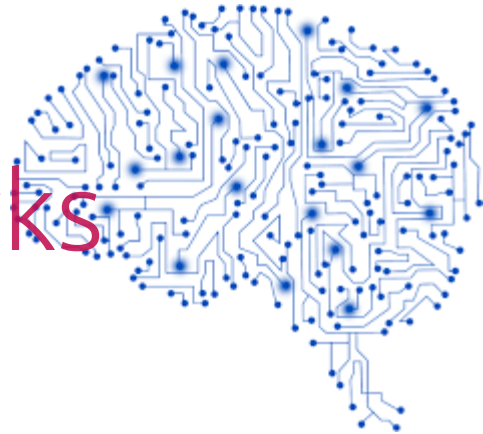


# Hands on neural networks



Alexandre Boucaud - [@alxbcd](#)

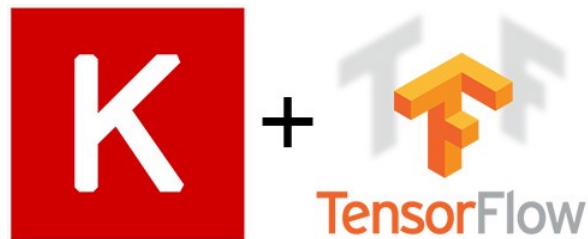
# Outline

- neurons
- hidden layers
- activation
- backpropagation
- training
- hands-on tutorial

# Foreword

The following slides provide examples of neural network models written in *Python*, using the [Keras](#) library and [TensorFlow](#) tensor ordering convention\*.

Keras provides a high level API to create deep neural networks and train them using numerical tensor libraries (*backends*) such as [TensorFlow](#), [CNTK](#) or [Theano](#).



\*channels last

What is a neural network  
made of ?

# A Neuron

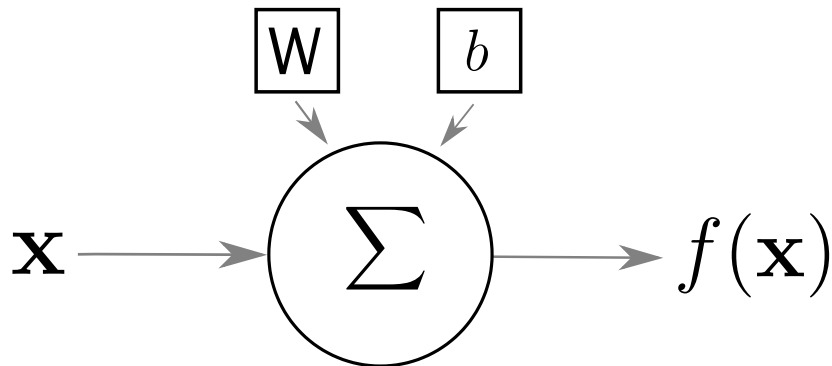
A neuron is a **linear system** with two attributes

the weight matrix  $\mathbf{W}$

the linear bias  $b$

It takes **multiple inputs** (from  $\mathbf{x}$ ) and returns **a single output**

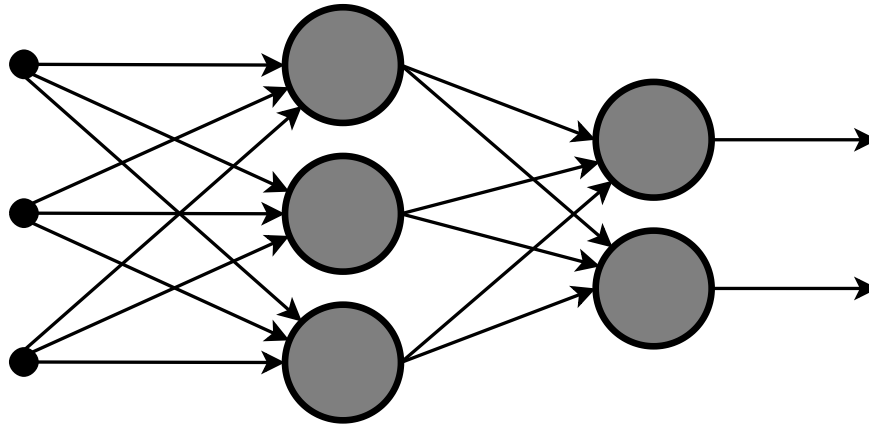
$$f(\mathbf{x}) = \mathbf{W} \cdot \mathbf{x} + b$$



# Linear layers

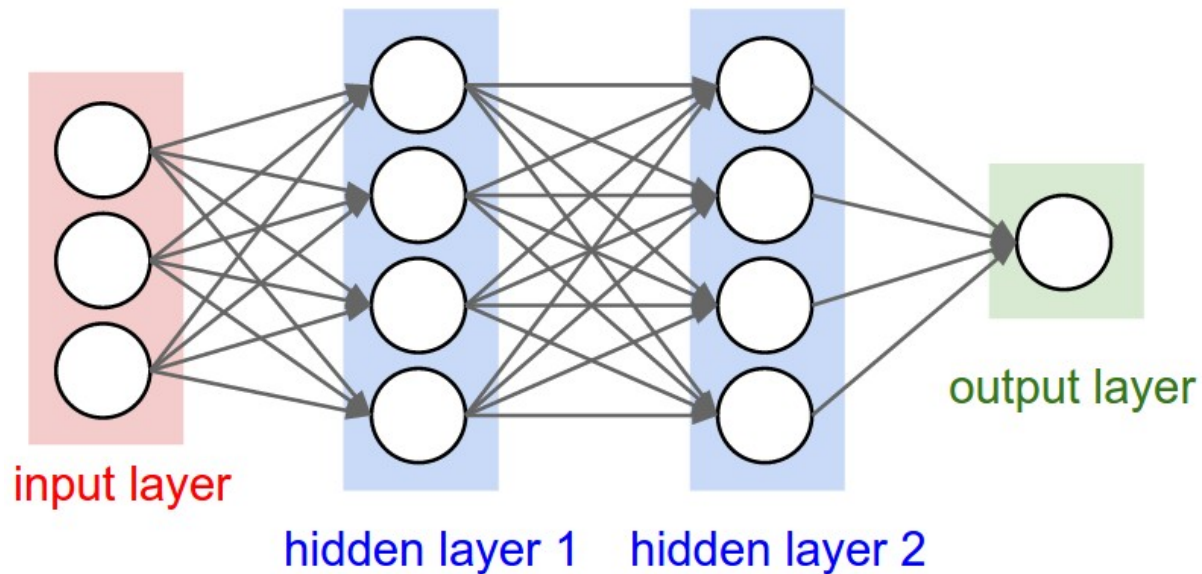
A linear layer is an **array of neurons**.

A layer has **multiple inputs** (same  $\mathbf{x}$  for each neuron) and returns **multiple outputs**.



# Hidden layers

All layers internal to the network (not input or output layer) are considered **hidden layers**.

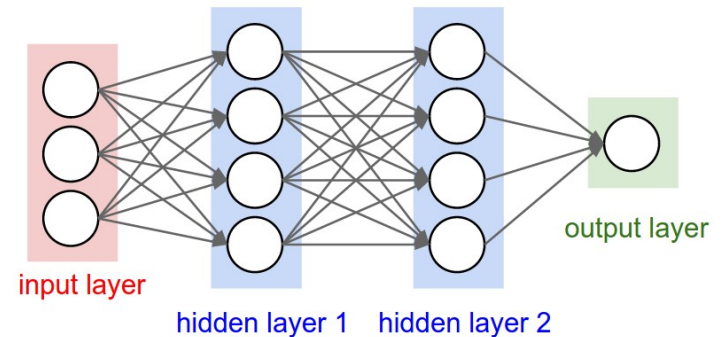


# Multi-layer perceptron (MLP)

```
from keras.models import Sequential
from keras.layers import Dense

# initialize model
model = Sequential()

# add layers
model.add(Dense(4, input_dim=3))
model.add(Dense(4))
model.add(Dense(1))
```



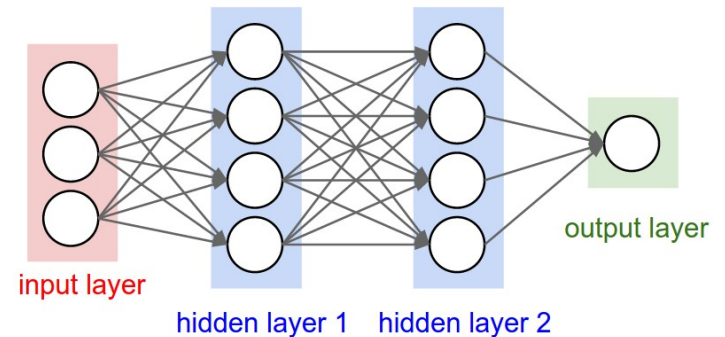


# Multi-layer perceptron (MLP)

```
from keras.models import Sequential
from keras.layers import Dense

# initialize model
model = Sequential()

# add layers
model.add(Dense(4, input_dim=3))
model.add(Dense(4))
model.add(Dense(1))
```



QUESTION:

How many **free parameters** has this model ?

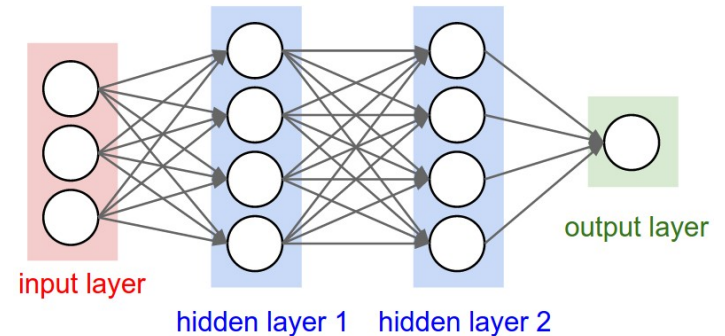
# Multi-layer perceptron (MLP)

```
from keras.models import Sequential
from keras.layers import Dense

model = Sequential()

model.add(Dense(4, input_dim=3))
model.add(Dense(4))
model.add(Dense(1))

# print model structure
model.summary()
```



Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 4)	16

$\leq$  W (3, 4)    b (4, 1)

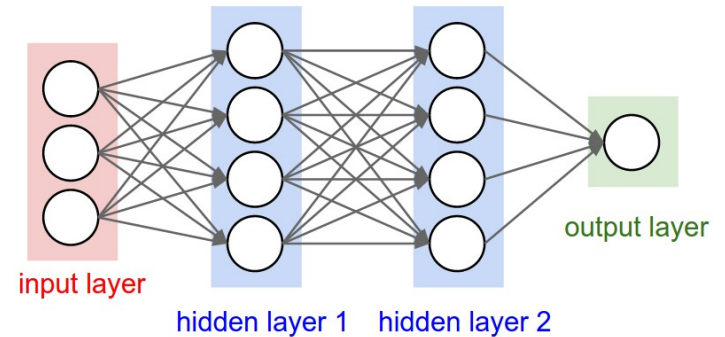
# Multi-layer perceptron (MLP)

```
from keras.models import Sequential
from keras.layers import Dense

model = Sequential()

model.add(Dense(4, input_dim=3))
model.add(Dense(4))
model.add(Dense(1))

# print model structure
model.summary()
```



Layer (type)	Output Shape	Param #
=====		
dense_1 (Dense)	(None, 4)	16
dense_2 (Dense)	(None, 4)	20

$\leq$  W (4, 4)    b (4, 1)

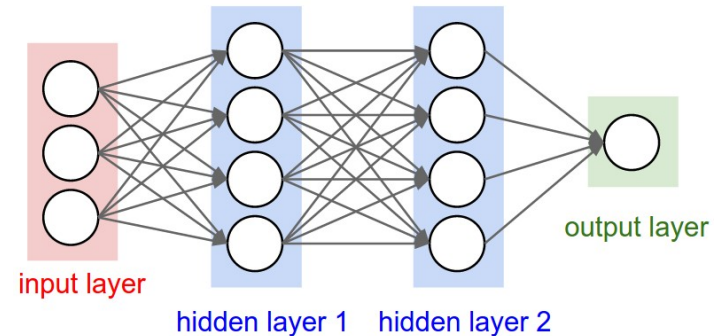
# Multi-layer perceptron (MLP)

```
from keras.models import Sequential
from keras.layers import Dense

model = Sequential()

model.add(Dense(4, input_dim=3))
model.add(Dense(4))
model.add(Dense(1))

# print model structure
model.summary()
```



Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 4)	16
dense_2 (Dense)	(None, 4)	20
dense_3 (Dense)	(None, 1)	5

Total params: 41  
Trainable params: 41  
Non-trainable params: 0

$\leq W(4, 1) \quad b(1, 1)$

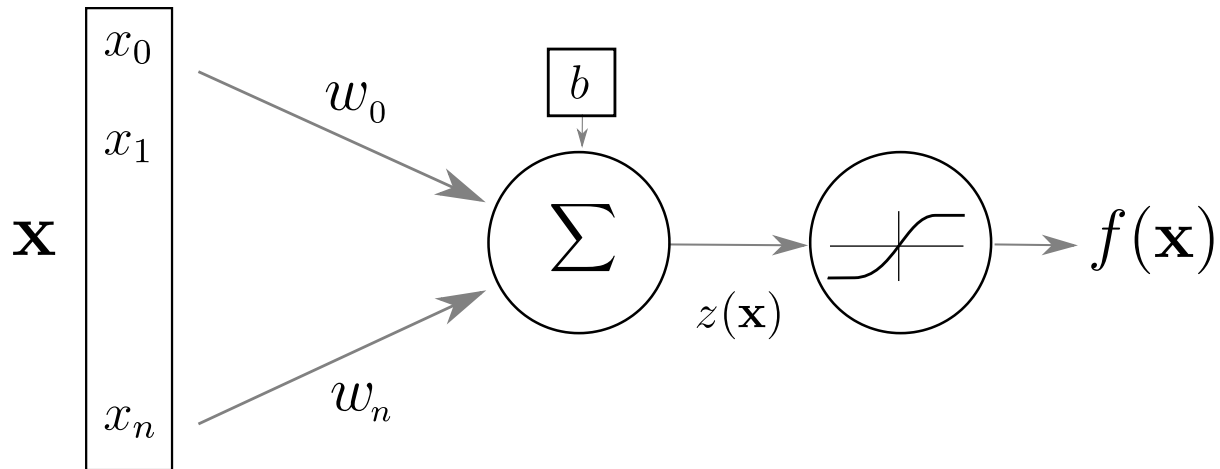
# Adding non linearities

A network with several linear layers remains a **linear system**.

# Adding non linearities

A network with several linear layers remains a **linear system**.

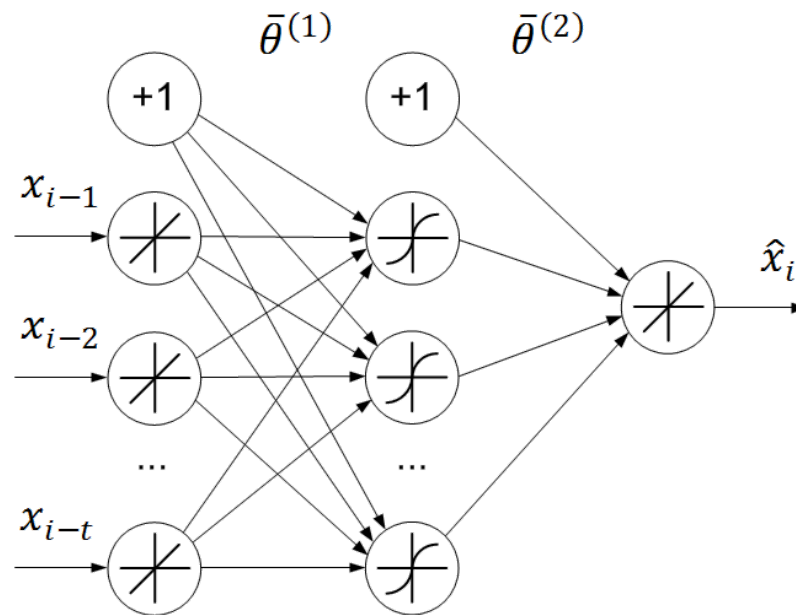
To add non-linearities to the system, **activation functions** are introduced.



# Adding non linearities

A network with several linear layers remains a **linear system**.

To add non-linearities to the system, **activation functions** are introduced.

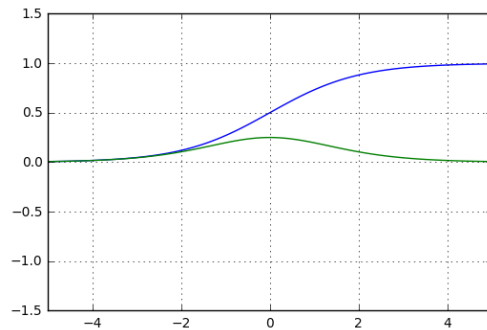


credit: Alexander Chekunkov

# Activation functions

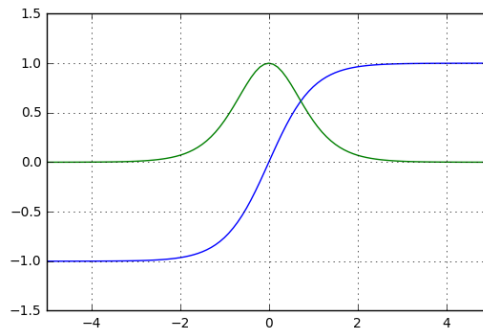
activation function

its derivative



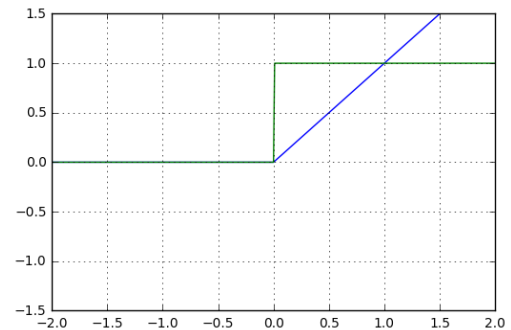
$$\text{sigm}(x) = \frac{1}{1 + e^{-x}}$$

$$\text{sigm}'(x) = \text{sigm}(x)(1 - \text{sigm}(x))$$



$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

$$\tanh'(x) = 1 - \tanh(x)^2$$



$$\text{relu}(x) = \max(0, x)$$

$$\text{relu}'(x) = 1_{x>0}$$

An extended list of activation functions can be found on [wikipedia](https://en.wikipedia.org/wiki/Activation_function).



# Activation layer

There are two different syntaxes whether the activation is seen as a **property** of the neuron layer

```
model = Sequential()  
model.add(Dense(4, input_dim=3, activation='sigmoid'))
```

# Activation layer

There are two different syntaxes whether the activation is seen as a **property** of the neuron layer

```
model = Sequential()  
model.add(Dense(4, input_dim=3, activation='sigmoid'))
```

or as an **additional layer** to the stack

```
from keras.layers import Activation  
  
model = Sequential()  
model.add(Dense(4, input_dim=3))  
model.add(Activation('tanh'))
```

# Activation layer

There are two different syntaxes whether the activation is seen as a **property** of the neuron layer

```
model = Sequential()  
model.add(Dense(4, input_dim=3, activation='sigmoid'))
```

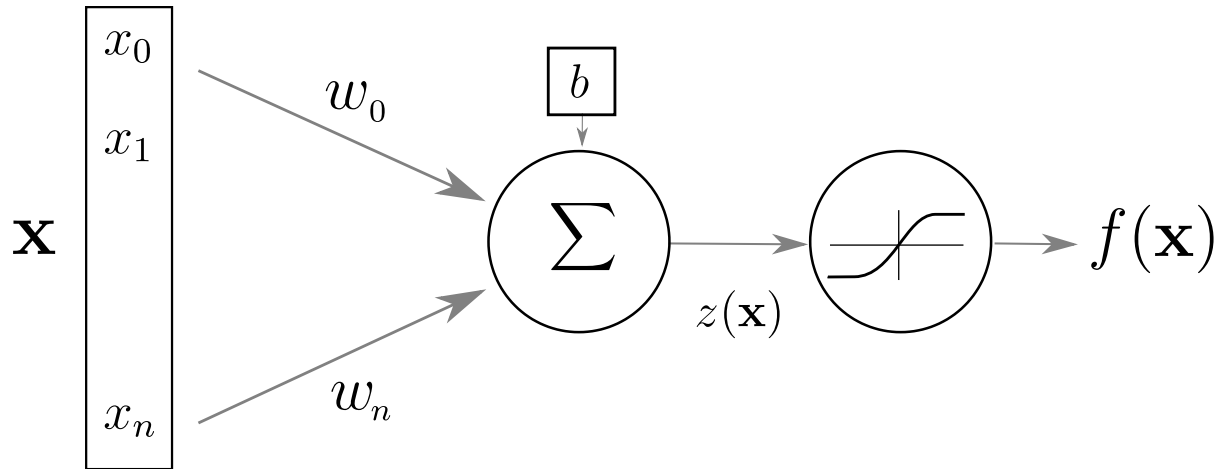
or as an **additional layer** to the stack

```
from keras.layers import Activation  
  
model = Sequential()  
model.add(Dense(4, input_dim=3))  
model.add(Activation('tanh'))
```

The activation layer **does not add** any **depth** to the network.

# Simple network

One neuron, one activation function.



$$x \xrightarrow{\text{neuron}} z(x) = wx + b \xrightarrow{\text{activation}} a(x) = g(z(x)) = y$$

# Feed forward

$$x \xrightarrow{\text{neuron}} z(x) = wx + b \xrightarrow{\text{activation}} a(x) = g(z(x)) = y$$

We propagate an input  $x$  through the network and compare the result  $y$  with the expected value  $y_t$  using the loss function

$$\ell = \text{loss}(y, y_t)$$

# Feed forward

$$x \xrightarrow{\text{neuron}} z(x) = wx + b \xrightarrow{\text{activation}} a(x) = g(z(x)) = y$$

We propagate an input  $x$  through the network and compare the result  $y$  with the expected value  $y_t$  using the loss function

$$\ell = \text{loss}(y, y_t)$$

How do we quantify the impact of  $w$  and  $b$  on the loss  $\ell$ ?

# Feed forward

$$x \xrightarrow{\text{neuron}} z(x) = wx + b \xrightarrow{\text{activation}} a(x) = g(z(x)) = y$$

We propagate an input  $x$  through the network and compare the result  $y$  with the expected value  $y_t$  using the loss function

$$\ell = \text{loss}(y, y_t)$$

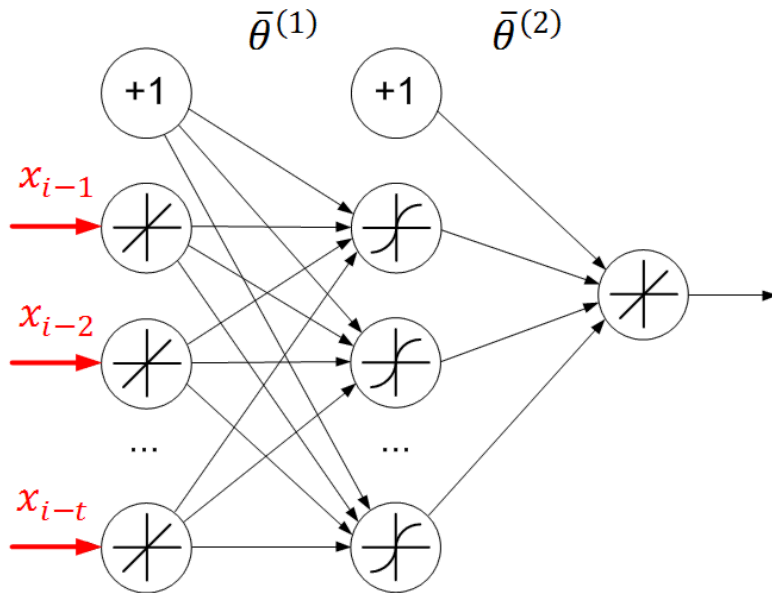
How do we quantify the impact of  $w$  and  $b$  on the loss  $\ell$  ?

$$\frac{\partial \ell}{\partial w} \quad \text{and} \quad \frac{\partial \ell}{\partial b}$$

# Backpropagation

A 30-years old algorithm (Rumelhart et al., 1986)

which is **key** for the re-emergence of neural networks today.



credit: Alexander Chekunkov



# Chain rule

Backpropagation works if networks are **differentiable**.

**Each layer** must have an analytic derivative expression.

$$x \xrightarrow{\text{neuron}} z(x) = wx + b \xrightarrow{\text{activation}} a(x) = g(z(x)) = y$$

# Chain rule

Backpropagation works if networks are **differentiable**.

**Each layer** must have an analytic derivative expression.

$$x \xrightarrow{\text{neuron}} z(x) = wx + b \xrightarrow{\text{activation}} a(x) = g(z(x)) = y$$

Since  $w$  and  $b$  are also variables

$$z(x, w, b) = wx + b,$$

the gradients can be expressed as

$$\frac{\partial z}{\partial x} = w, \quad \frac{\partial z}{\partial w} = x \quad \text{and} \quad \frac{\partial z}{\partial b} = 1.$$

# Chain rule

Backpropagation works if networks are **differentiable**.

**Each layer** must have an analytic derivative expression.

$$x \xrightarrow{\text{neuron}} z(x) = wx + b \xrightarrow{\text{activation}} a(x) = g(z(x)) = y$$

Then the **chain rule** can be applied :

$$\frac{\partial \ell}{\partial w} = \frac{\partial \ell}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w} = \nabla \text{loss}(y) \cdot g'(z) \cdot x$$

and

$$\frac{\partial \ell}{\partial b} = \frac{\partial \ell}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial b} = \nabla \text{loss}(y) \cdot g'(z)$$

# Network with more layers

Let's add one layer (with a single neuron) with the **same** activation  $g$

$$\begin{aligned}x &\longrightarrow z_1 = w_1 x + b_1 \longrightarrow a_1 = g(z_1(x)) \longrightarrow \\&\longrightarrow z_2 = w_2 a_1 + b_2 \longrightarrow a_2 = g(z_2(x)) = y\end{aligned}$$

# Network with more layers

Let's add one layer (with a single neuron) with the **same** activation  $g$

$$x \longrightarrow z_1 = w_1 x + b_1 \longrightarrow a_1 = g(z_1(x)) \longrightarrow$$

$$\longrightarrow z_2 = w_2 a_1 + b_2 \longrightarrow a_2 = g(z_2(x)) = y$$

How do we compute the gradients of  $w_1 : \frac{\partial \ell}{\partial w_1}$  ?

# Network with more layers

Let's add one layer (with a single neuron) with the **same** activation  $g$

$$x \longrightarrow z_1 = w_1 x + b_1 \longrightarrow a_1 = g(z_1(x)) \longrightarrow$$

$$\longrightarrow z_2 = w_2 a_1 + b_2 \longrightarrow a_2 = g(z_2(x)) = y$$

How do we compute the gradients of  $w_1 : \frac{\partial \ell}{\partial w_1}$  ?

Hint: remember the algorithm is called **backpropagation**

# Network with more layers

Let's add one layer (with a single neuron) with the **same** activation  $g$

$$\begin{aligned}x &\longrightarrow z_1 = w_1 x + b_1 \longrightarrow a_1 = g(z_1(x)) \longrightarrow \\&\longrightarrow z_2 = w_2 a_1 + b_2 \longrightarrow a_2 = g(z_2(x)) = y\end{aligned}$$

We use the **chain rule**

$$\frac{\partial \ell}{\partial w_1} = \frac{\partial \ell}{\partial a_2} \cdot \frac{\partial a_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial a_1} \cdot \frac{\partial a_1}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_1}$$

# Network with more layers

Let's add one layer (with a single neuron) with the **same** activation  $g$

$$\begin{aligned}x &\longrightarrow z_1 = w_1 x + b_1 \longrightarrow a_1 = g(z_1(x)) \longrightarrow \\&\longrightarrow z_2 = w_2 a_1 + b_2 \longrightarrow a_2 = g(z_2(x)) = y\end{aligned}$$

We use the **chain rule**

$$\frac{\partial \ell}{\partial w_1} = \frac{\partial \ell}{\partial a_2} \cdot \frac{\partial a_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial a_1} \cdot \frac{\partial a_1}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_1}$$

which simplifies to

$$\frac{\partial \ell}{\partial w_1} = \nabla \text{loss}(y) \cdot g'(z_2) \cdot w_2 \cdot g'(z_1) \cdot x$$



# Network with more layers

From the latest expression

$$\frac{\partial \ell}{\partial w_1} = (\nabla \text{loss}(y) \cdot g'(z_2) \cdot w_2) \cdot g'(z_1) \cdot x$$

one can derive the algorithm to compute the gradient for a layer  $z_i$

$$\frac{\partial \ell}{\partial z_i} = ((\nabla \text{loss}(y) \cdot g'(z_n) \cdot w_n) \cdot g'(z^{n-1}) * w^{n-1}) [\dots] \cdot g'(z_i)$$

which can be re-written as a recursion

$$\frac{\partial \ell}{\partial z_i} = \frac{\partial \ell}{\partial z^{i+1}} \cdot w^{i+1} \cdot g'(z_i)$$

# Network with more layers

From the latest expression

$$\frac{\partial \ell}{\partial w_1} = (\nabla \text{loss}(y) \cdot g'(z_2) \cdot w_2) \cdot g'(z_1) \cdot x$$

one can derive the algorithm to compute the gradient for a layer  $z_i$

$$\frac{\partial \ell}{\partial z_i} = ((\nabla \text{loss}(y) \cdot g'(z_n) \cdot w_n) \cdot g'(z^{n-1}) * w^{n-1}) [\dots] \cdot g'(z_i)$$

which can be re-written as a recursion

$$\frac{\partial \ell}{\partial z_i} = \frac{\partial \ell}{\partial z^{i+1}} \cdot w^{i+1} \cdot g'(z_i)$$

find a clear and more detailed explanation of backpropagation [here](#)

# Loss and optimizer

Once your architecture (`model`) is ready, a **loss function** and an **optimizer must** be specified

```
model.compile(optimizer='adam', loss='binary_crossentropy')
```

or with better access to optimization parameters

```
from keras.optimizers import Adam
from keras.losses import binary_crossentropy

model.compile(optimizer=Adam(lr=0.01, decay=0.1),
              loss=binary_crossentropy)
```

Choose both according to the target output.

# Network update

1. feedforward and compute loss gradient on the output

$$\nabla loss(y)$$

2. for each layer in the backward direction,
  - **receive** the gradients from the previous layer,
  - **compute** the gradient of the current layer
  - **multiply** with the weights and **pass** the results on to the next layer
3. for each layer, update their weight and bias using their own gradient, following the optimisation scheme (e.g. gradient descent)

# Training

It's time to **train** your model on the data (X\_train, y\_train).

```
model.fit(X_train, y_train,  
          batch_size=32,  
          epochs=50,  
          validation_split=0.3) # % of data being used for val_loss evaluation
```

- **batch\_size:** # of images used before updating the model  
32 is a very good compromise between precision and speed\*
- **epochs:** # of times the model is trained with the full dataset

After each epoch, the model will compute the loss on the validation set to produce the **val\_loss**.

The closer the values of **loss** and **val\_loss**, the better the training.

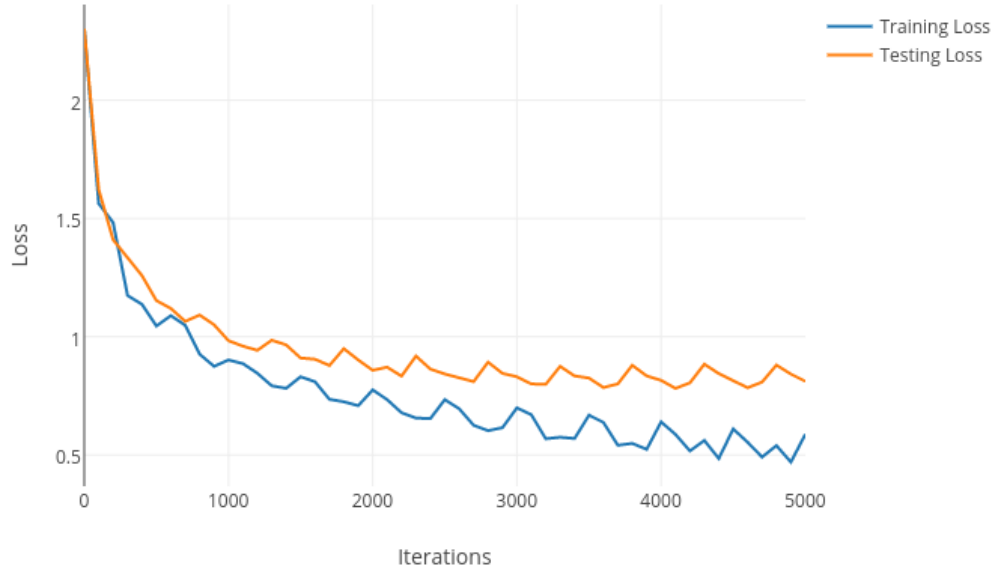
\*see [Masters et al. \(2018\)](#)

# Plot the training loss

```
import matplotlib.pyplot as plt

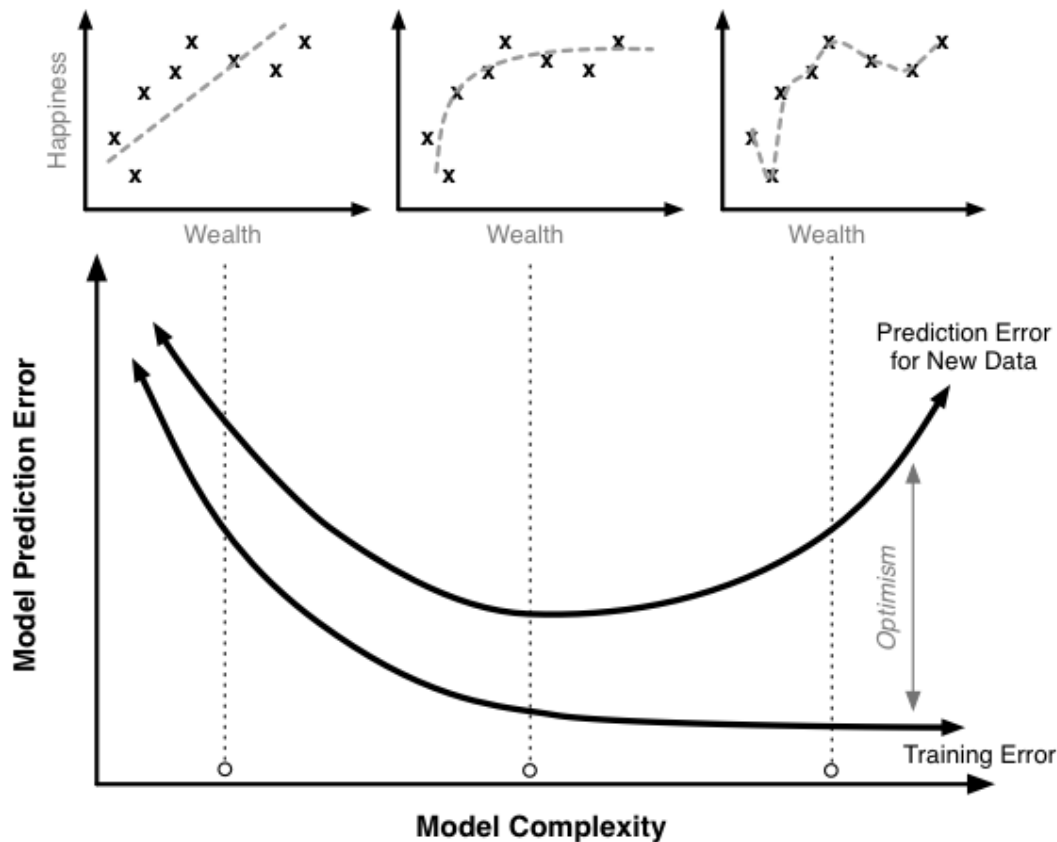
history = model.fit(X_train, y_train, validation_split=0.3)

# Visualizing the training
plt.plot(history.history['loss'], label='training')
plt.plot(history.history['val_loss'], label='validation')
plt.xlabel('epochs'); plt.ylabel('loss'); plt.legend()
```



# Plot the training loss

And look for the training **sweet spot** (before **overfitting**).



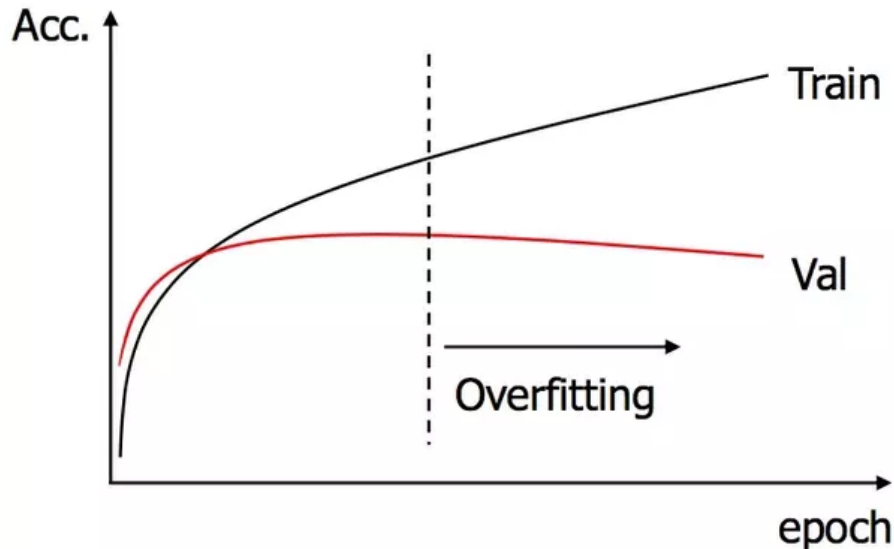
# Plot other metrics

```
import matplotlib.pyplot as plt

model.compile(..., metrics=['acc']) # computes other metrics, here accuracy

history = model.fit(X_train, y_train, validation_split=0.3)

# Visualizing the training
plt.plot(history.history['acc'], label='training')
plt.plot(history.history['val_acc'], label='validation')
plt.xlabel('epochs'); plt.ylabel('accuracy'); plt.legend()
```














# Common architectures

# Zoo of neural networks #1

A mostly complete chart of

## 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

Perceptron (P)



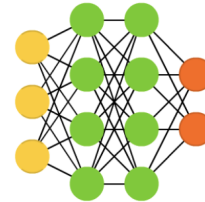
Feed Forward (FF)



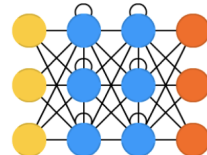
Radial Basis Network (RBF)



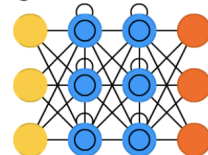
Deep Feed Forward (DFF)



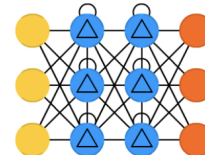
Recurrent Neural Network (RNN)



Long / Short Term Memory (LSTM)



Gated Recurrent Unit (GRU)



Auto Encoder (AE)



Variational AE (VAE)



Denoising AE (DAE)



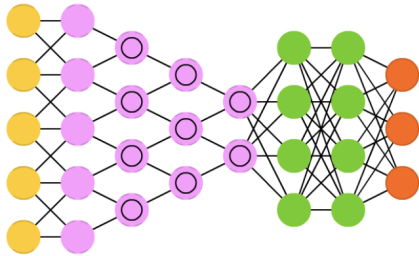
Sparse AE (SAE)



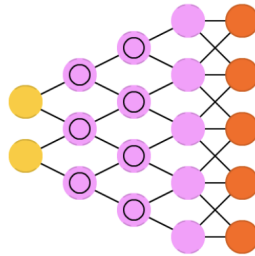
Neural network zoo - Fjodor van Veen (2016)

# Zoo of neural networks #2

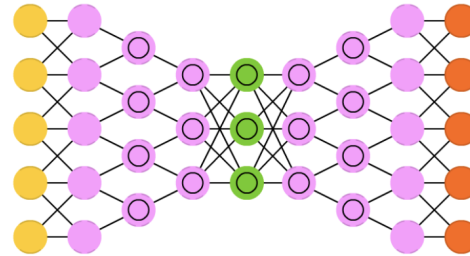
Deep Convolutional Network (DCN)



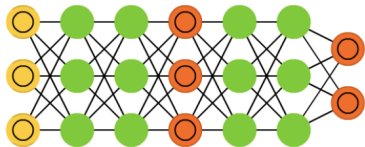
Deconvolutional Network (DN)



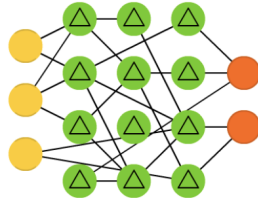
Deep Convolutional Inverse Graphics Network (DCIGN)



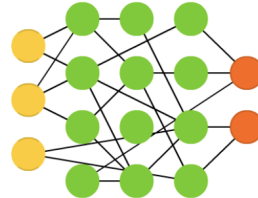
Generative Adversarial Network (GAN)



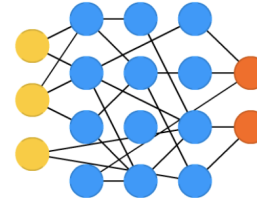
Liquid State Machine (LSM)



Extreme Learning Machine (ELM)



Echo State Network (ESN)



# Thank you

Contact info:

[aboucaud.github.io](https://aboucaud.github.io)

@aboucaud on GitHub, GitLab

[@alxbcd](#) on Twitter

This presentation is licensed under a  
[Creative Commons Attribution-ShareAlike 4.0 International License](#)

