# Machine learning in R - Day 3

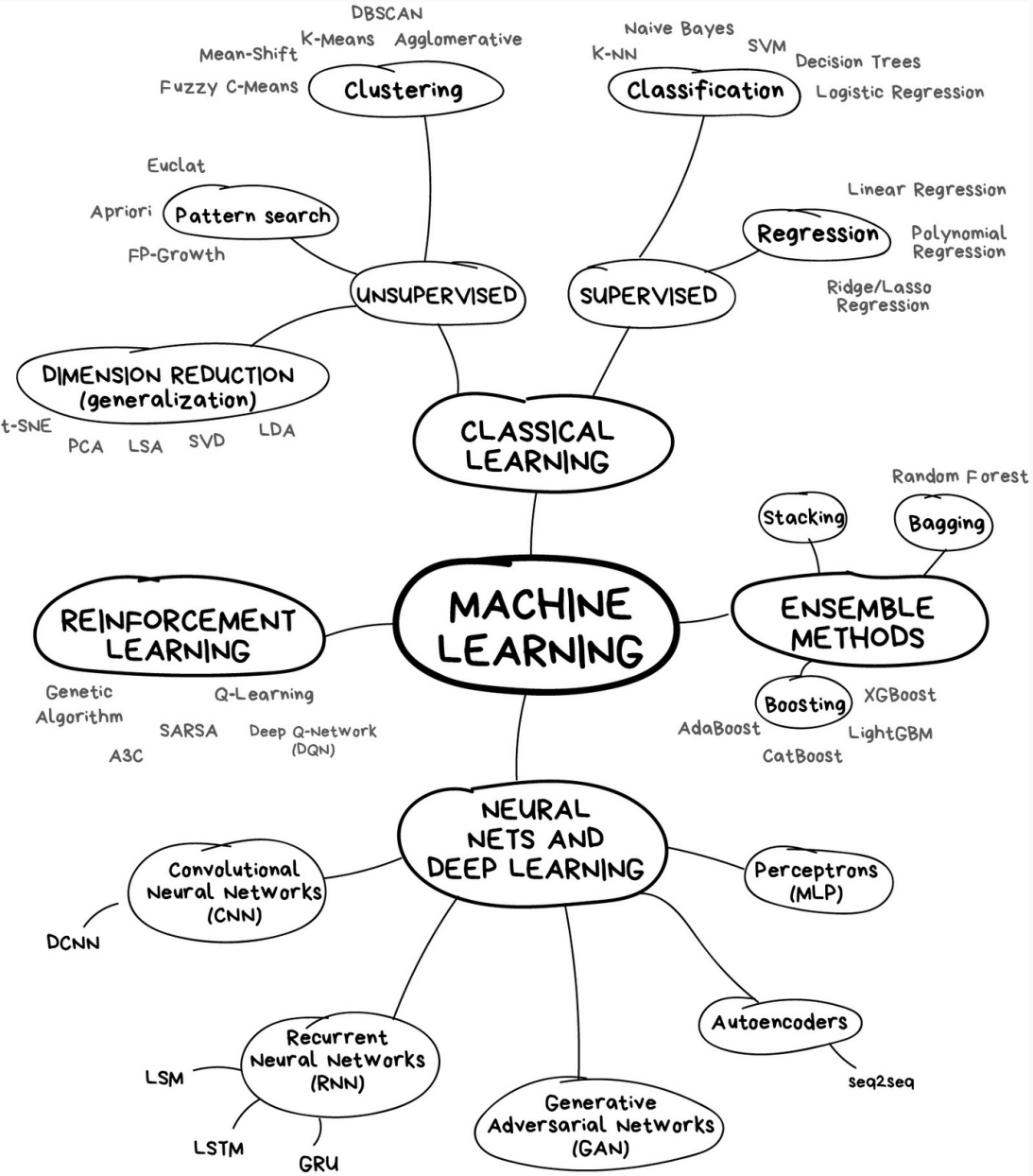
## Hands-on workshop at Nationale Nederlanden

Katrien Antonio, Jonas Crevecoeur and Roel Henckaerts

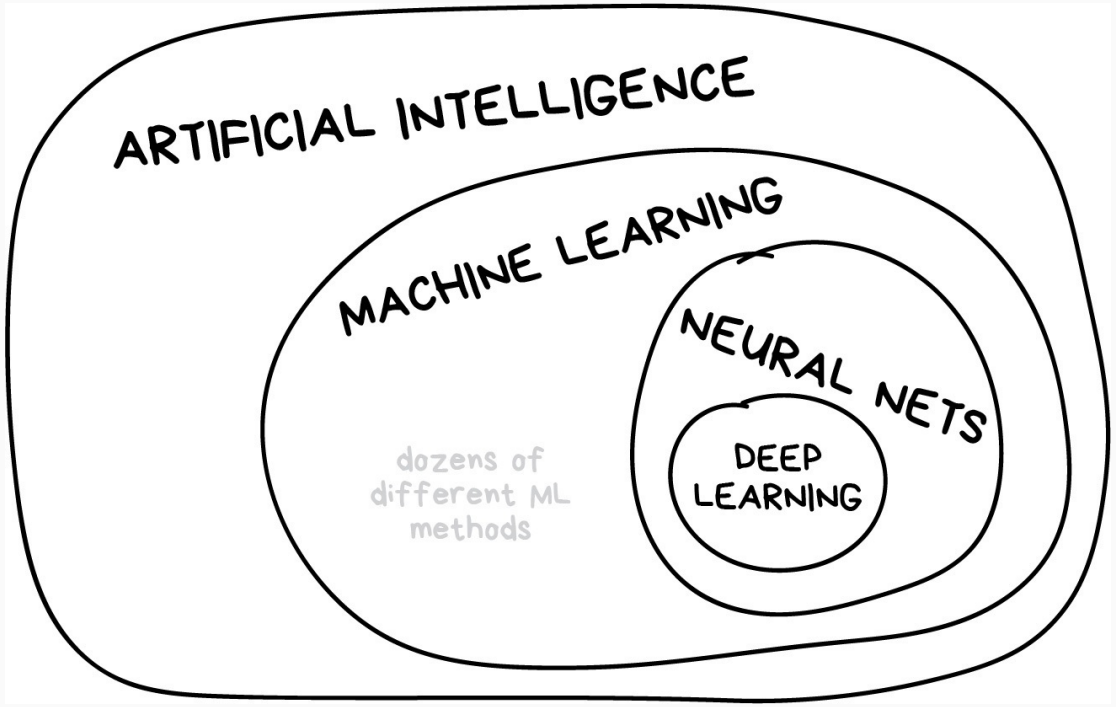
NN ML workshop | February 11-13, 2020

# Today's Outline

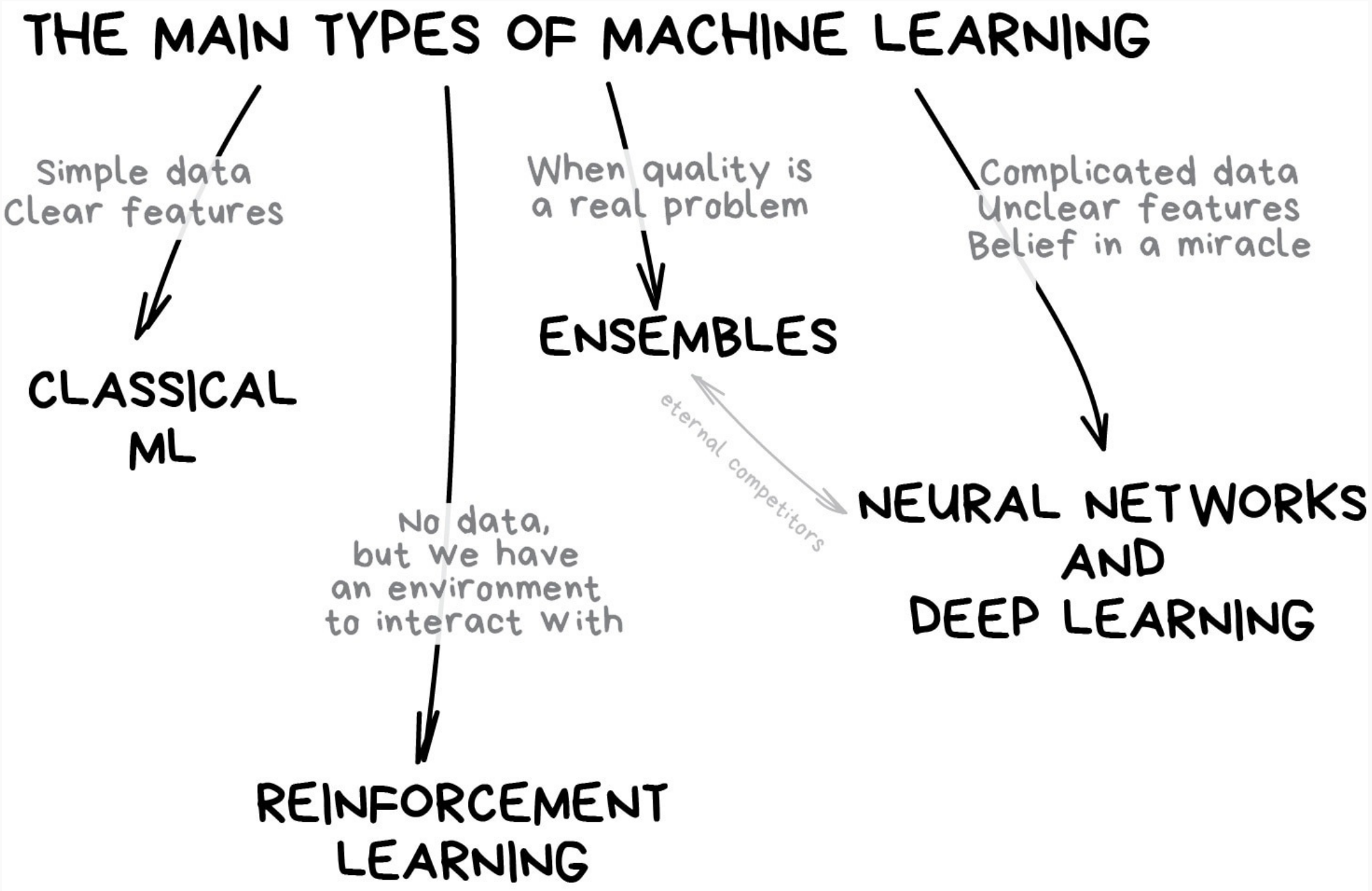
- Introduction
  - Unpacking our toolbox
  - Tensors
- Neural network fundamentals
  - Essential concepts
  - Model building and training in {keras}
  - Building your first ANN
  - Evaluating your model
  - Interpretation tools
- Auto encoders
  - Data compression and feature extraction
  - Evaluation
- Convolutional neural networks
  - Handling new data formats
  - Convolutional layers explained
  - Evaluation and intepretation
- Regression with neural networks
  - Redefining GLMs as a neural network
  - Including exposure
  - Case study




## Some roadmaps to explore the ML landscape...




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


THE MAIN TYPES OF MACHINE LEARNING
Simple data
Clear features
CLASSICAL ML
No data,
but we have
an environment
to interact with
REINFORCEMENT LEARNING
When quality is
a real problem
ENSEMBLES
eternal competitors
Complicated data
Unclear features
Belief in a miracle
NEURAL NETWORKS AND DEEP LEARNING

# Introduction

# Getting started

Download the GitHub repository for this course and open the Rproject file Neural networks.Rproj.

Today's session will make extensive use of {keras} and {tidyverse}.
Don't forget to load these packages in your R session.

```
require(keras)
require(tidyverse)
```

# The programming framework for today

- Tensorflow:
  Open source platform for machine learning developped by google.
- Keras:
  An inuitive high level interface for Tensorflow.
- R:
  Our favorite programming language.

# Why is it called tensorflow?

Untill now, all the **input** features of our model were individual data points, i.e. **zero dimensional**:

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

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

- time series: 1-dimensional,
- sound fragment: 2-dimensional,
- image in color: 3-dimensional,
- movie: 4-dimensional,
- ...

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

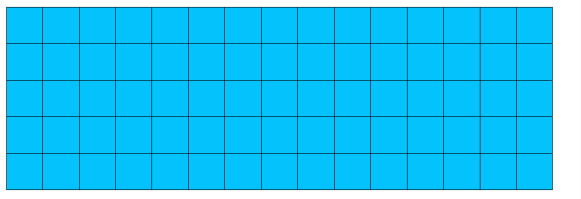
# Why is it called tensorflow? (cont.)

**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

- A 2-dimensional tensor is a matrix



- ...

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.

# 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),
                shape = c(2,2))
x
```

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

```
k_mean(x, axis = 2)
```

```
## tf.Tensor([1.5 3.5], shape=(2,), dtype=float32)
```

# Your turn

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

- Create a 3-dimensional tensor in R with values `1,2,...,8` and shape `(2, 2, 2)`.
- Calculate the logarithm of this tensor.
- Calculate the mean of this tensor over the third axis.

```
require(keras)
x <- k_constant(1:8, shape = c(2,2, 2))
k_log(x)
```

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

```
k_mean(x, axis = 3)
```

```
## tf.Tensor(
## [[1.5 3.5]
##  [5.5 7.5]], shape=(2, 2), dtype=float32)
```

`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:

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

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

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

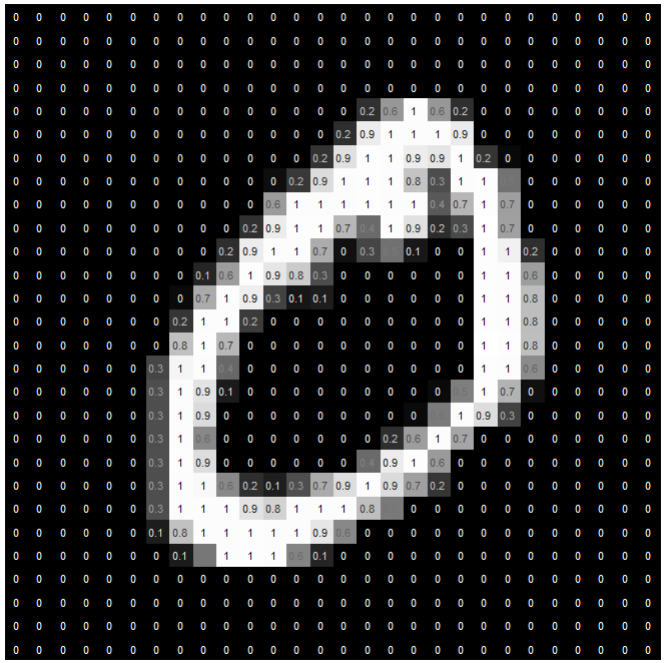
```
## Error in py_call_impl(callable, dots$args, dots$keyw
```

# Neural network fundamentals

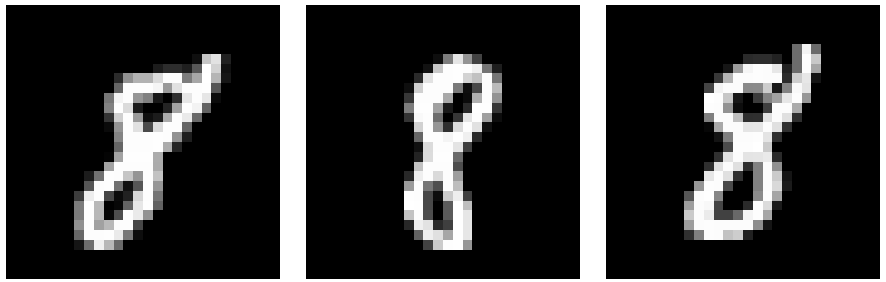
# The MNIST dataset

We will use the MNIST dataset, which contains 70.000 labeled images of handwritten digits.

Each image is stored as a 28x28 intensity matrix.



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 (intensity matrix) and the output (0-9) is complicated.

# Loading the MNIST dataset

`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 <- 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:

```
dim(input)
```

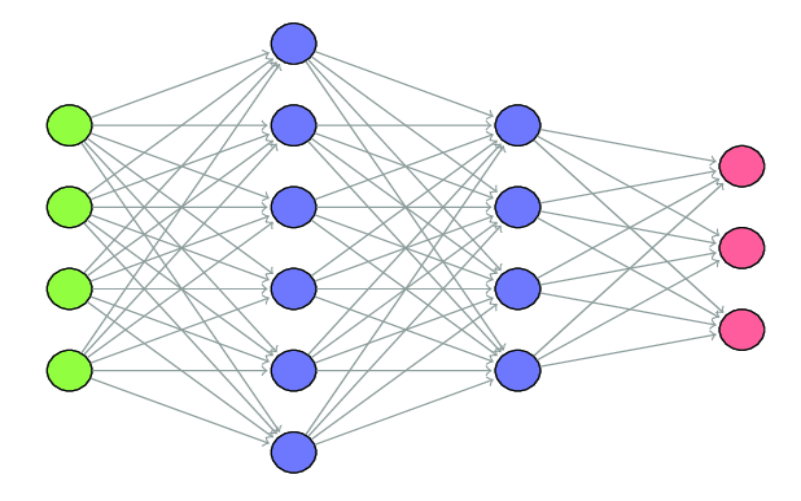
```
## [1] 60000    28    28
```

Select the first image:

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

# Neural networks

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

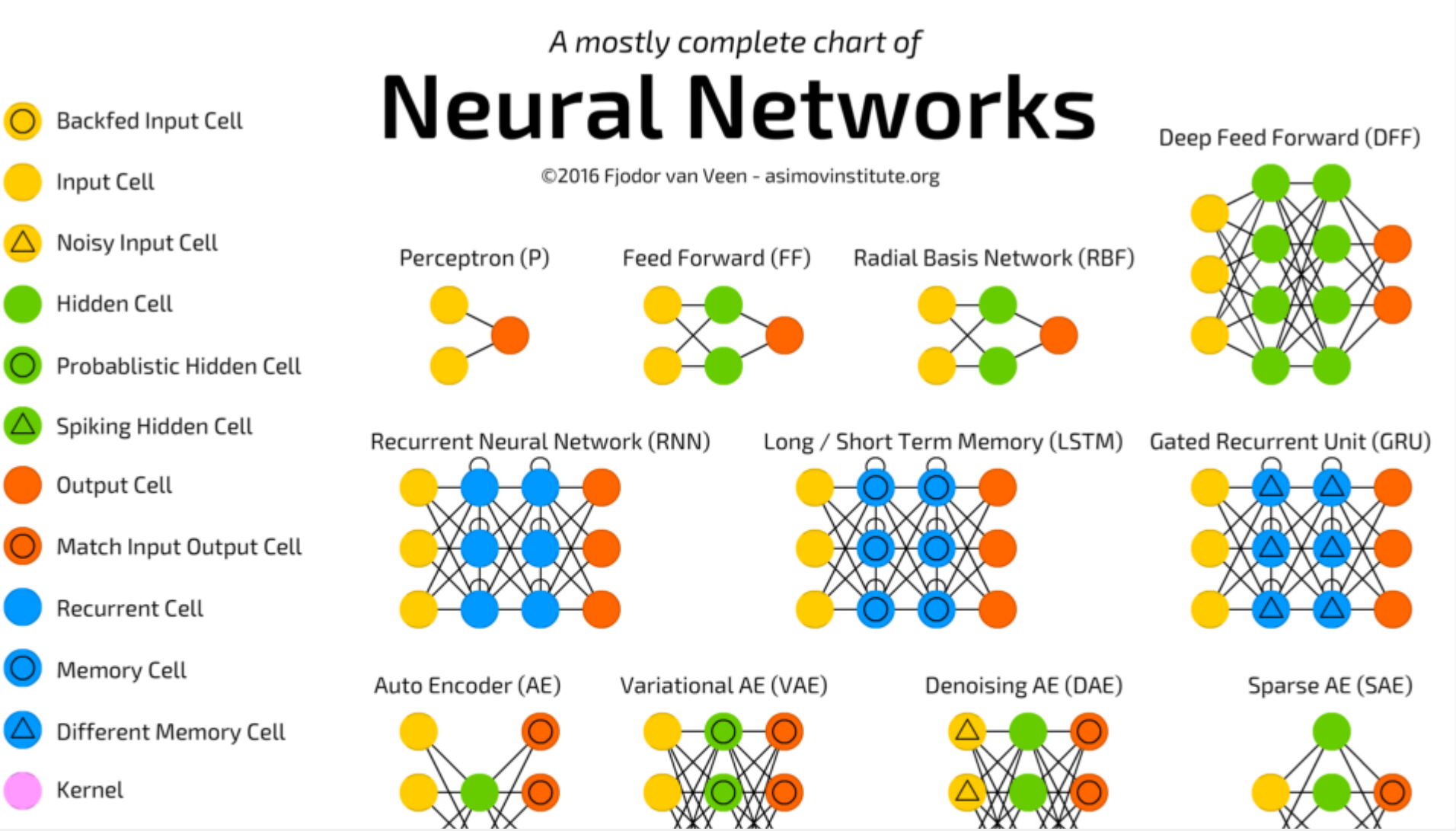


This sequential, layer structure is at the core of the Keras libary.

```
model <-
  keras_model_sequential() %>%
  layer_dense(...) %>%
  layer_dense(...)
```

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

# Meet the neural networks family

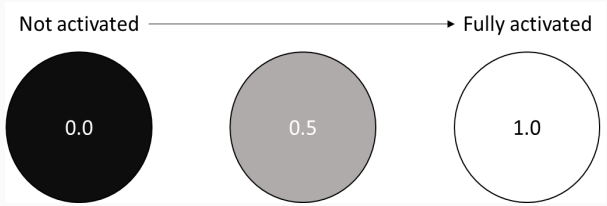


# Nodes

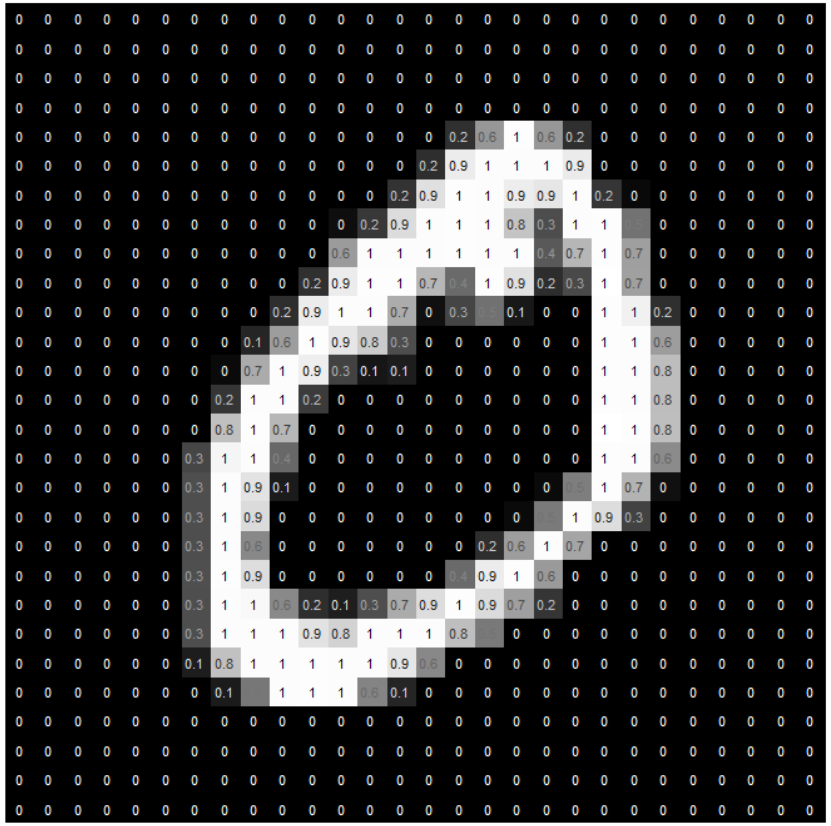
Nodes (called neurons) contain a numeric value.

In this course we will mostly consider nodes with values between zero and one.

- zero: the feature is not present in the data
- one: the feature is present in the data



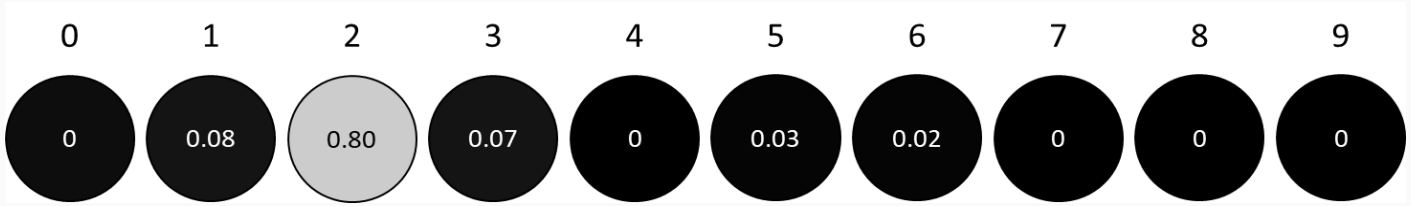

In our example, the input layer consists of 784 (=28 x 28) nodes.



# The output layer

The output layer in our example will consist of 10 nodes (1 per digit).

The values of these nodes will sum to one, such that we can interpret the outcomes as **probabilities**.




**Interpretation**: The model assigns a probability of 80% to the image representing a two.

# Your turn

We have proposed a structure with 10 nodes for the output layer.

**Q**: We list some other structures for encoding the output variable. What do you think are the **advantages** and **disadvantages** of each representation?

- Continuous encoding in 1 node.
- Binary encoding in 4 nodes, i.e. 6 = 0110.
- 9 nodes for levels 1-9 with zero as the default category.

# Hidden layer

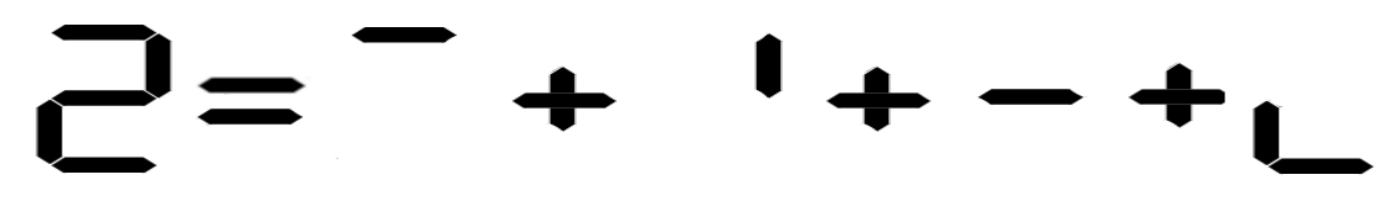
All layers between the input and output layer are called **hidden layers**.

Nodes in the hidden layer(s) represent intermediary features that we don't explicitely 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.



# Hidden layer (cont.)

```
model <-
  keras_model_sequential() %>%
  layer_dense(units = 16,
              input_shape = c(784)) %>%
  layer_dense(units = 10)
```

- `layer_dense`:

Add a fully connected hidden layer to the neural network, i.e. there is a connection between every node in this layer and the previous.

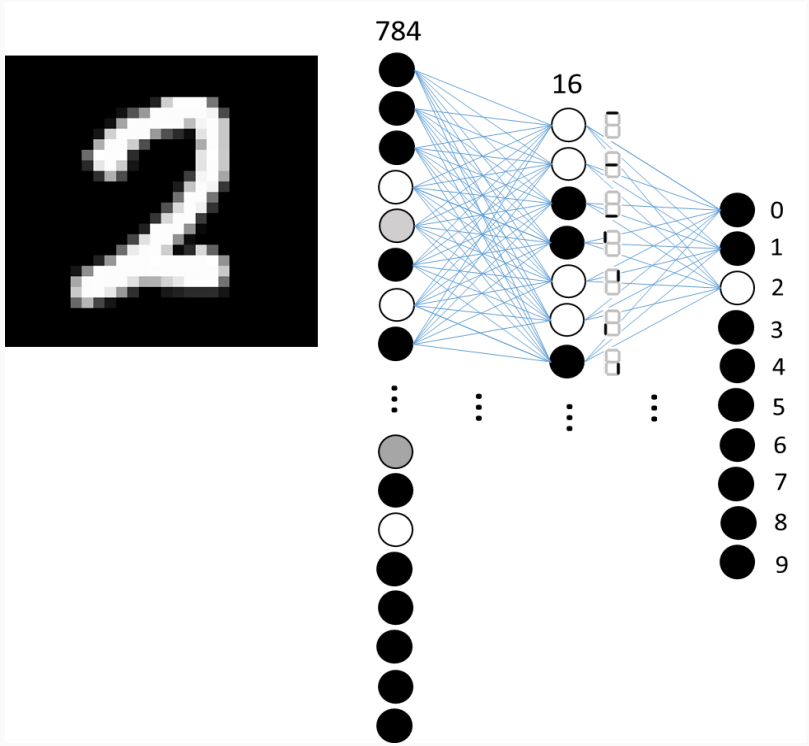
- `input_shape = c(784)`:

Define the dimension of the input tensor.
We flatten the image into a vector of 784 nodes.

- `units = 16`:

Set the number of nodes in this hidden layer.

# Connecting the dots




A pattern of connections in the first layer will activate certain features in the hidden layer.

These features in the hidden layer will in turn activate cells in the output layer.

The output cell with the highest value is our predicted outcome.

**Q**: How is the value of cells in the hidden and output layer determined?

# Connecting the dots (cont.)

The model assigns a **weight** to each connection.

**First** the model computes the **linear combination**:

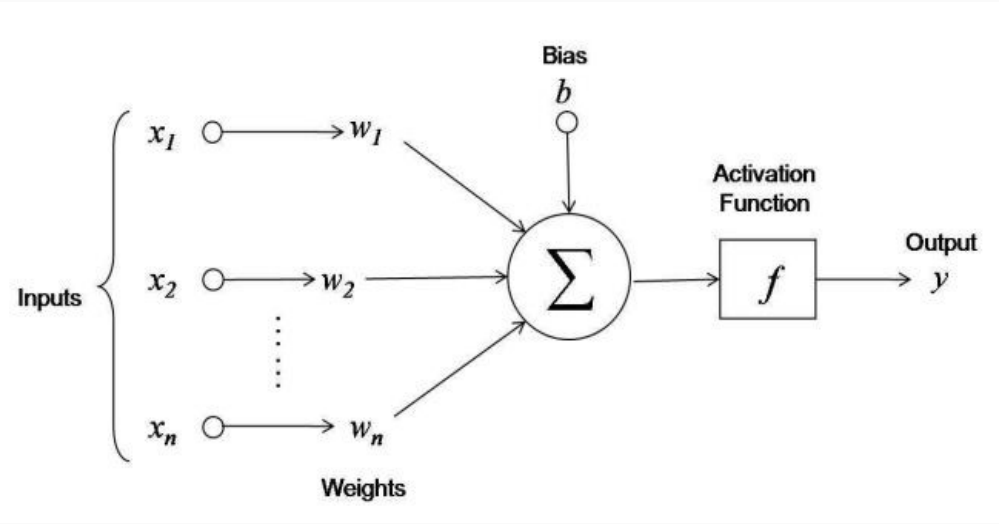
$$z = b + w_1 x_1 + w_2 x_2 + \ldots w_n x_n = b + w \cdot x$$

An extra bias term is added, which can be seen as a node in the previous layer that is always activated.

**Second** an **activation function** is applied:

$$y = f(b + w \cdot x)$$

The activation function adds non-linearity in the model. Without the activation function the model would be identical to linear regression.




Source: Arthur Arnx

# Connecting the dots (cont.)

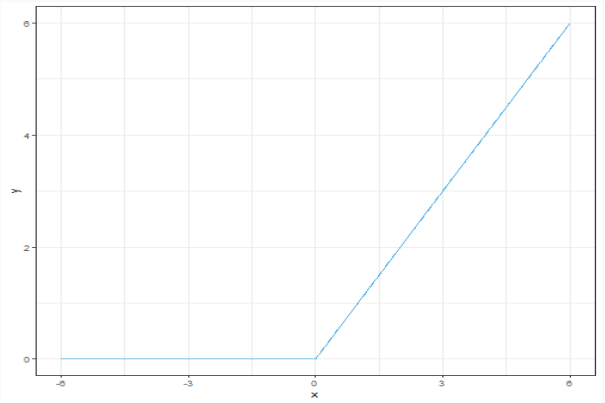
There are many popular choices for the activation function $f(.)$

$$y = f(z) = f(b + w \cdot x)$$

Many activation functions are available:

- **ReLU**

- Very popular way to introduce non-linearity.
- Simple derivative (fast calibration).
- Not used in the output layer.



$$f(x) = \max(x, 0)$$

# Connecting the dots (cont.)

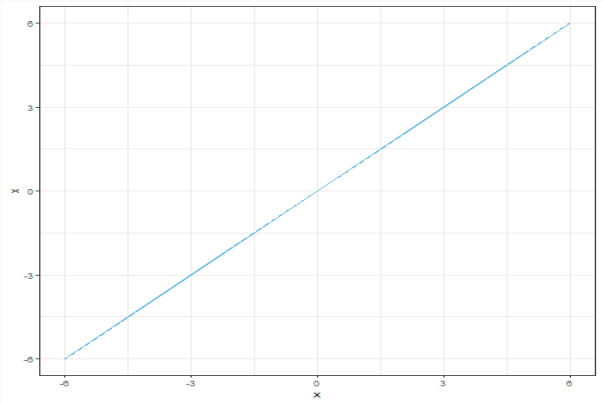
There are many popular choices for the activation function $f(.)$

$$y = f(z) = f(b + w \cdot x)$$

Many activation functions are available:

- ReLU
- **identity: linear regression**

- Only used in the output layer.



$$f(x) = x$$

# Connecting the dots (cont.)

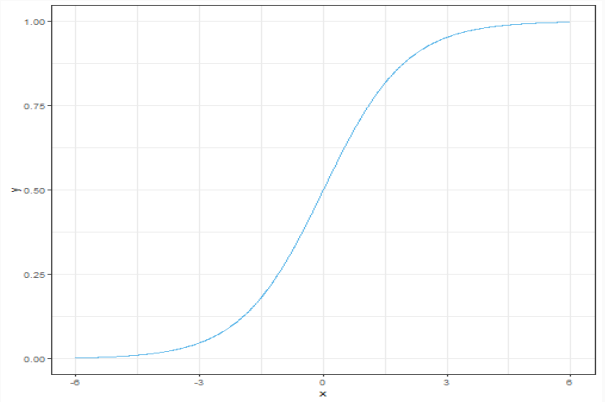
There are many popular choices for the activation function $f(.)$

$$y = f(z) = f(b + w \cdot x)$$

Many activation functions are available:

- ReLU
- identity: linear regression
- **sigmoid: binary classification**

- Transform $z$ to $(0, 1)$.
- Focus on values around zero.




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

# Connecting the dots (cont.)

There are many popular choices for the activation function $f(.)$

$$y = f(z) = f(b + w \cdot x)$$

Many activation functions are available:

- ReLU
- identity: linear regression
- sigmoid: binary classification
- **softmax: multi-class classification**

- Normalize the sum of the nodes in a layer to one.
- Depends on all values $z$ of the previous layer.
- Only used in the output layer.

$$y_k = \frac{e^{z_k}}{\sum_j e^{z_j}}.$$

# Connecting the dots (cont.)

There are many popular choices for the activation function $f(.)$

$$y = f(z) = f(b + w \cdot x)$$

Many activation functions are available:

- ReLU
- identity: linear regression
- sigmoid: binary classification
- softmax: multi-class classification
- **other activation functions**

See the Keras documentation:

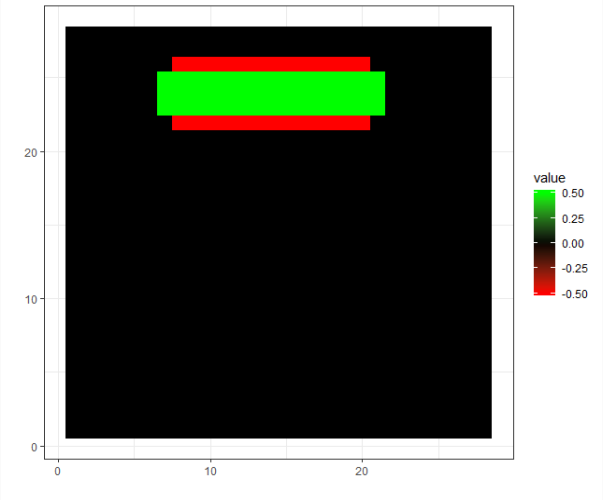
- elu
- selu
- softplus
- softsign
- tanh
- hard_sigmoid
- exponential
- leaky ReLU
- PReLU
- Threshold ReLU

# Connecting the dots (cont.)

How can our network detect a horizontal line at the top of the image?

Each node in the hidden layer is connected with the 784 (28x28) nodes in the input layer.

We can visualize these weights in a (28x28) image:




For the image on the left, the weighted sum is maximal when there is a horizontal line at the top:

$$z = b + W \cdot X.$$

Since there are more positive weights, the weighted sum is mostly positive for randomly generated data. A negative bias reduces the probability of activing a node.

# Defining a neural network in {keras}

```
model <-
  keras_model_sequential()
```

We construct the neural network by sequentially adding layers.

# Defining a neural network in {keras}

```
model <-
  keras_model_sequential() %>%
  layer_dense(units = 16,
              activation = 'sigmoid',
              input_shape = 784)
```

Specify the number of input nodes as a parameter in the first layer of the model.

Add a fully connected, dense layer with 16 nodes.
Many other layer types are available.

This layer uses the sigmoid activation function.

# Defining a neural network in {keras}

```
model <-
  keras_model_sequential() %>%
  layer_dense(units = 16,
              activation = 'sigmoid',
              input_shape = 784) %>%
  layer_dense(units = 10,
              activation = 'softmax')
```

Add the output layer with 10 units.

The `softmax` activation function forces the sum of the outputs to be one.

# Defining a neural network in {keras}

```
model <-
  keras_model_sequential() %>%
  layer_dense(units = 16,
              activation = 'sigmoid',
              input_shape = 784) %>%
  layer_dense(units = 10,
              activation = 'softmax')

  summary(model)
```

The model contains 12730 parameters:

- 784 * 16 weights between layer one and two.
- 16 bias terms in layer two.
- 16 * 10 weights between layer two and three.
- 10 bias terms in layer three.

```
## Model: "sequential"
## _________________________________________________________________
## Layer (type)                 Output Shape              Param #
## =================================================================
## dense (Dense)                (None, 16)                12560
## _________________________________________________________________
## dense_1 (Dense)              (None, 10)                170
## =================================================================
## Total params: 12,730
## Trainable params: 12,730
## Non-trainable params: 0
## _________________________________________________________________
```

# Calibrating the model

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

Selects a loss function, distribution for the outcome.

Keras includes many common losses:

- `"mse"`: gaussian
- `"poisson"`: poisson
- `"binary_crossentropy"`: bernoulli, non-exclusive classes
- `"categorical_crossentropy"`: bernoulli, exclusive classes
- other distributions, see the keras documentation

Or define your own loss function

```
mse <- function(y_true, y_pred) {
  k_mean((y_true - y_pred)^2, axis = 2)
}

model <- model %>%
  compile(
    loss = mse,
    optimizer = optimizer_rmsprop(),
    metrics = c('accuracy'))
```

- `k_mean` is the keras implementations of `mean` that take a tensor as input.
- `axis = 2` calculates the mean over the different output nodes.
- `axis = 1` would have calculated the mean over the different records in the input dataset.

# Calibrating the model (cont.)

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

Neural networks typically have many degrees of freedom. Keras includes several optimizers for finding a (local) minimum of the loss function.

Popular choices are:

- `optimizer_rmsprop()`
- `optimizer_adam()`
- other optimizers, see the keras documentation

All optimizers more or less do gradient descent

$$y_i = y_i - \delta \cdot \frac{\partial \mathcal{L}}{\partial y_i},$$

where $\mathcal{L}$ is the loss function and $y_i$ is one of the many tuning parameters.

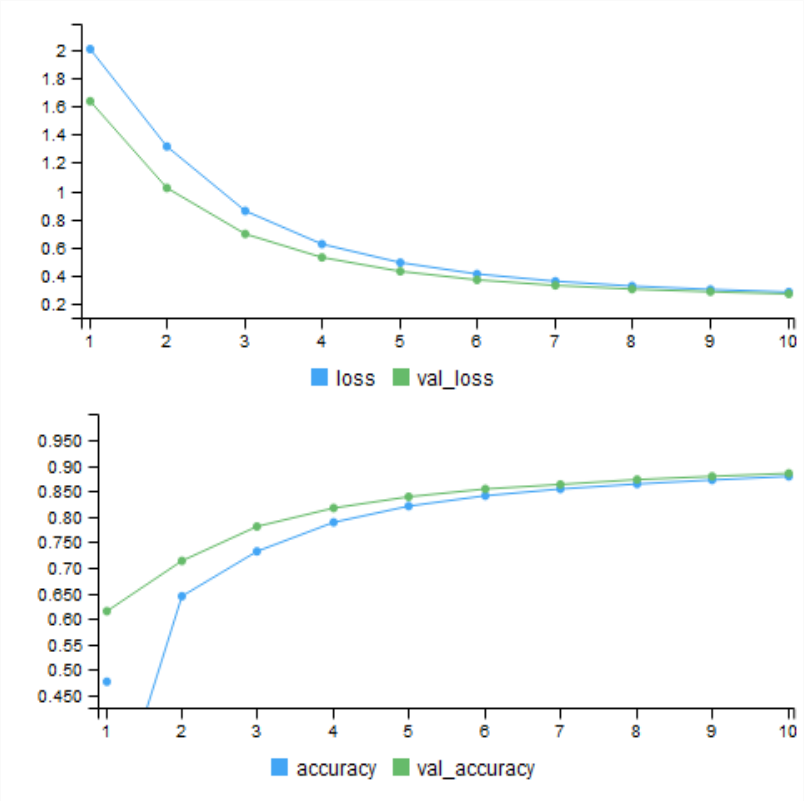
The gradient, $\frac{\partial \mathcal{L}}{\partial y_i}$, is calculated using a technique called backpropagation, i.e. a fancy name for the chain rule from mathematics.

# Calibrating the model (cont.)

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

In addition to the loss function, other performance measures 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



# Preparing the MNIST dataset

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

```
input <- array_reshape(input,
            c(nrow(input), 28*28)) / 255

test_input <- 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 <- to_categorical(output, 10)
test_output <- to_categorical(test_output, 10)
```

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

```
plot_image(input[17, ])
```

# Your turn

As discussed, any loss function can be used as a metric.

In the case of the MNIST dataset, we search for a model with a high accuracy.

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

# Calibrating the model (cont.)

`fit(.)` tunes the model parameters (weights and biasses).
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)
```

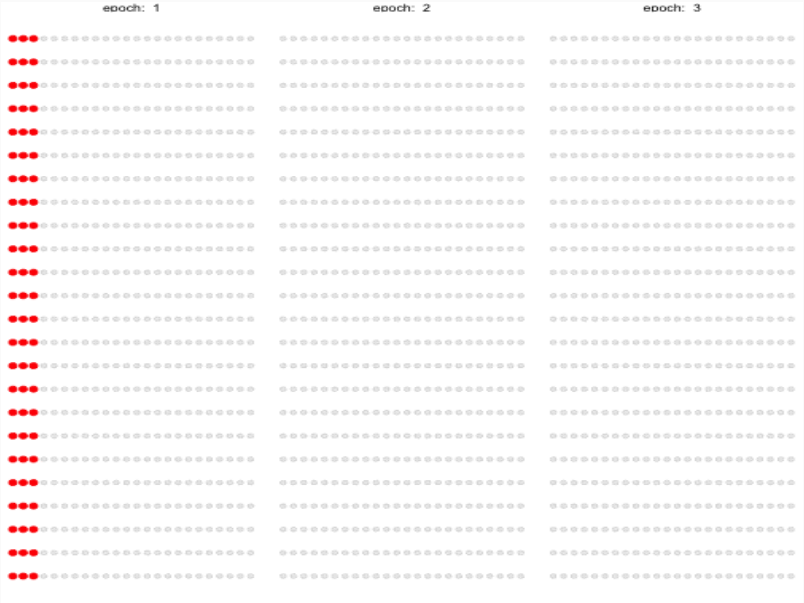
The first arguments are the input data (images) and the corresponding class (0-9).

# Calibrating the model (cont.)

`fit(.)` tunes the model parameters (weights and biasses).
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)
```

Parameter updates are calculated based on small subsets with `batch_size` elements. An `epoch` is one iteration of the algorithm over the full dataset.




Source: Bradley Boehmke, Deep Learning with Keras and TensorFlow in R

# Your turn

You will now **design** and **calibrate** your own neural network for the MNIST dataset. As a form of paralellized model selection, you will all use different model parameters. This way we gain insight into which parameter values work well for this dataset.

**Base model:** Neural network with a single hidden layer with 11-20 nodes.

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

**Adding hidden layers.**
The number of nodes in subsequent layers should decrease.

**Changing the batch size.**

**Changing the activation function.**

# Your turn

## Adding some new layer types:

- `layer_gaussian_noise`: ddds gaussian noise $\mathcal{N}(0, \texttt{stddev})$ 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()
```

# Model evaluation

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

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

```
## $loss
## [1] 0.2336413
##
## $accuracy
## [1] 0.9341
```

`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
```

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
```

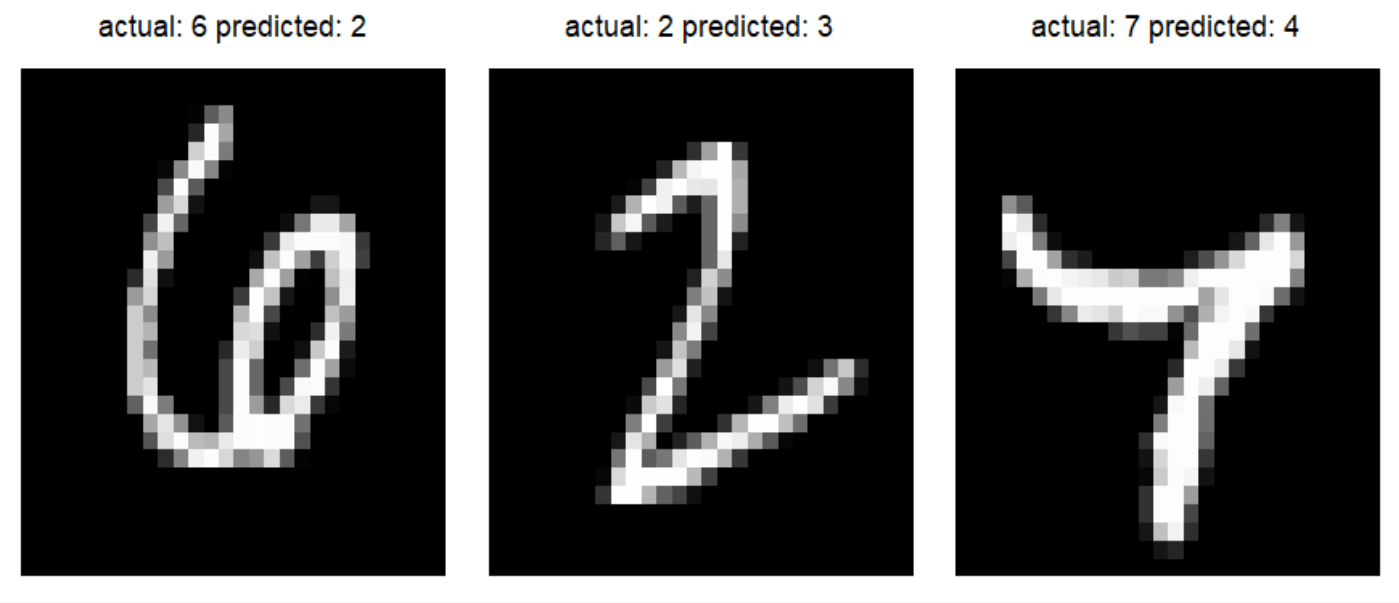
# Model evaluation (cont.)

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

```
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))
```

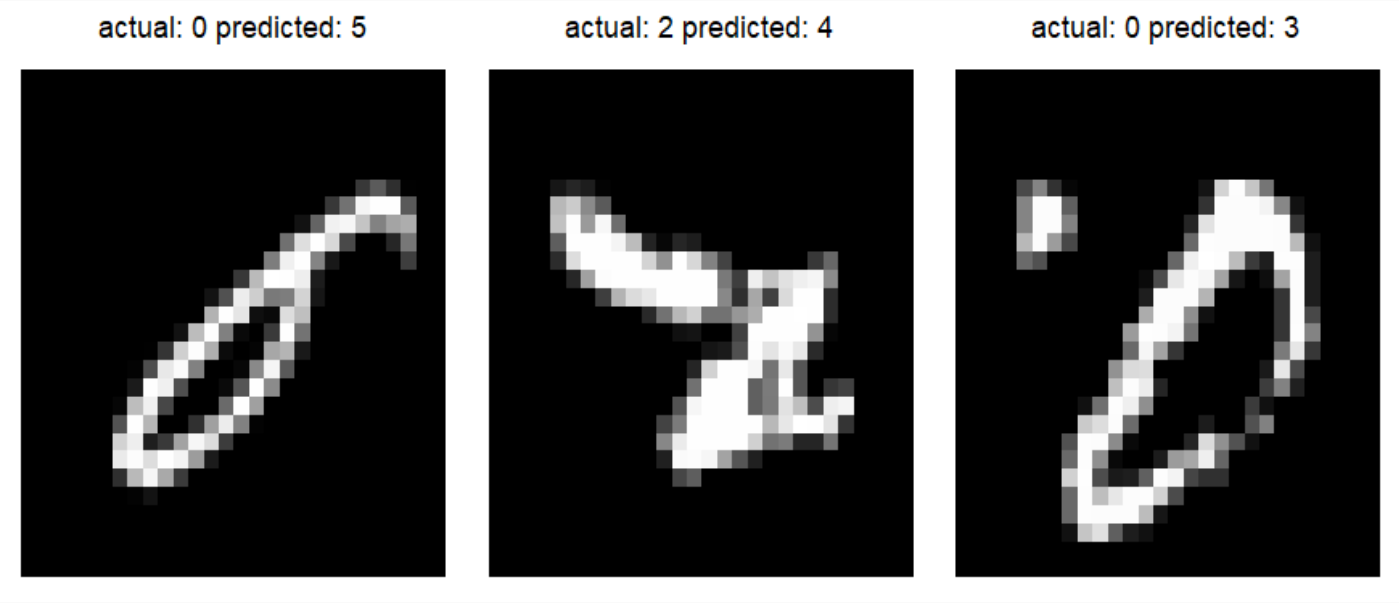


# Model evaluation (cont.)

We inspect the images for which the model asigns 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
```



# Your turn

You will now evaluate your own model!

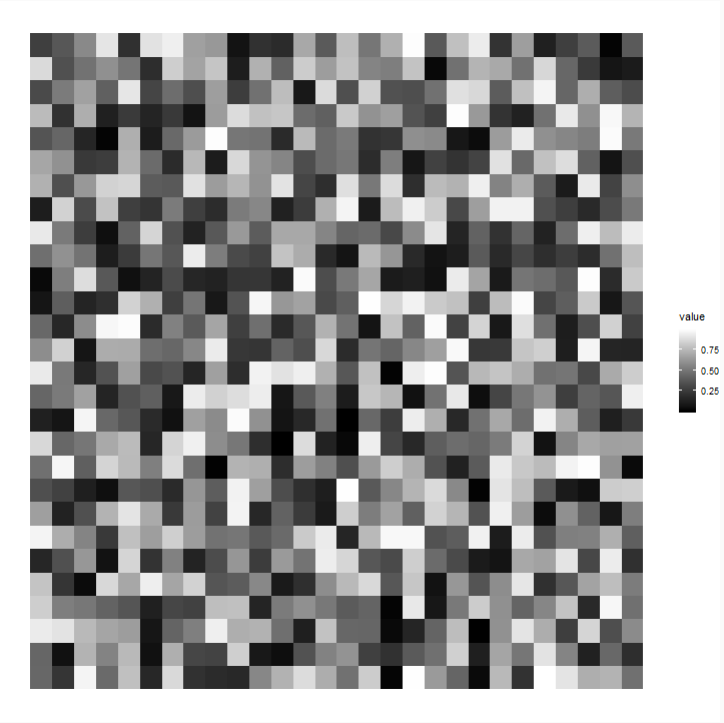
1. Calculate the accuracy of your model on the test set.

2. Visualize some of the misclassified images from your model.

3. Generate an image consisting of random noise and let the model classify this image. What do you think of the results?
   The 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
```

The model is pretty sure that the input on the left is a three!

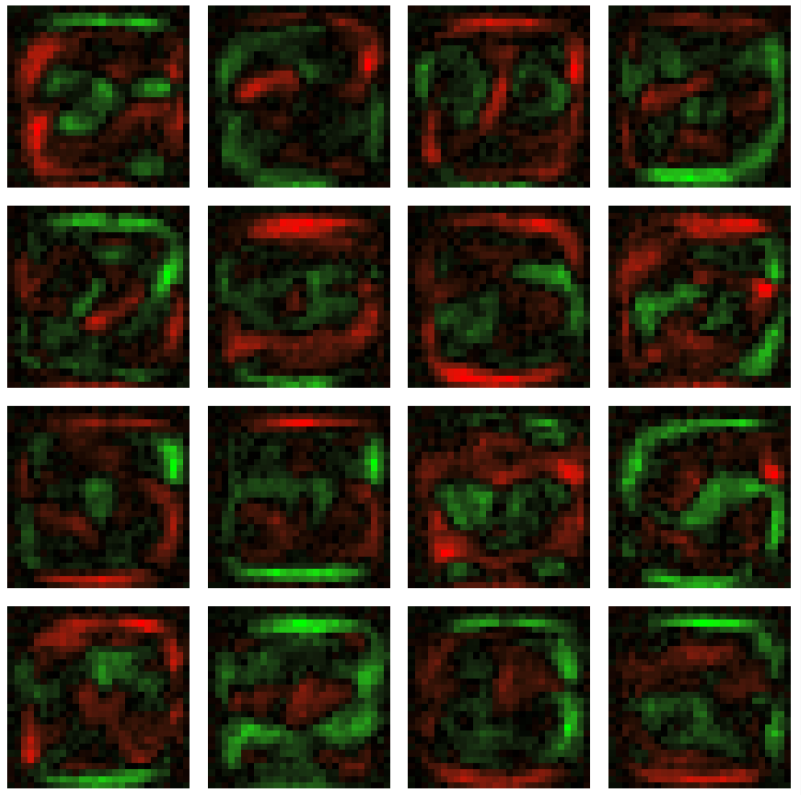
# Model understanding

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$weights[[2*(layer-1) + 1]][, node]

plot_image(as.numeric(weights))
```

Visualization of the calibrated weights for the 16 nodes in the first hidden layer.

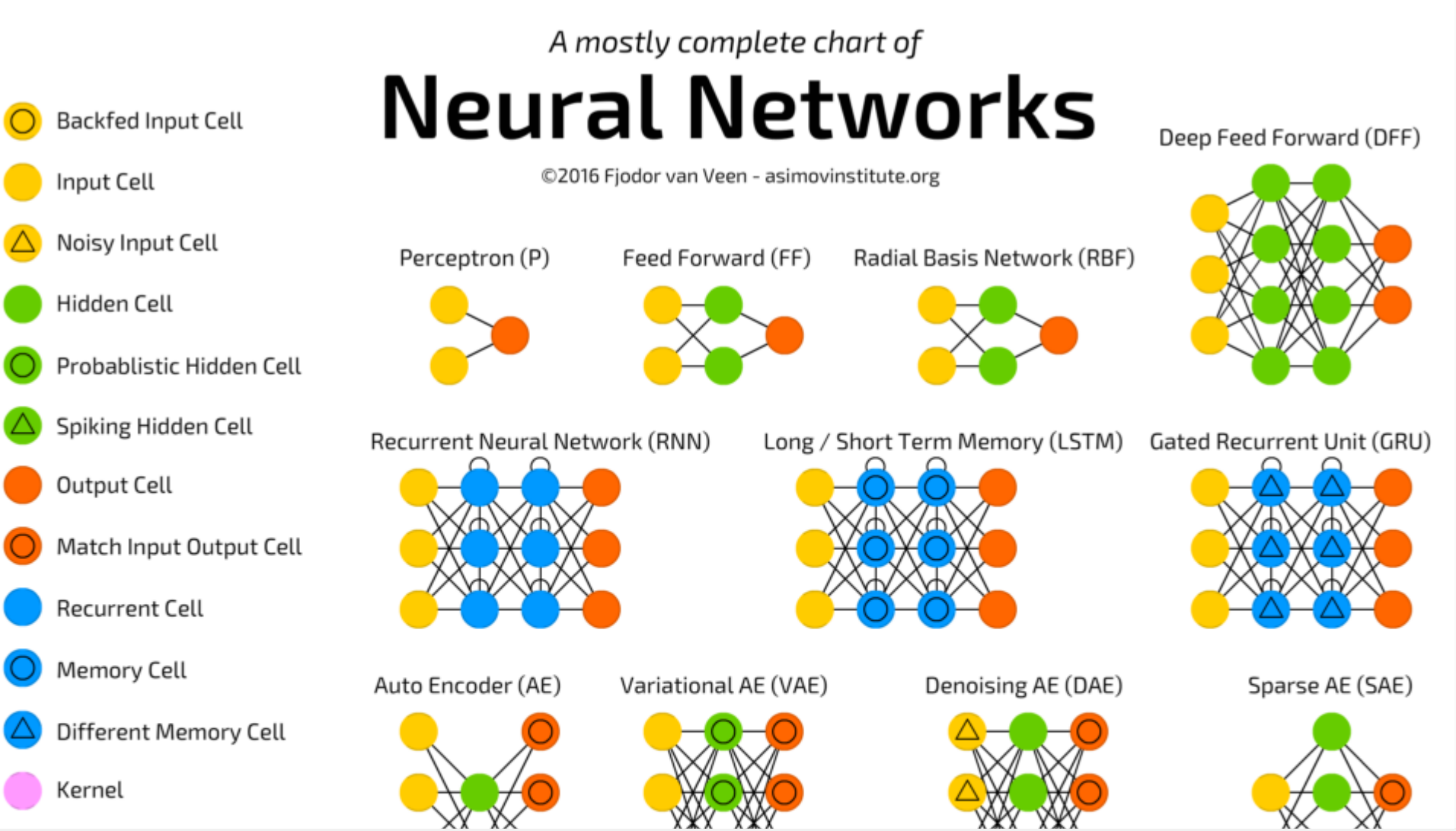


# Summary(fundamentals)

- Define neural networks **sequentially** in {keras}
  `keras_model_sequential`
- Layers consist of **nodes** and **connections**
- The vanilla choice is a **fully connected layer**
  `layer_dense`

- In building a neural network we can **tune**:
  - 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

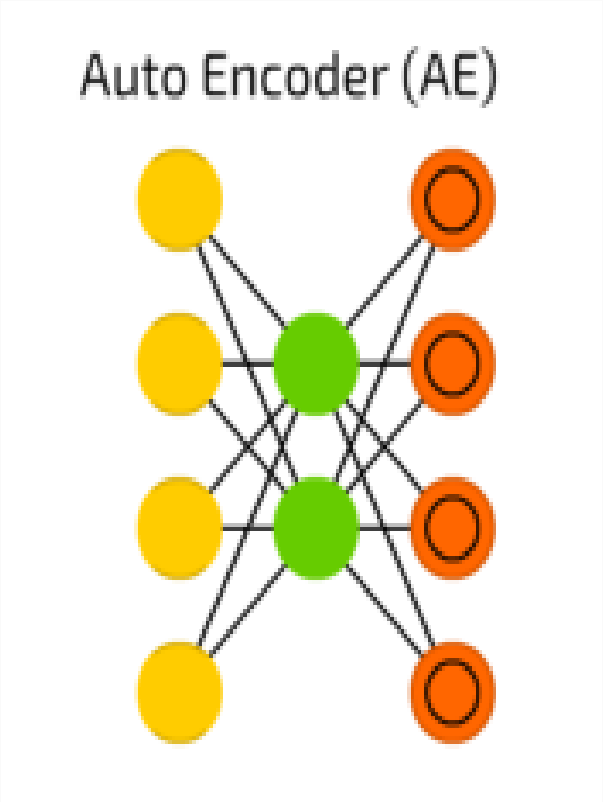
# Meet the neural networks family



# 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 functions 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.
- Expirement 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)
```

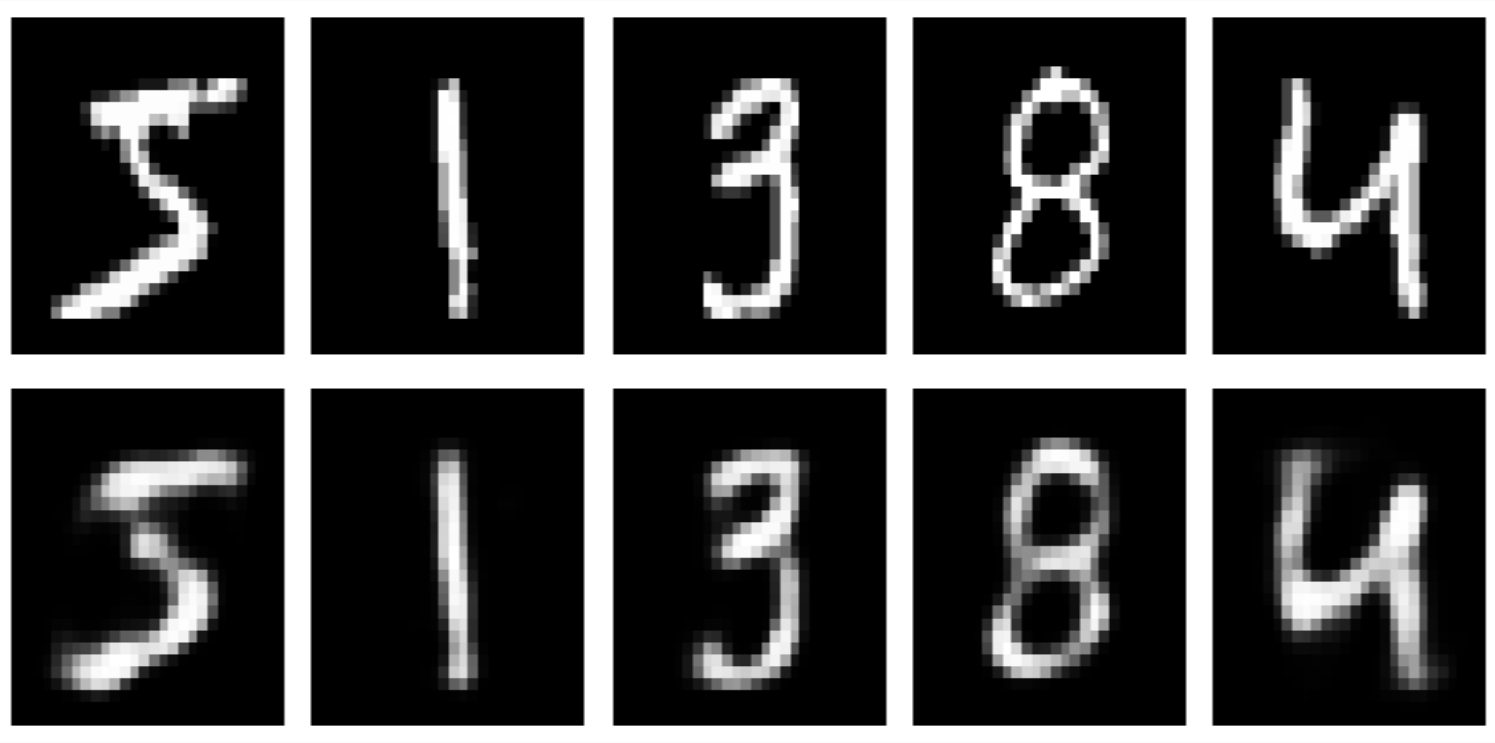
The `input` variable is also passed to the model as the `output` parameter.

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

# The big test

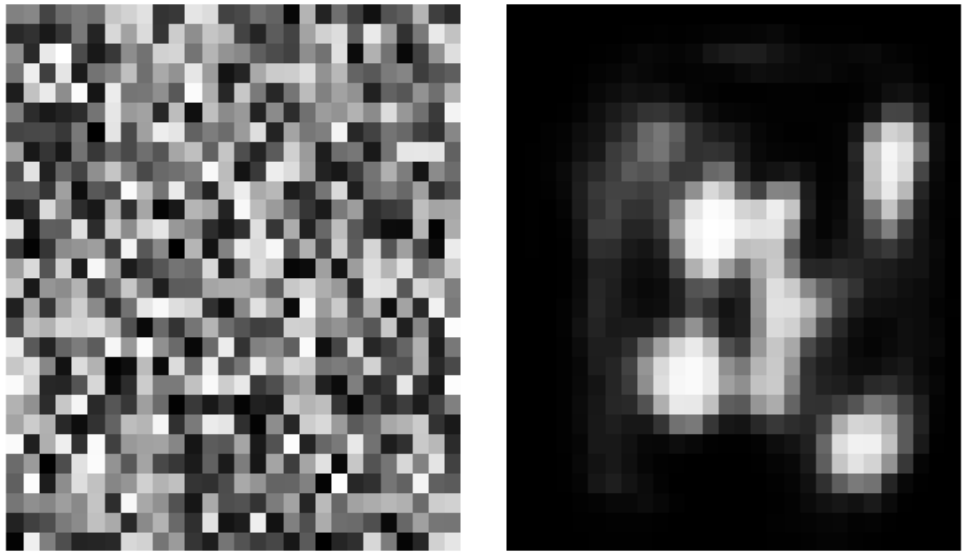
Let's compare the input and output of our auto encoder.

```
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)

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)

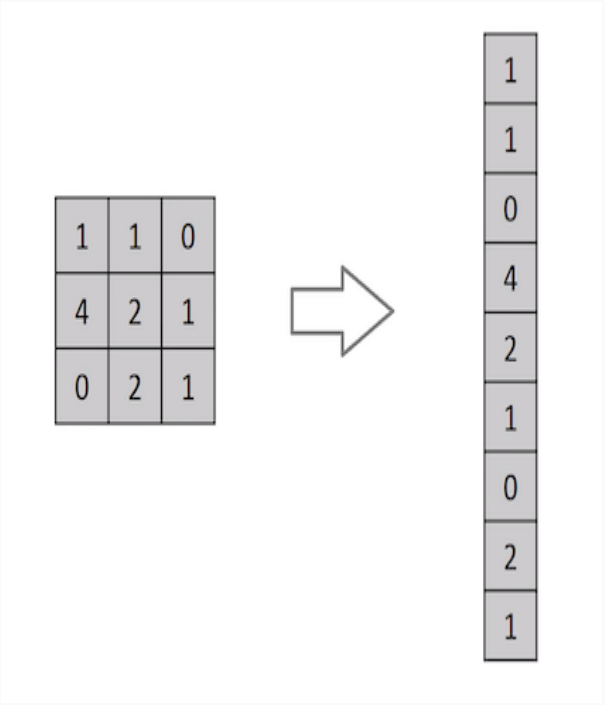
# Convolutional neural networks (CNN)

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

```
input <- 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

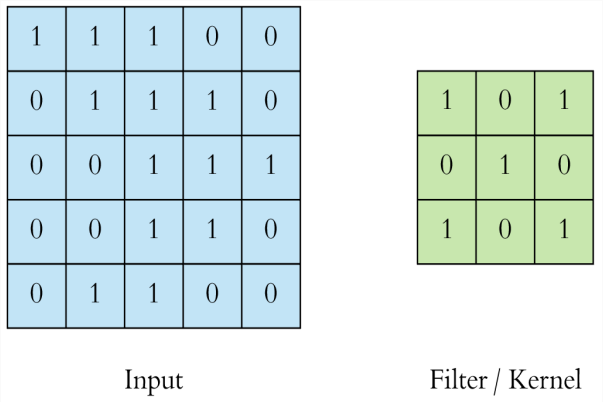
**Convolutional layers** allow to handle dimensional data, **without** flattening.

# Convolutional layers

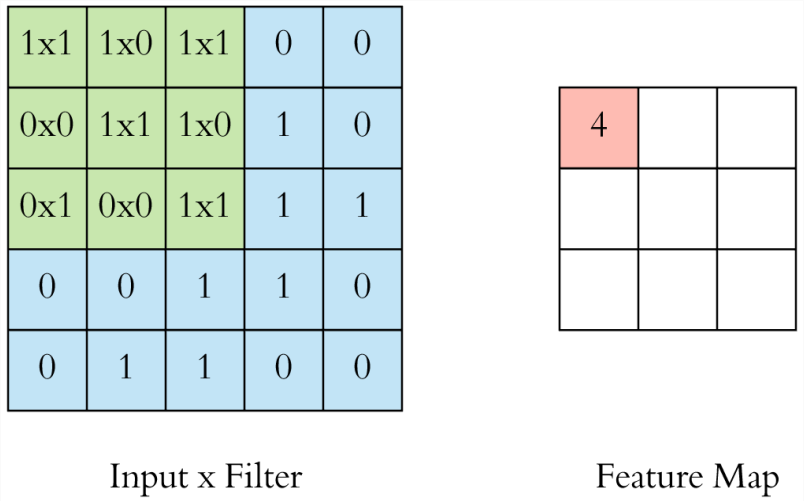
Classical hidden layers use **1 dimensional inputs** to construct **1 dimensional features**.

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.




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



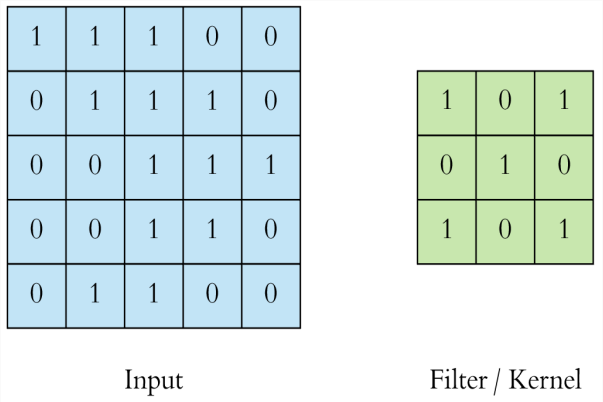

Source: Bradley Boehmke

# Convolutional layers (cont.)

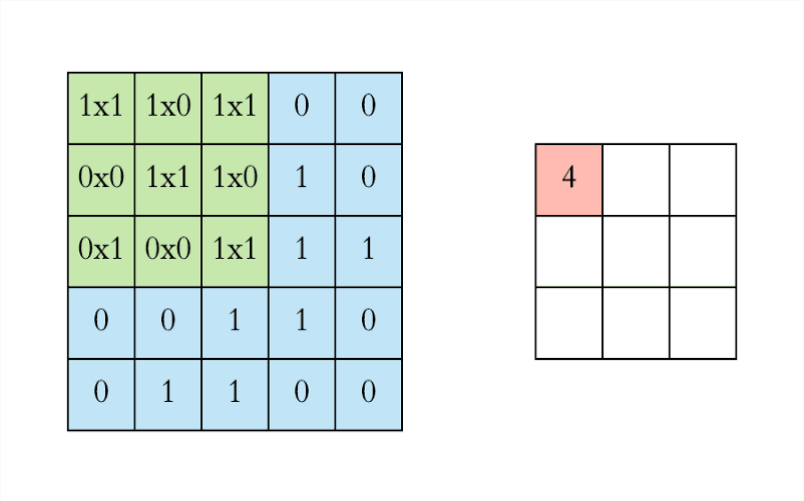
Classical hidden layers use **1 dimensional inputs** to construct **1 dimensional features**.

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.




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



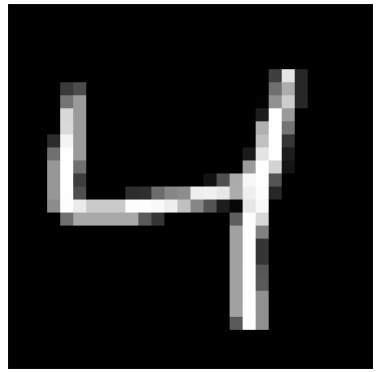

Source: Bradley Boehmke

# 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?

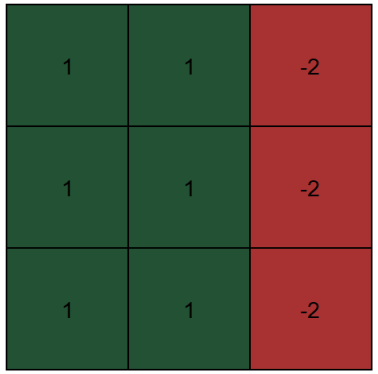


# Convolutional layers (cont.)

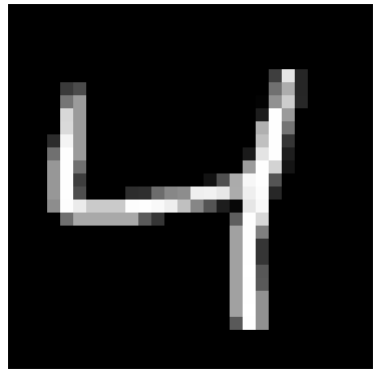
2d convolutional layers can **detect** the same, **local feature anywhere** in the image.

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

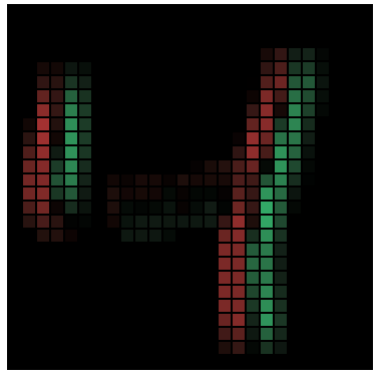
**Q**: How should the kernel look to detect this feature?




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))
```

- `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 the most 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.

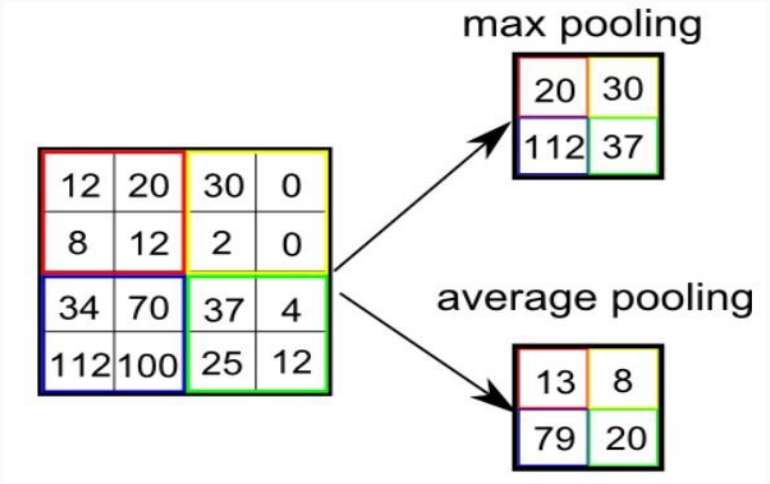
Two common operations are:

- 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)`:

Aggregate 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.

```
input_conv <- mnist$train$x / 255
input_conv <- k_expand_dims(input_conv, axis = 4)

test_input_conv <- mnist$test$x / 255
test_input_conv <- k_expand_dims(test_input_conv, axis = 4)
```

```
model <- load_model_tf("model_c")
```

# 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 (28x28x1)
input_conv <- k_expand_dims(input_conv, axis = 4)
```

Once you are finished 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
## [1] 0.1224037
##
## $accuracy
## [1] 0.9659
```

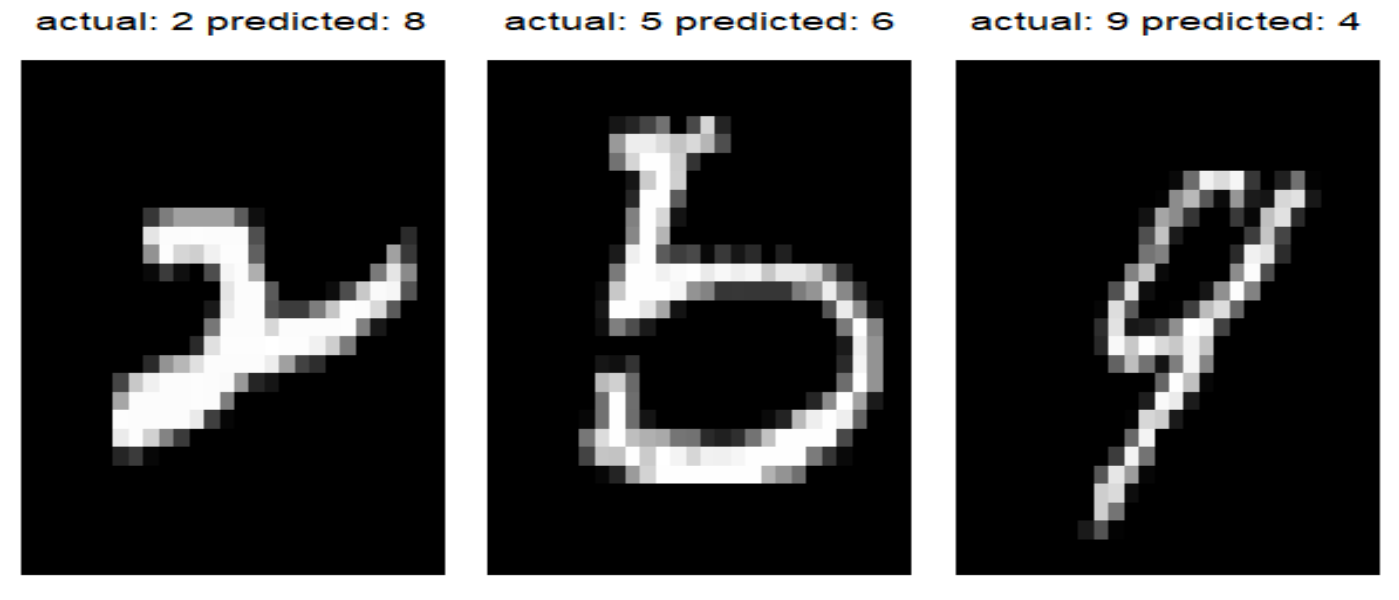
Model accuracy has increased significantly and improves further with more epochs.

# 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))
```

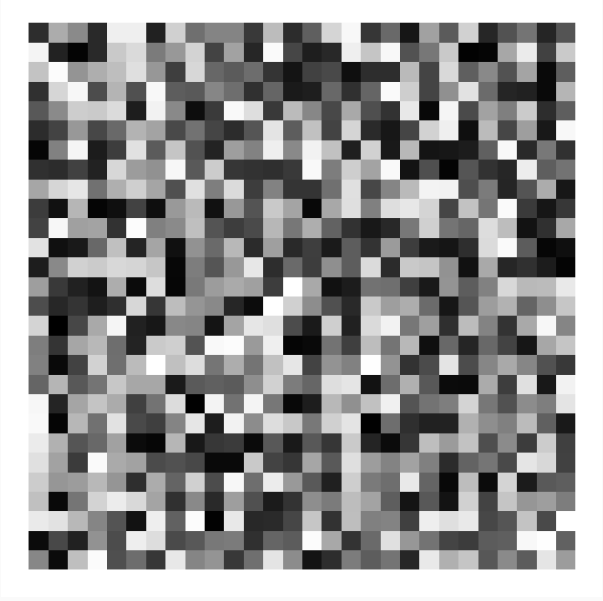
```
## [1]   9  19  93 248 260 319
```

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



# 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)
```

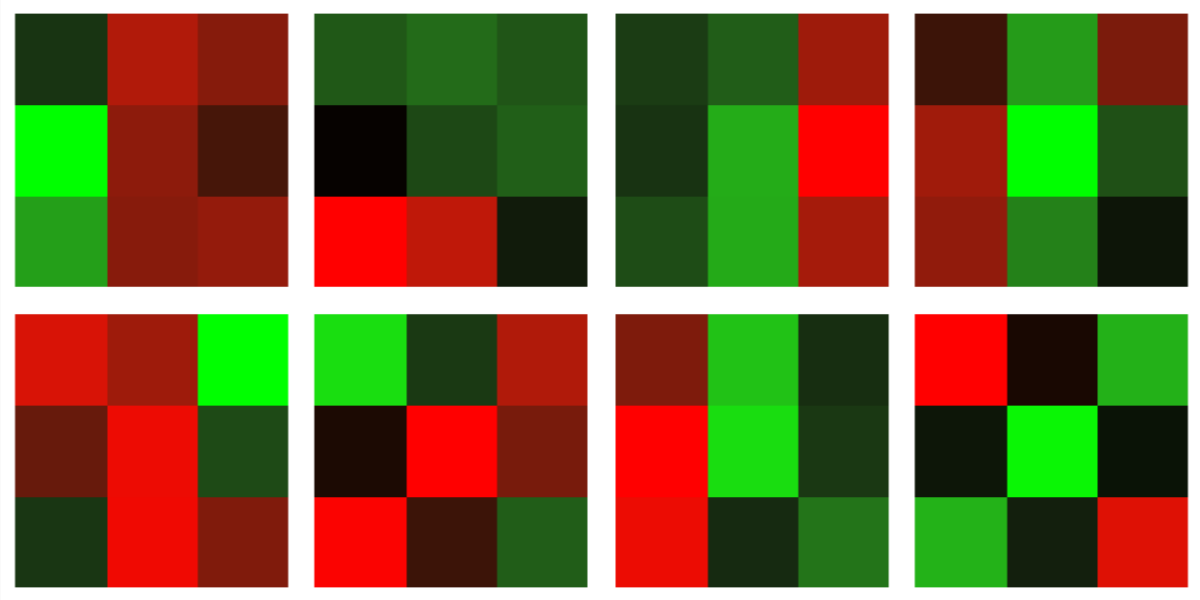
```
##               [,1]         [,2]         [,3]          
## [1,] 7.361005e-15 2.655474e-30 2.406724e-12 2.88801
##               [,6]         [,7]         [,8]       [,
## [1,] 3.955061e-15 6.562733e-10 2.638563e-21 0.99999
```

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

# Inspecting the filter/kernel

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

```
require(tidyverse)
require(gridExtra)
weights <- map(1:8, function(x) {plot_image(as.numeric(model$weights[[1]][,,,x]), FALSE)})
weights[['nrow']] <- 2
do.call(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

# Regression

# Regression with neural networks

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

Load the dataset:

```
mtpl_orig <- read.table('./data/P&Cdata.txt',
                        header = TRUE)
mtpl_orig <- as_tibble(mtpl_orig)
mtpl <- mtpl_orig %>%
  rename_all(function(.name) {.name %>% tolower }) %>%
  rename(expo = exp)
```

Create a training and test set:

```
require(rsample)

data_split <- initial_split(mtpl)
mtpl_train <- training(data_split)
mtpl_test  <- testing(data_split)

mtpl_train <- mtpl_train[sample(nrow(mtpl_train)), ]
```

# Regression with neural networks (cont.)

On Day 1 we fitted GLM regression models

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

We now reconstruct this model as a neural network:

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

# Your first NN regression model

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

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

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

- `layer_dense`:

There are no hidden layers, the input layer is directly connected to the output layer.

- `units = 1`:

There is a single output node.

- `input_shape = c(1)`:

There is one input covariate, i.e. the intercept which will be constant one.

# Your first NN regression model (cont.)

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

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

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

- `use_bias = FALSE`:

We don't need a bias, since we explicitely include a constant covariate.

- `activation = 'exponential'`:

We use an exponential inverse link function.

- `loss = 'poisson'`:

We minimize the poisson likelihood

# Your first NN regression model (cont.)

Prepare the input and output of the neural network.
Both should be of type **matrix** or tensor.

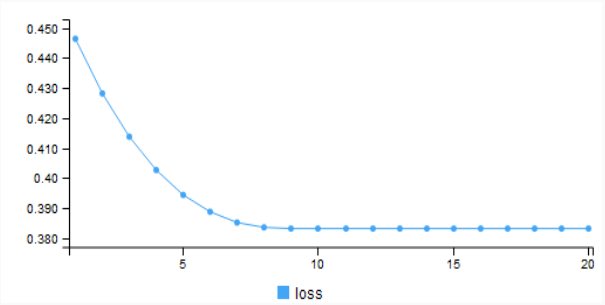
```
# (n x 1) matrix of constant one.
input <- matrix(1,
                nrow = nrow(mtpl_train),
                ncol = 1)
input_test <- matrix(1,
                     nrow = nrow(mtpl_test),
                     ncol = 1)

# (n x 1) matrix with the claim counts
output <- matrix(mtpl_train %>% pull(nclaims),
                 nrow = nrow(mtpl_train),
                 ncol = 1)
output_test <- matrix(mtpl_test %>% pull(nclaims),
                      nrow = nrow(mtpl_test),
                      ncol = 1)
```

Calibrate the neural network:

```
claim_count_model %>% fit(input,
                          output,
                          epochs = 20,
                          batch_size = 1024,
                          validation_split = 0)
```

- `validation_split = 0`:

We don't use a validation set, because we will compare the results with the glm fit.



# Comparing our neural network with a GLM

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

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

glm_fit
```

```
##
## Call:  glm(formula = nclaims ~ 1, family = poisson(l
##
## Coefficients:
## (Intercept)
##      -2.087
##
## Degrees of Freedom: 122423 Total (i.e. Null);  12242
## Null Deviance:        67510
## Residual Deviance: 67510     AIC: 95860
```

```
claim_count_model$weights
```

```
## [[1]]
## <tf.Variable 'dense_2/kernel:0' shape=(1, 1) dtype=f
```

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

| | (Intercept) |
|---|---|
| GLM | -2.088 |
| Neural network | -2.079 |

# Comparing our neural network with a GLM (cont.)

We implement the Poisson loss function:

$$P(Y = k) = \frac{e^{-\lambda}\lambda^k}{k!}$$

$$\log(P(Y = k)) = -\lambda + k \cdot \log(\lambda) - \log(k!)$$

```
poisson_loss <- function(pred, actual) {
  mean(pred - actual * log(pred))
}

poisson_loss(predict(glm_fit,
                     mtpl_test,
                     type = 'response'),
             mtpl_test$nclaims)
```

```
## [1] 0.3824852
```

```
evaluate(claim_count_model,
         input_test,
         output_test,
         verbose = FALSE)
```

```
## $loss
## [1] 0.3824905
##
## $mse
## [1] 0.1366765
```

| | In-sample loss | Out-of-sample loss |
|---|---|---|
| GLM | 0.3829 | 0.3831 |
| Neural network | 0.3829 | 0.3831 |

# Your turn

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

1. Can you adapt this code to replicate a binomial GLM with a logit link function. The `sigmoid` activation function is the inverse of the logit link function.
2. Calibrate the model defined in 1 on the outcome variable (nclaims > 0), i.e.~modelling no claim versus having at least one claim. Add accuracy as a metric in your model.
3. Compare your fitted neural network with a glm model.

```
claim_model_binair <- 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'))

claim_model_binair %>%
  fit(input,
      output > 0,
      epochs = 40,
      batch_size = 1024,
      validation_split = 0)

glm_binair <- glm((nclaims > 0) ~ 1,
                  data = mtpl_train,
                  family = binomial(link = logit))
```

The `sigmoid` activation function is the inverse of the logit link function.

`binary_crossentropy` minimizes the loglikelihood of bernoulli distributed random variables:

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

# Adding exposure

Each observation in the MTPL dataset has an exposure associated to it.

The loss function, including exposure, is

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

where:

- $\mathbf{expo}_i$ is the exposure of observation $i$,
- $\lambda_i$ is the fitted intensity,
- $k_i$ is the observed claim count.

We rewrite this loss function as:

$$\begin{aligned}\mathcal{L} &= \sum_i \mathbf{expo}_i \cdot \lambda_i - k_i \cdot \log(\mathbf{expo}_i \cdot \lambda_i) \\ &= \mathbf{expo}_i \cdot (\lambda_i - \frac{k_i}{\mathbf{expo}_i} \log(\lambda_i)).\end{aligned}$$

This is the loss function for a Poisson regression model with:

- observations $\frac{k_i}{\mathbf{expo}_i}$,
- weights $\mathbf{expo}_i$.

The parameter estimates of the following two GLMs are identical:

```
glm(nclaims ~ offset(log(expo)) + ageph, family = poisson, data = mtpl_train)
glm(nclaims / expo ~ ageph, family = poisson, data = mtpl_train, weights = expo)
```

# Adding exposure (cont.)

```
exposure <- as.numeric(mtpl_train  %>% pull(expo),
                       nrow = nrow(mtpl_train),
                       ncol = 1)

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

claim_count_model %>%
  fit(input,
      output / exposure,
      epochs = 20,
      batch_size = 1024,
      validation_split = 0,
      sample_weight = exposure)
```

We create a vector of exposures with the same length as our training dataset.

The model does not change, only the callibration procedure.

We divide the output by the exposure and include the exposure vector as weights.

# Adding exposure (cont.)

```
exposure_test <- k_constant(mtpl_test %>% pull(expo),
                            shape = c(nrow(mtpl_test)))

claim_count_model %>%
  evaluate(input_test,
           output_test,
           sample_weight = exposure_test,
           verbose = FALSE)
```

When evaluating the model, the weights have to be provided as a 1-dimensional tensor.

# Adding predictors and hidden layers

We have covered the fundamentals and are ready to build some more interesting neural networks.
We start by adding a single covariate `ageph`.

```
input <- matrix(mtpl_train %>% pull(ageph),
                nrow = nrow(mtpl_train),
                ncol = 1)

claim_count_model <- keras_model_sequential() %>%
  layer_batch_normalization(input_shape = c(1)) %>%
  layer_dense(units = 5, activation = 'relu') %>%
  layer_dense(units = 1,
              activation = 'exponential',
              use_bias = TRUE) %>%
  compile(loss = 'poisson',
          optimize = optimizer_rmsprop(),
          metrics = c('mse'))
```

`ageph` is added to the input matrix.
We remove the constant one from the input matrix and instead set `bias = TRUE`.

The first layer in the network is `layer_batch_normalization`.
This layer centers and scales the input variables, conform the preprocessing steps discussed on Day 1. Centering and scaling helps the model to faster converge.

Our model has a single hidden layer with 5 nodes.

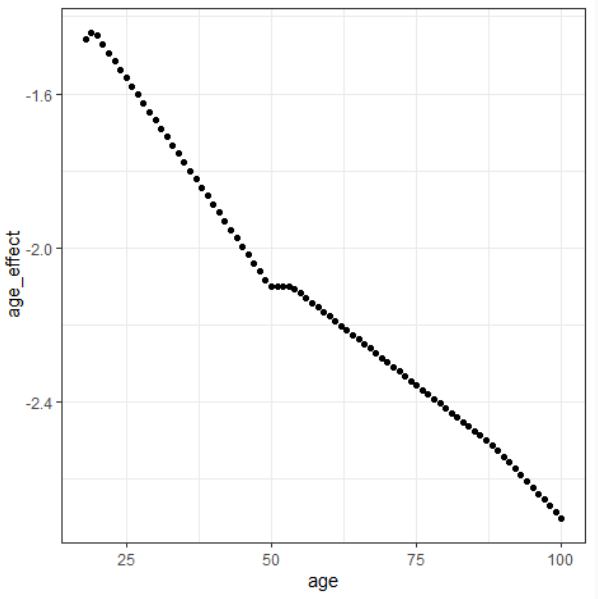
# Adding predictors and hidden layers (cont.)

We train the model on the training data.

```
claim_count_model %>%
  fit(input,
      output / exposure,
      epochs = 20,
      batch_size = 1024,
      validation_split = 0.2,
      sample_weight = exposure)
```

Let's visualize the fitted effect for ages between 18 and 100.

```
age <- 18:100 + 0.001
age_effect <- log(claim_count_model %>%
                    predict(age))

ggplot() +
  theme_bw() +
  geom_point(aes(age, age_effect))
```



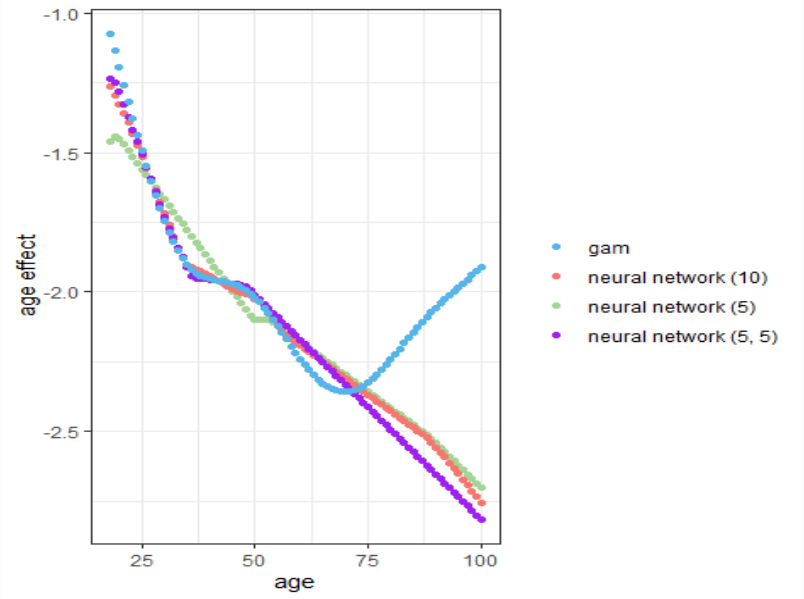
# Adding predictors and hidden layers (cont.)

We compare the fitted effect of age with the predictions of a `gam` model.

```
require(mgcv)
gam_age <- gam(nclaims/exposure ~ s(ageph),
               data = mtpl_train,
               family = poisson,
               weights = exposure)

df <- data.frame(ageph = age)
age_effect_gam <- log(
  predict(gam_age,
          newdata = data.frame(ageph = age),
          type = 'response'))
```

The figure below includes a neural network with:

- 1 hidden layer with 5 nodes
- 1 hidden layer with 10 nodes
- 2 hidden layers each with 5 nodes



# Your turn

It is time to combine everything you learned during this course in a case study!

In this case study you will predict the average claim size in the MTPL protfolio with a neural network.

**Step 1**: Data preparation:

- Filter the MTPL dataset on `nclaims > 0`.
- Construct a random training and test split.

**Step 2**: Data exploration:

- Visualize the density of `avg` and `log(avg)`.
- Which distribution would be suitable for modelling the claim size?

Irrespective of your choice we advise to estimate a lognormal distribution.
If you feel very brave you could implement and optimize your own gamma loss function.

# Your turn

**Step 3**: More data preparation:

- Write a recipe for preparing the input data set.
  Check the reference page for inspiration.
- Your recipe should atleast contain the following steps:
  - Remove the variables: id, nclaims, amount, exp, town and pc.
  - Normalize the data.
  - Take the logarithm of avg.
  - ...
- Bake the training and test dataset following your recipe.

# Your turn

**Step 4.** Model building:

- Construct and train at least five different neural networks on the training dataset.
- You can tune:
  - 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,
  - ...

**Step 5.** Model evaluation:

- Evaluate your models on the test dataset.
- Compare your selected neural network with a glm or gbm. Which model is preferred?

Reading in the data:

```
mtpl_orig <- read.table('./data/P&Cdata.txt',
                        header = TRUE)
mtpl_orig <- as_tibble(mtpl_orig)
mtpl <- mtpl_orig %>%
  rename_all(function(.name) {.name %>% tolower })
```

Split the data into a training and test dataset.

```
require(rsample)
set.seed(1)
data_split <- initial_split(mtpl)
mtpl_train <- training(data_split)
mtpl_test  <- testing(data_split)
mtpl_train <- mtpl_train[sample(nrow(mtpl_train)), ]
```

Select only records with atleast one claim:

```
mtpl <- mtpl %>% filter(nclaims > 0)
```

```
colnames(mtpl)
```

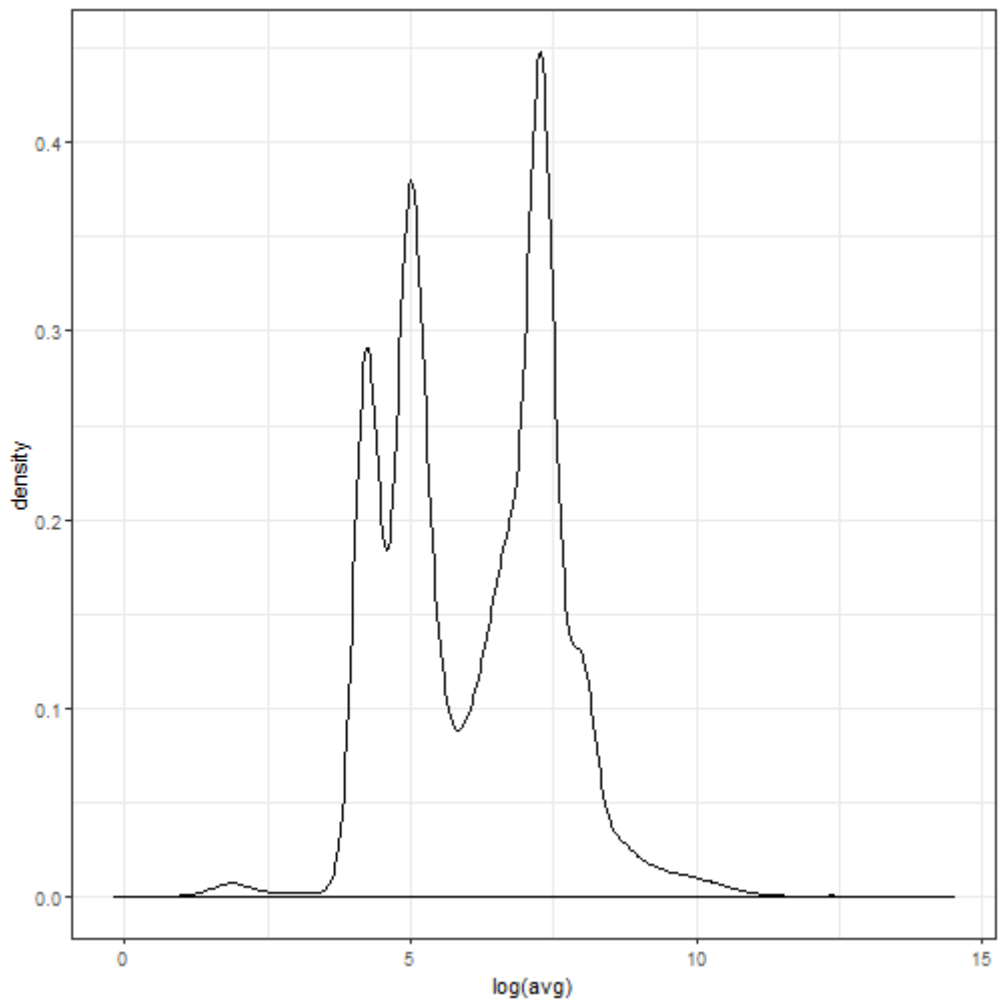
```
##  [1] "id"       "nclaims"  "amount"   "avg"      "exp"      "coverage"
##  [7] "fuel"     "use"      "fleet"    "sex"      "ageph"    "bm"
## [13] "agec"     "power"    "pc"       "town"     "long"     "lat"
```

Visualizing the outcome variable:

```
ggplot(mtpl_train) +
  theme_bw() +
  geom_density(aes(log(avg)))
```

The loss function for a gamma distribution:

```
gamma_loss <- function(y_true, y_pred) {
  k_mean(y_true / y_pred +  k_log(y_pred), axis = 2)
}
```



```
require(recipes)
mtpl_recipe <- recipe(avg ~ ., data = mtpl_train) %>%
  step_rm(id, nclaims, amount, exp, town, pc) %>%
  step_nzv(all_predictors()) %>%
  step_normalize(all_numeric(), -all_outcomes()) %>%
  step_dummy(all_nominal(), one_hot = TRUE) %>%
  step_log(avg) %>%
  prep(mtpl_train)

baked_mtpl_train <- bake(mtpl_recipe,
                         new_data = mtpl_train)
baked_mtpl_test <- bake(mtpl_recipe,
                        new_data = mtpl_test)

input_mtpl_train <-select(baked_mtpl_train, -avg) %>%
  as.matrix()
input_mtpl_test <- select(baked_mtpl_test, -avg) %>%
  as.matrix()
output_mtpl_train <- baked_mtpl_train %>% pull(avg)
output_mtpl_test <- baked_mtpl_test %>% pull(avg)
```

- `step_rm`:

Remove specific variables from the dataset.

- `step_nzv`:

Remove variables with near zero variance form the dataset.

- `step_normalize`:

Normalize all numeric predictor variables.

- `step_dummy`:

Construct dummy variables with one-hot encoding.

- `step_log`:

Take the logarithm of `avg`.

An first neural network, many other configuration are possible:

```
model <- keras_model_sequential() %>%
  layer_dense(units = 16,
              activation = 'relu',
              input_shape = ncol(input_mtpl_train)) %>%
  layer_dense(units = 1,
                  activation = NULL) %>%
  compile(loss = 'mse',
              optimize = optimizer_rmsprop(),
              metrics = c('mse'))

model %>% fit(
  input_mtpl_train,
  output_mtpl_train,
  epochs = 30,
  batch_size = 256,
  validation_split = 0.2
)
```

`units = 16`:

I start with a low number of nodes, since we do not have many input variables.

```
dim(input_mtpl_train)
```

```
## [1] 13722    13
```

`activation = NULL`:

No activation in the output layer, this is similar to linear regression.

`loss = 'mse'`:

We estimate a gaussian distribution for `log(avg)`.

Compare the loss of the neural network and the glm.

```
evaluate(model,
         input_mtpl_test,
         output_mtpl_test, verbose = FALSE)

fit_glm <- lm(avg ~ ., data = baked_mtpl_train)
mean((predict(fit_glm,
              newdata = baked_mtpl_test) - baked_mtpl_test$avg)^2)
```

# Thanks!


Slides created with the R package xaringan.

Course material available via

https://github.com/katrienantonio/workshop-ML