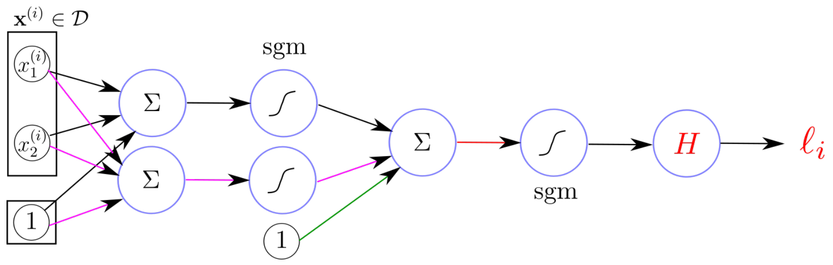
ML3 - Practical Class n°2

In this second practical class, we will implement a one-hidden-layer MLP to illustrate how the backpropagation algorithm works in practice. Our target MLP architecture is the same as the one studied in class :



Preliminaries

Instructions :

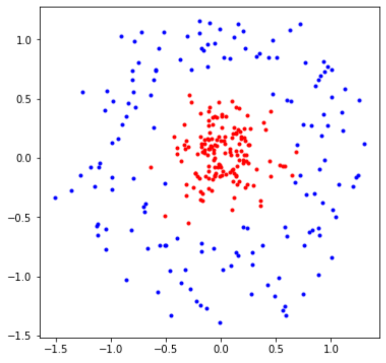
1. We begin with a number of imports that will rest with us throughout this mini-project.

```
import numpy as np
from sklearn.datasets import make_circles
import matplotlib.pyplot as plt
import copy
```

2. Second, we will generate a non-trivial 2D dataset using the `make_circles` function :

```
X,y = make_circles(n_samples = 300, noise=0.2, factor=0.1)
n,d = X.shape
plt.figure(1,figsize=[6,6])
plt.plot(X[:,0][np.where(y==1)],X[:,1][np.where(y==1)],'.c')
plt.plot(X[:,0][np.where(y==0)],X[:,1][np.where(y==0)],'.m')
```

You understand that this dataset has $n = 300$ training examples and corresponds to a binary classification problem, i.e. $\mathcal{Y} = \{0; 1\}$. The obtained figure displays the dataset and should look like this :



3. Also, please copy and paste the following functions that will be instrumental at some point :

```
def sigmoid(x):
    res = 1 / (1 + np.exp(-x))
    return res
```

The function is the sigmoid function which we will use as activation function in the sequel.

Exercise n°1 : Network creation

In this exercise, we will implement python classes allowing to instantiate neural units (with the sigmoid function as activation function), inputs and the network itself.

1. We shall start with a python class that will dispatch a training vector **x** using several "input units" in order to stick to the presentation of backprop from the lecture. Indeed, in what was discussed in class, each entry of a vector **x** is regarded as a unit whose job is just to provide one entry of **x** to the neural units that are connected to it.
   The bare structure of this class is

```
class InputUnit:
    def __init__(self,data):
        self.data = data #one column of matrix X
        self.n = data.shape[0] #dataset size
        self.k = 0 #layer number
        self.z = 0 #unit output
```

   For the time being, an instance of this class is just meant to store in the attribute `data` one raw feature of the inputs, i.e. a column of the array `X`. Create $d = 2$ instances of this class for each column of `X`. The other attributes are either self-explanatory or will be useful later.
2. Next, we will create a python class for neural units. Each unit will have to compute an output given a set of incoming numbers but this job will be implemented in the next exercise. For now, we only ask a unit to store its ancestors and its children in the network graph :

```
class NeuralUnit:
    #Constructor
    def __init__(self,k,u):
        self.u = u #unit number
        self.preceding = [] #list of preceding neurons
        self.npr = 0 #length of list preceding
        self.following = [] #list of following neurons
        self.nfo = 0 #length of list following
        self.k = k #layer number
        self.w = 0 #unit weights
        self.b = 0 #unit intercept
        self.z = 0 #unit output

    def reset_params(self):
        self.w = np.random.randn(self.npr)
        self.b = np.random.randn()
```

   To instantiate a unit, you see that we need to specify its layer index `k` as well as its personal index `u` in this layer. The method `reset_params` allows to initialize the trainable parameters of the unit. When we will launch the training loop, the goal will be to converge to appropriate values of these parameters.
   Observe that there are also two attributes which are lists and are meant to store pointers to the units from which incoming data arrives (`preceding`) and pointers to units to which the activation of the unit is sent (`following`). The sizes of these lists are also stored.
   Create 3 instances of this class corresponding to the targeted network (2 layers made of respectively 2 and 1 unit).

3. To finalize the network, we also need a python class for the loss unit. In the next exercise, the job of this class will be to compute the cross-entropy loss ℓ_i (see slides for its definition) for each input $\mathbf{x}^{(i)}$. For now, we just want this class to complete the network architecture.

```
class Loss:
    #Constructor
    def __init__(self,y,k):
        self.preceding = [] #list of preceding neurons
        self.npr = 0 #length of list preceding
        self.y = y #array of class labels of the training data
        self.k = k #layer index
```

   To instantiate a loss, you see that we need to specify its layer index `k`. We also add the array `y` of class labels because we need it as part of cross-entropy computations. This class also memorizes the units from which incoming data arrives in the `preceding` attribute as well as the length of this list.

   Create an instance of this class.

4. **Main job 1 :** in the `InputUnit` and `NeuralUnit` classes, add a method `plug` that allows to fill correctly the lists `preceding` of the neural units and the loss as well as the lists `following` of the neural units.

5. Now it is time to organize things in a more convenient way and to define an MLP python class that will be a container of the previously defined classes. The code skeleton of this class is

```
class MLP:
    #Constructor
    def __init__(self,X,y,archi):
        self.archi = archi
        self.X = X
        self.n = X.shape[0]
        self.y = y
        self.K = len(archi) #number of layers (including input layer but omitting loss layer)
        #creating network
        net = []
        #.... MISSING CODE ....
        self.net = net
```

   Basically, an instance of this class is an MLP network. It stores pointers to training examples in `X` and training class labels `y` for convenience but most importantly it connects instances of `InputUnit`, `NeuralUnit` and `Loss` in a meaningful way entirely described by parameter `archi`. This parameter is a list of integers. The size of the list is the number of layers of the network (including the input layer but without the loss layer) so in our case, this length is 3. Each of the integers is the number of units of the layer. In our case the list is thus $[2, 2, 1]$.

   Main job 2 : fill the gap in the constructor of the MLP class. The missing code is meant to progressively append layers to the empty list `net`. To do that, you need to loop on layers and for each layer to loop on units. Inside this double loop, depending on the layer index, you either instantiate an `InputUnit`, a `NeuralUnit` or a `Loss`. When necessary, invoke the `plug` method of the newly instantiated object.

   To complete the constructor, call the `reset_params` method of each neural unit. Instantiate a network corresponding to the target architecture and check unit connections as well as parameter initializations.

Exercise n°2 : Forward pass

In this exercise, we will add methods `forward` to each of the previously defined python classes in the previous exercise. This will allow us to map inputs $\mathbf{x}^{(i)}$ to ℓ_i.

1. Add a method `forward` to the `InputUnit` class. It takes only one argument : `i` the index of the example for which we wish to perform a forward pass of the network. Return the corresponding entry of the array `data` and also set `self.z` to this returned value.
2. Add a method `forward` to the `NeuralUnit` class. It takes only one argument : `i` the index of the example for which we wish to perform a forward pass of the network. Return the unit output. If the incoming

data for this unit is stored in vector $\mathbf{z}_{in}$, the unit output is sgm $\left(\mathbf{w}^T \cdot \mathbf{z}_{in} + b\right)$. To obtain the vector $\mathbf{z}_{in}$, loop on the members of the list `self.preceding` and invoke the `forward` method of each unit in this list.

Also set `self.z` to the returned value.

3. Add a method `forward` to the `Loss` class. It takes only one argument : `i` the index of the example for which we wish to perform a forward pass of the network. Return loss ℓ_i. Normally, this loss unit has only one neural unit in `self.preceding`. By calling the `forward` method of neural unit, you can access a scalar number z_{in} which is regarded as the probability that the network input $\mathbf{x}$ has class label $y = 1$. The loss is given by

$$\ell_i = \begin{cases} -\log\left(1 - z_{in}\right) & \text{if } y^{(i)} = 0 \\ -\log z_{in} & \text{if } y^{(i)} = 1 \end{cases}.$$

4. Also add a method `forward` to the `MLP` class. It takes only one argument : `i` the index of the example for which we wish to perform a forward pass of the network. It consists only in calling the `forward` method of the loss unit of the network and returning the value provided by it.

   You can check the methods are correctly programmed : for instance, if $y^{(i)} = 0$ and the output of the second layer is close to 1, then the returned loss should be high.

Exercise n°3 : Backprop

In this exercise, we will add methods `backprop` to each of the previously defined python classes in the previous exercise. This will allow us to compute derivatives $\frac{\partial \ell_i}{\partial w_{v,u}^{(k)}}$ and $\frac{\partial \ell_i}{\partial b_u^{(k)}}$.

Nota Bene : Keep in mind that a call to the `backprop` methods that we will implement comes always after a call to the `forward` method of the network. In particular, this means that each unit knows its current output which is stored in their `z` attribute.

1. Add a method `backprop` to the `Loss` class. It takes only one argument : `i` the index of the example for which we wish to compute gradients. The function returns nothing but stores the computed derivatives in a new class attribute. For the loss, recall that we solely need to compute $\delta = \frac{\partial \ell_i}{z_{in}}$ where z_{in} is the output of the previous layer. We have

$$\frac{\partial \ell_i}{z_{in}} = \begin{cases} \frac{1}{1-z_{in}} & \text{if } y^{(i)} = 0 \\ -\frac{1}{z_{in}} & \text{if } y^{(i)} = 1 \end{cases}.$$

   This derivative is a scalar, yet it will be later more convenient to have it stored in a `numpy array` therefore initialize a new class attribute as `self.delta = np.zeros((1,))` and save the derivative in `self.delta[0]`.

2. Add a method `backprop` to the `NeuralUnit` class. It takes two arguments : `i` the index of the example for which we wish to compute gradients and an array `delta` which contains the delta derivatives coming from the following layer. The function returns nothing but stores the computed derivatives in new class attributes. For neural units there are two types of derivatives to compute, i.e. the delta ones and the parameter ones :

   — for the delta ones : we actually need to compute

$$\delta_v^{(k)} := \frac{\partial \ell_i}{\partial z_{in,v}} = \sum_u \frac{dz_u}{dz_{in,v}} \times \delta_u^{(k+1)}, \forall v \text{ (unit index in the preceding layer)}, \qquad (1)$$

   where u is the unit index in the current layer. This computation will be dispatched on each unit of the layer. So the job of one unit is just to compute

$$\frac{dz_u}{dz_{in,v}} \times \delta_u^{(k+1)}, \forall v \text{ (unit index in the preceding layer)}.$$

   where u is the unit index stored in `self.u`.
   Start by initializing a new class attribute as `self.delta = np.zeros(self.w.shape)`. Then, loop on preceding units. For each preceding unit v, compute the above formula where z_u is the current

output of unit u which is stored in `self.z`. The value of $\delta_u^{(k+1)}$ is provided by `deltas[self.u]`. Furthermore, when the activation function is the sigmoid function, we have

$$\frac{dz_u}{dz_{in,v}} = z_u\,(1 - z_u)\,w_{v,u}^{(k)},$$

where $w_{v,u}^{(k)}$ is the v^{th} parameter of the neural unit stored in `self.w[v]`.
Store the derivative in `self.delta[v]`.

— for parameter ones : start by initializing two class attributes as `self.w_grad = np.zeros(self.w.shape)` and `self.b_grad = 0`. Then, loop on preceding units. For each preceding unit v, we need to compute

$$\frac{\partial \ell_i}{\partial w_{v,u}^{(k)}} = \frac{dz_u}{dw_{v,u}^{(k)}} \times \delta_u^{(k+1)}.$$

Again, when the activation function is the sigmoid function, we have

$$\frac{dz_u}{dw_{v,u}^{(k)}} = z_u\,(1 - z_u)\,z_{in,v},$$

where $z_{in,v}$ is the output of the preceding unit number v. This value is stored in `self.preceding[v].z`. Store the derivative in `self.w_grad[v]`. Do not forget to also compute $\frac{\partial \ell_i}{\partial b_u^{(k)}}$ which is given by

$$\frac{dz}{db_u^{(k)}} \times \delta_u^{(k+1)} = z_u\,(1 - z_u) \times \delta_u^{(k+1)}.$$

Store the derivative in `self.b_grad`

3. To finish this exercise, you also need to add a method `backprop` to the `MLP` class. It takes only one argument : `i` the index of the example for which we wish to compute gradients. This function will iterate on layers, but remember that we go backward, so we start from the final (loss) layer and decrement the layer index k. The loop stops at $k = 1$ (first hidden layer) as there are no gradient to compute in the input layer.

   At each iteration k, loop on the units of the layer. For each unit, invoke its method `backprop` :

   — In the first iteration (loss layer), call `self.net[k][0].backprop(i)`. We need to save the delta derivatives to prepare things for the next iteration. Add the following line of code : `deltas = self.net[k][0].delta`.

   — In the following iterations (neural layers) :

      — Initialize a numpy array as `deltas_new = np.zeros((self.net[k][0].npr,))`.

      — Loop on units of the current layer. For each unit u, call `self.net[k][u].backprop(i,deltas)`. Accumulate delta derivatives as `deltas_new += self.net[k][u].delta`. This accumulation allows to perform the sum in equation (1).

      — After the loop, update the delta derivatives as `deltas = deltas_new`.

4. We are now ready to launch a backprop step to obtain all necessary derivatives for a gradient descent step. You can check that the computed derivatives are correct by comparing them to their numerical approximations. One such approximation can be obtained for instance as follows :

```
i = 0
archi = [d,10,1]
mlp = MLP(X,y,archi)
mlp.forward(i)
mlp.backprop(i)
epsi=1e-3
mlp2 = copy.deepcopy(mlp)
mlp2.net[1][0].w[0] = mlp.net[1][0].w[0] + epsi
print("numerical␣derivative␣is:",(mlp2.forward(i) - mlp.forward(i))/epsi)
print("computed␣derivative␣is:",mlp.net[1][0].w_grad[0])
```

In the above code, it may be necessary to adjust the coefficient `epsi`.

Exercise n°4 : SGD training of the network

In this final exercise, we will train the network on our data and evaluate its performances on newly sampled data.

1. As part of SGD, each parameter is updated based on the parameter derivative computed using a given input $\mathbf{x}^{(i)}$. Add a method `update` to the `MLP` class. It takes one argument : `eta` the learning rate. The function returns nothing but performs the parameter update for each neural unit in `self.net` according to

$$\begin{aligned} \mathbf{w} &= \mathbf{w} - \eta \frac{\partial \ell_i}{\partial \mathbf{w}}, \\ b &= b - \eta \frac{\partial \ell_i}{\partial b}. \end{aligned} \tag{2}$$

2. Add the following method to the `MLP` class :

```
    def train(self, epochs,eta):
        for epoch in range(epochs):
            for i in range(self.n):
                self.forward(i)
                self.backprop(i)
                self.update(eta)
```

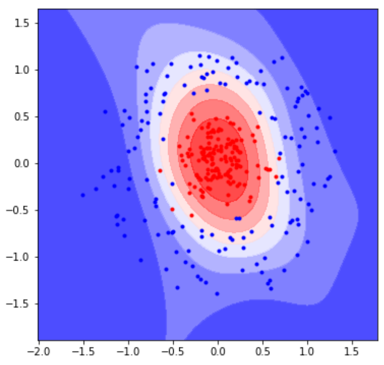
   Run this method by calling `mlp.train(50,0.01)`

3. Add a method `predict` to the `MLP` class. It takes one argument : `i` the index of the example for which we want prediction. The function returns a scalar which is the output of the network (without the loss layer) and is interpreted as the probability that the class label of $\mathbf{x}^{(i)}$ is 1.

4. Evaluate the trained model using the following code

```
h = .02
x1_min, x1_max = X[:, 0].min() - .5, X[:, 0].max() + .5
x2_min, x2_max = X[:, 1].min() - .5, X[:, 1].max() + .5
x11, x22 = np.meshgrid(np.arange(x1_min, x1_max, h),
                       np.arange(x2_min, x2_max, h))
X_disp = np.c_[x11.ravel(), x22.ravel()]
n_disp = X_disp.shape[0]
Z = []
for u in range(mlp.archi[0]):
    mlp.net[0][u].data = X_disp[:,u]
for i in range(n_disp):
    Z.append(mlp.predict(i))
for u in range(mlp.archi[0]):
    mlp.net[0][u].data = X[:,u]
Z = np.array(Z)
Z = Z.reshape(x11.shape)
plt.figure(2,figsize=[6,6])
plt.plot(X[:,0][np.where(y==1)],X[:,1][np.where(y==1)],'.r')
plt.plot(X[:,0][np.where(y==0)],X[:,1][np.where(y==0)],'.b')
plt.contourf(x11, x22, Z, cmap=plt.cm.bwr, alpha=.8)
```

   If everything went alright, you should obtain a figure close to the following one :



You may re-run the code to see the influence of the weight initialization. Observe that the convergence is not monitored so we cannot be sure that the chosen number of epochs is always correct.