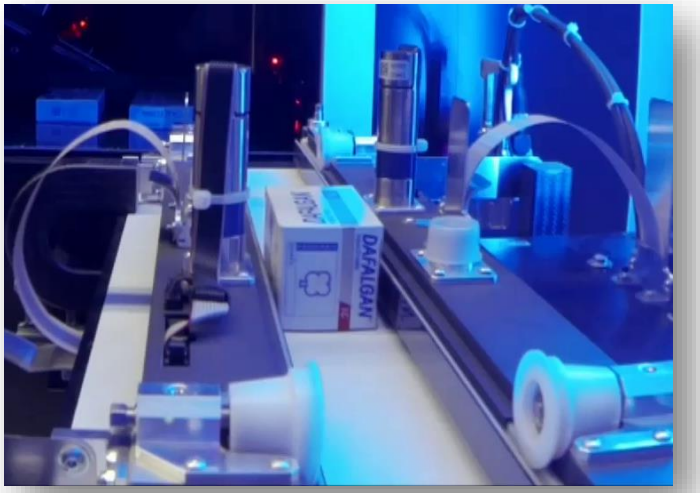
TIPE : RECONNAISSANCE DE BOÎTE DE MÉDICAMENTS AUTOMATISÉE PAR RÉSEAU DE NEURONES

TRABET Clément

N° 16888

MOTIVATION



PROBLÉMATIQUE

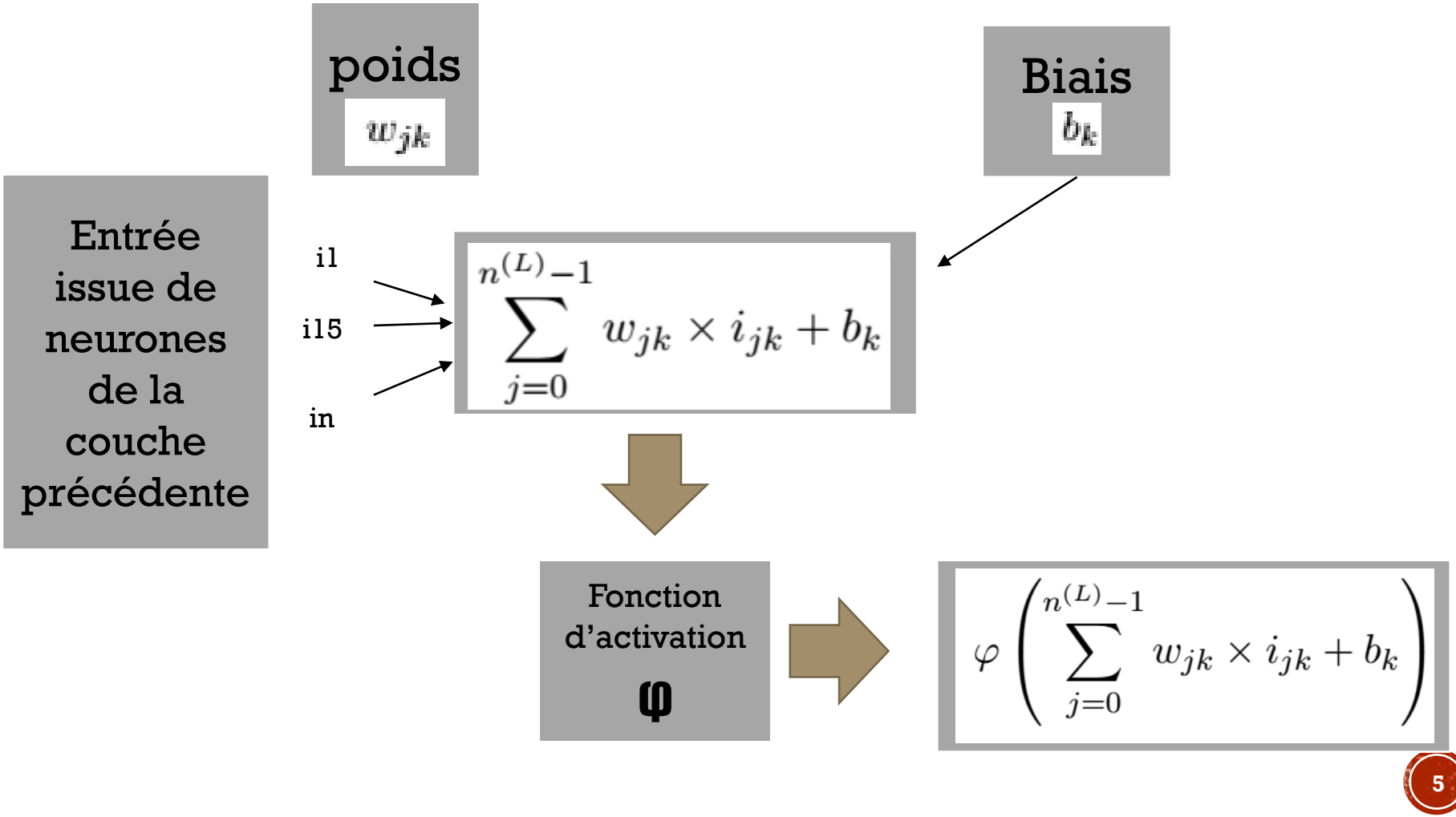
Algorithme → HUMEX

Tri des boites de médicament pour recyclage

Etudes fonctionnement réseau de neurones

SOMMAIRE

FONCTIONNEMENT D'UN NEURONE



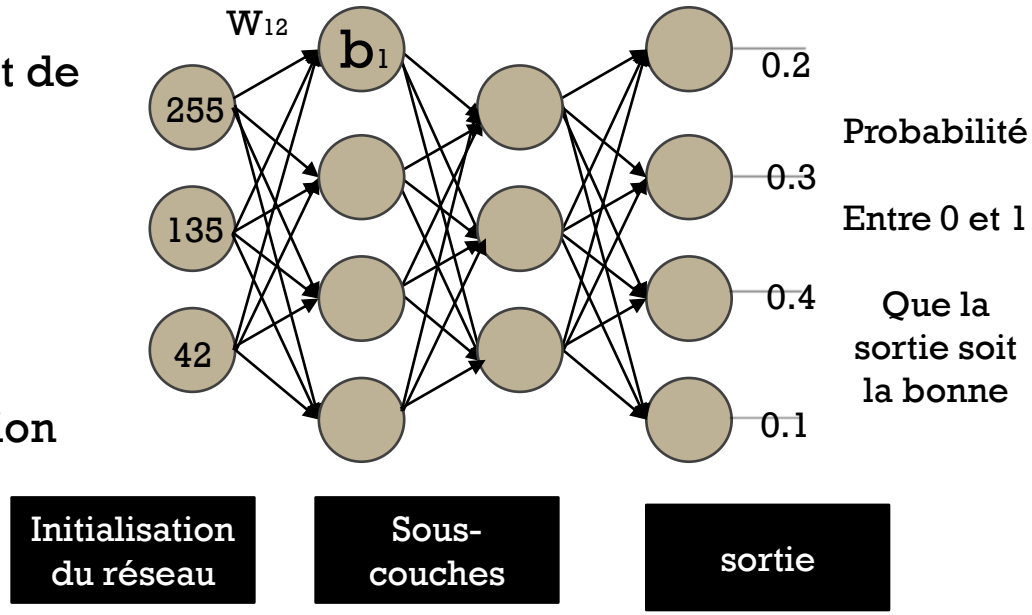
INITIALISATION DU RÉSEAU

Couche : regroupement de neurones en colonne (non connectés)

W_{jk} Poids : facteur multiplicatif qui permet de passer d'une couche à l'autre

b_k Biais : constante que l'on ajoute à la somme des entrées fois les poids

φ Fonction d'activation : tel que la fonction sigmoïde (non linéaire)



FONCTIONNEMENT DU RÉSEAU DE NEURONES

Objectif : renvoyer une probabilité que l'image corresponde à une lettre

Initialisation : Image de taille 28*28 reconnaissable à l'œil

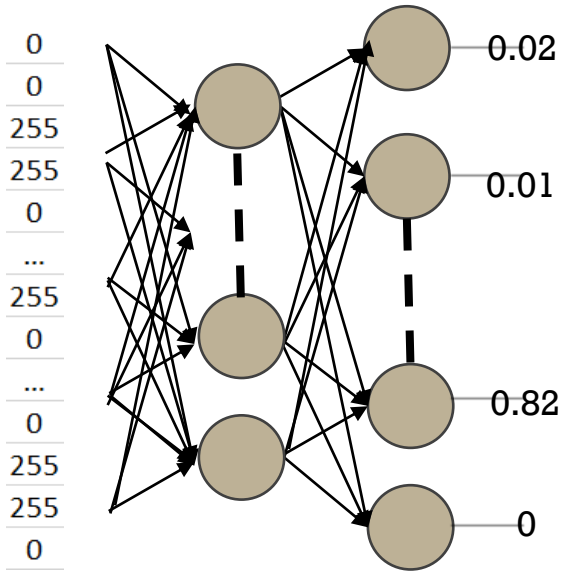
0	0	0	0	0	0	0	0	0
0	0	0	255	255	255	0	0	0
0	0	255	255	255	255	0	0	0
0	255	255	255	255	255	0	0	0
0	255	0	0	255	255	0	0	0
0	0	0	0	255	255	0	0	0
0	0	0	0	255	255	0	0	0
0	0	0	0	255	255	0	0	0
0	0	0	0	255	255	0	0	0
0	0	0	0	255	255	0	0	0
0	0	255	255	255	255	255	255	0
0	0	255	255	255	255	255	255	0
0	0	0	0	0	0	0	0	0

Matrice (28,28)

0
0
255
255
0
...
255
0
...
0
255
255
0

Vecteur (784,1)

RÉSEAU DE NEURONES UTILISÉ:



ASCII	valeur
48	0
49	1
121	y
122	z

Fonction coût

$$Cout = \sum_{j=0}^{n^{(L)}-1} (a_j^{(L)} - y_j)^2$$

Erreur quadratique moyenne

$$y_j = \delta_{j,k}$$

ENTRAINEMENT DU RÉSEAU DE NEURONES

Dataset : EMNIST
Stanford University

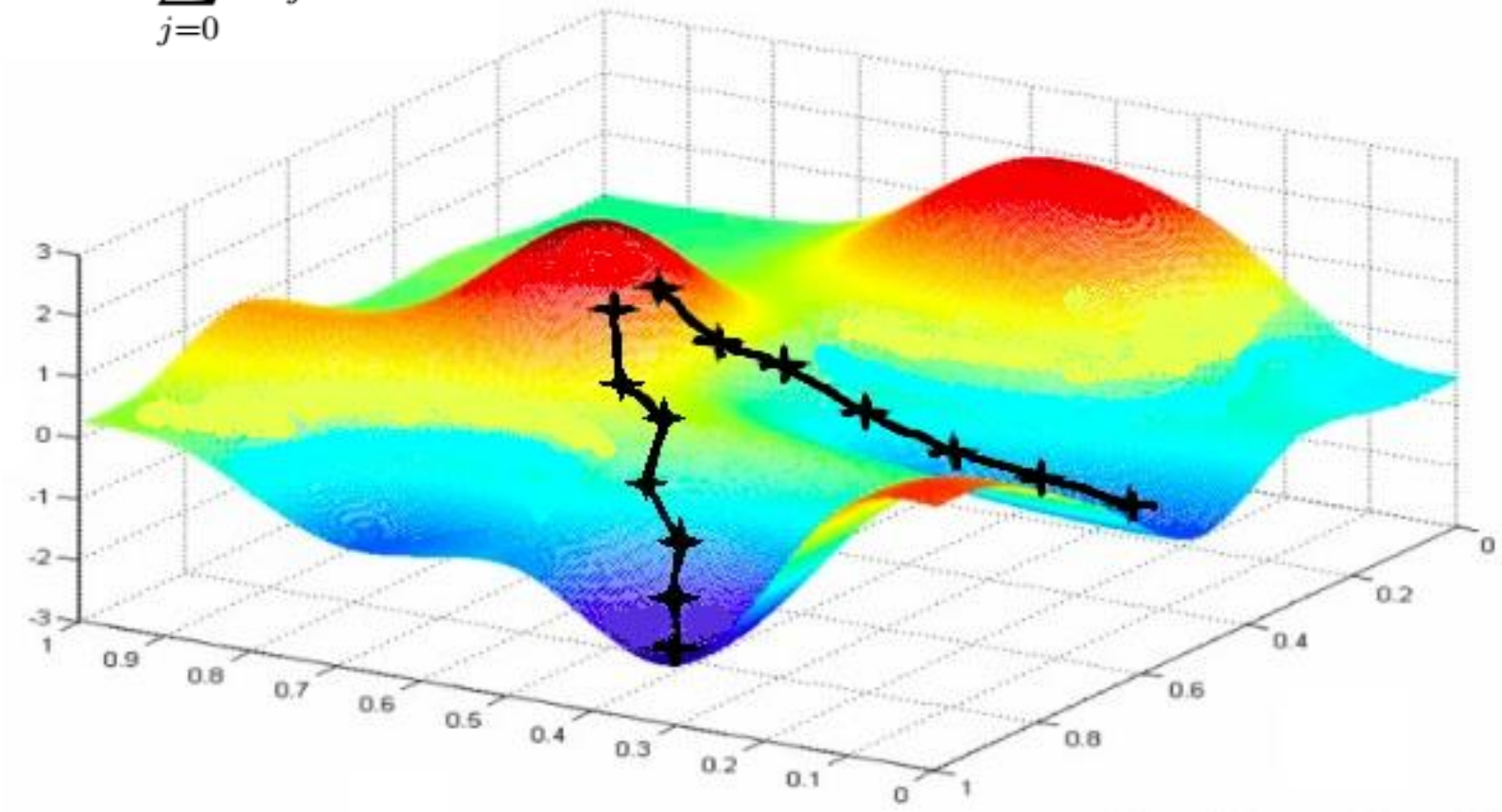
131 600 images

A-Z a-z 0-9
ASCII

Image 28*28
pixels

ALGORITHME DE RÉTRO PROPAGATION / GRADIENT

$$Cout = \sum_{j=0}^{n^{(L)}-1} (a_j^{(L)} - y_j)^2$$



biais

poids

ALGORITHME DE RÉTRO PROPAGATION / GRADIENT

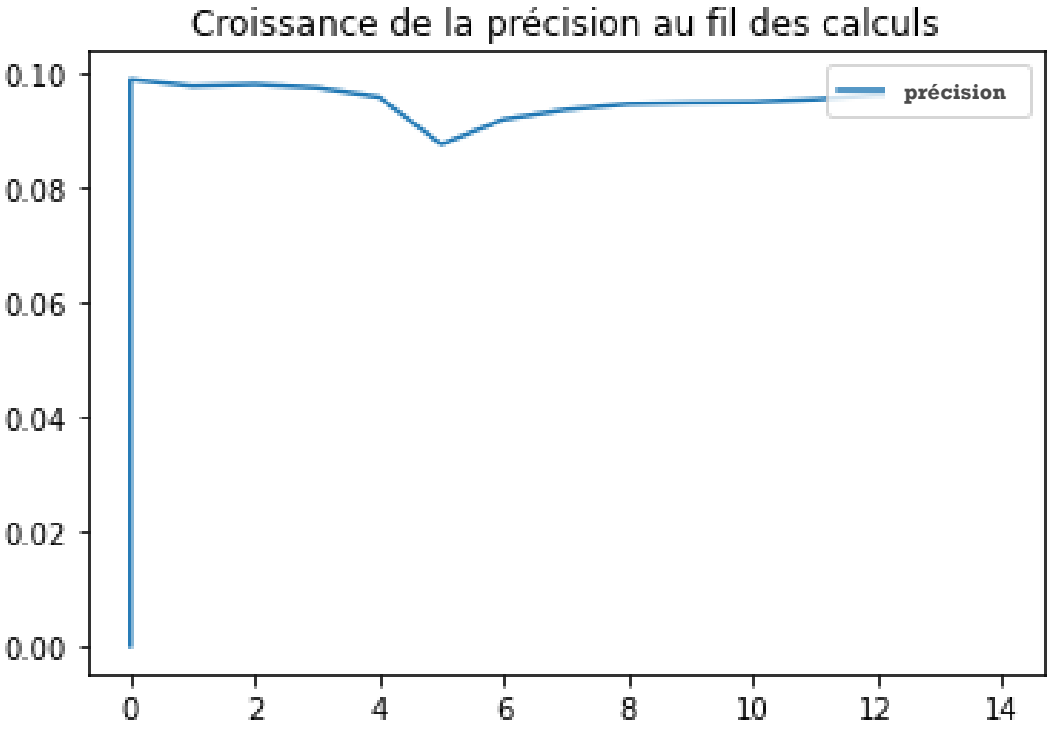
$$Cout = \sum_{j=0}^{n^{(L)}-1} (a_j^{(L)} - y_j)^2$$

Formules de récurrences :

$$w_{jk} = w_{jk} - \eta \times \frac{1}{n^{(L)}} \times \sum_{i=0}^{n^{(L)}-1} \frac{\partial C_i}{\partial w_{ik}^{(L)}}$$

$$b_k = b_k - \eta \times \frac{1}{n^{(L)}} \times \sum_{i=0}^{n^{(L)}-1} \frac{\partial C_i}{\partial b_k^{(L)}}$$

PREMIÈRE EXPÉRIENCE

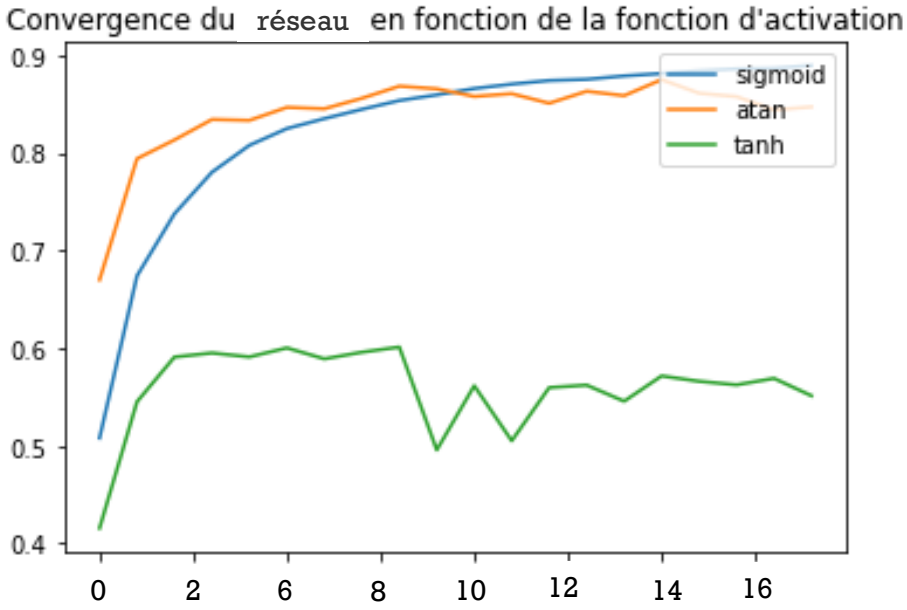


Moins bien que le hasard

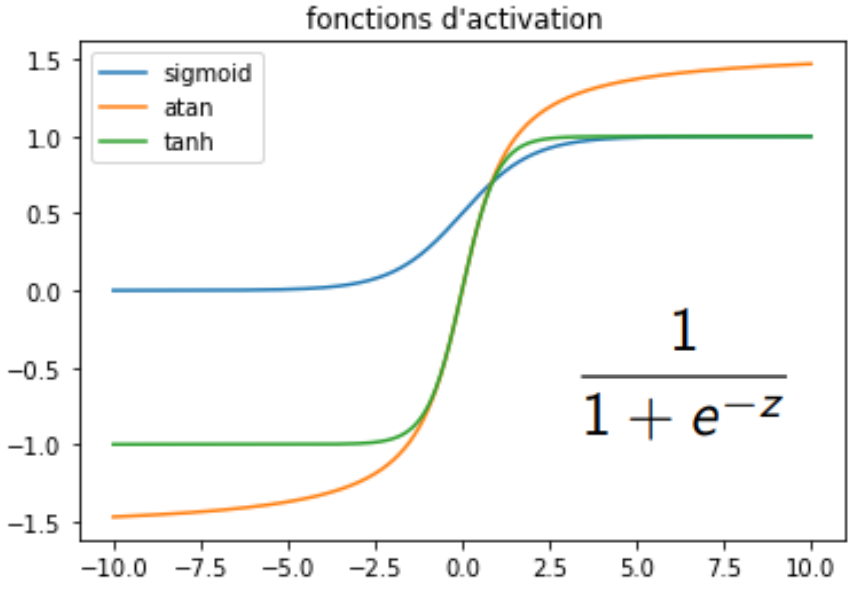
0-9

100 images test

PARAMÈTRE : FONCTION D'ACTIVATION

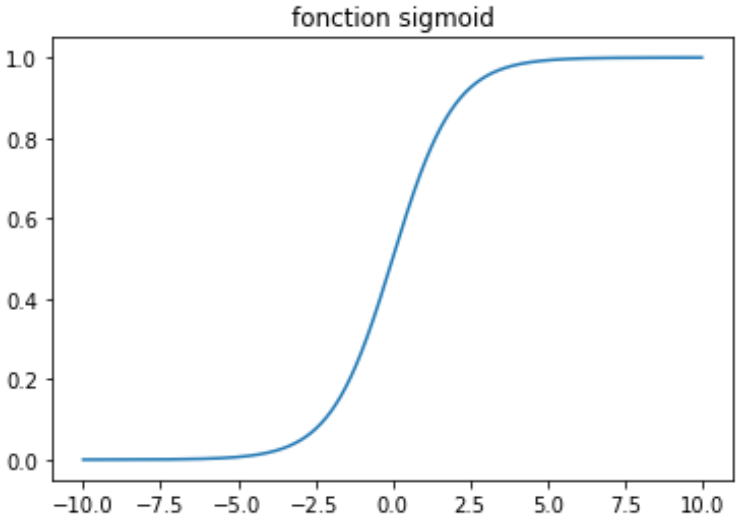


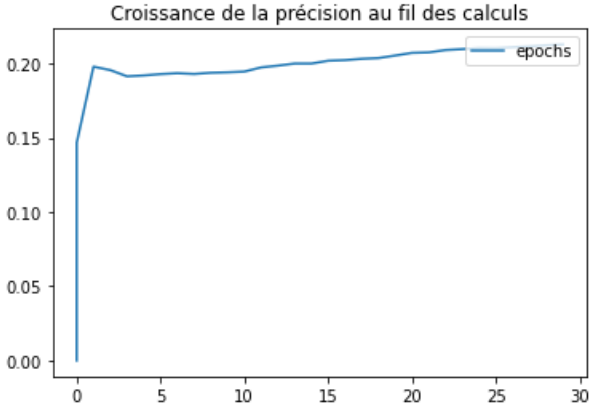
Sigmoïde plus stable, plus efficace

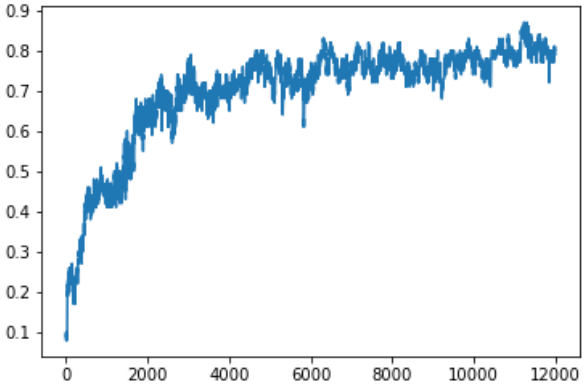


Dérivées & Initialisation

INFLUENCE DE L'INITIALISATION DES VALEURS



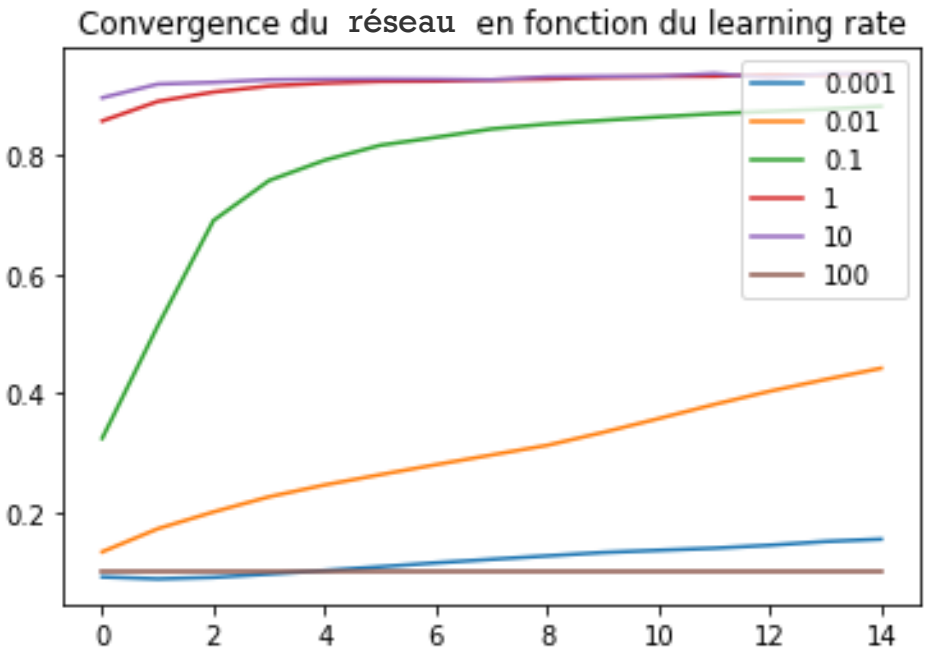




Précision atteinte de 20% pour des valeurs choisis aléatoirement entre [-10,10]

Valeurs non normalisées (valeurs entre [0,255]

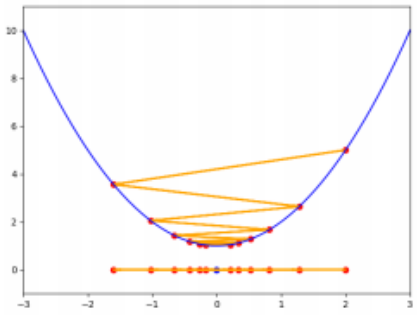
PARAMÈTRE : COEFFICIENT D'APPRENTISSAGE

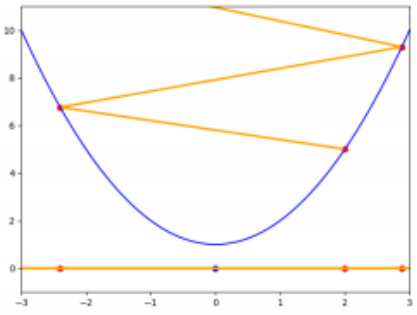


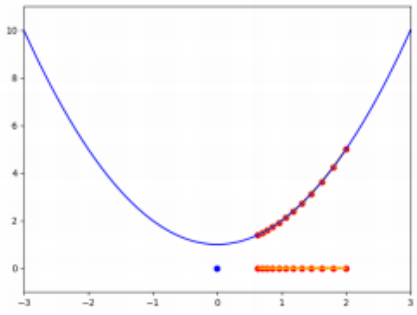
Formules de récurrences :

$$b_k = b_k - \eta \times \frac{1}{n^{(L)}} \times \sum_{j=0}^{n^{(L)}-1} \frac{\partial C_j}{\partial b_k^{(L)}}$$

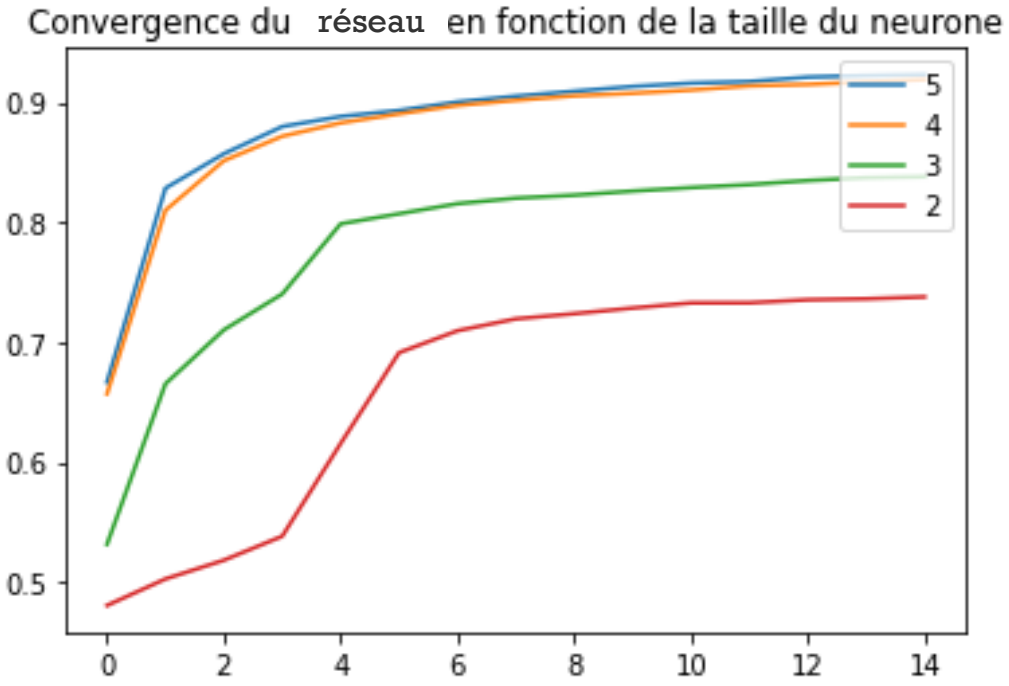
$$w_{jk} = w_{jk} - \eta \times \frac{1}{n^{(L)}} \times \sum_{j=0}^{n^{(L)}-1} \frac{\partial C_j}{\partial w_{jk}^{(L)}}$$



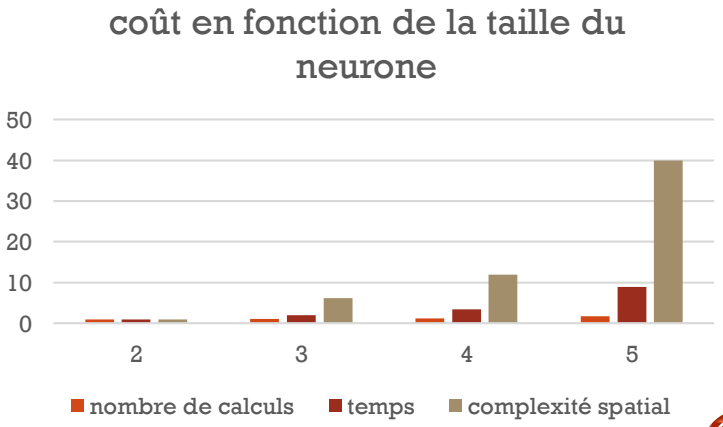




PARAMÈTRE TAILLE DU RÉSEAU DE NEURONES



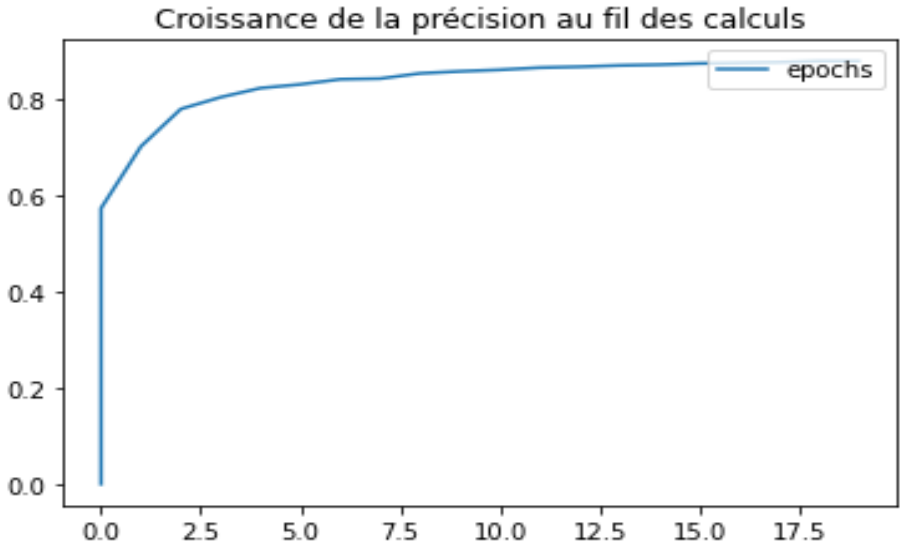
Taille	Nombre d'opérations	Temps	Espace
5	2,3 10^9	18min	1 Gb
4	1,6 10^9	7 min	300 Mb
3	1,4 10^9	4 min	154 Mb
2	1,3 10^9	2 min	25 Mb

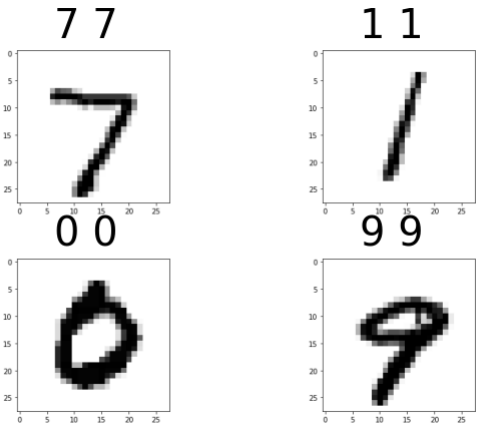


Coûts spatiaux et temporels

CONCLUSION DE L'ÉTUDE DE L'ALGORITHME

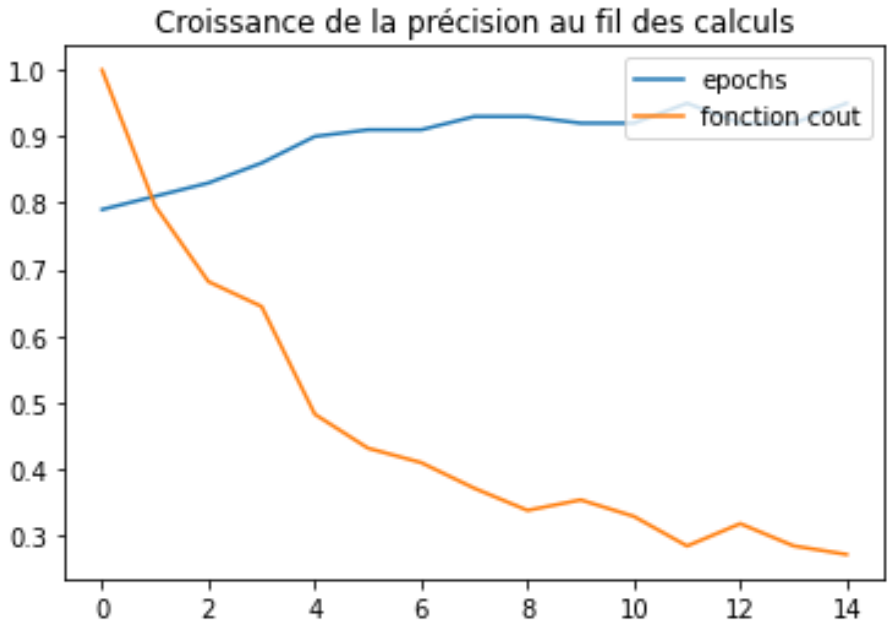
Paramètre	
Fonction d'activation	sigmoïde
Coefficient d'apprentissage	10
Taille du réseau	4 couches
Méthode stochastique	NON
Normalisation des vecteurs	OUI
Initialisation	[-1,1]

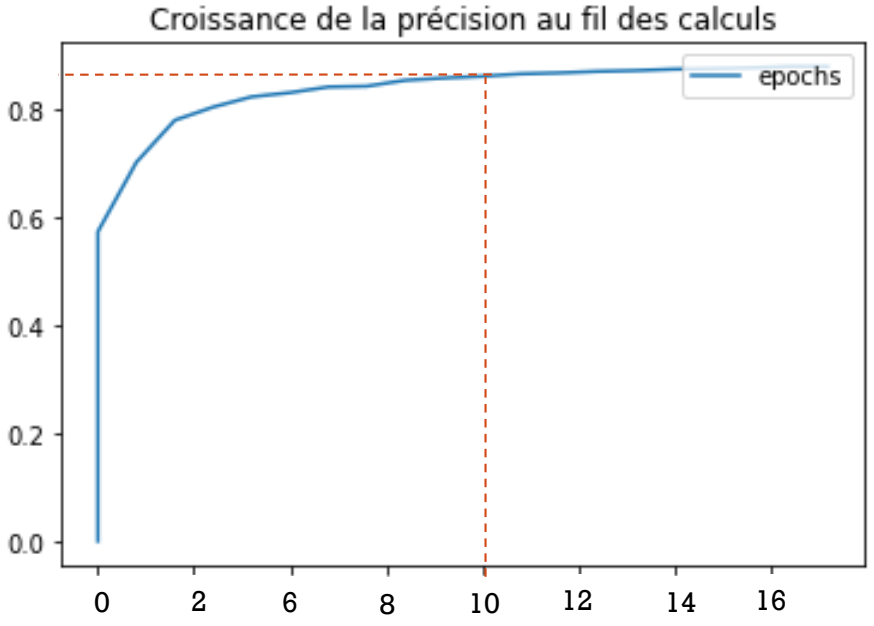




EXPÉRIENCES FINALES :

96,4 %





Critères validés :

- Obtenir une précision supérieure à 95%
- Parcourir moins de 10 fois les données

OBJECTIFS DU MCOT

2) Exploitation de l'algorithme:

Étude des paramètres du réseau de neurones :

-optimiser la vitesse de convergence du réseau 

-optimiser la précision du réseau 

COMMENT EXTRAIRE DES MOTS D'UNE IMAGE ?

RÉCUPÉRATION DES CONTOURS PAR MÉTHODE DE SOBEL

90	100	180	180	180
90	100	180	180	180
80	90	100	180	180
50	80	90	180	180
20	50	80	90	90

horizontale

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * A$$

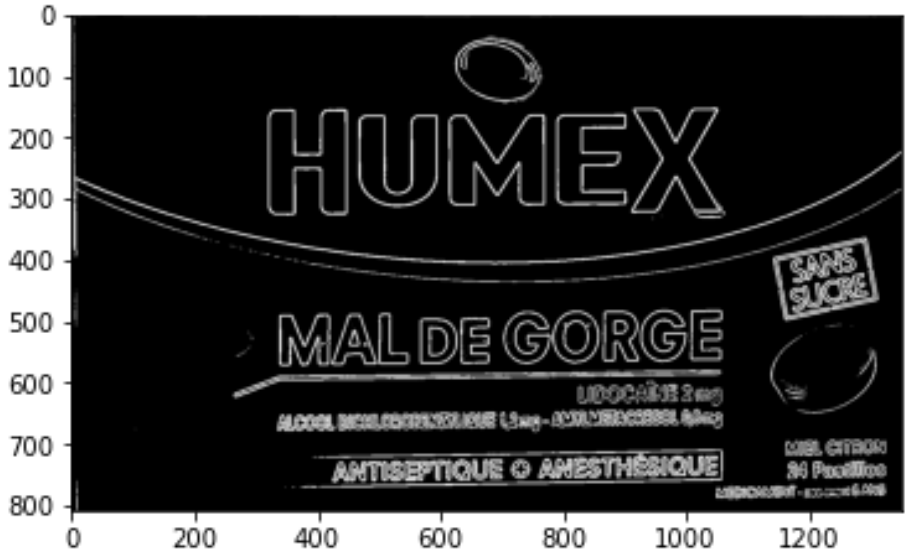
verticale

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * A$$

310	371	113
240	411	265
226	371	396

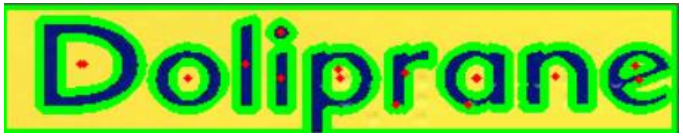
$$G = \sqrt{{G_x}^2 + {G_y}^2}$$

APPLICATION DU FILTRE DE SOBEL



Dessins, traits

PARCOURS DES CONTOURS



En utilisant une librairie

En utilisant aucune librairie

Barycentration

Entourage

Problème mots diacritiques

Récupération des contours (algorithme)

RÉSULTAT ALGORITHME :

Dollprane

HUMEX

MALDEGORGE O

ETAPE FINALE :

Comparaison à la BDPM (Base de Donnée Publique des Médicament)

Trouver le mot le plus proche par distance de Levenshtein

Dollprane

Doliprane

Distance 1

Algorithme réseau de neurones et extraction image

Algorithme Distance Levenshtein

OBJECTIFS MCOT :

Expérience :

A partir de photos de boîtes de médicament abîmées

-obtenir une précision de plus de 90%

OBJECTIFS DÉFINIS DANS LE MCOT:

1) Modélisation informatique :

Mise en œuvre d'un algorithme capable de reconnaître des lettres détériorées

- Obtenir une précision supérieure à 95 %
- Parcourir moins de 10 fois les données

Mise en œuvre d'un algorithme capable de reconnaître le nom d'un médicament

2) Exploitation de l'algorithme :

Étude des paramètres du réseau de neurones :

-optimiser la vitesse de convergence du réseau

-optimiser la précision du réseau

3) Expérience :

A partir de photos de boîtes de médicament abîmées

-obtenir une précision de plus de 90%

ANNEXE

Optimisation possible

- Evaluer la taille de la lettre
- Agrandir la zone rectangulaire 28*28
- Améliorer le dataset
- Compléter l'entrée du réseau avec une case « autre »

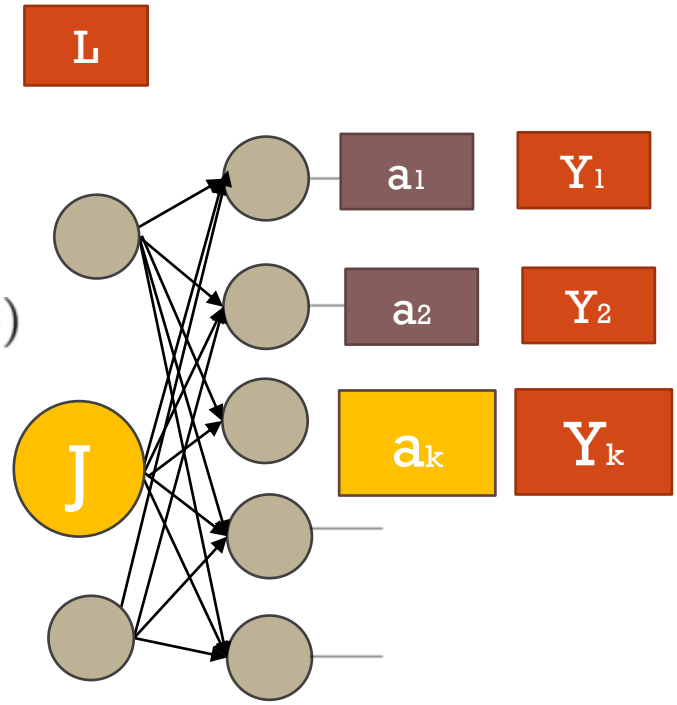
ALGORITHME DE RÉTROPROPAGATION/GRADIENT

$$\frac{\partial Cout}{\partial \omega_{jk}^{(L)}}$$

$$Cout = \sum_{j=0}^{n^{(L)}-1} (a_j^{(L)} - y_j)^2$$

$$a_k^{(L)} = \phi(Z_k^{(L)})$$

$$Z_k^{(L)} = \sum_{i=0}^{n^{(L)}-1} (\omega_{ik}^{(L)} \times a_i^{(L)-1}) + b_j^{(L)}$$



$\omega_{jk}^{(L)}$

$$\frac{\partial Cout}{\partial \omega_{jk}^{(L)}} = \frac{\partial Cout}{\partial a_k^{(L)}} \times \frac{\partial a_k^{(L)}}{\partial Z_k^{(L)}} \times \frac{\partial Z_k^{(L)}}{\partial \omega_{jk}^{(L)}}$$

Règle de la chaîne

ALGORITHME DE RÉTROPROPAGATION/GRADIENT

$$\frac{\partial Cout}{\partial \omega_{jk}^{(L)}} = \frac{\partial Cout}{\partial a_k^{(L)}} \times \frac{\partial a_k^{(L)}}{\partial Z_k^{(L)}} \times \frac{\partial Z_k^{(L)}}{\partial \omega_{jk}^{(L)}}$$

$$Cout = \sum_{j=0}^{n^{(L)}-1} (a_j^{(L)} - y_j)^2$$

$$\frac{\partial Cout}{\partial a_k^{(L)}} = 2 \times (a_k^{(L)} - y_k)$$

$$a_k^{(L)} = \phi(Z_k^{(L)})$$

$$\frac{\partial a_k^{(L)}}{\partial Z_k^{(L)}} = \phi'(Z_k^{(L)})$$

$$Z_k^{(L)} = \sum_{i=0}^{n^{(L)}-1} (\omega_{ik}^{(L)} \times a_i^{(L)-1}) + b_j^{(L)}$$

$$\frac{\partial Z_k^{(L)}}{\partial \omega_{jk}^{(L)}} = a_j^{(L-1)}$$

FORMULE DE RÉCURRENCE

Si (L) est la dernière couche:

$$\frac{\partial Cout}{\partial \omega_{jk}^{(L)}} = 2 \times (a_k^{(L)} - y_k) \times \phi'(Z_k^{(L)}) \times a_j^{(L-1)}$$

Sinon :

$$\frac{\partial Cout}{\partial \omega_{jk}^{(I)}} = \frac{\partial Cout}{\partial a_k^{(I_{+1})}} \frac{\partial a_k^{(I_{+1})}}{\partial \omega_{jk}^{(L)}} = \frac{\partial Cout}{\partial a_k^{(I_{+1})}} \phi'(Z_k^{(I)}) \times a_j^{(I-1)}$$

CODE DESCENTE DE GRADIENT

```
101
102     def descente_de_gradient(self, training_data, epochs, taille_mini_batch, eta, test_data,f,df):
103         """ descente de gradient sur les minibatch """
104         X_epoch=[0]  #initialisation des listes/compteur pour afficher la convergence
105         Y=[0]
106         Xcout=[]
107         Yc=[]
108         Ycout=[]
109         i=0
110         plt.clf()
111         n = len(training_data)
112         nd= len(test_data)
113         for j in range(epochs):
114             random.shuffle(training_data)  #on mélange les batchs
115             mini_batches = [training_data[k:k+taille_mini_batch] for k in range(0, n, taille_mini_batch)]
116             for mini_batch in mini_batches:
117                 i=i+1
118                 Xcout.append(i)
119                 #Ycout.append(self.cost(test_data,f)/nd)
120                 #Yc.append( self.eval(test_data,f) /len(test_data))
121                 self.update_mini_batch(mini_batch, eta,f,df)
122             X_epoch.append(j)
123             Y.append( self.eval(test_data,f) /len(test_data))
124
125         plt.title("Croissance de la précision au fil des calculs")
126         plt.plot(X_epoch,Y,label="epochs")
127         #plt.plot(Xcout,self.normalise_lst(Ycout),label="fonction cout")
128         #plt.plot(Xcout,Yc,label="batch")
129         plt.legend(loc=1)
130         print(max(Y))
131         plt.show()
```

```
33      def calc_forward(self, v,f):
34          """calcul la sortie du neurone pour un vecteur donné v """
35          for b, w in zip(self.biais, self.poids):
36              v = f(np.dot(w, v)+b)
37          return v
38
39      def update_mini_batch(self, mini_batch, eta,f,df):
40          """calcul des derivé partielles des biais et des poids"""
41          deriv_b = [np.zeros(b.shape) for b in self.biais]
42          deriv_w = [np.zeros(w.shape) for w in self.poids]
43          for x, y in mini_batch:
44              lst_deriv_b, lst_deriv_w = self.calc_backward(x, y,f,df)
45              deriv_b = [nb+dnb for nb, dnb in zip(deriv_b, lst_deriv_b)]
46              deriv_w = [nw+dnw for nw, dnw in zip(deriv_w, lst_deriv_w)]
47          self.poids = [w-(eta/len(mini_batch))*nw for w, nw in zip(self.poids, deriv_w)]
48          self.biais = [b-(eta/len(mini_batch))*nb for b, nb in zip(self.biais, deriv_b)]
49
50      def calc_backward(self, x, y,f,df):
51          """backward propagation """
52          deriv_b = [np.zeros(b.shape) for b in self.biais]
53          deriv_w = [np.zeros(w.shape) for w in self.poids]
54
55          activation = x
56          activations = [x]
57          zs = []
58          for b, w in zip(self.biais, self.poids):
59              z = np.dot(w, activation)+b
60              zs.append(z)
61              activation = sigmoid(z)
62              activations.append(activation)
63
64          delta = self.cost_derivative(activations[-1], y) *  df(zs[-1])
65          deriv_b[-1] = delta
66          deriv_w[-1] = np.dot(delta, activations[-2].transpose())
67
68          for l in range(2, self.nb_couches):
69              z = zs[-l]
70              delta = np.dot(self.poids[-l+1].transpose(), delta) * df(z)
71              deriv_b[-l] = delta
72              deriv_w[-l] = np.dot(delta, activations[-l-1].transpose())
73          return (deriv_b, deriv_w)
```

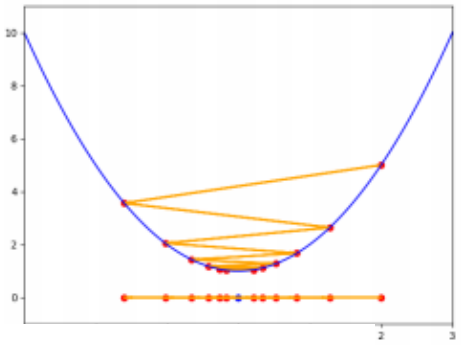
IMPLÉMENTATION FONCTION COÛT :

```
86      def cost(self,test_data,f):
87          s=0
88          for x,y in test_data:
89              a=self.calc_forward(x,f)
90              for j in range(len(a)):
91                  if j==y:
92                      s=s+(a[j]-1)**2
93                  else:
94                      s=s+(a[j])**2
95          return s
```

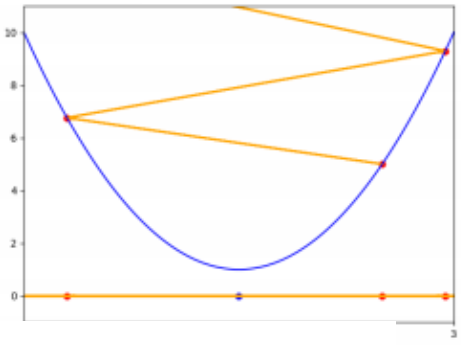
$$Cout = \sum_{j=0}^{n_{L-1}} (a_j^{(L)} - y_j)^2$$

COMMENT AMÉLIORER LES PERFORMANCES ?

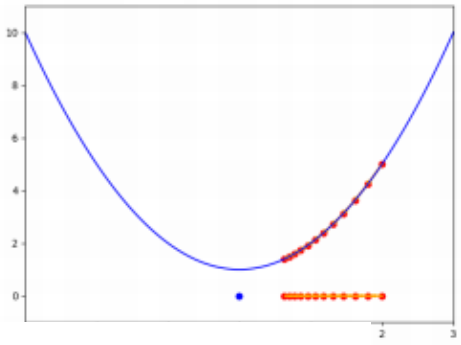
$\alpha = 0.9$



$\alpha = 1.1$

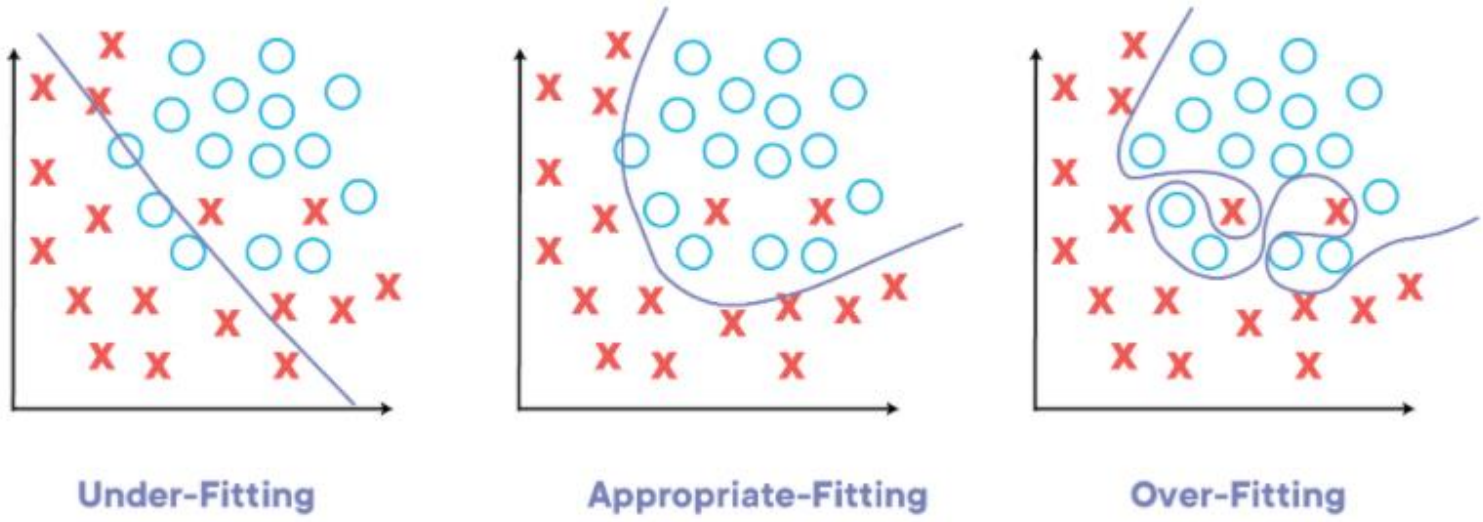


$\alpha = 0.05$



α	α	$\alpha_k = \frac{\alpha}{k}$	$\alpha_k = \frac{\alpha}{(k)^2}$	Divisions par 2 toutes les 200 itérations
0,1	Pas de convergence	9 itérations	Pas de convergence	3000 itérations
0,042	Pas de convergence	7 itérations	Pas de convergence	+10000 itérations
0,035	114 itérations	5 itérations	Pas de convergence	75 itérations

OVER-FITTING ET UNDER-FITTING



DISTANCE ENTRE LES MOTS / LEVENSHTEIN

Banque médicament										Distance de Levenshtein
Mot départ	D	O	L	L	P	R	A	N	E	
Mot 1	D	O	L	I	P	R	A	N	E	1
Mot 2	H	U	M	E	X					5
Mot 3	D	O	L	I	R	H	U	M	E	5

Lettre différentes

Ajout de lettre

Suppression de lettre

```
448  def needlmann_wunsch(m1,m2):
449      n=len(m1)
450      p=len(m2)
451      mat=[[0 for i in range(p+1)] for j in range(n+1)]
452      for i in range(n+1):
453          mat[i][0]=i
454      for j in range(p+1):
455          mat[0][j]=j
456      for i in range(1,n+1):
457          for j in range(1,p+1):
458              if m1[i-1]==m2[j-1]:
459                  mat[i][j]=mat[i-1][j-1]
460              else:
461                  mat[i][j]=min(mat[i-1][j], mat[i][j-1])+ 1
462      return mat[n][p]
```

If (lettre_mot_A=lettre_mot_B):
Tab[i][j]<- Tab[i-1,j-1]
Else:
Tab[i,j] <-Min(Tab[i,j-1], Tab[i-1,j]) +1

```
464  def plus_proche(m):
465      with open('data_propre.txt','r') as file:
466          mot=[]
467          ecart=len(m)
468          i=0
469          donne=file.readlines()
470          for mot_test in donne:
471              i=i+1
472              n=needlmann_wunsch(m, mot_test)
473              if n<ecart:
474                  ecart=n
475                  mot=[mot_test]
476              elif n==ecart:
477                  mot.append(mot_test)
478      return (mot,ecart)
```

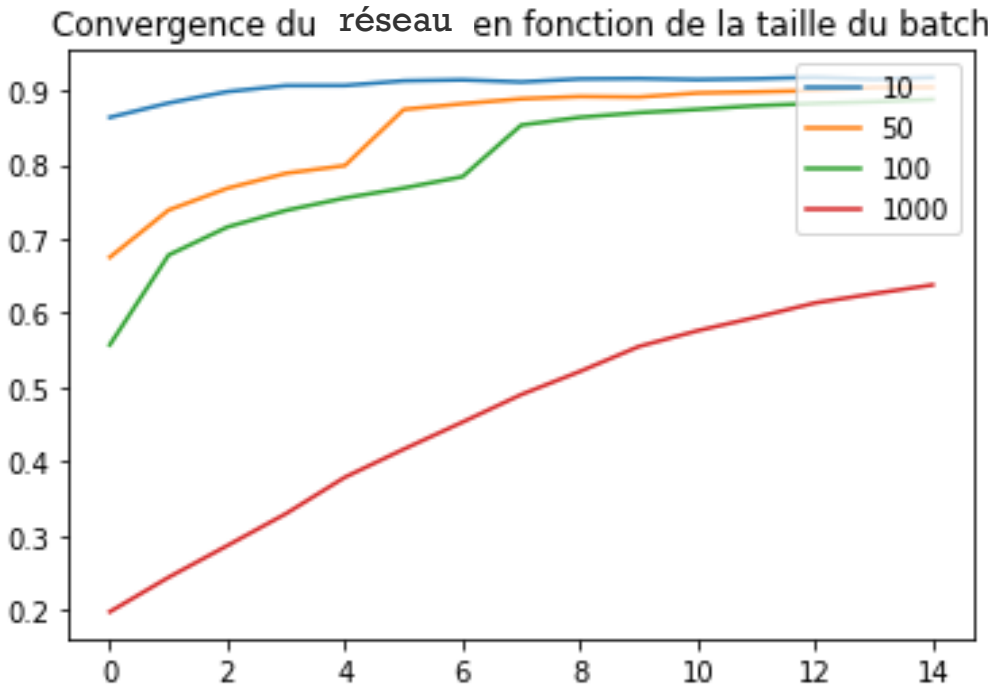
		G	C	A
	0	1	2	3
G	1	0	1	2
A	2	1	2	1
A	3	2	3	2

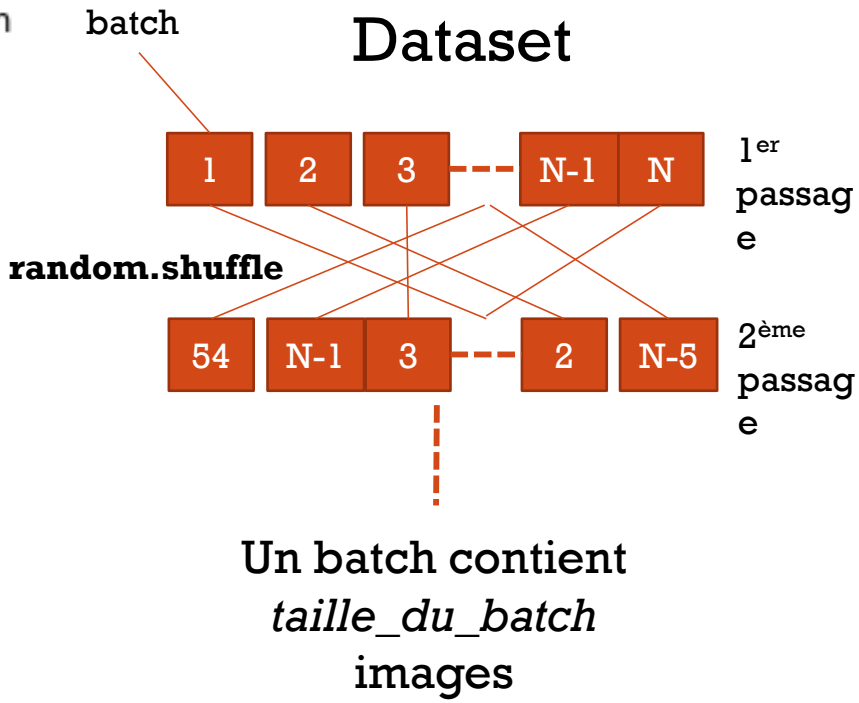
Encore plus d'éfficacité

IMPLÉMENTATION DU NEURONE

```
26  class Network(object):
27          def __init__(self, lst):
28              self.nb_couches = len(lst)
29              self.list_taille = lst
30              self.biais = [np.random.randn(y, 1) for y in lst[1:]]
31              self.poids = [np.random.randn(y, x) for x, y in zip(lst[:-1], lst[1:])]
```

UTILISATION DE LA MÉTHODE STOCHASTIQUE





Peu efficace dans ce cas

```
328  def filtre_de_Sobel(image,thresh=220):
329      n,p=np.shape(image)
330      new_image=[[[0,0,0] for i in range(p)] for j in range(n)]
331      x=np.array([[-1, 0, 1],
332                  [-2, 0, 2],
333                  [-1, 0, 1]])
334      y=np.array([[-1, -2, -1],
335                  [0, 0, 0],
336                  [1, 2, 1]])
337
338      for i in range (1,n):
339          for j in range(1,p):
340
341              im=np.array(image[i-1:i+2,j-1:j+2])
342              if (np.shape(im)==(3,3)):
343                  gx=x*im
344                  gy=y*im
345                  s=np.sum(gx)**2+np.sum(gy)**2
346                  s=np.sqrt(s)
347                  if s>=thresh:
348                      new_image[i][j]=[255,255,255]
349
350      plt.imshow(new_image)
351      plt.show()
```

```
def nom_boite(image):
    #image=cv2.resize ( image ,(800 ,300), interpolation= cv2.INTER_LINEAR )
    imgray_1 = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    #blur = cv2.GaussianBlur(imgray_1,(1,5),0)
    ret, thresh = cv2.threshold(imgray_1,0 , 255, 0)
    contours, hierarchy = cv2.findContours(thresh, cv2.RETR_TREE, cv2.CHAIN_APPROX_NONE)

    centres=[]
    mot=[]

    for cnt in contours:

        x,y=tr_centre(cnt)
        centres.append((x,y))

    print(centres[0])
    print(contours[0][0])
    print(contours[0][0][0][0])
    #for i in range (len(centres)):
    for i in range(len(contours)):
        if len(contours[i])>150:
            if distance(centres[i],centres,contours,len(contours[i])):

                min_x,min_y,max_x,max_y=carre_lettre(contours[i])

                if max_y-min_y >10 and max_x-min_x>10 and max_y-min_y<50 and max_x-min_x<40:
                    cv2.drawContours(image, contours[i], -1,(0,2550,0),3)
                    cv2.drawContours(imgray_1, contours[i], -1,(0,255,0),3)
                    cv2.circle(image, centres[i], radius=2, color=(0, 0, 255), thickness=-1)
                    cv2.rectangle(image, (min_x,min_y),(max_x,max_y) , color=(0,0,255), thickne
                    img_extract=image[min_y : max_y , min_x : max_x]
                    cv2.imshow('extrait',img_extract)
                    lettre=Net.calc(img_extract)
                    mot.append(lettre)
                cv2.waitKey(0)
    cv2.imshow('Image', image)
    cv2.imshow('Image GRAY', imgray_1)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

```
""" Partie mise en forme des données """

#loading
(train_X, train_y), (test_X, test_y) = mnist.load_data()


def resultat_form(result):
    """ met sous la forme (0,0,0,1,0,0,0,0,0,0) """

    new_res=[]
    i=0
    for y in result:
        Y=np.array([[0],[0],[0],[0],[0],[0],[0],[0],[0],[0]],float)
        Y[y][0]=1
        new_res.append(Y)
        i=i+1
    return new_res

#########################
def to_28_28 ( image ) :
    new = cv2 . resize ( image ,(28 ,28) )
    return new

def to_vector ( image ) :
    x = cv2 . cvtColor ( to_28_28(image) , cv2 . COLOR_BGR2GRAY )
    return np . reshape (1.0 - x /255 , (784 , 1) )

#########################

def convertion_image(entreeM):
    """ met la matrice sous la forme d'un vecteur de taille (784,1)"""
    new_entree=[]
    for x in entreeM:
        X=[np.concatenate([np.diagonal(x[::-1,:], k)[::(2*(k % 2)-1)] for k in range(1-x.shape[0], x.shape[0])])]
        new_entree.append(np.transpose(X))
    return new_entree

def concat(X,Y):
    tab=[]
    for i in range(len(X)):
        x=X[i]
        y=Y[i]
        tab.append((x,y))
    return tab

tab_apprend=concat(convertion_image(train_X[:50000]), resultat_form(train_y[:50000]))
tab_test=concat(convertion_image(test_X[:100]), resultat_form(test_y[:100]))

""" tab est une liste de tuples tq (matrice, resultat sous forme tablau avec 1 et 0) """
```

```
""" Partie Neurone """
#import emnist as emn
training_data,test_data=load()

class Network(object):
        def __init__(self, lst):
            self.nb_couches = len(lst)
            self.list_taille = lst
            self.biais = [np.random.randn(y, 1) for y in lst[1:]]
            self.poids = [np.random.randn(y, x) for x, y in zip(lst[:-1], lst[1:])]

        def calc_forward(self, v,f):
            """calcul la sortie du neurone pour un vecteur donné v """
            for b, w in zip(self.biais, self.poids):
                v = f(np.dot(w, v)+b)
            return v

        def update_mini_batch(self, mini_batch, eta,f,df):
            """calcul des derivé partielles des biais et des poids"""
            deriv_b = [np.zeros(b.shape) for b in self.biais]
            deriv_w = [np.zeros(w.shape) for w in self.poids]
            for x, y in mini_batch:
                lst_deriv_b, lst_deriv_w = self.calc_backward(x, y,f,df)
                deriv_b = [nb+dnb for nb, dnb in zip(deriv_b, lst_deriv_b)]
                deriv_w = [nw+dnw for nw, dnw in zip(deriv_w, lst_deriv_w)]
            self.poids = [w-(eta/len(mini_batch))*nw for w, nw in zip(self.poids, deriv_w)]
            self.biais = [b-(eta/len(mini_batch))*nb for b, nb in zip(self.biais, deriv_b)]

        def calc_backward(self, x, y,f,df):
            """backward propagation """
            deriv_b = [np.zeros(b.shape) for b in self.biais]
            deriv_w = [np.zeros(w.shape) for w in self.poids]

            activation = x
            activations = [x]
            zs = []
            for b, w in zip(self.biais, self.poids):
                z = np.dot(w, activation)+b
                zs.append(z)
                activation = sigmoid(z)
                activations.append(activation)

            delta = self.cost_derivative(activations[-1], y) *  df(zs[-1])
            deriv_b[-1] = delta
            deriv_w[-1] = np.dot(delta, activations[-2].transpose())

            for l in range(2, self.nb_couches):
                z = zs[-l]
                delta = np.dot(self.poids[-l+1].transpose(), delta) * df(z)
                deriv_b[-l] = delta
                deriv_w[-l] = np.dot(delta, activations[-l-1].transpose())
            return (deriv_b, deriv_w)
```

```
    def eval(self, test_data,f):
        """evalue la convergence du neurone test sur le dataset le nombre de réponses correct"""
        test_results = [(np.argmax(self.calc_forward(x,f)), y)
                        for (x, y) in test_data]
        return sum(int(x == y) for (x, y) in test_results)

    def cost_derivative(self, output_activations, y):
        """dérivé de la fonction cout"""
        return (output_activations-y)

    def cost(self,test_data,f):
        s=0
        for x,y in test_data:
            a=self.calc_forward(x,f)
            for j in range(len(a)):
                if j==y:
                    s=s+(a[j]-1)**2
                else:
                    s=s+(a[j])**2
        return s

    def calc(self, vect,f):
        """calcul la sortie troiver pour un vecteur"""
        return np.argmax(self.calc_forward(vect,f))


    def descente_de_gradient(self, training_data, epochs, taille_mini_batch, eta, test_data,f,df):
        """ descente de gradient sur les lots i.e batchs """
        X_epoch=[0]  #initialisation des listes/compteur pour afficher la convergence
        Y=[0]
        Xcout=[]
        Yc=[]
        Ycout=[]
        i=0
        plt.clf()
        n = len(training_data)
        nd= len(test_data)
        for j in range(epochs):
            random.shuffle(training_data)  #on mélange les batchs
            mini_batches = [training_data[k:k+taille_mini_batch] for k in range(0, n, taille_mini_batch)]
            for mini_batch in mini_batches:
                i=i+1
                Xcout.append(i)
                #Ycout.append(self.cost(test_data,f)/nd)
                #Yc.append( self.eval(test_data,f) /len(test_data))
                self.update_mini_batch(mini_batch, eta,f,df)
            X_epoch.append(j)
            Y.append( self.eval(test_data,f) /len(test_data))

        plt.title("Croissance de la précision au fil des calculs")
        plt.plot(X_epoch,Y,label="epochs")
        #plt.plot(Xcout,self.normalise_lst(Ycout),label="fonction cout")
        #plt.plot(Xcout,Yc,label="batch")
        plt.legend(loc=1)
        print(max(Y))
        plt.show()
```

```
def sigmoid(z):
    return 1.0/(1.0+np.exp(-z))

def dsigmoid(z):
    return sigmoid(z)*(1-sigmoid(z))

#### fonction traitement d'image

def to_28_28(image):
    down_width = 28
    down_height = 28
    down_points = (down_width, down_height)
    resized_down = cv2.resize(image, down_points, interpolation= cv2.INTER_LINEAR)
    #cv2.imshow('Resized Down by defining height and width', resized_down)
    #cv2.waitKey()
    #cv2.destroyAllWindows()
    return resized_down

def to_vector ( image ) :
    x = cv2 . cvtColor ( to_28_28(image) , cv2 . COLOR_BGR2GRAY )
    return 255*np . reshape (1.0 - x /255 , (784 , 1) )

def image_test():
    lst=['tt7.png','tt1.png','tt0.png','tt9.png']
    n_lst=[0,2,3,9]
    fig=plt.figure(figsize=(10,7))
    rows = 2
    columns = 2
    plt.axis('off')
    for i in range (1,5):

        image=img.imread(lst[i-1])
        fig.add_subplot(rows,columns,i)
        plt.imshow(image,cmap='Greys')
        plt.axis('off')
        #vect=to_vector(cv2.imread(lst[i-1]))
        y=Net.calc(test_data[n_lst[i-1]][0],sigmoid)
        y_2=test_data[n_lst[i-1]][1]

        plt.title(str(y)+' '+str(y_2),fontsize="40")
    plt.axis('off')
    plt.show()



def image(img,net):
    imgray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    img=cv2.resize ( imgray ,(28 ,28), interpolation= cv2.INTER_LINEAR )
    v=np . reshape (1.0 - img/255 , (784 , 1) )
    return Net.calc(v,sigmoid)
```

```
#### fonction test sur neurones

Net=Network([784,50,10])

Net.descente_de_gradient(training_data, 15, 100, 2, test_data,sigmoid,dsigmoid)

print("le neurone à appris")

""" Fin partie Neurone """

""" Début partie analyse image """


def voisins(i,j,n,p):
    if i==0:
        if j==0:
            return [(i,j+1),(i+1,j),(i+1,j+1)]
        if j==(p-1):
            return [(i,j-1),(i+1,j),(i-1,j-1)]
        return [(i,j-1),(i,j+1),(i+1,j+1),(i+1,j-1),(i+1,j)]
    if i==n-1:
        if j==0:
            return [(i,j+1),(i-1,j),(i-1,j+1)]
        if j==(p-1):
            return [(i-1,j-1),(i-1,j),(i,j-1)]
        return [(i-1,j-1),(i-1,j),(i-1,j+1),(i,j-1),(i,j+1)]
    if j==p-1:
        return [(i-1,j-1),(i-1,j),(i,j-1),(i+1,j-1),(i+1,j)]
    if j==0:
        return [(i-1,j+1),(i,j+1),(i+1,j+1),(i+1,j),(i-1,j)]
    return [(i-1,j-1),(i-1,j),(i-1,j+1),(i,j-1),(i,j+1),(i+1,j+1),(i+1,j-1),(i+1,j)]
```

```
#récupération contour de l'image

def contour_sobel(image):
    lst_contours=[]
    n,p=len(image[0]),len(image)
    mat=np.zeros((n+1,p+1))
    for i in range(n):
        mat[i][0]=-1
        mat[i][p-1]=-1
    for j in range(p):
        mat[0][j]=-1
        mat[n-1][j]=-1
    c=1
    def aux(i,j,n):
        if (mat[i][j]==0 and image[i][j]>67):
            mat[i][j]=n
            for (k,l) in voisins(i,j,n,p):
                aux(k,l,n)

    for i in range (n):
        for j in range (p-1):
            l=aux(i,j,c)
            c=c+1
    l=[[] for i in range (c) ]
    for i in range(n-1):
        for j in range(p):
            n=mat[i][j]
            if n != 0 :
                l[int(n)].append((i,j))

    for x in l:
        if len(x)>0:
            lst_contours.append(x)
    return lst_contours

```

```python
def entourage(h : list, i : int, j : int):
    #renvoie l'entourage les 8 voisins de l[i][j]
    voisins = []
    coordonnees = [(i-1, j-1), (i, j-1), (i+1, j-1), (i+1, j), (i+1, j+1), (i, j+1), (i-1, j+1), (i-1, j)]
    for k,l in coordonnees:
        voisins.append(h[k][l])
    return voisins

def tr_centre(tab):
    X=0
    Y=0
    for t in tab:
        x,y=t[0][0],t[0][1]
        X=X+x
        Y=Y+y
    X=int(X/len(tab))
    Y=int(Y/len(tab))
    return (X,Y)
```

```python
def distance(centre,c,contours,l):
    x,y=centre
    for i in range (len(c)):
        (xc,yc)=c[i]

        if np.sqrt((xc-x)**2+(yc-y)**2)<=10  and l<len(contours[i]) and len(contours[i])<500 :
            return False
    return True

def carre_lettre(contour):
    min_x=contour[0][0][0]
    min_y=contour[0][0][0]
    max_x=contour[0][0][1]
    max_y=contour[0][0][1]
    print(min_x)
    for i in range (len(contour)):
        if contour[i][0][0]<=min_x:
            min_x=contour[i][0][0]
        if contour[i][0][1]<=min_y:
            min_y=contour[i][0][1]
        if contour[i][0][0]>=max_x:
            max_x=contour[i][0][0]
        if contour[i][0][1]>=max_y:
            max_y=contour[i][0][1]
    return min_x,min_y,max_x,max_y
```

```
140         def convergence_fun_lr(self, training_data, epochs, taille_mini_batch, test_data,f,df):
141             """ étude de la convergence en fontion du learning rate pour sigmoid """
142 
143             plt.clf()
144             l_r_lst=[0.001,0.01,0.1,1,10,100]
145             n = len(training_data)
146             i_biais=self.biais
147             i_poids=self.poids
148             for l_r in l_r_lst:
149                 self.biais=i_biais
150                 self.poids=i_poids
151 
152                 X=[]  #initialisation des listes/compteur pour afficher la convergence
153                 Y=[]
154                 for j in range(epochs):
155                     random.shuffle(training_data)  #on mélange les batchs
156                     mini_batches = [training_data[k:k+taille_mini_batch] for k in range(0, n, taille_mini_batch)]
157                     for mini_batch in mini_batches:
158                         self.update_mini_batch(mini_batch, l_r,f,df)
159                     X.append(j)
160 
161                     Y.append( self.eval(test_data,f) /len(test_data))
162 
163                 plt.plot(X,Y,label=str(l_r))
164             plt.legend(loc=1)
165             plt.title("Convergence du neurone en fonction du learning rate")
166             plt.show()
```

```
168         def convergence_fun_f(self, training_data, epochs, taille_mini_batch, lr, test_data):
169             """ étude de la convergence en fonction de la fonction d'activation """
170 
171             plt.clf()
172             f_lst=[sigmoid,atan,tanh]
173             nom_f_lst=['sigmoid','atan','tanh']
174             df_lst=[dsigmoid,datan,dtanh]
175             n = len(training_data)
176             i_biais=self.biais
177             i_poids=self.poids
178             for (f,df,nf) in zip(f_lst,df_lst,nom_f_lst):
179                 self.biais=i_biais
180                 self.poids=i_poids
181                 i=0
182                 X=[]  #initialisation des listes/compteur pour afficher la convergence
183                 Y=[]
184                 for j in range(epochs):
185                     random.shuffle(training_data)  #on mélange les batchs
186                     mini_batches = [training_data[k:k+taille_mini_batch] for k in range(0, n, taille_mini_batch)]
187                     for mini_batch in mini_batches:
188                         self.update_mini_batch(mini_batch, lr,f,df)
189                     Y.append( self.eval(test_data,f) /len(test_data))
190                     X.append(j)
191                 plt.plot(X,Y,label=str(nf))
192             plt.legend(loc=1)
193             plt.title("Convergence du neurone en fonction de la fonction d'activation")
194             plt.show()
```

```
196         def convergence_fun_batch_taille(self, training_data, epochs, lr, test_data,f,df):
197             """ étude de la convergence en fonction de la fonction d'activation """
198 
199             plt.clf()
200             lst=[10,50,100,1000]
201             n = len(training_data)
202             i_biais=self.biais
203             i_poids=self.poids
204             for taille_mini_batch in lst:
205                 self.biais=i_biais
206                 self.poids=i_poids
207                 i=0
208                 X=[]  #initialisation des listes/compteur pour afficher la convergence
209                 Y=[]
210                 for j in range(epochs):
211                     random.shuffle(training_data)  #on mélange les batchs
212                     mini_batches = [training_data[k:k+taille_mini_batch] for k in range(0, n, taille_mini_batch)]
213                     for mini_batch in mini_batches:
214                         self.update_mini_batch(mini_batch, lr,f,df)
215                         X.append(i)
216                         i=i+taille_mini_batch
217                         Y.append( self.eval(test_data,f) /len(test_data))
218                 plt.plot(X,Y,label=str(taille_mini_batch))
219             plt.legend(loc=1)
220             plt.title("Convergence du neurone en fonction de la taille du batch")
221             plt.show()
```

```
223         def convergence_fun_taille_neurone(self, training_data, epochs, taille_mini_batch, eta, test_data,f,df):
224             """ descente de gradient sur les minibatch """
225             X=[]  #initialisation des listes/compteur pour afficher la convergence
226             Y=[]
227             i=0
228             n = len(training_data)
229             for j in range(epochs):
230                 random.shuffle(training_data)  #on mélange les batchs
231                 mini_batches = [training_data[k:k+taille_mini_batch] for k in range(0, n, taille_mini_batch)]
232                 for mini_batch in mini_batches:
233                     self.update_mini_batch(mini_batch, eta,f,df)
234                     X.append(i)
235                     i=i+1
236                     Y.append( self.eval(test_data,f) /len(test_data))
237             return (X,Y)
```