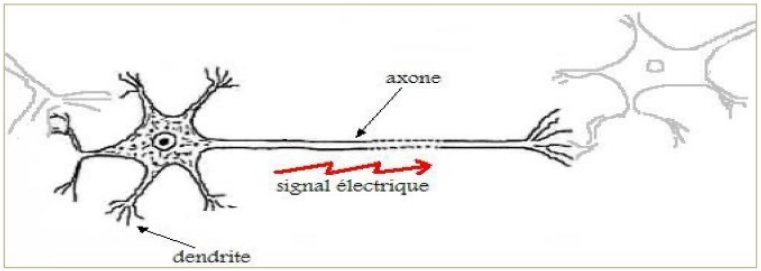
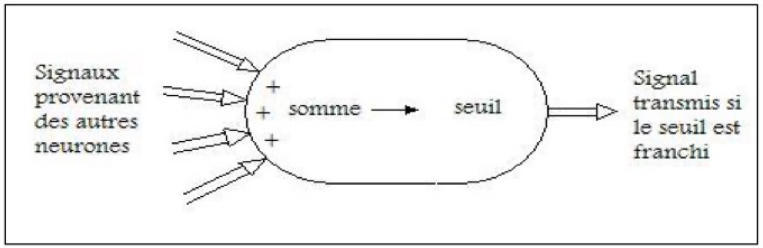
From biological neuron to a multi-layer perceptron interpretation

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



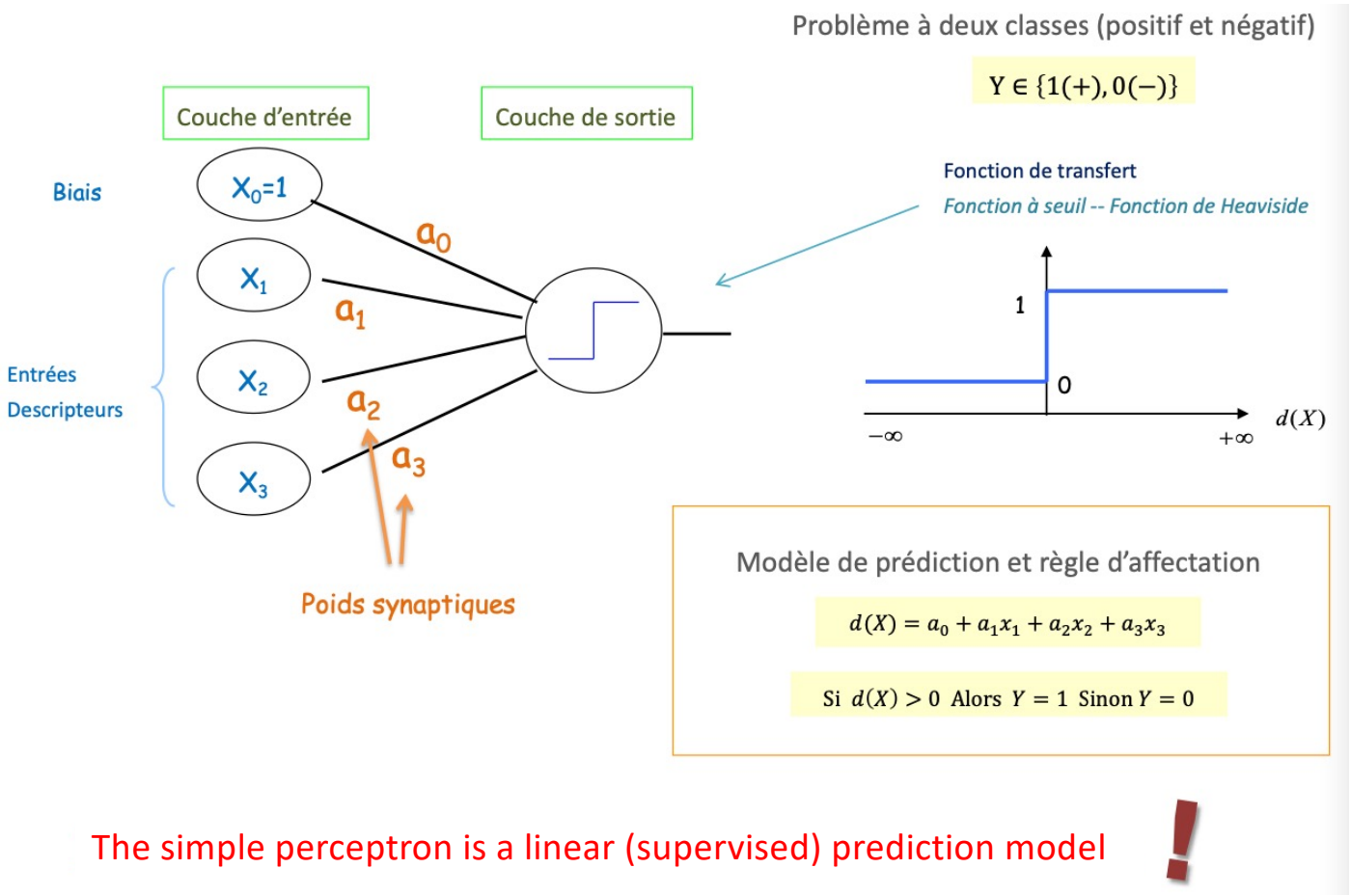
Key idea to remember



Key steps

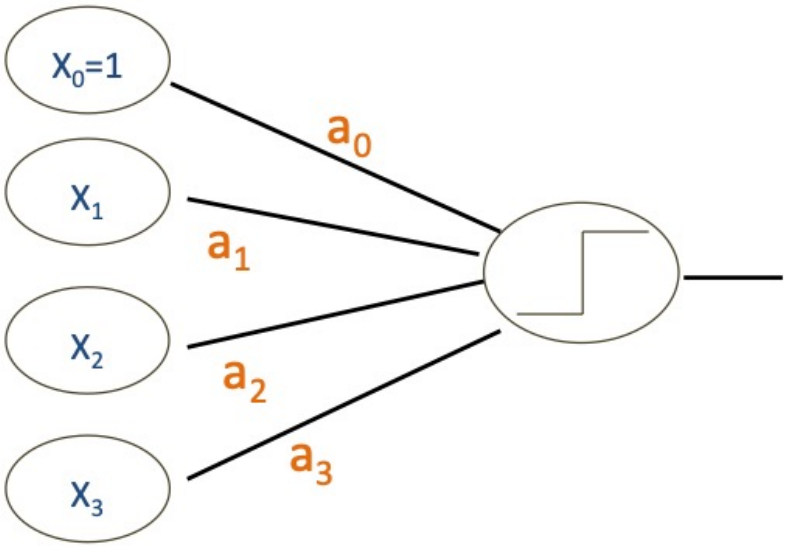
- Receiving information (signal)
- Activation + Processing (simple) by a neuron
- Transmission to other neurons (if threshold is crossed)
- In the long run: reinforcement of certain links -> Learning

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

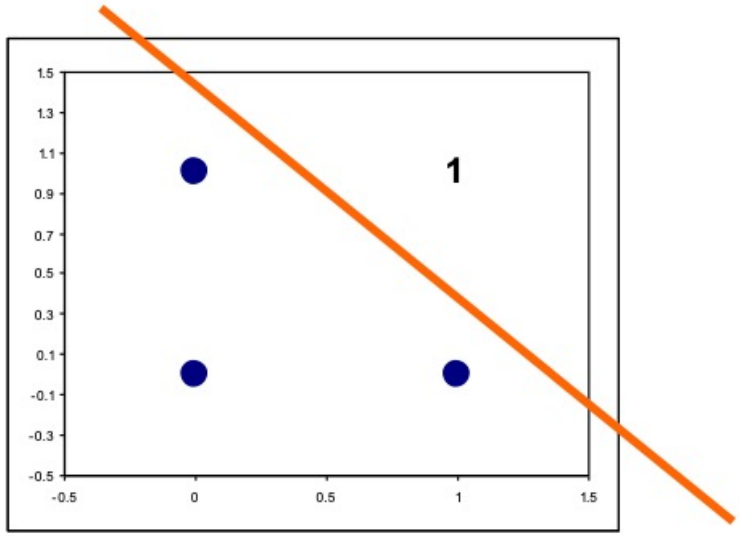
We can make the parallel with the regression and the least squares

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

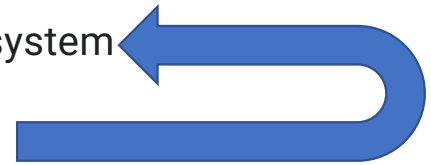
The data



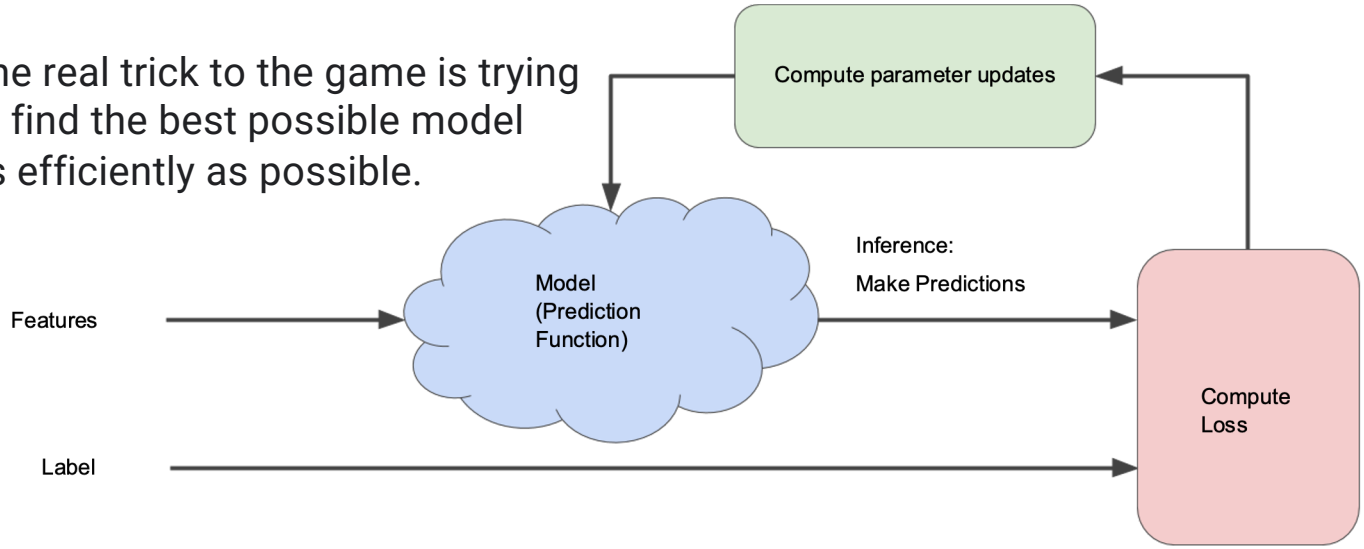
Representation in the plane

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.
- You choose a random value
- You submit the value to the model and wait for the system to tell you what the loss is.
- Then, you'll correct the initial value and try
- OK, you're getting warmer. Actually, if you play this game right, you'll usually be getting warmer.



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

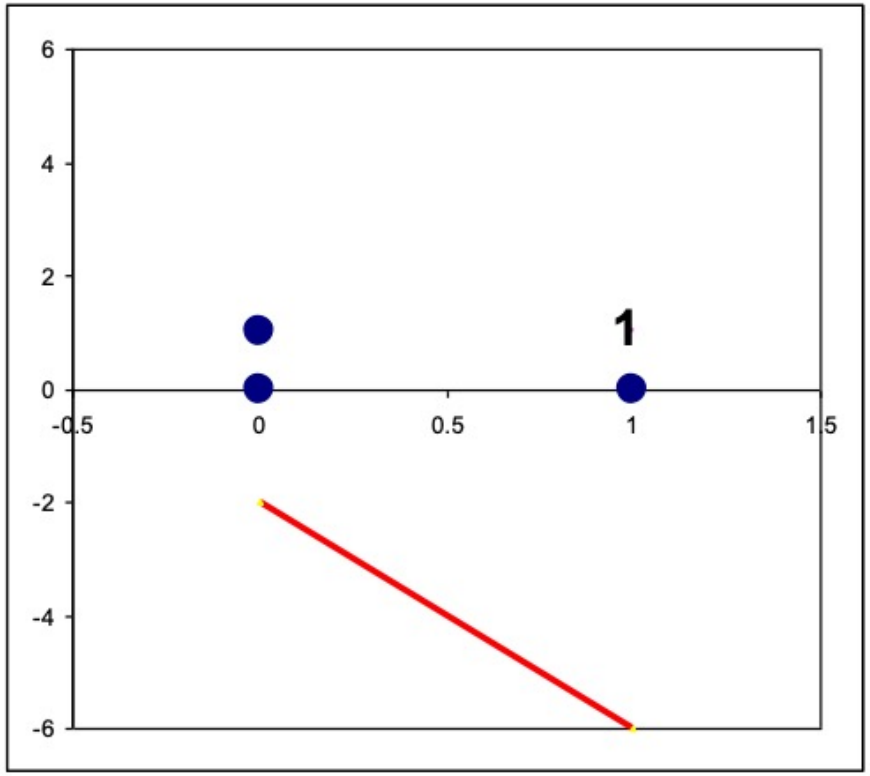


The learning Algorithm

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
   - $a_j \leftarrow a_j + \Delta a_j \; avec \; \Delta a_j = \eta(y - \hat{y})x_j$ (j = {0, 1}, here only 2 features)
     - x_j : input signal for weight j
     - $(y - \hat{y})$: 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
   - Convergence condition
     - 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 :
  - $0.1+0.2x_1 +0.05x_2 =0 \rightarrow x_2=-4.0x_1 -2$



Example AND

Observation à traiter

$$\begin{cases} x_0 = 1 \\ x_1 = 0 \\ x_2 = 0 \\ y = 0 \end{cases}$$

Valeur observée de Y et prédiction ne matchent pas, une correction des coefficients sera effectuée.

Appliquer le modèle

$$0.1\times1+0.2\times0+0.05\times0=0.1$$
$$\Rightarrow \hat{y}=1$$

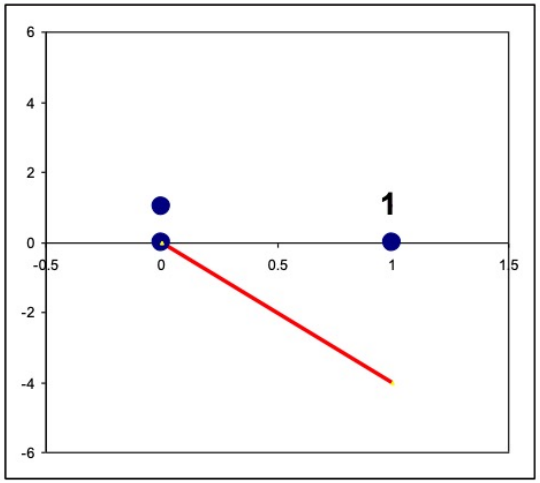
Màj des poids

$$\begin{cases} \Delta a_0 = 0.1\times(-1)\times1 = -0.1 \\ \Delta a_1 = 0.1\times(-1)\times0 = 0 \\ \Delta a_2 = 0.1\times(-1)\times0 = 0 \end{cases}$$

Signal nul ($x_1 = 0$, $x_2 = 0$), seule la constante a_0 est corrigée.

Nouvelle frontière :

$$0.0+0.2x_1+0.05x_2=0 \Leftrightarrow x_2=-4.0x_1+0.0$$



Example AND

Observation à traiter

$$\begin{cases} x_0 = 1 \\ x_1 = 1 \\ x_2 = 0 \\ y = 0 \end{cases}$$

Appliquer le modèle

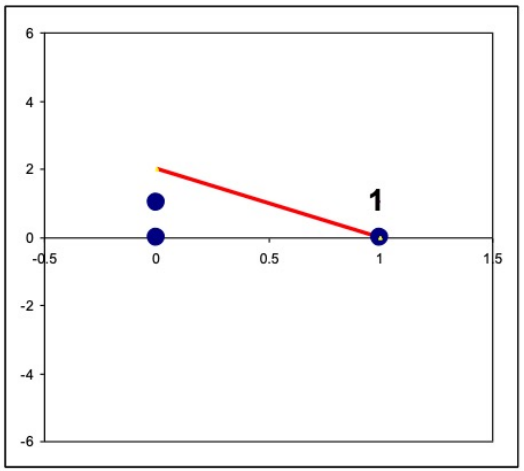
$$0.0 \times 1 + 0.2 \times 1 + 0.05 \times 0 = 0.2$$
$$\Rightarrow \hat{y} = 1$$

Màj des poids

$$\begin{cases} \Delta a_0 = 0.1 \times (-1) \times 1 = -0.1 \\ \Delta a_1 = 0.1 \times (-1) \times 1 = -0.1 \\ \Delta a_2 = 0.1 \times (-1) \times 0 = 0 \end{cases}$$

Nouvelle frontière :

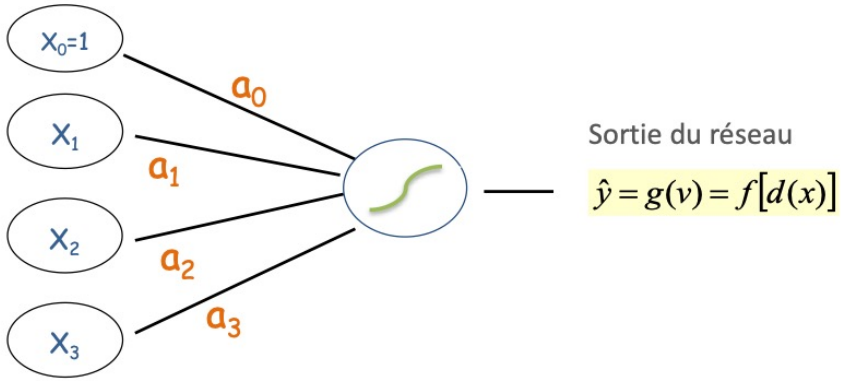
$$-0.1 + 0.1x_1 + 0.05x_2 = 0 \Leftrightarrow x_2 = -2.0x_1 + 2.0$$



More on the notebook perceptron

One more step towards today's neurons

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

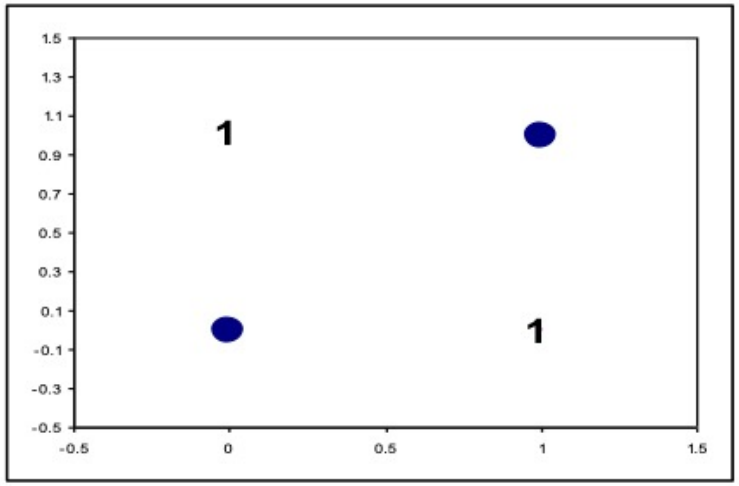


Multilayer perceptron
XOR problem

- 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)

- http://playground.tensorflow.org

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

- We need
  - Loss function → used for leaning weights by gradient descent & backpropagation of correction
  - Metric → in order to evaluate the model

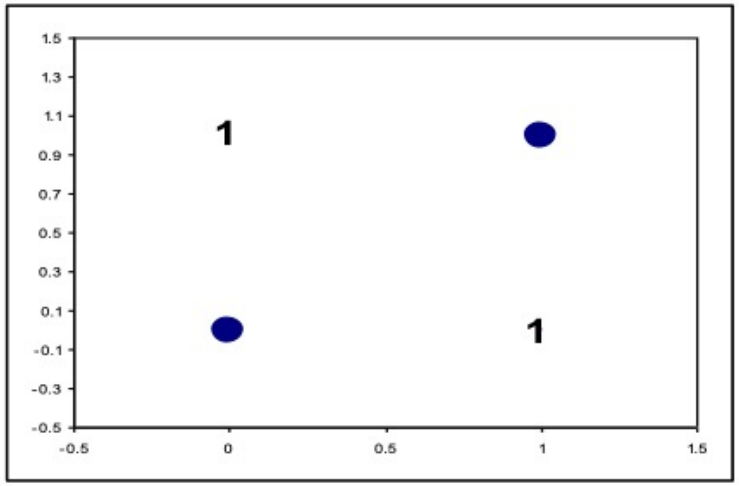
Neural network
Multilayer perceptron

Multilayer 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



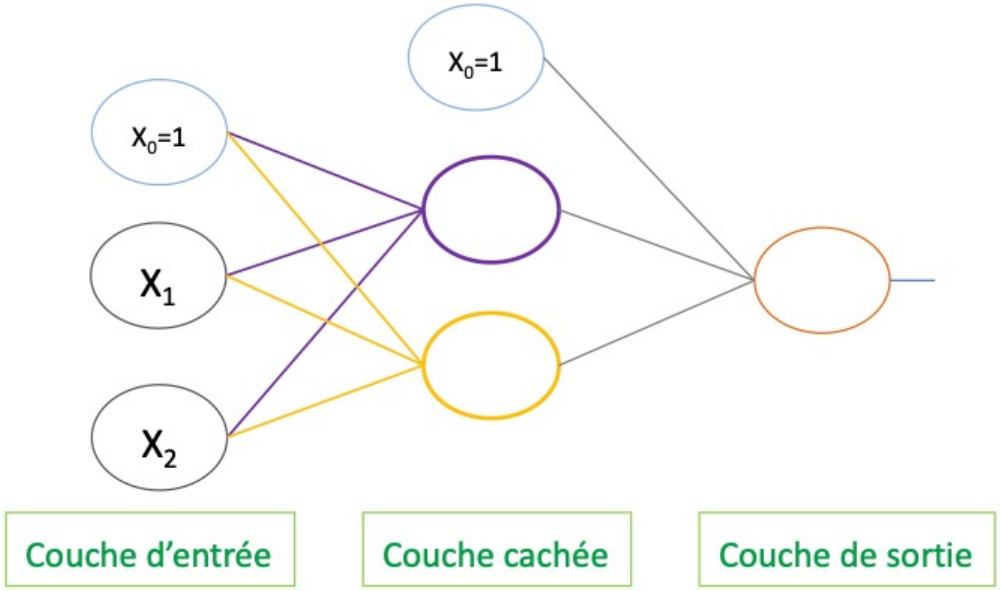
Non séparable linéairement

(Minsky & Papert, 1969)

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

Multi-layer perceptron

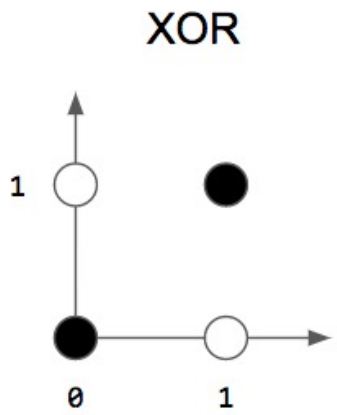
- First layer output
  - $u_1 = sig(v_1)\ with\ v_1 = \sum a_i x_i$
  - $u_2 = sig(v_2)\ with\ v_2 = \sum b_i x_i$
- Network output
  - $\hat{y} = sig(z)\ with\ z = \sum c_i u_i$

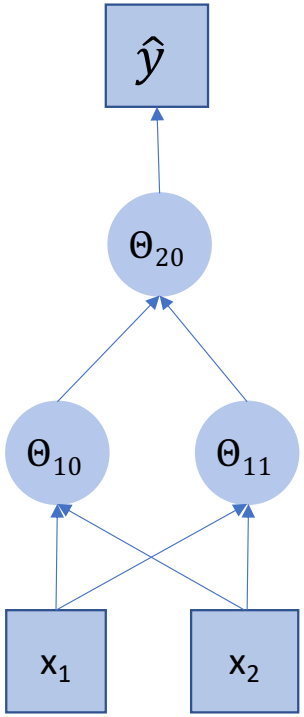


- Fundamental Property: The multilayer perceptron is capable of approximating any continuous function provided that the number of neurons in the hidden layer is appropriately fixed.

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

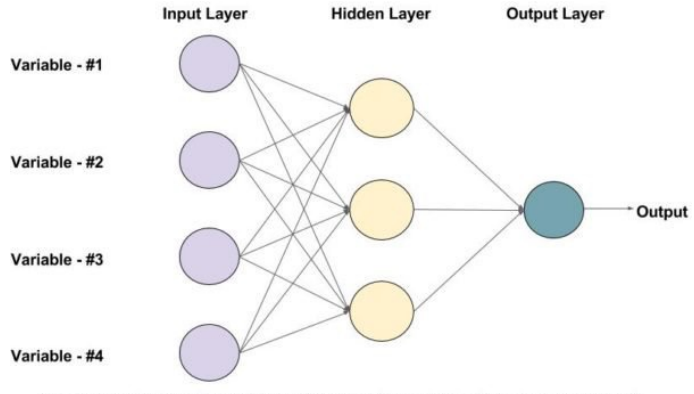




SOLUTION: ß = [[1, 1, -0.5],
[1, 1, -1.5],
[1, -2, -0.5]
i.e. Output is 1 if and only if
(x1+x2-0.5>0) + -2*(x1+x2-1.5>0) - 0.5 > 0

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



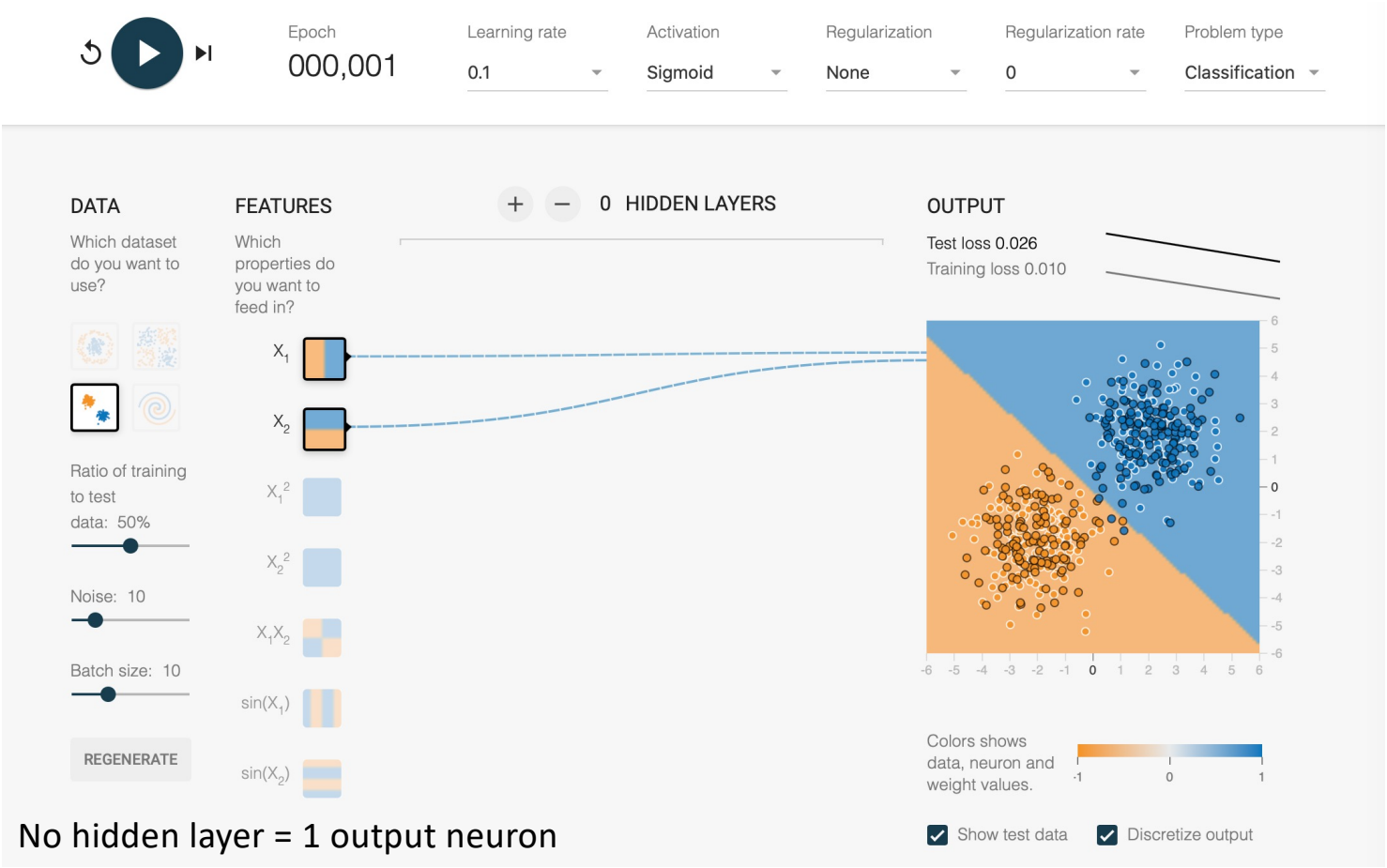
- $output = \sum_j w_j^{(2)} \left(\sum_i w_i^{(1)} x_i \right) = \sum_j \sum_i w_j^{(2)} w_i^{(1)} x_i = \sum_i \sum_j w_j^{(2)} w_i^{(1)} x_i$
- $output = \sum_i (\sum_j w_j^{(2)} w_i^{(1)}) x_i = \sum_i \hat{w}_i x_i = \hat{w}.x$

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)

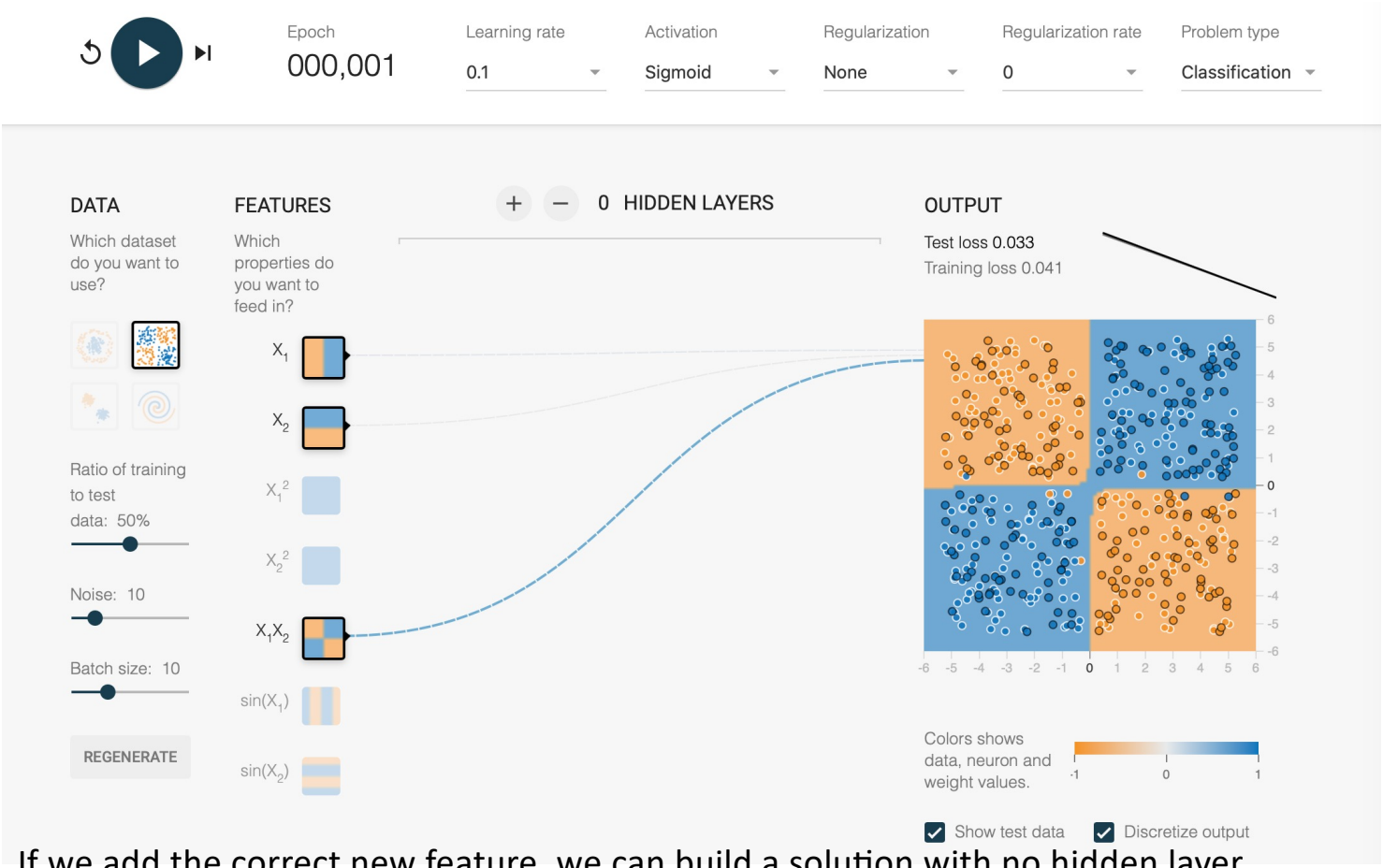
1. Choose a linearly separable dataset
   - Build the smaller network to fit them
2. Choose a non linearly separable dataset
   1. Add some extra feature in order to fit them with only one neuron
      - extra feature=x^2, xy, sin(x)
   2. Remove the extra feature and try to build the smaller network 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



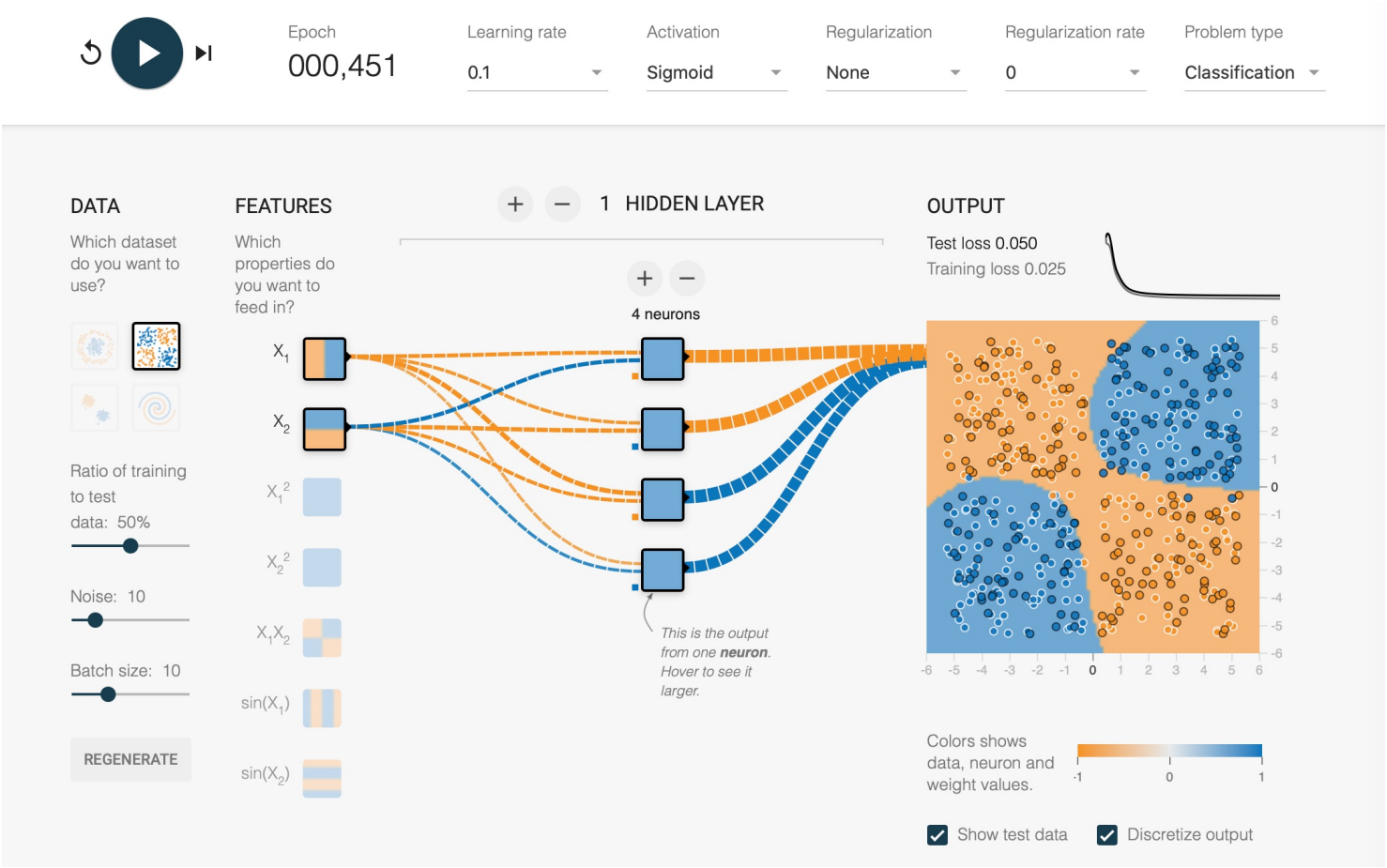
No hidden layer = 1 output neuron

Non linearly separable dataset



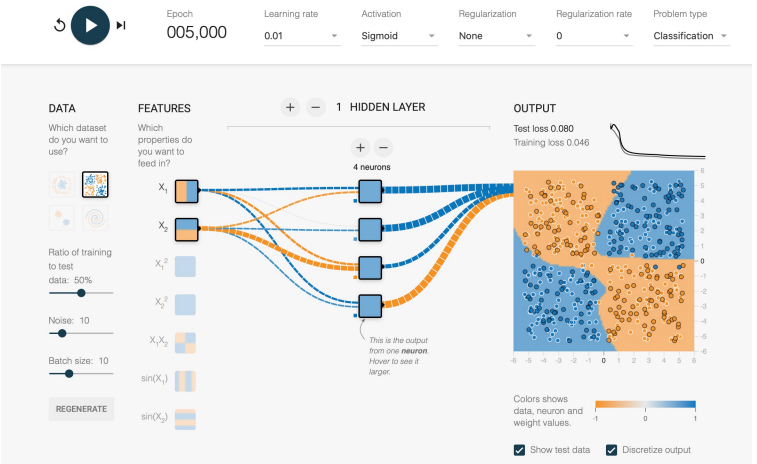
If we add the correct new feature, we can build a solution with no hidden layer i.e. with only 1 output neuron

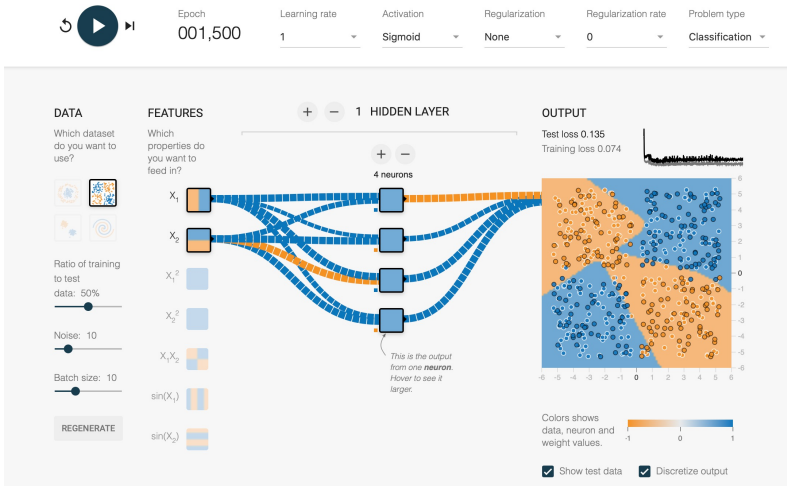
Non linearly separable dataset



Non linearly separable dataset

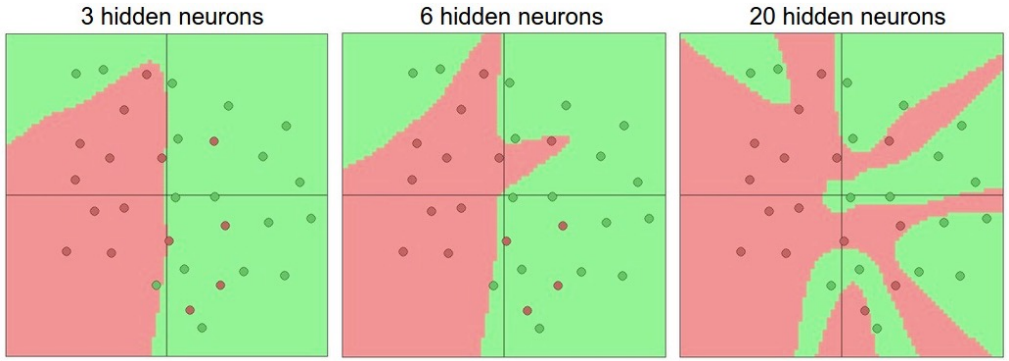
- we look at how the 2 curves evolve:
  - train: does the model learn well
  - test : does the model generalize well
  - Lr = 0,01 → after 5.000 epoch, loss = 0.08
  - Lr = 0,1 → 450 epochs, loss = 0.05
  - Lr = 1 → 1.500 epochs, loss between 0.05-1 unstable for train
  - Lr = 10 → 1.500 epochs loss doesn't converge





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

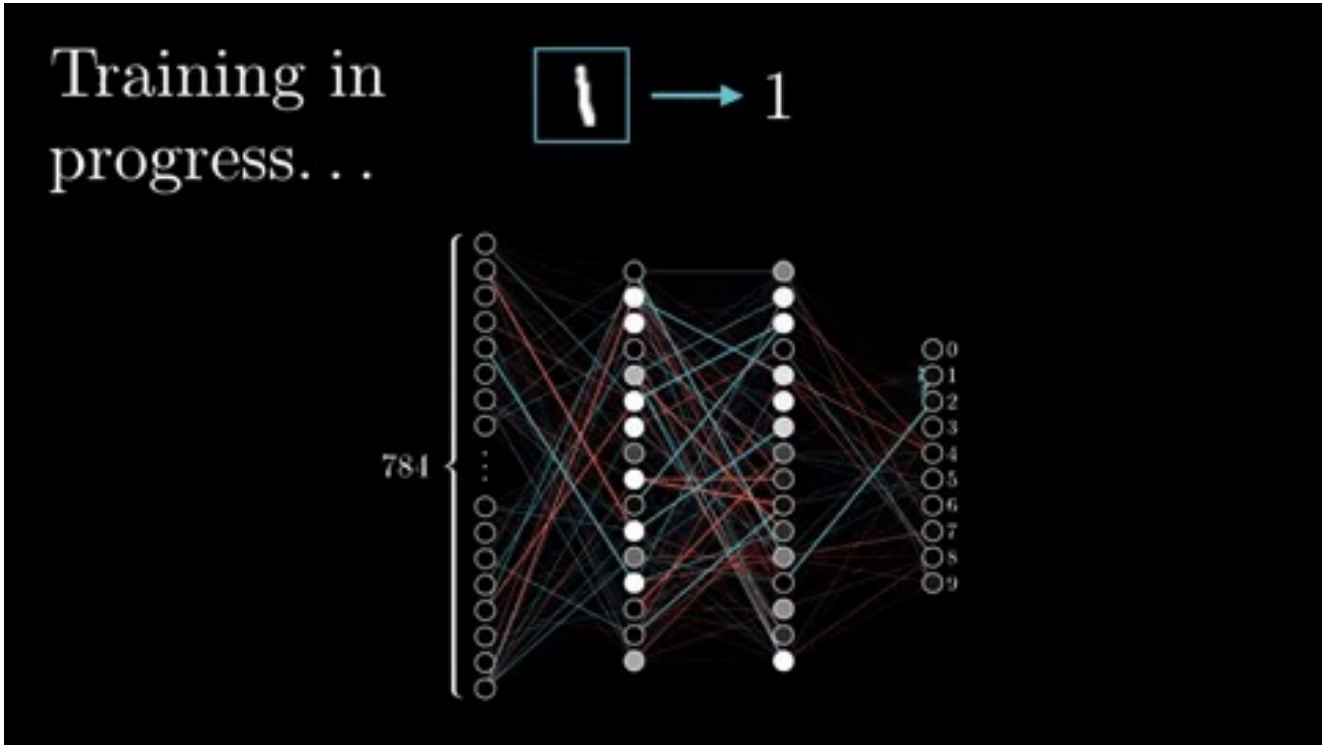


Learning process

Learning through back propagation

- Works in 2 waves :

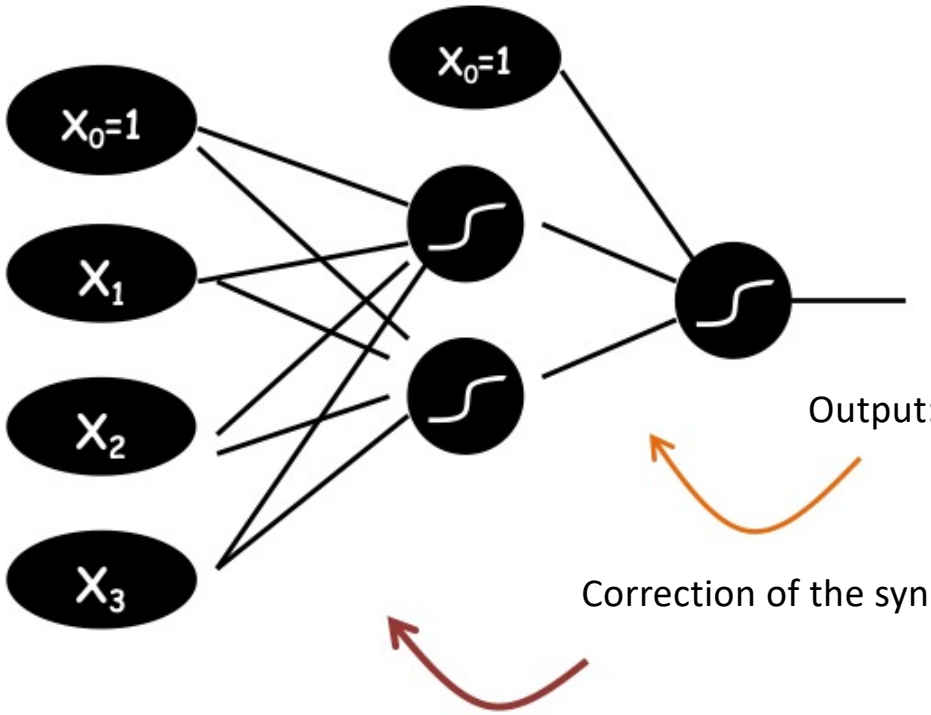
1. propagation, we calculate the output
   → error measurement
2. back-propagation
   → weight correction



Source: https://www.3blue1brown.com/neural-networks

26

Learning = gradient backpropagation



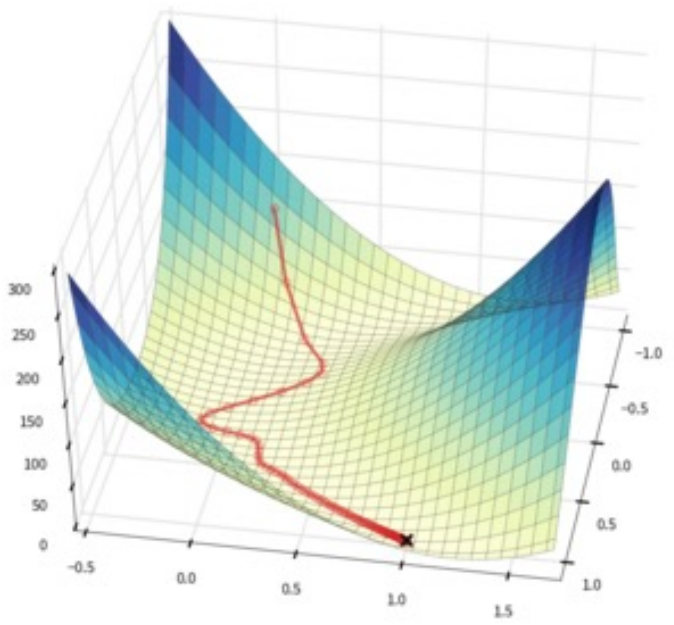
Output: measured error (for an individual)

Correction of the synaptic weights leading to the output layer

Propagation (backwards) of corrections in the intermediate layers

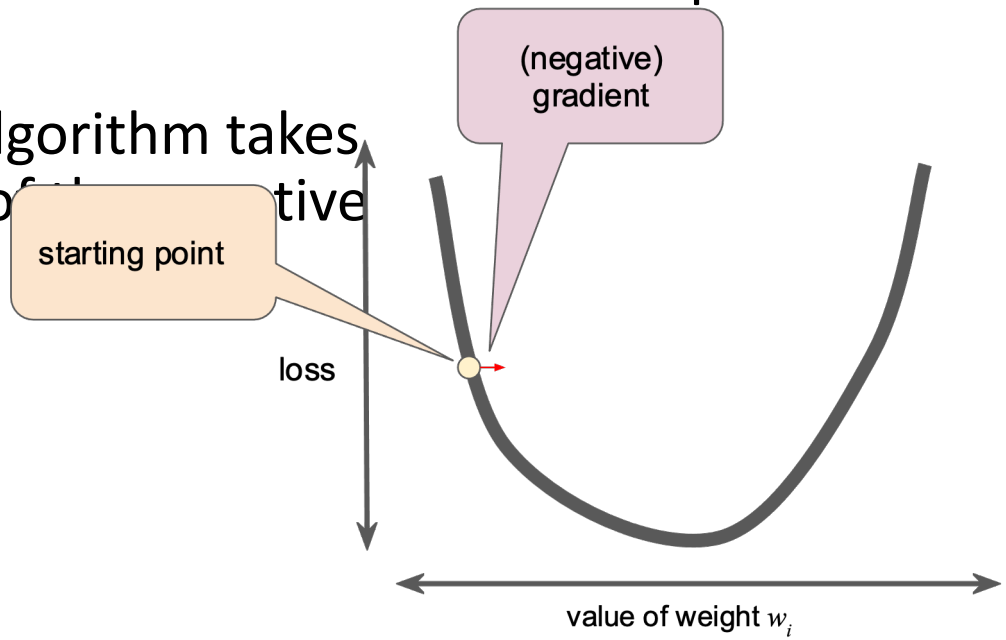
How does the network learn?

- 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
  - Regression : MAE, MSE
  - Classification : CrossEntropy



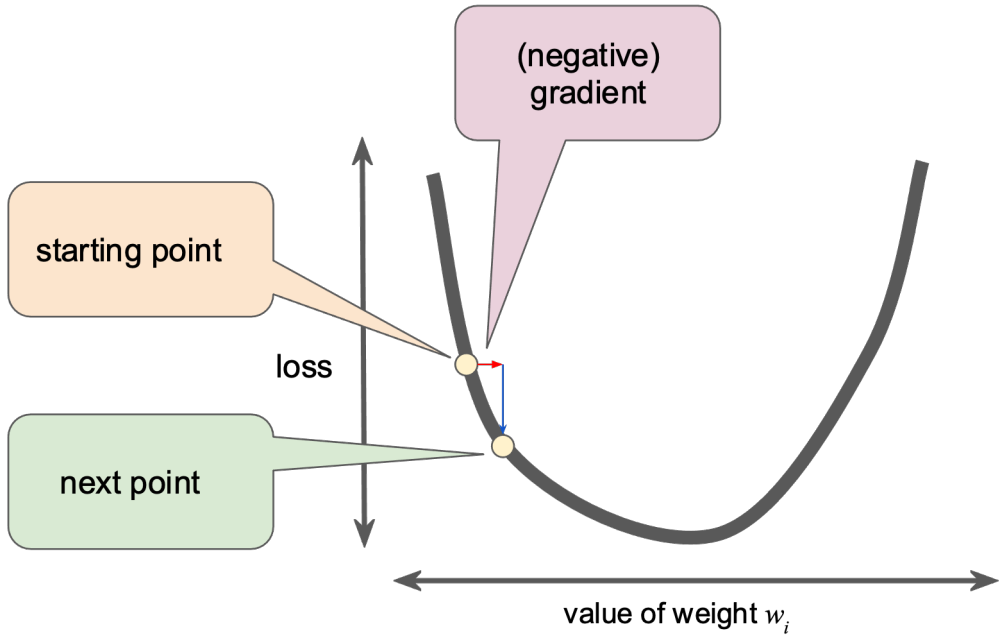
Gradient descent

- 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.



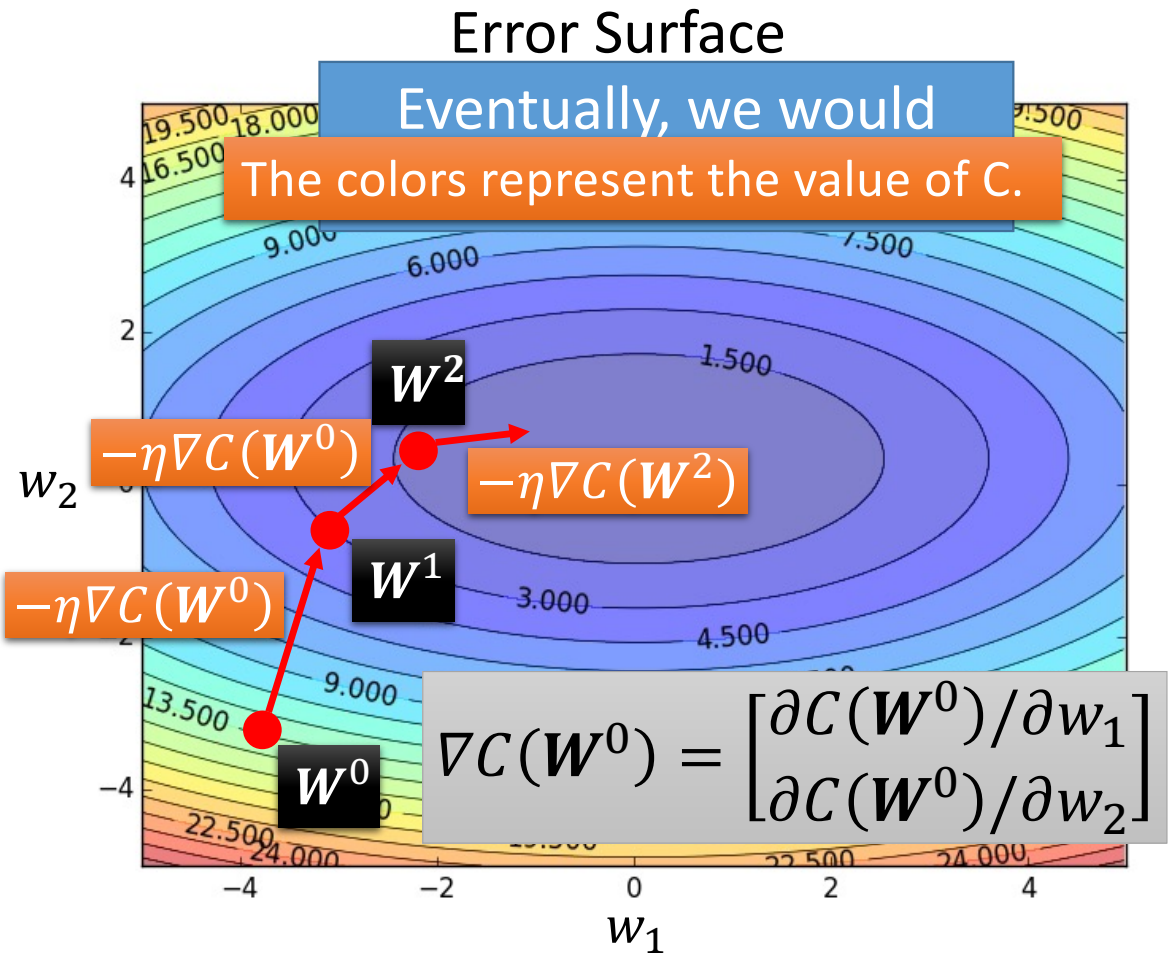
Gradient descent

- 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: $x_{k+1} = xk - \alpha \nabla f(x_k)$
     - α is correct, if $f(x_{k+1}) < f(x_k)$

How to train the network? Gradient descent



$\boldsymbol{W} = \{w_1, w_2\}$

Randomly pick a starting point $\boldsymbol{W}^0$

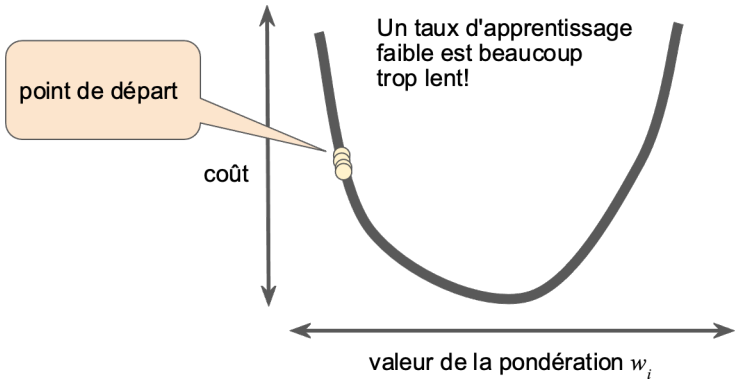
Compute the negative gradient at $\boldsymbol{W}^0$

$\rightarrow -\nabla C(\boldsymbol{W}^0)$

Times the learning rate η

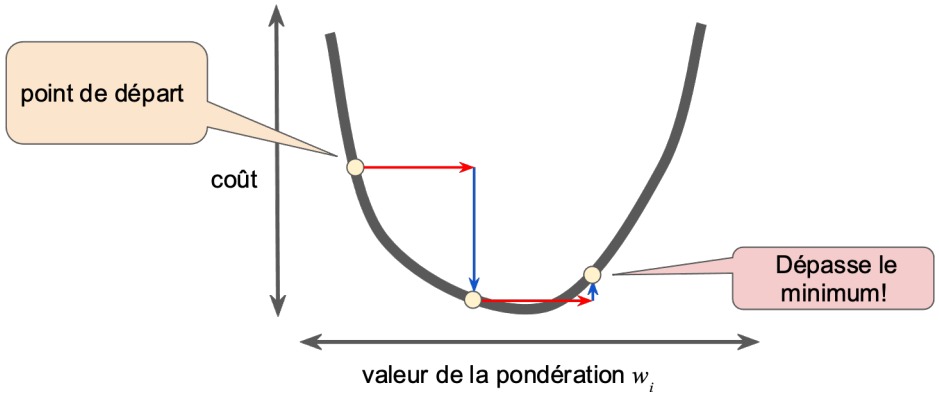
$\rightarrow -\eta\nabla C(\boldsymbol{W}^0)$

Gradient descent

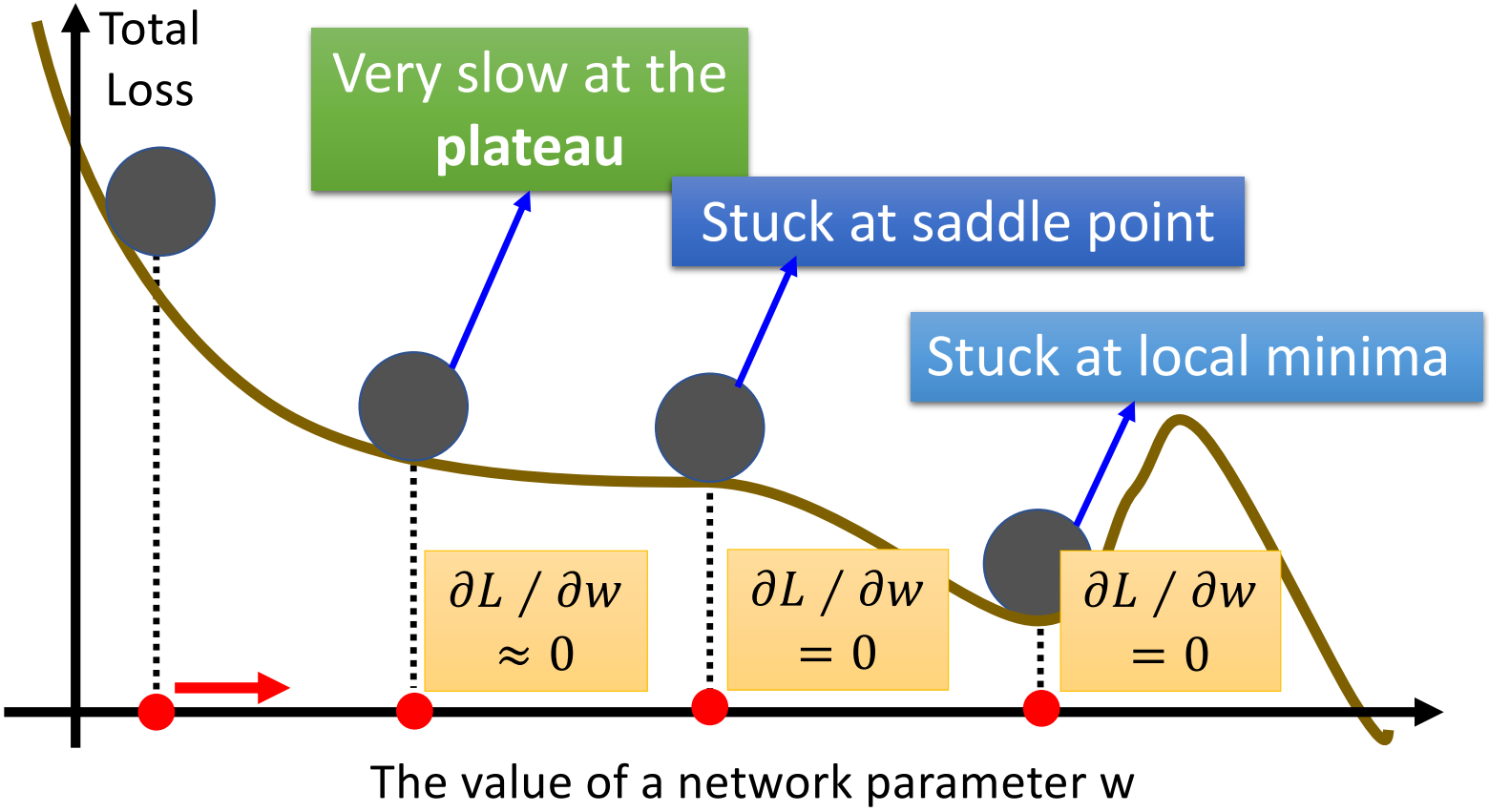


If learning rate is too small
→ the minimum is found after a very long time

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



Hard to find optimal network parameters by gradient descent…



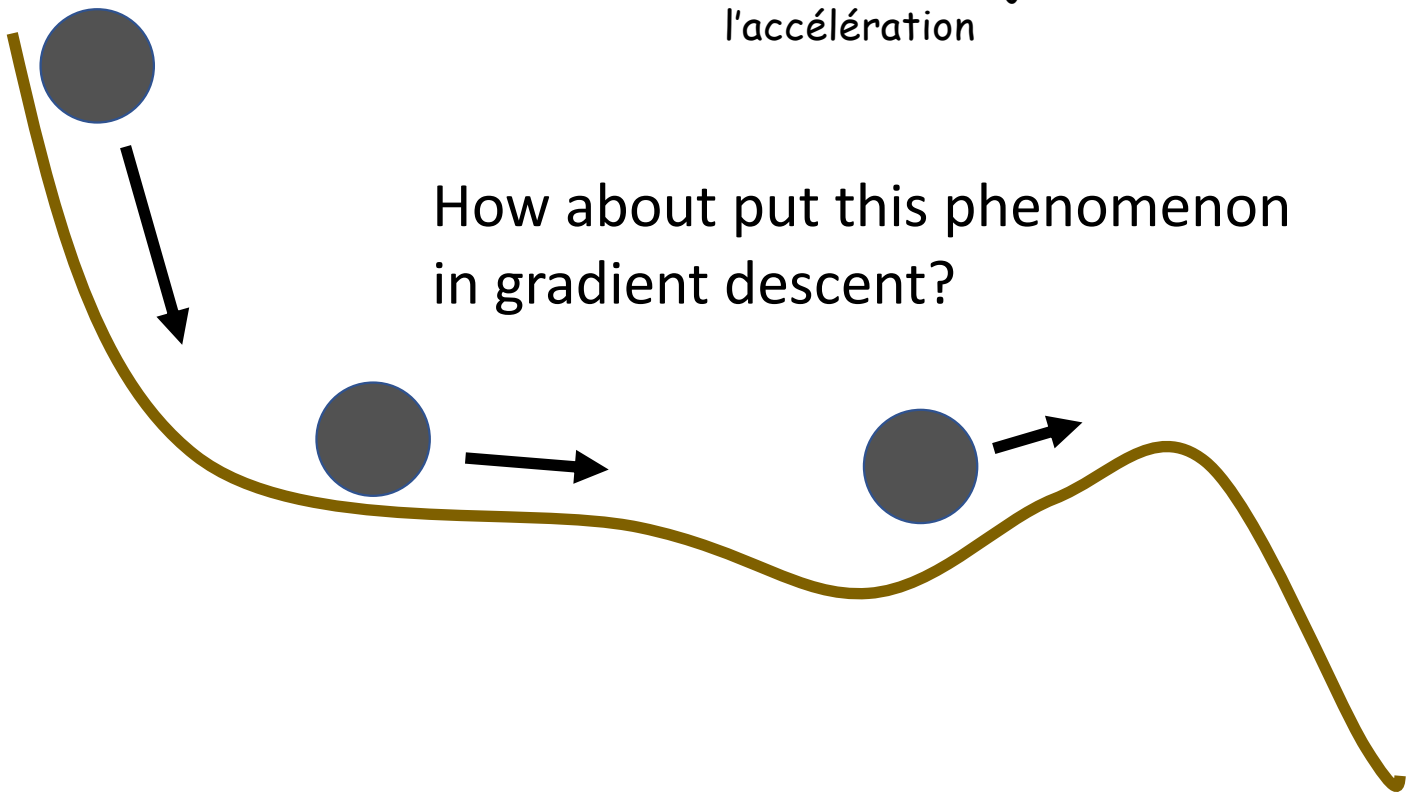
In physical world ……

- **Momentum = “a little flick”**

Gradient
= dérivée de la fonction à la position X
= vitesse

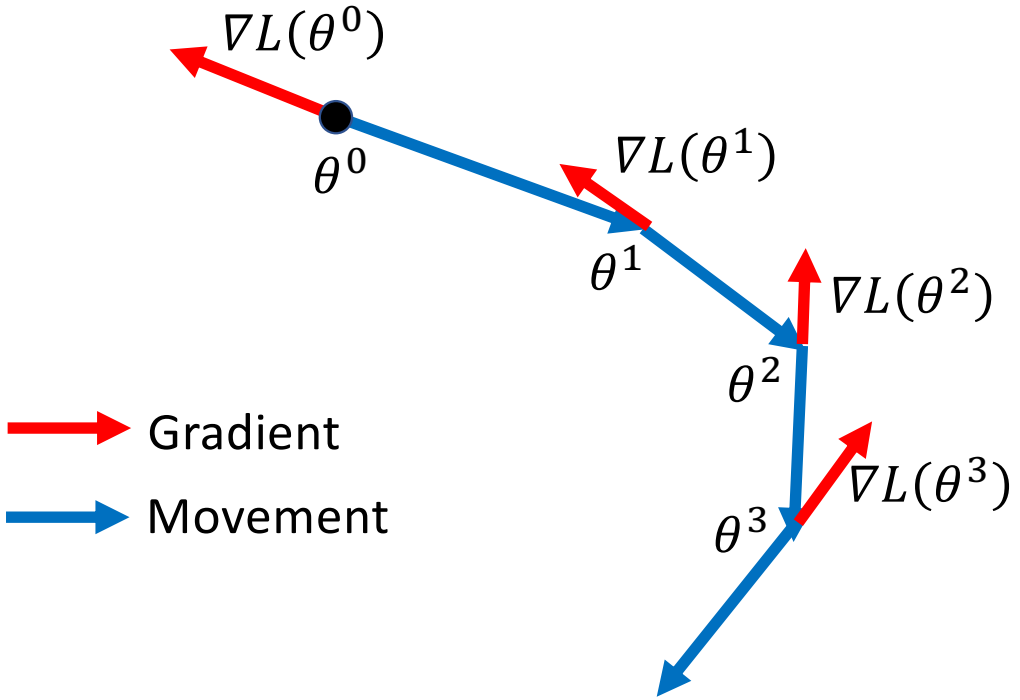
Si on dérive une seconde fois
= accélération

Momentum ≈ ajout d’une information sur l’accélération



How about put this phenomenon in gradient descent?

Vanilla Gradient Descent



Start at position θ^0

Compute gradient at θ^0

Move to $\theta^1 = \theta^0 - \eta\nabla L(\theta^0)$

Compute gradient at θ^1

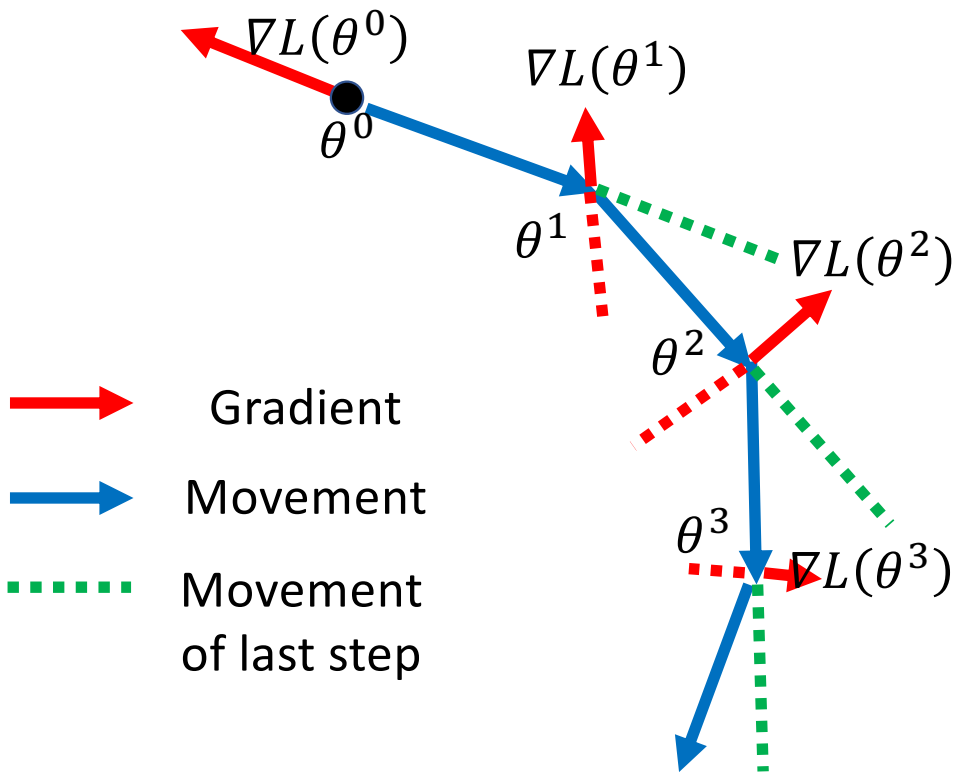
Move to $\theta^2 = \theta^1 - \eta\nabla L(\theta^1)$

⋮

Stop until $\nabla L(\theta^t) \approx 0$

Momentum = “a little flick”

Movement: movement of last step minus gradient at present



Start at point θ^0

Movement $v^0=0$

Compute gradient at θ^0

Movement $v^1 = \lambda v^0 - \eta \nabla L(\theta^0)$

Move to $\theta^1 = \theta^0 + v^1$

Compute gradient at θ^1

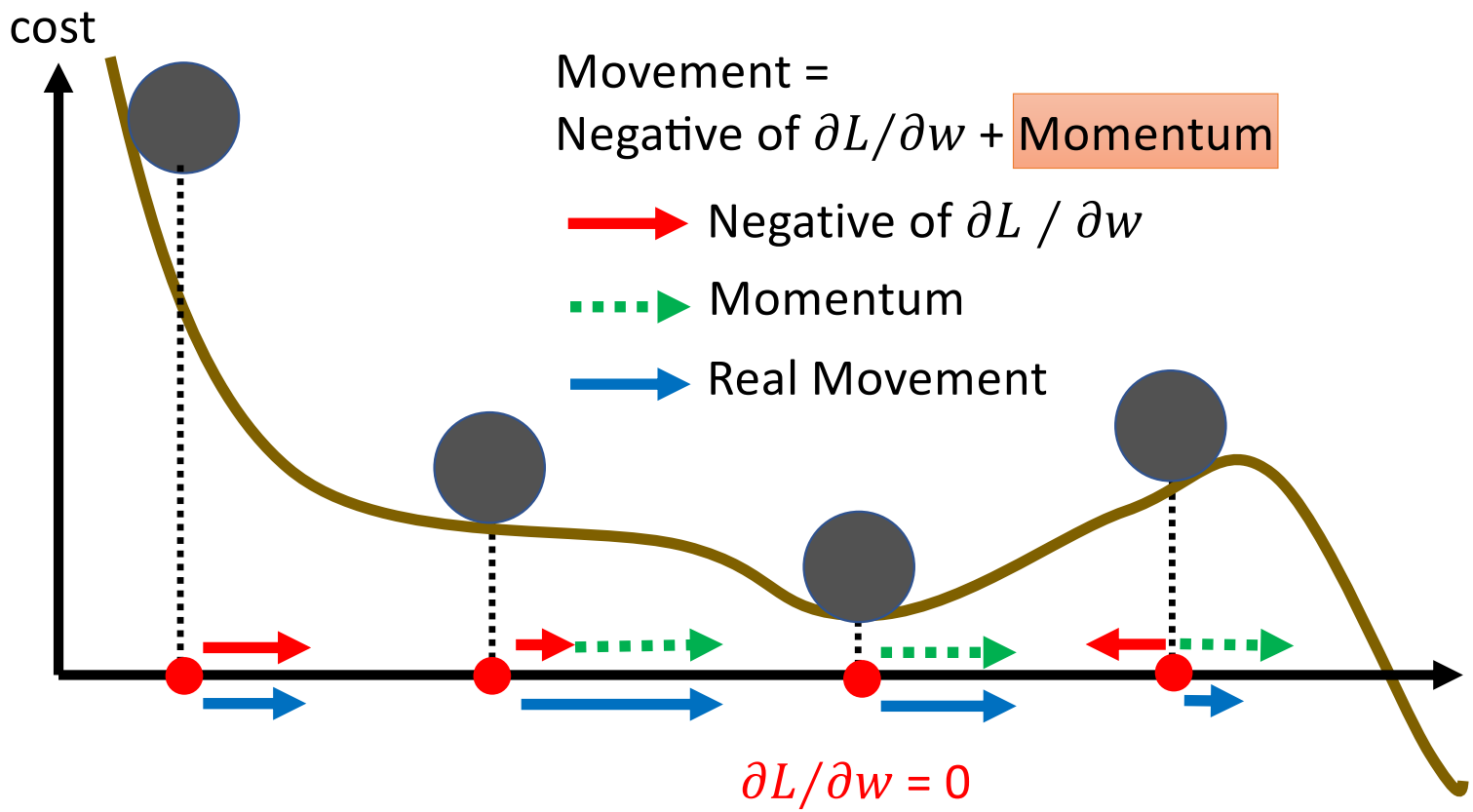
Movement $v^2 = \lambda v^1 - \eta \nabla L(\theta^1)$

Move to $\theta^2 = \theta^1 + v^2$

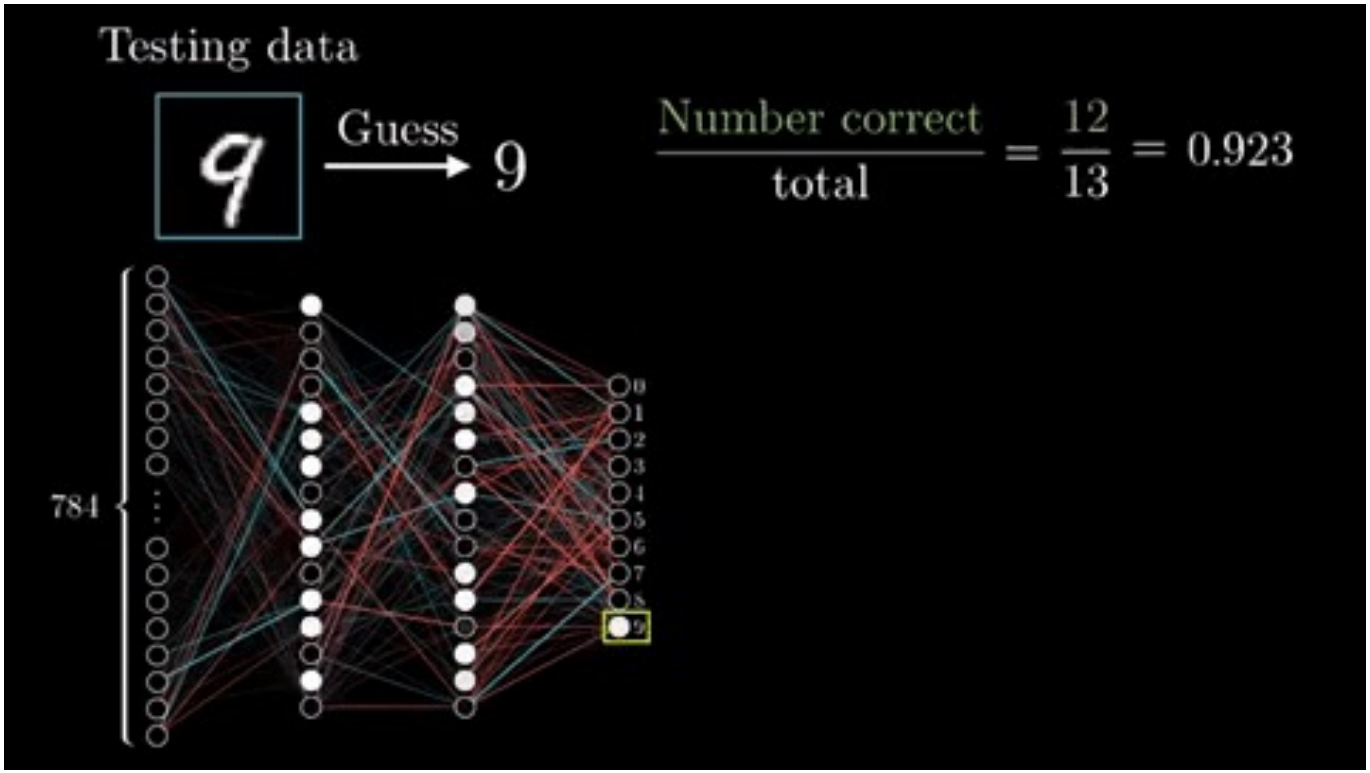
Movement not just based on gradient, but previous movement.

Momentum =

Still not guarantee reaching global minima, but give some hope ……



Inference/Prediction



Source: https://www.3blue1brown.com/neural-networks

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)


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 the 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
  - Run the same code with... – TensorFlow, CNTK, Theano, …
- 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
  - Subclassing

Functional API
(our favorite API)

```
1. import keras
2. from layers import Input, Dense

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

7. model = keras.Model(inputs, outputs)
8. model.compile(optimizer='sgd', loss='mean_squared_error')

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

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

Practical issues

Choose the network architecture

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

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

Choose the network architecture

- **Output layer**

→Regression

→ One neuron by function to fit

→ activation funtion: Linear

- outputs = Dense(**1,** activation='**linear**')(x)

→Classification

→2 classes = 1 neuron

→→ activation funtion: sigmoid

- outputs = Dense(**1**, activation='**sigmoid**')(x)
  - y = [0, 1]

→ N classes = N neurons

→ activation funtion: sofmax

- outputs = Dense(**n_classes**, activation='**softmax**')(x)
  - Y = one hot encoding
  - Y = [0, 1, 0, 0] for example for 4 classes

Number classes = 5

Size one hot encoding = 5

Classes = [0, 4]

Encode class 3

→ [0 0, 1, 0 0]

→ Sklearn →OneHotEncoding

→ Pandas.to_categorical or dummies gives methodological error

Summary: https://www.youtube.com/watch?v=IHZwWFHWa-w

Lab

Lab work

- Part 1
  - Build a 1 hidden neuron network
    - Just to learn the main function
    - Compare the result if you change the activation function ‘sigmoid’, ‘tanh’ or ‘relu’
    - Plot learning curve
  - Build a MLP in a various configurtion
    - For exemple 1, 10, 100 hidden layer and 32, 256, 512 neurons by layer

    – What did you observe ? Wht is the best ?

- Part 2
  - Build an MLP for classifying mnist

  -- I give you some of the functions take the time to understand the code, next time you will have to build the whole network.