Lectures 4 & 5: Feed-forward Neural Networks

Deep Learning for Actuarial Modeling
36th International Summer School SAA
University of Lausanne

Ronald Richman, Salvatore Scognamiglio, Mario V. Wüthrich

2025-09-09

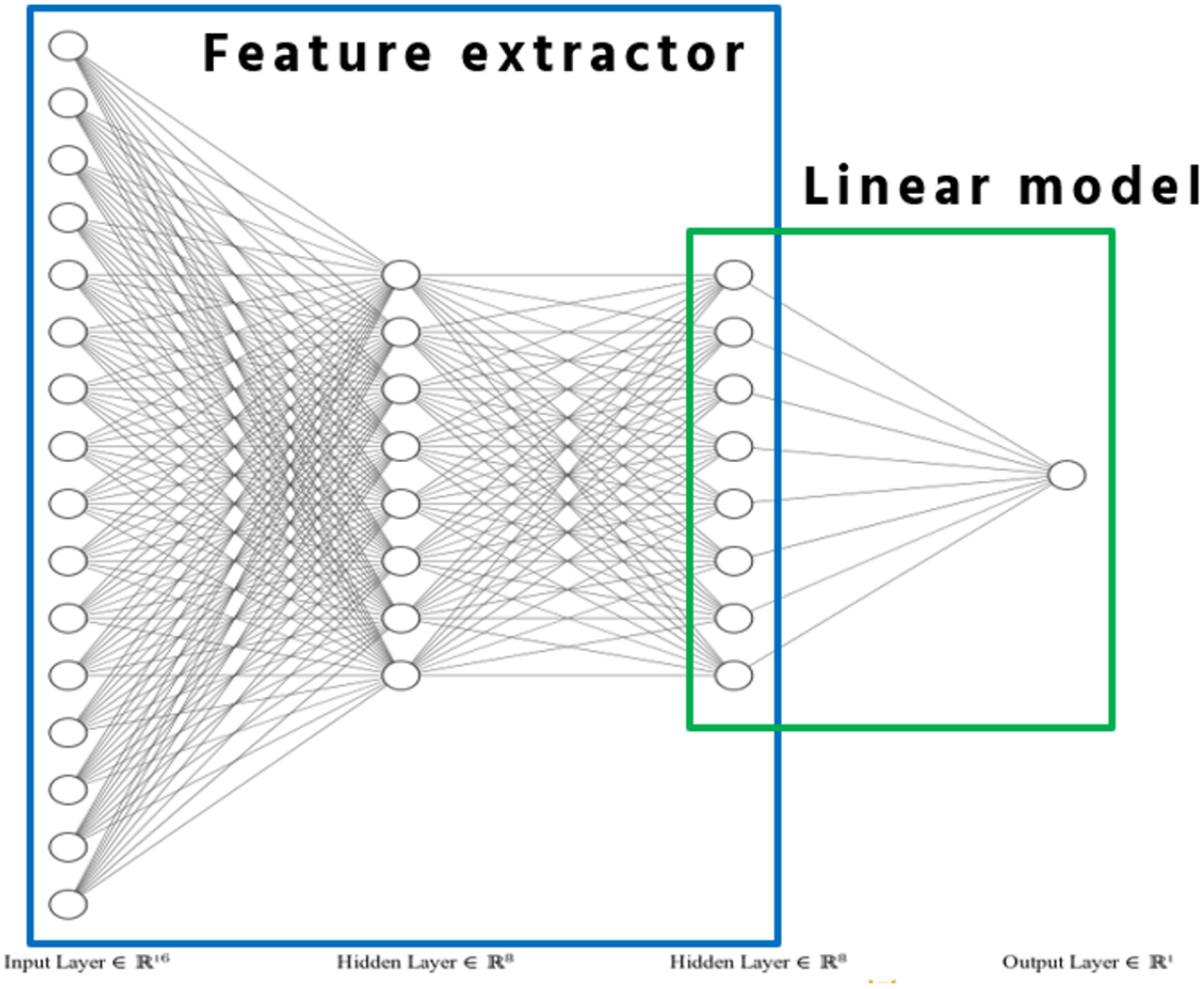
Feed-forward neural networks

Overview

This lecture introduces feed-forward neural networks (FNNs), it explains the building blocks of FNNs, and it shows how FNNs can be seen as an extension of GLMs. This lecture will also lay the foundation for more sophisticated deep learning methods. Moreover, we spend quite some time on explaining gradient descent fitting.

This lecture covers Chapter 5 of Wüthrich *et al.* (2025).

Feature extractor and GLM readout



- In a nutshell, networks perform *representation learning*, meaning that multi-layer networks learn in each layer of their architecture a *new representation* of the covariates $\boldsymbol{X}$ (inputs).
- This multi-layer module is illustrated by the *feature extractor* in the blue box of the graph.
- This newly learned representation of the feature extractor then serves as the new covariates for a (generalized) linear model, called *readout* and illustrated by the green box in the graph.
- Formally, one can write this as

$$\boldsymbol{X} \mapsto \mu(\boldsymbol{X}) = g^{-1}\left\langle \boldsymbol{w}^{(d+1)}, \boldsymbol{z}^{(d:1)}(\boldsymbol{X}) \right\rangle,$$

and we will introduce all building blocks of this archhitecture.

Feed-forward neural network architecture

- FNN architectures consist of (hidden) *FNN layers*

$$\boldsymbol{z}^{(m)} : \mathbb{R}^{q_{m-1}} \to \mathbb{R}^{q_m}, \qquad m \geq 1.$$

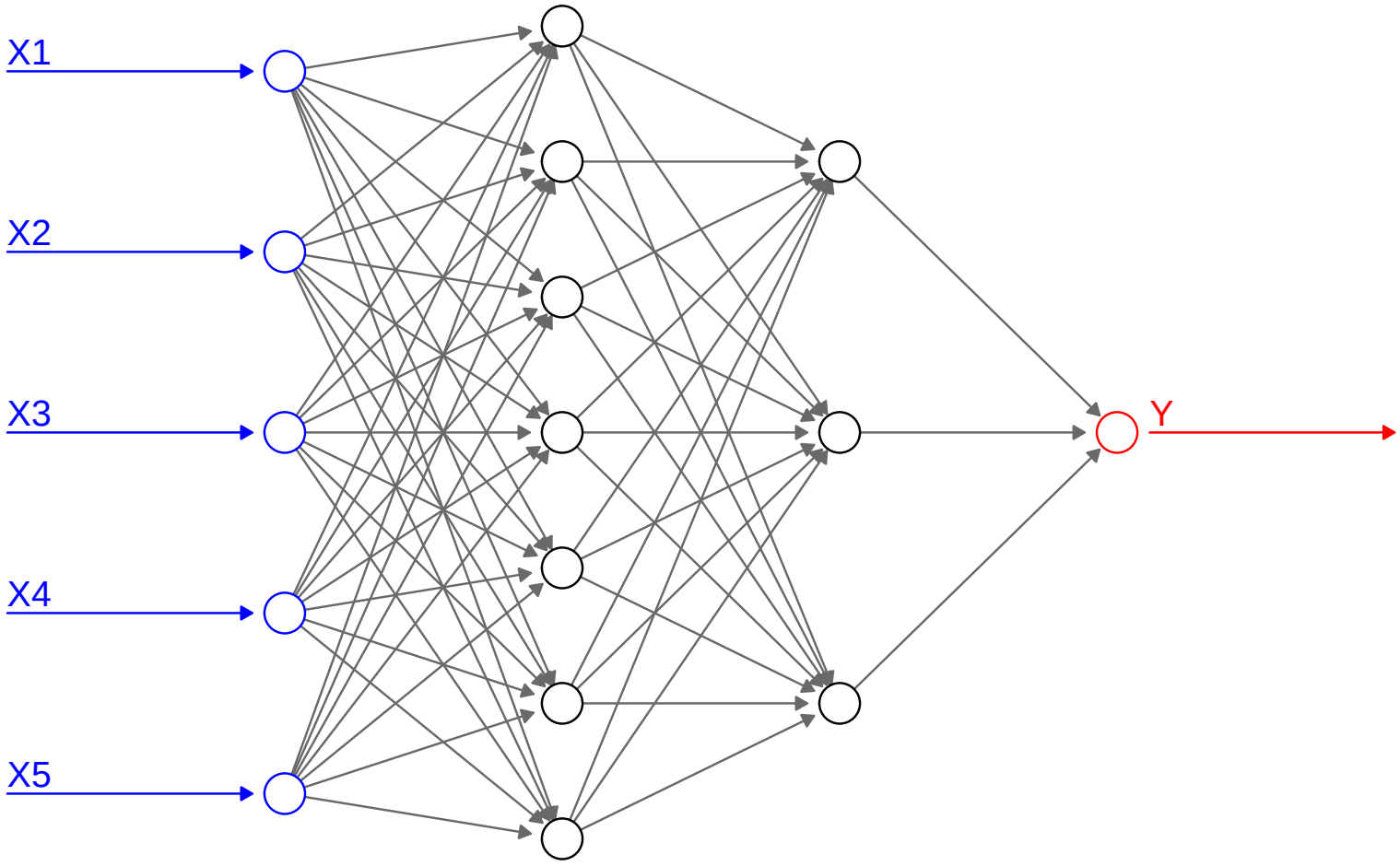
- Each FNN layer performs a *non-linear transformation* of the covariates.
- The main ingredients of such a FNN layer $\boldsymbol{z}^{(m)}$ are:
  (a) the number $q_m \in \mathbb{N}$ of *neurons* (also called *units*);
  (b) the non-linear *activation function* $\phi : \mathbb{R} \to \mathbb{R}$; and
  (c) the *network weights* (representing part of the model parameter ϑ).
- Items (a) and (b) are hyper-parameters selected by the modeler, and the network weights of item (c) are parameters that are learned during network training (model fitting).
- We discuss these items in detail below.

- Select d FNN layers $(\boldsymbol{z}^{(m)})_{m=1}^{d}$ with matching input and output dimensions.
- A *feature extractor* of *depth* d is obtained by the *composition*

$$\boldsymbol{X} \;\mapsto\; \boldsymbol{z}^{(d:1)}(\boldsymbol{X}) := \left(\boldsymbol{z}^{(d)} \circ \cdots \circ \boldsymbol{z}^{(1)}\right)(\boldsymbol{X}) \;\in\; \mathbb{R}^{q_d}.$$

- The input dimension of the 1st FNN layer $\boldsymbol{z}^{(1)}$ is the dimension of the covariates $\boldsymbol{X} \in \mathbb{R}^{q}$, that is, $q_0 = q$.
- The following graph illustrates a FNN architecture of depth $d = 2$:
  - with input dimension $q_0 = q = 5$, i.e., $\boldsymbol{X} = (X_1, \ldots, X_5)^\top$, and
  - units $q_1 = 7$ and $q_2 = 3$ in the two FNN layers.

Illustration of a FNN architecture of depth $d = 2$.



- The final step of the FNN architecture is the readout function on the feature extracted information

$$\boldsymbol{X} \;\mapsto\; \mu(\boldsymbol{X}) = g^{-1}\left\langle \boldsymbol{w}^{(d+1)}, \boldsymbol{z}^{(d:1)}(\boldsymbol{X}) \right\rangle,$$

with *readout parameter* $\boldsymbol{w}^{(d+1)} \in \mathbb{R}^{q_d+1}$ and inverse link function g^{-1}.

- There remains the discussion of the specification of the FNN layers

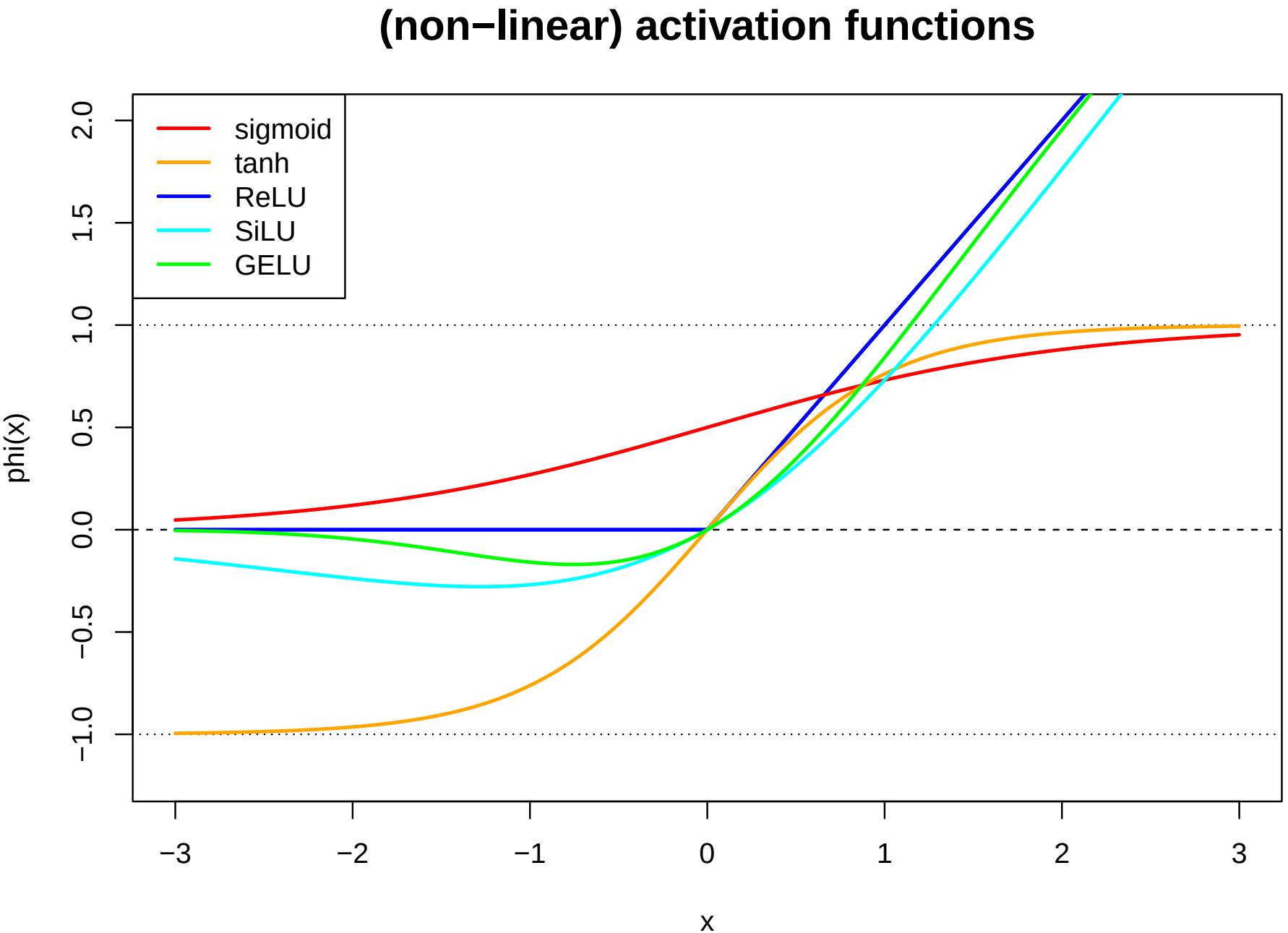
$$\boldsymbol{z}^{(m)} : \mathbb{R}^{q_{m-1}} \to \mathbb{R}^{q_m}, \qquad 1 \leq m \leq d.$$

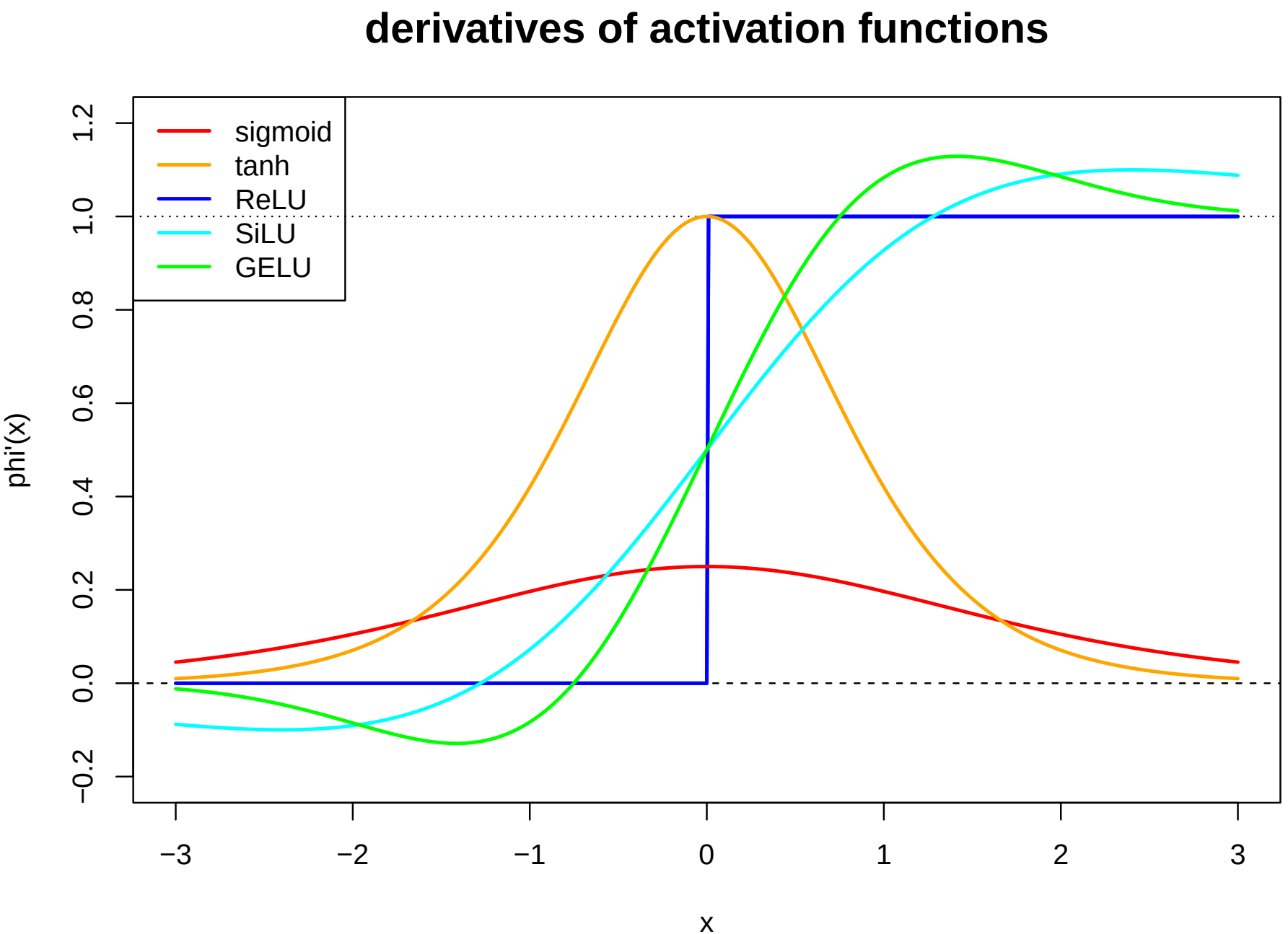
This is done next.

Activation functions

Since feature extractors should be able to extract non-linear structure of the original covariates, non-linear activation functions ϕ are needed. Commonly used examples are:

name	*activation function* ϕ	*derivative* ϕ'
sigmoid (logistic)	$\phi(x) = \sigma(x) = (1 + e^{-x})^{-1}$	$\phi(1 - \phi)$
hyperbolic tangent (tanh)	$\phi(x) = \tanh(x) = 2\sigma(2x) - 1$	$1 - \phi^2$
rectified linear unit (ReLU)	$\phi(x) = x\mathbf{1}_{\{x \geq 0\}}$	$\mathbf{1}_{\{x>0\}}, \quad x \neq 0$
sigmoid linear unit (SiLU)	$\phi(x) = x\sigma(x)$	$\sigma(x)(1 - \phi(x)) + \phi(x)$
Gaussian error linear unit (GELU)	$\phi(x) = x\Phi(x)$	$\Phi(x) + x\Phi'(x)$


(non−linear) activation functions
sigmoid
tanh
ReLU
SiLU
GELU
phi(x)
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−3
−2
−1
0
1
2
3
x


derivatives of activation functions
sigmoid
tanh
ReLU
SiLU
GELU
phi'(x)
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−3
−2
−1
0
1
2
3
x

- The presented activation functions have different properties, e.g.:
  - sigmoid and tanh are bounded which can be an advantage or a disadvantage, depending on the problem to be solved.
  - tanh is symmetric in zero which can be an advantage over the sigmoid in deep neural network fitting (because there is a natural calibration to zero that does not require to adjust biases).
  - ReLU is an activation function that is very popular in the machine learning community that can lead to sparsity in the activations, it is not differentiable in zero but it has a sub-gradient (it is convex).
  - SiLU is a smooth version of ReLU, but it is neither monotone nor convex.
  - GELU has recently gained popularity in transformer architectures.
- For fast gradient descent fitting it is important for ϕ to have a simple derivative.
- It is difficult to give a general advise for a specific selection of the 'best' activation function, but this is part of hyper-parameter tuning.

Feed-forward neural network layer

- Select an activation function ϕ.
- Define the FNN layer $\boldsymbol{z}^{(m)} : \mathbb{R}^{q_{m-1}} \to \mathbb{R}^{q_m}$ as follows

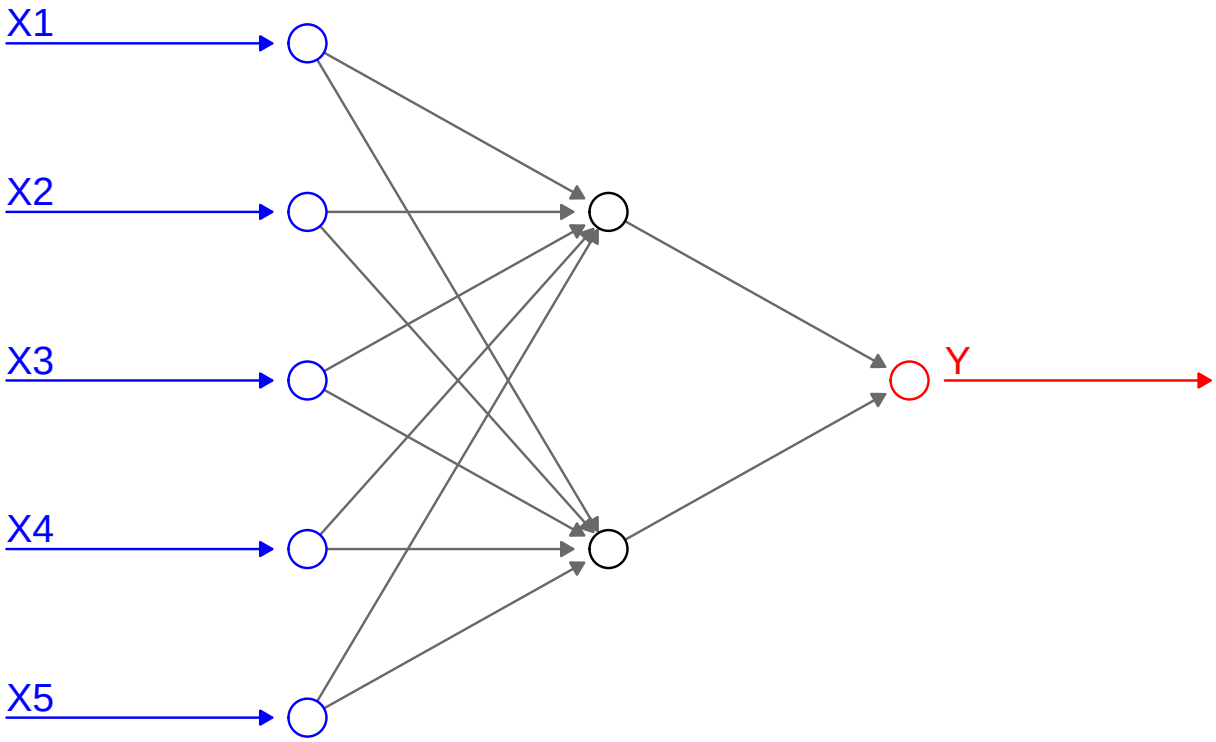
$$\boldsymbol{z}^{(m)}(\boldsymbol{x}) = \left(z_1^{(m)}(\boldsymbol{x}), \ldots, z_{q_m}^{(m)}(\boldsymbol{x})\right)^\top, \qquad \text{for } \boldsymbol{x} \in \mathbb{R}^{q_{m-1}},$$

with *neurons* (*units*), for $1 \leq j \leq q_m$,

$$z_j^{(m)}(\boldsymbol{x}) = \phi\left(w_{j,0}^{(m)} + \sum_{k=1}^{q_{m-1}} w_{j,k}^{(m)} x_k\right) =: \phi\langle \boldsymbol{w}_j^{(m)}, \boldsymbol{x}\rangle.$$

- $\boldsymbol{w}_j^{(m)} = (w_{j,0}^{(m)}, \ldots, w_{j,q_{m-1}}^{(m)})^\top \in \mathbb{R}^{q_{m-1}+1}$ are called *network weights*.
- Each neuron $z_j^{(m)}$ performs a *GLM operation* (data compression).

Illustration of GLM operations (data compressions) in the (two) neurons.



Since each data compression results in a loss of information, one needs multiple neurons to extract different relevant information.

Summary: Feed-forward neural network architecture

- Each FNN layer $\boldsymbol{z}^{(m)}$ has network weights $(\boldsymbol{w}_1^{(m)}, \ldots, \boldsymbol{w}_{q_m}^{(m)})$ of dimension $q_m(q_{m-1}+1)$.
- Collecting all network weights of all layers, including the readout parameter, gives all the network weights

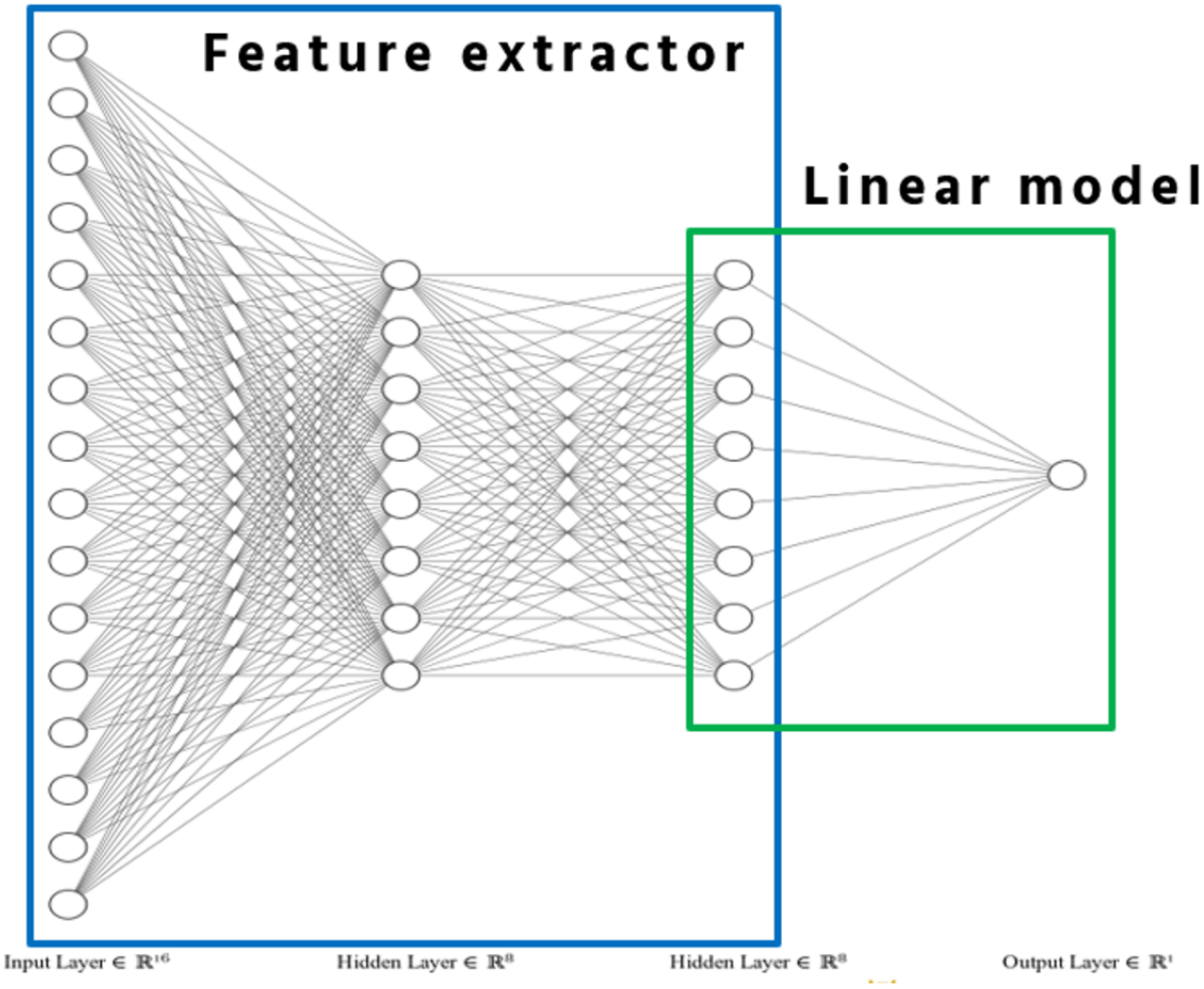
$$\vartheta = \left(\boldsymbol{w}_1^{(1)}, \ldots, \boldsymbol{w}_{q_d}^{(d)}, \boldsymbol{w}^{(d+1)}\right) \in \mathbb{R}^r,$$

  of dimension $r = \sum_{m=1}^{d} q_m(q_{m-1}+1) + (q_d+1)$.
- Indicating the network parameter results in the *FNN architecture*

$$\boldsymbol{X} \mapsto \mu_\vartheta(\boldsymbol{X}) = g^{-1}\left\langle \boldsymbol{w}^{(d+1)}, \boldsymbol{z}^{(d:1)}(\boldsymbol{X})\right\rangle.$$

- These FNN architectures give a class of parametric regression functions $\mathcal{M} = \{\mu_\vartheta\}_\vartheta$, parametrized through the network weights $\vartheta \in \mathbb{R}^r$.
- In summary, a FNN architecture is determined by the hyper-parameters:
  (a) the depth d of the FNN architecture;
  (b) the numbers q_m of neurons in the hidden layers $\boldsymbol{z}^{(m)}$, $1 \leq m \leq d$;
  (c) the non-linear activation function $\phi : \mathbb{R} \to \mathbb{R}$ in all the neurons; and
  (d) the output activation g^{-1}.
- The network weights $\vartheta \in \mathbb{R}^r$ represent the model parameter that parametrizes this family $\mathcal{M} = \{\mu_\vartheta\}_\vartheta$ of FNN architectures.

Feature extractor and GLM readout, revisited



Example

- We discuss the above FNN example of depth $d = 2$:
  - It has a 16-dimensional covariate vector $\boldsymbol{X}$ providing $q_0 = q = 16$.
  - The 1st hidden layer $\boldsymbol{z}^{(1)} : \mathbb{R}^{q_0} \to \mathbb{R}^{q_1}$ has $q_1 = 8$ neurons providing $8 \cdot 17 = 136$ network weights.
  - The 2nd hidden layer $\boldsymbol{z}^{(2)} : \mathbb{R}^{q_1} \to \mathbb{R}^{q_2}$ has $q_2 = 8$ neurons providing $8 \cdot 9 = 72$ network weights.
  - The readout parameter has dimension 9.
  - Altogether this FNN architecture has network weights $\vartheta \in \mathbb{R}^r$ of dimension $r = 217$.

- We implement this FNN architecture in R-Keras using the log-link $g(\cdot) = \log(\cdot)$.
- We additionally allow for a multiplicative exposure scaling using the volumes $v > 0$ (for more discussion, see the Poisson GLM).

```
# load the necessary libraries


library(tensorflow)
library(keras)


# keras3 requires small adaptions to the code below because indices in
↪  arrays are shifted from keras2 to keras3
```

```
# define a FNN architecture function of depth d=2 (qq gives the units)
FNN <- function(seed, qq){
    k_clear_session()
    set.seed(seed)
    set_random_seed(seed)
    Design  <- layer_input(shape = c(qq[1]), dtype = 'float32')
    Volume  <- layer_input(shape = c(1), dtype = 'float32')
    Network = Design %>%
          layer_dense(units=qq[2], activation='tanh') %>%
          layer_dense(units=qq[3], activation='tanh') %>%
          layer_dense(units=1, activation='exponential')
    Response = list(Network, Volume) %>% layer_multiply()
    keras_model(inputs = c(Design, Volume), outputs = c(Response))
    }

# define the FNN architecture/model from the above graph
(model <- FNN(seed=100, qq=c(16, 8, 8)))
```

```
Model: "model"
__________________________________________________________________________________________________
 Layer (type)                Output Shape              Param #   Connected to
==================================================================================================
 input_1 (InputLayer)        [(None, 16)]              0         []
 dense_2 (Dense)             (None, 8)                 136       ['input_1[0][0]']
 dense_1 (Dense)             (None, 8)                 72        ['dense_2[0][0]']
 dense (Dense)               (None, 1)                 9         ['dense_1[0][0]']
 input_2 (InputLayer)        [(None, 1)]               0         []
 multiply (Multiply)         (None, 1)                 0         ['dense[0][0]',
                                                                  'input_2[0][0]']
==================================================================================================
Total params: 217 (868.00 Byte)
Trainable params: 217 (868.00 Byte)
Non-trainable params: 0 (0.00 Byte)
__________________________________________________________________________________________________
```

Universality theorems

- The main universality theorem states that 'any compactly supported continuous (regression) function can be approximated arbitrarily well by a suitable (and sufficiently large) FNN'.
- This approximation can be w.r.t. different norms and the assumptions for such a statement to hold are comparably weak, e.g., the sigmoid activation function leads to a class of FNNs that are universal in the above sense.
- For precise mathematical statements and proofs about these denseness results, see Cybenko (1989), Hornik, Stinchcombe and White (1989), Hornik (1991) and Leshno *et al.* (1993); and there is a vast literature with similar statements and proofs.

Consequences for networks

- The universality statements imply that basically any regression function can be approximated arbitrarily well within the class of FNNs.
- This sounds very promising:
  - It means that the class of FNNs is very rich and flexible.
  - No matter what the specific true data generating model looks like, there is a FNN that is similar to this data generating mechanism, and our aim is to find it using the learning sample $\mathcal{L}$ that has generated that data.

- Unfortunately, there is a backside of this coin:
  - There is no hope to find 'the best' FNN (on finite samples), and there are infinitely many (almost equally) good candidate FNNs. Typically, one can only distinguish clearly better from clearly worse on finite samples.
  - The model selection/fitting problem is very high-dimensional and non-convex (for any reasonable choice of objective function).
  - Model selection within the class of FNN involves several elements of randomness, e.g., a fitting algorithm needs to be (randomly) initialized and this impacts the selected solution. To be able to replicate results, seeds of random number generators need to be stored.
- Some of the previous items will only become clear once we have introduced stochastic gradient descent fitting, and one should keep these (critical) items in mind for the discussions below.

Gradient descent algorithm

- Based on the non-uniqueness of a best FNN approximation to the true model on finite samples, one tries to find a *reasonably good* FNN approximation to the true data generating mechanism.
- Reasonably good means that it usually outperforms a classical GLM, but at the same time there are infinitely many other FNNs that have a similarly good predictive performance (generalization to new data).
- Due to the non-convexity and the complexity of the problem, computational aspects are crucial in designing a good FNN training algorithm.
- The main tool is *stochastic gradient descent* (SGD).

Objective function

- Choose a strictly consistent loss function L for mean estimation.
- Denote the learning sample by $\mathcal{L} = (Y_i, \boldsymbol{X}_i, v_i)_{i=1}^n$.
- The empirical loss in network parameter ϑ on $\mathcal{L}$ is defined by

$$L(\vartheta; \mathcal{L}) := \frac{1}{n} \sum_{i=1}^{n} \frac{v_i}{\varphi} L(Y_i, \mu_\vartheta(\boldsymbol{X}_i)),$$

  where μ_ϑ is a FNN with network weights $\vartheta \in \mathbb{R}^r$.
- We add the learning sample $\mathcal{L}$ to the loss notation $L(\vartheta; \mathcal{L})$ because for SGD we will vary over different learning (sub-)samples.

Gradient descent step

- Assume we have network weights $\vartheta^{[t]} \in \mathbb{R}^r$ at step t providing the empirical loss $L(\vartheta^{[t]}; \mathcal{L})$.
- The goal is to *stepwise adaptively* improve these network weights

$$\vartheta^{[t]} \;\mapsto\; \vartheta^{[t+1]},$$

  such that the empirical loss decreases in each step $t \to t+1$.
- Determine a small perturbation of $\vartheta^{[t]}$ leading to a *local improvement*.
- Local changes can be described by 1st order Taylor expansions

$$L\left(\vartheta^{[t+1]}; \mathcal{L}\right) \;\approx\; L\left(\vartheta^{[t]}; \mathcal{L}\right) + \nabla_\vartheta L\left(\vartheta^{[t]}; \mathcal{L}\right)^\top \left(\vartheta^{[t+1]} - \vartheta^{[t]}\right),$$

  for $\vartheta^{[t+1]}$ close to $\vartheta^{[t]}$.
- This becomes minimal if the last term is as negative as possible.

- Thus, the update in the network weights should point into the opposite direction of the gradient.
- This motivates the *gradient descent update*

$$\vartheta^{[t]} \;\mapsto\; \vartheta^{[t+1]} = \vartheta^{[t]} - \varrho_{t+1} \nabla_\vartheta L\left(\vartheta^{[t]}; \mathcal{L}\right),$$

  where $\varrho_{t+1} > 0$ is a (small) *learning rate*, also called *step size*.
- The learning rate needs to be small for the 1st order Taylor expansion to be a valid approximation. But the learning rate should not be too small, otherwise we need too many gradient descent steps.
- The initial value $\vartheta^{[0]}$ of the gradient descent algorithm should be selected at random to avoid starting the algorithm in a saddlepoint of the loss surface $\vartheta \mapsto L(\vartheta; \mathcal{L})$.
- Popular initializer: glorot-uniform of Glorot and Bengio (2010) selecting a random uniform initialization adapted to the layer sizes.

Open points

The following points need to be discussed:

(a) Covariate pre-processing.

(b) Efficient calculations of the gradients $\nabla_\vartheta L(\vartheta; \mathcal{L})$.

(c) Selection of the learning rate and higher order Taylor approximations.

(d) A stopping rule for the algorithm.

(e) Regularization and drop-out.

(f) Dealing with big data, i.e., big learning samples $\mathcal{L}$.

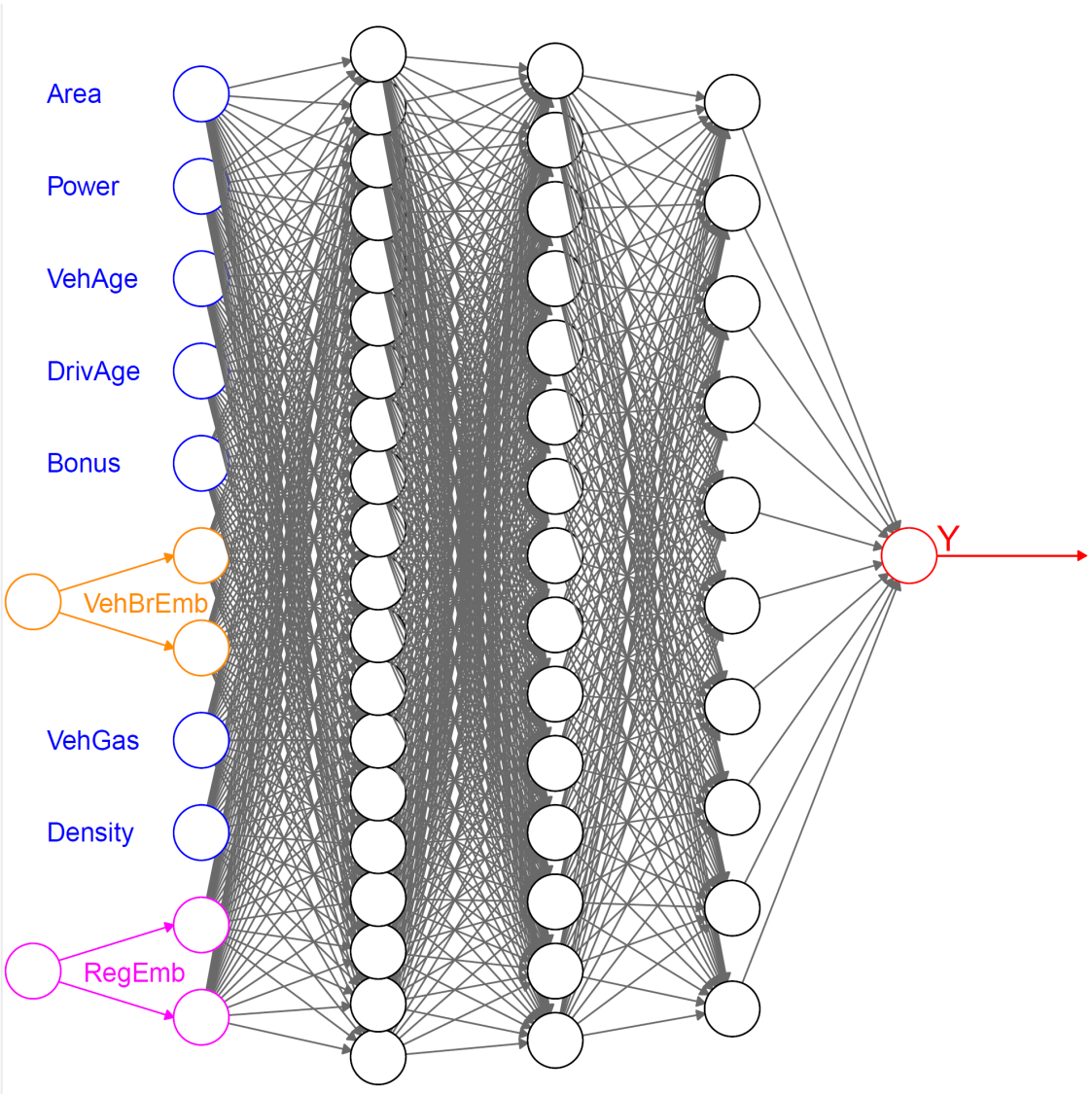
These items are discussed in the following paragraphs.

Covariate pre-processing

- Covariate pre-processing is discussed in detail in the next lecture and we just briefly highlight some important points here.
- It is important for the gradient descent algorithm to work properly that all covariate components live on the *same scale*. Otherwise some covariate components will dominate the gradient, and the algorithm is not able to extract systematic structure from all covariate components.
- For this reason, continuous covariates should be *standardized* or the *MinMaxScaler* should be applied (see next lecture).
- If the skewness of a continuous covariate is large, i.e., if it lives on different scales (magnitudes), one should first apply a log-transformation.

- For categorical covariates usually one-hot encoding is used in the first place (see next lecture).
- *High-cardinality categorical covariates* lead to large input dimensions q_0 to the feature extractor. This is generally problematic in network fitting as it gives a high potential for over-fitting.
- Generally, we recommend to use an *entity embedding* with a low-dimensional embedding dimension b for categorical covariates.
- The entity embedded variables are concatenated with the continuous ones, to jointly enter the feature extractor of the FNN architecture.
- More details are provided in the next lecture, but we briefly show an example.

Example: two-dimensional embedding ($b = 2$) of VehBrand and Region.



Gradient calculation via back-propagation

- Generally, gradient computations $\nabla_\vartheta L(\vartheta; \mathcal{L})$ are high-dimensional and computationally intensive. The network weights ϑ enter the readout and the different FNN layers $(\boldsymbol{z}^{(m)})_{m=1}^{d}$ of the feature extractor.
- Theoretically, the gradient can be worked out using standard calculus, but through the iterated application of the chain-rule the computations become very tedious.
- The workhorse to compute these gradients efficiently is the *back-propagation method* of Rumelhart, Hinton and Williams (1986). Mathematically speaking, the back-propagation method is a clever re-parametrization to efficiently compute these gradients recursively.
- We skip more technical details about back-propagation, but refer to the (ready-to-use) standard software, such as TensorFlow.

Learning rate and higher order Taylor approximations

- The gradient descent algorithm is based on a 1st order Taylor expansion.
- 1st order Taylor expansions compute slopes and, hence, directions of optimal local updates.
- The optimal (directional) learning rates $\varrho_{t+1} > 0$ are determined by the curvature of the loss surface described by 2nd order derivatives (Hessians) of the empirical loss $\vartheta \mapsto L(\vartheta; \mathcal{L})$, i.e., the Newton method.
- Unfortunately, it is computationally unfeasible to compute Hessians in (bigger) FNNs, therefore, we cannot determine the optimal learning rates by 2nd order derivatives.

Momentum based methods

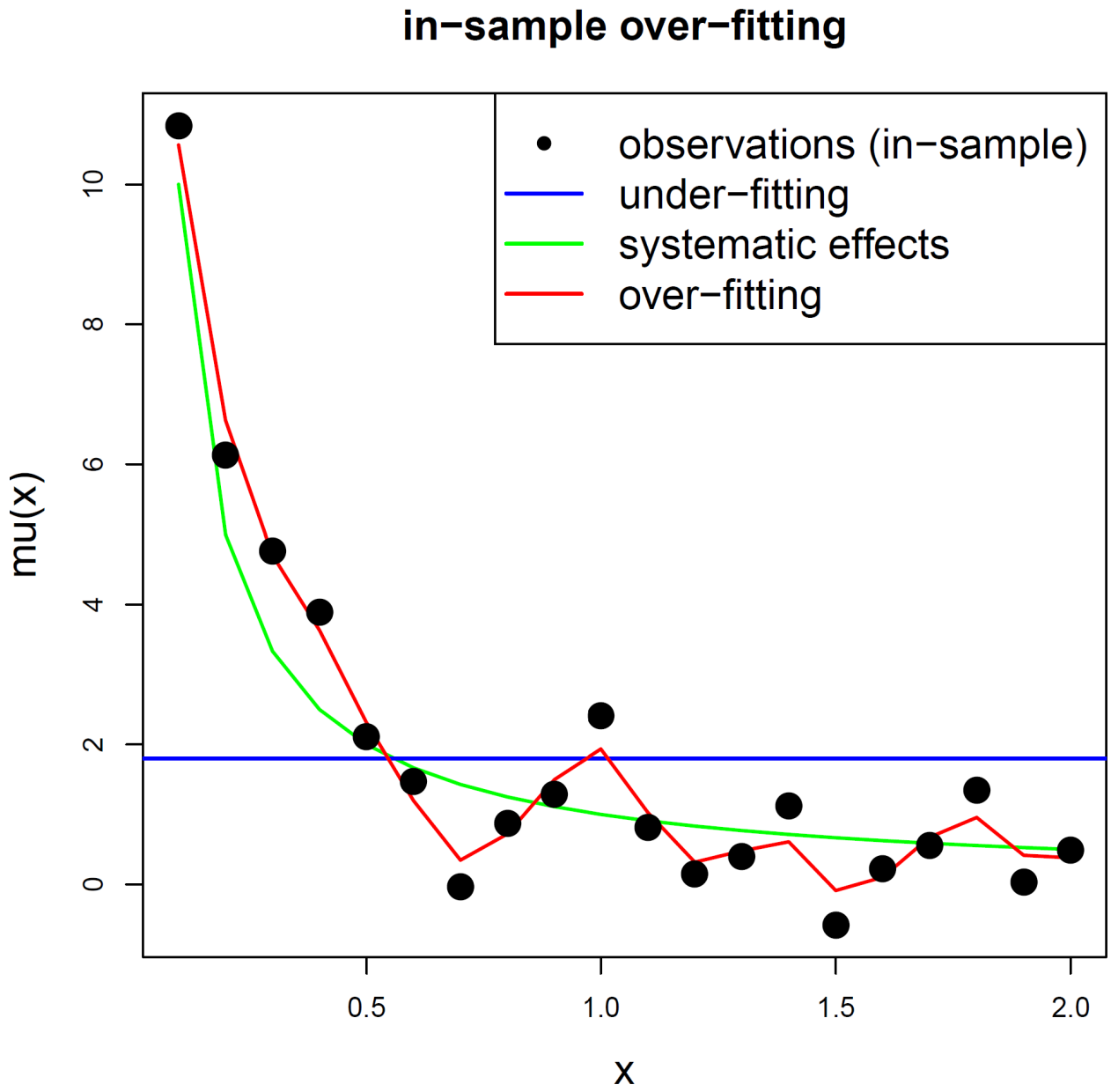
- In physics, 1st order derivatives are related to speed and 2nd order derivatives to acceleration.
- Since one cannot compute 2nd order derivatives, inspired by physics, one mimics how speed and velocity build up by computing momentums from past velocities. This is a way of mimicking 2nd order derivatives.
- Standard momentum based algorithms are **rmsprop** or **adam**; see Hinton, Srivastava and Swersky (2014), Kingma and Ba (2017).
- We do not discuss this any further here, but we just use the implemented methods, usually **adam** or its Nesterov (2007) accelerated version **nadam**.
- For transformers, there are more specialized gradient descent methods, e.g., **adamW** of Loshchilov and Hutter (2019) which better adapts to problems where the variables live on different scales.

Early stopping

- Having a reasonably large FNN architecture, is is very flexible because it is capable to approximate a fairly large function class.
- This implies that computing the MLE

$$\widehat{\vartheta}^{\mathrm{MLE}} \in \arg\min_{\vartheta} L(\vartheta; \mathcal{L}) = \arg\min_{\vartheta} \frac{1}{n} \sum_{i=1}^{n} \frac{v_i}{\varphi} L(Y_i, \mu_\vartheta(\boldsymbol{X}_i)),$$

  is *not* a sensible problem.
- This MLE fitted FNN does not only extract the *structural part* (*systematic effects*) from the learning sample $\mathcal{L} = (Y_i, \boldsymbol{X}_i, v_i)_{i=1}^n$, but it also largely adapts to the *noisy part* (*pure randomness*) in $\mathcal{L}$.
- Obviously, such a FNN will badly generalize and it will have a poor predictive performance on out-of-sample test data $\mathcal{T}$.


in−sample over−fitting
observations (in−sample)
under−fitting
systematic effects
over−fitting
mu(x)
0
2
4
6
8
10
0.5
1.0
1.5
2.0
x

- The above figure gives an example that in-sample over-fits:
  - The black dots are the observed responses Y_i (in learning sample $\mathcal{L}$).
  - The true regression function is shown in green color.
  - The red graph shows a fitted regression model that in-sample over-fits to the learning sample $\mathcal{L}$. It follows the black dots quite closely, significantly deviating from the true green regression function.
- Out-of-sample (repeating this experiment), the black dots likely also lie on the other side of the green line. Thus, the red estimated model badly generalizes.
- Consequently, within a highly flexible model class we need to try to find a model that only extracts the systematic part from a noisy sample.
- **Early stopping** is the crucial technology that solves this problem.

- Coming back to the gradient of the empirical loss

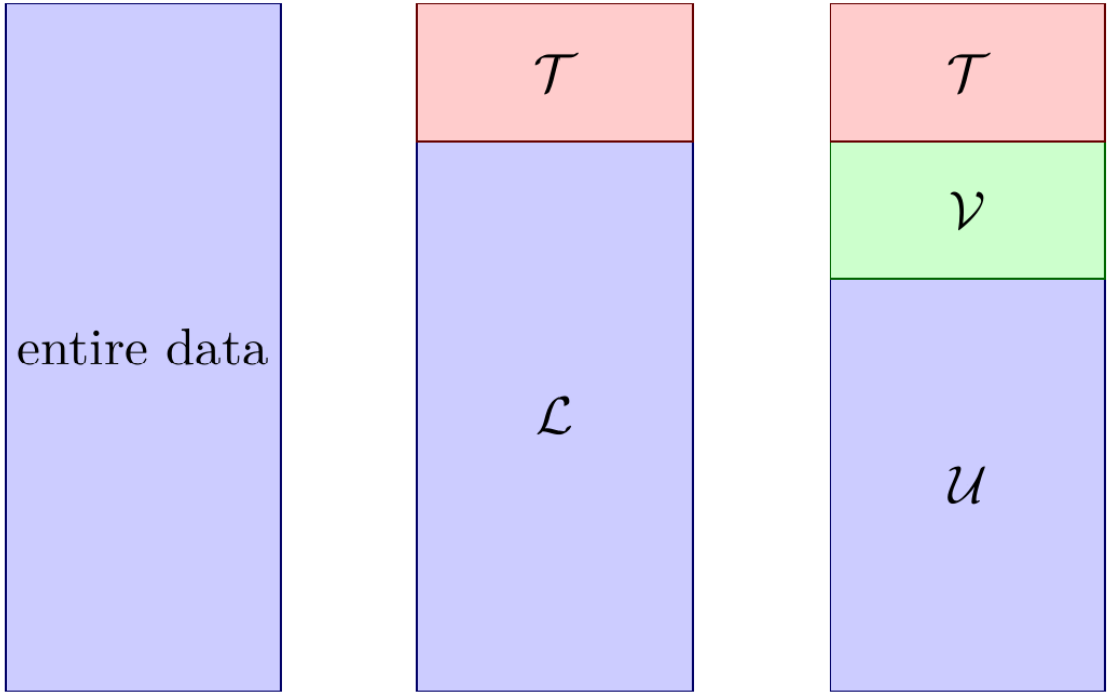
$$\nabla_\vartheta L(\vartheta; \mathcal{L}) = \frac{1}{n} \sum_{i=1}^{n} \frac{v_i}{\varphi} \nabla_\vartheta L(Y_i, \mu_\vartheta(\boldsymbol{X}_i)).$$

- This gradient consists of a sum of individual gradients over all instances $1 \leq i \leq n$.
- Systematic effects impact many individual instances (otherwise they would not be systematic).
- At the beginning of the gradient descent algorithm, before having found these systematic effects, they dominate the gradient descent steps.
- Once these systematic effects are found, the relative importance of instance-individual factors (noise) starts to increase.
- This is precisely the time-point to early stop the algorithm.

Training, validation and test samples

- Implementation of early stopping requires a careful treatment of the available learning sample $\mathcal{L}$.
- For this we partition the learning sample $\mathcal{L}$ at random into a *training sample* $\mathcal{U}$ and a *validation sample* $\mathcal{V}$.
- The training sample $\mathcal{U}$ is used for computing the gradient descent steps, and the validation sample $\mathcal{V}$ is used to track over-fitting by an instantaneous (out-of-sample) validation analysis.
- The *test sample* $\mathcal{T}$ is used to compare different models, i.e., it is not used during model fitting.

Training, validation and test samples: illustration



Training sample $\mathcal{U}$, validation sample $\mathcal{V}$ and test sample $\mathcal{T}$.

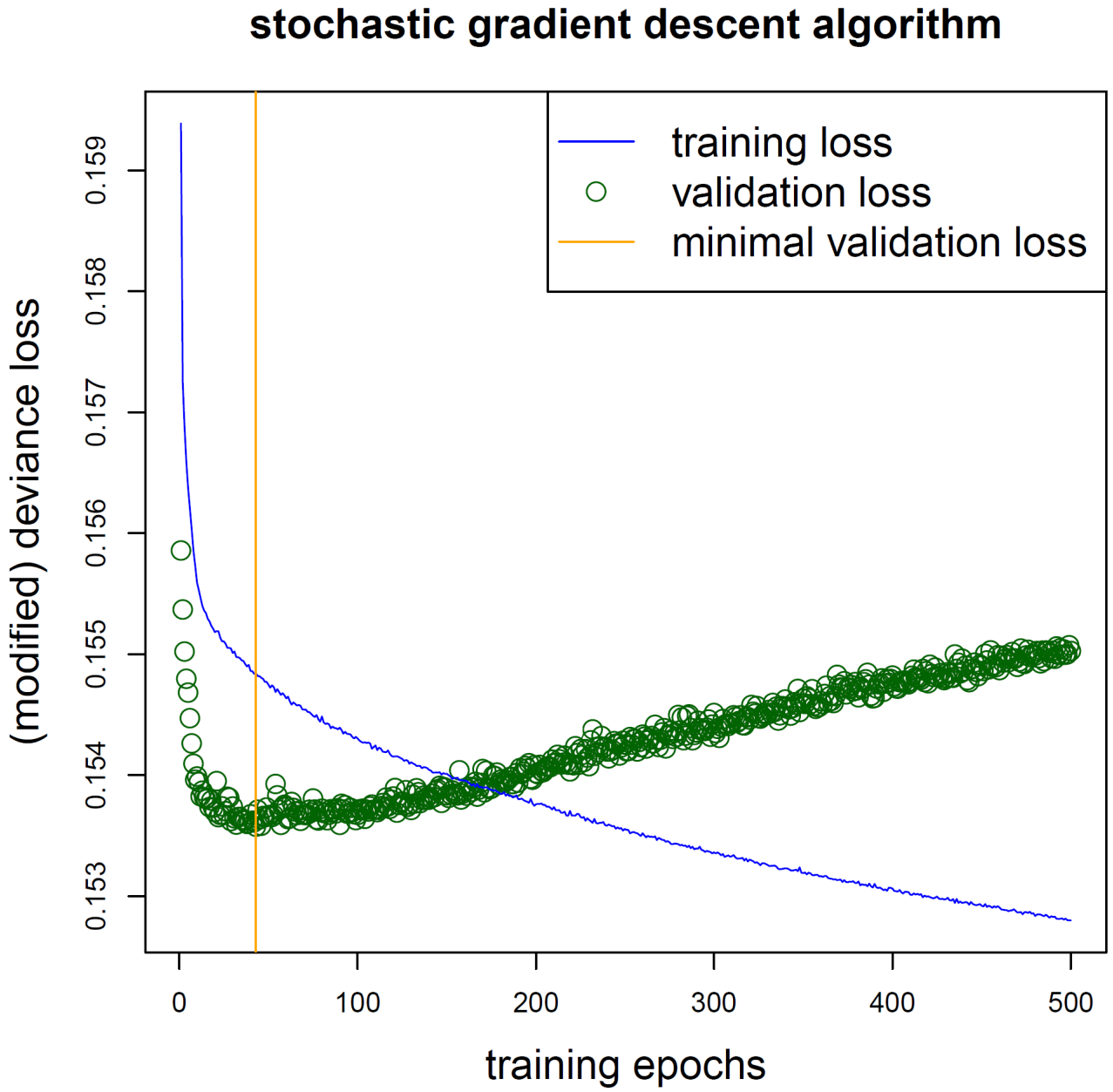
- Perform the gradient descent steps **only** on the training sample $\mathcal{U}$

$$\nabla_\vartheta L(\vartheta;\mathcal{U}) = \frac{1}{|\mathcal{U}|}\sum_{i\in\mathcal{U}}\frac{v_i}{\varphi}\nabla_\vartheta L(Y_i,\mu_\vartheta(\boldsymbol{X}_i)).$$

- Perform an instantaneous out-of-sample validation on $\mathcal{V}$

$$L(\vartheta^{[t]};\mathcal{V}) = \frac{1}{|\mathcal{V}|}\sum_{i\in\mathcal{V}}\frac{v_i}{\varphi}L(Y_i,\mu_{\vartheta^{[t]}}(\boldsymbol{X}_i)).$$

- Naturally, the *training loss* $L(\vartheta^{[t]};\mathcal{U})$ should decrease for $t\to\infty$.
- The *validation loss* $L(\vartheta^{[t]};\mathcal{V})$ decreases as long as systematic effects are learned, then it increases (deteriorates) once the noisy part is learned.
- This change of behavior gives the *early stopping point* $t^\star$, and the network weights are estimated by $\widehat{\vartheta}=\vartheta^{[t^\star]}$; see next graph.


stochastic gradient descent algorithm
training loss
validation loss
minimal validation loss
(modified) deviance loss
0.153
0.154
0.155
0.156
0.157
0.158
0.159
0
100
200
300
400
500
training epochs

Concluding remarks on early stopping

- The validation sample $\mathcal{V}$ should be sufficiently large so that a reliable validation loss $L(\vartheta^{[t]}; \mathcal{V})$ can be calculated, e.g., 10% or 20% of the learning sample $\mathcal{L}$.
- The difference $L(\vartheta^{[t]}; \mathcal{U}) - L(\vartheta^{[t]}; \mathcal{V})$ can have any sign, this depends on the specific random choices of $\mathcal{U}$ and $\mathcal{V}$.
- Practically, for gradient descent training, one installs a so-called **callback** that saves every weight $\vartheta^{[t]}$ which decreases the validation loss $L(\vartheta^{[t]}; \mathcal{V})$. After running the algorithm one *calls back* the weight $\vartheta^{[t^\star]}$ with the minimal validation loss.

Regularization and drop-out

- There is no difficulty in using a regularized loss in gradient descent fitting; we discuss regularization in a later lecture.
- A popular method to prevent from (in-sample) over-fitting is *drop-out* by Srivastava *et al.* (2014) and Wager, Wang and Liang (2013).
- Drop-out is an additional network layer between two FNN layers that removes neurons $z_j^{(m)}$ at random from the network *(only) during gradient descent training* (and in each gradient descent step resampled). This regularizes gradient descent training and can lead to better predictive models.

Stochastic gradient descent

- Typically, gradient computations on large samples involve large matrix multiplications. These are very slow which hinders fast network fitting.
- For this reason, use a *stochastic gradient descent* (SGD) algorithm.
- For SGD one chooses a fixed *batch size* $s \in \mathbb{N}$, and randomly partitions the training sample $\mathcal{U} = (Y_i, \boldsymbol{X}_i, v_i)_{i=1}^n$ into *(mini-)batches* $\mathcal{U}_1, \ldots, \mathcal{U}_{\lfloor n/s \rfloor}$ of roughly the same size s.
- One then considers the SGD updates

$$\vartheta^{[t]} \;\mapsto\; \vartheta^{[t+1]} = \vartheta^{[t]} - \varrho_{t+1} \nabla_\vartheta L\left(\vartheta^{[t]}; \mathcal{U}_k\right),$$

cyclically visiting the batches $(\mathcal{U}_k)_{k=1}^{\lfloor n/s \rfloor}$.

- The size $s \in \mathbb{N}$ of the batches $(\mathcal{U}_k)_{k=1}^{\lfloor n/s \rfloor}$ should neither be too small nor too big.
- Assuming i.i.d. observations $(Y_i, \boldsymbol{X}_i, v_i)_{i=1}^{s}$, the law of large numbers gives the *locally* optimal gradient descent step for batch size $s \to \infty$.
- But computational reasons force us to choose small(er) batch sizes. These may give certain *erratic* gradient descent updates.
- However, some erratic steps can be beneficial for finding better network weights, as long as these erratic steps are not too numerous (and not too large): SGD only always considers the next best step, but this may miss the long-run optimal step.
- Certain erratic steps may help one to escape from saddlepoints or an unwanted local optimal behavior.
- That is, a few erratic steps lead to better fitted FNNs (this is similar to explore vs. exploit in reinforcement learning).

A summary on network training

We have now introduced the whole FNN toolbox, and we are ready to apply our first FNN regression model!

- The first attempts on real data will likely result in a disappointment because working with FNNs requires quite some practical experience.
- There is the recurrent question of how to select good FNN architectures.
- A general principle is to select a network architecture that is not too small to be sufficiently flexible to approximate all potentially suitable regression functions.
- Generally, it is a bad guidance to attempt for a minimal FNN.

- Usually, there are many different, roughly equally good FNN approximations to a given real data problem, and the SGD algorithm can only find (some of) those if it has sufficiently many degrees of freedom to exploit the (full) parameter space.
- This contradicts parsimony, and is against actuarial thinking, but it is required for successful SGD fitting.
- Optimizing neural network architectures should not be the target, and ensembling (discussed below) helps to reduce model variations.

FNN example: French MTPL data

- We revisit the French MTPL claims count data set 'freMTPL2freq' of Dutang, Charpentier and Gallic (2024).
- We use the data cleaning procedure of Wüthrich and Merz (2023).
- We model these MTPL claims counts by fitting a FNN regression function with log-link, and using the Poisson deviance loss.
- We use one-hot encoding for categorical covariates (in detail explained in the next lecture).
- We use standardization for continuous covariates (in detail explained in the next lecture).
- We benchmark the FNN results by the GLM ones.

One-hot encoding (explained later)

- We start by a one-hot encoding function that also adds suitable labels to the columns of the design matrix.

```
# function for one-hot encoding of categorical covariates;
# this is based on the command to_categorical from the Keras library

PreProcess.OneHot <- function(var1, name, dat2){
   names(dat2)[names(dat2) == var1]  <- "V1"
   XX <- data.frame(to_categorical(as.integer(dat2$V1)))
   colnames(XX) <- paste0(name, c(1:ncol(XX)))
   names(dat2)[names(dat2) == "V1"]  <- var1
   cbind(dat2, XX)
   }
```

Standardization (explained later)

- The following code standardizes the continuous covariates by centering with the empirical mean and scaling with the standard deviation.

```
# standardization of continuous covariates
PreProcess.Continuous <- function(var1, dat2){
   names(dat2)[names(dat2) == var1]  <- "V1"
   dat2$X <- as.numeric(dat2$V1)
   dat2$X <- (dat2$X-mean(dat2$X))/sd(dat2$X)
   names(dat2)[names(dat2) == "V1"]  <- var1
   names(dat2)[names(dat2) == "X"]   <- paste(var1,"X", sep="")
   dat2
   }
```

- We apply this standardization *simultaneously* to the learning and to the test data sets. For new data, one needs to store the scaling constants to be able to pre-process new data in the identical way.

Load TensorFlow and Keras libraries

```
library(tensorflow)
library(keras)       # this notebook uses Keras 2
#library(keras3)     # Keras3 needs a slight adaption to the code below
```

- This uses Keras 2, but there is also Keras 3.
- A main difference between the two Keras versions is that arrays run from $0 : n-1$ in the latter compared to $1 : n$ in the former. This needs quite some care!

Covariate pre-processing for FNN fitting

```
Features.PreProcess <- function(dat2){
   dat2 <- PreProcess.Continuous("Area", dat2) # transformed to continuous
   dat2 <- PreProcess.Continuous("VehPower", dat2)
   dat2$VehAge <- pmin(dat2$VehAge,20)      # censoring at age 20
   dat2 <- PreProcess.Continuous("VehAge", dat2)
   dat2$DrivAge <- pmin(dat2$DrivAge,90)    # censoring at age 90
   dat2 <- PreProcess.Continuous("DrivAge", dat2)
   dat2$BonusMalus <- pmin(dat2$BonusMalus,150) # censoring at level 150
   dat2 <- PreProcess.Continuous("BonusMalus", dat2)
   dat2 <- PreProcess.OneHot("VehBrand", "B", dat2)
   dat2$VehGasX <- as.integer(dat2$VehGas)-1    # this is binary
   dat2$Density <- round(log(dat2$Density),2)   # log-scale and censoring
   dat2 <- PreProcess.Continuous("Density", dat2)
   PreProcess.OneHot("Region", "R", dat2) }
#
dat <- Features.PreProcess(dat)
```

Constructing learning and test samples

```
## learning and test sample partition
learn <- dat[which(dat$LearnTest=='L'),]
test  <- dat[which(dat$LearnTest=='T'),]
```

- **Important:** Learning and test samples use the identical pre-processing.
- We use the learning-test sample partition of Wüthrich and Merz (2023), and all results are directly comparable to the ones in that reference.
- Generally, the ordering of the data should be randomized, e.g., if the data is ordered w.r.t. the accident year. This is important for the SGD algorithm to work properly (we come back to this below).
- One could also stratify the allocation to learning and test samples, so that they are more similar, e.g., w.r.t. large claims.

Prepare data for FNN fitting

```
# considered covariates
features <- c("AreaX", "VehPowerX", "VehAgeX", "DrivAgeX", "BonusMalusX",
              "VehGasX", "DensityX", paste0("B", c(1:11)), paste0("R",
              ↪  c(1:22)))

# learning and test samples
Xlearn <- as.matrix(learn[, features])  # design matrix learning sample
Xtest  <- as.matrix(test[, features])   # design matrix test sample
Ylearn <- as.matrix(learn$ClaimNb)      # response learning sample
Ytest  <- as.matrix(test$ClaimNb)       # response test sample
Vlearn <- as.matrix(learn$Exposure)     # time exposure learning sample
Vtest  <- as.matrix(test$Exposure)      # time exposure test sample
```

FNN architecture of depth 3 (with one-hot encoding)

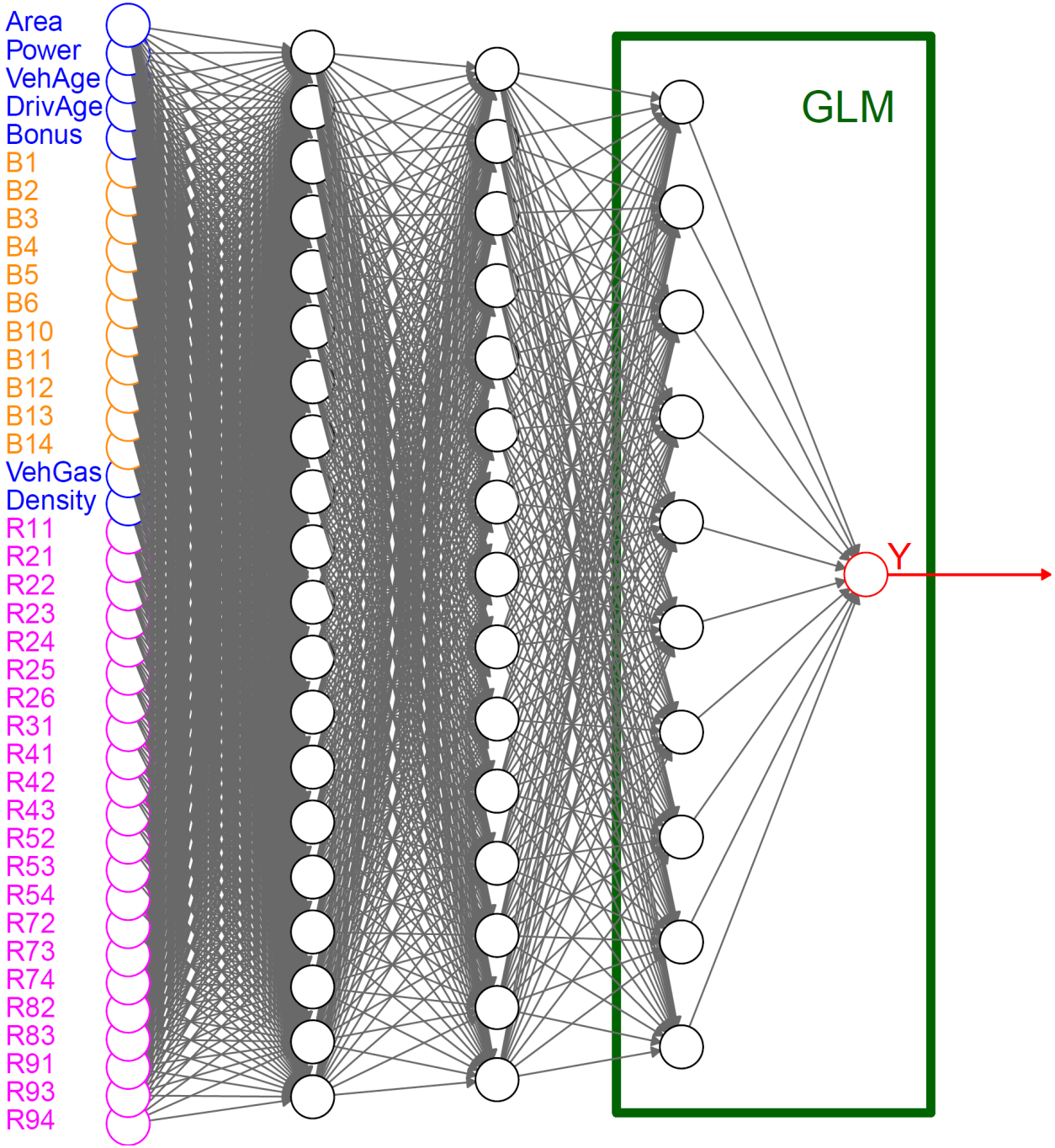
```
FNN <- function(seed, qq){      # tanh activations and log-link output
    tf$keras$backend$clear_session()
    set.seed(seed)
    set_random_seed(seed)
    Design  <- layer_input(shape = c(qq[1]), dtype = 'float32')
    Volume  <- layer_input(shape = c(1), dtype = 'float32')
    Network = Design %>%        # depth d=3 network
          layer_dense(units=qq[2], activation='tanh') %>%
          layer_dense(units=qq[3], activation='tanh') %>%
          layer_dense(units=qq[4], activation='tanh', name="FE") %>%
          layer_dense(units=1, activation='exponential')
    Response = list(Network, Volume) %>% layer_multiply()
    keras_model(inputs = c(Design, Volume), outputs = c(Response))
    }
```

Define FNN architecture

```
# homogeneous mean (empirical frequency on learning sample);
# this will be used to initialize the FNN to the homogeneous model
mu.hom <- sum(learn$ClaimNb)/sum(learn$Exposure)

# define the FNN architecture
q0    <- length(features)
qq    <- c(q0, c(20,15,10))    # selected FNN architecture
seed  <- 100
model <- FNN(seed, qq)
model                          # illustrate the FNN architecture
```

```
Model: "model"
__________________________________________________________________________________
 Layer (type)                Output Shape              Param #   Connected to
==================================================================================
 input_1 (InputLayer)        [(None, 40)]              0         []
 dense_2 (Dense)             (None, 20)                820       ['input_1[0][0]']
 dense_1 (Dense)             (None, 15)                315       ['dense_2[0][0]']
 FE (Dense)                  (None, 10)                160       ['dense_1[0][0]']
 dense (Dense)               (None, 1)                 11        ['FE[0][0]']
 input_2 (InputLayer)        [(None, 1)]               0         []
 multiply (Multiply)         (None, 1)                 0         ['dense[0][0]',
                                                                  'input_2[0][0]']
==================================================================================
Total params: 1306 (5.10 KB)
Trainable params: 1306 (5.10 KB)
Non-trainable params: 0 (0.00 Byte)
__________________________________________________________________________________
```


Area
Power
VehAge
DrivAge
Bonus
B1
B2
B3
B4
B5
B6
B10
B11
B12
B13
B14
VehGas
Density
R11
R21
R22
R23
R24
R25
R26
R31
R41
R42
R43
R52
R53
R54
R72
R73
R74
R82
R83
R91
R93
R94
GLM
Y

Initializing to the homogeneous model (without covariates)

```
## initialize to the homogeneous model (this is an intercept only model)
w0      <- get_weights(model)
w0[[7]] <- array(0, dim=dim(w0[[7]]))             # all signals are zero
w0[[8]] <- array(log(mu.hom), dim=dim(w0[[8]]))   # only bias is non-zero
set_weights(model, w0)
```

- We initialize the output weights so that we obtain the homogeneous model, i.e., the null model not considering any covariates. This is done by offsetting the output weights to receive an intercept-only model.
- All other network weights are randomly initialized using the glorot_uniform initializer of Glorot and Bengio (2010).

```
# prediction in the homogeneous model
learn.hom <- model %>% predict(list(Xlearn, Vlearn), batch_size=10^6)
test.hom  <- model %>% predict(list(Xtest, Vtest), batch_size=10^6)
```

In-sample and out-of-sample Poisson deviance losses

$$\text{on } \mathcal{L}: \quad \frac{1}{\sum_{i=1}^{n} v_i} \sum_{i=1}^{n} 2 v_i \left(\widehat{\mu}(\boldsymbol{X}_i) - Y_i - Y_i \log \left(\frac{\widehat{\mu}(\boldsymbol{X}_i)}{Y_i} \right) \right),$$

$$\text{on } \mathcal{T}: \quad \frac{1}{\sum_{t=1}^{m} v_t} \sum_{t=1}^{m} 2 v_t \left(\widehat{\mu}(\boldsymbol{X}_t) - Y_t - Y_t \log \left(\frac{\widehat{\mu}(\boldsymbol{X}_t)}{Y_t} \right) \right),$$

where the fitted model $\widehat{\mu}$ uses the learning data $\mathcal{L}$ only.

```
Poisson.Deviance <- function(pred, obs, weight){   # scale 10^2
  10^2 * 2*(sum(pred)-sum(obs)+sum(log((obs/pred)^(obs))))/sum(weight) }


## homogeneous case not considering any covariates
loss.hom  <- round(c(Poisson.Deviance(learn.hom, Ylearn, Vlearn),
↪  Poisson.Deviance(test.hom, Ytest, Vtest)), 3)
loss.hom
```

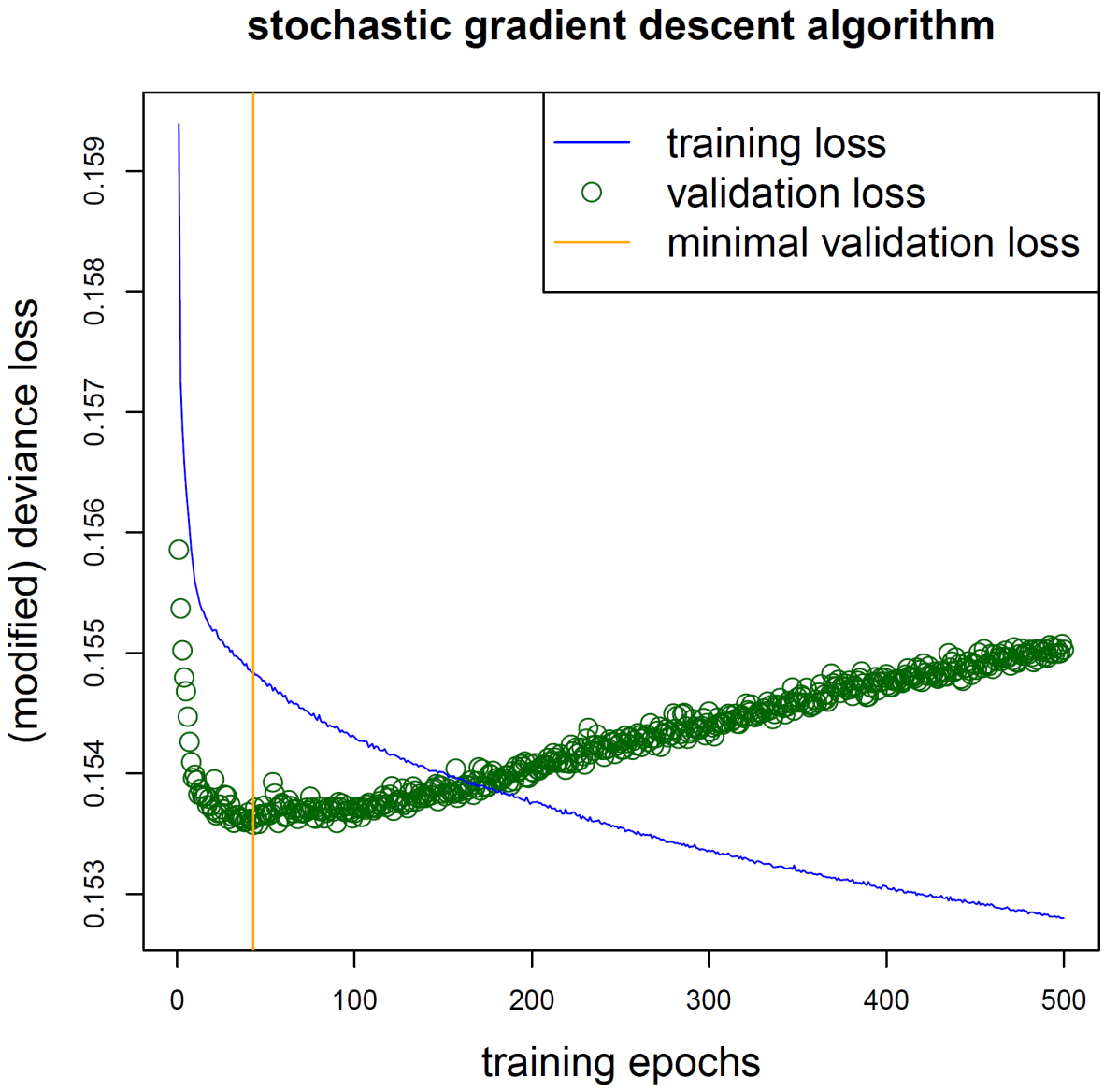
```
[1] 47.722 47.967
```

Stochastic gradient descent fitting (with early stopping)

```
## define the callback for early stopping
if (!dir.exists("./Networks")){dir.create("./Networks")}
path1 <- paste0("./Networks/FNN1_",seed,".h5")
CBs   <- callback_model_checkpoint(path1, monitor = "val_loss", verbose =
↪  0, save_best_only = TRUE, save_weights_only = TRUE)

## recall: w0 is initialized to the homogeneous model
model %>% compile(loss = 'poisson', optimizer = 'nadam')
#
fit <- model %>% fit(list(Xlearn, Vlearn),  Ylearn,
                     validation_split=0.1, batch_size=5000, epochs=500,
                     ↪  verbose=0, callbacks=CBs)
#
which.min(fit[[2]]$val_loss)   # early stopping time
```

```
[1] 43
```


stochastic gradient descent algorithm
training loss
validation loss
minimal validation loss
(modified) deviance loss
0.153
0.154
0.155
0.156
0.157
0.158
0.159
0
100
200
300
400
500
training epochs

- We take 10% of the learning sample $\mathcal{L}$ as validation data $\mathcal{V}$. In Keras, these are simply the last 10% of the instances of the learning sample. Therefore, it is important that the learning sample has a randomized order, because the algorithm does not automatically shuffle the data!
- The levels of the training and validation losses depend on the partition of the learning sample $\mathcal{L}$ into the training sample $\mathcal{U}$ and the validation sample $\mathcal{V}$. Here, $\mathcal{V}$ seems more typical, because it leads to a steeper decrease of the loss during the first 40 gradient descent steps compared to the training data $\mathcal{U}$. This indicates that the systematic effects are more dominant in the validation data, here.
- Careful: The order of the learning sample and all seeds matter, and other choices will provide other graphs and other results (with different stopping times and different minimas).

FNN architecture: Poisson deviance results

```
# load optimal weights (from early stopping)
load_model_weights_hdf5(model, path1)

# compute FNN estimated predictive means
learn.NN <- model %>% predict(list(Xlearn, Vlearn), batch_size=10^6)
test.NN  <- model %>% predict(list(Xtest, Vtest), batch_size=10^6)

# compute in-sample and out-of-sample Poisson deviance losses
loss.FNN <- round(c(Poisson.Deviance(learn.NN, Ylearn, Vlearn),
↪  Poisson.Deviance(test.NN, Ytest, Vtest)), 3)
```

In-sample and out-of-sample Poisson deviance losses of the FNN model:

```
loss.FNN       # => this outperforms the GLM (a summary is given below)
```

```
[1] 44.846 44.925
```

Results

We collect the results.

model	*in-sample loss*	*out-of-sample loss*	*balance (in %)*
Poisson null model	47.722	47.967	7.36
Poisson GLM	45.585	45.435	7.36
Poisson FNN	44.846	44.925	7.17

- The GLM results are taken from Table 5.5 in Wüthrich and Merz (2023); this GLM considers all covariates like our FNN:
  - The deviance loss scaling in Wüthrich and Merz (2023) is different, and the values of 24.084 and 24.102 from this reference need to be scaled by $n/\sum_{i=1}^{n} v_i = 1.89$ and $m/\sum_{t=1}^{m} v_t = 1.89$ to receive our in-sample and out-of-sample Poisson deviance loss scalings.

- We note that the FNN outperforms the GLM, i.e., there are some features in the data that cannot be captured by the proposed GLM.
- From a Poisson simulation analysis we conclude that at least 42.726 of the loss can be allocated to the irreducible risk (pure randomness), and the difference is (probably model error).
- The last column shows the average frequency over the whole portfolio. We observe an under-estimation of the FNN, and we come back to this issue when discussing the balance property.
- We give clear preference to the FNN over the GLM.

Copyright

- © The Authors
- This notebook and these slides are part of the project "AI Tools for Actuaries". The lecture notes can be downloaded from:

https://papers.ssrn.com/sol3/papers.cfm?abstract_id=5162304

References I

Cybenko, G.V. (1989) 'Approximation by superpositions of a sigmoidal function', *Mathematics of Control, Signals and Systems*, 2, pp. 303–314. Available at: https://doi.org/10.1007/BF02551274.

Dutang, C., Charpentier, A. and Gallic, E. (2024) 'Insurance dataset'. Available at: https://dutangc.github.io/CASdatasets/.

Glorot, X. and Bengio, Y. (2010) 'Understanding the difficulty of training deep feedforward neural networks', in Y.W. Teh and M. Titterington (eds) *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. PMLR (Proceedings of machine learning research), pp. 249–256. Available at: https://proceedings.mlr.press/v9/glorot10a.html.

References II

Hinton, G., Srivastava, N. and Swersky, K. (2014) 'Neural networks for machine learning'. Available at: https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

Hornik, K. (1991) 'Approximation capabilities of multilayer feedforward networks', *Neural Networks*, 4(2), pp. 251–257. Available at: https://doi.org/10.1016/0893-6080(91)90009-T.

Hornik, K., Stinchcombe, M. and White, H. (1989) 'Multilayer feedforward networks are universal approximators', *Neural Networks*, 2(5), pp. 359–366. Available at: https://doi.org/10.1016/0893-6080(89)90020-8.

Kingma, D.P. and Ba, J. (2017) 'Adam: A method for stochastic optimization'. Available at: https://arxiv.org/abs/1412.6980.

References III

Leshno, M. *et al.* (1993) 'Multilayer feedforward networks with a nonpolynomial activation function can approximate any function', *Neural Networks*, 6(6), pp. 861–867. Available at: https://doi.org/10.1016/S0893-6080(05)80131-5.

Loshchilov, I. and Hutter, F. (2019) 'Decoupled weight decay regularization'. Available at: https://arxiv.org/abs/1711.05101.

Nesterov, Y. (2007) 'Gradient methods for minimizing composite objective function'. Available at: https://cdn.uclouvain.be/public/Exports%20reddot/core/documents/coredp2007_76.pdf.

References IV

Rumelhart, D., Hinton, G. and Williams, R. (1986) 'Learning representations by back-propagating errors', *Nature*, 323, pp. 533–536. Available at: https://doi.org/10.1038/323533a0.

Srivastava, N. *et al.* (2014) 'Dropout: A simple way to prevent neural networks from overfitting', *Journal of Machine Learning Research*, 15(56), pp. 1929–1958. Available at: http://jmlr.org/papers/v15/srivastava14a.html.

Wager, S., Wang, S. and Liang, P. (2013) 'Dropout training as adaptive regularization'. Available at: https://arxiv.org/abs/1307.1493.

References V

Wüthrich, M.V. *et al.* (2025) 'AI Tools for Actuaries', *SSRN Manuscript* [Preprint]. Available at: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=5162304.

Wüthrich, M.V. and Merz, M. (2023) *Statistical foundations of actuarial learning and its applications*. Springer. Available at: https://doi.org/10.1007/978-3-031-12409-9.