

## Practical session : Introduction to multi-layer perceptron and classification

We consider the example of a classification problem with a fully connected neuron network.

### 1 Example 2D points datasets

We will first consider the example of binary classification of 2D data points using the datasets illustrated in Figure 1 below.

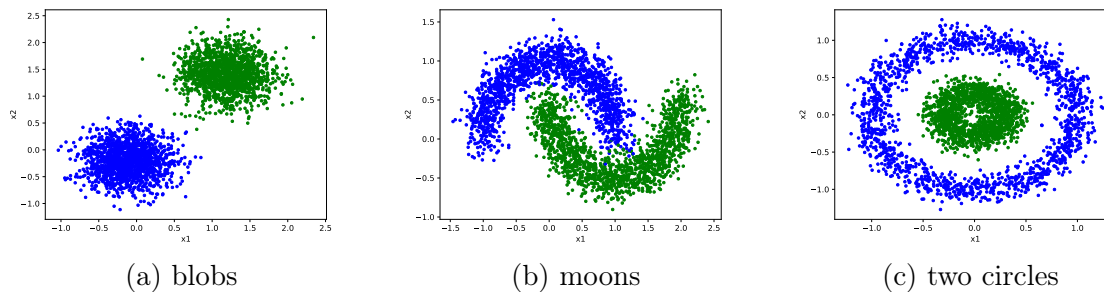


FIGURE 1 – 2D points datasets

The 2 different classes are represented visually with 2 different colors.

- Dataset (a) is linearly separable
- Dataset (b) called "moons" is not linearly separable
- Dataset (c) is made with two sets of point forming a circle inside another one.

### 2 Configuration

Using Anaconda navigator, create a keras environment (or add **keras** package to the environment you used in previous sessions)

### 3 Classification using single layer perceptron

To illustrate the use of Keras, we consider a simple 2D points binary classification problem (see figure 1).

For this example we will define linear classifier using a single neuron model.

- Download the example script : `1_linear_perceptron.py`. Test it using the dataset "blob", and then on the dataset "moon". Keras syntax is described in the appendix of this document.
- From this example, implement a perceptron with one hidden layer of  $n$  neurons. You will test it on the datasets "blob" and "moon", with different  $n$  values.
- Observe the influence of the iteration number, the batch size and the number  $n$  of neurons in the hidden layer on the performances.

## Appendix : Introduction to Keras

This section gives a fast introduction to implementing neural networks with Keras.

Keras is a Python library which provides a high level API (programming interface) for implementing neural networks. It can be used on top of other libraries such as TensorFlow, CNTK ou Theano. Keras allows fast and easy neural network programming.

In Keras the reference data structure is the model. There are two main model construction modes in Keras :

- Sequential modeling, in which a model is defined as a stack of layers
- Functional modeling, which allows to build arbitrary graphs of layers for more complex models

Here we will only consider Sequential models.

### Sequential models

To build a sequential model, you will import the `keras` module :

```
from keras.models import Sequential
```

The module `keras.layers` contains all the standard layers of neural networks (dense layers, convolutional layers, pooling layers, activations, dropout, ...)

```
from keras.layers import Dense, Activation
```

**Model definition :** Building the model is done by adding successive layers. For instance, a model with :

- one input layer of 100 neurons
- two hidden layers of 128 neurons with ReLU activation
- one output layer of 10 neurons with Softmax activation

will be written as :

```
model = Sequential()
model.add(Dense(units=128, activation='relu', input_dim=100))
model.add(Dense(units=128, activation='relu'))
model.add(Dense(units=10, activation='softmax'))
```

The `Dense` layers used in this example correspond to fully connected layers. We will not review all layers here, but they are referenced in the online documentation. In practical sessions we will mainly use dense layers, convolutional layers and pooling layers. An example of convolutional layer is given here :

```
# 8 convolutions of size 3x3,
# with stride 1 and zero padding.
model.add(Conv2D(8, (3, 3), padding='same', strides=(1,1), input_shape=(28,28,1)))
# Activation ReLU
model.add(Activation('relu'))
# Max pooling of size 2 by 2, with stride 2
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2,2)))
```

Model definition is also the step where the weight initialisation method is defined. By default, the weight initialization is Uniform Xavier : weights are drawn randomly from a uniform distribution where the variance depends on the number of neurons of input and output of the layer. This is usually a good default choice, but if you want to change it, for example using to a Normal Xavier initialization you can use :

```
model.add(Dense(128, kernel_initializer='glorot_normal', bias_initializer='zeros'))
```

Biases usually stay initialized at zero.

Once the model built, it can be displayed using :

```
print(model.summary())
```

We can also visualize graphically the network (here registered in a .png file) :

```
import keras.utils.plot_model
plot_model(model, to_file='model.png', show_shapes=True, show_layer_names=True)
```

**Training Configuration :** Once the architecture definition is done, the training strategy is defined :

- the loss function) to minimize ( mean squared error, cross entropy,...)
- the optimizer (SGD, RMSProp, AdaGrad, Adam...)
- the metric(s) computed to evaluate the performances. Metrics can be chosen from loss functions or other criteria such as accuracy, or even hand defined.

```
model.compile(loss='binary_crossentropy', optimizer='sgd', metrics=['accuracy'])
```

The choice of loss function and output activation function depend on the problem to solve (voir figure 3)

Problem	Output	Output activation	Loss function
Regression	Real	Linear	MSE : $\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$
Classification	Binary	Sigmoid : $\sigma(z) = \frac{1}{1+e^{-z}}$	Binary cross entropy : $-(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y}))$
Classification	One label, several classes	Softmax : $f(z) = \frac{\exp(z)}{\sum_i \exp(z_i)}$	cross entropy : $-\sum_i^M y_i \log(\hat{y}_i)$
Classification	Several labels, several classes	Sigmoid : $\sigma(z) = \frac{1}{1+e^{-z}}$	Binary cross entropy : $-\sum_i^M (y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i))$

FIGURE 2 – Typical choices of loss function and output activation function in classification and regression problems

We can then train the model by defining the way data will be presented :

- One epoch correspond to one pass through all the data
- A batch is a subset of the dataset which will be used in one iteration of the optimization process. Thus the batch size will define the number of examples which will be used at each weight update step.

```
model.fit(x_train, y_train, epochs=3, batch_size=32)
```

In the script below, `x_train` and `y_train` are arrays of training data and labels.

**Performances Evaluation :** During training we observe the loss value and metrics values through the epochs (depending on the `verbose` mode) Once the model is trained, we can evaluate its performance on the test set : Une fois le modèle entraîné, on pourra évaluer sa performance sur les données de la base de test :

```
measures_perf_test = model.evaluate(x_test, y_test)
print('test results: ', measures_perf_test)
```

**Using a trained network :** To make predictions using a trained model :

```
predictions = model.predict(x_new)
```

**Save :** Saving weights :

```
model.save_weights("./trained_models/my_model")
```