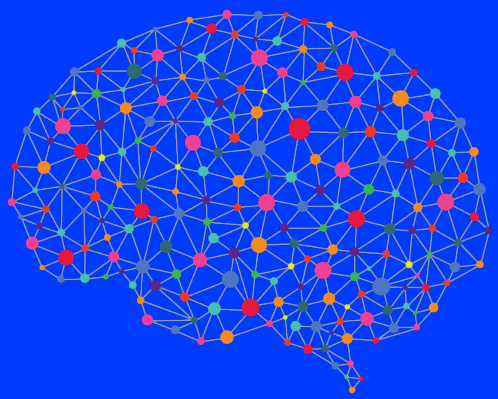
{EPITECH}

MY_TORCH - BOOTSTRAP_

< INTRODUCING ARTIFICIAL NEURAL NETWORKS />



MY_TORCH - BOOTSTRAP

In 1948, in a paper named Intelligent Machinery, Alan TURING introduced a type of neural networks named B-type unorganized machine that he considered as the simplest possible model of the nervous system.

The objectives of this bootstrap are:

- ✓ to give you a first approach to artificial neural networks
- ✓ to have you implement the algorithms at the heart of the training process.

⚠ Libraries that handles the implementation of artificial neural networks are forbidden.
You must code everything from scratch

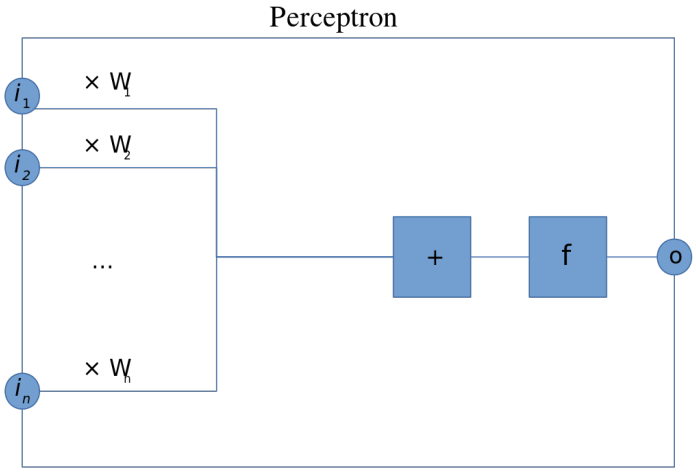
📢 You can use libraries to handle linear algebra and to display the learning curves.

The perceptron

Invented in 1943 by McCULLOCH and PITTS, the perceptron is a **single layer linear classifier**.
The first hardware implementation was built in 1957 by ROSENBLATT.

The perceptron mimics a neuron inside a human brain and provides a single output.
This single node is activated depending on the values of its **inputs** and its **activation function**.

Each connection contains a weight (a number generally between 0 and 1) which is learned during the **training phase**. An extra number is generally provided, called the **bias**, which can be interpreted as an extra weight from an input whose value will always be 1.



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

Terminal

```
$>  ./my_perceptron --help
USAGE
    ./my_perceptron [--new NB_INPUTS | --load LOADFILE] [--save SAVEFILE] FILE

DESCRIPTION
    --new      Creates a new perceptron with NB_INPUTS inputs.
    --load     Loads an existing perceptron from LOADFILE.
    --save     Save the perceptron's state into SAVEFILE. If not provided, the state of the
perceptron will be displayed on standard output.
    FILE     a file containing a list of inputs (and expected outputs) that the perceptron
needs to evaluate (either for training, or predicting).
```

Step I. The basics

A perceptron is a simple linear classifier. It is only defined by:

- ✓ input weights
- ✓ the bias
- ✓ the activation function
- ✓ the learning rate

Basically, that means that your **perceptron** will just make a weighted sum of all the inputs, add the bias, and send the result to an activation function: if it's above a certain **threshold**, the output will be 1, and 0 otherwise. Different activation function will lead to different results!

If some of these concepts are a bit vague, try to do some research before going further!

First, create a program that generates a new perceptron with a user-defined number of random weights (and a random bias), a fixed learning rate, and stores its state in a file.

We generally use the Heaviside function as the perceptron's activation function, but you can choose another one if you wish (like the hyperbolic tangent)

Step II. Your first perceptron

Generate a perceptron with 2 inputs. We will try to make it mimic the behavior of an AND gate. Obviously, this problem could easily be solved programmatically, or even by specifying the right weights yourself. You need to make your perceptron "*learn*" how to be an AND gate from a random configuration.

Don't forget your truth tables

Now let's try to train your perceptron. First, you need to generate your training data. As this is a really easy configuration, there are only 4 cases to handle.
Your perceptron will need a way to correct its weights depending on its output. Therefore, you need to implement a **learning algorithm**.
A simple learning algorithm can be found easily.

Make a program that takes a perceptron, and will train it in two phases:

- ✓ phase 1: Accuracy checking. During this phase, send every configuration possible (only 4 here) to your perceptron, and check the output for each one. If at least one is wrong, go to phase 2, otherwise stop your program: the perceptron is now valid!
- ✓ phase 2: Training. Choose a random configuration and send it to your perceptron. Then, update the weights using the learning algorithm. Repeat this phase a fixed number of times (10 times? 50 times? 1,000,000,000 times?), then go back to phase 1.

Your perceptron should converge toward an optimal solution with 100% accuracy quite quickly.

If you got ”lucky” and generated a perceptron that already computes an AND gate perfectly without training, try generating a new one to test your training algorithm!

Step III. Automation

As the previous example was pretty simple, a lot of steps could be done by hand. In order to facilitate future debugging and learning, try to automate all this process using scripts!

You could also consider plotting your results to get a better understanding of what's happening, and how fast your perceptron is ”learning”

During the learning phase, checking the percentage of accuracy of your model could be an interesting data to know

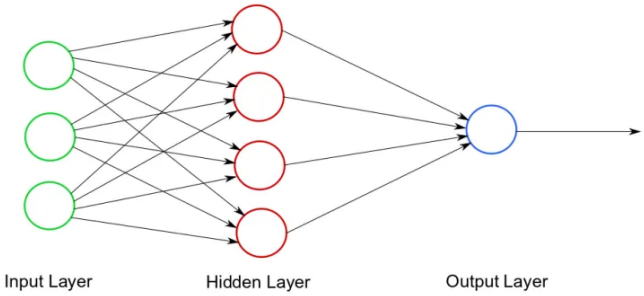
Step IV. The limit

Now that you have a perceptron that ”learned” how an AND gate works, and thanks to your scripts to automate all the learning process, try to generate a new perceptron with 2 inputs and 1 output. This time, your perceptron should mimic a XOR gate. That should be quite easy, right?

The multilayer perceptron

Unfortunately, the results of Frank's perceptron were not as good as expected and the idea was abandoned. Moreover, as you may have seen while trying to mimic a XOR gate, a single-layer perceptron only allows you to compute linear functions.

To compute more complex functions, the multilayer perceptron (MLP) was developed. The idea is to connect multiple neuron layers between them, each layer containing potentially a different number of neurons. Each neuron from a layer will be connected to every neurons from the next layer. Each neuron from a layer is a single layer perceptron, whose inputs are every neuron from the previous layer, and whose output is one input for every neuron of the next layer.



MLPs are able to distinguish data that are not linearly separable.

Pros:

- ✓ MLP models can solve complex non-linear problems ;
- ✓ they work well with both small and large input data ;
- ✓ they quickly provide some predictions after the training.

Cons:

- ✓ computations are time-consuming and complex ;
- ✓ it's hard to predict how much the dependent variable affects each independent variable ;
- ✓ the model functioning depends on the quality of training.

> Modern feedforward networks are trained using the **backpropagation** method and are colloquially referred to as the *vanilla neural networks*.
>
> **— Wikipedia —**

The XOR gate

Now, try to generalize your perceptron such that you can have multiple layers of multiple neurons. We will generate a neural networks with this configuration:

- ✓ Input layer: 2 neurons with 1 input each (No activation function, they simply transmit inputs to the hidden layer)
- ✓ One hidden layer: 2 neurons
- ✓ Output layer: 1 neuron

Each neuron from a layer takes as input the outputs of every neuron from the previous layer. Use the sigmoid function as the activation function for every neurons of the hidden and the output layers.

Train your new neural network until it gets perfect on all 4 cases of a XOR gate. Then, try multiple configuration for the hidden layer and the learning rate, and keep the data for how fast your MLPs converge toward an optimal solution.

A higher learning rate generally means a quicker convergence toward an optimal solution, but also a higher volatility!

Activation functions for a MLP are generally different for the hidden layers and the output layers

Tic-tac-toe

Simulating logical gates is good, but still a bit too easy: there are always only 4 configurations to test on a classical 2 inputs/1 output door. Let's try to go a bit further in complexity.

Try to create a program that loads a MLP which takes a tic-tac-toe board as input, and outputs the state of the game. Try to think carefully of how many neurons you need in each layer!
You should try to handle the 3 possible states of the game:

- ✓ ”Nothing”: the game is still on ;
- ✓ ”Win”: a player won ;
- ✓ ”Draw”: the game is finished without a winner.

However, on the first attempt, you may want to only get a binary output: ”Nothing” (includes ”nothing” and ”draw”) and ”Win”. Then, when you're satisfied with your results, try to re-train it in order to get the 3 different states and, why not, tell which player has won (O or X).

As we train or AI using supervised learning, we need labeled data. Fortunately, the number of possible boards for a Tic-Tac-Toe is not that large: you could write a script that will generate every tic-tac-toe boards labeled with the expected output for your program (or try to find if it already exists online).

Getting enough labeled data is a major issue of supervised learning; too few data leading, among other things, to **overfitting** and poor generalization.

Now, train your neural network, update your weights using backpropagation, and repeat the process until you get a good accuracy (90% of good predictions is a good start, but try to see how accurate your neural network can get!).

In order to get good results and quick convergence toward an optimal solution, you will need to implement several learning optimizations, such as a **gradient descent** and **hyperparameter optimizations** (for example: learning rate decay, grid/random search, etc.).

What is the difference between **(Mini-)Batch Gradient Descent** and **Stochastic Gradient Descent**?

To go further

Resources to dig and interactive learning tools to play with:

- ✓ backpropagation ;
- ✓ tensorflow playground ;
- ✓ mladdict.

v 2