PMR5251 - Assessment of Mechanical Behavior of Materials using Machine Learning Approach

ARTIFICIAL NEURAL NETWORKS (ANNS)

Izabel F. Machado
Larissa Driemeier

SCHEDULE

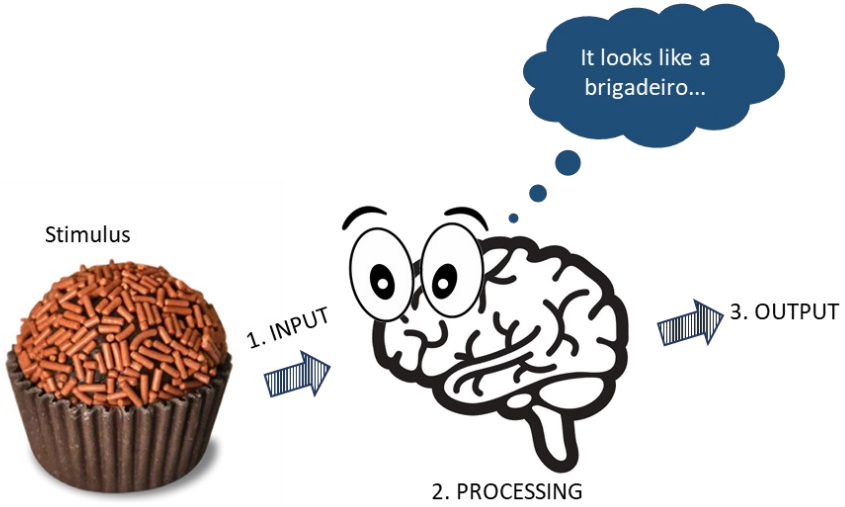
Date	Contents	Activities	Prof.
23/06	Introduction to Machine Learning main concepts	Notebook and Abaqus	Larissa
30/06	Review mechanical behavior of materials and microstructure characterization	List of exercises	Izabel
07/07	Discussion about list of exercises	List of Exercises MS evolution	Izabel
14/07	Neural Networks – theory	List of exercises	Larissa
21/07	Neural Networks – application in structural analyses	Notebook/article	Larissa
28/07	Convolutional Neural Network – application	Notebook MS evolution	Larissa
04/08	Damage	List of Exercises	Izabel
11/08	Discussion about list of exercises	List of Exercises MS evolution	Izabel
18/08	Discussion about manuscripts	Manuscript	Izabel/Larissa
25/08	Presentations	10min presentation	Izabel/Larissa

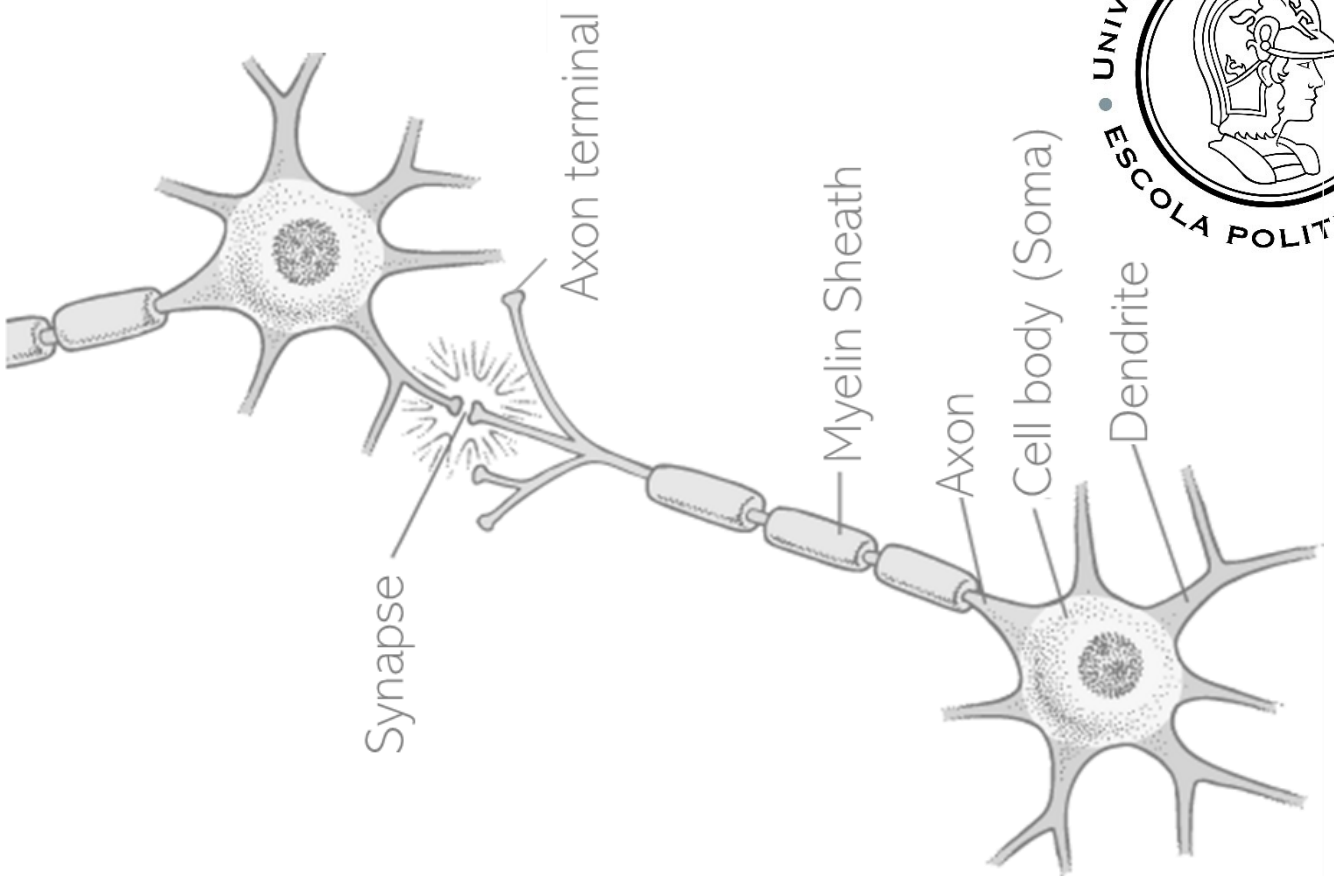
WHAT IS IT?

Definition

Structure

NEURAL NETWORK



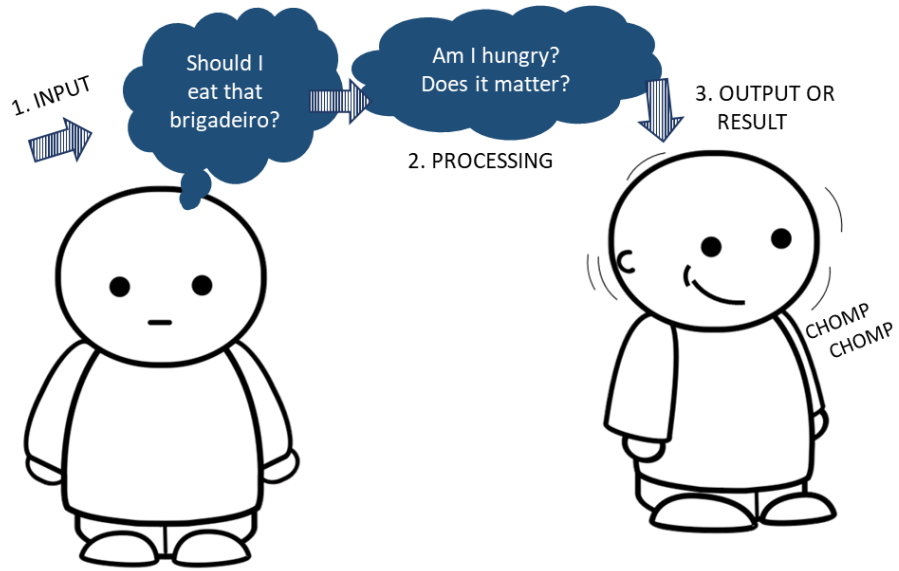


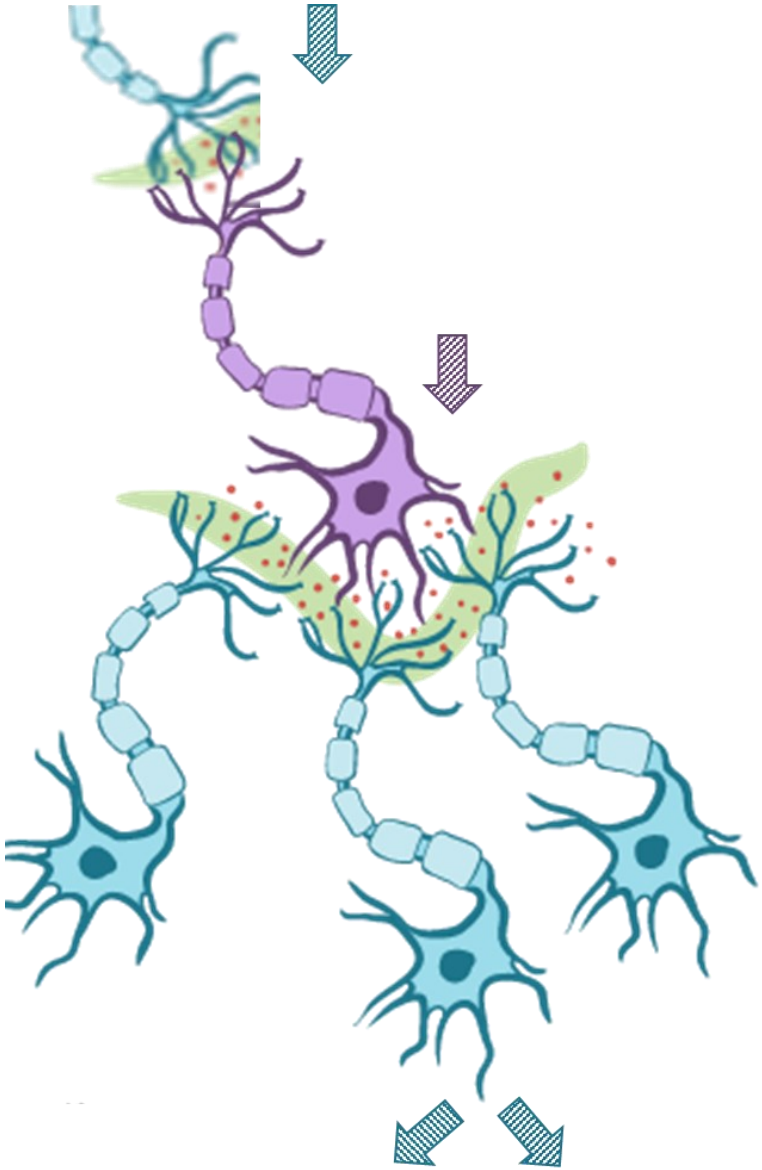
The neuron is the basic computing unit in the brain.

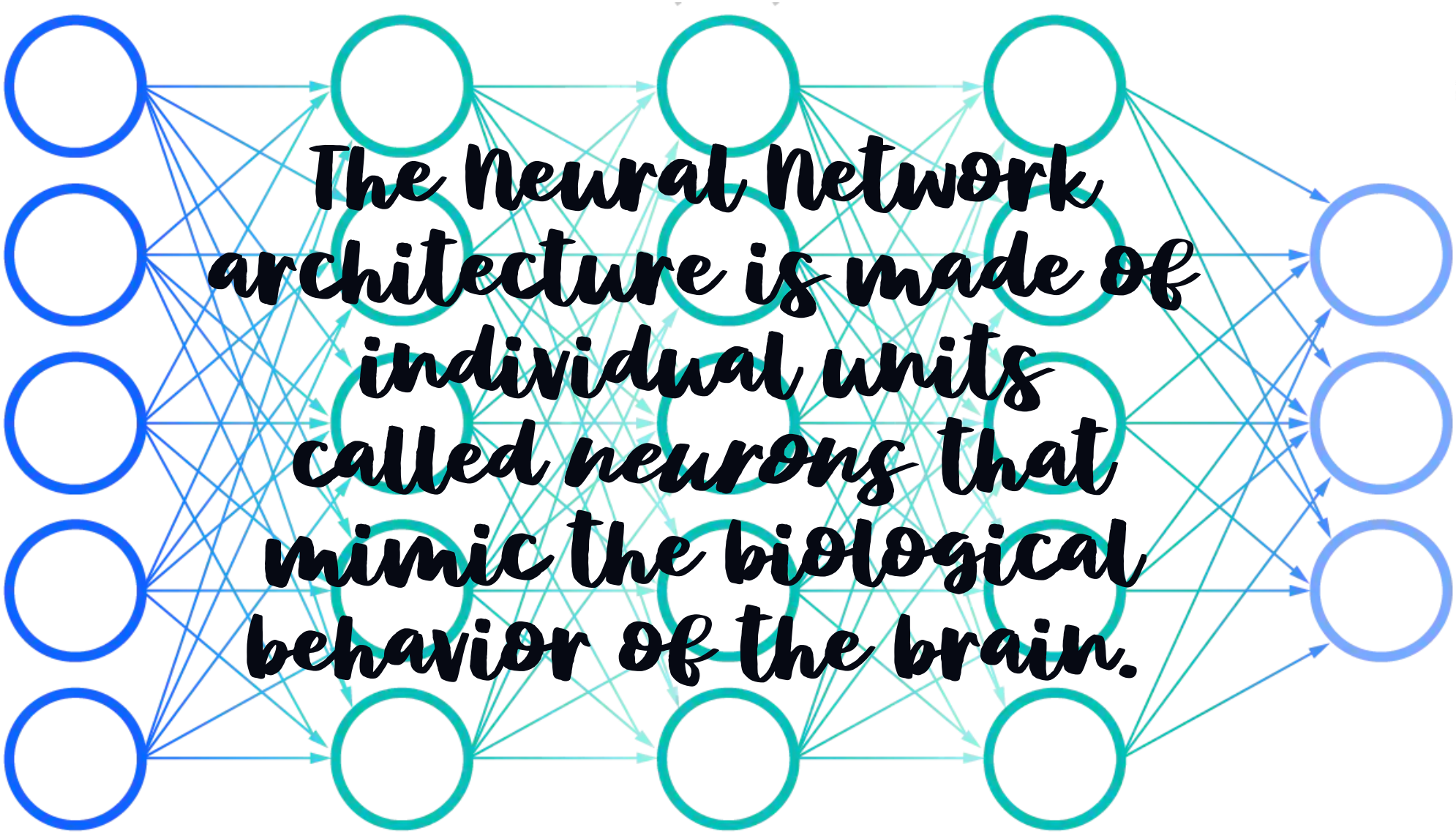
Our brain is made up of billions of neurons with hundreds of billions of connections between them, forming a huge communication network, the neural network.

- Dendrite: Receives signals from other neurons
- Soma: Processes the information
- Axon: Transmits the output of a neuron
- Synapse: Point of connection to other neurons

NEURAL NETWORK



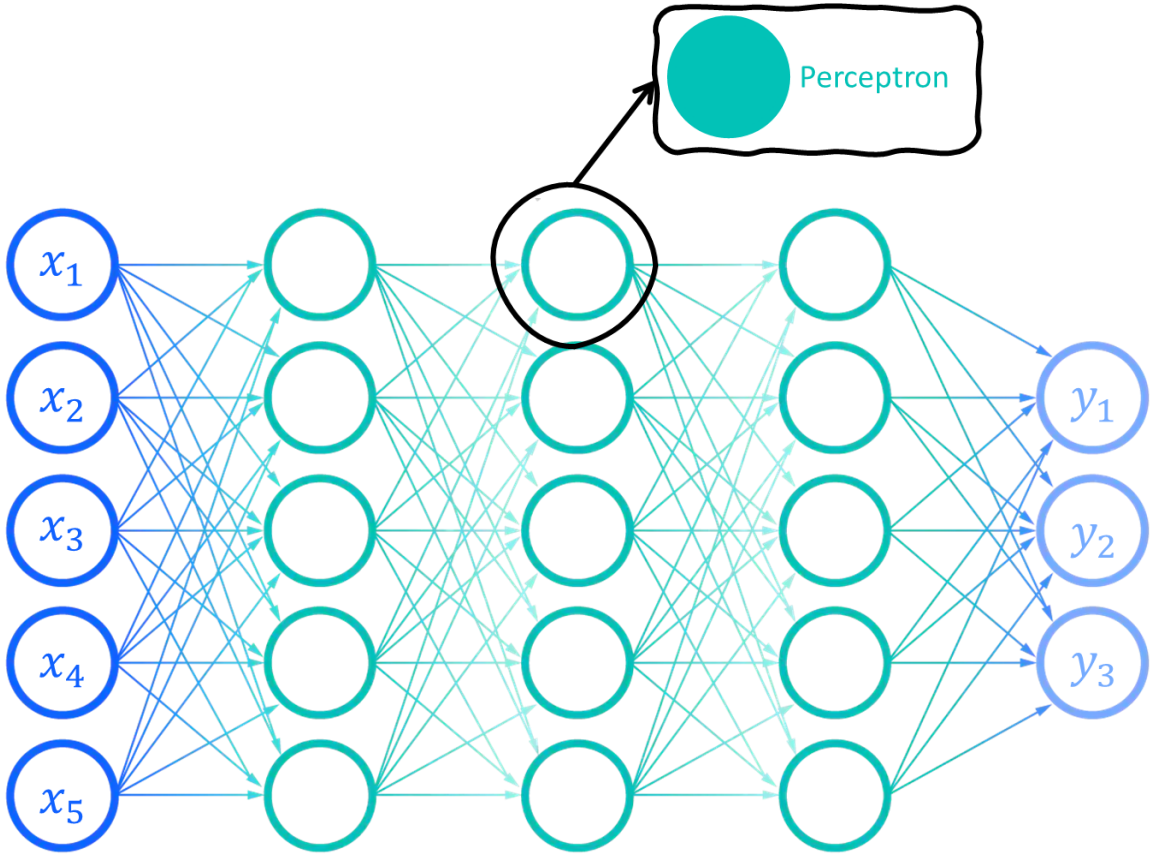




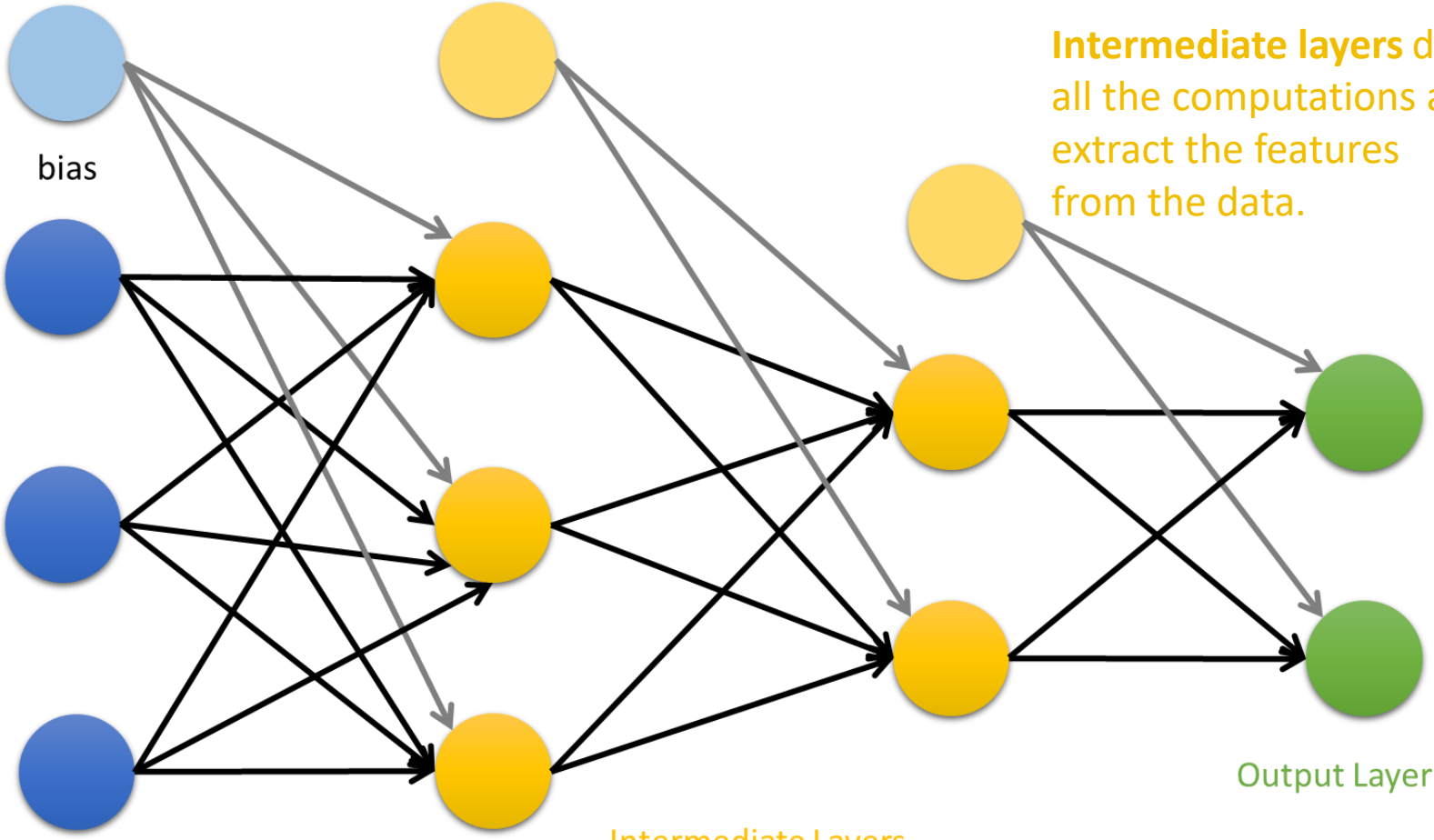
The Neural Network architecture is made of individual units called neurons that mimic the biological behavior of the brain.

MAIN CHARACTERISTICS OF BIOLOGICAL NEURAL NETWORKS PRESENT IN ANNS

- An ANN consists of many simple processing units (neurons) interconnected;
- Each artificial neuron, called perceptron, receives many signals;
- These are modified by a weight in the receptor synapses;
- Neurons sum up the weighted inputs;
- Neurons define the importance of information and transmit a single output;
- The output of one neuron is transmitted to many other neurons;
- A network can have several layers of neurons.



NEURAL NETWORK: ARCHITECTURE

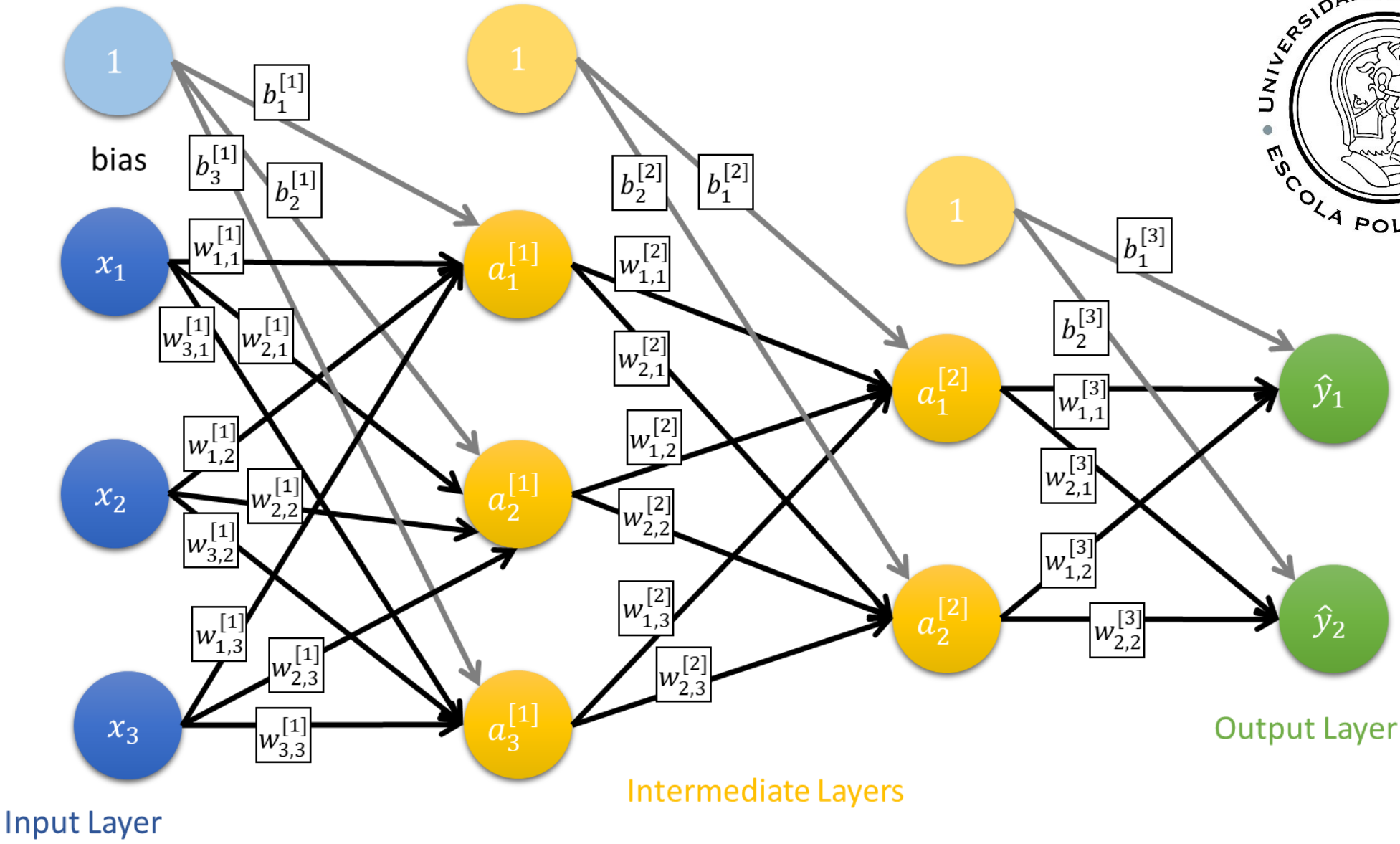


The data that we feed to the model is loaded into the **input layer** from external sources like a CSV file or a web service.

Intermediate layers do all the computations and extract the features from the data.

The **output layer** takes input from preceding hidden layers and comes to a final prediction based on the model's learnings. It is where we get the result.

The ANN has $L = 3$ hidden layers (see that the last, output layer is not a hidden layer).



The neural network maps an input dataset $\boldsymbol{x_i} \in \boldsymbol{X}$ to the desired target value $\boldsymbol{y_i} \in \boldsymbol{Y}$

ANN IS A SUPERVISED LEARNING ALGORITHM

Neural Network (or Artificial Neural Network) can learn by examples.

For an ANN to be able to map the problem, being it classification or regression, it needs to be trained. The supervised learning of an ANN initially requires a set of labeled and classified data.

During learning, the outputs generated by the ANN are compared with the desired outputs and the differences between them are used for training.

DATASET

x_1

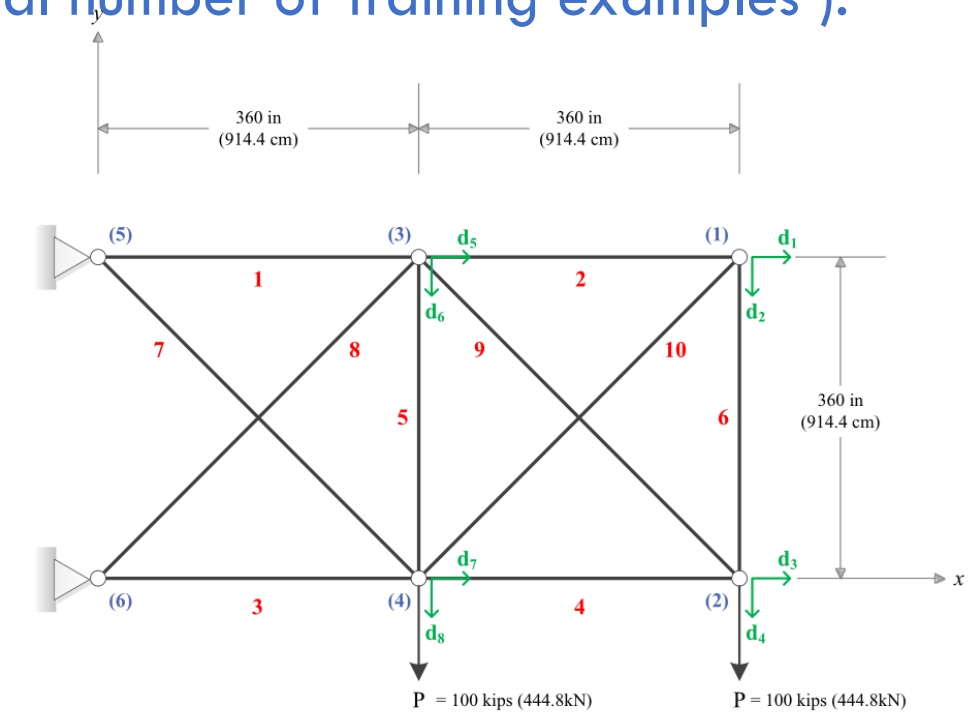
x_2

x_3

$\vdots$

x_{10}

$\boldsymbol{X} \in \mathbb{R}^{n_x,m}$, where n_x is the number of features in the problem (number of entries for each example) and m is the number of available data (total number of training examples).



The output data is defined by $\boldsymbol{Y} \in \mathbb{R}^{n_y,m}$, where n_y is the number of ANN outputs.

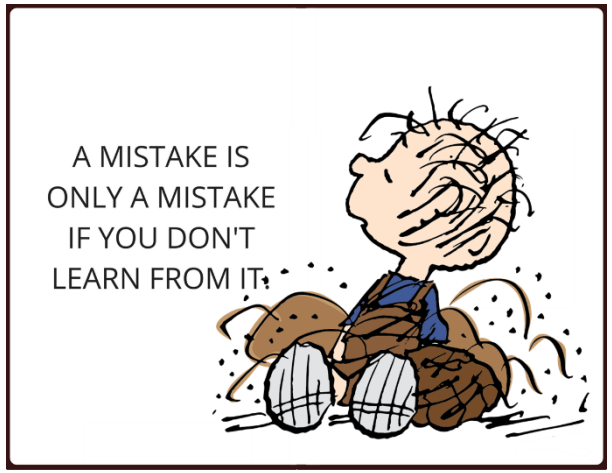
$\hat{y}_1$

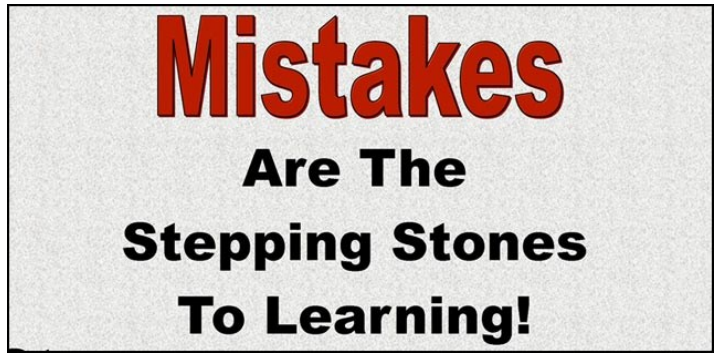
$\hat{y}_2$

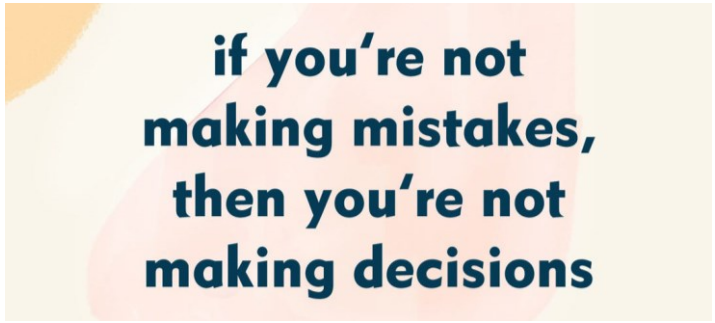
$\vdots$

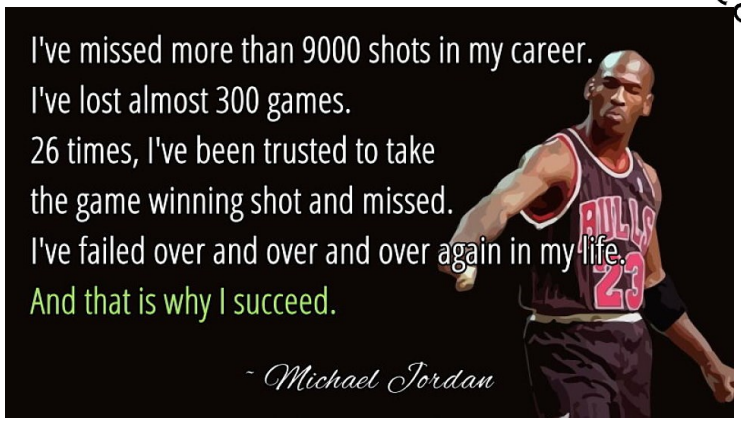
$\hat{y}_8$

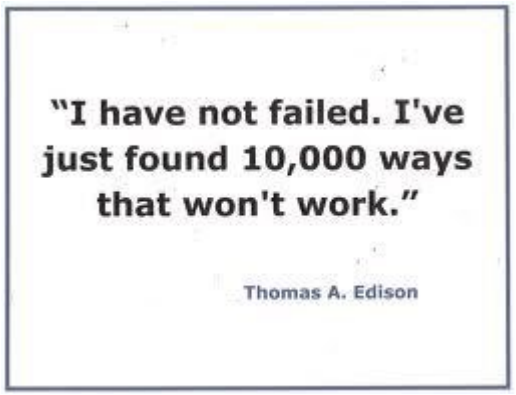
ANNs KNOW THINGS.....

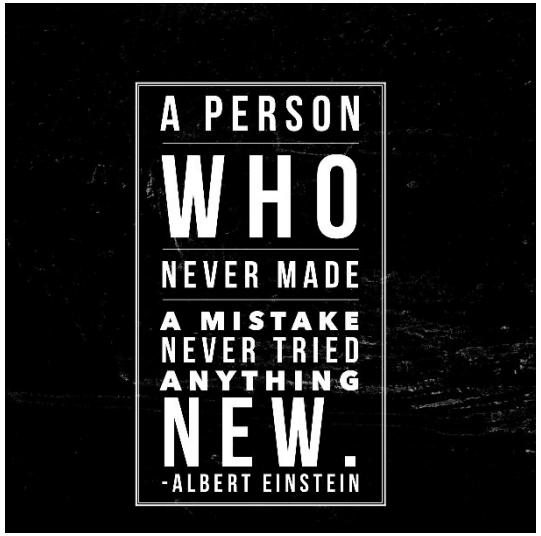








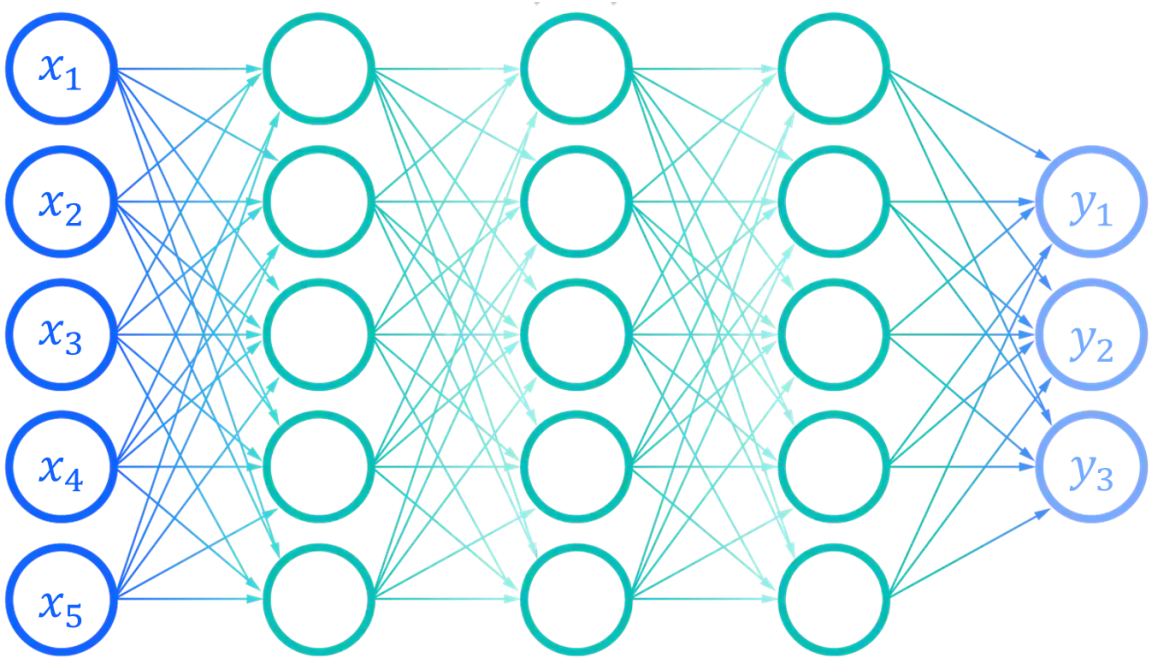




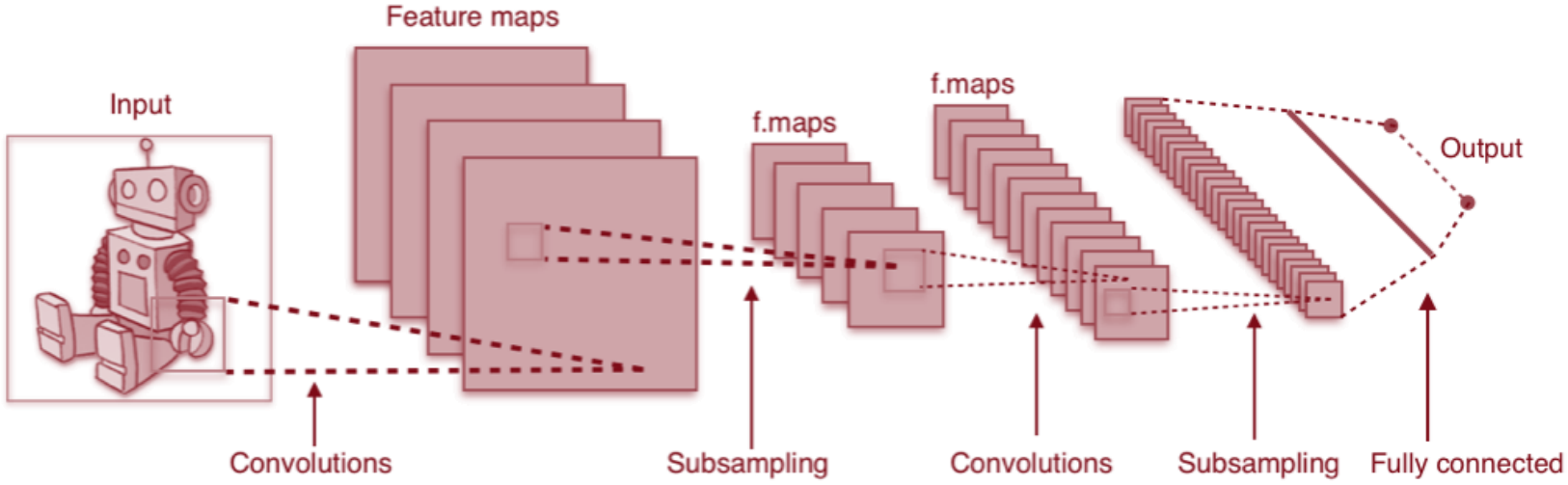
TYPES OF ANNS...

The fundamental structure of ANNs is the layer! There are several types of layers, each type being specific for a given tensor shape and for a given type of processing.

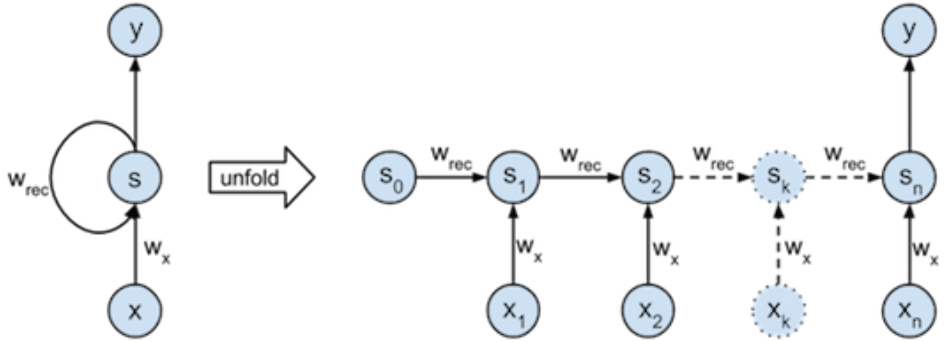
Data in the form of vectors is stored in 2D tensors (1st axis: examples; 2nd axis: features) and normally processed in densely connected layers, called dense layers;



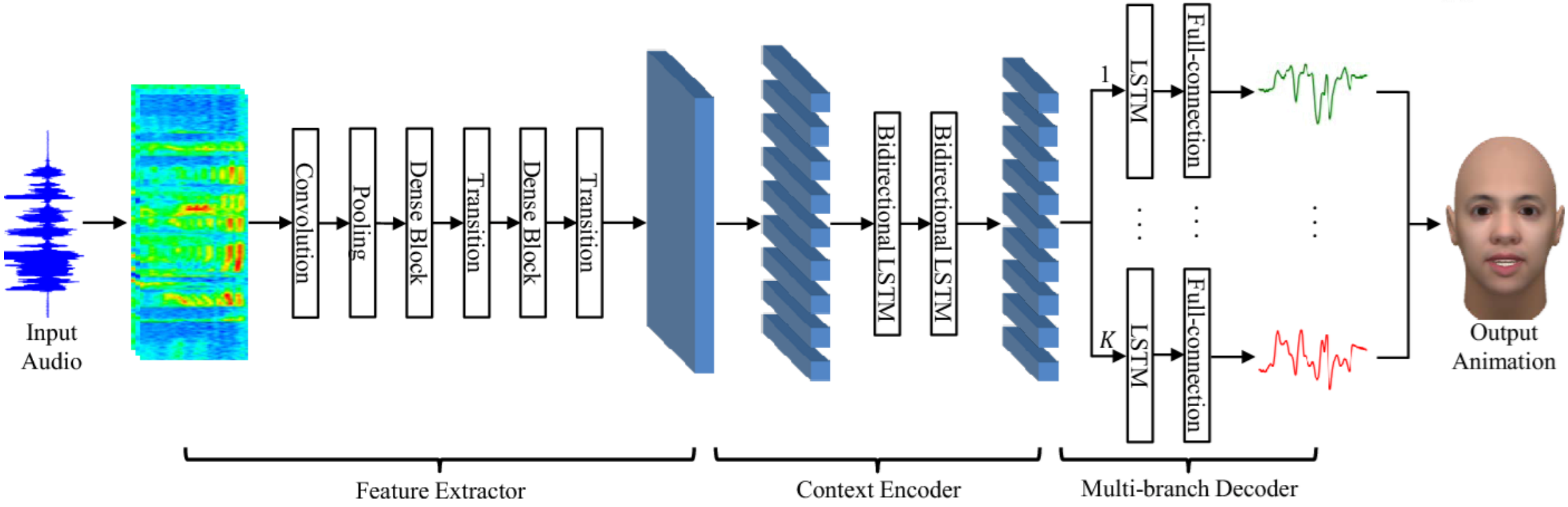
Grayscale image data is stored in 3D tensors (1st axis: examples; 2nd axis: height; 3rd axis: width) and normally processed in convolutional layers



Temporal data sequences are stored in 3D tensors (1st axis: examples; 2nd axis: time; 3rd axis: characteristics) and normally processed in recurring layers, for example LSTM (Long Short-Term Memory) or GRU (Gated Recurrent Unit) layers.



DENSE CONVOLUTIONAL RECURRENT NEURAL NETWORK (DENSECRNN)



L. Xiao and Z. Wang, "Dense Convolutional Recurrent Neural Network for Generalized Speech Animation," *2018 24th International Conference on Pattern Recognition (ICPR)*, 2018, pp. 633-638, doi: 10.1109/ICPR.2018.8545744.

PERCEPTRON

Frank Rosenblatt
Cornell Aeronautical Laboratory

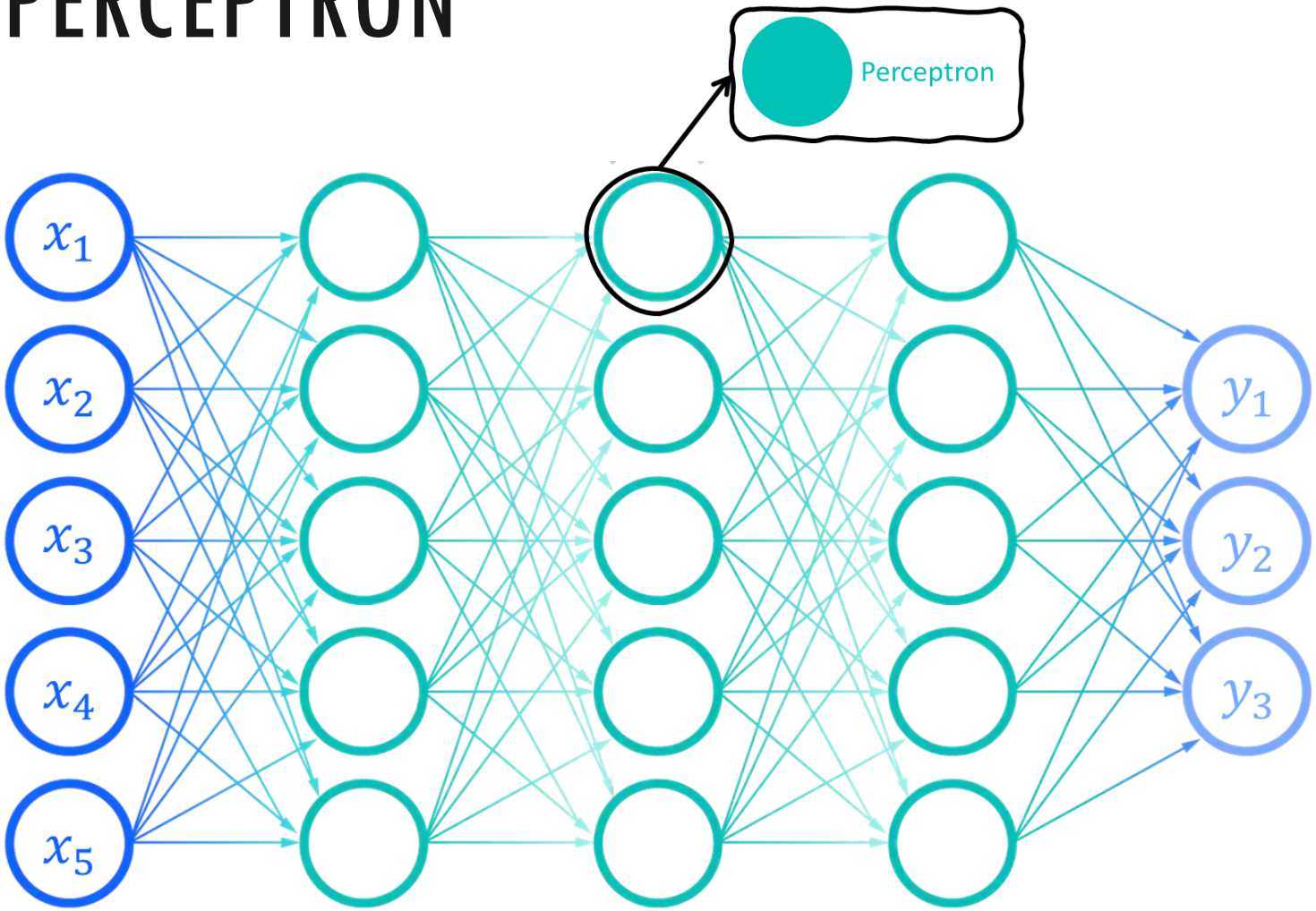
In 1958, Rosenblatt created the perceptron, an algorithm for pattern recognition. The simplest **neural network**, well known as the Rosenblatt Perceptron, is a neural network compound of a single artificial neuron.

THE PERCEPTRON

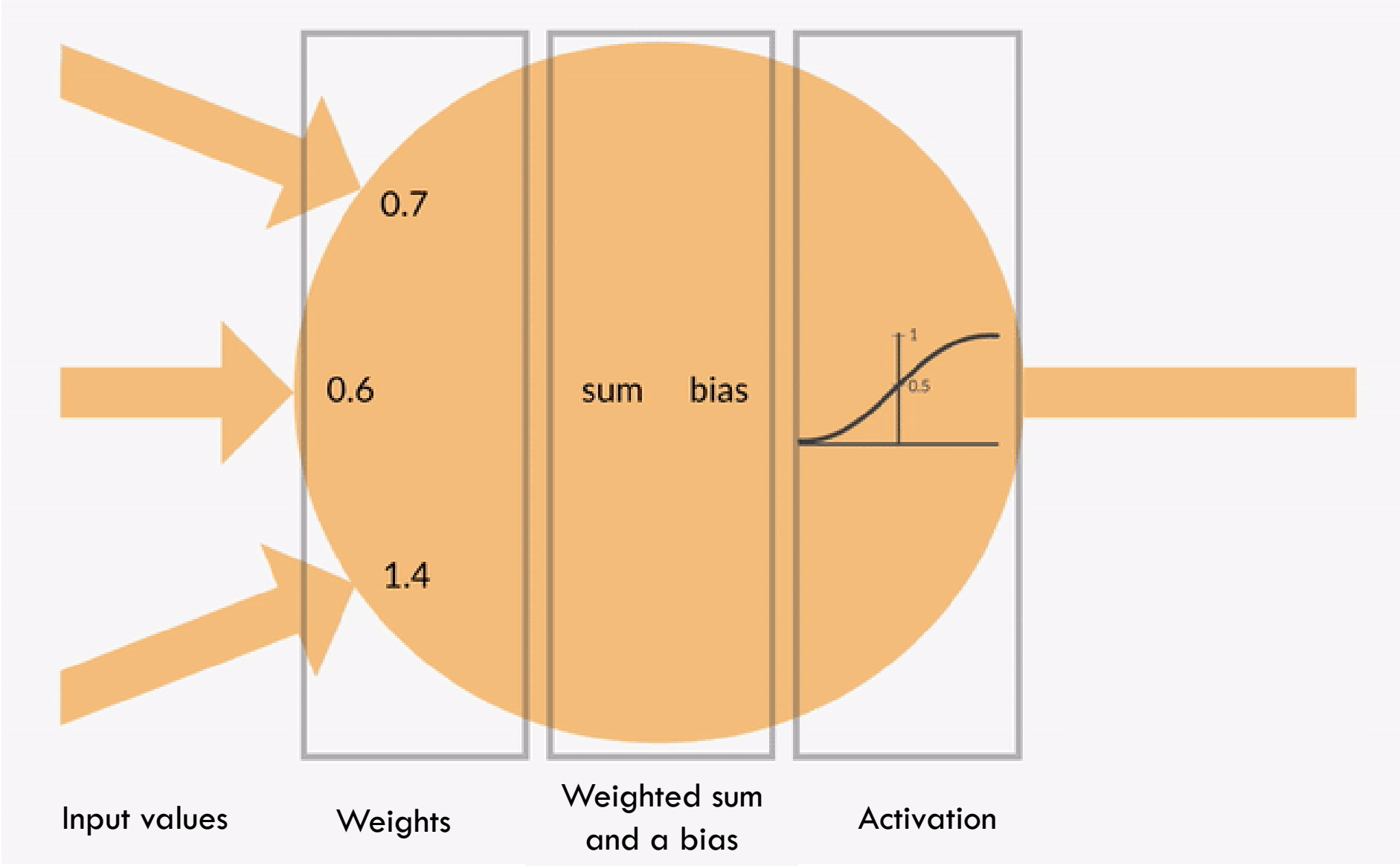
What is it?

Python: sklearn

PERCEPTRON



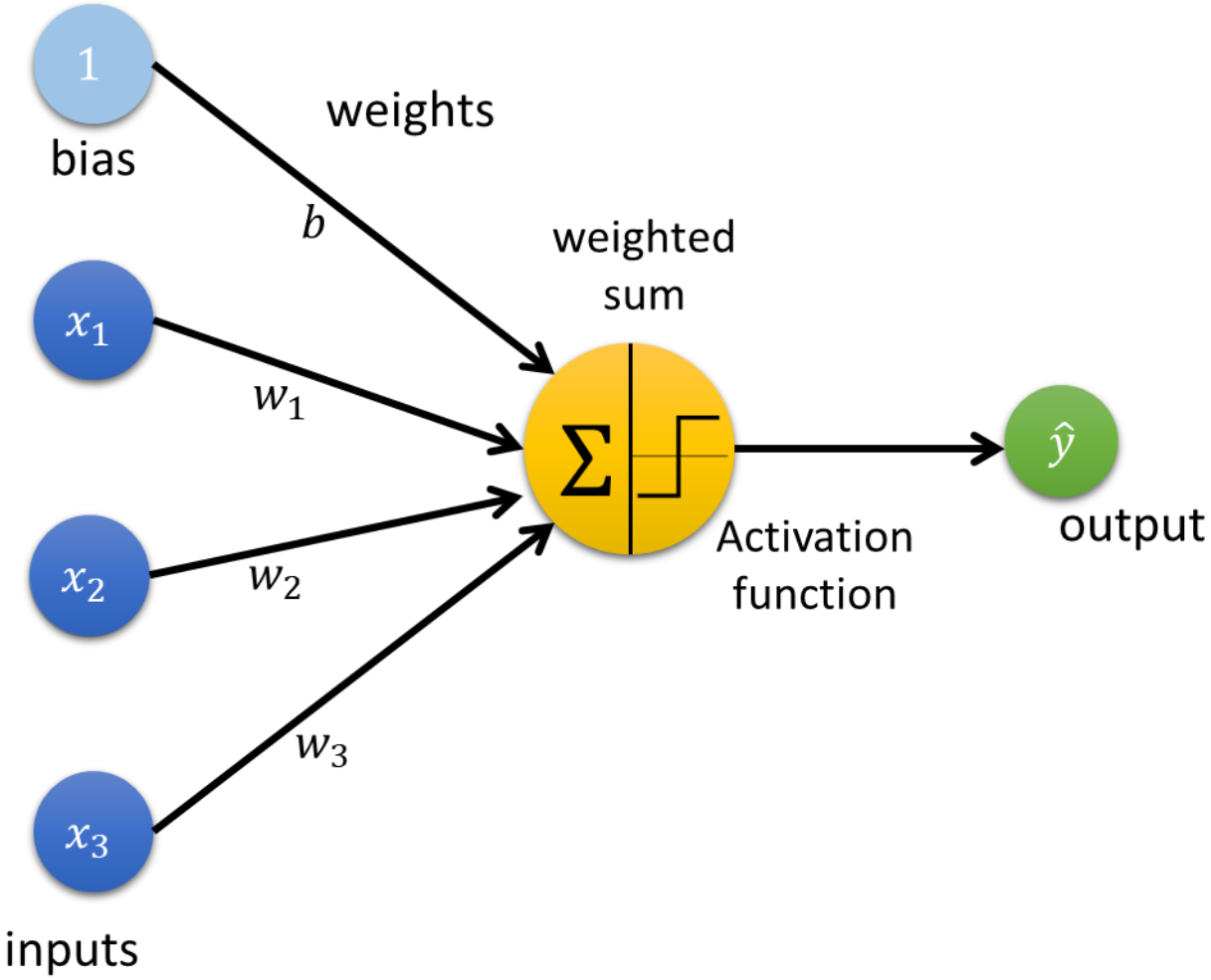
Rosenblatt Perceptrons are considered as the first generation of **neural networks.**
This **artificial neuron model is the basis of today's complex neural networks**

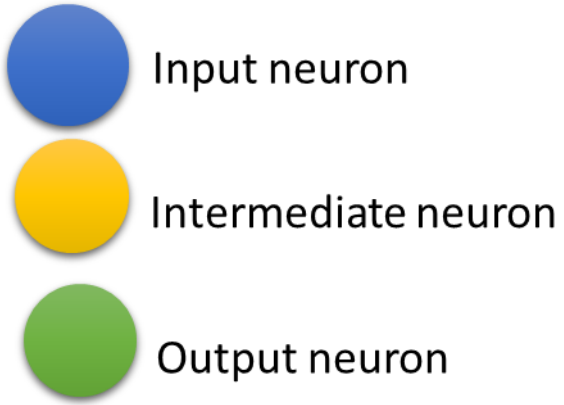


A single-layer perceptron is the basic unit of a neural network.
It consists of four main parts, including **input values, weights and a bias, a weighted sum, and activation function..**

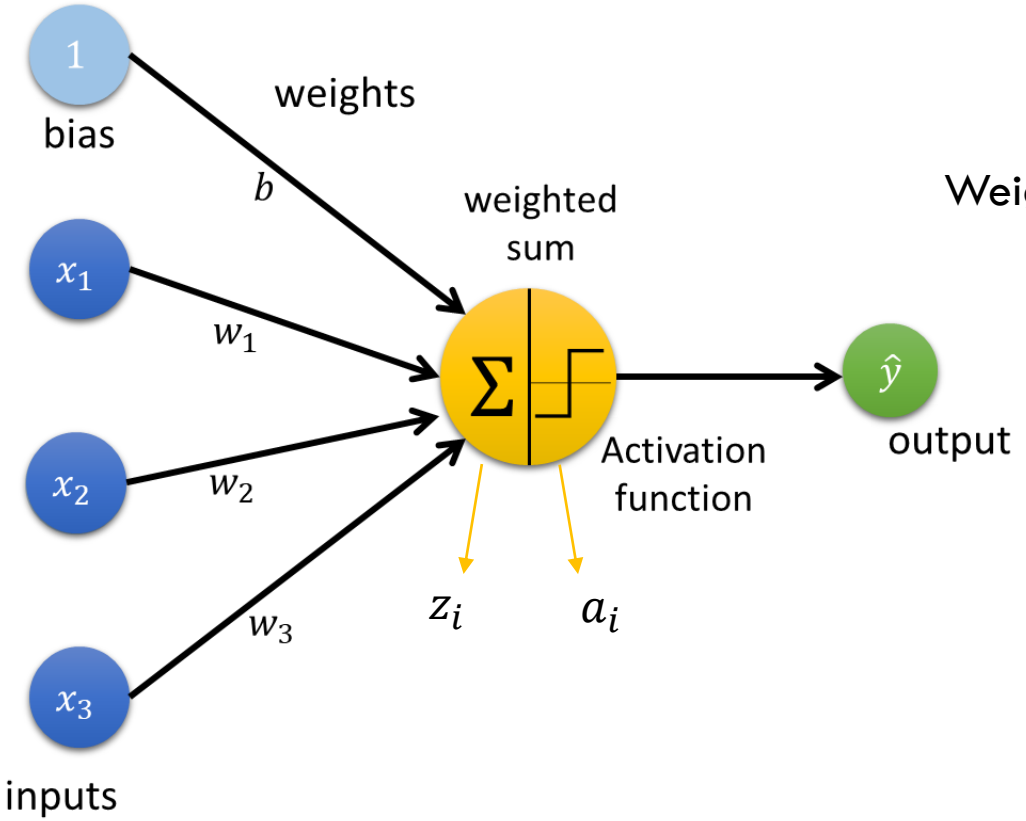
Extraído de:
https://www.analyticsvidhya.com/blog/2020/02/cnn-vs-rnn-vs-mlp-analyzing-3-types-of-neural-networks-in-deep-learning/

NOMENCLATURE





MODELO DO PERCEPTRON



Input vector, $(n_x, 1)$

$$\mathbf{x}^{(i)} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n_x} \end{bmatrix}$$

Weight vector, $(n_x, 1)$

$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_{n_x} \end{bmatrix}$$

Neuron state (scalar)

$$z^{(i)} = w_1 x_1 + \cdots + w_{n_x} x_{n_x} + b = \sum_{j=1}^{n_x} x_j^{(i)} w_j + b = \mathbf{w}^T \mathbf{x}^{(i)} + b$$

Activation, scalar

$$a_i = g(z_i)$$

Prediction, scalar

$$\hat{y}_i = a_i$$

ACTIVATION FUNCTION

The purpose of the activation function is to introduce nonlinearity into the output of a neuron.

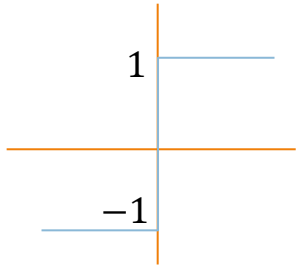
This nonlinearity is one of the factors that affect our results and the accuracy of our model.

If we use ANN with many hidden layers, a linear activation function will simply generate a series of affine transformations, so that the intermediate layers would be useless.

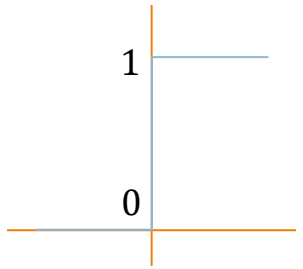
Unless we “transmit” nonlinearity, we are not computing interesting models, even if we delve deeper into neural networks.

Degrau Unitário (Unit step):

$$g(z) = \begin{cases} 1 & z \geq 0 \\ -1 & cc \end{cases}$$

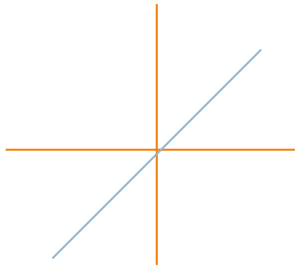


$$g(z) = \begin{cases} 1 & z \geq 0 \\ 0 & cc \end{cases}$$



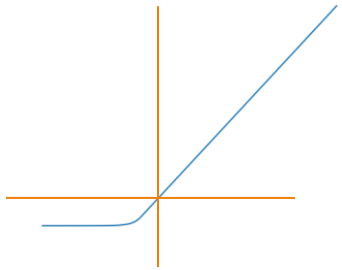
Linear:

$$g(z) = z$$



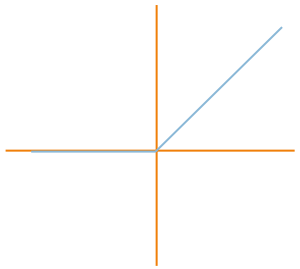
ELU (Exponential Linear Unit):

$$g(z) = \max(\alpha(e^z - 1), z)$$



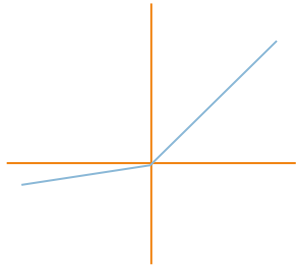
ReLU (Rectified Linear Unit):

$$g(z) = \max(0, z)$$



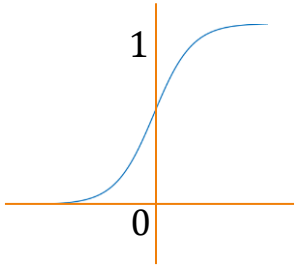
Leaky ReLU:

$$g(z) = \max(0.1z, z)$$



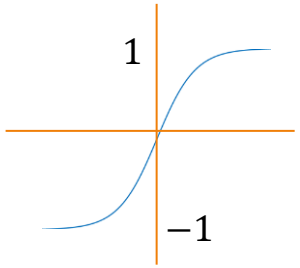
Sigmoide (logistic, sigmoid):

$$g(z) = \frac{1}{1 + e^{-z}}$$



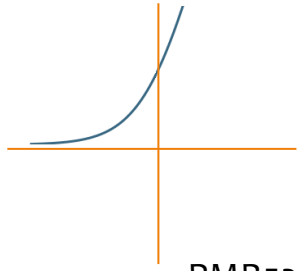
tanh (Hyperbolic tangent):

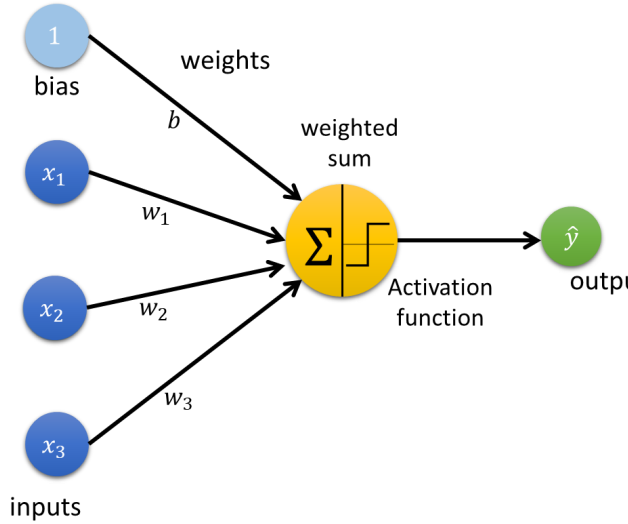
$$g(z) = \frac{e^{2z} - 1}{e^{2z} + 1}$$

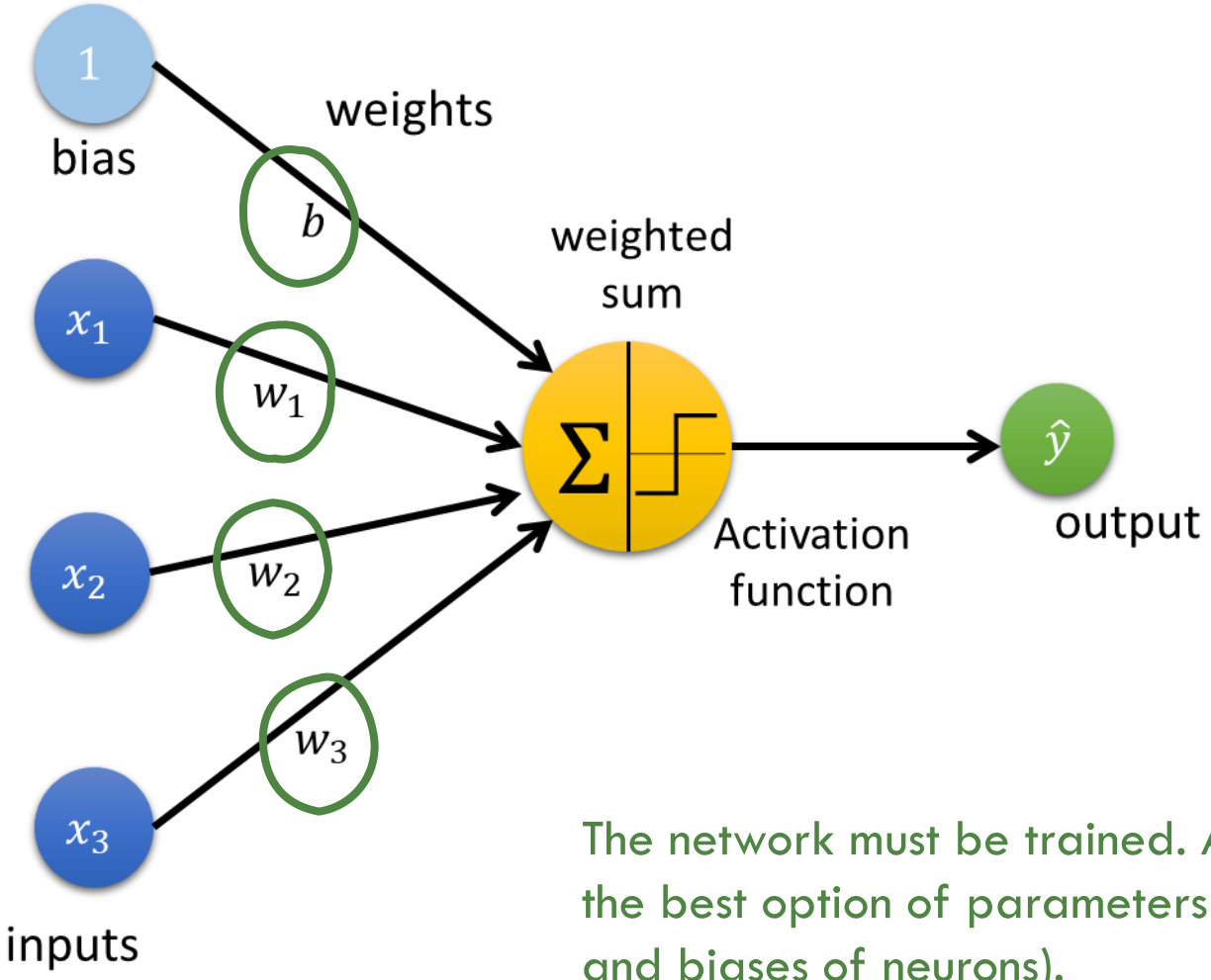


Softplus

$$g(z) = \ln 1 + e^z$$







The network must be trained. And it is the training that will show us the best option of parameters to be used (weights of connections and biases of neurons).

HOW TO TRAIN OUR NN?

The objective of training an ANN is to calculate its parameters (weights and biases) in order to minimize a cost function.

The cost function is based on a function of the difference between the actual Y and predicted $\hat{Y}$.

Mean Squared Error

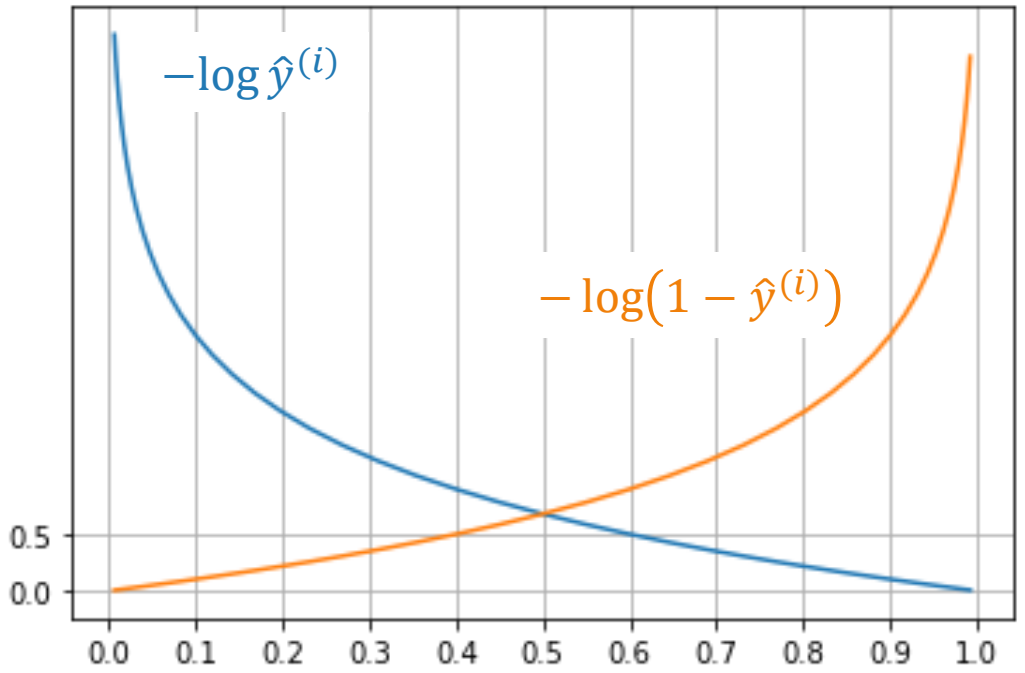
$$E(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)}) = \sum_{j=1}^{n_y} \left(\hat{y}_j^{(i)} - y_j^{(i)}\right)^2 = \left\|\hat{\mathbf{y}}^{(i)} - \mathbf{y}^{(i)}\right\|_2^2$$

Cost function

$$J(\mathbf{W}, \mathbf{B}) = \frac{1}{m} \sum_{i=1}^{m} E(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)}) = \frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{n_y} \left(\hat{y}_j^{(i)} - y_j^{(i)}\right)^2 = \frac{1}{m} \sum_{i=1}^{m} \left\|\hat{\mathbf{y}}^{(i)} - \mathbf{y}^{(i)}\right\|_2^2$$

Logistic error $(0-1)$

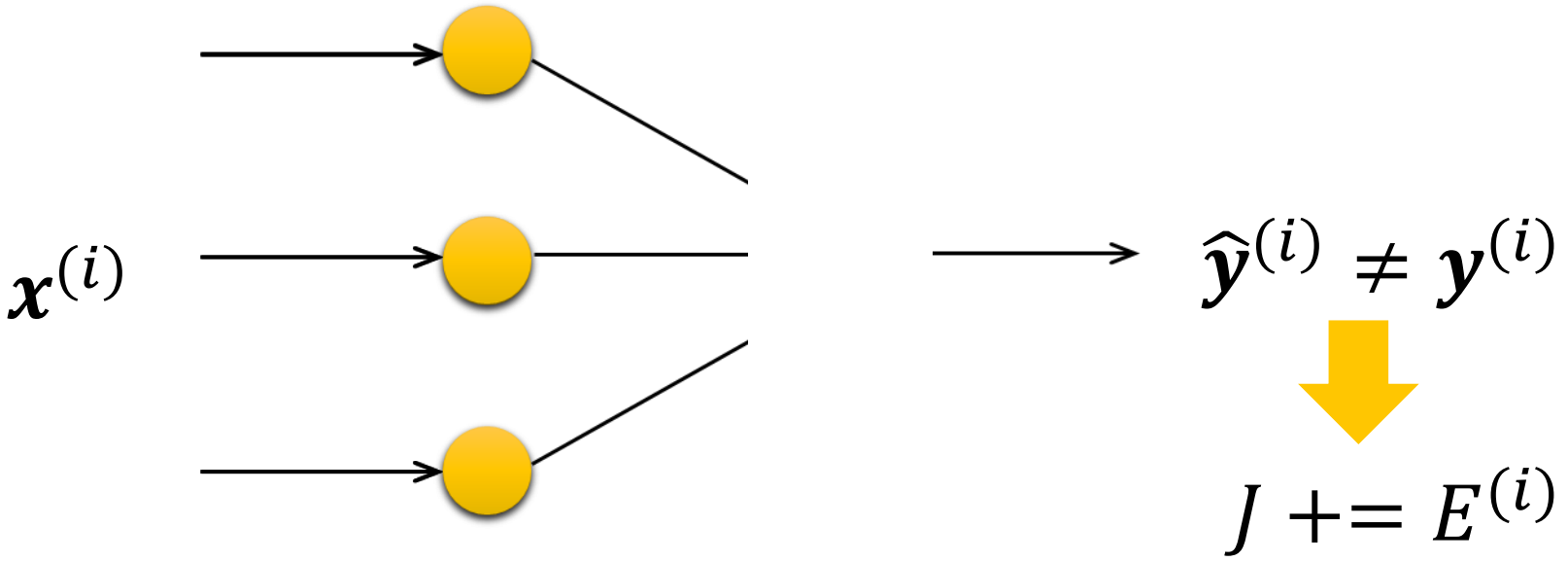
$$L(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)}) = -y^{(i)} \log \hat{y}^{(i)} - \left(1 - y^{(i)}\right) \log \left(1 - \hat{y}^{(i)}\right)$$



Cost function

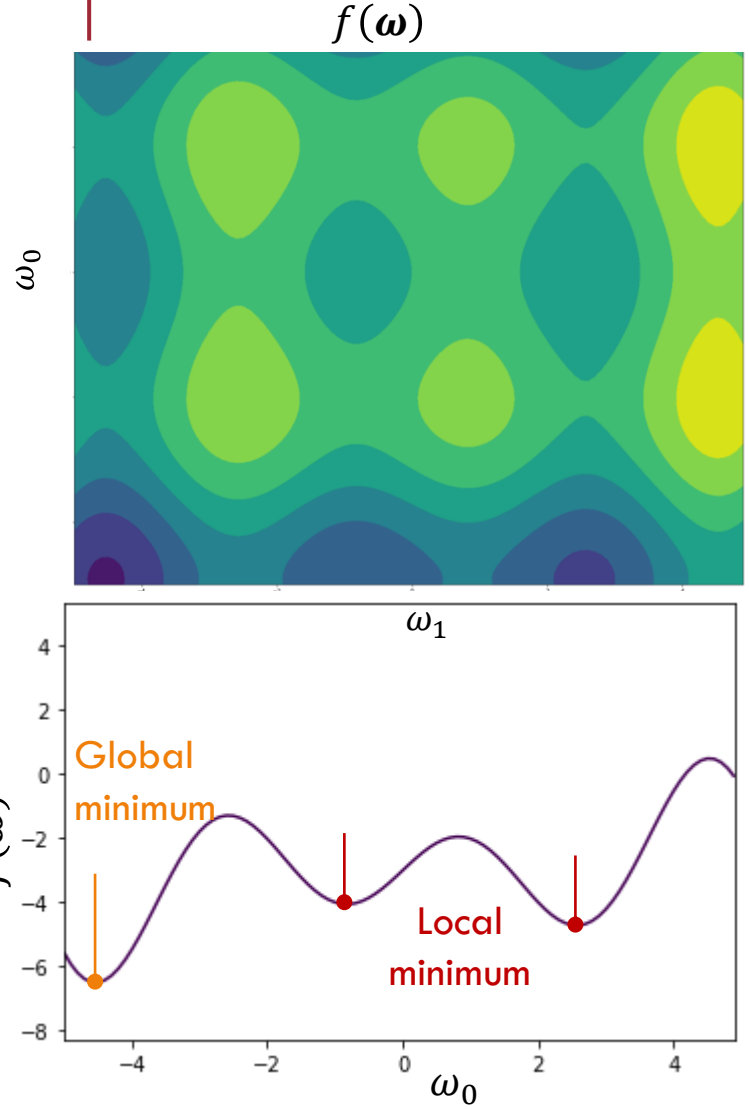
$$J(\mathbf{W}, \mathbf{B}) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)}) = \frac{1}{m} \sum_{i=1}^{m} -y^{(i)} \log \hat{y}^{(i)} - \left(1 - y^{(i)}\right) \log \left(1 - \hat{y}^{(i)}\right)$$

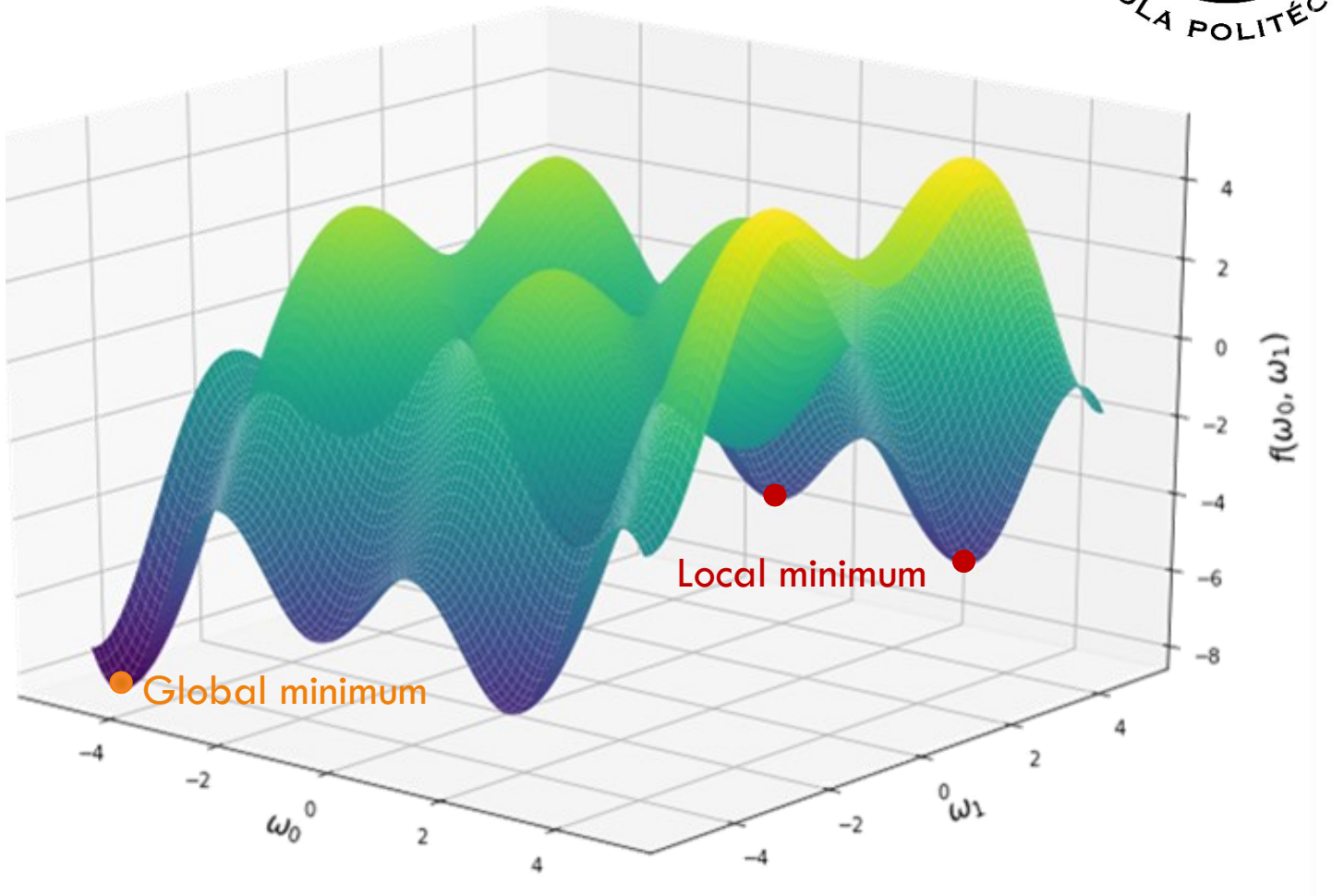
WHAT DOES TRAINING NEURAL NETWORKS MEAN?



OPTIMIZATION PROBLEM

HOW TO SOLVE THE OPTIMIZATION PROBLEM?





GRADIENT DESCENTE ALGORITHM

- How does gradient descent work?
- What types are used today?
- What are its advantages and disadvantages?

WHY STUDY GRADIENT DESCENT?

Just because Gradient Descent is the heart and soul of most machine learning algorithms.

Gradient Descent is by far the most popular optimization strategy used in machine learning and deep learning right now.

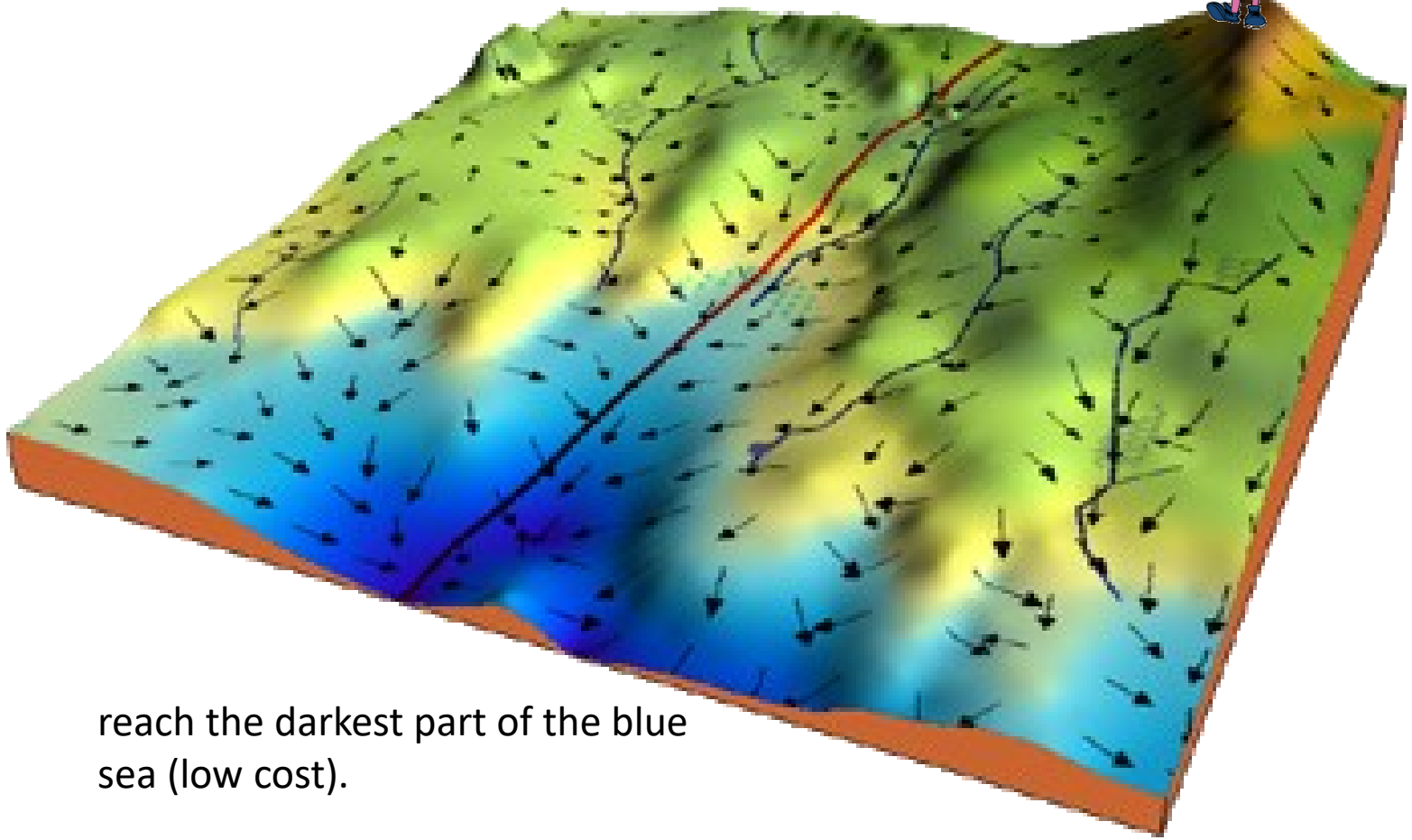
It is used in training data models, can be combined with all algorithms, and is easy to understand and implement.

Everyone who works with machine learning **must** understand its concept.

Suppose you are at the top of a very high hill.

Your objective is to reach the sea as fast as possible.

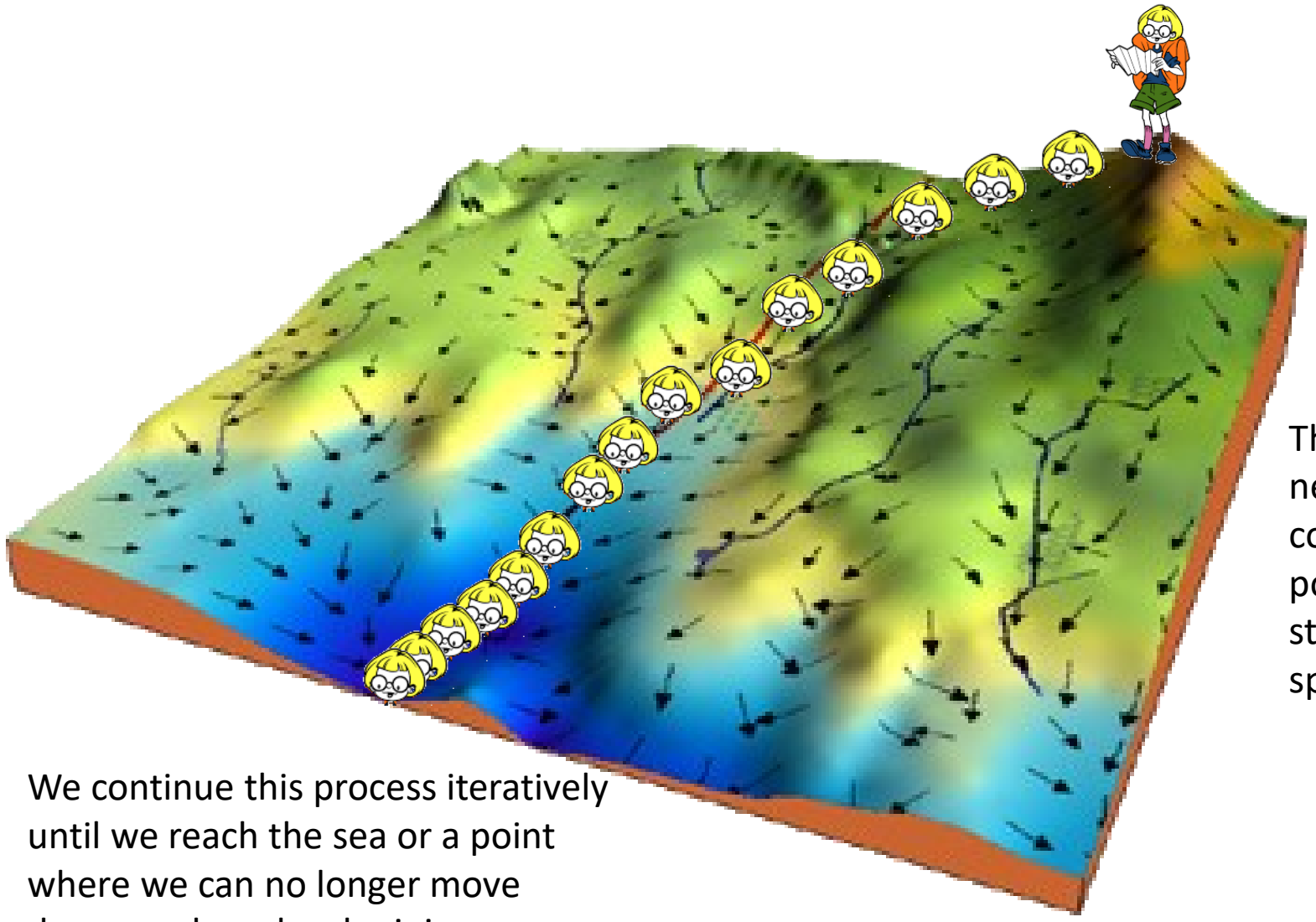
Starting from the top of the hill (high cost)



reach the darkest part of the blue sea (low cost).

The arrows represent the direction of steepest descent (negative gradient) from any given point - the direction that decreases the cost function as quickly as possible.

Fonte: https://ml-cheatsheet.readthedocs.io/en/latest/gradient_descent.html

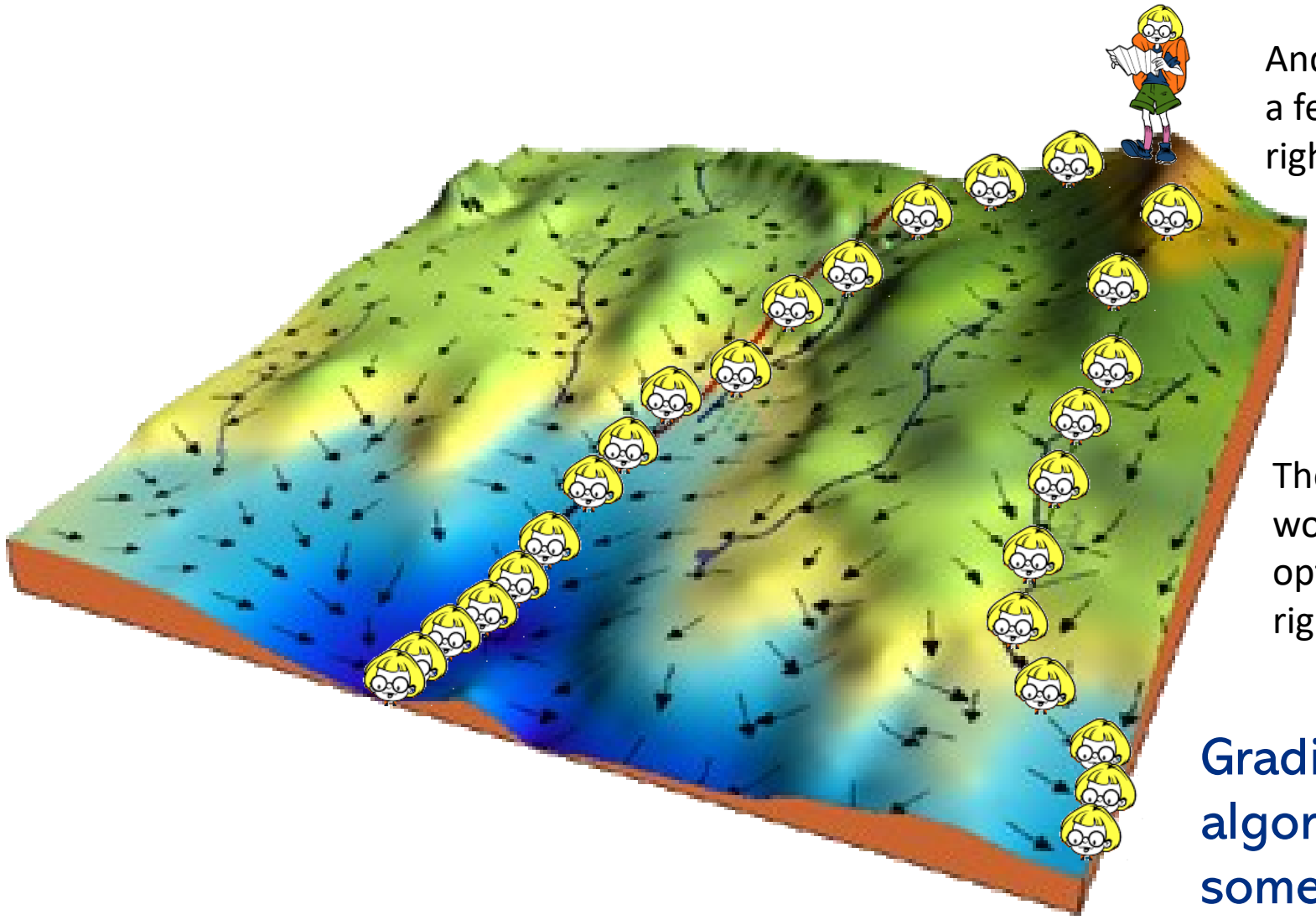


We take our first step downhill in the direction specified by the negative gradient.

Then we recalculate the negative gradient (via the coordinates of our new point) and take another step in the direction it specifies.

We continue this process iteratively until we reach the sea or a point where we can no longer move downwards - a local minimum.

Fonte: https://ml-cheatsheet.readthedocs.io/en/latest/gradient_descent.html

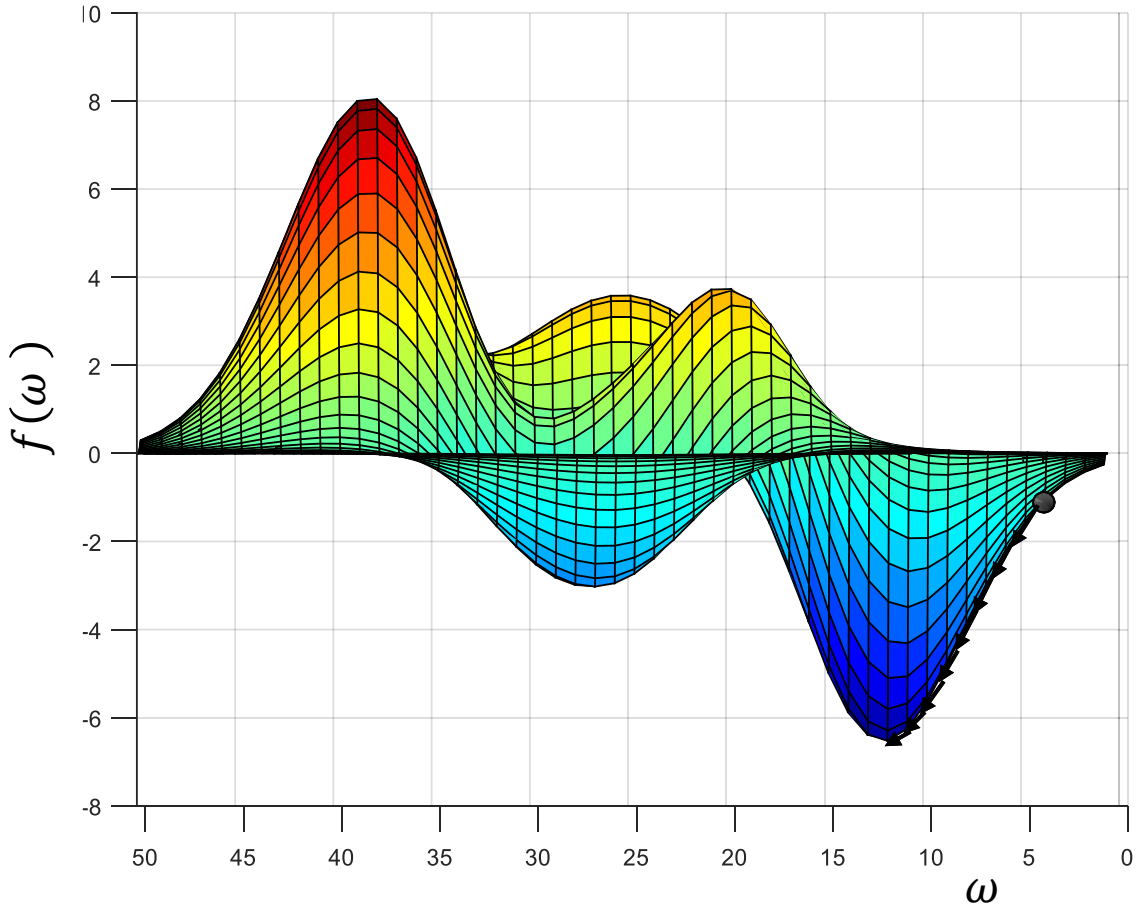


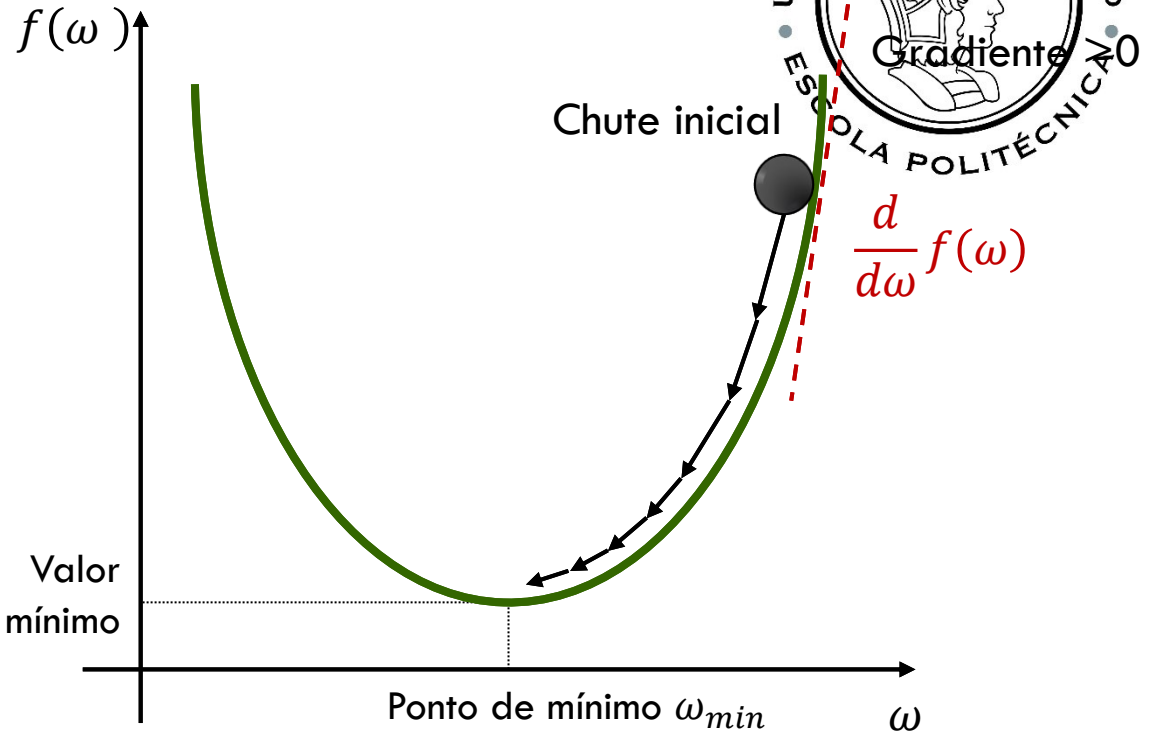
And if you started just a few steps to the right?

Then the gradient descent would take you to another optimal location on the right.

Gradient Descending (GD) is an algorithm used to find the minimum of some function by iteratively moving in the direction of **steepest descent**.

Fonte: https://ml-cheatsheet.readthedocs.io/en/latest/gradient_descent.html





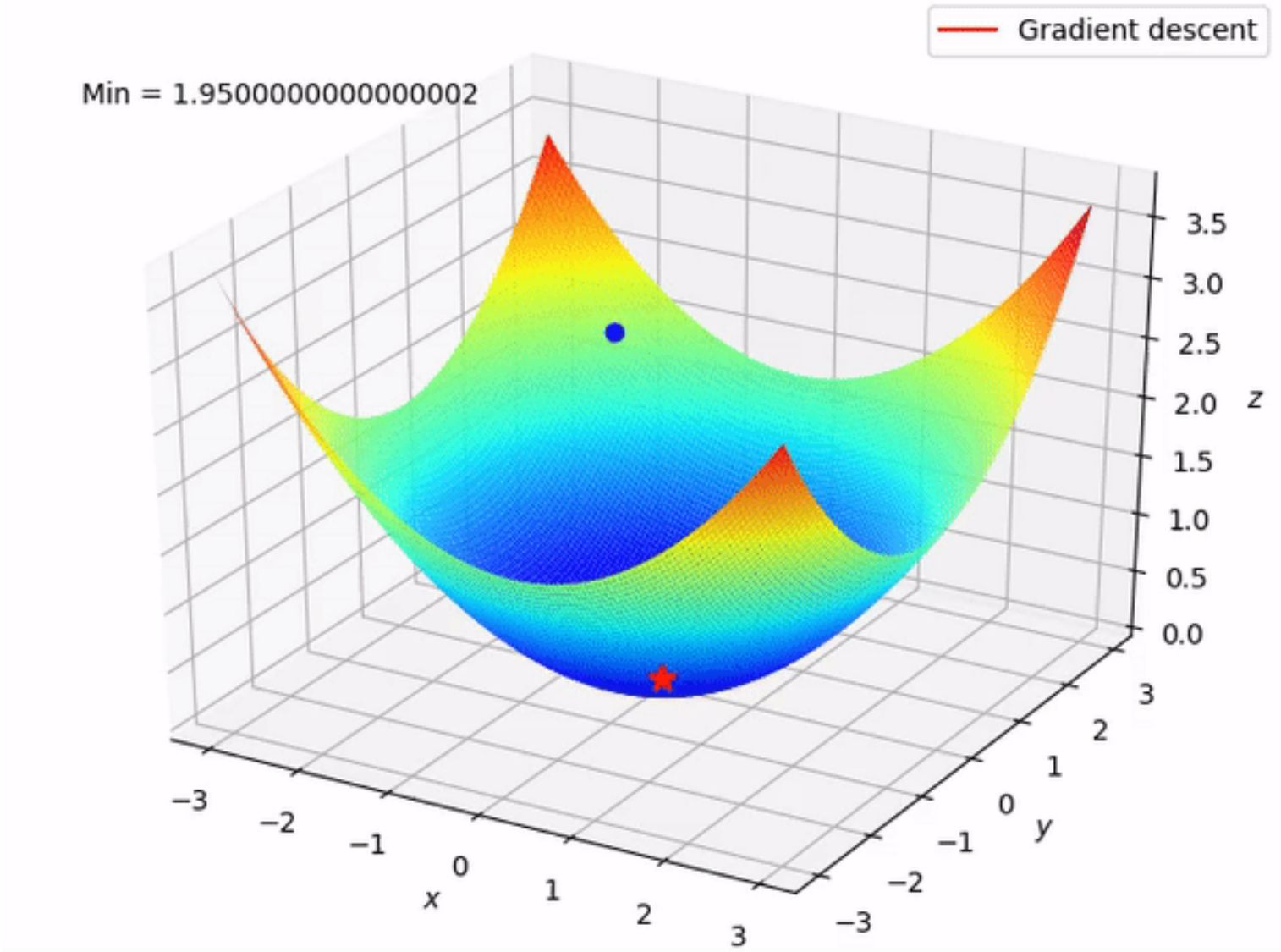
While $\|\nabla f(\omega)\| > \varepsilon$:

$$\omega^{(i+1)} := \omega^{(i+1)} - \alpha \nabla f\left(\omega^{(i)}\right)$$

$$i \mathrel{+}= 1$$

We start with a random point in the function and …

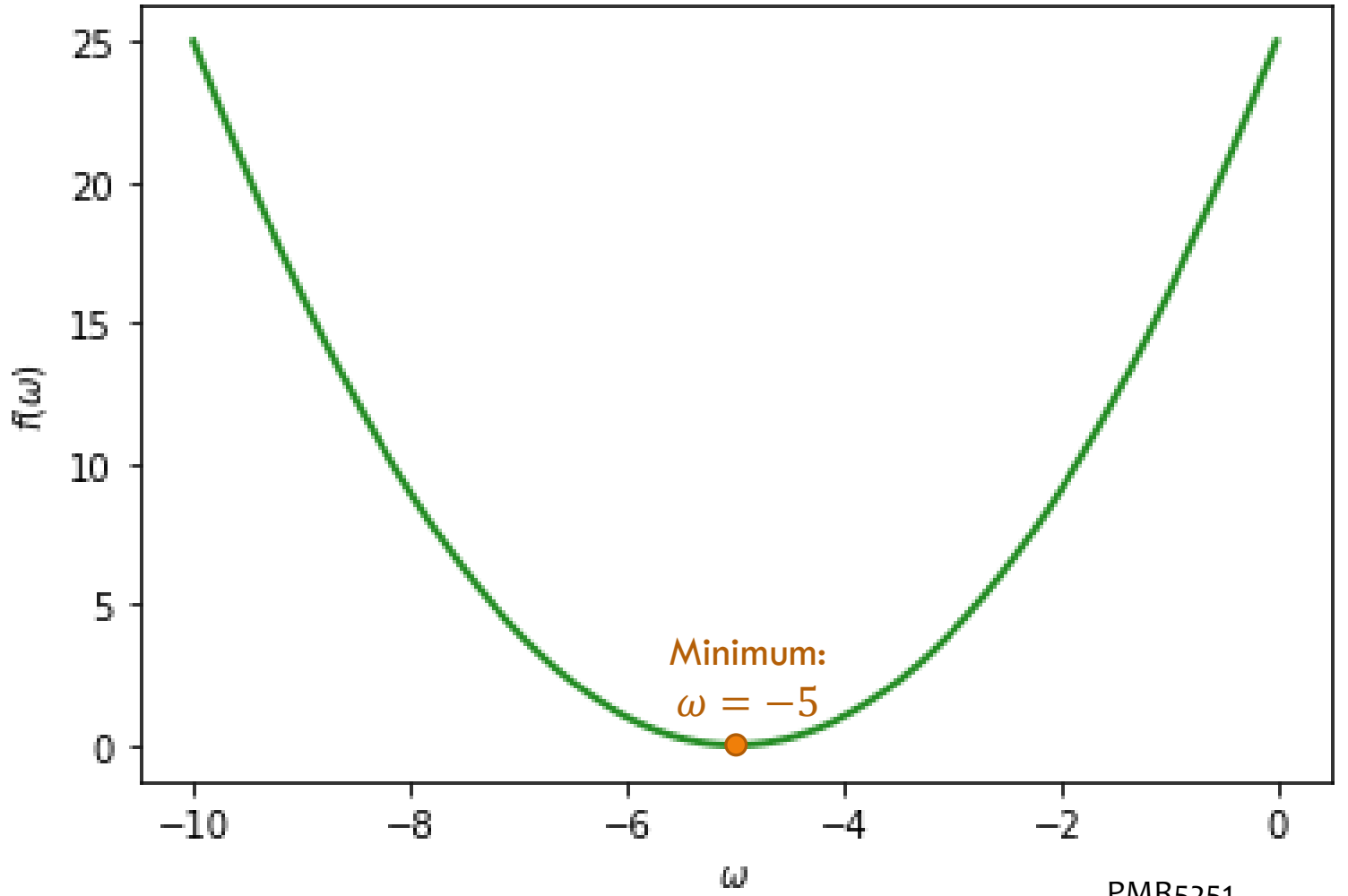
… move in the negative direction of the function's gradient, to reach the local/global minima.



Fonte: https://medium.com/@lucasoliveiras/regress%C3%A3o-linear-do-zero-com-python-ef74a81c4b84

EXAMPLE

$$f(\omega) = (\omega + 5)^2$$



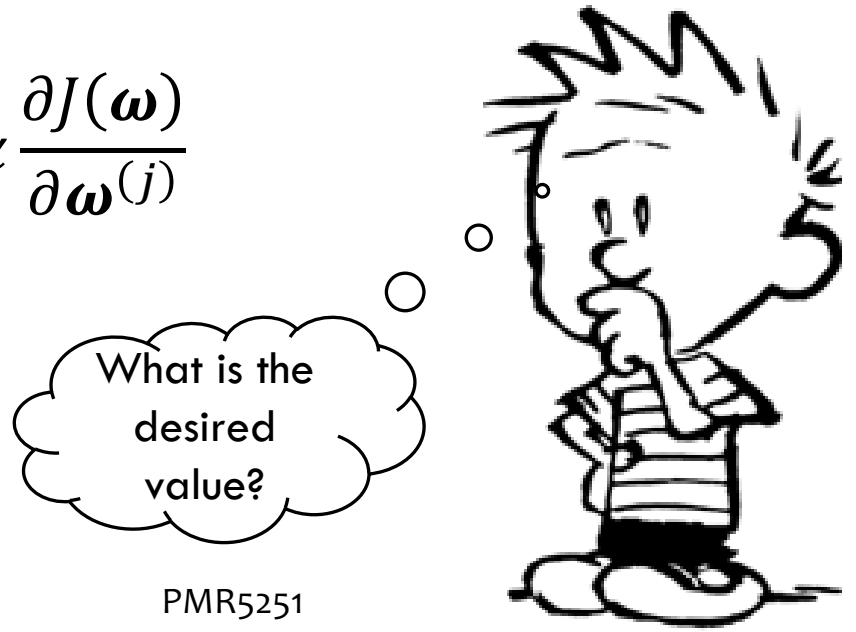
Find the minimum value of the function $\omega = -5$ using the Gradient Descent method.

GRADIENT DESCENT ALGORITHM

1. Parameter initialization: Assign an initial value $\boldsymbol{\omega}_0$ to the parameters;
2. Calculate the Jacobian;
3. Update the parameters in the opposite direction of the gradient in order to minimize the value of the function,

$$\boldsymbol{\omega}^{(j+1)} = \boldsymbol{\omega}^{(j)} - \alpha \frac{\partial J(\boldsymbol{\omega})}{\partial \boldsymbol{\omega}^{(j)}}$$

Repeat steps 2 and 3 until the desired value is obtained.



WHEN TO STOP?

When the variation $\boldsymbol{\omega}^{(j+1)} - \boldsymbol{\omega}^{(j)}$ for menor que uma precisão ε. For example, $\varepsilon = 0{,}000001$.

When the number of iterations exceeds a limit T. For example, $T = 10000$.

LET'S FOLLOW THE STEPS

$$f(\omega) = (\omega + 5)^2, \qquad \frac{df}{d\omega} = 2(\omega + 5)$$

1. Initialize with $x = 3$.
2. Modify ω in the negative direction of the gradient. How much to move? For this we need the learning rate. Let's suppose $\alpha \leftarrow 0{,}01$,
   - **Iteration 01**

$$J_{\omega^0} = \frac{df}{d\omega} = 2(3 + 5) = 16 \qquad \omega^1 = \omega^0 - \alpha J_{\omega^0} = 3 - 0{,}01 \times 16 = 2{,}84$$

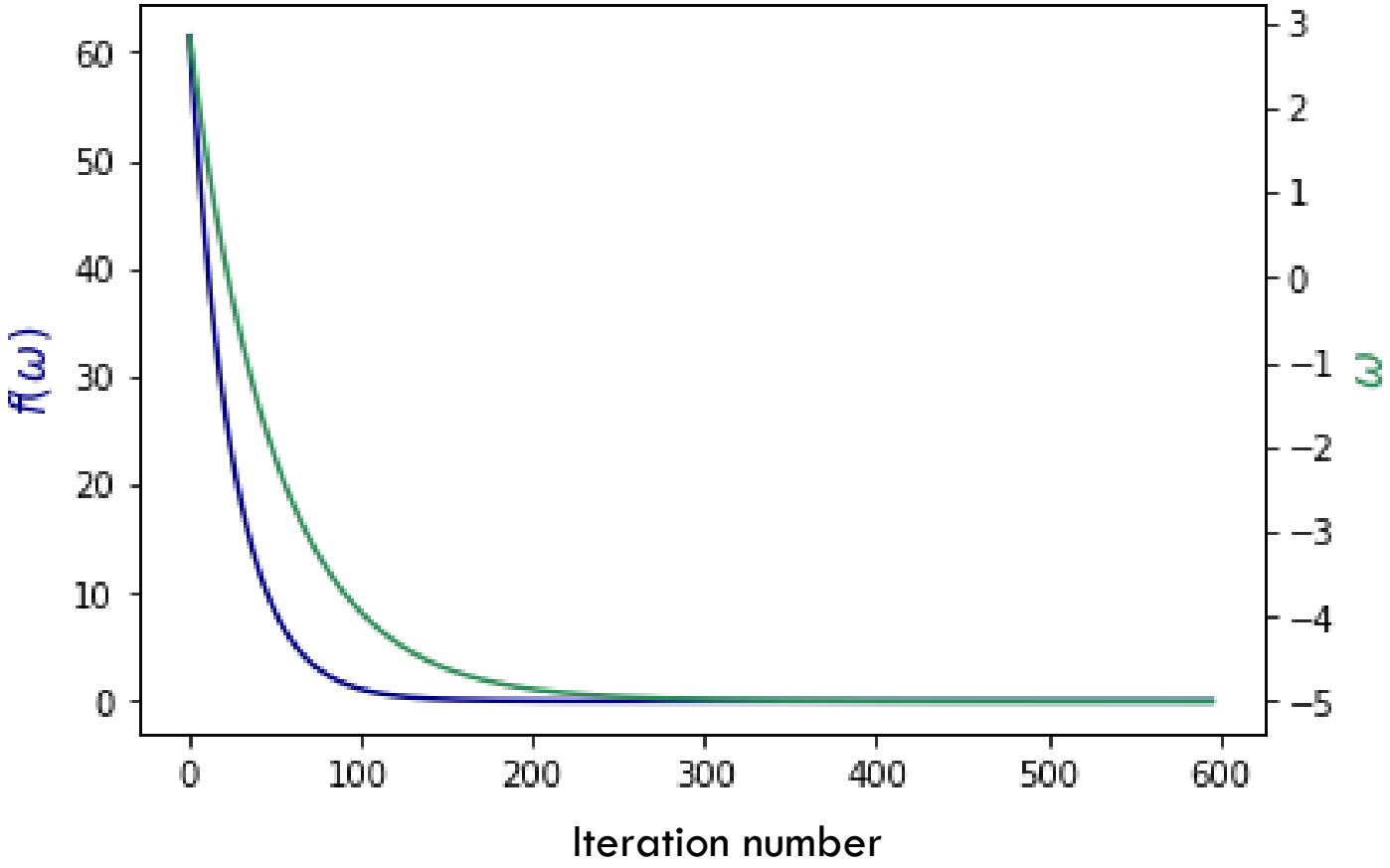
   - **Iteration 02**

$$J_{\omega^1} = \frac{df}{d\omega} = 2(2{,}84 + 5) = 15{,}7 \qquad \omega^2 = \omega^1 - \alpha J_{\omega^1} = 2{,}84 - 0{,}01 \times 15{,}7 = 2{,}68$$

   - **Iteration 03**

$$J_{\omega^2} = \frac{df}{d\omega} = 2(2{,}68 + 5) = 15{,}36 \omega^3 = \omega^2 - \alpha J_{\omega^2} = 2{,}68 - 0{,}01 \times 15{,}36 = 2{,}53$$

Slowly on the way to the minimum...


f(ω)
3
Iteration number
0
10
20
30
40
50
60
−5
−4
−3
−2
−1
0
1
2
3
0
100
200
300
400
500
600

LEARNING RATE α

Learning rate α controls the step size in each iteration.

Selecting the correct value is a critical model decision, α cannot be too big or too small...

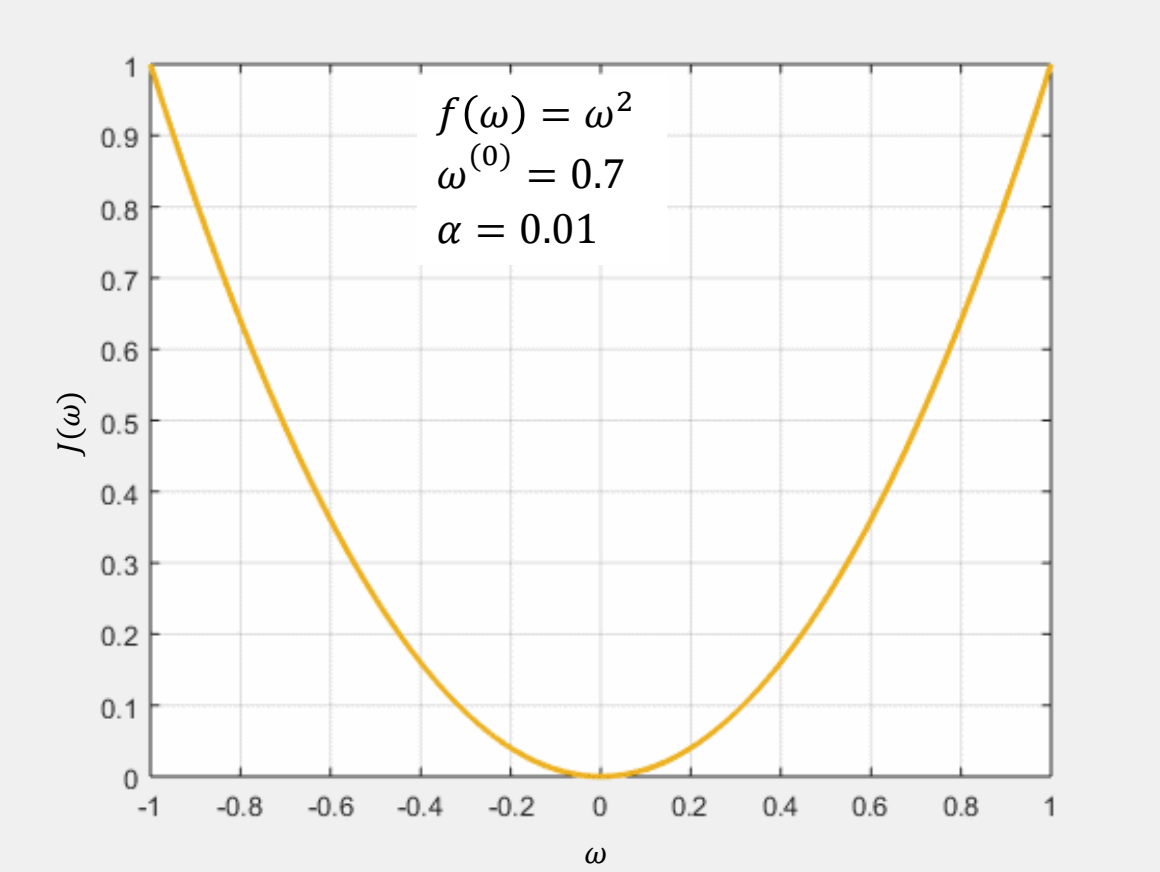
As we approach a local minimum, the gradient descent will automatically take smaller steps. So, there is no need to decrease α over time.

Try ... $0,001 - 0,003 - 0,01 - 0,03 - 0,1 - 0,3 - 1$ ...

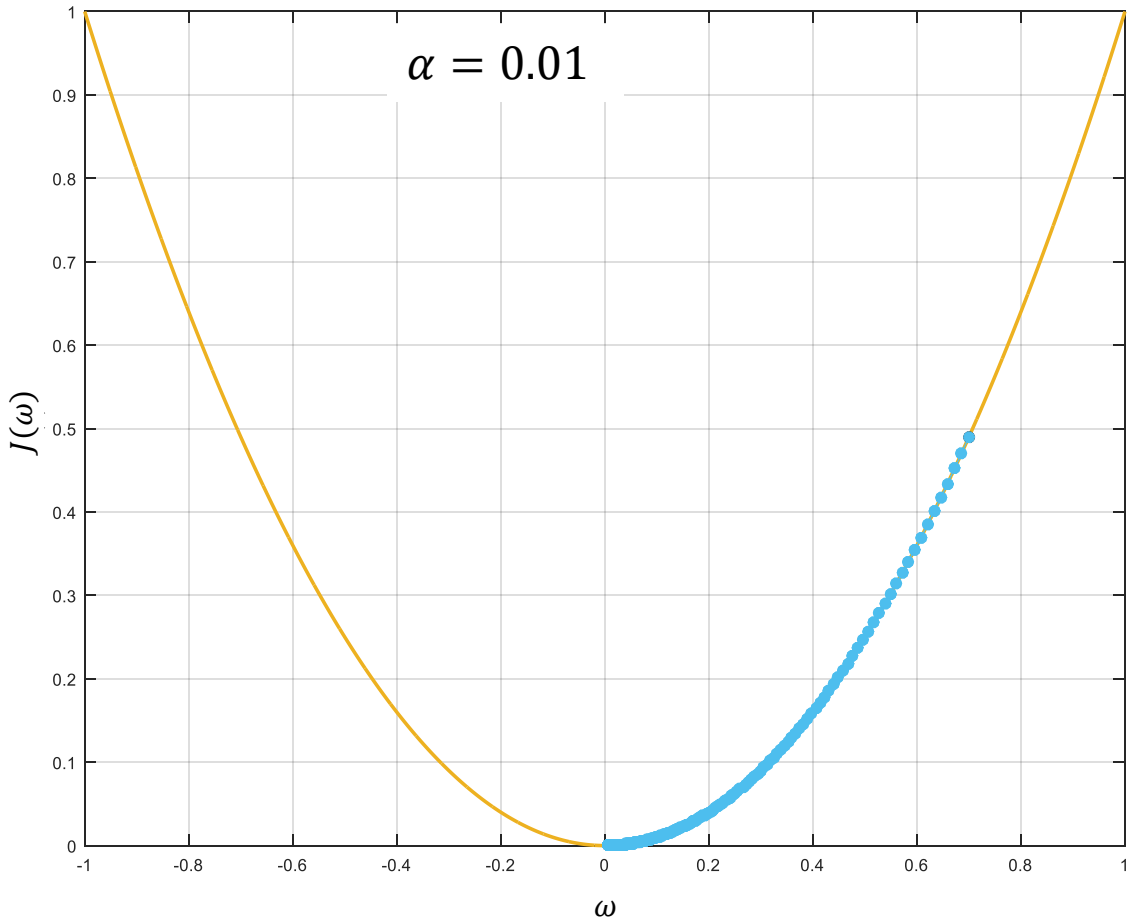
Increase/decrease ~ 3x the previous value

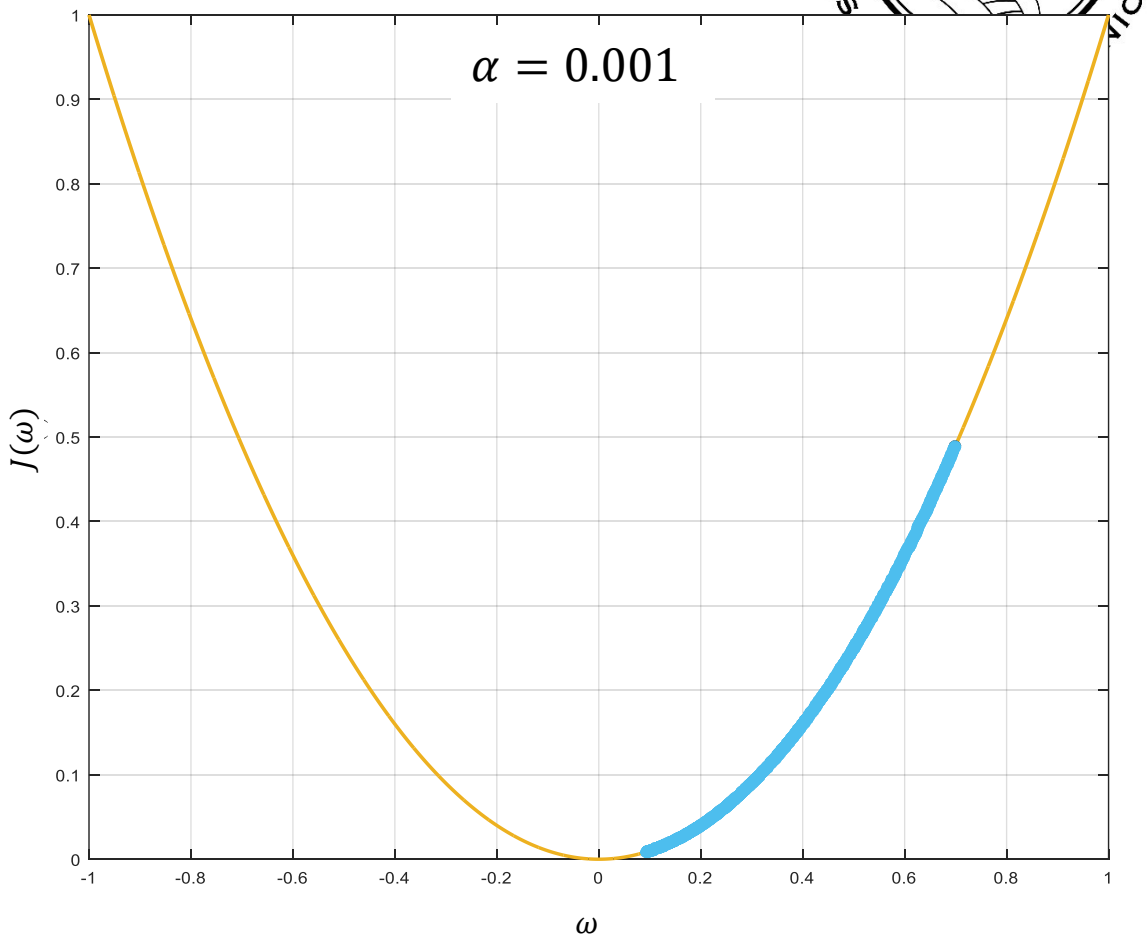
LOW α

Let's simplify the cost function something like $f(\omega) = \omega^2$

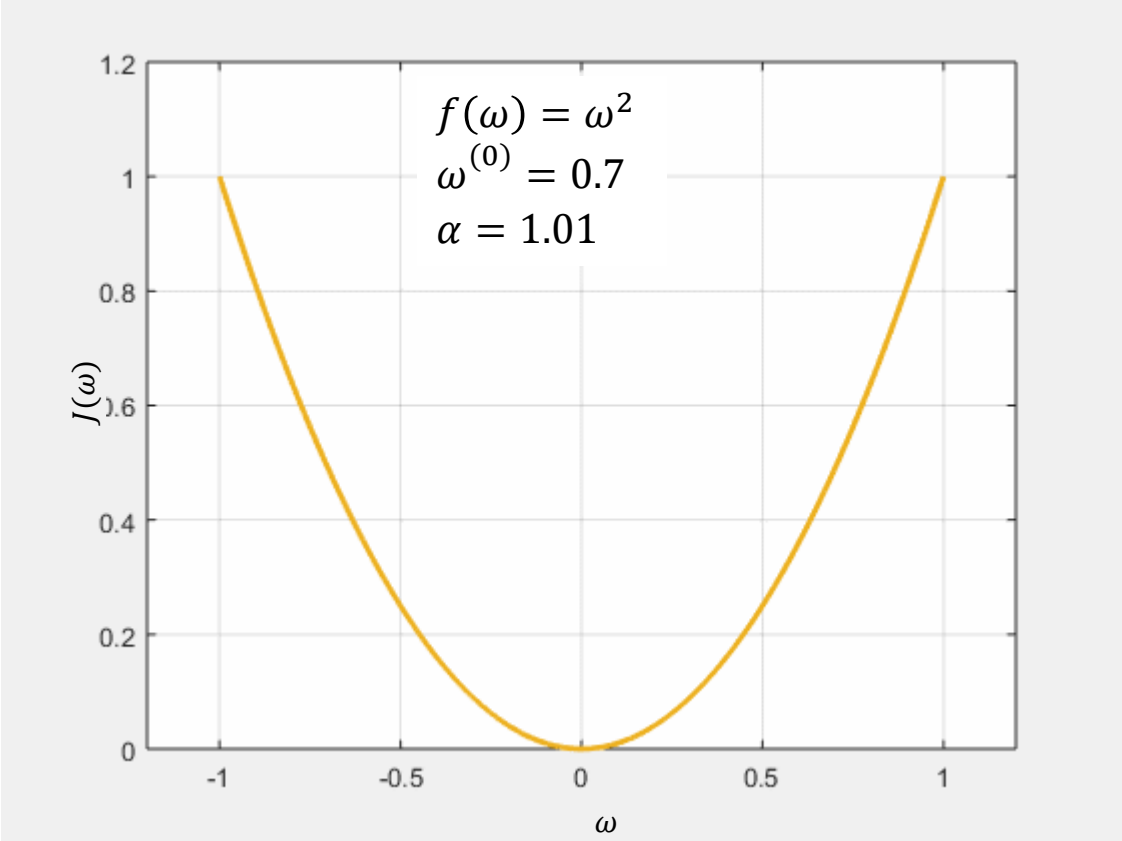


A very low learning rate is more accurate, but the gradient calculation is time consuming, so it will take a long time to get to the end. ... We can confidently move in the direction of the negative gradient, as we are recalculating it very often, but "we will go down too slowly";


$\alpha = 0.01$
$J(\omega)$
ω

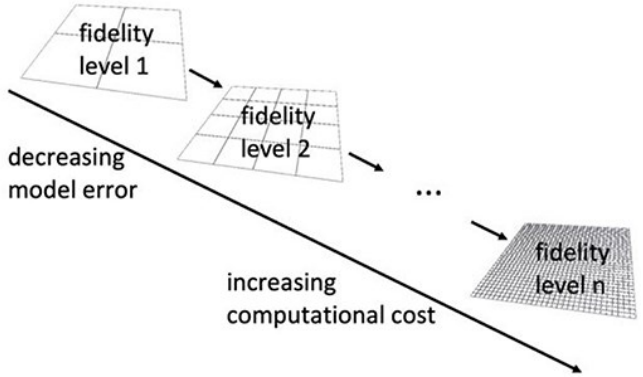

$\alpha = 0.001$
$J(\omega)$
ω

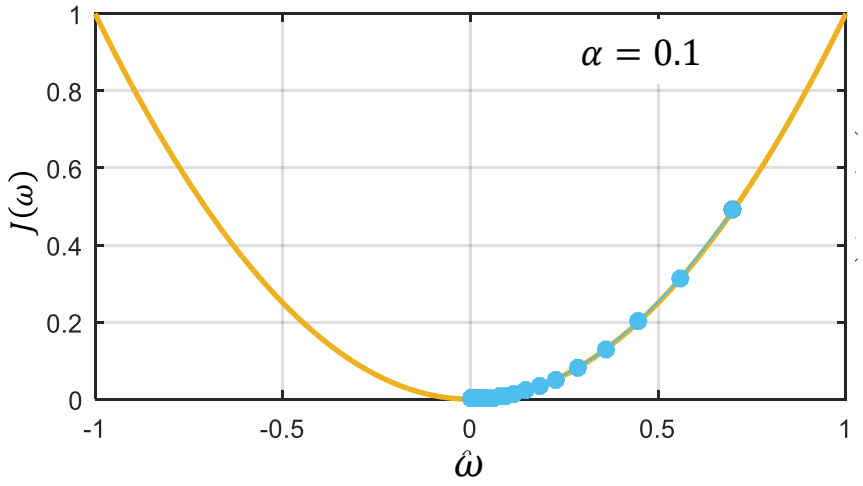
HIGH α

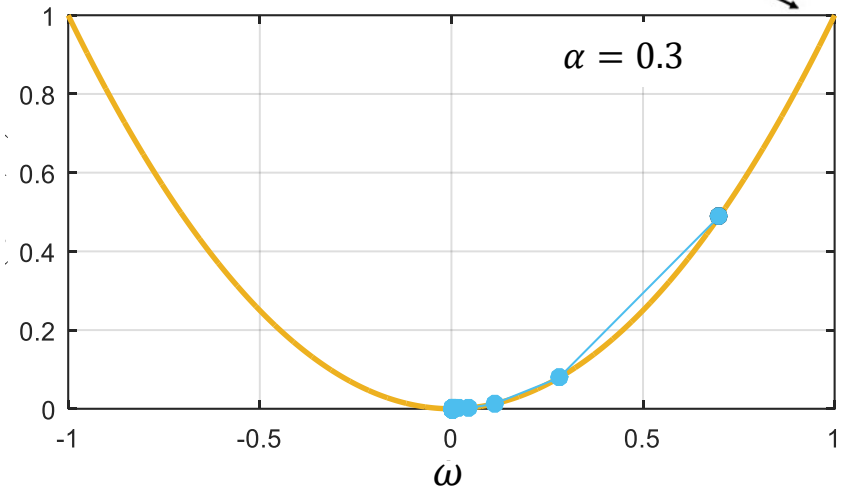


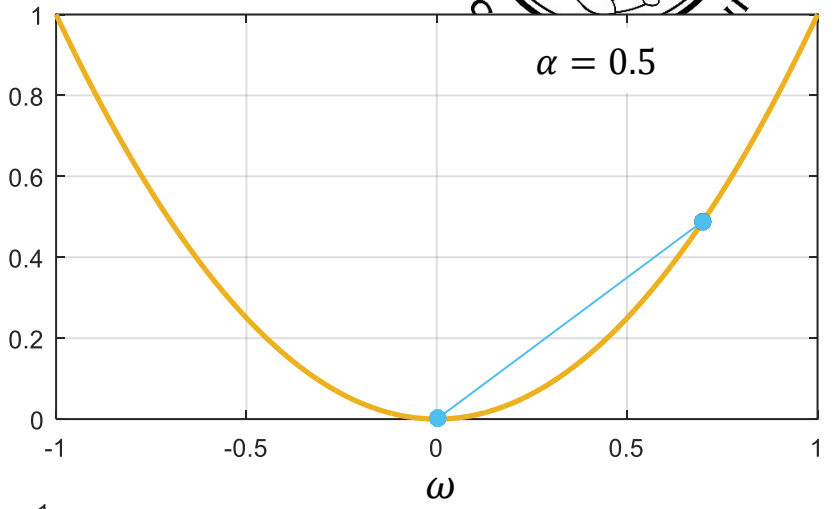
If we use a very high learning rate, we can cover more distance with each step, but we run the risk of going over the low point, as the gradient is constantly changing. The method diverges because we are "going down too fast".

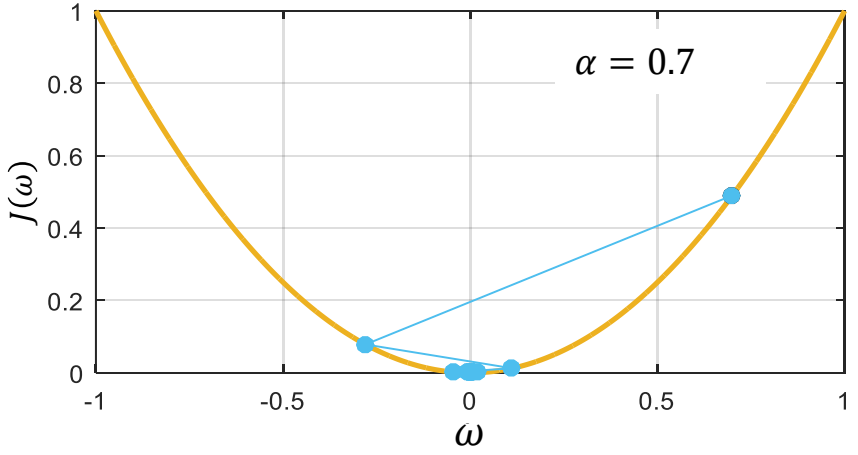
$$f(\omega) = \omega^2$$
$$\omega^{(0)} = 0.7$$

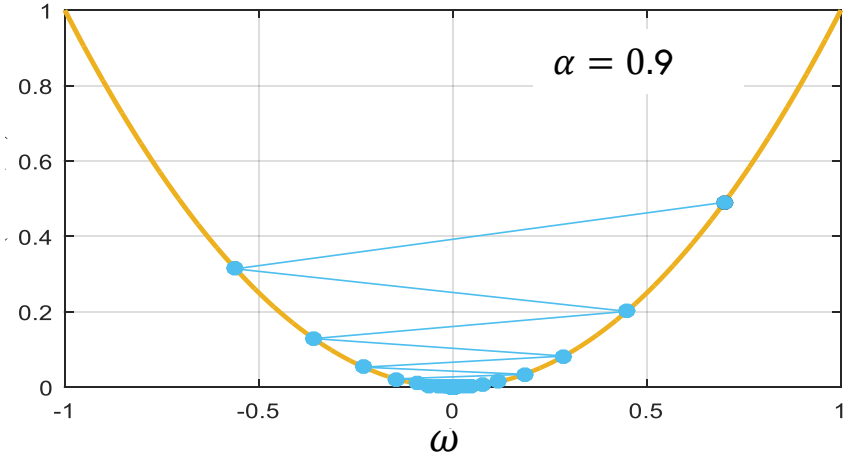


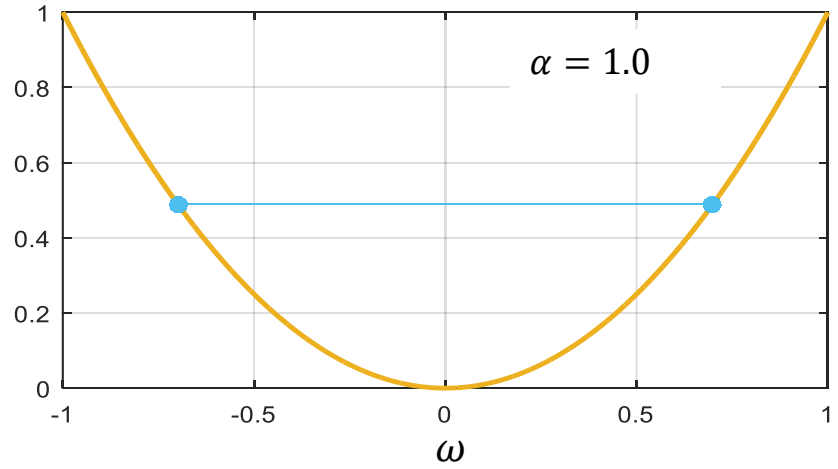




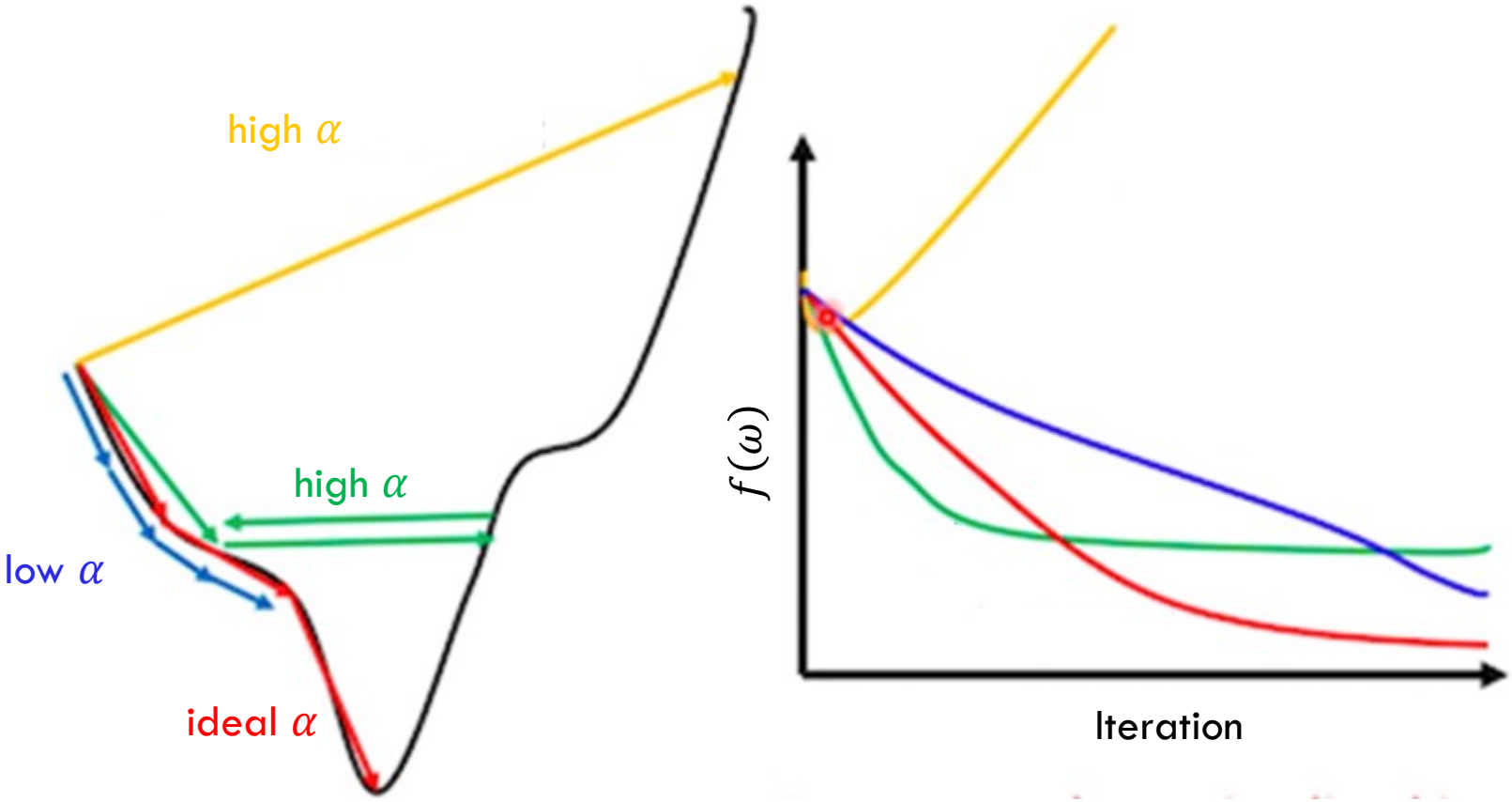




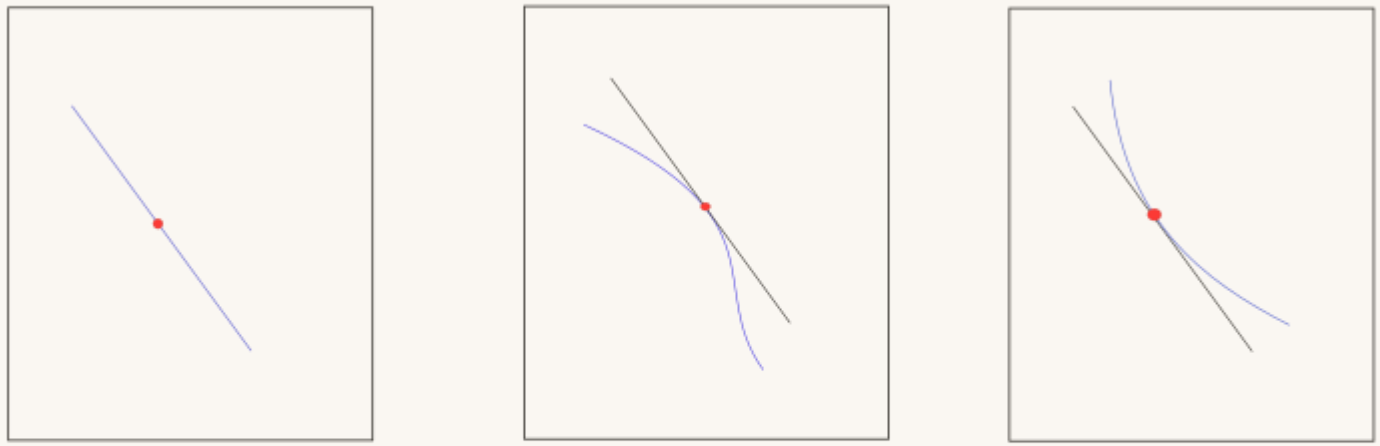




IDEAL α

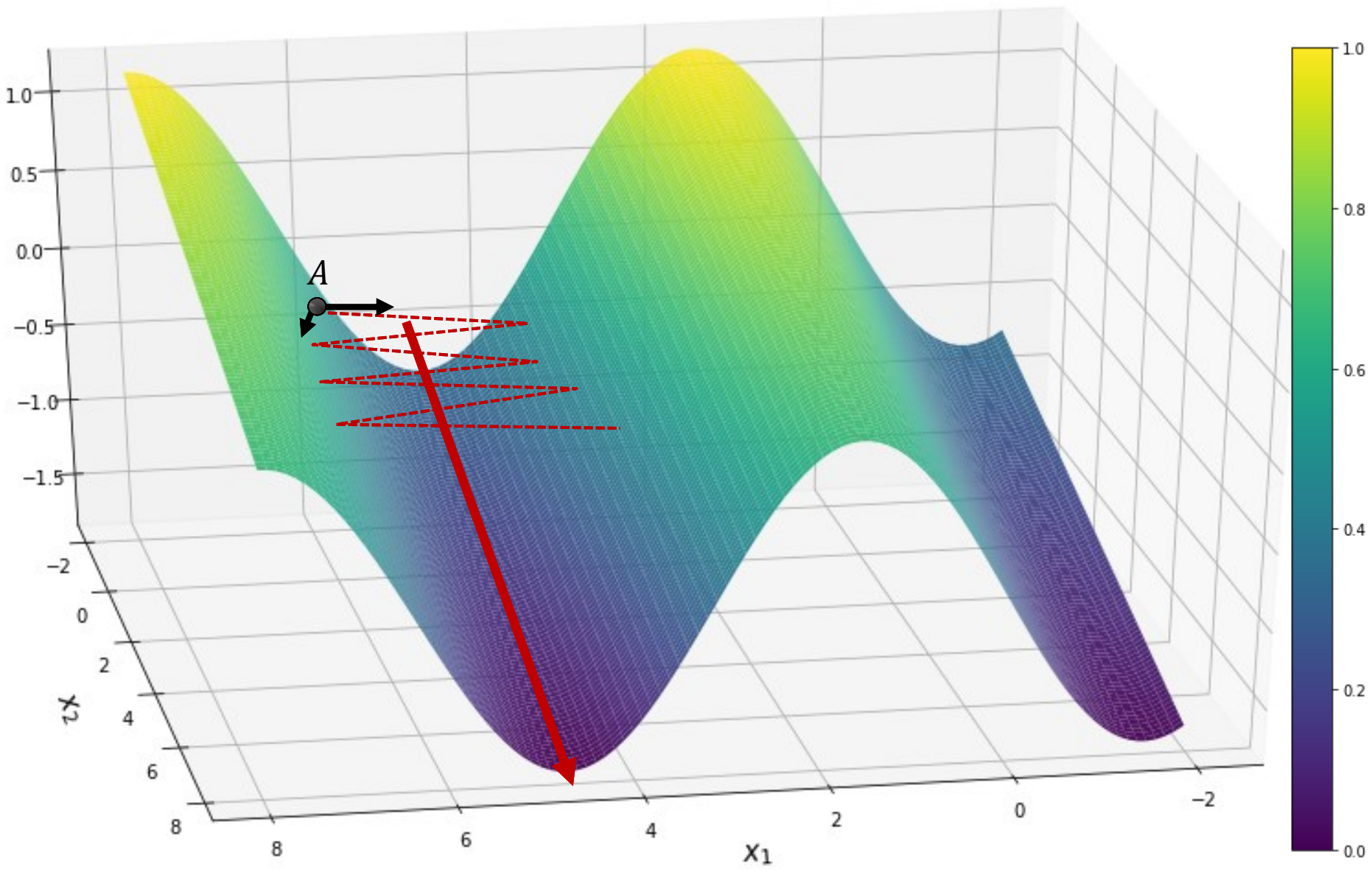


It decreases sharply and then becomes smoother...

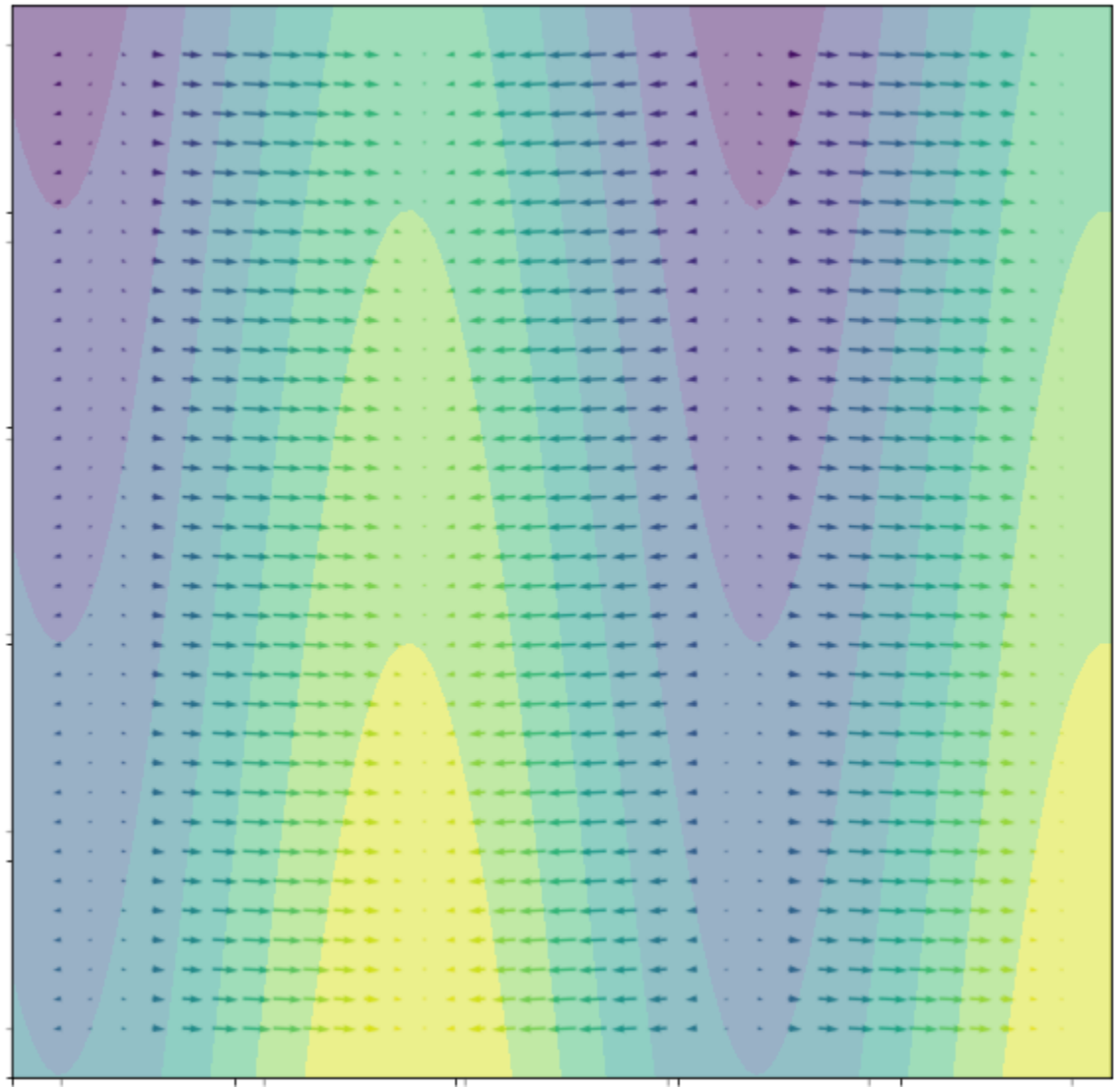


"Gradient descent is a **First Order Optimization Method**. It only takes the first order derivatives of the loss function into account and not the higher ones. What this basically means it has no clue about the curvature of the loss function. **It can tell whether the loss is declining and how fast, but cannot differentiate between whether the curve is a plane, curving upwards or curving downwards.**"

https://blog.paperspace.com/intro-to-optimization-momentum-rmsprop-adam/



One problem with the gradient descent algorithm is that searching in the steepest direction, that is, in the direction perpendicular to the contours, can lead to a zig-zag and convergence will be very slow. For example, if we start from point A, the direction perpendicular to the contour points in a direction almost 90^o degrees from the direction to the minimum point.



IS HESSIAN THE SOLUTION?

Hessian is a matrix that organizes all the second-order partial derivatives of a function.

BUT it requires you to calculate gradients of the loss function with respect to each combination of parameters ω_i, ω_j. For modern problems, the number of parameters can be in the billions, and having to calculate a billion square gradients makes using higher-order optimization methods computationally intractable.

HOWEVER, second-order optimization consists of incorporating information about how the gradient is changing. While we cannot accurately calculate this information, we can choose to follow heuristics that guide our search for optima based on past gradient behavior.

MOMENTUM

The Momentum proposes the following adjustment for the gradient descent.

$$m = \beta m - \alpha \nabla J(\omega)$$

$$\omega = \omega + m$$

m is the gradient that is maintained in previous iterations. This retained gradient is multiplied by a value called "Coefficient of Momentum" β, which is the percentage of the gradient retained at each iteration.

If we set the initial value of m to 0 and choose our coefficient as $\beta = 0.9$, the subsequent update equations will be,

$$m_1 = -G^{(1)}$$

$$m_2 = -0.9G^{(1)} - G^{(2)}$$

$$m_3 = -0.9\left(0.9G^{(1)} + G^{(2)}\right) - G_3 = -0.81G^{(1)} - 0.9G^{(2)} - G^{(3)}$$

$$G^{(i)} = \alpha \nabla J\left(\omega^{(i)}\right)$$

We give gradient descent a short-term memory!

GD vs GD COM MOMENTO

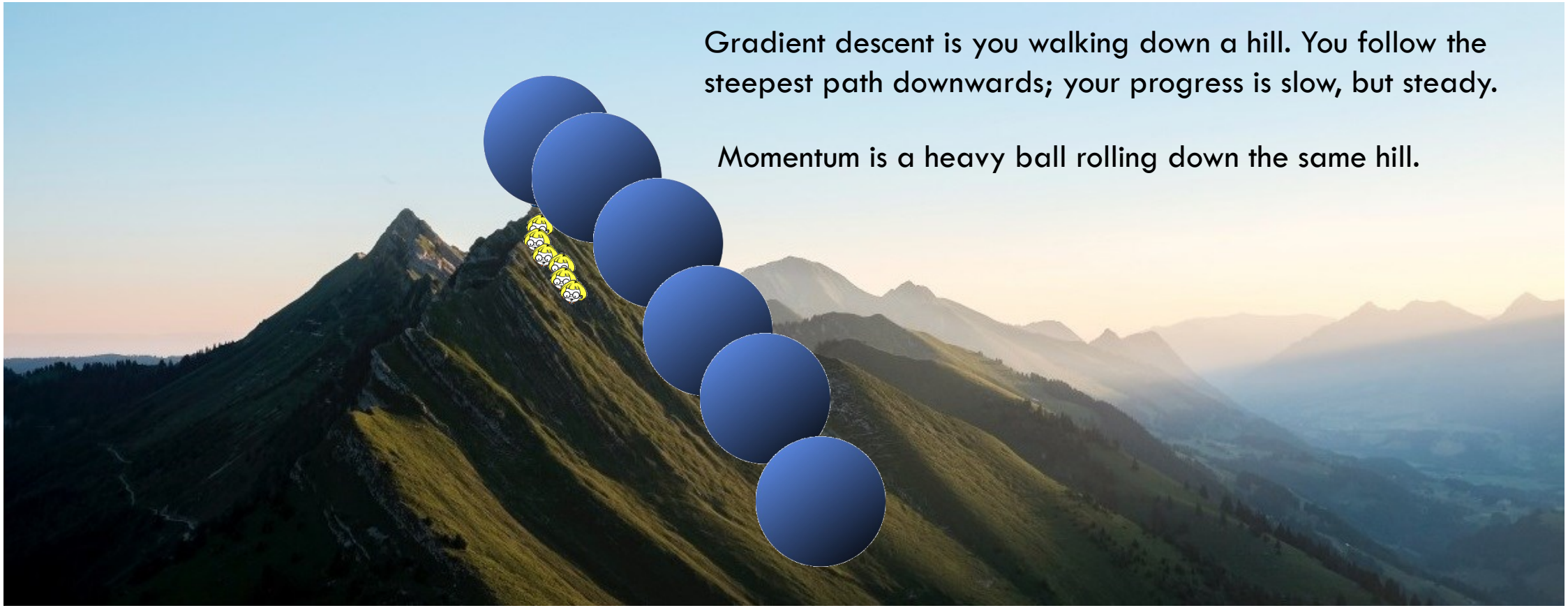
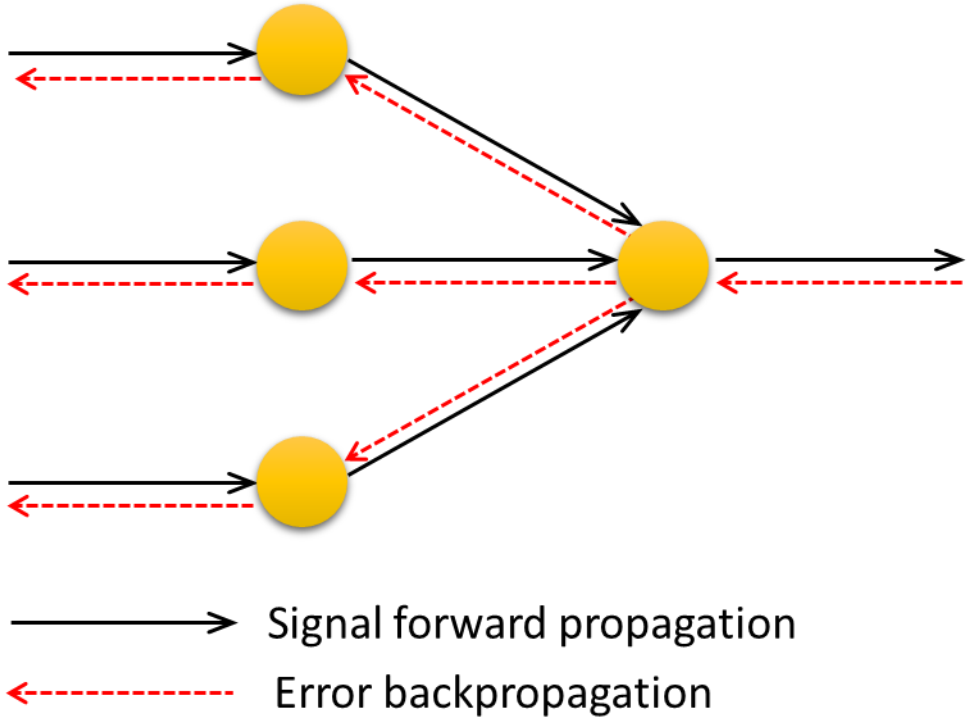


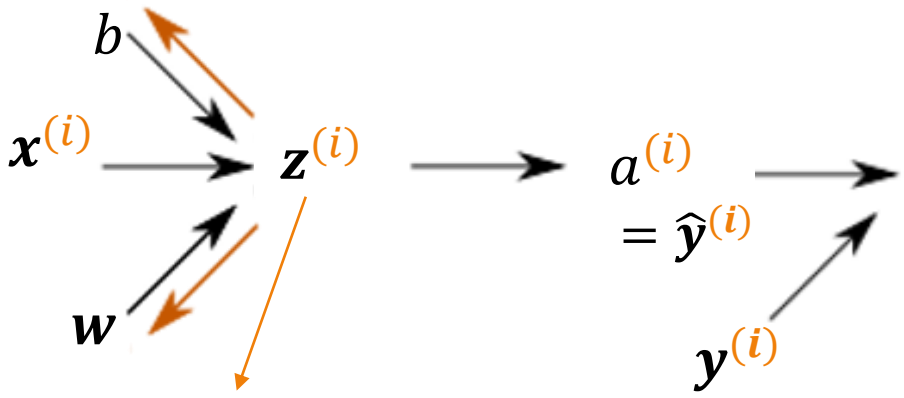
Figura extraída de:
https://www.itread01.com/content/1543467366.html

BACKPROPAGATION



WHY BACKPROPAGATION?

For each epoch k,



$$z_k^{(i)} = 1b_k^{(i)} + x_1^{(i)} w_{k,1} + x_2^{(i)} w_{k,2} + \cdots + x_{n_x}^{(i)} w_{k,n_x}$$

$$E_k{}^{(i)}$$

$$J_k += E_k^{(i)}$$

$$\Delta w_{j,k} = -\alpha \frac{\partial J_{h,k}}{\partial w_{j,k}}$$

$$\Delta b_k = -\alpha \frac{\partial J_{h,k}}{\partial b_k}$$

$$b_{k+1} = b_k + \Delta b_k$$

$$w_{j,k+1} = w_{j,k} + \Delta w_{j,k}$$

$$z_k^{(i)} = 1b_k^{(i)} + x_1^{(i)} w_{k,1} + x_2^{(i)} w_{k,2} + \cdots + x_{n_x}^{(i)} w_{k,n_x}$$

$$\Delta w_{j,k} = -\alpha \frac{\partial J_{h,k}}{\partial w_{j,k}}$$

$$\Delta b_k = -\alpha \frac{\partial J_{h,k}}{\partial b_k}$$

$$\frac{\partial J_k}{\partial w_{j,k}} = \sum_{i=1}^{m} \frac{\partial E(\hat{y}^{(i)}, y^{(i)})}{\partial w_{j,k}} = \sum_{i=1}^{m} \frac{\partial E(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)})}{\partial z_k^{(i)}} \frac{\partial z_k^{(i)}}{\partial w_{j,k}}$$

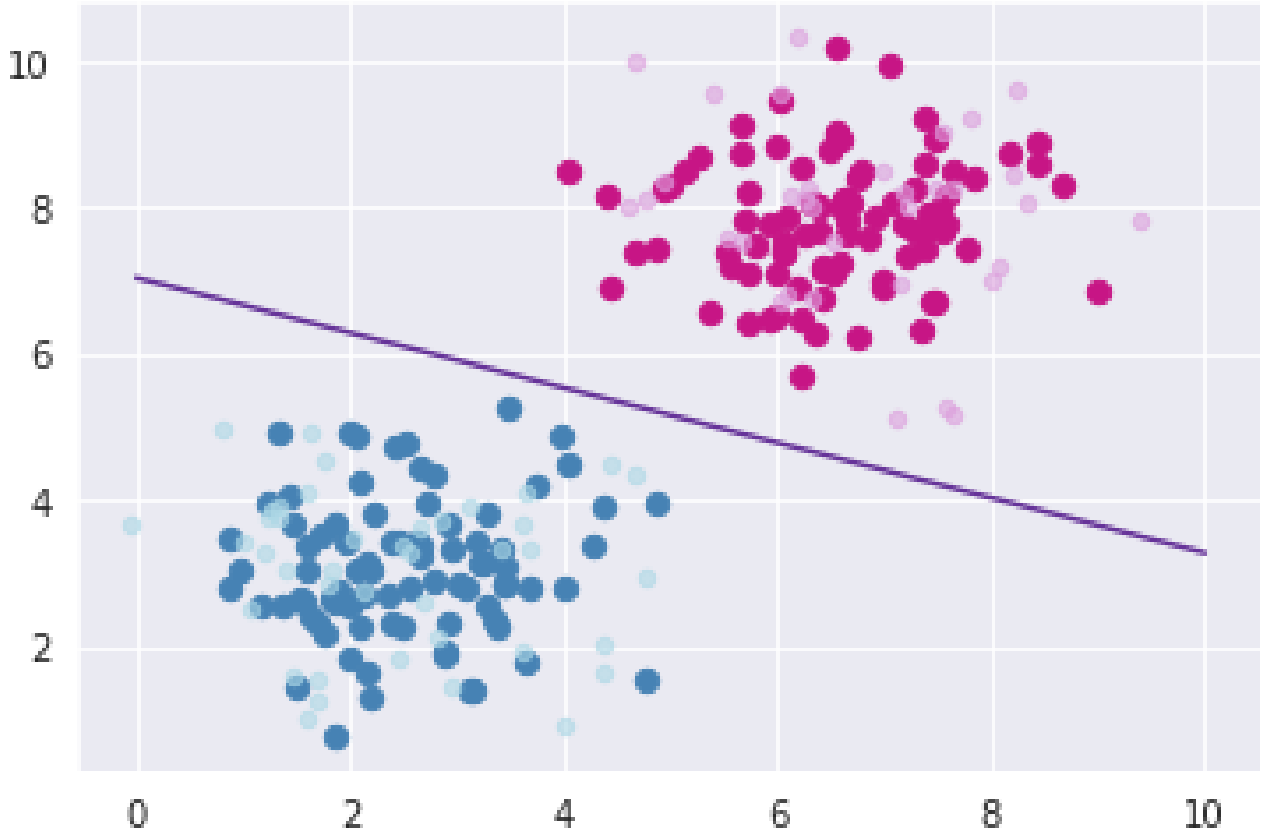
$$\frac{\partial J_k}{\partial b_k} = \sum_{i=1}^{m} \frac{\partial E(\hat{y}^{(i)}, y^{(i)})}{\partial b_k} = \sum_{i=1}^{m} \frac{\partial E(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)})}{\partial z_k^{(i)}} \frac{\partial z_k^{(i)}}{\partial b_k}$$

EXAMPLE: SYNTHETIC DATA

```
from sklearn.datasets import make_blobs

n_samples = 250
x, y = make_blobs(n_samples=n_samples,
                  centers=([2.5, 3], [6.7, 7.9]),
                  random_state=0)
colours = ('steelblue', 'mediumvioletred')
fig, ax = plt.subplots()


for n_class in range(2):
   ax.scatter(x[y==n_class][:, 0], x[y==n_class][:, 1],
           c=colours[n_class], s=10, label=str(n_class))
```

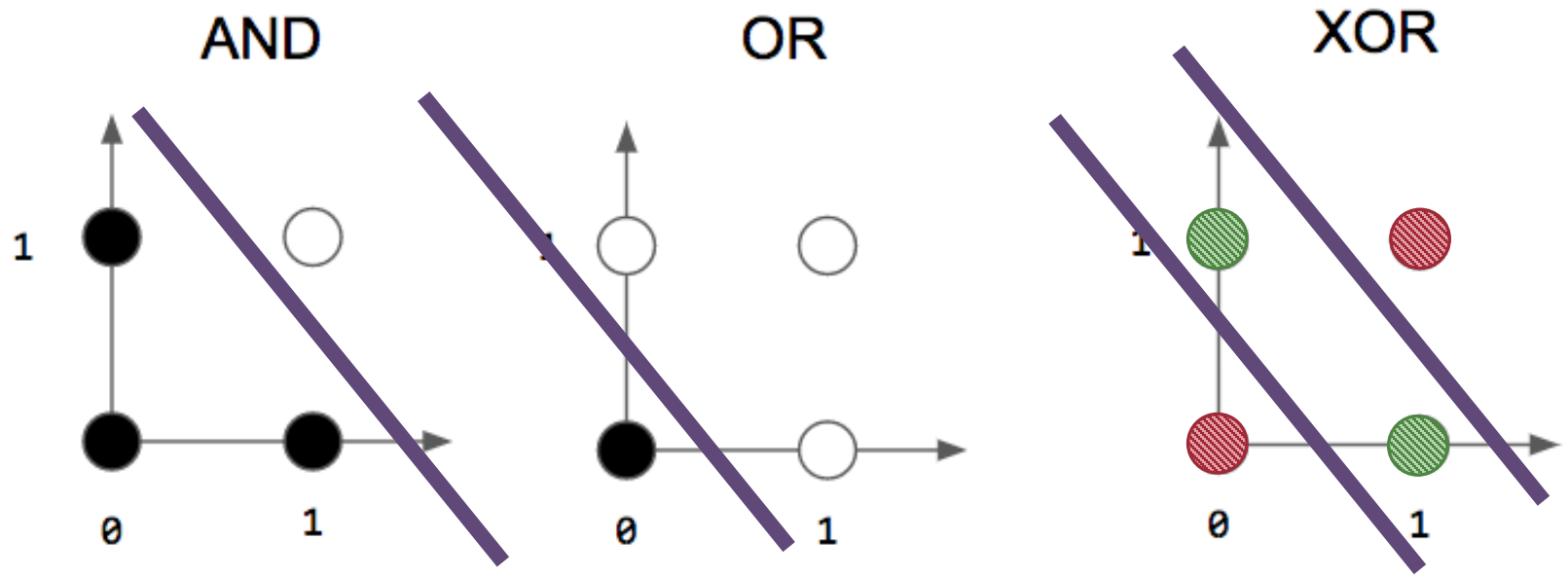

```python
#Com a biblioteca python
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Perceptron
from sklearn.metrics import accuracy_score

X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.3, random_state=0)

modelo = Perceptron(alpha=0.1, random_state=0)
modelo.fit(X_train, y_train)
y_pred = modelo.predict(X_test)
print('Total de testes:{:d}, erros:{:d}'.format(len(y_test), (y_test != y_pred).sum()))
print('Acurácia: {:3.2f}'.format(accuracy_score(y_test, y_pred)))
```

```
fig, ax = plt.subplots()

# plotting learn data
colours = ('steelblue', 'mediumvioletred')
for n_class in range(2):
    ax.scatter(X_train[y_train==n_class][:, 0],
               X_train[y_train==n_class][:, 1],
               c=colours[n_class], s=40)

# plotting test data
colours = ('lightblue', 'plum')
for n_class in range(2):
    ax.scatter(X_test[y_test==n_class][:, 0],
               X_test[y_test==n_class][:, 1],
               c=colours[n_class], s=20, alpha = 0.6)

print(modelo.coef_) # coeficientes
print(modelo.intercept_)  # intercept/w0

print(np.max(x[:]))
t = np.arange(np.max(x[:]))
m = -modelo.coef_[0][0] / modelo.coef_[0][1]
c = -modelo.intercept_ / modelo.coef_[0][1]
ax.plot(t, m*t+c, color = 'rebeccapurple' )
```



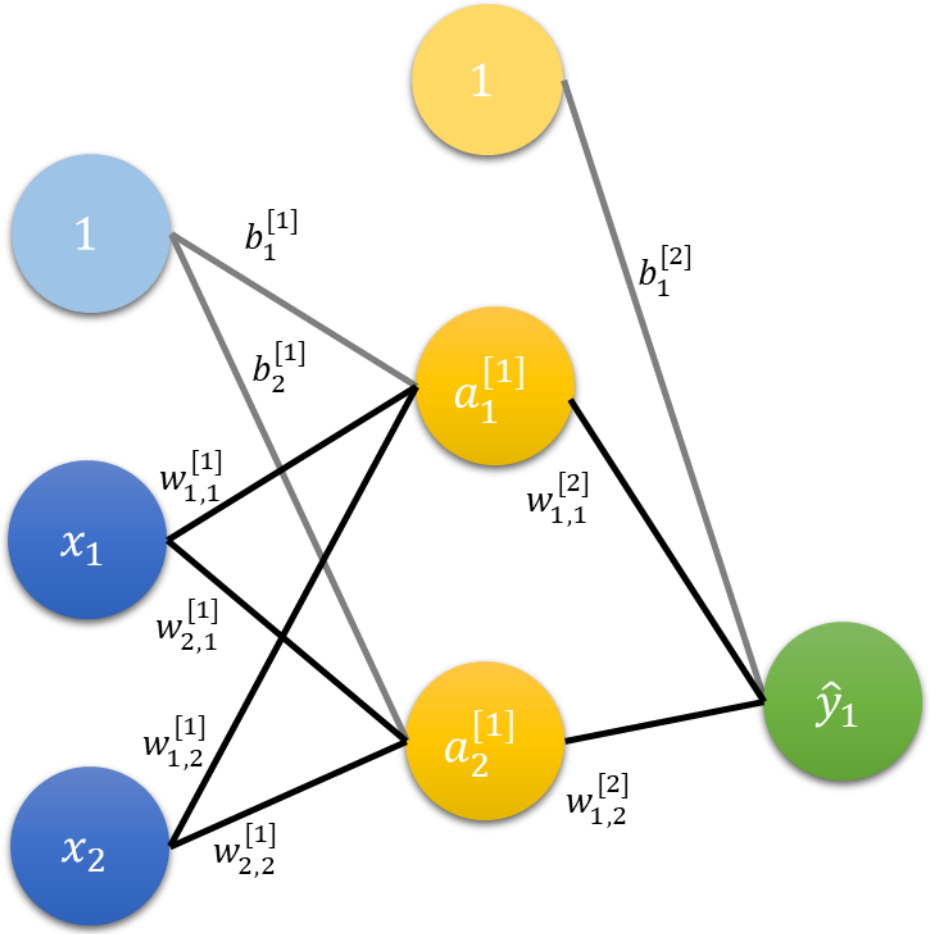
XOR PROBLEM

It is clear that a single perceptron will not serve our purpose: the classes aren't **linearly separable.**



MULTI-LAYERED PERCEPTRON (MLP)

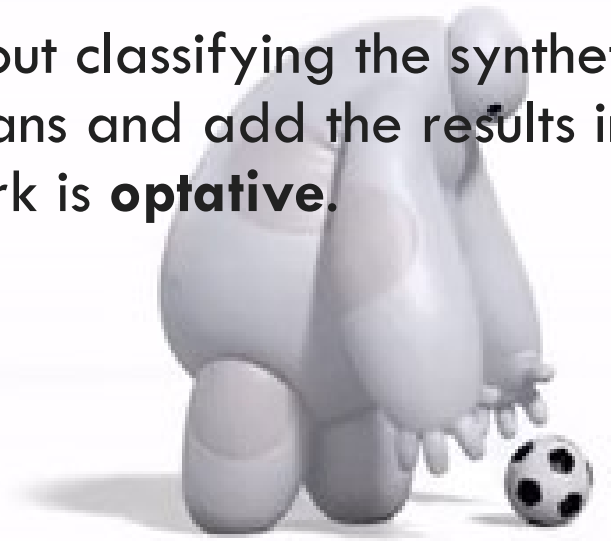
An MLP is generally restricted to having a single intermediary layer.



HOMEWORK – PART I

Solve the XOR Problem defined in the previous slide using a MLP.

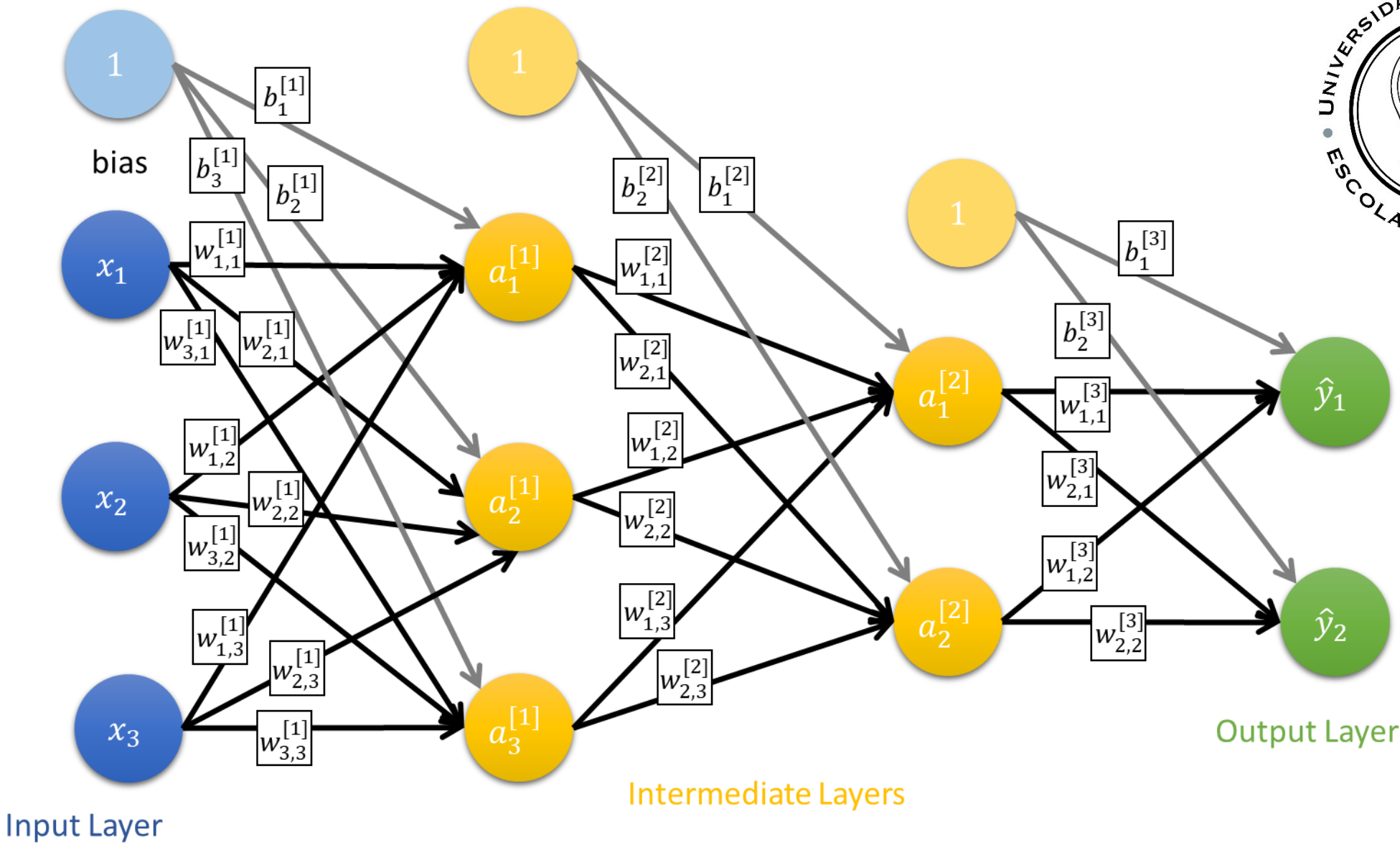
How about classifying the synthetic data using SVM (Support Vector Machine) or K-means and add the results in the graph we did? This part of the homework is **optative**.

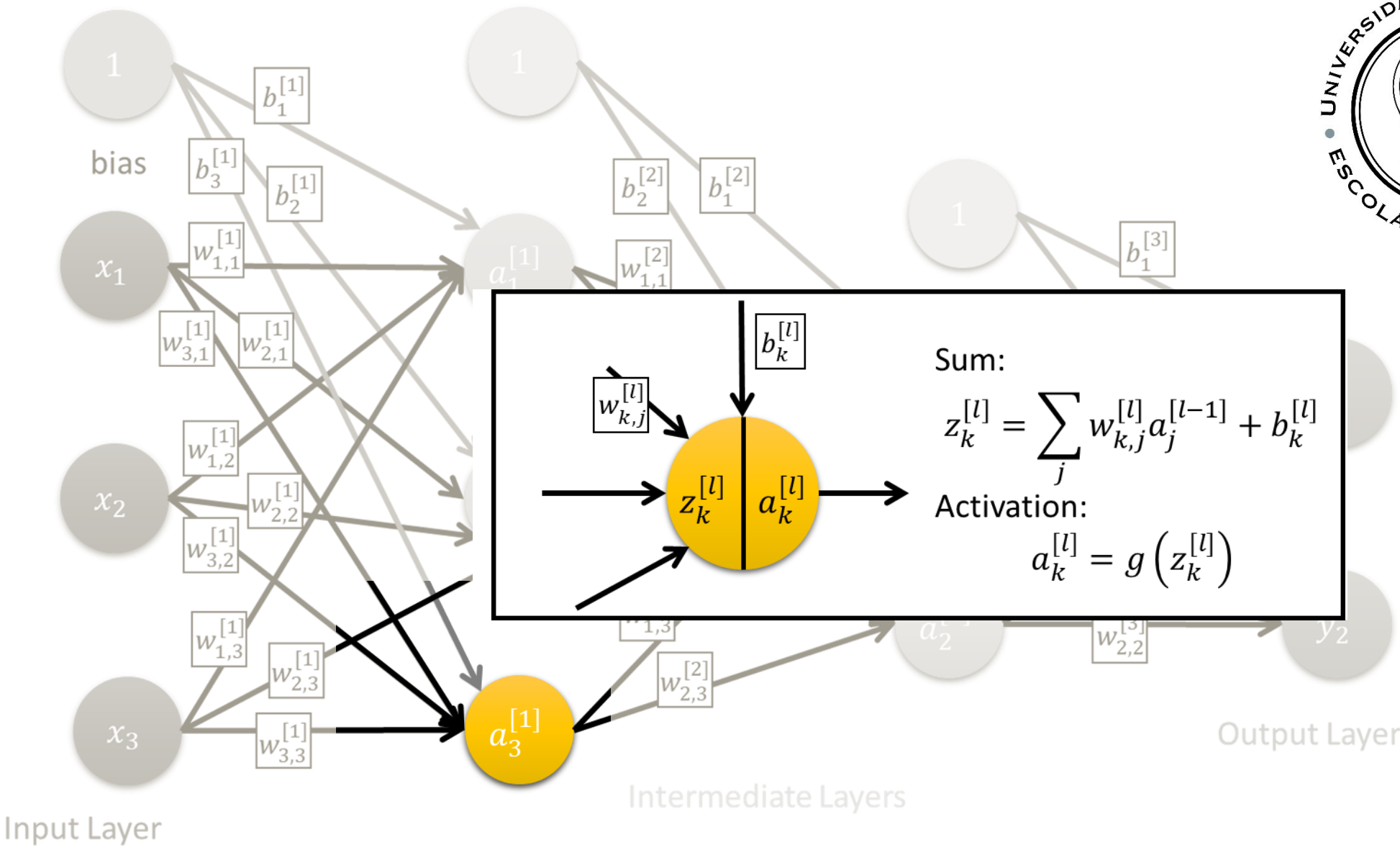


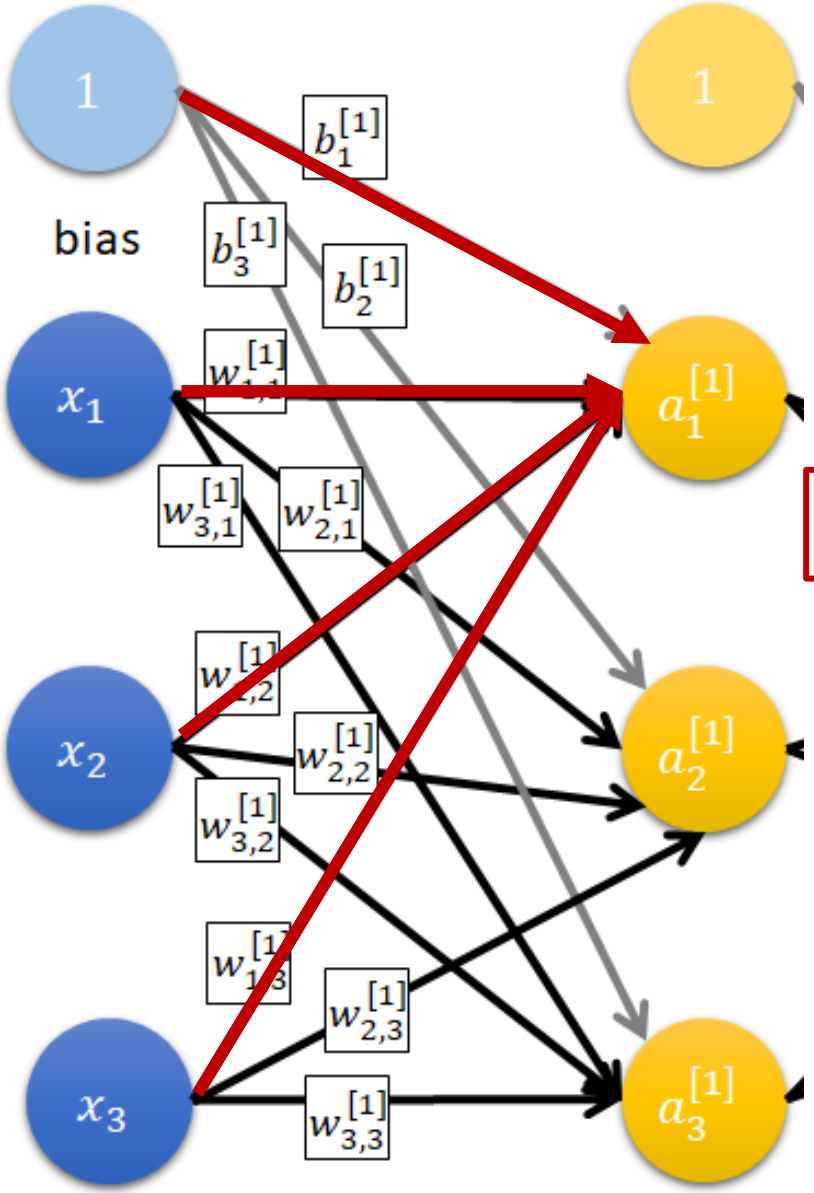
Although our classes have an end, theory doesn't!!!!

DEEP NEURAL NETWORKS

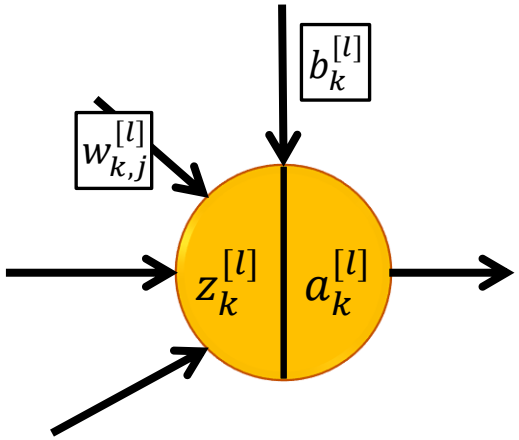
What will change if we have multiple layers in between?


1
bias
x_1
x_2
x_3
$b_1^{[1]}$
$b_3^{[1]}$
$b_2^{[1]}$
$w_{1,1}^{[1]}$
$w_{3,1}^{[1]}$
$w_{2,1}^{[1]}$
$w_{1,2}^{[1]}$
$w_{2,2}^{[1]}$
$w_{3,2}^{[1]}$
$w_{1,3}^{[1]}$
$w_{2,3}^{[1]}$
$w_{3,3}^{[1]}$
1
$a_1^{[1]}$
$a_2^{[1]}$
$a_3^{[1]}$
$b_2^{[2]}$
$b_1^{[2]}$
$w_{1,1}^{[2]}$
$w_{2,1}^{[2]}$
$w_{1,2}^{[2]}$
$w_{2,2}^{[2]}$
$w_{1,3}^{[2]}$
$w_{2,3}^{[2]}$
1
$a_1^{[2]}$
$a_2^{[2]}$
$b_1^{[3]}$
$b_2^{[3]}$
$w_{1,1}^{[3]}$
$w_{2,1}^{[3]}$
$w_{1,2}^{[3]}$
$w_{2,2}^{[3]}$
$\hat{y}_1$
$\hat{y}_2$
Input Layer
Intermediate Layers
Output Layer


1
bias
x_1
x_2
x_3
$b_1^{[1]}$
$b_3^{[1]}$
$b_2^{[1]}$
$w_{1,1}^{[1]}$
$w_{3,1}^{[1]}$
$w_{2,1}^{[1]}$
$w_{1,2}^{[1]}$
$w_{2,2}^{[1]}$
$w_{3,2}^{[1]}$
$w_{1,3}^{[1]}$
$w_{2,3}^{[1]}$
$w_{3,3}^{[1]}$
$a_3^{[1]}$
$b_2^{[2]}$
$b_1^{[2]}$
$w_{1,1}^{[2]}$
$w_{2,3}^{[2]}$
$b_1^{[3]}$
$w_{2,2}^{[3]}$
y_2
$b_k^{[l]}$
$w_{k,j}^{[l]}$
$z_k^{[l]}$
$a_k^{[l]}$
Sum:
$z_k^{[l]} = \sum_j w_{k,j}^{[l]} a_j^{[l-1]} + b_k^{[l]}$
Activation:
$a_k^{[l]} = g\left(z_k^{[l]}\right)$
Input Layer
Intermediate Layers
Output Layer



Camada de entrada



Somatório:

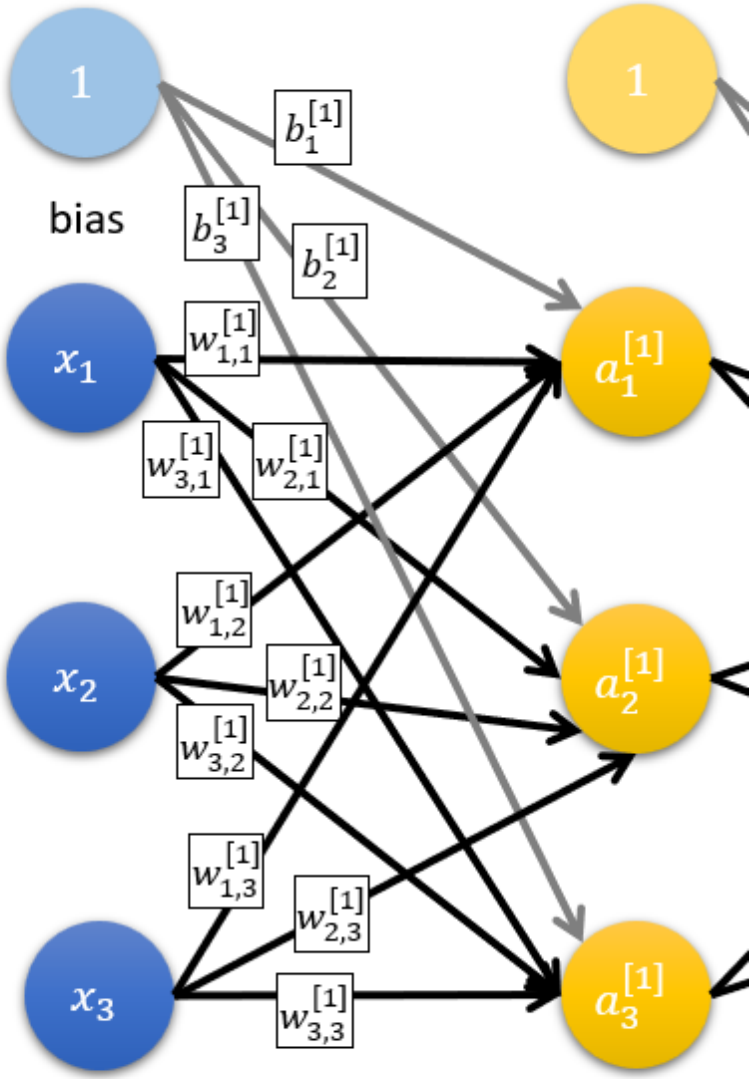
$$z_k^{[l]} = \sum_j w_{k,j}^{[l]} a_j^{[l-1]} + b_k^{[l]}$$

Ativação:

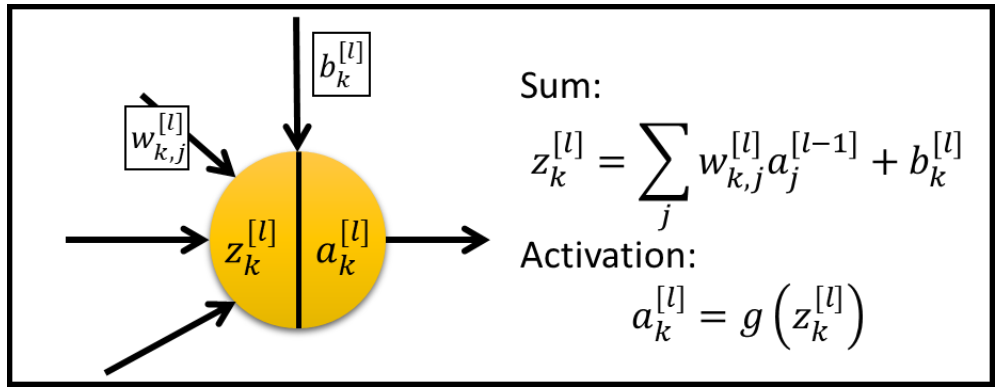
$$a_k^{[l]} = g\left(z_k^{[l]}\right)$$

$$\begin{bmatrix} z_1^{[1](i)} \\ z_2^{[1](i)} \\ z_3^{[1](i)} \end{bmatrix} = \begin{bmatrix} w_{1,1}^{[1]} & w_{1,2}^{[1]} & w_{1,3}^{[1]} \\ w_{2,1}^{[1]} & w_{2,2}^{[1]} & w_{2,3}^{[1]} \\ w_{3,1}^{[1]} & w_{3,2}^{[1]} & w_{3,3}^{[1]} \end{bmatrix} \begin{bmatrix} x_1^{(i)} \\ x_2^{(i)} \\ x_3^{(i)} \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \end{bmatrix}$$

$$\begin{bmatrix} a_1^{[1](i)} \\ a_2^{[1](i)} \\ a_3^{[1](i)} \end{bmatrix} = g^{[1]} \begin{bmatrix} z_1^{[1](i)} \\ z_2^{[1](i)} \\ z_3^{[1](i)} \end{bmatrix}$$



Input Layer



Sum:

$$z_k^{[l]} = \sum_j w_{k,j}^{[l]} a_j^{[l-1]} + b_k^{[l]}$$

Activation:

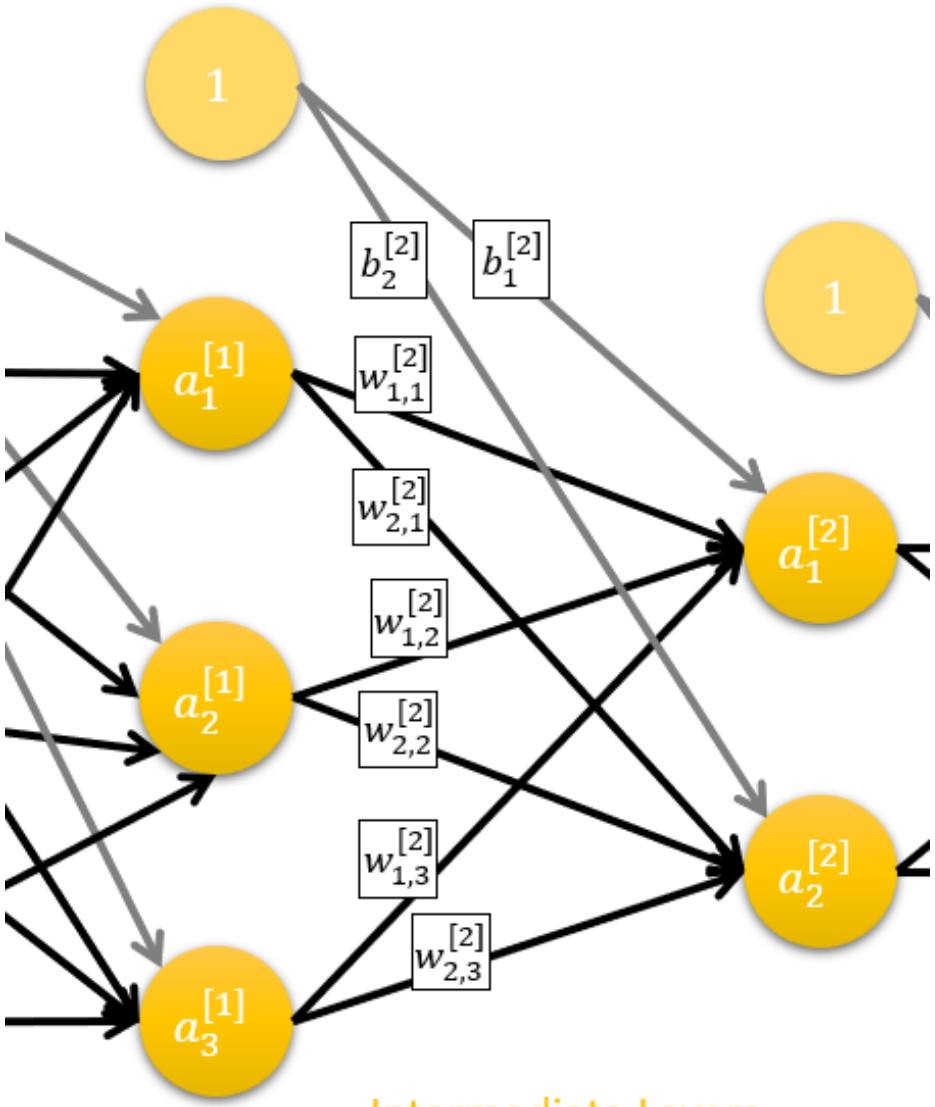
$$a_k^{[l]} = g\left(z_k^{[l]}\right)$$

$$\mathbf{z}^{[1](i)} = \mathbf{W}^{[1]}\mathbf{x}^{(i)} + \mathbf{b}^{[1]}$$

$$\begin{bmatrix} z_1^{[1](i)} \\ z_2^{[1](i)} \\ z_3^{[1](i)} \end{bmatrix} = \begin{bmatrix} w_{1,1}^{[1]} & w_{1,2}^{[1]} & w_{1,3}^{[1]} \\ w_{2,1}^{[1]} & w_{2,2}^{[1]} & w_{2,3}^{[1]} \\ w_{3,1}^{[1]} & w_{3,2}^{[1]} & w_{3,3}^{[1]} \end{bmatrix} \begin{bmatrix} x_1^{(i)} \\ x_2^{(i)} \\ x_3^{(i)} \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \end{bmatrix}$$

$$\mathbf{a}^{[1](i)} = g^{[1]}\left(\mathbf{z}^{[1](i)}\right)$$

$$\begin{bmatrix} a_1^{[1](i)} \\ a_2^{[1](i)} \\ a_3^{[1](i)} \end{bmatrix} = g^{[1]} \begin{bmatrix} z_1^{[1](i)} \\ z_2^{[1](i)} \\ z_3^{[1](i)} \end{bmatrix}$$

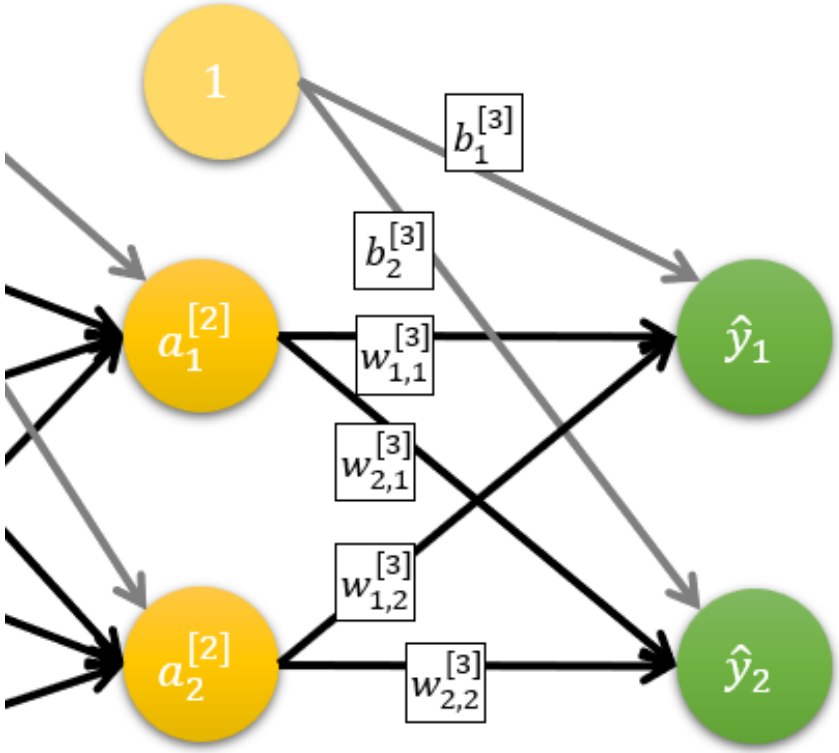


$$\begin{bmatrix} z_1^{[2](i)} \\ z_2^{[2](i)} \end{bmatrix} = \begin{bmatrix} w_{1,1}^{[2]} & w_{1,2}^{[2]} & w_{1,3}^{[2]} \\ w_{2,1}^{[2]} & w_{2,2}^{[2]} & w_{2,3}^{[2]} \end{bmatrix} \begin{bmatrix} a_1^{[1](i)} \\ a_2^{[1](i)} \\ a_3^{[1](i)} \end{bmatrix} + \begin{bmatrix} b_1^{[2]} \\ b_2^{[2]} \end{bmatrix}$$

$$\mathbf{z}^{[2](i)} = \mathbf{W}^{[2]}\mathbf{a}^{[1](i)} + \mathbf{b}^{[2]}$$

$$\mathbf{z}^{[l](i)} = \mathbf{W}^{[l]}\mathbf{a}^{[l-1](i)} + \mathbf{b}^{[l]}$$

$$\begin{bmatrix} a_1^{[2](i)} \\ a_2^{[2](i)} \end{bmatrix} = \sigma^{[2]} \begin{bmatrix} z_1^{[2](i)} \\ z_2^{[2](i)} \end{bmatrix} \qquad \mathbf{a}^{[l](i)} = g^{[l]}\left(\mathbf{z}^{[l](i)}\right)$$

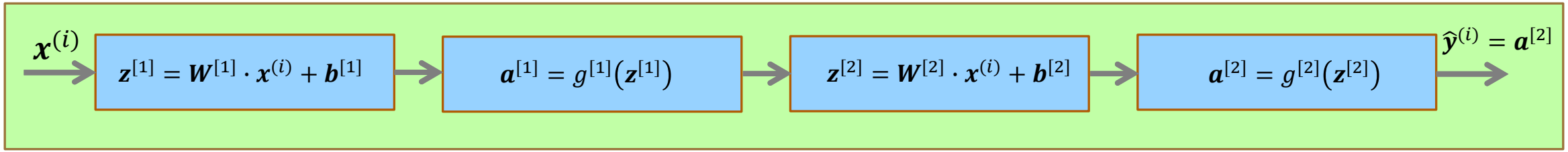


Output Layer

$$\begin{bmatrix} \hat{y}_1^{(i)} \\ \hat{y}_2^{(i)} \end{bmatrix} = \begin{bmatrix} a_1^{[3](i)} \\ a_2^{[3](i)} \end{bmatrix}$$

$$\hat{\mathbf{y}}^{(i)} = \mathbf{a}^{[L](i)}$$

ANN DATA FLOW DIAGRAM

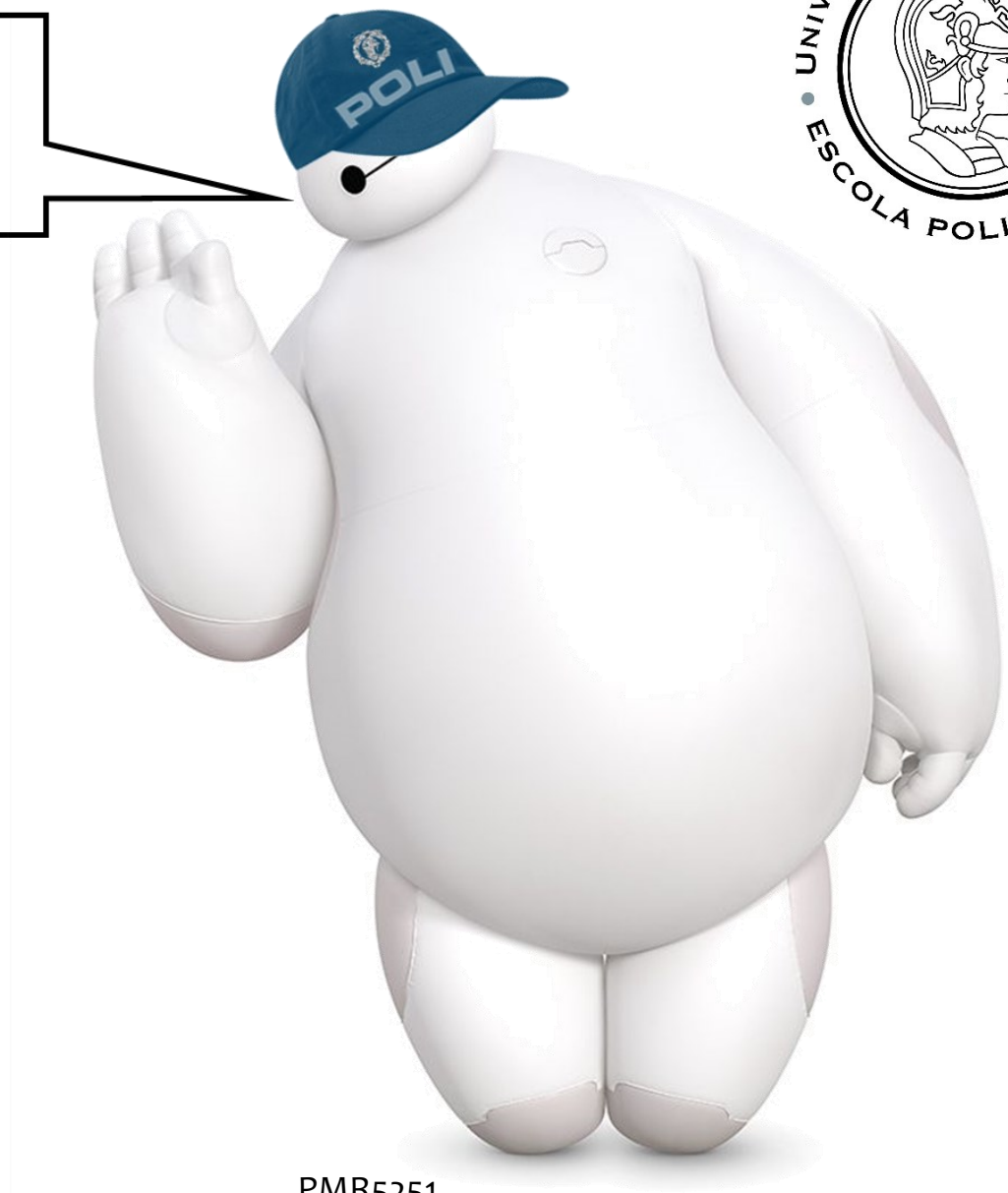


Each block represents a matrix equation, and for each layer of the ANN there are two equations: one to calculate the states $z^{[i]}$ of the neurons of the layer and another to calculate the activations $a^{[i]}$ of the neurons in the layer.

The output of one layer is the input of the next layer.

THE QUESTION IS:

How to find the weights $w_{k,j}^{[l]}$ and bias $b_k^{[l]}$?



ALGORITMO DE GRADIENTE DESCENDENTE

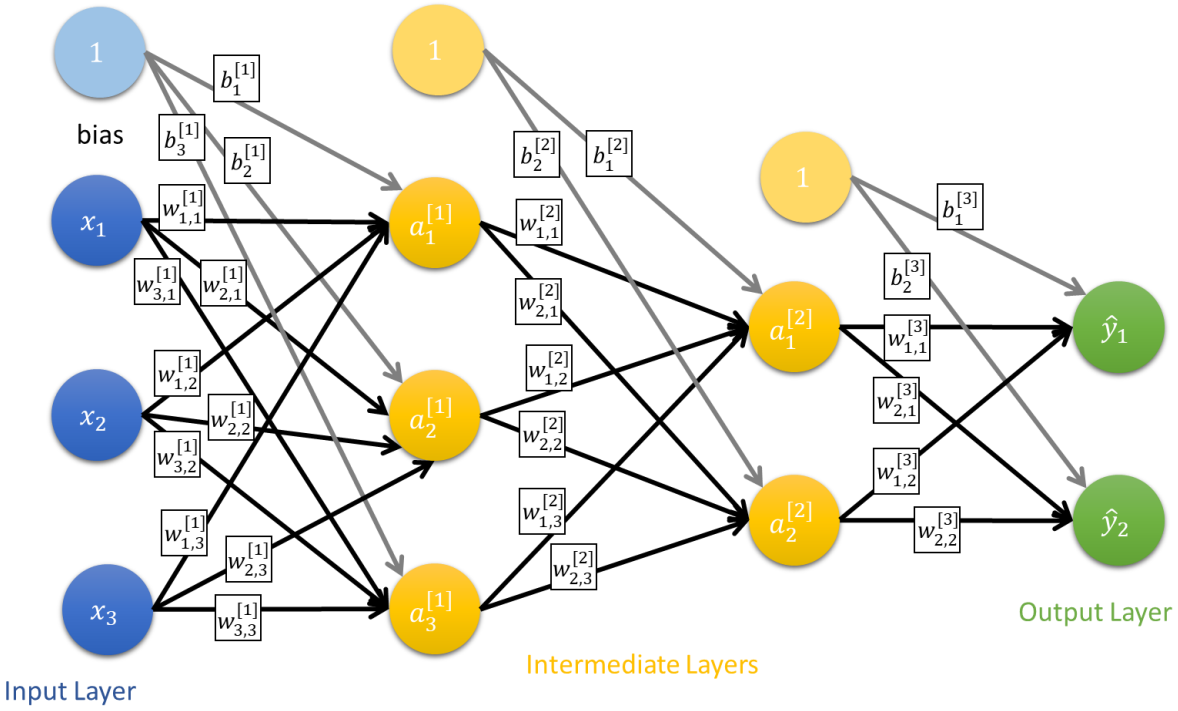
1. ANN parameter initialization: Assign initial value to $\boldsymbol{W}, \boldsymbol{b}$;
2. Execution of the ANN **for all examples of the training data set,** so that given the inputs, the outputs predicted by the ANN are calculated;
3. Calculate the cost function for all training examples, through the sum of the error function;
4. Calculate the cost function gradient in relation to all ANN parameters;
5. Update of the ANN parameters in the opposite direction of the gradient in order to reduce the value of the cost function

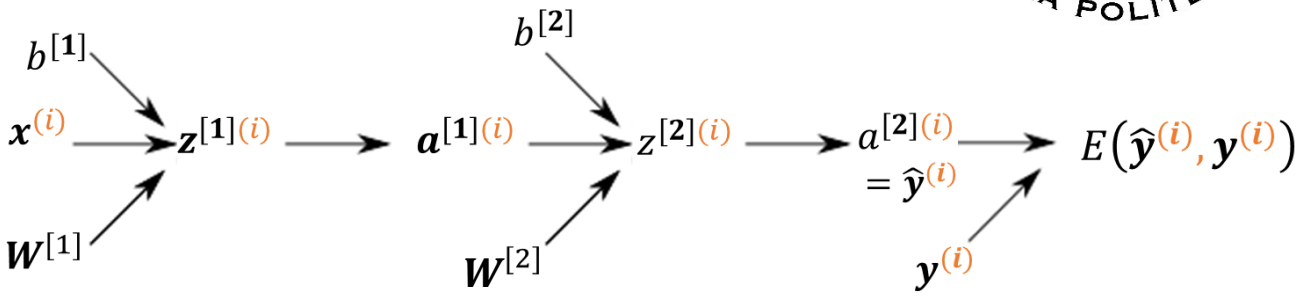
$$w_{k,j}^{[l]} = w_{k,j}^{[l]} - \alpha \frac{\partial J(\mathbf{W}, \mathbf{b})}{\partial w_{k,j}^{[l]}}$$

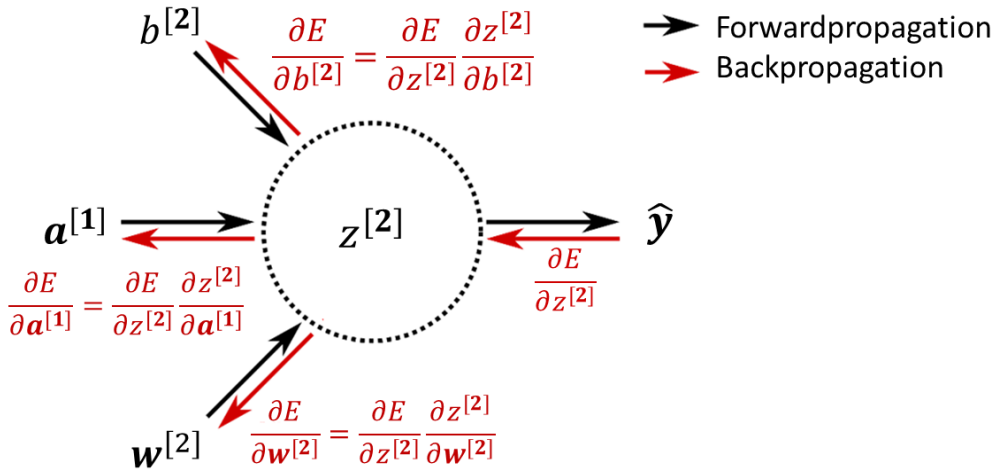
$$b_{k}^{[l]} = b_{k}^{[l]} - \alpha \frac{\partial J(\mathbf{W}, \mathbf{b})}{\partial b_{k}^{[l]}}$$

Repeat steps 2 to 5 until the cost function value is low enough, or up to the limit number of epochs.

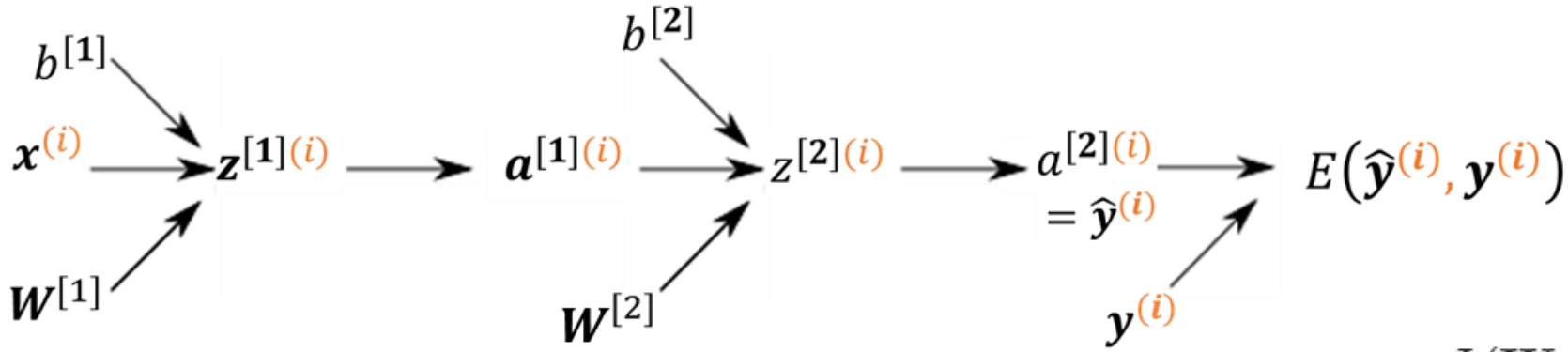
ANN IN ONE SLIDE







WHY *BACKPROPAGATION*?



$$\frac{\partial E\left(\hat{y}^{(i)}, y^{(i)}\right)}{\partial w_{k,j}^{[l]}} = \frac{\partial E\left(\hat{y}^{(i)}, y^{(i)}\right)}{\partial a_k^{[l](i)}} \frac{\partial a_k^{[l](i)}}{\partial z_k^{[l](i)}} \frac{\partial z_k^{[l](i)}}{\partial w_{k,j}^{[l]}}$$

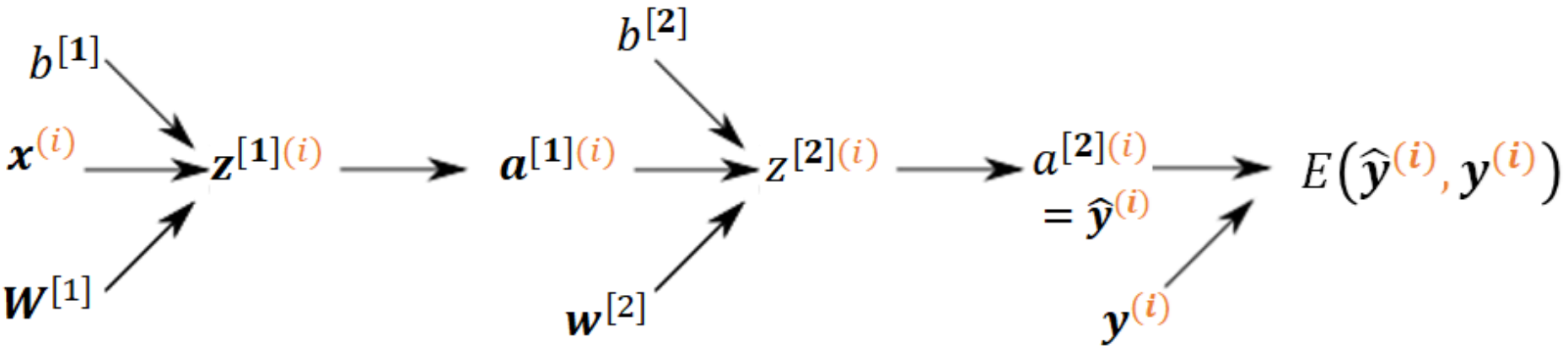
$$\frac{\partial E\left(\hat{y}^{(i)}, y^{(i)}\right)}{\partial b_k^{[l]}} = \frac{\partial E\left(\hat{y}^{(i)}, y^{(i)}\right)}{\partial a_k^{[l](i)}} \frac{\partial a_k^{[l](i)}}{\partial z_k^{[l](i)}} \frac{\partial z_k^{[l](i)}}{\partial b_k^{[l]}}$$

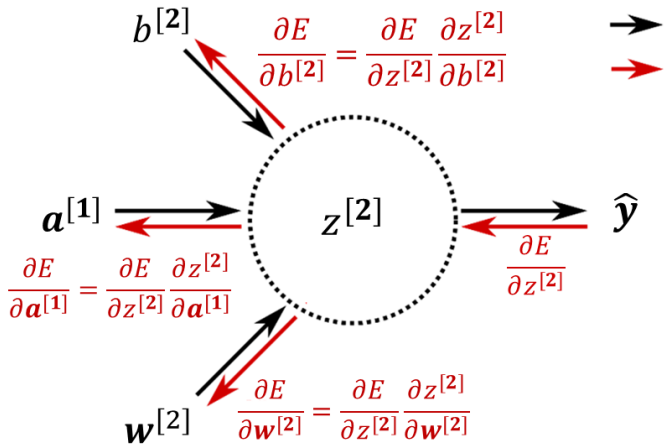
$$J(\mathbf{W}, \mathbf{b}) = \frac{1}{m} \sum_{i=1}^{m} E\left(\hat{y}^{(i)}, y^{(i)}\right)$$

$$w_{k,j}^{[l]} = w_{k,j}^{[l]} - \alpha \frac{\partial J(\mathbf{W}, \mathbf{b})}{\partial w_{k,j}^{[l]}}$$

$$b_k^{[l]} = b_k^{[l]} - \alpha \frac{\partial J(\mathbf{W}, \mathbf{b})}{\partial b_k^{[l]}}$$

FOR THE LAST LAYER...





$$\frac{\partial E}{\partial w_{k,j}^{[2]}} = \frac{\partial E}{\partial \hat{y}^{(i)}} \frac{\partial \hat{y}^{(i)}}{\partial w_{k,j}^{[2]}} = \frac{\partial E}{\partial \hat{y}^{(i)}} \frac{\partial \hat{y}^{(i)}}{\partial z^{[2](i)}} \frac{\partial z^{[2](i)}}{\partial w_{k,j}^{[2]}}$$

$$\frac{\partial E}{\partial w_{k,j}^{[1]}} = \boxed{\frac{\partial E}{\partial a^{[2](i)}} \frac{\partial a^{[2](i)}}{\partial z^{[2](i)}}} \frac{\partial z^{[2](i)}}{\partial a_k^{[1](i)}} \frac{\partial a_k^{[1](i)}}{\partial z_k^{[1](i)}} \frac{\partial z_k^{[1](i)}}{\partial w_{k,j}^{[1]}}$$

You can reuse some of the calculations performed during the gradient evaluation.

Obviously, everything applies in the solution to the gradient with respect to b

$$\frac{\partial E}{\partial b_k^{[1]}} = \frac{\partial E}{\partial a^{[2](i)}} \frac{\partial a^{[2](i)}}{\partial z^{[2](i)}} \frac{\partial z^{[2](i)}}{\partial a_k^{[1](i)}} \frac{\partial a_k^{[1](i)}}{\partial z_k^{[1](i)}} \frac{\partial z_k^{[1](i)}}{\partial b_k^{[1]}}$$

$$\frac{\partial E}{\partial w_{k,j}^{[1]}} = \boxed{\frac{\partial E}{\partial a^{[2](i)}} \frac{\partial a^{[2](i)}}{\partial z^{[2](i)}} \frac{\partial z^{[2](i)}}{\partial a_k^{[1](i)}}} \frac{\partial a_k^{[1](i)}}{\partial z_k^{[1](i)}} \frac{\partial z_k^{[1](i)}}{\partial w_{k,j}^{[1]}}$$

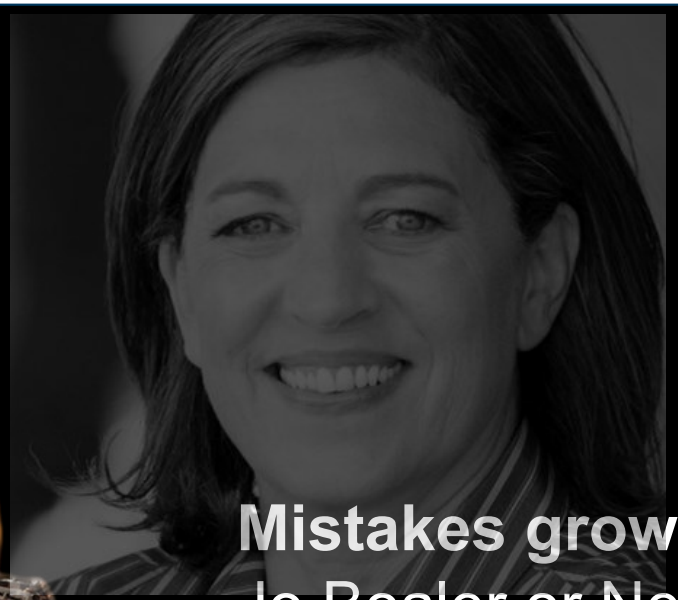
The first four terms on the right side of the equation are equal

Homework

Review the Notebook and do the proposed activities.

Upload the complete Notebook until 23/07, 23:59.

THE END



Mistakes grow your brain.
Jo Boaler or Neural Networks?