

# Demystifying neural networks and their use in actuarial science

---

CNP Assurances & Chaire DIALog, petit déjeuner

Katrien Antonio

December 16, 2021

Based on (ongoing) work with PhD Roel Henckaerts, MSc Simon Gielis and MSc Freek Holvoet

# Why this topic?

## Learning outcomes

- **de-mystify** neural networks in light of increasing literature on the use of neural nets in actuarial science
- develop foundations of working with (different types of) **neural networks**
- focus on the use of neural networks for the **analysis of claim frequency + severity data**, also in combination with GLMs or tree-based ML models
- present some **ongoing research** on this topic.

# Want to read more?

This presentation is based on

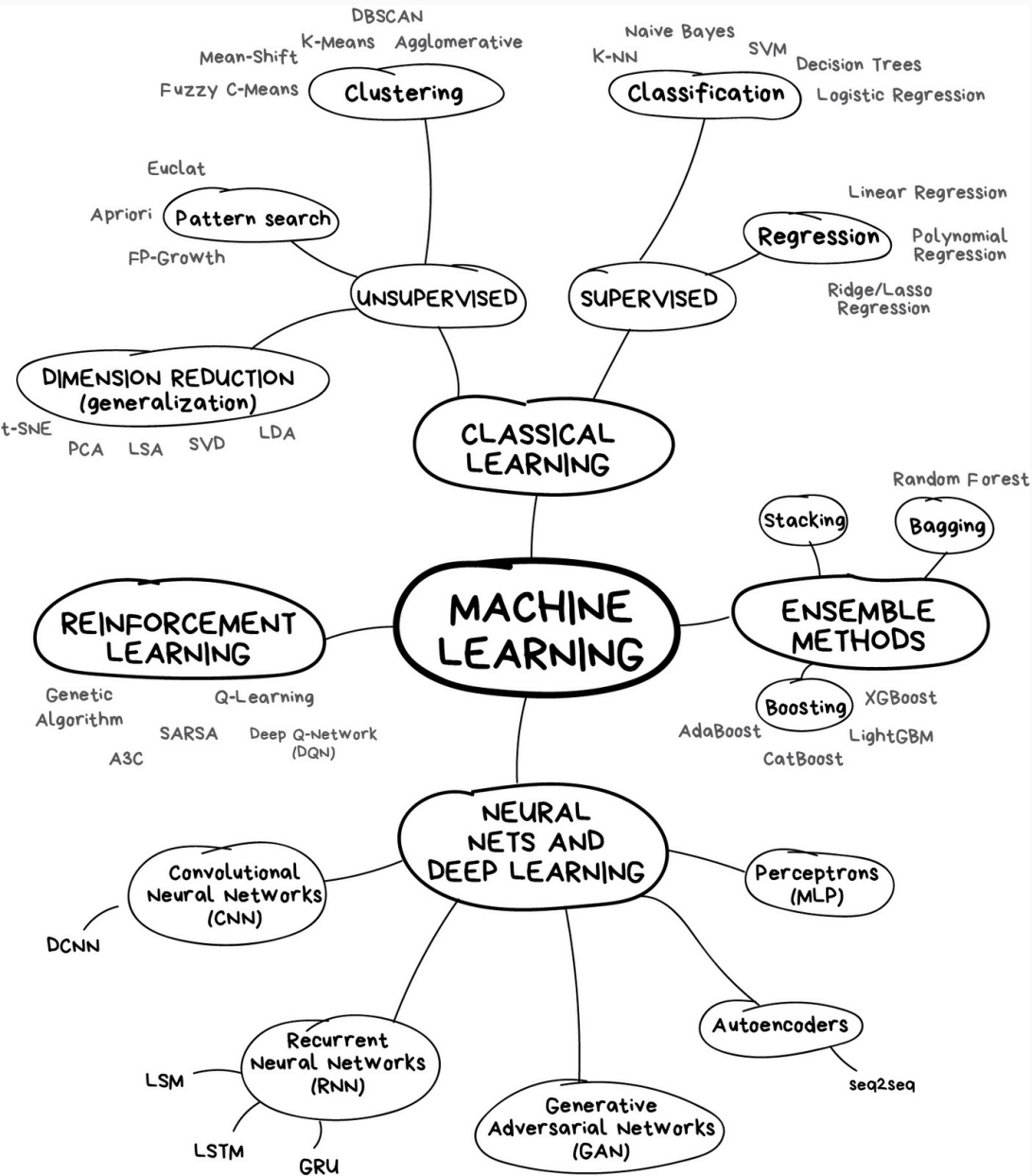
- Michael A. Nielsen (2015) [Neural networks and deep learning](#)
- the work of prof. Taylor Arnold, in particular Chapter 8 in the book [A computational approach to statistical learning](#) by Arnold, Kane & Lewis (2019)
- Boehmke (2020) on [Deep Learning with R: using Keras with TensorFlow backend](#).

Actuarial modelling with neural nets is covered in (among others)

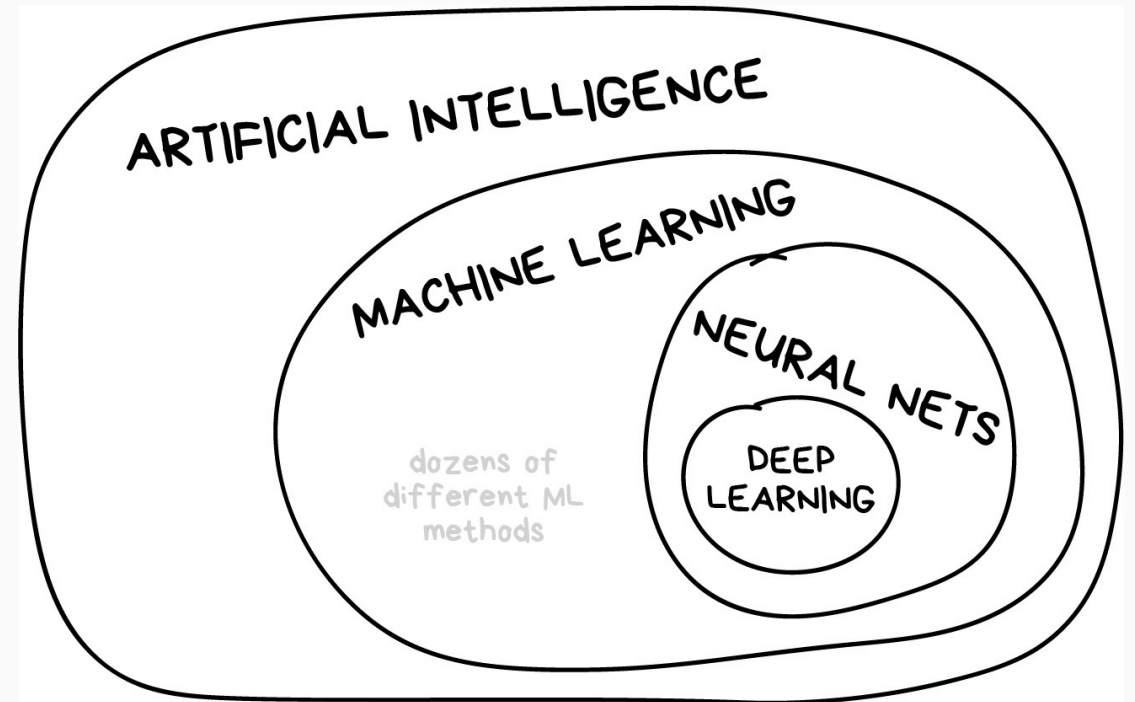
- Wüthrich & Buser (2020) [Data analytics for non-life insurance pricing](#), in particular Chapter 5
- Wüthrich (2019) [From Generalized Linear Models to neural networks, and back](#)
- Wüthrich & Merz (2019) [Editorial: Yes, we CANN!](#), in ASTIN Bulletin 49/1
- Denuit, Hainaut & Trufin (2019) [Effective Statistical Learning Methods for Actuaries: Neural Networks and Extensions](#), Springer Actuarial Lecture Notes
- A series of (working) papers covering the use of neural nets in insurance pricing (classic, and with telematics collected data), mortality forecasting, reserving, ...

# Outline

- Getting started
  - Unpacking our toolbox
  - Tensors
- De-mystifying neural networks
  - What's in a name?
  - A simple neural network
- Neural network architecture
  - An architecture with layers in {keras}
- Network compilation
  - Loss function and forward pass
  - Gradient descent and backpropagation
- Regression with neural networks
  - Redefining GLMs as a neural network
  - Including exposure
  - Skip connection and Combined Actuarial Neural Networks (CANNs)
  - Ongoing research
- Conclusions



Some roadmaps to explore the ML landscape...



Source: [Machine Learning for Everyone In simple words. With real-world examples. Yes, again.](#)

# Getting started

---

# What's the excitement about?

💡 Neural networks are an exciting topic to explore, because:

- They are a **biologically-inspired programming paradigm** that enables a computer to learn from data.
- **Deep learning** is a powerful set of techniques for learning in neural networks.
- Neural networks and deep learning provide **best-in-class solutions** when **going beyond tabular data**, e.g. many problems in image recognition, speech recognition and natural language processing.
- The **universal approximation theorem** (Hornik et al., 1989; Cybenko, 1989) states that neural networks with a single hidden layer can be used to approximate any continuous function to any desired precision.

# An accessible programming framework



- R:  
With interface to Keras and TensorFlow.
- Keras:  
An intuitive high level Python interface to TensorFlow.
- TensorFlow:  
Open source platform for machine learning developed by the Google Brain Team, see <https://www.tensorflow.org/>.  
Special focus on training deep neural networks.
- Tensors generalize vectors and matrices to an arbitrary number of dimensions, cfr. image, video or sound data.
- TensorFlow is a flexible framework which consists of highly optimized functions based on tensors.



# Example of a 4D tensor

Let's picture an image data set where

- each image has a specific height and width
- three color channels (Red, Green, Blue) are registered
- multiple images (`samples`) are stored.

Then, a collection of images can be stored in a 4D tensor (`samples, height, width, channels`).



Source: [Bradley Boehmke](#)

# De-mystifying neural networks

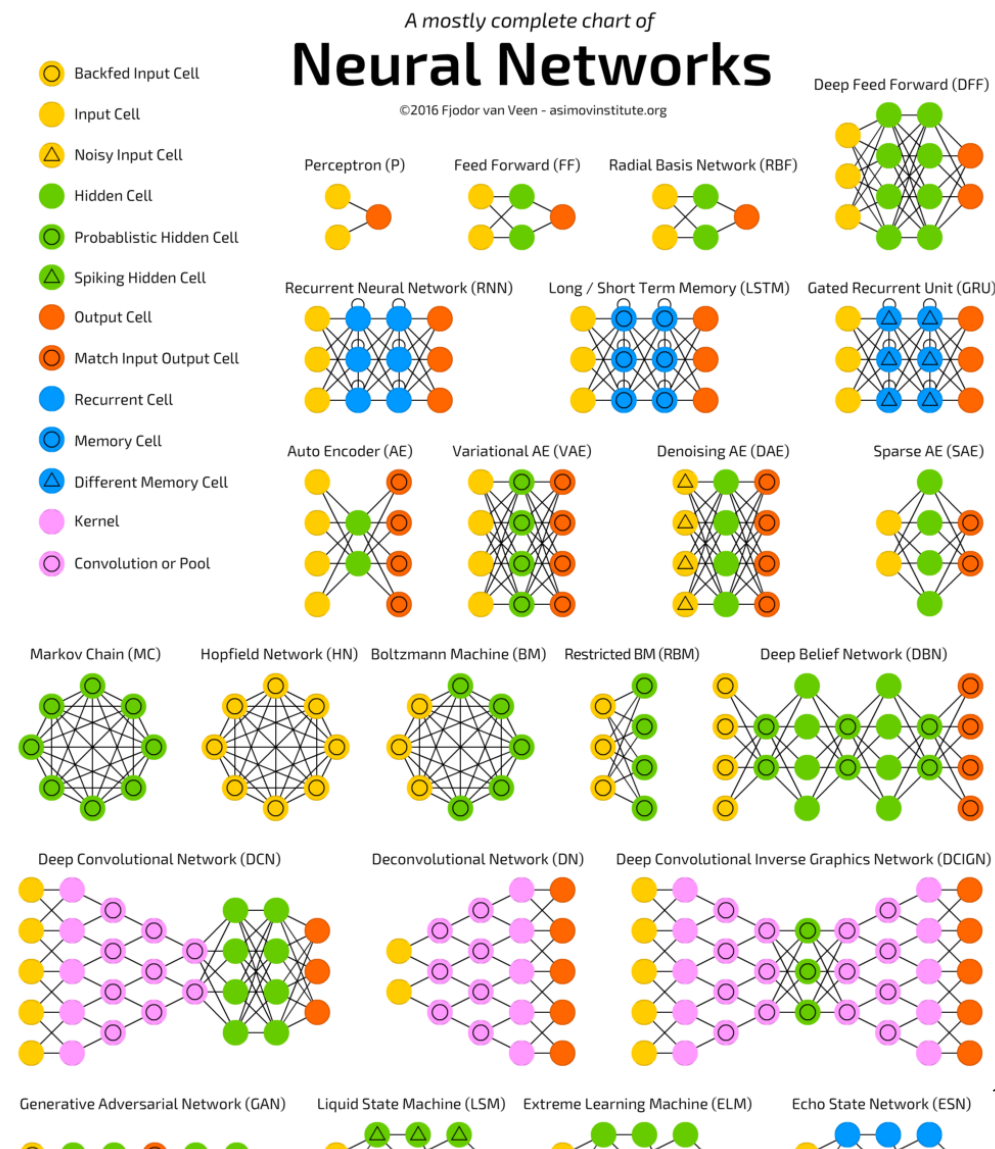
---

# What's in a name?

Different types of neural networks and their applications:

- **ANN**: Artificial Neural Network for regression and classification problems, with vectors as input data
- **CNN**: Convolutional Neural Network for image processing, image/face/... recognition, with images as input data
- **RNN**: Recurrent Neural Network for sequential data such as text or time series

... and many more!



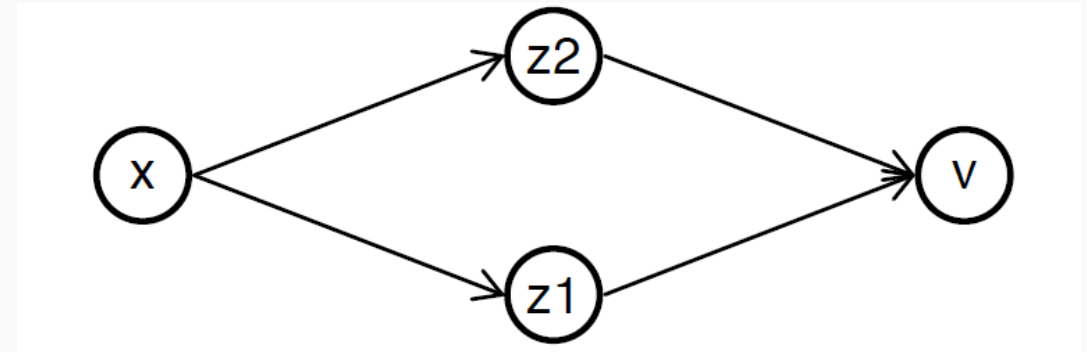
# A simple neural network

De-mystify artificial neural networks (ANNs):

- a collection of inter-woven linear models
- extending linear approaches to detect **non-linear** interactions in **high-dimensional** data.

See the picture on the right.

**Goal:** predict a scalar response  $y$  from scalar input  $x$ .



Some terminology:

- $x$  is the **input layer**
- $v$  is the **output layer**, to predict  $y$
- middle layer is a **hidden layer**
- four neurons:  $x$ ,  $z_1$ ,  $z_2$  and  $v$ .

# A simple neural network (cont.)

First, we apply two independent **linear models**:

$$z_1 = b_1 + x \cdot w_1$$

$$z_2 = b_2 + x \cdot w_2$$

using four parameters: two intercepts and two slopes.

Next, we construct **another linear model** with the  $z_j$  as inputs:

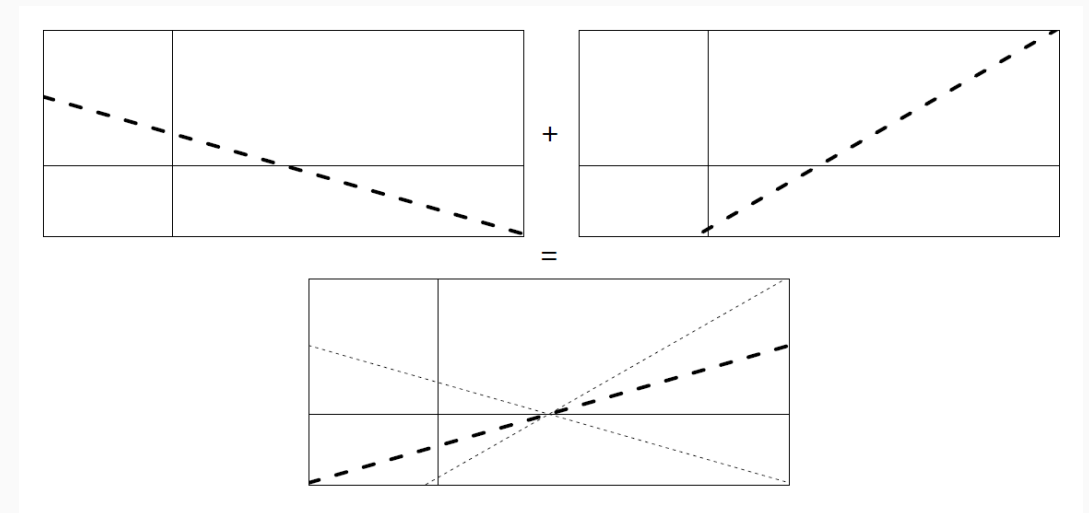
$$\hat{y} := v = b_3 + z_1 \cdot u_1 + z_2 \cdot u_2.$$

Putting it all together:

$$\begin{aligned} v &= b_3 + z_1 \cdot u_1 + z_2 \cdot u_2 \\ &= b_3 + (b_1 + x \cdot w_1) \cdot u_1 + (b_2 + x \cdot w_2) \cdot u_2 \\ &= (b_3 + u_1 \cdot b_1 + u_2 \cdot b_2) + (w_1 \cdot u_1 + w_2 \cdot u_2) \cdot x \\ &= (\text{intercept}) + (\text{slope}) \cdot x. \end{aligned}$$

Model is over-parametrized, with infinitely many ways to describe the same model.

Essentially, still a linear model!



# A simple neural network (cont.)

We capture **non-linear** relationships between  $x$  and  $v$  by replacing

$$v = b_3 + z_1 \cdot u_1 + z_2 \cdot u_2.$$

with

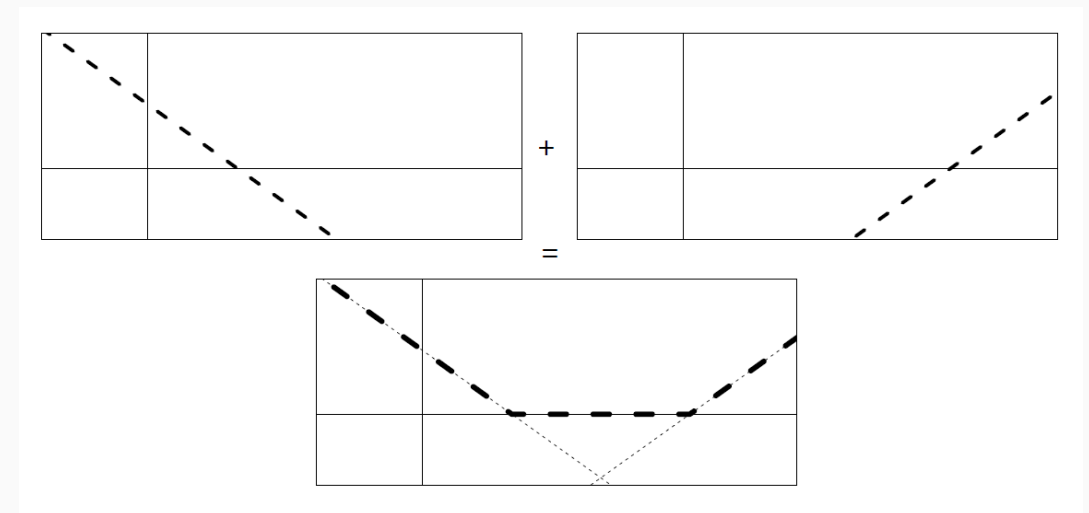
$$\begin{aligned} v &= b_3 + \sigma(z_1) \cdot u_1 + \sigma(z_2) \cdot u_2 \\ &= b_3 + \sigma(b_1 + x \cdot w_1) \cdot u_1 + \sigma(b_2 + x \cdot w_2) \cdot u_2, \end{aligned}$$

where  $\sigma(\cdot)$  is an **activation function**, a mapping from  $\mathbb{R}$  to  $\mathbb{R}$ .

Adding an activation function greatly increases the **set of possible relations** between  $x$  and  $v$ !

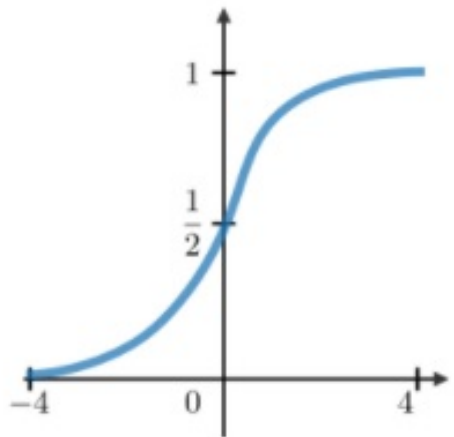
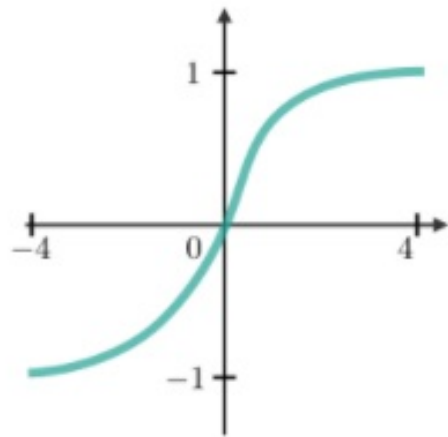
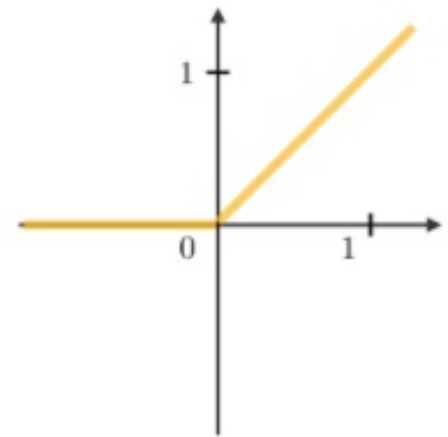
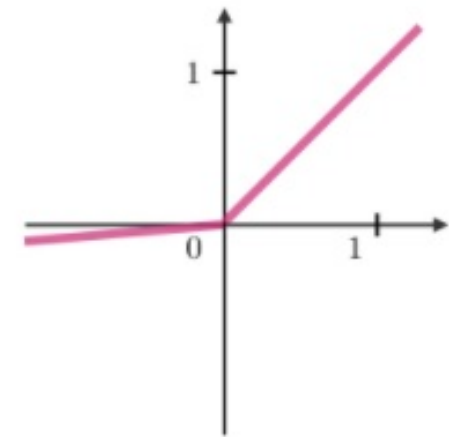
For example, the rectified linear unit (ReLU) activation function:

$$\text{ReLU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{otherwise.} \end{cases}$$



Many more activation functions: sigmoid, softmax, identity, etc. (see further).

# Examples of activation functions

Sigmoid	Tanh	ReLU	Leaky ReLU
$g(z) = \frac{1}{1 + e^{-z}}$	$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	$g(z) = \max(0, z)$	$g(z) = \max(\epsilon z, z)$ with $\epsilon \ll 1$
			

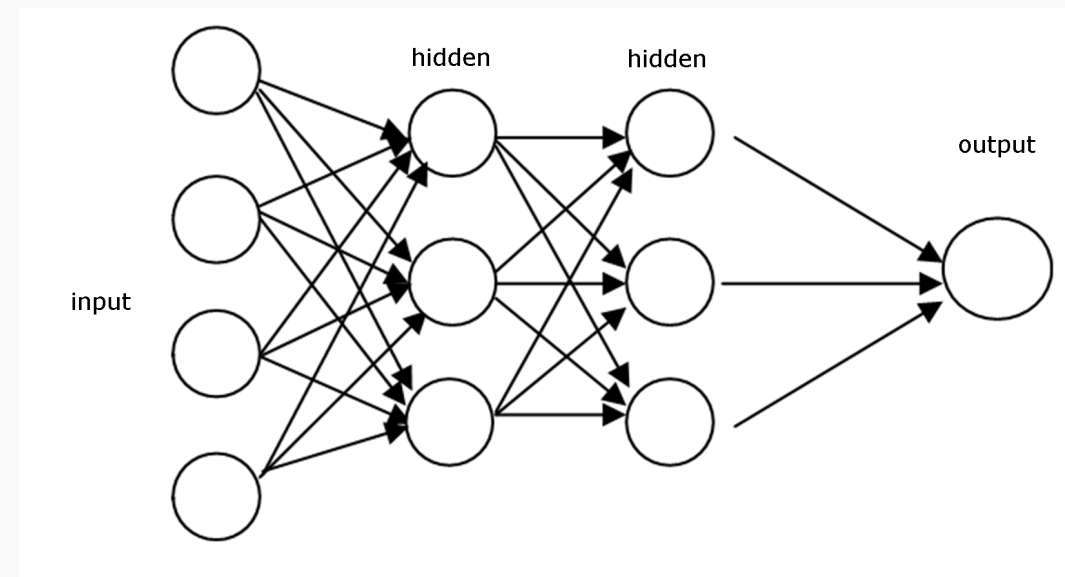
Source: <https://stanford.edu/~shervine/teaching/cs-229/cheatsheet-deep-learning>

# From the simple neural network to ANNs

Artificial Neural networks (ANNs):

- a collection of neurons
- organized into an ordered set of layers
- directed connections pass signals between neurons in adjacent layers
- **to train:**  
update parameters describing the connections by minimizing loss function over training data
- **to predict:**  
pass  $\mathbf{x}_i$  to first layer, output of final layer is  $\hat{y}_i$ .

The network is **dense** or **densely connected** if each neuron in a layer receives an input from all the neurons present in the previous layer.



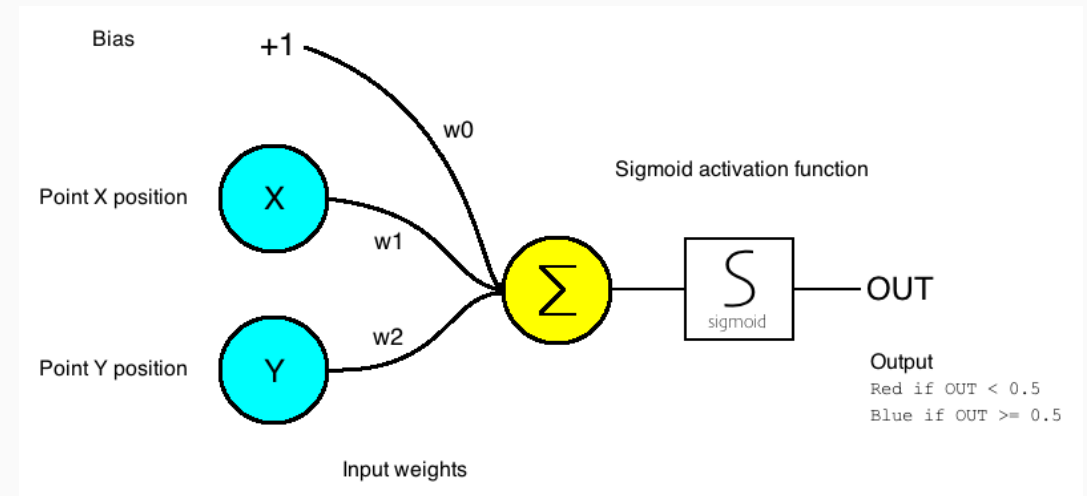
This is a **feedforward** neural network - no loops!



# The neural nets' terminology

Using the neural nets terminology or language:

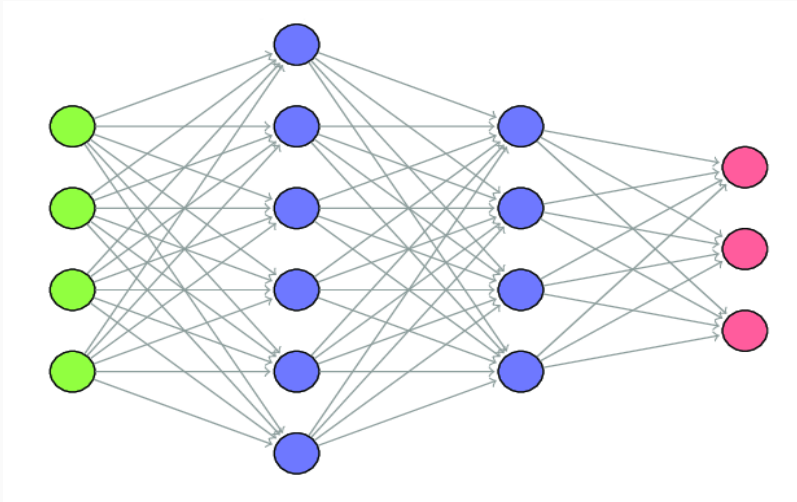
- intercept called **the bias**
- slopes called **weights**
- $L + 1$  layers in total, with input layer denoted as layer 0 and output layer as  $L$
- technically, **deep learning** refers to any neural network that has 2 or more hidden layers.



A single layer ANN, also called perceptron or artificial neuron.

# An architecture with layers

In a neural network, **input** travels through a sequence of **layers**, and gets transformed into the **output**.



This sequential layer structure is really at the core of the Keras library.

```
model ←  
keras_model_sequential() %>%  
  layer_dense( ... ) %>%  
  layer_dense( ... )
```

**Layers** consist of nodes and the **connections** between these nodes and the previous layer.

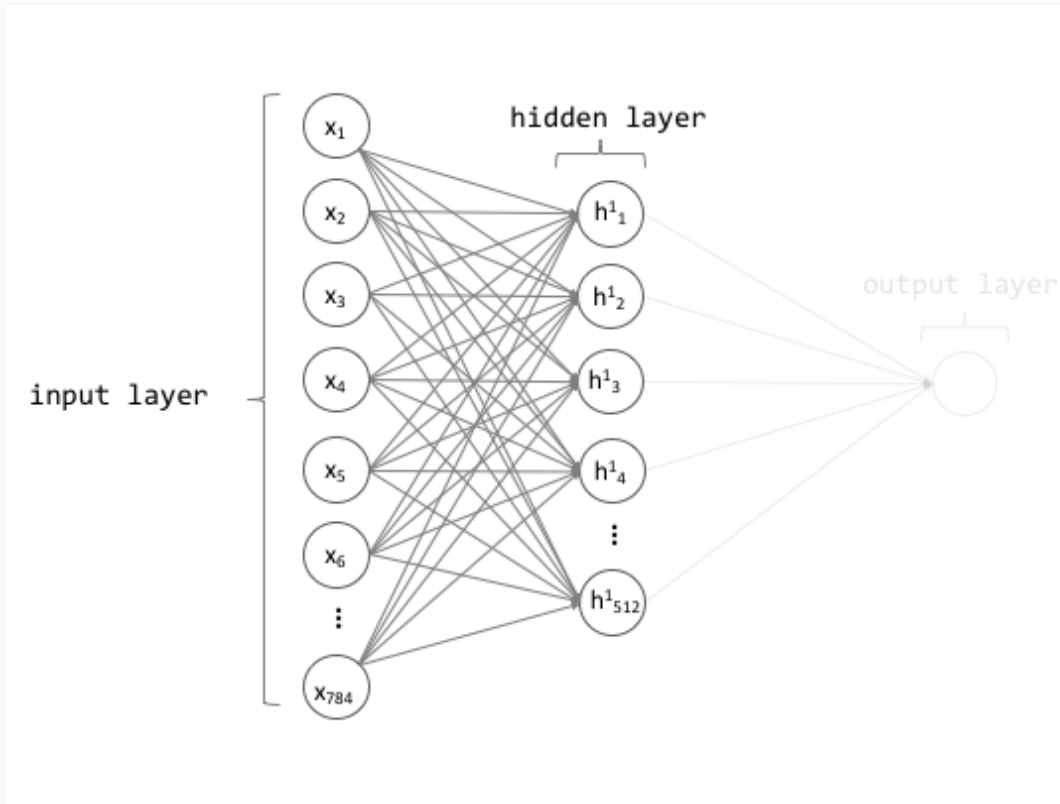
`layer_dense()` is creating a fully connected feed forward neural network.

# A hidden layer

```
model ← keras_model_sequential() %>%  
  layer_dense(units = 512, activation = 'relu', input_shape = c(784)) # hidden layer
```

- `units = 512`: number of nodes in the given layer
- `input_shape = c(784)`
  - tells the first hidden layer how many input features there are
  - only required for the first `layer_dense`
- `activation = 'relu'`: this hidden layer uses the ReLU activation function.

Here: a (28x28) picture is flattened to a an input vector of length 784.



# Output layer

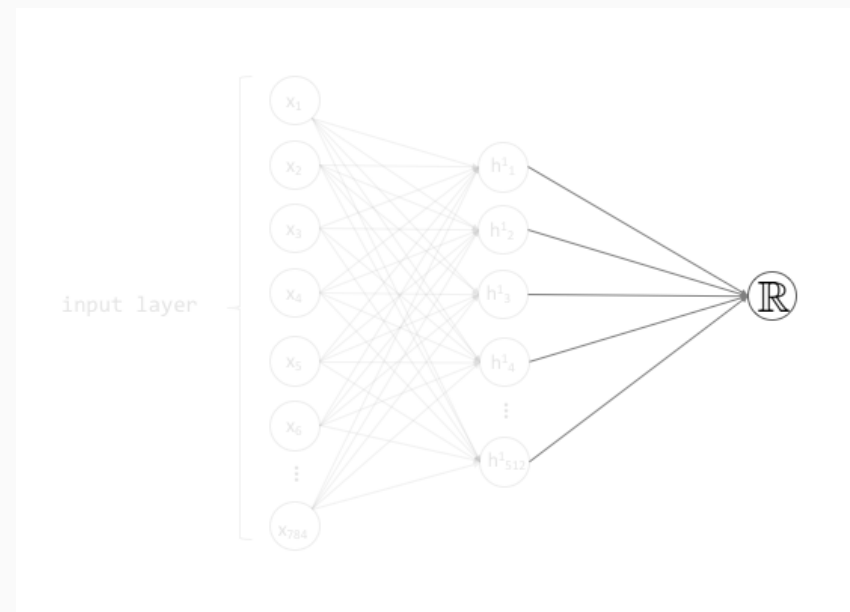
```
model ← keras_model_sequential() %>%  
  layer_dense(units = 512, activation = 'relu', input_shape = c(784)) %>%  
  layer_dense(units = 10, activation = 'softmax')
```

The choice of the `units` and `activation` function in the output layer depend on the type of prediction!

Two primary arguments of concern for the final output layer:

1. number of units

- regression: `units = 1`:



# Output layer

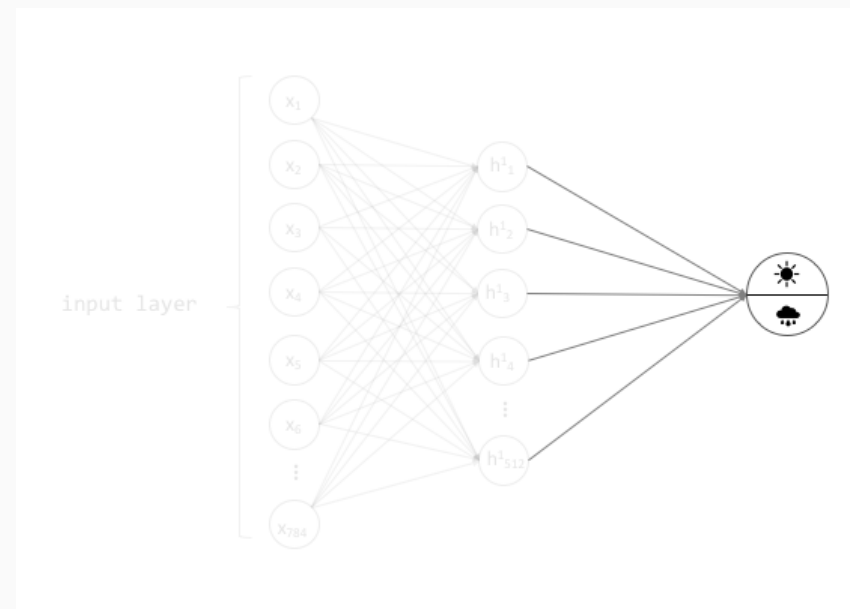
```
model ← keras_model_sequential() %>%  
  layer_dense(units = 512, activation = 'relu', input_shape = c(784)) %>%  
  layer_dense(units = 10, activation = 'softmax')
```

The choice of the `units` and `activation` function in the output layer depend on the type of prediction!

Two primary arguments of concern for the final output layer:

1. number of units

- regression: `units = 1`
- binary classification: `units = 1`



# Output layer

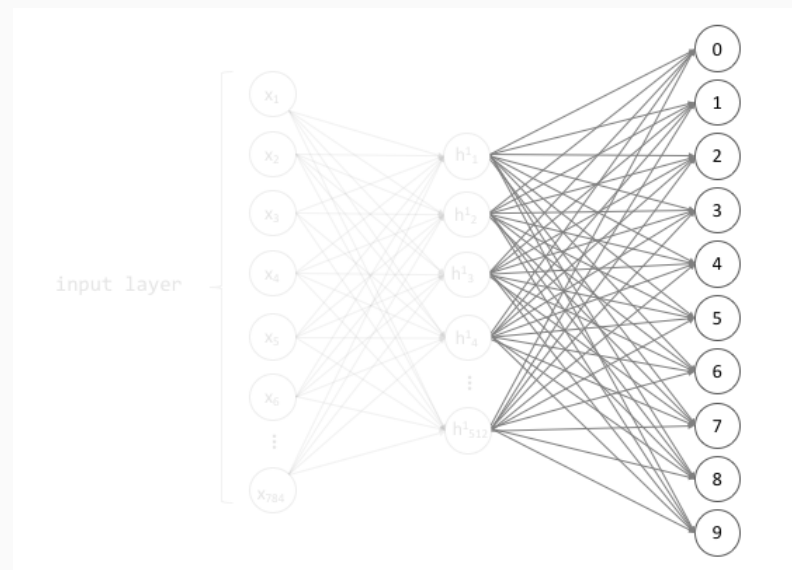
```
model ← keras_model_sequential() %>%  
  layer_dense(units = 512, activation = 'relu', input_shape = c(784)) %>%  
  layer_dense(units = 10, activation = 'softmax')
```

The choice of the `units` and `activation` function in the output layer depend on the type of prediction!

Two primary arguments of concern for the final output layer:

## 1. number of units

- regression: `units = 1`
- binary classification: `units = 1`
- multi-class classification: `units = n`



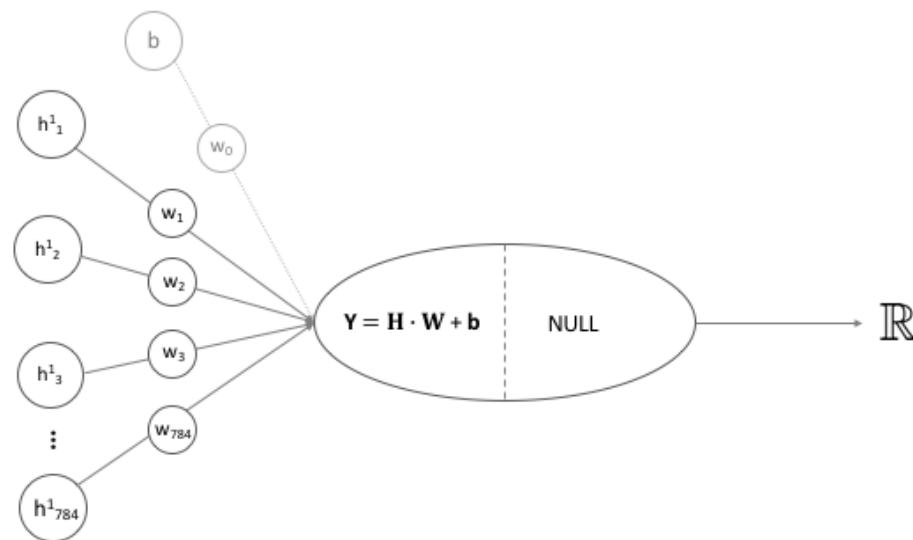
# Output layer

```
model ← keras_model_sequential() %>%  
  layer_dense(units = 512, activation = 'relu', input_shape = c(784)) %>%  
  layer_dense(units = 10, activation = 'softmax')
```

The choice of the `units` and `activation` function in the output layer depend on the type of prediction!

Two primary arguments of concern for the final output layer:

1. number of units
2. activation function
  - regression: `activation = NULL` (identity function)



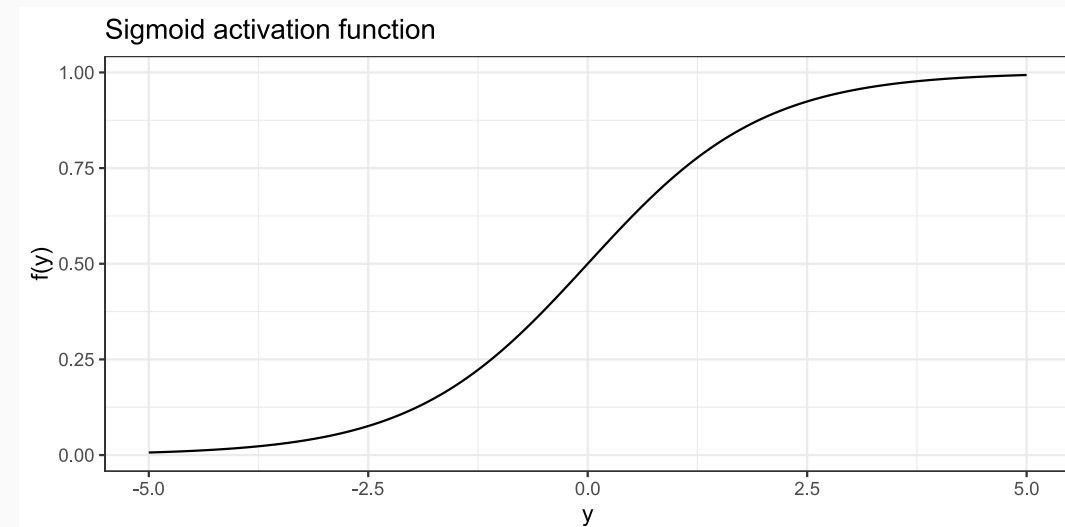
# Output layer

```
model ← keras_model_sequential() %>%  
  layer_dense(units = 512, activation = 'relu', input_shape = c(784)) %>%  
  layer_dense(units = 10, activation = 'softmax')
```

The choice of the `units` and `activation` function in the output layer depend on the type of prediction!

Two primary arguments of concern for the final output layer:

1. number of units
2. activation function
  - regression: `activation = NULL` (identity function)
  - binary classification: `activation = 'sigmoid'`



$$f(y) = \frac{1}{1+e^{-y}}$$



# Output layer

```
model ← keras_model_sequential() %>%  
  layer_dense(units = 512, activation = 'relu', input_shape = c(784)) %>%  
  layer_dense(units = 10, activation = 'softmax')
```

The choice of the `units` and `activation` function in the output layer depend on the type of prediction!

Two primary arguments of concern for the final output layer:

1. number of units
2. activation function
  - regression: `activation = NULL` (identity function)
  - binary classification: `activation = 'sigmoid'`
  - multi-class classification: `activation = 'softmax'`

<u>Output node</u>	<u>Linear transformation</u>	<u>Softmax Activation</u>	<u>Probabilities</u>
0	$Y_0 = H \cdot W + b$	$f(y_0) = \frac{e^{y_0}}{\sum_i e^{y_i}}$	0.01
1	$Y_1 = H \cdot W + b$	$f(y_1) = \frac{e^{y_1}}{\sum_i e^{y_i}}$	0.09
2	$Y_2 = H \cdot W + b$	$f(y_2) = \frac{e^{y_2}}{\sum_i e^{y_i}}$	0.85
3	$Y_3 = H \cdot W + b$	$f(y_3) = \frac{e^{y_3}}{\sum_i e^{y_i}}$	0.11
⋮	⋮	⋮	
8	$Y_8 = H \cdot W + b$	$f(y_8) = \frac{e^{y_8}}{\sum_i e^{y_i}}$	0.01
9	$Y_9 = H \cdot W + b$	$f(y_9) = \frac{e^{y_9}}{\sum_i e^{y_i}}$	<u>0.01</u>
			1.00

# Network compilation

---

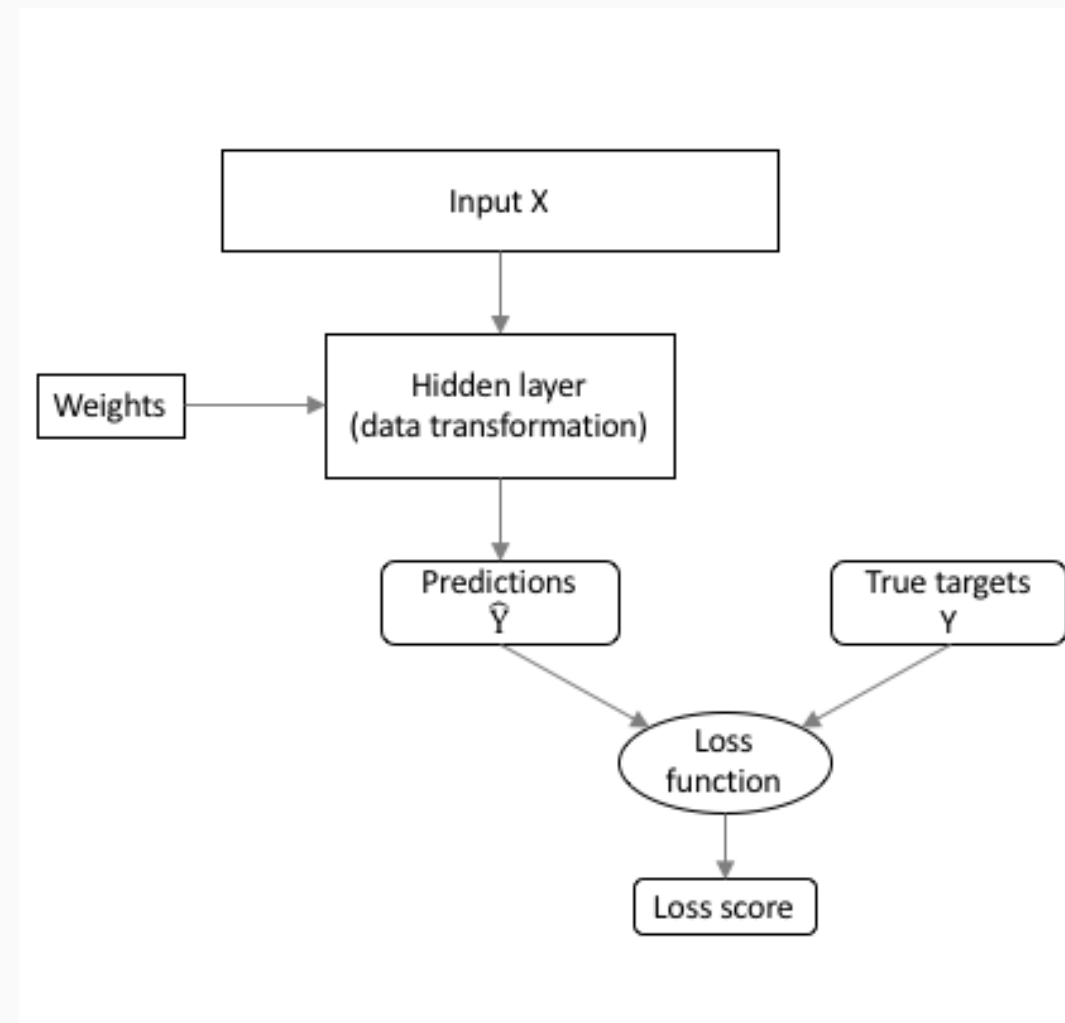
# Loss function and forward pass

- Initialize weights (randomly).
- The forward pass then results in predicted values  $\hat{y}$ , to be compared with  $y$ .
- The difference is measured with a loss function, the quantity that will be minimized during training.

Some **common loss functions**:

- "mse": Gaussian
- "poisson": Poisson
- "binary\_crossentropy": binary classification
- "categorical\_crossentropy": multi-class classification
- many others, see the [Keras documentation](#)

Pick a loss function that aligns best to the problem at hand!



# Compiling the model

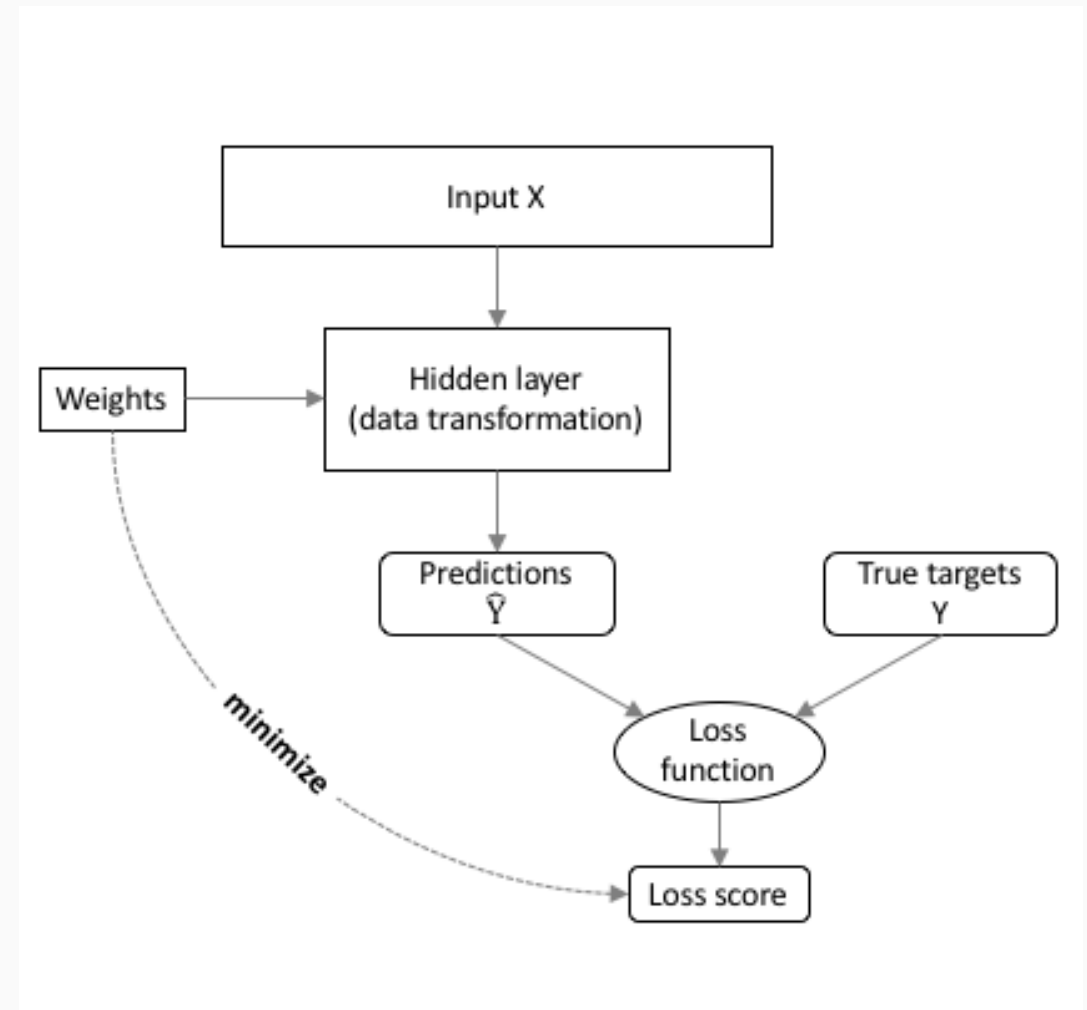
```
model ← model %>%  
  compile(loss = "categorical_crossentropy",  
          optimize = optimizer_rmsprop(),  
          metrics = c('accuracy'))
```

Keras includes several **optimizers** for minimizing the loss function.

Popular choices are:

- `optimizer_rmsprop()`
- `optimizer_adam()`
- other optimizers, see the [Keras documentation](#)

The goal is to find weights and bias terms that **minimize the loss function**.



# Gradient descent and backpropagation

In general terms, we want to find (with  $w$  for all unknown parameters)

$$\min_w \mathcal{L}(w).$$

With **gradient descent**: we'll move in the direction the loss locally decreases the fastest!

Thus,

$$w_{\text{new}} = w_{\text{old}} - \eta \cdot \nabla_w \mathcal{L}(w_{\text{old}}),$$

with learning rate  $\eta$ .

With a loss function evaluated over  $n$  training data points (cfr. supra on *epochs* and *minibatches*)

$$\nabla_w \mathcal{L}(w) = \frac{1}{n} \sum_{i=1}^n \nabla_w \mathcal{L}_i$$

# Gradient descent and backpropagation

In general terms, we want to find (with  $w$  for all unknown parameters)

$$\min_w \mathcal{L}(w),$$

With **gradient descent**: we'll move in the direction the loss locally decreases the fastest!

Thus,

$$w_{\text{new}} = w_{\text{old}} - \eta \cdot \nabla_w \mathcal{L}(w_{\text{old}}),$$

with learning rate  $\eta$ .

With a loss function evaluated over  $n$  training data points (cfr. supra on *epochs* and *minibatches*)

$$\nabla_w \mathcal{L}(w) = \frac{1}{n} \sum_{i=1}^n \nabla_w \mathcal{L}_i$$

Computing the gradient of the loss function wrt all trainable parameters:

- tons of parameters
- need for efficient algorithm to calculate gradient
- need for generic algorithm usable for arbitrary number of layers and neurons in each layer.

The strategy (Rumelhart et al., 1986, Nature)

- **backpropagation**
- derivatives in outer layer  $L$  are easy
- derivatives in layer  $l$  as a function of derivatives in layer  $l + 1$
- all about the **chain rule** for derivatives!

# Three variants of gradient descent

With **batch** gradient descent:

- compute loss for each observation in the training data
- update parameters after all training examples have been evaluated
- **con**: scales horribly to bigger data sets.

With **stochastic** gradient descent:

- randomly select an observation, compute gradient
- update parameters after this single observation has been evaluated
- **con**: takes a long time to convergence.

With **mini-batch** gradient descent:

- randomly select a subset of the training observations, compute gradient
- update parameters after this subset has been evaluated.

**Pros:**

- balance efficiency of batch vs stochastic
- balance robust convergence of batch with some stochastic nature to avoid local minima.

**Cons:**

- additional tuning parameter.

# Summary of the fundamentals

Our introduction to neural nets focused on:

- design neural networks **sequentially** in {keras}  
`keras_model_sequential`
- layers consist of **nodes** and **connections**
- vanilla choice is a **fully connected layer**  
`layer_dense`
- **fit** the model via gradient descent (i.e. backpropagation).

List of **tuning/architectural** choices: (*more on these would require more time*)

- the number of layers
- the number of nodes per layer
- the activation functions
- the layer type
- the loss function
- the optimization algorithm
- the batch size
- the number of epochs
- ...



# Claim frequency and severity regression

---

# Regression with neural networks

Actuaries often consider **GLMs**, for instance for claim frequency data:

$$Y \sim \text{Poisson}(\lambda = \exp(x' \beta)).$$

We now **redefine** this model as a **neural network**:

Formula	GLM	Neural network
$Y$	response	output node
Poisson	distribution	loss function
exp	inverse link function	activation function
$x$	predictors	input nodes
$\beta$	fitted effect	weights

# Adding a skip connection in a neural network

So far, we stayed in a **purely sequential** architecture.

Now, we will allow some input nodes to be connected directly to the output node, i.e., **skip connections**.

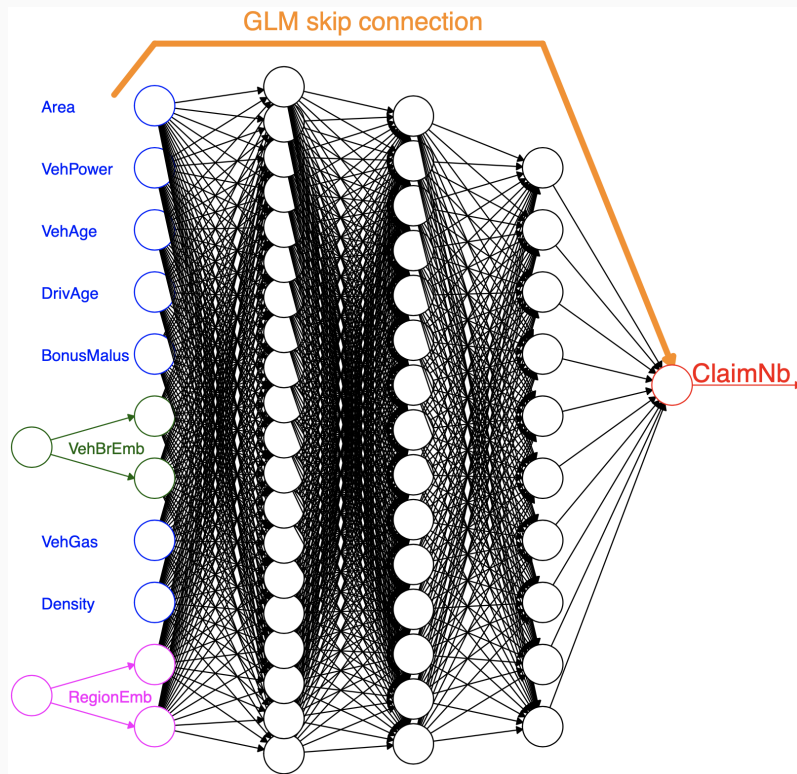


Figure taken from [Schelldorfer and Wuthrich \(2019\)](#).

The output node, without skip connection, calculates (with  $\sigma(\cdot)$  the activation function):

$$\sigma\left(\sum_i w_i h_i + b\right).$$

With a skip connection, this simply becomes:

$$\sigma\left(\sum_i w_i h_i + b + s\right).$$

We take a **linear** combination of the last hidden layer outputs and **add** the skip input, **before** applying the activation function.

So, what can we do with this?

# Adding a skip connection in a neural network (cont.)

Let's take a **claim frequency** example with the `exponential` activation function.

- Adding exposure as an **offset** term:

$$output = \exp\left(\sum_i w_i h_i + b + \log(expo)\right) = expo \cdot \exp\left(\sum_i w_i h_i + b\right).$$

- Adding a **base** prediction:

$$output = \exp\left(\sum_i w_i h_i + b + \log(base)\right) = base \cdot \exp\left(\sum_i w_i h_i + b\right).$$

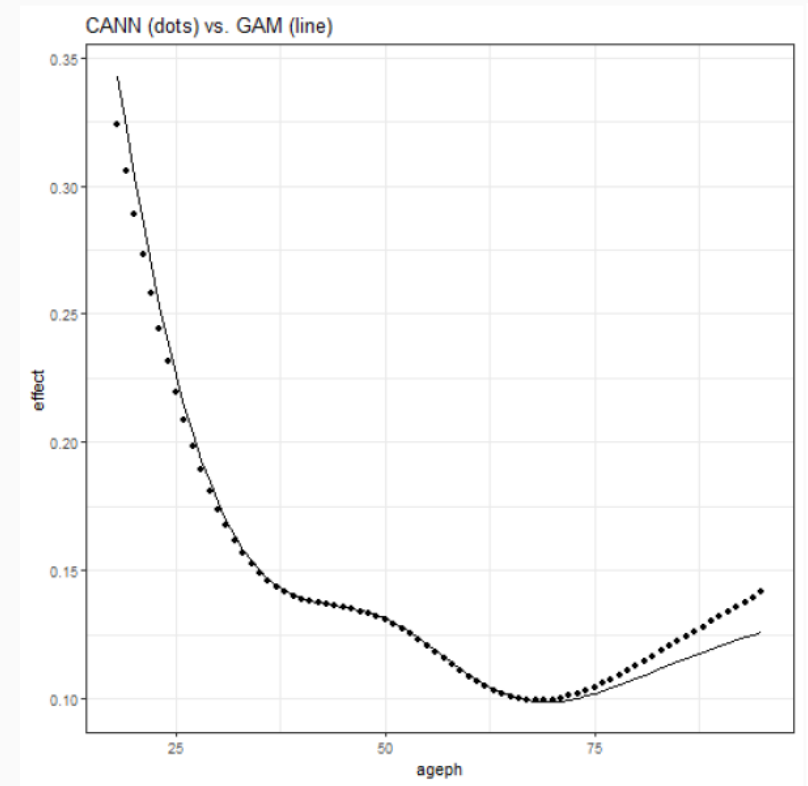
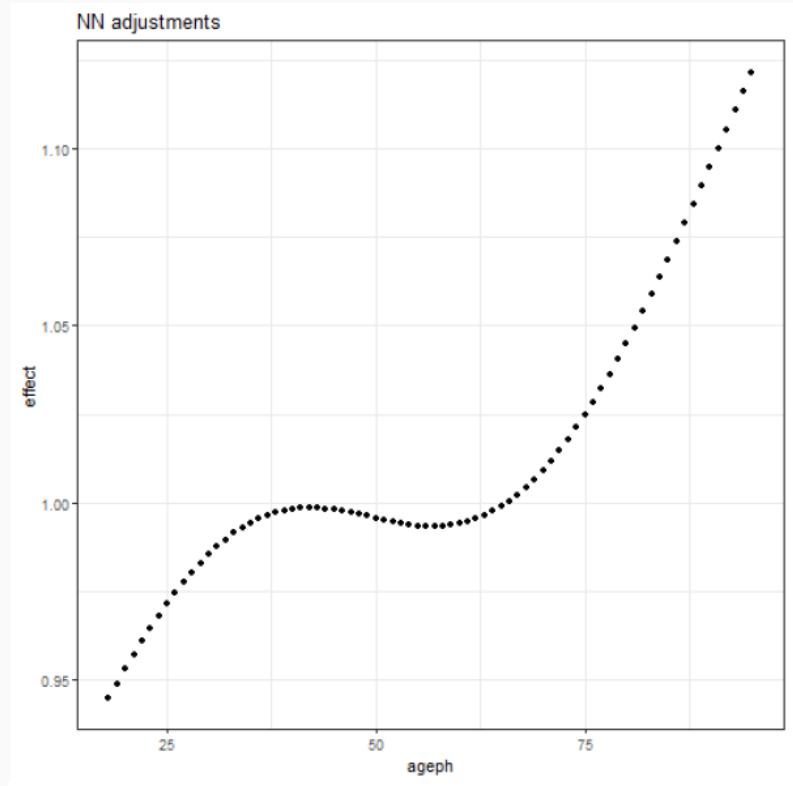
- The **combination** of both:

$$output = \exp\left(\sum_i w_i h_i + b + \log(expo \cdot base)\right) = expo \cdot base \cdot \exp\left(\sum_i w_i h_i + b\right).$$

A skip connection allows us to guide the neural net in the right direction and to model **adjustments** on top of the base predictions, for example obtained via a GLM or GAM.

In the actuarial lingo this is called a **C**ombined **A**ctuarial **N**eural **N**etwork (**CANN**).

# Adding a skip connection in a neural network (cont.)



# Throwback to last year's seminar



Henckaerts et al. (2021) paper on [Boosting insights in insurance tariff plans with tree-based machine learning methods](#)

- full algorithmic details of regression trees, bagging, random forests and gradient boosting machines
- with focus on claim frequency and severity modelling
- including interpretation tools (VIP, PDP, ICE, H-statistic)
- model comparison (GLMs, GAMs, trees, RFs, GBMs)
- managerial tools (e.g. loss ratio, discrimination power).

The paper comes with two notebooks, see [examples tree-based paper](#) and [severity modelling](#).

The paper comes with an R package for fitting random forests on insurance data, see [distRforest](#).

# Ongoing research

- ANNs and CANNs for both **claim frequency and severity (seperately)**, and then their **combination into a technical tariff**
- CANNs with input from (smartly engineered) GLM and GBM, with
  - **fixed** input (say  $\hat{y}_i^{(in)}$ ) used via skip connection
  - input used via skip, but **flexible** (weights are trained)

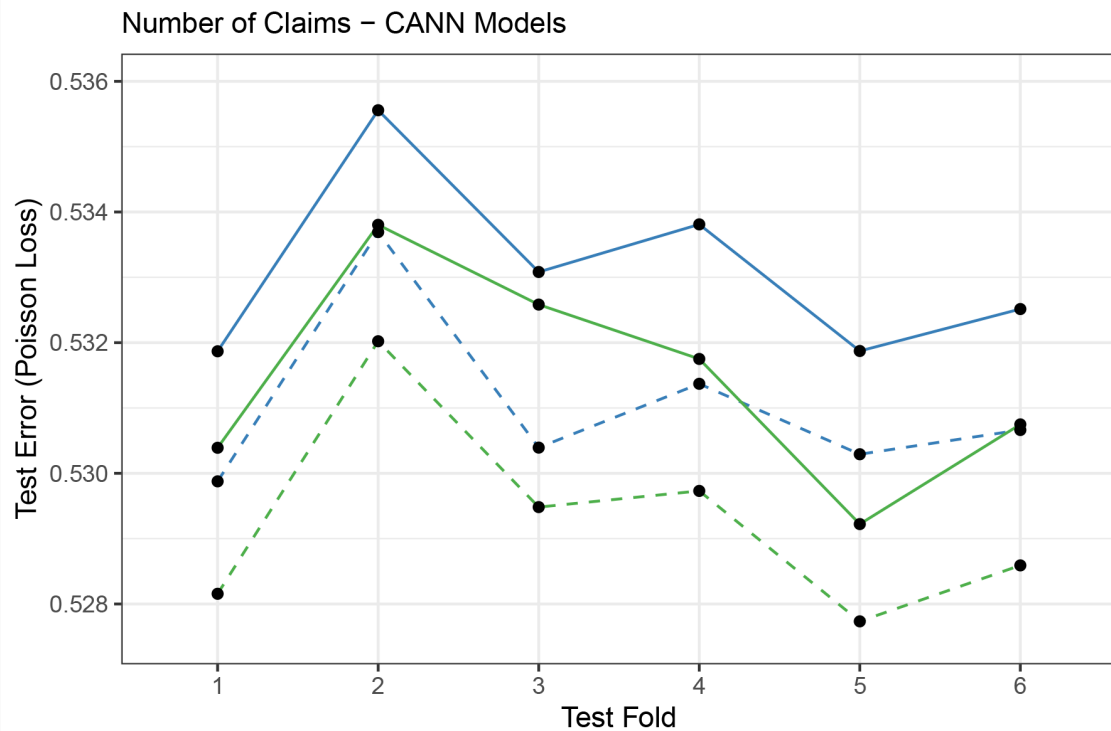
$$f^{fixed}(\mathbf{x}_i, \hat{y}_i^{(in)}) = \exp\left(\ln(\hat{y}_i^{(in)}) + \hat{y}_i^{(adj)}\right)$$

$$f^{flex}(\mathbf{x}_i, \hat{y}_i^{(in)}) = \exp\left([w_1 \quad w_2] \cdot \left[\ln(\hat{y}_i^{(in)}) \quad \hat{y}_i^{(adj)}\right]^t + b\right)$$

- **bias regularization**
  - in a GLM with canonical link  $\sum_i y_i = \sum_i \hat{f}(\mathbf{x}_i)$
  - how to restore this balance in a neural net?
- **preprocessing** steps of categorical inputs
  - one-hot encoding:  $p$  levels into  $p$  binary inputs
  - embedding layers: transform  $p$  levels into  $\mathbb{R}^d$
- **interpretation tools**
  - partial dependence plots (PDPs)
  - variable importance plots.

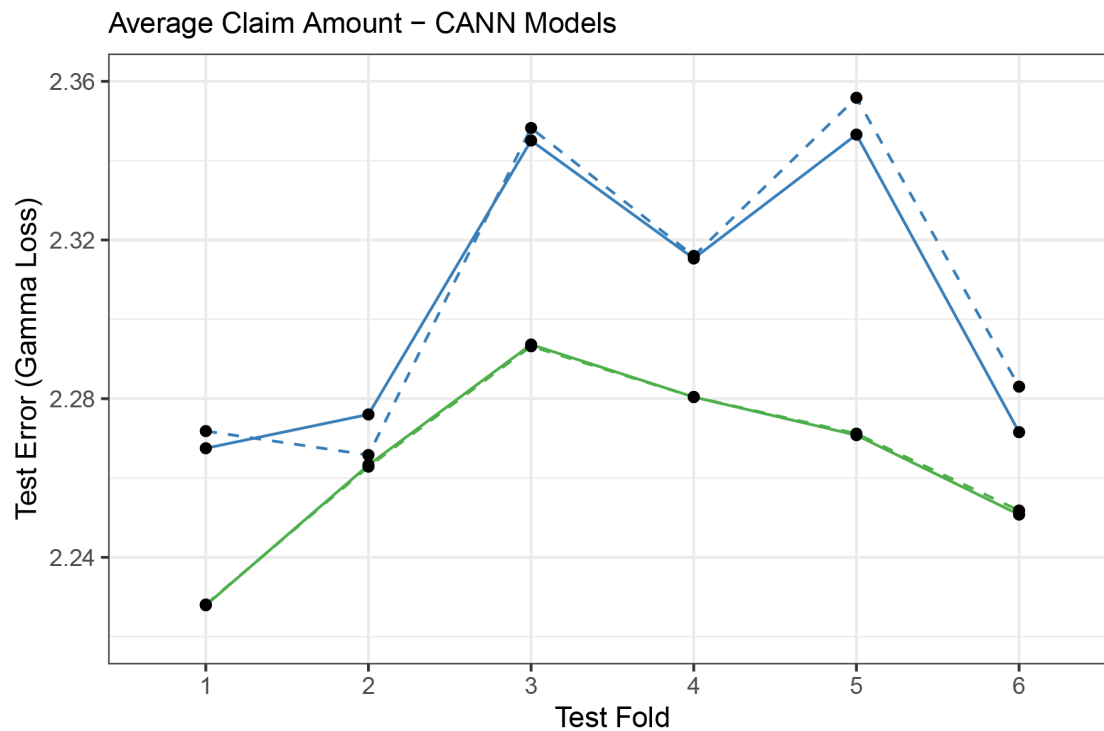
# Ongoing research (cont.)

## Test Performance all CANN Models



### Model

- CANN model with GLM input
- - CANN model with GLM input (Trainable Output)
- CANN model with GBM input
- - CANN model with GBM input (Trainable Output)

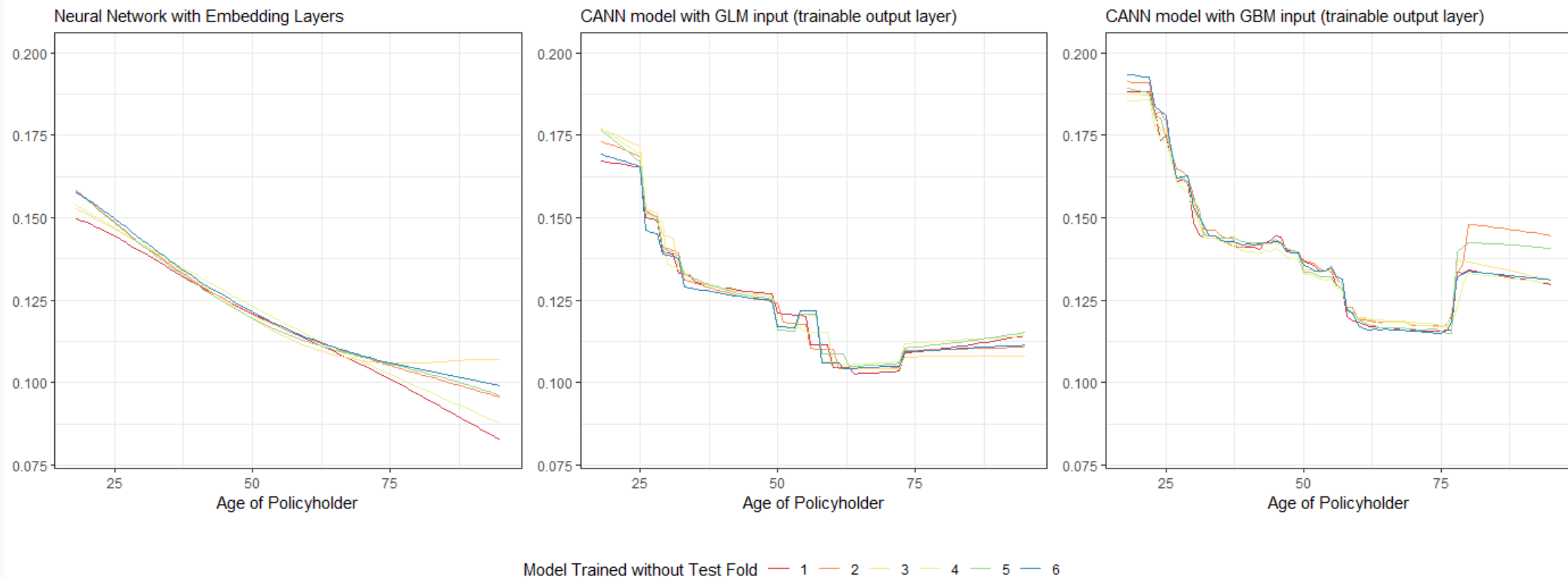


### Model

- CANN model with GLM input
- - CANN model with GLM input (Trainable Output)
- CANN model with GBM input
- - CANN model with GBM input (Trainable Output)

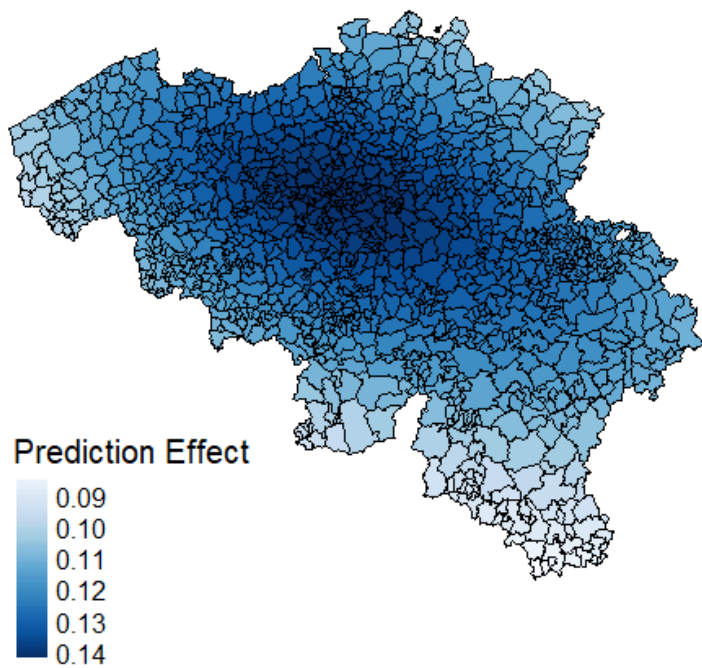


# Ongoing research (cont.)

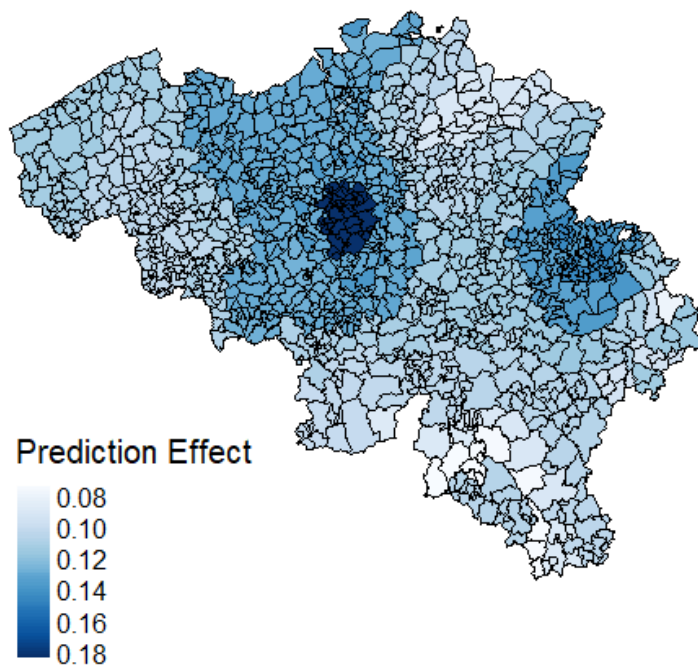


# Some first results (cont.)

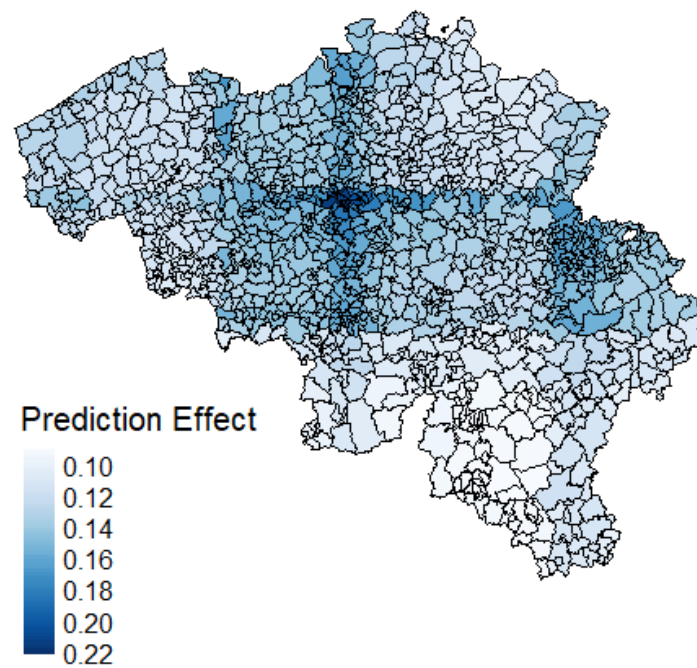
Regular NN with Embedding



CANN with GLM Input (Fixed)



CANN with GBM Input (Fixed)



# Conclusions

- **Insights in the working principles** behind (simple) neural networks, and their use for regression problems with tabular data.
- However, first experiments indicate that **such neural nets need the input of a base model** (e.g., a GLM or GBM) to be competitive with these actuarial predictive models in terms of predictive accuracy as well as interpretation of fitted effects of variables.
- But, they have a competitive advantage when **input data become more large and more complex** (e.g., v-a heat maps collected with telematics devices, together with more traditional input features).

# Thanks!



Thanks to the organizers, CNP Assurances and the chaire DIALog.

Slides created with the R package `xaringan`.

For more information please visit

 <https://github.com/katrienantonio>

 <https://katrienantonio.github.io>

 <https://chaire-dialog.fr/>