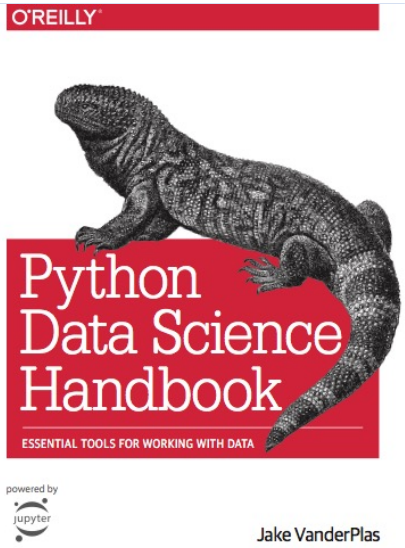

The
Perceptron

Main elements of the course

AI – Introduction to Deep Learning

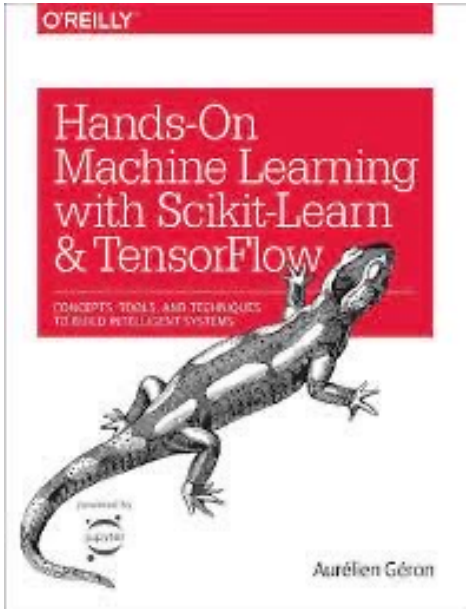
- Moodle's course name:
  - Artificial Intelligence : Introduction to Deep Learning
- Course code:
  - KMUIAU
- URL:
  - https://lms.univ-cotedazur.fr/course/view.php?id=17889
- Access password:
  - Intro2Deep!
- Prerequisites
  - Python Data Science Handbook, Jake VanderPlas, 2017, O'REILLY (Part 1 to 4)
  - Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, Aurélien Géron, September 2019, O'REILLY (Part I)



Part I to 4
Is assumed to be know

Syllabus – Advanced ML

- Michel Riveill
  1. Deep learning – General principles
  2. ML - An introduction to Natural Language Processing
  3. Deep learning - Multi-Layers perceptron
  4. ML - Recommender Systems
  5. Deep learning - Recurrent Neural Network
     - Test during of 5th class, 25% of total grade
- Diane Lingrand
  6. Deep learning - Convolutional Neural Network
  7. Deep learning – Model Explainability
  8. Deep learning - Reinforcement Learning
     - Test during of 8th class, 25% of total grade



Part I – is assumed to be know

Part II – this lecture

Organisation

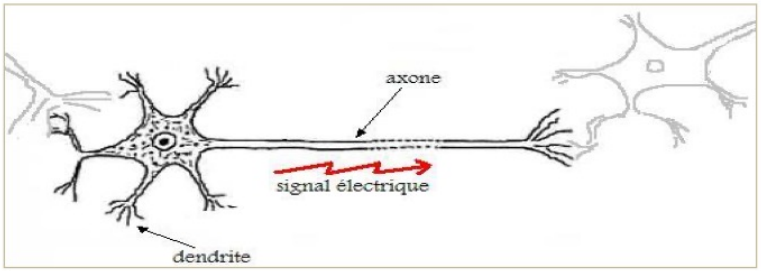
- Labs
  - Using Python Jupyter notebook and the libraries: tensorflow-keras, sklearn, numpy, pandas, nltk and matplotlib
  - On your workstation or via the internet with Google C

- Evaluation
  - Course attendance
    - Upload your lab each week
  - 2 MCQ (50 % of the grade)
  - 2 or 3 labs (50 % of the grade)

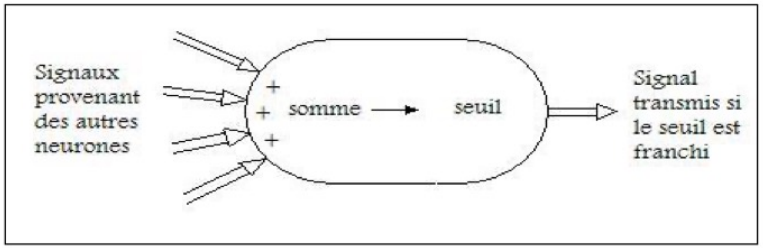

From biological neuron
to a multi-layer perceptron interpretation

Biological metaphor and mathematical transposition
Simple perceptron - invented in 1943 by McCulloch and Pitts

Biological metaphor and mathematical transposition Simple perceptron - invented in 1943 by McCulloch and Pitts



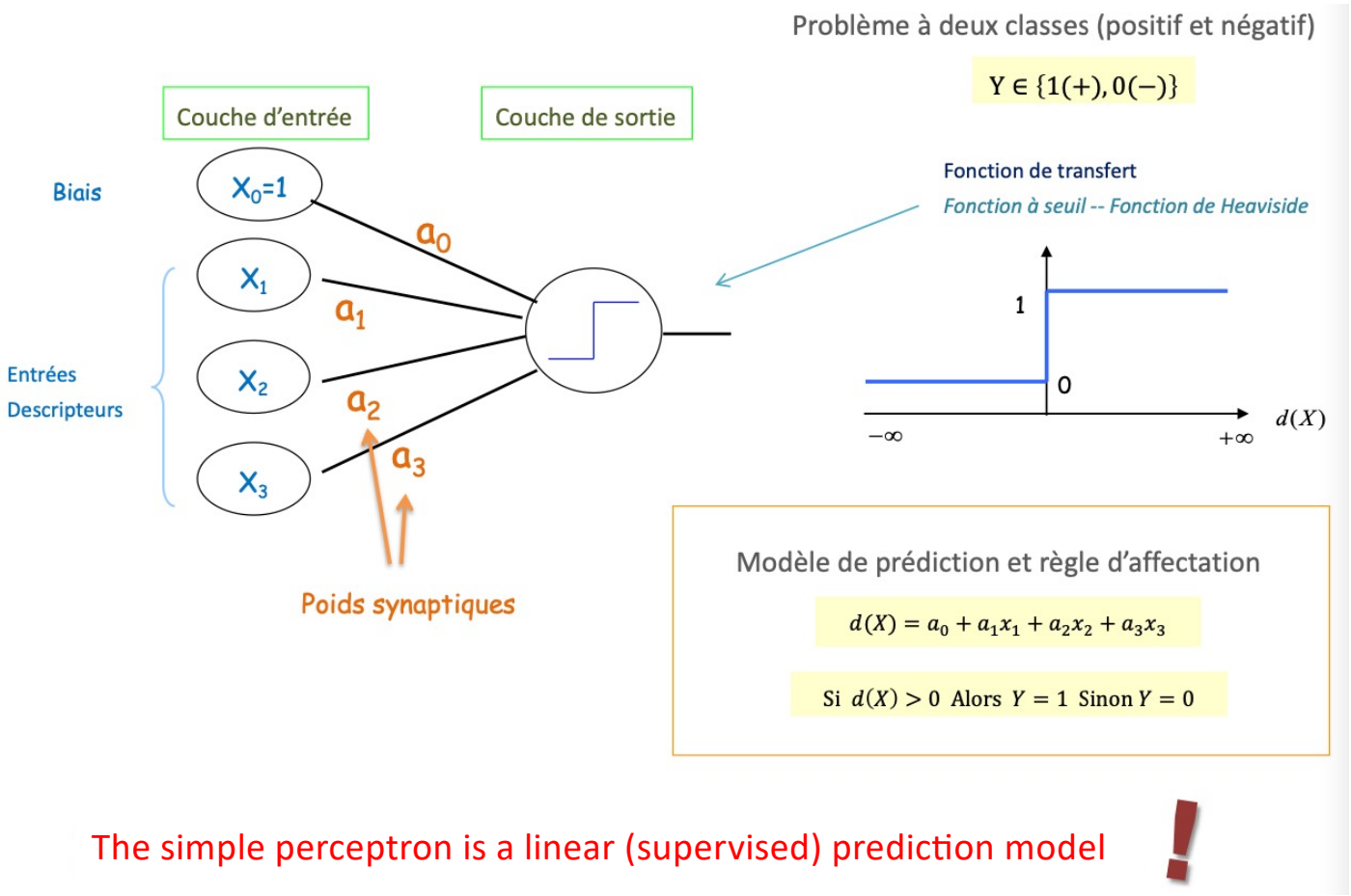
Key idea to remember



Key steps

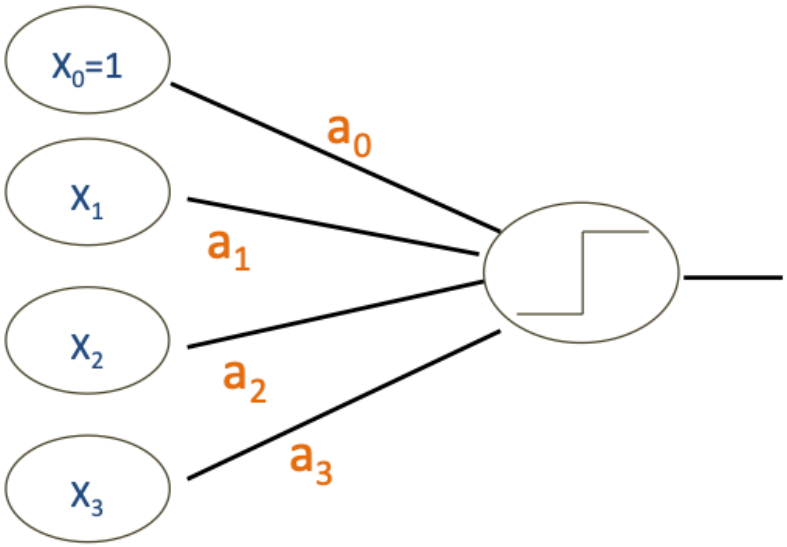
- Receiving information (signal)
- Activation + Processing (simple) by a neuron
- Transmission to other neurons (if threshold is crossed)
- In the long run: reinforcement of certain links -> Learning

Mc Colluch and Pitts model
The simple perceptron



The simple perceptron is a linear (supervised) prediction model

Learning



How to calculate synaptic weights from a (X, y) data file

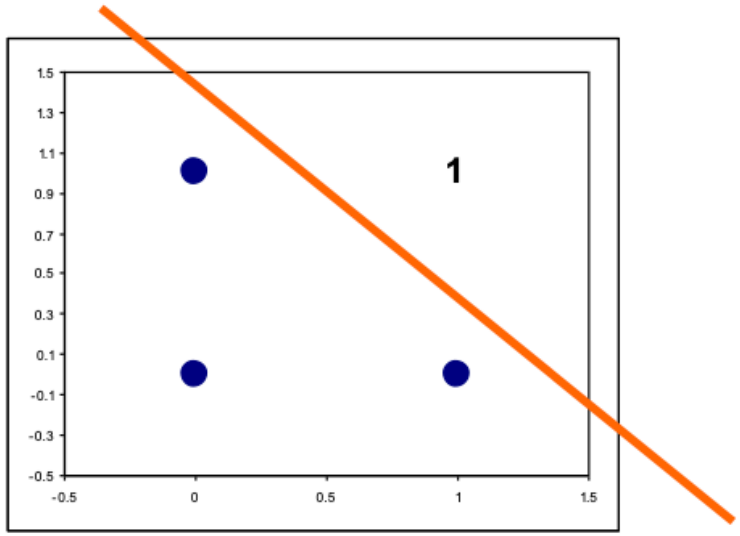
We can make the parallel with the regression and the least squares

1. Which criterion to optimize ? → minimizing the prediction error
2. How to optimize ? → incrementally

Example
Learning the AND function (logical AND)

X1	X2	Y
0	0	0
0	1	0
1	0	0
1	1	1

The data



Representation in the plane

How does the network learn?

- Iterative learning = "Hot and Cold" kid's game for finding a hidden object.
- In this game, the "hidden object" is the best possible model.

How does the network learn?

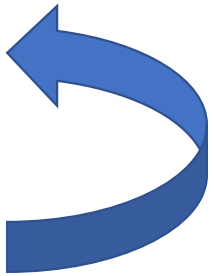
- Iterative learning = "Hot and Cold" kid's game for finding a hidden object.
- In this game, the "hidden object" is the best possible model.

- You start by choosing a **random value**
- You **submit the value to the model** and wait for the result
- The **system tells you** what the loss is.

How does the network learn?

- Iterative learning = "Hot and Cold" kid's game for finding a hidden object.
- In this game, the "hidden object" is the best possible model.

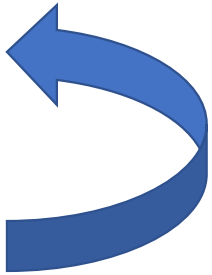
- You start by choosing a **random value**
- You **submit the value to the model** and wait for the result
- The **system tells you** what the loss is.
- Then, you'll **correct the initial value** and try again



How does the network learn?

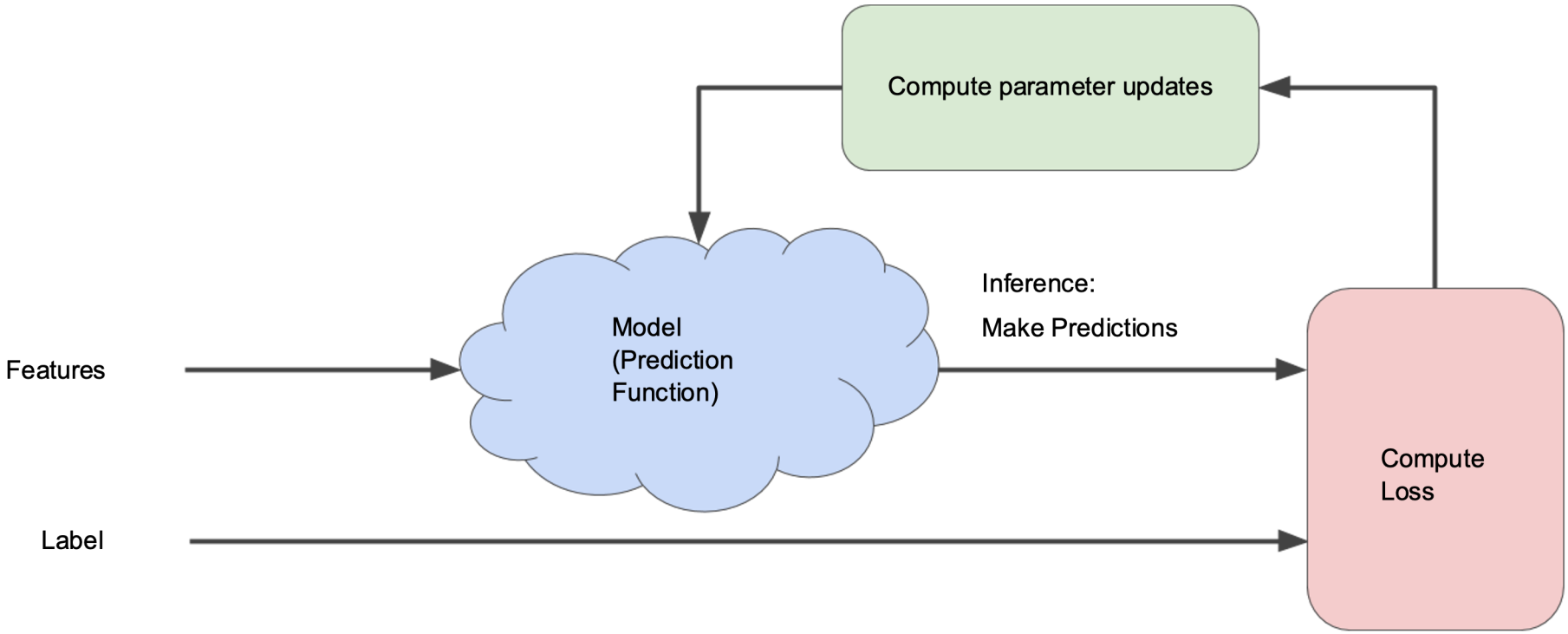
- Iterative learning = "Hot and Cold" kid's game for finding a hidden object.
- In this game, the "hidden object" is the best possible model.

- You start by choosing a **random value**
- You **submit the value to the model** and wait for the result
- The **system tells you** what the loss is.
- Then, you'll **correct the initial value** and try again



- OK, you're getting warmer. Actually, if you play this game right, you'll usually be getting warmer.

How does the network learn?



The real trick to the game is trying to **find the best possible** model **as efficiently as possible.**

The learning Algorithm

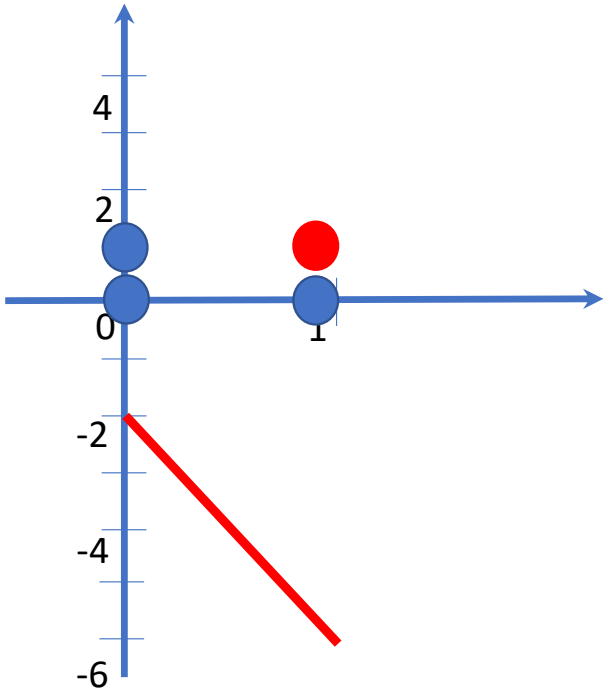
1. Randomly initialize the synaptic weights
2. Randomly shuffle the observations
3. Run the observations one by one and calculate the prediction
4. Compute the sum of the prediction errors
5. Update the synaptic weights
   - $a_j \leftarrow a_j + \Delta a_j \; with \; \Delta a_j = \eta(y - \hat{y})x_j$
     - x_j : signal
     - $(y - \hat{y})$: possible error
     - η: learning rate: too small slow convergence, too large oscillation
6. If convergence condition is not met, we start again at step 3

The learning Algorithm

1. Randomly initialize the synaptic weights
2. Randomly shuffle the observations
3. Run the observations one by one and calculate the prediction
4. Compute the sum of the prediction errors
5. Update the synaptic weights
6. If convergence condition is not met, we start again at step 3
   - No more corrections made by passing everyone
   - The global error does not decrease "significantly" anymore
   - The weights are stable
   - A maximum number of iterations is fixed
   - We set a minimum error to reach

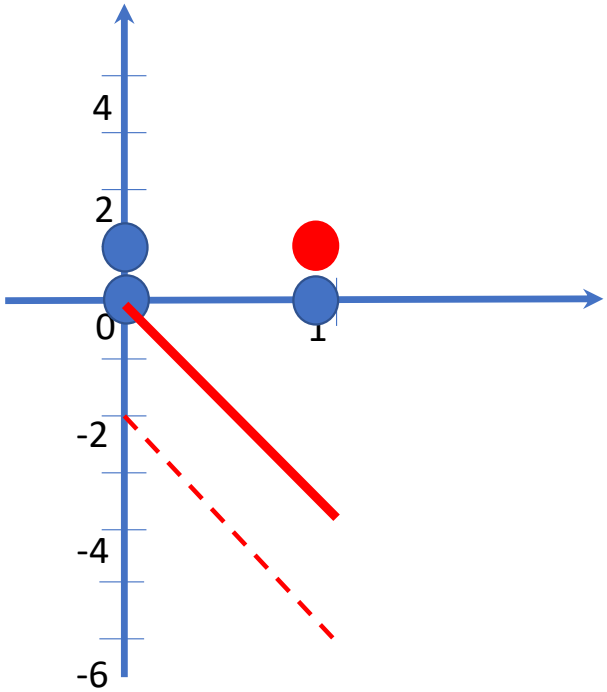
Exemple AND

- Random initialization of weights:
  - $a_0 = 0.1$
  - $a_1 = 0.2$
  - $a_2 = 0.05$
- Border :
  - $0.1\,x_0 + 0.2x_1 + 0.05x_2 = 0$
  - By convention x_0 is always 1
    → $x_2 = -4.0x_1 - 2$
    → $x_1 = 0$, the $x_2 = -4$
    → $x_1 = 1$, the $x_2 = -6$



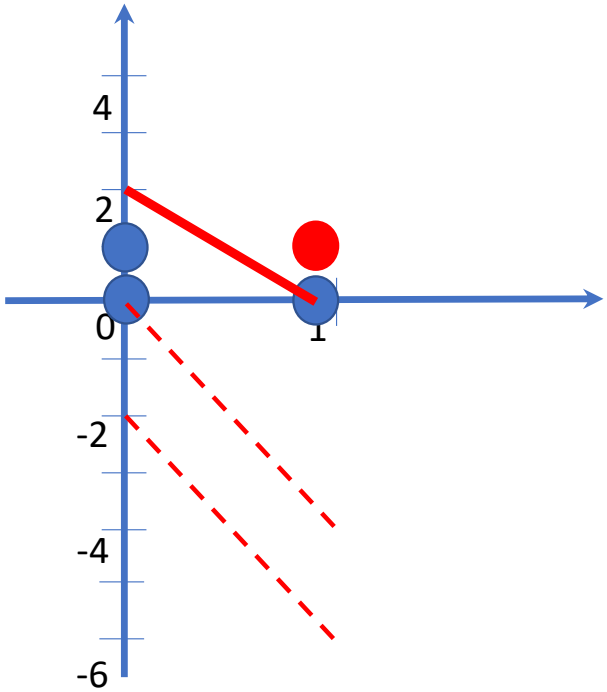
Exemple AND – treat one observation.

- Weights:
  - a_0 = 0.1
  - a_1 = 0.2
  - a_2 = 0.05
- One observation
  - x_0 = 1 (always 1), x_1 = 0, x_2 = 0, y=0
- Apply the model
  - $\widehat{result} = 0.1+0.2x_1 +0.05x_2$
  - → $\widehat{result} = 0.1+0.2*0 +0.05*0 =0.1$
  - → $\widehat{result} > 0$ ➔ $\hat{y} = 1$
- Compute the error
  - $\varepsilon = y - \hat{y} = 0 - 1 = -1$
- Update the weigh
  - $a_j \leftarrow a_j + \Delta a_j \; with \; \Delta a_j = \eta(y - \hat{y})x_j$
  - We choose η = 0.1
  - $a_0 = a_0 + \eta(-1)1 = a_0 - 0.1 = 0$
  - $a_1 = a_1 + \eta(-1)0 = a_1$
  - $a_2 = a_2 + \eta(-1)0 = a_2$
- New border
  - 0.0 x_0 +0.2x_1 +0.05x_2 = 0 ⇔ $x_2 = -4.0x_1$



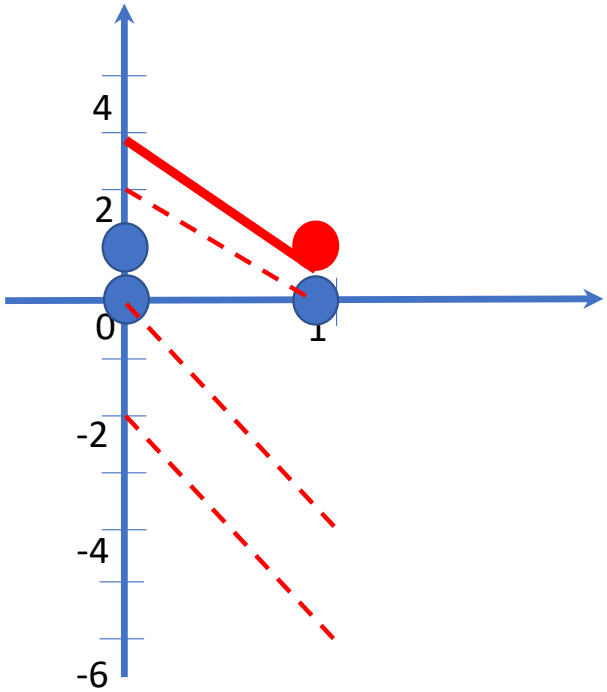
Exemple AND – treat another observation.

- Weights:
  - $a_0 = 0.0$
  - $a_1 = 0.2$
  - $a_2 = 0.05$
- One observation
  - $x_0 = 1$ (always 1), $x_1 = 1$, $x_2 = 0$, y=0
- Apply the model
  - $\widehat{result} = 0.1+0.2x_1 +0.05x_2$
  - → $\widehat{result} = 0.1+0.2*1 +0.05*0 =0.3$
  - → $\widehat{result} > 0$ ➔ $\hat{y} = 1$
- Compute the error
  - $\varepsilon = y - \hat{y} = 0 - 1 = -1$
- Update the weigh
  - $a_j \leftarrow a_j + \Delta a_j \; with \; \Delta a_j = \eta(y - \hat{y})x_j$
  - We choose $\eta = 0.1$
  - $a_0 = a_0 + \eta(-1)1 = a_0 - 0.1 = -0.1$
  - $a_1 = a_1 + \eta(-1)1 = a_1 - 0.1 = 0.1$
  - $a_2 = a_2 + \eta(-1)0 = a_2$
- New border
  - -0.1 x_0 +0.1x_1 +0.05x_2 = 0 ⇔ $x_2 = -2.0x_1 + 2.0$



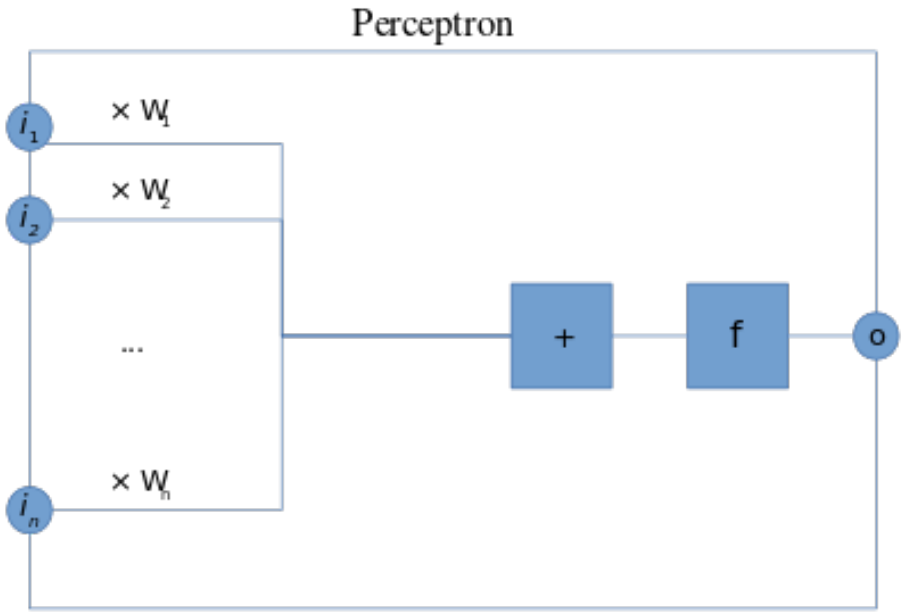
Exemple AND – treat another observation.

- Etc. etc. etc.
- Until : No more corrections made by passing everyone
- Final weights:
  - $a_0 = -0.45$
  - $a_1 = 0.31$
  - $a_2 = 0.16$
-



Original Perceptron- summary

- Invented in 1957 by Frank Rosenblatt
- The perceptron is a supervised learning algorithm that separates 2 classes
  - It is a formal neuron
  - Learning rule for determining synaptic weights

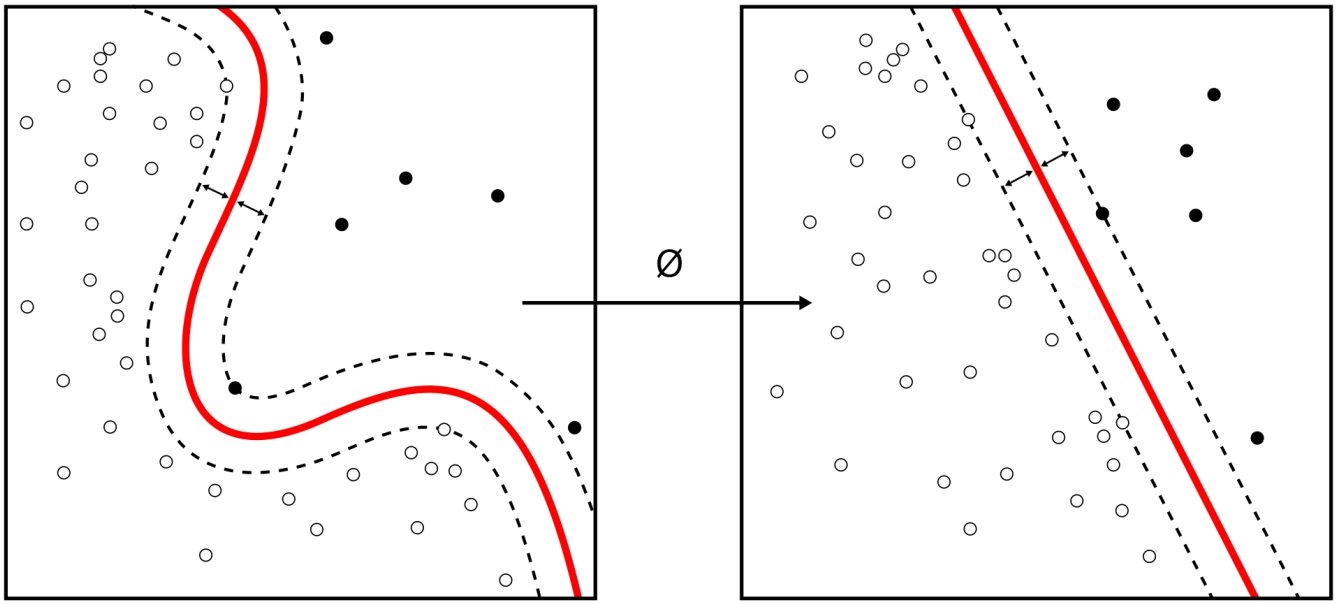


$$o = f\left(\sum_{k=1}^{n} i_k \cdot W_k\right)$$

f : activation function

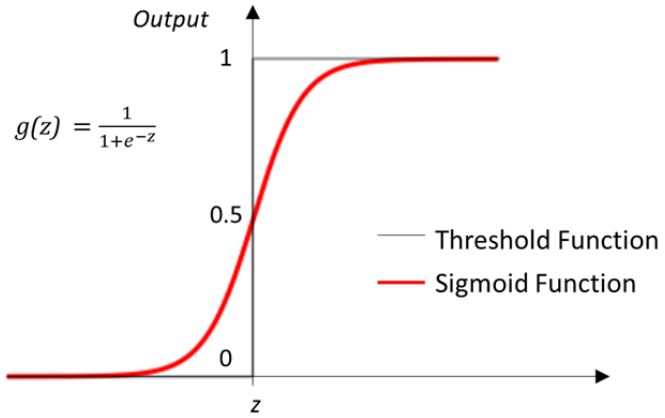
Perceptron theorem

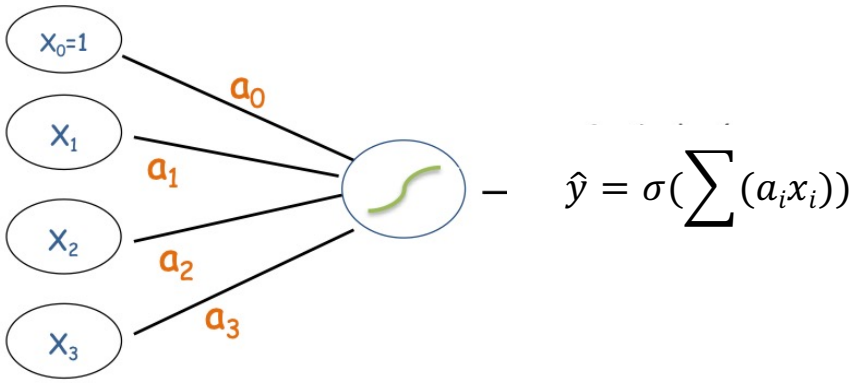
- If the problem is linearly separable, there are synaptic weights to separate the two classes.
  - Main problem : how many weights



Actual perceptron

- The perceptron proposes a ranking Y|X
- In some cases, we look for a probability P(Y|X)
- Modification of the activation function
  - The threshold function is replaced by the sigmoid function
    - $\sigma(x) = \frac{1}{1+e^{-x}}$ and $\frac{d\sigma}{dx} = \sigma(x)(1-\sigma(x))$
  - Modification of the decision rule
    - if $\sigma(\sum(a_i x_i))$ then y=1 else y=0
    - Equivalent to if $\sum(a_i x_i)$>0 then y=1 else y=0
    - Same criteria as threshold
  - Modification of the criterion to optimize
    - $E = \frac{1}{2}\sum(y-\hat{y})^2$
  - Updating the weights using gradient descent
    - $a_j = a_j - \eta(y - \hat{y})g'(\sum a_i x_i)x_j$



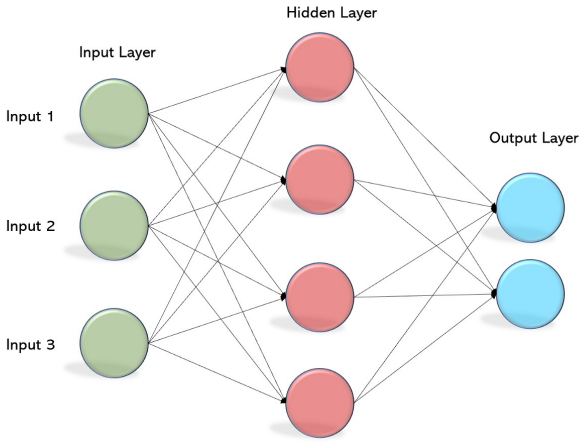


Multi-layer perceptron (MLP)

- A multilayer perceptron (MLP) is a class of fully connected artificial neural networks (ANNs).

- The term MLP is used ambiguously:
  - sometimes strictly to refer to networks composed of multiple layers of perceptrons (with threshold activation),
  - sometimes broadly to refer to any feedforward artificial neural network.

- For us: it is the second definition,
  as threshold neurons are very rarely used.

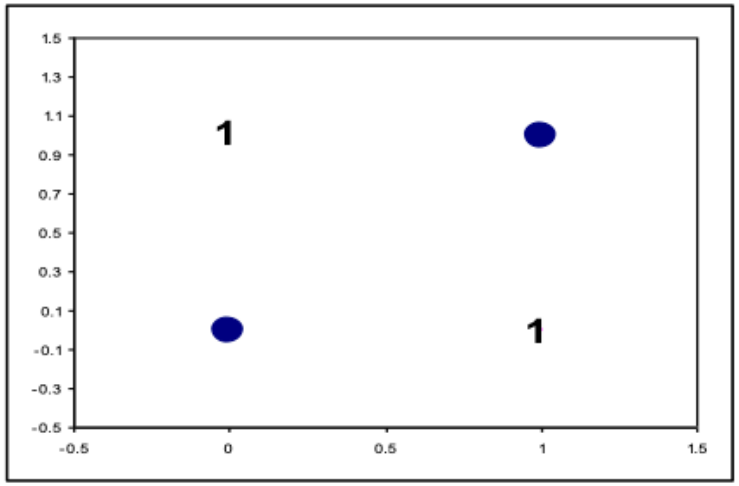


Why multi-layer perceptron
XOR problem

- A perceptron can only handle linearly separable problems

X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	0

Données



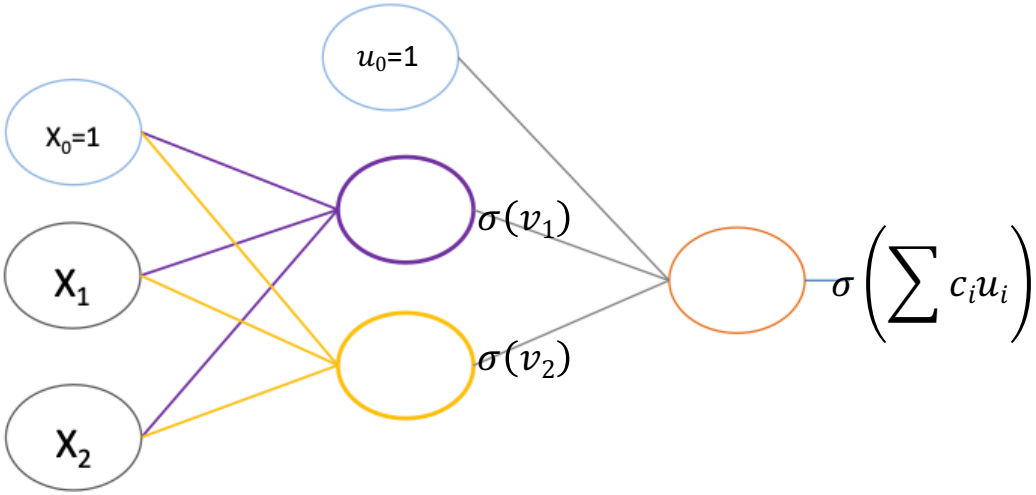
Non séparable linéairement

(Minsky & Papert, 1969)

- Multi-layer perceptron: A combination of linear separators produces a global non-linear separator (Rumelhart, 1986).

Multi-layer perceptron

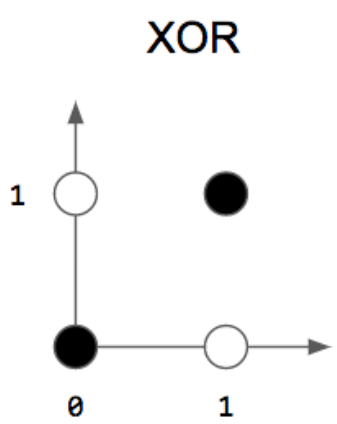
- First layer output
  - $\mathrm{u}_1 = \sigma(v_1)\ with\ v_1 = \sum a_i x_i$
  - $\mathrm{u}_2 = \sigma(v_2)\ with\ v_2 = \sum b_i x_i$
- Network output
  - $\hat{y} = \sigma(z)\ with\ z = \sum c_i u_i$

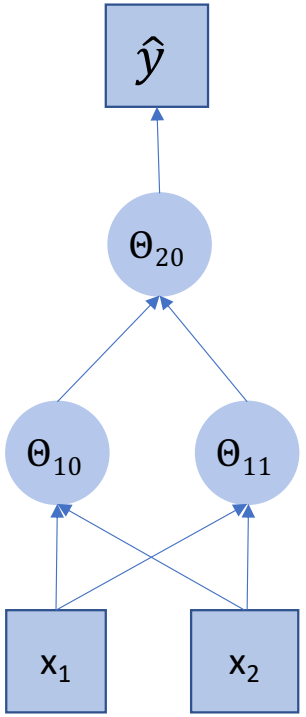


- Fundamental Property: The multilayer perceptron is capable of approximating any continuous function provided that the number of neurons in the hidden layer is appropriately fixed.

Multi-layer perceptron for non Linearly separable

x_1	x_2	x_1 and x_2
1	1	1
0	1	0
1	0	0
0	0	0



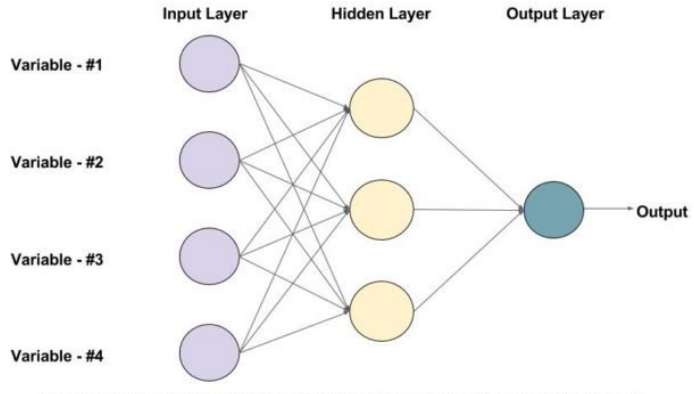


SOLUTION: ß = [[1, 1, -0.5],
[1, 1, -1.5],
[1, -2, -0.5]

i.e. Output is 1 if and only if
(x1+x2-0.5>0) + -2*(x1+x2-1.5>0) - 0.5 > 0

Linear regression neural networks

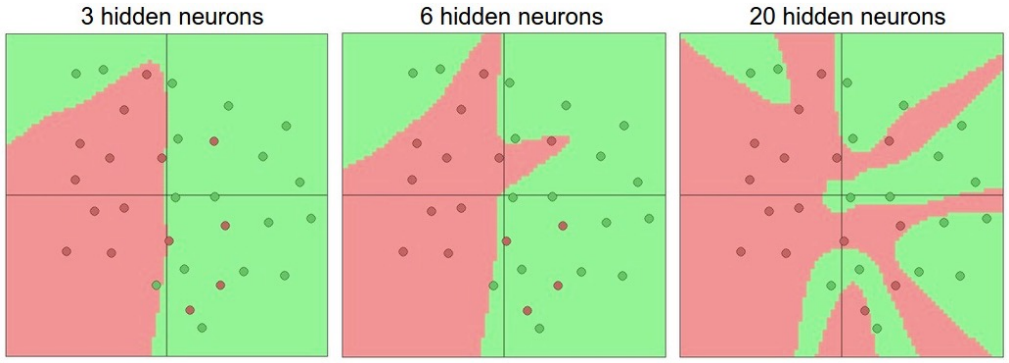
- What happens when we arrange **linear neurons** in a multilayer network?
  - Linear neurons: $\sigma(x) = x$
  - Nothing special happens.
  - The product of two linear transformations is itself a linear transformation



- $output = \sum_j w_j^{(2)} \left(\sum_i w_i^{(1)} x_i \right) = \sum_j \sum_i w_j^{(2)} w_i^{(1)} x_i = \sum_i \sum_j w_j^{(2)} w_i^{(1)} x_i$
- $output = \sum_i (\sum_j w_j^{(2)} w_i^{(1)}) x_i = \sum_i \hat{w}_i x_i = \hat{w}.x$

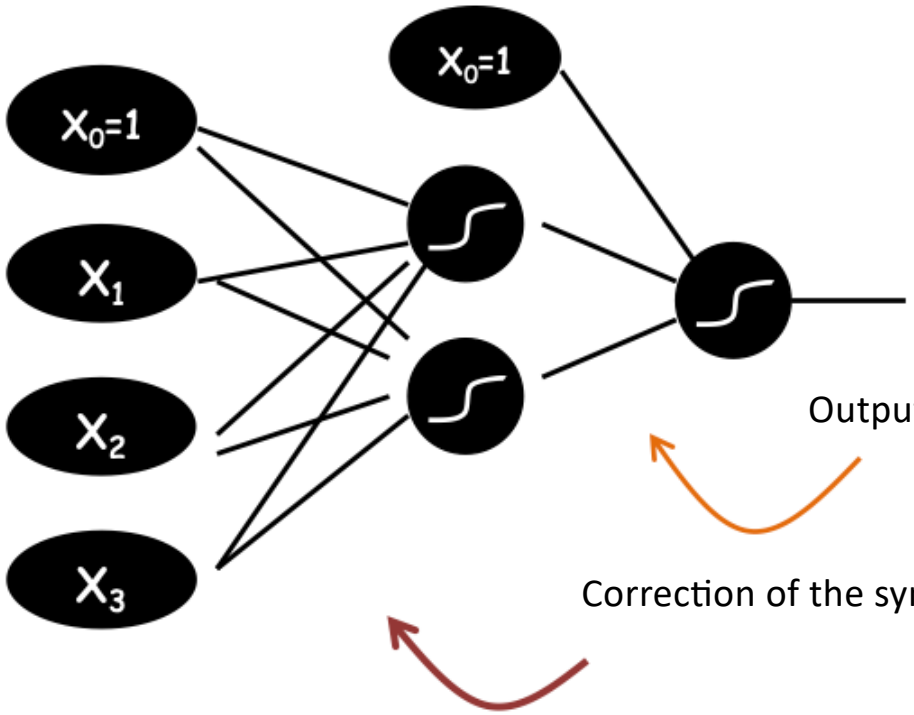
Neural Networks

- We must introduce non-linearities to the network
  - Non-linearities allow a network to identify complex regions in space
  - replace linear active function for non linear active function
- A one-layer cannot handle XOR
  - More layers can handle more complicated spaces – but require more parameters
  - Each node splits the feature space with a hyperplane
- A two-layer network can represent any convex region
  - provided that the number of neurons is large enough in the hidden layer



Learning process

Learning = gradient backpropagation



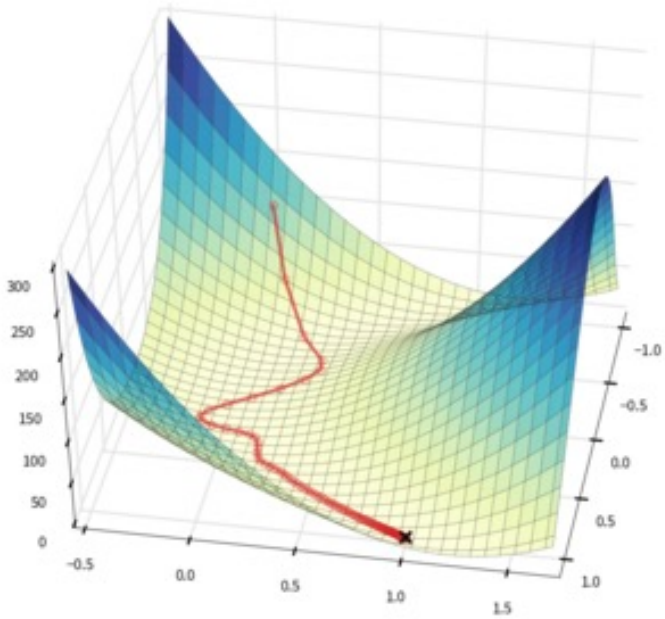
Output: measured error (for an individual)

Correction of the synaptic weights leading to the output layer

Propagation (backwards) of corrections in the intermediate layers

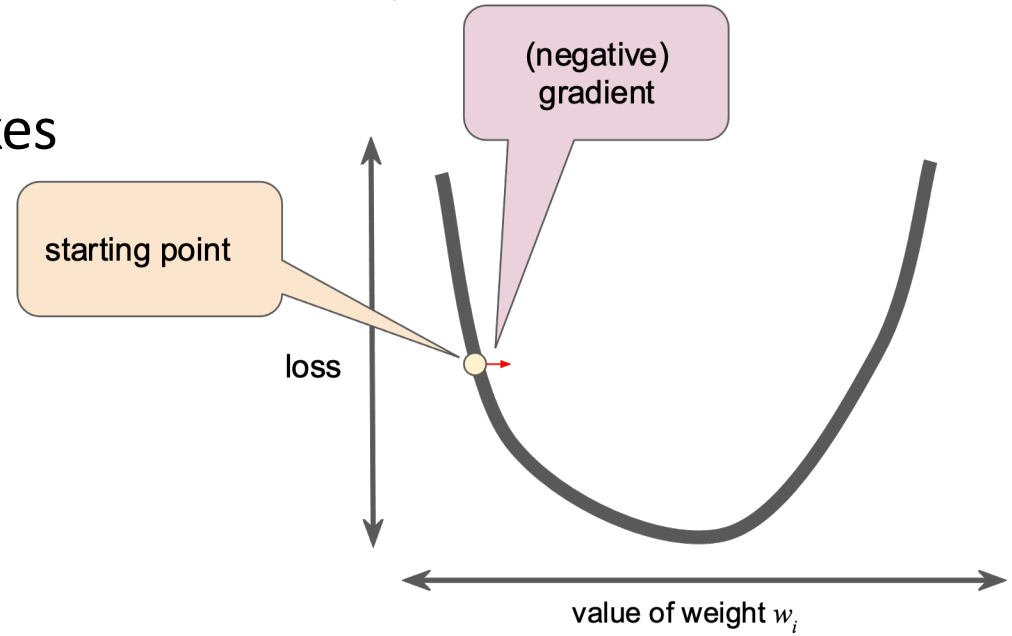
How does the network learn?

- In order to play this game we use “Gradient Descent”
  - Gradient Descent = help us to find the minimum of a function
- In Neural Network
  - Find the minimum of the loss function
- Loss function depend of the problem
  - Regression : MAE, MSE
  - Classification : CrossEntropy
  - Clusterisation : KL divergence



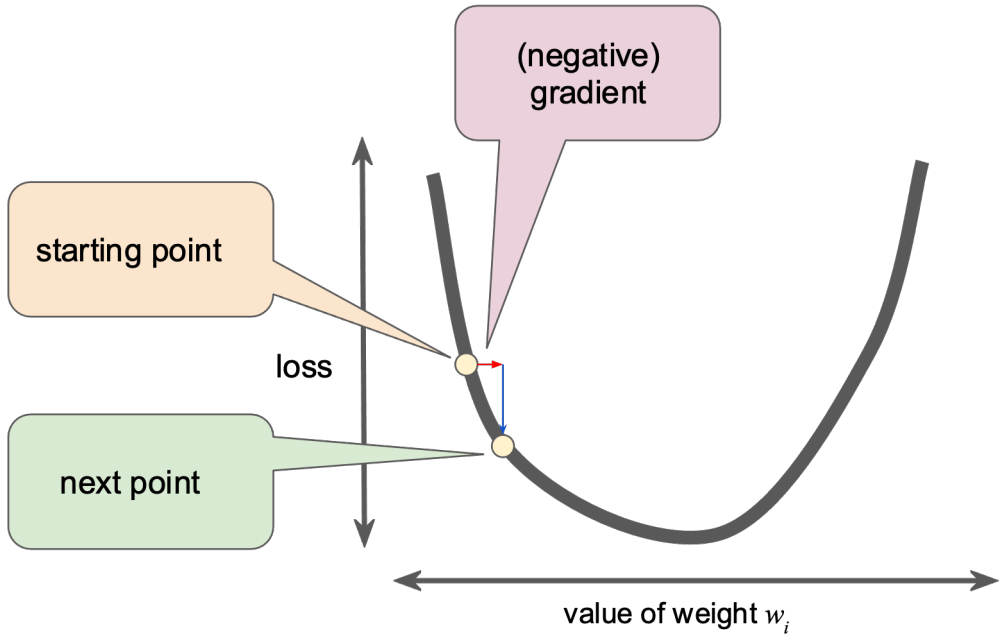
Gradient descent

- A gradient is a vector, so it has both of the following characteristics:
  - a direction
  - a magnitude
- The gradient always points in the direction of steepest increase in the loss function.
- The gradient descent algorithm takes a step in the direction of the negative gradient in order to reduce loss as quickly as possible.



Gradient descent

- To determine the next point along the loss function curve, the gradient descent algorithm adds some fraction of the gradient's magnitude to the current point.
- The gradient descent then repeats this process, edging ever closer to the minimum.



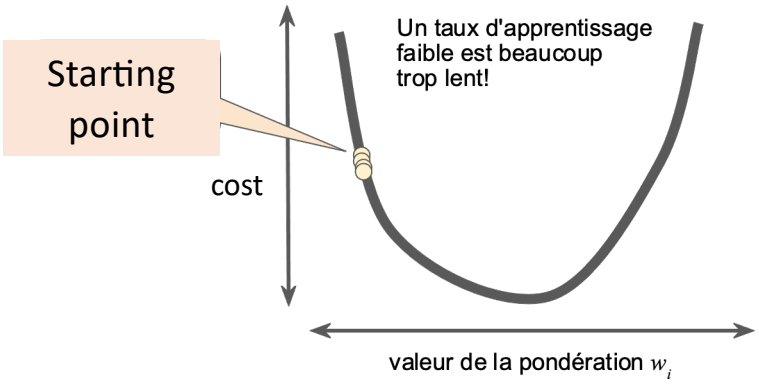
Gradient descent

- Objective: find a minimum of a differentiable function

- Main variables
  - Choose the initial point x_0
  - Choose the learning rate $\alpha \geq 0$ -- for the new point
  - Choose the tolerance level $\varepsilon \geq 0$ -- stop the algorithm

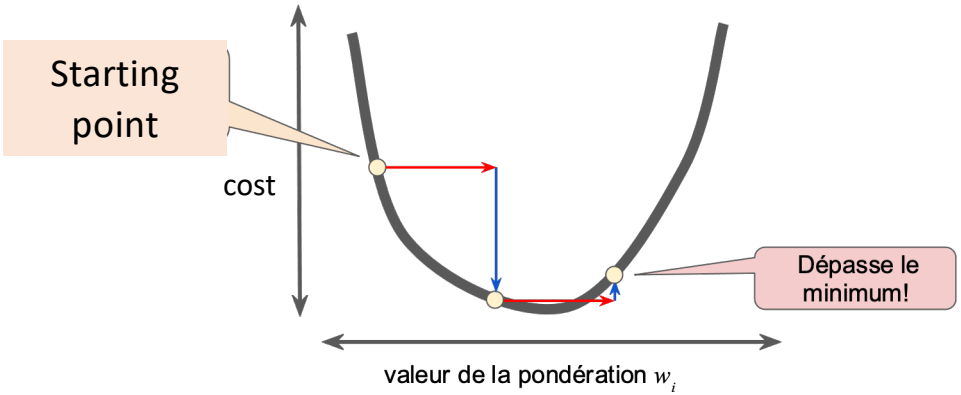
- Execute the following algorithm
  1. Calculate $\nabla f(x_k)$
  2. Stop if $\|\nabla f(x_k)\| \leq \varepsilon$
  3. Calculate the new value of x: $x_{k+1} = x_k - \alpha \nabla f(x_k)$
     - α is correct, if $f(x_{k+1}) < f(x_k)$

Gradient descent



If learning rate is too small
→ the minimum is found after a very long time

If learning rate is too big
→ the minimum is never found

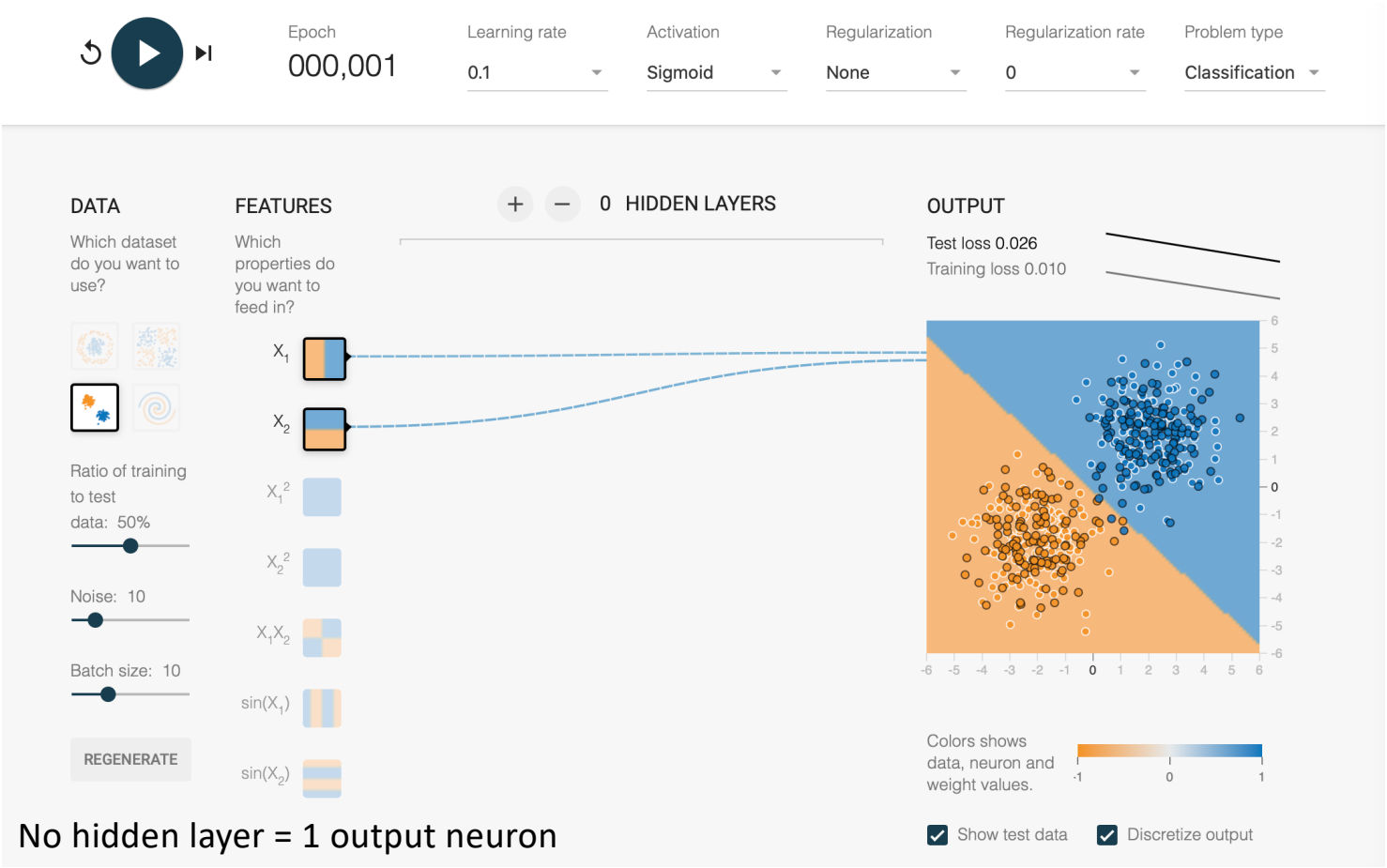


It's time to play

- Go to playground: **http://playground.tensorflow.org**
  - Choose classification problem
  - Fix:
    - Ratio: 50%
    - Noise: 10%
    - Regularization: None
    - Batch size: 10
    - Other paramters: Learning rate, Activation
  - When you fit the model, stop training when test loss < 0.05
    - make a screenshot (or note epoch number, training loss and test loss)

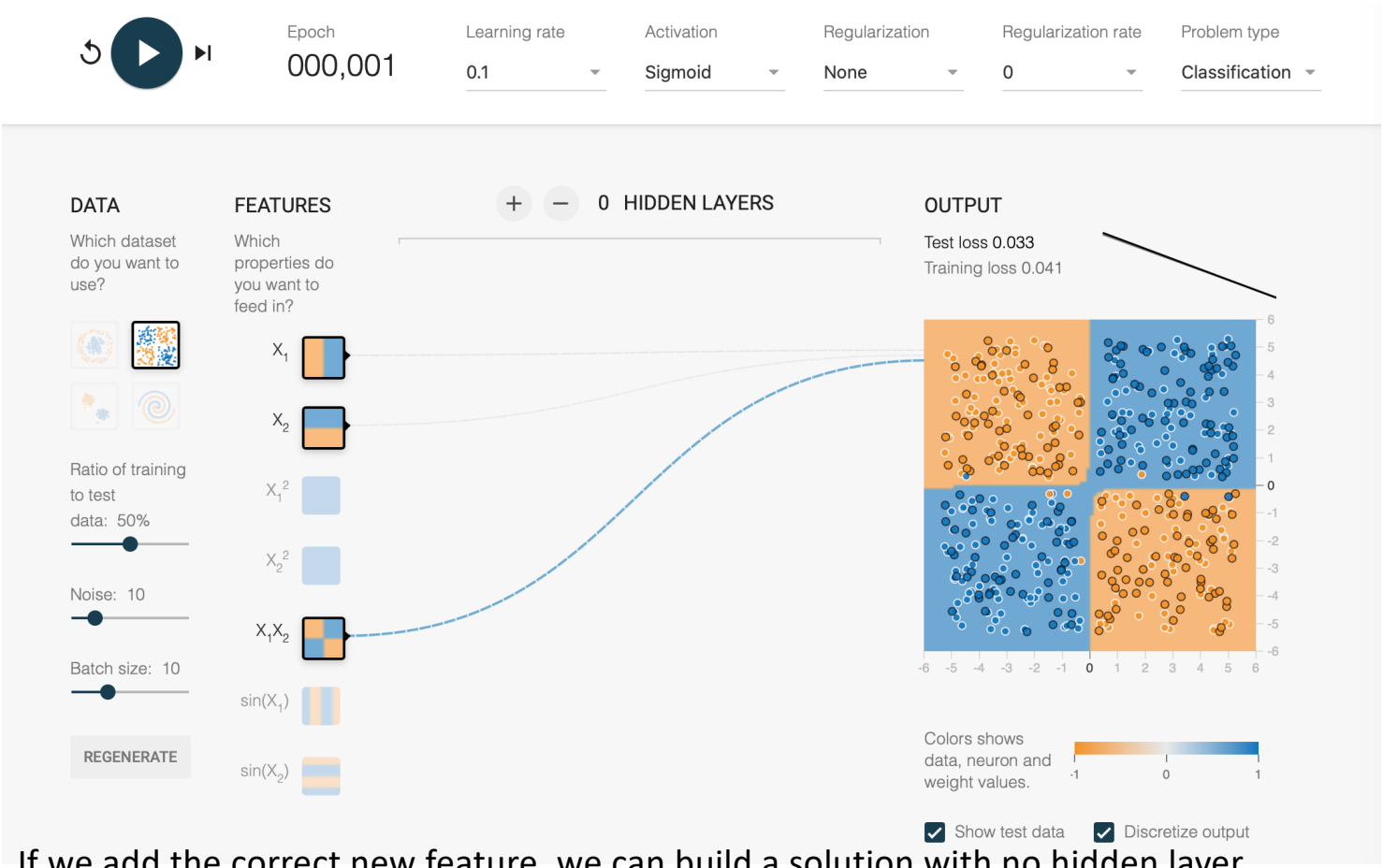
1. Choose a linearly separable dataset
   - Build the smaller network to fit them
2. Choose a non linearly separable dataset
   1. Add some extra feature in order to fit them with only one neuron
      - extra feature=x^2, xy, sin(x)
   2. Remove the extra feature and try to build the smaller network to fit them
   3. Is it possible to fit the function with linear activation function ?
   4. With the same example
      - Increase learning (0,1 → 1) rate and then, decrease learning (0,1 → 0,01)

linearly separable dataset



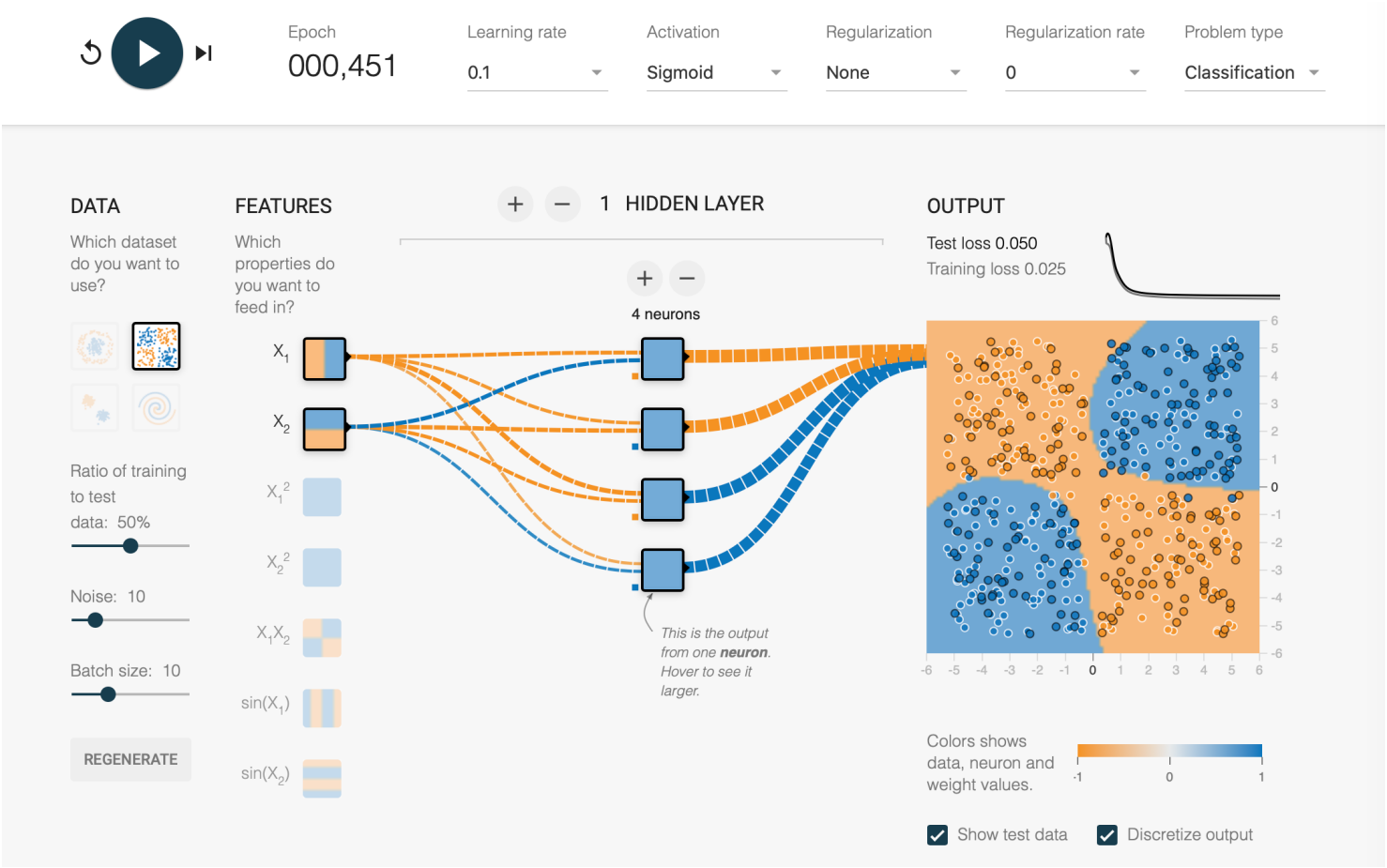
No hidden layer = 1 output neuron

Non linearly separable dataset



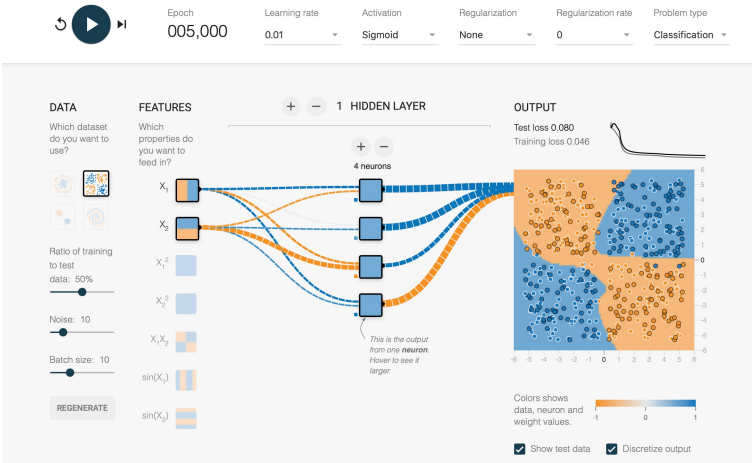
If we add the correct new feature, we can build a solution with no hidden layer i.e. with only 1 output neuron

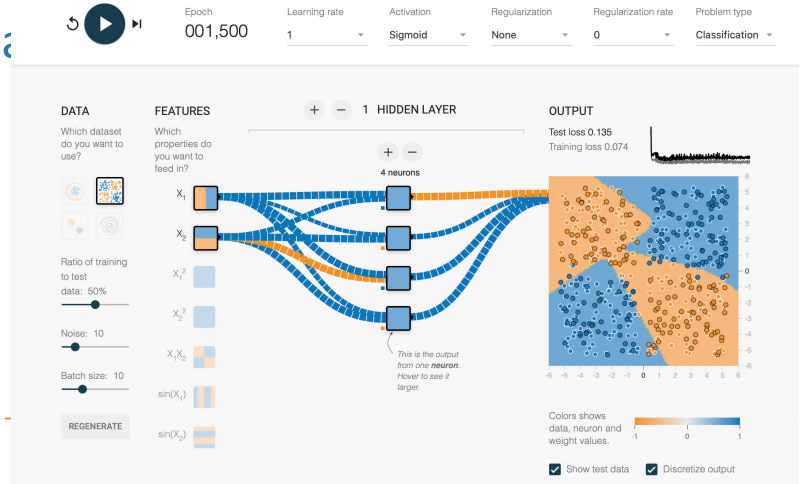
Non linearly separable dataset



Non linearly separable dataset

- Same example
  - Lr = 0,01 → after 5.000 epoch, loss = 0.08
  - Lr = 0,1 → 450 epochs, loss = 0.05
  - Lr = 1 → 1.500 epochs loss a





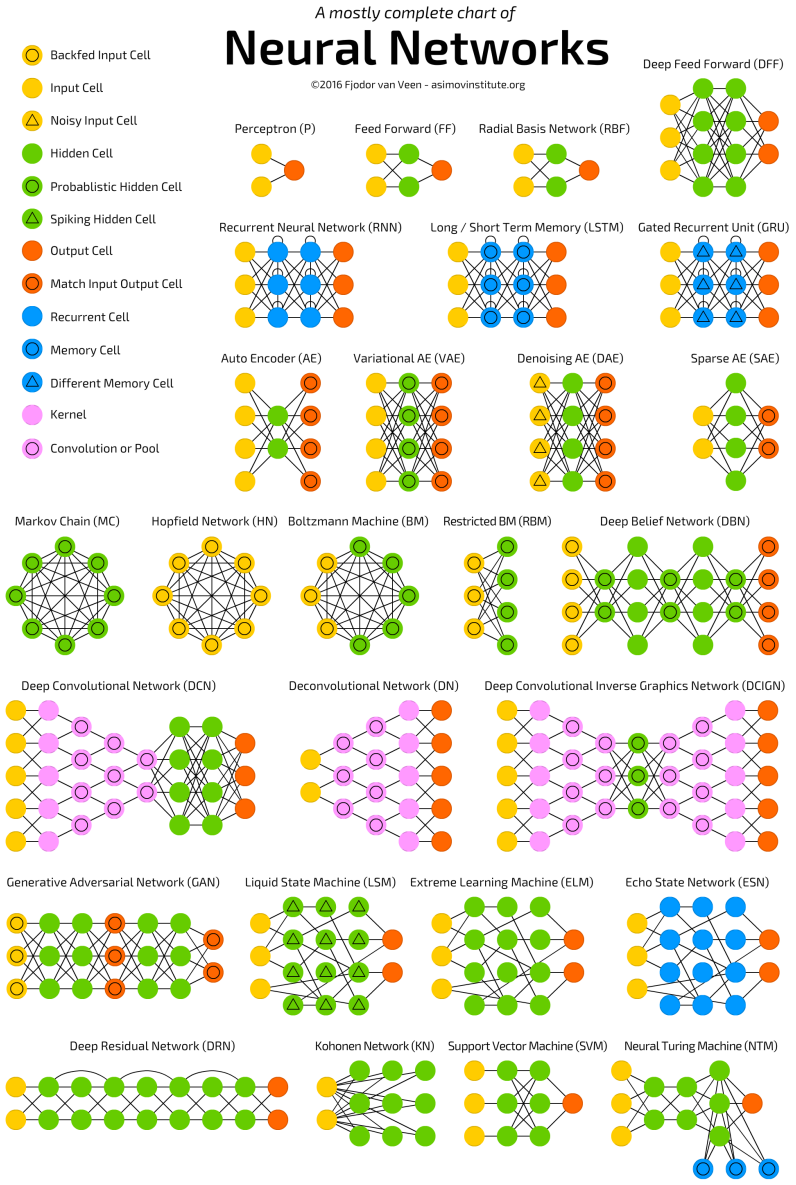
Multi-layer perceptron

- Advantages
  - Very accurate classifier (if well set up)
  - Incrementality
  - Scalability (ability to be implemented on large databases)
- Disadvantages
  - Black box model (causality between descriptor and variable to be predicted)
  - Difficulty of parameterization (number of neurons in the hidden layer)
  - Convergence problem (local optimum)
  - Danger of over-learning (too many neurons in the hidden layer)

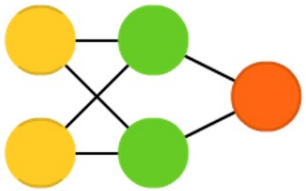
Main neural network architectures

Topologies of Neural Networks

http://www.asimovinstitute.org/neural-network-zoo/



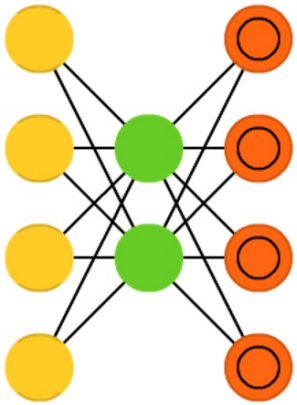
Feed forward neural networks (FF or FFNN) or MLP



Original paper: *Rosenblatt, Frank. "The perceptron: a probabilistic model for information storage and organization in the brain." Psychological review 65.6 (1958): 386.*

- Feedforward:
  - No loops
  - Very simple architecture: input → [hidden layers]$^{+}$ → output
- Supervised networks use a "teacher"
  - The desired output for each input is provided by user

Autoencoders (AE)

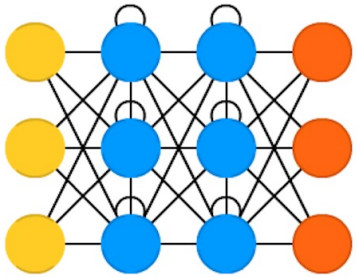


Original paper: *Bourlard, Hervé, and Yves Kamp. "Auto-association by multilayer perceptrons and singular value decomposition." Biological cybernetics 59.4-5 (1988): 291-294.*

The basic idea behind autoencoders is to compress information automatically

Unsupervised networks find hidden statistical patterns in input data

Recurrent neural networks (RNN)



Original paper: *Elman, Jeffrey L. "Finding structure in time." Cognitive science 14.2 (1990): 179-211.*

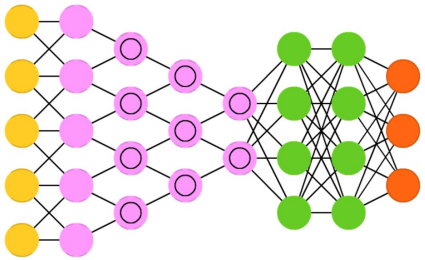
Add memory state to a traditional neuron

They have connections between passes, connections through time.

Neurons are feed information not just from the previous layer but also from themselves from the previous pass.

Convolutional neural networks (CNN)

- In mathematics (in particular, functional analysis), convolution is a mathematical operation on two functions (f and g) that produces a third function (f * g) that expresses how the shape of one is modified by the other.
- The term convolution refers to both the result function and to the process of computing it.



Original paper: *LeCun, Yann, et al. "Gradient-based learning applied to document recognition." Proceedings of the IEEE 86.11 (1998): 2278-2324.*

CNN are quite different from most other networks. They are primarily used for image processing but can also be used for other types of input such as audio or text.

Each node only concerns itself with close neighbouring cells.


Deep learning in Python

Which library ?

- Some deep learning libraries under Python
  - Theano: provides an API for writing digital algorithms using the GPU
    - Not specialized in deep learning even if it offers many features for this purpose
    - kind of numpy GPU
  - Tensorflow: Originally developed by researchers and engineers from Google
    - Strong support for deep learning
  - Pytorch
  - Keras: an over-layer to different at deep learning library

- I’ve choose **Keras+Tensorflow** as backend

- Comparatifs: github.com/zer0n/deepframeworks

Keras properties

- Keras is an API designed for human beings
  - Minimizes the number of user actions required for common use cases
  - Offers consistent & simple APIs
  - Keras is the official high-level API of TensorFlow
- Keras is multi-platform, multi-backend,
  - Develop in Python, R
  - On Unix, Windows, OSX
- Allow access to low-level API
  - Full access to TensorFlow API
- Keras is easy to learn and easy to use
  - Easy to use models in production
- **Three API**
  - Sequential
  - **Functional** ← our favorite one: it's used for all examples in the lecture
  - Subclassing

Sequential API
(only for sequential model – not enough for us)

```
1. from keras import models, layers

2. model = models.Sequential()
3. model.add(layers.Dense(16, input_dim=N_features, activation='relu'))
4. model.add(layers.Dense(8, activation='relu'))
5. model.add(layers.Dense(1))

6. model.compile(optimizer='sgd', loss=mse’)

7. model.fit(X_train, y_train, epochs=100, validation_split=0.33)

8. y_pred = model.predict(X_test)
```

Only sequential architecture

Functional API
(our favorite API)

```
1. from keras import models, layers

2. inputs = layers.Input(shape=(N_features, ))
3. x = layers.Dense(16, activation='relu')(inputs)
4. x = layers. Dense(8, activation='relu')(x)
5. outputs = layers. Dense(1)(x)

6. model = models.Model(inputs, outputs)
7. model.compile(optimizer='sgd', loss=mse')

8. model.fit(X_train, y_train, epochs=100, validation_split=0.33)

9. y_pred = model.predict(X_test)
```

All kind of architecture

Model subclassing
(You can work without)

```
1.  from keras import models, layers

2.  class MyModel(models.Model):
3.      def __init__(self):
4.          super(MyModel, self).__init__()
5.          self.dense1 = layers.Dense(16, activation='relu')
6.          self.dense2 = layers.Dense(8, activation='relu')
7.          self.dense3 = layers.Dense(1)
8.      def call(self, inputs):
9.          x = self.dense1(inputs)
10.         x = self.dense2(x)
11.         return self.dense3(x)

12. model = MyModel()
13. model.compile(optimizer='sgd', loss='mse')
14. model.fit(X_train, y_train, epochs=100, validation_split=0.33)
15. y_pred = model.predict(X_test)
```

Used to build new kind of neurons
Not in the scope of this lecture


MLP
Practical issues

Choose the network architecture

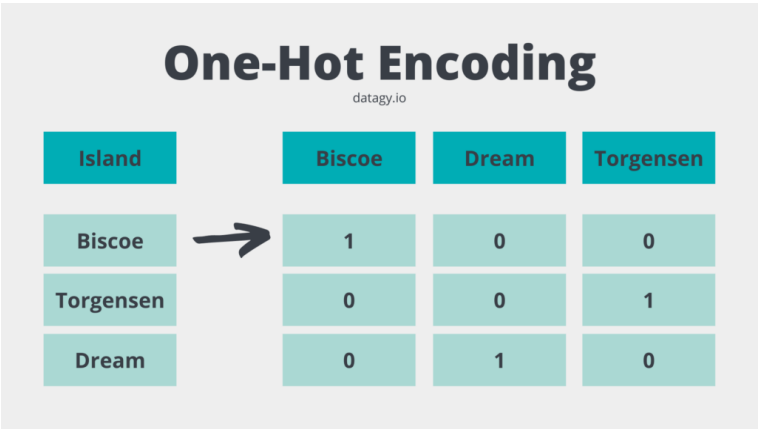
- Input layer → number of feature
  - inputs = layers.Input(shape=(N_features,))

- Hidden layer → number of layers / number of neurons by layer
- → activation funtion: relu / sigmoid / tanh
  - x = layers.Dense(16, activation='relu')(inputs)
  - x = layers.Dense(8, activation='relu')(x)

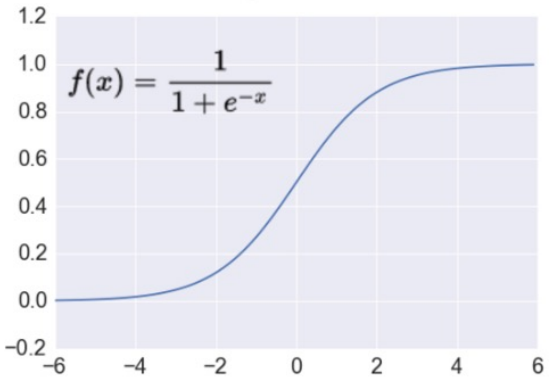
Choose the network architecture

Output layer: depends on the problem

- Regression
  - → One neuron by function to fit
  - → activation function: linear
  - outputs = layers.Dense(1, activation='linear')(x)
- Classification
  - 2 classes = 1 neuron
  - → activation function: sigmoid
  - outputs = layers.Dense(1, activation='sigmoid')(x)
    - $\hat{y} \in [0,1]$
  - N classes = N neurons
  - → activation funtion: sofmax
  - outputs = layers.Dense(n_classes, activation='softmax')(x)
    - one hot encode y
    - $\hat{y}$ is a vector, with a prob of each classes
  - Please use sklearn OneHotEncoder
  - Not pandas.dummies or to_categorical (error prone)



Activation: Sigmoid



http://adilmoujahid.com/images/activation.png

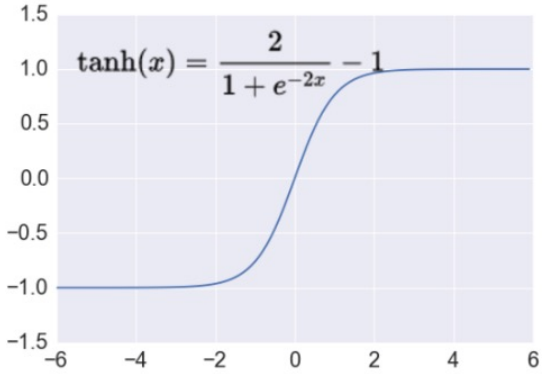
Takes a real-valued number and
"squashes" it into range between 0 and 1.

$$R^n \rightarrow [0,1]$$

\+ Nice interpretation as the **firing rate** of a neuron
- 0 = not firing at all
- 1 = fully firing

\- Sigmoid neurons **saturate** and **kill gradients**, thus NN will barely learn
- when the neuron's activation are 0 or 1 (saturate)
  gradient at these regions almost zero
  almost no signal will flow to its weights
  if initial weights are too large then most neurons would saturate

Activation: Tanh



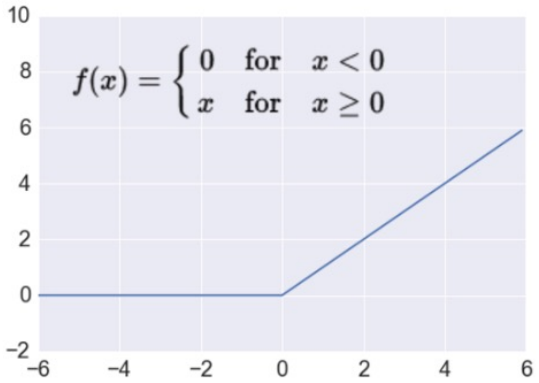
http://adilmoujahid.com/images/activation.png

Takes a real-valued number and "squashes" it into range between -1 and 1.

$$R^n \rightarrow [-1,1]$$

- 🙁Like sigmoid, tanh neurons **saturate**
- 🙂Unlike sigmoid, output is **zero-centered**
- Tanh is a **scaled sigmoid**: $\tanh(x) = 2sigm(2x) - 1$

Activation: ReLU



http://adilmoujahid.com/images/activation.png

Takes a real-valued number and thresholds it at zero $f(x) = \max(0, x)$

Most Deep Networks use ReLU nowadays

Trains much **faster**
- accelerates the convergence of SGD
- due to linear, non-saturating form

Less expensive operations
- compared to sigmoid/tanh (exponentials etc.)
- implemented by simply thresholding a matrix at zero

More **expressive**

Prevents the **gradient vanishing problem**

Hyper-parameters fitting

- There are many other parameters to set in order to efficiently converge the network to the slightest error while maintaining a strong generalization capacity.

- Batch size
- Optimizer
- Number of epochs
- Loss / Metrics
- Deal with underfitting and overfitting
- Vanishing gradient
- Etc.

Another lecture

Conclusion

- Neural Network: a flexible model but also a black box model
  - Difficult to interpret
  - Regression
    - $output \in]-\infty, +\infty[$, an activation is not necessary
    - $output \in]-1, +1[$, you can use also tanh function
  - Classification
    - Output function = softmax
    - 2 classes (softmax = sigmoid)
      - Interpret $\hat{y}$ as probability of belonging to the class: P(C=c|X)
    - More than 2 classes, do the same i.e.
      - Sofmax: normalize the output probabilities: $\sum P(C = ci|X) = 1$
- Many libraries in many programming languages
- Tuning is crucial and not obvious