Evolution du Deep Learning

1943

Invention des Premiers Neurones Artificiels

1947

Invention du Perceptron

1986

Invention du Perceptron Multicouche

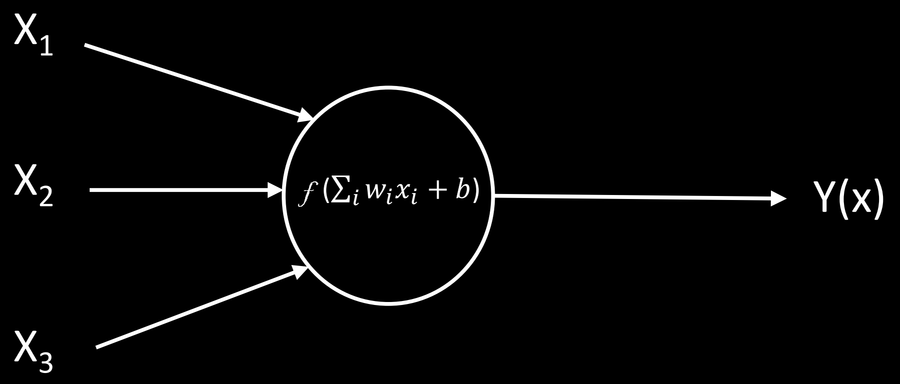
1990

Apparition des Reseaux de Neurones Convolutifs

Le Perceptron

1. Neurone Formel et le perceptron Multicouche.
2. Fonction d’activation, en particulier la function sigmoide (Logistique).
3. Fonction Cout et minimisation de l’erreur.
4. La Descente du gradient.
5. Algorithme de la descente de gradient.

Neurone formel



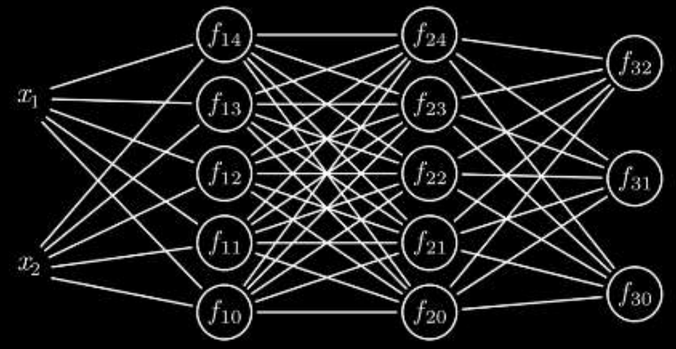
X_i : Entrées

f : Fonction d'activation

Y : Sortie

Perceptron multicouche

Entrées



Sortie

$f_{i,j}$: Fonction d'activation

Fonction d'activation

Sigmoid

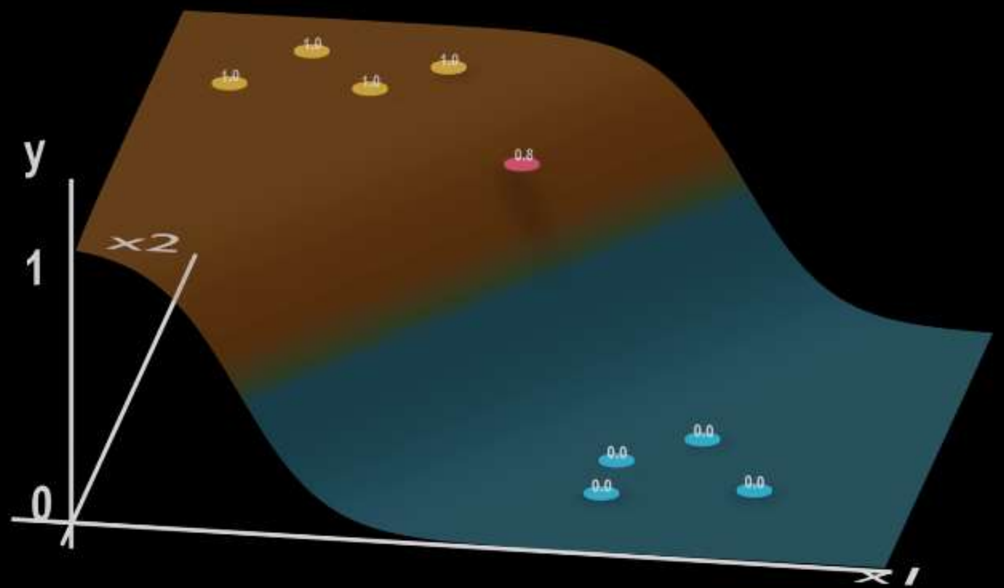
$$f(x) = \frac{1}{1 + e^{-x}}$$

Relu

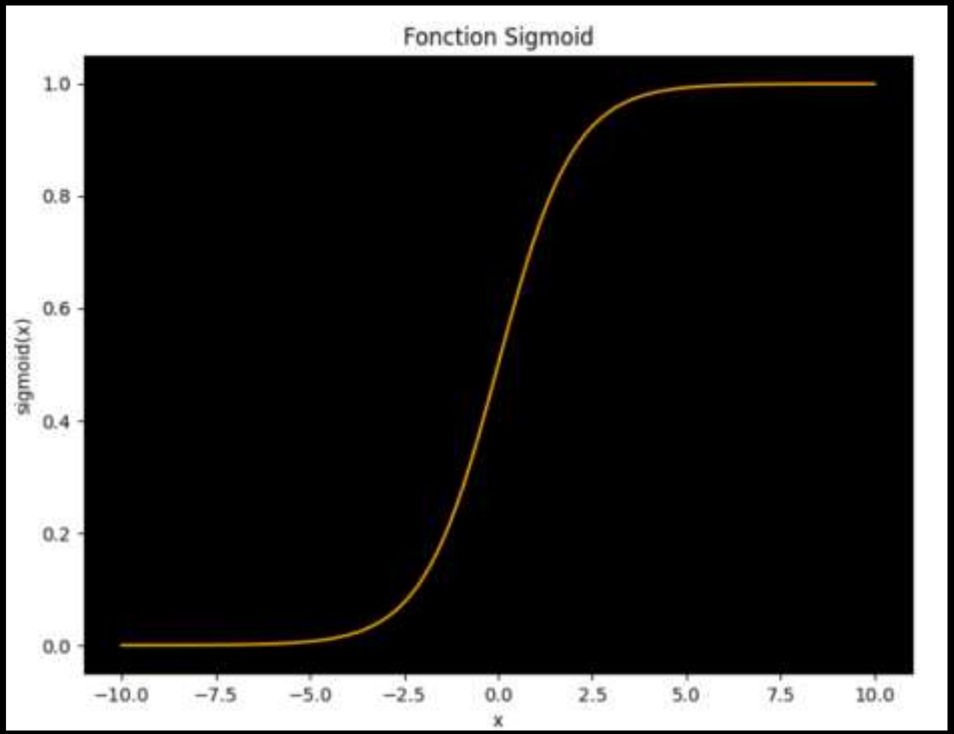
$$f(x) = \max(0, x)$$

Heaviside

$$H(x) = \begin{cases} 1, Si\ x > 0 \\ \frac{1}{2}, Si\ x = 0 \\ 0, Si\ x < 0 \end{cases}$$



Source : Machine Learnia



Minimisation de l'erreur

Pour minimiser les erreurs de notre modèle, nous utiliserons la fonction de coût ***Log Loss:***

$$\boxed{-\frac{1}{m} * \sum_{i=1}^{m} \mathrm{y_i} * \log(\mathrm{a_i}) + (1 - \mathrm{y_i}) * \log(1 - \mathrm{a_i})}$$

$$\begin{cases} a_i\text{: } \textit{la probabilité d'etre dans la classe 1 pour la sortie } i \\ y_i\text{: } \textit{la classe de l'echantillon} \end{cases}$$

Cette formule est dérivée de la définition statistique de ***Log-vraisemblance*** :

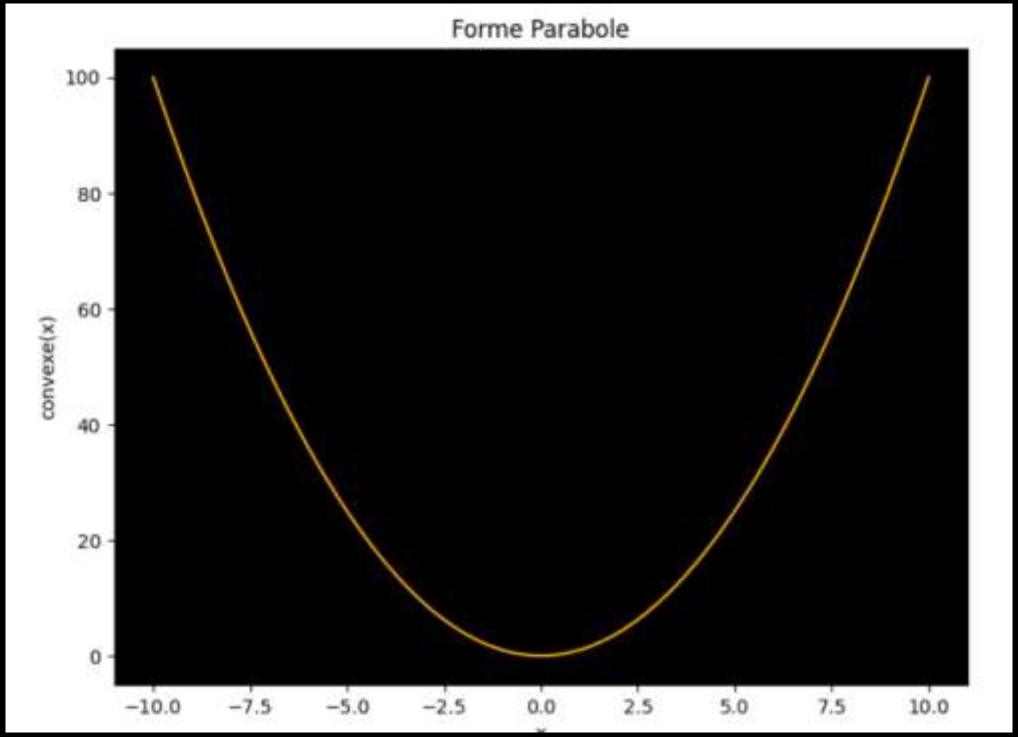
$$\boxed{\log\left(\prod_{i=1}^{m} P(X = x_i)\right) = \log\left(\prod_{i=1}^{m} a_i^{x_i} * (1 - a_i)^{1-x_i}\right)}$$

$$\textit{avec } x_i \in \{0,1\}$$

En effet, pour maximiser la vraisemblance, nous minimisons la fonction de coût car il existe de nombreux algorithmes d'optimisation pour minimiser les fonctions.

L’ algorithme de la descente du gradient

Il s’agit d’un algorithme de minimisation qui consiste à ajuster les parametres W et le biais b afin de minimiser la fonction coût. On calcule ainsi le gradient de la fonction coût



- Si $\frac{\partial F}{\partial x} > 0$, la fonction F augmente lorsque x augmente
- Sinon, la fonction F diminue lorsque x augmente

Dans notre cas, nous chercherons à atteindre un minimum global pour la fonction de coût. Nous dériverons donc la fonction par rapport aux paramètres, et nous augmenterons la valeur de ces paramètres en fonction du signe de la dérivée jusqu'à atteindre le minimum.

La formule conçue pour ce travail est :

$$\boxed{W_{i+1} = W_i - \mu * \frac{\partial L}{\partial W_i}}$$

En resumé

Neurone Formel

$$z = \sum_{i=1}^{m} w_i x_i + w_0$$

$$y(sortie) = \begin{cases} 0\,, Si\ z < 0 \\ 1, Si\ z \geq 0 \end{cases}$$

Perceptron

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

$$-\frac{1}{m} * \sum_{i=1}^{m} \mathrm{y_i} * \log(\mathrm{a_i}) + (1 - \mathrm{y_i}) * \log(1 - \mathrm{a_i})$$

$$W_{i+1} = W_i - \mu * \frac{\partial L}{\partial W_i}$$

Vectorisation des équations pour des donnés larges

1) Vectorisation du modele:

$$\begin{cases} X = \begin{bmatrix} x_1^{(1)} & \cdots & x_n^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(m)} & \cdots & x_n^{(m)} \end{bmatrix} \\ W = \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix} \\ B = \begin{bmatrix} b \\ \vdots \\ b \end{bmatrix} \end{cases} \Rightarrow \quad Z = \begin{bmatrix} z^{(1)} \\ \vdots \\ z^{(m)} \end{bmatrix} = \begin{bmatrix} w_1 * x_1^{(1)} + w_2 * x_2^{(1)} + \cdots + w_n * x_n^{(1)} + b \\ \vdots \\ w_1 * x_1^{(m)} + w_2 * x_2^{(m)} + \cdots + w_n * x_n^{(m)} + b \end{bmatrix} = X * W + B$$

2) Vectorisation de la fonction d'activation:

$$A = \begin{bmatrix} \sigma(z^{(1)}) \\ \vdots \\ \sigma(z^{(m)}) \end{bmatrix} = \sigma\left(\begin{bmatrix} z^{(1)} \\ \vdots \\ z^{(m)} \end{bmatrix}\right) = \sigma(Z)$$

$$avec : \sigma(z) = \frac{1}{1 + e^{-z}} \quad (fonction\ logistique)$$

3) Vectorisation de la fonction cout:

Rappelons que :

$$L = -\frac{1}{m} * \sum_{i=1}^{m} y_i * \log(a_i) + (1 - y_i) * \log(1 - a_i)$$

$$L = -\frac{1}{m} * Somme\_des\_lignes(\underbrace{y * \log}(A) + (1 - y) * \log(1 - A)) \quad , avec \begin{cases} y = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{bmatrix} \\ A = \begin{bmatrix} \sigma(z^{(1)}) \\ \vdots \\ \sigma(z^{(m)}) \end{bmatrix} \end{cases}$$

*Produit terme à terme

4) Vectorisation de l'algorithme de la descente de gradient:

$$W = W - \mu * \frac{\partial L}{\partial W}$$

$$B = B - \mu * \frac{\partial L}{\partial B}$$

$$\begin{cases} W = \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix} \\ B = \begin{bmatrix} b \\ \vdots \\ b \end{bmatrix} \\ \frac{\partial L}{\partial W} = \begin{bmatrix} \frac{\partial L}{\partial w_1} \\ \vdots \\ \frac{\partial L}{\partial w_n} \end{bmatrix} \\ \frac{\partial L}{\partial B} = \begin{bmatrix} \frac{\partial L}{\partial b} \\ \vdots \\ \frac{\partial L}{\partial b} \end{bmatrix} \end{cases}$$

Calcule des gradients

1) Calcule de $\frac{\partial L}{\partial w_i}$:

On a $L = -\frac{1}{m} * \sum_{i=1}^{m} y_i * \log(a_i) + (1 - y_i) * \log(1 - a_i)$

alors $L = -\frac{1}{m} * \sum_{i=1}^{m} y_i * \log\left(\frac{1}{1+ e^{-z_i}}\right) + (1 - y_i) * \log\left(1 - \frac{1}{1+ e^{-z_i}}\right)$

alors $L = -\frac{1}{m} * \sum_{i=1}^{m} (- y_i * \log(1 + e^{-z_i})) + (1 - y_i) * (-\log(1 + e^{-z_i}) - z_i)$

Donc $L = -\frac{1}{m} * \sum_{i=1}^{m} -\log(e^{z_i} + 1) + y_i z_i$

Finalement : $$\boxed{\frac{\partial L}{\partial w_j} = -\frac{1}{m} * \sum_{i=1}^{m} (y_i - a_i) \times x_j^{(i)}}$$

2) Calcule de $\frac{\partial L}{\partial b}$:

De façon analogue : $$\boxed{\frac{\partial L}{\partial b} = -\frac{1}{m} * \sum_{i=1}^{m} (y_i - a_i)}$$

Ainsi sous une forme vectorisée :

$$\frac{\partial L}{\partial W}=\begin{bmatrix}\dfrac{\partial L}{\partial w_1}\\ \vdots \\ \dfrac{\partial L}{\partial w_n}\end{bmatrix}=-\frac{1}{m}\begin{bmatrix}\displaystyle\sum_{i=1}^{m}(y_i-a_i)\ \times \boldsymbol{x_1^{(i)}}\\ \vdots \\ \displaystyle\sum_{i=1}^{m}(y_i-a_i)\ \times \boldsymbol{x_n^{(i)}}\end{bmatrix}=-\ \frac{1}{m}\begin{bmatrix}\boldsymbol{x_1^{(1)}} & \dots & \boldsymbol{x_1^{(m)}}\\ \vdots & \vdots & \vdots \\ \boldsymbol{x_n^{(1)}} & \dots & \boldsymbol{x_n^{(m)}}\end{bmatrix}.\left(\begin{bmatrix}y_1\\ \vdots \\ y_n\end{bmatrix}-\begin{bmatrix}a_1\\ \vdots \\ a_n\end{bmatrix}\right)=-\frac{1}{m}\ X^T.(y-A)$$

De meme:

$$\frac{\partial L}{\partial B}=\begin{bmatrix}\dfrac{\partial L}{\partial b}\\ \vdots \\ \dfrac{\partial L}{\partial b}\end{bmatrix}=-\frac{1}{m}\begin{bmatrix}\displaystyle\sum_{i=1}^{m}(y_i-a_i)\\ \vdots \\ \displaystyle\sum_{i=1}^{m}(y_i-a_i)\end{bmatrix}=-\frac{1}{m}\begin{bmatrix}\sum(\,y-A)\\ \vdots \\ \sum(\,y-A)\end{bmatrix}$$

Programmation du neurone artificiel

Notre Dataset
(X, y)

Init_w_b(n)
avec n:nombre de parameter x_i

algorithme itératif

Model(X, W, b)

Calcule de Z
Return A

Cost(A, y)

Return La function
LogLoss.

Gradient(A, X, y)

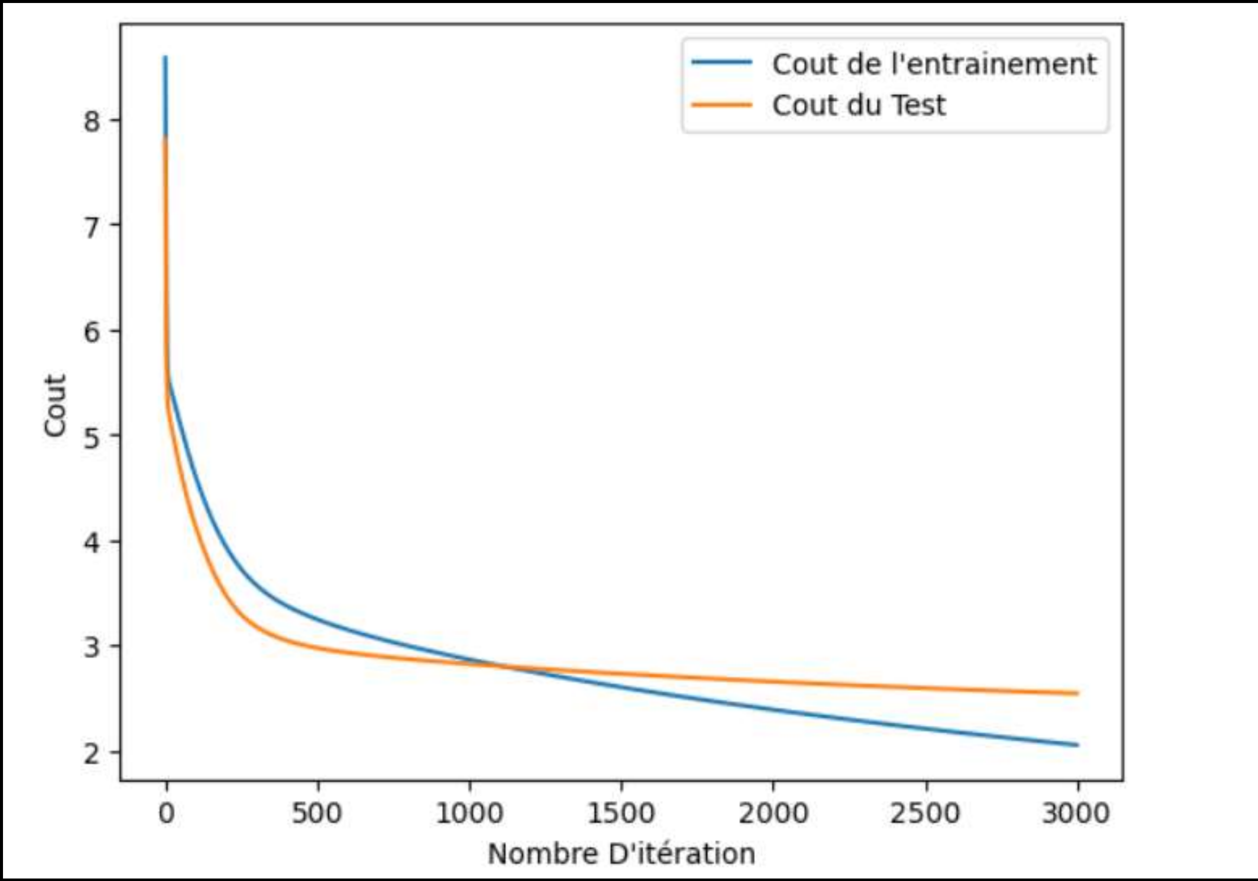
Calcule des gradients.

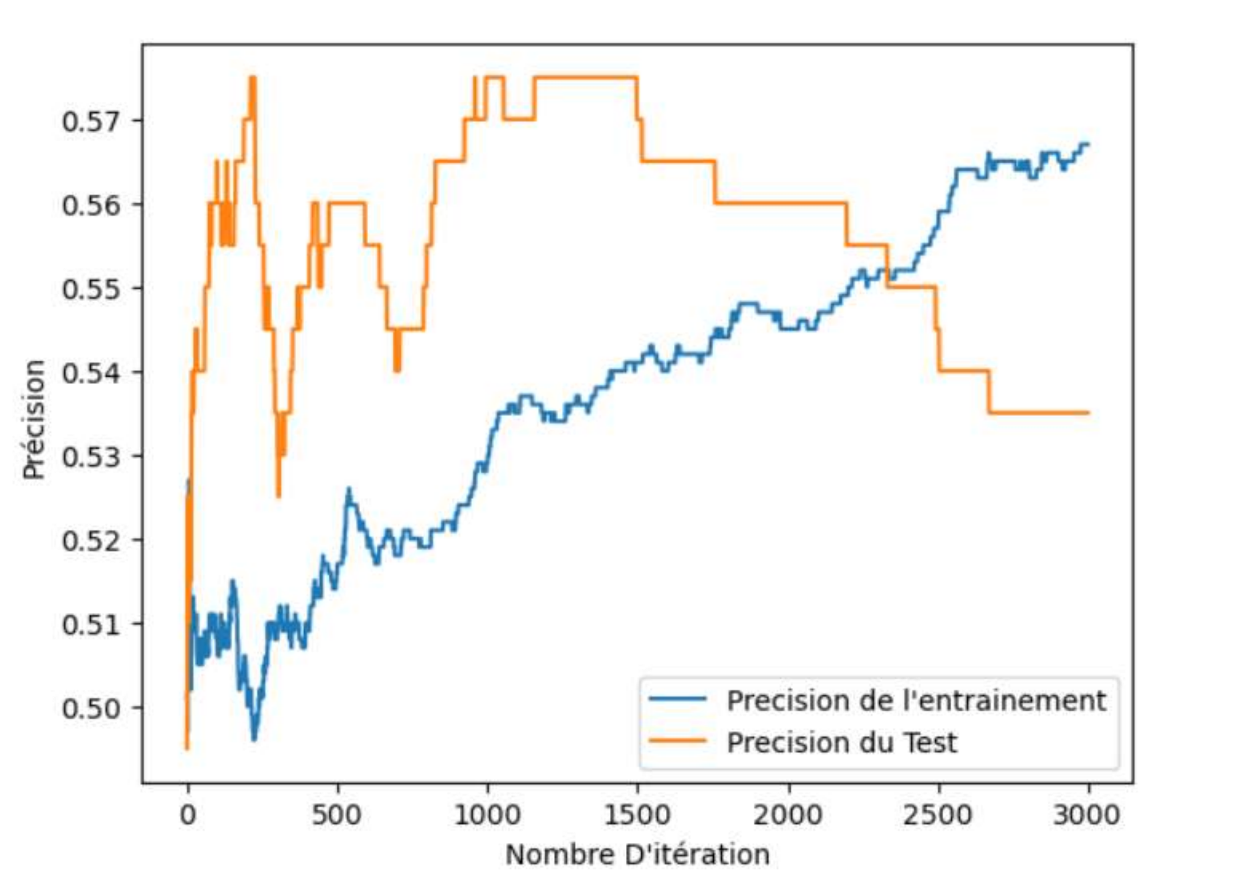
Changement_des_valeurs_w_b(W, b, g_w, g_b)

Return les nouvelles
valeurs de w et b

Application : Chat ou Chien ?

Résultat:

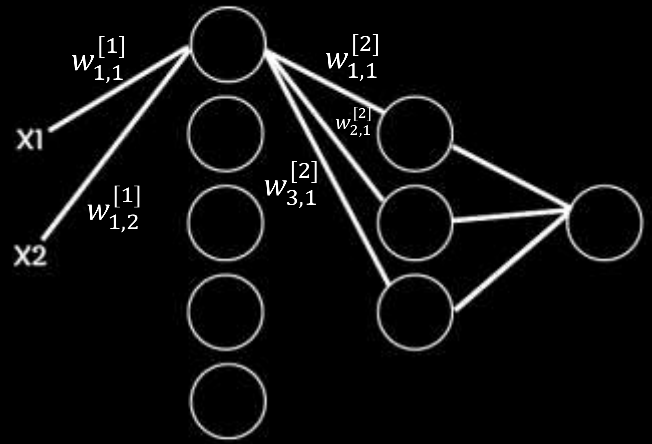




Vers les réseaux de neurones

L’insuffisance du perceptron tout seul nous oblige à travailler avec plusieurs neurones connectés entre eux pour faire des taches plus complexes et plus précis :

1) Forward Propagation:



Notation:

$W_{i,j}^{[\alpha]}$: *poid associé au neurone i et provenant de l'entrée j et α le numero de la couche*

Véctorisation des couches d’un réseau de neurone:

Fonction d’agrégation:

$$\begin{cases} Z^{[1]} = W^{[1]}.X + B^{[1]} \\ Z^{[\alpha]} = W^{[\alpha]}.A^{[\alpha-1]} + B^{[\alpha]} \end{cases}$$

Fonction d’activation:

$$A^{[\alpha]} = \frac{1}{1+e^{-Z^{[\alpha]}}}$$

2) Back Propagation:

On procède de la meme façon que pour un perceptron, sauf qu'ici pour actualiser le poids de connexion $w_{j,q}^{h}$ du neurone j de la couche h-1 vers le neurone q de la couche h, nous devons calculer $\frac{\partial L}{\partial w_{j,q}^{h}}$. Pour ce faire, nous allons appliquer le théoreme de dérivation des fonctions composées (Règle de la chaine).

D'après les calules faits sur la fonction cout d'un perceptron, on obtient :

<u>Cas de C = 2:</u>

$$\frac{\partial L}{\partial W^{[2]}} = \frac{\partial L}{\partial A^{[2]}} \times \frac{\partial A^{[2]}}{\partial Z^{[2]}} \times \frac{\partial Z^{[2]}}{\partial W^{[2]}}$$

$$\frac{\partial L}{\partial B^{[2]}} = \frac{\partial L}{\partial A^{[2]}} \times \frac{\partial A^{[2]}}{\partial Z^{[2]}} \times \frac{\partial Z^{[2]}}{\partial B^{[2]}}$$

$$\frac{\partial L}{\partial W^{[1]}} = \frac{\partial L}{\partial A^{[2]}} \times \frac{\partial A^{[2]}}{\partial Z^{[2]}} \times \frac{\partial Z^{[2]}}{\partial A^{[1]}} \times \frac{\partial A^{[1]}}{\partial Z^{[1]}} \times \frac{\partial Z^{[1]}}{\partial W^{[1]}}$$

$$\frac{\partial L}{\partial B^{[1]}} = \frac{\partial L}{\partial A^{[2]}} \times \frac{\partial A^{[2]}}{\partial Z^{[2]}} \times \frac{\partial Z^{[2]}}{\partial A^{[1]}} \times \frac{\partial A^{[1]}}{\partial Z^{[1]}} \times \frac{\partial Z^{[1]}}{\partial B^{[1]}}$$

Avec:

$$\frac{\partial L}{\partial A} = \sum \frac{-y}{A} + \frac{1-y}{A}$$

$$\frac{\partial A}{\partial Z} = A \times (1 - A)$$

$$\frac{\partial Z}{\partial W} = A$$

On constate alors une répetition de calcule, on utilisera alors la <u>mémoïsation</u> pour optimiser nos calcules.

<u>Notation:</u>

[C] : numéro de la couche.

Notation Majuscule: pour les vecteurs

On pose :

$$base^{[2]} = \frac{\partial L}{\partial A^{[2]}} \times \frac{\partial A^{[2]}}{\partial Z^{[2]}}$$

$$base^{[1]} = \frac{\partial L}{\partial A^{[2]}} \times \frac{\partial A^{[2]}}{\partial Z^{[2]}} \times \frac{\partial Z^{[2]}}{\partial A^{[1]}} \times \frac{\partial A^{[1]}}{\partial Z^{[1]}} = \boldsymbol{base}^{[2]} \times \frac{\partial Z^{[2]}}{\partial A^{[1]}} \times \frac{\partial A^{[1]}}{\partial Z^{[1]}} = \boldsymbol{W}^{[2]^T} \times \boldsymbol{base}^{[2]} \times \boldsymbol{A}^{[1]} \times (\mathbf{1} - \boldsymbol{A}^{[1]})$$

Cas général:

En prenant en considération la technique de mémoïsation, on obtient les équations suivantes pour une couche [C]:

$$\frac{\partial L}{\partial W^{[C]}} = \frac{1}{m} \times base^{[C]}.A^{[C-1]^T}$$: la somme est inclue dans le produit matricielle

$$\frac{\partial L}{\partial B^{[C]}} = \frac{1}{m} \times base^{[C]}$$

$$base^{[C-1]} = W^{[C]^T} \times base^{[C]} \times A^{[C-1]} \times \left(1 - A^{[C-1]}\right)$$: technique de mémoïsation.

Convolution

1)Operation de convolution: (Padding = ‘same’ ou ‘valid’, Stride = (x, y), kernel = taille du filtre)

0	0	0	0	0
0	2	50	20	0
0	20	18	20	0
0	20	20	20	0

Portion d’une image

$*$

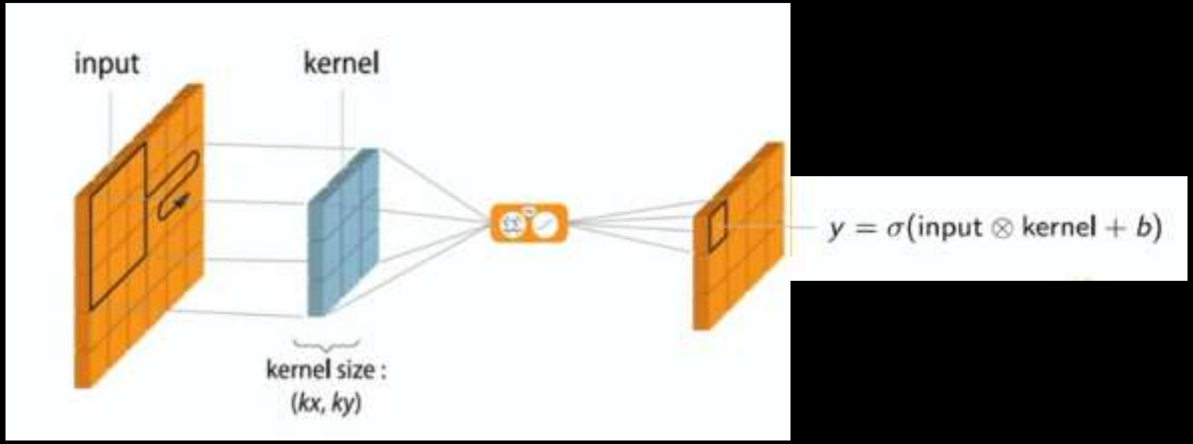
-1	2	2
2	6	2
2	2	-1

Filtre (Kernel)

$=$

134			

2)Neurone de Convolution:



Source : youtube, @CNRS - Formation FIDLE

Maxpooling

1)Operation de maxpooling: (Stride = (x, y), kernel = taille du filtre)

0	0	0	0
0	2	50	20
0	20	18	20
0	20	20	20

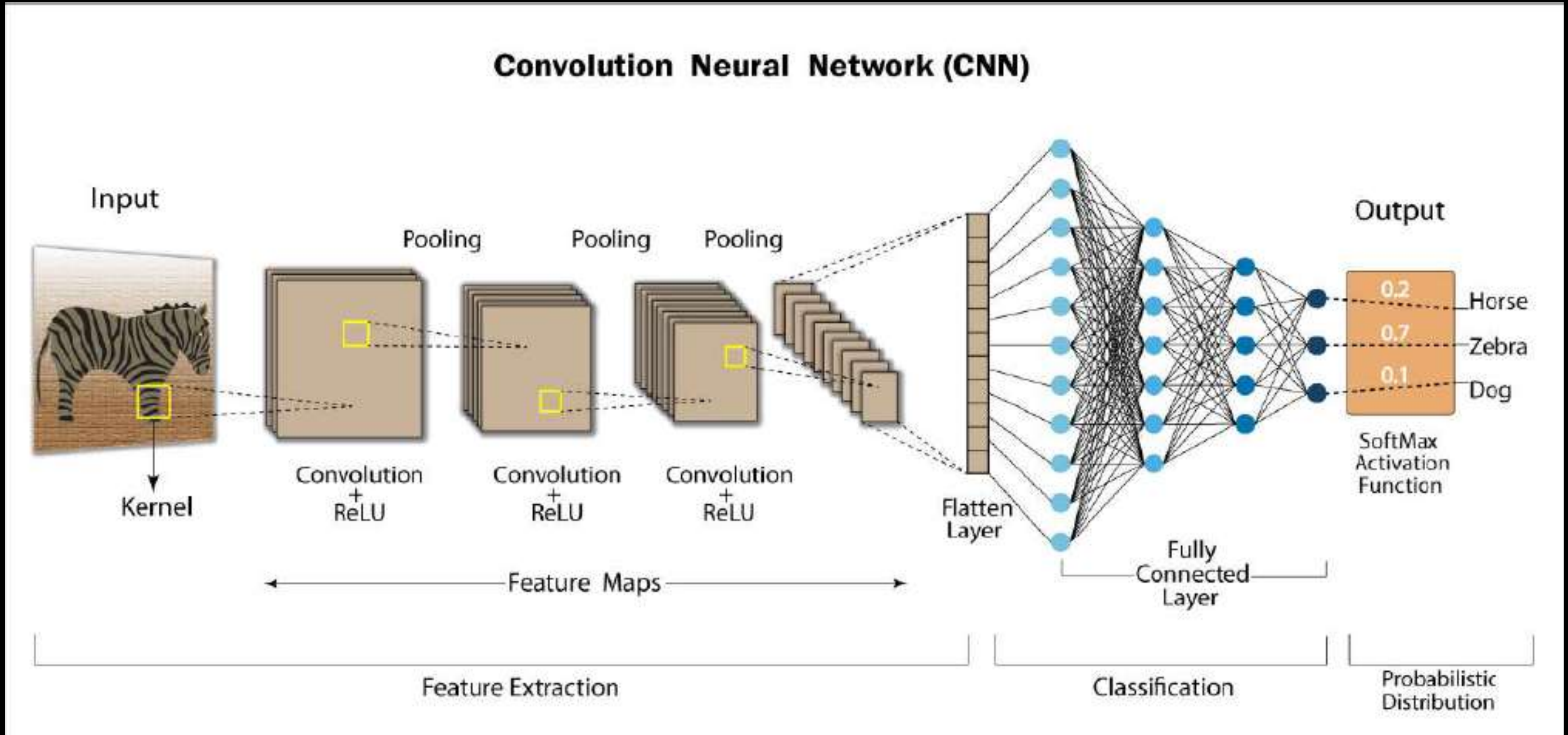
$\xrightarrow{\text{2x2 maxpooling}}$

2	50
20	20

2)Neurone de MaxPooling :

$$\vec{X} \longrightarrow maxpooling(\vec{X}) \longrightarrow \vec{Y}$$

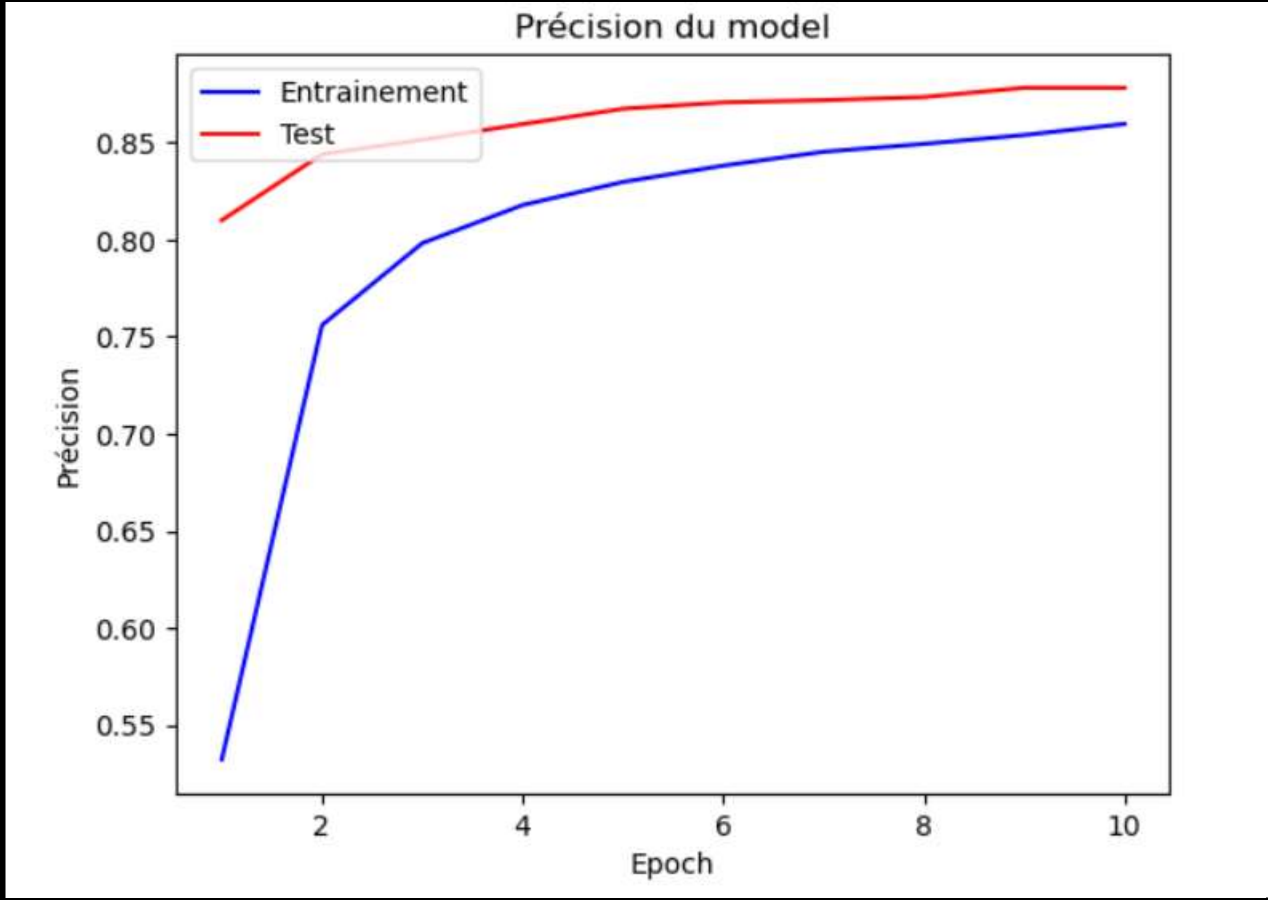
Reconnaissance d’image

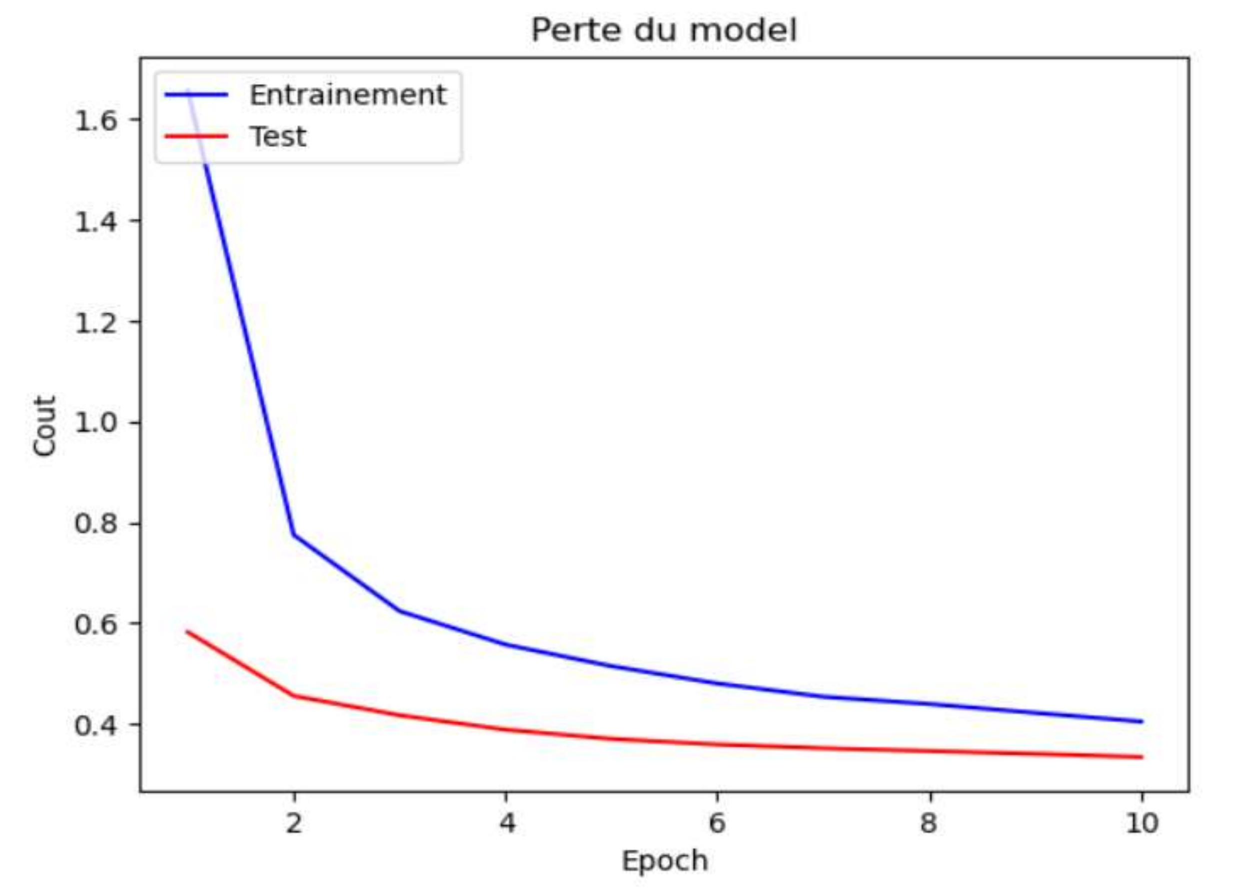


Structure type d’un réseau de classification d’images

Source : https://www.analyticsvidhya.com/

Résultat d'entrainement d'un reseau de neurone convolutif sur la dataset EMNIST





Conclusion

Nous avons réussi à construire un perceptron et un réseau de neurones convolutif. Nous les avons appliqués à la dataset EMNIST et avons interprété les résultats obtenus. Pour mettre à jour les différentes valeurs du réseau, nous avons utilisé les algorithmes de rétropropagation du gradient. Enfin, nous avons élaboré une conception pour la détection des infractions routières.

Annexe 1: Code Perceptron

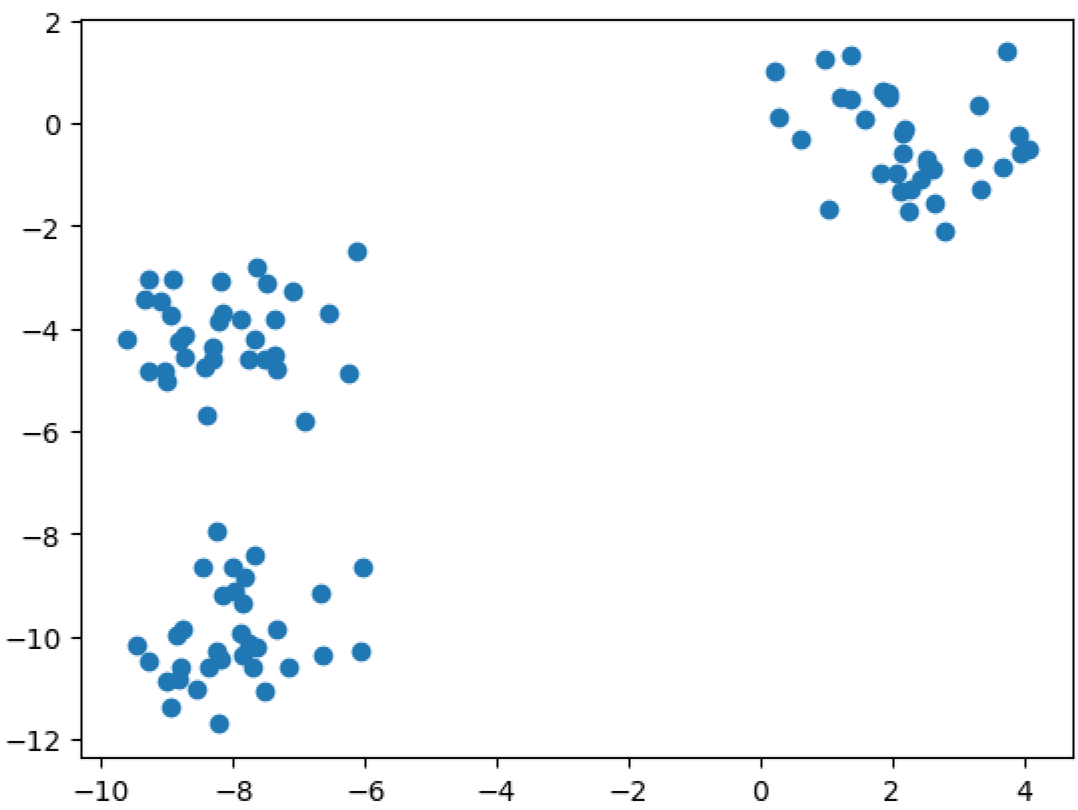
In [31]:
```
#Calcule
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from scipy.special import expit
#Importation de notre dataset
from utilities import *
```

```
Requirement already satisfied: tqdm in c:\users\amine\anaconda3\lib\site-packages (4.64.
1)
Requirement already satisfied: colorama in c:\users\amine\anaconda3\lib\site-packages (f
rom tqdm) (0.4.6)
```

In [4]:
```
#generer un Dataset:
number_of_arguments = 100
X, y = make_blobs(n_samples=100, n_features=number_of_arguments)
```

In [5]:
```
print(f"dim X : {X.shape}")
print(f"dim y : {y.shape}")
colors = [i for i in range(100)]
plt.scatter(X[:,0], X[:,80])
plt.show()
```

```
dim X : (100, 100)
dim y : (100,)
```



In [6]:
```
def init_w_b(n):
    W = np.random.randn(n,1)
    b = np.random.randn(1)
    return (W, b)
```

In [15]:
```
def model(X, W, b):
    Z = X.dot(W) + b
```

```python
    A = 1/(1+np.exp(-Z))
    return A
```

In [16]:
```python
def LogLoss(A, y):
    m = len(y)
    epsilon = 1e-10 #Eviter le probleme du 0
    return (-1/m)*np.sum(y*np.log(A + epsilon)+(1-y)*np.log(1 - A + epsilon)) #Somme des
```

In [69]:
```python
def gradients(A, X, y):
    m = len(y)
    y = y.reshape(y.shape[0], 1) #Numpy donne (100,)
    dw = (1/m)* np.dot(X.T, A-y)
    db = (1/m)* np.sum(A-y)
    return (dw, db)

print(X.shape)
```

```
(100, 100)
```

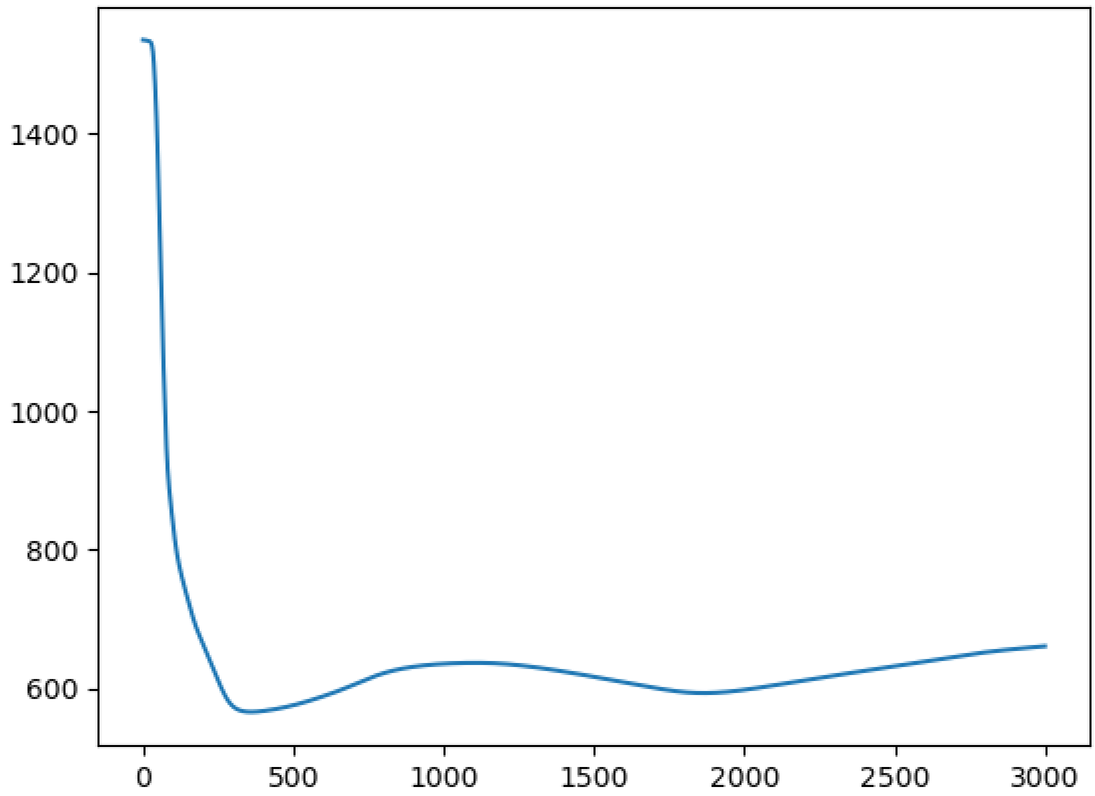
In [62]:
```python
def changementde_w_b(dw, db, W, b, pas_dapprentissage):
    W = W - pas_dapprentissage*dw
    b = b - pas_dapprentissage*db
    return (W, b)
```

Assemblage du neurone

In [2]:
```python
from sklearn.metrics import accuracy_score

def predict(A):
    return A>= 0.5

def neurone_artificiel(x_train, y_train, x_test, y_test, pas_dapprentissage=0.001, nb_di
    #Initialisation:
    W, b     = init_w_b(x_train.shape[1])

    cout_train     = []
    accuracy_train = []
    cout_test      = []
    accuracy_test  = []


    #L'algorithme itératif
    for i in range(nb_diteration):
        A_train = model(x_train, W, b)
        C_train = LogLoss(A_train, y_train)
        G = gradients(A_train, x_train, y_train)
        W, b = changementde_w_b(G[0], G[1], W, b, pas_dapprentissage)
        #ENTRAINEMENT:
        cout_train.append(C_train)
        accuracy_train.append(accuracy_score(y_train, predict(A_train)))
        #TEST:
        A_test = model(x_test, W, b)
        C_test = LogLoss(A_test, y_test)
        cout_test.append(C_test)
        accuracy_test.append(accuracy_score(y_test, predict(A_test)))


    return (W,b, cout_train, accuracy_train, cout_test, accuracy_test)

#result = neurone_artificiel(X, y)
#W, b, cout_train, accuracy_train, cout_test, accuracy_test= result[0], result[1], resul
```

In [23]:
```python
plt.plot([i for i in range(3000)], cout_train)
```

```
plt.show()
```



In [ ]:

In [217…

Annexe 2: Chien ou chat

In [26]:

```
#Importation de notre Dataset:
x_train, y_train, x_test, y_test = load_data()

print(x_train.shape)
print(y_train.shape)
print(x_test.shape)
print(y_test.shape)
```

```
(1000, 64, 64)
(1000, 1)
(200, 64, 64)
(200, 1)
```

Premier problème rencontré (SHAPE)

In [242…

```
#On applique introduit notre dataset au neurone:
na = neurone_artificiel(x_train, y_train)
W, b, cout_train, accuracy_train, cout_test, accuracy_test = na[0], na[1], na[2], na[3],
```

```
C:\Users\amine\AppData\Local\Temp\ipykernel_22324\1146679145.py:3: RuntimeWarning: overf
low encountered in exp
  A = 1/(1+np.exp(-Z))

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[242], line 2
      1 #On applique introduit notre dataset au neurone:
```

```
----> 2 na = neurone_artificiel(x_train, y_train)
      3 W, b, cout = na[0], na[1], na[2]

Cell In[239], line 9, in neurone_artificiel(X, y, pas_dapprentissage, nb_diteration)
      7 for i in range(nb_diteration):
      8     A = model(X, W, b)
----> 9     C = LogLoss(A, y)
     10     cout.append(C)
     11     G = gradients(A, X, y)

Cell In[191], line 4, in LogLoss(A, y)
      2 m = len(y)
      3 epsilon = 1e-10 #Eviter le probleme du 0
----> 4 return (-1/m)*np.sum(y*np.log(A + epsilon)+(1-y)*np.log(1 - A + epsilon))

ValueError: operands could not be broadcast together with shapes (1000,1) (1000,64,1)
```

In [27]:

```
#on applatit les données d'entrée afin qu'elles soient compatibles avec notre modele:
x_train_redimentionné = x_train.reshape(1000, -1) # -1 : reorgonise le reste.. qui est e
x_test_redimentionné = x_test.reshape(200, -1)
print(x_train_redimentionné.shape)
print(x_test_redimentionné.shape)
```

```
(1000, 4096)
(200, 4096)
```

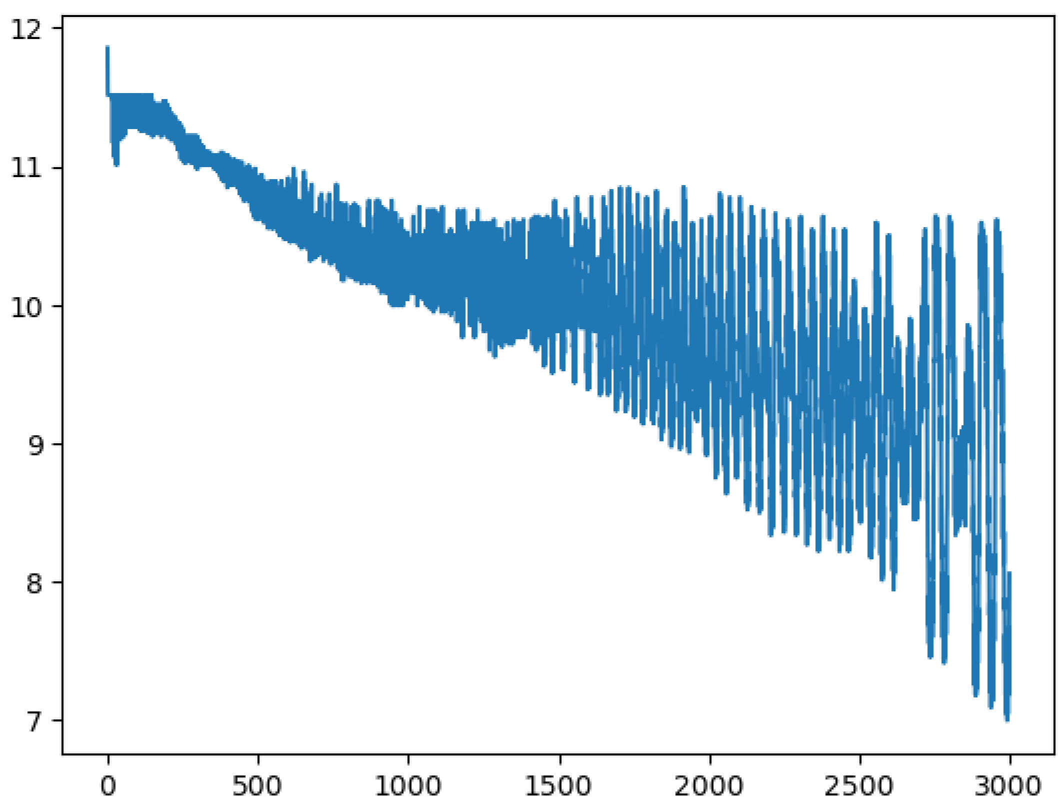
In [243…

```
#ReTEST du programme:
res = neurone_artificiel(x_train_redimentionné, y_train,0.1,3000)

W, b, cout_train, accuracy_train, cout_test, accuracy_test = res[0], res[1], res[2], res
```

```
C:\Users\amine\AppData\Local\Temp\ipykernel_22324\1146679145.py:3: RuntimeWarning: overf
low encountered in exp
  A = 1/(1+np.exp(-Z))
```

In [244…

```
plt.plot([i for i in range(3000)], cout_train)
plt.show()
```



Deuxieme problème rencontré (0 DANS LE LOGARITHME)

In [246…

```
# Fixé en avance par un epsilon qui represente le 0
#(-1/m)*np.sum(y*np.log(A + epsilon)+(1-y)*np.log(1 - A + epsilon))
```

Troisieme problème rencontré (OVERFLOW DE L'EXPONENTIELLE)

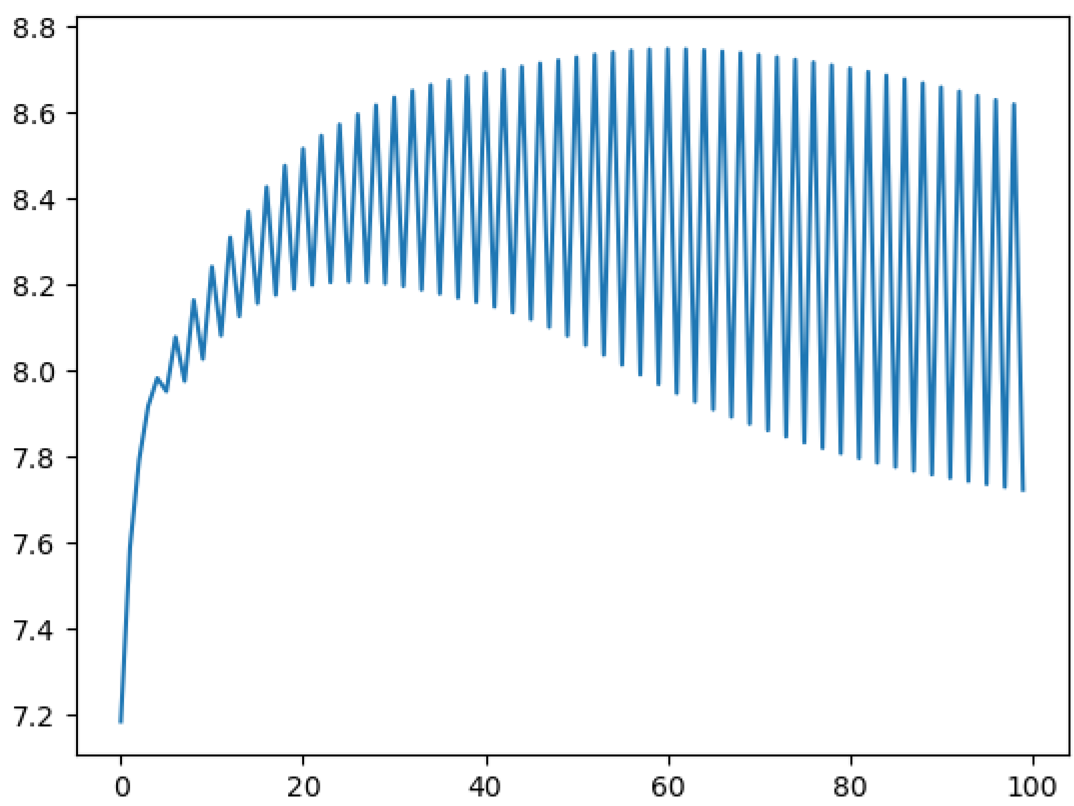
Normalisation MinMax

In [28]:

```
#Parmi Les Solutions pour alleger ce problème est la normalisation:
#X = (X-Xmin)/(Xmax - Xmin)
#Pour Notre Cas les images sont codés en 8 bit donc Xmax = 255,Xmin = 0 generalement:
x_train_redimentionné = x_train.reshape(1000, -1)/x_train.max()
x_test_redimentionné = x_test.reshape(200, -1)/x_test.max()
```

In [258…

```
#Re-entraine notre model:
res = neurone_artificiel(x_train_redimentionné, y_train, pas_dapprentissage=0.1, nb_dite
W, b, cout_train, accuracy_train, cout_test, accuracy_test = res[0], res[1], res[2], res

plt.plot([i for i in range(100)], cout_train)
plt.show()
```



In [250…

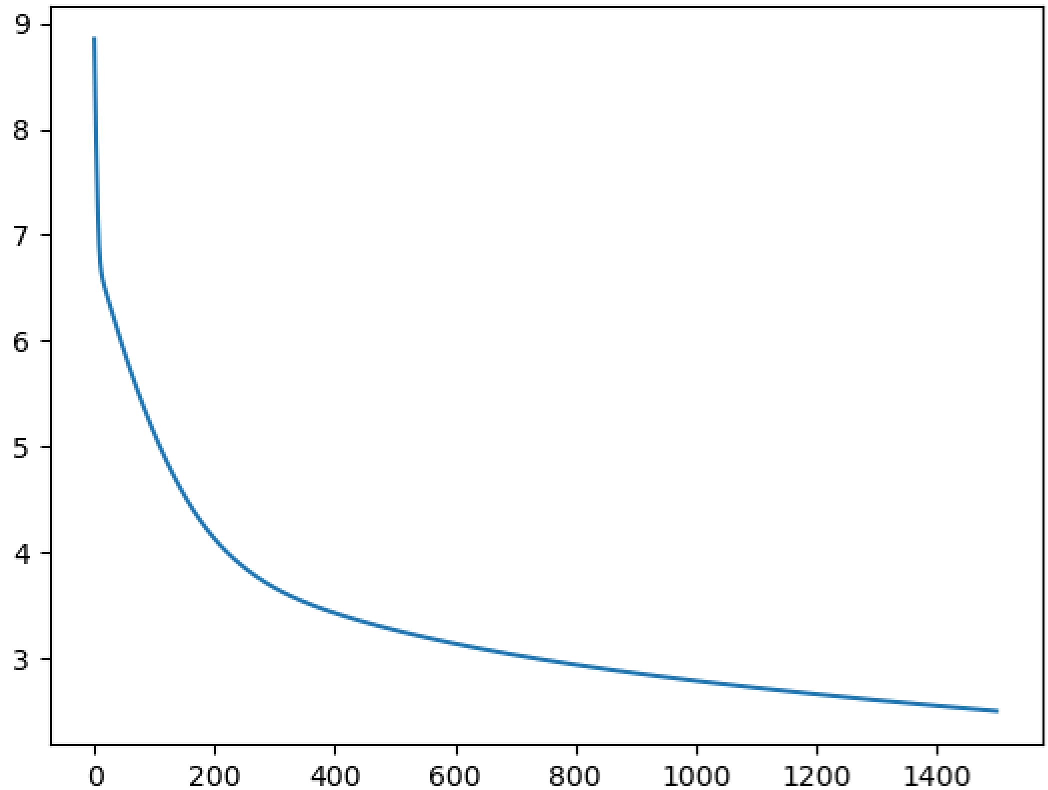
```
#Pas d'overflow!
```

Quatrieme problème rencontré (Le Rebondissement de la fonction cout)

In [263…

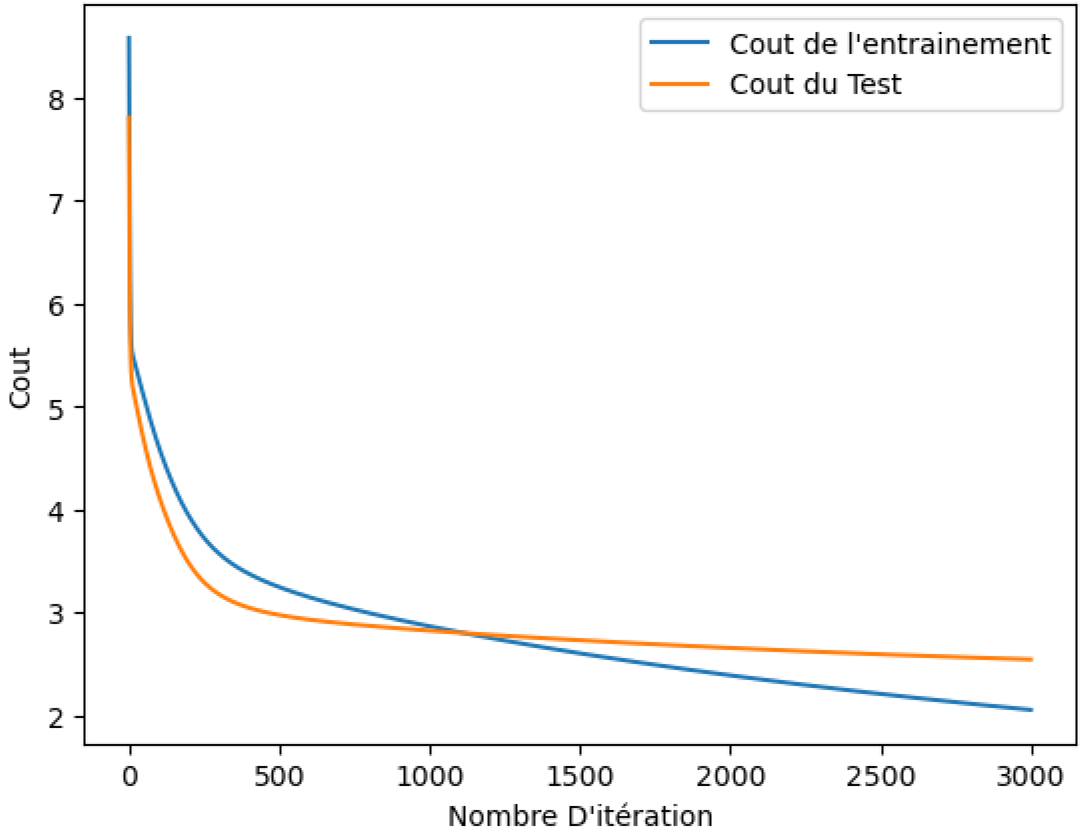
```
#Pour cela il suffit de regler le parametre du neurone (pas d'apprentissage):
"""
Pour un pas d'apprentissage assez grand, notre fonction cout oscille autour de son minim
```

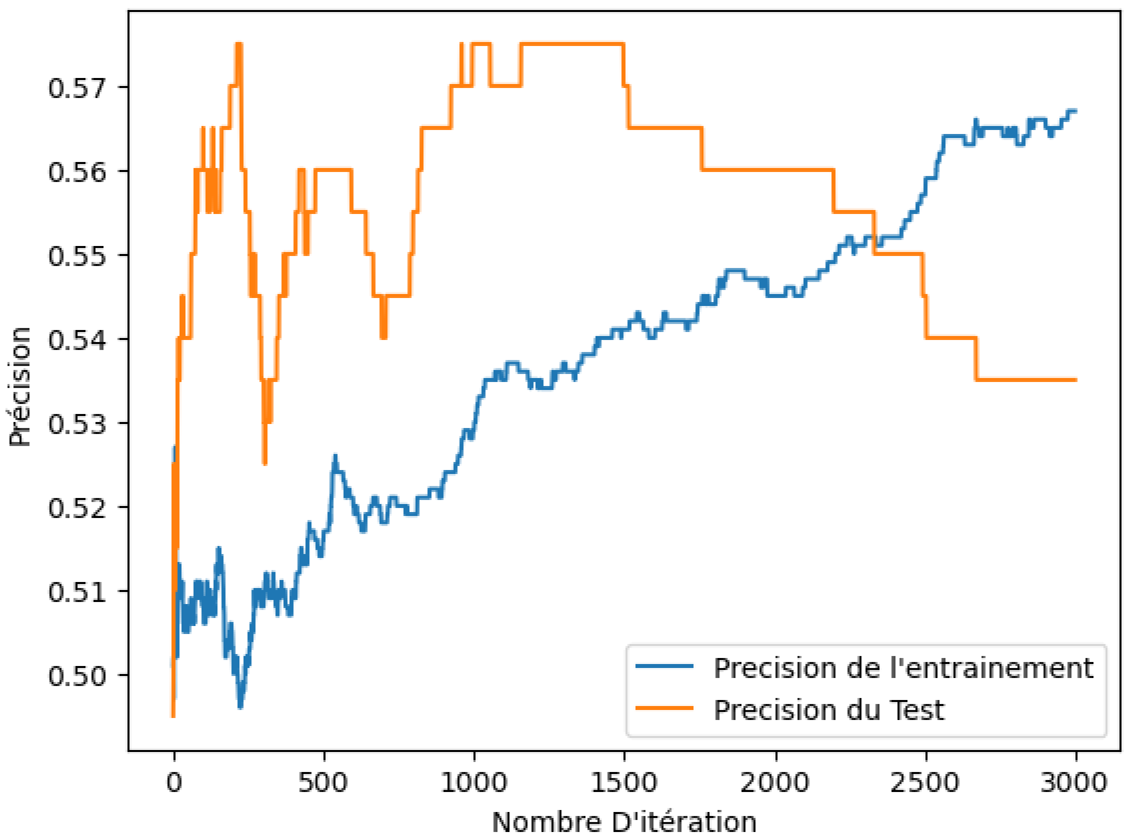
```
de part et d'autre de la valeur minimal, on réduit alors le pas d'apprentissage (pas tro
 à effacer !
"""

res = neurone_artificiel(x_train_redimentionné, y_train, pas_dapprentissage=0.01, nb_dit
W, b, cout_train, accuracy_train, cout_test, accuracy_test = res[0], res[1], res[2], res
plt.plot([i for i in range(1500)], cout_train)
plt.show()
```



In [61]:

```
#On remarque qu'il peut apprendre plus:
n_diteration = 3000
res = neurone_artificiel(x_train_redimentionné, y_train,x_test_redimentionné,y_test, pas_
W, b, cout_train, accuracy_train, cout_test, accuracy_test = res[0], res[1], res[2], res
plt.figure(2)
plt.plot([i for i in range(n_diteration)], cout_train, label="Cout de l'entrainement")
plt.plot([i for i in range(n_diteration)], cout_test, label="Cout du Test")
plt.xlabel("Nombre D'itération")
plt.ylabel("Cout")

plt.legend()
plt.show()

plt.plot([i for i in range(n_diteration)], accuracy_train, label="Precision de l'entrain
plt.plot([i for i in range(n_diteration)], accuracy_test, label="Precision du Test")
plt.xlabel("Nombre D'itération")
plt.ylabel("Précision")

plt.legend()
plt.show()
```


Cout de l'entrainement
Cout du Test
Cout
Nombre D'itération
0
500
1000
1500
2000
2500
3000
2
3
4
5
6
7
8


Precision de l'entrainement
Precision du Test
Précision
Nombre D'itération
0
500
1000
1500
2000
2500
3000
0.50
0.51
0.52
0.53
0.54
0.55
0.56
0.57

In [ ]:

Annexe 3: Réseau de neurone

Codage du Forward Propagation

In [51]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs, make_circles
from sklearn.metrics import accuracy_score, log_loss
from utilities import *
```

In [121…
```
def initialisation(dimensions):
    ln_dim = len(dimensions)
    parametres = {}
    for i in range(1, ln_dim):
        parametres['W' + str(i)] = np.random.randn(dimensions[i], dimensions[i-1])
        parametres['b' + str(i)] = np.random.randn(dimensions[i], 1)
    return parametres
```

```
(16, 1)
```

In [91]:
```
def sigmoid(Z):
    return 1/(1+np.exp(-Z))

def forward_propagation(X, parametres):
    activations = {'A0': X}
    couches = len(parametres) // 2 #chaque couche contient Wi et Bi,nb de couches
    for couche in range(1, couches + 1):
        W = parametres['W' + str(couche)]
        B = parametres['b' + str(couche)]
        Z = W.dot(activations['A' + str(couche-1)]) + B #BroadCasting
        activations['A' + str(couche)] = sigmoid(Z)

    return activations
```

Codage de la Retropropagation

In [191…
```
def back_propagation(y, parametres, activations):
    m = y.shape[1]
    C = len(parametres) // 2

    dZ = activations['A' + str(C)] - y
    gradients = {}

    for c in reversed(range(1, C + 1)):
        print(dZ.shape)
        print(activations['A' + str(c - 1)].T.shape)
        gradients['∂W' + str(c)] = 1/m * np.dot(dZ, activations['A' + str(c - 1)].T)
        gradients['∂b' + str(c)] = 1/m * np.sum(dZ, axis=1, keepdims=True)
        if c > 1:
            dZ = np.dot(parametres['W' + str(c)].T, dZ) * activations['A' + str(c - 1)]

    return gradients
```

Mise à jour des parametres

In [192…
```
def changement_w_b(gradients, parametres, learning_rate):
```

```python
    couches = len(parametres) // 2
    for couche in range(1, couches +1):
        ec_w = learning_rate*gradients['∂W'+str(couche)]
        ec_b = learning_rate*gradients['∂b'+str(couche)]
        parametres['W'+str(couche)] = parametres['W'+str(couche)] - ec_w
        parametres['b'+str(couche)] = parametres['b'+str(couche)] - ec_b

    return parametres
```

Codage du reseau de neurone

In [230…

```python
def LogLoss(A, y):
    m = len(y)
    epsilon = 1e-10 #Eviter le probleme du 0
    return (-1/m)*np.sum(y*np.log(A + epsilon)+(1-y)*np.log(1 - A + epsilon)) #Somme des


def predict(X, parametres):
    activations = forward_propagation(X, parametres)
    C = len(parametres) // 2
    Af = activations['A' + str(C)]
    return Af >= 0.5

def reseau_de_neurone(X, y, hidden_layers = (16, 16, 16), learning_rate = 0.001, nb_dite

    # initialisation parametres
    dimensions = list(hidden_layers)
    dimensions.insert(0, X.shape[0])
    dimensions.append(y.shape[0])
    np.random.seed(1)
    parametres = initialisation(dimensions)

    # tableau numpy contenant les futures accuracy et log_loss
    training_history = np.zeros((int(nb_diteration), 2))

    C = len(parametres) // 2

    # gradient descent
    for i in range(nb_diteration):

        activations = forward_propagation(X, parametres)
        gradients = back_propagation(y, parametres, activations)
        parametres = changement_w_b(gradients, parametres, learning_rate)
        Af = activations['A' + str(C)]

        # calcul du log_loss et de l'accuracy
        training_history[i, 0] = (log_loss(y.flatten(), Af.flatten()))
        y_pred = predict(X, parametres)
        training_history[i, 1] = (accuracy_score(y.flatten(), y_pred.flatten()))

    # Plot courbe d'apprentissage
    plt.figure(figsize=(12, 4))
    plt.subplot(1, 2, 1)
    plt.plot(training_history[:, 0], label='train loss')
    plt.legend()
    plt.subplot(1, 2, 2)
    plt.plot(training_history[:, 1], label='train acc')
    plt.legend()
    plt.show()

    return training_history
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

Annexe 4: Codage d'un réseau de neurone

Couches

In [1]:
```
import numpy as np # Essentiel pour les array, calculs, ...
import matplotlib.pyplot as plt # Pour les graphiques/images
from scipy import signal # Calcul de convolution : le coder soi-même est simple mais les
# sont très couteuses et n'optimisent pas les calculs (réseau trop lent)
from skimage.measure import block_reduce # Calcul de maxpooling (même raison que convolu
# la rétropropagation, il n'y a pas de telle fonction donc on utilise
# des boucles etc ce qui rend le programme plus lent)
import time # Chronomètre
import scipy # Pour la
```

In [ ]:
```
class Dense():
    def __init__(self,taille_entree,taille_sortie):
        self.poids = np.random.randn(taille_sortie, taille_entree)
        self.biais = np.random.randn(taille_sortie,1)

    def propagation_directe(self, entree):
        self.entree = entree
        return (np.dot(self.poids, self.entree) + self.biais)

    def retropropagation(self, grad_sortie, pas_apprentissage):
        grad_poids = np.dot(grad_sortie, self.entree.T)
        self.poids -= pas_apprentissage * grad_poids
        self.biais -= pas_apprentissage * grad_sortie
        return np.dot(self.poids.T, grad_sortie)
```

In [2]:
```
class Convolution():
    def __init__(self, dimensions_entree, taille_filtre, profondeur_sortie):
        """Profondeur_sortie = nombre de filtres"""
        self.profondeur_sortie = profondeur_sortie
        self.profondeur_entree, self.hauteur_entree, self.largeur_entree = dimensions_en
        self.dimensions_entree = dimensions_entree
        self.dimensions_sortie = (profondeur_sortie, self.hauteur_entree - taille_filtre
                                  - taille_filtre + 1)
        self.dimensions_filtres = (profondeur_sortie, self.profondeur_entree, taille_fil
        self.filtres = np.random.randn(*self.dimensions_filtres)
        self.biais = np.random.randn(*self.dimensions_sortie)
    def propagation_directe(self, entree):
        self.entree = entree
        self.sortie = np.copy(self.biais)
        for i in range(self.profondeur_sortie) :
            for j in range(self.profondeur_entree):
                self.sortie[i] += signal.correlate2d(self.entree[j], self.filtres[i,j],
        return self.sortie


    def retropropagation(self, grad_sortie, pas_apprentissage):
        grad_filtres = np.zeros(self.dimensions_filtres)
        grad_entree = np.zeros(self.dimensions_entree)
        for i in range(self.profondeur_sortie):
            for j in range(self.profondeur_entree):
                grad_filtres[i,j] = signal.correlate2d(self.entree[j], grad_sortie[i], "
                grad_entree[j] += signal.correlate2d(grad_sortie[i], self.filtres[i,j],
                grad_biais = grad_sortie
        # Mise à jour
        self.filtres -= pas_apprentissage * grad_filtres
```

```
        self.biais -= pas_apprentissage * grad_biais
        return grad_entree
```

In [4]:
```
class Maxpooling():
    def __init__(self, dimensions_entree, taille_filtre, stride):
        self.profondeur_entree, self.hauteur_entree, self.largeur_entree = dimensions_en
        self.dimensions_entree = dimensions_entree
        self.taille_filtre = taille_filtre
        self.filtre_h, self.filtre_l = taille_filtre
        self.stride = stride
        self.hauteur_sortie = int(1 + (self.hauteur_entree - self.filtre_h) / stride)
        self.largeur_sortie = int(1 + (self.largeur_entree - self.filtre_l) /stride)
        self.profondeur_sortie = self.profondeur_entree
    def propagation_directe(self, entree):
        self.entree = entree
        sortie = np.zeros((self.profondeur_sortie, self.hauteur_sortie, self.largeur_sor
        stride = self.stride
        for c in range(self.profondeur_sortie):
            for i in range(self.hauteur_sortie):
                for j in range(self.largeur_sortie):
                    sortie[c, i, j] = np.max(entree[c, i * stride : i * stride + self.fi
                                                                + self.filtre_l ])
        return sortie
    def retropropagation(self, grad_sortie, pas_apprentissage):
        grad_entree = np.zeros((self.profondeur_entree, self.hauteur_entree, self.largeu
        entree = self.entree
        stride = self.stride
        for c in range(self.profondeur_sortie):
            for i in range(self.hauteur_sortie):
                for j in range(self.largeur_sortie):
                    intermediaire = entree[c, i * stride : i * stride + self.filtre_h ,
                    i_max, j_max = np.where(np.max(intermediaire) == intermediaire)
                    i_max, j_max = i_max[0], j_max[0]
                    grad_entree[c, i * stride : i * stride + self.filtre_h, j *stride :

        return grad_entree
```

In [5]:
```
class Dropout():
    def __init__(self, q_bernouilli):
        self.p_bernouilli = 1 - q_bernouilli

    def propagation_directe(self, entree):
        self.entree = entree
        self.masque_binaire = np.random.binomial(1, self.p_bernouilli, size = entree.sha
        self.sortie = entree * self.masque_binaire
        return self.sortie

    def retropropagation(self, grad_sortie, pas_apprentissage):
        return grad_sortie * self.masque_binaire
```

In [6]:
```
class Dimension():
    def __init__(self,dimension_entree, dimension_sortie):
        self.dimension_entree = dimension_entree
        self.dimension_sortie = dimension_sortie

    def propagation_directe(self, entree):
        return np.reshape(entree, self.dimension_sortie)

    def retropropagation(self, grad_sortie, pas_apprentissage):
        return np.reshape(grad_sortie, self.dimension_entree)
```

Fonctions d'activation

In [7]:
```python
# Tangente hyperbolique
class Tanh():
    def __init__(self):
        tanh = lambda x : np.tanh(x)
        tanh_p = lambda x : 1 - np.tanh(x)**2
        self.activation = tanh
        self.derivee_activation = tanh_p

    def propagation_directe(self, entree):
        self.entree = entree
        return self.activation(self.entree)

    def retropropagation(self, grad_sortie, pas_apprentissage):
        return np.multiply(grad_sortie, self.derivee_activation(self.entree))
```

In [8]:
```python
# Sigmoide
class Sigmoide():
    def __init__(self):
        def sigmoide(x):
            return scipy.special.expit(x)

        def sigmoide_p(sig):
            return sig * (1- sig) #derive

        self.activation = sigmoide
        self.derivee_activation = sigmoide_p

    def propagation_directe(self, entree):
        self.entree = entree
        self.sig = self.activation(self.entree)
        return self.sig

    def retropropagation(self, grad_sortie, pas_apprentissage):
        return np.multiply(grad_sortie, self.derivee_activation(self.sig))
```

In [12]:
```python
# ReLU
class Relu():
    def propagation_directe(self, entree):
        self.entree = entree
        self.sortie = np.maximum(0, entree)
        return self.sortie

    def retropropagation(self, grad_sortie, pas_apprentissage):
        inter = grad_sortie.copy()
        inter[inter <= 0] = 0
        return inter
```

In [13]:
```python
# Softmax
class Softmax():
    def propagation_directe(self, entree):
        maxi = np.max(entree)
        entree = entree - maxi
        expo = np.exp(entree)
        self.sortie = expo/np.sum(expo)
        return self.sortie

    def retropropagation(self, grad_sortie, pas_apprentissage): # L'erreur utilisée étan
# en complément de softmax, et comme on a déjà calculé la dérivée de l'erreur par rappor
# l'entrée du softmax, on la transmet (cf cce)
        return grad_sortie
```

Erreur

In [10]:
```
# Erreur quadratique moyenne:
def eqm(sortie_voulue, sortie):
    return np.mean(np.power(sortie_voulue - sortie, 2))

def eqm_derivee(sortie_voulue, sortie):
    return 2 * (sortie - sortie_voulue) / np.size(sortie)
```

In [ ]:
```
# Erreur croisée binaire (binary crossentropy)
def bce(sortie_voulue, sortie):
    sortie_voulue = np.clip(sortie_voulue, 1e-7, 1 - 1e-7) # Pour ne pas avoir de divisi
    sortie = np.clip(sortie, 1e-7, 1 - 1e-7) # Pour ne pas avoir de division par zero/ l
    return -np.mean(sortie_voulue * np.log(sortie) + (1 - sortie_voulue) * np.log(1-sort

def bce_derivee(sortie_voulue, sortie):
    sortie = np.clip(sortie, 1e-7, 1 - 1e-7)
    sortie_voulue = np.clip(sortie_voulue, 1e-7, 1 - 1e-7)
    return ((1 - sortie_voulue) / (1 - sortie) - sortie_voulue/ (sortie)) / np.size(sort
```

In [14]:
```
#Erreur croisée (categorical crossentropy noté cce)
def cce(sortie_voulue, sortie):
    sortie_voulue = np.clip(sortie_voulue, 1e-7, 1 - 1e-7)
    sortie = np.clip(sortie, 1e-7, 1 - 1e-7)
    return -np.sum(np.log(sortie) * sortie_voulue)
def cce_derivee(sortie_voulue, sortie): # Ici, pour un soucis de rapidité de calcul, on
# la dérivée de l'erreur par rapport à l'entrée du softmax en utilisant
# cce comme erreur à la dernière couche. En effet, la formule est bien
# plus simple comme ceci, et on utilisera toujours Softmax comme fonction
# d'activation en dernière couche avec cce (cf. Softmax)
    sortie_voulue = np.clip(sortie_voulue, 1e-7, 1 - 1e-7)
    sortie = np.clip(sortie, 1e-7, 1 - 1e-7)
    return sortie - sortie_voulue
```

Assemblage du réseau

In [16]:
```
def precision_erreur(res, entree_t, sortie_t):
    succes = 0
    total = 0
    e = 0
    for i in range(len(entree_t)):
        s = res.prediction(entree_t[i])
        e += res.erreur(sortie_t[i],s)
        maxi = np.argmax(s)
        if maxi == np.argmax(sortie_t[i]):
            succes += 1

        total += 1
    return (succes/total,e/total)
```

In [18]:
```
class Reseau():

    def __init__(self, couches, erreur, erreur_derivee):
        self.couches = couches
        self.erreur = erreur
        self.erreur_derivee = erreur_derivee

    def prediction(self, entree):
        sortie = entree
        for couche in self.couches:
            sortie = couche.propagation_directe(sortie)
```

```python
        return sortie

    def entrainement(self,entree_e , sortie_e , entree_t, sortie_t, iterations, pas_appr
        nb_entrainements = len(entree_e)
        liste_erreur = []
        liste_erreur_t = []
        precision_e = []
        precision_t = []
        tini = time.time()
        for itera in range(iterations):
            print("Itération numéro ", itera+1)
            erreur = 0
            titer = time.time()
            succes_e = 0
            tot_e = 0
            for i in range(nb_entrainements):
                # Propagation directe
                sortie = entree_e[i]

                for couche in self.couches:
                    sortie = couche.propagation_directe(sortie)
                if np.argmax(sortie) == np.argmax(sortie_e[i]):
                    succes_e += 1
                tot_e += 1

                # Ajout de l'erreur
                erreur += self.erreur(sortie_e[i],sortie)

                # Rétropropagation
                sortie_retro = self.erreur_derivee(sortie_e[i], sortie)
                for couche in reversed(self.couches):
                    sortie_retro = couche.retropropagation(sortie_retro,pas_apprentissag

            # Traitement de l'erreur
            erreur /= nb_entrainements
            liste_erreur.append(erreur)
            print("Erreur : ", erreur)
            prec_test, err_test = precision_erreur(self, entree_t, sortie_t)
            liste_erreur_t.append(err_test)
            print("Erreur sur la base de test :", liste_erreur_t[-1])
            precision_e.append(succes_e/tot_e)
            precision_t.append(prec_test)
            print("Précision sur la base entrainement :", precision_e[-1])
            print("Précision sur la base test :", precision_t[-1])
            print("Durée de l'itération :", round(time.time() - titer,2) , "s")
            print()

        print()
        print("Fin de l'apprentissage")
        print("Durée de l'apprentissage : ", round(time.time() - tini,2) , "s")
        return (liste_erreur, liste_erreur_t, precision_e, precision_t)

    def test(self, entree_t, sortie_t):
        nb_entrainements = len(entree_t)
        erreur = 0
        for i in range(nb_entrainements):

            # Propagation directe
            sortie = entree_t[i]
            for couche in self.couches:
                sortie = couche.propagation_directe(sortie)

            # Ajout de l'erreur
            erreur += self.erreur(sortie_t[i],sortie)
```

```
        erreur /= nb_entrainements
        return erreur
```

In [ ]:

```

```

Annexe 5 : Entrainement d'un reseau de neurone convolutif sur la dataset EMNIST en utilisant keras

In [ ]:
```
# Importation des librairies
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
import cv2

#keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.utils import np_utils
import sklearn.metrics as metrics
```

In [30]:
```
train = pd.read_csv(r"C:\Users\amine\TIPE_CODE\EMNIST\emnist-balanced-train.csv",delimit
test = pd.read_csv(r"C:\Users\amine\TIPE_CODE\EMNIST\emnist-balanced-test.csv", delimite
mapp = pd.read_csv(r"C:\Users\amine\TIPE_CODE\EMNIST\emnist-balanced-mapping.txt", delim
                   index_col=0, header=None, squeeze=True)
print("Train: %s, Test: %s, Map: %s" %(train.shape, test.shape, mapp.shape))
```

```
Train: (112799, 785), Test: (18799, 785), Map: (47,)
```

```
C:\Users\amine\AppData\Local\Temp\ipykernel_19076\1858441103.py:3: FutureWarning: The sq
ueeze argument has been deprecated and will be removed in a future version. Append .sque
eze("columns") to the call to squeeze.


  mapp = pd.read_csv(r"C:\Users\amine\TIPE_CODE\EMNIST\emnist-balanced-mapping.txt", del
imiter = ' ', \
```

In [31]:
```
# Constants
HEIGHT = 28
WIDTH = 28
```

In [32]:
```
# Split x and y
train_x = train.iloc[:,1:]
train_y = train.iloc[:,0]
del train

test_x = test.iloc[:,1:]
test_y = test.iloc[:,0]
del test
```

In [ ]:

In [33]:
```
def rotate(image):
    image = image.reshape([HEIGHT, WIDTH])
    image = np.fliplr(image)
    image = np.rot90(image)
    return image
```

In [34]:
```
train_x = np.asarray(train_x)
train_x = np.apply_along_axis(rotate, 1, train_x)
print ("train_x:",train_x.shape)

test_x = np.asarray(test_x)
```

```
test_x = np.apply_along_axis(rotate, 1, test_x)
print ("test_x:",test_x.shape)
```

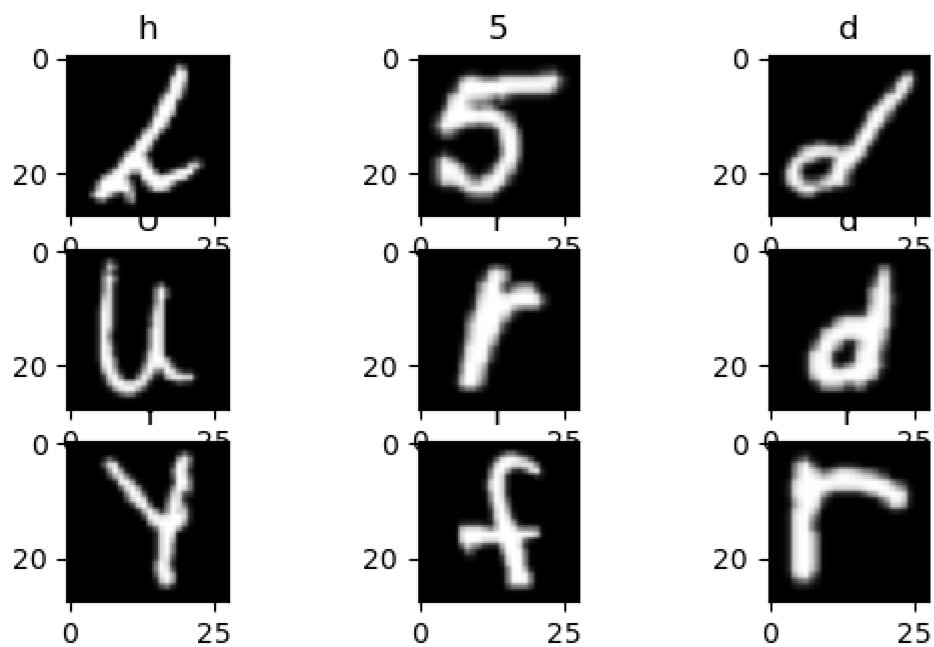
```
train_x: (112799, 28, 28)
test_x: (18799, 28, 28)
```

In [35]:

```
# Normalisation
train_x = train_x.astype('float32')
train_x /= 255
test_x = test_x.astype('float32')
test_x /= 255
```

In [12]:

```
# Affichage de qql elements de la dataset
for i in range(100, 109):
    plt.subplot(330 + (i+1))
    plt.imshow(train_x[i], cmap=plt.get_cmap('gray'))
    plt.title(chr(mapp[train_y[i]]))
```



In [13]:

```
# nombre de classes:
num_classes = train_y.nunique()
```

In [14]:

```
# Affichage plus clair:
train_y = np_utils.to_categorical(train_y, num_classes)
test_y = np_utils.to_categorical(test_y, num_classes)
```

```
train_y:  (112799, 47)
test_y:  (18799, 47)
```

In [15]:

```
# Redimentionne l'entrée pour qu'elle 'fit' à notre model
train_x = train_x.reshape(-1, HEIGHT, WIDTH, 1)
test_x = test_x.reshape(-1, HEIGHT, WIDTH, 1)
```

In [16]:

```
# Division de notre dataset:
train_x, val_x, train_y, val_y = train_test_split(train_x, train_y, test_size= 0.10, ran
```

In [17]:

```
#Création du model
model = Sequential()

model.add(Conv2D(filters=128, kernel_size=(5,5), padding = 'same', activation='relu',\
                 input_shape=(HEIGHT, WIDTH,1)))
model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))
model.add(Conv2D(filters=64, kernel_size=(3,3) , padding = 'same', activation='relu'))
```

```
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Flatten())
model.add(Dense(units=128, activation='relu'))
model.add(Dropout(.5))
model.add(Dense(units=num_classes, activation='softmax'))
```

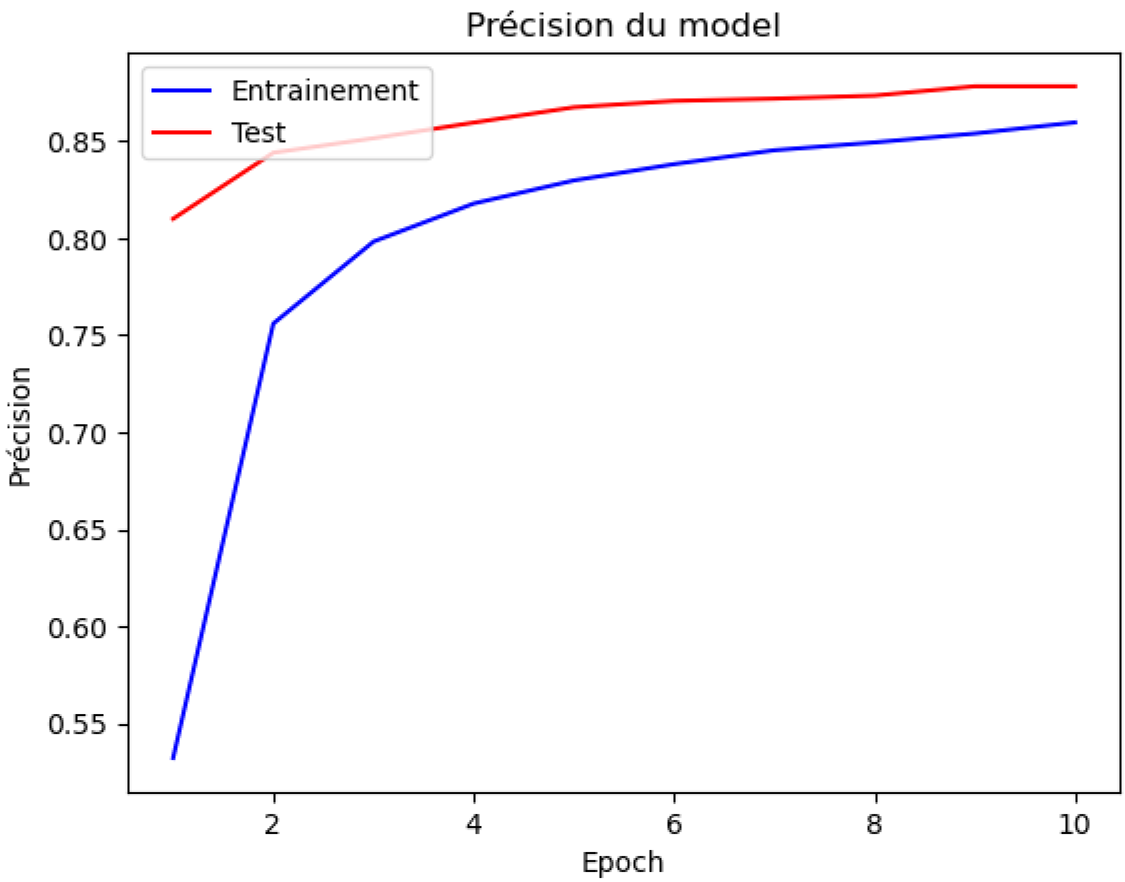
In [18]:
```
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

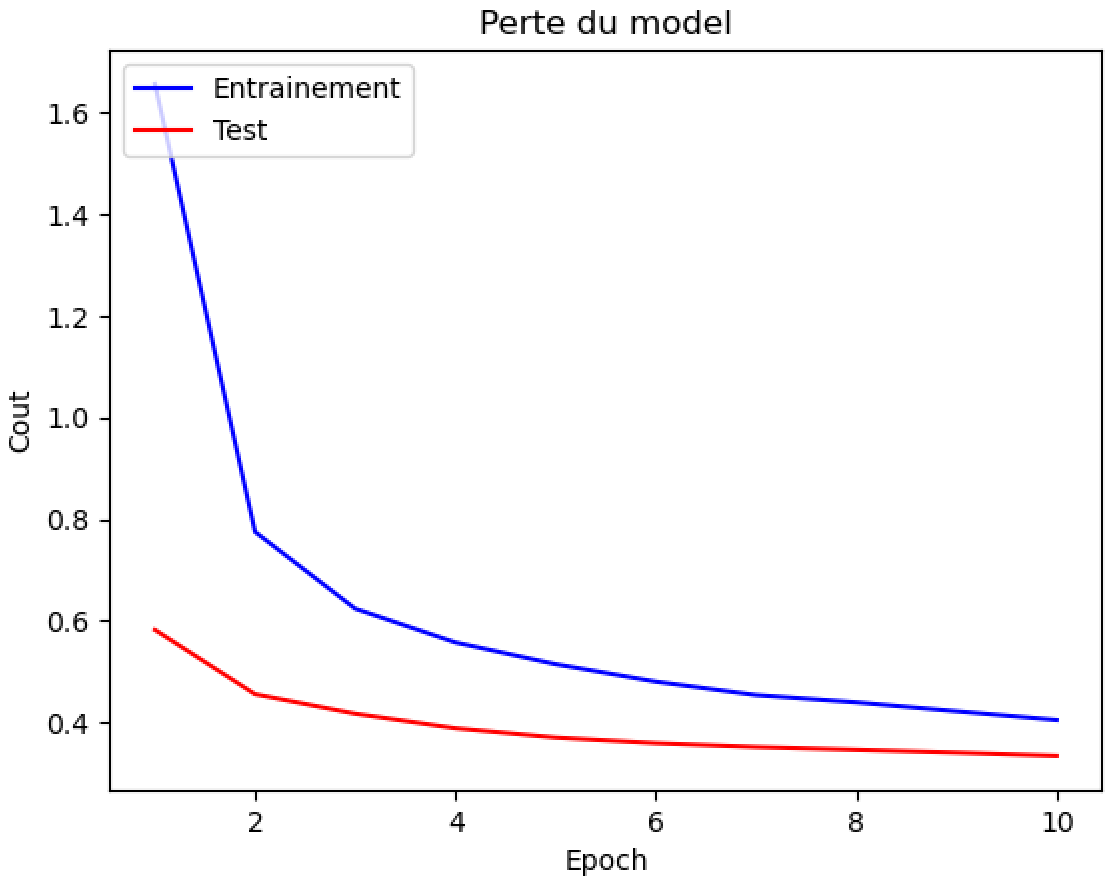
In [19]:
```
#Entrainement
history = model.fit(train_x, train_y, epochs=10, batch_size=512, verbose=1, \
                    validation_data=(val_x, val_y))
```

```
Epoch 1/10
199/199 [==============================] - 94s 470ms/step - loss: 1.6558 - accuracy: 0.5
322 - val_loss: 0.5825 - val_accuracy: 0.8100
Epoch 2/10
199/199 [==============================] - 94s 473ms/step - loss: 0.7751 - accuracy: 0.7
560 - val_loss: 0.4559 - val_accuracy: 0.8441
Epoch 3/10
199/199 [==============================] - 98s 494ms/step - loss: 0.6239 - accuracy: 0.7
983 - val_loss: 0.4175 - val_accuracy: 0.8515
Epoch 4/10
199/199 [==============================] - 98s 493ms/step - loss: 0.5576 - accuracy: 0.8
179 - val_loss: 0.3890 - val_accuracy: 0.8596
Epoch 5/10
199/199 [==============================] - 99s 499ms/step - loss: 0.5149 - accuracy: 0.8
298 - val_loss: 0.3707 - val_accuracy: 0.8675
Epoch 6/10
199/199 [==============================] - 99s 496ms/step - loss: 0.4806 - accuracy: 0.8
382 - val_loss: 0.3596 - val_accuracy: 0.8707
Epoch 7/10
199/199 [==============================] - 97s 489ms/step - loss: 0.4542 - accuracy: 0.8
453 - val_loss: 0.3521 - val_accuracy: 0.8719
Epoch 8/10
199/199 [==============================] - 90s 450ms/step - loss: 0.4399 - accuracy: 0.8
494 - val_loss: 0.3467 - val_accuracy: 0.8735
Epoch 9/10
199/199 [==============================] - 90s 450ms/step - loss: 0.4224 - accuracy: 0.8
539 - val_loss: 0.3411 - val_accuracy: 0.8783
Epoch 10/10
199/199 [==============================] - 93s 465ms/step - loss: 0.4052 - accuracy: 0.8
597 - val_loss: 0.3347 - val_accuracy: 0.8783
```

In [24]:
```
# plot de la precision et de la fonction cout
def plotgraph_acc(epochs, acc, val_acc):
    # Plot training & validation accuracy values
    plt.plot(epochs, acc, 'b')
    plt.plot(epochs, val_acc, 'r')
    plt.title('Précision du model')
    plt.ylabel('Précision')
    plt.xlabel('Epoch')
    plt.legend(['Entrainement', 'Test'], loc='upper left')
    plt.show()

def plotgraph_loss(epochs, acc, val_acc):
    plt.plot(epochs, acc, 'b')
    plt.plot(epochs, val_acc, 'r')
    plt.title('Perte du model')
    plt.ylabel('Cout')
    plt.xlabel('Epoch')
    plt.legend(['Entrainement', 'Test'], loc='upper left')
    plt.show()

acc = history.history['accuracy']
```

```
val_acc = history.history['val_accuracy']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1,len(acc)+1)


plotgraph_acc(epochs, acc, val_acc)

plotgraph_loss(epochs, loss, val_loss)

score = model.evaluate(test_x, test_y, verbose=0)
print("Test loss:", score[0])
print("Test accuracy:", score[1])
```






```
Test loss: 0.35347482562065125
Test accuracy: 0.8765891790390015
```

In [ ]: