SIMULATIONS NUMÉRIQUES - SAGEMATH/PYTHON

Ines Abdeljaoued Tej [1]

Réseaux de neurones et Application à la reconnaissance d'images

Nous vous proposons un TP sur Python2/SageMath portant sur la manipulation et l'apprentissage des réseaux neuronaux. Le plan que nous allons suivre est le suivant :

1 Brève introduction sur le perceptron

Ici, nous utilisons les notions de poids synaptiques et de fonction d'activation (appelée aussi fonction de transfert). Nous montrons deux exemples de perceptron (les fonctions logiques *et* et *ou*) ainsi que la limite du perceptron.

```
# importer les packages de calcul scientifique et numérique :
import numpy as np
import matplotlib.pyplot as plt
from pylab import *
```

1.1 Brève introduction sur le perceptron

1.1.1 La fonction logique OR

```
# Input data
X = np.array([ [0,0],[0,1],[1,0],[1,1] ])

# Output data
y = np.array([[0,1,1,1]]).T

# Fonction à seuil
def transfert(x):
    S = []
    for j in x:
        if j<1:
            S += [0]
        else:
```

1. MA à l'ESSAI, membre du Laboratoire BIMS, inestej@gmail.com

```
            S += [1]
    return(S)

w = np.array([1,1])
b = transfert(np.dot(X,w))
b
```

1.1.2 La fonction AND

```
# Input data
X = np.array([ [0,0],[0,1],[1,0],[1,1] ])

# Output data
y = np.array([[0,0,0,1]]).T

w = np.array([0.5,0.5])
b = transfert(np.dot(X,w))
b
```

1.1.3 Apprentissage

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

# Input data
X = np.array([ [0,0],[0,1],[1,0],[1,1] ])

# Output data
y = np.array([[0,1,1,1]]).T

w = 2*np.random.random((2,1)) - 1
for iter in xrange(100000):

    # forward propagation
    a = X
    b = sigmoid(np.dot(a,w))

    # Erreur (Output désiré - Output calculé par le RN)
    b_erreur = y - b

    # Descente donnée par l'erreur multipliée * la sigmoide en b
    b_delta = b_erreur * sigmoid(b)

    # Mettre à jour les poids synaptiques
    w += np.dot(a.T,b_delta)
```

```
print(w)
print(b)

# Input data
X = np.array([  [0,0,1],
                [0,1,1],
                [1,0,1],
                [1,1,1] ])

# Output data
y = np.array([[0,0,1,1]]).T

# seed random numbers to make calculation
# deterministic (just a good practice)
np.random.seed(1)

# Initialiser les poids synaptiques aléatoirement
# (avec une moyenne nulle)
w = 2*np.random.random((3,1)) - 1

def apprentissage(w,X):
    for iter in xrange(100000):
        # forward propagation
        a = X
        b = sigmoid(np.dot(a,w))
        # Erreur (Output désiré - Output calculé par le RN)
        b_erreur = y - b
        # Descente donnée par l'erreur multipliée * la sigmoide en b
        b_delta = b_erreur * sigmoid(b)
        # Mettre à jour les poids synaptiques
        w += np.dot(a.T,b_delta)
    return(w, b)

apprentissage(w,X)
```

1.1.4 La fonction logique XOR

```
# Input data
X = np.array([ [0,0],[0,1],[1,0],[1,1] ])

# Output data
y = np.array([[0,1,1,0]]).T

w = 2*np.random.random((2,1)) - 1
print w
```

```
apprentissage(w,X)
```

Nous constatons la limite du modèle de perceptron, avec XOR. Il est nécessaire d'utiliser des structures plus complexes, avec des couches de neurones cachées.

2 Perceptron multi-couches (PML) et Apprentissage

Nous traitons l'exemple *XOR* qui nécessite une classification moins triviale que la fonction *ou*. Il s'agit de mettre en place le PML dans le but de fournir une classification des données plus pertinente. Pour cela, nous utilisons une fonction à seuil puis une sigmoide, comme fonction de mise à feu (ou fonction de transfert).
Sur un RN à plusieurs couches, on effectue une recherche des meilleurs poids synaptiques, en utilisant l'algorithme de descente du gradient. Cet algorithme fonctionne de la manière suivante : $w_{i+1} = w_i + \delta_i$ où δ_i est l'erreur (entre activations et objectifs) multipliée par la dérivée partielle de la fonction coût.
Ci-dessous, un exemple sur R (penser à changer de noyau) pour la mise en place d'un RN qui calcule la racine carré.

```
# install.packages('neuralnet')
library(neuralnet)

input_app <- as.data.frame(runif(50, min=0, max=100))
output_app <- sqrt(input_app)
data_app <- cbind(input_app, output_app)
colnames(data_app) <- c('Input', 'Output')

print(head(data_app))

net.sqrt <- neuralnet(Output~Input, data_app, hidden=10, threshold=0.01)

data_test <- as.data.frame((1:10)^2)
net.results <- compute(net.sqrt, data_test)

print(net.results$net.result)

plot(net.sqrt, rep="best")
```

3 Traitement de la base de données MNIST

Nous proposons une étude détaillée d'un célèbre exemple tirée des travaux de Yann LeCun (Courant Institute, NYU), Corinna Cortes (Google Labs, New York) et Christopher J.C. Burges (Microsoft Research, Redmond).

Un petit exemple de fonctionnement de cette base est illustré dans ce lien http ://myselph.de/neuralNet.html où on peut écrire à la main un chiffre et un RN (implémenté sous matlab) permet de classifier correctement ce chiffre et donc le reconnaître.
Pour reproduire les travaux de LeCun et al., nous nous inspirons fortement du code développé par Evert Nieuwenburg dans https ://people.phys.ethz.ch/ evertv/
Sur 'Python3', il est nécessaire de préciser que les divisions doivent être calculées dans IR (et non dans IQ comme c'est fait par défaut) :

```
#from __future__ import division
```

4 Packages nécessaires à la manipulation de la BD

Il faut installer les packages utiles à la suite des opérations. Ce sont des packages comme 'gzip' ou 'pickle'.

```
# pickle et gzip permettent de télécharger
# la base de données et la décompresser :
import pickle
from pickle import load
import gzip

# Pour que les figures apparaissent dans cette feuille de travail :
\%matplotlib inline

# We seed the random number generator,
# so that we can reproducibly train the network
np.random.seed(1990)
```

Nous téléchargeons la base de données MNIST de Yann LeCun, qui est constituée de près de 60000 chiffres manuscrits, stockés dans des images de taille 28x28 pixels (i.e. de longueur 784).

```
with gzip.open('mnist.pkl.gz', 'r') as f:
    train_set, valid_set, test_set = pickle.load(f)
```

Nous restructurons la base de données, de telle sorte qu'elle corresponde à l'interface du réseau de neurones souhaité. Ici, nous avons besoin d'un input X et d'un output y où X correspond aux 784 pixels et y à un chiffre compris entre 0 et 9.
La sortie est en réalité une liste de 10 nombres, comprenant des 0 et un seul 1. L'indice de ce 1 correspond au chiffre de l'image (en input).

```
learn_data       = [(train_set[0][i], [1 if j == train_set[1][i] else 0 for j in range(10)])
                    for i in np.arange(len(train_set[0]))]
test_data        = [(test_set[0][i], [1 if j == test_set[1][i] else 0 for j in range(10)]) \
                    for i in np.arange(len(test_set[0]))]
validation_data  = [(valid_set[0][i], [1 if j == valid_set[1][i] else 0 for j in range(10)])
                    for i in np.arange(len(valid_set[0]))]
imgnr = 12
```

```
ax = plt
ax.matshow( np.reshape(test_set[0][imgnr], (28,28) ), cmap=cm.gray )
```

4.1 Définition de la fonction d'activation

Nous considérons la fonction sigmoïde ainsi que sa dérivée :

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

def sigmoid_deriv( x ):
    return sigmoid(x)*(1-sigmoid(x))
```

Nous introduisons la fonction coût : c'est la fonction à minimiser et elle dépend des poids synaptiques. Cette minimisation fait appel à une variante de l'algorithme de descente du gradient. Nous utilisons l'algorithme de backpropagation pour entrainer le RN (ce qui est basé sur un calcul de dérivées partielles de la fonction coût par rapport aux poids synaptiques w).

```
class QuadraticCost:
    """ Fonction coût E(w). """

    @staticmethod
    def fn(activations, objectifs):
        """ Evaluation du coût quadratique.
        C'est une optimisation classique. """
        return 0.5*(activations - objectifs)**2

    @staticmethod
    def fn_deriv(activations, objectifs):
        """ La fonction dérivée du coût. """
        return activations - objectifs

    @staticmethod
    def delta(inputs, activations, objectifs):
        """ Calcule l'erreur delta à la dernière couche
        (output) de la fonction coût. """
        return (activations - objectifs)*sigmoid_deriv(inputs)
```

Ce n'est pas l'unique manière de définir l'erreur dans un RN. Ci-dessous, il y a la fonction CrossEntropy définie comme suit :
"def fn(activations, objectifs) : return (objectifs*log(activations) + (1-objectifs)*log(1 - activations))"
La dérivée est donnée par
'-(objectifs/activations - (1-objectifs)/(1-activations))'
C'est le terme 'activations' qui dépend de w. Nous dérivons donc ce terme dans 'fn'.

4.2 Le réseau de neurone et son apprentissage

La définition de la classe neuralnetwork est développée ci-dessous par Evert Nieuwenburg. Nous supposons que le réseau est fortement connecté, dans le sens où tous les neurones d'une couche sont liés aux neurones de la couche suivante. Pour d'autres variantes de réseaux de neurones, et pour comprendre de plus près ce code, il est recommandé de visiter le site web de Michael Nielsens.

```
class neuralnetwork:
    """ Un réseau de neurones codé par Evert Nieuwenburg. """

    def __init__(self, shape, cost=QuadraticCost):
        """ Initialisation du RN """
        # La taille du RN
        self.shape = shape
        # Nombre de couches du RN
        self.number_of_layers = len(shape)
        # Définir la fonction coût
        self.cost = cost
        # Initialiser la marice des poids,
        # revoir l'échelle de la Gaussienne pour donner à chaque
        # neurone relatively peaked activation
        self.weights = [ np.random.normal(0,1/np.sqrt(shape[i+1]),(shape[i], shape[i+1])) \
                         for i in range(self.number_of_layers-1) ]
        # Initialiser les biais pour toutes les couches
        # (en dehors de la couche des inputs)
        self.biases  = [ np.random.normal(0,1,(shape[i])) \
                         for i in range(1,self.number_of_layers) ]

    def feedforward( self, inputdata ):
        """ Feed les input à travers le RN """

        # Fixer les inputs et outputs pour chaque couche
        self.input_to_layer = {}
        self.output_from_layer = {}
        # Pour la couche input, nous n'utilisons pas de
        # fonction d'activation
        self.input_to_layer[0] = inputdata
        self.output_from_layer[0] = np.array(inputdata)
        # Feed les input à travers les couches
        for layer in range(1,self.number_of_layers):
            self.input_to_layer[layer]    = np.dot( self.output_from_layer[layer-1], \
                                                    self.weights[layer-1] ) + self.biases[layer-1
            self.output_from_layer[layer] = np.array( sigmoid( self.input_to_layer[layer] ) )
        # Retourner l'output de la dernière couche du RN
        return self.output_from_layer[self.number_of_layers-1]
```

```
def backpropagate( self, targets ):
    """ Propager l'erreur utilisée par la descente du gradient"""
    self.delta = {}
    self.del_cost_del_bias = {}
    self.del_cost_del_weight = {}
    # Delta in dans l'output final
    self.delta[self.number_of_layers-1] = \
        (self.cost).delta(self.input_to_layer[self.number_of_layers-1], \
        self.output_from_layer[self.number_of_layers-1], targets )

    # Calculer les deltas pour les autres couches :
    for layer in np.arange(self.number_of_layers-2, -1, -1):
        self.delta[layer] = np.dot( self.delta[layer+1],  self.weights[layer].T ) * \
                                    sigmoid_deriv( np.array(self.input_to_layer[layer]) )

    # Calculer les dérivées partielles de l'Erreur E(w)
    # par rapport aux biais et aux poids synaptiques
    for layer in np.arange(self.number_of_layers-1, 0, -1):
        self.del_cost_del_bias[layer]   = self.delta[layer]
        self.del_cost_del_weight[layer] = np.dot( self.output_from_layer[layer-1].T, \
                                                  self.delta[layer] )

    return self.del_cost_del_bias, self.del_cost_del_weight

def train_mini_batch( self, data, rate, l2 ):
    """ Train the network on a mini-batch """

    # Diviser les données en deux : input et output
    inputs  = [ entry[0] for entry in data ]
    objectifs = [ entry[1] for entry in data ]

    # Feed les input à travers le réseau
    self.feedforward( inputs )
    # Propager l'erreur backwards
    self.backpropagate( objectifs )

    # Mettre à jour les biais et les poids
    n = len(objectifs)
    for layer in np.arange(1,self.number_of_layers):
        self.biases[layer-1]  -= (rate)*np.mean(self.del_cost_del_bias[layer], axis=0)
        self.weights[layer-1] -= (rate/n)*self.del_cost_del_weight[layer] - \
                                 rate*l2*self.weights[layer-1]

def stochastic_gradient_descent( self, data, number_of_epochs, mini_batch_size, \
```

```
                                        rate = 1, l2 = 0.1, test_data = None ):
    """ Apprentissage du RN en utilisant la méthode
    de descente du gradient stochastique. """

    # For every epoch:
    for epoch in np.arange(number_of_epochs):
        # Randomly split the data into mini_batches
        np.random.shuffle(data)
        batches = [ data[x:x+mini_batch_size] \
                    for x in np.arange(0, len(data), mini_batch_size) ]

        for batch in batches:
            self.train_mini_batch( batch, rate, l2 )

        if test_data != None:
            print("Epoch {0}: {1} / {2}".format(epoch, self.evaluate(test_data), \
                                                len(test_data)))

def evaluate(self, test_data):
    """ Evaluer la performance, en calculant
    le nombre de tests effectué. """
    count = 0
    for testcase in test_data:
        answer = np.argmax( testcase[1] )
        prediction = np.argmax( self.feedforward( testcase[0] ) )
        count = count + 1 if (answer - prediction) == 0 else count
    return count

def save(self, filename):
    """ Sauvegarder les poids synaptiques
    dans un fichier. """
    with open(filename, 'wb') as f:
        pickle.dump({'biases':self.biases, 'weights':self.weights}, f )

def load(self, filename):
    """ Charger les poids synaptiques à partir d'un fichier. """
    with open(filename, 'rb') as f:
        data = pickle.load(f)

    # Affecter les biais et les poids synaptiques
    self.biases = data['biases']
    self.weights = data['weights']
```

4.3 Apprentissage avec l'algorithme du gradient

Les images en input de la base de données MNIST data sont sous forme 28x28 pixels. Ils sont reconfigurés sous forme d'un vecteur de longueur 784. A la première couche du RN, il y a donc 784 inputs (ou neurones). La sortie (dernière couche) du RN contient 10 neurones. Nous allons entrainer un réseau avec une seule couche cachée, contenant 100 neurones.
Nous entraînons le RN en 25 epochs en utilisant un mini-batch de taille 10, avec un taux d'apprentissage de 0.1 et un paramètre de régularsation L2 égal à 0.1/longueur de l'ensemble. Durant l'apprentissage, on évalue la performance du réseau sur la base de données de validation.

```
mynet = neuralnetwork( [784,100,10] )
mynet.stochastic_gradient_descent( learn_data, 25, 10, 0.1, 0.1/len(train_set[0]), \
                                   test_data = validation_data )

# Sauvegarder les poids synaptiques obtenus dans le fichier MNIST
mynet.save("MNIST-QuadraticCost-Network")
```

4.4 Exemples de reconnaissance d'image avec le RN

Let's just look at the predictions of our network by showing the input image, printing the actual digit according to MNIST and including a plot of the output neurons of our network. Try running this many times and see on what kinds of digits the network does badly (usually 0 vs 6 or 3 vs 5 and 8)

```
mynet = neuralnetwork( [784,100,10] )
mynet.load("MNIST-QuadraticCost-Network")

# Choisir une entrée au hasard à partir de test-data.
imgnr = np.random.randint(0,10000)
# Feed cet input à travers le réseau pour avoir la prévision
prevision = mynet.feedforward( test_set[0][imgnr] )
print("L image numéro {0} est un {1} et notre RN a reconnu un {2}".format(imgnr, test_set[1]|

# Montrer l'image avec l'output du RN
fig, ax = plt.subplots(1,2,figsize=(8,4))
ax[0].matshow( np.reshape(test_set[0][imgnr], (28,28) ), cmap=cm.gray )
ax[1].plot( prevision, lw=3 )
ax[1].set_aspect(9)
```