Deep generative neural networks Fundamentals & problem solving

JAMAL TOUTOUH
jamal@uma.es
jamal.es
@jamtou

Sergio Nesmachnow
sergion@fing.edu.uy

Jamal Toutouh, Ph.D.

Researcher Assistant Professor at the **University of Málaga** (Spain)

Affiliate Researcher at **Massachusetts Institute of Technology (MIT)**

- MIT Computer Science & Artificial Intelligence Lab

- PhD in Computer Science, **University of Malaga**
- M.Sc. in Software Engineering and Artificial Intelligence, **University of Malaga**
- M.Sc. in Information and Computer Sciences **University of Luxembourg**

jamal@uma.es

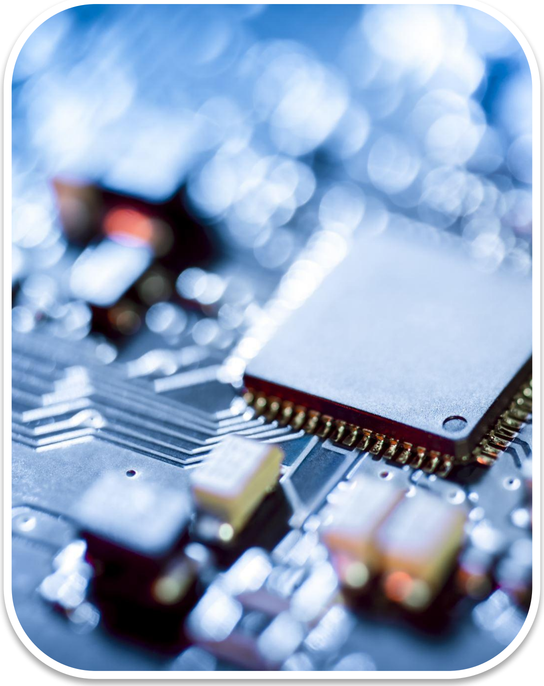
www.jamal.es

@jamtou

Intended Learning Outcomes

Attendees will, at the end of the course, be able to:

- describe the main principles of Artificial Neural Networks and Generative Adversarial Networks and their design
- identify problems that can be addressed by using Artificial Neural Networks
- use Python code to create and use Artificial Neural Networks to address classification and prediction problems
- identify problems that can be solved using Generative Machine Learning
- use Python code to create and use Generative Adversarial Networks to generate synthetic data



Generative Adversarial Networks for fun

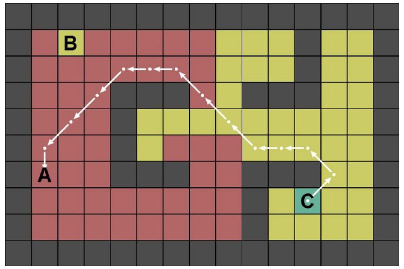
https://thispersondoesnotexist.com/

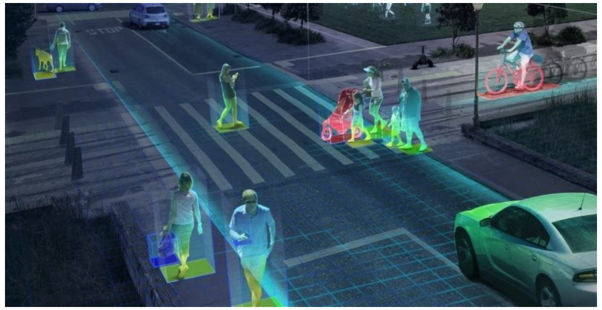
Main Concepts

Artificial Intelligence

- **Artificial Intelligence:** intelligence exhibited by machines and software
- Goal: automate “intellectual tasks” performed by humans

- AI models can be simple or complex
  - Simple: search, pathfinding, simple game playing, etc.
  - Complex: computer vision, control tasks, speech recognition, etc.



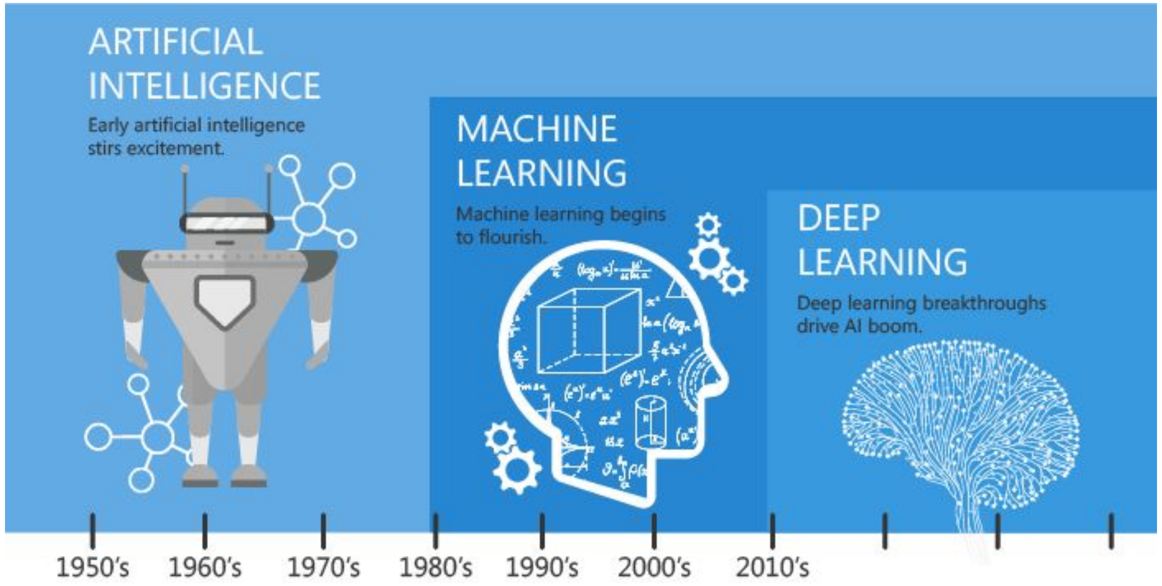


Artificial intelligence

- The main idea is that AI imitates the human cognition process (perception, learning, pattern recognition, etc.)
- Key aspects: reasoning, problem solving, learning, knowledge representation
- Many types of algorithms: search, optimization, logic programming, or **machine learning algorithms**.

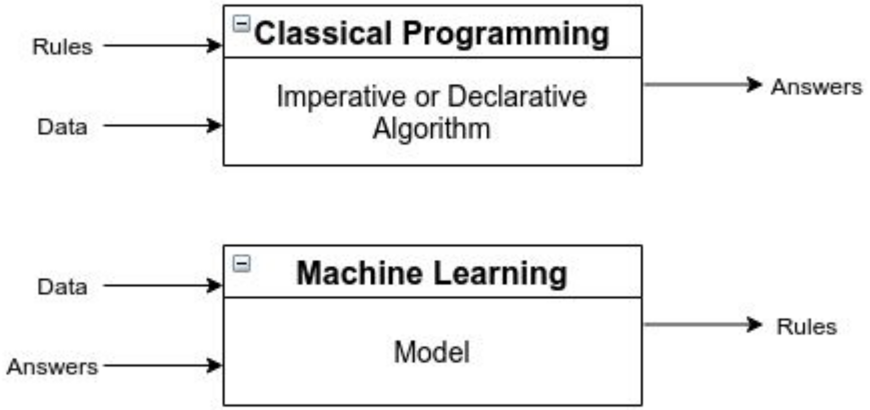
Machine Learning, Deep Learning, AI ….

- **Artificial Intelligence** is human-like “intelligence” exhibited by computers
- **Machine Learning** is the field of study that gives the computers the ability to learn without being explicitly programmed
- **Deep Learning** uses deep neural networks to implement machine learning



Machine Learning

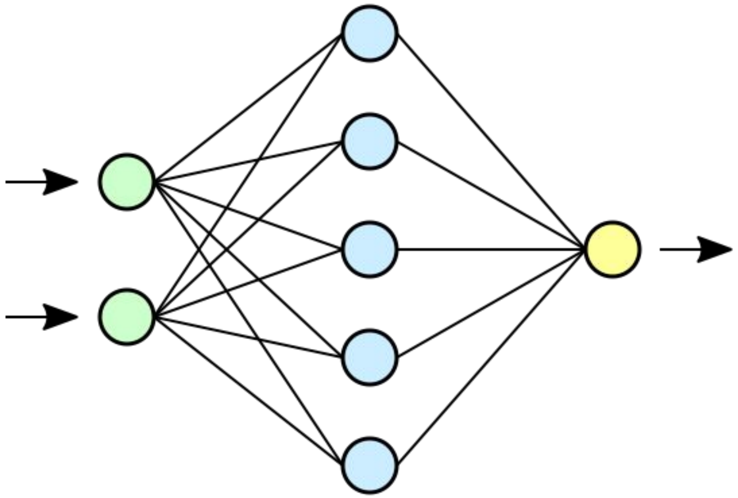
- Instead of being explicitly programmed (i.e. with a set of rules), machine learning algorithms try to infer the rules using a model.



Probabilistic (i.e., not deterministic) outputs.
Characterized by an accuracy rate.

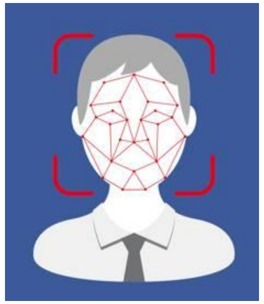
Neural networks

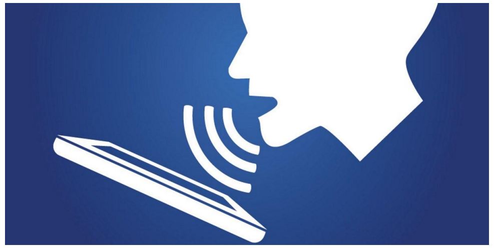
- A type of machine learning algorithms specialized on handling layered representations of data.
- Multi stage information extraction process: allows modeling complex (non-linear) functions.



Artificial Intelligence Applications

- Facial recognition
- Game playing
- Speech recognition
- Language translation
- Self-driving cars
- Image translation: edges to photo
- Fake images
- Fake videos





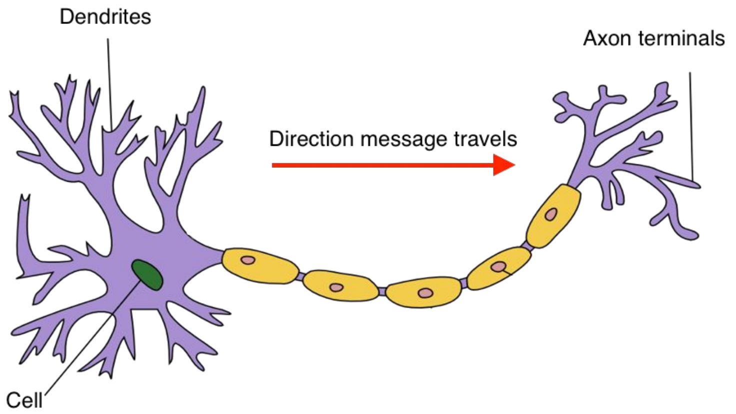

Artificial Neural Networks

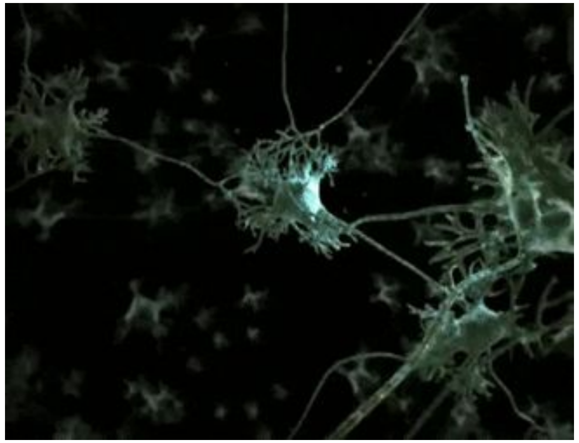
Simple Biological Neuron

The neuron is the fundamental cell responsible for **processing and transmitting information** throughout the nervous system.

A simple biological network has three major parts:

- **Dendrites:** They branch out into a tree around the cell body. They get incoming signals to cell body with their strength as weights.
- **Cell:** Collects input through dendrites and processes to produce output.
- **Axon:** Responsible for transmitting signals to other neurons.





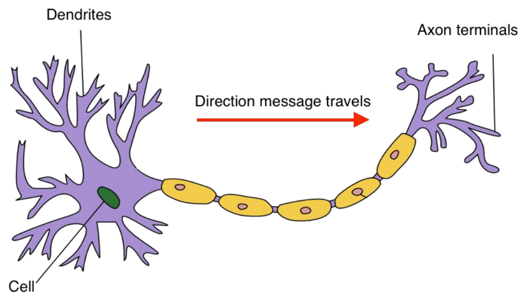
Collective Intelligence

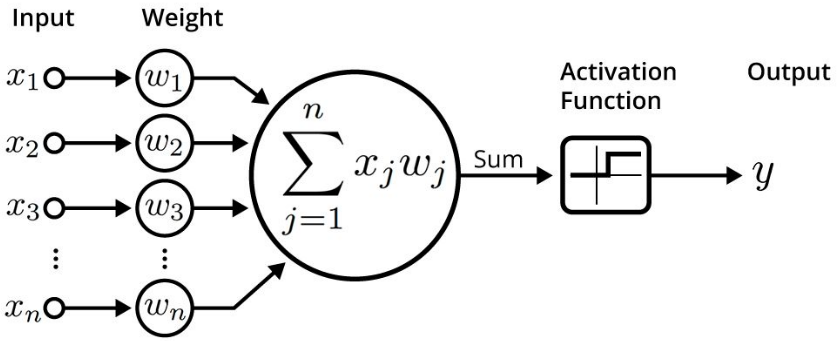
Shared or group intelligence that emerges from collaboration, collective efforts, and/or competition of many agents.

- **A single neuron has limited processing capabilities**: response speed is about several milliseconds.
- **However, the human brain is very powerful for problem solving:** it uses the aggregated power of millions of neurons.

Artificial Neuron

- An Artificial Neuron is a computational model of a biological neuron.
- The idea is that the artificial neuron receives input signals from other connected artificial neurons and via **a non-linear transmission function** emits a signal itself.
- Main operation:
  - Receives ***n inputs***
  - Computes the **weighted sum**
  - Passes through an **activation function**
  - Sends the signal to succeeding neurons





Artificial Neuron. Basic example

Two-inputs neuron operation:

1. Each input is multiplied by a **weight**

$$x_1 \rightarrow x_1 * w_1$$

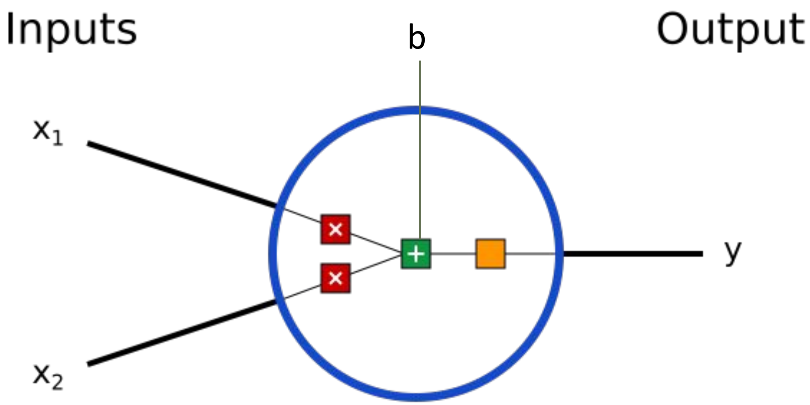
$$x_2 \rightarrow x_2 * w_2$$

2. All weighted sums are added with a **bias** b (***feedforward***)

$$(x_1 * w_1) + (x_2 * w_2) + b$$

3. The sum is passed through an **activation function**

$$y = f(x_1 * w_1 + x_2 * w_2 + b)$$



Code: *[basic-neuron.py]*

The output of the network depends on the **weights**, the **bias**, and the **activation function**

Artificial Neuron. Basic example

Two-inputs neuron operation:

1. Each input is multiplied by a **weight**

$$x_1 \rightarrow x_1 * w_1$$

$$x_2 \rightarrow x_2 * w_2$$

2. All weighted sums are added with a **bias** b (***feedforward***)

$$(x_1 * w_1) + (x_2 * w_2) + b$$

3. The sum is passed through an **activation function**

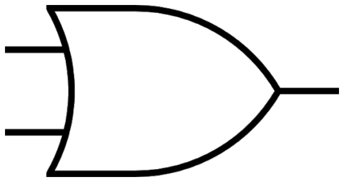
$$y = f(x_1 * w_1 + x_2 * w_2 + b)$$

```python
class BasicNeuron:
    """ It encapsulates a basic neuron that is constructed
    and an activation function. """

    def __init__(self, weights, bias, activation):
        self.weights = weights
        self.bias = bias
        self.activation = activation

    def feedforward(self, inputs):
        """ It applies the feedforward of the function: it
        function. """
        total = np.dot(self.weights, inputs) + self.bias
        return self.activation(total)
```

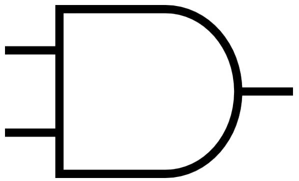
The output of the network depends on the **weights**, the **bias**, and the **activation function**

Artificial Neuron. What can we do?

- Try to implement a logic function with the two-input neuron.



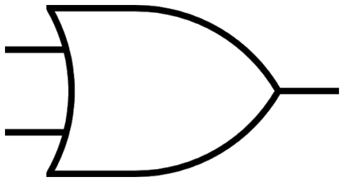
x_1	x_2	x_1 OR x_2
0	0	0
0	1	1
1	0	1
1	1	1



x_1	x_2	x_1 AND x_2
0	0	0
0	1	0
1	0	0
1	1	1

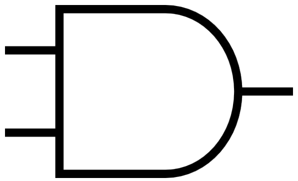
Artificial Neuron. What can we do?

- Try to implement a logic function with the two-input neuron.



x_1	x_2	x_1 OR x_2
0	0	0
0	1	1
1	0	1
1	1	1

```
weights = np.array([2, 2])  # w1 = 0, w2 = 1
bias = 0  # b = 0
or_function = BasicNeuron(weights, bias, step)
input = np.array([1, 0])  # x1 = 1, x2 = 0
output = or_function.feedforward(input)
print('OR({}) = {}'.format(input, output))
```



x_1	x_2	x_1 AND x_2
0	0	0
0	1	0
1	0	0
1	1	1

```
weights = np.array([1, 1])  # w1 = 1, w2 = 1
bias = 0  # b = 0
and_function = BasicNeuron(weights, bias, step)
input = np.array([0, 1])  # x1 = 0, x2 = 1
output = and_function.feedforward(input)
print('AND({}) = {}'.format(input, output))
```

Artificial Neuron. Basic example (2)

Two-inputs neuron operation:

1. Each input is multiplied by a **weight**

$$x_1 \rightarrow x_1 * w_1$$

$$x_2 \rightarrow x_2 * w_2$$

2. All weighted sums are added with a **bias** b (***feedforward***)

$$(x_1 * w_1) + (x_2 * w_2) + b$$

3. The sum is passed through an **activation function**

$$y = f(x_1 * w_1 + x_2 * w_2 + b)$$

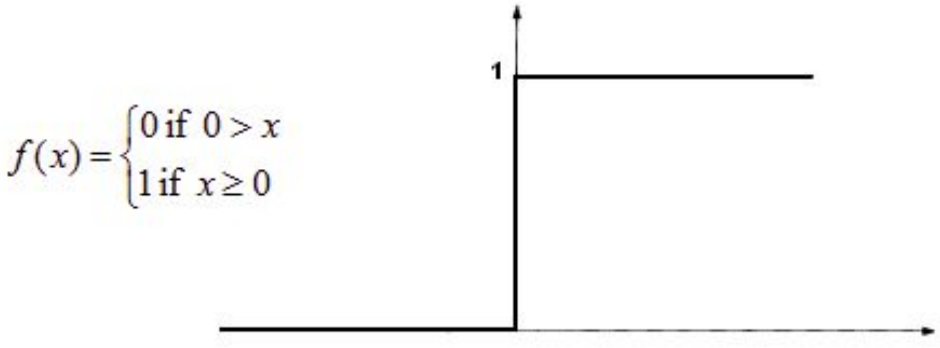
```python
class BasicNeuron:
    """ It encapsulates a basic neuron that is constructed
    and an activation function. """

    def __init__(self, weights, bias, activation):
        self.weights = weights
        self.bias = bias
        self.activation = activation

    def feedforward(self, inputs):
        """ It applies the feedforward of the function: it
        function. """
        total = np.dot(self.weights, inputs) + self.bias
        return self.activation(total)
```

The output of the network depends on the **weights**, the **bias**, and the **activation function**

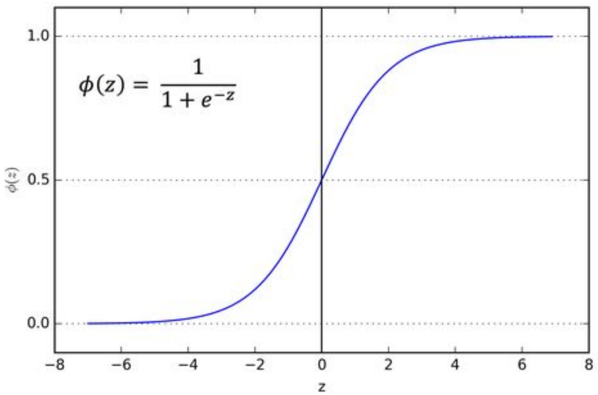
Activation Function

- Activation function decides whether a neuron should be **activated or not** by calculating the weighted sum and further adding bias to it. The motive is to **introduce non-linearity** into the output of a neuron.
- If we do not apply activation function then the output signal would be **simply linear function** (one-degree polynomial).
- Linear functions are limited in their complexity, have **less power**. Without activation function, our model cannot learn and model complicated data such as images, videos, audio, speech, etc.

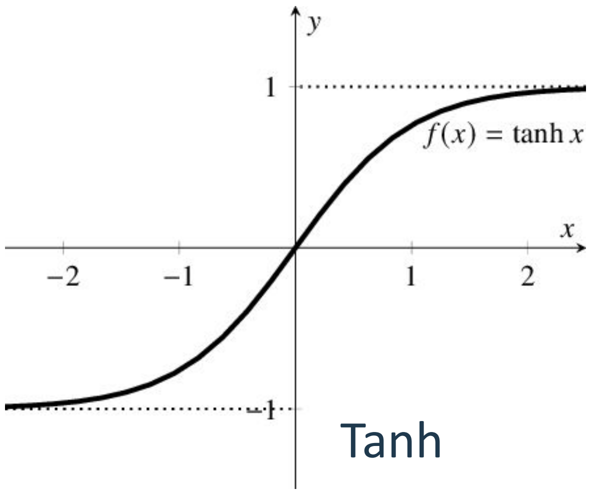
Activation Function. Types



Step

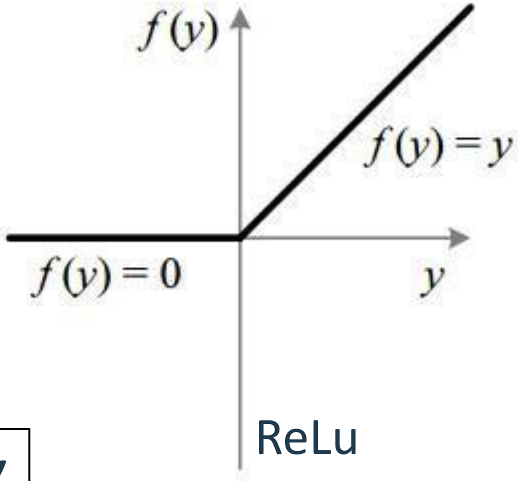


Sigmoid

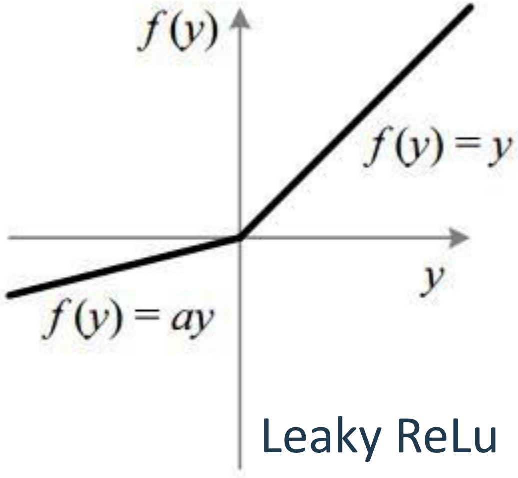


Tanh

Code: *neuron.py*



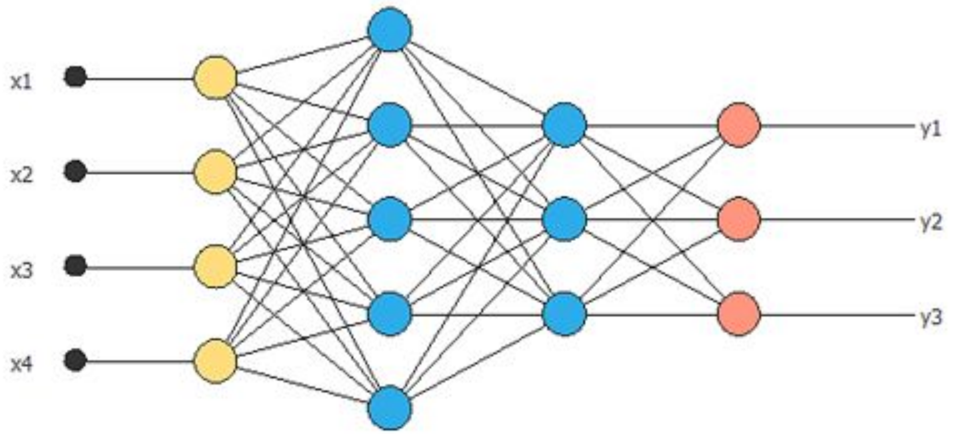
ReLu

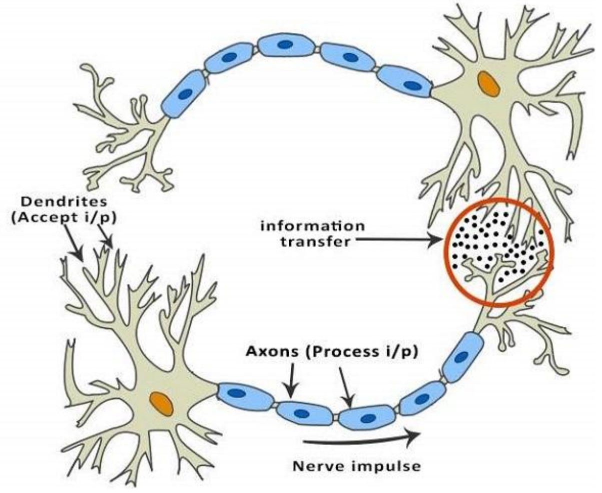


Leaky ReLu

Artificial Neural Networks

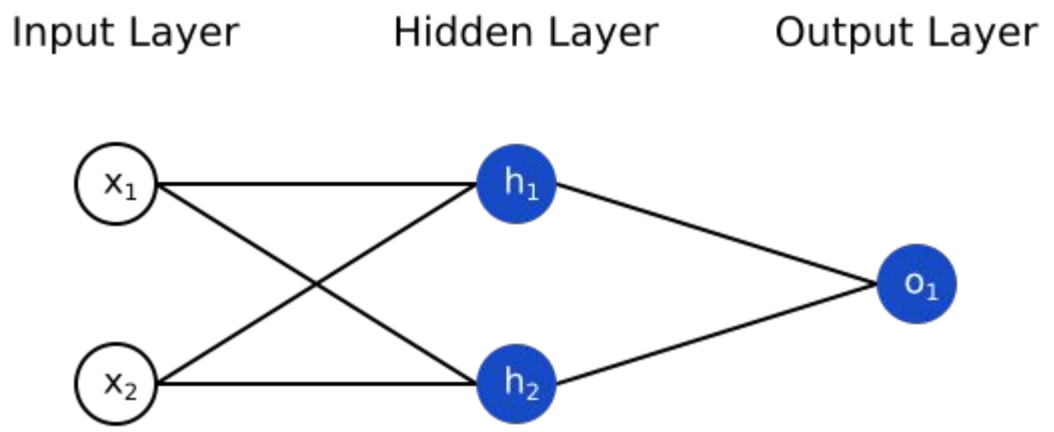
- A neural network is a bunch of neurons connected together.
- Neural networks are typically **organized in layers**.
- Layers are made up of a number of **interconnected neurons.**
- Inputs are presented to the network via the **input layer**, which communicates to **one or more hidden layers** through **weighted connections**.
- The hidden layers then link to an **output layer**.





Artificial Neural Networks. Code

- **Example 1**: Two inputs, two neurons in a hidden layer, and one output



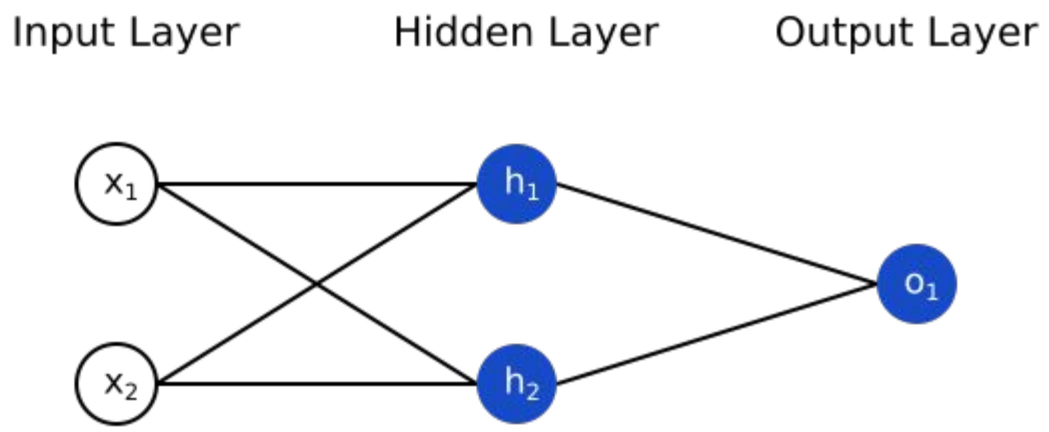
Code: *basic-two-layer-neural-network.py*

- **Example 2**: X inputs, H neurons in the hidden layer, and one output

Code: *two-layer-neural-network.py*

Artificial Neural Networks. Code

- **Example 1**: Two inputs, two neurons in a hidden lay



Code: *basic-two-layer-neural-network*

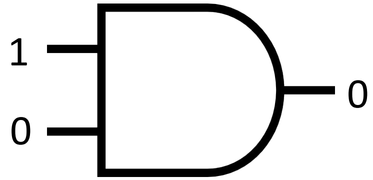
- **Example 2**: X inputs, H neurons in the hidden layer,

Code: *two-layer-neural-network.py*

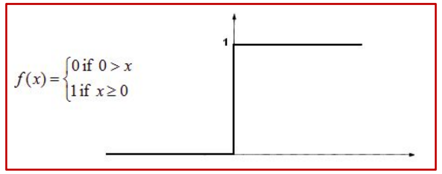
```python
class BasicNeuralNetwork:
    """
    A neural network with:
    - two inputs: x1, and x2
    - a hidden layer with two neurons: h1 and h2
    - an output layer with a neuron: o1
    The three neurons use the same weights and bias
    """

    def __init__(self, weights, bias, activation):
        self.h1 = Neuron(weights, bias, activation)
        self.h2 = Neuron(weights, bias, activation)
        self.o1 = Neuron(weights, bias, activation)

    def feedforward(self, x):
        """First we compute the output of the first layer"""
        output_h1 = self.h1.feedforward(x)
        output_h2 = self.h2.feedforward(x)

        """The outputs of the hidden layer h1 and h2 are the input of the
        output_o1 = self.o1.feedforward(np.array([output_h1, output_h2]))

        return output_o1
```

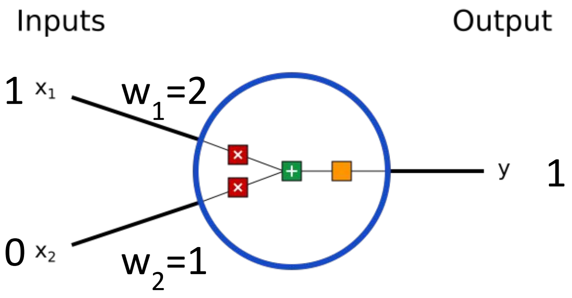
How do ANNs Learn?

- The **output** of the ANN depends on the **weights**
- **Learning** consist on **updating the weights** to get a desired output, i.e., **minimize the error**



$$y = f(x_1 * w_1 + x_2 * w_2 + b)$$



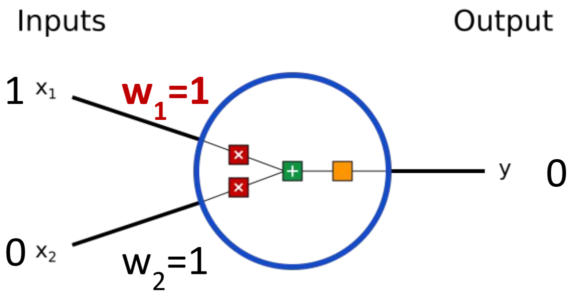


y = step(1 x 2 + 0 x 1, 2)
y = step(2, 2) = 1

Compute error

Error = MSE = 1

Update weights

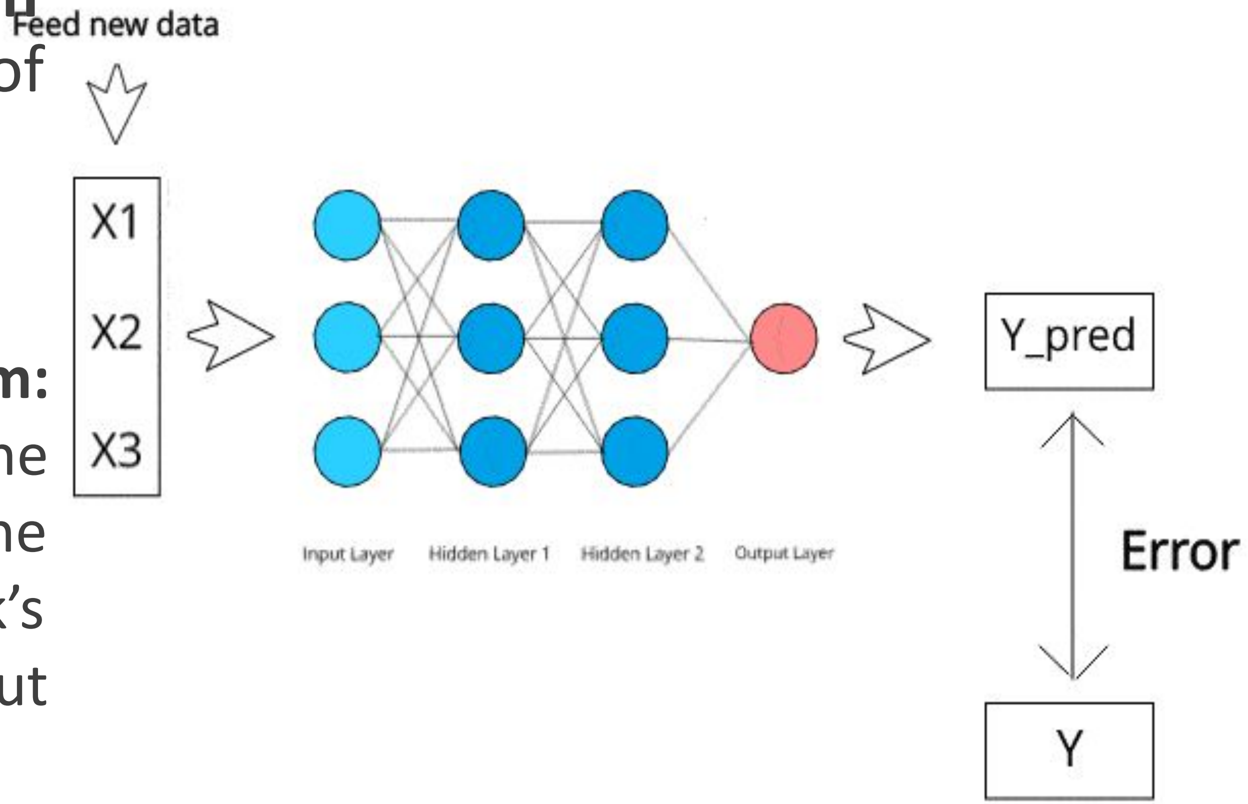


y = step(1 x 1 + 0 x 1, 2)
y = step(1, 2) = 0

How do ANNs Learn?

ANNs learn by solving the **optimization problem** of reducing the error in terms of ***error***, ***loss*** or ***cost function***.

Back Propagation Algorithm: It learns by example. If you submit to the algorithm the example of what you want the network to do, it changes the network's weights so that it can produce desired output for a particular input on finishing the training.



How do ANNs Learn? A simple example

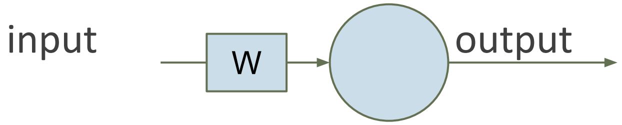
- **Objective:** Learn the following data

Input	Desired output
0	0
1	2
2	4
3	6
4	8

output = 2 x input

- **Error function**: mean squared error (MSE)

- **Neural network model:** output = W x input (W represents the weight)



How do ANNs Learn? A simple example

First step: Random weights initialization → W = 3 → output = 3 x input

Second step: Get actual output → Forward propagate input

Input	Actual output
0	0
1	3
2	6
3	9
4	12

How do ANNs Learn? A simple example

Third step: Get loss values ⟶ loss = f(actual output, desired output)

In our example f is mean squared error (MSE)

Input	Actual output	Desired output	Loss = Square error
0	0	0	0
1	3	2	1
2	6	4	4
3	9	6	9
4	12	8	16

Total loss is **30**

How do ANNs Learn? A simple example

Fourth step: Differentiation

In our numerical example: $-1000.0 < W < 1000.0$

We can move with steps of 0.0001

Optimization problem → finding W that minimizes loss

Differentiation allows us to address the problem

Remember → the derivative of a function at a certain point, gives the rate or the speed of which this function is changing its values at this point

In order to see the effect of the derivative, we can ask ourselves the following question: how much the total error will change if we change the internal weight of the neural network with a certain small value **$\delta W=0.0001$**

Loss with W=3.0001 → 30.006

Loss with W=2.9999 → 29.994

How do ANNs Learn? A simple example

Fourth step: Differentiation

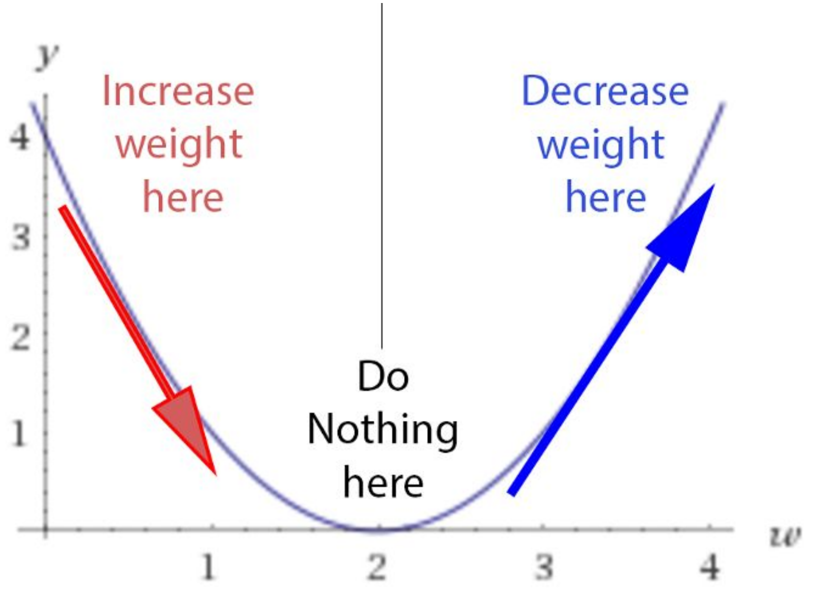
We could guess this rate by calculating directly the derivative of the loss function

Here is what our loss function looks like:

If **W=2**, we have a loss of 0, since the neural network actual output fits perfectly the training set

If **W<2**, we have a positive loss function, but the derivative is negative, meaning that an increase of weight will decrease the loss function

If **W>2**, we have a positive loss, but the derivative is positive, meaning that any more increase in the weight, will increase the losses even more



Arrows represent the derivative at the corresponding points

How do ANNs Learn? A simple example

Fifth step: Backpropagation

In this example, we used only one layer neural network

$\longrightarrow$ No backpropagation is needed!!!

In the case there are more layers, the process is the same but each layer (as the output layer with the loss function) requires to provide the function of its derivative.

Thus, we only need to keep a stack of the function calls during the forward pass and their parameters, in order to know the way back to back-propagate the errors using the derivatives of these functions. This can be done by de-stacking through the function calls. This technique is called auto-differentiation.

How do ANNs Learn? A simple example

Sixth step: Weight update

Thus as a general rule of weight updates is the delta rule:

New weight = old weight - derivative x learning rate

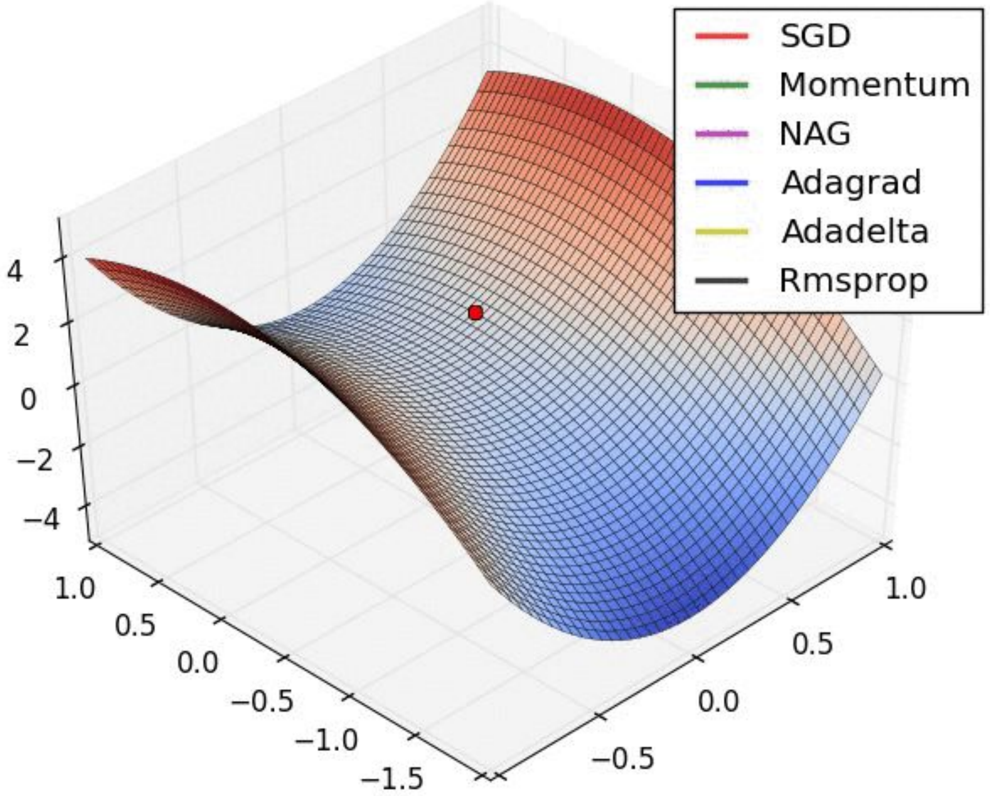
If the **derivative rate is positive**, it means that an increase in weight will increase the error, thus the new weight should be smaller.

If the **derivative rate is negative**, it means that an increase in weight will decrease the error, thus we need to increase the weights.

If the **derivative is 0**, it means that we are in a stable minimum. Thus, no update on the weights is needed -> we reached a stable state.

How do ANNs Learn? A simple example

The importance of the optimization method applied



https://towardsdatascience.com/overview-of-various-optimizers-in-neural-networks-17c1be2df6d5

Deep Learning

ANN results get better with

- more/better **data**
- bigger **models**
- more **computation**

Deep Learning

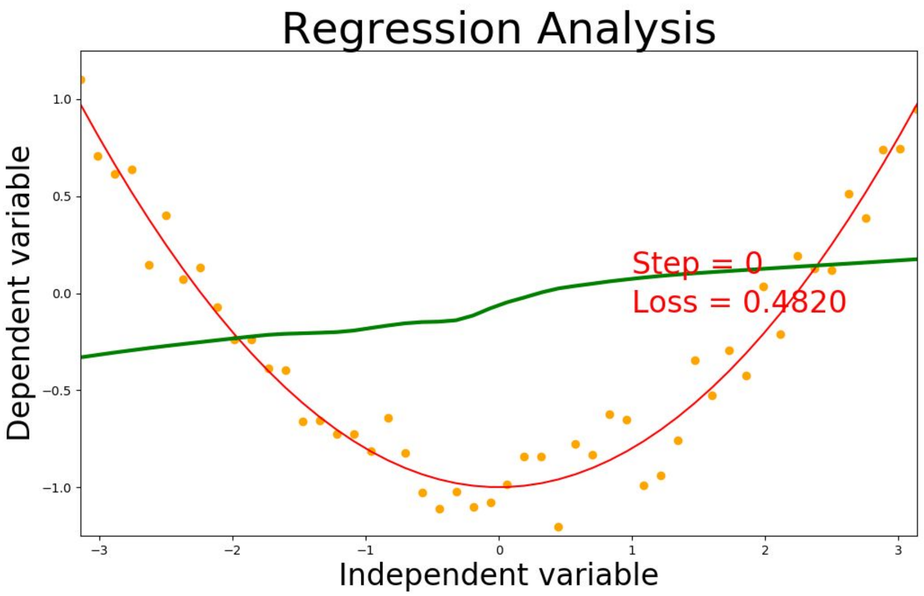
ANN results get better with

- more/better **data**
- bigger **models**
- more **computation**

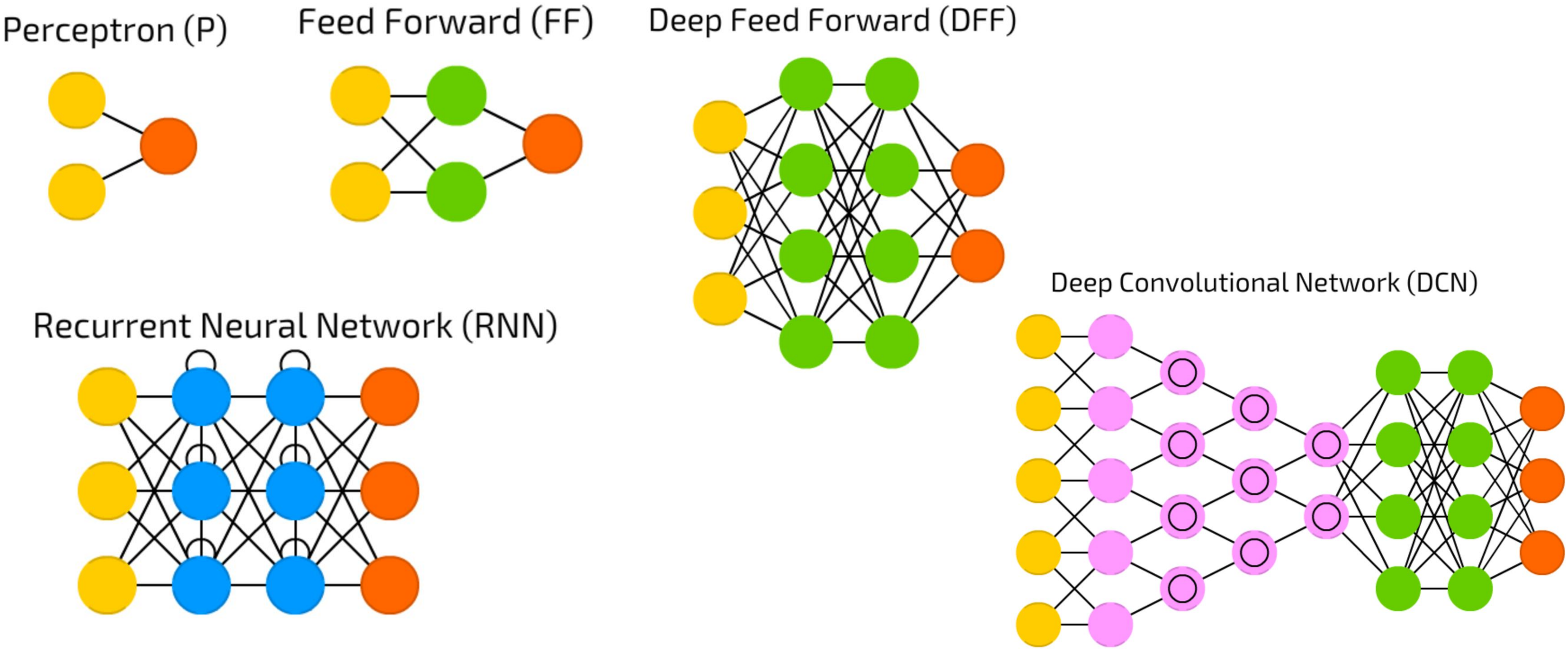
Example:

Deep Learning for regression

https://drive.google.com/file/d/115dbtD8Zxs2I98t0wSzvauEkie4IPI5P/view?usp=sharing



Artificial Neural Networks. Types



Thanks!

Comments?

JAMAL TOUTOUH
jamal@uma.es
jamal.es
@jamtou

Sergio Nesmachnow
sergion@fing.edu.uy