TIPE reconnaissance de nom de boîtes de médicament automatisé par réseaux de neurones : listing des codes informatiques

TRABET Clément

June 8, 2022

Contents

1 Mise en forme des données 2

2 Création du réseau de neurones 3
2.1 Fonctions d'activations 5
2.2 Conversion ASCII 5
2.3 Test sur le réseau 5

3 Algorithmes d'études des différents paramètres 6
3.1 Paramètre : Fonction d'activation 6
3.2 Paramètre : coefficient d'apprentissage 6
3.3 Paramètre : taille des lots 7
3.4 Paramètre : taille du réseau de neurones 7

4 Analyse d'image 9
4.1 Filtre de Sobel 9
4.2 Parcours des contours 9
4.3 Barycentration 10
4.4 Entourage 11

5 Comparaison à la base de donnée 12
5.1 Distance de Levenshtein 12
5.2 Mot le plus proche 12
5.3 Traitement de la base de données médicament 13

6 Fonction principale 14

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

2 Création du réseau de neurones

```
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)

```

```python
    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 normalise_lst(self,lst):
        l=[]
        m=max(lst)
        for i in range (len(lst)):
            l.append(lst[i]/m)
        return l
```

2.1 Fonctions d'activations

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

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


def atan(z):
    return np.arctan(z)

def datan(z):
    return 1/(1+z*z)

def tanh(z):
    return np.tanh(z)
def dtanh(z):
    return 4/((np.exp(-z)+ np.exp(z))**2)
```

2.2 Conversion ASCII

```
def value(n):
    if n<10:
        return chr(n+48)
    if n<35:
        return chr(n+55)
    else:
        return chr(n+61)
```

2.3 Test sur le réseau

```

Net=Network([784,50,10])
#Net.descente_de_gradient(training_data, 100, 100, 2, test_data,sigmoid,dsigmoid)
#print(image(img,Net))
#Net.calc_cout(training_data, 15, 20, 1, test_data[:100], sigmoid, dsigmoid)
#Net.descente_de_gradient(training_data, 15, 100, 2, test_data,sigmoid,dsigmoid)
#Net.convergence_fun_lr(training_data, 15, 100, test_data,sigmoid,dsigmoid)
#Net.convergence_fun_f(training_data,20, 100, 2, test_data)
#Net.convergence_fun_batch_taille(training_data, 15, 2, test_data,sigmoid,dsigmoid)
#convergence_f_taille_neurone()
print("le neurone à appris")
```

3 Algorithmes d'études des différents paramètres

3.1 Paramètre : Fonction d'activation

```
    def convergence_fun_f(self, training_data, epochs, taille_mini_batch, lr,
test_data):
        """ étude de la convergence en fonction de la fonction d'activation """

        plt.clf()
        f_lst=[sigmoid,atan,tanh]
        nom_f_lst=['sigmoid','atan','tanh']
        df_lst=[dsigmoid,datan,dtanh]
        n = len(training_data)
        i_biais=self.biais
        i_poids=self.poids
        for (f,df,nf) in zip(f_lst,df_lst,nom_f_lst):
            self.biais=i_biais
            self.poids=i_poids
            i=0
            X=[] #initialisation des listes/compteur pour afficher la convergence
            Y=[]
            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:
                    self.update_mini_batch(mini_batch, lr,f,df)
                Y.append( self.eval(test_data,f) /len(test_data))
                X.append(j)
            plt.plot(X,Y,label=str(nf))
        plt.legend(loc=1)
        plt.title("Convergence du neurone en fonction de la fonction d'activation")
        plt.show()
```

3.2 Paramètre : coefficient d'apprentissage

```
    def convergence_fun_lr(self, training_data, epochs, taille_mini_batch,
test_data,f,df):
        """ étude de la convergence en fontion du learning rate pour sigmoid """

        plt.clf()
        l_r_lst=[0.001,0.01,0.1,1,10,100]
        n = len(training_data)
        i_biais=self.biais
        i_poids=self.poids
        for l_r in l_r_lst:
            self.biais=i_biais
            self.poids=i_poids

            X=[] #initialisation des listes/compteur pour afficher la convergence
            Y=[]
            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:
                    self.update_mini_batch(mini_batch, l_r,f,df)
                X.append(j)

                Y.append( self.eval(test_data,f) /len(test_data))

            plt.plot(X,Y,label=str(l_r))
```

```
        plt.legend(loc=1)
        plt.title("Convergence du neurone en fonction du learning rate")
        plt.show()
```

3.3 Paramètre : taille des lots

```
    def convergence_fun_batch_taille(self, training_data, epochs, lr, test_data,f,df):
        """ étude de la convergence en fonction de la fonction d'activation """

        plt.clf()
        lst=[10,50,100,1000]
        n = len(training_data)
        i_biais=self.biais
        i_poids=self.poids
        for taille_mini_batch in lst:
            self.biais=i_biais
            self.poids=i_poids
            i=0
            X=[] #initialisation des listes/compteur pour afficher la convergence
            Y=[]
            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:
                    self.update_mini_batch(mini_batch, lr,f,df)
                    X.append(i)
                    i=i+taille_mini_batch
                    Y.append( self.eval(test_data,f) /len(test_data))
            plt.plot(X,Y,label=str(taille_mini_batch))
        plt.legend(loc=1)
        plt.title("Convergence du neurone en fonction de la taille du batch")
        plt.show()
```

3.4 Paramètre : taille du réseau de neurones

```
def convergence_fun_taille_neurone(self, training_data, epochs, taille_mini_batch, eta, test_data,f,df):
    """ descente de gradient sur les minibatch """
    X=[] #initialisation des listes/compteur pour afficher la convergence
    Y=[]
    i=0
    n = len(training_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:
            self.update_mini_batch(mini_batch, eta,f,df)
            X.append(i)
            i=i+1
            Y.append( self.eval(test_data,f) /len(test_data))
    return (X,Y)
def convergence_f_taille_neurone():
    lst=[[784,200,100,50,10],
         [784, 80, 30, 10],
         [784, 50, 10],
         [784, 10]]

    for l in lst:
        Net=Network(l)
        (X,Y)=Net.convergence_fun_taille_neurone(training_data, 1, 100, 2, test_data, sigmoid,dsigmoid)
        plt.plot(X,Y,label=str(len(l)))
    plt.legend(loc=1)
    plt.title("Convergence du neurone en fonction de la taille du neurone")
    plt.show()
```

4 Analyse d'image

4.1 Filtre de Sobel

```
def filtre_de_Sobel(image, thresh=220):
    n,p=np.shape(image)
    new_image=[[[0,0,0] for i in range(p)] for j in range(n)]
    x=np.array([[-1, 0, 1],
                [-2, 0, 2],
                [-1, 0, 1]])
    y=np.array([[-1, -2, -1],
                [0, 0, 0],
                [1, 2, 1]])

    for i in range (1,n):
        for j in range(1,p):

            im=np.array(image[i-1:i+2,j-1:j+2])
            if (np.shape(im)==(3,3)):
                gx=x*im
                gy=y*im
                s=np.sum(gx)**2+np.sum(gy)**2
                s=np.sqrt(s)
                if s>=thresh:
                    new_image[i][j]=[255,255,255]

    plt.imshow(new_image)
    plt.show()
```

4.2 Parcours des contours

```
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
```

4.3 Barycentration

```
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)

def entourage(h : list, i : int, j : int):
    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
```

4.4 Entourage

```
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
```

5 Comparaison à la base de donnée

5.1 Distance de Levenshtein

```
def levenshtein(m1,m2):
    n=len(m1)
    p=len(m2)
    mat=[[0 for i in range(p+1)] for j in range(n+1)]
    for i in range(n+1):
        mat[i][0]=i
    for j in range(p+1):
        mat[0][j]=j
    for i in range(1,n+1):
        for j in range(1,p+1):
            if m1[i-1]==m2[j-1]:
                mat[i][j]=mat[i-1][j-1]
            else:
                mat[i][j]=min(mat[i-1][j], mat[i][j-1])+ 1
    return mat[n][p]
```

5.2 Mot le plus proche

```
def plus_proche(m):
    with open('data_propre.txt','r') as file:
        mot=[]
        ecart=len(m)
        i=0
        donne=file.readlines()
        for mot_test in donne:
            i=i+1
            n=levenshtein(m, mot_test)
            if n<ecart:
                ecart=n
                mot=[mot_test]
            elif n==ecart:
                mot.append(mot_test)
    return (mot,ecart)
```

5.3 Traitement de la base de données médicament

```
import csv
with open('data.txt', 'w') as f:

    with open('CIS_bdpm.csv') as csv_file:
        data = csv.reader(csv_file, delimiter=',')
        line_count = 0

        for line in data:
            tab=line[0]
            liste=tab.split()
            mot=liste[1]+"\n"
            f.write((mot))

import csv

with open('data.txt','r') as data_file:
    data=data_file.readlines()
    mot=""
    with open('data_propre.txt','w') as f:

        for new_mot in data:
            if mot != new_mot:
                mot=new_mot
                f.write(new_mot)
```

6 Fonction principale

```python
def nom_boite(image):
    #image=cv2.resize ( image ,(800 ,300), interpolation= cv2.INTER_LINEAR )
    imgray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    contours=contours_sobel(imgray)

    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), thickness=1)
                    img_extract=image[min_y : max_y , min_x : max_x]
                    cv2.imshow('extrait',img_extract)
                    lettre=Net.image(forme(img_extract))
                    mot.append(lettre)
                cv2.waitKey(0)
    cv2.imshow('Image', image)
    cv2.imshow('Image GRAY', imgray_1)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
    return plus_proche(mot)
```