Feed-Forward Neural Networks

Mario V. Wüthrich
RiskLab, ETH Zurich

"Deep Learning with Actuarial Applications in R"
Swiss Association of Actuaries SAA/SAV, Zurich
October 14/15, 2021

Programme SAV Block Course

- Refresher: Generalized Linear Models (THU 9:00-10:30)
- Feed-Forward Neural Networks (THU 13:00-15:00)
- Discrimination-Free Insurance Pricing (THU 17:15-17:45)

- LocalGLMnet (FRI 9:00-10:30)
- Convolutional Neural Networks (FRI 13:00-14:30)
- Wrap Up (FRI 16:00-16:30)

Contents: Feed-Forward Neural Network

- The statistical modeling cycle
- Generic feed-forward neural networks (FNNs)
- Universality theorems
- Gradient descent methods for model fitting
- Generalization loss and cross-validation
- Embedding layers

- **The Statistical Modeling Cycle**

The Statistical Modeling Cycle

(1) data collection, data cleaning and data pre-processing ($> 80\%$ of total time)

(2) selection of model class (data or algorithmic modeling culture, Breiman 2001)

(3) choice of objective function

(4) 'solving' a (non-convex) optimization problem

(5) model validation and variable selection

(6) possibly go back to (1)

▷ 'solving' involves:

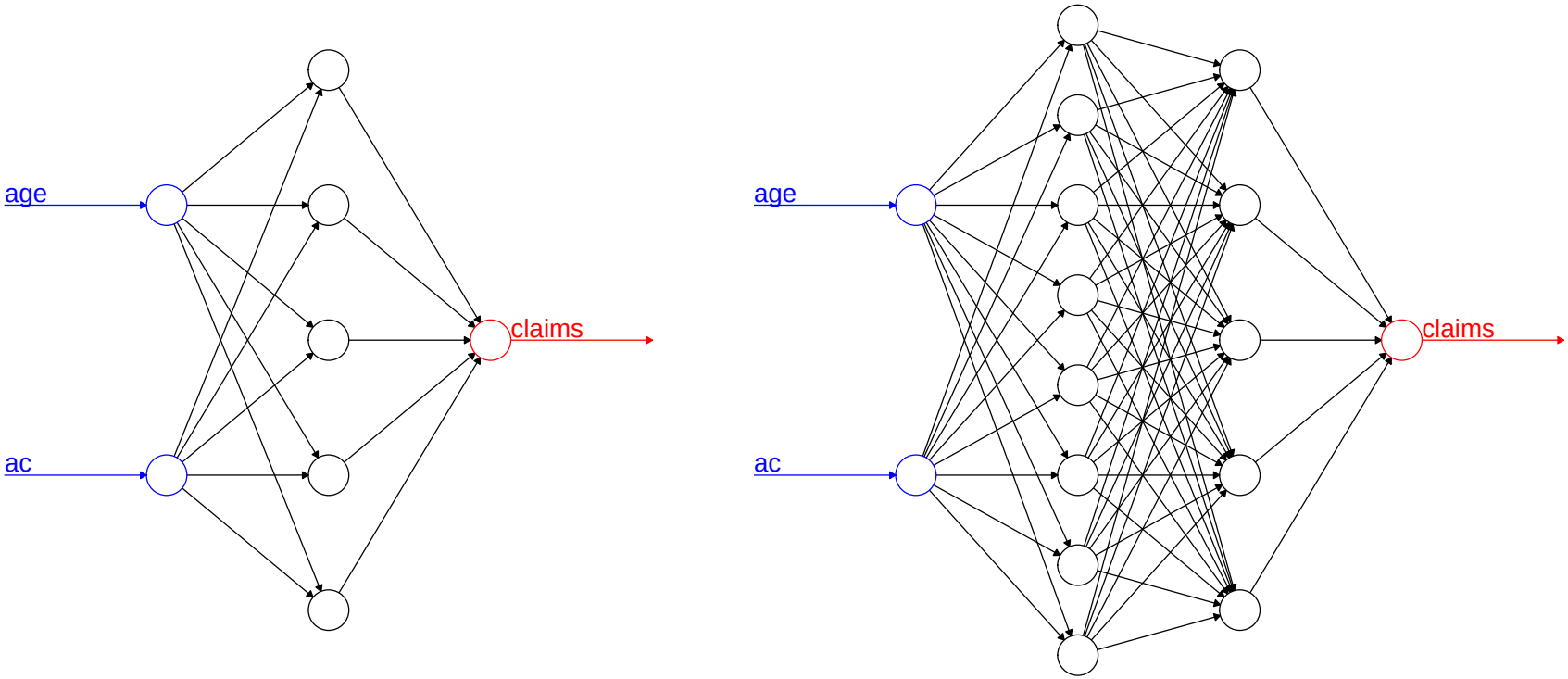
⋆ choice of algorithm
⋆ choice of stopping criterion, step size, etc.
⋆ choice of seed (starting value)

- **Generic Feed-Forward Neural Networks (FNNs)**

Neural Network Architectures

- Neural networks can be understood as an approximation framework.
- Here: neural networks generalize GLMs.
- There are different types of neural networks:
  - ⋆ Feed-forward neural network (FNN): Information propagates in one direction from input to output.
  - ⋆ Recurrent neural network (RNN): This is an extension of FNNs that allows for time series modeling (because it allows for time series (or causal) structures).
  - ⋆ Convolutional neural network (CNN): This is a type of network that allows for modeling temporal and spatial structure, e.g., in image recognition.
- FNNs have stacked hidden layers. If there is exactly one hidden layer, we call the network shallow; if there are multiple hidden layers, we call the network deep.
- There are many special neural network architectures such as generative-adversarial networks (GANs), bottleneck auto-encoder (AE) networks, etc.

Shallow and Deep Fully-Connected FNNs



These two examples are fully-connected FNNs.

Information is processed from the input (in blue) to the output (in red).

Representation Learning

- A GLM with link g has the following structure

$$\boldsymbol{x} \;\mapsto\; \mu(\boldsymbol{x}) = \mathbb{E}\left[Y\right] = g^{-1}\langle \boldsymbol{\beta}, \boldsymbol{x} \rangle.$$

  ▷ This requires manual feature engineering to bring $\boldsymbol{x}$ into the right form.

- Networks perform automated feature engineering.

- A layer is given by a mapping

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

  ▷ Each layer presents a new representation of the covariates.

- In general, compose layers

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

Fully-Connected FNN Layer

- Choose dimensions $q_{m-1}, q_m \in \mathbb{N}$ and activation function $\phi : \mathbb{R} \to \mathbb{R}$.

- A (hidden) FNN layer is a mapping

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

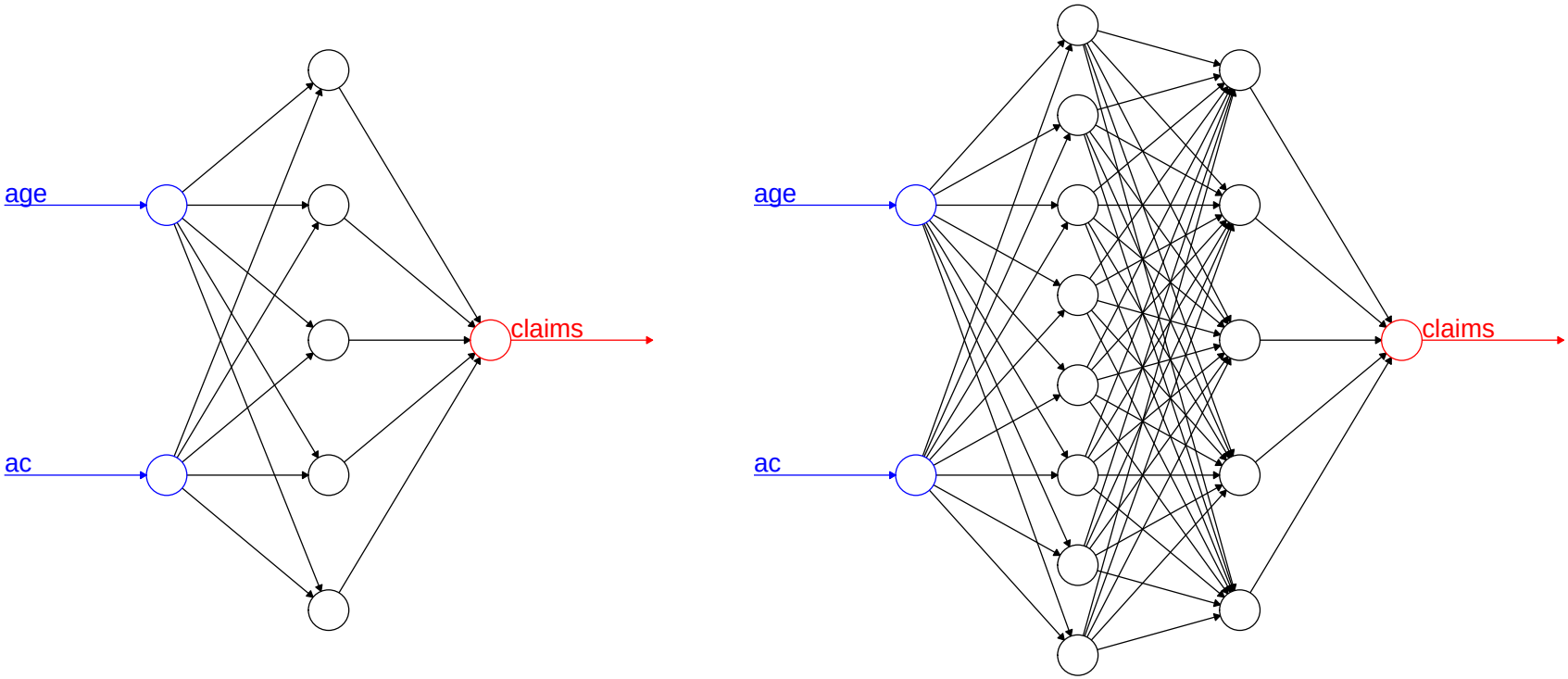
  with (hidden) neurons given by, $1 \leq j \leq q_m$,

$$z_j^{(m)}(\boldsymbol{x}) = \phi \left(w_{j,0}^{(m)} + \sum_{l=1}^{q_{m-1}} w_{j,l}^{(m)} x_l \right) \overset{\text{def.}}{=} \phi\langle \boldsymbol{w}_j^{(m)}, \boldsymbol{x} \rangle,$$

  for given network weights (parameters) $\boldsymbol{w}_j^{(m)} \in \mathbb{R}^{q_{m-1}+1}$.

- Every neuron $z_j^{(m)}(\boldsymbol{x})$ describes a GLM w.r.t. feature $\boldsymbol{x} \in \mathbb{R}^{q_{m-1}}$ and activation ϕ. The resulting function (called ridge function) reflects a compression of information.

Shallow and Deep Fully-Connected FNNs



These two examples are fully-connected FNNs.

Information is processed from the input (in blue) to the output (in red).

Activation Function

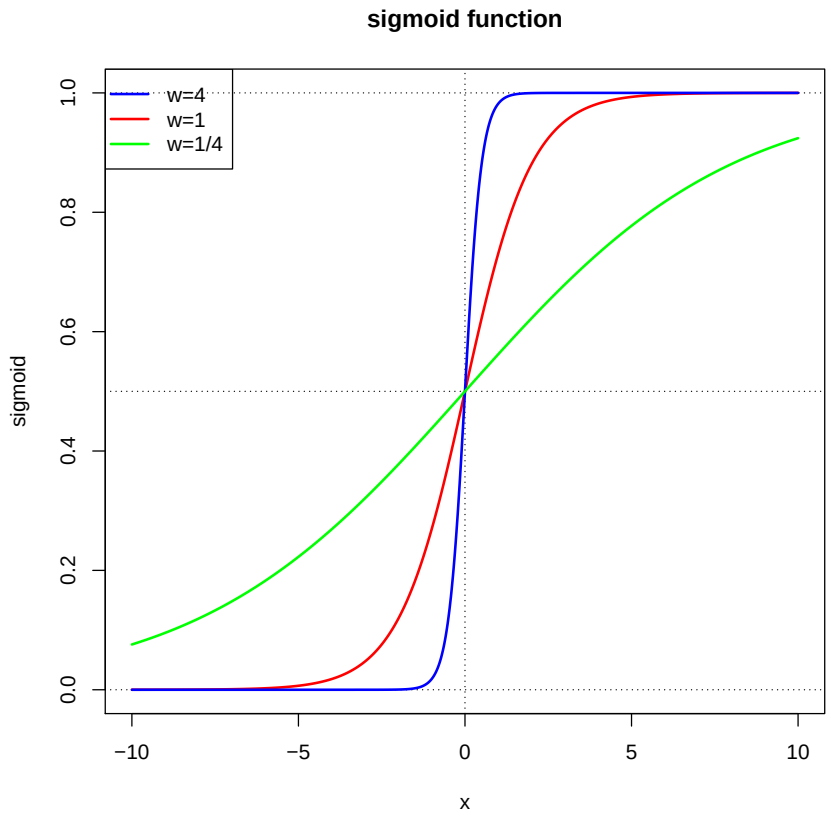
- The activation function $\phi : \mathbb{R} \to \mathbb{R}$ is an inverse link function $\phi = g^{-1}$.
- Since we would like to approximate non-linear regression functions, activation functions should be non-linear, too.
- The most popular choices of activation functions are

sigmoid/logistic function	$\phi(x) = (1 + e^{-x})^{-1} \in (0, 1)$	$\phi' = \phi(1 - \phi)$
hyperbolic tangent function	$\phi(x) = \tanh(x) \in (-1, 1)$	$\phi' = 1 - \phi^2$
exponential function	$\phi(x) = \exp(x) \in (0, \infty)$	$\phi' = \phi$
step function	$\phi(x) = \mathbb{1}_{\{x \geq 0\}} \in \{0, 1\}$	not differentiable in 0
rectified linear unit (ReLU)	$\phi(x) = x\mathbb{1}_{\{x \geq 0\}} \in [0, \infty)$	not differentiable in 0

- We mainly use hyperbolic tangent (with the following relationship to sigmoid)

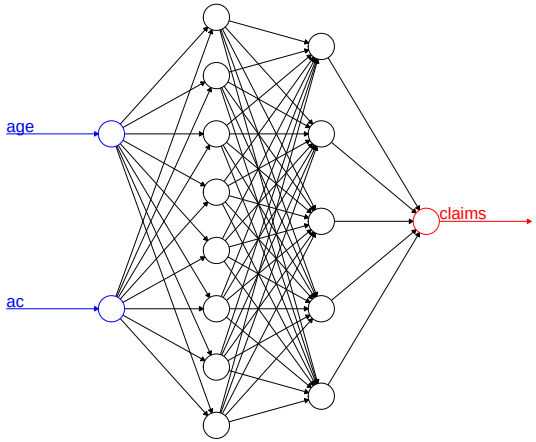
$$x \mapsto \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\left(1 + e^{-2x}\right)^{-1} - 1 = 2\,\mathrm{sigmoid}(2x) - 1.$$

Sigmoid Activation Function $\phi(x) = (1 + e^{-x})^{-1}$



- Sigmoid activation $x \mapsto \phi(wx)$ for weights $w \in \{1/4, 1, 4\}$ and $x \in (-10, 10)$:
  - ⋆ "deactivated" for small values x, i.e. $\phi(wx) \approx 0$ for x small,
  - ⋆ "activated" for big values x, i.e. $\phi(wx) \approx 1$ for x large.

Fully-Connected FNN Architecture



- Choose depth of the network $d \in \mathbb{N}$ and define the FNN layer composition

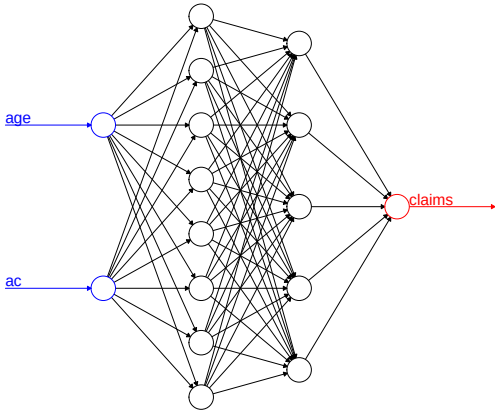
$$\boldsymbol{x} \;\mapsto\; \boldsymbol{z}^{(d:1)}(\boldsymbol{x}) \;\stackrel{\text{def.}}{=}\; \left(\boldsymbol{z}^{(d)} \circ \cdots \circ \boldsymbol{z}^{(1)}\right)(\boldsymbol{x}) \in \mathbb{R}^{q_d},$$

  with $q_0 = q$ for $\boldsymbol{x} \in \mathbb{R}^q$.

- Define output layer with link function g by

$$\boldsymbol{x}_i \;\mapsto\; \mu_i = \mathbb{E}[Y_i] = g^{-1}\left\langle \boldsymbol{\beta}, \boldsymbol{z}^{(d:1)}(\boldsymbol{x}_i) \right\rangle.$$

FNN Architecture: Interpretations



- Network mapping

$$\boldsymbol{x}_i \;\mapsto\; \mu_i = \mathbb{E}[Y_i] = g^{-1}\left\langle \boldsymbol{\beta}, \boldsymbol{z}^{(d:1)}(\boldsymbol{x}_i) \right\rangle .$$

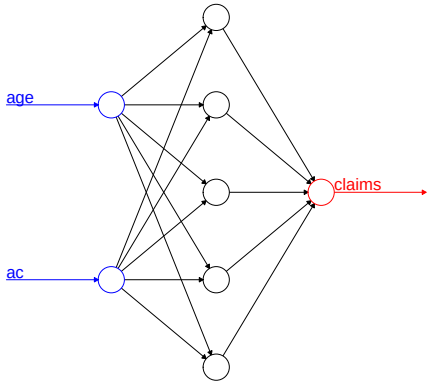
- Mapping $\boldsymbol{x}_i \mapsto \boldsymbol{z}_i = \boldsymbol{z}^{(d:1)}(\boldsymbol{x}_i)$ should be understood as feature engineering or representation learning.

- The linear activation function $\phi(x) = x$ provides a GLM (composition of linear functions is a linear function). Thus, a GLM is a special case of a FNN.

- For depth $d = 0$ we receive a GLM, too.

- **Universality Theorems**

Universality Theorems for FNNs



- Cybenko (1989) and Hornik et al. (1989): Any compactly supported continuous function can be approximated arbitrarily well (in sup- or L^2-norm) by shallow FNNs with sigmoid activation if allowing for arbitrarily many hidden neurons (q_1).

- Leshno et al. (1993): The universality theorem for shallow FNNs holds if and only if the activation function ϕ is non-polynomial.

- Grohs et al. (2019): Shallow FNNs with ReLU activation functions provide polynomial approximation rates, deep FNNs provide exponential rates.

Simple Example Supporting Deep FNNs

- Consider a 2-dimensional example $\mu : [0,1]^2 \to \mathbb{R}_+$

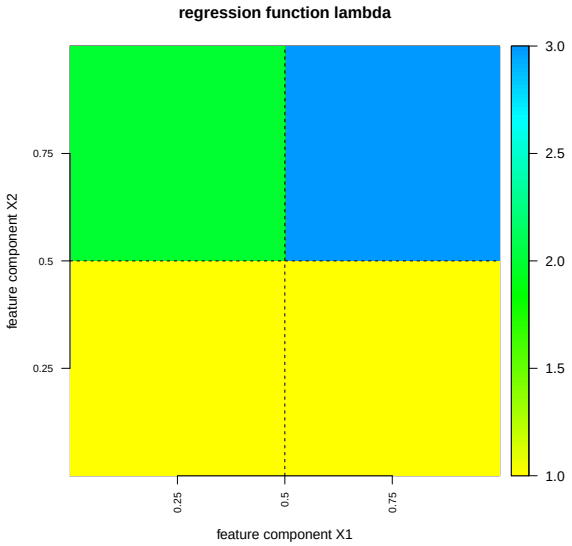
$$\boldsymbol{x} \;\mapsto\; \mu(\boldsymbol{x}) = 1 + \mathbb{1}_{\{x_2 \geq 1/2\}} + \mathbb{1}_{\{x_1 \geq 1/2,\; x_2 \geq 1/2\}} \;\in\; \{1,2,3\}.$$

- Choose step function activation $\phi(x) = \mathbb{1}_{\{x \geq 0\}}$.

- A FNN of depth $d = 2$ with $q_1 = q_2 = 2$

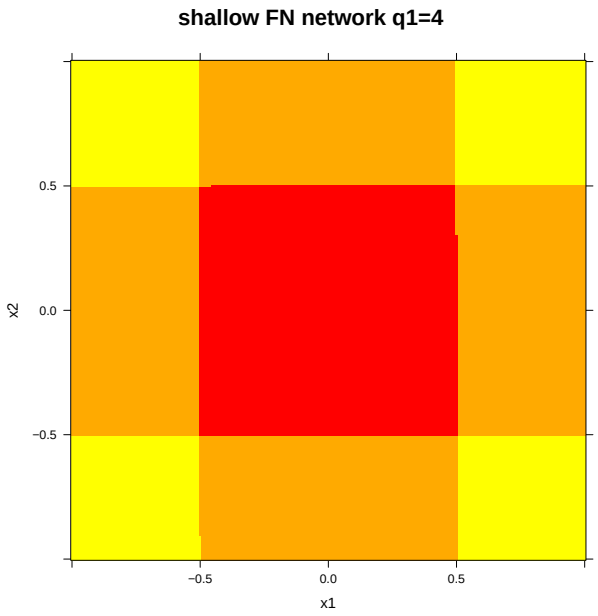
$$\left\langle \boldsymbol{\beta}, \boldsymbol{z}^{(2:1)}(\boldsymbol{x}) \right\rangle = \left\langle \boldsymbol{\beta}, (\boldsymbol{z}^{(2)} \circ \boldsymbol{z}^{(1)})(\boldsymbol{x}) \right\rangle,$$

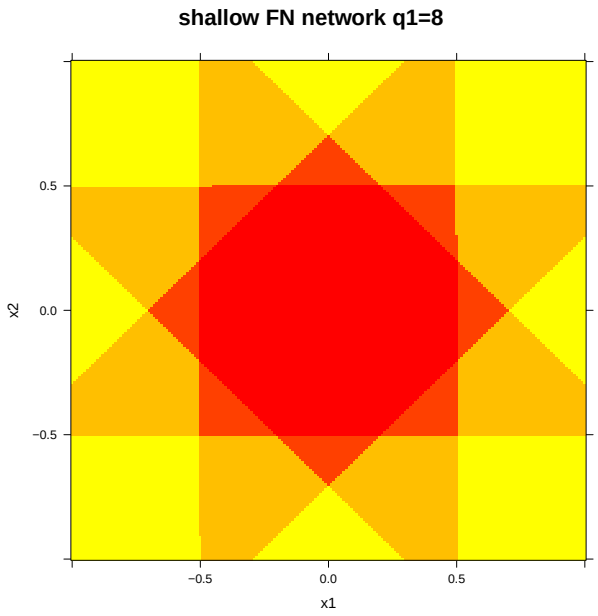
  can perfectly approximate function μ.

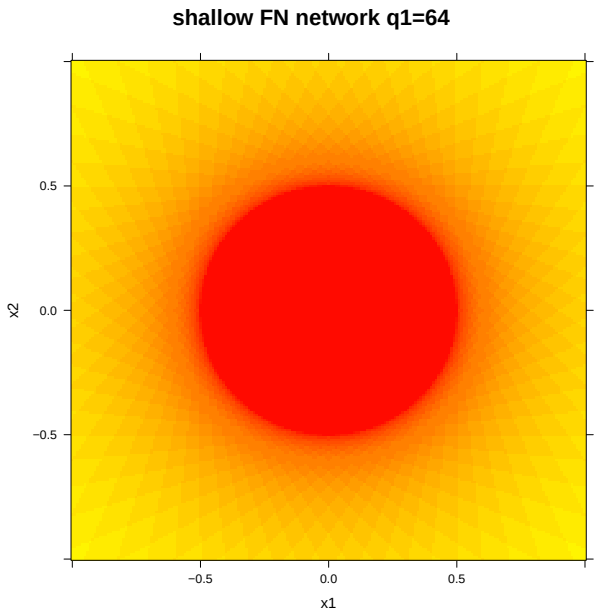


- Deep FNNs allow for more complex interactions of covariates through compositions of layers/functions: wide allows for superposition, and deep allows for composition.

Shallow Neural Networks







- **Gradient Descent Methods for Model Fitting**

Deviance Loss Function

- FNN mapping

$$\boldsymbol{x}_i \;\mapsto\; \mu_i = \mathbb{E}[Y_i] = g^{-1}\left\langle \boldsymbol{\beta}, \boldsymbol{z}^{(d:1)}(\boldsymbol{x}_i)\right\rangle,$$

has network parameter

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

of dimension $r = \sum_{m=1}^{d} q_m(q_{m-1}+1) + (q_d+1)$.

- The deviance loss function under independent observations $(Y_i)_{i=1}^n$

$$\begin{aligned}\vartheta \;\mapsto\; D^*(\boldsymbol{Y},\vartheta) &= 2\left[\ell_{\boldsymbol{Y}}(\boldsymbol{Y}) - \ell_{\boldsymbol{Y}}(\vartheta)\right] \\ &= 2\sum_{i=1}^{n}\frac{v_i}{\varphi}\Big[Y_i h(Y_i) - \kappa(h(Y_i)) - Y_i h(\mu_i) + \kappa(h(\mu_i))\Big] \;\geq\; 0.\end{aligned}$$

- Minimizing deviance loss $D^*(\boldsymbol{Y},\vartheta)$ in network parameter ϑ provides MLE $\widehat{\vartheta}$.

Plain Vanilla Gradient Descent Method (1/2)

- Gradient descent methods (GDMs) stepwise iteratively improve network parameter ϑ by moving into the direction of the maximal (local) decrease of $D^*(\boldsymbol{Y}, \vartheta)$.

- 1st order Taylor expansion of deviance loss in network parameter ϑ

$$D^*(\boldsymbol{Y}, \widetilde{\vartheta}) = D^*(\boldsymbol{Y}, \vartheta) + \nabla_\vartheta D^*(\boldsymbol{Y}, \vartheta)^\top \left(\widetilde{\vartheta} - \vartheta\right) + o\left(\|\widetilde{\vartheta} - \vartheta\|\right),$$

  for $\|\widetilde{\vartheta} - \vartheta\| \to 0$ (we suppose differentiability).

- Calculate the corresponding gradient

$$\nabla_\vartheta D^*(\boldsymbol{Y}, \vartheta) \;=\; \sum_{i=1}^{n} 2\left[\mu_i - Y_i\right] \nabla_\vartheta h(\mu_i).$$

- Back-propagation (Rumelhart et al. 1986) is an efficient way to calculate $\nabla_\vartheta h(\mu_i)$.

Plain Vanilla Gradient Descent Method (2/2)

- Negative gradient $-\nabla_\vartheta D^*(\boldsymbol{Y}, \vartheta)$ gives the direction for ϑ of the maximal local decrease in deviance loss.

- For a given learning rate $\varrho_{t+1} > 0$, the gradient descent algorithm updates network parameter $\vartheta^{(t)}$ iteratively by (adapted locally optimal)

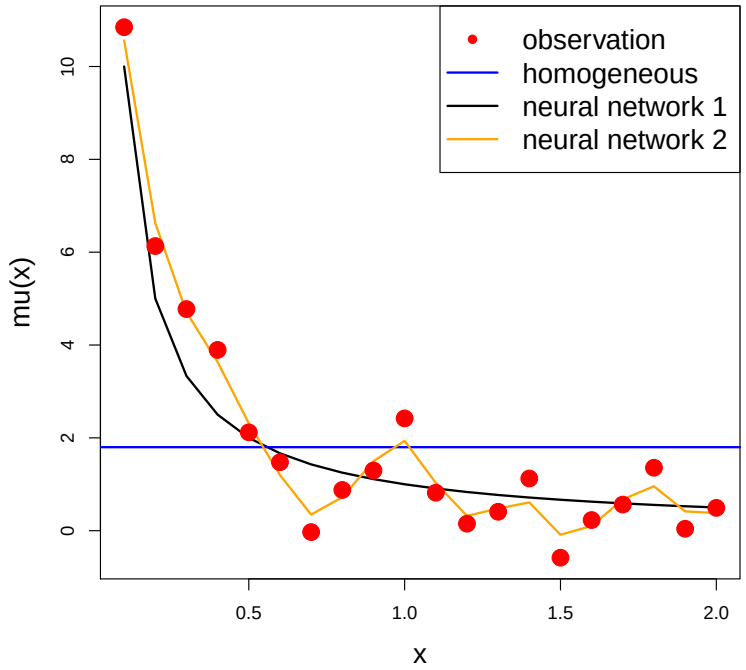
$$\vartheta^{(t)} \;\mapsto\; \vartheta^{(t+1)} = \vartheta^{(t)} - \varrho_{t+1} \nabla_\vartheta D^*(\boldsymbol{Y}, \vartheta^{(t)}).$$

- This update provides new (in-sample) deviance loss for $\varrho_{t+1} \to 0$

$$D^*(\boldsymbol{Y}, \vartheta^{(t+1)}) = D^*(\boldsymbol{Y}, \vartheta^{(t)}) - \varrho_{t+1} \left\| \nabla_\vartheta D^*(\boldsymbol{Y}, \vartheta^{(t)}) \right\|^2 + o\left(\varrho_{t+1}\right).$$

- Using a tempered learning rate $(\varrho_t)_{t \geq 1}$ the network parameter $\vartheta^{(t)}$ converges to a local minimum of $D^*(\boldsymbol{Y}, \cdot)$ for $t \to \infty$.

Over-Fitting in Complex FNNs

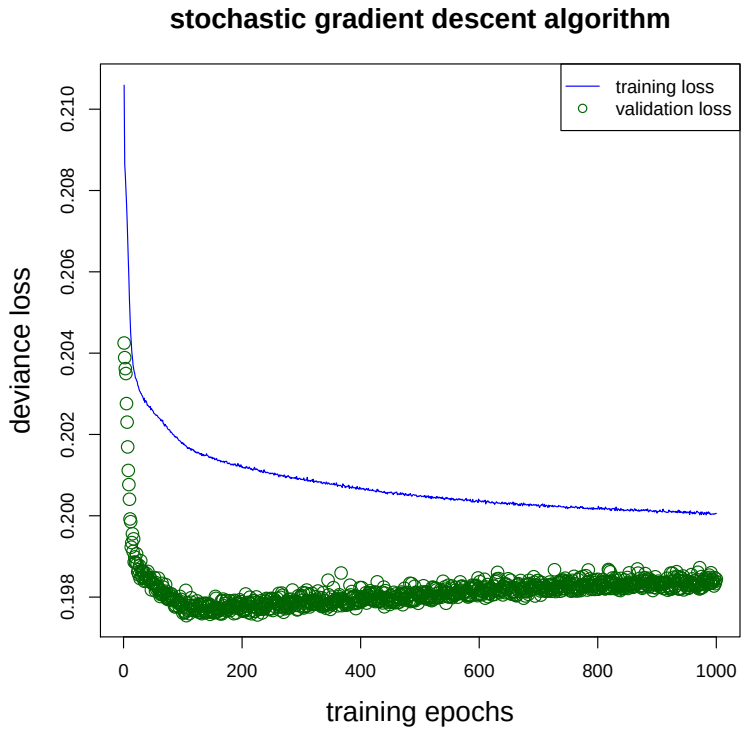


- Convergence to a local minimum of $D^*(\boldsymbol{Y}, \cdot)$ typically means over-fitting.

- Apply early stopping:

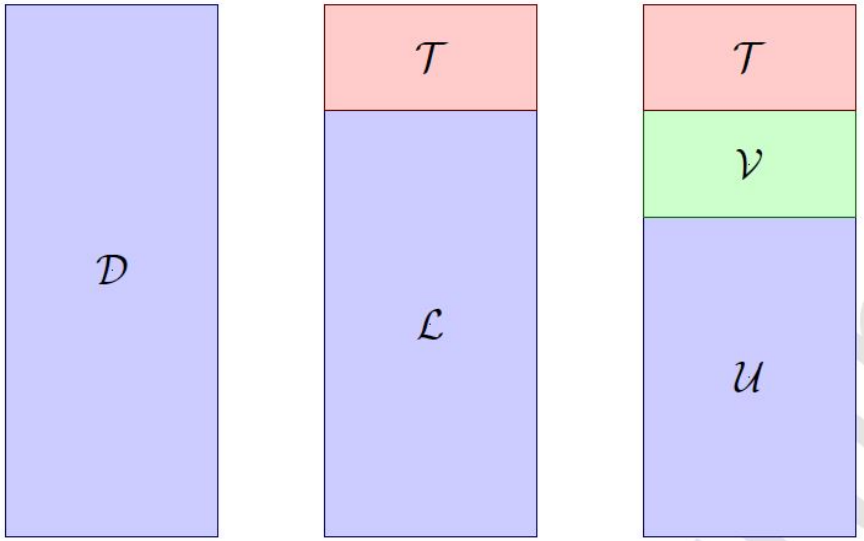
  ⋆ Partition data at random into training data $\mathcal{U}$ and validation data $\mathcal{V}$.
  ⋆ Fit ϑ on $\mathcal{U}$ (in-sample) and track over-fitting on $\mathcal{V}$ (out-of-sample).
  ⋆ The "best" model obviously is non-unique when we use early stopping.

Early Stopping of Gradient Descent Algorithm



- Convergence to a local minimum of $D^*(\boldsymbol{Y}, \cdot)$ typically means over-fitting.
- Apply early stopping:
  - ⋆ Partition data at random into training data $\mathcal{U}$ and validation data $\mathcal{V}$.
  - ⋆ Fit ϑ on $\mathcal{U}$ (in-sample) and track over-fitting on $\mathcal{V}$ (out-of-sample).
  - ⋆ The "best" model obviously is non-unique when we use early stopping.

Use of Data



$\mathcal{D}$	entire data,
$\mathcal{L}$	learning data (in-sample),
$\mathcal{T}$	test data (out-of-sample),
$\mathcal{U}$	training data,
$\mathcal{V}$	validation data

Computational Issues and Stochastic Gradient

- Gradient descent steps

$$\vartheta^{(t)} \;\mapsto\; \vartheta^{(t+1)} = \vartheta^{(t)} - \varrho_{t+1} \nabla_\vartheta D^*(\boldsymbol{Y}, \vartheta^{(t)}),$$

  involve high-dimensional matrix multiplications

$$\nabla_\vartheta D^*(\boldsymbol{Y}, \vartheta) \;=\; \sum_{i=1}^{n} 2\,[\mu_i - Y_i]\,\nabla_\vartheta h(\mu_i),$$

  which are computationally expensive if the size of the training data $\mathcal{U}$ is large.

- Partition training data $\mathcal{U}$ at random in mini batches $\mathcal{U}_k$ of a given size. Use for gradient descent steps one mini batch $\mathcal{U}_k$ at a time. This is called stochastic gradient descent (SGD) algorithm.

- Running through all mini batches $(\mathcal{U}_k)_k$ once is called a training epoch.

- Using the entire training data in each GDM step is called steepest gradient descent.

Size of Mini-Batches

- Partition training data $\mathcal{U}$ at random in mini batches $\mathcal{U}_1, \ldots, \mathcal{U}_K$, and use for each gradient descent step one mini batch $\mathcal{U}_k$ at a time

$$\nabla_\vartheta D^*(\mathcal{U}_k, \vartheta) \;=\; \sum_{i \in \mathcal{U}_k} 2\,[\mu_i - Y_i]\,\nabla_\vartheta h(\mu_i).$$

- Size of mini-batches for Poisson frequencies?

$$\left[\mu(\boldsymbol{x}) - 2\sqrt{\frac{\mu(\boldsymbol{x})}{v}}, \mu(\boldsymbol{x}) + 2\sqrt{\frac{\mu(\boldsymbol{x})}{v}}\right] = \left[5\% - 2\sqrt{\frac{5\%}{2000}}, 5\% + 2\sqrt{\frac{5\%}{2000}}\right] = [4\%, 6\%].$$

Note for Poisson case $\mathbb{E}[N] = \mathrm{Var}(N) = \mu(\boldsymbol{x})v$.

Momentum-Based Gradient Descent Methods

- Plain vanilla GDMs use 1st order Taylor expansions.
- To improve convergence rates we could use 2nd order Taylor expansions.
- 2nd order Taylor expansions involve calculations of Hessians.
- This is computationally not feasible.
- Replace Hessians by momentum methods (inspired by physics/mechanics).
- Choose a momentum coefficient $\nu \in [0, 1)$ and set initial speed $\mathbf{v}^{(0)} = 0 \in \mathbb{R}^r$. Replace plain vanilla GDM update by

$$\begin{aligned} \mathbf{v}^{(t)} &\mapsto \mathbf{v}^{(t+1)} = \nu \mathbf{v}^{(t)} - \varrho_{t+1} \nabla_\vartheta D^*(\boldsymbol{Y}, \vartheta^{(t)}), \\ \vartheta^{(t)} &\mapsto \vartheta^{(t+1)} = \vartheta^{(t)} + \mathbf{v}^{(t+1)}. \end{aligned}$$

Predefined Gradient Descent Methods

- 'rmsprop' chooses learning rates that differ in all directions by consider directional sizes ('rmsprop' stands for root mean square propagation);

- 'adam' stands for adaptive moment estimation, similar to 'rmsprop' it searches for directionally optimal learning rates based on the momentum induced by past gradients measured by an L^2-norm;

- 'nadam' is Nesterov (2007) accelerated version of 'adam' avoiding zig-zag behavior.

- For more details we refer to Chapter 8 of Goodfellow et al. (2016) and Section 7.2.3 in Wüthrich–Merz (2021)

- **Generalization Loss and Cross-Validation**

Empirical Generalization Loss

Typically, for neural network modeling one considers 3 disjoint sets of data.

- Training data $\mathcal{U}$: is used to fit the network parameter ϑ.
- Validation data $\mathcal{V}$: is used to track in-sample over-fitting (early stopping).
- Test data $\mathcal{T}$: is used to study out-of-sample generalization loss.

Assume that $\widehat{\vartheta}^{\mathcal{U},\mathcal{V}}$ is the estimated network parameter based on $\mathcal{U}$ and $\mathcal{V}$. The test data $\mathcal{T}$ is given by $(Y_t, \boldsymbol{x}_t, v_t)_{t=1}^T$. We have (out-of-sample) generalization loss (GL)

$$D^*(\boldsymbol{Y}, \widehat{\vartheta}^{\mathcal{U},\mathcal{V}}) = 2\sum_{t=1}^{T} \frac{v_t}{\varphi}\Big[Y_t h(Y_t) - \kappa(h(Y_t)) - Y_t h(\widehat{\mu}_t^{\mathcal{U},\mathcal{V}}) + \kappa(h(\widehat{\mu}_t^{\mathcal{U},\mathcal{V}}))\Big].$$

- This is an empirical generalization loss based on $\mathcal{T}$ mimicking portfolio distribution.

K-Fold Cross-Validation Loss

- If one cannot afford to partition the data $\mathcal{D}$ into 3 disjoint sets training data $\mathcal{U}$, validation data $\mathcal{V}$ and test data $\mathcal{T}$, one has to use the data more efficiently.
- K-fold cross-validation aims at doing so.
- Partition entire data at random in K subsets $\mathcal{D}_1, \ldots, \mathcal{D}_K$ of roughly equal size.
- Denote by $\widehat{\vartheta}^{(-\mathcal{D}_k)}$ the estimated network parameter based on all data except $\mathcal{D}_k$.
- The K-fold cross-validation loss is given by

$$D^{\mathrm{CV}} = \frac{1}{K} \sum_{k=1}^{K} \left(2 \sum_{t \in \mathcal{D}_k} \frac{v_t}{\varphi} \left[Y_t h(Y_t) - \kappa(h(Y_t)) - Y_t h(\widehat{\mu}_t^{(-\mathcal{D}_k)}) + \kappa(h(\widehat{\mu}_t^{(-\mathcal{D}_k)})) \right] \right).$$

- This mimics K times an out-of-sample generalization loss on $\mathcal{D}_k$, respectively.
- In neural network modeling K-fold cross-validation is computationally too costly.

- **Car Insurance Frequency Example**

Car Insurance Claims Frequency Data

```
'data.frame':   678013 obs. of  12 variables:
 $ IDpol      : num  1 3 5 10 11 13 15 17 18 21 ...
 $ ClaimNb    : num  1 1 1 1 1 1 1 1 1 1 ...
 $ 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   : int  5 5 6 7 7 6 6 7 7 7 ...
 $ VehAge     : int  0 0 2 0 0 2 2 0 0 0 ...
 $ DrivAge    : int  55 55 52 46 46 38 38 33 33 41 ...
 $ BonusMalus: int  50 50 50 50 50 50 50 68 68 50 ...
 $ VehBrand   : Factor w/ 11 levels "B1","B10","B11",..: 4 4 4 4 4 4 4 4 4 4 ...
 $ VehGas     : Factor w/ 2 levels "Diesel","Regular": 2 2 1 1 1 2 2 1 1 1 ...
 $ Density    : int  1217 1217 54 76 76 3003 3003 137 137 60 ...
 $ Region     : Factor w/ 22 levels "R11","R21","R22",..: 18 18 3 15 15 8 8 20 20 12
```

- 3 categorical covariates, 1 binary covariate and 5 continuous covariates
- Goal: Find systematic effects to explain/predict claim counts.

Feature Engineering

- Categorical features: use either dummy coding or one-hot encoding.
  PS: We come back to this choice below.

- Also continuous features need pro-processing. All feature components should live on a similar scale such that the GDM can be applied efficiently.

▷ Often, the MinMaxScaler is used

$$x_{i,l} \;\mapsto\; x_{i,l}^{\mathrm{MM}} = 2\,\frac{x_{i,l} - x_l^-}{x_l^+ - x_l^-} - 1 \;\in\; [-1,1],$$

where x_l^- and x_l^+ are the minimum and maximum of the domain of $x_{i,l}$.

- Successful application of MinMaxScaler pre-processing requires that the feature distribution is not "too skewed", otherwise pre-processing should be performed with a scaler that accounts for skewness (like the log function).

- Standardization with empirical mean and standard deviation is possible, too.

Deep FNN Coding in R keras

```
library(keras)

q0 <- 12   # dimension of input x
q1 <- 20
q2 <- 15
q3 <- 10

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

Network = Design %>%
          layer_dense(units=q1, activation='tanh', name='hidden1') %>%
          layer_dense(units=q2, activation='tanh', name='hidden2') %>%
          layer_dense(units=q3, activation='tanh', name='hidden3') %>%
          layer_dense(units=1,  activation='exponential', name='Network')

model <- keras_model(inputs = c(Design), outputs = c(Network))
model %>% compile(optimizer = optimizer_nadam(), loss = 'poisson')

summary(model)
```

Deep FNN with $(q_1, q_2, q_3) = (20, 15, 10)$

```
Layer (type)                 Output Shape              Param #
=================================================================
Design (InputLayer)          (None, 12)                0
_________________________________________________________________
hidden1 (Dense)              (None, 20)                260
_________________________________________________________________
hidden2 (Dense)              (None, 15)                315
_________________________________________________________________
hidden3 (Dense)              (None, 10)                160
_________________________________________________________________
Network (Dense)              (None, 1)                 11
=================================================================
Total params: 746
Trainable params: 746
Non-trainable params: 0
```

Poisson FNN Regression with Offset

- Poisson regression with offset and canonical link $g = h = \log$, set $N_i = v_i Y_i$

$$v_i \mu_i \;=\; \mathbb{E}\left[N_i\right] \;=\; v_i \kappa'(\theta_i) \;=\; v_i \exp(\theta_i) \;=\; \exp(\log v_i + \theta_i).$$

- The Poisson FNN regression is given by, set $\mu_i = \mu(\boldsymbol{x}_i)$,

$$\boldsymbol{x}_i \;\mapsto\; \log\left(\mathbb{E}\left[N_i\right]\right) = \log\left(v_i \mu(\boldsymbol{x}_i)\right) = \log v_i + \left\langle \boldsymbol{\beta}, \boldsymbol{z}^{(d:1)}(\boldsymbol{x}_i) \right\rangle.$$

- The Poisson deviance loss function is given by

$$D^*(\boldsymbol{N}, \vartheta) = \sum_{i=1}^{n} 2\; N_i \left[\frac{v_i \mu(\boldsymbol{x}_i)}{N_i} - 1 - \log\left(\frac{v_i \mu(\boldsymbol{x}_i)}{N_i} \right) \right] \;\geq\; 0,$$

  where the i-th term is set equal to $2 v_i \mu(\boldsymbol{x}_i)$ for $N_i = 0$.

- In `keras` the terms independent of ϑ are dropped in the deviance losses.

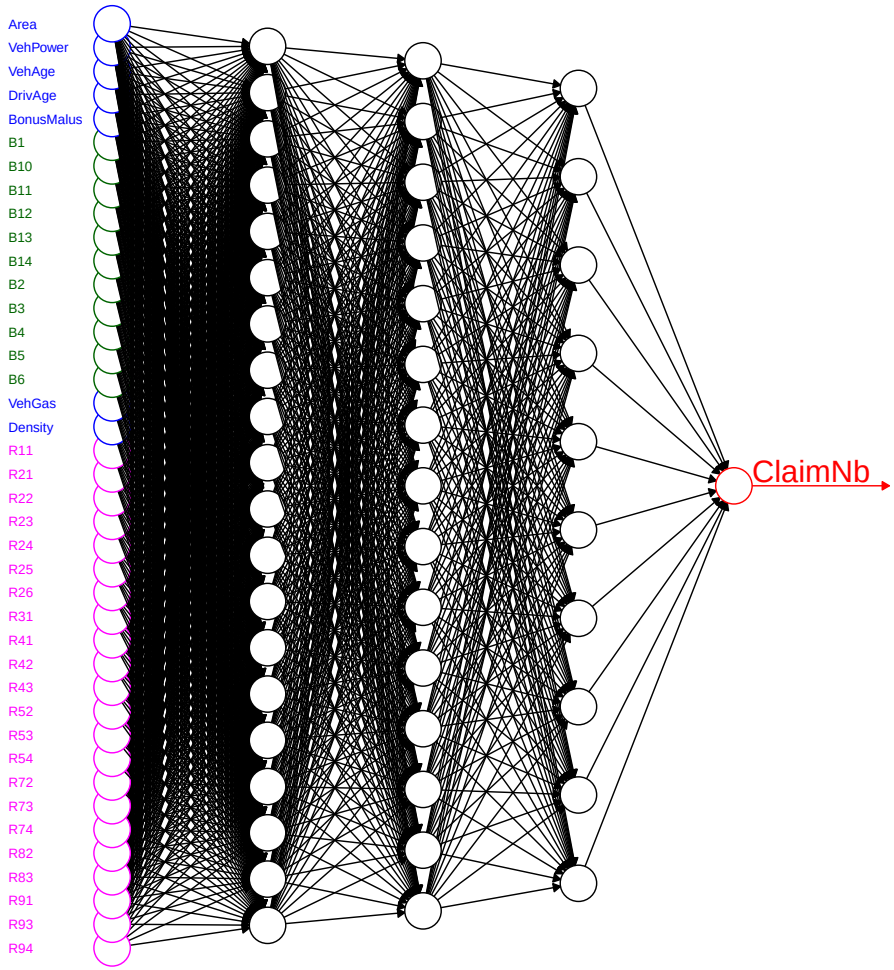
Deep FNN Coding in R keras with Offset

```
q0 <- 12     # dimension of input x
q1 <- 20
q2 <- 15
q3 <- 10
lambda.hom <- 0.05   # initialization of network output (homogeneous model)

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='hidden1') %>%
          layer_dense(units=q2, activation='tanh',   name='hidden2') %>%
          layer_dense(units=q3, activation='tanh',   name='hidden3') %>%
          layer_dense(units=1,  activation='linear', name='Network',
             weights=list(array(0, dim=c(10,1)), array(log(lambda.hom), dim=c(1))))

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

model <- keras_model(inputs = c(Design, LogVol), outputs = c(Response))
model %>% compile(optimizer = optimizer_nadam(), loss = 'poisson')
```

Deep FNN with $(q_1, q_2, q_3) = (20, 15, 10)$ with Offset

```
Layer (type)                 Output Shape              Param #   Connected to
==================================================================================
Design (InputLayer)          (None, 12)                0
----------------------------------------------------------------------------------
hidden1 (Dense)              (None, 20)                260       Design[0][0]
----------------------------------------------------------------------------------
hidden2 (Dense)              (None, 15)                315       hidden1[0][0]
----------------------------------------------------------------------------------
hidden3 (Dense)              (None, 10)                160       hidden2[0][0]
----------------------------------------------------------------------------------
Network (Dense)              (None, 1)                 11        hidden3[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: 748
Trainable params: 746
Non-trainable params: 2
```

Application to French MTPL Data



Input dimension is $q_0 = 40$ (one-hot encoding), this provides $r = 1'306$.

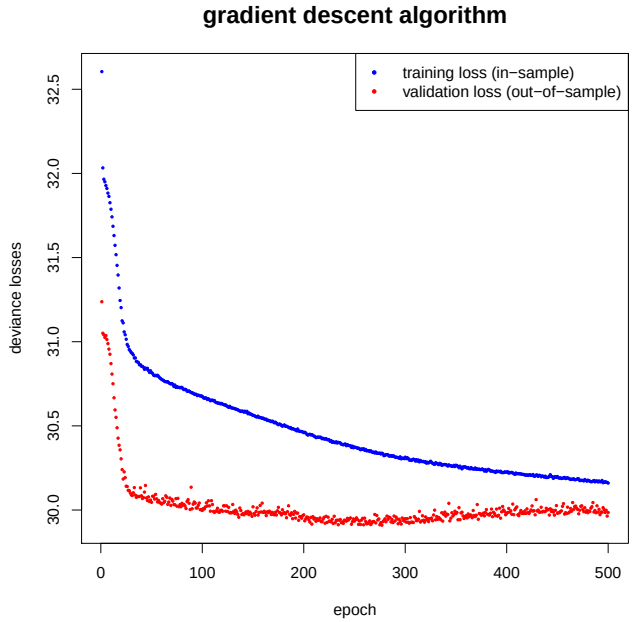
Results of Deep FNN Model

	epochs	run time	# param.	in-sample loss 10^{-2}	out-of-sample loss 10^{-2}	average frequency
homogeneous model		–	1	32.935	33.861	10.02%
Model GLM1		20s	49	31.267	32.171	10.02%
Deep FNN model	250	152s	1'306	30.268	31.673	10.19%

- Network fitting needs quite some run time.
- We perform early stopping.
- The best validation loss model can be retrieved with a callback, see next slide.
- We see a *substantial* improvement in out-of-sample loss on test data $\mathcal{T}$.
- Balance property fails to hold.
- Remark: AIC is not a sensible model selection criterion for FNNs (early stopping).

Callbacks in Gradient Descent Methods

```
path0 <- "./name0"
CB    <- callback_model_checkpoint(path0, monitor = "val_loss",
                        verbose = 0,  save_best_only = TRUE,
                        save_weights_only = TRUE)

model %>% fit(list(X.learn, LogVol.learn), Y.learn,
                       validation_split = 0.1, batch_size = 10000, epochs = 500,
                       verbose = 0, callbacks = CBs)

load_model_weights_hdf5(model, path0)
```



- **Embedding Layers for Categorical Variables**

Categorical Variables and Dummy/One-Hot Encoding

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

each row is in $\mathbb{R}^{10}$

B1 $\mapsto$ $\boldsymbol{e}_1$	1	0	0	0	0	0	0	0	0	0	0
B10 $\mapsto$ $\boldsymbol{e}_2$	0	1	0	0	0	0	0	0	0	0	0
B11 $\mapsto$ $\boldsymbol{e}_3$	0	0	1	0	0	0	0	0	0	0	0
B12 $\mapsto$ $\boldsymbol{e}_4$	0	0	0	1	0	0	0	0	0	0	0
B13 $\mapsto$ $\boldsymbol{e}_5$	0	0	0	0	1	0	0	0	0	0	0
B14 $\mapsto$ $\boldsymbol{e}_6$	0	0	0	0	0	1	0	0	0	0	0
B2 $\mapsto$ $\boldsymbol{e}_7$	0	0	0	0	0	0	1	0	0	0	0
B3 $\mapsto$ $\boldsymbol{e}_8$	0	0	0	0	0	0	0	1	0	0	0
B4 $\mapsto$ $\boldsymbol{e}_9$	0	0	0	0	0	0	0	0	1	0	0
B5 $\mapsto$ $\boldsymbol{e}_{10}$	0	0	0	0	0	0	0	0	0	1	0
B6 $\mapsto$ $\boldsymbol{e}_{11}$	0	0	0	0	0	0	0	0	0	0	1

each row is in $\mathbb{R}^{11}$

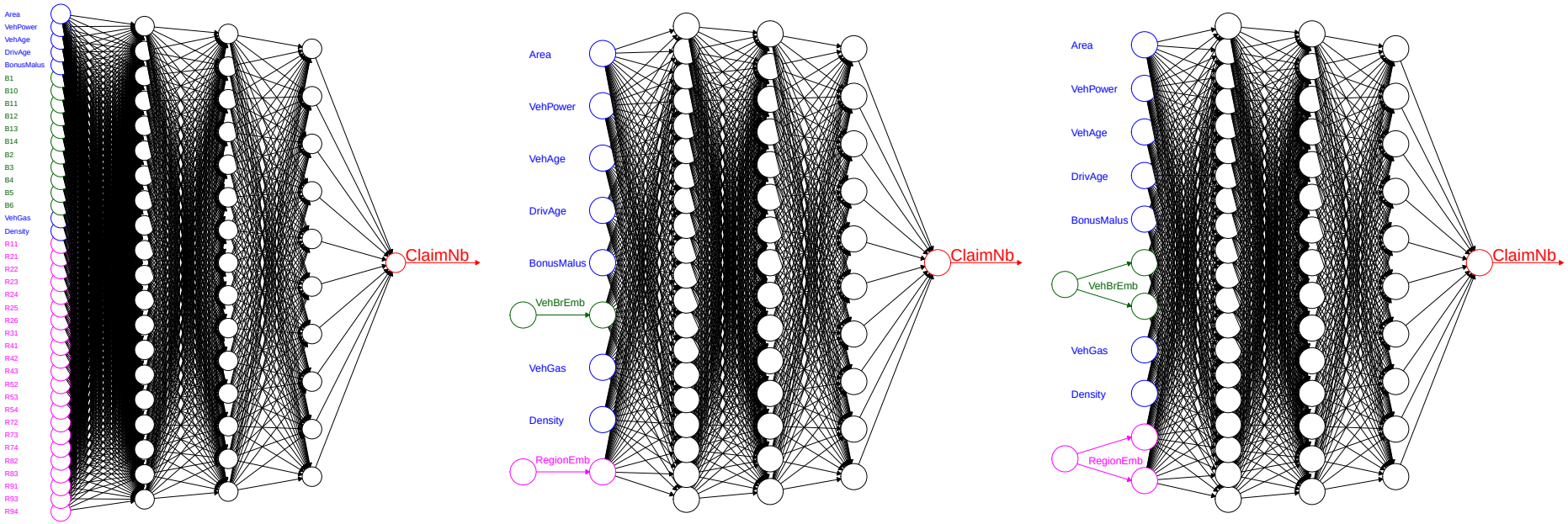
Embeddings for Categorical Variables

- One-hot encoding uses as many dimensions as there are labels (mapping to unit vectors in Euclidean space).
- All labels have the same distance from each other.
- From Natural Language Processing (NLP) we have learned that there are "better" codings in the sense that we should try to map to low-dimensional Euclidean spaces $\mathbb{R}^b$, and similar labels (w.r.t. the regression task) should have some proximity.
- Choose $b \in \mathbb{N}$ and consider an embedding mapping (representation)

$$\boldsymbol{e} : \{\texttt{B1}, \ldots, \texttt{B6}\} \to \mathbb{R}^b, \qquad \texttt{brand} \mapsto \boldsymbol{e}(\texttt{brand}) \stackrel{\text{def.}}{=} \boldsymbol{e}^{\texttt{brand}}.$$

- $\boldsymbol{e}^{\texttt{brand}} \in \mathbb{R}^b$ are called embeddings, and optimal embeddings for the regression task can be learned during GDM training. This amounts in adding an additional (embedding) layer to the FNN.

Deep FNN using Embedding Layers (1/2)



(left) one-hot enc. (middle) $b = 1$-dim. emb's (right) $b = 2$-dim. emb's

- Embedding weights are learned during network training (gradient descent).

Deep FNN using Embedding Layers (2/2)

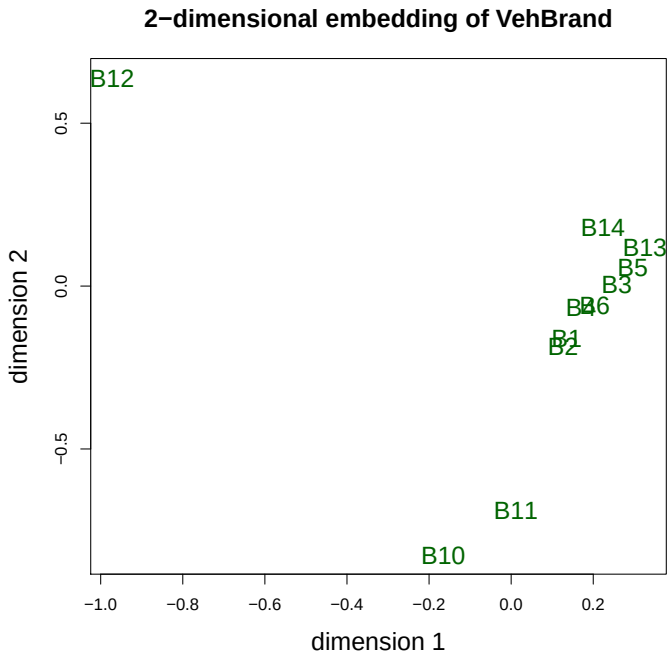
```
Design   <- layer_input(shape = c(7), dtype = 'float32', name = 'Design')
VehBrand <- layer_input(shape = c(1), dtype = 'int32',   name = 'VehBrand')
Region   <- layer_input(shape = c(1), dtype = 'int32',   name = 'Region')
LogVol   <- layer_input(shape = c(1), dtype = 'float32', name = 'LogVol')

BrEmb = VehBrand %>%
    layer_embedding(input_dim=11, output_dim=2, input_length=1, name='BrEmb') %>%
    layer_flatten(name='Br_flat')
ReEmb = Region %>%
    layer_embedding(input_dim=22, output_dim=2, input_length=1, name='ReEmb') %>%
    layer_flatten(name='Re_flat')

Network = list(Design, BrEmb, ReEmb) %>% layer_concatenate(name='concate') %>%
    layer_dense(units=20, activation='tanh',   name='hidden1') %>%
    layer_dense(units=15, activation='tanh',   name='hidden2') %>%
    layer_dense(units=10, activation='tanh',   name='hidden3') %>%
    layer_dense(units=1,  activation='linear', name='Network',
                weights=list(array(0, dim=c(10,1)), array(log(lambda.hom), dim=c(1))))

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

model <- keras_model(inputs=c(Design,VehBrand,Region,LogVol), outputs=c(Response))
```

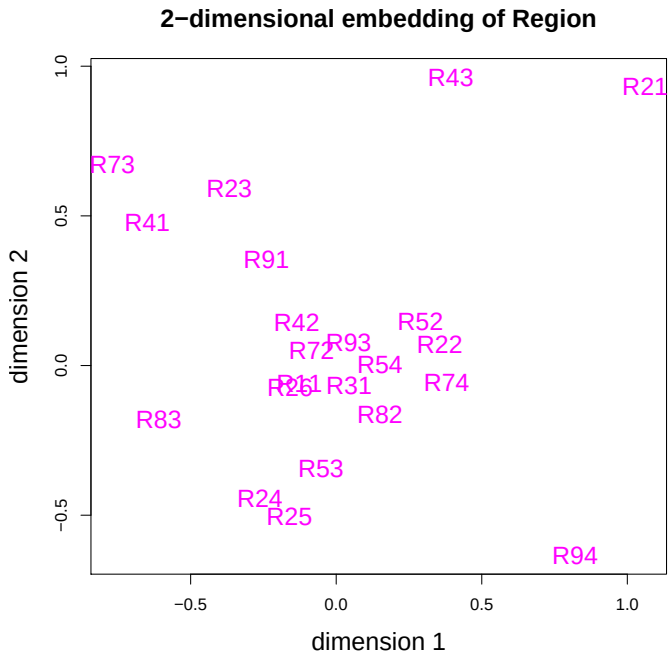
Results of Deep FNN Model with Embeddings

	epochs	run time	# param.	in-sample loss 10^{-2}	out-of-sample loss 10^{-2}	average frequency
homogeneous model		–	1	32.935	33.861	10.02%
Model GLM1		20s	49	31.268	32.171	10.02%
Deep FNN One-Hot	250	152s	1'306	30.268	31.673	10.19%
Deep FNN Emb($b = 1$)	700	419s	719	30.245	31.506	9.90%
Deep FNN Emb($b = 2$)	600	365s	792	30.165	31.453	9.70%

- Network fitting needs quite some run time.
- We perform early stopping using a callback.
- We see a substantial improvement in out-of-sample loss on test data $\mathcal{T}$.
- Balance property fails to hold.
- Remark: AIC is not a sensible model selection criterion for FNNs (early stopping).

Learned Two-Dimensional Embeddings





Two-dimensional embeddings can be nicely plotted and interpreted.

Special Purpose Layers and Other Features

- **Drop-out layers.** A method to prevent from over-training individual neurons to a certain task is to introduce so-called drop-out layers. A drop-out layer, say, after `'hidden2'` of the above listing would remove during a gradient descent step at random any of the 15 neurons in that layer with a given drop-out probability $p \in (0, 1)$, and independently from the other neurons. This random removal will imply that the composite of the remaining neurons needs to sufficiently well cover the dropped-out neurons. Therefore, a single neuron cannot be over-trained to a certain task because it may need to play several different roles at the same time. Drop-out can be interpreted in terms of ridge regression, see Section 18.6 in Efron–Hastie (2016).

- **Normalization layers.** Feature activations $\boldsymbol{z}^{(m:1)}(\boldsymbol{x})$ are scaled back to be centered and have unit standard variance (similar to MinMaxScaler).

- **Skip connections.** Certain layers are skipped in the network architecture, this is going to be used in the LocalGLMnet chapter.

References

- Breiman (2001). Statistical modeling: the two cultures. Statistical Science 16/3, 199-215.
- Cybenko (1989). Approximation by superpositions of a sigmoidal function. Mathematics of Control, Signals, and Systems 2, 303-314.
- Efron (2020). Prediction, estimation, and attribution. Journal American Statistical Association 115/539 , 636-655.
- Efron, Hastie (2016). Computer Age Statistical Inference: Algorithms, Evidence, and Data Science. Cambridge UP.
- Ferrario, Noll, Wüthrich (2018). Insights from inside neural networks. SSRN 3226852.
- Goodfellow, Bengio, Courville (2016). Deep Learning. MIT Press.
- Grohs, Perekrestenko, Elbrächter, Bölcskei (2019). Deep neural network approximation theory. IEEE Transactions on Information Theory.
- Hastie, Tibshirani, Friedman (2009). The Elements of Statistical Learning. Springer.
- Hornik, Stinchcombe, White (1989). Multilayer feedforward networks are universal approximators. Neural Networks 2, 359-366.
- Kingma, Ba (2014). Adam: A method for stochastic optimization. arXiv:1412.6980.
- Leshno, Lin, Pinkus, Schocken (1993). Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. Neural Networks 6/6, 861-867.
- Nesterov (2007). Gradient methods for minimizing composite objective function. Technical Report 76, Center for Operations Research and Econometrics (CORE), Catholic University of Louvain.
- Noll, Salzmann, Wüthrich (2018). Case study: French motor third-party liability claims. SSRN 3164764.
- Richman (2020a/b). AI in actuarial science – a review of recent advances – part 1/2. Annals of Actuarial Science.
- Rumelhart, Hinton, Williams (1986). Learning representations by back-propagating errors. Nature 323/6088, 533-536.
- Schelldorfer, Wüthrich (2019). Nesting classical actuarial models into neural networks. SSRN 3320525.
- Shmueli (2010). To explain or to predict? Statistical Science 25/3, 289-310.
- Wüthrich, Buser (2016). Data Analytics for Non-Life Insurance Pricing. SSRN 2870308, Version September 10, 2020.
- Wüthrich, Merz (2021). Statistical Foundations of Actuarial Learning and its Applications. SSRN 3822407.