Feed-Forward Neural Networks (FNN)

French Motor Third-Party Liability Claims

Daniel Meier and Jürg Schelldorfer, with support from Christian Lorentzen, Friedrich Loser, Michael Mayer, Mario V. Wüthrich and Mirai Solutions GmbH.

2021-15-10

Introduction

This notebook was created for the course "Deep Learning with Actuarial Applications in R" of the Swiss Association of Actuaries (https://www.actuaries.ch/).

This notebook serves as companion to the tutorial "Insights from Inside Neural Networks", available on SSRN.

The code is similar to the code used in above tutorial and combines the raw R code in the scripts, available on GitHub along with some more comments. Please refer to the tutorial for explanations.

Note that the results might vary depending on the R and Python package versions, see last section for the result of `sessionInfo()` and corresponding info on the Python setup.

Data Preparation

The tutorial uses the French MTPL data set available on openML (ID 41214).

Load packages and data

```
#library(mgcv)
library(keras)
library(magrittr)
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
library(tibble)
library(purrr)
```

```
##
## Attaching package: 'purrr'
```

```
## The following object is masked from 'package:magrittr':
##
##     set_names
```

```
library(ggplot2)
library(gridExtra)
```

```
##
## Attaching package: 'gridExtra'
```

```
## The following object is masked from 'package:dplyr':
##
##     combine
```

```
library(splitTools)
library(tidyr)
```

```
##
## Attaching package: 'tidyr'
```

```
## The following object is masked from 'package:magrittr':
##
##     extract
```

```
# plotting parameters in R Markdown notebook
knitr::opts_chunk$set(fig.width = 9, fig.height = 9)
# plotting parameters in Jupyter notebook
library(repr)  # only needed for Jupyter notebook
options(repr.plot.width = 9, repr.plot.height = 9)
```

Set global parameters

```
options(encoding = 'UTF-8')
```

```
# set seed to obtain best reproducibility. note that the underlying architecture may affect results non
seed <- 100
Sys.setenv(PYTHONHASHSEED = seed)
set.seed(seed)
reticulate::py_set_seed(seed)
tensorflow::tf$random$set_seed(seed)
```

The results below will not exactly match the results in the paper, since the underlying dataset and some packages are different. In addition the split into training and testing data is different as well. However, the general conclusions remain the same.

Helper functions

Subsequently, for ease of reading, we provide all the helper functions which are used in this tutorial in this section.

```
summarize <- function(...) suppressMessages(dplyr::summarize(...))
```

```
load_data <- function(file) {
  load(file.path("../0_data/", file), envir = parent.frame(1))
}
```

```
# Poisson deviance
PoissonDeviance <- function(pred, obs) {
```

```
    200 * (sum(pred) - sum(obs) + sum(log((obs/pred)^(obs)))) / length(pred)
}
```

```
plot_freq <- function(test, xvar, title, model, mdlvariant) {
  out <- test %>% group_by(!!sym(xvar)) %>%
            summarize(obs = sum(ClaimNb) / sum(Exposure), pred = sum(!!sym(mdlvariant)) / sum(Exposure))

  ggplot(out, aes(x = !!sym(xvar), group = 1)) +
    geom_point(aes(y = pred, colour = model)) +
    geom_point(aes(y = obs, colour = "observed")) +
    geom_line(aes(y = pred, colour = model), linetype = "dashed") +
    geom_line(aes(y = obs, colour = "observed"), linetype = "dashed") +
    ylim(0, 0.35) + labs(x = xvar, y = "frequency", title = title) +
    theme(legend.position = "bottom")
}
```

```
plot_loss <- function(x) {
  if (length(x) > 1) {
    df_val <- data.frame(epoch = 1:length(x$loss), train_loss = x$loss, val_loss = x$val_loss)
    df_val <- gather(df_val, variable, loss, -epoch)
    p <- ggplot(df_val, aes(x = epoch, y = loss)) + geom_line() +
      facet_wrap(~variable, scales = "free") + geom_smooth()
    suppressMessages(print(p))
  }
}
```

Load data

We consider the data `freMTPL2freq` included in the R package `CASdatasets` for claim frequency modeling. This data comprises a French motor third-party liability (MTPL) insurance portfolio with corresponding claim counts observed in one accounting year. We do not incorporate claim sizes which would also be available through freMTPL2sev.

As the current package version provides a slightly amended dataset, we use an older dataset available on openML (ID 41214). Before we can use this data set we need to do some data cleaning. It has been pointed out by F. Loser that some claim counts do not seem to be correct. Hence, we use the pre-processing of the data described in the book "Statistical Foundations of Actuarial Learning and its Applications" in Appendix A.1. This pre-processed data can be downloaded from the course GitHub page here.

```
load_data("freMTPL2freq.RData")
```

General data preprocessing

A priori, there is not sufficient information about this data to do a sensible decision about the best consideration of the exposure measure, either as feature or as offset. In the following we treat the exposure always as an offset.

Data preprocessing includes a couple of transformations. We ensure that `ClaimNb` is an integer, `VehAge`, `DrivAge` and `BonusMalus` have been capped for the plots at age 20, age 90 and bonus-malus level 150, respectively, to improve visualization. `Density` is logarithmized and `VehGas` is a categorical variable. We leave away the rounding used in the first notebook, which were mainly used for nicer visualizations of the data.

We are adding a group_id identifying rows possibly referring to the same policy. Respecting group_id in data splitting techniques (train/test, cross-validation) is essential. This is different to the tutorial where

another splitting has been used. As a consequence, the figures in this notebook do not match the figures in the tutorial, but the conclusions drawn are the same.

In addition to the previous tutorial, we decide to truncate the `ClaimNb` and the `$Exposure` in order to correct for unreasonable data entries and simplifications for the modeling part.

```
# Grouping id
distinct <- freMTPL2freq %>%
  distinct_at(vars(-c(IDpol, Exposure, ClaimNb))) %>%
  mutate(group_id = row_number())
```

```
dat <- freMTPL2freq %>%
  left_join(distinct) %>%
  mutate(ClaimNb = pmin(as.integer(ClaimNb), 4),
         VehAge = pmin(VehAge, 20),
         DrivAge = pmin(DrivAge, 90),
         BonusMalus = pmin(BonusMalus, 150),
         Density = round(log(Density), 2),
         VehGas = factor(VehGas),
         Exposure = pmin(Exposure, 1))
```

```
## Joining, by = c("Area", "VehPower", "VehAge", "DrivAge", "BonusMalus", "VehBrand", "VehGas", "Density
```

```
# Group sizes of suspected clusters
table(table(dat[, "group_id"]))
```

```
##
##      1      2      3      4      5      6      7      8      9     10     11
## 429576  84201  13940   2437    966    754    720    475    400    269    142
##     12     13     14     15     18     22
##    191      3      1      2      1      1
```

Feature pre-processing for generalized linear models

As previously mentioned, typically features x_i need pre-processing before being used for a specific model. In our Poisson GLM the regression function is modeled by a log-linear shape in the continuous feature components. From the marginal empirical frequency plots in the previous file we see that such a log-linear form is not always appropriate. We make the following choices here:

- `Area`: we choose a continuous (log-linear) feature component for $\{A, ..., F\} \mapsto \{1, ..., 6\}$
- `VehPower`: we choose a categorical feature component where we merge vehicle power groups bigger and equal to 9 (totally 6 classes)
- `VehAge`: we build 3 categorical classes $[0, 1), [1, 10], (10, \infty)$
- `DrivAge`: we build 7 categorical classes $[18, 21), [21, 26), [26, 31), [31, 41), [41, 51), [51, 71), [71, \infty)$
- `BonusMalus`: continuous log-linear feature component (we cap at value 150)
- `VehBrand`: categorical feature component (totally 11 classes)
- `VehGas`: binary feature component;
- `Density`: log-density is chosen as continuous log-linear feature component (note that we have very small volumes for small log-densities)
- `Region`: categorical feature component (totally 22 classes)

Thus, we consider 3 continuous feature components (`Area`, `BonusMalus`, `log-Density`), 1 binary feature component (`VehGas`) and 5 categorical feature components (`VehPower`, `VehAge`, `DrivAge`, `VehBrand`, `Region`). The categorical classes for `VehPower`, `VehAge` and `DrivAge` have been done based on expert opinion, only. This expert opinion has tried to find homogeneity within class labels (levels) and every class label should receive a sufficient volume (of observations). We could also make a data-driven choice by using a (marginal) regression tree for different feature components, see references in the tutorial.

```
dat2 <- dat %>% mutate(
  AreaGLM = as.integer(Area),
  VehPowerGLM = as.factor(pmin(VehPower, 9)),
  VehAgeGLM = cut(VehAge, breaks = c(-Inf, 0, 10, Inf), labels = c("1","2","3")),
  DrivAgeGLM = cut(DrivAge, breaks = c(-Inf, 20, 25, 30, 40, 50, 70, Inf), labels = c("1","2","3","4","5
  BonusMalusGLM = as.integer(pmin(BonusMalus, 150)),
  DensityGLM = as.numeric(Density),
  VehAgeGLM = relevel(VehAgeGLM, ref = "2"),
  DrivAgeGLM = relevel(DrivAgeGLM, ref = "5"),
  Region = relevel(Region, ref = "R24")
)
```

We remark that for categorical variables we use the data type factor in R. This data type automatically considers dummy coding in the corresponding R procedures. Categorical variables are initialized to one class (reference level). We typically initialize to the class with the biggest volume. This initialization is achieved by the command relevel, see above. This initialization does not influence the fitted means but provides a unique parametrization. See `?relevel` for further details.

Inspect the prepared dataset

```
knitr::kable(head(dat2))
```

IDpol	Exposure	Area	VehPower	VehAge	DrivAge	BonusMalus	VehBrand	VehGas	Density	Region	ClaimTotal	ClaimNb	group_id	AreaGLM	VehPowerGLM	VehAgeGLM	DrivAgeGLM	BonusMalusGLM	DensityGLM
1	0.10	D	5	0	55	50	B12	Regular	7.10	R82	0	0	1	4	5	1	6	50	7.10
3	0.77	D	5	0	55	50	B12	Regular	7.10	R82	0	0	1	4	5	1	6	50	7.10
5	0.75	B	6	2	52	50	B12	Diesel	3.99	R22	0	0	2	2	6	2	6	50	3.99
10	0.09	B	7	0	46	50	B12	Diesel	4.33	R72	0	0	3	2	7	1	5	50	4.33
11	0.84	B	7	0	46	50	B12	Diesel	4.33	R72	0	0	3	2	7	1	5	50	4.33
13	0.52	E	6	2	38	50	B12	Regular	8.01	R31	0	0	4	5	6	2	4	50	8.01

```
str(dat2)
```

```
## 'data.frame':    678007 obs. of  20 variables:
##  $ IDpol         : num  1 3 5 10 11 13 15 17 18 21 ...
##  $ Exposure      : num  0.1 0.77 0.75 0.09 0.84 0.52 0.45 0.27 0.71 0.15 ...
##  $ Area          : Factor w/ 6 levels "A","B","C","D",..: 4 4 2 2 2 5 5 3 3 2 ...
##  $ VehPower      : num  5 5 6 7 7 6 6 7 7 7 ...
##  $ VehAge        : num  0 0 2 0 0 2 2 0 0 0 ...
##  $ DrivAge       : num  55 55 52 46 46 38 38 33 33 41 ...
##  $ BonusMalus    : num  50 50 50 50 50 50 50 68 68 50 ...
##  $ VehBrand      : Factor w/ 11 levels "B1","B2","B3",..: 9 9 9 9 9 9 9 9 9 9 ...
##  $ VehGas        : Factor w/ 2 levels "Diesel","Regular": 2 2 1 1 1 2 2 1 1 1 ...
##  $ Density       : num  7.1 7.1 3.99 4.33 4.33 8.01 8.01 4.92 4.92 4.09 ...
##  $ Region        : Factor w/ 22 levels "R24","R11","R21",..: 18 18 4 15 15 8 8 20 20 12 ...
##  $ ClaimTotal    : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ ClaimNb       : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ group_id      : int  1 1 2 3 3 4 4 5 5 6 ...
##  $ AreaGLM       : int  4 4 2 2 2 5 5 3 3 2 ...
##  $ VehPowerGLM   : Factor w/ 6 levels "4","5","6","7",..: 2 2 3 4 4 3 3 4 4 4 ...
##  $ VehAgeGLM     : Factor w/ 3 levels "2","1","3": 2 2 1 2 2 1 1 2 2 2 ...
##  $ DrivAgeGLM    : Factor w/ 7 levels "5","1","2","3",..: 6 6 6 1 1 5 5 5 5 1 ...
##  $ BonusMalusGLM: int  50 50 50 50 50 50 50 68 68 50 ...
##  $ DensityGLM    : num  7.1 7.1 3.99 4.33 4.33 8.01 8.01 4.92 4.92 4.09 ...
```

```
summary(dat2)
```

```
##      IDpol            Exposure        Area          VehPower
##  Min.   :      1   Min.   :0.002732   A:103957   Min.   : 4.000
##  1st Qu.:1157948   1st Qu.:0.180000   B: 75459   1st Qu.: 5.000
##  Median :2272153   Median :0.490000   C:191880   Median : 6.000
##  Mean   :2621857   Mean   :0.528547   D:151590   Mean   : 6.455
##  3rd Qu.:4046278   3rd Qu.:0.990000   E:137167   3rd Qu.: 7.000
##  Max.   :6114330   Max.   :1.000000   F: 17954   Max.   :15.000
##
##      VehAge          DrivAge       BonusMalus        VehBrand
##  Min.   : 0.000   Min.   :18.0   Min.   : 50.00   B12    :166024
##  1st Qu.: 2.000   1st Qu.:34.0   1st Qu.: 50.00   B1     :162730
##  Median : 6.000   Median :44.0   Median : 50.00   B2     :159861
##  Mean   : 6.976   Mean   :45.5   Mean   : 59.76   B3     : 53395
##  3rd Qu.:11.000   3rd Qu.:55.0   3rd Qu.: 64.00   B5     : 34753
##  Max.   :20.000   Max.   :90.0   Max.   :150.00   B6     : 28548
##                                                   (Other): 72696
##      VehGas          Density           Region         ClaimTotal
##  Diesel :332136   Min.   : 0.000   R24    :160601   Min.   :      0
##  Regular:345871   1st Qu.: 4.520   R82    : 84752   1st Qu.:      0
##                   Median : 5.970   R93    : 79315   Median :      0
##                   Mean   : 5.982   R11    : 69791   Mean   :     88
##                   3rd Qu.: 7.410   R53    : 42122   3rd Qu.:      0
##                   Max.   :10.200   R52    : 38751   Max.   :4075401
##                                    (Other):202675
##     ClaimNb           group_id         AreaGLM     VehPowerGLM VehAgeGLM
##  Min.   :0.00000   Min.   :     1   Min.   :1.00   4:115343    2:434492
##  1st Qu.:0.00000   1st Qu.:149318   1st Qu.:2.00   5:124821    1: 57739
##  Median :0.00000   Median :273211   Median :3.00   6:148976    3:185776
##  Mean   :0.03891   Mean   :275320   Mean   :3.29   7:145401
##  3rd Qu.:0.00000   3rd Qu.:404072   3rd Qu.:4.00   8: 46956
##  Max.   :4.00000   Max.   :534079   Max.   :6.00   9: 96510
##
##  DrivAgeGLM BonusMalusGLM      DensityGLM
##  5:165185   Min.   : 50.00   Min.   : 0.000
##  1:  6816   1st Qu.: 50.00   1st Qu.: 4.520
##  2: 32079   Median : 50.00   Median : 5.970
##  3: 65594   Mean   : 59.76   Mean   : 5.982
##  4:170097   3rd Qu.: 64.00   3rd Qu.: 7.410
##  6:198871   Max.   :150.00   Max.   :10.200
##  7: 39365
```

Split train and test data

First, we split the dataset into train and test. Due to the potential grouping of rows in policies we can not just do a random split. For this purpose, we use the function `partition(...)` from the `splitTools` package.

```
ind <- partition(dat2[["group_id"]], p = c(train = 0.8, test = 0.2),
                 seed = seed, type = "grouped")
train <- dat2[ind$train, ]
test <- dat2[ind$test, ]
```

It describes our choices of the learning data set $\mathcal{D}$ and the test data set $\mathcal{T}$ That is, we allocate at random

80% of the policies to $\mathcal{D}$ and the remaining 20% of the policies to $\mathcal{T}$.

Usually, an 90/10 or 80/20 is used for training and test data. This is a rule-of-thumb and best practice in modeling. A good explanation can be found here, citing as follows: "There are two competing concerns: with less training data, your parameter estimates have greater variance. With less testing data, your performance statistic will have greater variance. Broadly speaking you should be concerned with dividing data such that neither variance is too high, which is more to do with the absolute number of instances in each category rather than the percentage."

```
# size of train/test
sprintf("Number of observations (train): %s", nrow(train))
```

```
## [1] "Number of observations (train): 542331"
```

```
sprintf("Number of observations (test): %s", nrow(test))
```

```
## [1] "Number of observations (test): 135676"
```

```
# Claims frequency of train/test
sprintf("Empirical frequency (train): %s", round(sum(train$ClaimNb) / sum(train$Exposure), 4))
```

```
## [1] "Empirical frequency (train): 0.0736"
```

```
sprintf("Empirical frequency (test): %s", round(sum(test$ClaimNb) / sum(test$Exposure), 4))
```

```
## [1] "Empirical frequency (test): 0.0736"
```

Store model results

As we are going to compare various models, we create a table which stores the metrics we are going to use for the comparison and the selection of the best model.

```
# initialize table to store all model results for comparison
df_cmp <- tibble(
 model = character(),
 epochs = numeric(),
 run_time = numeric(),
 parameters = numeric(),
 in_sample_loss = numeric(),
 out_sample_loss = numeric(),
 avg_freq = numeric(),
)
```

In the following chapters, we are going to fit various feed-forward neural networks for the data. At the end, we will compare the performance and quality of the several fitted models. We compare them by using the metrics defined above.

Model assumptions

Before fitting any neural network, we fit a generalized linear model (GLM) which serves as baseline model.

In the following, we will fit various claim frequency models based on a Poisson assumption, to be more precise we make the following assumptions:

Model Assumptions 1.1 (Model GLM1) *Choose feature space $\mathcal{X}$ as in (1.2) and define the regression function $\lambda : \mathcal{X} \to \mathbb{R}_+$ by*

$$\boldsymbol{x} \;\mapsto\; \log \lambda(\boldsymbol{x}) = \beta_0 + \sum_{l=1}^{q_0} \beta_l x_l \;\stackrel{\text{def.}}{=}\; \langle \boldsymbol{\beta}, \boldsymbol{x} \rangle, \tag{1.3}$$

for parameter vector $\boldsymbol{\beta} = (\beta_0, \ldots, \beta_{q_0})' \in \mathbb{R}^{q_0+1}$. Assume for $i \geq 1$

$$N_i \;\stackrel{\text{ind.}}{\sim}\; \text{Poi}\left(\lambda(\boldsymbol{x}_i) v_i\right).$$

The main problem to be solved is to find the regression function $\lambda(\cdot)$ such that it appropriately describes the data, and such that it generalizes to similar data which has not been seen, yet. Remark that the task of finding an appropriate regression function $\lambda : \mathcal{X} \to \mathbb{R}_+$ also includes the definition of the feature space $\mathcal{X}$ which typically varies over different modeling approaches.

Model 1: GLM

Definition

The defined feature components are continuous in nature, but we have been turning them into categorical ones for modeling purposes (as mentioned above). Having so much data, we can further explore these categorical feature components by trying to replace them by ordinal ones assuming an appropriate continuous functional form, still fitting into the GLM framework.

As an example we show how to bring DrivAge into a continuous functional form. We therefore modify the feature space $\mathcal{X}$ and the regression function $\lambda(\cdot)$ from (1.3). We replace the 7 categorical age classes by the following continuous function:

$$\texttt{DrivAge} \;\mapsto\; \beta_l \,\texttt{DrivAge} + \beta_{l+1} \log(\texttt{DrivAge}) + \sum_{j=2}^{4} \beta_{l+j} (\texttt{DrivAge})^j,$$

Thus, we replace the 7 categorical classes (involving 6 regression parameters from dummy coding) by the above continuous functional form having 5 regression parameters. The remaining parts of the regression function in (1.3) are kept unchanged

This model will be our baseline model with which we compare the subsequent fitted feed-forward neural networks.

Fitting

```
exec_time <- system.time(
  glm2 <- glm(ClaimNb ~ AreaGLM + VehPowerGLM + VehAgeGLM + BonusMalusGLM + VehBrand +
                        VehGas + DensityGLM + Region + DrivAge + log(DrivAge) +
                        I(DrivAge^2) + I(DrivAge^3) + I(DrivAge^4),
              data = train, offset = log(Exposure), family = poisson())
)
exec_time[1:5]
```

```
##  user.self   sys.self     elapsed user.child  sys.child
##     63.718     65.942      30.648      0.000      0.000
```

```
summary(glm2)
```

```
##
## Call:
## glm(formula = ClaimNb ~ AreaGLM + VehPowerGLM + VehAgeGLM + BonusMalusGLM +
##     VehBrand + VehGas + DensityGLM + Region + DrivAge + log(DrivAge) +
##     I(DrivAge^2) + I(DrivAge^3) + I(DrivAge^4), family = poisson(),
##     data = train, offset = log(Exposure))
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.4705  -0.3250  -0.2461  -0.1382   6.8980
##
## Coefficients:
##                  Estimate Std. Error z value Pr(>|z|)
## (Intercept)    7.653e+01  6.908e+00  11.079  < 2e-16 ***
## AreaGLM        4.681e-02  2.128e-02   2.199 0.027873 *
## VehPowerGLM5   5.782e-02  2.435e-02   2.374 0.017575 *
## VehPowerGLM6   9.069e-02  2.389e-02   3.796 0.000147 ***
## VehPowerGLM7   6.610e-02  2.377e-02   2.780 0.005428 **
## VehPowerGLM8   9.579e-02  3.381e-02   2.833 0.004605 **
## VehPowerGLM9   2.376e-01  2.655e-02   8.949  < 2e-16 ***
## VehAgeGLM1    -1.945e-02  3.419e-02  -0.569 0.569403
## VehAgeGLM3    -1.840e-01  1.633e-02 -11.272  < 2e-16 ***
## BonusMalusGLM  2.755e-02  4.127e-04  66.755  < 2e-16 ***
## VehBrandB2    -8.200e-03  1.933e-02  -0.424 0.671477
## VehBrandB3     5.923e-02  2.670e-02   2.219 0.026515 *
## VehBrandB4     5.602e-02  3.623e-02   1.546 0.122043
## VehBrandB5     8.473e-02  3.086e-02   2.746 0.006034 **
## VehBrandB6     1.326e-02  3.499e-02   0.379 0.704762
## VehBrandB10    8.286e-03  4.431e-02   0.187 0.851663
## VehBrandB11    1.860e-01  4.688e-02   3.967 7.27e-05 ***
## VehBrandB12   -2.505e-01  2.442e-02 -10.257  < 2e-16 ***
## VehBrandB13    5.417e-02  4.992e-02   1.085 0.277899
## VehBrandB14   -1.602e-01  9.794e-02  -1.636 0.101797
## VehGasRegular -1.569e-01  1.494e-02 -10.500  < 2e-16 ***
## DensityGLM     3.919e-02  1.581e-02   2.478 0.013211 *
## RegionR11     -9.825e-03  3.111e-02  -0.316 0.752112
## RegionR21      2.124e-03  1.314e-01   0.016 0.987099
## RegionR22      1.766e-01  6.414e-02   2.753 0.005906 **
## RegionR23     -4.110e-02  7.866e-02  -0.522 0.601323
## RegionR25     -3.692e-02  5.557e-02  -0.664 0.506376
## RegionR26      4.565e-02  6.128e-02   0.745 0.456283
## RegionR31      2.350e-02  4.055e-02   0.579 0.562303
## RegionR41     -1.508e-01  5.514e-02  -2.735 0.006241 **
## RegionR42      2.856e-02  1.168e-01   0.245 0.806763
## RegionR43     -1.427e-01  1.896e-01  -0.753 0.451682
## RegionR52      2.480e-02  3.201e-02   0.775 0.438538
## RegionR53      2.326e-02  2.952e-02   0.788 0.430749
## RegionR54      3.652e-02  4.265e-02   0.856 0.391850
## RegionR72      1.099e-01  3.728e-02   2.949 0.003185 **
## RegionR73     -1.697e-01  5.944e-02  -2.855 0.004304 **
## RegionR74      4.124e-01  7.951e-02   5.187 2.14e-07 ***
## RegionR82      2.234e-01  2.367e-02   9.441  < 2e-16 ***
```

```
## RegionR83      1.885e-02  9.388e-02   0.201 0.840865
## RegionR91     -2.618e-03  3.834e-02  -0.068 0.945544
## RegionR93      1.446e-01  2.669e-02   5.418 6.04e-08 ***
## RegionR94      1.518e-01  9.800e-02   1.549 0.121306
## DrivAge        3.874e+00  4.007e-01   9.670  < 2e-16 ***
## log(DrivAge)  -4.661e+01  4.231e+00 -11.018  < 2e-16 ***
## I(DrivAge^2)  -5.549e-02  6.728e-03  -8.248  < 2e-16 ***
## I(DrivAge^3)   4.399e-04  6.360e-05   6.917 4.61e-12 ***
## I(DrivAge^4)  -1.388e-06  2.421e-07  -5.733 9.85e-09 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for poisson family taken to be 1)
##
##     Null deviance: 136692  on 542330  degrees of freedom
## Residual deviance: 130634  on 542283  degrees of freedom
## AIC: 171306
##
## Number of Fisher Scoring iterations: 6
```

Exercise: One could possible reduce the number of parameters by reducing the variables included. Do so and compare the result to the currently used model.

Exercise: This glm might be improved from a modeling perspective by excluding non-significant variables.

The `summary()` function for a `glm` object provides the statistical tests of significance for every single parameter. However, with categorical variables the primary interest is to know if a categorical variables at all is significant. This can be done using the R function `drop1`, see its help file for further details. It performs a Likelihood Ratio Test (LRT) which confirms that only the p-value for AreaGLM is above 5%.

```
drop1(glm2, test = "LRT")
```

```
## Single term deletions
##
## Model:
## ClaimNb ~ AreaGLM + VehPowerGLM + VehAgeGLM + BonusMalusGLM +
##     VehBrand + VehGas + DensityGLM + Region + DrivAge + log(DrivAge) +
##     I(DrivAge^2) + I(DrivAge^3) + I(DrivAge^4)
##               Df Deviance    AIC    LRT  Pr(>Chi)
## <none>             130634 171306
## AreaGLM        1   130639 171308    4.8   0.02786 *
## VehPowerGLM    5   130721 171382   86.7 < 2.2e-16 ***
## VehAgeGLM      2   130763 171431  129.6 < 2.2e-16 ***
## BonusMalusGLM  1   134139 174809 3505.3 < 2.2e-16 ***
## VehBrand      10   130840 171492  206.6 < 2.2e-16 ***
## VehGas         1   130744 171414  110.3 < 2.2e-16 ***
## DensityGLM     1   130640 171310    6.1   0.01316 *
## Region        21   130836 171466  202.3 < 2.2e-16 ***
## DrivAge        1   130728 171397   93.9 < 2.2e-16 ***
## log(DrivAge)   1   130755 171424  120.7 < 2.2e-16 ***
## I(DrivAge^2)   1   130703 171372   68.9 < 2.2e-16 ***
## I(DrivAge^3)   1   130683 171352   48.7 2.971e-12 ***
## I(DrivAge^4)   1   130667 171337   33.6 6.767e-09 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Exercise: Consider a similar approach as DrivAge for another feature (e.g. BonusMalus)

Validation

```
# Predictions
train$fitGLM2 <- fitted(glm2)
test$fitGLM2 <- predict(glm2, newdata = test, type = "response")
dat$fitGLM2 <- predict(glm2, newdata = dat2, type = "response")
```

```
# in-sample and out-of-sample losses (in 10^(-2))
sprintf("100 x Poisson deviance GLM (train): %s", PoissonDeviance(train$fitGLM2, train$ClaimNb))
```

```
## [1] "100 x Poisson deviance GLM (train): 24.0874788981516"
```

```
sprintf("100 x Poisson deviance GLM (test): %s", PoissonDeviance(test$fitGLM2, test$ClaimNb))
```

```
## [1] "100 x Poisson deviance GLM (test): 24.1666108887936"
```

```
# Overall estimated frequency
sprintf("average frequency (test): %s", round(sum(test$fitGLM2) / sum(test$Exposure), 4))
```

```
## [1] "average frequency (test): 0.0737"
```

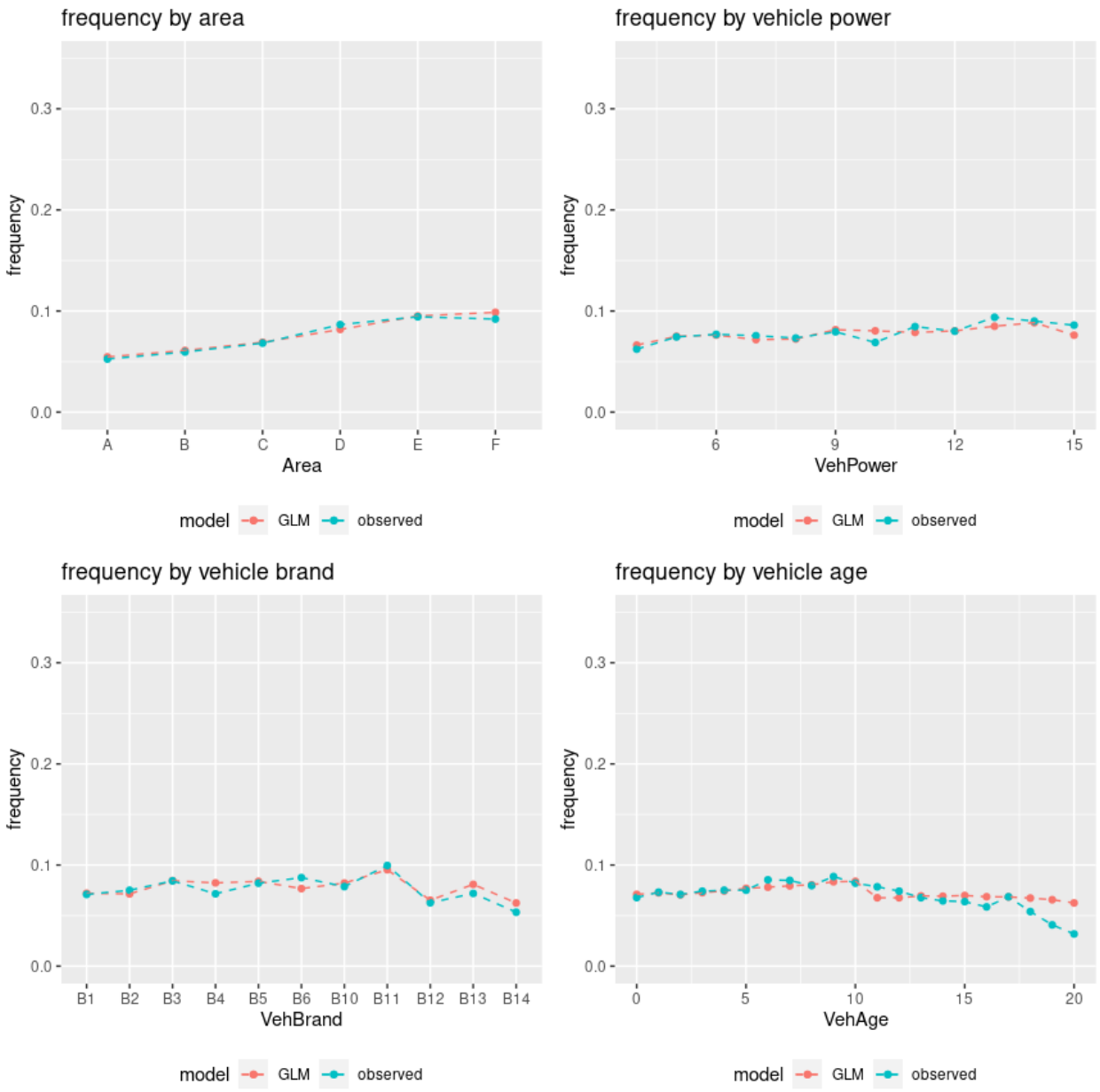
```
df_cmp %<>% bind_rows(
  data.frame(model = "M1: GLM", epochs = NA, run_time = round(exec_time[[3]], 0), parameters = length(c
             in_sample_loss = round(PoissonDeviance(train$fitGLM2, as.vector(unlist(train$ClaimNb))), 4
             out_sample_loss = round(PoissonDeviance(test$fitGLM2, as.vector(unlist(test$ClaimNb))), 4)
             avg_freq = round(sum(test$fitGLM2) / sum(test$Exposure), 4))
)
knitr::kable(df_cmp)
```

model	epochs	run_time	parameters	in_sample_loss	out_sample_loss	avg_freq
M1: GLM	NA	31	48	24.0875	24.1666	0.0737

Calibration

In addition to fitting and validating the model with a few metrics, it is important to check if the model is well calibrated across the feature space. E.g. it could be that the overall fit of a model is good, but that there are areas where the model under- and overestimates the claim frequencies. It is the goal of the subsequent calibration plots to ensure the proper fit along the whole feature space.

```
# Area
p1 <- plot_freq(test, "Area", "frequency by area", "GLM", "fitGLM2")
# VehPower
p2 <- plot_freq(test, "VehPower", "frequency by vehicle power", "GLM", "fitGLM2")
# VehBrand
p3 <- plot_freq(test, "VehBrand", "frequency by vehicle brand", "GLM", "fitGLM2")
# VehAge
p4 <- plot_freq(test, "VehAge", "frequency by vehicle age", "GLM", "fitGLM2")

grid.arrange(p1, p2, p3, p4)
```



Based on the charts, no issues are detected and the model seems to be well calibrated.

Exercise: Perform the calibration with other variables not yet in the charts above.

Pre-processing Neural Networks

Introduction

In this chapter, we explain how the data need to be pre-processed to be used in neural networks. It can not be processed in the same way as shown above for GLMs. Further details can be found in this tutorial on SSRN, chapter 2.

We are going to highlight a few important points in data pre-processing that are necessary for a successful application of networks.

In network modeling the choice of the scale of the feature components may substantially influence the fitting procedure of the predictive model. Therefore, data pre-processing requires careful consideration. We treat unordered categorical (nominal) feature components and continuous (or ordinal) feature components 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.

Unordered (nominal) categorical feature components

We need to transform (nominal) categorical feature components to numerical values. The most commonly used transformations are the so-called **dummy coding** and the **one-hot encoding**. Both methods construct binary representations for categorical labels. For dummy coding one label is chosen as reference level. Dummy coding then uses binary variables to indicate which label a particular policy possesses if it differs from the reference level. In our example we have two unordered categorical feature components, namely VehBrand and Region. We use VehBrand as illustration. It has 11 different labels $\{B_1, B_{10}, B_{11}, B_{12}, B_{13}, B_{14}, B_2, B_3, B_4, B_5, B_6\}$. We choose B_1 as reference label. Dummy coding then provides the coding scheme below (left). We observe that the 11 labels are replaced by 10-dimensional feature vectors $\{0,1\}^{10}$, with components summing up to either 0 or 1.

label	feature components $x^\star \in \{0,1\}^{10}$									
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

label	feature components $x^\star \in \{0,1\}^{11}$										
B1	1	0	0	0	0	0	0	0	0	0	0
B10	0	1	0	0	0	0	0	0	0	0	0
B11	0	0	1	0	0	0	0	0	0	0	0
B12	0	0	0	1	0	0	0	0	0	0	0
B13	0	0	0	0	1	0	0	0	0	0	0
B14	0	0	0	0	0	1	0	0	0	0	0
B2	0	0	0	0	0	0	1	0	0	0	0
B3	0	0	0	0	0	0	0	1	0	0	0
B4	0	0	0	0	0	0	0	0	1	0	0
B5	0	0	0	0	0	0	0	0	0	1	0
B6	0	0	0	0	0	0	0	0	0	0	1

In contrast to dummy coding, one-hot encoding does not choose a reference level, but uses an indicator for each label. In this way the 11 labels of VehBrand are replaced by the 11 unit vectors. The main difference between dummy coding and one-hot encoding is that the former leads to full rank design matrices, whereas the latter does not. This implies that under one-hot encoding there are identifiability issues in parametrizations. In network modeling identifiability is less important because we typically work in over-parametrized nonconvex optimization problems (with multiple equally good models/parametrizations); on the other hand, identifiability in GLMs is an important feature because one typically tries to solve a convex optimization problem, where the full rank property is important to efficiently and the (unique) solution.

Remark that other coding schemes could be used for categorical feature components such as Helmert's contrast coding. In classical GLMs the choice of the coding scheme typically does not influence the prediction, however, 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, and it provides sparsity in input features. Moreover, the Euclidean distance between any two labels in the one-hot encoding scheme is that same. From natural language processing (NLP) we have learned that

there are more efficient ways of representing categorical feature components, namely, by embedding them into lower-dimensional spaces so that proximity in these spaces has a useful meaning in the regression task. In networks this can be achieved by so-called embedding layers. In the context of our French MTPL example we refer to our next notebook.

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 at (also known as vanishing gradient problem). Often, one uses $[-1,+1]$ as the common scale because the/our choice of activation function is focused to that scale.

A popular transformation is the so-called MinMaxScaler. For this transformation we fix each continuous feature component of 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 ml or the maximum Ml is 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.

Remark that if we have outliers, the above transformations may lead to very concentrated transformed feature components x_l^*, $i = 1,...,n$, 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) across the real line.

Binary feature components

We observe that binary feature components (e.g. gender) are often embedded in the ML literature into a higher-dimensional space. However, we are of the opinion that this does not make sense. Hence, we suggest to set binary categorical feature components to $\pm 1/2$.

Summary

As a rule of thumb one could formulate it as follows:

- continuous features $\Rightarrow$ scale to $[-1,+1]$ (if no outliers)
- binary features $\Rightarrow$ set to $\{-1/2,+1/2\}$
- categorical features:
  - make them numerical $\Rightarrow$ scale to $[-1,+1]$
  - One-hot encoding $\Rightarrow$ no scaling
  - dummy encoding $\Rightarrow$ no scaling
  - embedding $\Rightarrow$ made numerical and no scaling

Pre-processing functions

In our example we use dummy coding for the feature components VehBrand and Region. We use the MinMaxScaler for Area (after transforming $\{A,...,F\} \mapsto \{1,...,6\}$), 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.

Below the corresponding pre-processing functions:

```
# MinMax scaler
preprocess_minmax <- function(varData) {
  X <- as.numeric(varData)
  2 * (X - min(X)) / (max(X) - min(X)) - 1
}

# Dummy coding
preprocess_catdummy <- function(data, varName, prefix) {
  varData <- data[[varName]]
  X <- as.integer(varData)
  n0 <- length(unique(X))
  n1 <- 2:n0
  addCols <- purrr::map(n1, function(x, y) {as.integer(y == x)}, y = X) %>%
    rlang::set_names(paste0(prefix, n1))
  cbind(data, addCols)
}

# Feature pre-processing using MinMax Scaler and Dummy Coding
preprocess_features <- function(data) {
  data %>%
    mutate_at(
      c(AreaX = "Area", VehPowerX = "VehPower", VehAgeX = "VehAge",
        DrivAgeX = "DrivAge", BonusMalusX = "BonusMalus", DensityX = "Density"),
      preprocess_minmax
    ) %>%
    mutate(
      VehGasX = as.integer(VehGas) - 1.5
    ) %>%
    preprocess_catdummy("VehBrand", "Br") %>%
    preprocess_catdummy("Region", "R")
}
```

Execute pre-processing

```
dat2 <- preprocess_features(dat)
```

Inspect the pre-processed data

```
knitr::kable(head(dat2))
```

ID	Exp	Area	VehPower	VehAge	DrivAge	BonusMalus	VehBrand	VehGas	Density	Region	ClaimNb	AreaX	VehPowerX	VehAgeX	DrivAgeX	BonusMalusX	DensityX	VehGasX	Br1	Br2	Br3	Br4	Br5	Br6	Br7	Br8	Br9	Br10	Br11	Br12	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20	R21	R22	R23	R24
1	0.10	D	5	0	55	50	B12	Regular	1823	0	1	0.00	0.53	0.25	0.02	0.77	0.38	0.05	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0.77	D	5	0	55	50	B12	Regular	1823	0	1	0.04	0.85	0.25	0.02	0.77	0.38	0.05	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0.75	B	6	2	52	50	B12	Diesel	3317	0	2	0.04	0.23	0.53	0.60	0.63	0.38	0.55	0.55	0.26	0.76	0.47	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

IDpol	Exposure	Area	VehPower	VehAge	DrivAge	BonusMalus	VehBrand	VehGas	Density	Region	ClaimTotal	ClaimNb	group_id	fitGLM2	AreaX	VehPowerX	VehAgeX	DrivAgeX	BonusMalusX	DensityX	VehGasX	Br2	Br3	Br4	Br5	Br6	Br7	Br8	Br9	Br10	Br11	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11
10	0.08	B	7	0	46	50	B12	Diesel	3R72	2	0	3	0.00458	57	-	-	-	-	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
														0.60	45	54	42	22	22	29	804																				
11	0.84	B	7	0	46	50	B12	Diesel	3R72	2	0	3	0.04279	97	-	-	-	-	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
														0.60	45	54	42	22	22	29	804																				
13	0.52	E	6	2	38	50	B12	Regular	8R31	0	0	4	0.02459	61	-	-	0.57	75	88	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
														0.63	86	36	44	44	44	44	444																				

```
str(dat2)
```

```
## 'data.frame':    678007 obs. of  53 variables:
##  $ IDpol      : num  1 3 5 10 11 13 15 17 18 21 ...
##  $ Exposure   : num  0.1 0.77 0.75 0.09 0.84 0.52 0.45 0.27 0.71 0.15 ...
##  $ Area       : Factor w/ 6 levels "A","B","C","D",..: 4 4 2 2 2 5 5 3 3 2 ...
##  $ VehPower   : num  5 5 6 7 7 6 6 7 7 7 ...
##  $ VehAge     : num  0 0 2 0 0 2 2 0 0 0 ...
##  $ DrivAge    : num  55 55 52 46 46 38 38 33 33 41 ...
##  $ BonusMalus : num  50 50 50 50 50 50 50 68 68 50 ...
##  $ VehBrand   : Factor w/ 11 levels "B1","B2","B3",..: 9 9 9 9 9 9 9 9 9 9 ...
##  $ VehGas     : Factor w/ 2 levels "Diesel","Regular": 2 2 1 1 1 2 2 1 1 1 ...
##  $ Density    : num  7.1 7.1 3.99 4.33 4.33 8.01 8.01 4.92 4.92 4.09 ...
##  $ Region     : Factor w/ 22 levels "R11","R21","R22",..: 18 18 3 15 15 8 8 20 20 12 ...
##  $ ClaimTotal : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ ClaimNb    : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ group_id   : int  1 1 2 3 3 4 4 5 5 6 ...
##  $ fitGLM2    : num  0.00583 0.04486 0.04233 0.00459 0.0428 ...
##  $ AreaX      : num  0.2 0.2 -0.6 -0.6 -0.6 0.6 0.6 -0.2 -0.2 -0.6 ...
##  $ VehPowerX  : num  -0.818 -0.818 -0.636 -0.455 -0.455 ...
##  $ VehAgeX    : num  -1 -1 -0.8 -1 -1 -0.8 -0.8 -1 -1 -1 ...
##  $ DrivAgeX   : num  0.0278 0.0278 -0.0556 -0.2222 -0.2222 ...
##  $ BonusMalusX: num  -1 -1 -1 -1 -1 -1 -1 -0.64 -0.64 -1 ...
##  $ DensityX   : num  0.392 0.392 -0.218 -0.151 -0.151 ...
##  $ VehGasX    : num  0.5 0.5 -0.5 -0.5 -0.5 0.5 0.5 -0.5 -0.5 -0.5 ...
##  $ Br2        : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ Br3        : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ Br4        : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ Br5        : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ Br6        : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ Br7        : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ Br8        : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ Br9        : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ Br10       : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ Br11       : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ R2         : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ R3         : int  0 0 1 0 0 0 0 0 0 0 ...
##  $ R4         : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ R5         : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ R6         : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ R7         : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ R8         : int  0 0 0 0 0 1 1 0 0 0 ...
##  $ R9         : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ R10        : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ R11        : int  0 0 0 0 0 0 0 0 0 0 ...
```

```
##  $ R12        : int  0 0 0 0 0 0 0 0 0 1 ...
##  $ R13        : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ R14        : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ R15        : int  0 0 0 1 1 0 0 0 0 0 ...
##  $ R16        : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ R17        : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ R18        : int  1 1 0 0 0 0 0 0 0 0 ...
##  $ R19        : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ R20        : int  0 0 0 0 0 0 0 1 1 0 ...
##  $ R21        : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ R22        : int  0 0 0 0 0 0 0 0 0 0 ...
```

```
summary(dat2)
```

```
##      IDpol            Exposure        Area          VehPower
##  Min.   :      1   Min.   :0.002732   A:103957   Min.   : 4.000
##  1st Qu.:1157948   1st Qu.:0.180000   B: 75459   1st Qu.: 5.000
##  Median :2272153   Median :0.490000   C:191880   Median : 6.000
##  Mean   :2621857   Mean   :0.528547   D:151590   Mean   : 6.455
##  3rd Qu.:4046278   3rd Qu.:0.990000   E:137167   3rd Qu.: 7.000
##  Max.   :6114330   Max.   :1.000000   F: 17954   Max.   :15.000
##
##      VehAge          DrivAge       BonusMalus        VehBrand
##  Min.   : 0.000   Min.   :18.0   Min.   : 50.00   B12    :166024
##  1st Qu.: 2.000   1st Qu.:34.0   1st Qu.: 50.00   B1     :162730
##  Median : 6.000   Median :44.0   Median : 50.00   B2     :159861
##  Mean   : 6.976   Mean   :45.5   Mean   : 59.76   B3     : 53395
##  3rd Qu.:11.000   3rd Qu.:55.0   3rd Qu.: 64.00   B5     : 34753
##  Max.   :20.000   Max.   :90.0   Max.   :150.00   B6     : 28548
##                                                   (Other): 72696
##      VehGas          Density            Region         ClaimTotal
##  Diesel :332136   Min.   : 0.000   R24    :160601   Min.   :      0
##  Regular:345871   1st Qu.: 4.520   R82    : 84752   1st Qu.:      0
##                   Median : 5.970   R93    : 79315   Median :      0
##                   Mean   : 5.982   R11    : 69791   Mean   :     88
##                   3rd Qu.: 7.410   R53    : 42122   3rd Qu.:      0
##                   Max.   :10.200   R52    : 38751   Max.   :4075401
##                                    (Other):202675
##     ClaimNb           group_id         fitGLM2              AreaX
##  Min.   :0.00000   Min.   :     1   Min.   :0.0000632   Min.   :-1.00000
##  1st Qu.:0.00000   1st Qu.:149318   1st Qu.:0.0118361   1st Qu.:-0.60000
##  Median :0.00000   Median :273211   Median :0.0329438   Median :-0.20000
##  Mean   :0.03891   Mean   :275320   Mean   :0.0389177   Mean   :-0.08412
##  3rd Qu.:0.00000   3rd Qu.:404072   3rd Qu.:0.0545373   3rd Qu.: 0.20000
##  Max.   :4.00000   Max.   :534079   Max.   :2.0223469   Max.   : 1.00000
##
##    VehPowerX          VehAgeX           DrivAgeX         BonusMalusX
##  Min.   :-1.0000   Min.   :-1.0000   Min.   :-1.00000   Min.   :-1.0000
##  1st Qu.:-0.8182   1st Qu.:-0.8000   1st Qu.:-0.55556   1st Qu.:-1.0000
##  Median :-0.6364   Median :-0.4000   Median :-0.27778   Median :-1.0000
##  Mean   :-0.5537   Mean   :-0.3024   Mean   :-0.23620   Mean   :-0.8049
##  3rd Qu.:-0.4545   3rd Qu.: 0.1000   3rd Qu.: 0.02778   3rd Qu.:-0.7200
##  Max.   : 1.0000   Max.   : 1.0000   Max.   : 1.00000   Max.   : 1.0000
##
##     DensityX          VehGasX              Br2              Br3
```

```
##  Min.   :-1.0000   Min.   :-0.50000   Min.   :0.0000   Min.   :0.00000
##  1st Qu.:-0.1137   1st Qu.:-0.50000   1st Qu.:0.0000   1st Qu.:0.00000
##  Median : 0.1706   Median : 0.50000   Median :0.0000   Median :0.00000
##  Mean   : 0.1729   Mean   : 0.01013   Mean   :0.2358   Mean   :0.07875
##  3rd Qu.: 0.4529   3rd Qu.: 0.50000   3rd Qu.:0.0000   3rd Qu.:0.00000
##  Max.   : 1.0000   Max.   : 0.50000   Max.   :1.0000   Max.   :1.00000
##
##       Br4               Br5               Br6               Br7
##  Min.   :0.00000   Min.   :0.00000   Min.   :0.00000   Min.   :0.00000
##  1st Qu.:0.00000   1st Qu.:0.00000   1st Qu.:0.00000   1st Qu.:0.00000
##  Median :0.00000   Median :0.00000   Median :0.00000   Median :0.00000
##  Mean   :0.03714   Mean   :0.05126   Mean   :0.04211   Mean   :0.02612
##  3rd Qu.:0.00000   3rd Qu.:0.00000   3rd Qu.:0.00000   3rd Qu.:0.00000
##  Max.   :1.00000   Max.   :1.00000   Max.   :1.00000   Max.   :1.00000
##
##       Br8               Br9               Br10              Br11
##  Min.   :0.00000   Min.   :0.0000   Min.   :0.00000   Min.   :0.000000
##  1st Qu.:0.00000   1st Qu.:0.0000   1st Qu.:0.00000   1st Qu.:0.000000
##  Median :0.00000   Median :0.0000   Median :0.00000   Median :0.000000
##  Mean   :0.02004   Mean   :0.2449   Mean   :0.01796   Mean   :0.005969
##  3rd Qu.:0.00000   3rd Qu.:0.0000   3rd Qu.:0.00000   3rd Qu.:0.000000
##  Max.   :1.00000   Max.   :1.0000   Max.   :1.00000   Max.   :1.000000
##
##        R2                R3                R4                R5
##  Min.   :0.000000   Min.   :0.00000   Min.   :0.00000   Min.   :0.0000
##  1st Qu.:0.000000   1st Qu.:0.00000   1st Qu.:0.00000   1st Qu.:0.0000
##  Median :0.000000   Median :0.00000   Median :0.00000   Median :0.0000
##  Mean   :0.004463   Mean   :0.01179   Mean   :0.01296   Mean   :0.2369
##  3rd Qu.:0.000000   3rd Qu.:0.00000   3rd Qu.:0.00000   3rd Qu.:0.0000
##  Max.   :1.000000   Max.   :1.00000   Max.   :1.00000   Max.   :1.0000
##
##        R6                R7                R8                R9
##  Min.   :0.00000   Min.   :0.00000   Min.   :0.00000   Min.   :0.00000
##  1st Qu.:0.00000   1st Qu.:0.00000   1st Qu.:0.00000   1st Qu.:0.00000
##  Median :0.00000   Median :0.00000   Median :0.00000   Median :0.00000
##  Mean   :0.01607   Mean   :0.01547   Mean   :0.04024   Mean   :0.01916
##  3rd Qu.:0.00000   3rd Qu.:0.00000   3rd Qu.:0.00000   3rd Qu.:0.00000
##  Max.   :1.00000   Max.   :1.00000   Max.   :1.00000   Max.   :1.00000
##
##       R10               R11               R12               R13
##  Min.   :0.000000   Min.   :0.000000   Min.   :0.00000   Min.   :0.00000
##  1st Qu.:0.000000   1st Qu.:0.000000   1st Qu.:0.00000   1st Qu.:0.00000
##  Median :0.000000   Median :0.000000   Median :0.00000   Median :0.00000
##  Mean   :0.003245   Mean   :0.001956   Mean   :0.05715   Mean   :0.06213
##  3rd Qu.:0.000000   3rd Qu.:0.000000   3rd Qu.:0.00000   3rd Qu.:0.00000
##  Max.   :1.000000   Max.   :1.000000   Max.   :1.00000   Max.   :1.00000
##
##       R14               R15               R16               R17
##  Min.   :0.00000   Min.   :0.00000   Min.   :0.00000   Min.   :0.000000
##  1st Qu.:0.00000   1st Qu.:0.00000   1st Qu.:0.00000   1st Qu.:0.000000
##  Median :0.00000   Median :0.00000   Median :0.00000   Median :0.000000
##  Mean   :0.02809   Mean   :0.04621   Mean   :0.02528   Mean   :0.006736
##  3rd Qu.:0.00000   3rd Qu.:0.00000   3rd Qu.:0.00000   3rd Qu.:0.000000
##  Max.   :1.00000   Max.   :1.00000   Max.   :1.00000   Max.   :1.000000
```

```
##
##       R18              R19                 R20              R21
##  Min.   :0.000   Min.   :0.000000   Min.   :0.0000   Min.   :0.000
##  1st Qu.:0.000   1st Qu.:0.000000   1st Qu.:0.0000   1st Qu.:0.000
##  Median :0.000   Median :0.000000   Median :0.0000   Median :0.000
##  Mean   :0.125   Mean   :0.007798   Mean   :0.0528   Mean   :0.117
##  3rd Qu.:0.000   3rd Qu.:0.000000   3rd Qu.:0.0000   3rd Qu.:0.000
##  Max.   :1.000   Max.   :1.000000   Max.   :1.0000   Max.   :1.000
##
##       R22
##  Min.   :0.000000
##  1st Qu.:0.000000
##  Median :0.000000
##  Mean   :0.006661
##  3rd Qu.:0.000000
##  Max.   :1.000000
##
```

Split train and test data

First, we split the dataset into train and test. Due to the potential grouping of rows in policies we can not just do a random split. For this purpose, we use the function `partition(...)` from the `splitTools` package.

```
ind <- partition(dat2[["group_id"]], p = c(train = 0.8, test = 0.2),
                 seed = seed, type = "grouped")
train <- dat2[ind$train, ]
test <- dat2[ind$test, ]
```

```
# size of train/test
sprintf("Number of observations (train): %s", nrow(train))
```

```
## [1] "Number of observations (train): 542331"
```

```
sprintf("Number of observations (test): %s", nrow(test))
```

```
## [1] "Number of observations (test): 135676"
```

```
# Claims frequency of train/test
sprintf("Empirical frequency (train): %s", round(sum(train$ClaimNb) / sum(train$Exposure), 4))
```

```
## [1] "Empirical frequency (train): 0.0736"
```

```
sprintf("Empirical frequency (test): %s", round(sum(test$ClaimNb) / sum(test$Exposure), 4))
```

```
## [1] "Empirical frequency (test): 0.0736"
```

Common neural network specifications

In this section, we define objects and parameters which are used for all subsequent neural networks considered, independent of their network structure.

We need to define which components in the pre-processed dataset `dat2` are used as input features. As we have added the pre-processed features appropriate for the neural networks to the original features, we only must use the relevant ones.

```
# select the feature space
col_start <- ncol(dat) + 1
```

```
col_end <- ncol(dat2)
features <- c(col_start:col_end)  # select features, be careful if pre-processing changes
print(colnames(train[, features]))
```

```
##  [1] "AreaX"       "VehPowerX"   "VehAgeX"     "DrivAgeX"    "BonusMalusX"
##  [6] "DensityX"    "VehGasX"     "Br2"         "Br3"         "Br4"
## [11] "Br5"         "Br6"         "Br7"         "Br8"         "Br9"
## [16] "Br10"        "Br11"        "R2"          "R3"          "R4"
## [21] "R5"          "R6"          "R7"          "R8"          "R9"
## [26] "R10"         "R11"         "R12"         "R13"         "R14"
## [31] "R15"         "R16"         "R17"         "R18"         "R19"
## [36] "R20"         "R21"         "R22"
```

The input to keras requires the train and testing data to be of matrix format, including all features used in the matrix and already correctly pre-processed.

```
# feature matrix
Xtrain <- as.matrix(train[, features])  # design matrix training sample
Xtest <- as.matrix(test[, features])    # design matrix test sample
```

Designing neural networks

The choice of a particular network architecture and its calibration involve many steps. In each of these steps the modeler has to make certain decisions, and it may require that each of these decisions is revised several times in order to get the best (or more modestly a good) predictive model.

In this section we provide a short explanation on the ones explicitly used below. We refer to this tutorial on SSRN and further literature (provided below in the last chapter) to learn more about these hyperparameters and the way to choose the right architecture.

The tutorial covers the choice of a particular network architecture and its calibration. In each of these steps the modeler has to make certain decisions, and it may require that each of these decisions is revised several times in order to get the best (or more modestly a good) predictive model. The choices involve:

(a) data cleaning and 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.

These choices correspond to the modeling cycle that is typically performed in statistical applications, the final validation step is not mentioned above.

Gradient descent methods

There are several optimizers available to find the solution to a neural networks, a short description as follows:

- The stochastic gradient descent method, called 'sgd', can be fine-tuned for the speed of convergence by using optimal learning rates, momentum-based improvements, the Nesterov acceleration and optimal batches. 'stochastic' gradient means that in contrast to (steepest) gradient descent, we explore locally optimally steps on random sub-samples
- '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 modi
ed 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 '2-norm;
- 'adamax' considers optimal learning rates as 'adam' but based on the l_1-norm;
- 'nadam' is a Nesterov accelerated version of 'adam'.

In the tutorial on SSRN, chapter 4 the performance of the various optimizers are analyzed. The study shows that `nadam` is a good candidate to be used.

```
# available optimizers for keras
# https://keras.io/optimizers/
optimizers <- c('sgd', 'adagrad', 'adadelta', 'rmsprop', 'adam', 'adamax', 'nadam')
```

Epochs and batches

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 Gradient Descent Method (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. 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.

Concretely, for the maximal batch size n we can do exactly one GDM step in one epoch, for batch size k we can do n/k GDM steps in one epoch. For the maximal batch size we need to calculate the gradient on the entire data $\mathcal{D}$. The latter is, of course, much faster but on the other hand we need to calculate n/k gradients to run through the entire data (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 happens especially if the chosen batch size is small and the expected frequency is low (class imbalance problem).

Initialization

A simple way to bring the initial network onto the right price level is to embed the homogeneous model into the neural network. This can be achieved by setting the output weights of the neurons equal to zero, and by initializing the output intercept to the homogeneous model. This is obtained subsequently by the code part defining the `weights`.

```
# homogeneous model (train)
lambda_hom <- sum(train$ClaimNb) / sum(train$Exposure)
```

Activation function

Next we discuss the choice of activation function $\phi(\cdot)$ Typical choices of activation functions are:

$\phi : \mathbb{R} \to \mathbb{R}$ 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

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

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 are in $(-1, +1)$, which is the domain of the main activation of the neurons in the next layer.

The step function activation is useful for theoretical considerations. 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 responses: 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.

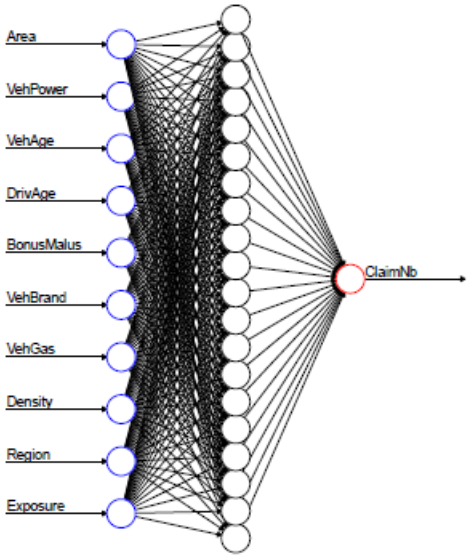
We remark that the ReLU activation function often leads to sparsity in deep network activations because some neurons remain unactivated for the entire input. Such an effect may be wanted because it reduces the

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. Moreover, ReLU may lead to arbitrarily large activations in neurons because it is an unbounded activation function, this may be an unwanted effect because it may need re-scaling of activations to the main domain around the origin.

Model 2: Shallow plain vanilla neural network (shNN)

Let us first fit the most simple feed-forward neural network, a so called shallow plain vanilla neural network. Before fitting and specifying a neural network, we highly recommend to draw it. This helps in using the `keras` function.

We choose a network with $K = 1$ **one hidden layer** and $q_1 = 20$ **neurons**, which looks as follows:



The dimension of the input layer is defined by the selected input feature dimension (see above).

Definition

Below we define the network parameters.

```
# define network and load pre-specified weights
q0 <- length(features)                  # dimension of features
q1 <- 20                                # number of hidden neurons in hidden layer

sprintf("Neural network with K=1 hidden layer")
```

```
## [1] "Neural network with K=1 hidden layer"
```

```
sprintf("Input feature dimension: q0 = %s", q0)
```

```
## [1] "Input feature dimension: q0 = 38"
```

```
sprintf("Number of hidden neurons: q1 = %s", q1)
```

```
## [1] "Number of hidden neurons: q1 = 20"
```

```
sprintf("Output dimension: %s", 1)
```

```
## [1] "Output dimension: 1"
```

Below we define the feed-forward shallow network with q_1 hidden neurons and hyperbolic tangent activation function, the output has exponential activation function due to the Poisson GLM.

The exposure is included as an offset, and we use the homogeneous model for initializing the output.

In this notebook we do not provide further details on the layers used and the structure, we refer to other notebooks or the references at the end of this notebook. A good reference is the `keras` cheat sheet.

```
Design  <- layer_input(shape = c(q0), dtype = 'float32', name = 'Design')
LogVol  <- layer_input(shape = c(1), dtype = 'float32', name = 'LogVol')

Network <- Design %>%
  layer_dense(units = q1, activation = 'tanh', name = 'layer1') %>%
  layer_dense(units = 1, activation = 'linear', name = 'Network',
              weights = list(array(0, dim = c(q1, 1)), array(log(lambda_hom), dim = c(1))))

Response <- list(Network, LogVol) %>%
  layer_add(name = 'Add') %>%
  layer_dense(units = 1, activation = k_exp, name = 'Response', trainable = FALSE,
              weights = list(array(1, dim = c(1, 1)), array(0, dim = c(1))))

model_sh <- keras_model(inputs = c(Design, LogVol), outputs = c(Response))
```

Compilation

Let us compile the model, using the Poisson deviance loss function as objective function, and nadam as the optimizer, and we provide a summary of the network structure.

For further details, we refer to the help file of `compile` here.

```
model_sh %>% compile(
  loss = 'poisson',
  optimizer = optimizers[7]
)

summary(model_sh)
```

```
## Model: "model"
## ____________________________________________________________________________
## Layer (type)              Output Shape      Param #  Connected to
## ============================================================================
## Design (InputLayer)       [(None, 38)]      0
## ____________________________________________________________________________
## layer1 (Dense)            (None, 20)        780      Design[0][0]
## ____________________________________________________________________________
## Network (Dense)           (None, 1)         21       layer1[0][0]
## ____________________________________________________________________________
## LogVol (InputLayer)       [(None, 1)]       0
## ____________________________________________________________________________
## Add (Add)                 (None, 1)         0        Network[0][0]
##                                                      LogVol[0][0]
## ____________________________________________________________________________
## Response (Dense)          (None, 1)         2        Add[0][0]
## ============================================================================
## Total params: 803
## Trainable params: 801
## Non-trainable params: 2
```

```
## _________________________________________________________________________
```

This summary is crucial for a good understanding of the fitted model. It contains the total number of parameters and shows what the exposure is included as an offset (without training the corresponding weight).

Fitting

For fitting a keras model and its arguments, we refer to the help of `fit` here. There you find details about the `validation_split` and the `verbose` argument.

If validation_split>0, then the training set is further subdivided into a new training and a validation set. The new training set is used for fitting the model and the validation set is used for (out-of-sample) validation. We emphasize that the validation set is chosen disjointly from the test data, as this latter data may still be used later for the choice of the optimal model (if, for instance, we need to decide between several networks).

With validation_split=0.2 we split the learning data 8:2 into training set and validation set. We fit the network on the training set and we out-of-sample validate it on the validation set.

```
# set hyperparameters
epochs <- 300
batch_size <- 10000
validation_split <- 0.2  # set to >0 to see train/validation loss in plot(fit)
verbose <- 1
```

```
# expected run-time on Renku 8GB environment around 70 seconds
exec_time <- system.time(
  fit <- model_sh %>% fit(
    list(Xtrain, as.matrix(log(train$Exposure))), as.matrix(train$ClaimNb),
    epochs = epochs,
    batch_size = batch_size,
    validation_split = validation_split,
    verbose = verbose
  )
)
exec_time[1:5]
```

```
##  user.self   sys.self     elapsed user.child  sys.child
##    312.604     39.495     249.034       0.000       0.000
```

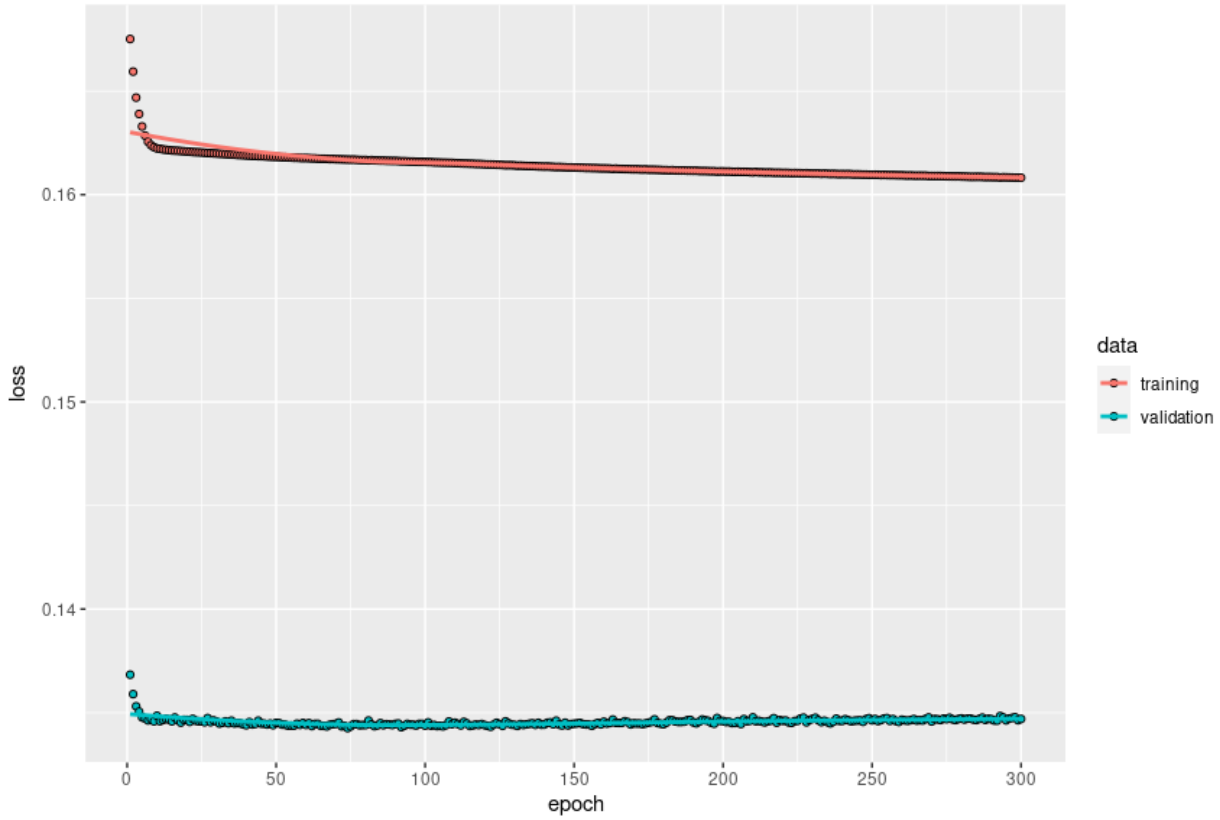
Let us illustrate the decrease of loss during the gradient descent algorithm. We provide two charts below

- First one: Depending on the argument validation_split you see one curve or two curves (training and validation).
- Second one: only shown if validation_split > 0

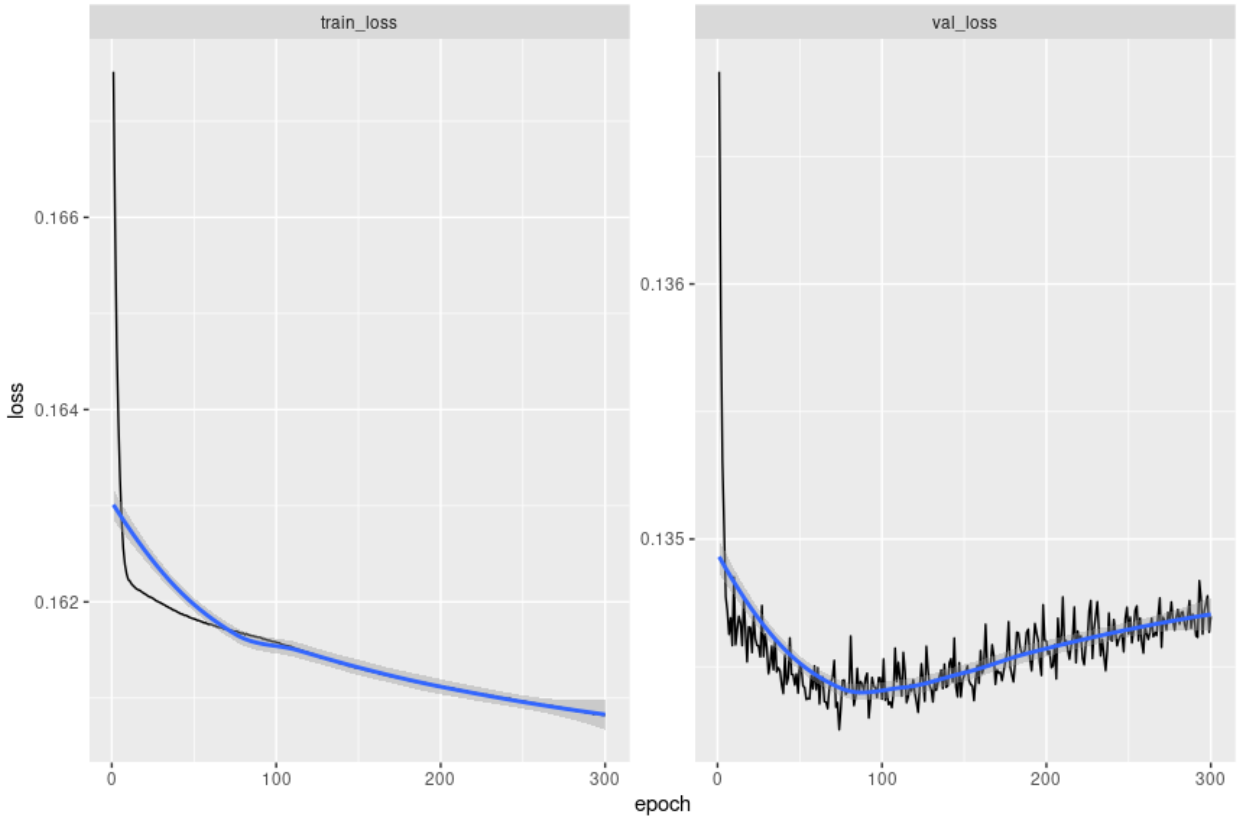
The plots help to determine the optimal number of epochs.

```
plot(fit)
```

```
## `geom_smooth()` using formula 'y ~ x'
```




```
plot_loss(x = fit[[2]])
```

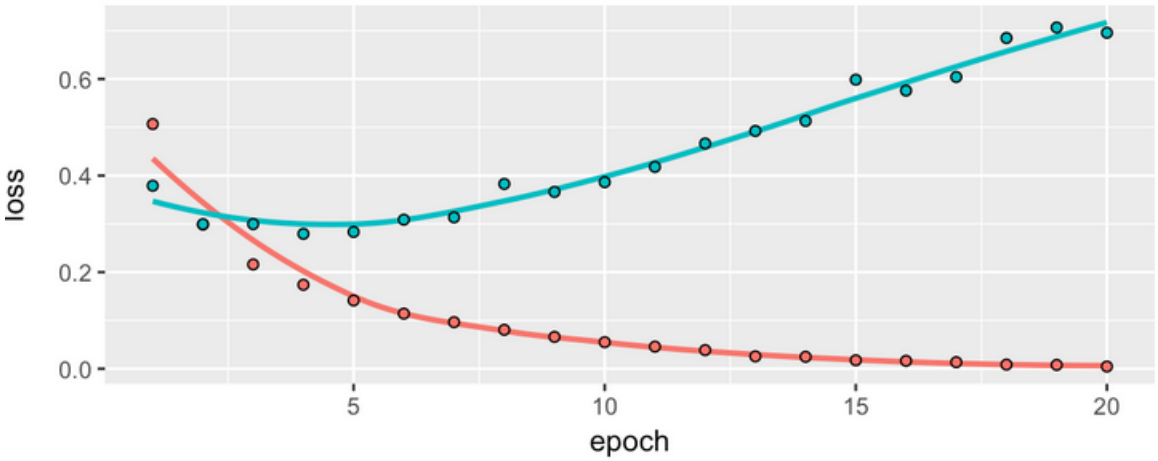


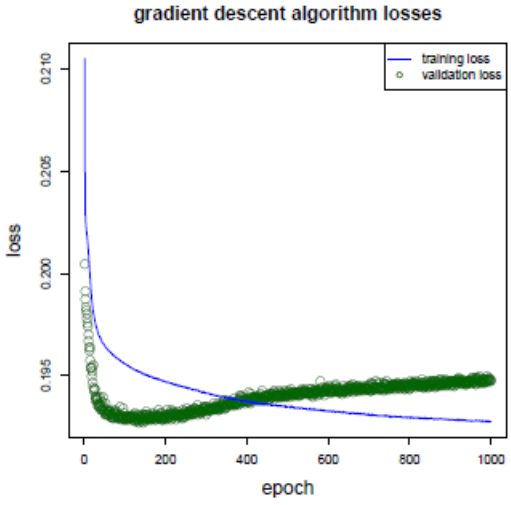
You can see the following on the charts:

- `validation_split` $= 0$: Decrease of the loss during the algorithm for the whole training data. You should expect to see a steady decrease and leveling off at some time. The more epochs, the smaller the loss.
- `validation_split` > 0: decrease of the loss during the algorithm, for training and validation data separately. The loss on the training data is expected to decrease with leveling off, whereas the loss on the validation data decreases and at some point increases again. The epoch with the smallest loss is then a good selection of the optimal number of epochs.

Below, we show two examples how the plots should ideally look like: * The corresponding tutorial on SSRN, Figure 14. * RStudio blog here

as follows, closely copy paste what you find on the linked websites as comments:





In the first chart (citing the comment on the page), the training loss decreases with every epoch. That's what you would expect when running a gradient-descent optimization – the quantity you're trying to minimize should be less with every iteration. But that isn't the case for the validation loss and accuracy: they seem to peak at the fourth epoch. This is an example of what we warned against earlier: a model that performs better on the training data isn't necessarily a model that will do better on data it has never seen before. In precise terms, what you're seeing is overfitting: after the second epoch, you're overoptimizing on the training data, and you end up learning representations that are specific to the training data and don't generalize to data outside of the training set.

In the second chart, the learning data $\mathcal{D}$ is split 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) validation. We emphasize that we choose the validation set disjointly from the test data $\mathcal{T}$. In the second figure the learning data is split 8:2 into training set (blue color) and validation set (green color). The network is fit on the training set and it is validated on the validation set. We observe that the model starts to over-fit to the learning data after roughly 150 epochs, since the validation loss (green color) starts to increase thereafter.

Exercise In the code here, we see the charts based on the training and validation set. Take this model and look at the curve on the test data.

Exercise For comparing the models, one should select the optimal number of epochs as selected above. Do so and look how the out-of-sample prediction changes.

Validation

```
# calculating the predictions
train$fitshNN <- as.vector(model_sh %>% predict(list(Xtrain, as.matrix(log(train$Exposure)))))
test$fitshNN <- as.vector(model_sh %>% predict(list(Xtest, as.matrix(log(test$Exposure)))))

# average in-sample and out-of-sample losses (in 10^(-2))
sprintf("100 x Poisson deviance shallow network (train): %s", PoissonDeviance(train$fitshNN, train$Clai
```

```
## [1] "100 x Poisson deviance shallow network (train): 23.9347984604197"
```

```
sprintf("100 x Poisson deviance shallow network (test): %s", PoissonDeviance(test$fitshNN, test$ClaimNb
```

```
## [1] "100 x Poisson deviance shallow network (test): 24.1111731499751"
```

```
# average frequency
sprintf("Average frequency (test): %s", round(sum(test$fitshNN) / sum(test$Exposure), 4))
```

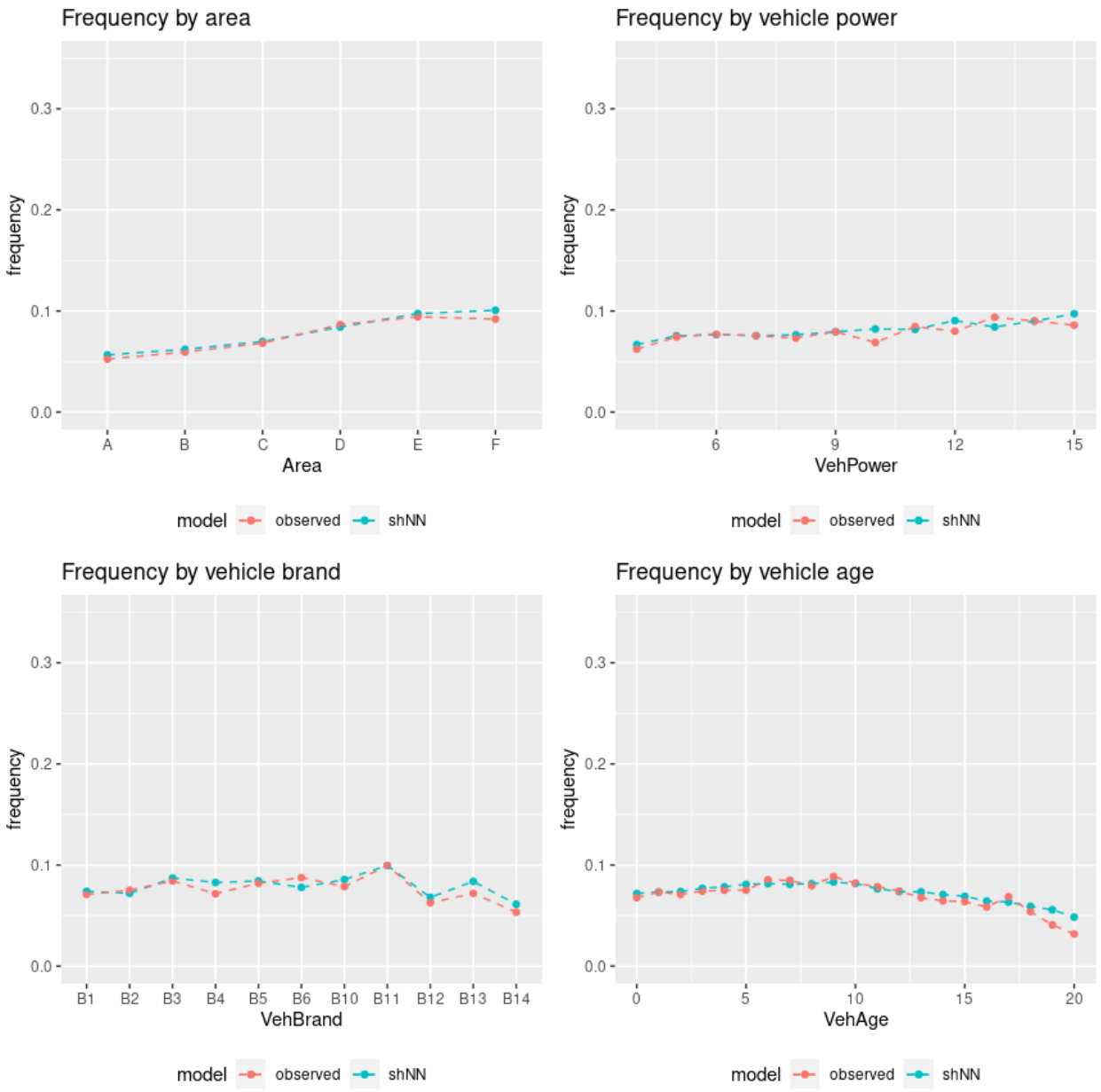
```
## [1] "Average frequency (test): 0.0754"
```

```
trainable_params <- sum(unlist(lapply(model_sh$trainable_weights, k_count_params)))
df_cmp %<>% bind_rows(
  data.frame(model = "M2: Shallow Plain Network", epochs = epochs,
             run_time = round(exec_time[[3]], 0), parameters = trainable_params,
             in_sample_loss = round(PoissonDeviance(train$fitshNN, train$ClaimNb), 4),
             out_sample_loss = round(PoissonDeviance(test$fitshNN, test$ClaimNb), 4),
             avg_freq = round(sum(test$fitshNN) / sum(test$Exposure), 4)
  ))

knitr::kable(df_cmp)
```

model	epochs	run_time	parameters	in_sample_loss	out_sample_loss	avg_freq
M1: GLM	NA	31	48	24.0875	24.1666	0.0737
M2: Shallow Plain Network	300	249	801	23.9348	24.1112	0.0754

Calibration

```
# Area
p1 <- plot_freq(test, "Area", "Frequency by area", "shNN", "fitshNN")
# VehPower
p2 <- plot_freq(test, "VehPower", "Frequency by vehicle power", "shNN", "fitshNN")
# VehBrand
p3 <- plot_freq(test, "VehBrand", "Frequency by vehicle brand", "shNN", "fitshNN")
# VehAge
p4 <- plot_freq(test, "VehAge", "Frequency by vehicle age", "shNN", "fitshNN")

grid.arrange(p1, p2, p3, p4)
```



Is worthwhile to remark that the fit is quite close to the observations and that the neural networks smooths out some jumps in the observations for VehAge which is in line with expectations.

Exercise: Change the number of neurons, compare the number of parameters and the fitted results.

Exercise: Change the input feature space by removing for example Area, VehPower from the input and compare the number of network parameters and the fitted results.

Exercise: Read the documentation to the optimizers and run the subsequent analysis with different optimizers and compare the results.

Exercise: Change the random seeds (at the beginning of the tutorial) and compare the results.

Exercise Run the same analysis and see if you can fully reproduce the results. What do you conclude from that?

Exercise: Change the activation function (only where it is appropriate) and compare the results.

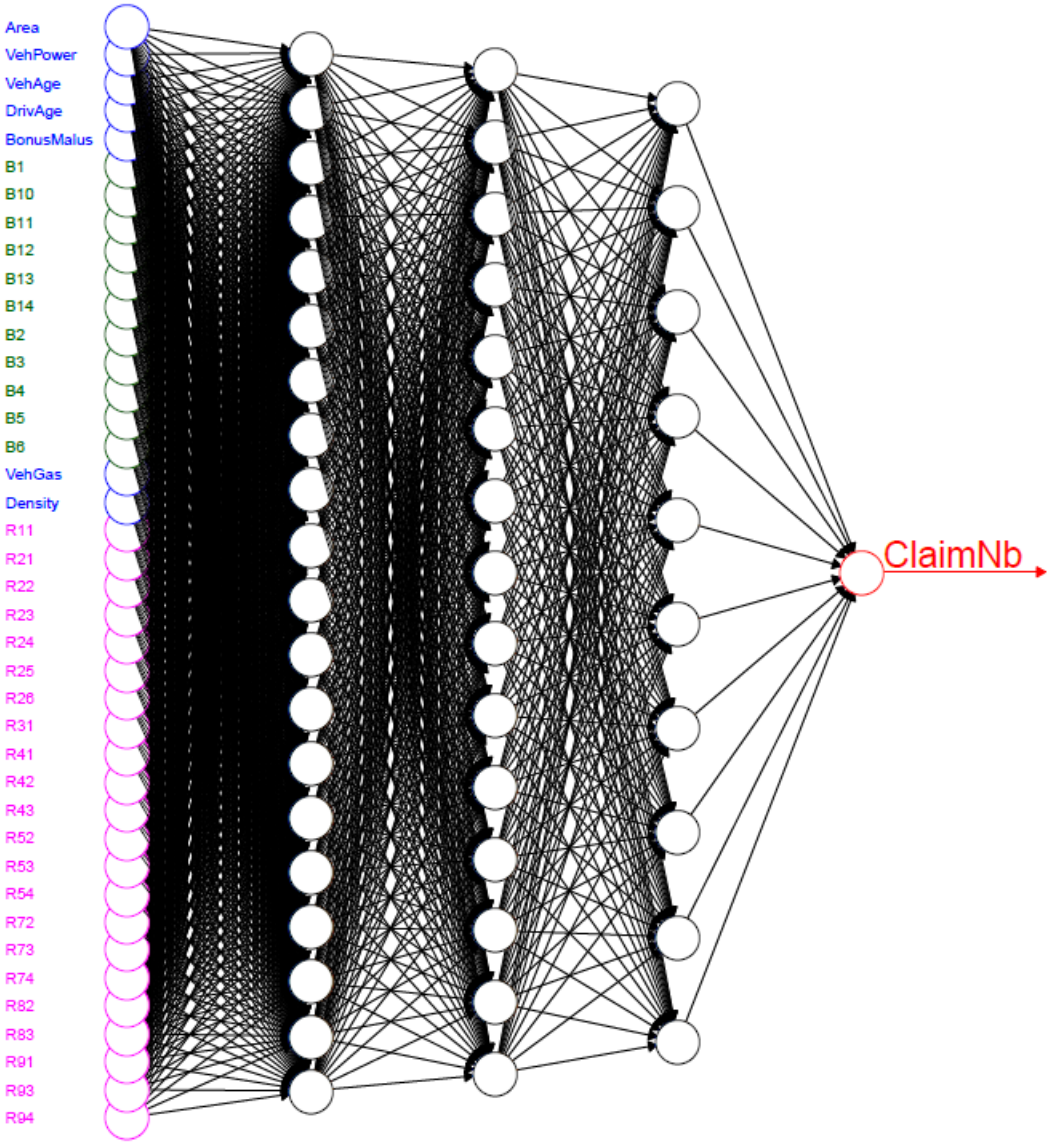
Exercise: Change the validation_split and verbose argument and see the difference in the fitting of the model.

Model 3: Deep plain vanilla neural network (dpNN)

Let us fit a deep neural network. In this chapter, we will not provide as many comments as above, as they remain valid for this chapter as well.

Before fitting and specifying a neural network, we highly recommend to draw it. This helps in using the `keras` function.

We choose a network with $K = 3$ **three hidden layer** and $q_1 = 20$, $q_1 = 15$ and $q_1 = 10$ **neurons**, which looks as follows:



The dimension of the input layer is defined by the selected input feature dimension (see above).

Definition

```
# define network
q0 <- length(features)   # dimension of features
q1 <- 20                 # number of neurons in first hidden layer
q2 <- 15                 # number of neurons in second hidden layer
q3 <- 10                 # number of neurons in second hidden layer

sprintf("Neural network with K=3 hidden layer")
```

```
## [1] "Neural network with K=3 hidden layer"
```

```
sprintf("Input feature dimension: q0 = %s", q0)
```

```
## [1] "Input feature dimension: q0 = 38"
```

```
sprintf("Number of hidden neurons first layer: q1 = %s", q1)
```

```
## [1] "Number of hidden neurons first layer: q1 = 20"
```

```
sprintf("Number of hidden neurons second layer: q2 = %s", q2)
```

```
## [1] "Number of hidden neurons second layer: q2 = 15"
```

```
sprintf("Number of hidden neurons third layer: q3 = %s", q3)
```

```
## [1] "Number of hidden neurons third layer: q3 = 10"
```

```
sprintf("Output dimension: %s", 1)
```

```
## [1] "Output dimension: 1"
```

Below we define the feed-forward shallow network with q_1 hidden neurons and hyperbolic tangent activation function, the output has exponential activation function due to the Poisson GLM.

The exposure is included as an offset, and we use the homogeneous model for initializing the output.

In this notebook we do not provide further details on the layers used and the structure, we refer to other notebooks or the references at the end of this notebook. A good reference is the `keras` cheat sheet.

```
Design <- layer_input(shape = c(q0), dtype = 'float32', name = 'Design')
LogVol <- layer_input(shape = c(1), dtype = 'float32', name = 'LogVol')

Network <- Design %>%
  layer_dense(units = q1, activation = 'tanh', name = 'layer1') %>%
  layer_dense(units = q2, activation = 'tanh', name = 'layer2') %>%
  layer_dense(units = q3, activation = 'tanh', name = 'layer3') %>%
  layer_dense(units = 1, activation = 'linear', name = 'Network',
              weights = list(array(0, dim = c(q3, 1)), array(log(lambda_hom), dim = c(1))))

Response <- list(Network, LogVol) %>%
  layer_add(name = 'Add') %>%
  layer_dense(units = 1, activation = k_exp, name = 'Response', trainable = FALSE,
              weights = list(array(1, dim = c(1, 1)), array(0, dim = c(1))))

model_dp <- keras_model(inputs = c(Design, LogVol), outputs = c(Response))
```

Compilation

```
model_dp %>% compile(
  loss = 'poisson',
  optimizer = optimizers[7]
)

summary(model_dp)
```

```
## Model: "model_1"
## __________________________________________________________________________________
## Layer (type)                Output Shape      Param #  Connected to
## ==================================================================================
## Design (InputLayer)         [(None, 38)]      0
## __________________________________________________________________________________
## layer1 (Dense)              (None, 20)        780      Design[0][0]
## __________________________________________________________________________________
## layer2 (Dense)              (None, 15)        315      layer1[0][0]
## __________________________________________________________________________________
## layer3 (Dense)              (None, 10)        160      layer2[0][0]
## __________________________________________________________________________________
## Network (Dense)             (None, 1)         11       layer3[0][0]
## __________________________________________________________________________________
## LogVol (InputLayer)         [(None, 1)]       0
## __________________________________________________________________________________
## Add (Add)                   (None, 1)         0        Network[0][0]
##                                                        LogVol[0][0]
## __________________________________________________________________________________
## Response (Dense)            (None, 1)         2        Add[0][0]
## ==================================================================================
## Total params: 1,268
## Trainable params: 1,266
## Non-trainable params: 2
## __________________________________________________________________________________
```

Fitting

For fitting a keras model and its arguments, we refer to the help of `fit` here. There you find details about the `validation_split` and the `verbose` argument.

```
# select number of epochs and batch_size
epochs <- 300
batch_size <- 10000
validation_split <- 0.2  # set to >0 to see train/validation loss in plot(fit)
verbose <- 1
```

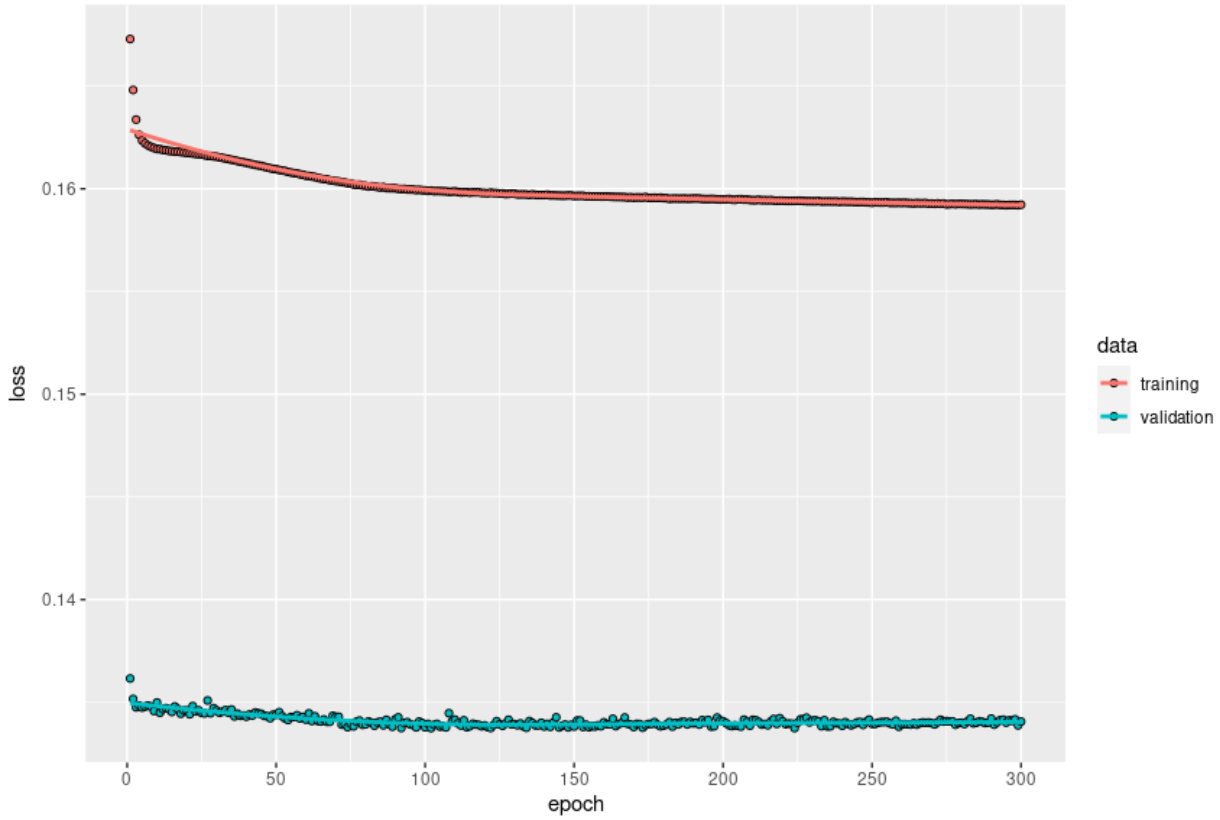
```
# expected run-time on Renku 8GB environment around 200 seconds
exec_time <- system.time(
  fit <- model_dp %>% fit(
    list(Xtrain, as.matrix(log(train$Exposure))), as.matrix(train$ClaimNb),
    epochs = epochs,
    batch_size = batch_size,
    validation_split = validation_split,
    verbose = verbose
  )
```

```
)
exec_time[1:5]
```

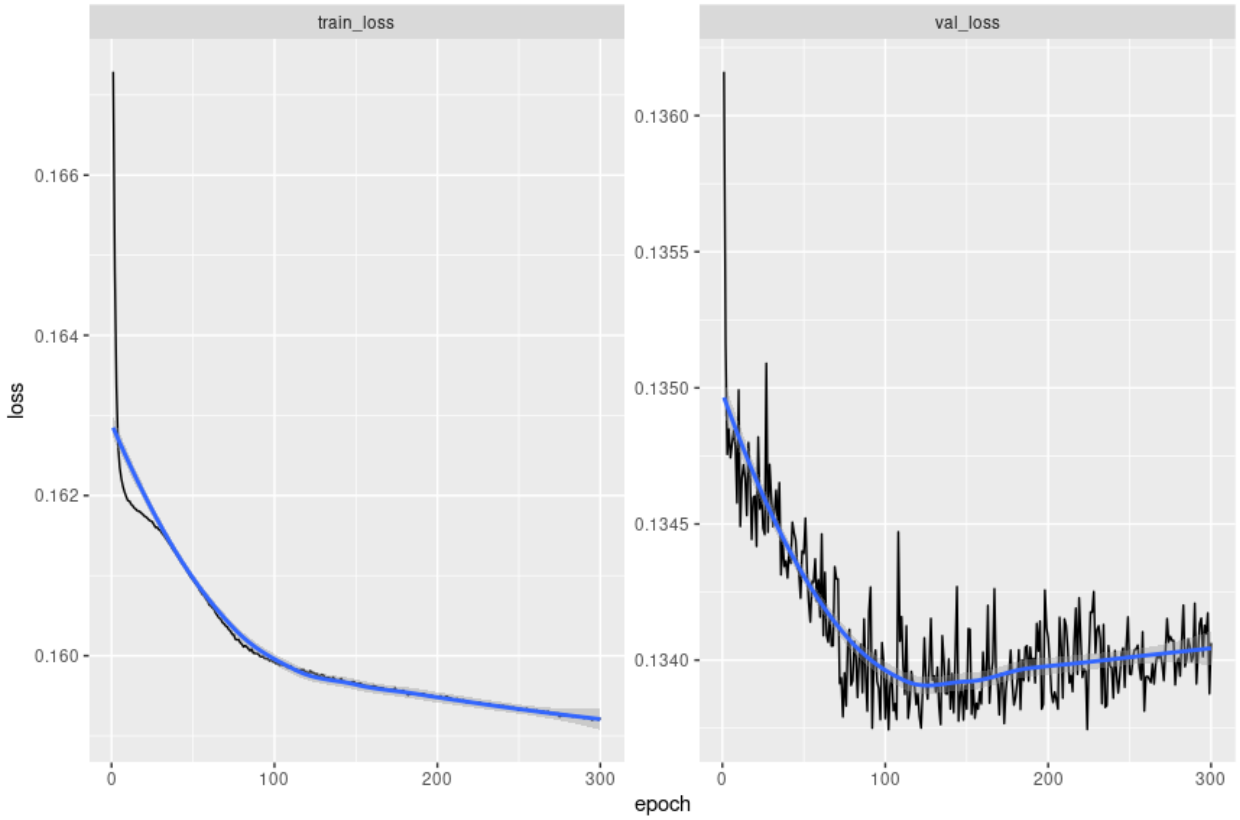
```
##  user.self   sys.self     elapsed user.child  sys.child
##    446.245     61.309     375.441       0.000       0.000
```

```
plot(fit)
```

```
## `geom_smooth()` using formula 'y ~ x'
```




```
plot_loss(x = fit[[2]])
```



Again, around 100 epochs seem to be optimal.

Validation

```
# Validation: Poisson deviance
train$fitdpNN <- as.vector(model_dp %>% predict(list(Xtrain, as.matrix(log(train$Exposure)))))
test$fitdpNN <- as.vector(model_dp %>% predict(list(Xtest, as.matrix(log(test$Exposure)))))

sprintf("100 x Poisson deviance shallow network (train): %s", PoissonDeviance(train$fitdpNN, train$Clai
```

```
## [1] "100 x Poisson deviance shallow network (train): 23.6396486382078"
```

```
sprintf("100 x Poisson deviance shallow network (test): %s", PoissonDeviance(test$fitdpNN, test$ClaimNb
```

```
## [1] "100 x Poisson deviance shallow network (test): 23.9750925146365"
```

```
# average frequency
sprintf("Average frequency (test): %s", round(sum(test$fitdpNN) / sum(test$Exposure), 4))
```

```
## [1] "Average frequency (test): 0.0747"
```

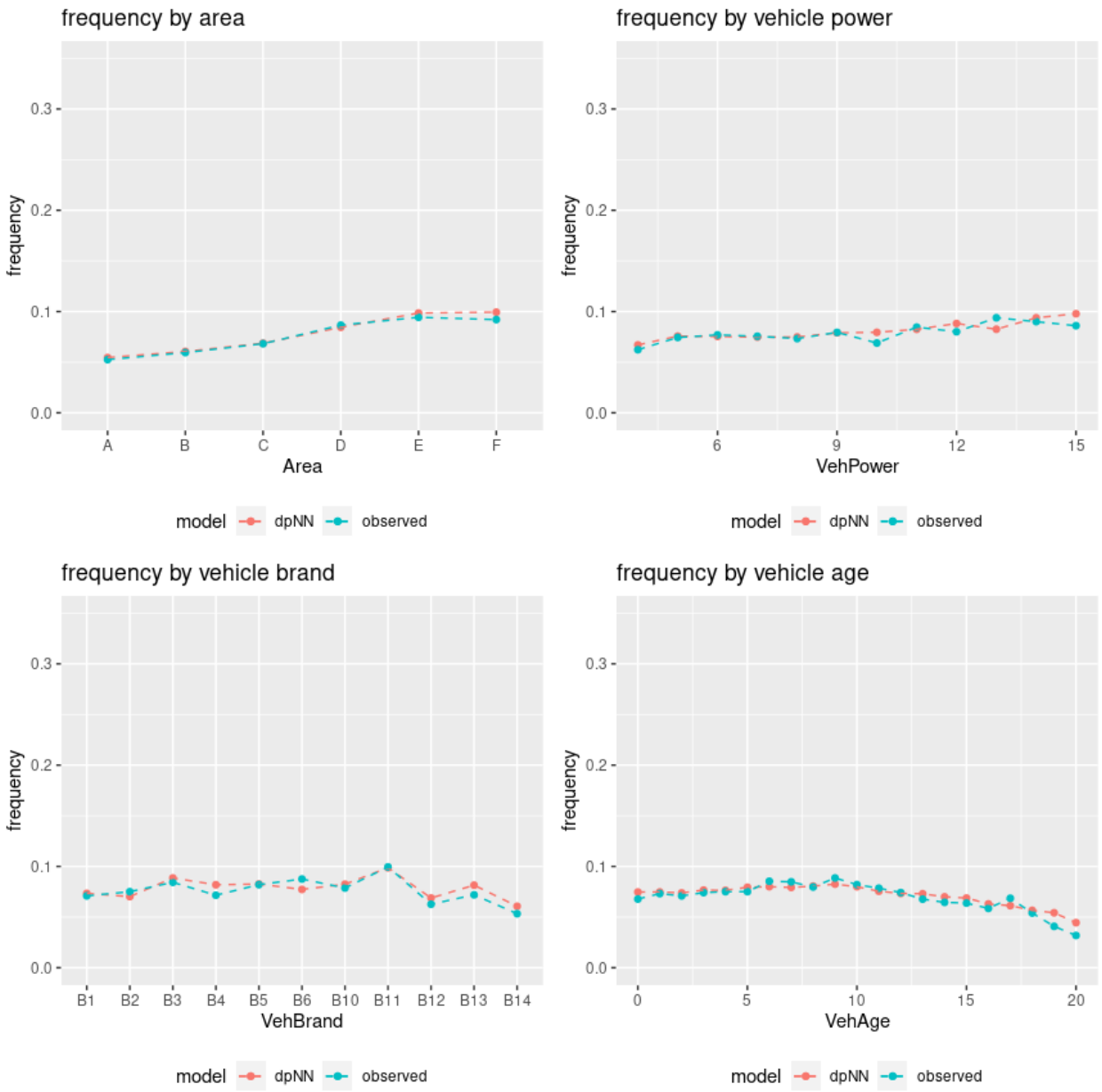
```
trainable_params <- sum(unlist(lapply(model_dp$trainable_weights, k_count_params)))
df_cmp %<>% bind_rows(
  data.frame(model = "M3: Deep Plain Network", epochs = epochs,
             run_time = round(exec_time[[3]], 0), parameters = trainable_params,
             in_sample_loss = round(PoissonDeviance(train$fitdpNN, train$ClaimNb), 4),
             out_sample_loss = round(PoissonDeviance(test$fitdpNN, test$ClaimNb), 4),
             avg_freq = round(sum(test$fitdpNN) / sum(test$Exposure), 4)
  ))
```

```
knitr::kable(df_cmp)
```

model	epochs	run_time	parameters	in_sample_loss	out_sample_loss	avg_freq
M1: GLM	NA	31	48	24.0875	24.1666	0.0737
M2: Shallow Plain Network	300	249	801	23.9348	24.1112	0.0754
M3: Deep Plain Network	300	375	1266	23.6396	23.9751	0.0747

Calibration

```
# Area
p1 <- plot_freq(test, "Area", "frequency by area", "dpNN", "fitdpNN")
# VehPower
p2 <- plot_freq(test, "VehPower", "frequency by vehicle power", "dpNN", "fitdpNN")
# VehBrand
p3 <- plot_freq(test, "VehBrand", "frequency by vehicle brand", "dpNN", "fitdpNN")
# VehAge
p4 <- plot_freq(test, "VehAge", "frequency by vehicle age", "dpNN", "fitdpNN")

grid.arrange(p1, p2, p3, p4)
```



Model 4: Deep neural network with dropout layers (drNN)

Definition

```
# define network
q0 <- length(features)  # dimension of input features
q1 <- 20                # number of neurons in first hidden layer
q2 <- 15                # number of neurons in second hidden layer
q3 <- 10                # number of neurons in second hidden layer
p0 <- 0.05              # dropout rate

sprintf("Neural network with K=3 hidden layer and 3 dropout layers")
```

```
## [1] "Neural network with K=3 hidden layer and 3 dropout layers"
```

```
sprintf("Input feature dimension: q0 = %s", q0)
```

```
## [1] "Input feature dimension: q0 = 38"
```

```
sprintf("Number of hidden neurons first layer: q1 = %s", q1)
```

```
## [1] "Number of hidden neurons first layer: q1 = 20"
```

```
sprintf("Number of hidden neurons second layer: q2 = %s", q2)
```

```
## [1] "Number of hidden neurons second layer: q2 = 15"
```

```
sprintf("Number of hidden neurons third layer: q3 = %s", q3)
```

```
## [1] "Number of hidden neurons third layer: q3 = 10"
```

```
sprintf("Output dimension: %s", 1)
```

```
## [1] "Output dimension: 1"
```

```
Design <- layer_input(shape = c(q0), dtype = 'float32', name = 'Design')
LogVol <- layer_input(shape = c(1), dtype = 'float32', name = 'LogVol')

Network <- Design %>%
  layer_dense(units = q1, activation = 'tanh', name = 'layer1') %>%
  layer_dropout(rate = p0) %>%
  layer_dense(units = q2, activation = 'tanh', name = 'layer2') %>%
  layer_dropout(rate = p0) %>%
  layer_dense(units = q3, activation = 'tanh', name = 'layer3') %>%
  layer_dropout(rate = p0) %>%
  layer_dense(units = 1, activation = 'linear', name = 'Network',
              weights = list(array(0, dim = c(q3, 1)), array(log(lambda_hom), dim = c(1))))

Response <- list(Network, LogVol) %>%
  layer_add(name = 'Add') %>%
  layer_dense(units = 1, activation = k_exp, name = 'Response', trainable = FALSE,
              weights = list(array(1, dim = c(1, 1)), array(0, dim = c(1))))

model_dr <- keras_model(inputs = c(Design, LogVol), outputs = c(Response))
```

Compilation

```
model_dr %>% compile(
  loss = 'poisson',
  optimizer = optimizers[7]
)

summary(model_dr)
```

```
## Model: "model_2"
## __________________________________________________________________________________
## Layer (type)                Output Shape      Param #  Connected to
## ==================================================================================
## Design (InputLayer)         [(None, 38)]      0
## __________________________________________________________________________________
## layer1 (Dense)              (None, 20)        780      Design[0][0]
```

```
## ____________________________________________________________________________
## dropout_2 (Dropout)       (None, 20)        0        layer1[0][0]
## ____________________________________________________________________________
## layer2 (Dense)            (None, 15)        315      dropout_2[0][0]
## ____________________________________________________________________________
## dropout_1 (Dropout)       (None, 15)        0        layer2[0][0]
## ____________________________________________________________________________
## layer3 (Dense)            (None, 10)        160      dropout_1[0][0]
## ____________________________________________________________________________
## dropout (Dropout)         (None, 10)        0        layer3[0][0]
## ____________________________________________________________________________
## Network (Dense)           (None, 1)         11       dropout[0][0]
## ____________________________________________________________________________
## LogVol (InputLayer)       [(None, 1)]       0
## ____________________________________________________________________________
## Add (Add)                 (None, 1)         0        Network[0][0]
##                                                      LogVol[0][0]
## ____________________________________________________________________________
## Response (Dense)          (None, 1)         2        Add[0][0]
## ============================================================================
## Total params: 1,268
## Trainable params: 1,266
## Non-trainable params: 2
## ____________________________________________________________________________
```

Fitting

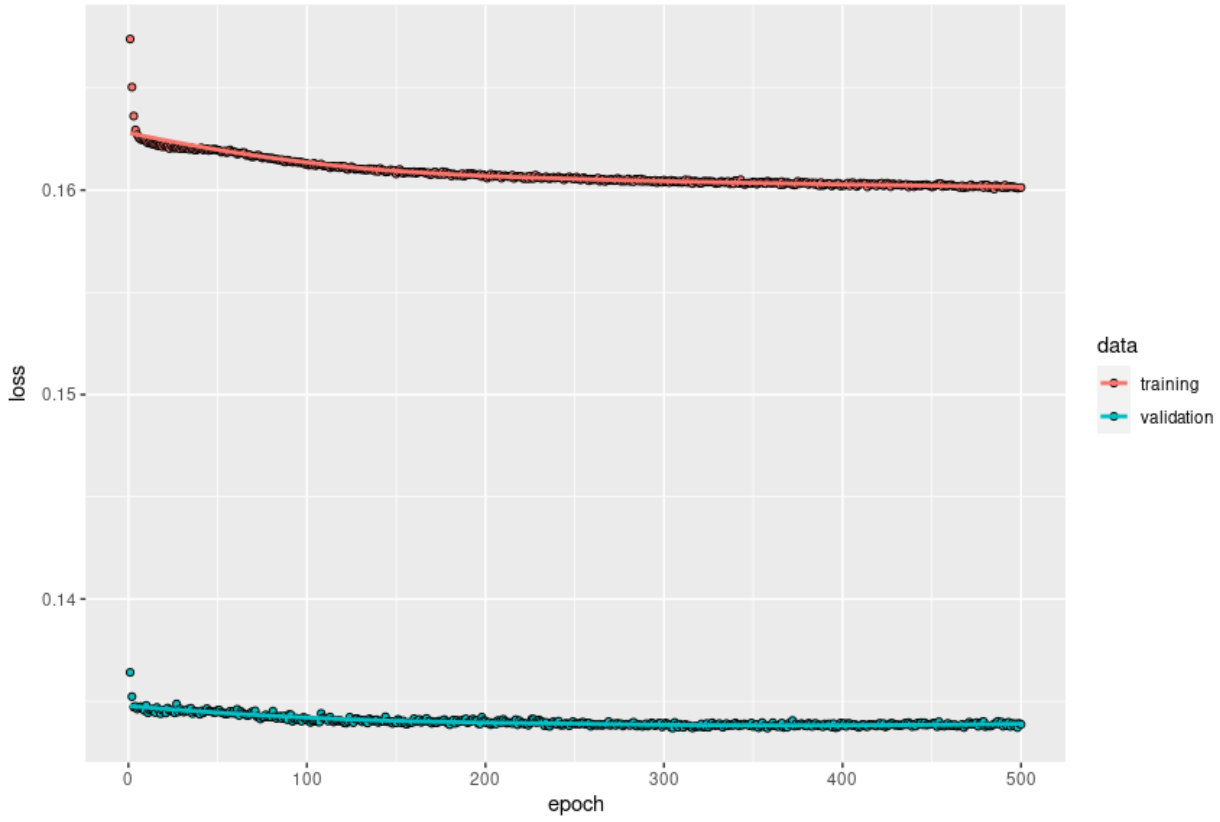
```
# select number of epochs and batch_size
epochs <- 500
batch_size <- 10000
validation_split <- 0.2  # set >0 to see train/validation loss in plot(fit)
verbose <- 1
```

```
# expected run-time on Renku 8GB environment around 110 seconds (for 100 epochs)
exec_time <- system.time(
  fit <- model_dr %>% fit(
    list(Xtrain, as.matrix(log(train$Exposure))), as.matrix(train$ClaimNb),
    epochs = epochs,
    batch_size = batch_size,
    validation_split = validation_split,
    verbose = verbose
  )
)
exec_time[1:5]
```

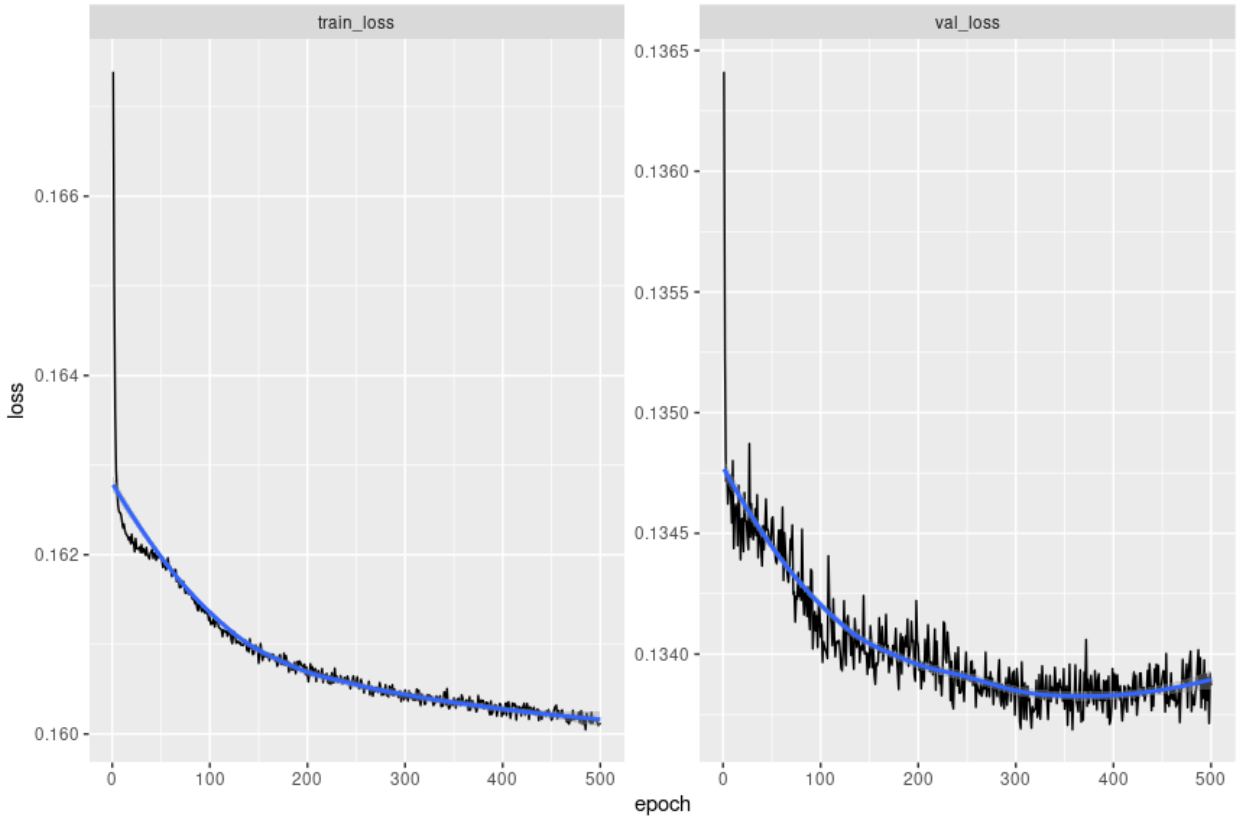
```
##  user.self   sys.self    elapsed user.child  sys.child
##    992.100    143.070    947.861      0.000      0.000
```

```
plot(fit)
```

```
## `geom_smooth()` using formula 'y ~ x'
```




```
plot_loss(x = fit[[2]])
```



Validation

```
# Validation: Poisson deviance
train$fitdrNN <- as.vector(model_dr %>% predict(list(Xtrain, as.matrix(log(train$Exposure)))))
test$fitdrNN <- as.vector(model_dr %>% predict(list(Xtest, as.matrix(log(test$Exposure)))))

sprintf("100 x Poisson deviance shallow network (train): %s", PoissonDeviance(train$fitdrNN, train$Clai
```

```
## [1] "100 x Poisson deviance shallow network (train): 23.7130495894515"
```

```
sprintf("100 x Poisson deviance shallow network (test): %s", PoissonDeviance(test$fitdrNN, test$ClaimNb
```

```
## [1] "100 x Poisson deviance shallow network (test): 23.9458073240059"
```

```
# average frequency
sprintf("Average frequency (test): %s", round(sum(test$fitdrNN) / sum(test$Exposure), 4))
```

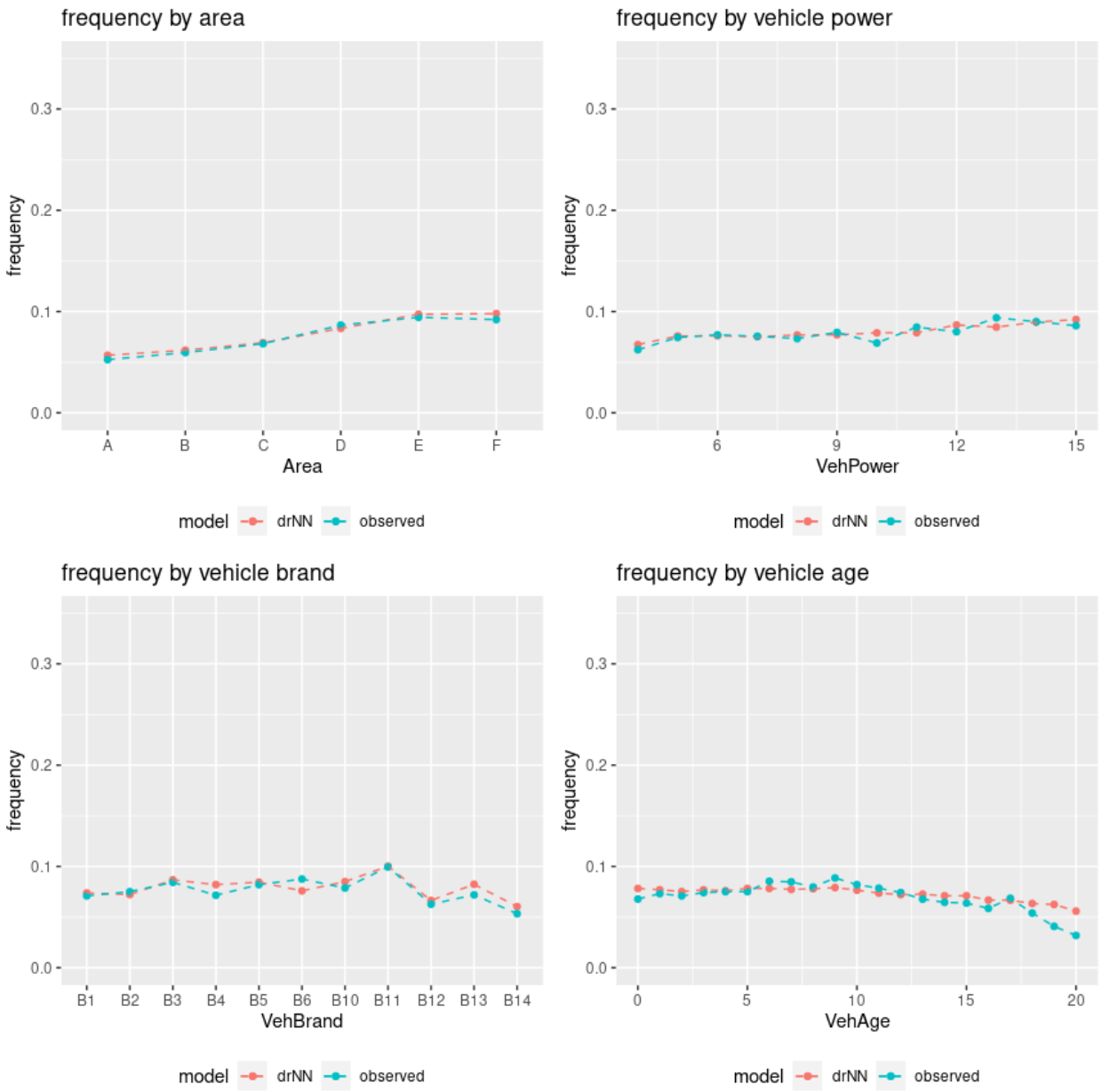
```
## [1] "Average frequency (test): 0.0749"
```

```
trainable_params <- sum(unlist(lapply(model_dr$trainable_weights, k_count_params)))
df_cmp %<>% bind_rows(
  data.frame(model = "M4: Deep Dropout Network", epochs = epochs,
             run_time = round(exec_time[[3]], 0), parameters = trainable_params,
             in_sample_loss = round(PoissonDeviance(train$fitdrNN, train$ClaimNb), 4),
             out_sample_loss = round(PoissonDeviance(test$fitdrNN, test$ClaimNb), 4),
             avg_freq = round(sum(test$fitdrNN) / sum(test$Exposure), 4)
  ))
knitr::kable(df_cmp)
```

model	epochs	run_time	parameters	in_sample_loss	out_sample_loss	avg_freq
M1: GLM	NA	31	48	24.0875	24.1666	0.0737
M2: Shallow Plain Network	300	249	801	23.9348	24.1112	0.0754
M3: Deep Plain Network	300	375	1266	23.6396	23.9751	0.0747
M4: Deep Dropout Network	500	948	1266	23.7130	23.9458	0.0749

Calibration

```
# Area
p1 <- plot_freq(test, "Area", "frequency by area", "drNN", "fitdrNN")
# VehPower
p2 <- plot_freq(test, "VehPower", "frequency by vehicle power", "drNN", "fitdrNN")
# VehBrand
p3 <- plot_freq(test, "VehBrand", "frequency by vehicle brand", "drNN", "fitdrNN")
# VehAge
p4 <- plot_freq(test, "VehAge", "frequency by vehicle age", "drNN", "fitdrNN")

grid.arrange(p1, p2, p3, p4)
```



Further network architectures

So far, we have fitted three different network architectures and we are comparing their performance and quality of fit.

As mentioned above, one can amend and change hyperparameters (optimizer, batch size, epochs,...) and see how the results change.

One can also change the neural network architecture, e.g. * change the number of layers and neurons * add additional layers, e.g. normalization layers * add ridge regularization

See: * the keras cheat sheet https://github.com/rstudio/cheatsheets/raw/master/keras.pdf. * the tutorial: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3226852

Exercise: Have a look at the keras cheat sheet, study the normalization layer and add them and compare

the results.

Exercise: Look at other layers, try to understand them and compare the impact of these layers.

Exercise: Apply ridge regularization, understand what it is and compare the results.

Exercise: In all models above, select the optimal number of epochs as described and compare the overall results.

With fitting neural networks, there are a couple of questions arising, and we like to share some of our experience here:

- The choice of the architecture can be considered more as art than science
- Neural networks move the challenge of feature engineering (which the network learns) to the challenge of selecting the architecture.
- There are a few rules of thumb on the architecture:
  - For structured data, only 3-5 layers are required, more layers do not improve the accuracy further
  - For finding interactions, the third and higher layers are "considering" them. The first two are for the "main effects"
  - In this tutorial on SSRN, there is a proposal how to choose the optimal batch size for insurance pricing data, see formula (4.5) on p. 25.
- The treatment of missing values is an open question.

Model Comparison

The results of the various models are as follows.

```
knitr::kable(df_cmp)
```

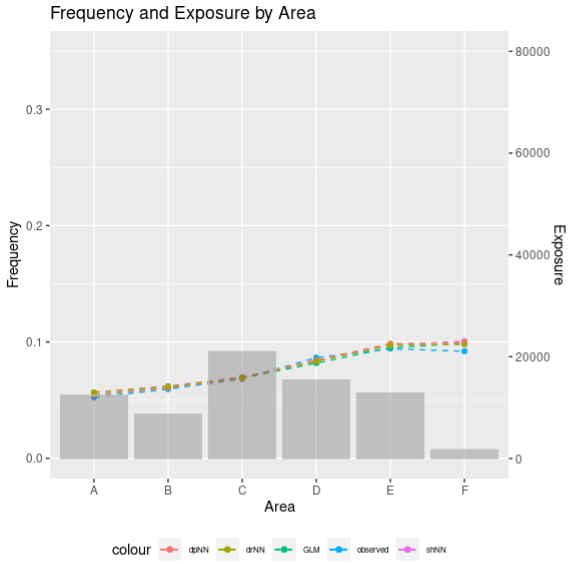
model	epochs	run_time	parameters	in_sample_loss	out_sample_loss	avg_freq
M1: GLM	NA	31	48	24.0875	24.1666	0.0737
M2: Shallow Plain Network	300	249	801	23.9348	24.1112	0.0754
M3: Deep Plain Network	300	375	1266	23.6396	23.9751	0.0747
M4: Deep Dropout Network	500	948	1266	23.7130	23.9458	0.0749

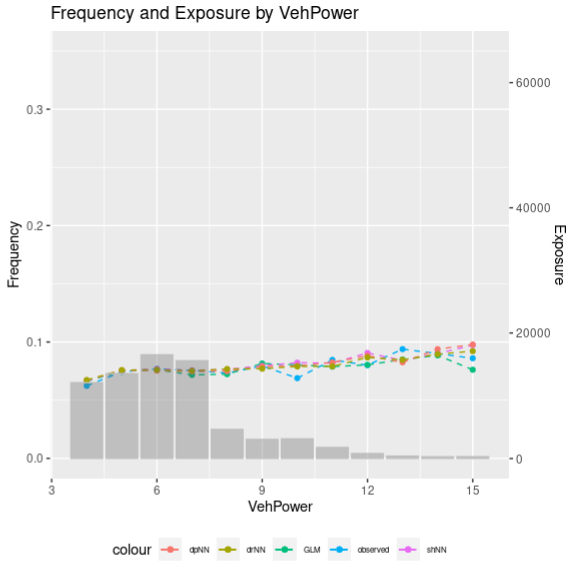
We can draw the following conclusions:

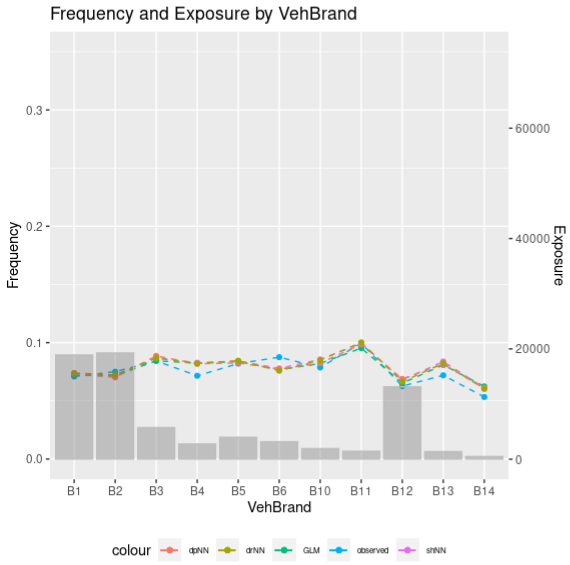
- The out-of sample loss of the models M3 and M4 is clearly better than for M1 and M2.
- The better performance of M3 and M4 indicates that some interactions are missing in M1 and M2. This coincides with the general understanding that as of 3 hidden layers interactions can be captured.
- The shallow plain network seems not to be able to model the interactions appropriately, and they are not included in the GLM.
- The fitted average claim frequency differs between the neural network models. This is different when comparing GLM's (see the glm tutorial on the same data) where the predicted claim frequency is the same for all models. This is the so called **bias regularization** issue of neural networks, which is discussed in the tutorial, section 6.5.
- We see that the `freqMTPL2freq` dataset is maybe not ideal and not representative for a standard primary insurance pricing dataset, due to:
  - The performance of the various models is quite similar and there are not many areas with highly different predictions.
  - The marginal predicted frequencies do not vary much across the various categorical feature levels.
  - The goal of the tutorial is more to demonstrate the techniques and compare them rather to nicely identify important features.

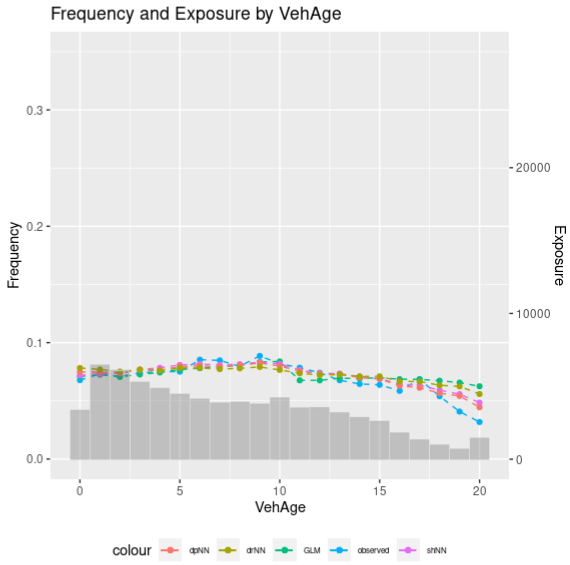
Calibration

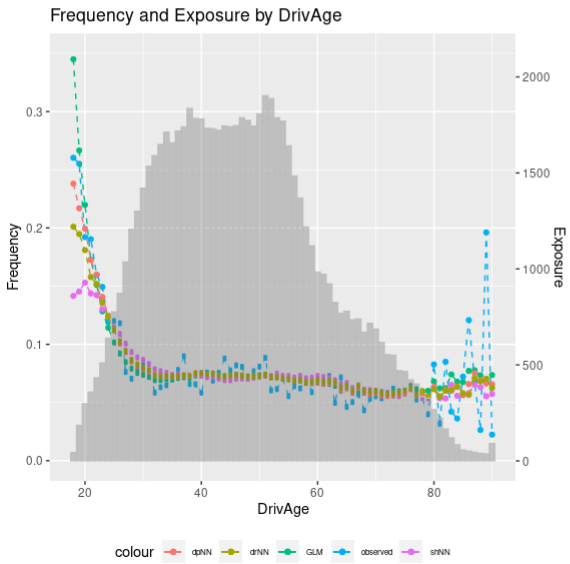
```
plot_cmp <- function(xvar, title, maxlim = 0.35) {
  out <- test %>% group_by(!!sym(xvar)) %>% summarize(
    vol = sum(Exposure),
    obs = sum(ClaimNb) / sum(Exposure),
    glm = sum(fitGLM2) / sum(Exposure),
    shNN = sum(fitshNN) / sum(Exposure),
    dpNN = sum(fitdpNN) / sum(Exposure),
    drNN = sum(fitdrNN) / sum(Exposure)
  )

  max_pri <- max(out$obs, out$glm, out$shNN, out$dpNN, out$drNN)
  max_sec <- 1.1 * max(out$vol)
  max_ratio <- max_pri / max_sec

  if (is.null(maxlim)) maxlim <- max_pri

  ggplot(out, aes(x = !!sym(xvar), group = 1)) +
    geom_point(aes(y = obs, colour = "observed")) + geom_line(aes(y = obs, colour = "observed"), linety
    geom_point(aes(y = glm, colour = "GLM")) + geom_line(aes(y = glm, colour = "GLM"), linetype = "dash
    geom_point(aes(y = shNN, colour = "shNN")) + geom_line(aes(y = shNN, colour = "shNN"), linetype = "
    geom_point(aes(y = dpNN, colour = "dpNN")) + geom_line(aes(y = dpNN, colour = "dpNN"), linetype = "
    geom_point(aes(y = drNN, colour = "drNN")) + geom_line(aes(y = drNN, colour = "drNN"), linetype = "
    geom_bar(aes(y = vol * (max_ratio)), colour = "grey", stat = "identity", alpha = 0.3) +
    scale_y_continuous(name = "Frequency", sec.axis = sec_axis( ~ . / (max_ratio), name = "Exposure"),
    labs(x = xvar, title = title) + theme(legend.position = "bottom", legend.text = element_text(size =
}

# Area
p1 <- plot_cmp("Area", "Frequency and Exposure by Area")
# VehPower
p2 <- plot_cmp("VehPower", "Frequency and Exposure by VehPower")
# VehBrand
p3 <- plot_cmp("VehBrand", "Frequency and Exposure by VehBrand")
# VehAge
p4 <- plot_cmp("VehAge", "Frequency and Exposure by VehAge")
# DrivAge plot with exposure distribution
p5 <- plot_cmp("DrivAge", "Frequency and Exposure by DrivAge")

grid.arrange(p1, p2, p3, p4, p5)
```


Frequency and Exposure by Area
Frequency
Exposure
Area
A
B
C
D
E
F
colour
dpNN
drNN
GLM
observed
shNN


Frequency and Exposure by VehPower
Frequency
Exposure
VehPower
colour
dpNN
drNN
GLM
observed
shNN


Frequency and Exposure by VehBrand
Frequency
Exposure
VehBrand
B1
B2
B3
B4
B5
B6
B10
B11
B12
B13
B14
colour
dpNN
drNN
GLM
observed
shNN

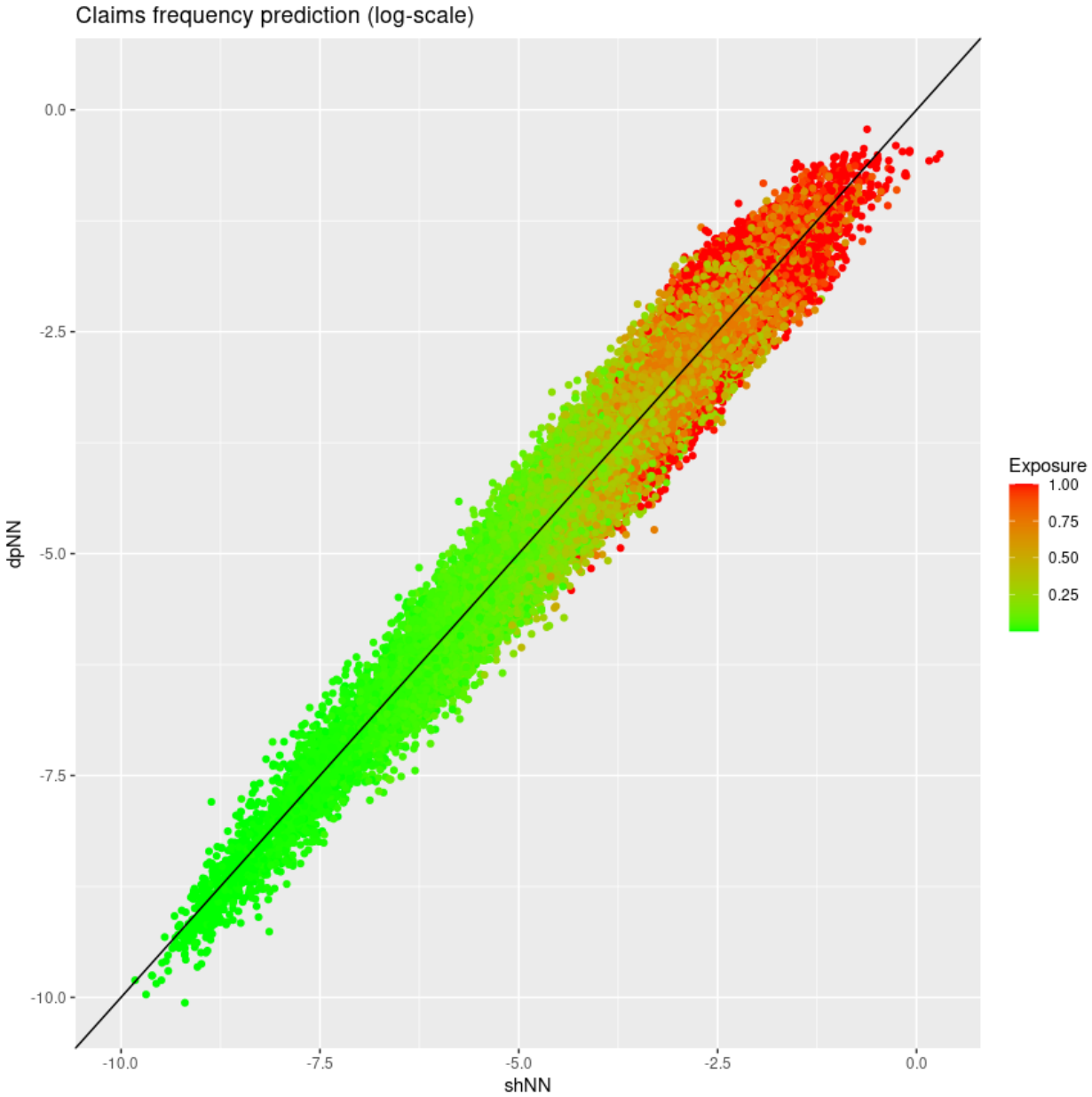

Frequency and Exposure by VehAge
Frequency
Exposure
VehAge
colour
dpNN
drNN
GLM
observed
shNN


Frequency and Exposure by DrivAge
Frequency
Exposure
DrivAge
colour
dpNN
drNN
GLM
observed
shNN

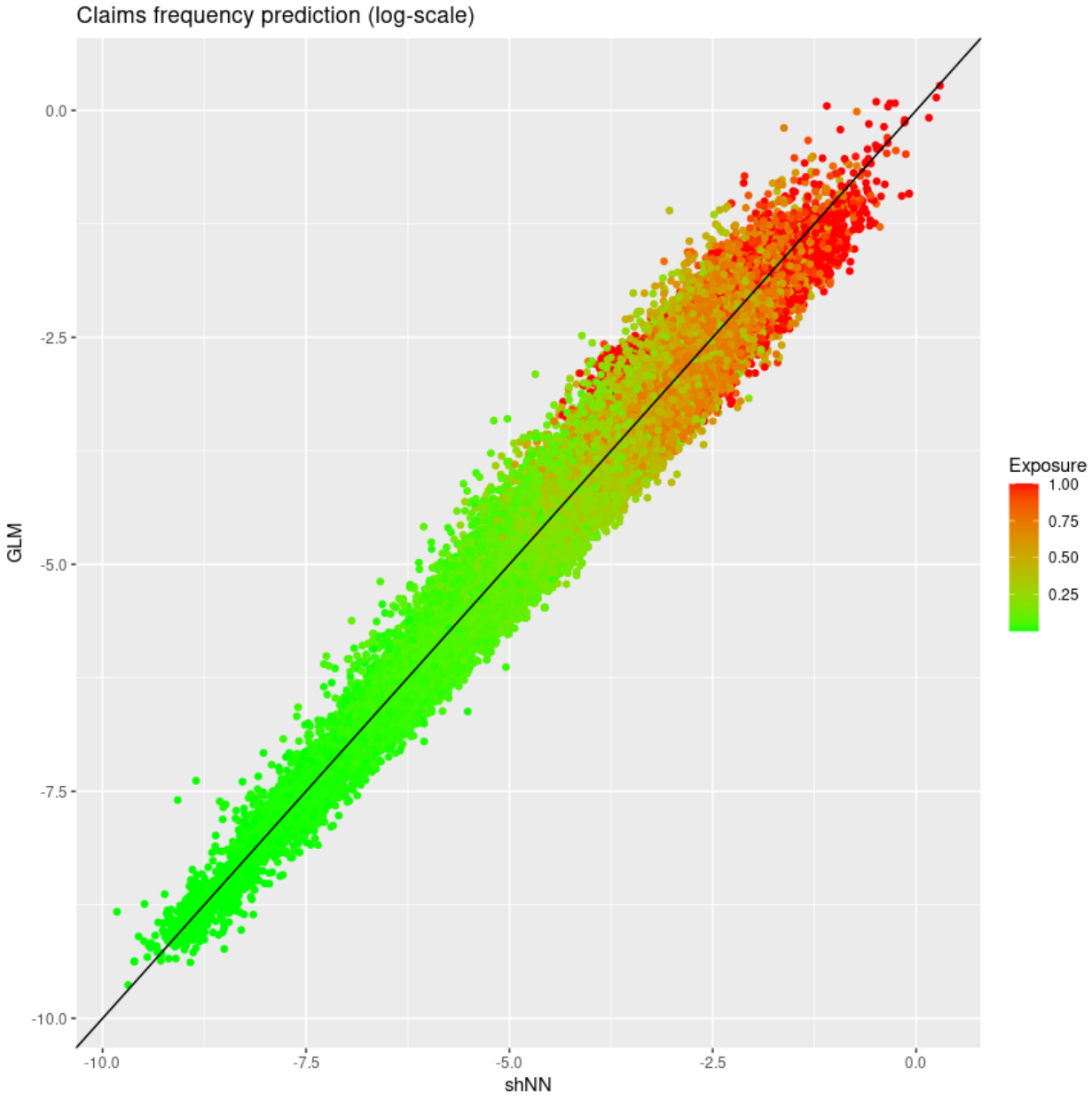
Out-of-sample claims frequency predictions (on log-scales) comparison.

```
plot_claims_freq <- function(xvar, yvar, xlab, ylab) {
  axis_min <- log(max(test[[xvar]], test[[yvar]]))
  axis_max <- log(min(test[[xvar]], test[[yvar]]))

  ggplot(test, aes(x = log(!!sym(xvar)), y = log(!!sym(yvar)), colour = Exposure)) + geom_point() +
    geom_abline(colour = "#000000", slope = 1, intercept = 0) +
    xlim(axis_max, axis_min) + ylim(axis_max, axis_min) +
    labs(x = xlab, y = ylab, title = "Claims frequency prediction (log-scale)") +
    scale_colour_gradient(low = "green", high = "red")
}
```

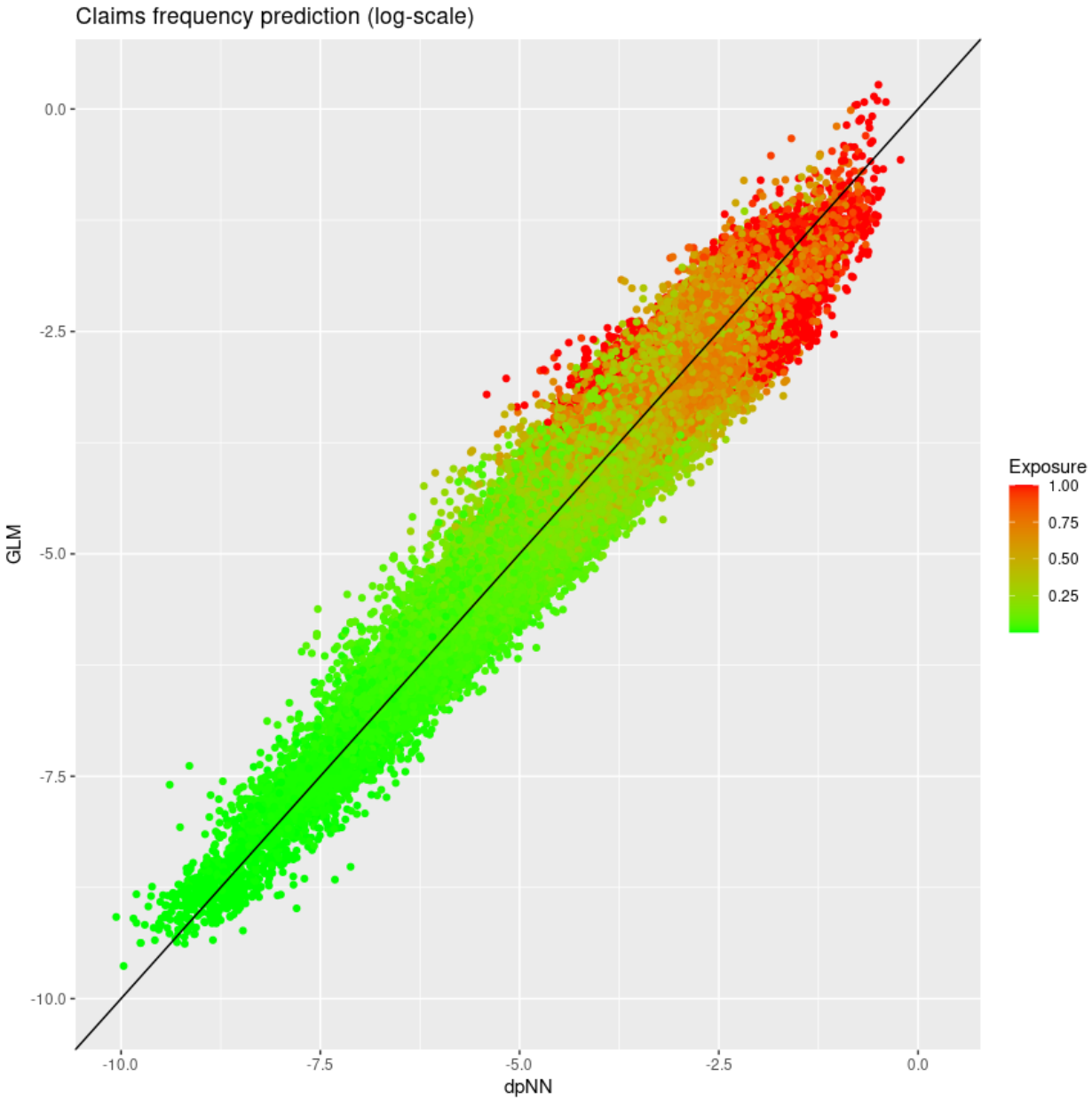
```
plot_claims_freq("fitshNN", "fitdpNN", "shNN", "dpNN")
```



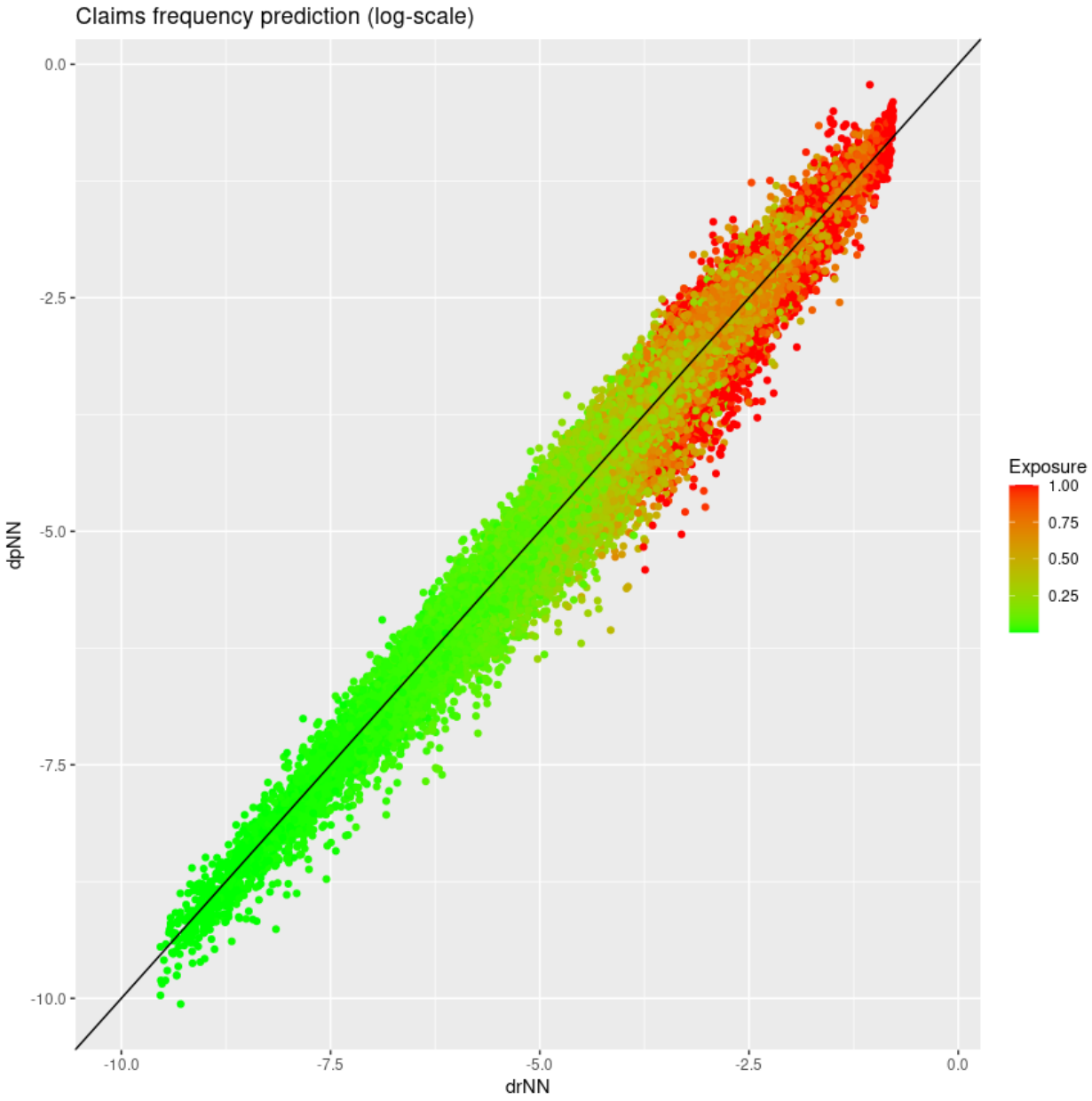
```
plot_claims_freq("fitshNN", "fitGLM2", "shNN", "GLM")
```




```
plot_claims_freq("fitdpNN", "fitGLM2", "dpNN", "GLM")
```




```
plot_claims_freq("fitdrNN", "fitdpNN", "drNN", "dpNN")
```



Exercise: The `keras(...)` function can also be used to fit a glm. Take the shallow network structure, reduce the hidden layer and fit a glm instead. Then compare it to the corresponding output from using the function `glm()`.

Exercise: As initial weight for the fitting, we use the homogeneous model fit (stored in glmHom). Examine the dependency of the output from changes to this input parameter.

Session Info

The html is generated with the following packages (which might be slightly newer than the ones used in the published tutorial).

```
sessionInfo()
```

```
## R version 4.0.5 (2021-03-31)
```

```
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 20.04.2 LTS
##
## Matrix products: default
## BLAS/LAPACK: /usr/lib/x86_64-linux-gnu/openblas-pthread/libopenblasp-r0.3.8.so
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
##  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=C
##  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
##  [1] repr_1.1.3       tidyr_1.1.3      splitTools_0.3.1 gridExtra_2.3
##  [5] ggplot2_3.3.5    purrr_0.3.4      tibble_3.1.4     dplyr_1.0.7
##  [9] magrittr_2.0.1   keras_2.6.0
##
## loaded via a namespace (and not attached):
##  [1] Rcpp_1.0.7        highr_0.9         pillar_1.6.2      compiler_4.0.5
##  [5] base64enc_0.1-3   tools_4.0.5       zeallot_0.1.0     digest_0.6.27
##  [9] lattice_0.20-41   nlme_3.1-152      jsonlite_1.7.2    evaluate_0.14
## [13] lifecycle_1.0.0   gtable_0.3.0      mgcv_1.8-34       pkgconfig_2.0.3
## [17] rlang_0.4.11      Matrix_1.3-2      DBI_1.1.1         yaml_2.2.1
## [21] xfun_0.23         withr_2.4.2       stringr_1.4.0     knitr_1.34
## [25] rappdirs_0.3.3    generics_0.1.0    vctrs_0.3.8       grid_4.0.5
## [29] tidyselect_1.1.1  reticulate_1.14   glue_1.4.2        R6_2.5.0
## [33] fansi_0.4.2       rmarkdown_2.11    farver_2.1.0      whisker_0.4
## [37] splines_4.0.5     scales_1.1.1      tfruns_1.5.0      ellipsis_0.3.2
## [41] htmltools_0.5.1.1 assertthat_0.2.1  colorspace_2.0-1  labeling_0.4.2
## [45] tensorflow_2.6.0  utf8_1.2.1        stringi_1.6.1     munsell_0.5.0
## [49] crayon_1.4.1
```

```
reticulate::py_config()
```

```
## python:         /home/rstudio/.virtualenvs/r-reticulate/bin/python
## libpython:      /opt/conda/lib/libpython3.9.so
## pythonhome:     /opt/conda:/opt/conda
## version:        3.9.5 | packaged by conda-forge | (default, Jun 19 2021, 00:32:32)  [GCC 9.3.0]
## numpy:          /home/rstudio/.virtualenvs/r-reticulate/lib/python3.9/site-packages/numpy
## numpy_version:  1.19.5
## tensorflow:     /home/rstudio/.virtualenvs/r-reticulate/lib/python3.9/site-packages/tensorflow
##
## python versions found:
##  /home/rstudio/.virtualenvs/r-reticulate/bin/python
##  /opt/conda/bin/python3
##  /usr/bin/python3
```

```
tensorflow::tf_version()
```

```
## [1] '2.6'
```

References

- https://tensorflow.rstudio.com/guide/
- https://github.com/rstudio/cheatsheets/raw/master/keras.pdf
- https://cran.r-project.org/web/packages/keras/vignettes/guide_keras.html
- https://keras.rstudio.com/articles/about_keras_models.html
- https://keras.rstudio.com/articles/functional_api.html
- https://cran.rstudio.com/web/packages/keras/vignettes/sequential_model.html
- https://www.rdocumentation.org/packages/keras/versions/2.3.0.0/topics/layer_dense
- https://www.rdocumentation.org/packages/keras/versions/2.1.6/topics/compile