KU LEUVEN

Demystifying neural networks and their use in actuarial tasks

IA|BE Data Science Certificate, edition October 2021

Katrien Antonio

November 23, 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
- discuss how to evaluate and interpret neural networks
- step from simple networks (for regression) to more complex types of networks (e.g., convolutional neural networks) (if time permits).

Want to read more?

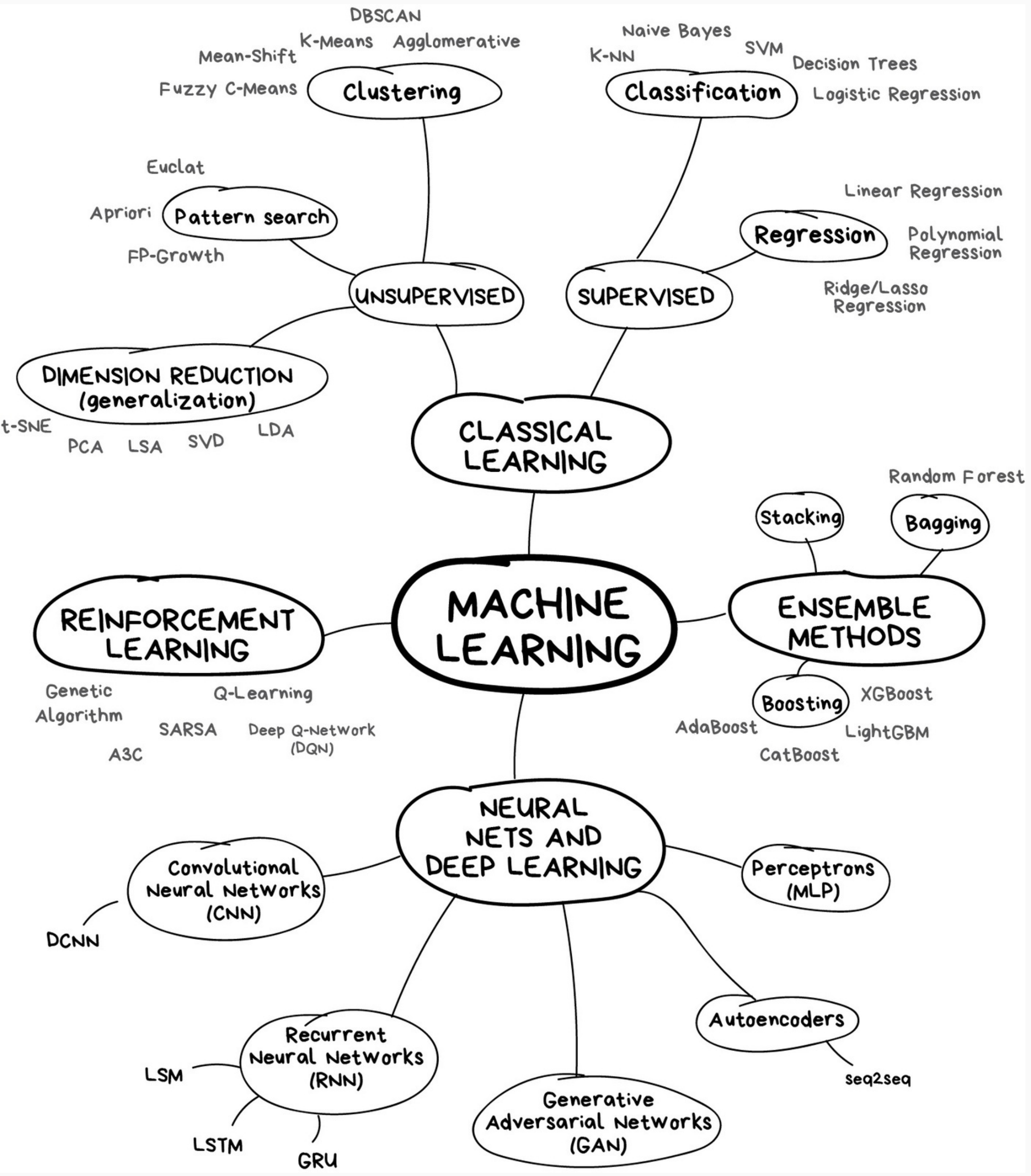
This presentation is based on

- 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.

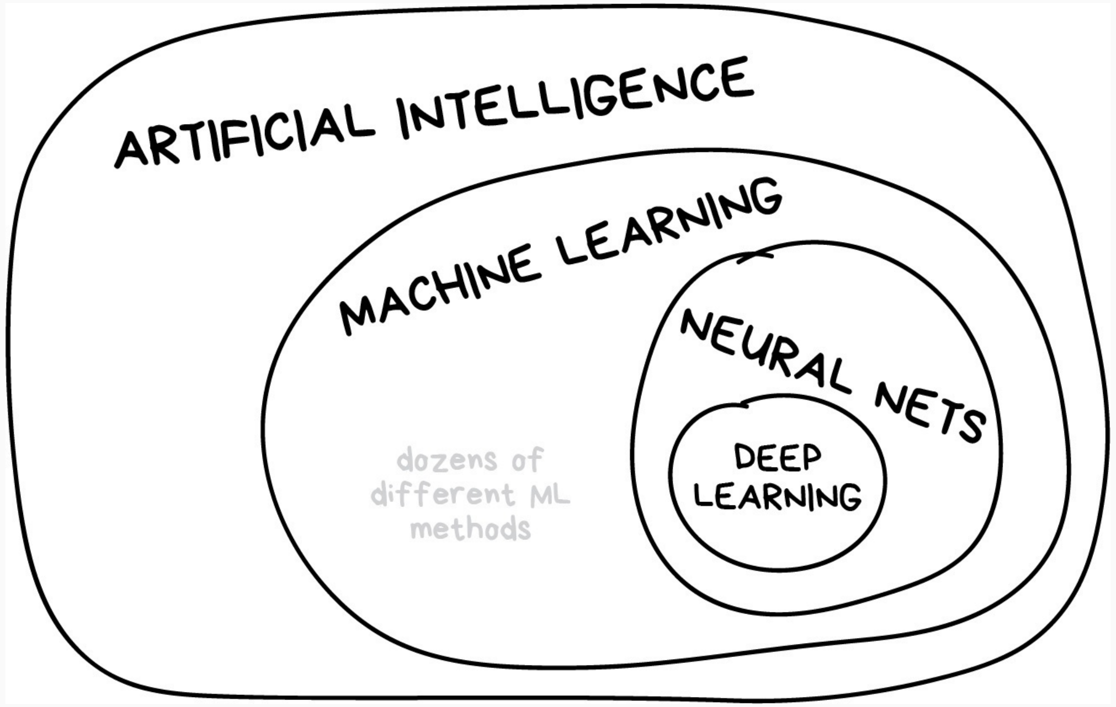
Actuarial modelling with neural nets is covered in (among others)

- 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



Some roadmaps to explore the ML landscape...



Source: Machine Learning for Everyone In simple words. With real-world examples. Yes, again.

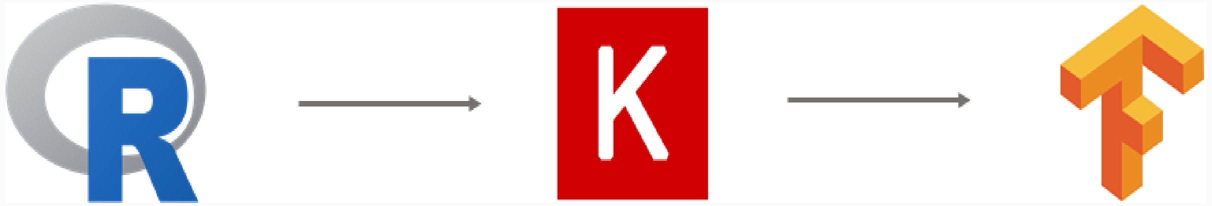
Getting started

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 to 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 inuitive 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.

Why is this thing called TensorFlow?

A scalar is a single number, or a 0D tensor, i.e. **zero dimensional**:

`age_car = 5`, `fuel = gasoline`, `bm = 10`

In tensor parlance a scalar has 0 axes.

In a **big data world** with structured and unstructured data, our **input** can be a

- a single time series: 1-dimensional, with 1 axis
- a sound fragment: 2-dimensional, with 2 axes
- an image in color: 3-dimensional, with 3 axes
- a movie: 4-dimensional, with 4 axes
- ...

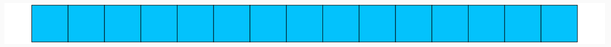
We require a framework that can flexibly adjust to all these data structures!

Why is this thing called TensorFlow? (cont.)

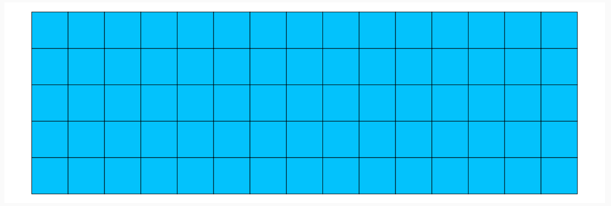
TensorFlow is this flexible framework which consists of highly optimized functions based on **tensors**.

What is a **tensor**?

- A 1-dimensional tensor is a vector (e.g. closing daily stock price during 250 days)



- A 2-dimensional tensor is a matrix (e.g. a tabular data set with observations and features)



- ...

Tensors generalize vectors and matrices to an arbitrary number of dimensions.

Many matrix operations, such as the matrix product, can be generalized to tensors.

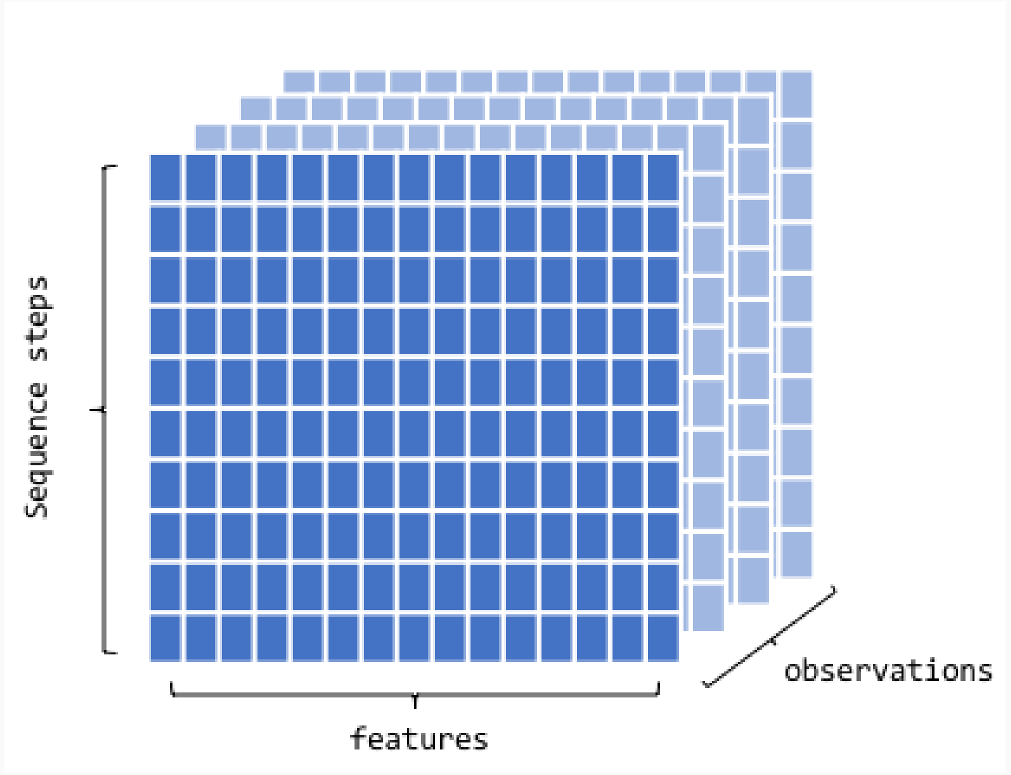
Luckily Keras provides a high level interface to TensorFlow, such that we will have only minimal exposure to tensors and the complicated math behind them.

Example of a 3D tensor

Let's picture a stock price dataset where

- each minute we record the current price, lowest price and highest price
- a trading day has 390 minutes and a trading year has 250 days.

Then, one year of data can then be stored in a 3D tensor `(samples, timesteps, features)`, here: `(250, 390, 3)`.



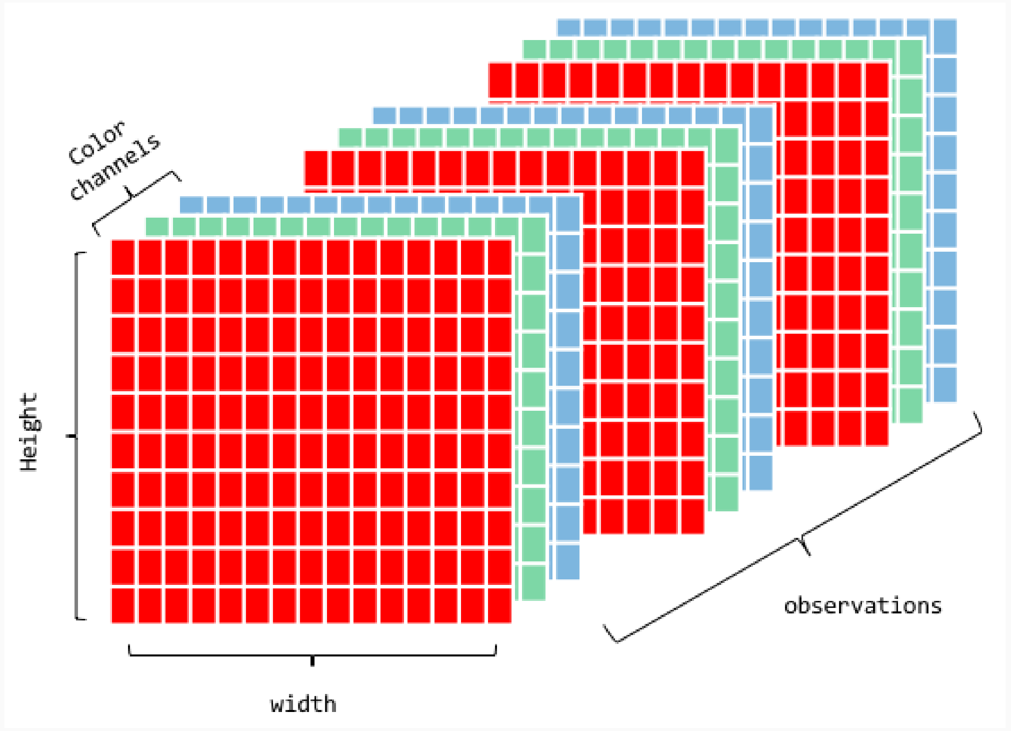
Source: Bradley Boehmke

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

Example of a 5D tensor

Let's picture a video data set where

- each video sample is one minute long and has a number of frames per second (e.g. 4 frames per second)
- each frame has a specific height (e.g. 256 pixels) and width (e.g. 144 pixels)
- three color channels (Red, Green, Blue)
- multiple images (`samples`) are stored.

Then, a collection of images can be stored in a 5D tensor `(samples, frames, height, width, channels)` which becomes here `(samples, 240, 256, 144, 3)`.

Source: Bradley Boehmke

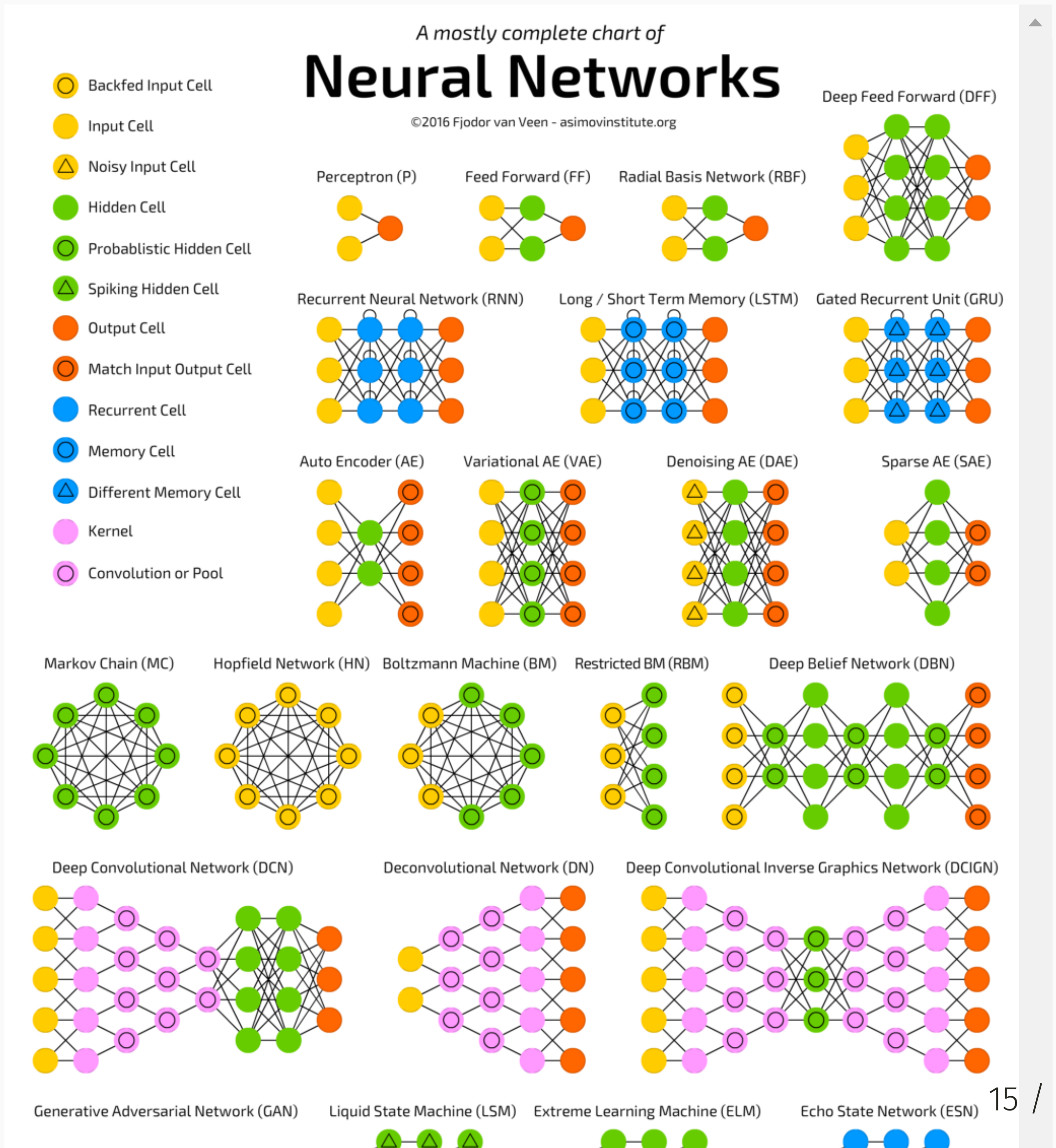
De-mystifying neural networks

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!



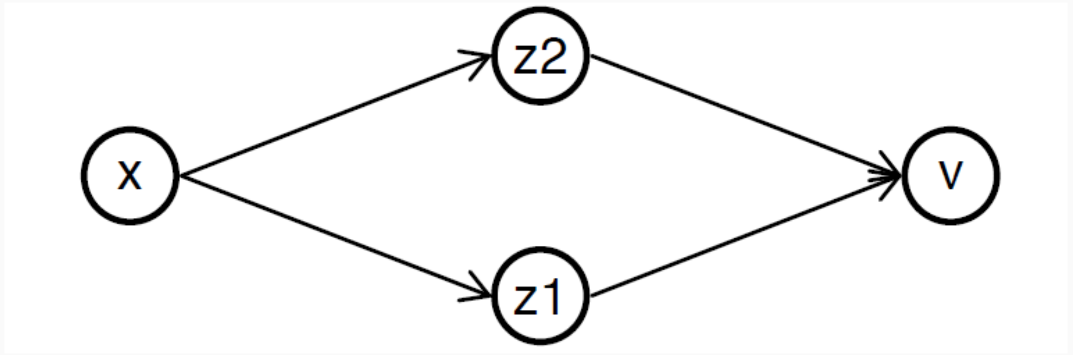
A simple neural network

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:

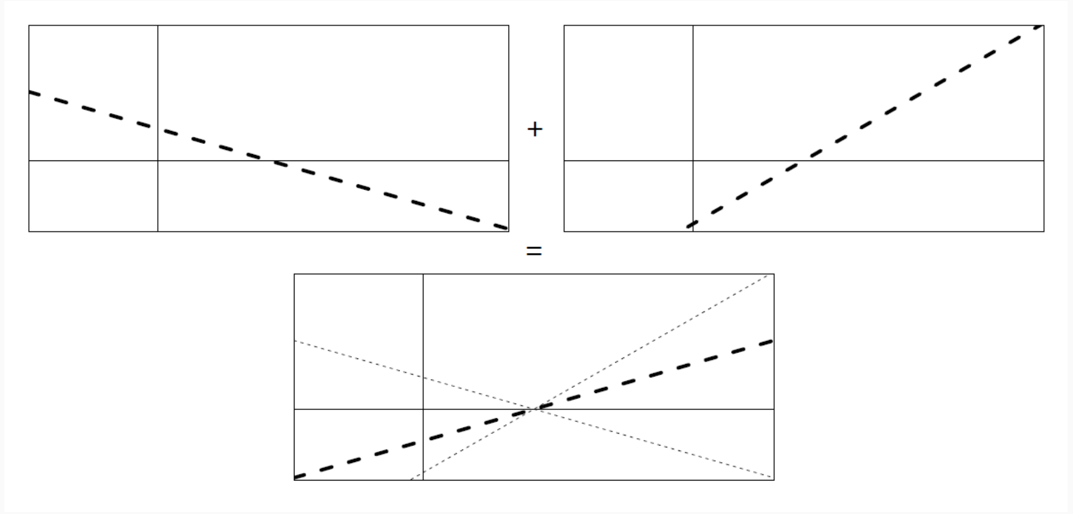
$$\hat{y} := v = b_3 + z_1 \cdot u_1 + z_2 \cdot u_2.$$

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

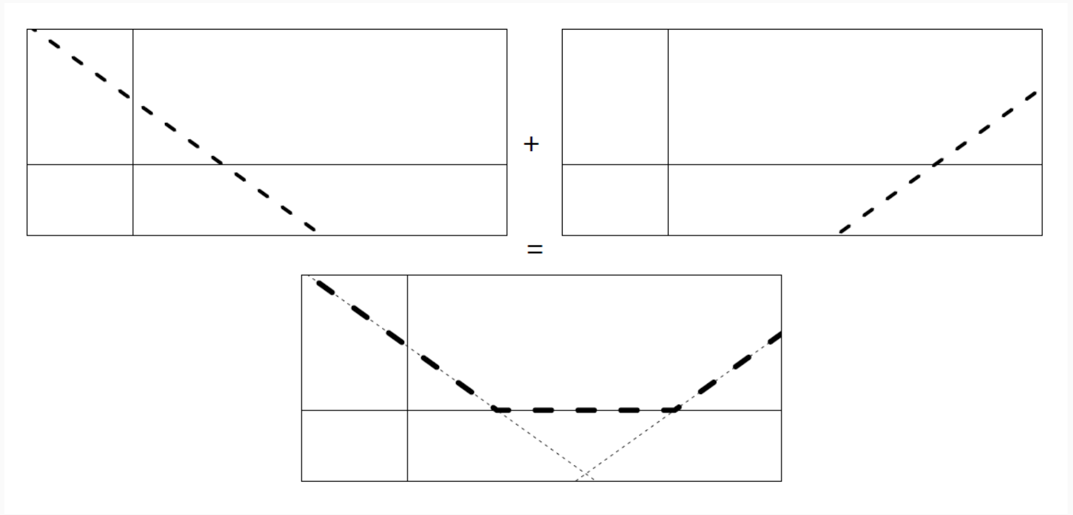
$$\begin{aligned} v &= b_3 + \sigma(z_1) \cdot u_1 + \sigma(z_2) \cdot u_2 \\ &= b_3 + \sigma(b_1 + x \cdot w_1) \cdot u_1 + \sigma(b_2 + x \cdot w_2) \cdot u_2, \end{aligned}$$

where $\sigma(.)$ 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!

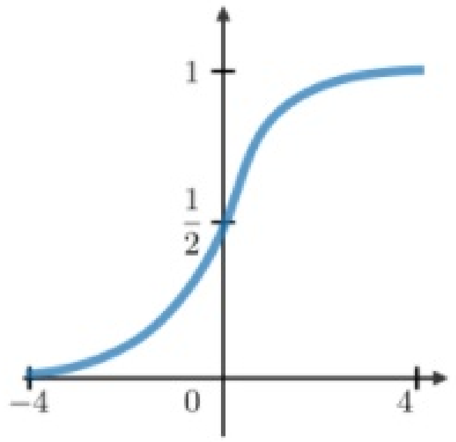
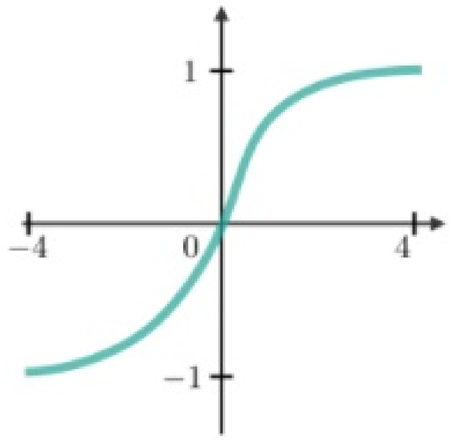
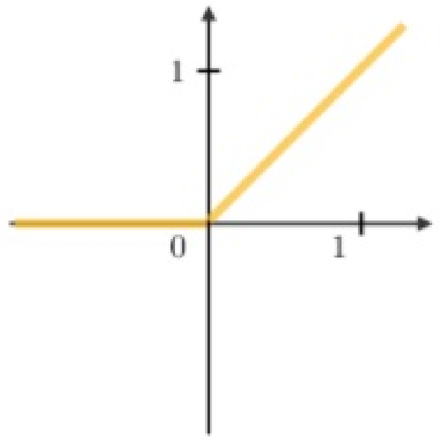
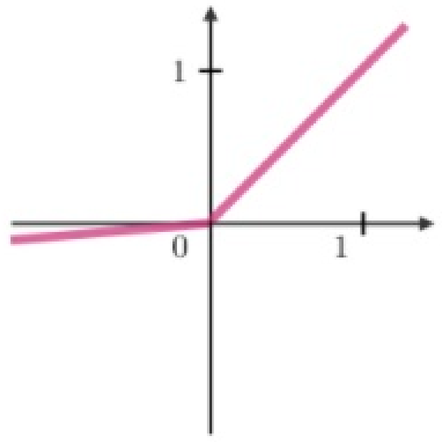
For example, the rectified linear unit (ReLU) activation function:

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



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

Examples of activation functions

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

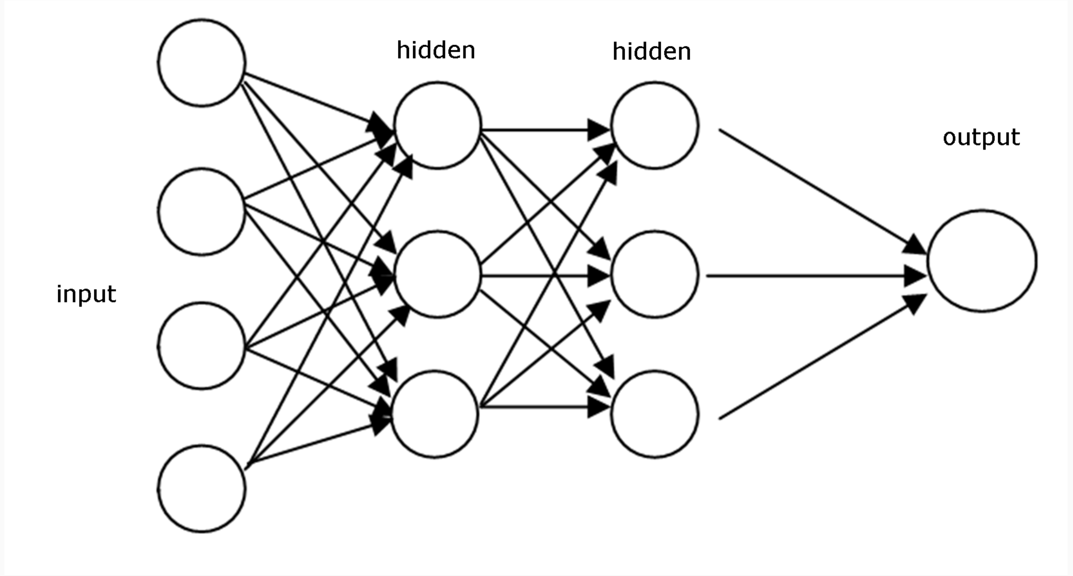
Source: https://stanford.edu/~shervine/teaching/cs-229/cheatsheet-deep-learning

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 $\boldsymbol{x}_i$ to first layer, output of final layer is $\hat{\boldsymbol{y}}_i$.

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.

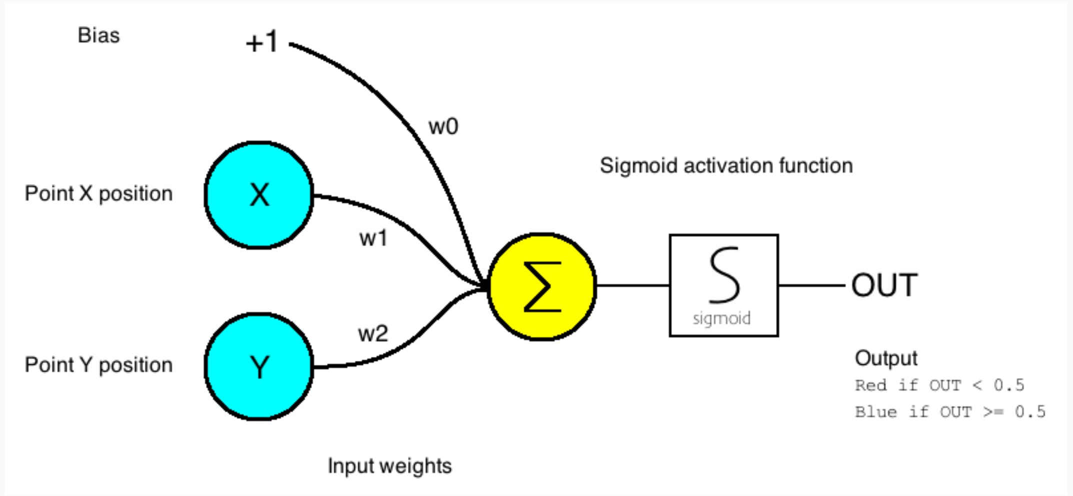


This is a feedforward neural network - no loops!

The neural nets' terminology

Using the neural nets terminology or language:

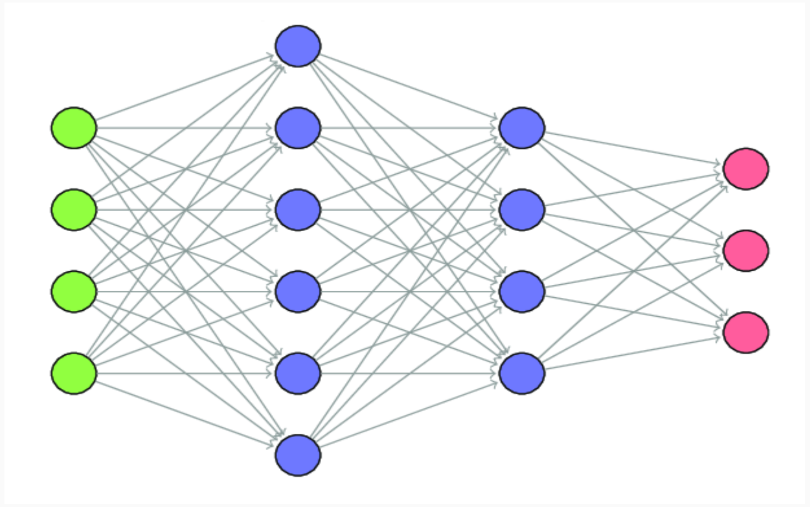
- intercept called **the bias**
- slopes called **weights**
- $L+1$ layers in total, with input layer denoted as layer 0 and output layer as L
- technically, **deep learning** refers to any neural network that has 2 or more hidden layers.



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 libary.

```
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.

An architecture with layers (cont.)

```
model ← keras_model_sequential() %>%
  layer_dense() %>% # hidden layer
  layer_dense() # output layer
```

Each `layer_dense()` represents a hidden layer *or* the final output layer.

```
model ← keras_model_sequential() %>%
  layer_dense() %>% # hidden layer 1
  layer_dense() %>% # hidden layer 2
  layer_dense() %>% # hidden layer 3
  layer_dense() # output layer
```

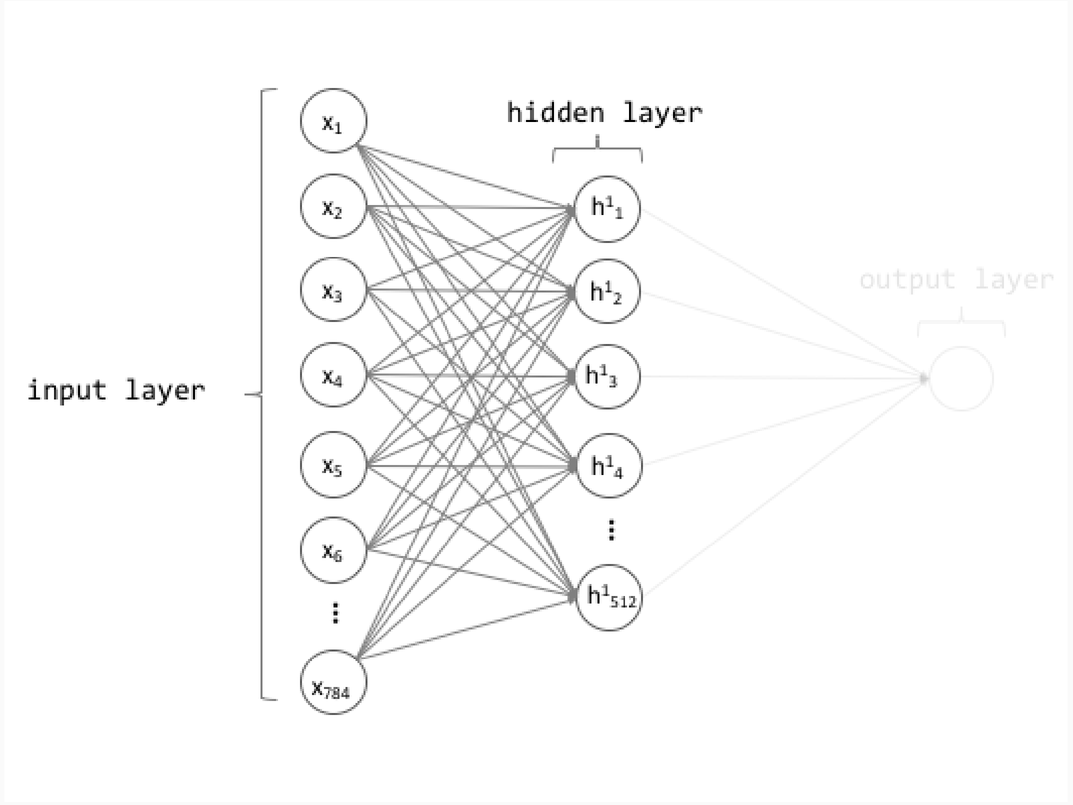
- We can add multiple hidden layers by adding more `layer_dense()` functions.
- The last `layer_dense()` will always represent the output layer.

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.



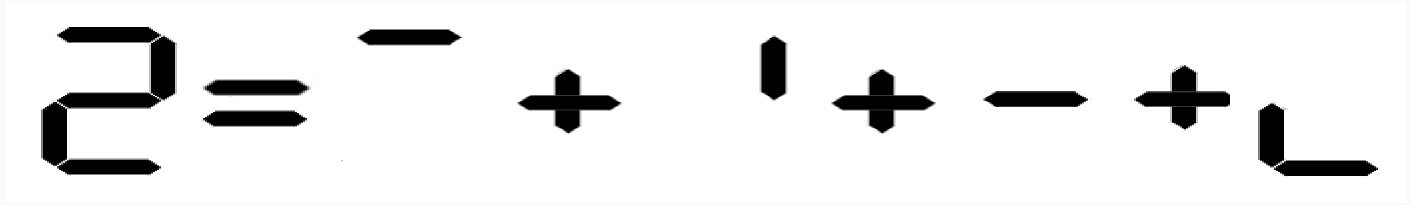
A hidden layer - some intuition

Nodes in the hidden layer(s) represent intermediary features that we do not explicitely define.

We let the model decide the optimal features.

For example, recognizing a digit is more difficult than recognizing a horizontal or vertical line.

Hidden layers automatically split the problem into smaller problems that are easier to model.



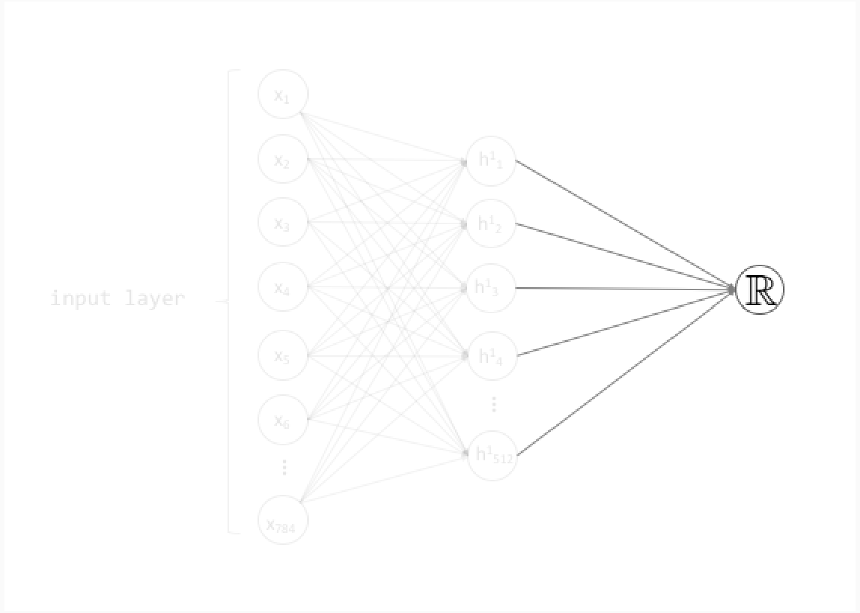
Output layer

```r
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`:



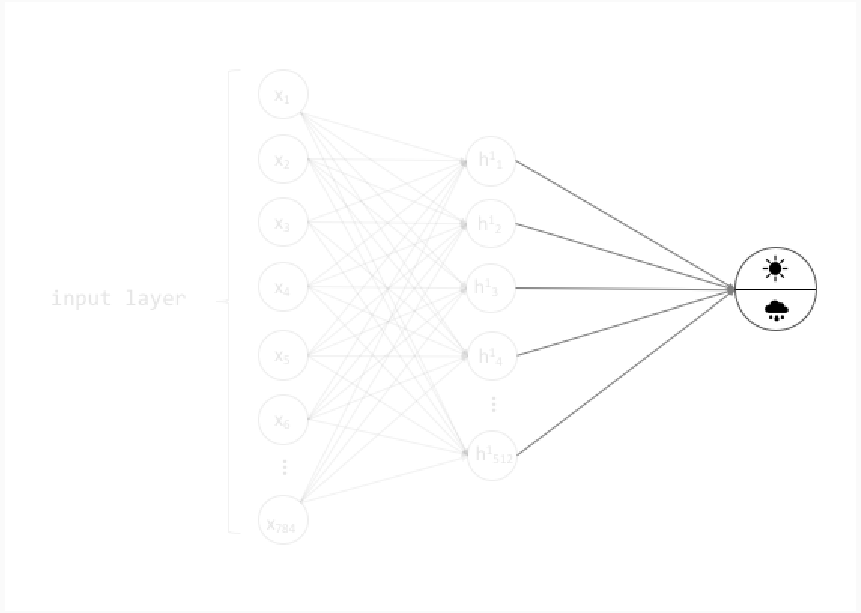
Output layer

```
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`



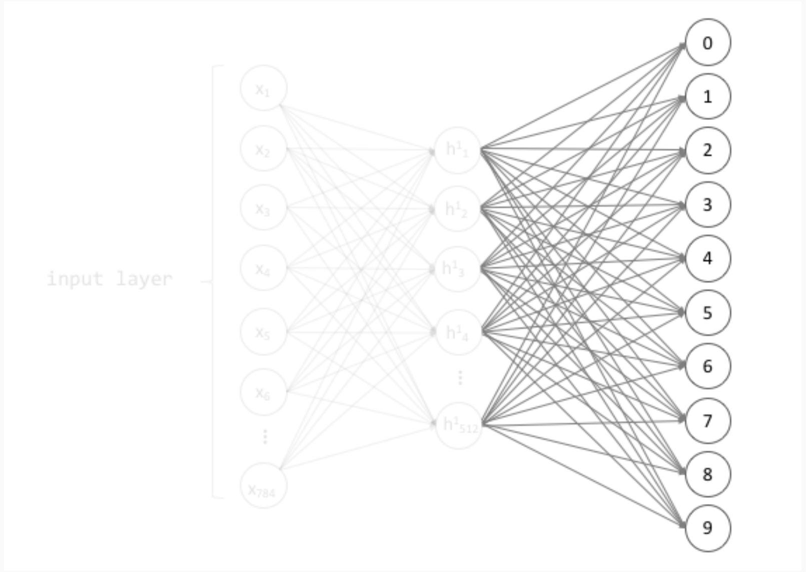
Output layer

```
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`



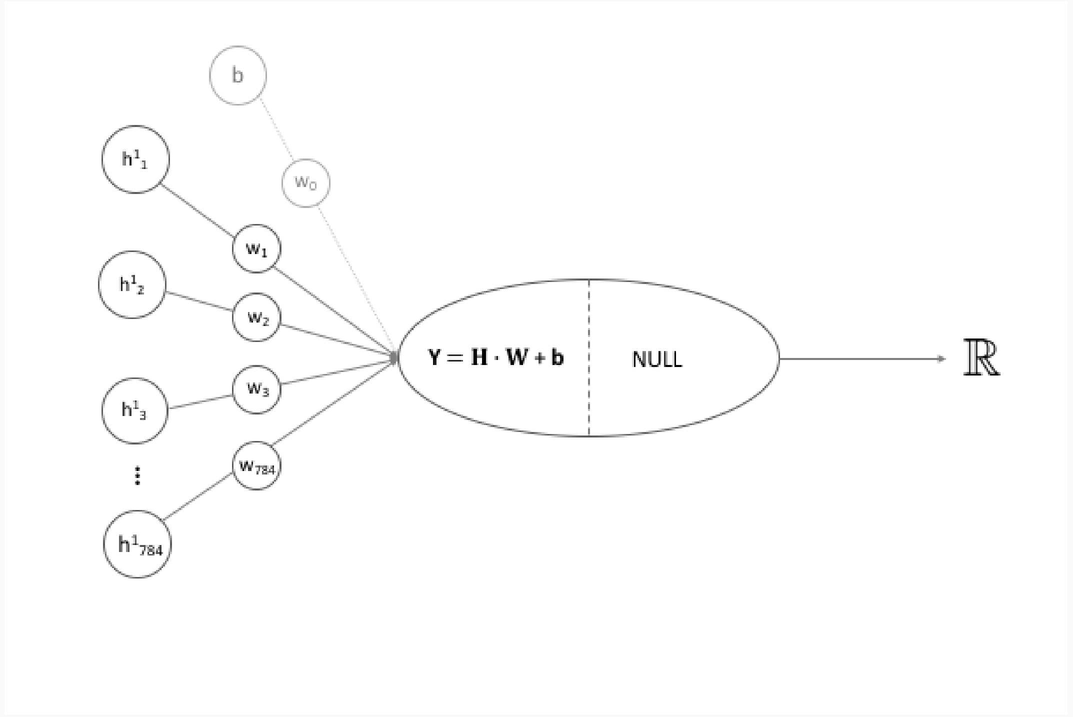
Output layer

```r
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)



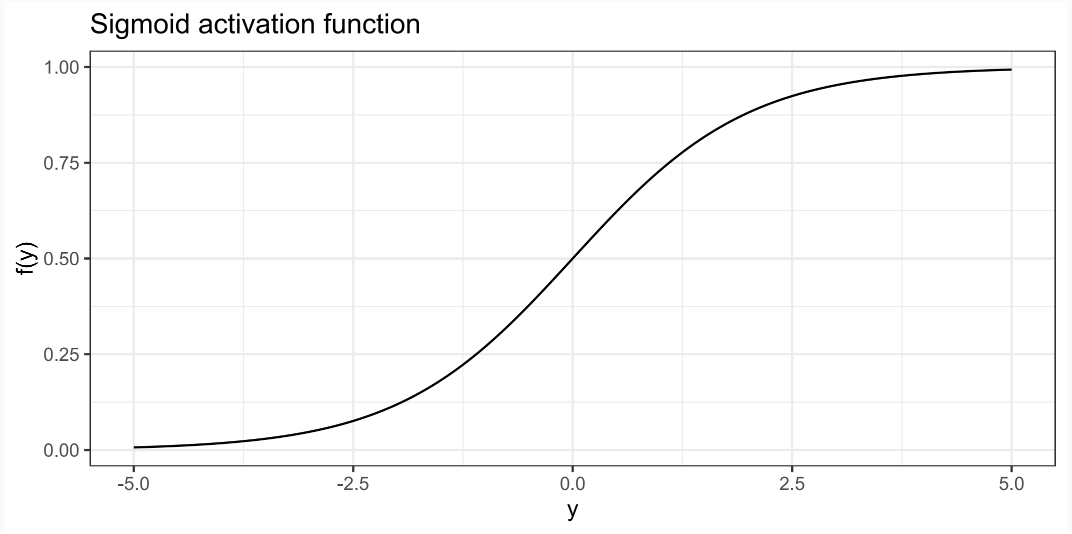
Output layer

```
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'`



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

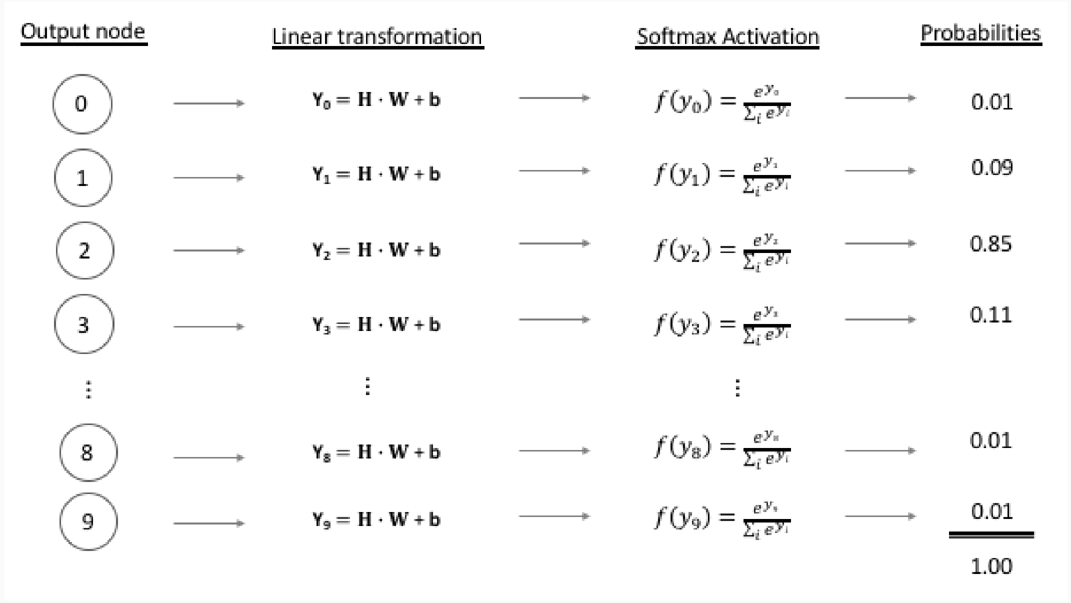
Output layer

```r
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'`



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

Your turn

Ultimately, here is a summary of the network architecture discussed so far

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

Can you figure out how many parameters will be trained for this network?

```
## Model: "sequential"
## ___________________________________________________
## Layer (type)                    Output Shape
## ===================================================
## dense_1 (Dense)                 (None, 512)
## ___________________________________________________
## dense (Dense)                   (None, 10)
## ===================================================
## Total params: 407,050
## Trainable params: 407,050
## Non-trainable params: 0
## ___________________________________________________
```

The model has 407,050 parameters:

- 784 inputs (28x28 pixels in a single image)
- 1 hidden layer, with
  - 512 nodes and ReLU activation
  - thus, (784 x 512) + 512 = 401,920 parameters
- multi-class output layer, with
  - 10 nodes
  - softmax activation function
  - thus, (512 x 10) + 10 = 5,130 parameters
- all together, that makes 407,050 parameters!

Network compilation

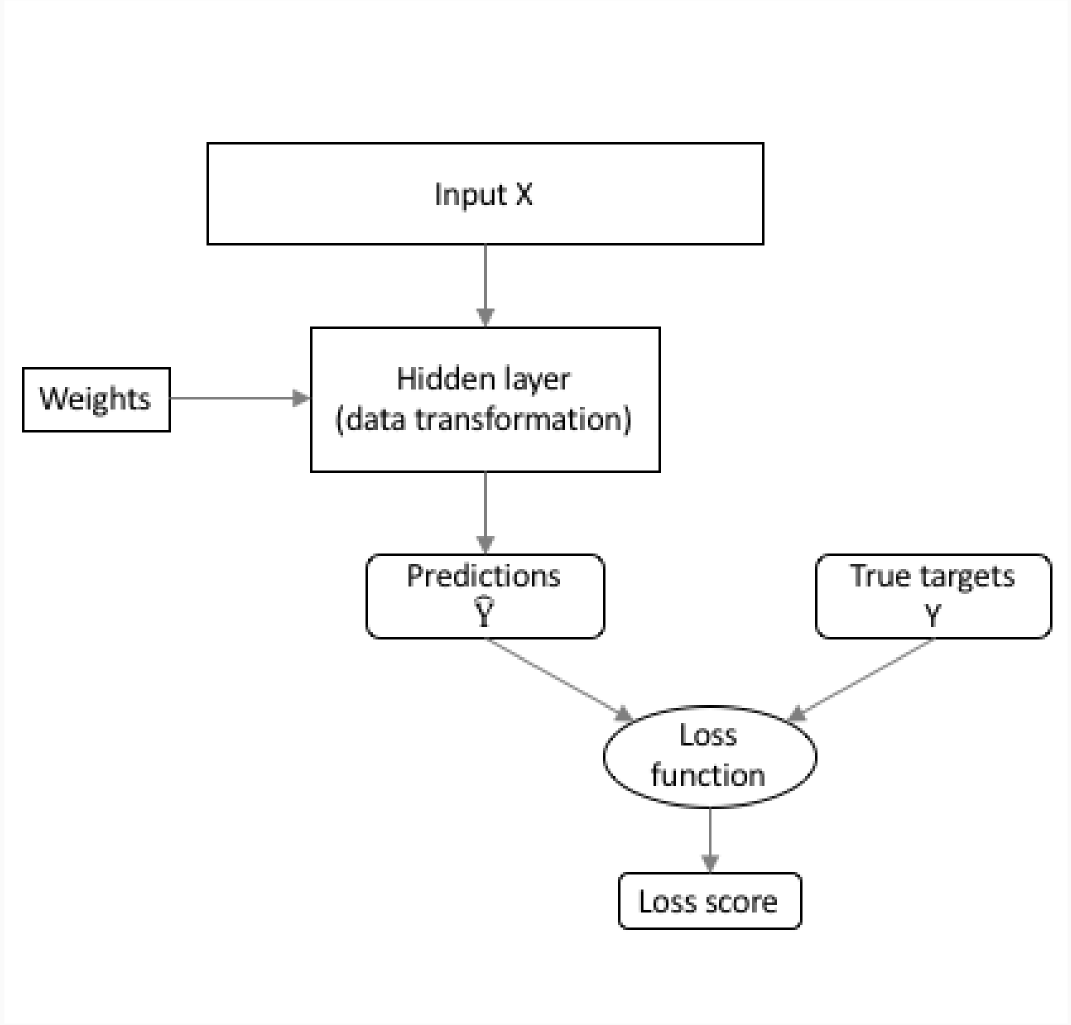
Loss function and forward pass

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

Keras includes many **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

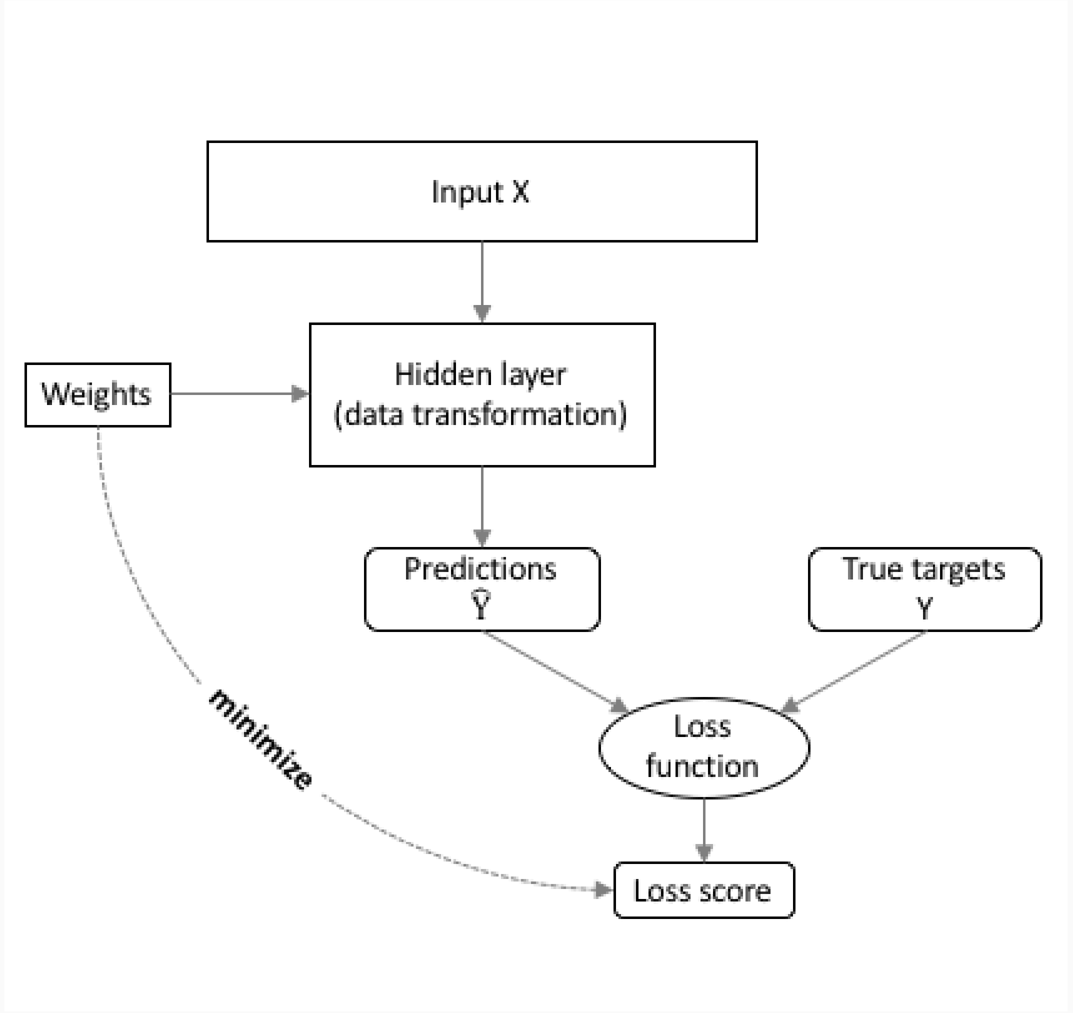
```
model ← model %>%
  compile(loss = "categorical_crossentropy",
          optimize = optimizer_rmsprop(),
          metrics = c('accuracy'))
```

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 η.

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 η.

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

We discussed so far:

- 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:

- the number of layers
- the number of nodes per layer
- the activation functions
- the layer type *(more on this would require more time)*
- 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:

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

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
β	fitted effect	weights

Your first claim frequency neural network

Let's start with a model with **only an intercept**:

$$Y \sim \texttt{Poisson}(\lambda = \exp(1 \cdot \beta)).$$

```r
nn_freq_intercept <-
  keras_model_sequential() %>%
  layer_dense(units = 1,
              activation = 'exponential',
              input_shape = c(1),
              use_bias = FALSE) %>%
  compile(loss = 'poisson',
          optimize = optimizer_rmsprop())
```

Q.: How many parameters does this model have?

- `layer_dense`: there are **no hidden layers**, the input layer is directly connected to the output layer.
- `units = 1`: there is **one** output node.
- `activation = 'exponential'`: we use an **exponential** inverse link function.
- `input_shape = c(1)`: there is **one** input node, i.e., the intercept which will be constant one.
- `use_bias = FALSE`: we don't need a **bias** term, since we explicitly include an input node equal to one.
- `loss = 'poisson'`: we maximize the **Poisson** likelihood, i.e., minimize the Poisson deviance.

Your first claim frequency neural network (cont.)

Create **vectors** for the input and output:

```
intercept ← rep(1, nrow(data_train))

counts ← data_train$nclaims
```

Fit the neural network:

```
nn_freq_intercept %>% fit(x = intercept,
                          y = counts,
                          epochs = 30,
                          batch_size = 1024,
                          validation_split = 0,
                          verbose = 0)
```

- `x = intercept`: use the intercept as **feature**.
- `y = counts`: use the claim counts as **target**.
- `epochs = 20`: perform 20 training **iterations** over the complete data.
- `batch_size = 1024`: use **batches** with 1024 observations to update weights.
- `validation_split = 0`: don't use a **validation** set, so all observations are used for training.
- `verbose = 0`: **silence** keras such that no output is generated during fitting.

Comparing our neural network with a GLM

We compare the results of our neural network with the same model specified as a GLM:

```r
glm_freq_intercept <- glm(nclaims ~ 1,
                          data = mtpl_train,
                          family = poisson(link = 'log'))

# GLM coefficients
glm_freq_intercept$coefficients
## (Intercept)
##   -2.084486

## NN weights
nn_freq_intercept$get_weights()
## [[1]]
##            [,1]
## [1,] -2.085138
```

There is a small difference in the parameter estimate, resulting from a **different optimization technique**.

Taking exposure into account in a neural network

The Poisson loss function, including **exposure**, is

$$\mathcal{L} = \sum_i \texttt{expo}_i \cdot \lambda_i - y_i \cdot \log(\texttt{expo}_i \cdot \lambda_i),$$

which is proportional to:

$$\mathcal{L} = \sum_i \texttt{expo}_i \cdot (\lambda_i - \frac{y_i}{\texttt{expo}_i} \log(\lambda_i)).$$

This is the loss function for a Poisson model with:

- observations $\frac{y_i}{\texttt{expo}_i}$ and
- weights $\texttt{expo}_i$.

Notice indeed how the parameter estimates of the following two GLMs are **identical**:

```
glm_offset <- glm(nclaims ~ ageph,
                  family = poisson(link = 'log'),
                  data = mtpl_train,
                  offset = log(expo))
glm_offset$coefficients
## (Intercept)       ageph
## -1.24456257 -0.01582612
```

```
glm_weights <- glm(nclaims / expo ~ ageph,
                   family = poisson(link = 'log'),
                   data = mtpl_train,
                   weights = expo)
glm_weights$coefficients
## (Intercept)       ageph
## -1.24456257 -0.01582612
```

Taking exposure into account in a neural net (cont.)

Nothing changes in our neural network model **architecture**:

```
nn_freq_exposure ←
  keras_model_sequential() %>%
  layer_dense(units = 1,
              activation = 'exponential',
              input_shape = c(1),
              use_bias = FALSE) %>%
  compile(loss = 'poisson',
          optimize = optimizer_rmsprop())
```

It is however **good practice** to always recompile.

Otherwise the neural network will pick up where it left off last time, with the optimal weights after fitting.

Create a **vector** with exposure values:

```
exposure ← data_train$expo
```

Divide claim counts by exposure and use weights:

```
nn_freq_exposure %>%
  fit(x = intercept,
      y = counts / exposure,
      sample_weight = exposure,
      epochs = 20,
      batch_size = 1024,
      validation_split = 0,
      verbose = 0)
```

Stay tuned to find out how to include exposure via an **offset** term!

Adding an input feature and a hidden layer

Let's start by adding **one feature**, namely `ageph`:

```
ageph ← data_train$ageph
```

Define the neural network **architecture** with a hidden layer:

```
nn_freq_ageph ←
  keras_model_sequential() %>%
  layer_batch_normalization(input_shape = c(1)) %>%
  layer_dense(units = 5,
              activation = 'tanh') %>%
  layer_dense(units = 1,
              activation = 'exponential',
              use_bias = TRUE) %>%
  compile(loss = 'poisson',
          optimize = optimizer_rmsprop())
```

- Pre-processing:

`layer_batch_normalization` **centers** and **scales** the input features (here only one) **per mini-batch**.

- Hidden layer:

`layer_dense` with five nodes and the `tanh` activation function.

- Output layer:

`layer_dense` with one node and the `exponential` activation function.

Notice how we set `use_bias = TRUE` for the **intercept.**

Adding an input feature and a hidden layer (cont.)

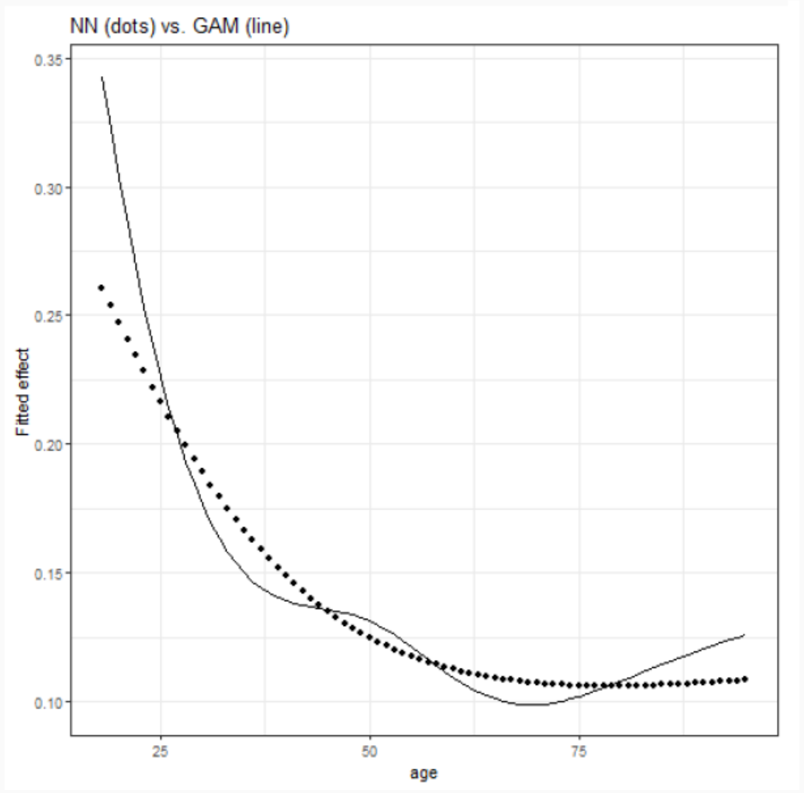
Let's **fit** our brand new neural net:

```r
nn_freq_ageph %>%
  fit(x = ageph,
      y = counts / exposure,
      sample_weight = exposure,
      epochs = 30,
      batch_size = 1024,
      validation_split = 0,
      verbose = 0)
```

We also fit a **GAM** with a smooth effect for `ageph`:

```r
library(mgcv)
gam_ageph ← gam(nclaims ~ s(ageph),
                 data = mtpl_train,
                 family = poisson(link = 'log'),
                 offset = log(expo))
```

Q.: What do you think about those fits?



Adding a skip connection in a neural network

So far, we stayed in a **purely sequential** architecture with `keras_model_sequential()`.

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

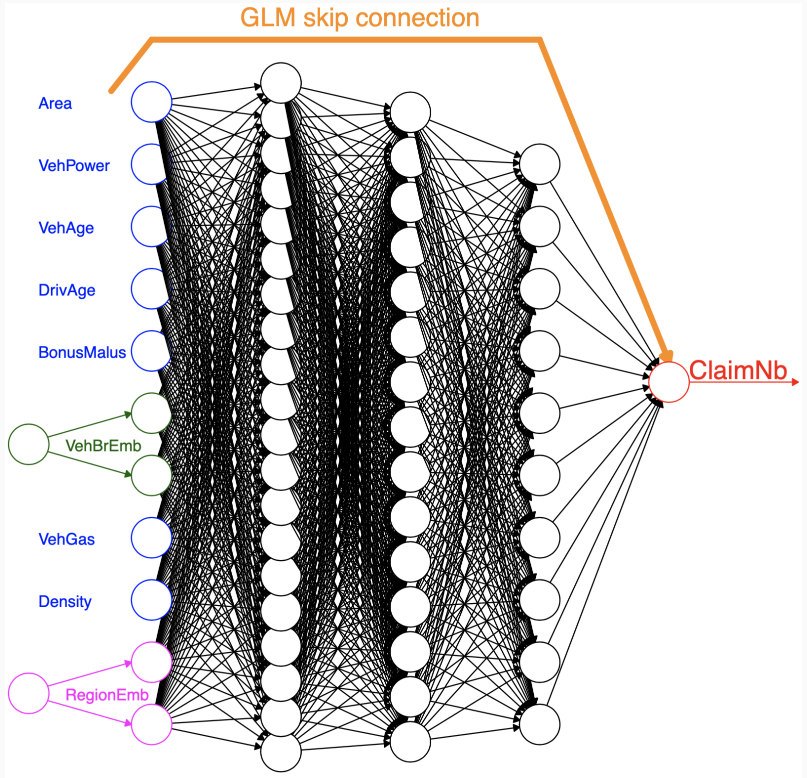


Figure taken from Schelldorfer and Wuthrich (2019).

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

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

With a skip connection, this simply becomes:

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

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:

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

- Adding a **base** prediction:

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

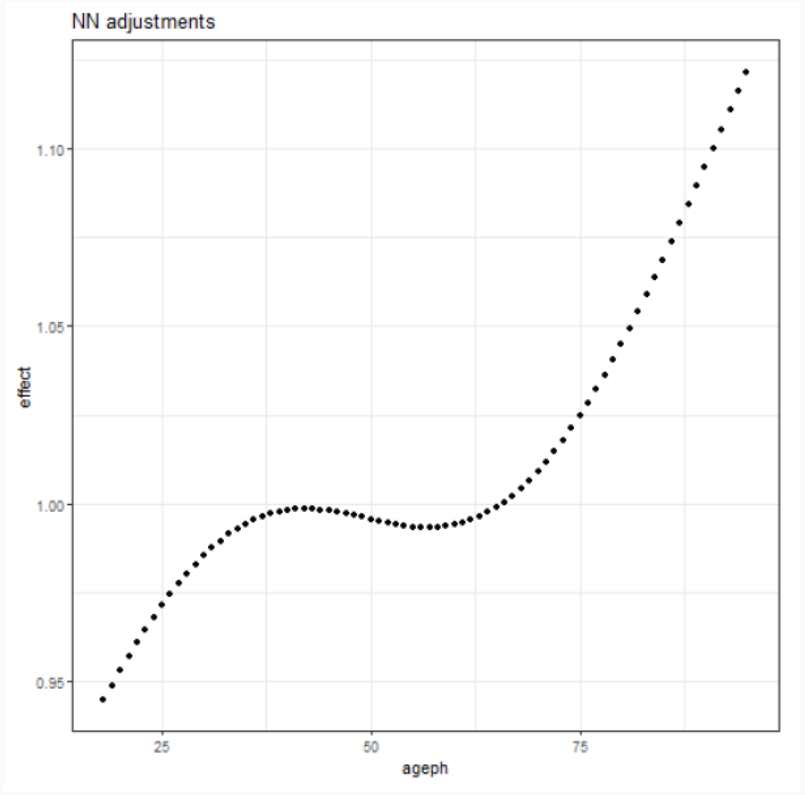
- The **combination** of both:

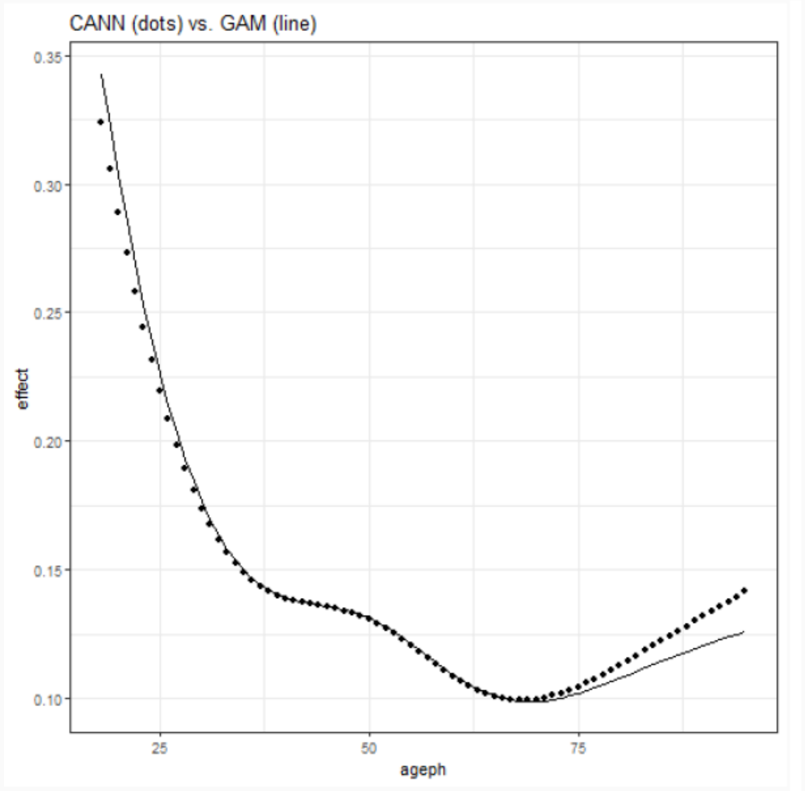
$$output = \exp(\sum_i w_i h_i + b + \log(expo \cdot base)) = expo \cdot base \cdot \exp(\sum_i w_i h_i + b).$$

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 Combined Actuarial Neural Network (**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 study with Roel, Freek and Simon

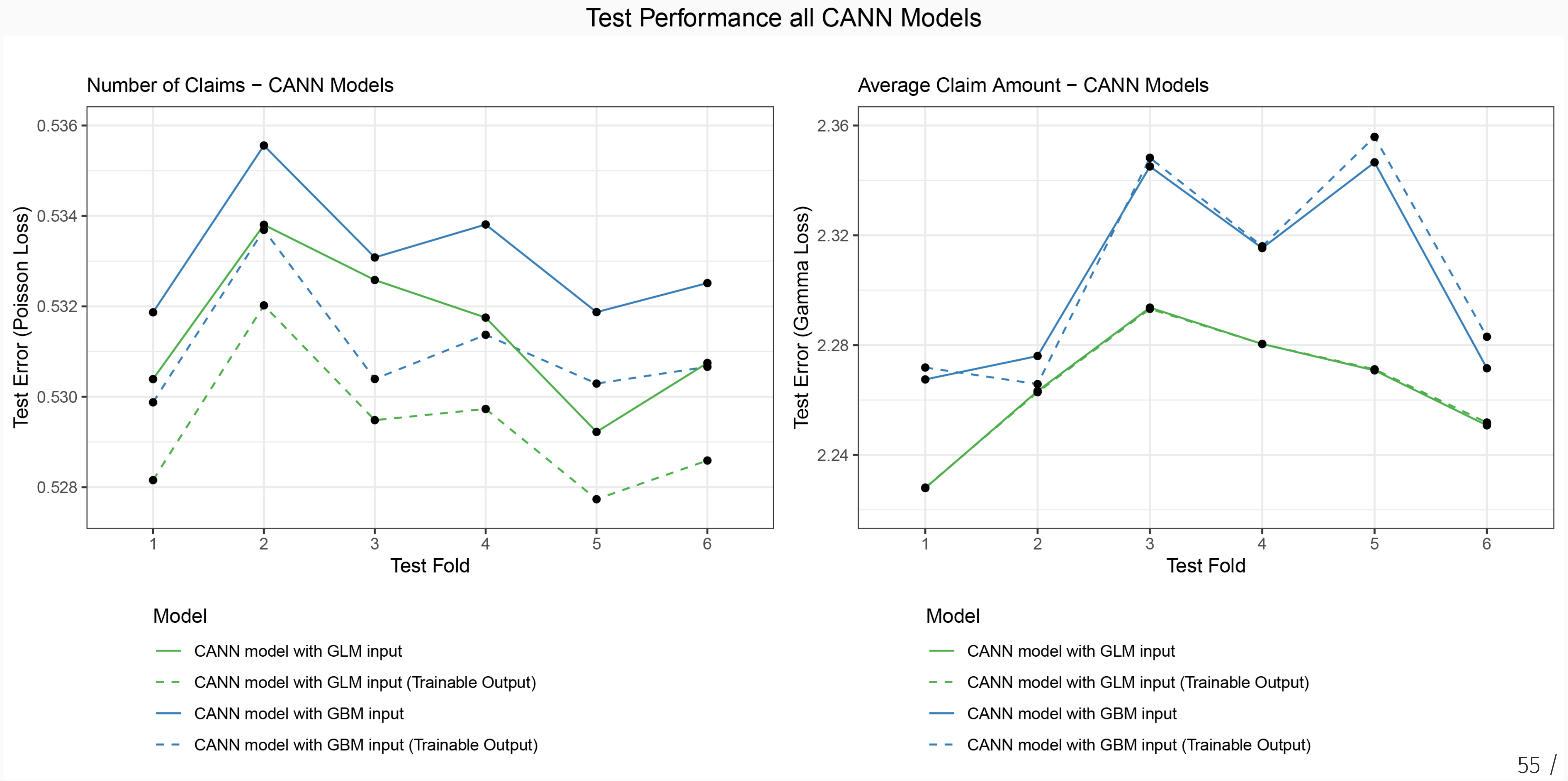
- ANNs and CANNs for both claim frequency and severity (seperately), and then their combination into a technical tariff
- CANNs with input from (smartly engineered) GLM and GBM, with
  - fixed input (say $\hat{\boldsymbol{y}}_i^{(in)}$) used via skip connection
  - input used via skip, but flexible (weights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

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

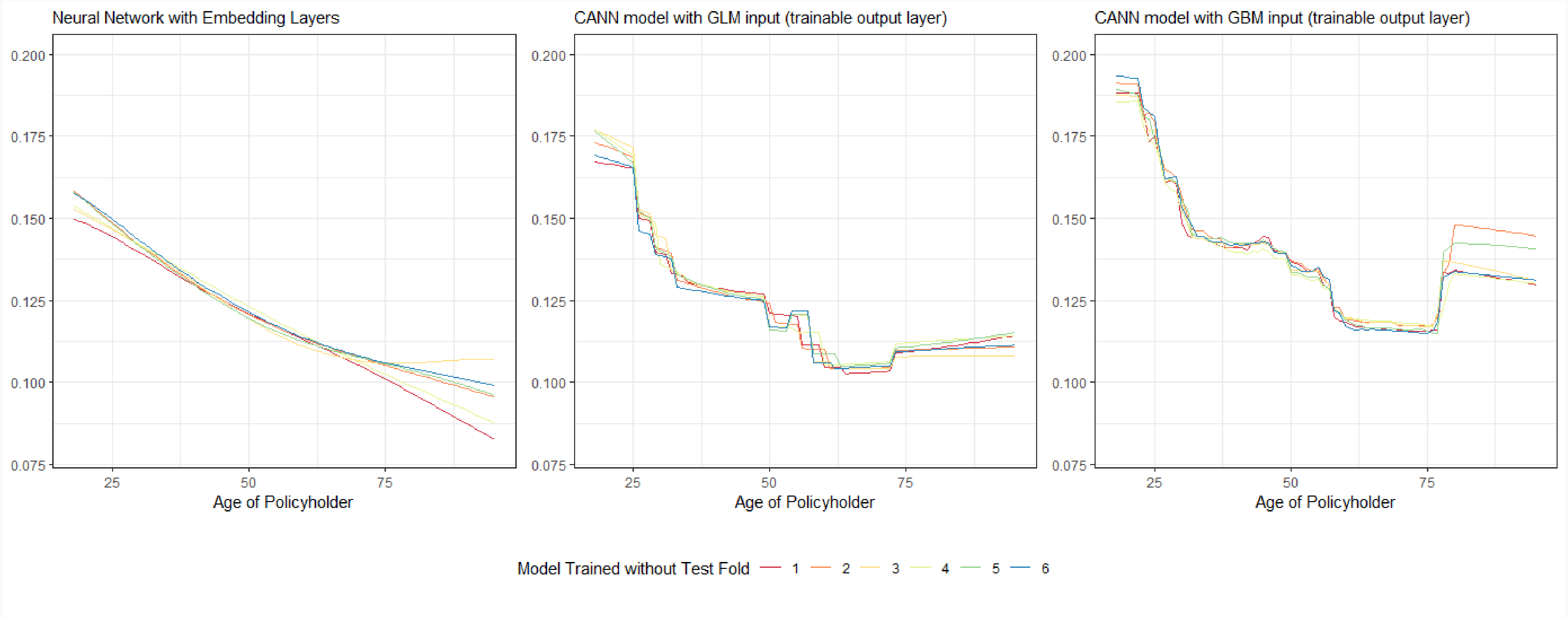
$$f^{flex}\left(\boldsymbol{x}_i, \hat{y}_i^{(in)}\right) = \exp\left(\begin{bmatrix} w_1 & w_2 \end{bmatrix} \cdot \begin{bmatrix} \ln(\hat{y}_i^{(in)}) & \hat{y}_i^{(adj)} \end{bmatrix}^t + b\right)$$

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

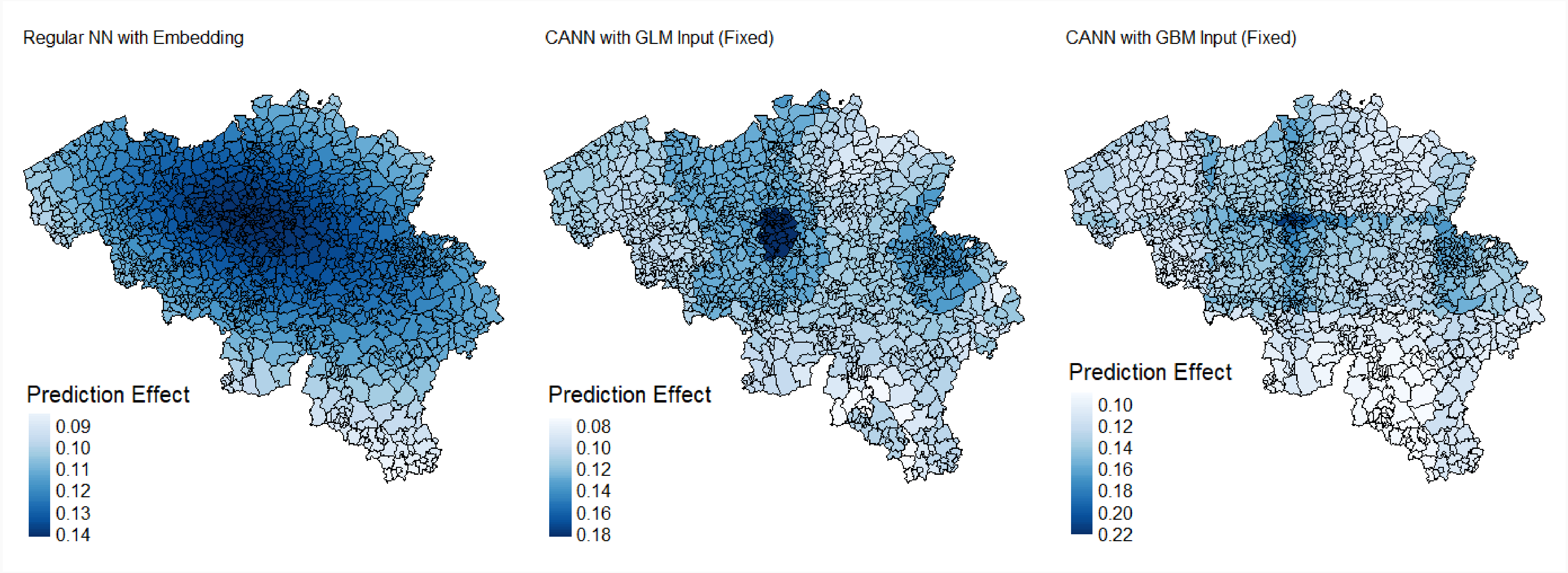
Some first results



Some first results (cont.)



Some first results (cont.)



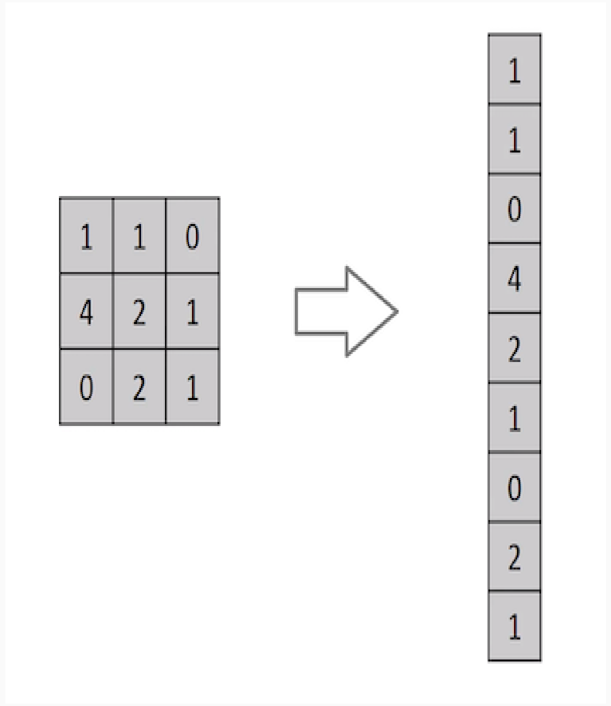
An outlook to convolutional neural networks (CNNs)

The problems with flattening

With ANNs, our first step when working with images was to **flatten** the image matrix into a vector.

This approach

- is not **translation** invariant. A completely different set of nodes gets activated when the image is shifted.
- ignores the **dependency** between nearby pixels.
- requires a **large number** of parameters/weights as each node in the first hidden layer is connected to all nodes in the input layer.



Source: Sumit Saha

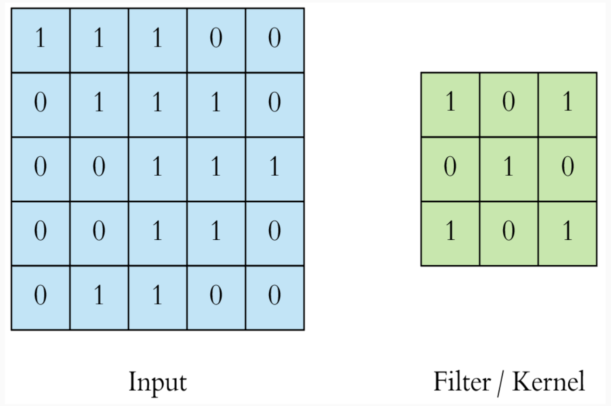
Convolutional layers allow to handle multi-dimensional data, **without** flattening.

Convolutional layers

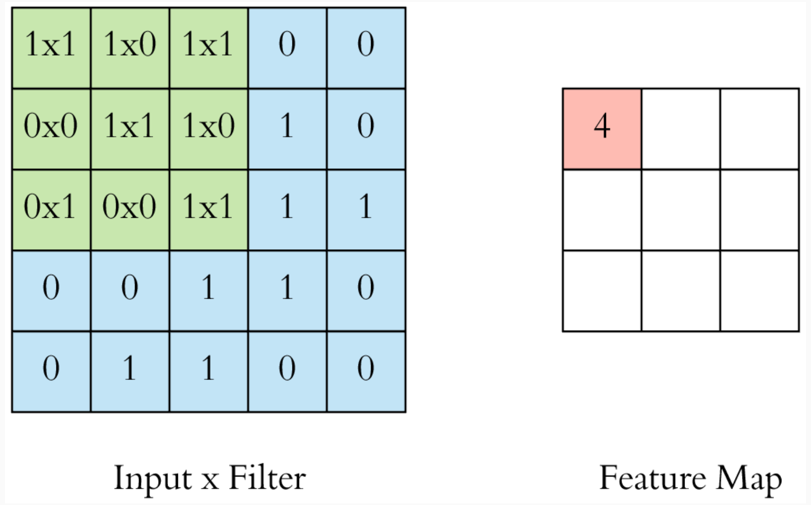
Classical hidden layers (as we have seen so far) use **1 dimensional inputs** to construct **1 dimensional features**.

2d convolutional layers use **2 dimensional input** (for example images) to construct **2 dimensional feature maps**.

The weights in a 2d convolutional layer are structured in a small image, called the **kernel** or the **filter**.



We slide the kernel over the input image, multiply the selected part of the image and the kernel elementwise and sum:



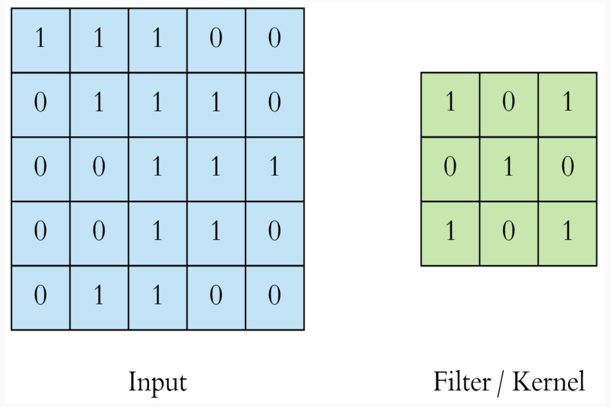
Source: Bradley Boehmke

Convolutional layers (cont.)

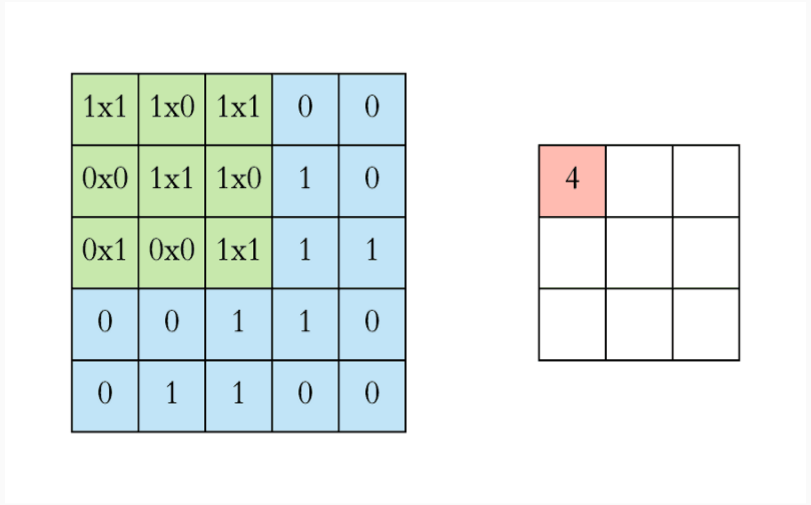
Classical hidden layers (as we have seen so far) use **1 dimensional inputs** to construct **1 dimensional features**.

2d convolutional layers use **2 dimensional input** (for example images) to construct **2 dimensional feature maps**.

The weights in a 2d convolutional layer are structured in a small image, called the **kernel** or the **filter**.



We slide the kernel over the input image, multiply the selected part of the image and the kernel elementwise and sum:



Source: Bradley Boehmke

Pooling layers

A convolution layer is typically followed by a **pooling step**, which reduces the size of the feature maps.

Pooling layers divide the image in blocks of equal size and then **aggregate** the data per block.

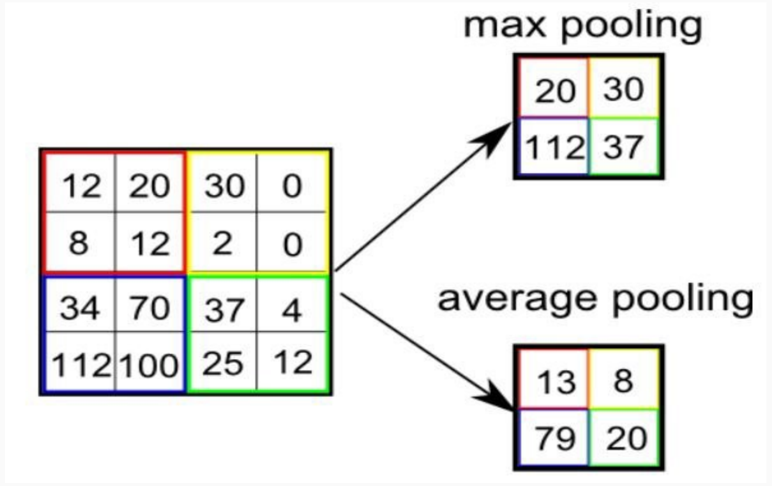
Two common operations are:

- average pooling

```
layer_average_pooling_2d(pool_size = c(2, 2),
                         strides = c(2, 2))
```

- max pooling

```
layer_max_pooling_2d(pool_size = c(2, 2),
                     strides = c(2, 2))
```



- `pool_size = c(2, 2)`:

  Pool blocks of 2x2

- `strides = c(2, 2)`:

  Move in steps of size 2 in both the horizontal and vertical direction.

Flattening layers

When all features are extracted, the data is flattened.

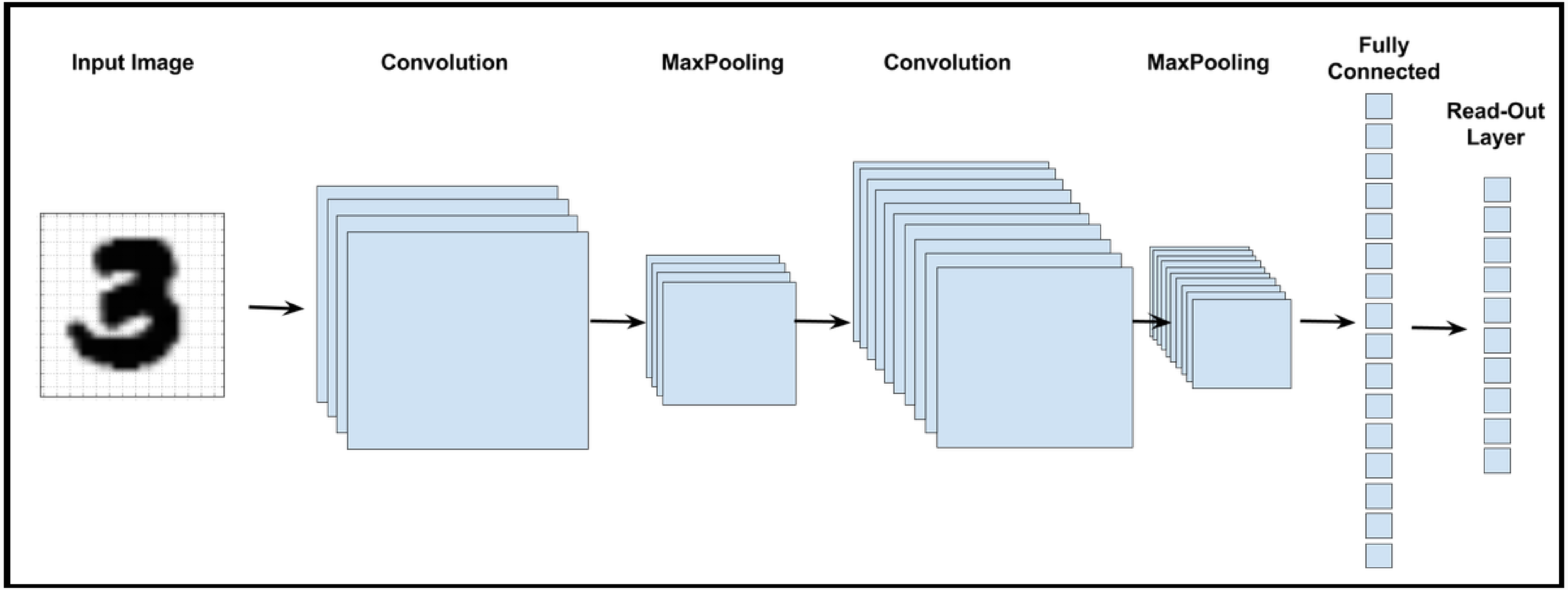
This data can be seen as **engineered features**, automatically created by the CNN architecture.

In a next step, a **feed-forward ANN** is used to analyze these local features.

```
keras_model_sequential() %>%
  layer_conv_2d() %>%
  layer_max_pooling_2d() %>%
  layer_flatten()
```

```
keras_model_sequential() %>%
  layer_conv_2d() %>%
  layer_max_pooling_2d() %>%
  layer_flatten() %>%
  layer_dense() %>%
  layer_dense() %>%
  compile()
```

A CNN architecture



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 accurary 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!

This research is supported by the Ageas - KU Leuven research chair on insurance analytics. The support of Ageas is gratefully acknowledged.

Slides created with the R package xaringan.

For more information please visit

https://github.com/katrienantonio

https://katrienantonio.github.io