we obtain the result on the last line of Table 13. Note that this model outperforms the model on the third line (with $q_1 = 30$) because we start the GDM in a more appropriate network parameter in the latter fitting. Thus, we save run time by using results from smaller network architectures as initial values because the bigger model already has a reasonable starting value.

Remark that this model averaging does not directly focus on the weaknesses of the individual models. That is, if the models based on $\mu^a(\cdot)$ and $\mu^b(\cdot)$ have the same weakness, also the averaged model will have this weakness. In the next section we analyze these weaknesses more systematically.

6.6 Outlook

In this section we have been discussing several strategies in choosing a neural network architecture. The answers provided may not seem sufficiently satisfying from a practical point of view because the actuarial profession may want to have a more systematic approach to improve their models. In Wüthrich–Merz [28] we provide such a more systematic approach, namely, we explain how classical actuarial regression models can be nested into neural networks. The neural networks then challenge these classical actuarial models for additional model structure that the classical model has not seen, yet. If the network can find such structure, the classical model should be enhanced, otherwise we can be confidently working with the classical actuarial model.

7 Challenging the networks with boosting

7.1 Boosting challenge

In (6.6) we have been aggregating models. However, there would be a better way of extending a model, namely, by including structure that has not been seen by the first model. This idea is very close to the boosting machine idea. Assume we have fitted a first regression function $\widehat{\mu} : (\boldsymbol{x}, v) \mapsto \widehat{\mu}(\boldsymbol{x}, v)$ which provides a first (estimated) model

$$N_i \overset{\text{ind.}}{\sim} \text{Poi}\left(\widehat{\mu}(\boldsymbol{x}_i, v_i)\right), \qquad \text{for } i = 1, \ldots, n.$$

We can now try to enhance this model by considering an additional regression function $\chi : \mathcal{X}^+ \to \mathbb{R}_+$, $(\boldsymbol{x}, v) \mapsto \chi(\boldsymbol{x}, v)$ such that we obtain a new model

$$N_i \overset{\text{ind.}}{\sim} \text{Poi}\left(\chi(\boldsymbol{x}_i, v_i)\widehat{\mu}(\boldsymbol{x}_i, v_i)\right) \overset{\text{(d)}}{=} \text{Poi}\left(\chi(\boldsymbol{x}_i, v_i)\nu_i\right), \qquad \text{for } i = 1, \ldots, n,$$

where we consider the working weights $\nu_i = \widehat{\mu}(\boldsymbol{x}_i, v_i) > 0$ as offsets in a Poisson regression model.

If this new regression function turns out to be identically equal to 1, i.e. $\chi(\cdot) \equiv 1$, then the first regression function $\widehat{\mu}(\cdot)$ is perfect. Otherwise, the first regression function should be enhanced to a new regression function $(\boldsymbol{x}, v) \mapsto \widehat{\mu}^+(\boldsymbol{x}, v) = \chi(\boldsymbol{x}, v)\widehat{\mu}(\boldsymbol{x}, v)$.

In order to assess the quality of the (network) regression function $\widehat{\mu}(\cdot)$ we use a Poisson boosting machine to estimate the additional regression function $\chi(\cdot)$. For the Poisson boosting machine we refer to Section 5 of Noll et al. [18]. In order to determine an appropriate number of boosting steps we use stratified 10-fold cross-validation, that is, we partition the learning data $\mathcal{D}$ into 10 stratified subsets $\mathcal{D}_1, \ldots, \mathcal{D}_{10}$. As trigger for stratifying we use the number of claims N_i. The corresponding R code is provided in Listing 6 (note that the learning data $\mathcal{D}$ is denoted by

Listing 6: R script for choosing 10 stratified subsets

```
1  set.seed(100)
2  # random ordering of learning data
3  learn$K <- runif(nrow(learn))
4  learn <- learn[order(learn$K),]
5  # order learning data according to the number of claims
6  learn <- learn[order(learn$ClaimNb, decreasing=TRUE),]
7  # choosing stratifying allocation of learning data
8  learn$K <- rep(1:10, length = nrow(learn))
9  # choose validation and training set for some k=1,...,10
10 validation_k <- learn[which(learn$K==k),]
11 train_k <- learn[which(learn$K!=k),]
```

`learn`). For every $k = 1, \ldots, 10$ we receive validation set $\mathcal{D}_k$ and training set $\mathcal{D}_{-k} = \mathcal{D} \setminus \mathcal{D}_k$ on lines 10-11 of Listing 6. Based on these sets, we fit a Poisson boosting model $\widehat{\chi}^{(-k)}(\cdot)$ on the training data $\mathcal{D}_{-k}$ and we do an out-of-sample model validation on the validation set $\mathcal{D}_k$. This gives us out-of-sample validation losses for $k = 1, \ldots, 10$

$$\mathcal{L}_k = \mathcal{L}(\mathcal{D}_k, \widehat{\chi}^{(-k)}(\cdot)) \;=\; \frac{1}{|\mathcal{D}_k|} \sum_{(\boldsymbol{x}_i, v_i) \in \mathcal{D}_k} 2N_i \left[\frac{\widehat{\chi}^{(-k)}(\boldsymbol{x}_i, v_i)\nu_i}{N_i} - 1 - \log\left(\frac{\widehat{\chi}^{(-k)}(\boldsymbol{x}_i, v_i)\nu_i}{N_i} \right) \right].$$

This provides us with a stratified 10-fold cross-validation loss and the corresponding standard deviation estimate of, respectively,

$$\mathrm{CV}^{(10)} = \frac{1}{10}\sum_{k=1}^{10}\mathcal{L}_k \qquad \text{and} \qquad \sigma_{\mathrm{CV}}^{(10)} = \sqrt{\frac{1}{9}\sum_{k=1}^{10}\left(\mathcal{L}_k - \mathrm{CV}^{(10)}\right)^2}.$$

The aim will be to see whether the Poisson boosting machine is able to lower these cross-validation losses compared to the network estimations. We do this with two examples at hand.

7.1.1 Boosting challenge 1: ridge regularized shallow network of Table 9

As a first network model for the challenge we choose the ridge regularized shallow network of Table 9 which provides an in-sample loss of 29.09129 and an out-of-sample loss of 30.26989. We then apply a Poisson boosting machine based on binary regression trees of maximal depth $\texttt{J0} = 1, 2, 3$ and for $D = 30$ boosting iterations to each training set $\mathcal{D}_{-k}$. The corresponding R code is provided in Listing 7. This gives us for each maximal tree depth $\texttt{J0} = 1, 2, 3$ and for each

Listing 7: R script for cross-validated Poisson boosting

```
1  D <- 30
2  validation.loss <- array(NA, c(10+2, D+1))
3  learn$predBB <- learn$predNN   # initialize working weights nu_i with network predictions
4
5  for (k in 1:10){
6     validation_k <- learn[which(learn$K==k),]
7     train_k <- learn[which(learn$K!=k0),]
8     validation.loss[k,1] <- loss.function(validation_k$predBB, validation_k$ClaimNb))
9     for (d in 1:D){
10       tree_k <- rpart(cbind(predBB,ClaimNb) ~ Area + VehPower + VehAge + DrivAge +
11                               BonusMalus + VehBrand + VehGas + Density + Region + Exposure,
12                               data=train_k, method="poisson", control=rpart.control(xval=1,
13                               minbucket=10000, maxdepth=J0, maxsurrogate=0, cp=0.00001))
14       train_k$predBB <- predict(tree_k) * train_k$predBB
15       validation_k$predBB <- predict(tree_k) * validation_k$predBB
16       validation.loss[k,d+1] <- loss.function(validation_k$predBB, validation_k$ClaimNb))
17                       }}
18 for (d in 1:(D+1)){
19    validation.loss[K+1,d] <- mean(validation.loss[1:K,d])
20    validation.loss[K+2,d] <- sd(validation.loss[1:K,d])
21                       }
```

boosting iteration $d = 1, \ldots, D$ the resulting stratified 10-fold cross-validation losses $\mathrm{CV}^{(10)}_{\texttt{J0},d}$ and the corresponding standard deviation estimates $\sigma^{(10)}_{\mathrm{CV}_{\texttt{J0},d}}$, see also lines 18-21 in Listing 7. These results are presented in Figure 18 for maximal tree depth $\texttt{J0} = 1$ (lhs, red), maximal tree depth $\texttt{J0} = 2$ (middle, blue), and maximal tree depth $\texttt{J0} = 3$ (rhs, green). The horizontal lines show the 1-SD model selection rule for the necessary number of boosting steps. For maximal tree depth $\texttt{J0} = 1$ we see that all cross-validation losses $\mathrm{CV}^{(10)}_{1,d}$, $d = 1, \ldots, D$, (black dots) lie below the 1-SD rule line (horizontal red line in Figure 18), therefore, a boosting improvement with $\texttt{J0} = 1$ is not justified. For maximal tree depths $\texttt{J0} = 2, 3$ we receive roughly the same 1-SD rule lines (horizontal blue and green lines in Figure 18). For $\texttt{J0} = 2$ this justifies 3 boosting iterations and for $\texttt{J0} = 3$ this gives 2 boosting iterations because $\mathrm{CV}^{(10)}_{2,d=3}$ and $\mathrm{CV}^{(10)}_{3,d=2}$ (black dots) lie above the 1-SD rule lines. The discrepancy between $\texttt{J0} = 1$ and $\texttt{J0} \geq 2$ can be explained by the

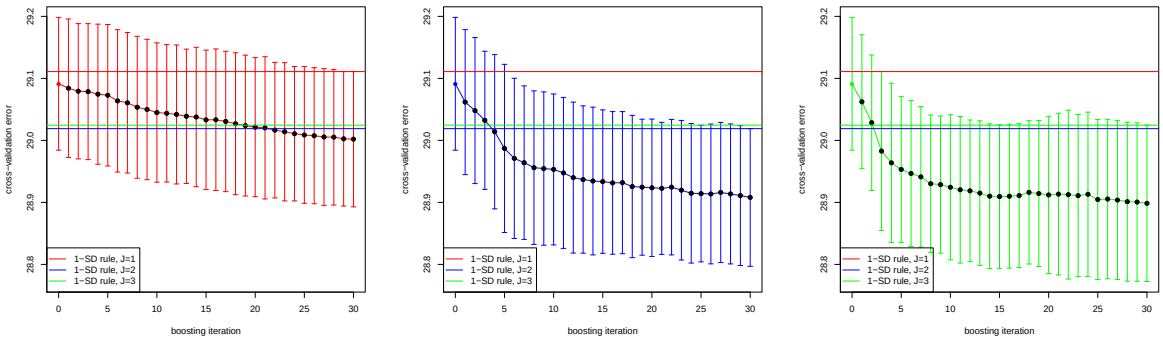

Figure 18: boosting challenge of the ridge regularized shallow network of Table 9: stratified 10-fold cross-validation losses $\mathrm{CV}^{(10)}_{\mathtt{J0},d} \pm \sigma^{(10)}_{\mathrm{CV}_{\mathtt{J0}},d}$ for iterations $d = 1, \ldots, D = 30$ for (lhs) maximal tree depth $\mathtt{J0} = 1$; (middle) maximal tree depth $\mathtt{J0} = 2$; and (rhs) maximal tree depth $\mathtt{J0} = 3$.

argument that the ridge regularized shallow network prediction captures additive interactions on the log-scale, but it misses some of the more complex interactions which can more easily be received by deeper trees. This is exactly the same consideration as in Example 6.2.
We now perform these boosting steps for $\mathtt{J0} = 2, 3$ on the entire learning data $\mathcal{D}$ and calculate the corresponding in- and out-of-sample losses. These results are presented in Table 14. In both

	in-sample loss	out-of-sample loss
initial ridge regularized model with $q_1 = 20$	29.09129	30.26989
1st boosting step depth $\mathtt{J0} = 2$	29.05953	30.22339
2nd boosting step depth $\mathtt{J0} = 2$	29.04934	30.21130
3rd boosting step depth $\mathtt{J0} = 2$	29.04156	30.20938
1st boosting step depth $\mathtt{J0} = 3$	29.04627	30.22593
2nd boosting step depth $\mathtt{J0} = 3$	28.96970	30.16496

Table 14: boosting steps with Poisson regression trees of depth $\mathtt{J0} = 2, 3$; losses are in 10^{-2}.

cases we see an improvement of the loss figures, and the initial ridge regularized shallow network estimator $\widehat{\mu}(\cdot)$ should be enhanced by the resulting boosting regression estimators $\chi(\cdot)$, resulting in the new regression function $\widehat{\mu}^{+}(\cdot) = \chi(\cdot)\widehat{\mu}(\cdot)$. This provides out-of-sample losses of 30.20938 and 30.16496 for $\mathtt{J0} = 2$ and $\mathtt{J0} = 3$, respectively.
In Figure 19 we analyze the two boosting steps that lead to the out-of-sample loss of 30.16496 for $\mathtt{J0} = 3$. First boosting step in Figure 19 (lhs): The first 3 splits are done for the feature component `Exposure`, i.e. our shallow network does not consider the exposures in a fully appropriate way, yet. Thereafter, we observe interactions between `Exposure`, `VehBrand` and `VehAge`, as well as between `Exposure`, `BonusMalus` and `DrivAge`. The second boosting step in Figure 19 (rhs) provides interactions and fine-tuning of `BonusMalus` and `DrivAge`. Thus, we receive quite some insight how the ridge regularized shallow network should be improved.

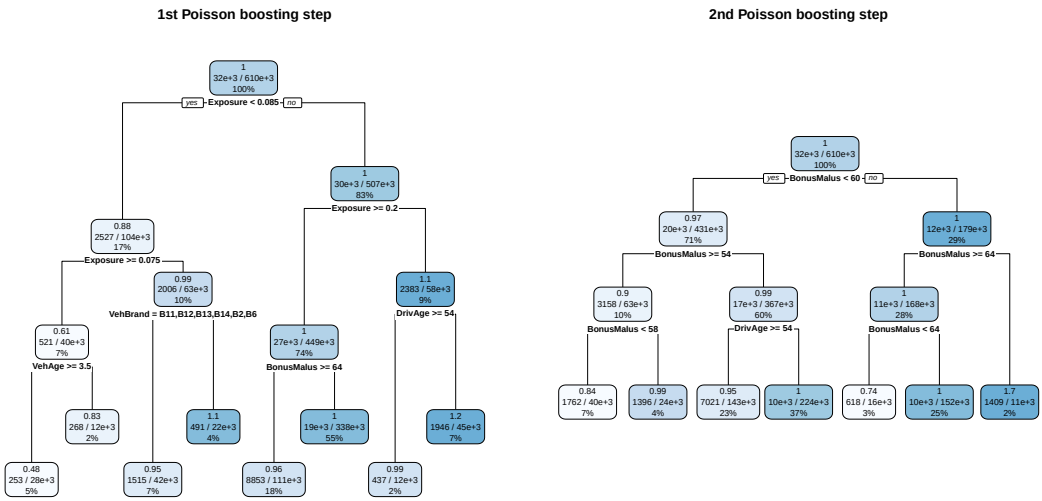

Figure 19: boosting challenge of ridge regularized shallow network: (lhs) 1st iteration of boosting algorithm; (rhs) 2nd iteration of boosting algorithm for maximal tree depth $\texttt{J0} = 3$ (note that these regression trees have been plotted with the R function `rpart.plot` which applies rounding to the labels).

7.1.2 Boosting challenge 2: deep network of Table 12

As a second example we choose a deep network of Table 12 having $K = 4$ hidden layers with hidden neurons $(q_1, q_2, q_3, q_4) = (20, 15, 10, 5)$ and hyperbolic tangent activation function. Running this network with normalization layers and dropouts $p = 5\%$ for 500 epochs we were able to improve the in-sample loss from 29.02275 to 28.79469 and the out-of-sample loss from 30.29956 to 30.13439. We choose this latter deep network as starting value $\widehat{\mu}(\cdot)$ of the boosting algorithm. Then we perform stratified 10-fold cross-validation completely similarly as above. The results

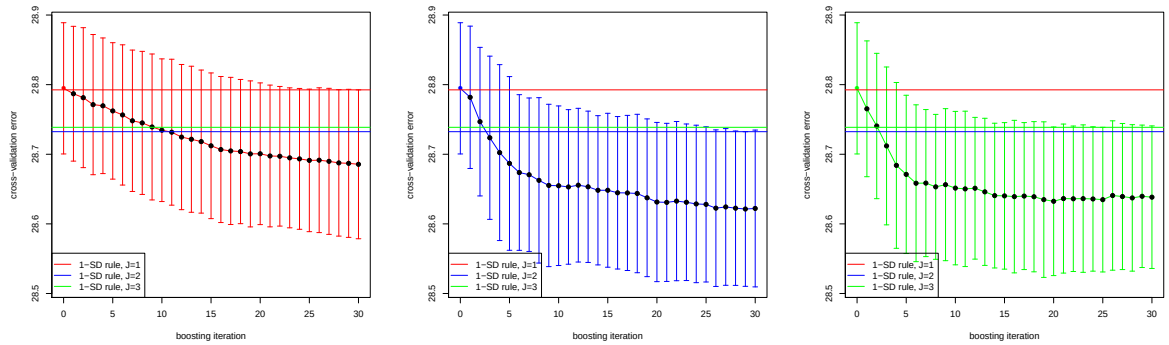

Figure 20: boosting challenge of a deep network of Table 12: stratified 10-fold cross-validation losses $\mathrm{CV}^{(10)}_{\texttt{J0},d} \pm \sigma^{(10)}_{\mathrm{CV}_{\texttt{J0},d}}$ for iterations $d = 1, \ldots, D = 30$ for (lhs) maximal tree depth $\texttt{J0} = 1$; (middle) maximal tree depth $\texttt{J0} = 2$; and (rhs) maximal tree depth $\texttt{J0} = 3$.

are presented in Figure 20, they look similar to Subsection 7.1.1. In the boosting we should

allow for more complex interactions and choose J0 $= 2, 3$, resulting in the graphs in the middle and rhs of Figure 20 (blue and green). For both tree depths we conclude from stratified 10-fold cross-validation that we should perform an additional two boosting steps.

	in-sample loss	out-of-sample loss
deep network $(q_1, q_2, q_3, q_4) = (20, 15, 10, 5)$	28.79469	30.13439
1st boosting step depth J0 $= 2$	28.77370	30.11873
2nd boosting step depth J0 $= 2$	28.70221	30.06373
1st boosting step depth J0 $= 3$	28.74757	30.08976
2nd boosting step depth J0 $= 3$	28.72844	30.06688

Table 15: boosting steps with Poisson regression trees of depth J0 $= 2, 3$; losses are in 10^{-2}.

We perform these two boosting steps for J0 $= 2, 3$ and obtain the results in Table 15. The out-of-sample analysis justifies these two boosting steps, and we slightly prefer the simpler model with J0 $= 2$. These two boosting steps are illustrated in Figure 21. The first one provides an

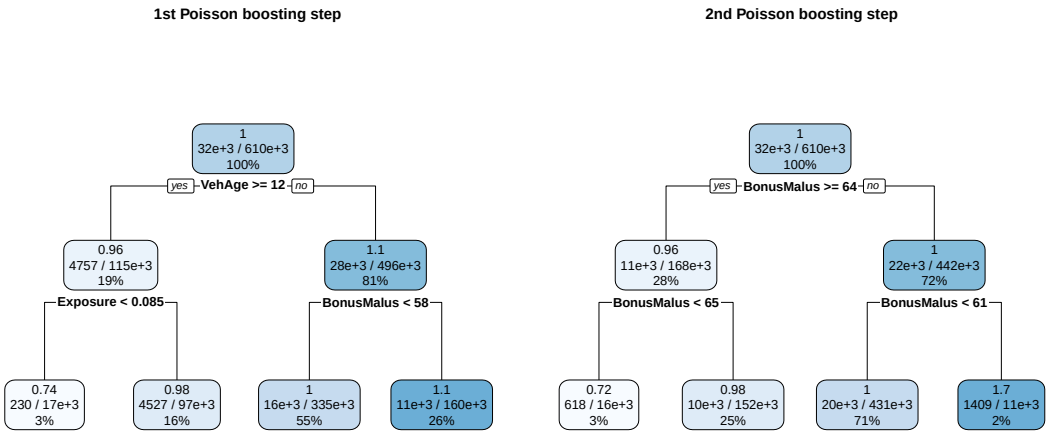

Figure 21: boosting challenge of deep network $(q_1, q_2, q_3, q_4) = (20, 15, 10, 5)$: (lhs) 1st iteration of boosting algorithm; (rhs) 2nd iteration of boosting algorithm for J0 $= 2$.

interaction between VehAge, Exposure and BonusMalus, whereas the second boosting step is only needed to fine-tune the BonusMalus level.

7.2 Outlook: enhancement of networks

In the previous subsections we have provided boosting improved networks, see Table 14 and 15. We can now work with these results. However, these might be a bit unsatisfactory because we may want to fully rely on networks.[21] One way to proceed is to use the working weights

[21] Remark that boosting with regression trees leads to discontinuities in regression functions which is sometimes not wanted. Moreover, also extrapolation is more questionable with regression trees because slopes are zero at boundaries.

$\nu_i = \widehat{\mu}(\boldsymbol{x}_i, v_i) > 0$ as offsets in a next network Poisson regression modeling approach. That is, we design a network $\chi : \mathcal{X}^+ \to \mathbb{R}_+$ under the model assumption

$$N_i \overset{\text{ind.}}{\sim} \text{Poi}\left(\chi(\boldsymbol{x}_i, v_i)\nu_i\right), \qquad \text{for } i = 1, \ldots, n. \tag{7.1}$$

The resulting regression function $\widehat{\mu}^+(\cdot) = \chi(\cdot)\widehat{\mu}(\cdot)$ can then be understood as a new network, in a similar fashion as in (6.6). To benefit from the boosting analysis, we could only include the feature components that are relevant for the boosting splits, see Figures 19 and 21, or we could initialize the network weights for the modeling of the regression function $\chi(\cdot)$ in such a way that they reflect these boosting splits.

Listing 8: enhancement of networks

```
1  dat.X <- Xlearn[,c(3,6,8)]   # select Exposure, VehAge, BonusMalus
2  dat.W <- learn$predNN        # define offset
3  q1 <- 5                      # first hidden layer
4  q2 <- 3                      # second hidden layer
5
6  features <- layer_input(shape=c(ncol(dat.X)))
7  net <- features %>%
8       layer_dense(units = q1, activation = 'tanh') %>%
9       layer_dense(units = q2, activation = 'tanh') %>%
10      layer_dense(units = 1, activation = k_exp)
11 volumes <- layer_input(shape=c(1))
12 offset <- volumes %>%
13      layer_dense(units = 1, activation = 'linear', use_bias=FALSE, trainable=FALSE,
14                                                    weights=list(array(1, dim=c(1,1))))
15 merged <- list(net, offset) %>%
16      layer_multiply()
17 model <- keras_model(inputs=list(features, volumes), outputs=merged)
18 summary(model)
19 model %>% compile(loss = 'poisson', optimizer = 'rmsprop')
20
21 fit <- model %>% fit(list(dat.X, dat.W), learn$ClaimNb, epochs=100, batch_size=10000)
```

In Listing 8 we give an example of such a neural network enhancement (7.1). On line 1 we select the three feature components `Exposure`, `VehAge` and `BonusMalus`. These are used in a deep neural network of depth $K = 2$ with $q_1 = 5$ and $q_2 = 3$ hidden neurons, respectively, see lines 3-10 of Listing 8. This provides regression function $\chi(\cdot)$ in (7.1). On line 2 we define the working weights ν_i which act as offsets on lines 11-14 of Listing 8. On lines 15-17 we merge regression function $\chi(\cdot)$ and offsets ν_i by multiplying them to the new regression function $\widehat{\mu}^+(\cdot)$. On line 21 we fit this enhancement. This additional calibration step has led to a new in-sample loss of 28.78369 and an out-of-sample loss of 30.09593. Thus, we receive an improvement compared to the original model (see first line of Table 15), but the improvement is slightly less pronounced compared to the boosting versions in Table 15.

8 Analysis of out-of-sample results

In this section we provide a final analysis of our results. This is done in a similar spirit as in the last section of Noll et al. [18]. We compare the Poisson boosting machine model PBM3 of Noll et al. [18], the boosting improved ridge regularized shallow network of Table 14, and the boosting improved deep network $(q_1, q_2, q_3, q_4) = (20, 15, 10, 5)$ of Table 15. The corresponding

method	in-sample loss	out-of-sample loss
boosting machine model PBM3 of Noll et al. [18], Table 10	30.09093	31.41314
ridge regularized shallow network $q_1 = 20$	29.09129	30.26989
boosting improved, Table 14	28.96970	30.16496
deep network $(q_1, q_2, q_3, q_4) = (20, 15, 10, 5)$	28.79469	30.13439
boosting improved, Table 15	28.70221	30.06373

Table 16: boosting steps with Poisson regression trees of depth $\texttt{J0} = 2, 3$; losses are in 10^{-2}.

in- and out-of-sample losses are summarized in Table 16. We remark that this comparison is

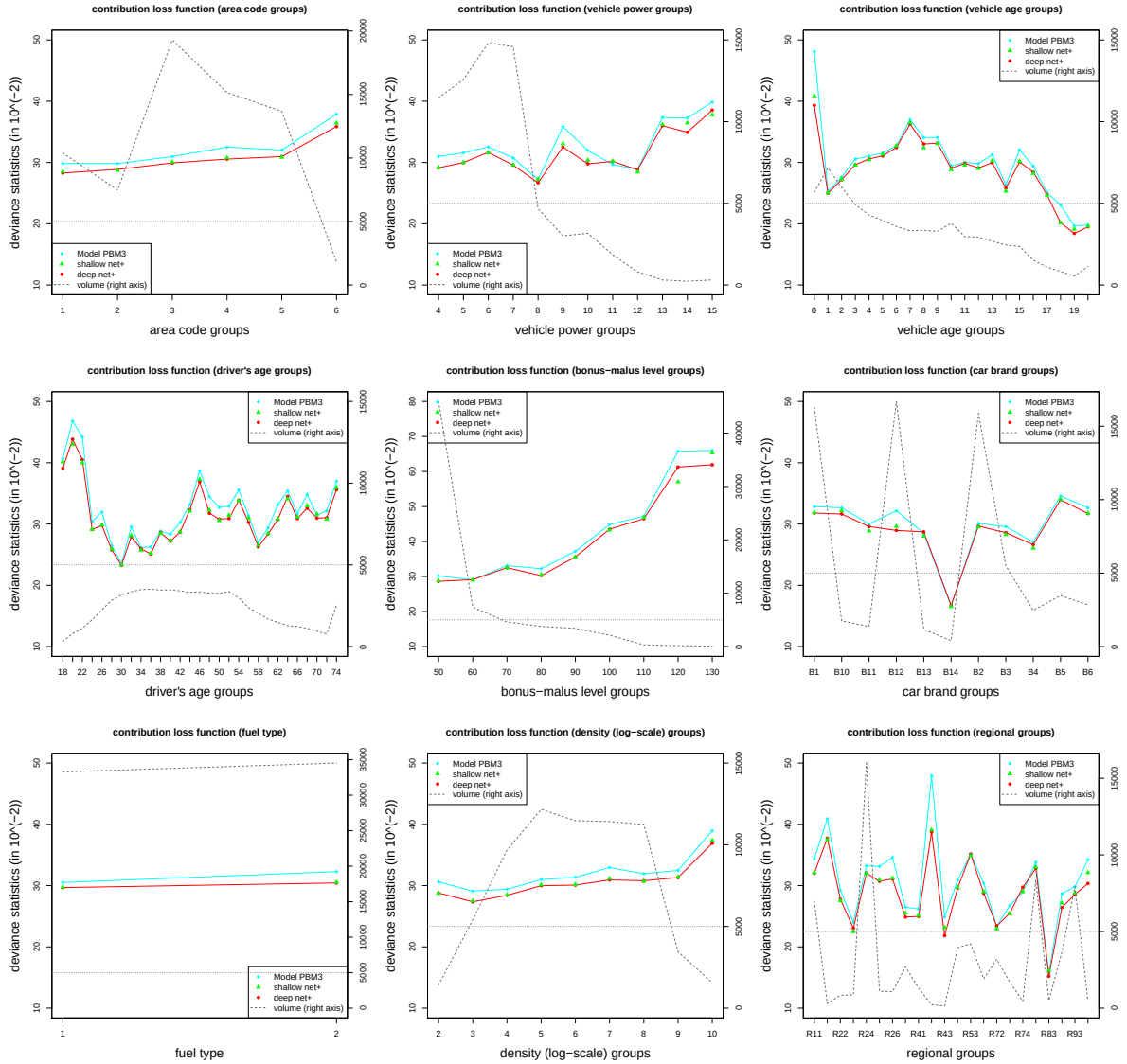

Figure 22: (lhs) average marginal out-of-sample loss per label on test data set $\mathcal{T}$: comparison of Model PBM3, boosting improved ridge regularized shallow network, and boosting improved deep network, units on the y-axes are in 10^{-2}.

not completely fair because in model PBM3 we have not been modeling the exposures, but we have worked with model assumption (1.1). To make the comparison more fair we could also lift model PBM3 to model assumptions (1.2), and then run the boosting algorithm under these model assumptions (which also partitions w.r.t. `Exposure`). The resulting regression function will have discontinuities in the exposure which we consider as a major disadvantage. Therefore, we refrain from further refining model PBM3.

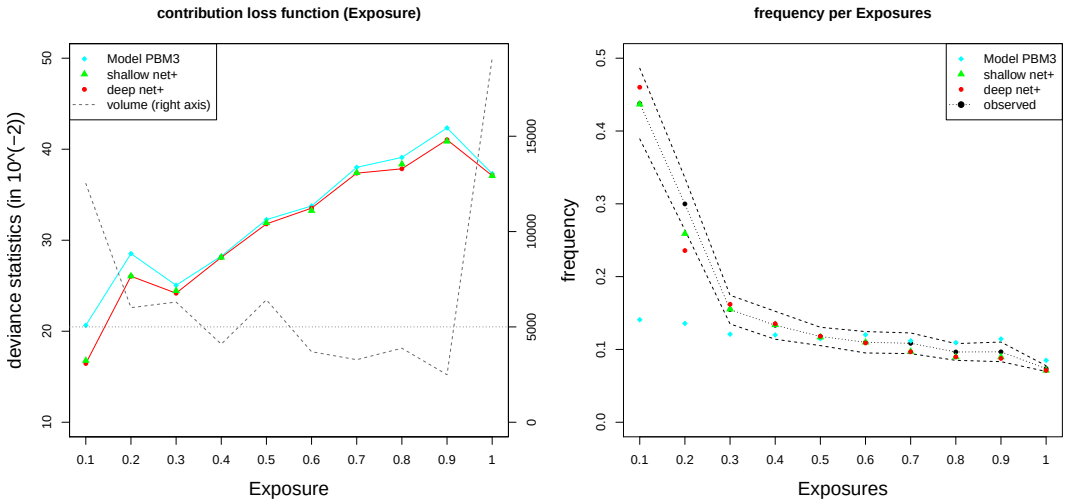

Figure 23: average marginal out-of-sample loss per exposure level and marginal frequency on test data set $\mathcal{T}$: comparison of Model PBM3, boosting improved ridge regularized shallow network, and boosting improved deep network, units on the y-axes are in 10^{-2}.

We calculate for every label the average marginal out-of-sample loss on the test data $\mathcal{T}$. These are given by

$$\frac{1}{\sum_{t=1}^{n_{\mathcal{T}}} \mathbb{1}_{\{x_{t,l}=x\}}} \sum_{t=1}^{n_{\mathcal{T}}} 2N_t \left[\frac{\widehat{\mu}(\boldsymbol{x}_t, v_t)}{N_t} - 1 - \log\left(\frac{\widehat{\mu}(\boldsymbol{x}_t, v_t)}{N_t} \right) \right] \mathbb{1}_{\{x_{t,l}=x\}},$$

for x being in the domain of the l-th component of $\boldsymbol{x} \in \mathcal{X}$, and $\widehat{\mu}(\cdot)$ being an estimated model. These statistics are plotted in Figure 22: Model PBM3 in cyan color, the boosting improved ridge regularized shallow network in green color, and boosting improved deep network in red color; moreover, the gray dotted line shows the volumes (number of policies) per label (with y-axis on the rhs). We start by discussing Model PBM3 (cyan) and the deep network results (red). From the Figure 22 (top, lhs) we can see that the network results are better over all `Area` codes. Big improvements in out-of-sample losses are made by the latter model for `VehAge` $= 0$ and `VehBrand` $=$ B12. In Figure 23 (lhs) we provide the same figure, but as a function of the exposure. Recall that for Model PBM3 we have model assumption (1.1) (which is linear in the exposure) and for the network approaches we have model assumption (1.2) (which allows for non-linearities in the exposure modeling). We observe major improvements in the volume modeling for `Exposure` $\in (0, 0.2]$, i.e. for short exposures, see loss statistics in Figure 23 (lhs). This short exposures also interact with `VehAge` $= 0$ and `VehBrand` $=$ B12, see Figure 2. This explains the previously mentioned improvements in out-of-sample losses on these labels. In

Figure 23 (rhs) we also provide the resulting marginal frequencies for different exposures. We observe that this is clearly non-linear in the exposure and that the models $\mu(\boldsymbol{x}, v)$ are better able to capture this non-linearity.
The out-of-sample losses of the two network models appear to be rather similar, see green and red colors in Figures 22 and 23 (lhs). Bigger differences mainly occur on labels with small volumes, and maybe the major difference is on `VehAge` $= 0$.

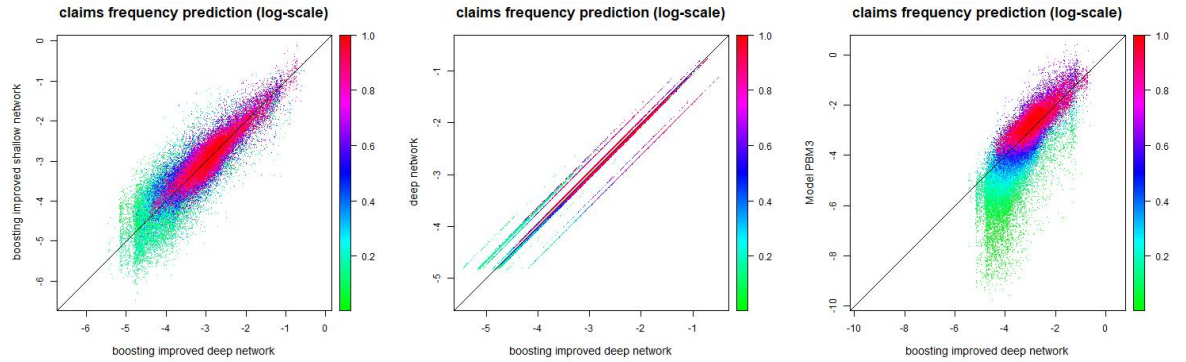

Figure 24: Out-of-sample claims frequency predictions (on log-scales) of the deep network, the boosting improved deep network, the boosting improved shallow network and the Model PBM3 of [18].

In Figure 24 we present the resulting claims frequency predictions of the 4 methods: deep network, the boosting improved deep network, the boosting improved shallow network and the Model PBM3 of [18]. The figure on the lhs compares the boosting improved deep network (x-axis) to the boosting improved shallow network (y-axis). Each dot represents a policy $t = 1, \ldots, n_{\mathcal{T}}$ and the colors are given according to the corresponding exposures. We observe that there are quite some differences between the two models on the individual policy level. Of course, this is very unpleasant from a practical point of view because it implies that the pricing may be too much model dependent (or even dependent on the calibration chosen). This suggests averaging over different models, in the sense of Section 6.5, which will robustify the pricing.
Figure 24 (middle) provides the comparison between the deep network and its boosting improved counterpart of Table 16. The boosting improvement results in the multiplication of twice 4 leaves, see Figure 21. This provides the improved portfolios on the diagonal lines in Figure 24 (middle).
Finally, we compare the boosting improved deep network to the Model PBM3 of [18] in Figure 24 (rhs). Not surprisingly, the latter model provides substantially lower frequencies for smaller exposures. This finishes our example.

9 Conclusions and outlook

This tutorial is considering the use of neural network regression models for claims frequency modeling in insurance. We have discussed all aspects of designing such a neural network regression model: this includes feature pre-processing, choice of loss function, choice of explicit neural network architecture, class imbalance problem, over-fitting, regularization, dropout, as well as model blending. For the numerical analysis we have used the R interface to Keras (an API to

TensorFlow). Moreover, we have provided graphical tools to analyze neural networks, we have discussed model improvements by boosting, and as a side product we have seen that a linear time exposure is non-optimal for our claims frequency data set.
A main deficiency of our considerations is that we have not been discussing prediction uncertainty which, in particular, should also include model uncertainty. At the moment, this is an open issue that requires further research. This will require a more systematic selection of the model and the model calibration, some advice is given in [28].
The previous analysis has been focusing on claims frequencies in motor third party liability insurance which, by nature, is rather nice data for sufficiently large insurance portfolios. We expect much more difficulties for claims size modeling, in particular, caused by larger claims. It is another open issue to deal with such situations.
Another open point is to deal with high-dimensional feature spaces that contain many categorical feature components. In such situations one can easily run into curse of dimensionality problems that have not been present in our analysis. We believe that a more efficient way to represent categorical feature components is to use embedding layers, this has recently been demonstrated in [22]. Moreover, missing values will pose another challenge we may have to deal with, here maybe generative-adversarial networks (GAN) are of help because these manage to impute missing values.

Acknowledgment. We would like to kindly thank Jürg Schelldorfer (Swiss Re), Ronald Richman (AIG), Christian Lorentzen (Mobiliar), Andrea Gabrielli (ETH Zurich), John Ery (ETH Zurich) and Ulrich Riegel (Munich Re) for detailed comments that have helped us to substantially improve this tutorial.

References

[1] Arnold, V.I. (1957). On functions of three variables. *Doklady Akademii Nauk SSSR* **114/4**, 679-681.

[2] Barron, A.R. (1993). Universal approximation bounds for superpositions of sigmoidal functions. *IEEE Transactions of Information Theory* **39/3**, 930-945.

[3] CASdatasets Package Vignette (2016). Reference Manual, May 28, 2016. Version 1.0-6. Available from `http://cas.uqam.ca`.

[4] Charpentier, A. (2015). *Computational Actuarial Science with R.* CRC Press.

[5] Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems* **2**, 303-314.

[6] Döhler, S., Rüschendorf, L. (2001). An approximation result for nets in functional estimation. *Statistics and Probability Letters* **52**, 373-380.

[7] Glorot, X., Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In: Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, *Proceedings of Machine Learning Research* **9**, 249-256.

[8] Gneiting, T. (2011). Making and evaluation point forecasts. *Journal of the American Statistical Association* **106/494**, 746-762.

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

[10] Hastie, T., Tibshirani, R., Friedman, J. (2009). *The Elements of Statistical Learning. Data Mining, Inference, and Prediction.* 2nd edition. Springer Series in Statistics.

[11] Hornik, K., Stinchcombe, M., White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks* **2**, 359-366.

[12] Isenbeck, M., Rüschendorf, L. (1992). Completeness in location families. *Probability and Mathematical Statistics* **13**, 321-343.

[13] Kolmogorov, A. (1957). On the representation of continuous functions of many variables by superposition of continuous functions of one variable and addition. *Doklady Akademii Nauk SSSR* **114/5**, 953-956.

[14] Leshno, M., Lin, V.Y., Pinkus, A., Schocken, S. (1993). Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks* **6/6**, 861-867.

[15] Makavoz, Y. (1996). Random approximants and neural networks. *Journal of Approximation Theory* **85**, 98-109.

[16] Montúfar, G., Pascanu, R., Cho, K., Bengio, Y. (2014). On the number of linear regions of deep neural networks. *Neural Information Processing Systems Proceedings*$^{\beta}$ **27**, 2924-2932.

[17] Nielsen, M. (2017). *Neural Networks and Deep Learning.* Online book available on http://neuralnetworksanddeeplearning.com

[18] Noll, A., Salzmann, R., Wüthrich, M.V. (2018). Case study: French motor third-party liability claims. *SSRN Manuscript* ID 3164764. Version November 8, 2018.

[19] Park, J., Sandberg, I. (1991). Universal approximation using radial-basis function networks. *Neural Computation* **3**, 246-257.

[20] Park, J., Sandberg, I. (1993). Approximation and radial-basis function networks. *Neural Computation* **5**, 305-316.

[21] Petrushev, P. (1999). Approximation by ridge functions and neural networks. *SIAM Journal on Mathematical Analysis* **30/1**, 155-189.

[22] Richman, R., Wüthrich, M.V. (2018). A neural network extension of the Lee-Carter model to multiple populations. SSRN Manuscript, ID 3270877, Version October 22, 2018.

[23] Rüger, S.M., Ossen, A. (1997). The metric structure of weight space. *Neural Processing Letters* **5**, 63-72.

[24] Shaham, U., Cloninger, A., Coifman, R.R. (2015). Provable approximation properties for deep neural networks. *arXiv:*1509.07385v3. Version March 28, 2016.

[25] Verbelen, R., Antonio, K., and Claeskens, G. (2016). Unraveling the predictive power of telematics data in car insurance. To appear in *Journal of the Royal Statistical Society: Series C (Applied Statistics).*

[26] Wüthrich, M.V. (2013). *Non-Life Insurance: Mathematics & Statistics.* SSRN Manuscript ID 2319328. Version December 21, 2017.

[27] Wüthrich, M.V., Buser, C. (2016). *Data Analytics for Non-Life Insurance Pricing.* SSRN Manuscript ID 2870308. Version June 13, 2018.

[28] Wüthrich, M.V., Merz, M. (2019). Editorial: Yes, we CANN! *ASTIN Bulletin* **49/1**.

[29] Yukich, J., Stinchcombe, M., White, H. (1995). Sup-norm approximation bounds for networks through probabilistic methods. *IEEE Transactions on Information Theory* **41/4**, 1021-1027.

[30] Zaslavsky, T. (1975). Facing up to arrangements: face-count formulas for partitions of space by hyperplanes. *Memoirs of the American Mathematical Society* **154**.