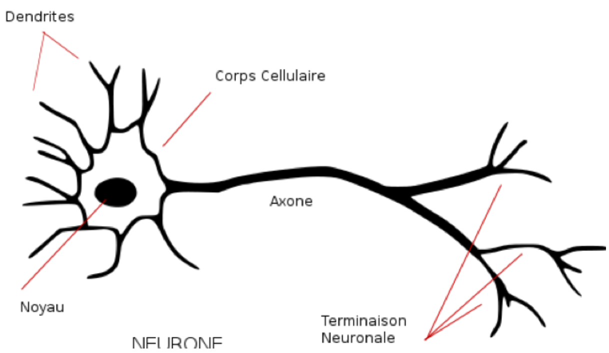
Les réseaux de neurones

Les neurones
"Un neurone, ou une cellule nerveuse, est une cellule excitable constituant l'unité fonctionnelle de base du système nerveux. Les neurones assurent la transmission d'un signal bioélectrique appelé influx nerveux. Ils ont deux propriétés physiologiques : l'excitabilité, c'est-à-dire la capacité de répondre aux stimulations et de convertir celles-ci en impulsions nerveuses, et la conductivité, c'est-à-dire la capacité de transmettre les impulsions." (Wikipedia)

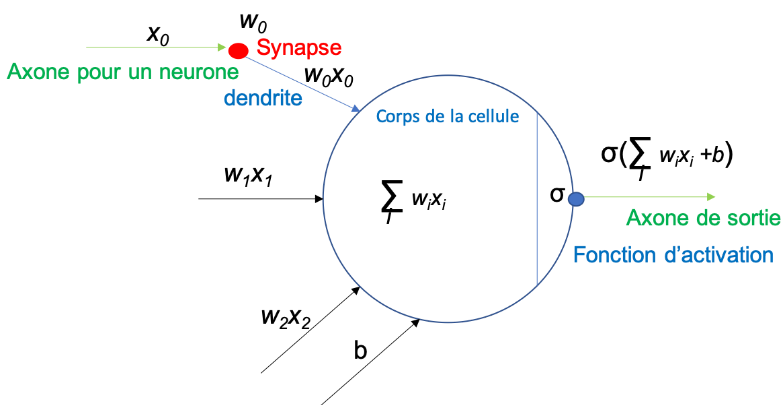
La structure d'un neurone (source : https://fr.wikipedia.org/wiki/Fichier:Neurone_-_commenté.svg (https://fr.wikipedia.org/wiki/Fichier:Neurone_-_commenté.svg)) :



Le fonctionnement est le suivant : tout d'abord, les dendrites reçoivent l'influx nerveux d'autres neurones. Le neurone évalue alors l'ensemble de la stimulation reçue. Si celle-ci est suffisante, il est excité : il transmet un signal (0/1) le long de l 'axone et l'excitation est propagée jusqu'aux autres neurones qui y sont connectés via les synapses.

"Un réseau de neurones artificiels, ou réseau neuronal artificiel, est un système dont la conception est à l'origine schématiquement inspirée du fonctionnement des neurones biologiques" (Wikipedia)

La structure d'un neurone artificiel :



Comme nous le constatons, un neurone artificiel est assez similaire à un neurone. Il comprend un ensemble d'entrées (synapses) auxquelles un ensemble de poids sont ajoutés (dans le notebook sur la descente de gradient, ces poids correspondent aux paramètres qu'il fallait trouver pour les fonctions linéaires - θ). Il possède également une entrée particulière appelée *biais*. Une fonction additive (combinaison linéaire) calcule la somme pondérée des entrées : $\sum_i w_i x_i$. La sortie du noeud est déterminée en appliquant une fonction de transfert non-linéaire, $\sigma(\sum_i w_i x_i + b)$.

Lorsque la régression logistique ne fonctionne plus

Dans le notebook sur la descente de gradient, nous avons terminé par la régression logistique et avons vu qu'il était possible d'afficher les limites de décision (une droite) pour classer les iris. Considérons, à présent, la figure suivante :

In [1]:

```
from sklearn.datasets import make_moons
import matplotlib.pyplot as plt
import matplotlib
from matplotlib.colors import ListedColormap
import numpy as np
matplotlib.rcParams['figure.figsize'] = (6.0, 6.0) # pour avoir des figures d
np.random.seed(0)
X, y = make_moons(n_samples=1000, noise=0.1)
cm_bright = ListedColormap(['#FF0000', '#0000FF'])
plt.scatter(X[:,0], X[:,1], s=40, c=y, cmap=cm_bright)#cmap=plt.cm.PiYG)
```

Out[1]:

```
<matplotlib.collections.PathCollection at 0x118c77860>
```

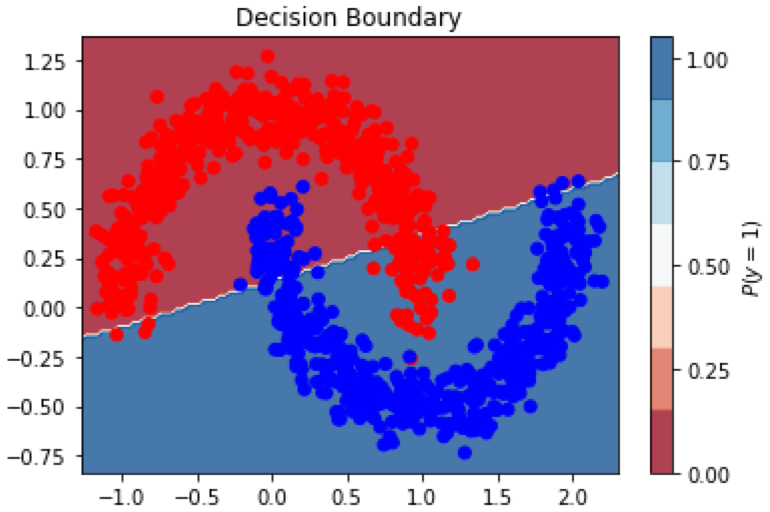
In [2]:

```
#from :
#https://github.com/ardendertat/Applied-Deep-Learning-with-Keras/blob/master/

def plot_decision_boundary(func, X, y):
    amin, bmin = X.min(axis=0) - 0.1
    amax, bmax = X.max(axis=0) + 0.1
    hticks = np.linspace(amin, amax, 101)
    vticks = np.linspace(bmin, bmax, 101)
    aa, bb = np.meshgrid(hticks, vticks)
    ab = np.c_[aa.ravel(), bb.ravel()]
    c = func(ab)
    cc = c.reshape(aa.shape)
    cm = plt.cm.RdBu
    cm_bright = ListedColormap(['#FF0000', '#0000FF'])
    fig, ax = plt.subplots()
    contour = plt.contourf(aa, bb, cc, cmap=cm, alpha=0.8)
    ax_c = fig.colorbar(contour)
    ax_c.set_label("$P(y = 1)$")
    ax_c.set_ticks([0, 0.25, 0.5, 0.75, 1])

    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cm_bright)
    plt.xlim(amin, amax)
    plt.ylim(bmin, bmax)
    plt.title("Decision Boundary")
```

Appliquons, à présent, la logistic regression de sickit learn pour afficher la frontière de decision :

In [3]:

```
from sklearn import linear_model
clf = linear_model.LogisticRegression(solver='lbfgs')
clf.fit(X, y)
#Plot the decision boundary
plot_decision_boundary(lambda x: clf.predict(x),X,y)
```



Comme nous pouvons le constater les deux ensembles ne peuvent pas être séparés linéairement. Nous avons besoin de quelque chose de plus sophistiqué : les réseaux de neurones.

Les réseaux de neurones

Les réseaux de neurones se composent des éléments suivants :

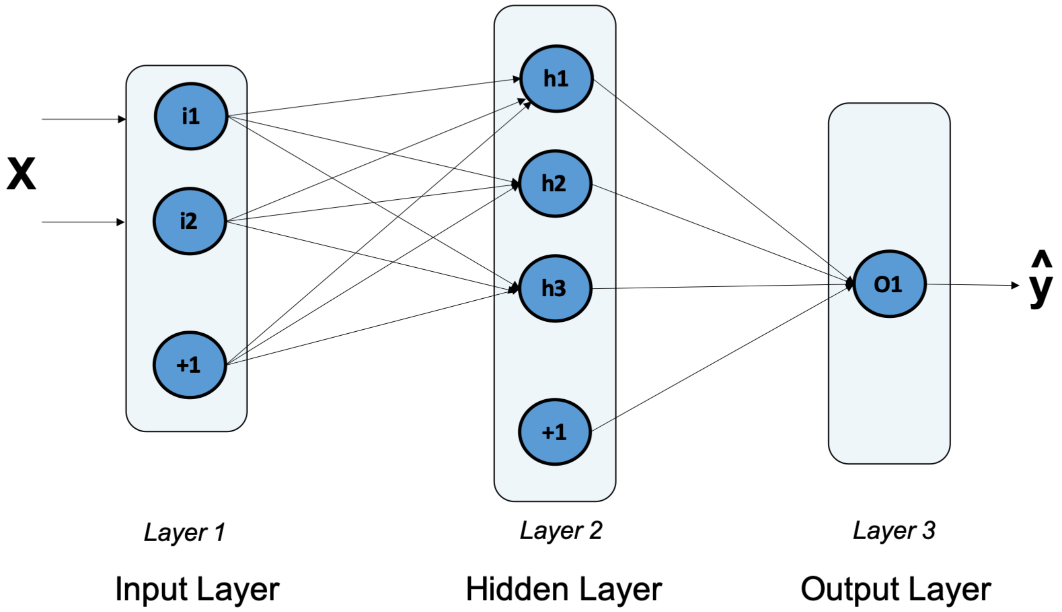
- Une couche d'entrée qui reçoit l'ensemble des caractéristiques (features), i.e. les variables prédictives.
- Un nombre arbitraire de couches cachées.
- Une couche de sortie, ŷ, qui contient la variable à prédire.
- Un ensemble de poids W qui vont être ajoutés aux valeurs des features et de biais b entre chaque couche
- Un choix de fonction d'activation pour chaque couche cachée, σ.

Remarque La couche de sortie doit avoir autant de neurones qu'il y a de sorties au problème de classification :

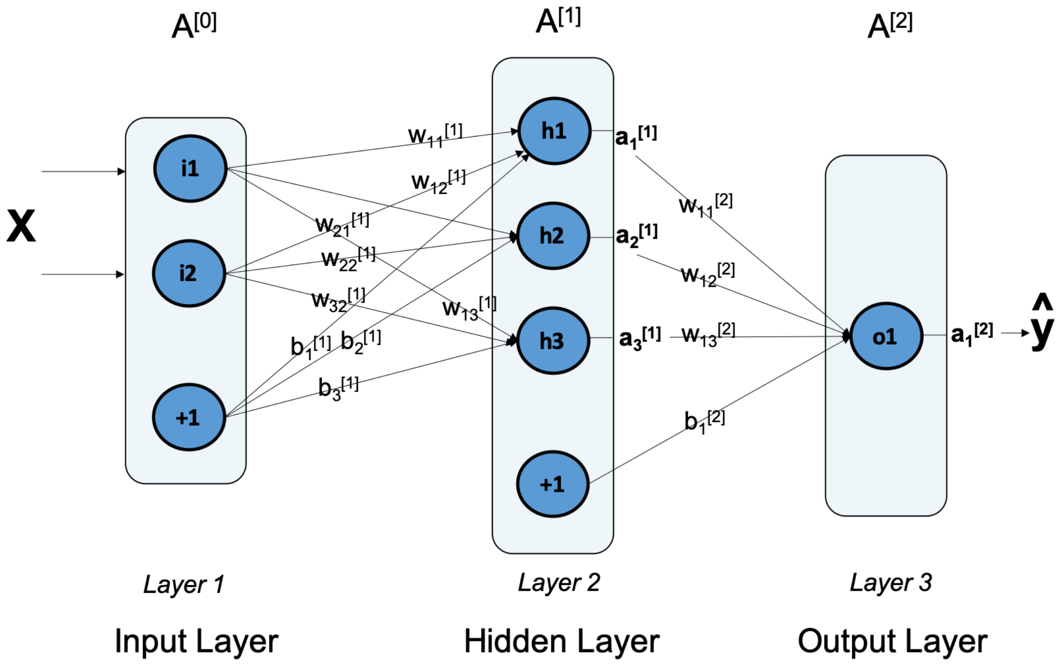
- *régression* : 1 seul neurone (C.f. notebook descente de gradient)
- *classification binaire* : 1 seul neurone avec une fonction d'activation qui sépare les deux classes.
- *classification multi-classe* : 1 neurone par classe et une fonction d'activation Softmax pour avoir la classe appropriée en fonction des probabilités de l'entrée appartenant à chaque classe.

La figure suivante illustre un exemple de réseau avec 3 couches :

- le layer 1 correspond au layer d'entrée (*input layer*), il reçoit l'ensemble des variables prédicives et est composé de 2 neurones. Le neurone avec +1 correspond au biais qui est ajouté.
- le layer 2 est appelé couche cachée (*hidden layer*), il possède 3 neurones et aussi un biais.
- le layer 3 correspond à la couche de sortie (*output layer*), la sortie de ce layer correspond à la prédiction.

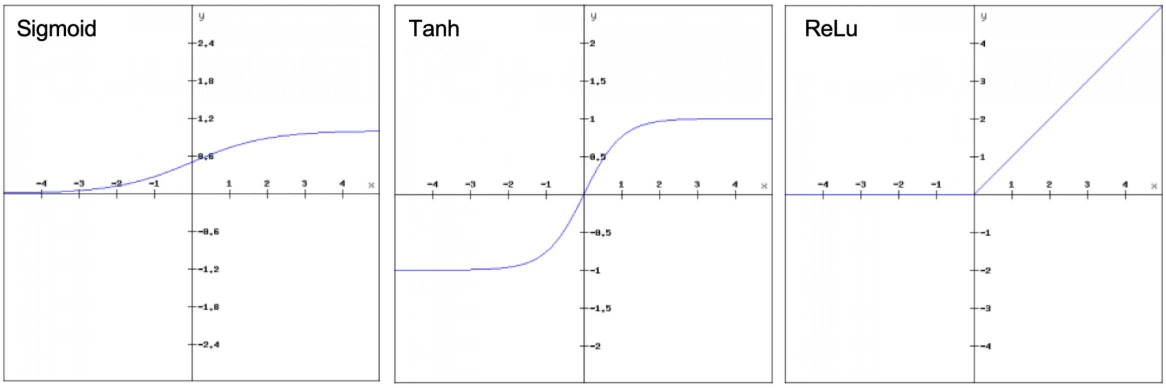


La figure suivante illustre le même réseau avec les poids affectés.



Notations : X correspond aux variables prédictives. Le poids est identifié de la manière suivante : $w_{ij}^{[l]}$ où l correspond au niveau du layer cible, i correspond au numéro du nœud de la connection dans la couche $l-1$ et j correspond au numéro du nœud de la connection dans la couche l. Par exemple, le poids entre le nœud 1 dans le layer 1 et le nœud 2 dans le layer 2 est noté : $w_{12}^{[2]}$. Un biais est connecté à chaque nœud de la couche suivante. La notation est similaire : $b_i^{[l]}$ où i est le numéro du nœud de la couche supérieure. La sortie d'un nœud est notée = $a_i^{[l]}$ où i correspond au numéro du nœud dans la couche l. **ŷ** correspond à la variable prédite.

Choix de la fonction d'activation

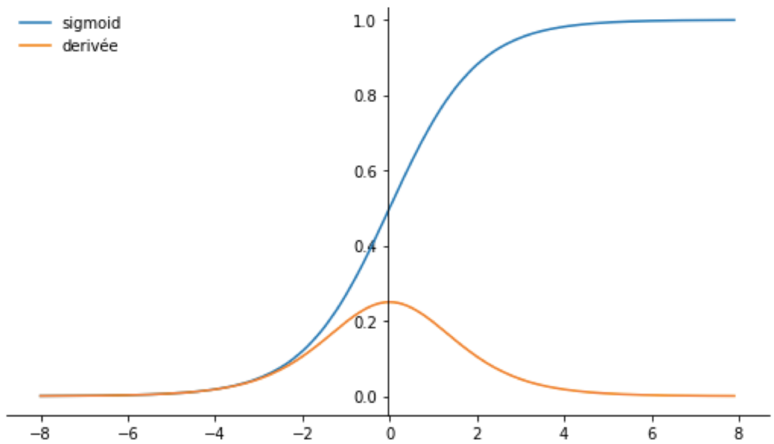


Il existe de très nombreuses fonctions d'activation qui peuvent être utilisées :

- Binary Step
- Sigmoid
- Tanh
- ReLU
- Leaky ReLU
- Softmax
- ...

Elles n'ont pas les mêmes propriétés.

Nous verrons, par la suite, que les réseaux de neurones utilisent la descente de gradient, le comportement de la dérivée des fonctions est donc important. Par exemple, nous avons vu que la sigmoid va transformer de grandes valeurs d'entrée dans des valeurs comprises entre 0 et 1. Cela veut dire qu'une modification importante de l'entrée entraînera une modification mineure de la sortie (C.f. notebook descente de gradient). Par conséquent la dérivée devient plus petite comme l'illustre l'image ci-dessous :



En fait, pour corriger les erreurs, les dérivées du réseau vont être propagées layer par layer de l'output layer à l'input layer. Le problème est que celles-ci sont multipliées entre chaque layer afin de connaître les valeurs de dérivées utiles pour l'input layer : le gradient décroît de façon exponentielle à mesure que nous nous propagons jusqu'aux couches initiales.

Pour choisir les fonctions d'activation, il faut considérer les propriétés principales suivantes :

- La disparition du gradient (*vanishing gradient*) : le problème intervient généralement dans des réseaux avec de très nombreux layer. Comme les descentes de gradient sont propagées dans tout le réseau, de trop petites valeurs de gradient (le gradient de la fonction de perte approche 0) indiquent que les poids des premiers layers ne seront pas mis à jour efficacement à chaque étape. Ceci entraîne donc une imprécision globale du réseau. Cela peut arriver si le réseau est composé de nombreuses couches avec une sigmoid.

- Disparition de neurones (*dead neuron*) : un neurone mort est un neurone qui, lors de l'apprentissage, ne s'active plus. Cela est lié au fait que les dérivées sont très petites ou nulles. Le neurone ne peut donc pas mettre à jour les poids. Les erreurs ne se propageant plus, ce neurone peut affecter les autres neurones du réseau. C'est, par exemple, le cas avec ReLu qui renvoie 0 quand l'entrée est inférieure ou égale à 0. Si chaque exemple donne une valeur négative, le neurone ne s'active pas et après la descente de gradient le neurone devient 0 donc ne sera plus utilisé. Le Leaky Relu permet de résoudre ce problème.
- Explosion du gradient (*Explosing gradient*) : le problème se pose lorsque des gradients d'erreur important s'accumulent et entraînent des mises à jour importantes des poids. Cela amène un réseau instable : les valeurs de mises à jour des poids peuvent être trop grandes et être remplacées par des NaN donc non utilisables (s'il n'y a pas d'erreurs d'exécution bien sûr !). Le problème est lié au type de descente de gradient utilisé (Batch vs mini-batch), au fait qu'il y a peut être trop de couches dans le réseau et bien sûr à certaines fonctions d'activation qui favorisent ce problème.
- Saturation de neurones (*Saturated neurons*) : le problème est lié au fait que les valeurs grandes (resp. petites) atteignent un plafond et qu'elles ne changent pas lors de la propagation dans le réseau. Ce problème est principalement lié aux fonctions sigmoid et tanh. En effet, sigmoid, pour toutes les valeurs supérieures à 1 va arriver sur un plateau et retournera toujours 1. Pour cela, ces deux fonctions d'activations sont assez déconseillées en deep learning (préférer LeRu ou Leaky Relu).

Pour avoir une idée du comportement des différentes fonctions d'activation et de leurs conséquences : ”

Une fois que tout est fixe, le réseau de neurones s'exécute alors en deux étapes :

- Forward Propagation
- Backward Propagation

Forward Propagation

L'objectif de cette étape est de déterminer la valeur de sortie du réseau : **ŷ**.

Comme nous avons vu dans le notebook descente de gradient, pour chaque neurone de la couche, nous effectuons une application affine en considérant les valeurs issues de la couche précédente (i.e. a représente le résultat de la fonction d'activation de la couche précédente) :

$$\mathbf{z}_i^{[l]} = \mathbf{w}_i^T \bullet \mathbf{a}^{[l-1]} + b_i \qquad \mathbf{a}_i^{[l]} = \sigma^{[l]}(\mathbf{z}_i^{[l]})$$

Que nous pouvons donc généraliser en utilisant les matrices :

$$\begin{aligned} \mathbf{Z}^{[l]} &= \mathbf{W}^{[l]} \bullet \mathbf{A}^{[l-1]} + \mathbf{b}^{[l]} \\ \mathbf{A}^{[l]} &= \sigma^{[l]}(\mathbf{Z}^{[l]}) \end{aligned}$$

Si nous reprenons l'exemple de réseau précédent avec ReLu pour le hidden layer et sigmoid pour le layer de sortie nous avons donc :

$$\mathbf{A}^{[0]} = X$$

où $\mathbf{A}^{[0]} = X$ la matrice contenant les exemples d'apprentissage.

$$\begin{aligned} \mathbf{Z}^{[1]} &= \mathbf{W}^{[1]} \bullet \mathbf{A}^{[0]} + \mathbf{b}^{[1]} \\ \mathbf{A}^{[1]} &= ReLu^{[1]}(\mathbf{Z}^{[1]}) \\ \mathbf{Z}^{[2]} &= \mathbf{W}^{[2]} \bullet \mathbf{A}^{[1]} + \mathbf{b}^{[2]} \\ \mathbf{A}^{[2]} &= Sigmoid^{[2]}(\mathbf{Z}^{[2]}) \end{aligned}$$

Finalement : $\hat{y} = \mathbf{A}^{[2]}$

Le code ci-dessous illustre un exemple simple de la phase forward propagation pour notre exemple comportant deux variables prédictives. Les fonctions d'activations sont respectivement Relu pour le hidden layer et sigmoid pour le dernier layer.

In [4]:

```
import numpy as np

#fonctions d'activation
def sigmoid(Z):
    return 1/(1+np.exp(-Z))

def relu(Z):
    return np.maximum(0,Z)

def valueoutput(y_hat):
    for i in range(len(y_hat)):
        if y_hat[i]>0.5:
            y_hat[i]=1
        else:
            y_hat[i]=0
    return y_hat

# les donnees d'entree sous la forme d'une matrice (array en python)
X = np.array(([0, 1],
              [0, 0],
              [1, 0],
              [1, 1],
              [1, 1]))
# les donnees de sortie sous la forme d'un vecteur
y = np.array(([1],
             [0],
             [0],
             [1],
             [1]), dtype=float)


# initialisation des poids de manière aléatoire ainsi que des biais
inputSize = 2
hiddenSize = 3
outputSize = 1
W1=np.random.rand(2, 3)
W2=np.random.rand(3, 1)
b1 = np.random.rand(3)
b2 = np.random.rand(1)
print ("Les données d'entrées : \n",X)
print ('Les valeurs de poids et de biais initialisees aléatoirement : \n')
print ('\t(layer input vers layer 1) : W1 \n',W1,'\n')
print ('\t(layer input vers layer 1) : b1\n',b1,'\n')
print ("\t(layer 1 vers layer 2) : W2\n",W2,'\n')
print ("\t(layer 1 vers layer 2) : b2\n",b2,'\n')
print ("Etape 1 : ")
print ("\n A0=X\n")

A0=X
Z1 = np.dot(A0,W1)+b1
print ("\n Z1 = W1.A0 + b1  \n",Z1,'\n')
A1 = relu(Z1)
print ('\nA1 = relu(Z1)\n',A1,'\n')
Z2 = np.dot(A1,W2)+b2
print ("\n Z2 = W2.A1 + b2  \n",Z2,'\n')
A2 = sigmoid(Z2)
print ('\nA2 = sigmoid(Z2)\n',A2,'\n')
y_hat = sigmoid(Z2)
print ('yhat\n',y_hat)
```

```python



print ("Les données d'entrées : \n",X)
print ("Les sorties predites : \n", str(valueoutput(y_hat)))

print ("Les sorties reelles attendues : \n", str(y))
```

```
Les données d'entrées :
 [[0 1]
 [0 0]
 [1 0]
 [1 1]
 [1 1]]
Les valeurs de poids et de biais initialisees aléatoirement :

        (layer input vers layer 1) : W1
 [[0.90496764 0.66934312 0.61669425]
 [0.70675322 0.21538845 0.58636595]]

        (layer input vers layer 1) : b1
 [0.55441685 0.50237972 0.86147085]

        (layer 1 vers layer 2) : W2
 [[0.68936531]
 [0.25632519]
 [0.84027211]]

        (layer 1 vers layer 2) : b2
 [0.18734206]

Etape 1 :

 A0=X


 Z1 = W1.A0 + b1
 [[1.26117007 0.71776817 1.4478368 ]
 [0.55441685 0.50237972 0.86147085]
 [1.45938449 1.17172285 1.4781651 ]
 [2.16613771 1.38711129 2.06453105]
 [2.16613771 1.38711129 2.06453105]]


A1 = relu(Z1)
 [[1.26117007 0.71776817 1.4478368 ]
 [0.55441685 0.50237972 0.86147085]
 [1.45938449 1.17172285 1.4781651 ]
 [2.16613771 1.38711129 2.06453105]
 [2.16613771 1.38711129 2.06453105]]


 Z2 = W2.A1 + b2
 [[2.45730789]
 [1.4221803 ]
 [2.73579409]
 [3.77092168]
 [3.77092168]]


A2 = sigmoid(Z2)
 [[0.92109422]
```

```
[[0.92109422]
 [0.80567999]
 [0.93910602]
 [0.97748765]
 [0.97748765]]

yhat
 [[0.92109422]
 [0.80567999]
 [0.93910602]
 [0.97748765]
 [0.97748765]]
Les données d'entrées :
 [[0 1]
 [0 0]
 [1 0]
 [1 1]
 [1 1]]
Les sorties predites :
 [[1.]
 [1.]
 [1.]
 [1.]
 [1.]]
Les sorties reelles attendues :
 [[1.]
 [0.]
 [0.]
 [1.]
 [1.]]
```

Comme nous pouvons le constater il y a des erreurs dans les sorties prédites. C'est là qu'intervient la seconde phase.

Backward Propagation

L'objectif de la Backward Propagation est tout d'abord d'évaluer la différence entre la valeur prédite et la valeur réelle.

Etape 1 : (calcul du coût)

Nous avons vu dans le notebook de la descente de gradient que la différence entre la valeur obtenue dans l'étape précédente et la valeur réelle correspond au coût. Plus la différence est élevée, plus le coût sera élevé. Pour minimiser ce coût, il faut trouver les valeurs de poids et de biais pour lesquelles la fonction de coût renvoie la plus petite valeur possible. Plus le coût est faible, plus les prévisions sont exactes. Nous retrouvons donc le problème rencontré pour la descente de gradient.

Précédemment nous avons vu que la cross entropy, comme fonction de coût, était bien adaptée à notre problème de classification binaire, donc :

$$\mathbf{C(\hat{y}, y) = -y\log(\hat{y}) - (1 - y)\log(1 - \hat{y})}$$

L'objectif, à présent, est de propager cette erreur dans tout le réseau pour mettre à jour les différents poids.

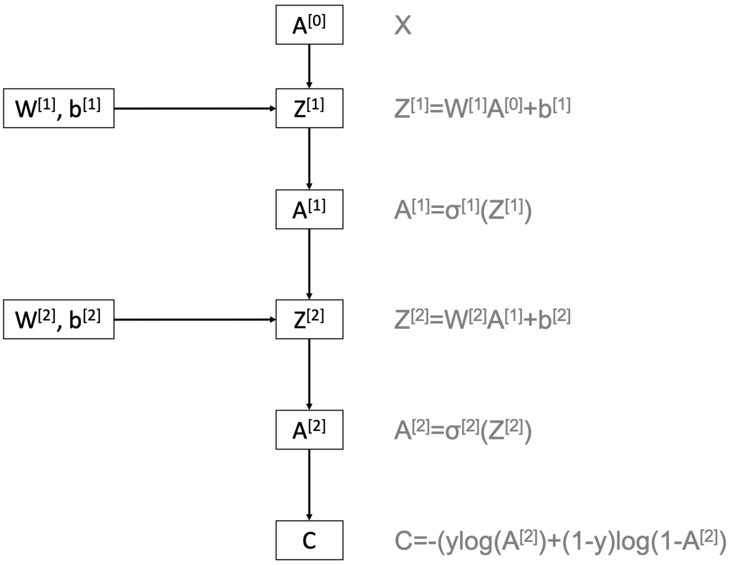
Etape 2 : (backpropagation)
Comprendre ce qui est derrière

Nous avons vu précédemment, lors de la phase de forward, que l'exécution était de la forme :

$$\begin{aligned}
\mathbf{A}^{[0]} &= X \\
\mathbf{Z}^{[1]} &= \mathbf{W}^{[1]} \bullet \mathbf{A}^{[0]} + \mathbf{b}^{[1]} \\
\mathbf{A}^{[1]} &= \sigma^{[1]}(\mathbf{Z}^{[1]}) \\
\mathbf{Z}^{[2]} &= \mathbf{W}^{[2]} \bullet \mathbf{A}^{[1]} + \mathbf{b}^{[2]} \\
\mathbf{A}^{[2]} &= \sigma^{[2]}(\mathbf{Z}^{[2]}) \\
&\vdots \\
\mathbf{Z}^{[L]} &= \mathbf{W}^{[L]} \bullet \mathbf{A}^{[L-1]} + \mathbf{b}^{[L]} \\
\mathbf{A}^{[L]} &= \sigma^{[L]}(\mathbf{Z}^{[L]}) = \hat{y}
\end{aligned}$$

où L est le output layer.

La figure suivante illustre les étapes jusqu'à la fonction de coût pour notre réseau exemple :



L'objectif de la backward propagation est de reporter, dans le réseau, l'ensemble des modifications à apporter aux poids entre les couches. Pour cela il faut repartir en sens inverse pour calculer les dérivées partielles du coût. Elle repose sur la règle de dérivation en chaîne (*chain rule*) qui est une formule qui explicite la dérivée d'une fonction composée pour deux fonctions dérivables :

$$\mathbf{\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}}$$

Lorsque l'on regarde la fin du réseau, nous constatons que $\mathbf{C}$ est une fonction qui dépend de $\mathbf{A^{[2]}}$, que $\mathbf{A^{[2]}}$ dépend, elle même, d'une fonction $\mathbf{Z^{[2]}}$ et que finalement $\mathbf{Z^{[2]}}$ dépend de $\mathbf{W^{[2]}}$ et de $\mathbf{b^{[2]}}$.

$$\frac{\partial \mathbf{C}}{\partial \mathbf{W^{[2]}}} = \frac{\partial \mathbf{C}}{\partial \mathbf{A^{[2]}}} \bullet \frac{\partial \mathbf{A^{[2]}}}{\partial \mathbf{Z^{[2]}}} \bullet \frac{\partial \mathbf{Z^{[2]}}}{\partial \mathbf{W^{[2]}}}$$

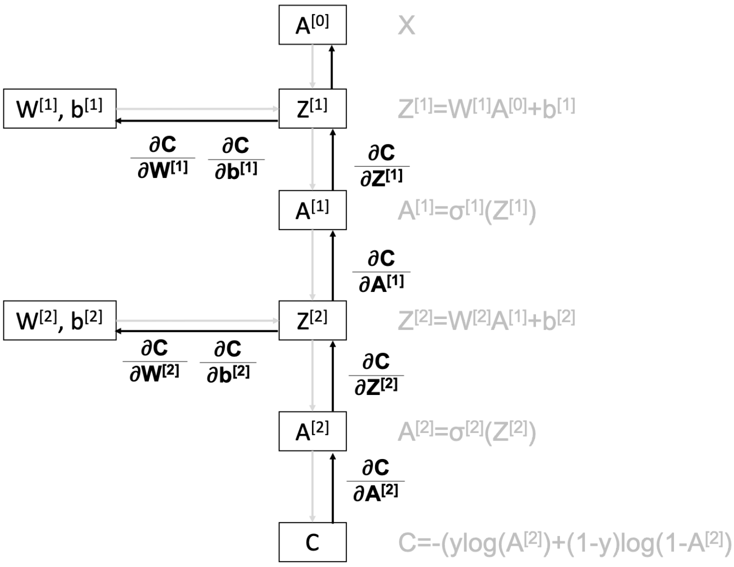
$$\frac{\partial \mathbf{C}}{\partial \mathbf{b^{[2]}}} = \frac{\partial \mathbf{C}}{\partial \mathbf{A^{[2]}}} \bullet \frac{\partial \mathbf{A^{[2]}}}{\partial \mathbf{Z^{[2]}}} \bullet \frac{\partial \mathbf{Z^{[2]}}}{\partial \mathbf{b^{[2]}}}$$

De la même manière, pour avoir la dérivée partielle de $\mathbf{C}$ par rapport à $\mathbf{W^{[1]}}$ et $\mathbf{b^{[1]}}$, nous voyons, sur la figure, que $\mathbf{Z^{[2]}}$ est une fonction qui dépend de $\mathbf{A^{[1]}}$, qui, elle-même, dépend de $\mathbf{Z^{[1]}}$ et que finalement $\mathbf{Z^{[1]}}$ dépend de $\mathbf{W^{[1]}}$ et $\mathbf{b^{[1]}}$.

$$\frac{\partial \mathbf{C}}{\partial \mathbf{W^{[1]}}} = \frac{\partial \mathbf{C}}{\partial \mathbf{A^{[2]}}} \bullet \frac{\partial \mathbf{A^{[2]}}}{\partial \mathbf{Z^{[2]}}} \bullet \frac{\partial \mathbf{Z^{[2]}}}{\partial \mathbf{A^{[1]}}} \bullet \frac{\partial \mathbf{A^{[1]}}}{\partial \mathbf{Z^{[1]}}} \bullet \frac{\partial \mathbf{Z^{[1]}}}{\partial \mathbf{W^{[1]}}}$$

$$\frac{\partial \mathbf{C}}{\partial \mathbf{b}^{[1]}} = \frac{\partial \mathbf{C}}{\partial \mathbf{A}^{[2]}} \bullet \frac{\partial \mathbf{A}^{[2]}}{\partial \mathbf{Z}^{[2]}} \bullet \frac{\partial \mathbf{Z}^{[2]}}{\partial \mathbf{A}^{[1]}} \bullet \frac{\partial \mathbf{A}^{[1]}}{\partial \mathbf{Z}^{[1]}} \bullet \frac{\partial \mathbf{Z}^{[1]}}{\partial \mathbf{b}^{[1]}}$$

Les différentes étapes sont résumées sur la figure suivante :



Pour résumer, les équations pour calculer la dérivée partielle de la fonction de coût en fonction des poids et des biais d'une couche $\mathbf{l}$ sont :

$$\frac{\partial \mathbf{C}}{\partial \mathbf{W}^{[\mathbf{l}]}} = \frac{\partial \mathbf{C}}{\partial \mathbf{Z}^{[\mathbf{l}]}} \bullet \frac{\partial \mathbf{Z}^{[\mathbf{l}]}}{\partial \mathbf{W}^{[\mathbf{l}]}}$$

$$\frac{\partial \mathbf{C}}{\partial \mathbf{b}^{[\mathbf{l}]}} = \frac{\partial \mathbf{C}}{\partial \mathbf{Z}^{[\mathbf{l}]}} \bullet \frac{\partial \mathbf{Z}^{[\mathbf{l}]}}{\partial \mathbf{b}^{[\mathbf{l}]}}$$

Donc, pour obtenir les dérivées partielles de $\mathbf{C}$ par rapport à $\mathbf{W}^{[\mathbf{l}]}$ et $\mathbf{b}^{[\mathbf{l}]}$, nous devons calculer :

$$\frac{\partial \mathbf{C}}{\partial \mathbf{A}^{[\mathbf{L}]}}, \frac{\partial \mathbf{C}}{\partial \mathbf{Z}^{[\mathbf{L}]}}, \frac{\partial \mathbf{C}}{\partial \mathbf{Z}^{[\mathbf{l}]}} \frac{\partial \mathbf{Z}^{[\mathbf{l}]}}{\partial \mathbf{W}^{[\mathbf{l}]}}, \frac{\partial \mathbf{Z}^{[\mathbf{l}]}}{\partial \mathbf{b}^{[\mathbf{l}]}}$$

Par la suite, et par simplification, nous considérons que les deux fonctions d'activation dans notre réseau sont Relu (pour $A^{[1]}$) et sigmoid (pour $A^{[2]}$). Le principe est le même quelques soient les fonctions, il suffit juste de connaître la dérivée des fonctions d'activation.

Pour la dérivée partielle de $\mathbf{C}$ par rapport à $\mathbf{A}^{[\mathbf{L}]}$:

$$\frac{\partial \mathbf{C}}{\partial \mathbf{A}^{[\mathbf{L}]}} = \frac{\partial(-\mathbf{y}\mathbf{log}(\mathbf{A}^{[\mathbf{L}]}) - (\mathbf{1}-\mathbf{y})\mathbf{log}(\mathbf{1}-\mathbf{A}^{[\mathbf{L}]}))}{\partial \mathbf{A}^{[\mathbf{L}]}}$$

Nous savons que la dérivée d'une fonction log(x) est :

$$\frac{\partial \mathbf{log}(\mathbf{x})}{\partial \mathbf{dx}} = \frac{\mathbf{1}}{\mathbf{x}}$$

Pour la partie gauche $-\mathbf{y}\mathbf{log}(\mathbf{A}^{[\mathbf{L}]})$ nous avons donc comme dérivée :

$$\frac{-\mathbf{y}}{\mathbf{A}^{[\mathbf{L}]}}$$

Pour la partie droite $-(\mathbf{1}-\mathbf{y})\mathbf{log}(\mathbf{1}-\mathbf{A}^{[\mathbf{L}]})$, il faut juste appliquer la formule de la dérivée d'une fonction :

$$\frac{\partial \mathbf{log}(\mathbf{g}(\mathbf{x}))}{\partial \mathbf{dx}} = \frac{\mathbf{1}}{\mathbf{g}(\mathbf{x})}\mathbf{g}'(\mathbf{x}))$$

comme la dérivée de $\mathbf{1 - A^{[L]}}$ est $\mathbf{-1}$ nous avons au final :

$$\frac{\partial \mathbf{C}}{\partial \mathbf{A^{[L]}}} = \frac{\mathbf{-y}}{\mathbf{A^{[L]}}} - (-)\frac{\mathbf{(1-y)}}{\mathbf{(1-A^{[L]})}}$$

$$= \left(\frac{\mathbf{-y}}{\mathbf{A^{[L]}}} + \frac{\mathbf{(1-y)}}{\mathbf{(1-A^{[L]})}}\right)$$

Donc :

$$\boxed{\frac{\partial \mathbf{C}}{\partial \mathbf{A^{[L]}}} = \left(\frac{\mathbf{-y}}{\mathbf{A^{[L]}}} + \frac{\mathbf{(1-y)}}{\mathbf{(1-A^{[L]})}}\right)}$$

Considérons, à présent la dérivée partielle de $\mathbf{C}$ par rapport à $\mathbf{Z^{[L]}}$:

$$\frac{\partial \mathbf{C}}{\partial \mathbf{Z^{[L]}}}$$

En utilisant la chaîne de dérivation :

$$\frac{\partial \mathbf{C}}{\partial \mathbf{Z^{[L]}}} = \frac{\partial \mathbf{C}}{\partial \mathbf{A^{[L]}}} \bullet \frac{\partial \mathbf{A^{[L]}}}{\partial \mathbf{Z^{[L]}}} = \frac{\partial \mathbf{C}}{\partial \mathbf{A^{[L]}}} * \sigma'^{\mathbf{[L]}}(\mathbf{Z^{[L]}})$$

$\sigma'^{\mathbf{[L]}}(\mathbf{Z^{[L]}})$ correspond simplement à la dérivée de la sigmoid. Nous avons vu dans le notebook sur la descente de gradient, que cette dérivée est :

$$\frac{\partial \mathbf{A^{[L]}}}{\partial \mathbf{Z^{[L]}}} = \mathbf{sigmoid(Z^{[L]})(1 - sigmoid(Z^{[L]})) = A^{[L]}(1 - A^{[L]})}$$

Donc :

$$\frac{\partial \mathbf{C}}{\partial \mathbf{A^{[L]}}} \bullet \frac{\partial \mathbf{A^{[L]}}}{\partial \mathbf{Z^{[L]}}} = \left(\frac{\mathbf{-y}}{\mathbf{A^{[L]}}} + \frac{\mathbf{(1-y)}}{\mathbf{(1-A^{[L]})}}\right)\mathbf{A^{[L]}(1 - A^{[L]})}$$

Nous pouvons multiplier par $\mathbf{(1 - A^{[L]})}$ et $\mathbf{(A^{[L]})}$ pour simplifier :

$$= \left(\frac{\mathbf{-y(1-A^{[L]})}}{\mathbf{A^{[L]}(1-A^{[L]})}} + \frac{\mathbf{A^{[L]}(1-y)}}{\mathbf{A^{[L]}(1-A^{[L]})}}\right)\mathbf{A^{[L]}(1 - A^{[L]})}$$

$$= \left(\frac{\mathbf{-y(1-A^{[L]}) + A^{[L]}(1-y)}}{\mathbf{A^{[L]}(1-A^{[L]})}}\right)\mathbf{A^{[L]}(1 - A^{[L]})}$$

en supprimant $\mathbf{A^{[L]}(1 - A^{[L]})}$ nous avons :

$$= \left(\mathbf{-y(1-A^{[L]}) + A^{[L]}(1-y)}\right)$$

$$= \mathbf{-y + yA^{[L]} + A^{[L]} - A^{[L]}y}$$

$$= \mathbf{-y + A^{[L]}}$$

Donc :

$$\boxed{\frac{\partial \mathbf{C}}{\partial \mathbf{Z^{[L]}}} = \mathbf{A^{[L]} - y}}$$

Dérivée partielle de $\mathbf{C}$ par rapport à $\mathbf{Z^{[l]}}$:

Nous souhaitons, à présent, obtenir la dérivée partielle de $\mathbf{C}$ par rapport à un niveau $\mathbf{l}$, i.e. $\mathbf{Z^{[l]}}$. Le principe étant que si l'on connaît $\mathbf{Z^{[L]}}$, il est possible de déduire $\mathbf{Z^{[L-1]}}$, $\mathbf{Z^{[L-2]}}$, ...

Nous savons, en appliquant la chaîne de dérivation, qu'il est possible de calculer la dérivée de $\frac{\partial \mathbf{C}}{\partial \mathbf{Z^{[l]}}}$ par :

$$\frac{\partial \mathbf{C}}{\partial \mathbf{Z^{[l]}}} = \frac{\partial \mathbf{C}}{\partial \mathbf{Z^{[l+1]}}} \bullet \frac{\partial \mathbf{Z^{[l+1]}}}{\partial \mathbf{A^{[l]}}} \bullet \frac{\partial \mathbf{A^{[l]}}}{\partial \mathbf{Z^{[l]}}}$$

Comme :

$$\mathbf{Z^{[l+1]}} = \mathbf{W^{[l+1]}} \bullet \mathbf{A[l] + b[l+1]}$$

$$\frac{\partial \mathbf{Z^{[l+1]}}}{\partial \mathbf{A^{[l]}}} = \frac{\partial(\mathbf{W^{[l+1]}} \bullet \mathbf{A[l] + b[l+1]})}{\partial \mathbf{A^{[l]}}}$$

$$= \mathbf{W^{[l+1]}}$$

Nous avons également :

$$\frac{\partial \mathbf{A^{[l]}}}{\partial \mathbf{Z^{[l]}}} = \sigma'^{[l]}(\mathbf{Z^{[l]}})$$

Donc :

$$\boxed{\frac{\partial \mathbf{C}}{\partial \mathbf{Z^{[l]}}} = (\mathbf{W^{[l+1]^T}} \bullet \frac{\partial \mathbf{C}}{\partial \mathbf{Z^{[l+1]}}}) * \sigma'^{[l]}(\mathbf{Z^{[l]}})}$$

Dérivée partielle de $\mathbf{Z^{[l]}}$ par rapport à $\mathbf{W^{[l]}}$:

Dans un premier temps nous calculons la dérivée partielle de $\mathbf{Z^{[l]}}$ par rapport à $\mathbf{W^{[l]}}$ pour, par la suite déterminer la dérivée partielle de $\mathbf{C}$ par rapport à $\mathbf{W^{[l]}}$.

Comme :

$$\mathbf{Z^{[l]}} = \mathbf{W^{[l]}} \bullet \mathbf{A[l-1] + b[l]}$$

$$\frac{\partial \mathbf{Z^{[l]}}}{\partial \mathbf{W^{[l]}}} = \frac{\partial(\mathbf{W^{[l]}} \bullet \mathbf{A[l-1] + b[l]})}{\partial \mathbf{W^{[l]}}}$$

$$= \mathbf{A^{[l-1]}}$$

Dérivée partielle de $\mathbf{C}$ par rapport à $\mathbf{W^{[l]}}$:

A partir du résultat précédent, nous avons :

$$\boxed{\frac{\partial \mathbf{C}}{\partial \mathbf{W^{[l]}}} = \frac{\partial \mathbf{C}}{\partial \mathbf{Z^{[l]}}} \bullet \mathbf{A^{[l-1]^T}}}$$

Dérivée partielle de $\mathbf{Z^{[l]}}$ par rapport à $\mathbf{b^{[l]}}$:

Comme précédemment, nous calculons la dérivée partielle de $\mathbf{Z^{[l]}}$ par rapport à $\mathbf{b^{[l]}}$ pour, par la suite déterminer la dérivée partielle de $\mathbf{C}$ par rapport à $\mathbf{b^{[l]}}$.

Comme :

$$\mathbf{Z^{[l]}} = \mathbf{W^{[l]}} \bullet \mathbf{A[l-1] + b[l]}$$

$$\frac{\partial \mathbf{Z^{[l]}}}{\partial \mathbf{b^{[l]}}} = \frac{\partial(\mathbf{W^{[l]}} \bullet \mathbf{A[l-1] + b[l]})}{\partial \mathbf{b^{[l]}}}$$

$$= \mathbf{1}$$

Dérivée partielle de $\mathbf{C}$ par rapport à $\mathbf{b^{[l]}}$:

A partir du résultat précédent, nous avons :

A partir du résultat précédent, nous avons :

$$\boxed{\frac{\partial \mathbf{C}}{\partial \mathbf{b^{[l]}}} = \frac{\partial \mathbf{C}}{\partial \mathbf{Z^{[l]}}}}$$

Pour résumer

A présent, pour le dernier layer $\mathbf{L}$, nous sommes capable de calculer la dérivée partielle du coût par rapport à $\mathbf{A^{[L]}}$, $\mathbf{Z^{[L]}}$, $\mathbf{W^{[L]}}$ et $\mathbf{b^{[L]}}$:

$\frac{\partial \mathbf{C}}{\partial \mathbf{A^{[L]}}}$	$\left(\frac{-\mathbf{y}}{\mathbf{A^{[L]}}} + \frac{(\mathbf{1-y})}{(\mathbf{1-A^{[L]}})}\right)$
$\frac{\partial \mathbf{C}}{\partial \mathbf{Z^{[L]}}}$	$(\mathbf{A^{[L]}} - \mathbf{y})$
$\frac{\partial \mathbf{C}}{\partial \mathbf{W^{[L]}}}$	$\frac{\partial \mathbf{C}}{\partial \mathbf{Z^{[L]}}} \bullet (\mathbf{A^{[L-1]^T}})$
$\frac{\partial \mathbf{C}}{\partial \mathbf{b^{[L]}}}$	$\frac{\partial \mathbf{C}}{\partial \mathbf{Z^{[L]}}}$

Pour n'importe quel layer $\mathbf{l}$, nous avons :

$\frac{\partial \mathbf{C}}{\partial \mathbf{Z^{[l]}}}$	$(\mathbf{W^{[l+1]^T}} \bullet \frac{\partial \mathbf{C}}{\partial \mathbf{Z^{[l+1]}}}) * \sigma'^{[\mathbf{l}]}(\mathbf{Z^{[l]}})$
$\frac{\partial \mathbf{C}}{\partial \mathbf{W^{[l]}}}$	$\frac{\partial \mathbf{C}}{\partial \mathbf{Z^{[l]}}} \bullet \mathbf{A^{[l-1]^T}}$
$\frac{\partial \mathbf{C}}{\partial \mathbf{b^{[l]}}}$	$\frac{\partial \mathbf{C}}{\partial \mathbf{Z^{[l]}}}$

La descente de gradient

Attention, parfois, la backward propagation est considérée comme la descente de gradient. Ce n'est pas le cas. Elle a pour seul objectif de calculer les gradients pour les opérations à chaque niveau. La descente de gradient intervient après, elle permet de pouvoir mettre automatiquement les poids des différents layer en appliquant justement les gradients obtenus dans l'étape précédente.

La descente de gradient se fait comme dans le notebook : il faut boucler jusqu'au premier layer pour appliquer la formule du gradient à l'aide des dérivées calculées précédemment :

For $\mathbf{l}$ in enumate (dernier_layer,1) {

$$\mathbf{W^{[l]}} = \mathbf{W^{[l]}} - \eta \frac{\partial \mathbf{C}}{\partial \mathbf{W^{[l]}}}$$
$$\mathbf{b^{[l]}} = \mathbf{b^{[l]}} - \eta \frac{\partial \mathbf{C}}{\partial \mathbf{b^{[l]}}}$$

}

Remarque : comme nous l'avons vu lors des dérivations, il est nécessaire de sauvegarder $\frac{\partial \mathbf{C}}{\partial \mathbf{W^{[l]}}}$ et $\frac{\partial \mathbf{C}}{\partial \mathbf{b^{[l]}}}$ pour pouvoir les réutiliser lors de la descente de gradient.

Cas de la classification multi-classes

Jusqu'à présent nous avons vu comment faire de la classification binaire, i.e. la fonction d'activation est une sigmoid. Pour faire de la classification multi-classe, il faut utiliser la fonction d'activation softmax. Elle attribue des probabilités à chaque classe d'un problème à plusieurs classes et la somme de ces probabilités doit être égale à 1. Formellement softmax, prend en entrée un vecteur de C-dimensions (le nombre de classes possibles) $\mathbf{z}$ et retourne un autre vecteur de C-dimensions $\mathbf{a}$ de valeurs réelles comprises entre 0 et 1.

Pour $i = 1 \cdots C$:

$$\mathbf{a_i} = \frac{\mathbf{e^{z_i}}}{\mathbf{\sum_{k=1}^{C} e^{z_k}}}$$

$$avec \sum_{i=1}^{C} = 1$$

où C correspond au nombre de classes.

In [5]:

```
def softmax(z):
    expz = np.exp(z)
    return expz / expz.sum(axis=0, keepdims=True)

nums = np.array([4, 5, 6])
print(softmax(nums))
print ("la somme des probabilités donne 1")
```

```
[0.09003057 0.24472847 0.66524096]
la somme des probabilités donne 1
```

Cependant, cette fonction n'est pas très stable : elle génère souvent des nan pour des grands nombres par exemple.

In [6]:

```
nums = np.array([4000, 5000, 6000])
print(softmax(nums))
```

```
[nan nan nan]

/Users/pascalponcelet/Desktop/Sicki-learn/Tools/tools/lib/python3.6/si
te-packages/ipykernel_launcher.py:2: RuntimeWarning: overflow encounte
red in exp

/Users/pascalponcelet/Desktop/Sicki-learn/Tools/tools/lib/python3.6/si
te-packages/ipykernel_launcher.py:3: RuntimeWarning: invalid value enc
ountered in true_divide
  This is separate from the ipykernel package so we can avoid doing im
ports until
```

Aussi il est fréquent de multiplier le numérateur par une constante : généralement $-max(z)$:

$$\mathbf{a_i} = \frac{\mathbf{e^{z_i - max(z)}}}{\mathbf{\sum_{k=1}^{C} e^{z_k - max(z)}}}$$

In [7]:

```python
def softmax(z):
    expz = np.exp(z - np.max(z))
    return expz / expz.sum(axis=0, keepdims=True)

nums = np.array([4, 5, 6])
print(softmax(nums))
print ("la somme des probabilités donne 1")
nums = np.array([4000, 5000, 6000])
print(softmax(nums))
```

```
[0.09003057 0.24472847 0.66524096]
la somme des probabilités donne 1
[0. 0. 1.]
```

Dérivée de la fonction softmax

Elle est basée sur le fait de considérer la ré-écriture suivante :

$$\mathbf{g(x) = e^{z_i}}$$

et

$$\mathbf{h(x) = \sum_{k=1}^{C} e^{z_k}}$$

Nous savons que la dérivée d'une fonction $\mathbf{f(x) = \frac{g(x)}{h(x)}}$ est :

$$\mathbf{f'(x) = \frac{g'(x)h(x) - h'(x)g(x)}{h(x)^2}}$$

Par simplification, nous notons :

$$\mathbf{\sum_C = \sum_{k=1}^{C} e^{z_k}}$$

Pour $i = 1 \cdots C$, nous avons :

$$\mathbf{a_i = \frac{e^{z_i}}{\sum_C}}$$

La dérivée $\mathbf{\frac{\partial a_i}{\partial z_j}}$ de la sortie de softmax $\mathbf{a}$ par rapport à $\mathbf{z}$:

- Si $\mathbf{i = j}$,
$$\mathbf{\frac{\partial a_i}{\partial z_i} = \frac{\partial(\frac{e^{z_i}}{\sum_C})}{\partial z_i} = \frac{e^{z_i}\sum_C - e^{z_i}e^{z_i}}{\sum_C^2} = \frac{e^{z_i}}{\sum_C}\frac{\sum_C - e^{z_i}}{\sum_C} = \frac{e^{z_i}}{\sum_C}(1 - \frac{e^{z_i}}{\sum_C}) = a_i(1 - a_i)}$$
Remarque : il s'agit du même résultat que pour la dérivée de la sigmoid.
- Si $\mathbf{i \neq j}$,
$$\mathbf{\frac{\partial a_i}{\partial z_j} = \frac{\partial(\frac{e^{z_i}}{\sum_C})}{\partial z_j} = \frac{0 - e^{z_i}e^{z_j}}{\sum_C^2} = \frac{e^{z_i}}{\sum_C}\frac{e^{z_j}}{\sum_C} = -a_i a_j}$$

Dérivée par rapport à la fonction de coût

En appliquant le même principe que précédemment, on trouve :

$$\frac{\partial \mathbf{C}}{\partial \mathbf{Z}^{[\mathbf{L}]}} = \mathbf{A}^{[\mathbf{L}]} - \mathbf{y}$$

Donc toutes les dérivées précédentes pour $\mathbf{W}$ et $\mathbf{b}$ sont également similaires.

Il est par contre nécessaire de redéfinir la fonction de prédiction. Généralement lorsqu'il y a plusieurs classes, il convient de transformer le $\mathbf{y}$ initial en utilisant la fonction *OneHotEncoder*. Parfois, au préalable, il est indispensable de transformer les labels s'il s'agit d'attributs catégoriels en nombre via *LabelEncoder*. (Cf. notebook ingénierie des données).

In [8]:

```
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
import numpy as np

# Transformation du label
labelencoder = LabelEncoder()
exemple=np.array(["positif","negatif","positif","negatif"])
print ("Exemple ",exemple)
label_encoded=labelencoder.fit_transform(exemple)
print ("labels encodés :",label_encoded)


# Encodage via OneHotEncoder
onehot_encoder = OneHotEncoder(sparse=False,categories='auto')
#il est indispensable de faire un reshape

integer_encoded = label_encoded.reshape(len(label_encoded), 1)
final = onehot_encoder.fit_transform(integer_encoded)
print ("Encodage final :\n", final)
```

```
Exemple  ['positif' 'negatif' 'positif' 'negatif']
labels encodés : [1 0 1 0]
Encodage final :
 [[0. 1.]
 [1. 0.]
 [0. 1.]
 [1. 0.]]
```

Dans le cas de la prédiction, l'objectif est de retourner la classe qui a la plus forte probabilité à la sortie de softmax. Pour cela nous pouvons utiliser la fonction *argmax*.

In [9]:

```
exemple = np.array ([[0,2,25,4],
                     [1,11,8,10],
                     [200,7,5,10]])
print (exemple)

```

```
[[  0   2  25   4]
 [  1  11   8  10]
 [200   7   5  10]]
```

```
Position des différents éléments dans la matrice
                     #0      #1   #2   #3
exemple = np.array ([[0,     2,   25,   4],   #0
```

```
                  [1,   11,   8, 10],  #1
                  [200,  7,   5, 10]]  #2
                )
```

In [10]:

```
print ("\nPosition du plus grand élément : ",np.argmax(exemple))
print ("200 est le 8 ième élement (on compte à partir de 0)\n")

print ("Axis = 0 : la fonction cherche la valeur maximale sur les colonne de
print("\nIndice de l'élément max en considérant les colonnes : ", np.argmax(e
print ("colonne 0 : 200 est le plus grand de la colonne (retourne ligne 2)")
print ("colonne 1 : 11 est le plus grand de la colonne (retourne ligne 1)")
print ("colonne 2 : 25 est le plus grand de la colonne (retourne ligne 0)")
print ("colonne 3 : 10 est le plus grand de la colonne (retourne ligne 1)\n")

print ("Axis = 1 : la fonction cherche la valeur maximale sur les lignes de l
print("\nIndices of Max element : ", np.argmax(exemple, axis = 1))
print ("ligne 0 : 25 est le plus grand de la colonne (retourne colonne 2)")
print ("ligne 1 : 11 est le plus grand de la colonne (retourne colonne 1)")
print ("ligne 2 : 200 est le plus grand de la colonne (retourne colonne 0)")
```

```
Position du plus grand élément :  8
200 est le 8 ième élement (on compte à partir de 0)

Axis = 0 : la fonction cherche la valeur maximale sur les colonne de l
a matrice

Indice de l'élément max en considérant les colonnes :  [2 1 0 1]
colonne 0 : 200 est le plus grand de la colonne (retourne ligne 2)
colonne 1 : 11 est le plus grand de la colonne (retourne ligne 1)
colonne 2 : 25 est le plus grand de la colonne (retourne ligne 0)
colonne 3 : 10 est le plus grand de la colonne (retourne ligne 1)

Axis = 1 : la fonction cherche la valeur maximale sur les lignes de la
matrice

Indices of Max element :  [2 1 0]
ligne 0 : 25 est le plus grand de la colonne (retourne colonne 2)
ligne 1 : 11 est le plus grand de la colonne (retourne colonne 1)
ligne 2 : 200 est le plus grand de la colonne (retourne colonne 0)
```

Implémentations

Implémentation sous la forme de fonctions

Récupération des librairies utiles. Dans la mesure du possible il est préférable de les mettre au tout début.

In [11]:

```
# importations utiles
import numpy as np
from sklearn.datasets import make_moons
import matplotlib.pyplot as plt
import matplotlib
from sklearn.model_selection import train_test_split
from sklearn import linear_model
from matplotlib.legend_handler import HandlerLine2D
import keras
from keras.models import Sequential
from keras.layers import Dense
from keras.utils import np_utils
from keras import regularizers
from keras.optimizers import SGD
from sklearn.metrics import accuracy_score

matplotlib.rcParams['figure.figsize'] = (6.0, 6.0) # pour avoir des figures d
```

```
Using TensorFlow backend.
```

Dans un premier temps il est nécessaire de construire le réseau en indiquant le nombre de layers, de neurones par layer et les fonctions d'activation. Nous stockons ici cette information sous la forme d'un dictionnaire python.

In [12]:

```
#reseau de l'exemple du notebook
# il contient trois layer : input - hidden - output
#    par défaut on ne spécifie pas la couche d'entrée (couche 0) qui contient
#    hidden layer : elle reçoit en entrée les variables prédictives X avec de
#               (input_dim=2) elle contient 5 neurones, la fonction d'activati
#    output layer : les dimensions d'entrée = 5 (il s'agit de la sortie de l'
#                  elle donne le résultat. Ici la fonction d'activation est
#                  classification binaire
#

layers = [
    {"input_dim": 2, "output_dim": 3, "activation": "relu"},
    {"input_dim": 3, "output_dim": 1, "activation": "sigmoid"}
]


```

La fonction suivante permet de créer le réseau à partir du dictionnaire passé en paramètre. Elle initialise avec un nombre aléatoire les poids et le biais. Les différents paramètres du réseau (poids) et biais sont stockés sous la forme d'un dictionnaire python avec une clé qui indentifie le layer auquel il appartient.

In [13]:

```
def init_layers(layers):

    seed=30
    np.random.seed(seed)

    # nombre de layers dans le réseau
    number_of_layers = len(layers)
    # pour stocker les différentes valeurs des paramètres

    paramameters = {}

    # Pour toutes les couches du réseau
    for idx, layer in enumerate(layers):
        # Par simplification on commence la numérotation du layer à 1
        # correspond aux données d'entrées (input), i.e. les variables prédic
        layer_idx = idx + 1

        layer_input_size = layer["input_dim"]
        layer_output_size = layer["output_dim"]

        # Initialisation des valeurs de la matrice W et du vecteur b
        # pour les différentes couches

        paramameters['W' + str(layer_idx)] = np.random.randn(
            layer_output_size, layer_input_size) * 0.1
        paramameters['b' + str(layer_idx)] = np.random.randn(
            layer_output_size, 1) * 0.1

    return paramameters
```

Définition des différentes fonctions d'activation (Relu, Sigmoid, Tanh) ainsi que les dérivées qui seront utilisées par la suite.

In [14]:

```
def sigmoid(Z):
    return 1/(1+np.exp(-Z))

def relu(Z):
    return np.maximum(0,Z)

def tanh(Z):
    return np.tanh(Z)

def sigmoid_backward(dA, Z):
    sig = sigmoid(Z)
    return dA * sig * (1 - sig)

def relu_backward(dA, Z):
    dZ = np.array(dA, copy = True)# pour ne pas effacer dA
    dZ[Z <= 0] = 0;
    return dZ;

def tanh_backward (dA, Z):
     return 1- dA**2


```

La fonction suivante réalise la forward propagation mais uniquement d'un layer. Elle effectue donc le produit matriciel avec ajout du biais pour obtenir Z. Elle applique ensuite la fonction d'activation du layer.

In [15]:

```
def one_layer_forward_propagation(A_prev, W_curr, b_curr, activation="relu"):

    # regression linéaire sur les entrées du layer
    Z_curr = np.dot(W_curr, A_prev) + b_curr

    # selection de la fonction d'activation à utiliser dans la couche
    if activation == "relu":
        activation_func = relu
    elif activation == "sigmoid":
        activation_func = sigmoid
    elif activation == "tanh":
        activation_func = tanh

    # A est la sortie de la fonction d'activation
    A=activation_func(Z_curr)
    # retourne la fonction d'activation calculée et la matrice intermédiaire
    return A, Z_curr
```

La fonction suivante fait la forward propagation sur tout le réseau. Elle sauvegarde aussi les valeurs de A et Z dans un dictionnaire python indexé par ces lettres afin de pouvoir les retrouver facilement lors de l'étape de backpropagation.

In [16]:

```
def forward_propagation(X, parameters, layers):

    # Création d'un cache temporaire qui contient les valeurs intermédiaires
    # utiles lors de la phase de backward. Le fait de les sauvegarder permet
    # de ne pas les recalculer lors du backward
    cache = {}

    # A_curr correspond à la sortie du layer 0, i.e. les variables prédictive
    A_curr = X

    # iteration pour l'ensemble des couches du réseau
    for idx, layer in enumerate(layers):
        # La numérotation des couches commence à 1 (0 pour la couche des donn
        layer_idx = idx + 1

        # Recupération de l'activation de l'itération précédente
        A_prev = A_curr

        # Récupération du nom de la fonction d'activation du layer courant
        activ_function_curr = layer["activation"]

        # Récupération du W et du b du layer courant
        W_curr = parameters["W" + str(layer_idx)]
        b_curr = parameters["b" + str(layer_idx)]


        # calcul de la fonction d'activation pour le layer courant
        A_curr, Z_curr = one_layer_forward_propagation(A_prev, W_curr, b_curr

        # Sauvegarde dans le cache pour la phase de backward
        cache["A" + str(idx)] = A_prev
        cache["Z" + str(layer_idx)] = Z_curr

    # retourne le vecteur de prédiction à la sortie du réseau
    # et un dictionnaire contenant toutes les valeurs intermédiaire
    # pour faciliter la descente de gradient

    return A_curr, cache
```

Le calcul de la fonction de coût (ici la cross entropy). y_hat correspond à la sortie du réseau après application de la forward propagation. y est la valeur réelle.

In [17]:

```
def cost_function(y_hat, y):
    # calcul du coût (cross-entropy)
    cost = (-y * np.log(y_hat) - (1 - y) * np.log(1 - y_hat)).mean()
    return cost
```

Les deux fonctions suivantes permettent de calculer l'accuracy du modèle. Elles suivent le même principe que celles vues dans le notebook descente de gradient.

In [18]:

```
def convert_prob_into_class(probs):
    probs = np.copy(probs)#pour ne pas perdre probs
    probs[probs > 0.5] = 1
    probs[probs <= 0.5] = 0
    return probs

def accuracy(y_hat, y):
    y_hat_ = convert_prob_into_class(y_hat)
    return (y_hat_ == y).all(axis=0).mean()
```

Propagation d'un niveau. Tout d'abord nous considérons la descente de gradient sur un niveau. Elle applique tout d'abord la dérivée de la fonction d'activation passée en paramètre, dW, db et donc la dérivée de Z pour un layer L.

In [19]:

```
def one_layer_backward_propagation(dA_curr, W_curr, b_curr, Z_curr, A_prev, a

    # nombres d'exemples venant de la fonction d'activation précédente
    m = A_prev.shape[1]

    # selection de la fonction d'activation à appliquer
    if activation =="relu":
        backward_activation_func = relu_backward
    elif activation == "sigmoid":
        backward_activation_func = sigmoid_backward
    elif activation == "tanh":
        backward_activation_func = tanh_backward

    # calcul de la dérivée de la fonction d'activation
    dZ_curr = backward_activation_func(dA_curr, Z_curr)

    # dérivée de la matrice W
    dW_curr = np.dot(dZ_curr, A_prev.T) / m
    # dérivée du vecteur b
    db_curr = np.sum(dZ_curr, axis=1, keepdims=True) / m
    # dérivée de la matrice A_Prev
    dA_prev = np.dot(W_curr.T, dZ_curr)

    return dA_prev, dW_curr, db_curr
```

La propagation sur tout le réseau commence par calculer la dérivée de la fonction de coût :

$$\mathbf{\frac{\partial L}{\partial A} = \left(\frac{-y}{A} + \frac{(1-y)}{(1-A)}\right)}$$

ou

$$\mathbf{-\left(\frac{y}{A} - \frac{(1-y)}{(1-A)}\right)}$$

et boucle sur les autres niveaux. Le cache est utilisé pour récupérer les valeurs de A et de Z en fonction de leur layer et ce cache est utilisé pour sauvegarder dW et db qui seront utilisés lors de la descente de gradient.

In [20]:

```python
def backward_propagation(y_hat, y, cache, parameters, layers):


    # Création d'un cache temporaire qui contient les dérivées (gradients) po
    # les différentes couches. Il est utilisé pour mettre à jour les paramètr
    derivatives = {}

    # nombre d'exemples
    m = y.shape[1]

    # pour garantir que y a la même forme que y_hat
    y = y.reshape(y_hat.shape)

    #  Initialisation du calcul de la dérivée de la fonction de coût
    # par rapport à A pour la couche L
    dA_prev = - (np.divide(y, y_hat) - np.divide(1 - y, 1 - y_hat))

    # parcours du réseau de la fonction finale vers celle d'entrée
    for layer_idx_prev, layer in reversed(list(enumerate(layers))):
        # Comme précédemment la numérotation des couches commence
        # à 1. On ne modifie donc pas la couche 0
        layer_idx_curr = layer_idx_prev + 1

        # Récupération du nom de la fonction d'activation du layer courant
        activ_function_curr = layer["activation"]

        dA_curr = dA_prev

        # Récupération dans le cache de la sortie précédente (A_prev)
        # et de la matrice Z correspondant à l'application de la
        # regression linéaire du niveau courant. Ceci évite de les recalculer
        A_prev = cache["A" + str(layer_idx_prev)]
        Z_curr = cache["Z" + str(layer_idx_curr)]

        # Récupération dans parameters des valeurs de paramètres W et b du la
        W_curr = parameters["W" + str(layer_idx_curr)]
        b_curr = parameters["b" + str(layer_idx_curr)]

        #application de la backwart propagation pour le layer afin d'avoir
        #la valeur des dérivées (gradients)
        dA_prev, dW_curr, db_curr = one_layer_backward_propagation(
            dA_curr, W_curr, b_curr, Z_curr, A_prev, activ_function_curr)

        # sauvegarde pour mettre à jour les paramètres
        derivatives["dW" + str(layer_idx_curr)] = dW_curr
        derivatives["db" + str(layer_idx_curr)] = db_curr

    return derivatives
```

Application de la descente de gradient pour mettre à jour les paramètres. Comme tous les gradients ont été calculés précédemment (et sauvegardés dans un dictionnaire), il suffit de les appliquer. Ici la descente est faite à la manière d'une descente par lot (*batch gradient descent*) et peut être facilement modifiée en mini-batch gradient descent (C.f. notebook descente de gradient) avec optimisation.

In [21]:

```
def update(parameters, derivatives, layers, eta):

    # Mise à jour des paramètres sur les différentes couches
    for layer_idx, layer in enumerate(layers, 1):
        parameters["W" + str(layer_idx)] -= eta * derivatives["dW" + str(laye
        parameters["b" + str(layer_idx)] -= eta * derivatives["db" + str(laye

    return parameters
```

La fonction train permet de lancer les différentes phases en fonction du nombre d'epochs. Elle retourne l'historique du coût et de l'accuracy pour pouvoir les afficher.

In [22]:

```
def fit(X, y, layers, epochs, eta):

    # Initialisation des paramètres du réseau de neurones
    parameters = init_layers(layers)

    # sauvegarde historique coût et accuracy pour affichage
    cost_history = []
    accuracy_history = []

    # Descente de gradient
    for i in range(epochs):

        # forward progragation
        y_hat, cache = forward_propagation(X, parameters, layers)

        # backward propagation - calcul des gradients
        derivatives = backward_propagation(y_hat, y, cache, parameters, layer

        # Mise à jour des paramètres
        parameters = update(parameters, derivatives, layers, eta)

        # sauvegarde des historiques
        current_cost=cost_function(y_hat, y)
        cost_history.append(current_cost)
        curent_accuracy = accuracy(y_hat, y)
        accuracy_history.append(curent_accuracy)

        if(i % 100 == 0):
            print("Epoch : #%s - cost : %.3f - accuracy : %.3f"%(i, float(cur

    return parameters,cost_history,accuracy_history
```

Premier test avec la configuration de l'exemple

In [23]:

```
def plot_histories (eta,epochs,cost_history,accuracy_history):
    fig,ax = plt.subplots(figsize=(5,5))
    ax.set_ylabel(r'$J(\theta)$')
    ax.set_xlabel('Epochs')
    ax.set_title(r"$\eta$ :{}".format(eta))
    line1, = ax.plot(range(epochs),cost_history,label='Cost')
    line2, = ax.plot(range(epochs),accuracy_history,label='Accuracy')
    plt.legend(handler_map={line1: HandlerLine2D(numpoints=4)})
```

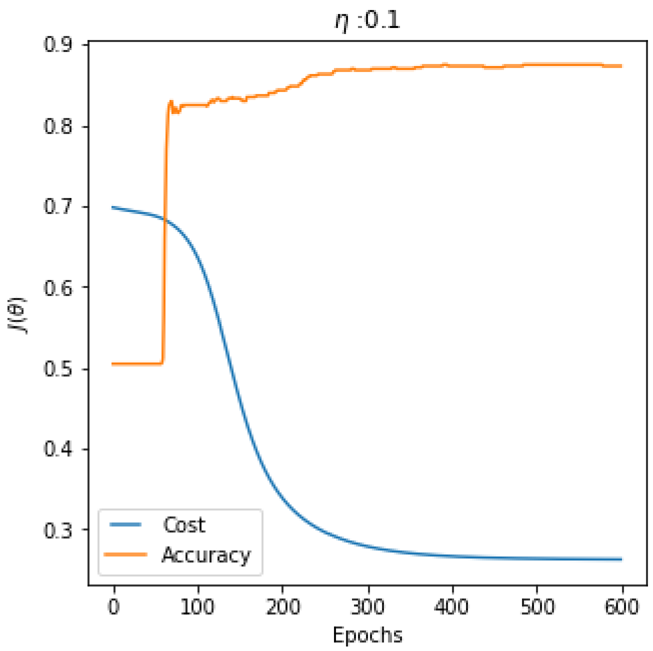
In [24]:

```
X,y = make_moons(n_samples=1000, noise=0.1)


validation_size=0.6 #40% du jeu de données pour le test

testsize= 1-validation_size
seed=30
# séparation jeu d'apprentissage et jeu de test
X_train,X_test,y_train,y_test=train_test_split(X,
                                               y,
                                               train_size=validation_size,
                                               random_state=seed,
                                               test_size=testsize)

# sauvegarde pour comparaison avec Keras
X_train_init=X_train
X_test_init=X_test
y_train_init=y_train
y_test_init=y_test

#transformation des données pour être au bon format
X_train=np.transpose(X_train)
X_test=np.transpose(X_test)
y_train=np.transpose(y_train.reshape((y_train.shape[0], 1)))
y_test=np.transpose(y_test.reshape((y_test.shape[0], 1)))
```

In [25]:

```
epochs = 600
eta = 0.1

parameters,cost_history,accuracy_history = fit(X_train,
                                               y_train,
                                              layers, epochs, eta)


# Calcul de l'accuracy sur le jeu de test
y_test_hat,_= forward_propagation(X_test, parameters, layers)
accuracy_test = accuracy(y_test_hat, y_test)
print("Accuracy : %.3f"%accuracy_test)

plot_histories (eta,epochs,cost_history,accuracy_history)
```

```
Epoch : #0 - cost : 0.698 - accuracy : 0.505
Epoch : #100 - cost : 0.636 - accuracy : 0.825
Epoch : #200 - cost : 0.339 - accuracy : 0.843
Epoch : #300 - cost : 0.279 - accuracy : 0.868
Epoch : #400 - cost : 0.267 - accuracy : 0.873
Epoch : #500 - cost : 0.264 - accuracy : 0.875
Accuracy : 0.887
```

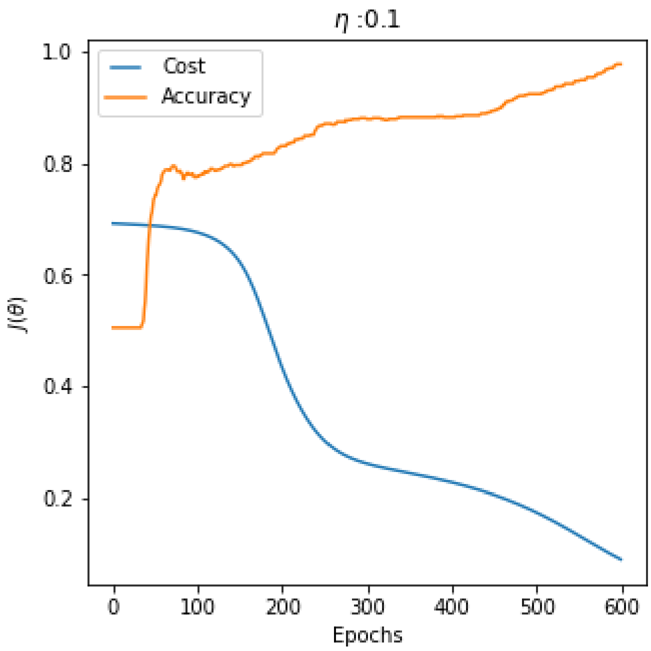


Test en modifiant le réseau pour voir si le résultat est meilleur. Attention au surapprentissage dans ce cas.

In [26]:

```
layers = [
    {"input_dim": 2, "output_dim": 25, "activation": "relu"},
    {"input_dim": 25, "output_dim": 25, "activation": "relu"},
    {"input_dim": 25, "output_dim": 25, "activation": "relu"},
    {"input_dim": 25, "output_dim": 1, "activation": "sigmoid"},
]


epochs = 600
eta = 0.1
parameters,cost_history,accuracy_history = fit(X_train,
                                               y_train,
                                              layers, epochs, eta)

# Calcul de l'accuracy sur le jeu de test
y_test_hat,_= forward_propagation(X_test, parameters, layers)
accuracy_test = accuracy(y_test_hat, y_test)
print("Accuracy : %.3f"%accuracy_test)

plot_histories (eta,epochs,cost_history,accuracy_history)
```

```
Epoch : #0 - cost : 0.692 - accuracy : 0.505
Epoch : #100 - cost : 0.676 - accuracy : 0.778
Epoch : #200 - cost : 0.432 - accuracy : 0.830
Epoch : #300 - cost : 0.261 - accuracy : 0.878
Epoch : #400 - cost : 0.228 - accuracy : 0.882
Epoch : #500 - cost : 0.174 - accuracy : 0.923
Accuracy : 0.983
```



Essai avec Keras pour voir les résulats.

In [27]:

```
#Construction du modèle
model = Sequential()
model.add(Dense(25, input_dim=2,activation='relu'))
model.add(Dense(25, activation='relu'))
model.add(Dense(25, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
epochs = 600
eta = 0.1

gd = SGD(lr=eta)
model.compile(loss='binary_crossentropy', optimizer="sgd", metrics=['accuracy


# Training
history = model.fit(X_train_init, y_train_init, epochs=600, verbose=0)


```

In [28]:

```
y_test_hat = model.predict_classes(X_test_init)
accuracy_test = accuracy_score(y_test_init, y_test_hat)
print("Accuracy pour Keras : %.3f"%accuracy_test)
```

```
Accuracy pour Keras : 1.000
```

Implémentation avec une classe simple

Dans un premier temps, nous créons une classe simple qui reprend les fonctions précédentes et dans laquelle nous ajoutons simplement les fonctions d'activation Relu, LeakyRelu, Tanh, Sigmoid et Softmax. Cette classe permet de faire de la classification multi-classes.

Récupération des librairies utiles

In [29]:

```python
# importations utiles
import numpy as np
from sklearn.datasets import make_moons
from sklearn.datasets import make_circles
from sklearn import datasets
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import OneHotEncoder
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt
import matplotlib
from sklearn.model_selection import train_test_split
from sklearn import linear_model
from matplotlib.legend_handler import HandlerLine2D
import keras
from keras.models import Sequential
from keras.layers import Dense
from keras.utils import np_utils
from keras import regularizers
from keras.optimizers import SGD
from sklearn.metrics import accuracy_score
import pandas as pd
import seaborn as sns
import numpy as np
matplotlib.rcParams['figure.figsize'] = (6.0, 6.0) # pour avoir des figures d
```

Fonctions d'affichage.

In [30]:

```
def plot_histories (eta,epochs,cost_history,accuracy_history):
    fig,ax = plt.subplots(figsize=(5,5))
    ax.set_ylabel(r'$J(\theta)$')
    ax.set_xlabel('Epochs')
    ax.set_title(r"$\eta$ :{}".format(eta))
    line1, = ax.plot(range(epochs),cost_history,label='Cost')
    line2, = ax.plot(range(epochs),accuracy_history,label='Accuracy')
    plt.legend(handler_map={line1: HandlerLine2D(numpoints=4)})

def plot_decision_boundary(func, X, y):
    amin, bmin = X.min(axis=0) - 0.1
    amax, bmax = X.max(axis=0) + 0.1
    hticks = np.linspace(amin, amax, 101)
    vticks = np.linspace(bmin, bmax, 101)

    aa, bb = np.meshgrid(hticks, vticks)
    ab = np.c_[aa.ravel(), bb.ravel()]
    c = func(ab)
    cc = c.reshape(aa.shape)

    cm = plt.cm.RdBu
    cm_bright = ListedColormap(['#FF0000', '#0000FF'])

    fig, ax = plt.subplots()
    contour = plt.contourf(aa, bb, cc, cmap=cm, alpha=0.8)

    ax_c = fig.colorbar(contour)
    ax_c.set_label("$P(y = 1)$")
    ax_c.set_ticks([0, 0.25, 0.5, 0.75, 1])

    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cm_bright)
    plt.xlim(amin, amax)
    plt.ylim(bmin, bmax)
    plt.title("Decision Boundary")
```

Fonctions d'activations et dérivées des fonctions : sigmoid, tanh, relu, leakyrelu et softmax.

In [31]:

```
#fonctions utiles
def sigmoid(x):
    return 1/(1 + np.exp(-x))

def sigmoid_prime(x):
    return sigmoid(x)*(1.0 - sigmoid(x))

def tanh(x):
    return np.tanh(x)

def tanh_prime(x):
    return 1 - x ** 2

def relu(x):
    return np.maximum(0,x)

def relu_prime(x):
    x[x<=0] = 0
    x[x>0] = 1
    return x

def leakyrelu(x):
    return np.maximum(0.01,x)

def leakyrelu_prime(x):
    x[x<=0] = 0.01
    x[x>0] = 1
    return x

def softmax(x):
    expx = np.exp(x - np.max(x))
    return expx / expx.sum(axis=0, keepdims=True)
```

Nous définissons, à présent, une classe Layer. Son objectif est de pouvoir être utilisée de la même manière que dans Keras. Un layer peut être ajouté à notre réseau. Ce dernier va contenir le nombre de neurones de la couche ainsi que la fonction d'activation de la couche.
La première couche doit spécifier le nombre de neurones de la couche d'entrée.

Par exemple :

- *Layer(10,input=4,activation="leakyrelu")*

aura pour effet de créer un layer contenant 10 neurones, dont la couche d'entrée contient 4 neurones et pour lequel la fonction d'activation est leakyrelu.

Pour les autres couches il suffit de spécifier le nombre de couches et la fonction d'activation :

- *Layer(10,activation="sotfmax")*

ajoutera une couche contenant 10 neurones et dont la fonction d'activation est softmax (ici il s'agit de la dernière couche du réseau).

Les différents initialisations sont créées dans le constructeur de la classe Layer. La méthode *initParams* sera appelée par une méthode de la classe suivante et aura pour effet d'initaliser les valeurs de W et de b avec des nombres aléatoires. W est initialisé en prenant en compte l'optimisation de [He et al 2015] (Source: [https://arxiv.org/pdf/1502.01852.pdf) (https://arxiv.org/pdf/1502.01852.pdf)](https://arxiv.org/pdf/1502.01852.pdf).

In [32]:

```
class Layer:
    def __init__(self,output,*args,**kwargs):

        self.output = output # Number of neurons at  layer i (current layer)
        self.input = kwargs.get("input",None) # Number of neurons at layer i-
        self.activ_function_curr = kwargs.get("activation",None) # Activation
        self.parameters ={}
        self.derivatives={}
        self.activation_func=None
        if self.activ_function_curr == "relu":
            self.activation_func = relu
            self.backward_activation_func = relu_prime
        elif self.activ_function_curr == "sigmoid":
            self.activation_func = sigmoid
            self.backward_activation_func = sigmoid_prime
        elif self.activ_function_curr == "tanh":
            self.activation_func = tanh
            self.backward_activation_func = tanh_prime
        elif self.activ_function_curr == "leakyrelu":
            self.activation_func = leakyrelu
            self.backward_activation_func = leakyrelu_prime
        elif self.activ_function_curr == "softmax":
            self.activation_func = softmax
            self.backward_activation_func = softmax

    def initParams(self):
        # initialisation du dictionnaire de données parameters contenant W, A
        seed=30
        np.random.seed(seed)
        self.parameters['W']=np.random.randn(self.output,self.input)*np.sqrt(
        self.parameters['b']=np.random.randn(self.output,1)*0.1

    def setW(self,matW):
        self.parameters['W']=np.copy(matW)

    def setA(self,matA):
        self.parameters['A']=np.copy(matA)

    def setZ(self,matZ):
        self.parameters['Z']=np.copy(matZ)

    def setB(self,matB):
        self.parameters['b']=np.copy(matB)

    def setdW(self,matdW):
        self.parameters['dW']=np.copy(matdW)

    def setdA(self,matdA):
        self.parameters['dA']=np.copy(matdA)

    def setdZ(self,matdZ):
        self.parameters['dZ']=np.copy(matdZ)

    def setdB(self,matdB):
        self.parameters['db']=np.copy(matdB)

```

Définition de la classe MyNeuralNetwork.

L'ajout de couche se fait à l'aide de la méthode *addLayer*. Cette fonction permet de mettre à jour les objets de la classe Layer.
Par exemple :

network = MyNeuralNetwork()

network.addLayer(Layer(3,input=2,activation="relu"))
network.addLayer(Layer(1,activation="sigmoid"))

permet de créer un réseau contenant 2 couches. La première est composée de 3 neurones, la couche d'entrée est composée de 2 neurones, elle utilise la fonction d'activation relu. La seconde couche, dernière couche, contient 1 neurone et la fonction d'activation est une sigmoid.

La classe est assez similaire à l'approche par fonctions. La principale différence est dans les fonctions forward et backward qui traitent différemment le cas des fonctions d'activations. Ici la descente de gradient se fait par mini-batch : d'où appel à la fonction *next_batch* vue dans le notebook descente de gradient qui retourne un batch de la taille de batchsize.

La classe contient différentes méthodes supplémentaires :

- *info* qui donne les informations sur les différentes couches (nombre de neurones par couche, valeurs des W et des b lors de l'initialisation
- *set_parametersW_b (numlayer,matX,matb)* qui permet de mettre des valeurs d'initialisation aux W et b pour une couche. Les valeurs doivent être sous la forme d'une matrice pour W et d'un vecteur pour b.
- *plot_W_b_epoch (epoch,parameter_history)* qui permet de connaître les valeurs de W (visualisation) à une epoch donnée. Elle utilise l'historique d'exécution du réseau.

Les différentes valeurs de A,Z, W, b ainsi que quelques statistiques pour chaque valeur d'epoch sont sauvegardées dans un tableau de dictionnaire.

L'entrainement du classifieur se fait à l'aide de la fonction fit :
Les premières valeurs correspondent à X et y, ensuite les paramètre sont (sans ordre particulier) :

- "epochs". Par défaut 20.
- "verbose"=(True|False). Par défaut False, qui affiche le coût et l'accuracy pour chaque epoch.
- "eta". Par défaut 0.01.
- "batchsize". Par défaut 32.

Les valeurs de retour sont :

- la liste des W et b à la fin de l'apprentissage (retour de la variable layer).
- les historiques des cost et accuracy pour chaque epoch afin de faire de pouvoir afficher l'historique.
- l'historique des A, Z, b, W pour chaque epoch.

Exemple d'utilisation :

layers,cost_history,accuracy_history,parameter_history=network.fit(X_train, y_train, verbose=True, epochs=100, eta=0.1, batchsize=128)

permet de lancer le classifieur avec comme paramètres X_train et y_train, en affichant les valeurs de cost et d'accuracy pour chaque epoch (verbose=True), sur 100 epochs (epochs=100), avec un learning rate de 0.1 (eta=0.1) et avec des batchsize de 128 (batchsize=128).

In [33]:

```
class MyNeuralNetwork:
    def __init__(self):
        self.nbLayers=0
        self.layers=[]

    def info(self):
        print("Content of the network:");
        j=0;
        for i in range(len(self.layers)):
            print("Layer n° ",i," => ")
            print ("\tInput ", self.layers[i].input,
                   "\tOutput", self.layers[i].output)
            if (i != 0):
                print ("\tActivation Function",self.layers[i].activation_func
                print ("\tW", self.layers[i].parameters['W'].shape,self.layer
                print ("\tb", self.layers[i].parameters['b'].shape,self.layer


    def addLayer(self,layer):
        self.nbLayers += 1;
        if (self.nbLayers==1):
            # this is the first layer so adding a layer 0
            layerZero=Layer(layer.input)
            self.layers.append(layerZero)

        self.layers.append(layer)
        self.layers[self.nbLayers].input=self.layers[self.nbLayers-1].output
        self.layers[self.nbLayers].output=self.layers[self.nbLayers].output
        layer.initParams()



    def set_parametersW_b (self,numlayer,matX,matb):
        self.layers[numlayer].parameters['W']=np.copy(matX)
        self.layers[numlayer].parameters['b']=np.copy(matb)


    def forward_propagation(self, X):
        #Init predictive variables for the input layer
        self.layers[0].setA(X)

        #Propagation for all the layers
        for l in range(1, self.nbLayers + 1):
            # Compute Z
            self.layers[l].setZ(np.dot(self.layers[l].parameters['W'],
                                       self.layers[l-1].parameters['A'])+sel
            # Applying the activation function of the layer to Z
            self.layers[l].setA(self.layers[l].activation_func(self.layers[l


    def cost_function(self,y):
        return (-(y*np.log(self.layers[self.nbLayers].parameters['A']+1e-8) -

    def backward_propagation(self,y):
        #calcul de dZ dW et db pour le dernier layer
        self.layers[self.nbLayers].derivatives['dZ']=self.layers[self.nbLaye
        self.layers[self.nbLayers].derivatives['dW']=np.dot(self.layers[self.
                                                              np.transpose(se
        m=self.layers[self.nbLayers].parameters['A'].shape[1]#égal au nombre
```

```
            self.layers[self.nbLayers].derivatives['db']=np.sum(self.layers[self.
                                                               axis=1, keepdims=True)

            #calcul de dZ dW db pour les autres layers
            for l in range(self.nbLayers-1,0,-1) :
                self.layers[l].derivatives['dZ']=np.dot(np.transpose(self.layers[
                                                self.layers[l+1].derivatives['dZ

                self.layers[l].derivatives["dW"]=np.dot(self.layers[l].derivative
                                                np.transpose(self.layers[l-1].par

                m=self.layers[l-1].parameters['A'].shape[1]#égal au nombre de co
                self.layers[l].derivatives['db']=np.sum(self.layers[l].derivative
                                                           axis=1, keepdims=True)

        def update_parameters(self, eta) :
            for l in range(1,self.nbLayers+1) :
                self.layers[l].parameters['W']-=eta*self.layers[l].derivatives['d
                self.layers[l].parameters["b"]-=eta*self.layers[l].derivatives["d

        def convert_prob_into_class(self,probs):
            probs = np.copy(probs)#pour ne pas perdre probs, i.e. y_hat
            probs[probs > 0.5] = 1
            probs[probs <= 0.5] = 0
            return probs

        def plot_W_b_epoch (self,epoch,parameter_history):
            mat=[]
            max_size_layer=0
            for l in range(1, self.nbLayers+1):
                value=parameter_history[epoch]['W'+str(l)]
                if (parameter_history[epoch]['W'+str(l)].shape[1]>max_size_layer)
                    max_size_layer=parameter_history[epoch]['W'+str(l)].shape[1]
                mat.append(value)
            figure=plt.figure(figsize=((self.nbLayers+1)*3,int (max_size_layer/2)
            for nb_w in range (len(mat)):
                    plt.subplot(1, len(mat), nb_w+1)
                    plt.matshow(mat[nb_w],cmap = plt.cm.gist_rainbow,fignum=False
                    plt.colorbar()
            thelegend="Epoch "+str(epoch)
            plt.title (thelegend)

        def accuracy(self,y_hat, y):
            if self.layers[self.nbLayers].activation_func==softmax:
                # si la fonction est softmax, les valeurs sont sur différentes d
                # il faut utiliser argmax avec axis=0 pour avoir un vecteur qui 
                # où est la valeur maximale à la fois pour y_hat et pour y
                # comme cela il suffit de comparer les deux vecteurs qui indiquen
                # dans quelle ligne se trouve le max
                y_hat_encoded=np.copy(y_hat)
                y_hat_encoded = np.argmax(y_hat_encoded, axis=0)
                y_encoded=np.copy(y)
                y_encoded=np.argmax(y_encoded, axis=0)
                return (y_hat_encoded == y_encoded).mean()
            # la dernière fonction d'activation n'est pas softmax.
            # par exemple sigmoid pour une classification binaire
            # il suffit de convertir la probabilité du résultat en classe
            y_hat_ = self.convert_prob_into_class(y_hat)
            return (y_hat_ == y).all(axis=0).mean()

        def predict(self, x):
```

```
        self.forward_propagation(x)
        return self.layers[self.nbLayers].parameters['A']

    def next_batch(self,X, y, batchsize):
        # pour avoir X de la forme : 2 colonnes, m lignes (examples) et égal
        # cela permet de trier les 2 tableaux avec un indices de permutation
        X=np.transpose(X)
        y=np.transpose(y)

        m=len(y)
        # permutation aléatoire de X et y pour faire des batchs avec des val
        indices = np.random.permutation(m)
        X = X[indices]
        y = y[indices]
        for i in np.arange(0, X.shape[0], batchsize):
            # creation des batchs de taille batchsize
            yield (X[i:i + batchsize], y[i:i + batchsize])
    def fit(self, X, y, *args,**kwargs):
        epochs=kwargs.get("epochs",20)
        verbose=kwargs.get("verbose",False)
        eta =kwargs.get("eta",0.01)
        batchsize=kwargs.get("batchsize",32)
    #def fit(self, X, y, epochs, eta = 0.01,batchsize=64) :
        # sauvegarde historique coût et accuracy pour affichage
        cost_history = []
        accuracy_history = []
        parameter_history = []
        for i in range(epochs):
            i+=1
            # sauvegarde des coûts et accuracy par mini-batch
            cost_batch = []
            accuracy_batch = []
            # Descente de gradient par mini-batch
            for (batchX, batchy) in self.next_batch(X, y, batchsize):
                # Extraction et traitement d'un batch à la fois

                # mise en place des données au bon format
                batchX=np.transpose(batchX)
                if self.layers[self.nbLayers].activation_func==softmax:
                    # la classification n'est pas binaire, y a utilisé one-h
                    # le batchy doit donc être transposé et le résultat doit
                    # être sous la forme d'une matrice de taille batchy.shap

                    batchy=np.transpose(batchy.reshape((batchy.shape[0], bat
                else:
                    # il s'agit d'une classification binaire donc shape[1] n
                    # pas
                    batchy=np.transpose(batchy.reshape((batchy.shape[0], 1))
                #batchy=np.transpose(batchy.reshape((batchy.shape[0], 1)))
                self.forward_propagation(batchX)
                self.backward_propagation(batchy)
                self.update_parameters(eta)

                # sauvegarde pour affichage
                current_cost=self.cost_function(batchy)
                cost_batch.append(current_cost)
                y_hat = self.predict(batchX)
                current_accuracy = self.accuracy(y_hat, batchy)
                accuracy_batch.append(current_accuracy)

            # SaveStats on W, B as well as values for A,Z, W, b
```

```
            save_values = {}
            save_values["epoch"]=i
            for l in range(1, self.nbLayers+1):
                save_values["layer"+str(l)]=l
                save_values["Wmean"+ str(l)]=np.mean(self.layers[self.nbLaye
                save_values["Wmax"+ str(l)]=np.amax(self.layers[self.nbLayer
                save_values["Wmin"+str(l)]=np.amin(self.layers[self.nbLayers
                save_values["Wstd"+str(l)]=np.std(self.layers[self.nbLayers]
                save_values["bmean"+ str(l)]=np.mean(self.layers[self.nbLaye
                save_values["bmax"+ str(l)]=np.amax(self.layers[self.nbLayer
                save_values["bmin"+str(l)]=np.amin(self.layers[self.nbLayers
                save_values["bstd"+str(l)]=np.std(self.layers[self.nbLayers]
                # be careful A,Z,W and b must be copied otherwise it is a re
                save_values['A'+str(l)]=np.copy(self.layers[self.nbLayers].pa
                save_values['Z'+str(l)]=np.copy(self.layers[self.nbLayers].pa
                save_values['W'+str(l)]=np.copy(self.layers[self.nbLayers].pa
                save_values['b'+str(l)]=np.copy(self.layers[self.nbLayers].pa

            parameter_history.append(save_values)
            # sauvegarde de la valeur moyenne des coûts et de l'accuracy du l
            current_cost=np.average(cost_batch)
            cost_history.append(current_cost)
            current_accuracy=np.average(accuracy_batch)
            accuracy_history.append(current_accuracy)

            if(verbose == True):
                print("Epoch : #%s/%s - %s/%s - cost : %.4f - accuracy : %.4

        return self.layers, cost_history, accuracy_history, parameter_history

```

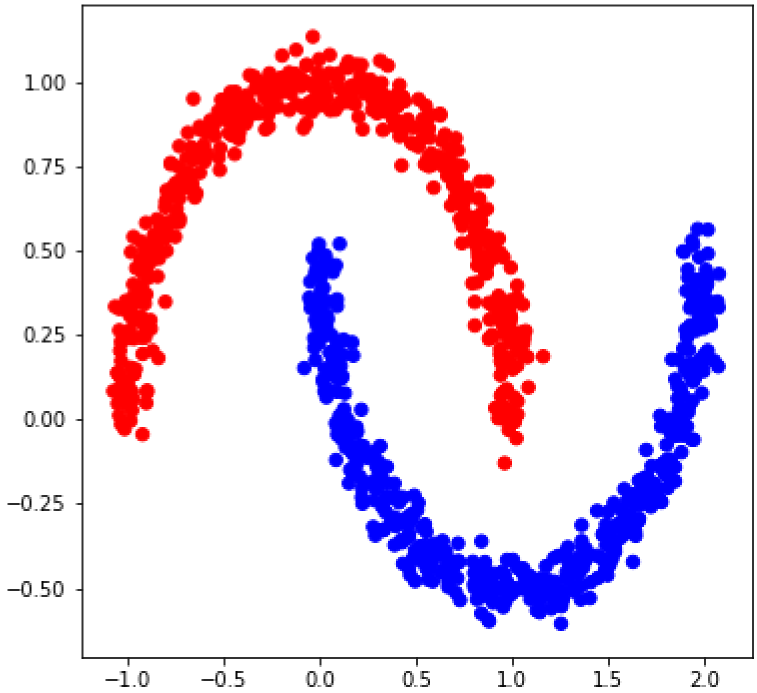
Utilisation de la classe MyNeural Network pour une classification binaire.

In [34]:

```
X, y = make_moons(n_samples=1000, noise=0.05, random_state=0)
cm_bright = ListedColormap(['#FF0000', '#0000FF'])
plt.scatter(X[:,0], X[:,1], s=40, c=y, cmap=cm_bright)#cmap=plt.cm.PiYG)
```

Out[34]:

```
<matplotlib.collections.PathCollection at 0x12d1a3748>
```



Création d'un jeu de données d'apprentissage et de test.

In [35]:

```
#création d'un jeu d'apprentissage et de test

validation_size=0.6 #40% du jeu de données pour le test

testsize= 1-validation_size
seed=30
# séparation jeu d'apprentissage et jeu de test
X_train,X_test,y_train,y_test=train_test_split(X,
                                               y,
                                               train_size=validation_size,
                                               random_state=seed,
                                               test_size=testsize)
```

Attention, les variables prédictives X doivent être sous la forme m colonnes où m est le nombre d'exemples d'apprentissage et l lignes où l est le nombre de variables prédictives. Dans notre exemple du notebook, il y a deux variables prédictives donc il faut passer que X soit de la forme m colonnes et 2 lignes.

In [36]:

```
#transformation des données pour être au bon format
# X_train est de la forme : 2 colonnes, m lignes (examples)
# y_train est de la forme : m colonnes, 1 ligne

# La transposée de X_train est de la forme : m colonnes (exemples), 2 lignes
X_train=np.transpose(X_train)

# y_train est forcé pour être un tableau à 1 ligne contenant m colonnes
y_train=np.transpose(y_train.reshape((y_train.shape[0], 1)))

# mêmes traitements pour le jeu de test
X_test=np.transpose(X_test)
y_test=np.transpose(y_test.reshape((y_test.shape[0], 1)))

```

Création du réseau et affichage des informations.

In [37]:

```
network = MyNeuralNetwork()

network.addLayer(Layer(3,input=2,activation="relu"))
network.addLayer(Layer(1,activation="sigmoid"))

network.info()


```

```
Content of the network:
Layer n°  0  =>
        Input  None     Output 2
Layer n°  1  =>
        Input  2        Output 3
        Activation Function <function relu at 0x12e6941e0>
        W (3, 2) [[-1.26405266  1.52790535]
 [-0.97071094  0.47055962]
 [-0.10069672  0.30379318]]
        b (3, 1) [[-0.17259624]
 [ 0.15850954]
 [ 0.01342966]]
Layer n°  2  =>
        Input  3        Output 1
        Activation Function <function sigmoid at 0x12e6940d0>
        W (1, 3) [[-1.03209468  1.2475295  -0.79258216]]
        b (1, 1) [[0.04705596]]
```

Entraînement du classifieur et prédiction.

In [38]:

```
epochs = 100
eta = 0.01
batchsize=32

#Entraînement du classifieur
layers,cost_history,accuracy_history,parameter_history=network.fit(X_train, y


#Prédiction
y_pred=network.predict(X_test)
accuracy_test = network.accuracy(y_pred, y_test)
print("Accuracy test: %.3f"%accuracy_test)
```

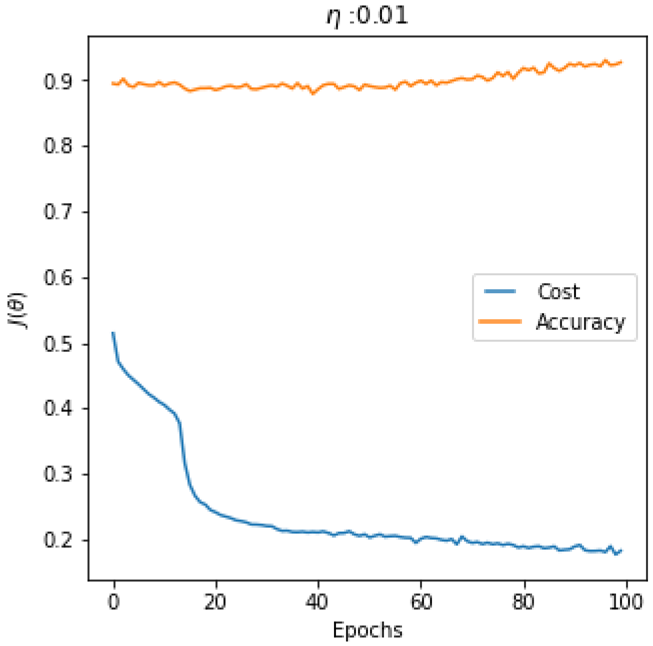
```
Epoch : #1/100 - 600/600 - cost : 0.5145 - accuracy : 0.8947
Epoch : #2/100 - 600/600 - cost : 0.4712 - accuracy : 0.8931
Epoch : #3/100 - 600/600 - cost : 0.4599 - accuracy : 0.9019
Epoch : #4/100 - 600/600 - cost : 0.4503 - accuracy : 0.8920
Epoch : #5/100 - 600/600 - cost : 0.4434 - accuracy : 0.8893
Epoch : #6/100 - 600/600 - cost : 0.4366 - accuracy : 0.8958
Epoch : #7/100 - 600/600 - cost : 0.4290 - accuracy : 0.8936
Epoch : #8/100 - 600/600 - cost : 0.4215 - accuracy : 0.8920
Epoch : #9/100 - 600/600 - cost : 0.4162 - accuracy : 0.8920
Epoch : #10/100 - 600/600 - cost : 0.4098 - accuracy : 0.8969
Epoch : #11/100 - 600/600 - cost : 0.4052 - accuracy : 0.8914
Epoch : #12/100 - 600/600 - cost : 0.3981 - accuracy : 0.8947
Epoch : #13/100 - 600/600 - cost : 0.3920 - accuracy : 0.8964
Epoch : #14/100 - 600/600 - cost : 0.3772 - accuracy : 0.8931
Epoch : #15/100 - 600/600 - cost : 0.3164 - accuracy : 0.8871
Epoch : #16/100 - 600/600 - cost : 0.2832 - accuracy : 0.8832
Epoch : #17/100 - 600/600 - cost : 0.2665 - accuracy : 0.8854
Epoch : #18/100 - 600/600 - cost : 0.2568 - accuracy : 0.8876
Epoch : #19/100 - 600/600 - cost : 0.2531 - accuracy : 0.8876
Epoch : #20/100 - 600/600 - cost : 0.2449 - accuracy : 0.8882
Epoch : #21/100 - 600/600 - cost : 0.2415 - accuracy : 0.8849
Epoch : #22/100 - 600/600 - cost : 0.2371 - accuracy : 0.8871
Epoch : #23/100 - 600/600 - cost : 0.2348 - accuracy : 0.8904
Epoch : #24/100 - 600/600 - cost : 0.2325 - accuracy : 0.8914
Epoch : #25/100 - 600/600 - cost : 0.2291 - accuracy : 0.8887
Epoch : #26/100 - 600/600 - cost : 0.2279 - accuracy : 0.8898
Epoch : #27/100 - 600/600 - cost : 0.2260 - accuracy : 0.8936
Epoch : #28/100 - 600/600 - cost : 0.2227 - accuracy : 0.8865
Epoch : #29/100 - 600/600 - cost : 0.2226 - accuracy : 0.8860
Epoch : #30/100 - 600/600 - cost : 0.2218 - accuracy : 0.8882
Epoch : #31/100 - 600/600 - cost : 0.2204 - accuracy : 0.8904
Epoch : #32/100 - 600/600 - cost : 0.2201 - accuracy : 0.8920
Epoch : #33/100 - 600/600 - cost : 0.2157 - accuracy : 0.8898
Epoch : #34/100 - 600/600 - cost : 0.2129 - accuracy : 0.8947
Epoch : #35/100 - 600/600 - cost : 0.2133 - accuracy : 0.8914
Epoch : #36/100 - 600/600 - cost : 0.2113 - accuracy : 0.8871
Epoch : #37/100 - 600/600 - cost : 0.2111 - accuracy : 0.8953
Epoch : #38/100 - 600/600 - cost : 0.2117 - accuracy : 0.8871
Epoch : #39/100 - 600/600 - cost : 0.2106 - accuracy : 0.8909
Epoch : #40/100 - 600/600 - cost : 0.2113 - accuracy : 0.8788
Epoch : #41/100 - 600/600 - cost : 0.2106 - accuracy : 0.8860
Epoch : #42/100 - 600/600 - cost : 0.2121 - accuracy : 0.8920
Epoch : #43/100 - 600/600 - cost : 0.2098 - accuracy : 0.8942
Epoch : #44/100 - 600/600 - cost : 0.2059 - accuracy : 0.8942
Epoch : #45/100 - 600/600 - cost : 0.2095 - accuracy : 0.8865
Epoch : #46/100 - 600/600 - cost : 0.2097 - accuracy : 0.8893
```

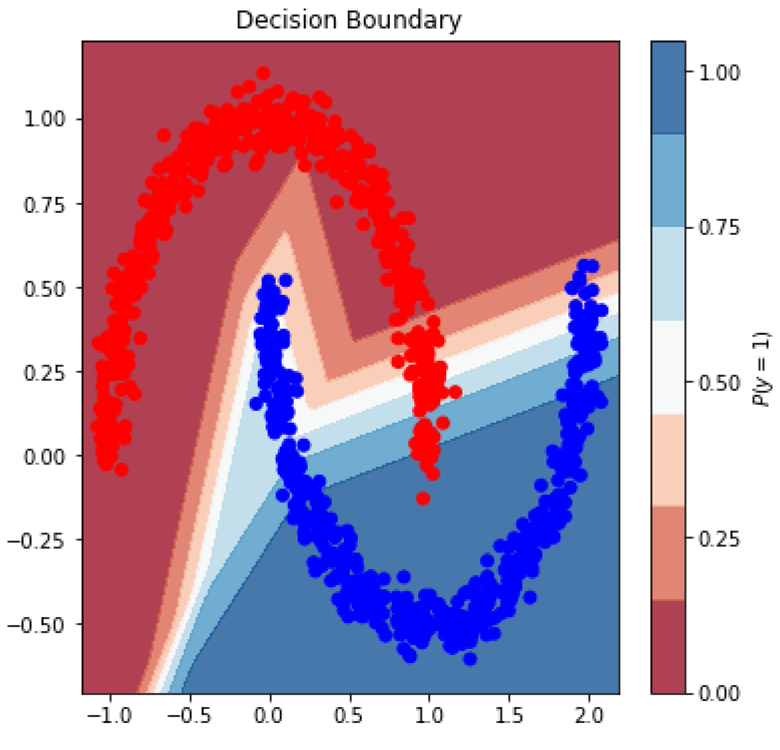
```
Epoch : #47/100 - 600/600 - cost : 0.2124 - accuracy : 0.8920
Epoch : #48/100 - 600/600 - cost : 0.2077 - accuracy : 0.8904
Epoch : #49/100 - 600/600 - cost : 0.2054 - accuracy : 0.8849
Epoch : #50/100 - 600/600 - cost : 0.2077 - accuracy : 0.8931
Epoch : #51/100 - 600/600 - cost : 0.2029 - accuracy : 0.8909
Epoch : #52/100 - 600/600 - cost : 0.2055 - accuracy : 0.8893
Epoch : #53/100 - 600/600 - cost : 0.2077 - accuracy : 0.8882
Epoch : #54/100 - 600/600 - cost : 0.2041 - accuracy : 0.8887
Epoch : #55/100 - 600/600 - cost : 0.2049 - accuracy : 0.8914
Epoch : #56/100 - 600/600 - cost : 0.2054 - accuracy : 0.8854
Epoch : #57/100 - 600/600 - cost : 0.2035 - accuracy : 0.8953
Epoch : #58/100 - 600/600 - cost : 0.2025 - accuracy : 0.8975
Epoch : #59/100 - 600/600 - cost : 0.2023 - accuracy : 0.8909
Epoch : #60/100 - 600/600 - cost : 0.1946 - accuracy : 0.8958
Epoch : #61/100 - 600/600 - cost : 0.2010 - accuracy : 0.8991
Epoch : #62/100 - 600/600 - cost : 0.2036 - accuracy : 0.8942
Epoch : #63/100 - 600/600 - cost : 0.2021 - accuracy : 0.8991
Epoch : #64/100 - 600/600 - cost : 0.2015 - accuracy : 0.8920
Epoch : #65/100 - 600/600 - cost : 0.1993 - accuracy : 0.8969
Epoch : #66/100 - 600/600 - cost : 0.1980 - accuracy : 0.8958
Epoch : #67/100 - 600/600 - cost : 0.2004 - accuracy : 0.8986
Epoch : #68/100 - 600/600 - cost : 0.1924 - accuracy : 0.9013
Epoch : #69/100 - 600/600 - cost : 0.2045 - accuracy : 0.9030
Epoch : #70/100 - 600/600 - cost : 0.1974 - accuracy : 0.9008
Epoch : #71/100 - 600/600 - cost : 0.1946 - accuracy : 0.9013
Epoch : #72/100 - 600/600 - cost : 0.1959 - accuracy : 0.9068
Epoch : #73/100 - 600/600 - cost : 0.1928 - accuracy : 0.9046
Epoch : #74/100 - 600/600 - cost : 0.1950 - accuracy : 0.8991
Epoch : #75/100 - 600/600 - cost : 0.1926 - accuracy : 0.9030
Epoch : #76/100 - 600/600 - cost : 0.1940 - accuracy : 0.9117
Epoch : #77/100 - 600/600 - cost : 0.1913 - accuracy : 0.9062
Epoch : #78/100 - 600/600 - cost : 0.1933 - accuracy : 0.9123
Epoch : #79/100 - 600/600 - cost : 0.1914 - accuracy : 0.9019
Epoch : #80/100 - 600/600 - cost : 0.1876 - accuracy : 0.9112
Epoch : #81/100 - 600/600 - cost : 0.1894 - accuracy : 0.9183
Epoch : #82/100 - 600/600 - cost : 0.1868 - accuracy : 0.9150
Epoch : #83/100 - 600/600 - cost : 0.1888 - accuracy : 0.9189
Epoch : #84/100 - 600/600 - cost : 0.1895 - accuracy : 0.9101
Epoch : #85/100 - 600/600 - cost : 0.1866 - accuracy : 0.9117
Epoch : #86/100 - 600/600 - cost : 0.1873 - accuracy : 0.9254
Epoch : #87/100 - 600/600 - cost : 0.1893 - accuracy : 0.9183
Epoch : #88/100 - 600/600 - cost : 0.1831 - accuracy : 0.9139
Epoch : #89/100 - 600/600 - cost : 0.1841 - accuracy : 0.9183
Epoch : #90/100 - 600/600 - cost : 0.1847 - accuracy : 0.9249
Epoch : #91/100 - 600/600 - cost : 0.1891 - accuracy : 0.9221
Epoch : #92/100 - 600/600 - cost : 0.1914 - accuracy : 0.9260
Epoch : #93/100 - 600/600 - cost : 0.1834 - accuracy : 0.9205
Epoch : #94/100 - 600/600 - cost : 0.1821 - accuracy : 0.9232
Epoch : #95/100 - 600/600 - cost : 0.1823 - accuracy : 0.9243
Epoch : #96/100 - 600/600 - cost : 0.1830 - accuracy : 0.9211
Epoch : #97/100 - 600/600 - cost : 0.1806 - accuracy : 0.9304
Epoch : #98/100 - 600/600 - cost : 0.1896 - accuracy : 0.9227
Epoch : #99/100 - 600/600 - cost : 0.1770 - accuracy : 0.9238
Epoch : #100/100 - 600/600 - cost : 0.1826 - accuracy : 0.9271
Accuracy test: 0.945
```

Affichage de l'historique de la fonction de coût et de l'accuracy ainsi que de la frontière de décision.

In [39]:

```
# Affichage des historiques
plot_histories (eta,epochs,cost_history,accuracy_history)
# Affichage de la frontière de décision
plot_decision_boundary(lambda x: network.predict(np.transpose(x)), X, y)
#plot_decision_boundary(lambda x: clf.predict(np.transpose(x)), X, y)
```

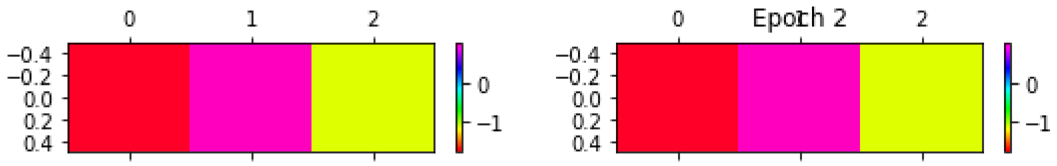


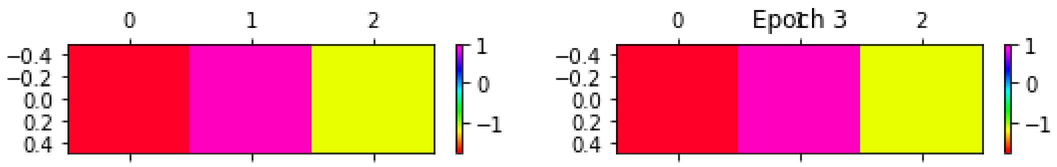


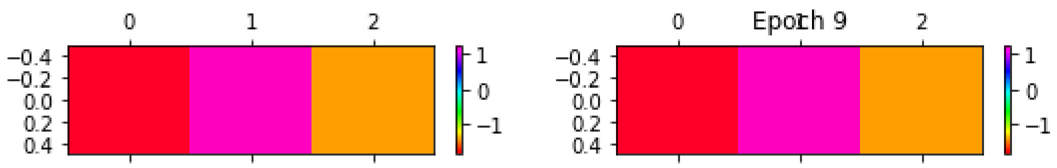
Affichage des valeurs de W pour les layers à différentes epochs.

In [40]:

```
layer_to_check=[2, 3, 9]
for i,e in enumerate(layer_to_check):
    network.plot_W_b_epoch(e,parameter_history)

```



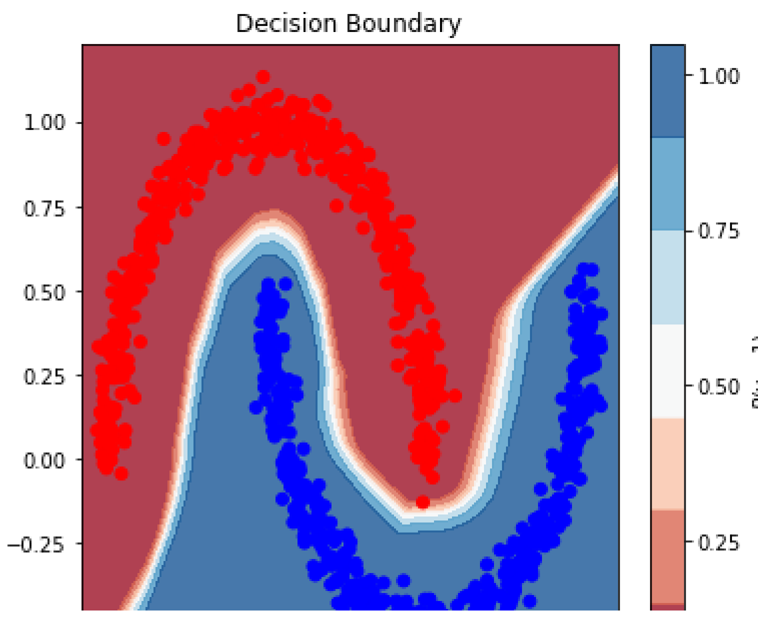




En changeant la configuration du réseau. Attention au surapprentissage.

In [41]:

```
network = MyNeuralNetwork()

network.addLayer(Layer(25,input=2,activation="relu"))
network.addLayer(Layer(25,activation="relu"))
network.addLayer(Layer(3,activation="relu"))
network.addLayer(Layer(1,activation="sigmoid"))

epochs = 100
eta = 0.01
batchsize=32

#Entraînement du classifieur
layers,cost_history,accuracy_history,parameter_history=network.fit(X_train, y


#Prédiction
y_pred=network.predict(X_test)
accuracy_test = network.accuracy(y_pred, y_test)
print("Accuracy test: %.3f"%accuracy_test)
# Affichage des historiques
plot_histories (eta,epochs,cost_history,accuracy_history)
# Affichage de la frontière de décision
plot_decision_boundary(lambda x: network.predict(np.transpose(x)), X, y)
```



Classification sur 3 classes à l'aide de softmax. Utilisation du jeu de données IRIS.

Sélection du jeu de données.

In [42]:

```
url="https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data
names = ['SepalLengthCm', 'SepalWidthCm',
         'PetalLengthCm', 'PetalWidthCm',
         'Species']

df = pd.read_csv(url, names=names)

# mélange des données
df=df.sample(frac=1).reset_index(drop=True)


array = df.values #necessité de convertir le dataframe en numpy
#X matrice de variables prédictives - attention forcer le type à float
X = array[:,0:4].astype('float64')
#y vecteur de variable à prédire
y = array[:,4]

```

Première étape : normalisation des données

In [43]:

```
# normalisation de X
sc_X=StandardScaler()
X=sc_X.fit_transform(X)
```

Seconde étape : transformation des variables prédites à l'aide de OneHotEncoder : mettre 1 colonne par classe. Quand un exemple d'apprentissage correspond à la classe, il y a un 1 à la ligne et la colonne correspondante.

In [44]:

```
# Conversion de la variable à prédire via OneHotEncoder
# Dans IRIS il y a 3 classes -> création de 3 colonnes pour y
# 1 colonne correspond à 1 classe -> 1 si la ligne est du type de la classe
# 0 sinon

# Integer encode
label_encoder = LabelEncoder()
integer_encoded = label_encoder.fit_transform(y)

# binary encode
onehot_encoder = OneHotEncoder(sparse=False,categories='auto')
integer_encoded = integer_encoded.reshape(len(integer_encoded), 1)
y = onehot_encoder.fit_transform(integer_encoded)
```

Comme précédemment création du jeu d'apprentissage et de test. Attention, il faut mettre les variables prédictives et prédites au bon format.

In [45]:

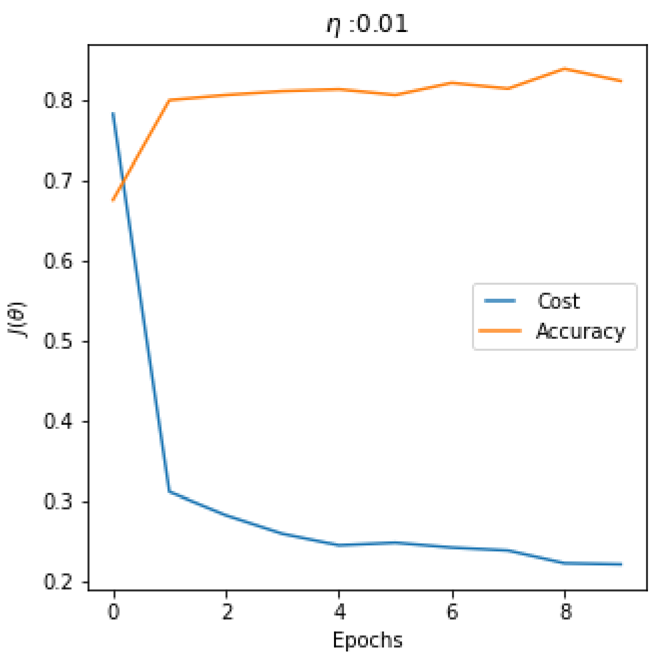
```
# Jeu de test/apprentissage
validation_size=0.6 #40% du jeu de données pour le test

testsize= 1-validation_size
seed=30
# séparation jeu d'apprentissage et jeu de test
X_train,X_test,y_train,y_test=train_test_split(X,
                                                y,
                                                train_size=validation_size,
                                                random_state=seed,
                                                test_size=testsize)


#transformation des données pour être au bon format
# X_train est de la forme : n colonnes (variables à prédire après OneHotEncod
# y_train est de la forme : m colonnes, n lignes (variables à prédire après O

# La transposée de X_train est de la forme : m colonnes (exemples), n lignes
X_train=np.transpose(X_train)

# y_train est forcé pour être un tableau à 1 ligne contenant m colonnes
y_train=np.transpose(y_train.reshape((y_train.shape[0], y_train.shape[1])))

# mêmes traitements pour le jeu de test
X_test=np.transpose(X_test)
y_test=np.transpose(y_test.reshape((y_test.shape[0], y_test.shape[1])))
```

Création du réseau et lancement du classifieur.

In [46]:

```
network = MyNeuralNetwork()

network.addLayer(Layer(10,input=4,activation="leakyrelu"))
network.addLayer(Layer(3,activation="softmax"))


epochs = 10
eta = 0.01
batchsize=128

#Entraînement du classifieur
layers,cost_history,accuracy_history,parameter_history=network.fit(X_train, y


#Prédiction
y_pred=network.predict(X_test)
accuracy_test = network.accuracy(y_pred, y_test)
print("Accuracy test: %.3f"%accuracy_test)

# Affichage des historiques
plot_histories (eta,epochs,cost_history,accuracy_history)
```

```
Epoch : #1/10 - 90/90 - cost : 0.7825 - accuracy : 0.6755
Epoch : #2/10 - 90/90 - cost : 0.3108 - accuracy : 0.7997
Epoch : #3/10 - 90/90 - cost : 0.2818 - accuracy : 0.8061
Epoch : #4/10 - 90/90 - cost : 0.2588 - accuracy : 0.8109
Epoch : #5/10 - 90/90 - cost : 0.2443 - accuracy : 0.8133
Epoch : #6/10 - 90/90 - cost : 0.2475 - accuracy : 0.8061
Epoch : #7/10 - 90/90 - cost : 0.2415 - accuracy : 0.8213
Epoch : #8/10 - 90/90 - cost : 0.2378 - accuracy : 0.8141
Epoch : #9/10 - 90/90 - cost : 0.2219 - accuracy : 0.8389
Epoch : #10/10 - 90/90 - cost : 0.2205 - accuracy : 0.8237
Accuracy test: 0.750
```

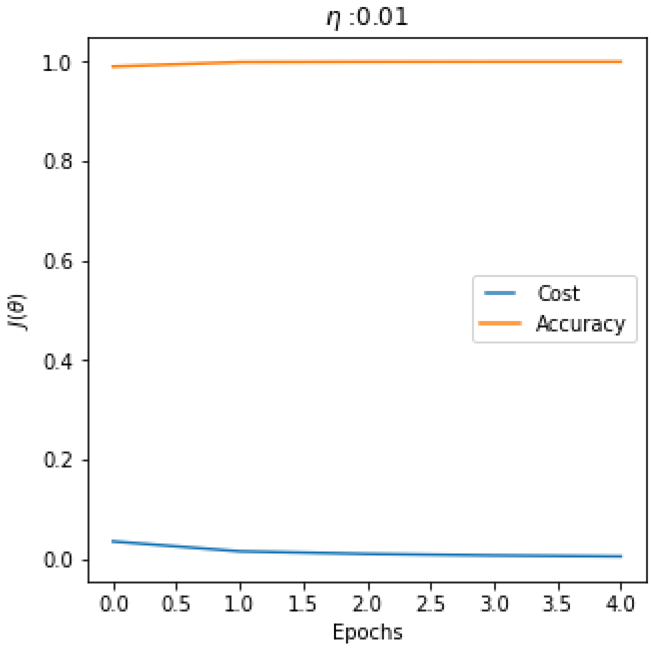


Essai sur le jeu de données MNIST. Un peu plus long car 60000 exemples d'apprentissage.

In [47]:

```
from tensorflow.keras.datasets import mnist
from keras.utils import to_categorical


(X_train_orig, y_train_orig), (X_test_orig, y_test_orig) = mnist.load_data()


y_tr_resh = y_train_orig.reshape(60000, 1)
y_te_resh = y_test_orig.reshape(10000, 1)
y_tr_T = to_categorical(y_tr_resh, num_classes=10)
y_te_T = to_categorical(y_te_resh, num_classes=10)
y_train = y_tr_T.T
#Y_train = Y_tr_resh.T
y_test = y_te_T.T


X_train_flatten = X_train_orig.reshape(X_train_orig.shape[0], -1).T
#X_train_flatten=np.transpose(X_train)
X_test_flatten = X_test_orig.reshape(X_test_orig.shape[0], -1).T


X_train = X_train_flatten / 255.
X_test = X_test_flatten / 255.


network = MyNeuralNetwork()

network.addLayer(Layer(512,input=784,activation="relu"))
network.addLayer(Layer(10,activation="softmax"))

#network.info()
epochs = 5
eta = 0.01
batchsize=128
beta=0.9

#Entraînement du classifieur
layers,cost_history,accuracy_history,parameter_history=network.fit(X_train, y


#Prédiction
y_pred=network.predict(X_test)
accuracy_test = network.accuracy(y_pred, y_test)
print("Accuracy test: %.3f"%accuracy_test)

# Affichage des historiques
plot_histories (eta,epochs,cost_history,accuracy_history)

```

```
Epoch : #1/5 - 60000/60000 - cost : 0.0350 - accuracy : 0.9902
Epoch : #2/5 - 60000/60000 - cost : 0.0150 - accuracy : 0.9991
Epoch : #3/5 - 60000/60000 - cost : 0.0099 - accuracy : 0.9997
Epoch : #4/5 - 60000/60000 - cost : 0.0068 - accuracy : 0.9998
Epoch : #5/5 - 60000/60000 - cost : 0.0048 - accuracy : 0.9999
Accuracy test: 0.978
```



Classe avec optimisation (momentum, adam)

Dans cette section, nous présentons une classe plus complète qui permet de faire des optimisations : momentum et adam.

Ces modifications sont répercutées sur la classe *Layer* et *MyNeuralNetwork*.

In [48]:

```
# importations utiles
import numpy as np
from sklearn.datasets import make_moons
from sklearn.datasets import make_circles
from sklearn import datasets
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import OneHotEncoder
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt
import matplotlib
from sklearn.model_selection import train_test_split
from sklearn import linear_model
from matplotlib.legend_handler import HandlerLine2D
import keras
from keras.models import Sequential
from keras.layers import Dense
from keras.utils import np_utils
from keras import regularizers
from keras.optimizers import SGD
from sklearn.metrics import accuracy_score
import pandas as pd
import seaborn as sns
import numpy as np
matplotlib.rcParams['figure.figsize'] = (6.0, 6.0) # pour avoir des figures d
```

Fonctions d'affichage inchangées.

In [49]:

```
def plot_histories (eta,epochs,cost_history,accuracy_history):
    fig,ax = plt.subplots(figsize=(5,5))
    ax.set_ylabel(r'$J(\theta)$')
    ax.set_xlabel('Epochs')
    ax.set_title(r"$\eta$ :{}".format(eta))
    line1, = ax.plot(range(epochs),cost_history,label='Cost')
    line2, = ax.plot(range(epochs),accuracy_history,label='Accuracy')
    plt.legend(handler_map={line1: HandlerLine2D(numpoints=4)})

def plot_decision_boundary(func, X, y):
    amin, bmin = X.min(axis=0) - 0.1
    amax, bmax = X.max(axis=0) + 0.1
    hticks = np.linspace(amin, amax, 101)
    vticks = np.linspace(bmin, bmax, 101)

    aa, bb = np.meshgrid(hticks, vticks)
    ab = np.c_[aa.ravel(), bb.ravel()]
    c = func(ab)
    cc = c.reshape(aa.shape)

    cm = plt.cm.RdBu
    cm_bright = ListedColormap(['#FF0000', '#0000FF'])

    fig, ax = plt.subplots()
    contour = plt.contourf(aa, bb, cc, cmap=cm, alpha=0.8)

    ax_c = fig.colorbar(contour)
    ax_c.set_label("$P(y = 1)$")
    ax_c.set_ticks([0, 0.25, 0.5, 0.75, 1])

    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cm_bright)
    plt.xlim(amin, amax)
    plt.ylim(bmin, bmax)
    plt.title("Decision Boundary")
```

Fonctions d'activations et dérivées inchangées.

In [50]:

```
#fonctions utiles
def sigmoid(x):
    return 1/(1 + np.exp(-x))

def sigmoid_prime(x):
    return sigmoid(x)*(1.0 - sigmoid(x))

def tanh(x):
    return np.tanh(x)

def tanh_prime(x):
    return 1 - x ** 2

def relu(x):
    return np.maximum(0,x)

def relu_prime(x):
    x[x<=0] = 0
    x[x>0] = 1
    return x

def leakyrelu(x):
    return np.maximum(0.01,x)

def leakyrelu_prime(x):
    x[x<=0] = 0.01
    x[x>0] = 1
    return x

def softmax(x):
    expx = np.exp(x - np.max(x))
    return expx / expx.sum(axis=0, keepdims=True)


```

Le constructeur et la méthode *initParams* ont été modifiés pour pouvoir prendre en compte les optimisations de descente de gradient.

In [51]:

```
class Layer:
    def __init__(self,output,*args,**kwargs):
        self.output = output # Nombre de neurones au layer i (actuel)
        self.input = kwargs.get("input",None) # Nombre de neurones au layer i
        self.activ_function_curr = kwargs.get("activation",None) # fonction d
        self.parameters ={}
        self.derivatives={}
        #for momentum
        self.v={}
        #for adam
        self.s={}

        self.activation_func=None
        if self.activ_function_curr == "relu":
            self.activation_func = relu
            self.backward_activation_func = relu_prime
        elif self.activ_function_curr == "sigmoid":
            self.activation_func = sigmoid
            self.backward_activation_func = sigmoid_prime
        elif self.activ_function_curr == "tanh":
            self.activation_func = tanh
            self.backward_activation_func = tanh_prime
        elif self.activ_function_curr == "leakyrelu":
            self.activation_func = leakyrelu
            self.backward_activation_func = leakyrelu_prime
        elif self.activ_function_curr == "softmax":
            self.activation_func = softmax
            self.backward_activation_func = softmax

    def initParams(self,optimizer):
        seed=30
        np.random.seed(seed)
        self.parameters['W']=np.random.randn(self.output,self.input)*np.sqrt(
        self.parameters['b']=np.random.randn(self.output,1)*0.1
        if optimizer=="momentum":
                # Sauvegarde velocity, v, pour momentum
                self.v['dW'] = np.zeros_like(self.parameters['W'])
                self.v['db'] = np.zeros_like(self.parameters['b'])
        elif optimizer=='adam':
                self.v['dW'] = np.zeros_like(self.parameters['W'])
                self.v['db'] = np.zeros_like(self.parameters['b'])
                self.s['dW'] = np.zeros_like(self.parameters['W'])
                self.s['db'] = np.zeros_like(self.parameters['b'])


    def setW(self,matW):
        #self.parameters['W']=matW
        self.parameters['W']=np.copy(matW)

    def setA(self,matA):
        self.parameters['A']=np.copy(matA)

    def setZ(self,matZ):
        self.parameters['Z']=np.copy(matZ)

    def setB(self,matB):
        self.parameters['b']=np.copy(matB)

    def setdW(self,matdW):
```

```
            self.parameters['dW']=np.copy(matdW)

        def setdA(self,matdA):
            self.parameters['dA']=np.copy(matdA)

        def setdZ(self,matdZ):
            self.parameters['dZ']=np.copy(matdZ)

        def setdB(self,matdB):
            self.parameters['db']=np.copy(matdB)

```

La structure de la classe *MyNeuralNetwork* est assez similaire à la précédente.

- Le constructeur prend par défaut *optimizer="bgd"* (*batch gradient descent*) (les valeurs possibles sont "*momentum*" ou "*adam*").
- La méthode update prend en compte le fait que l'optimisation puisse être de type momentum ou adam.

Par exemple :

network = MyNetwork(optimizer="adam")

aura pour effet de créer un réseau qui utilise l'optimisation adam.

Enfin lors de l'appel de la méthode fit les paramètres beta, beta1 et epsilon doivent être spécifiées (en cas d'appel d'un optimizer. Ici les valeurs par défaut sont : beta=0.9,beta2=0.999, epsilon=1e-8.

Par exemple :

layers,cost_history,accuracy_history,parameter_history=network.fit(X_train, y_train, verbose=True, epochs=40, eta=0.01, beta=0.8)

aura pour effet de lancer le classifieur sur X_train, y_train, en affichant le coût et l'accuracy (verbose=True), pour 40 epochs (epochs=40), avec un learning rate de 0.01 (eta=0.01) et un beta de 0.8 (beta=0.8).

In [52]:

```
class MyNeuralNetwork:
    def __init__(self,optimizer="bgd"):
        self.nbLayers=0
        self.layers=[]
        self.optimizer=optimizer
        self.t=2

    def info(self):
        print("Content of the network:");
        j=0;
        for i in range(len(self.layers)):
            print("Layer n° ",i," => ")
            print ("\tInput ", self.layers[i].input,
                   "\tOutput", self.layers[i].output)
            if (i != 0):
                print ("\tActivation Function",self.layers[i].activation_func
                print ("\tW", self.layers[i].parameters['W'].shape,self.layer
                print ("\tb", self.layers[i].parameters['b'].shape,self.layer


    def addLayer(self,layer):
        self.nbLayers += 1;
        if (self.nbLayers==1):
            # this is the first layer so adding a layer 0
            layerZero=Layer(layer.input)
            self.layers.append(layerZero)

        self.layers.append(layer)
        self.layers[self.nbLayers].input=self.layers[self.nbLayers-1].output
        self.layers[self.nbLayers].output=self.layers[self.nbLayers].output
        layer.initParams(self.optimizer)



    def set_parametersW_b (self,numlayer,matX,matb):
        self.layers[numlayer].parameters['W']=np.copy(matX)
        self.layers[numlayer].parameters['b']=np.copy(matb)


    def forward_propagation(self, X):
        #Initialisation des variables prédictives pour la couche d'entrée
        self.layers[0].setA(X)

        #Propagation pour tous les layers
        for l in range(1, self.nbLayers + 1):
            # Calcul de Z
            self.layers[l].setZ(np.dot(self.layers[l].parameters['W'],
                                       self.layers[l-1].parameters['A'])+sel
            # Application de la fonction d'activation à Z
            self.layers[l].setA(self.layers[l].activation_func(self.layers[l


    def cost_function(self,y):
        return (-(y*np.log(self.layers[self.nbLayers].parameters['A']+1e-8) -

    def backward_propagation(self,y):
        #calcul de dZ dW et db pour le dernier layer
        self.layers[self.nbLayers].derivatives['dZ']=self.layers[self.nbLaye
        self.layers[self.nbLayers].derivatives['dW']=np.dot(self.layers[self.
```

```
                                                                   np.transpose(se
        m=self.layers[self.nbLayers].parameters['A'].shape[1]#égal au nombre
        self.layers[self.nbLayers].derivatives['db']=np.sum(self.layers[self.
                                                              axis=1, keepdims=True

        #calcul de dZ dW db pour les autres layers
        for l in range(self.nbLayers-1,0,-1) :
            self.layers[l].derivatives['dZ']=np.dot(np.transpose(self.layers[
                                            self.layers[l+1].derivatives['dZ

            self.layers[l].derivatives["dW"]=np.dot(self.layers[l].derivative
                                            np.transpose(self.layers[l-1].pa

            m=self.layers[l-1].parameters['A'].shape[1]#égal au nombre de co
            self.layers[l].derivatives['db']=np.sum(self.layers[l].derivative
                                                              axis=1, keepdims=True


    def update_parameters(self, eta,beta, beta2=0.999, epsilon=1e-8):
        # Descente de gradient

        if self.optimizer=="adam":
            v_corrected = {}            # Initialisation de la première esti
            s_corrected = {}            # Initialisation de la seconde estim

        for l in range(1, self.nbLayers+1):
            if self.optimizer=="momentum":
                # Calcul de la vitesse
                self.layers[l].v['dW'] = beta * self.layers[l].v['dW'] + (1 -
                self.layers[l].v['db'] = beta * self.layers[l].v['db'] + (1 -

                # Mise à jour des paramètres
                self.layers[l].parameters['W'] -= eta*self.layers[l].v['dW']
                self.layers[l].parameters['b'] -= eta*self.layers[l].v['db']
            elif self.optimizer=="adam":
                # Calcul de la vitesse
                self.layers[l].v['dW'] = beta * self.layers[l].v['dW'] + (1 -
                self.layers[l].v['db'] = beta * self.layers[l].v['db'] + (1 -

                # Calcul de la première estimation du moment (correction du b
                v_corrected['dW' + str(l)] = self.layers[l].v['dW'] / (1 - np
                v_corrected['db' + str(l)] = self.layers[l].v['db'] / (1 - np

                 # déplacement moyen des gradients au carré
                self.layers[l].s['dW'] = beta2 * self.layers[l].s['dW'] + (1
                self.layers[l].s['db'] = beta2 * self.layers[l].s['db'] + (1

                # Calcul de la seconde estimation du moment (correction du b
                s_corrected["dW" + str(l)] = self.layers[l].s['dW'] / (1 - np
                s_corrected["db" + str(l)] = self.layers[l].s['db'] / (1 - np

                # Mise à jour des paramètres
                self.layers[l].parameters['W'] -= eta*v_corrected['dW' + str(
                self.layers[l].parameters['b'] -= eta*v_corrected['db' + str(

            else: #descente par mini-lots
                self.layers[l].parameters['W'] -= eta*self.layers[l].derivati
                self.layers[l].parameters['b'] -= eta*self.layers[l].derivati


    def convert_prob_into_class(self,probs):
```

```
            probs = np.copy(probs)#pour ne pas perdre probs, i.e. y_hat
            probs[probs > 0.5] = 1
            probs[probs <= 0.5] = 0
            return probs

        def plot_W_b_epoch (self,epoch,parameter_history):
            mat=[]
            max_size_layer=0
            for l in range(1, self.nbLayers+1):
                value=parameter_history[epoch]['W'+str(l)]
                if (parameter_history[epoch]['W'+str(l)].shape[1]>max_size_layer)
                    max_size_layer=parameter_history[epoch]['W'+str(l)].shape[1]
                mat.append(value)
            figure=plt.figure(figsize=((self.nbLayers+1)*3,int (max_size_layer/2)
            for nb_w in range (len(mat)):
                    plt.subplot(1, len(mat), nb_w+1)
                    plt.matshow(mat[nb_w],cmap = plt.cm.gist_rainbow,fignum=False
                    plt.colorbar()
            thelegend="Epoch "+str(epoch)
            plt.title (thelegend)



        def accuracy(self,y_hat, y):
            if self.layers[self.nbLayers].activation_func==softmax:
                # si la fonction est softmax, les valeurs sont sur différentes d
                # il faut utiliser argmax avec axis=0 pour avoir un vecteur qui i
                # où est la valeur maximale à la fois pour y_hat et pour y
                # comme cela il suffit de comparer les deux vecteurs qui indiquen
                # dans quelle ligne se trouve le max
                y_hat_encoded=np.copy(y_hat)
                y_hat_encoded = np.argmax(y_hat_encoded, axis=0)
                y_encoded=np.copy(y)
                y_encoded=np.argmax(y_encoded, axis=0)
                return (y_hat_encoded == y_encoded).mean()
            # la dernière fonction d'activation n'est pas softmax.
            # par exemple sigmoid pour une classification binaire
            # il suffit de convertir la probabilité du résultat en classe
            y_hat_ = self.convert_prob_into_class(y_hat)
            return (y_hat_ == y).all(axis=0).mean()

        def predict(self, x):
            self.forward_propagation(x)
            return self.layers[self.nbLayers].parameters['A']

        def next_batch(self,X, y, batchsize):
            # pour avoir X de la forme : 2 colonnes, m lignes (examples) et égale
            # cela permet de trier les 2 tableaux avec un indices de permutation
            X=np.transpose(X)
            y=np.transpose(y)

            m=len(y)
            # permutation aléatoire de X et y pour faire des batchs avec des vale
            indices = np.random.permutation(m)
            X = X[indices]
            y = y[indices]
            for i in np.arange(0, X.shape[0], batchsize):
                # creation des batchs de taille batchsize
                yield (X[i:i + batchsize], y[i:i + batchsize])
```

```
    #def fit(self, X, y, epochs, verbose=True,eta = 0.01,batchsize=32,beta=0
    def fit(self, X, y, *args,**kwargs):
        epochs=kwargs.get("epochs",20)
        verbose=kwargs.get("verbose",False)
        eta =kwargs.get("eta",0.01)
        batchsize=kwargs.get("batchsize",32)
        beta=kwargs.get("beta",0.9)
        beta2=kwargs.get("beta2",0.999)
        epsilon=kwargs.get("epsilon",1e-8)
        # sauvegarde historique coût, accuracy pour affichage
        # parameter_history sauvegarde des statistiques, W et b pour toutes 
        # epochs afin de pouvoir étudier l'historique
        cost_history = []
        accuracy_history = []
        parameter_history = []

        for i in range(epochs):
            i+=1
            # sauvegarde des coûts et accuracy par mini-batch
            cost_batch = []
            accuracy_batch = []
            # Gradient descent per mini-batch

            for (batchX, batchy) in self.next_batch(X, y, batchsize):

                # Extract and process one batch at a time

                # Data must be at the appropriate format
                batchX=np.transpose(batchX)

                if self.layers[self.nbLayers].activation_func==softmax:
                    # Not binary classification. one-hot-encoder has been use
                    # the batchy must be transformed and the result
                    # must be a matrice with size batchy.shape[1]

                    batchy=np.transpose(batchy.reshape((batchy.shape[0], batc
                else:
                    # It is a binary classification so shape[1] does not exi
                    batchy=np.transpose(batchy.reshape((batchy.shape[0], 1))


                self.forward_propagation(batchX)
                self.backward_propagation(batchy)
                if self.optimizer=="adam":
                    self.t=self.t+1
                self.update_parameters(eta,beta,beta2,epsilon)

                # save for output
                current_cost=self.cost_function(batchy)
                cost_batch.append(current_cost)
                y_hat = self.predict(batchX)
                current_accuracy = self.accuracy(y_hat, batchy)
                accuracy_batch.append(current_accuracy)

            # SaveStats on W, B as well as values for A,Z, W, b
            save_values = {}
            save_values["epoch"]=i
            for l in range(1, self.nbLayers+1):
                save_values["layer"+str(l)]=l
                save_values["Wmean"+ str(l)]=np.mean(self.layers[self.nbLaye
                save_values["Wmax"+ str(l)]=np.amax(self.layers[self.nbLayers
```

```
                save_values["Wmin"+str(l)]=np.amin(self.layers[self.nbLayers
                save_values["Wstd"+str(l)]=np.std(self.layers[self.nbLayers].
                save_values["bmean"+ str(l)]=np.mean(self.layers[self.nbLaye
                save_values["bmax"+ str(l)]=np.amax(self.layers[self.nbLayers
                save_values["bmin"+str(l)]=np.amin(self.layers[self.nbLayers
                save_values["bstd"+str(l)]=np.std(self.layers[self.nbLayers].
                # be careful A,Z,W and b must be copied otherwise it is a re
                save_values['A'+str(l)]=np.copy(self.layers[self.nbLayers].pa
                save_values['Z'+str(l)]=np.copy(self.layers[self.nbLayers].pa
                save_values['W'+str(l)]=np.copy(self.layers[self.nbLayers].pa
                save_values['b'+str(l)]=np.copy(self.layers[self.nbLayers].pa

            parameter_history.append(save_values)

            # save avg value for cost and accuracy
            current_cost=np.average(cost_batch)
            cost_history.append(current_cost)
            current_accuracy=np.average(accuracy_batch)
            accuracy_history.append(current_accuracy)

            if(verbose == True):
                print("Epoch : #%s/%s - %s/%s - cost : %.4f - accuracy : %.4

        return self.layers, cost_history, accuracy_history, parameter_history
```

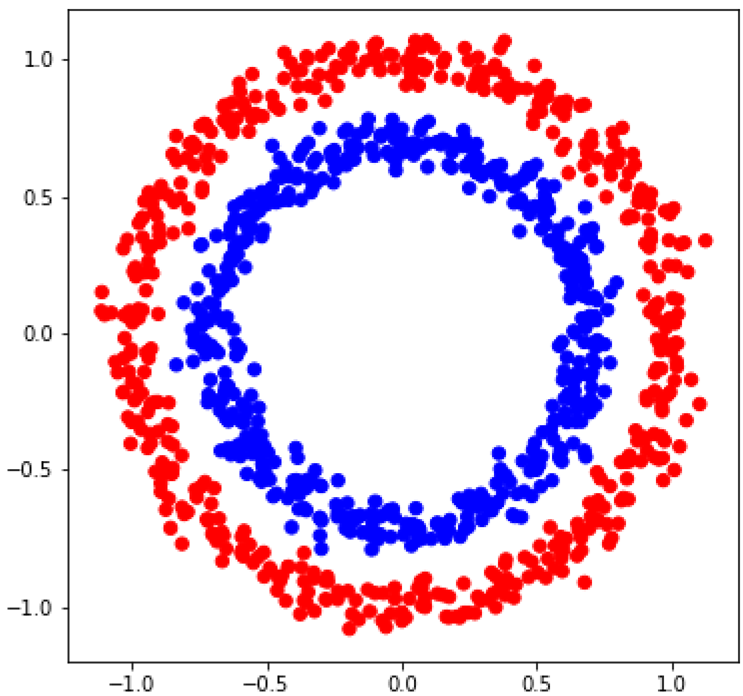
Test sur un jeu de données

In [53]:

```
X, y = make_circles(n_samples=1000, noise=0.05, factor=0.7, random_state=0)
cm_bright = ListedColormap(['#FF0000', '#0000FF'])
plt.scatter(X[:,0], X[:,1], s=40, c=y, cmap=cm_bright)#cmap=plt.cm.PiYG)
```

Out[53]:

<matplotlib.collections.PathCollection at 0x12ce94fd0>



Préparation des données et du jeu d'apprentissage et de test.

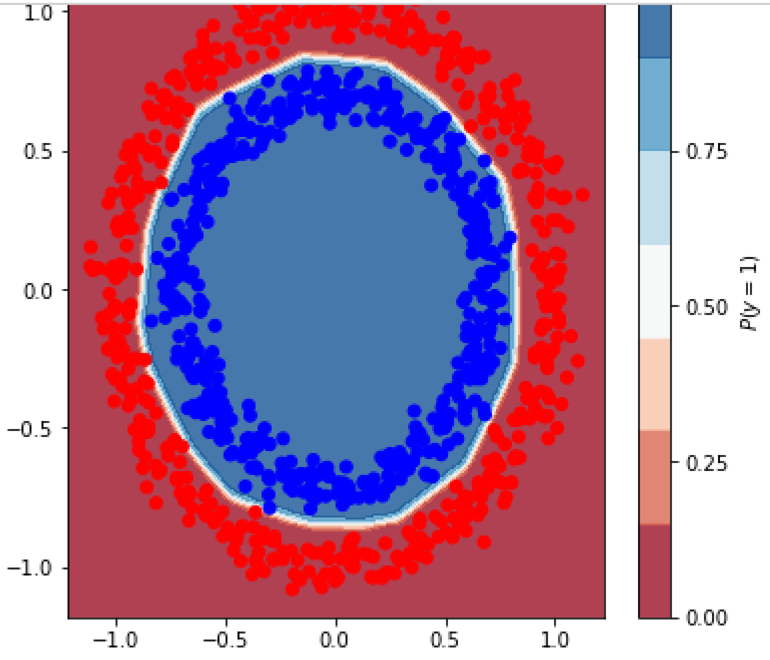
In [54]:

```
validation_size=0.6 #40% du jeu de données pour le test

testsize= 1-validation_size
seed=30
# séparation jeu d'apprentissage et jeu de test
X_train,X_test,y_train,y_test=train_test_split(X,
                                               y,
                                               train_size=validation_size,
                                               random_state=seed,
                                               test_size=testsize)


#transformation des données pour être au bon format
# X_train est de la forme : 2 colonnes, m lignes (examples)
# y_train est de la forme : m colonnes, 1 ligne

# La transposée de X_train est de la forme : m colonnes (exemples), 2 lignes
X_train=np.transpose(X_train)

# y_train est forcé pour être un tableau à 1 ligne contenant m colonnes
y_train=np.transpose(y_train.reshape((y_train.shape[0], 1)))

# mêmes traitements pour le jeu de test
X_test=np.transpose(X_test)
y_test=np.transpose(y_test.reshape((y_test.shape[0], 1)))
```

Création du réseau en utilisant *momentum*.

In [55]:

```
Myoptimizer="momentum"
Myepochs = 100
Myeta = 0.01
Mybatchsize=32
Mybeta=0.9

network = MyNeuralNetwork(optimizer=Myoptimizer)

network.addLayer(Layer(25,input=2,activation="relu"))
network.addLayer(Layer(25,activation="relu"))
network.addLayer(Layer(3,activation="relu"))
network.addLayer(Layer(1,activation="sigmoid"))



#Entraînement du classifieur
layers,cost_history,accuracy_history,parameter_history=network.fit(X_train, y
                                                                    eta=Myeta, batchsize=Mybatch


#Prédiction
y_pred=network.predict(X_test)
accuracy_test = network.accuracy(y_pred, y_test)
print("Accuracy test: %.3f"%accuracy_test)

# Affichage des historiques

plot_histories (Myeta,Myepochs,cost_history,accuracy_history)
# Affichage de la frontière de décision
# Affichage de la frontière de décision
plot_decision_boundary(lambda x: network.predict(np.transpose(x)), X, y)


```



Le même en utilisant *adam*.

In [56]:

```
Myoptimizer="adam"
Myepochs = 100
Myeta = 0.01
Mybatchsize=32
Mybeta=0.9
Mybeta2=0.999
Myepsilon=1e-8

network = MyNeuralNetwork(optimizer=Myoptimizer)

network.addLayer(Layer(25,input=2,activation="relu"))
network.addLayer(Layer(25,activation="relu"))
network.addLayer(Layer(3,activation="relu"))
network.addLayer(Layer(1,activation="sigmoid"))



#Entraînement du classifieur
layers,cost_history,accuracy_history,parameter_history=network.fit(X_train, y
                                                 eta=Myeta, batchsize=Mybatch
                                                   epsilon=Myepsilon)



#Prédiction
y_pred=network.predict(X_test)
accuracy_test = network.accuracy(y_pred, y_test)
print("Accuracy test: %.3f"%accuracy_test)

# Affichage des historiques

plot_histories (Myeta,Myepochs,cost_history,accuracy_history)
# Affichage de la frontière de décision

plot_decision_boundary(lambda x: network.predict(np.transpose(x)), X, y)
```

Utilisation de softmax et d'adam sur le jeu de données IRIS.

In [57]:

```
url="https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data
names = ['SepalLengthCm', 'SepalWidthCm',
         'PetalLengthCm', 'PetalWidthCm',
         'Species']

df = pd.read_csv(url, names=names)

# mélange des données
df=df.sample(frac=1).reset_index(drop=True)


array = df.values #necessité de convertir le dataframe en numpy
#X matrice de variables prédictives - attention forcer le type à float
X = array[:,0:4].astype('float32')
#y vecteur de variable à prédire
y = array[:,4]

# normalisation de X
sc_X=StandardScaler()
X=sc_X.fit_transform(X)
# Conversion de la variable à prédire via OneHotEncoder
# Dans IRIS il y a 3 classes -> création de 3 colonnes pour y
# 1 colonne correspond à 1 classe -> 1 si la ligne est du type de la classe
# 0 sinon

# Integer encode
label_encoder = LabelEncoder()
integer_encoded = label_encoder.fit_transform(y)

# binary encode
onehot_encoder = OneHotEncoder(sparse=False,categories='auto')
integer_encoded = integer_encoded.reshape(len(integer_encoded), 1)
y = onehot_encoder.fit_transform(integer_encoded)
# Jeu de test/apprentissage
validation_size=0.6 #40% du jeu de données pour le test

testsize= 1-validation_size
seed=30
# séparation jeu d'apprentissage et jeu de test
X_train,X_test,y_train,y_test=train_test_split(X,
                                               y,
                                               train_size=validation_size,
                                               random_state=seed,
                                               test_size=testsize)


#transformation des données pour être au bon format
# X_train est de la forme : n colonnes (variables à prédire après OneHotEncod
# y_train est de la forme : m colonnes, n lignes (variables à prédire après O

# La transposée de X_train est de la forme : m colonnes (exemples), n lignes
X_train=np.transpose(X_train)

# y_train est forcé pour être un tableau à 1 ligne contenant m colonnes
y_train=np.transpose(y_train.reshape((y_train.shape[0], y_train.shape[1])))

# mêmes traitements pour le jeu de test
X_test=np.transpose(X_test)
y_test=np.transpose(y_test.reshape((y_test.shape[0], y_test.shape[1])))
```

```

Myoptimizer="adam"
Myepochs = 100
Myeta = 0.01
Mybatchsize=10
Mybeta=0.9
Mybeta2=0.999
Myepsilon=1e-8

network = MyNeuralNetwork(optimizer=Myoptimizer)

network.addLayer(Layer(10,input=4,activation="leakyrelu"))

network.addLayer(Layer(3,activation="softmax"))



#Entraînement du classifieur
layers,cost_history,accuracy_history,parameter_history=network.fit(X_train, y
                                                  eta=Myeta, batchsize=Mybatch
                                                    epsilon=Myepsilon)



#Prédiction
y_pred=network.predict(X_test)
accuracy_test = network.accuracy(y_pred, y_test)
print("Accuracy test: %.3f"%accuracy_test)

# Affichage des historiques

plot_histories (Myeta,Myepochs,cost_history,accuracy_history)
# Affichage de la frontière de décision

```