

Hands-on Machine Learning with R - Module 3

Hands-on webinar

Katrien Antonio & Roel Henckaerts

[hands-on-machine-learning-R-module-3](#) | January 28 & February 4, 2021

Prologue

Introduction

Course

⌚ <https://github.com/katrienantonio/hands-on-machine-learning-R-module-3>

The course repo on GitHub, where you can find the data sets, lecture sheets, R scripts and R markdown files.

Us

🔗 <https://katrienantonio.github.io/> & <https://henckr.github.io/>

✍ katrien.antonio@kuleuven.be & roel.henckaerts@kuleuven.be

🎓 (Katrien) Professor in insurance data science

🎓 (Roel) PhD student in insurance data science

Checklist

- Do you have a fairly recent version of R?

```
version$version.string  
## [1] "R version 4.0.3 (2020-10-10)"
```

- Do you have a fairly recent version of RStudio?

```
RStudio.Version()$version  
## Requires an interactive session but should return something like "[1] '1.3.1093'"
```

- Have you installed the R packages listed in the software requirements?

or

- Have you created an account on RStudio Cloud (to avoid any local installation issues)?

Why this course?

The goals of this module

- **de-mystify** neural networks
- 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 **auto encoders** and **convolutional** networks.

Want to read more?

This presentation is based on

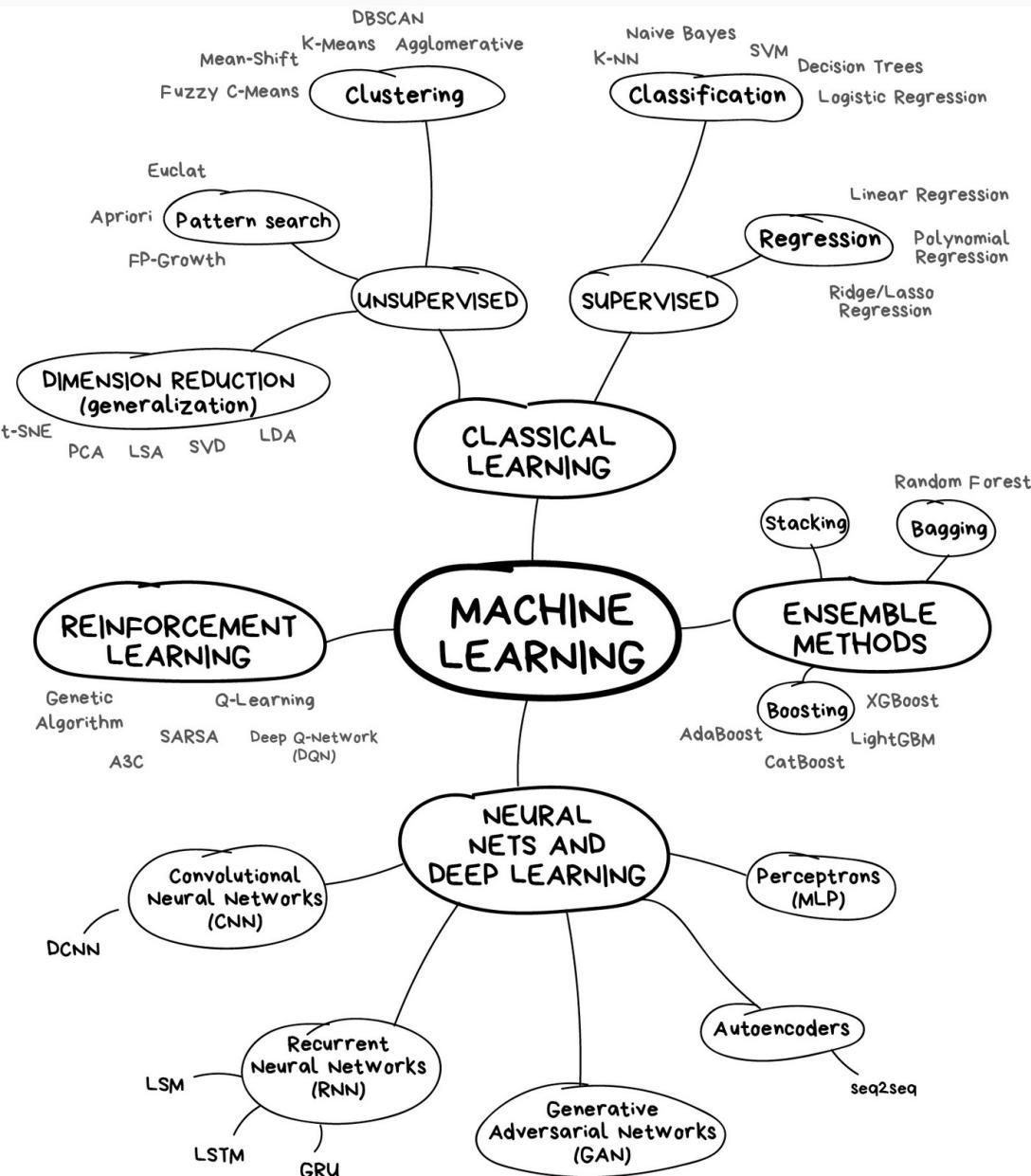
- Michael A. Nielsen (2015) [Neural networks and deep learning](#)
- the work of prof. Taylor Arnold, in particular Chapter 8 in the book [A computational approach to statistical learning](#) by Arnold, Kane & Lewis (2019)
- Boehmke (2020) on [Deep Learning with R: using Keras with Tensorflow backend.](#)

Actuarial modelling with neural nets is covered in

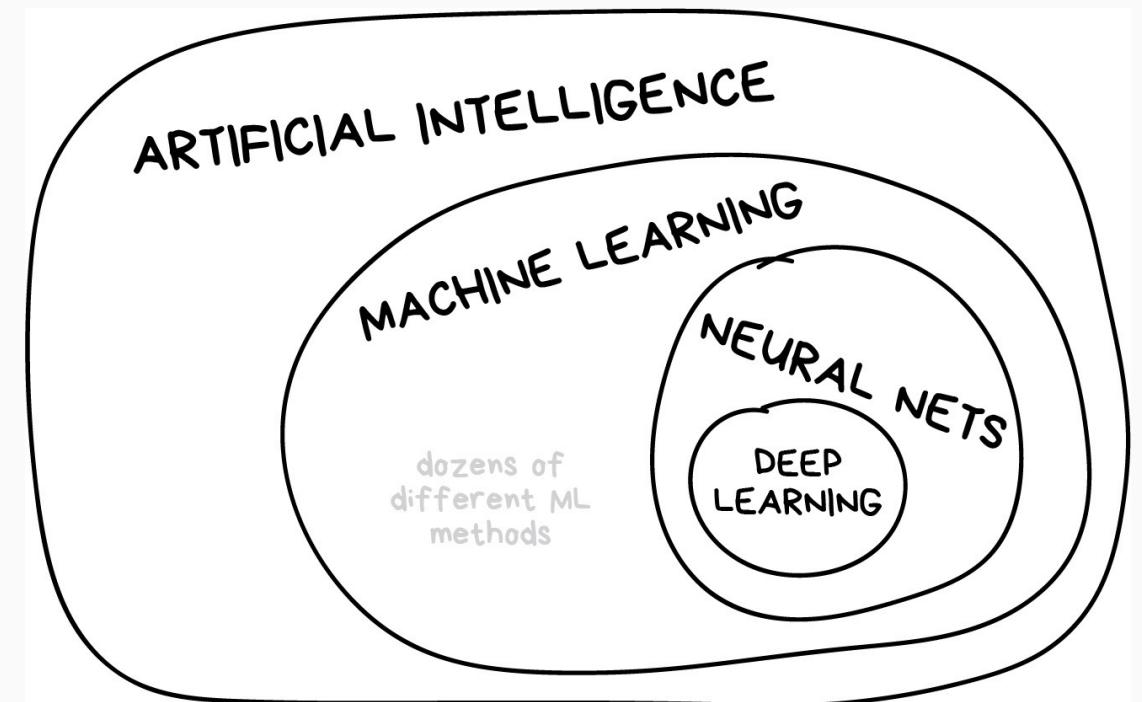
- Wüthrich & Buser (2020) [Data analytics for non-life insurance pricing](#), in particular Chapter 5
- Wüthrich (2019) [From Generalized Linear Models to neural networks, and back](#)
- Wüthrich & Merz (2019) [Editorial: Yes, we CANN!](#), in ASTIN Bulletin 49/1.

Module 3's 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
 - Performance metrics
 - Model evaluation
- Regression with neural networks
 - Redefining GLMs as a neural network
 - Including exposure
 - Case study
- Auto encoders
 - Data compression and feature extraction
 - Evaluation
- Convolutional neural networks
 - Handling new data formats
 - Convolutional layers explained
 - Evaluation and interpretation



Some roadmaps to explore the ML landscape...



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

Getting started

The programming framework for today



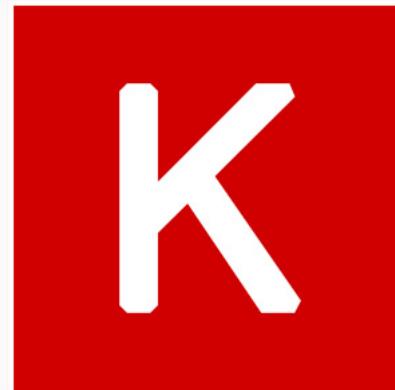
- R:
Our favorite programming language, including an R interface to Keras and TensorFlow.
- Keras:
An intuitive high level Python interface to TensorFlow.
- TensorFlow:
Open source platform for machine learning developed by the Google Brain Team, see <https://www.tensorflow.org/>.
Special focus on training deep neural networks.

R packages

Today's session will make extensive use of {keras} and {tidyverse}.

Do not forget to load these packages in your R session.

```
library(keras)  
library(tidyverse)
```



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
- one sound fragment: 2-dimensional, with 2 axes
- one image in color: 3-dimensional, with 3 axes
- one 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.)

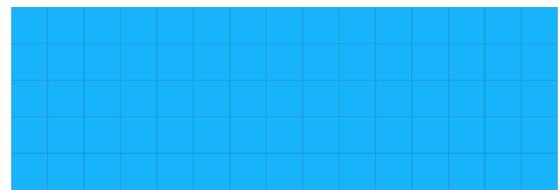
TensorFlow is this flexible framework which consists of highly optimized functions based on **tensors**.

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.

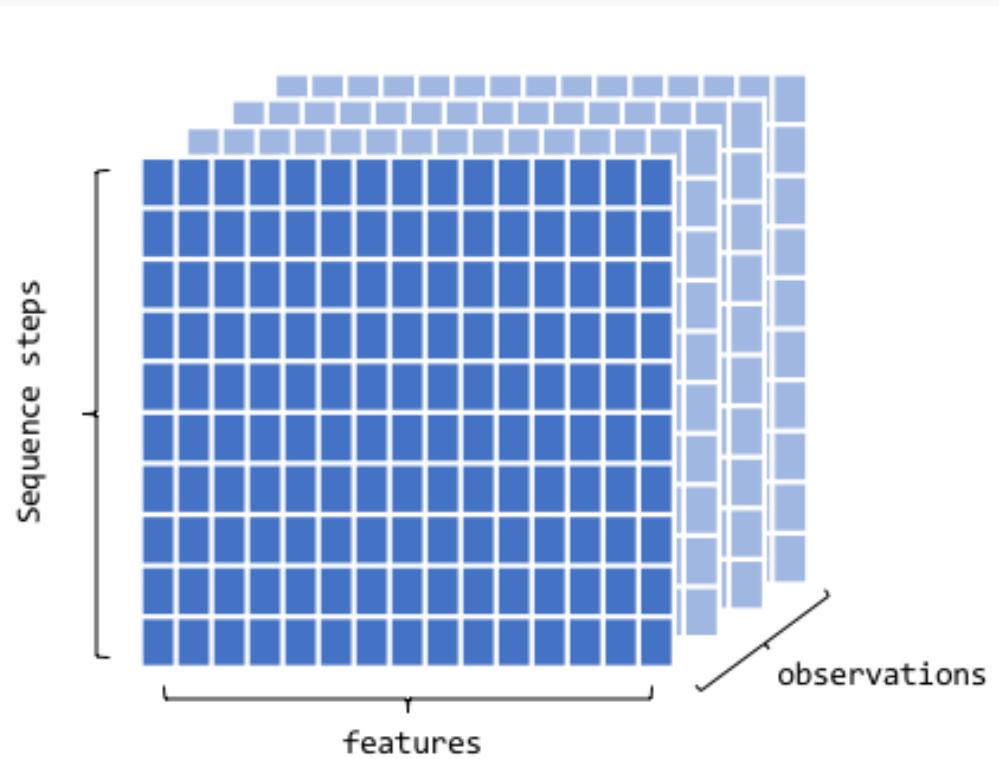
Luckily Keras provides a high level interface to TensorFlow, such that we will have only minimal exposure to tensors and the complicated math behind them.

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: $(250, 390, 3)$.



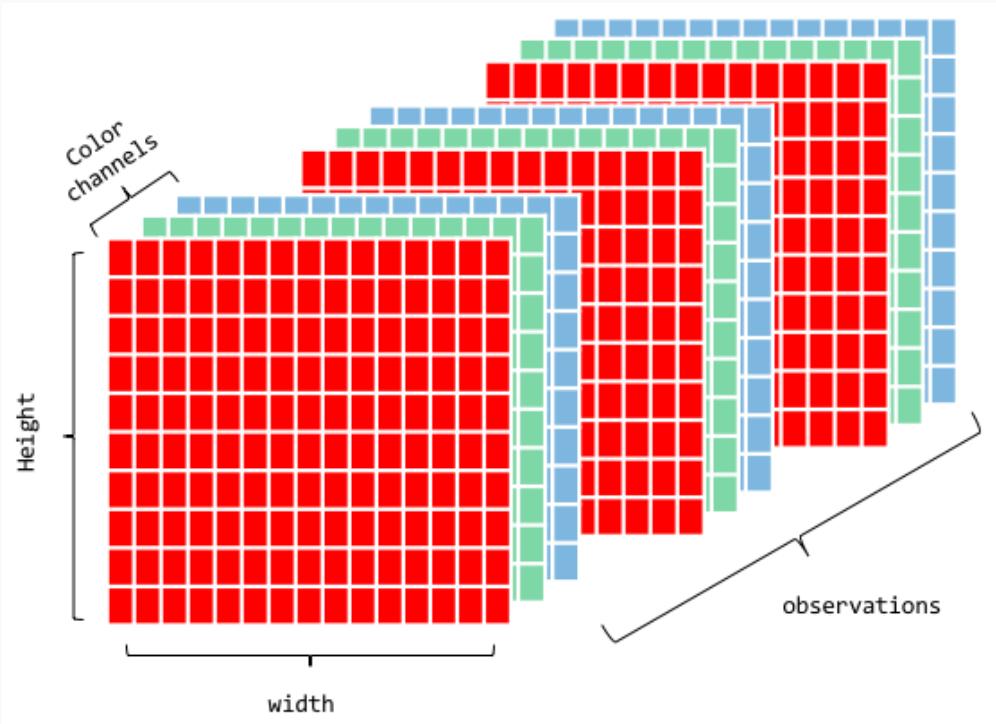
Source: Bradley Boehmke

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).



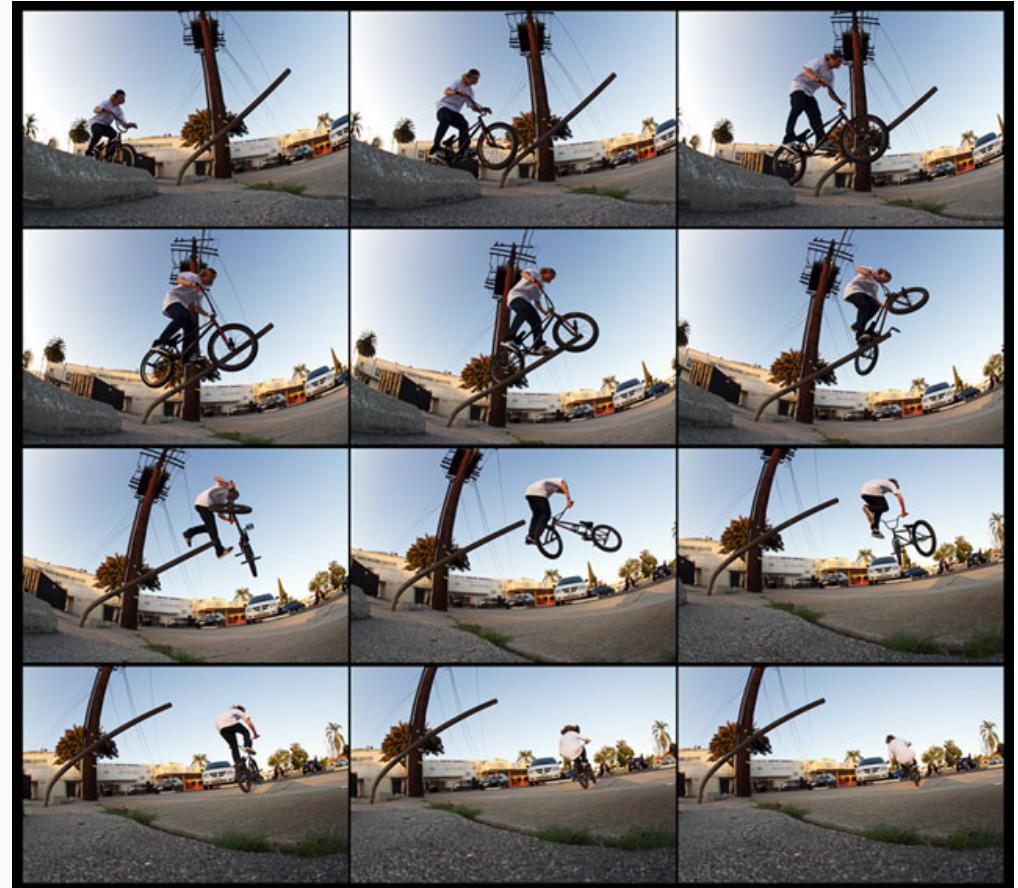
Source: Bradley Boehmke

Example of a 5D tensor

Let's picture an 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 4D tensor
(samples, frames, height, width, channels) which becomes here (samples, 240, 256, 144, 3).



Source: Bradley Boehmke

Tensor functions

Keras generalizes common R functions for inputs of type tensor. These functions can be recognized by the prefix `k_`.

See the [Keras documentation](#) for a list of all tensor functions.

- `k_constant`: create and initialise a tensor.

```
x ← k_constant(c(1, 2, 3, 4, 5, 6),
                 shape = c(3, 2))
x
## tf.Tensor(
## [[1. 2.]
## [3. 4.]
## [5. 6.]], shape=(3, 2), dtype=float32)
```

Most tensor operations require an axis parameter to specify the dimensions over which the function should be performed.

- `k_mean`: calculate the mean of the tensor.

```
k_mean(x, axis = 1)
## tf.Tensor([3. 4.], shape=(2,), dtype=float32)
```

```
k_mean(x, axis = 2)
## tf.Tensor([1.5 3.5 5.5], shape=(3,), dtype=float32)
```



In this warming up exercise you **create a tensor** and **apply basic tensor functions**.

Your turn

- **Q.1:** create a 3-dimensional tensor in R with values `1, 2, ... , 12` and shape `(2, 3, 2)`.
- **Q.2:** calculate the logarithm of this tensor.
- **Q.3:** calculate the mean of this tensor over the third axis.

Q.1: create a tensor

```
x ← k_constant(1:12, shape = c(2, 3, 2))
x
## tf.Tensor(
## [[[ 1.  2.]
##   [ 3.  4.]
##   [ 5.  6.]]
##
##  [[ 7.  8.]
##   [ 9. 10.]
##   [11. 12.]]], shape=(2, 3, 2), dtype=float32)
```

Q.2: calculate the logarithm

```
k_log(x)
## tf.Tensor(
## [[[0.          0.6931472]
##   [1.0986123 1.3862944]
##   [1.609438  1.7917595]]
##
##  [[1.9459102 2.0794415]
##   [2.1972246 2.3025851]
##   [2.3978953 2.4849067]]], shape=(2, 3, 2), dtype=fl
```

`log(x)` would have also resulted in the correct answer. However, it is best practice to use `k_log`, since the R-function `log` can not be evaluated within TensorFlow:

```
library(tensorflow)
tf$`function`(k_log)(x)
```

```
## tf.Tensor(
## [[[0.          0.6931472]
##   [1.0986123 1.3862944]
##   [1.609438  1.7917595]]
##
##  [[1.9459102 2.0794415]
##   [2.1972246 2.3025851]
##   [2.3978953 2.4849067]]], shape=(2, 3, 2), dtype=fl
```

```
tf$`function`(log)(x)
```

```
## Error in py_call_impl(...):
##   Unable to convert R object to Python type
```

Q.3: calculate the mean along the third axis

```
k_mean(x, axis = 3)
## tf.Tensor(
## [[ 1.5  3.5  5.5]
##  [ 7.5  9.5 11.5]], shape=(2, 3), dtype=float32)
```

... and explore the other axes as well

Q.3: mean along the first axis

```
k_mean(x, axis = 1)
## tf.Tensor(
## [[4. 5.]
##  [6. 7.]
##  [8. 9.]], shape=(3, 2), dtype=float32)
```

or the second axis

```
k_mean(x, axis = 2)
## tf.Tensor(
## [[ 3.  4.]
##  [ 9. 10.]], shape=(2, 2), dtype=float32)
```

Data sets used in the course

Data sets used in this course - MTPL



We will (once again) use the Motor Third Party Liability data set. There are 163,231 policyholders in this data set.

The frequency of claiming (`nclaims`) and corresponding severity (`avg`, the amount paid on average per claim reported by a policyholder) are the **target variables** in this data set.

Predictor variables are:

- the exposure-to-risk, the duration of the insurance coverage (max. 1 year)
- factor variables, e.g. gender, coverage, fuel
- continuous, numeric variables, e.g. age of the policyholder, age of the car
- spatial information: postal code (in Belgium) of the municipality where the policyholder resides.

More details in [Henckaerts et al. \(2018, Scandinavian Actuarial Journal\)](#) and [Henckaerts et al. \(2019, arxiv\)](#).



Data sets used in this course - MNIST

As discussed, not all data are in tabular format.

We analyze an **image database** from the Modified National Institute of Standards and Technology, short **MNIST**.

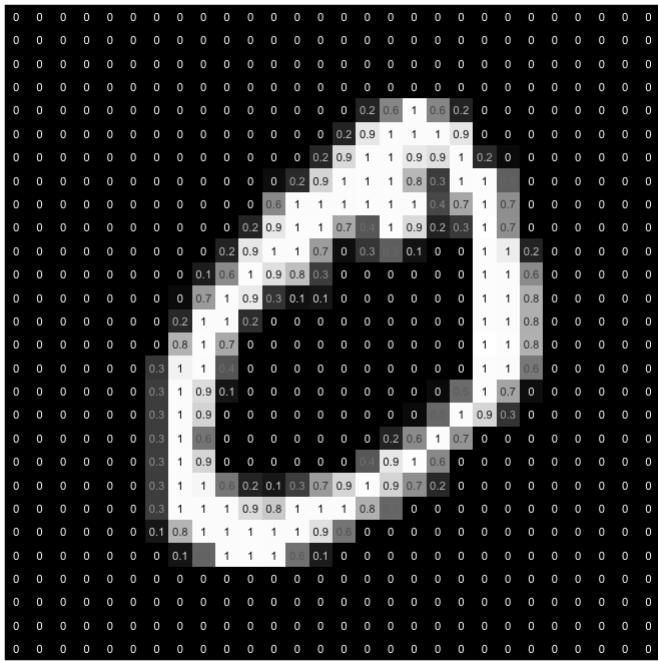
Working with MNIST will teach us how machine learning methods can be used to work with new data sources, such as images.

- Large database of 70,000 labeled images of handwritten digits, see
<http://yann.lecun.com/exdb/mnist/>
- Images are preprocessed, i.e. scaled and centered.
- Classic test case for machine learning classification algorithms. Current models achieve an accuracy of more than 99.5%.

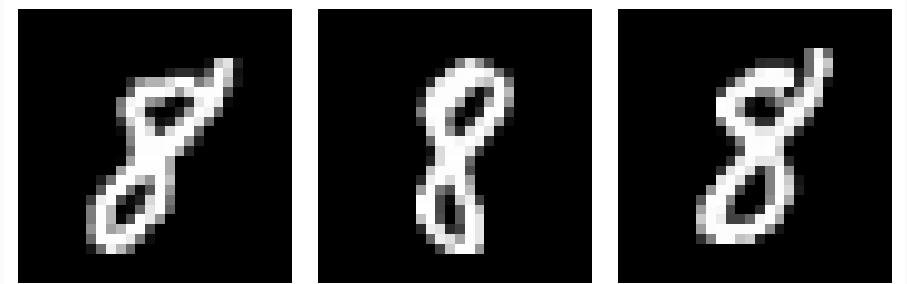
HANDWRITING SAMPLE FORM			
NAME	DATE	CITY	STATE ZIP
[REDACTED]	8/23/89	Leominster, MA 01453	
<small>This sample of handwriting is being collected for use in testing computer recognition of hand printed numbers and letters. Please print the following characters in the boxes that appear below:</small>			
0123456789	0123456789	0123456789	0123456789
0123456789	0123456789	0123456789	0123456789
07 508	4188	13183	793094
407	4298	72478	931465
2567	87516	492935	36 600
25649	274951	02	236 1838
035006	16	953	9458 6717
<small>zhbergfladijwnfkxsymipouvcg</small>			
<small>WPZBKIJFGRONCXQLDUEASHYNVT</small>			
<small>We, the People of the United States, in Order to form a more perfect Union, establish Justice, insure domestic Tranquility, provide for the common Defense, promote the general Welfare, and secure the Blessings of Liberty to ourselves and our posterity, do ordain and establish this CONSTITUTION for the United States of America.</small>			

Data sets used in this course - MNIST

The images are in grayscale. Each image is stored as a 28x28 intensity matrix, with intensity expressed on a scale from 0-255.



Recognizing that the images below all represent the digit 8 is trivial for humans, but difficult for computers.



Neural networks are ideal for situations where the relation between the input (here: intensity matrix) and the output (here: 0-9) is complicated.

Loading the MNIST dataset

`keras::dataset_mnist()` retrieves the MNIST dataset from the online repository.

Alternatively, the dataset can be loaded from the course directory.

```
# download the dataset from the online repository
mnist ← keras::dataset_mnist()
# load the dataset from the course files
load('..../data/mnist.RData')
```

Assign new names to the input and output data.

```
input ← mnist$train$x
output ← mnist$train$y
test_input ← mnist$test$x
test_output ← mnist$test$y
```

The input data is an `array`:

```
class(input)
```

```
## [1] "array"
```

with 60,000 28x28 images (the training data):

```
dim(input)
```

```
## [1] 60000    28    28
```

Select the first image:

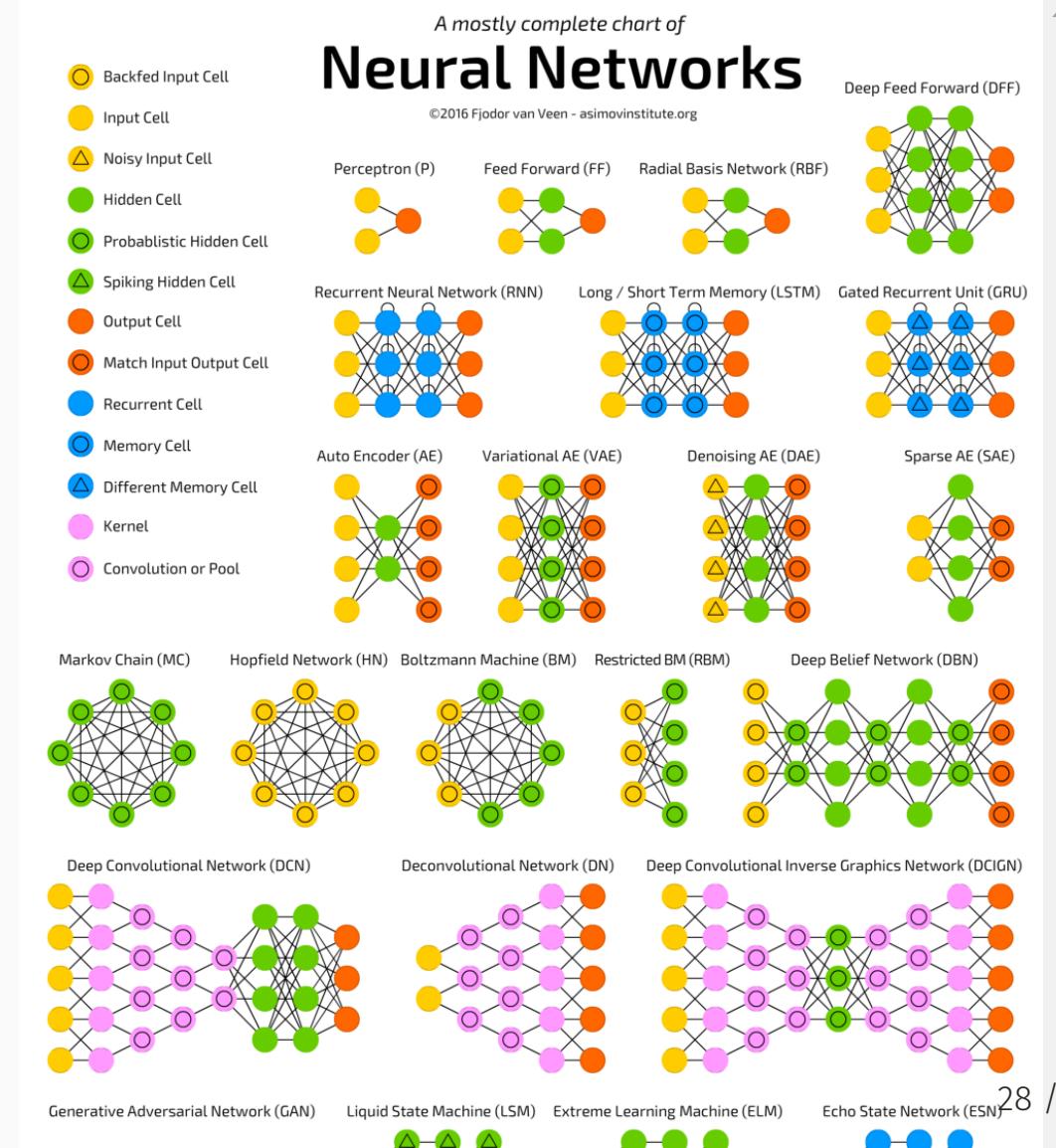
```
input[1, , ]
```

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!



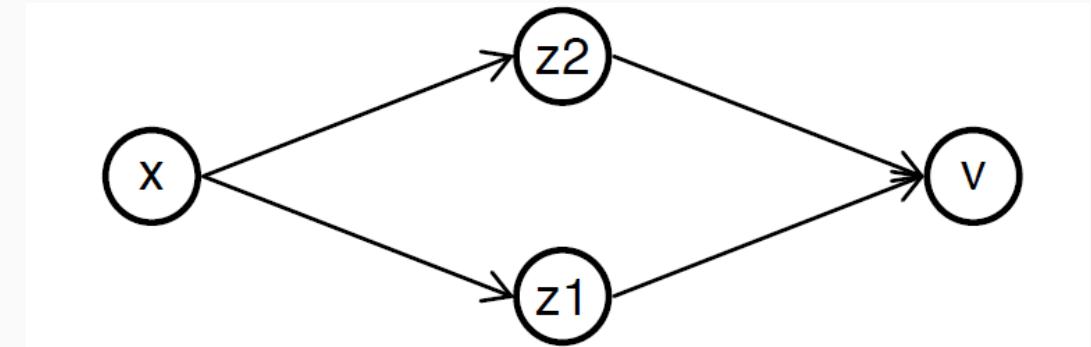
A simple neural network

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**
- 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**:

$$\begin{aligned}z_1 &= b_1 + x \cdot w_1 \\z_2 &= b_2 + x \cdot w_2\end{aligned}$$

using four parameters: two intercepts and two slopes.

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

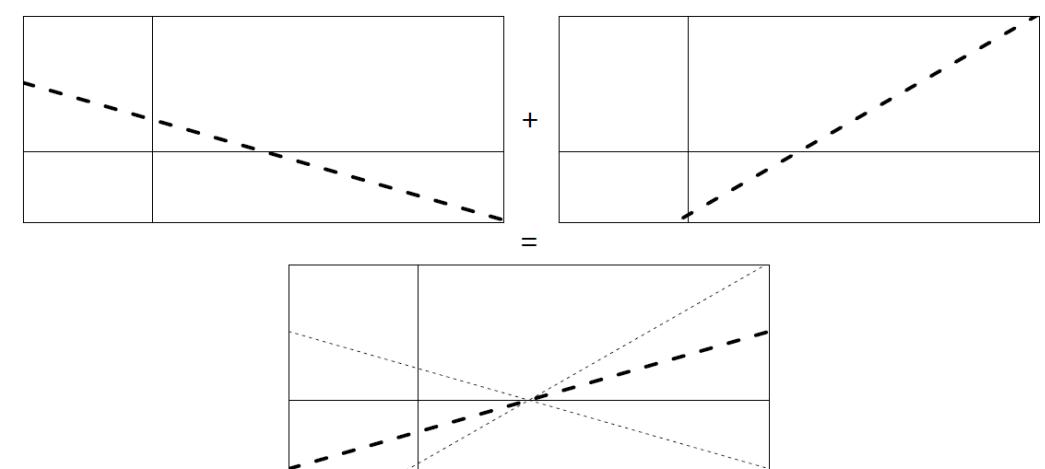
$$\hat{y} := v = b_3 + z_1 \cdot u_1 + z_2 \cdot u_2.$$

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

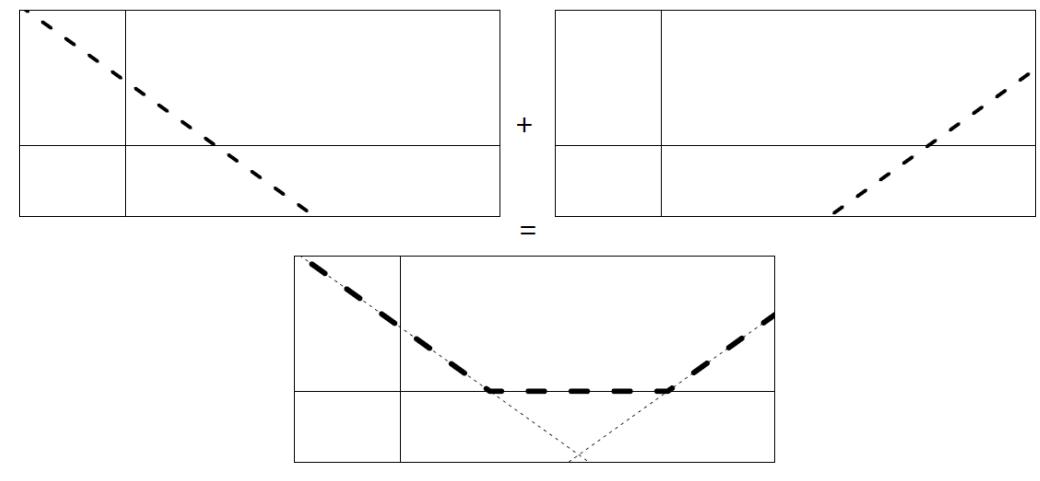
$$\begin{aligned} v &= b_3 + \sigma(z_1) \cdot u_1 + \sigma(z_2) \cdot u_2 \\ &= b_3 + \sigma(b_1 + x \cdot w_1) \cdot u_1 + \sigma(b_2 + x \cdot w_2) \cdot u_2, \end{aligned}$$

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:

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



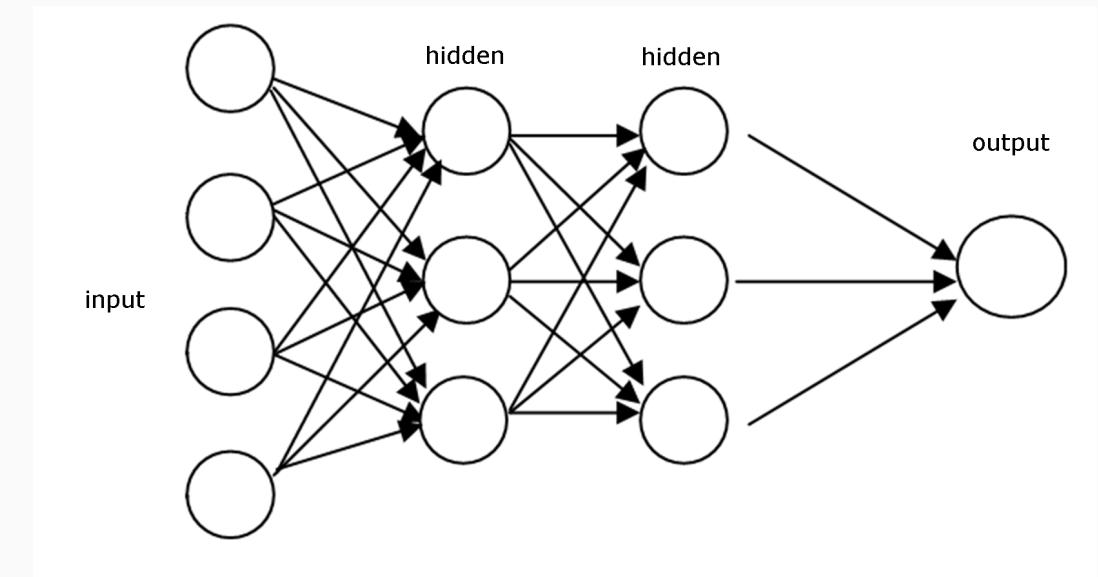
Many more activation functions: sigmoid, softmax, identity, etc. (see further).

Wrap-up on 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.

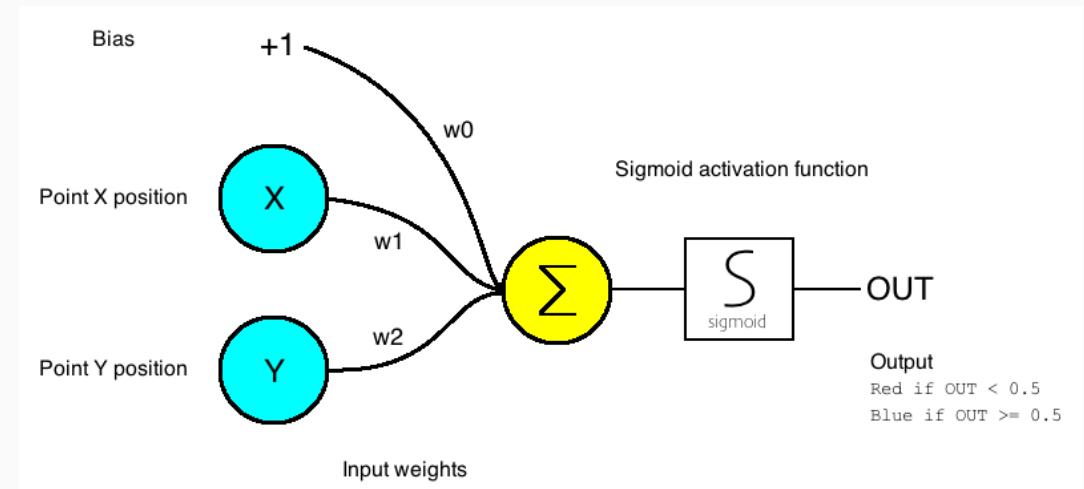


This is a **feedforward** neural network - no loops!

Wrap-up on ANNs - the neural nets' newspeak

Using the neural nets newspeak or language:

- intercept called **the bias**
- slopes called **weights**
- L layers in total, with input layer denoted as layer 0
- use a (from **activation**) to denote the output of a given neuron in a given layer.



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

Neural network architecture in Keras

Preparing the MNIST data for an ANN

Let's construct a densely connected, feed forward neural network for the multi-class classification problem in MNIST.

We **flatten** the image data (28x28 matrix) into a vector of length 784:

```
input ← tensorflow::array_reshape(input,  
                                    c(nrow(input), 28*28)) / 255  
  
test_input ← tensorflow::array_reshape(test_input,  
                                       c(nrow(test_input), 28*28)) / 255
```

Later in this course we will see how we can analyze this data **without flattening!**

We construct 10 dummy variables (0-9) for the output of the model:

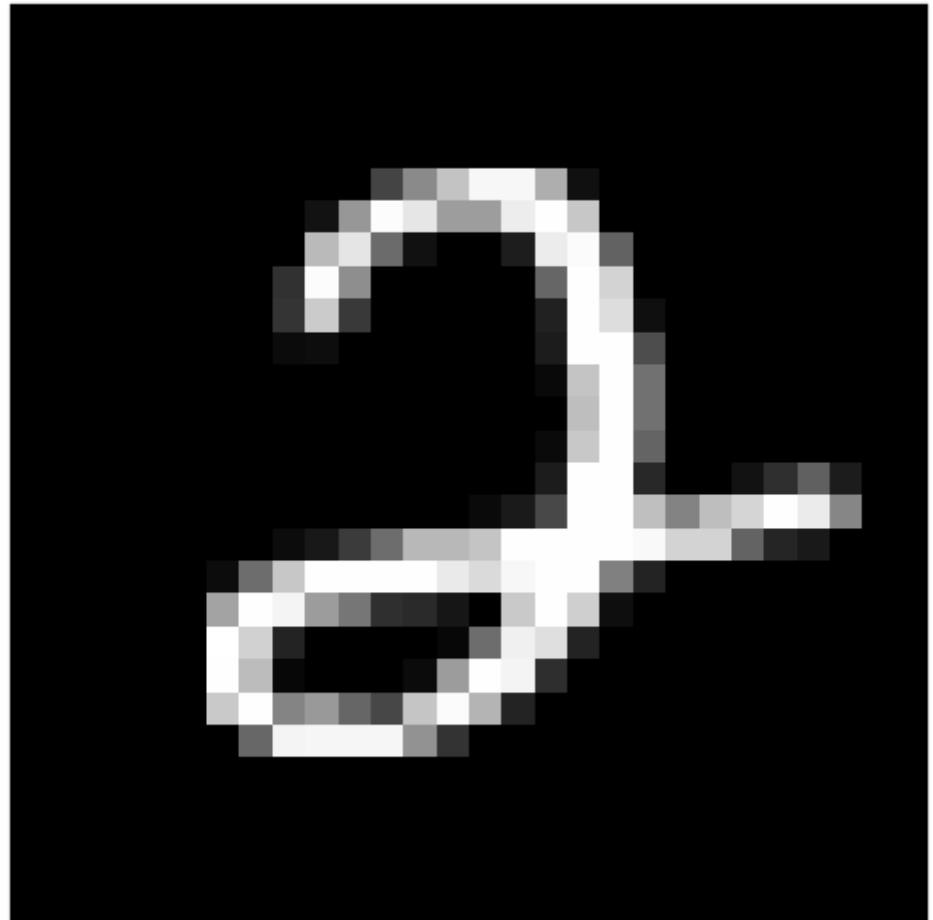
```
output ← keras::to_categorical(output, 10)
```

```
test_output ← keras::to_categorical(test_output, 10)
```

Preparing the MNIST data for an ANN (cont.)

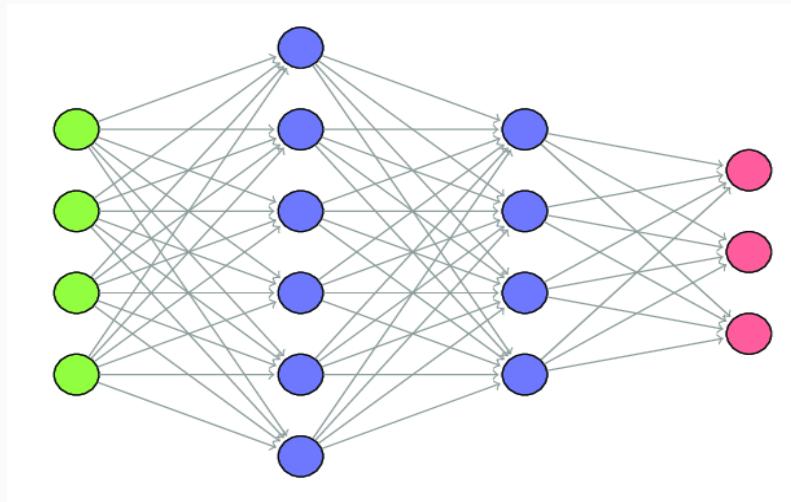
The R-script includes a function `plot_image` to visualize the input:

```
plot_image(input[17, ], legend = FALSE)
```



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.

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
```

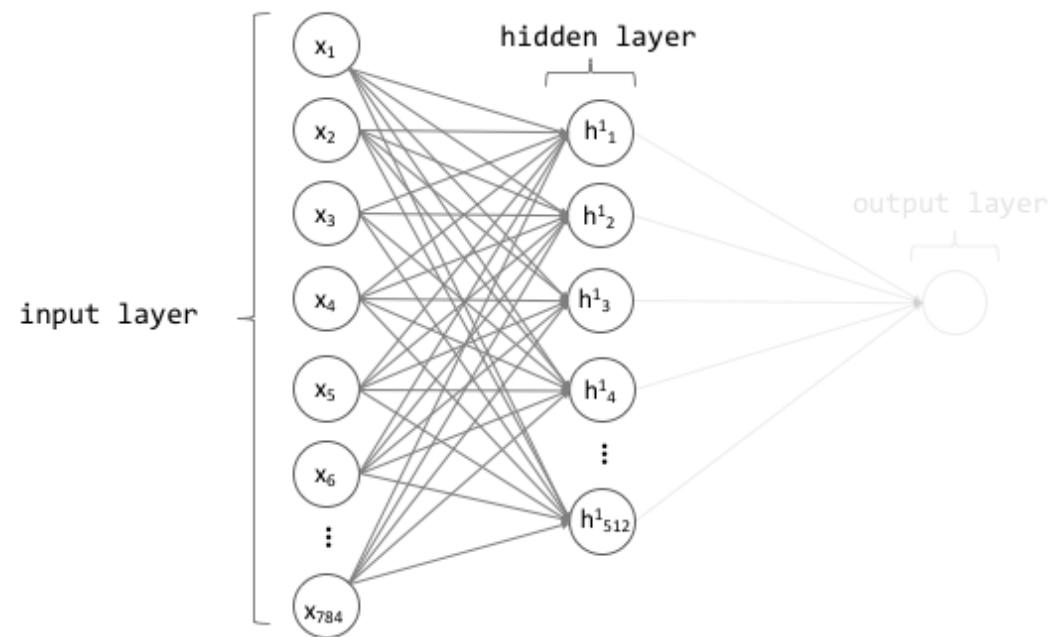
- We can add multiple hidden layers by adding more `layer_dense()` functions.
- Technically, **deep learning** refers to any neural network that has 2 or more hidden layers.
- The last `layer_dense()` will always represent the output layer.

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: the MNIST pictures (28x28) are 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 explicitly define.

We let the model decide the optimal features.



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.



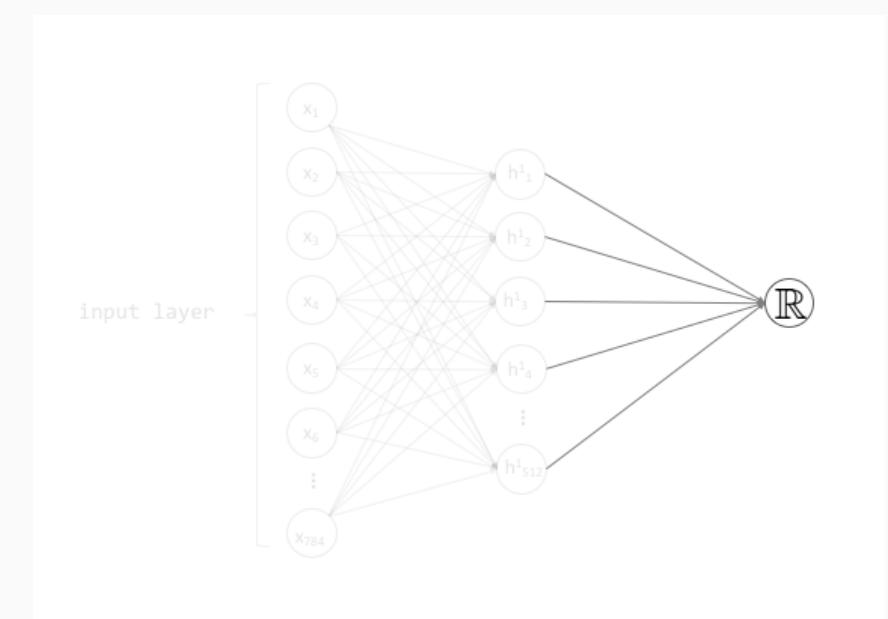
Output layer

```
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`:



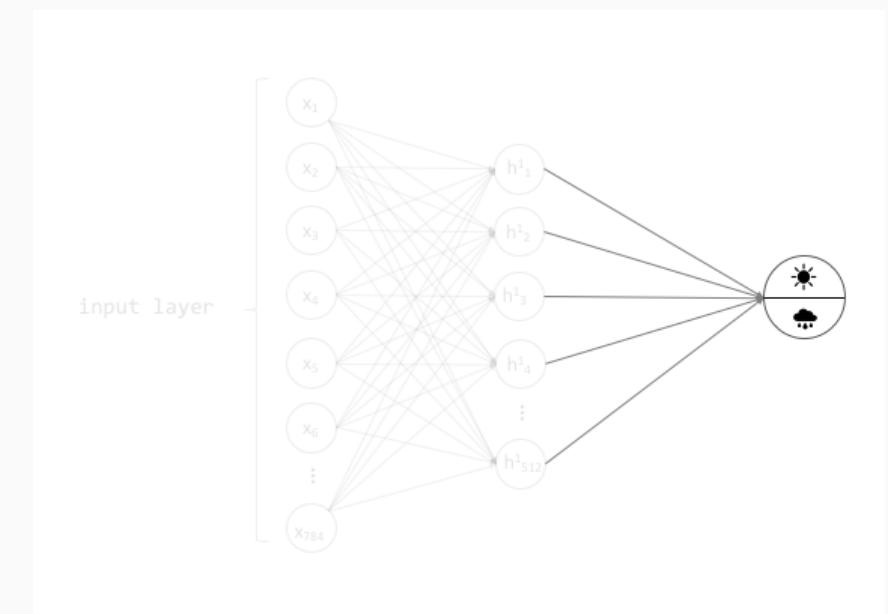
Output layer

```
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`



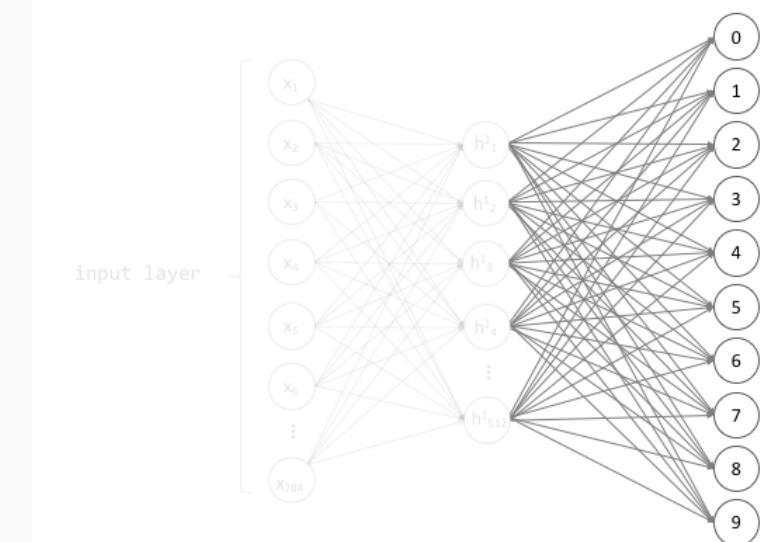
Output layer

```
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`



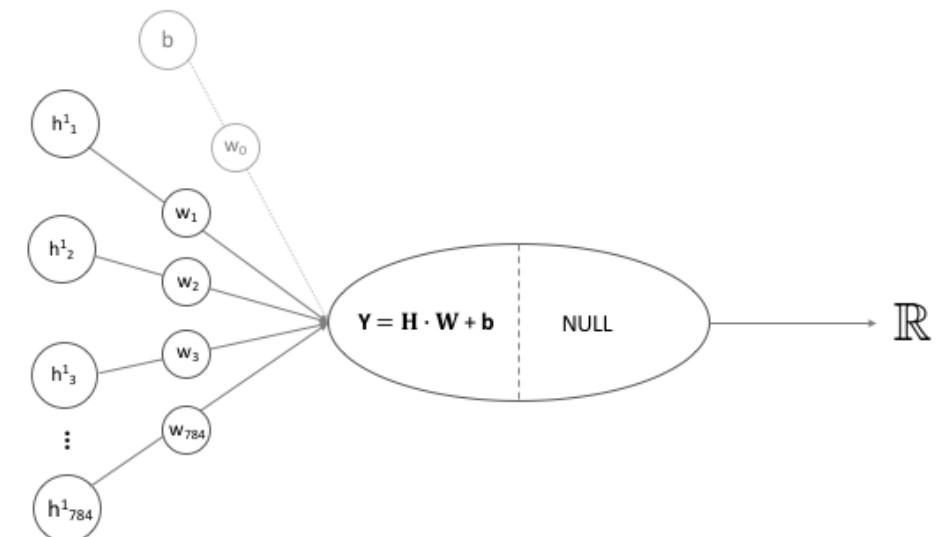
Output layer

```
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)



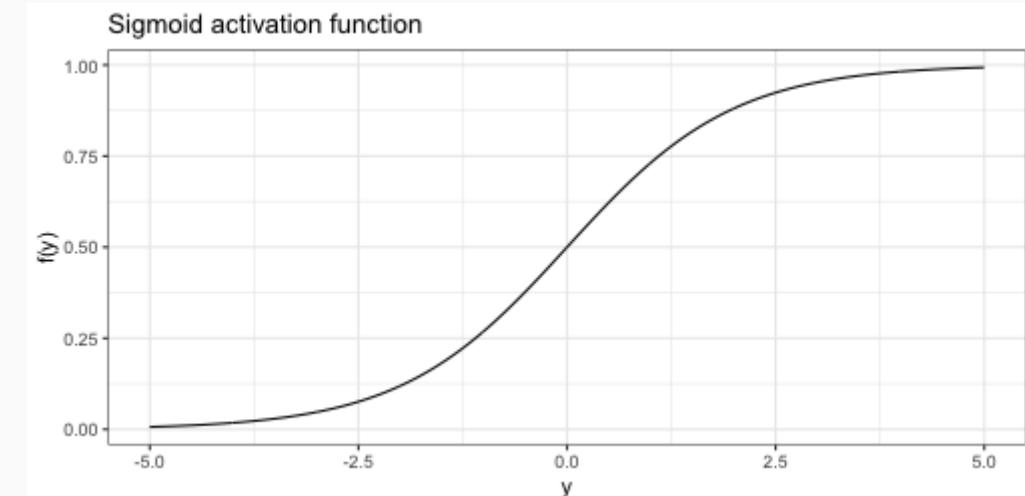
Output layer

```
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	$y_9 = H \cdot W + b$	$f(y_9) = \frac{e^{y_9}}{\sum_i e^{y_i}}$	0.01
			<u>1.00</u>



Your turn

Ultimately, here is a summary of the network architecture for the MNIST data

```
model <-  
  keras_model_sequential() %>%  
  layer_dense(units = 512,  
              activation = 'sigmoid',  
              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)
## -----
## dense_1 (Dense)
##
## -----
## dense (Dense)
## -----
## 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, $(784 \times 512) + 512 = 401,920$ parameters
- multi-class output layer, with
 - 10 nodes
 - softmax activation function
 - thus, $(512 \times 10) + 10 = 5,130$ parameters
- all together, that makes 407,050 parameters!

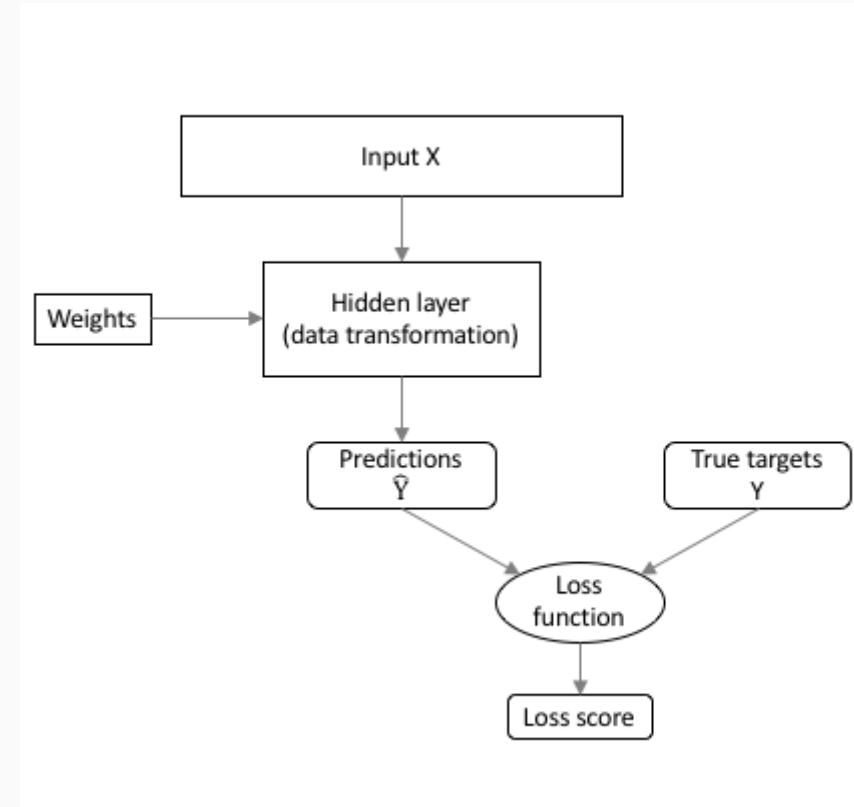
Network compilation

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'))
```

With `loss = "categorical_crossentropy"` the loss of a single training observation is

$$\sum_{j=0}^9 -p_j \cdot \log(f(y_j)),$$

where j runs over the classes in the multi-class prediction problem, $f(y_j)$ is the fitted probability of class j and p_j is a 0/1 hot-encoding of the truly observed class.

For instance, when the true input digit is 1 the vector p is $(0, 1, 0, 0, \dots, 0)$.

You can also define your **own loss** function in Keras, e.g.

```
mse ← function(y_true, y_pred) {  
  k_mean((y_true - y_pred)^2, axis = 2)  
}  
  
model ← model %>%  
  compile(loss = mse,  
          optimize = optimizer_rmsprop(),  
          metrics = c('accuracy'))
```

- `k_mean` is the keras implementation of `mean` that takes a tensor as input.
- `axis = 2` calculates the mean over the different output nodes.

Compiling the model (cont.)

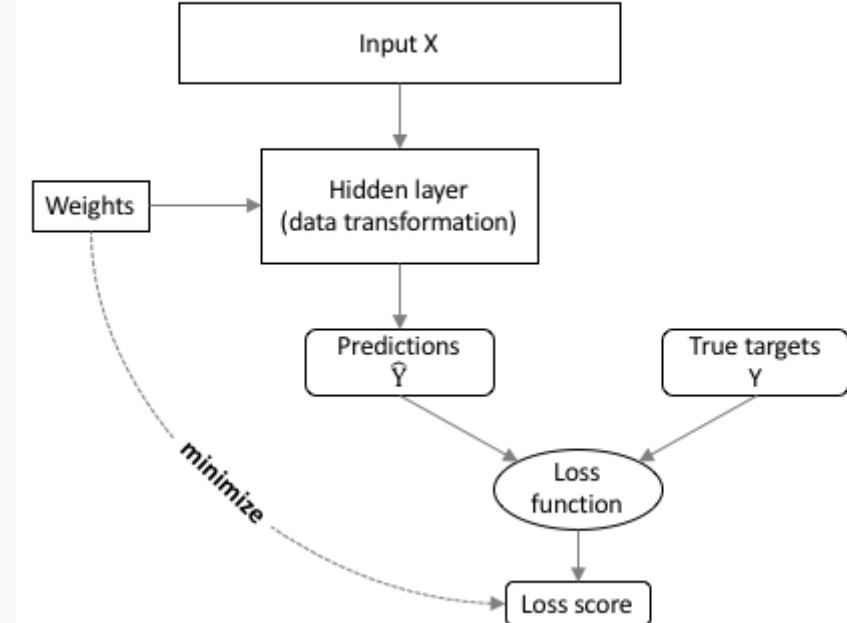
```
model ← model %>%  
  compile(loss = "categorical_crossentropy",  
          optimize = optimizer_rmsprop(),  
          metrics = c('accuracy'))
```

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)

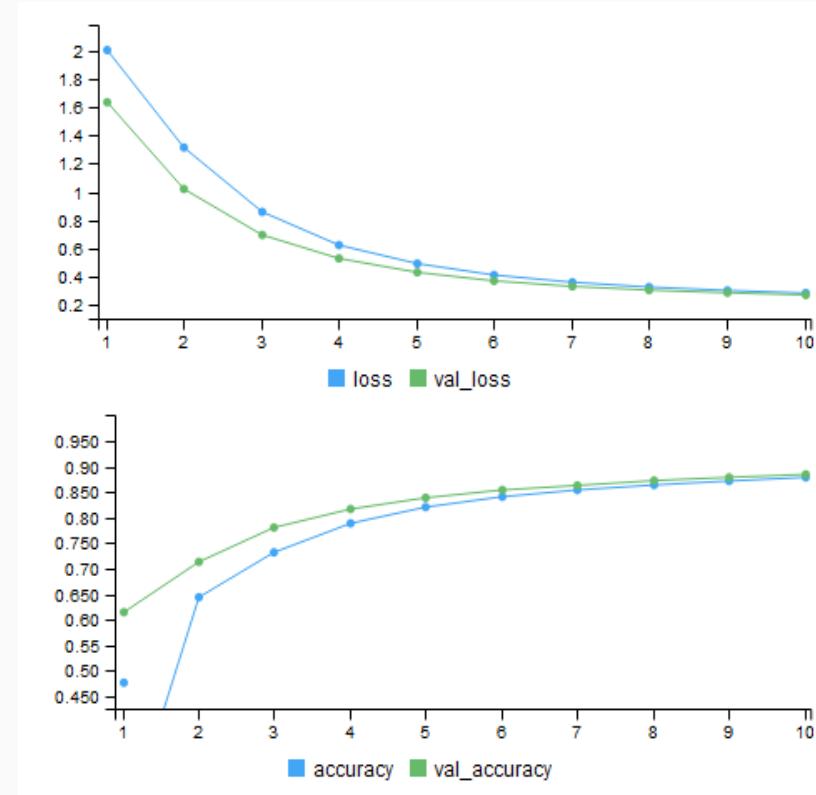
- **backpropagation**
- derivatives in layer L are easy
- derivatives in layer l as a function of derivatives in layer $l + 1$
- all about the **chain rule** for derivatives!

Performance metrics

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

In addition to the loss function, other **performance measures (metrics)** can be tracked while calibrating the model.

- accuracy (= categorical_accuracy)
- binary_accuracy
- categorical_accuracy
- sparse_categorical_accuracy
- top_k_categorical_accuracy
- sparse_top_k_categorical_accuracy
- cosine_proximity
- any loss function





Your turn

As discussed, any **loss function** can be also used as an accuracy **metric**.

In the case of the MNIST dataset, we search for a model with a high **accuracy**. Hereto, we calculate the number of times the class of the observed y equals the predicted class \hat{y} , and divide by the size of the (training) set.

Q: Why can we not use **accuracy** as our loss function?

Fitting the model

`fit(.)` tunes the model parameters (the weights and bias terms).

There is no need to save the result as the input model gets updated automatically.

```
model %>%  
  fit(input,  
       output,  
       batch_size = 128,  
       epochs = 10,  
       validation_split = 0.2)
```

The first arguments are the input data (here: training images stored in `input`) and their corresponding class (here: 0-9, the labels of the training data, stored in `output`).

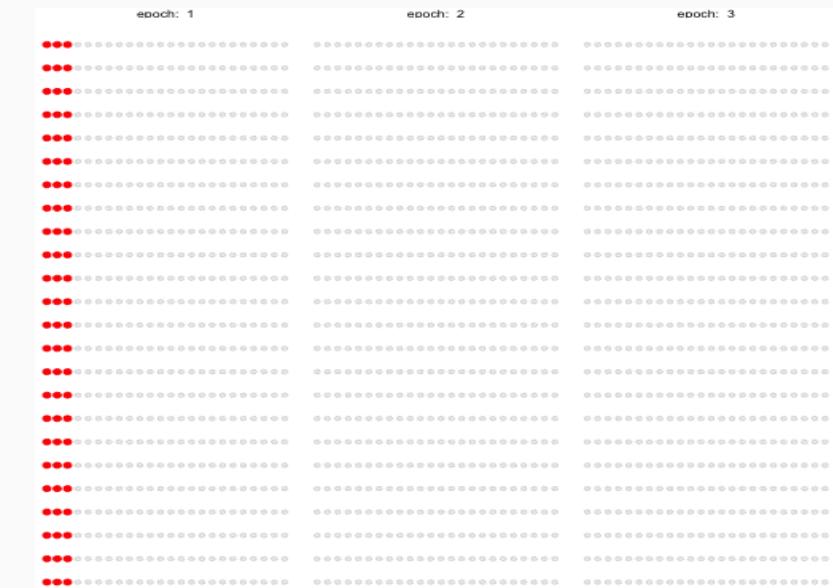
Fitting the model (cont.)

`fit(.)` tunes the model parameters (the weights and bias terms). We use `fit()` to start executing model training.

```
model %>%  
  fit(input,  
       output,  
       batch_size = 128,  
       epochs = 10,  
       validation_split = 0.2)
```

Parameter updates are calculated based on small subsets of the training data with `batch_size` elements.

An `epoch` is one iteration of the algorithm over the full dataset.



Source: Bradley Boehmke

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.

Fitting the model (cont.)

`fit(.)` tunes the model parameters (the weights and bias terms).

```
model %>%  
  fit(input,  
       output,  
       batch_size = 128,  
       epochs = 10,  
       validation_split = 0.2)
```

With the `validation_split = 0.2` we use the last 20% of our input training data as a hold-out validation set.

We evaluate the loss on this validation set at the end of each epoch.



Your turn

You will now **design, compile** and **fit** your own neural network for the MNIST dataset.

As a form of parallelized model selection, all of us will play with different model parameters. This way we gain insight into which parameter values work well for this dataset.

Base model: the neural network with a single hidden layer, as specified in the R script.

Try some of the following ideas to improve the model: (more ideas on the next slide!)

- **add hidden layers:** the number of nodes in subsequent layers should decrease
- **change batch size**
- **change the activation function.**



Your turn

Examples of layer types

- `layer_gaussian_noise`: adds gaussian noise $N(0, \text{stddev})$ to the nodes when training the model. This reduces the probability of overfitting.

```
model <- model %>%
  layer_gaussian_noise(stddev)
```

- `layer_dropout`: sets a fraction `rate` of the input units to zero. This reduces the probability of overfitting.

```
model <- model %>%
  layer_dropout(rate)
```

- `layer_batch_normalization`: centers and scales the values of each node in the previous layer.

```
model <- model %>%
  layer_batch_normalization()
```

Several methods proposed to reduce overfitting:

- try different weight initializations
- early stopping
- regularization
- dropout.

Available in {keras} as:

```
layer_dense(units = , activation = "relu",
            kernel_regularizer = regularizer_l2(l = 0.001))
```

```
layer_dropout(0.6)
```

```
# fit with callbacks
model %>% fit(___, ___, callbacks = list(
  callback_early_stopping(___)
))
```

With **early stopping**:

- calculate validation performance after each epoch
- stop when this no longer improves.

With **regularization** (cfr. *lasso* and *{glmnet}*): e.g.

$$\min_{w,b} \mathcal{L}(w, b) + \frac{\lambda}{2} \cdot \|w\|_2^2.$$

With **dropout**:

- randomly set activations to zero, with fixed probability p
- both in forward propagation as well as backpropagation
- only in training, all nodes turned on during prediction.

For more details, please consult

https://keras.rstudio.com/articles/training_callbacks.html.

Model evaluation

`evaluate(.)` calculates losses and metrics on the test dataset.

```
model %>%
  evaluate(test_input, test_output, verbose = 0)
##      loss accuracy
## 0.2336414 0.9341000
```

`predict(.)` returns a vector of length 10 with the probability per output node.

```
prediction ← model %>%
  predict(test_input)

round(prediction[1, ], 3)
## [1] 0.000 0.000 0.001 0.003 0.000 0.000 0.000 0.995 0.000 0.001
```

The predicted category is the node with the highest probability.

```
category ← apply(prediction, 1, which.max) - 1
actual_category ← apply(test_output, 1, which.max) - 1
```

Model evaluation (cont.)

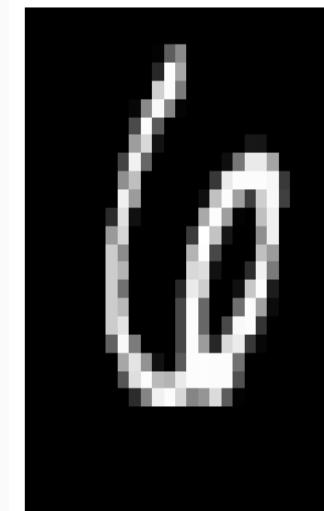
We inspect the misclassified images to gain more insight in the model.

Below we show some examples for our pre-trained MNIST neural network.

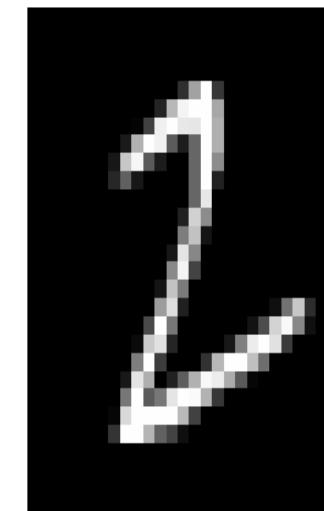
```
head(which(actual_category != category))  
## [1] 9 34 39 64 88 93
```

```
index ← 9  
plot_image(test_input[index, ]) +  
  ggtitle(paste(  
    'actual: ', actual_category[index],  
    ' predicted: ', category[index], sep='')) +  
  theme(legend.position = 'none',  
        plot.title = element_text(hjust = 0.5))
```

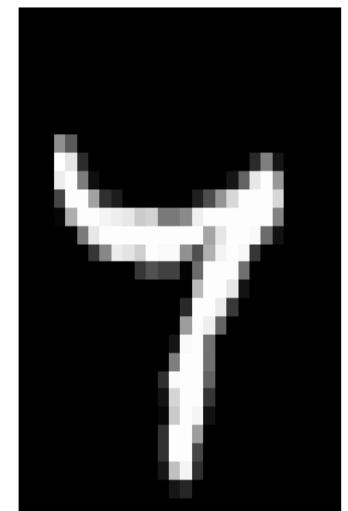
actual: 6 predicted: 2



actual: 2 predicted: 3



actual: 7 predicted: 4

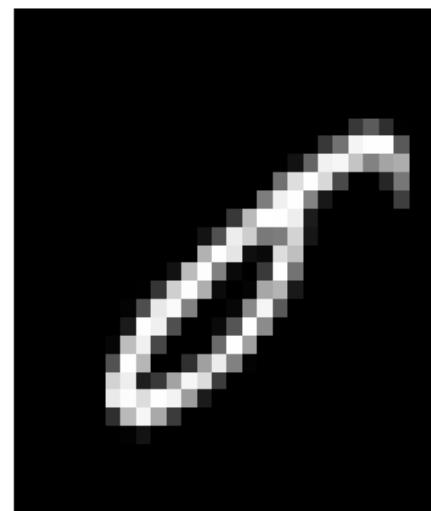


Model evaluation (cont.)

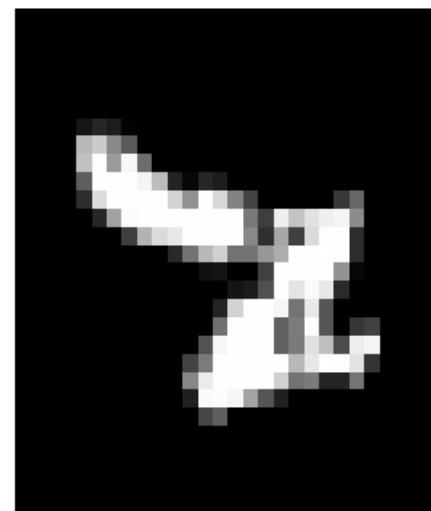
We inspect the images for which the model assigns the lowest probability to the correct class.

```
# select per row, the probability corresponding to the correct class  
prob_correct ← prediction[cbind(1:nrow(prediction), actual_category+1)]  
  
# get the index of the 5 lowest records in prob_correct  
which(rank(prob_correct) ≤ 5)  
## [1] 1501 5735 6167 6506 6652
```

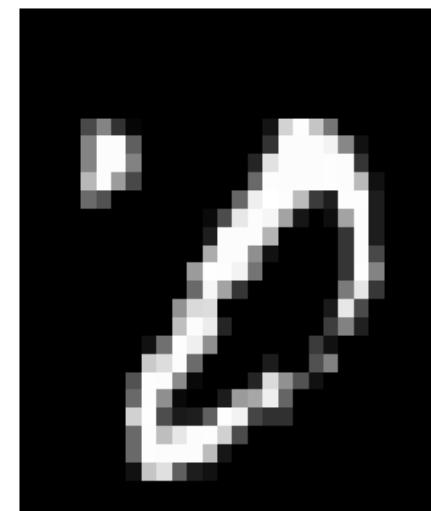
actual: 0 predicted: 5



actual: 2 predicted: 4



actual: 0 predicted: 3





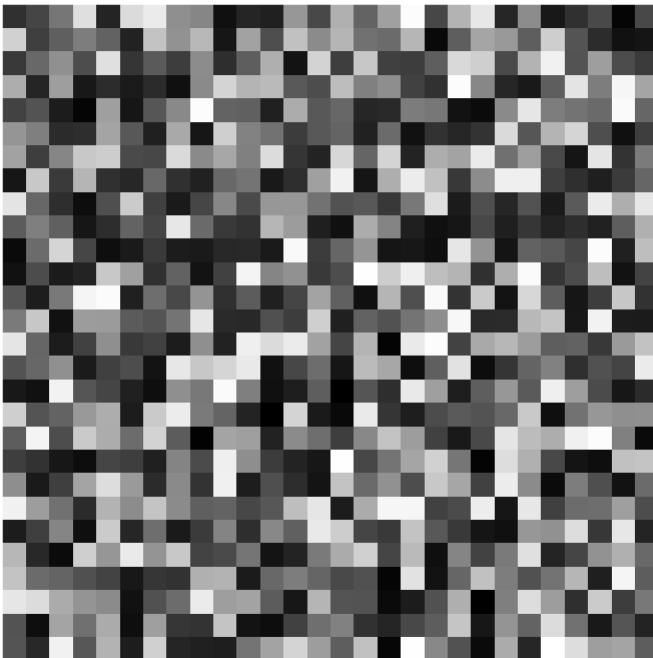
You will now **evaluate** your own model!

Your turn

- **Q.1:** calculate the accuracy of your model on the test set.
- **Q.2:** visualize some of the misclassified images from your model.
- **Q.3:** generate an image consisting of random noise and let the model classify this image.
What do you think of the results?
Remember: your input should be a 1x784 matrix with values in [0, 1].

Feeding random data to a neural network

```
random ← matrix(runif(28^2), nrow = 1)  
plot_image(random[1, ])
```



```
round(predict(model, random), 3)  
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]  
## [1,]     0    0 0.617 0.124 0.002 0.196 0.042 0.001 0
```

Our pre-trained MNIST model is pretty sure that the input on the left is a two!

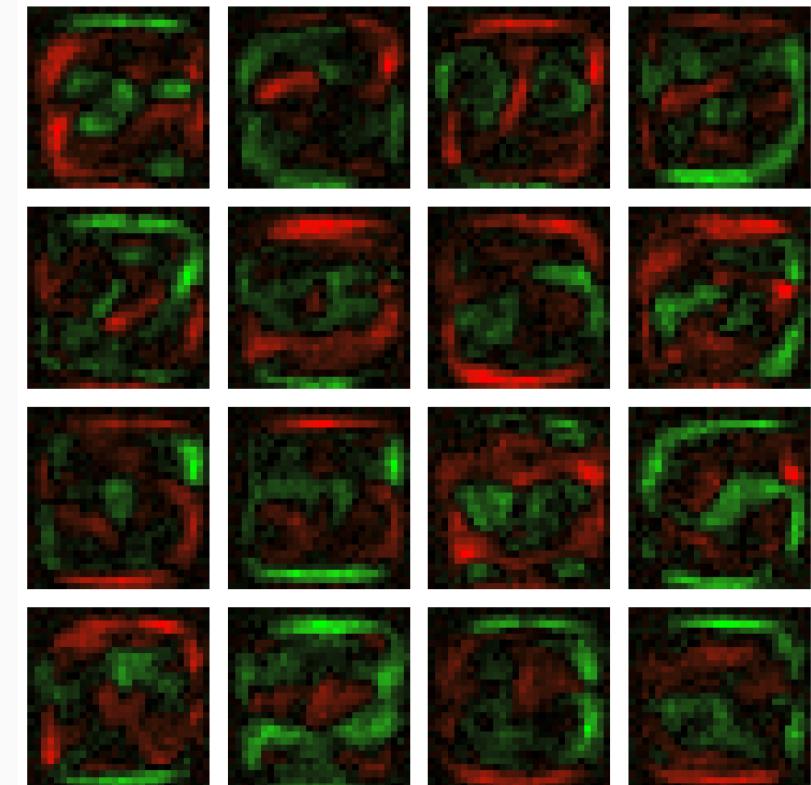
Model understanding

Inspecting the calibrated weights can provide some insight in the features created in the hidden layer.

Every node in the first hidden layer has 784 connections with the input layer.

The weights of these connections can be visualized as an 28x28 image.

```
node <- 9  
layer <- 1  
weights <- model$get_weights()[[2*(layer-1) + 1]][, noc  
plot_image(as.numeric(weights))
```



On the right we show a visualization of the calibrated weights for a pre-trained model with (only) 16 nodes in the first hidden layer.

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:

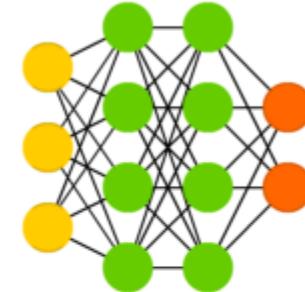
- the number of layers
- the number of nodes per layer
- the activation functions
- the layer type (*more on this coming soon*)
- the loss function
- the optimization algorithm
- the batch size
- the number of epochs
- ...

Neural Networks

©2016 Fjodor van Veen - asimovinstitute.org

- (○) Backfed Input Cell
- (○) Input Cell
- (△) Noisy Input Cell
- (●) Hidden Cell
- (○) Probabilistic Hidden Cell
- (△) Spiking Hidden Cell
- (●) Output Cell
- (○) Match Input Output Cell
- (●) Recurrent Cell
- (○) Memory Cell
- (△) Different Memory Cell
- (●) Kernel
- (○) Convolution or Pool

Deep Feed Forward (DFF)



Perceptron (P)



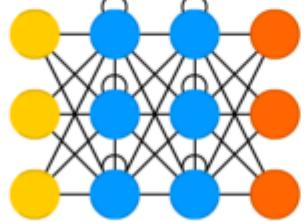
Feed Forward (FF)



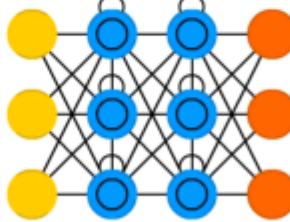
Radial Basis Network (RBF)



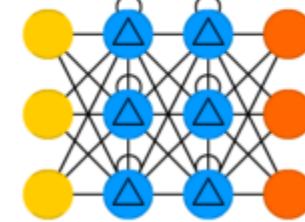
Recurrent Neural Network (RNN)



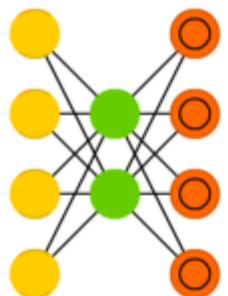
Long / Short Term Memory (LSTM)



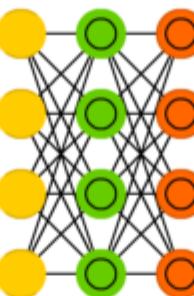
Gated Recurrent Unit (GRU)



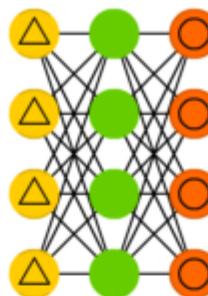
Auto Encoder (AE)



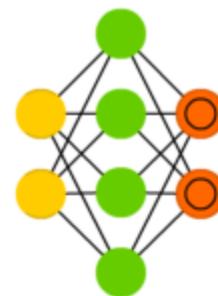
Variational AE (VAE)



Denoising AE (DAE)



Sparse AE (SAE)



Markov Chain (MC)



Hopfield Network (HN)



Boltzmann Machine (BM)



Restricted BM (RBM)



Deep Belief Network (DBN)



Claim frequency and severity regression

Preparing the MTPL data

We predict **claim frequency** and **severity** in the MTPL data with a **neural network**.

Load the **MTPL** data:

```
#setwd(dirname(rstudioapi::getActiveDocumentContext()$path))
mtpl <- read.table('..../data/PC_data.txt',
                    header = TRUE, stringsAsFactors = TRUE) %>%
  as_tibble() %>% rename_all(tolower) %>% rename(expo = exp)
```

Create a **training** and **test** set with {rsample}:

```
library(rsample)
set.seed(54321)
data_split <- initial_split(mtpl)
mtpl_train <- training(data_split)
mtpl_test <- testing(data_split)
# Reshuffling of the training observations
mtpl_train <- mtpl_train[sample(nrow(mtpl_train)), ]
```

Regression with neural networks

In Module 1 we fitted **GLMs** for claim frequency as follows:

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

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
β	fitted effect	weights

Your first claim frequency NN

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

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

```
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 a constant intercept.
- `loss = 'poisson'`: we maximize the **Poisson** likelihood.

Your first claim frequency NN

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

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

```
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?

```
nn_freq_intercept$count_params()  
## [1] 1
```

- `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 a constant intercept.
- `loss = 'poisson'`: we maximize the **Poisson** likelihood.

Your first claim frequency NN (cont.)

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

```
intercept <- rep(1, nrow(mtpl_train))  
  
counts <- mtpl_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:

```
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**.



Your turn

We have shown that a **Poisson GLM** can be implemented as a neural network.

- **Q1:** adapt this code to replicate a **binomial GLM** with a **logit** link function. Add **accuracy** as a **metric** in your model.

Hint 1: the `sigmoid` activation function is the inverse of the logit link function.

Hint 2: the `binary_crossentropy` loss maximizes the loglikelihood of Bernoulli outcomes:

$$\sum_{i=1}^n (\textcolor{orange}{y_i} \cdot \log(p_i) + (1 - \textcolor{orange}{y_i}) \cdot \log(1 - p_i)).$$

- **Q2:** **fit** your NN on the outcome variable (`nclaims > 0`), i.e. modeling no claim versus having at least one claim.
- **Q3:** **compare** your fitted neural network with a **GLM**.

Q1: set-up for binary classification

```
nn_binary <-  
keras_model_sequential() %>%  
layer_dense(units = 1,  
            activation = 'sigmoid',  
            input_shape = c(1),  
            use_bias = FALSE) %>%  
compile(loss = 'binary_crossentropy',  
        optimize = optimizer_rmsprop(),  
        metrics = c('accuracy'))
```

Q2: fit your NN

```
nn_binary %>% fit(x = intercept,  
                    y = counts > 0,  
                    epochs = 40,  
                    batch_size = 1024,  
                    validation_split = 0,  
                    verbose = 0)
```

Q3: compare with a GLM

```
glm_binary <- glm(nclaims > 0 ~ 1,  
                   data = mtpl_train,  
                   family = binomial(link = 'logit'))  
  
glm_binary$coefficients  
## (Intercept)  
## -2.064613  
nn_binary$get_weights()[[1]] %>% as.numeric()  
## [1] -2.063514  
  
unique(predict(glm_binary, type = 'response'))  
## [1] 0.1125841  
unique(predict(nn_binary, x = intercept))  
## [,1]  
## [1,] 0.112694
```

Taking exposure into account in a NN

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

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

which can be rewritten as follows:

$$\mathcal{L} = \sum_i \text{expo}_i \cdot \left(\lambda_i - \frac{Y_i}{\text{expo}_i} \log(\lambda_i) \right).$$

This is the loss function for a Poisson model with:

- observations $\frac{Y_i}{\text{expo}_i}$ and
- weights 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 NN (cont.)

Nothing changes in our NN model **specification**:

```
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 NN will pick up where it left off last time,
with the optimal weights after fitting.

Create a vector with exposure values:

```
exposure <- mtpl_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 ← mtpl_train$ageph
```

Define the NN **architecture**:

```
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** (see Module 1):

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

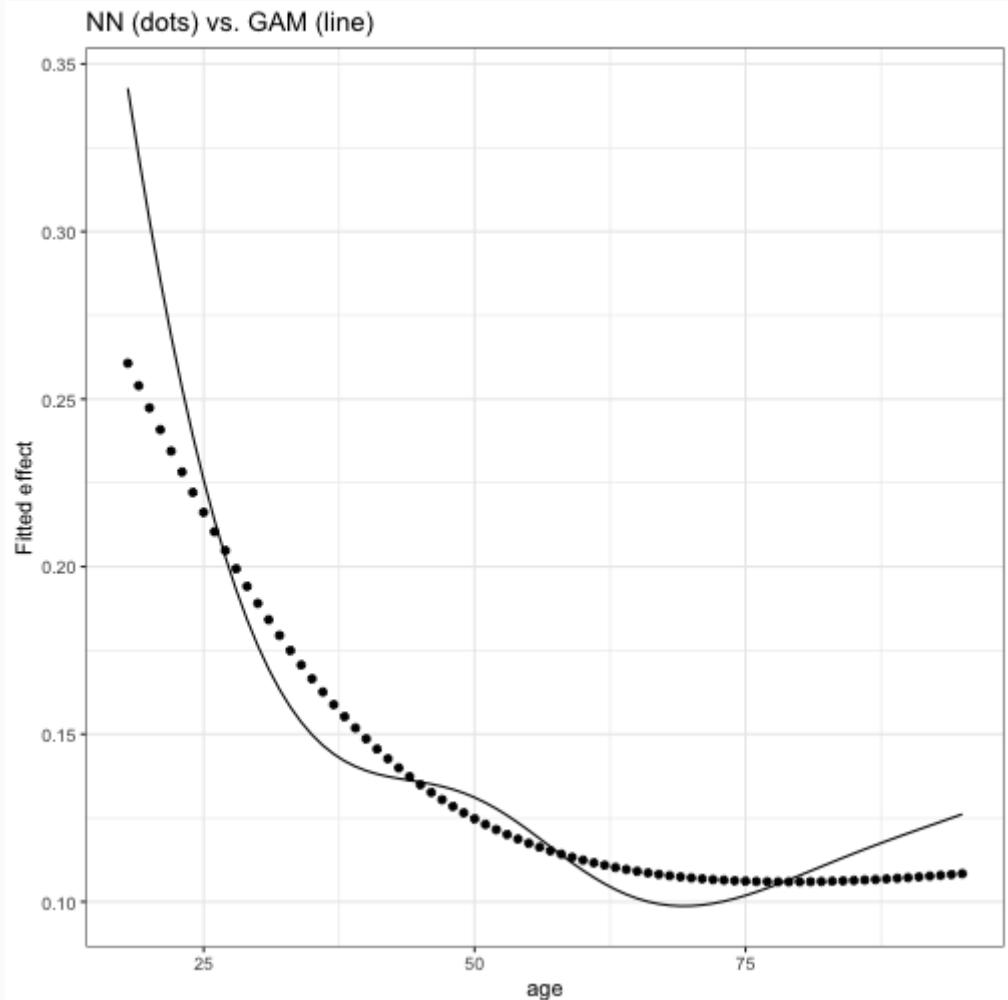
Adding an input feature and a hidden layer (cont.)

Let's **fit** our brand new NN:

```
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** to `ageph`:

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

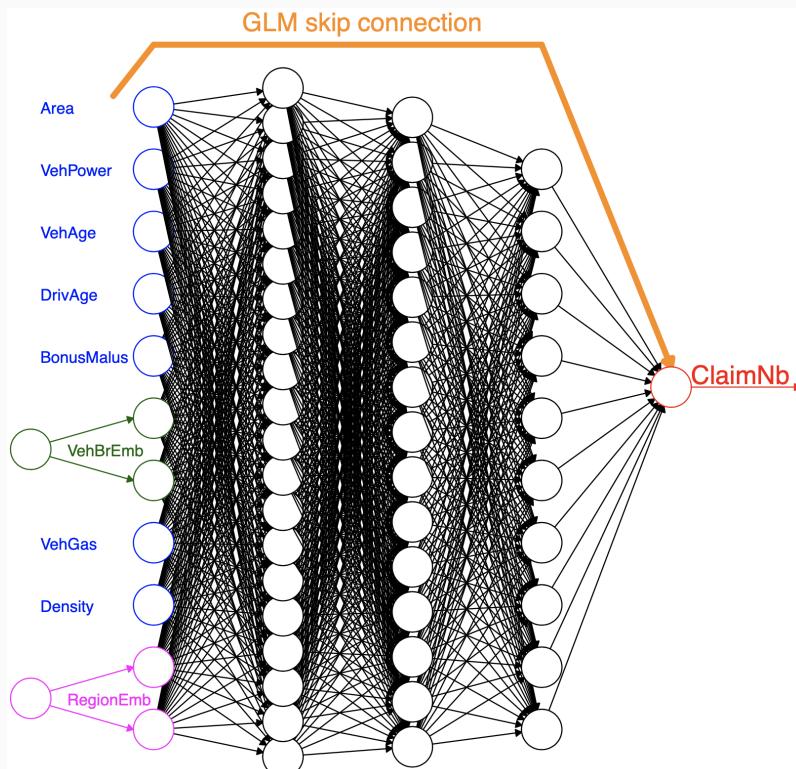


Q: What do you think about those fits?

Adding a skip connection in a NN

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**.



The output node, without skip connection, calculates:

$$f_{activation} \left(\sum_i w_i h_i + b \right).$$

With a skip connection, this simply becomes:

$$f_{activation} \left(\sum_i w_i h_i + b + s \right).$$

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?

Figure taken from Schelldorfer and Wuthrich (2019)

Adding a skip connection in a NN (cont.)

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

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

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

- Adding a **base** prediction:

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

- The **combination** of both:

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

A skip connection allows us to guide the NN in the right direction and model **adjustments** to 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 NN (cont.)

Create two **input layers** via `layer_input`, one for the skip connection and one for the NN:

```
input_skip ← layer_input(shape = c(1), name = 'skip')
input_nn ← layer_input(shape = c(1), name = 'nn')
```

Specify the **architecture** to use for the NN part with a **linear** combination of the hidden nodes outputs:

```
network ← input_nn %>% layer_batch_normalization() %>% layer_dense(units = 5, activation = 'tanh') %>%
  layer_dense(units = 1, activation = 'linear')
```

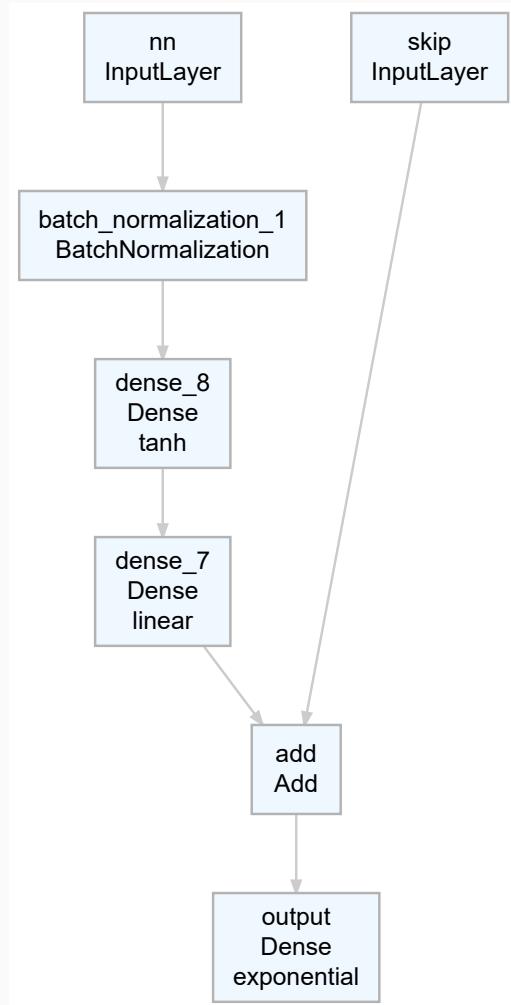
Combine the NN and skip connection via `layer_add` and pass through the `exponential` function without weights:

```
output ← list(network, input_skip) %>% layer_add() %>%
  layer_dense(units = 1, activation = 'exponential', trainable = FALSE, name = 'output',
              weights = list(array(1, dim = c(1,1)), array(0, dim = c(1))))
```

Define the full model with **inputs** and **output** via `keras_model` and **compile** as usual:

```
cann ← keras_model(inputs = list(input_nn, input_skip), outputs = output)
cann %>% compile(loss = 'poisson', optimizer = optimizer_rmsprop())
```

Adding a skip connection in a NN (cont.)



Calculate the GAM **base** predictions, including **exposure**:

```
gam_expo <- predict(gam_ageph) + log(mtpl_train$expo)
```

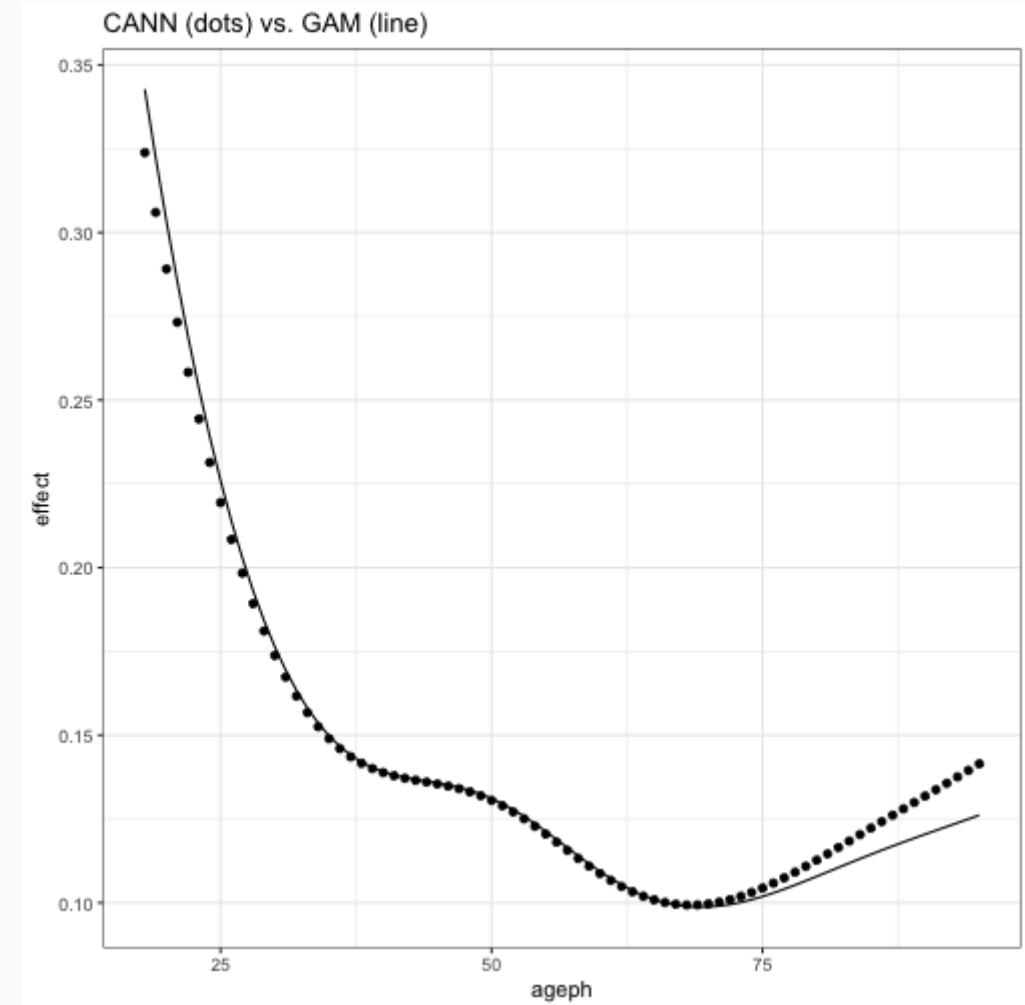
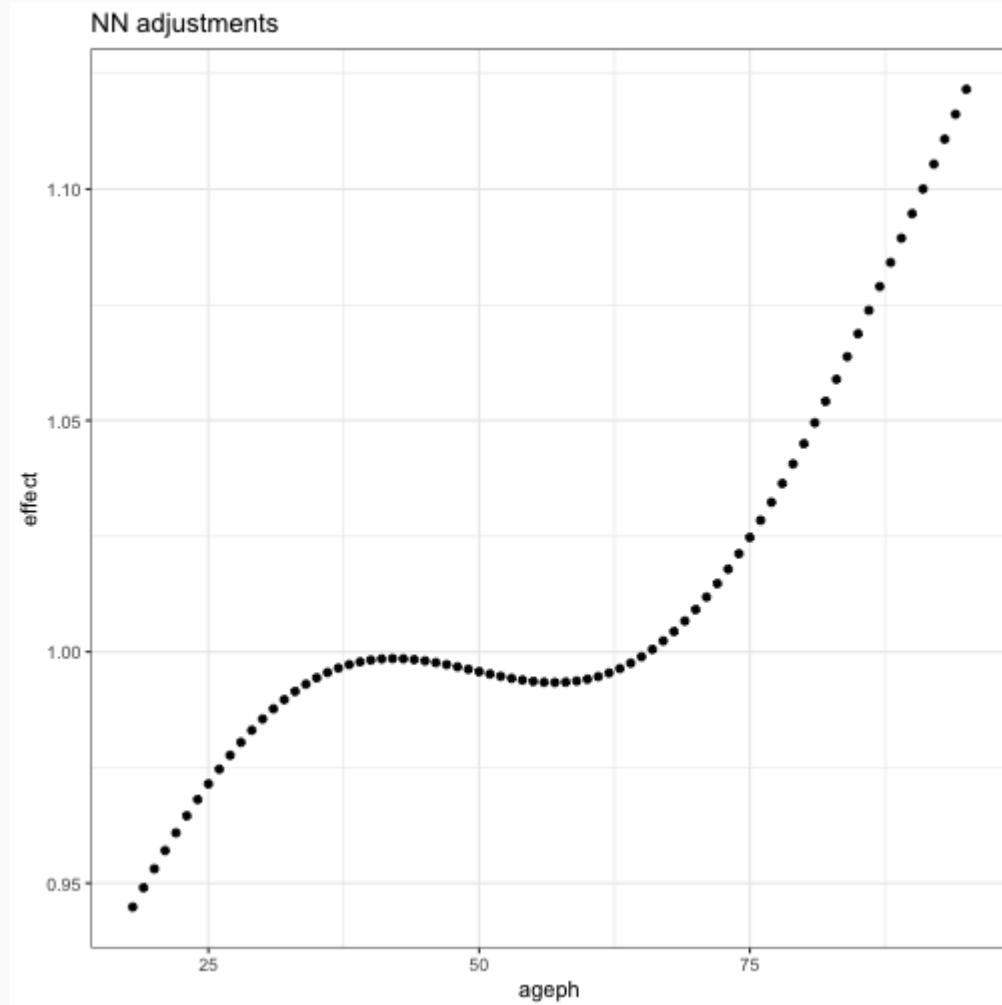
Collect the CANN input data in a **named list**:

```
cann_input <- list('nn' = mtpl_train$ageph,  
                     'skip' = gam_expo)
```

Fit the CANN like we have seen before:

```
cann %>% fit(x = cann_input,  
                 y = counts,  
                 epochs = 20,  
                 batch_size = 1024,  
                 validation_split = 0,  
                 verbose = 0)
```

Adding a skip connection in a NN (cont.)



Claim severity modeling with NNs

Within the GLM framework, **claim severity** is often modeled via a **lognormal** or **gamma** distribution.

Let's model the **log severity** with a **MSE** loss:

```
nn_sev_log <- keras_model_sequential() %>%
  layer_dense(units = 1, activation = 'linear',
              input_shape = c(1), use_bias = FALSE) %>%
  compile(loss = 'mse',
          optimize = optimizer_rmsprop())
claims <- mtpl_train %>% dplyr::filter(nclaims > 0)
nn_sev_log %>%
  fit(x = rep(1, nrow(claims)),
      y = log(claims$avg),
      epochs = 100, batch_size = 128,
      validation_split = 0, verbose = 0)
predict(nn_sev_log, 1) %>% exp() %>% as.numeric()
## [1] 451.989
claims$avg %>% summary()
##      Min.   1st Qu.   Median     Mean   3rd Qu.   19
##      0.0    142.2   523.5  1605.8  1426.4  19
```

Seems like we are heavily **underestimating** the mean?

The mean and median of the **lognormal** distribution:

$$\text{mean} = \exp(\mu + \sigma^2/2)$$

$$\text{median} = \exp(\mu)$$

We are doing the latter, but how to **estimate** σ in a NN?

Alternatives to consider:

- use MSE with **untransformed** severity amounts,
- implement **your own loss** function (e.g., gamma).



Your turn

Time to **apply** everything you learned on neural networks in a **claim frequency case study!**

The goal is to **fit** a couple of NNs on `mtpl_train` and **evaluate** those models on `mtpl_test`.

Step 1: Data pre-processing:

- Write a **recipe** to prepare the input data.
Feel free to check the {recipes} [reference page](#) for inspiration.
- Your recipe should contain at least the following **steps**:
 - Remove the features `id`, `amount`, `avg`, `town` and `pc` with `step_rm`.
 - Normalize all numeric features with `step_normalize`.
 - Dummy encode all nominal features with `step_dummy`.
- **Bake** the training and test data following your recipe.
- Make the train and test data **NN proof** as follows:
 - Create **vectors** for both `nclaims` and `expo`.
 - Create a **matrix** for the input features via `as.matrix()`.



Your turn

Step 2: Model building:

- **Compile** and **fit** a couple of neural networks on the pre-processed training data.
You are completely free to specify the architecture of your choice.
- You can **experiment** with the following options:
 - the number of layers,
 - the number of nodes per layer,
 - the batch size,
 - the activation functions,
 - adding dropout layers,
 - including batch normalization after some layers,
 - ...
- If you want to include exposure, either use **weights** or a **skip** connection.

Step 3. Model evaluation:

- **Evaluate** your models on the test data via `evaluate`. Which architecture is the winner?
- **Compare** your favorite neural network with a GLM, GAM or GBM. Which model is preferred?

Step 1: data pre-processing with {recipes}

```
library(recipes)

# Create and prepare the recipe
mpl_recipe <- recipe(nclaims ~ ., data = mpl_train) %>
  step_rm(id, amount, avg, town, pc) %>%
  step_nzv(all_predictors(), -expo) %>%
  step_normalize(all_numeric(), -c(nclaims, expo)) %>%
  step_dummy(all_nominal(), one_hot = TRUE) %>%
  prep(mpl_train)

# Bake the training and test data
mpl_train_b <- mpl_recipe %>% juice()
mpl_test_b <- mpl_recipe %>% bake(new_data = mpl_te

# Make the data NN proof
train_x <- mpl_train_b %>%
  dplyr::select(-c(nclaims, expo)) %>% as.matrix()
test_x <- mpl_test_b %>%
  dplyr::select(-c(nclaims, expo)) %>% as.matrix()

train_y <- mpl_train_b$nclaims
test_y <- mpl_test_b$nclaims

train_expo <- mpl_train_b$expo
test_expo <- mpl_test_b$expo
```

Step 2: a possible NN architecture with 2 hidden layers

```
nn_case <- keras_model_sequential() %>%
  layer_dense(units = 20,
              activation = 'relu',
              input_shape = ncol(train_x)) %>%
  layer_dense(units = 10,
              activation = 'relu') %>%
  layer_dense(units = 1,
              activation = 'exponential') %>%
  compile(loss = 'poisson',
          optimize = optimizer_nadam())

nn_case %>%
  fit(x = train_x,
      y = train_y / train_expo,
      sample_weight = train_expo,
      epochs = 20,
      batch_size = 1024,
      validation_split = 0,
      verbose = 0)
```

Step 3a: evaluate the NN on the test data

```
# Built-in evaluation
nn_case %>%
  evaluate(x = test_x,
            y = test_y,
            verbose = 0)
##      loss
## 0.3741113

# If you want to check the results
poisson_loss ← function(pred, actual) {
  mean(pred - actual * log(pred))
}
poisson_loss(predict(nn_case, test_x),
              test_y)
## [1] 0.3741113

# Use array for weights in evaluate
nn_case %>%
  evaluate(x = test_x,
            y = test_y,
            sample_weight = array(test_expo),
            verbose = 0)
##      loss
## 0.3413752
```

Step 3b: compare the NN performance with a GAM

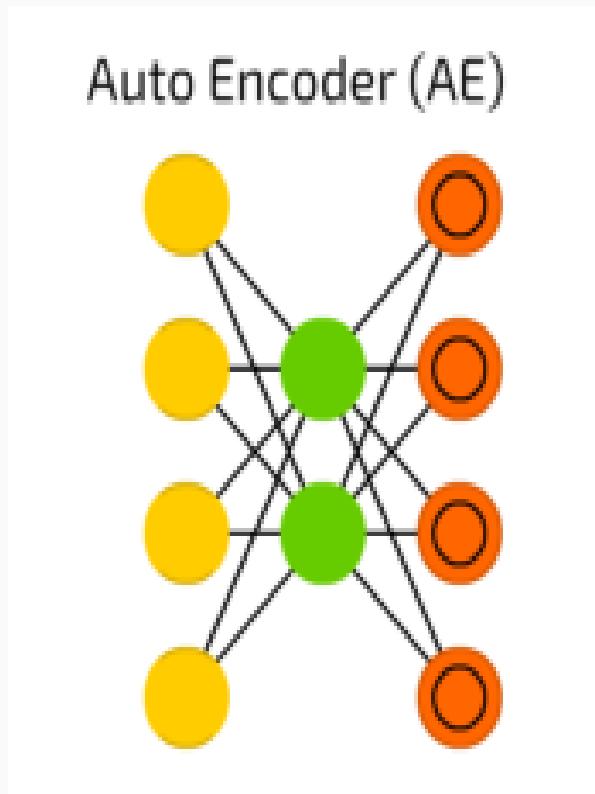
```
gam_case ← gam(
  nclaims ~ coverage + fuel + sex +
  s(ageph) + s(bm) + s(power) + s(agec),
  data = mtpl_train,
  offset = log(expo),
  family = poisson(link = 'log')
)

poisson_loss(predict(gam_case, mtpl_test,
                     type = 'response'),
              test_y)
## [1] 0.3746941
```

Auto encoders

Auto encoders

Auto encoders compress the input data into a limited number of features.



- **Unsupervised** machine learning algorithm.
- **Decorrelation** of the input data, comparable with PCA. The low dimensional compressed data is often used as an input in traditional statistical models.
- Input and output are identical.
- Few nodes in the center of the network. This is the compressed feature space.
- A high performing auto encoder is capable of reconstructing the input data based on compressed feature space.



Your turn

Auto encoders can be implemented in Keras using the same tools that you have already learned during this course.

The following steps guide you in **constructing** and **training** your personal auto encoder for the MNIST dataset.

- Make a sketch of the neural network that you will implement.
- Define a neural network with 5 layers:
 - Layer 1: input (784 nodes)
 - Layer 2: Hidden layer (128 nodes)
 - Layer 3: Hidden layer (32 nodes), this is the compressed feature space
 - Layer 4: Hidden layer (128 nodes)
 - Layer 5: Output layer (784 nodes)
- Choose an appropriate activation function for each layer:
 - identity
 - ReLU
 - sigmoid
 - softmax



Your turn

- Which of these loss functions can we use to train the model?
 - mse
 - binary_crossentropy
 - categorical_crossentropy
- Fit the model on the MNIST data in 10 epochs.
- Experiment with adding other layer types to the model:
 - layer_gaussian_noise(stddev)
 - layer_dropout(rate)
 - layer_batch_normalization()

```
encoder <- keras_model_sequential() %>%  
  layer_dense(units = 128, activation = 'sigmoid',  
              input_shape = c(784)) %>%  
  layer_dense(units = 32, activation = 'sigmoid')  
  
model <- encoder %>%  
  layer_batch_normalization() %>%  
  layer_dense(units = 128, activation = 'sigmoid') %>%  
  layer_dense(units = 784, activation = 'sigmoid') %>%  
  compile(loss = 'binary_crossentropy',  
          optimize = optimizer_rmsprop(),  
          metrics = c('mse'))  
  
model %>%  
  fit(input,  
       input,  
       epochs = 10,  
       batch_size = 256,  
       shuffle = TRUE,  
       validation_split = 0.2)
```

`encoder` contains the first part of the model for compressing the model.

`model` is the full auto encoder, including the encode and decode step.

By defining `model` as an extension of `encoder`, we can compress the data using `predict(encoder, ...)` after training the model.

```
encoder <- keras_model_sequential() %>%
  layer_dense(units = 128, activation = 'sigmoid',
              input_shape = c(784)) %>%
  layer_dense(units = 32, activation = 'sigmoid')

model <- encoder %>%
  layer_batch_normalization() %>%
  layer_dense(units = 128, activation = 'sigmoid') %>%
  layer_dense(units = 784, activation = 'sigmoid') %>%
  compile(loss = 'binary_crossentropy',
          optimize = optimizer_rmsprop(),
          metrics = c('mse'))

model %>%
  fit(input,
       input,
       epochs = 10,
       batch_size = 256,
       shuffle = TRUE,
       validation_split = 0.2)
```

I interpret the hidden nodes as binary features and therefore use a `sigmoid` activation function.

We no longer use the `softmax` activation function in the last layer, since multiple output nodes can be activated simultaneously.

I choose `binary_crossentropy` as a loss function, since we have independent bernoulli outcome variables.

Another good combination would have been:

- activation `ReLU` in the hidden layers
- activation `identity` in the output layer
- `mse` as the loss function

```
encoder <- keras_model_sequential() %>%
  layer_dense(units = 128, activation = 'sigmoid',
              input_shape = c(784)) %>%
  layer_dense(units = 32, activation = 'sigmoid')

model <- encoder %>%
  layer_batch_normalization() %>%
  layer_dense(units = 128, activation = 'sigmoid') %>%
  layer_dense(units = 784, activation = 'sigmoid') %>%
  compile(loss = 'binary_crossentropy',
          optimize = optimizer_rmsprop(),
          metrics = c('mse'))

model %>%
  fit(input,
       input,
       epochs = 10,
       batch_size = 256,
       shuffle = TRUE,
       validation_split = 0.2)
```

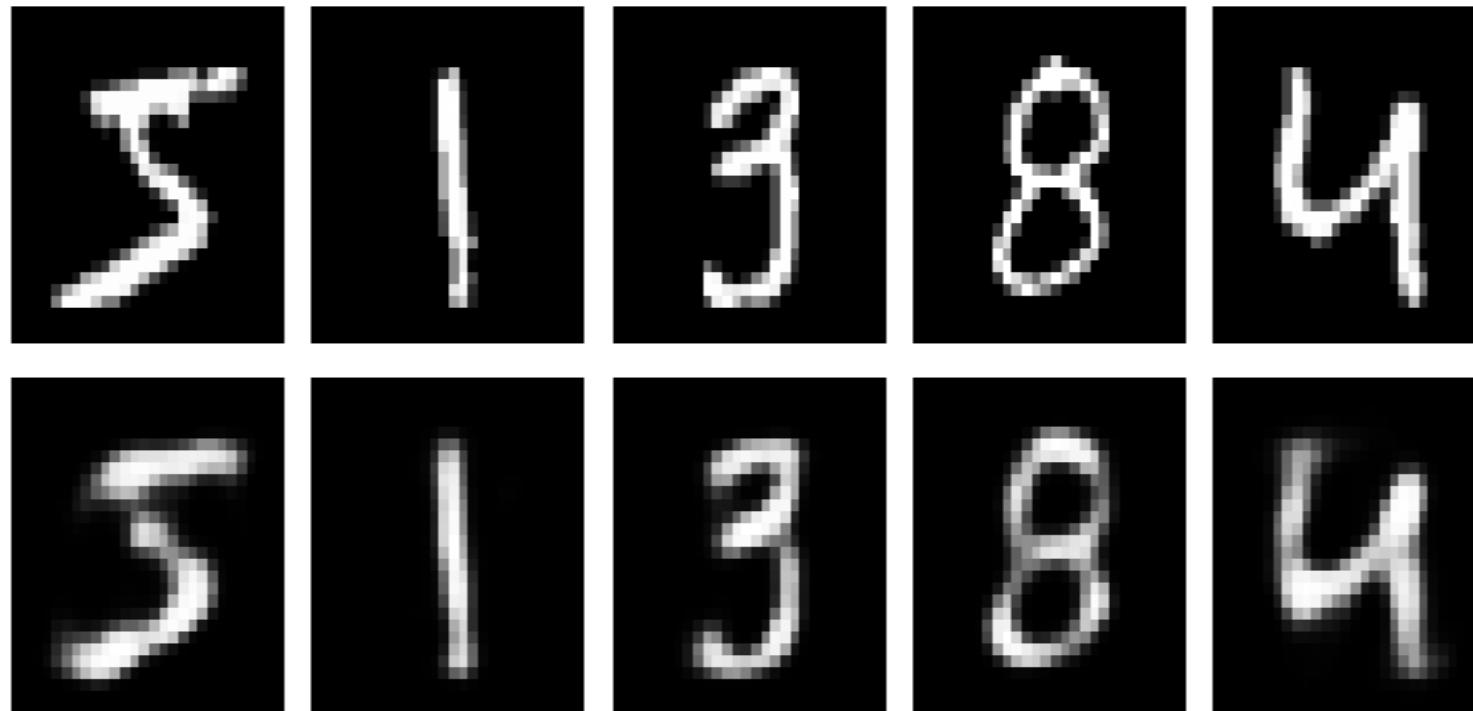
The `input` variable is also passed to the model as the `output` parameter.

The option `shuffle=TRUE` shuffles the training dataset after each epoch, such that the model is trained on different batches.

The big test

Let's compare the input and output of our auto encoder.

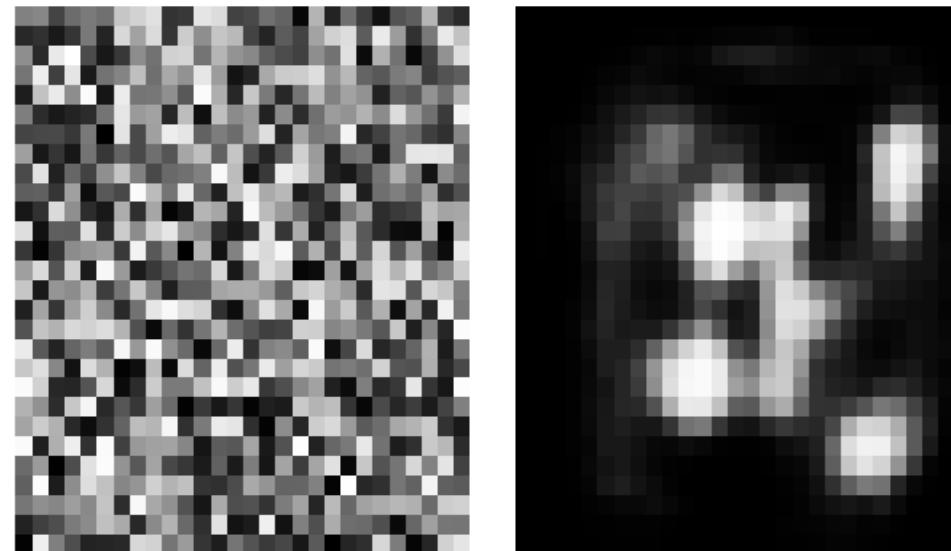
```
result ← predict(model, input[1, , drop = FALSE])
plot_image(input[1, ]) # the original image
plot_image(result[1, ]) # the reconstruction of the model
```



What happens with random noise?

```
random ← matrix(runif(28^2), nrow = 1)

gridExtra::grid.arrange(
  plot_image(random[1, ]) + theme(legend.position = 'none'),
  plot_image(predict(model, random)[1, ]) + theme(legend.position = 'none'),
  nrow = 1)
```



Convolutional neural networks (CNN)

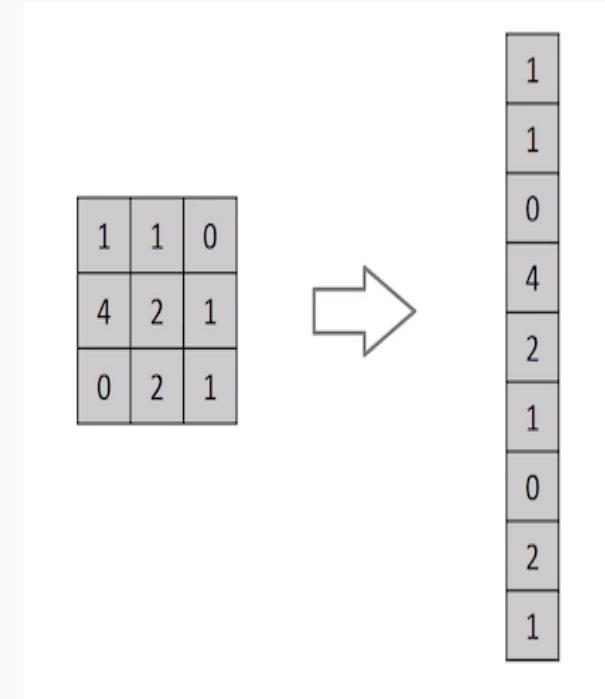
Convolutional neural networks (CNN)

So far, the first step in our analysis was to flatten the image matrix into a vector.

```
input ← tensorflow::array_reshape(  
  input, c(nrow(input), 28*28)) / 255
```

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 dimensional data, **without** flattening.

Convolutional layers

Classical hidden layers use **1 dimensional inputs** to construct **1 dimensional features**.

2d convolutional layers use **2 dimensional input** (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.

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Input

1	0	1
0	1	0
1	0	1

Filter / Kernel

We slide the kernel over the input image and compute matrix multiplications between the selected part of the image and the kernel.

1x1	1x0	1x1	0	0
0x0	1x1	1x0	1	0
0x1	0x0	1x1	1	1
0	0	1	1	0
0	1	1	0	0

Input x Filter

4		

Feature Map

Source: Bradley Boehmke

Convolutional layers (cont.)

Classical hidden layers use **1 dimensional inputs** to construct **1 dimensional features**.

2d convolutional layers use **2 dimensional input** (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.

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Input

1	0	1
0	1	0
1	0	1

Filter / Kernel

We slide the kernel over the input image and compute matrix multiplications between the selected part of the image and the kernel.

1x1	1x0	1x1	0	0
0x0	1x1	1x0	1	0
0x1	0x0	1x1	1	1
0	0	1	1	0
0	1	1	0	0

4		

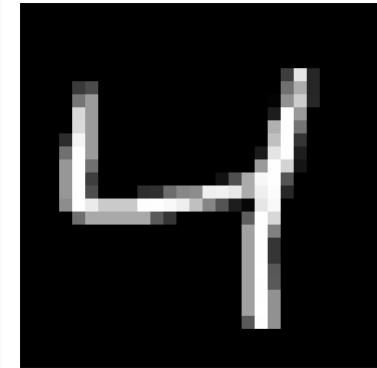
Source: Bradley Boehmke

Convolutional layers (cont.)

2d convolutional layers can **detect** the same, **local feature anywhere** in the image.

A useful feature for classifying the number four is the presence of straight, vertical lines.

Q: How should the kernel look to detect this feature?



Convolutional layers (cont.)

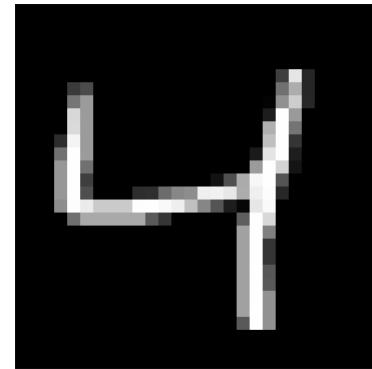
2d convolutional layers can **detect** the same, **local featur anywhere** in the image.

A useful feature for classifying the number four is the presence of straight, **vertical lines**.

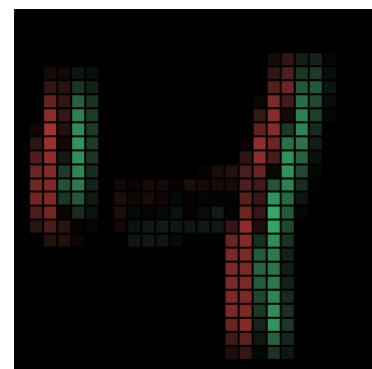
Q: How should the **kernel** look to detect this feature?

1	1	-2
1	1	-2
1	1	-2

original image:



feature map:



Convolutional layers in {keras}

```
keras_model_sequential() %>%  
  layer_conv_2d(filters = 8,  
                 kernel_size = c(3, 3),  
                 strides = c(1, 1),  
                 input_shape = c(28, 28, 1))
```

Add a **2d convolutional layer** with `layer_conv_2d()`:

- `filters = 8`:

We construct **8 feature maps** associated to different kernels/**filters**.

- `kernel_size = c(3, 3)`:

The filter/**kernel** has a size of **3x3**.

- `strides = c(1, 1)`:

We **move** the **kernel** in steps of **1** pixel in both the horizontal and vertical direction. This is a common choice.

- `input_shape = c(28, 28, 1)`:

If this is the first layer of the model, we also have to specify the dimensions of the input data. The input consists of **1 image** of size **28x28**.

Pooling

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

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

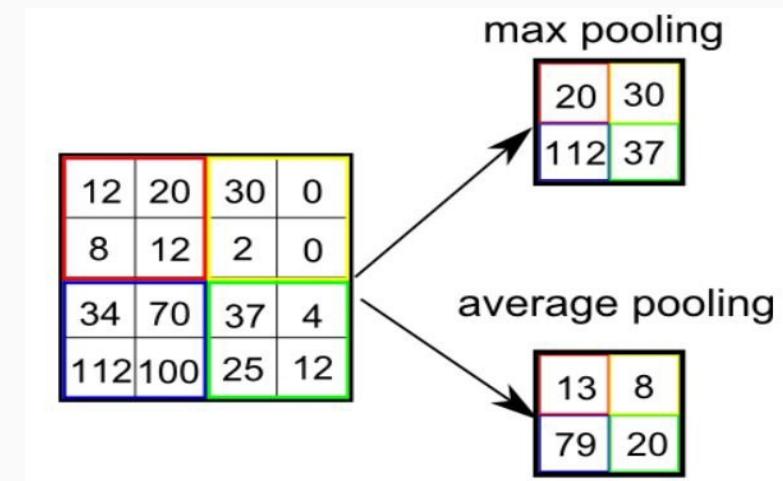
Two common operations are:

- 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.

Flatten

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

Finally, when all local features are extracted the data is **flattened**.

A classical neural network (as built in the first part of these course) analyzes these local features.



Your turn

Fit and **evaluate** a **convolutional neural network** on the MNIST dataset.

Before you start, you should restructure the input and test dataset in the required format.

```
# We start from the original data  
input_conv ← mnist$train$x / 255  
  
# We add a fourth dimension, such that one data point has size (28×28×1)  
input_conv ← k_expand_dims(input_conv, axis = 4)
```

Once you are satisfied with the performance of your model, you can plot some images that were misclassified by your model.

```
model ← keras_model_sequential() %>%
  layer_conv_2d(filters = 8,
                kernel_size = 3,
                input_shape = c(28, 28, 1)) %>%
  layer_max_pooling_2d(pool_size = 2) %>%
  layer_flatten() %>%
  layer_dense(units = 10,
              activation = 'softmax') %>%
  compile(loss = 'categorical_crossentropy',
          optimize = optimizer_rmsprop(),
          metrics = c('accuracy'))

model %>% fit(input_conv, output,
                 epochs = 10,
                 batch_size = 128,
                 validation_split = 0.2)

model %>%
  evaluate(test_input_conv,
           test_output, verbose = 0)
```

```
##      loss accuracy
## 0.1224037 0.9659000
```

Model accuracy has increased significantly and improves further with more epochs.

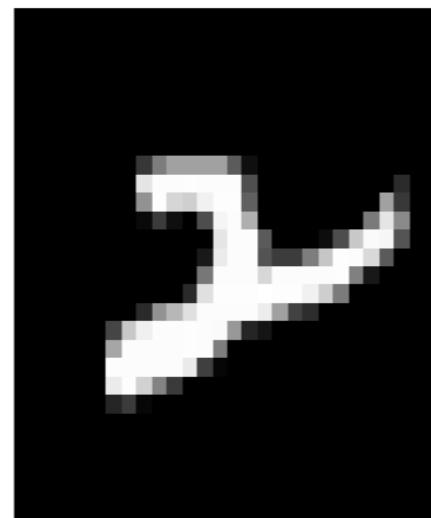
Misclassifications

```
prediction ← model %>% predict(test_input_conv)
category ← apply(prediction, 1, which.max)-1
actual_category ← apply(test_output, 1, which.max)-1
head(which(actual_category ≠ category))
```

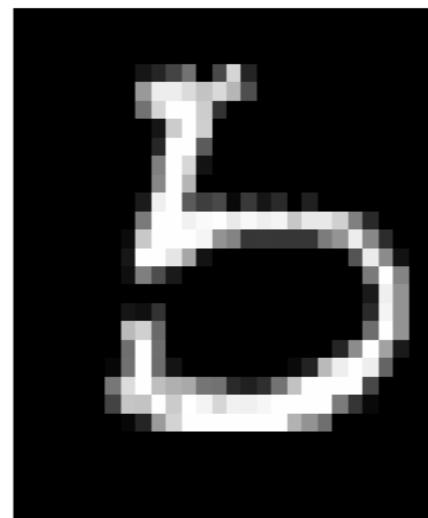
```
## [1] 9 19 93 248 260 319
```

```
plot_image(test_input[93, ])
```

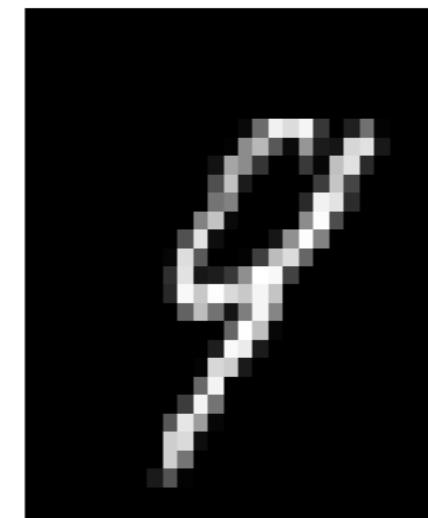
actual: 2 predicted: 8



actual: 5 predicted: 6

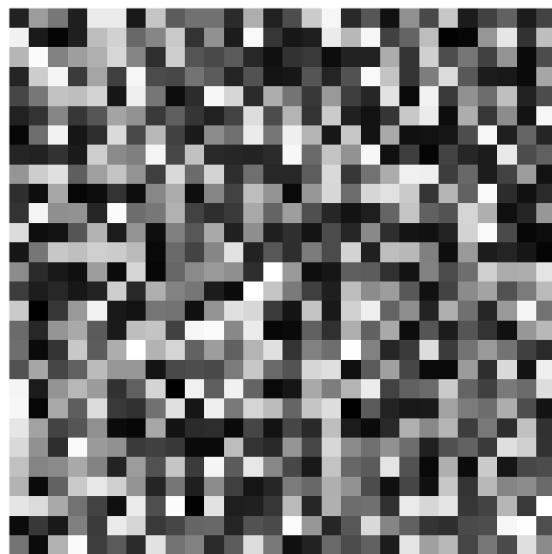


actual: 9 predicted: 4



What happens to random data?

```
random ← runif(28*28)
random_conv ← matrix(random, nrow = 28, ncol = 28)
random_conv ← k_expand_dims(random_conv, axis = 1)
random_conv ← k_expand_dims(random_conv, axis = 4)
plot_image(random)
```



There is not a single doubt, this has to be a nine!

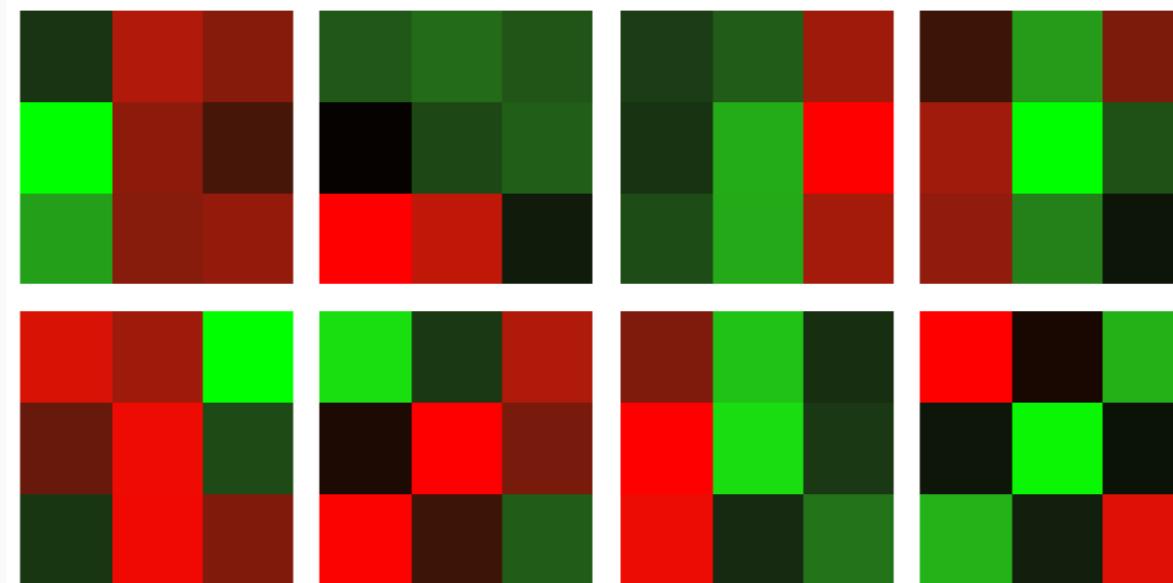
```
predict(model, random_conv)
## [,1] [,2] [,3]
## [1,] 7.361005e-15 2.655454e-30 2.406724e-12 2.888002
```

Actually almost all random images will be classified as a nine.

Inspecting the filter/kernel

The third filter/kernel is the one we inferred for detecting vertical lines. Can you find an interpretation for the other kernels?

```
weights ← purrr::map(1:8, function(x) {plot_image(as.numeric(model$get_weights()[[1]][,,,x]), FALSE)})  
weights[['nrow']] ← 2  
do.call(gridExtra::grid.arrange, weights)
```





Your turn

We have now built **convolutional neural** networks using **layer_conv_2d()**.
In addition {keras} defines **layer_conv_1d()** and **layer_conv_3d()**.

Q: For which data would you use **layer_conv_1d()**?

Q: Which function would you use to build a convolutional network for:

- Colored images
- Movie data

Thanks!



Slides created with the R package `xaringan`.

Course material available via

 <https://github.com/katrienantonio/hands-on-machine-learning-R-module-3>