

[Tutorial] Multi Layer Perceptron

This tutorial will introduce the **Multilayer Perceptron (MLP)**, which is the most basic neural network architecture. Later, we will review more complex neural networks architectures.

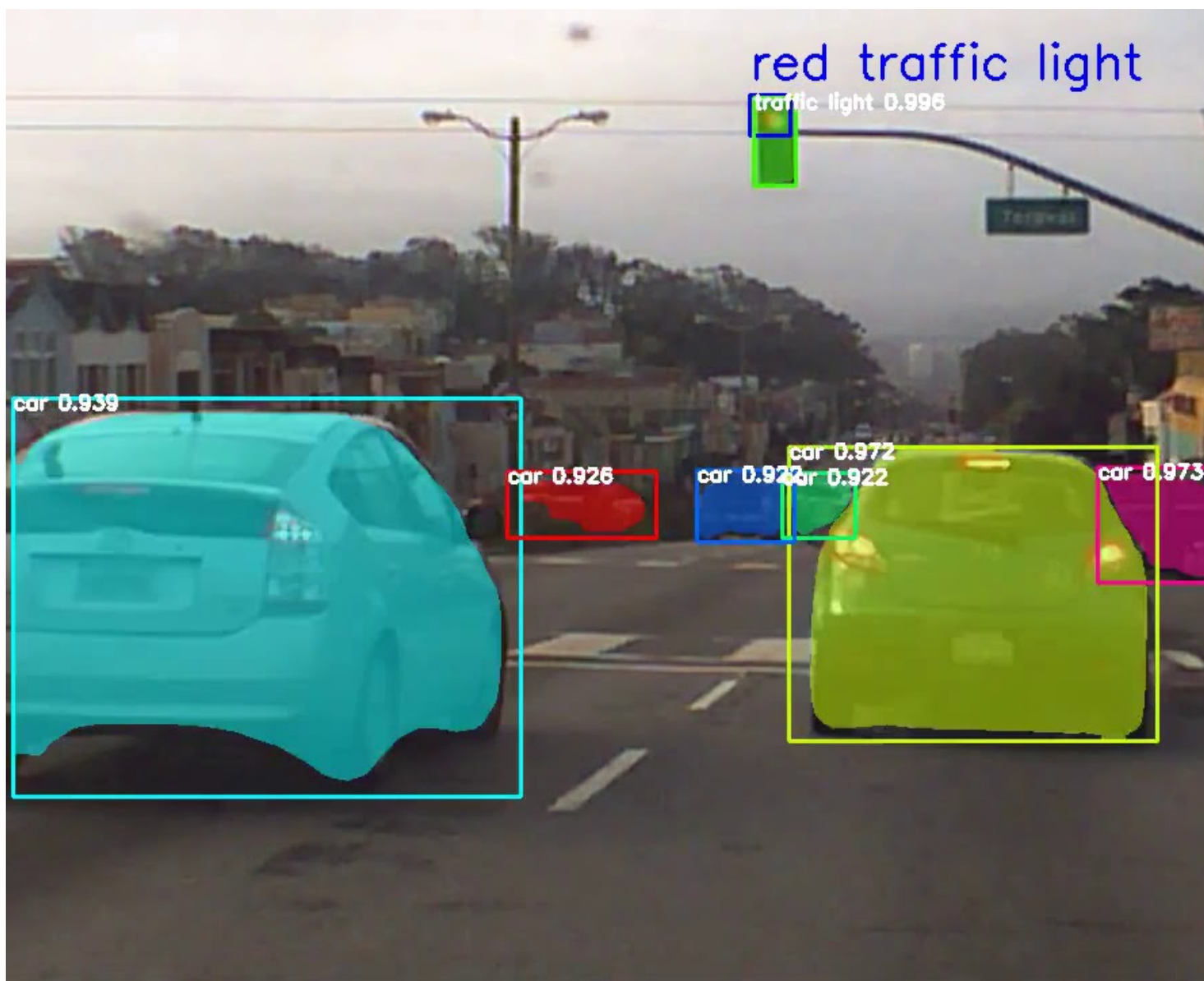
0. Introduction to Deep Learning

Deep Learning is the specific area of Machine Learning using Neural Networks. Deep Learning got some momentum with the rise of Big Data: a Deep Learning model **usually needs more data** than a regular Machine Learning one.

There are several types of Deep Learning architectures:

- Regular Neural Networks (or MLP)
- Convolutional Neural Networks
- Recurrent Neural Networks
- Generative Adversarial Networks
- ...

There are **many applications** of Deep Learning, here are some of the most common ones.









Google Translate

Japanese ▼



Tap to enter text



1. Perceptron

Perceptrons are the **building blocks** of neural networks (as a matter of fact, one perceptron is a **single layer neural network**) so having a good understanding of them now will be beneficial when learning about deep neural networks!

1.1. Neuron analogy

In 1957, [Frank Rosenblatt](https://en.wikipedia.org/wiki/Frank_Rosenblatt) [_ \(https://en.wikipedia.org/wiki/Frank_Rosenblatt\)](https://en.wikipedia.org/wiki/Frank_Rosenblatt) explored the following question:

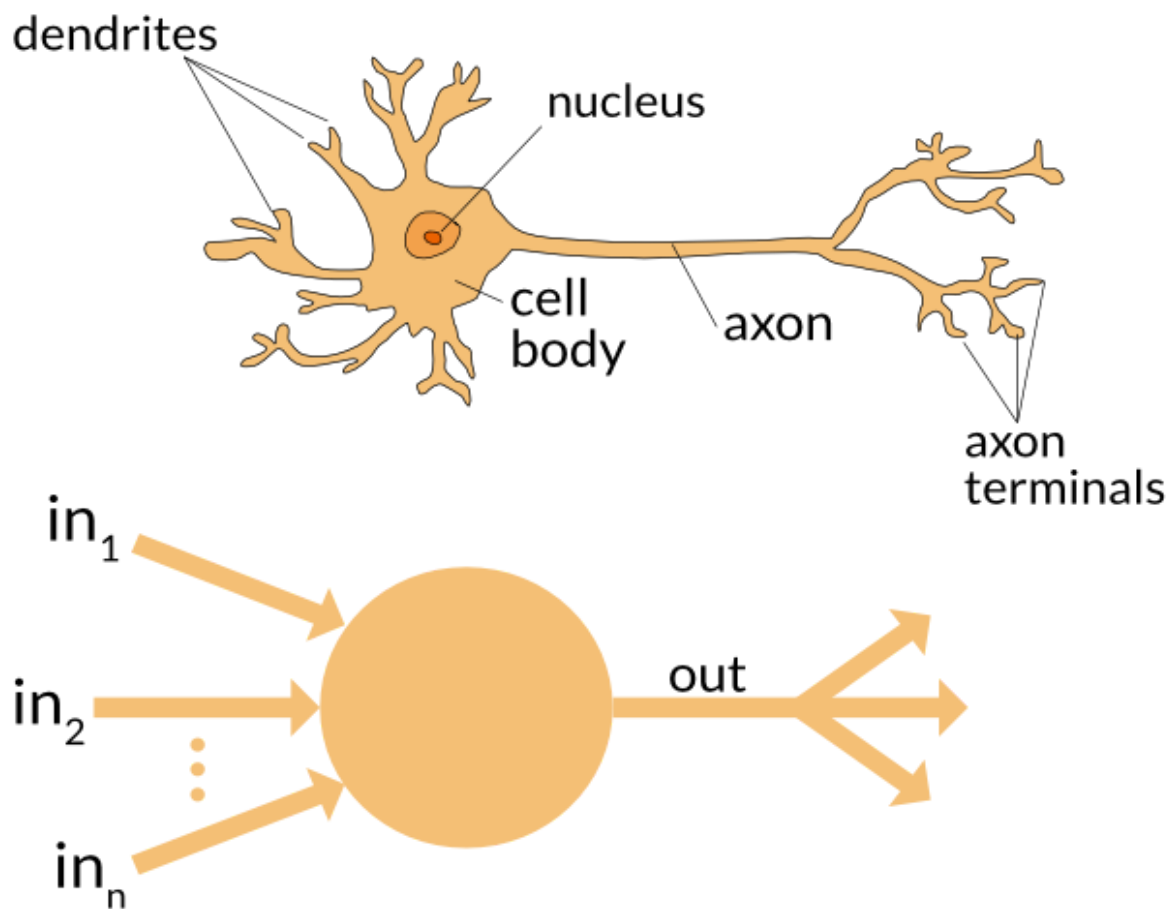
“How can computers solve such problems in the way a human brain does?” 🧠

This led him to the invention of the **Perceptron algorithm** that corresponds to an artificial neuron that simulate a biological neuron!

If you look closely, you might notice that the word "perceptron" is a combination of two words:

- **Perception**: the ability to sense something
- **Neuron**: a nerve cell in the human brain that turns sensory input into meaningful information

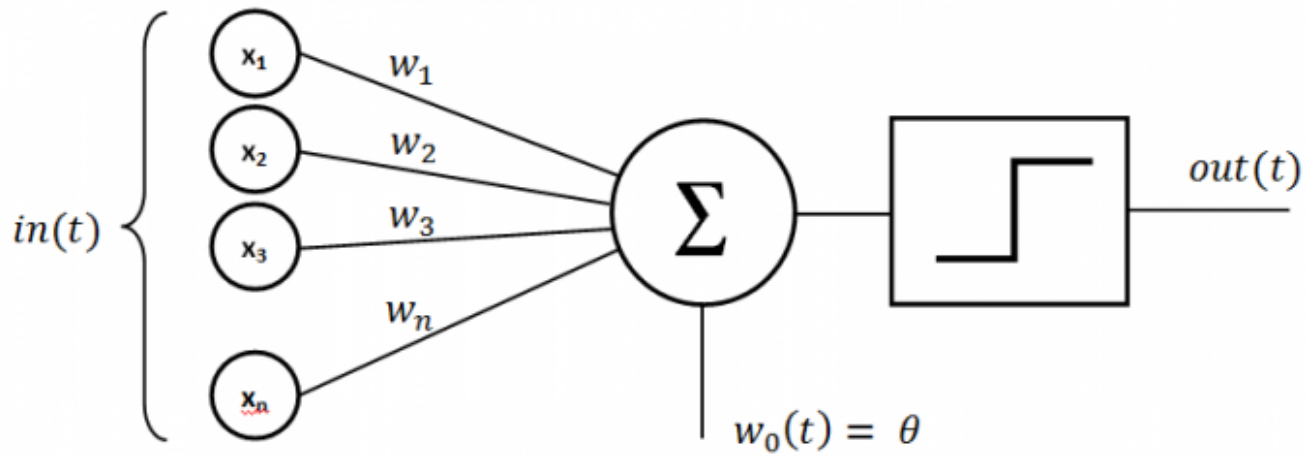
Just like a biological neuron has dendrites to receive signals, a cell body to process them, and an axon to send signals out to other neurons, the artificial neuron combines some **input** channels, a **processing stage** (based on specific rules), and fires an **output** (possibly sent to multiple other artificial neurons).



📖 **Resources:** More info about the **Perceptron** [_\(https://en.wikipedia.org/wiki/Perceptron\)_](https://en.wikipedia.org/wiki/Perceptron) and its history

1.2. Representation

A perceptron can simply be seen as a set of inputs, that are weighted and summed. This produces sort of a weighted sum of inputs, resulting in a constrained output.



1.2.1. Weighted sum

What is the **weighted sum**? This is just a number that gives a reasonable representation of the inputs. You obtain it by multiplying each weights with each inputs:

$$weightedsum = z = X_1 W_1 + X_2 W_2 + \dots + X_n W_n$$

It can also be called **dot product** (in `NumPy`, you can do a weighted sum like this : `np.dot(a, b)` with a and b two Numpy 1D arrays of same length).

We usually introduce a bias term w_0 or b that is also added to the weighted sum.

1.2.2. Step function

The output of the weighted sum is passed through a **step function**, resulting in a binary result:

$$Y = \begin{cases} 1 & \text{if } z > 0, \\ 0 & \text{otherwise} \end{cases} \quad Y = \begin{cases} 1 & \text{if } z > 0, \\ 0 & \text{otherwise} \end{cases}$$

1.2.3. Learning process

The learning process behind the perceptron is greatly inspired by **Hebb rule**:

« cells that fire together, wire together »

For each input fed to the model, we update the weights like this:

$$W'_i = W_i + \alpha(Y_t - Y)X_i$$

where:

- W'_i = the updated weight i
- W_i = the initial weight i

- Y_t = the expected output (the ground truth)
- \hat{Y} = the observed output (the prediction)
- α = the learning rate is an hyperparameter

1.3. `scikit-learn` implementation

Perceptron is available in `scikit-learn`, of course! The signature is the following:

```
class sklearn.linear_model.Perceptron(penalty=None, alpha=0.0001, fit_intercept=True, max_iter=None,
tol=None, shuffle=True, verbose=0, eta0=1.0, n_jobs=None, random_state=0, early_stopping=False, vali
dation_fraction=0.1, n_iter_no_change=5, class_weight=None, warm_start=False, n_iter=None)
```

Let's try it on the Iris dataset for example:

```
from sklearn.datasets import load_breast_cancer
from sklearn.linear_model import Perceptron
from sklearn.model_selection import train_test_split
# Load the data
X, y = load_breast_cancer(return_X_y=True)

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)

# Instantiate and fit the model
perceptron = Perceptron(tol=None)
perceptron.fit(X_train, y_train)

# Print the score
print('accuracy:', perceptron.score(X_test, y_test))
```

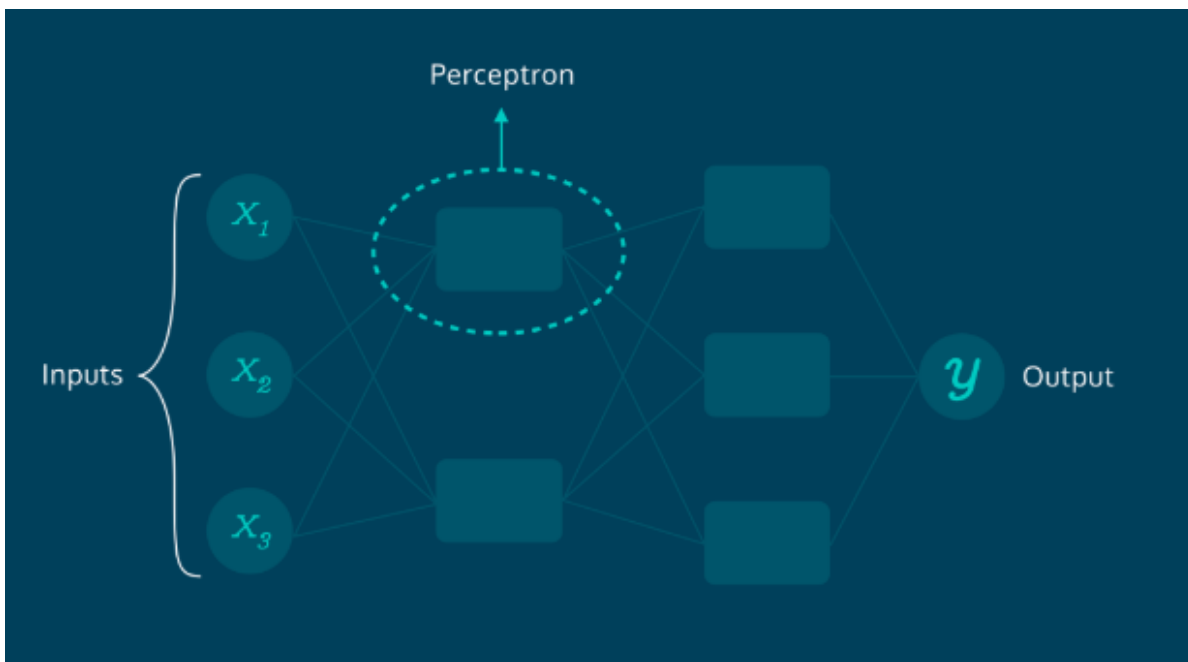
```
accuracy: 0.9473684210526315
```


2. Multilayer perceptron

A Perceptron is a **linear model**, meaning that - if we take the example of an input with 2 dimensions - it can separate data points that can be separated by a single line. The reason why it's so basic is because at its core is a simple activation function, which is a simple binary function, that only has two possible and calculable results.

What happens when the **data is not linearly separable** ? A single perceptron fed by two inputs would not work in such scenario.

But it can be solved ! By adding the number of features and perceptrons we can give birth to the **Multilayer Perceptrons (MLP)** also known as Neural Networks (as it represents a network of neurons).



 **Resources:** Fascinating and greatly explained neural networks (Perceptron are at the middle of the video):

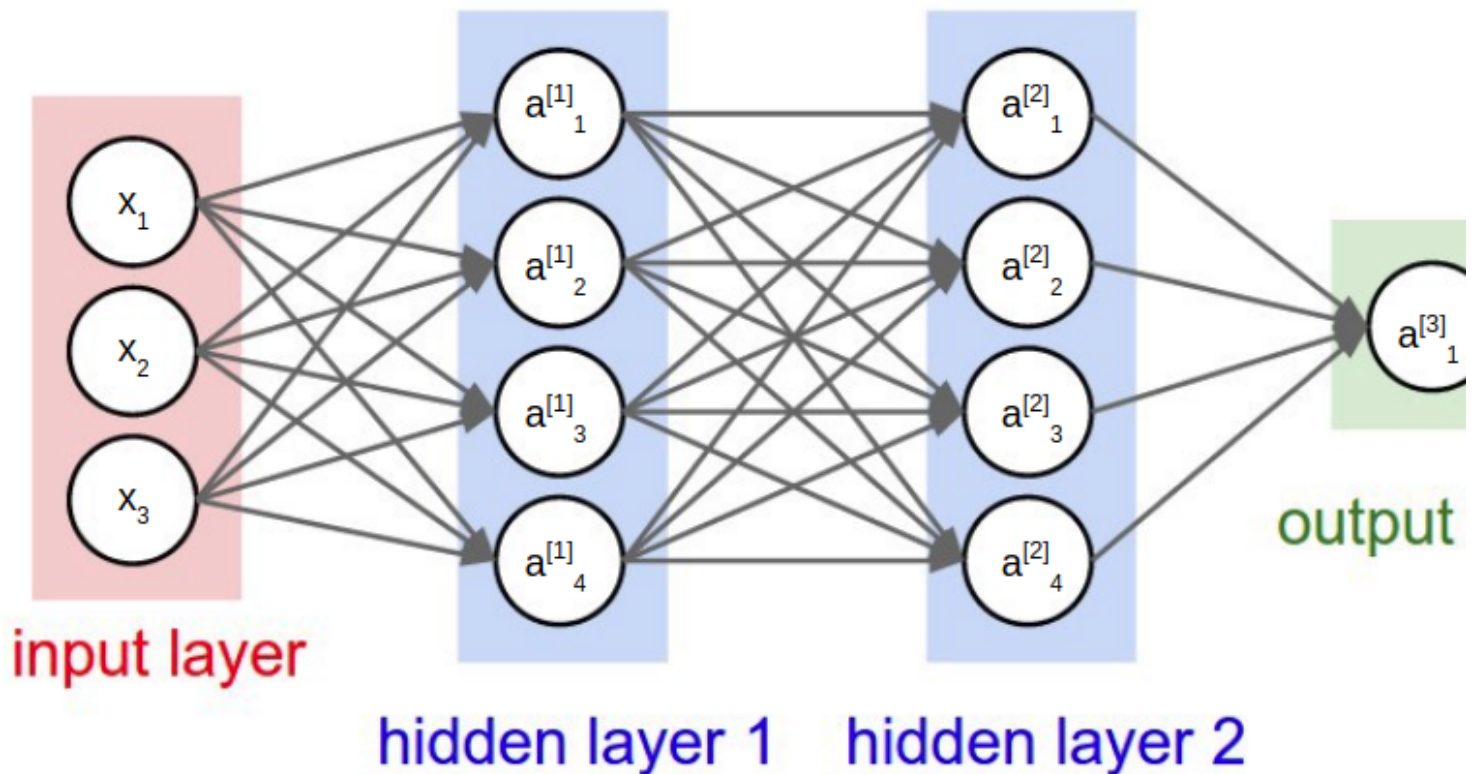
<https://www.youtube.com/watch?v=aircAruvnKk&t=6s> [_ \(https://www.youtube.com/watch?v=aircAruvnKk&t=6s\)](https://www.youtube.com/watch?v=aircAruvnKk&t=6s)



[\(https://www.youtube.com/watch?v=aircAruvnKk&t=6s\)](https://www.youtube.com/watch?v=aircAruvnKk&t=6s)

2.1. Neural networks representation

Neural networks are usually represented on diagrams using a standard convention:



This diagram has to be read from left to right.

On the left, here in red, is the **input layer**: this is actually the input features $X = (x_1, x_2, x_3)$ (e.g. number of rooms in a house, presence of a garden...). The number of **units** is the number of input features.

On the right, in green, is the **output layer**: this is the prediction of target value (e.g. the house price in a regression, or the class in classification). The number of **units** depends on the task (for regression it is usually one, for multiclass classification the number of classes).

There is always one and only one input layer and output layer.

In the middle, in blue, are the **hidden layers**. There can be an arbitrary number of hidden layers. The hidden layers also have an arbitrary number of **units**.

2.2. Forward propagation

Now that we understand the representation, we would understand how to compute the output predicted value, given input features $X = (x_1, x_2, \dots, x_N)$.

To compute this output predicted value, we need to weights W of each unit. Each unit i of a layer l (except the input layer) has associated weights $W^{[l]}_{i,j}$. We will use those weights compute the activation $a^{[l]}_i$ of each unit.

- Hidden layer 1

Considering our example in diagram, we could compute the activations of the first hidden layer using the following formulas:

$$a_1^{[1]} = W_1^{[1]} \times X + b_1^{[1]} \{1\} = W^{[1]}_{\{1\}} \times X + b^{[1]}_{\{1\}} \quad a1[1]=W1[1] \times X + b1[1]$$

$$a_2^{[1]} = W_2^{[1]} \times X + b_2^{[1]} \{2\} = W^{[1]}_{\{2\}} \times X + b^{[1]}_{\{2\}} \quad a2[1]=W2[1] \times X + b2[1]$$

$$a_3^{[1]} = W_3^{[1]} \times X + b_3^{[1]} \{3\} = W^{[1]}_{\{3\}} \times X + b^{[1]}_{\{3\}} \quad a3[1]=W3[1] \times X + b3[1]$$

$$a_4^{[1]} = W_4^{[1]} \times X + b_4^{[1]} \{4\} = W^{[1]}_{\{4\}} \times X + b^{[1]}_{\{4\}} \quad a4[1]=W4[1] \times X + b4[1]$$

Where $b_i^{[1]}$ $b1[1]$ is called the bias, and i just an additional parameter.

- Hidden layer 2

Now if we want to compute the activations of the second hidden layer $a_i^{[2]}$ $ai[2]$, we would use the exact same formulas, but with the activations of the first hidden layer as input ($a_i^{[1]}$ $ai[1]$), instead of the input features:

$$a_1^{[2]} = W_1^{[2]} \times a^{[1]} + b_1^{[2]} \{1\} = W^{[2]}_{\{1\}} \times a^{[1]} + b^{[2]}_{\{1\}} \quad a1[2]=W1[2] \times a[1] + b1[2]$$

$$a_2^{[2]} = W_2^{[2]} \times a^{[1]} + b_2^{[2]} \{2\} = W^{[2]}_{\{2\}} \times a^{[1]} + b^{[2]}_{\{2\}} \quad a2[2]=W2[2] \times a[1] + b2[2]$$

$$a_3^{[2]} = W_3^{[2]} \times a^{[1]} + b_3^{[2]} \{3\} = W^{[2]}_{\{3\}} \times a^{[1]} + b^{[2]}_{\{3\}} \quad a3[2]=W3[2] \times a[1] + b3[2]$$

$$a_4^{[2]} = W_4^{[2]} \times a^{[1]} + b_4^{[2]} \{4\} = W^{[2]}_{\{4\}} \times a^{[1]} + b^{[2]}_{\{4\}} \quad a4[2]=W4[2] \times a[1] + b4[2]$$

- Output layer

Finally, to compute the output layer would be exactly the same: $a_1^{[3]} = W_1^{[3]} \times a^{[2]} + b_1^{[3]} \{1\} = W^{[3]}_{\{1\}} \times a^{[2]} + b^{[3]}_{\{1\}} \quad a1[3]=W1[3] \times a[2] + b1[3]$

👉 As you can see, forward propagation is not really complicated, it is just multiplications and additions.

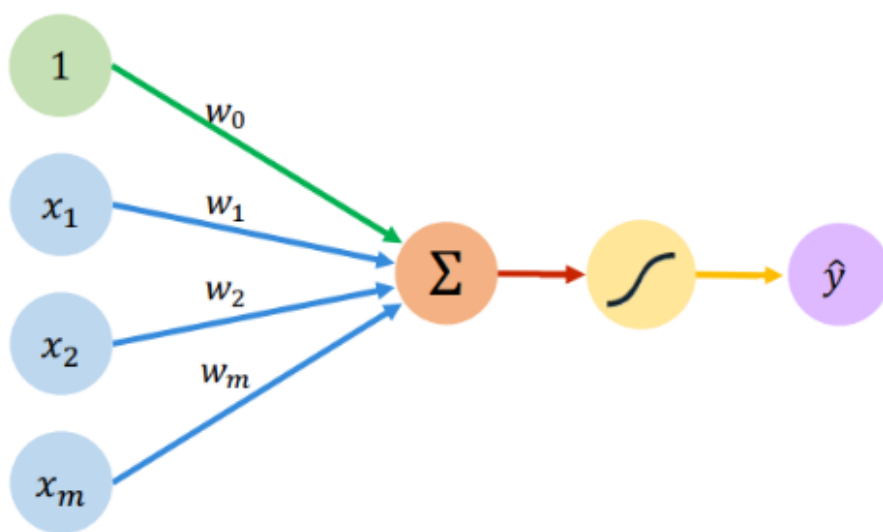
⚠️ You may also notice that the activation values $a_i^{[l]}$ $ai[l]$ have no boundaries: they could go up to \pm infinity. We are going to change that in a few moments.

2.3. Activation functions

2.3.1. Why do we add activation functions ?

However, adding layers of perceptrons as we know them doesn't change anything : **a weighted sum of a weighted sum is just... a weighted sum** ! So right now, our MLP is just a big perceptron. That's why activation functions were added to perceptrons !

The Perceptron: Forward Propagati



Output

Linear c

$$\hat{y} = g \left(w_0 + \sum_{i=1}^m \right)$$

Non-linear activation function

Bias

Inputs Weights Sum Non-Linearity Output

➡ Activation functions, often called g , **add non-linearity** to our MLP and allow to handle much more complex problems (without activation function, only linearly separable problems can be solved).

The purpose of activation functions is to introduce non linearity

➡ The formulas of our forward propagation are easily modified :

$$a_1^{[1]} = g(W_1^{[1]} \times X + b_1^{[1]}) = g(z_1^{[1]}) = g(W^{[1]}_{\{1\}} \times X + b^{[1]}_{\{1\}}) = g(z^{[1]}_{\{1\}}) \quad a1[1]=g(W1[1] \times X + b1[1])=g(z1[1])$$

$$a_2^{[1]} = g(W_2^{[1]} \times X + b_2^{[1]}) = g(z_2^{[1]}) = g(W^{[1]}_{\{2\}} \times X + b^{[1]}_{\{2\}}) = g(z^{[1]}_{\{2\}}) \quad a2[1]=g(W2[1] \times X + b2[1])=g(z2[1])$$

$$a_3^{[1]} = g(W_3^{[1]} \times X + b_3^{[1]}) = g(z_3^{[1]}) = g(W^{[1]}_{\{3\}} \times X + b^{[1]}_{\{3\}}) = g(z^{[1]}_{\{3\}}) \quad a3[1]=g(W3[1] \times X + b3[1])=g(z3[1])$$

$$a_4^{[1]} = g(W_4^{[1]} \times X + b_4^{[1]}) = g(z_4^{[1]}) = g(W^{[1]}_{\{4\}} \times X + b^{[1]}_{\{4\}}) = g(z^{[1]}_{\{4\}}) \quad a4[1]=g(W4[1] \times X + b4[1])=g(z4[1])$$

(and so on for the other layers)

2.3.2. Most common activation functions

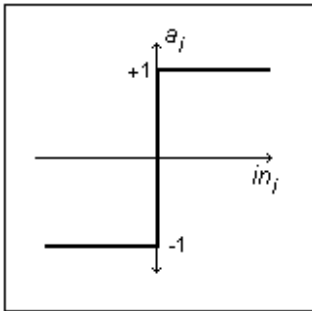
There are five main activation functions: *sigmoid*, *tanh*, *relu*, *softmax* and *linear*. We will make a review of them now and try to understand how we choose between them.

We will see later that they can all be called when instantiating a layer with the parameter `activation` (ex: `tf.keras.layers.Dense(activation='sigmoid')`).

2.3.2.1. Sign activation function

Let's imagine you wish to train a perceptron to detect whether a point is above or below a line, you might want the output to be a +1 or -1 label. You could use the **sign activation function** to help the perceptron make the decision:

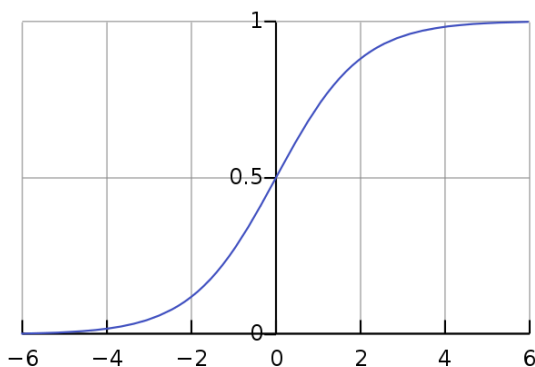
- If weighted sum is positive, return +1
- If weighted sum is negative, return -1



2.3.2.2 Sigmoid

Sigmoid is a historic activation function, but not the most efficient in general. It is now used mainly in **last layer of a binary classification**. Call it with `'sigmoid'`.

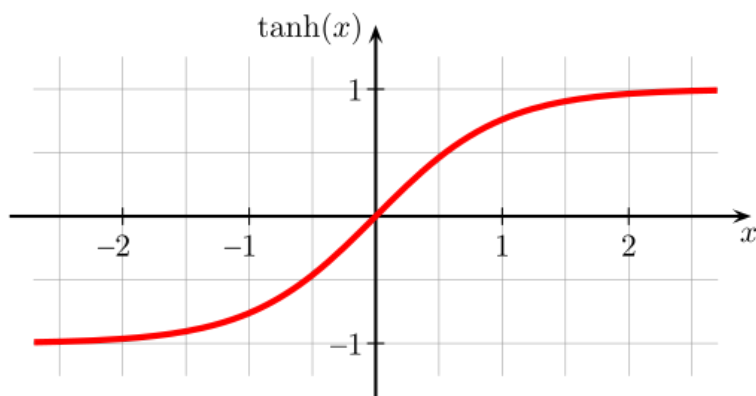
$$\text{sigmoid}(x) = \frac{1}{1+e^{-x}}$$



2.3.2.3. Hyperbolic tangent (tanh)

Hyperbolic tangent (or tanh) is quite similar to sigmoid function. Its shape is pretty close but it ranges between -1 and 1, while sigmoid ranges between 0 and 1. Therefore, it has a symmetry that sigmoid does not have. This is **not the most frequently used** activation function at first approach anymore. Call it with `'tanh'`.

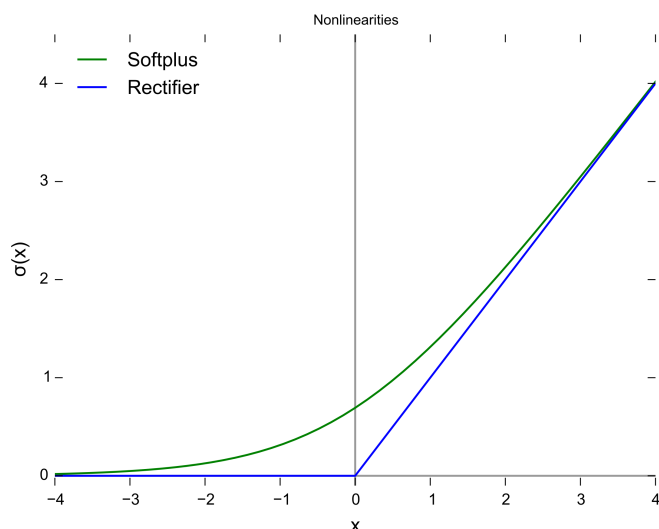
$$\text{tanh}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



2.3.2.4. Rectified linear unit (ReLU)

ReLU is currently **the most popular activation function in deep learning**. Even though it looks odd at first sight, it has the advantage of **avoiding the problem of vanishing gradient** (we'll delve into it later). Call it with `'relu'`.

$$relu(x) = \max(0, x)$$



2.3.2.5. Softmax

Softmax function is a special activation function, used in only one specific case: the **last layer of a multiclass classification**. Call it with `'softmax'`.

2.3.2.6. Linear

Linear function is just the identity, and is used mainly in the specific case of **last layer of a regression** (you may also use a ReLU if you know in advance your regression can't have any negative result). Call it with `'linear'`.

2.4. Backpropagation (optional)

Backward propagation is more complicated than forward propagation. Backpropagation is used to **apply the gradient descent algorithm to all weights of the neural networks**. In a word, this is the **optimization phase** of the neural network.

The concept is to backpropagate your error between the prediction of the output layer and the target value Y , and finally to update the values of the weights W (as well as the bias b) of each layer.

⚠ It's **not necessary to fully understand the way backpropagation works**, the most important is to remember that this is how neural networks are trained.

📖 **Resources:** we won't delve into the math of backward propagation, but if you're motivated you can read this [article](https://towardsdatascience.com/backpropagation-super-simplified-2b8631c0683d) [_\(https://towardsdatascience.com/backpropagation-super-simplified-2b8631c0683d\)](https://towardsdatascience.com/backpropagation-super-simplified-2b8631c0683d) and watch this

[video](https://www.youtube.com/watch?v=1N837i4s1T8&index=9&list=PL6Xpj9I5qXYEcOhn7TqghAJ6NAPrNmUBH) [_\(https://www.youtube.com/watch?v=1N837i4s1T8&index=9&list=PL6Xpj9I5qXYEcOhn7TqghAJ6NAPrNmUBH\)](https://www.youtube.com/watch?v=1N837i4s1T8&index=9&list=PL6Xpj9I5qXYEcOhn7TqghAJ6NAPrNmUBH)



[_\(https://www.youtube.com/watch?v=1N837i4s1T8&index=9&list=PL6Xpj9I5qXYEcOhn7TqghAJ6NAPrNmUBH\)](https://www.youtube.com/watch?v=1N837i4s1T8&index=9&list=PL6Xpj9I5qXYEcOhn7TqghAJ6NAPrNmUBH)

3. Implementation with TensorFlow

3.1. Introduction to TensorFlow



TensorFlow is an **open-source library**, first released in november 2015 by a Google subsidiary. It allows, among other things, to compute neural networks very efficiently. TensorFlow is **programmed in C++ high performance** and offers a **Python API**.



To install TensorFlow, use the following command: `pip install tensorflow`

Then to use it in our Python code, just import it this way:

```
import tensorflow as tf
```

Even though TensorFlow is very powerful and allows you to do pretty much anything you want in terms of neural network architecture, it is hard to handle for beginners. That's why we will work with Keras.

3.2. Introduction to Keras



Keras is an open-source library, initially developed by François Chollet, that allows to implement Deep Learning standard algorithms very easily, with a highly efficient backend (e.g. TensorFlow). Keras was historically separated from TensorFlow, but is now maintained by Google and is **an official API of TensorFlow 2.0**.

There are two ways to compute a neural network using Keras: the sequential or the API way. We will study the sequential way here, but the API way is not very different, no worries.

3.3. Building our first model

3.3.1. Define the layers

We will build a neural network with 2 hidden layers of 3 units, and an output layer of 1 unit. We will put all of it into a function called `model`, in order to reuse it easily.

```
# Import tensorflow
import tensorflow as tf

# Define a function
def model(input_dim):
    # We create a so called Sequential model
    model = tf.keras.models.Sequential()

    # Add the first "Dense" layer of 3 units, and give the input dimension (here 5)
    model.add(tf.keras.layers.Dense(3, input_dim=input_dim, activation='sigmoid'))
```

```
# Add the second "Dense" layer of 3 units
# This time the input dimension is not needed anymore: it is known from the previous layer
model.add(tf.keras.layers.Dense(3, activation='sigmoid'))

# Add finally the output layer with one unit: the predicted result
model.add(tf.keras.layers.Dense(1, activation='sigmoid'))

# return the created model
return model
```

You will soon get familiar with the classes of TensorFlow/Keras. Here we are using the following:

- `Sequential()`: this will contain all the layers of our neural network
- `Dense(units, activation=None)`: this is a classical multi layer perceptron, taking as input the number of units (i.e. Neurons) and the activation function

NB: the first layer of a `Sequential` model always takes as input the `input_dim`: this is the `shape` of your features! Otherwise TensorFlow does not know how many input features you have!

Finally, one can review a model with the `summary()` method:

```
my_model = model(input_dim=5)
my_model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 3)	18
dense_1 (Dense)	(None, 3)	12
dense_2 (Dense)	(None, 1)	4
Total params: 34		
Trainable params: 34		
Non-trainable params: 0		

3.3.2. Compile your model

Since TensorFlow does not fully work on Python, a compilation step is necessary. To perform this step, you need to provide several parameters :

- The optimizer is the optimization algorithm: you already know the gradient descent, you can call it by using `optimizer='SGD'`
- The loss function: for binary classification use `loss='binary_crossentropy'`, for regression use `loss='mean_squared_error'`

- You can also play with the metrics to display in real time, for example to display the accuracy, add the parameter `metrics=['accuracy']`

For more information, the documentation is [here](https://keras.io) `(https://keras.io)`.

```
# Then you want to compile your model
# here with Gradient Descent and binary cross entropy (for binary classification)
my_model = model(input_dim=30)
my_model.compile(optimizer='SGD', loss='binary_crossentropy', metrics=['accuracy'])
```

3.3.3. Train your model

Finally, you want to fit your model, just like we did in `scikit-learn` ! For this example, we will use the *Wisconsin Breast Cancer* dataset from `scikit-learn` and then fit our model.

To fit our model, some parameters have must be given:

- `x`: the input features
- `y`: the labels or target values
- `epochs`: the number of times you iterate over all the input samples
- `batch_size`: the number of samples used before updating the parameters of the model (we will speak more about it tomorrow)

```
# First import and load the data
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_breast_cancer
from sklearn.preprocessing import StandardScaler

X, y = load_breast_cancer(return_X_y=True)

# Split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)

# Scale the input features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Train the model, iterating on the data in batches of 32 samples
my_model.fit(x=X_train, y=y_train, validation_data=(X_test, y_test), epochs=10, batch_size=32)
```

```
Epoch 1/10
15/15 [=====] - 0s 31ms/step - loss: 0.7165 - accuracy: 0.3626 - val_loss:
0.7006 - val_accuracy: 0.4123
Epoch 2/10
15/15 [=====] - 0s 6ms/step - loss: 0.7069 - accuracy: 0.3736 - val_loss:
0.6941 - val_accuracy: 0.4298
Epoch 3/10
15/15 [=====] - 0s 7ms/step - loss: 0.6986 - accuracy: 0.4110 - val_loss:
0.6880 - val_accuracy: 0.4825
```

```

Epoch 4/10
15/15 [=====] - 0s 13ms/step - loss: 0.6908 - accuracy: 0.4945 - val_loss:
0.6829 - val_accuracy: 0.6053
Epoch 5/10
15/15 [=====] - 0s 9ms/step - loss: 0.6841 - accuracy: 0.5846 - val_loss:
0.6788 - val_accuracy: 0.6842
Epoch 6/10
15/15 [=====] - 0s 6ms/step - loss: 0.6786 - accuracy: 0.6505 - val_loss:
0.6751 - val_accuracy: 0.7193
Epoch 7/10
15/15 [=====] - 0s 5ms/step - loss: 0.6736 - accuracy: 0.6945 - val_loss:
0.6713 - val_accuracy: 0.7018
Epoch 8/10
15/15 [=====] - 0s 5ms/step - loss: 0.6683 - accuracy: 0.7275 - val_loss:
0.6685 - val_accuracy: 0.7105
Epoch 9/10
15/15 [=====] - 0s 5ms/step - loss: 0.6643 - accuracy: 0.7275 - val_loss:
0.6657 - val_accuracy: 0.6667
Epoch 10/10
15/15 [=====] - 0s 5ms/step - loss: 0.6602 - accuracy: 0.7099 - val_loss:
0.6632 - val_accuracy: 0.6491

```

At the end, you can just predict values on a new sample using the usual *predict* method.

3.3.4. Predict and evaluate

To predict, do not change your habits: use the function `.predict(X)`:

```

# Predict for some values
my_model.predict(X_test[:5])

```

```

array([[0.5057793 ],
       [0.5717841 ],
       [0.5540764 ],
       [0.555383  ],
       [0.56602347]], dtype=float32)

```

To evaluate, you can of course use scikit-learn's metrics as usual. But warning, the keras `.predict()` method does not return classes, but probabilities!

You can also use `.evaluate(X, y)`: This function returns a list with the loss and the metrics.

```

# Evaluate your model
loss, accuracy = my_model.evaluate(X_test, y_test, verbose=0)
print('loss is:', loss)
print('accuracy is:', accuracy)

```

```

loss is: 0.6631731390953064
accuracy is: 0.6491228342056274

```

🎉 Congratulations ! We've just built our first neural network 🎉
