TP Réseaux de neurones MLP

Préparation du TP

— Le cours d'introduction aux réseaux de neurones ;
— Bibliothèque de caractères `MNIST` ;

Le TP Réseaux de neurones MLP s'effectue sur deux séances de 2h et 4h, soit 6h présentielles sur machine. L'environnement de programmation utilisé est Python version 3+ avec le minimum de modules (*numpy* pour le calcul matriciel, mais ce n'est pas une obligation, et *matplotlib* pour l'affichage des données). L'objectif du TP est de mettre en œuvre un réseau de neurones de type *Multi-Layer Perceptron* sur un problème de reconnaissance de formes de chiffres manuscrits. Le réseau sera le plus simple possible et entièrement programmé *à la main*. Il existe des librairies spécialisées permettant de programmer des réseaux plus sophistiquées. Elles seront introduites dans les TPs suivants.

Les données d'apprentissage

Tout problème de reconnaissance de formes par apprentissage requiert l'utilisation d'une base d'exemples étiquetés. L'étiquetage signifie que l'on connait les *labels* ou *classes d'appartenance* des exemples de la base (apprentissage *supervisé*) :

$$\mathcal{A} = \left\{(\mathbf{x}^{(1)}, \mathbf{y}^{(1)}), (\mathbf{x}^{(2)}, \mathbf{y}^{(2)}), \ldots, (\mathbf{x}^{(k)}, \mathbf{y}^{(k)})\right\}$$

où les couples $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$, éléments de la base de donnée d'apprentissage, désignent les vecteurs de caractéristiques assignés en entrée du réseau et les vecteurs de sortie représentant les classes d'appartenance correspondantes.
La base de données de chiffres manuscrits MNIST (`http://yann.lecun.com/exdb/mnist`) propose un ensemble d'apprentissage de 60000 échantillons et un ensemble de test de 10000 échantillons. Les caractères ont été normalisés en taille et centrés pour constituer des images en niveau de gris (après normalisation) de 28×28 pixels (contre 20×20 au départ et en binaire). Cette base permet de tester facilement des algorithmes d'apprentissage sur des données réelles nécessitant un minimum de pré-traitements.

Chargement des données.

Les données sont sauvegardées dans quatre fichiers :
— `basetrain.npy` : les données d'apprentissage (matrice contenant une image par ligne)
— `labeltrain.npy` : les labels de la base d'apprentissage (classes des caractères de 0 à 9)
— `basetest.npy` : les données de test (matrice contenant une image par ligne)
— `labeltest.npy` : les labels de la base de test (classes des caractères de 0 à 9)
Exemple de code Python permettant de charger les fichiers fournis :

```
# import librairie numpy:
import numpy as np
```

```
# chargement des bases d'apprentissage et de test:
Basetrain = np.load('basetrain.npy')
Labeltrain = np.load('labeltrain.npy')
Basetest = np.load('basetest.npy')
Labeltest = np.load('labeltest.npy')
```

Ces bases contiennent respectivement 10000 échantillons pour l'apprentissage et 2000 échantillons pour la généralisation.

Visualisation des caractères.

L'exemple suivant permet de visualiser la première image de la base de test en niveaux de gris :

```
# import librairie numpy et matplotlib:
import numpy as np
import matplotlib.pyplot as plt

# affichage:
plt.figure(1, figsize=(3, 3))
plt.imshow(Basetest[1,:].reshape(28,28),cmap=plt.cm.gray_r)
plt.show()
```

Questions posées

- Pourquoi sépare-t-on les données en deux ensembles, d'apprentissage et de test ?
- Ecrire un module permettant de charger les données d'apprentissage et de test, ainsi que les labels associés, dans des variables qui pourront être utilisées par la suite ;
- Visualiser quelques images de la base d'apprentissage et de la base de test.

Perceptron simple

Dans le TP nous considérerons uniquement des neurones dont les sorties sont à valeurs dans $[-1, +1]$. La sortie d'un neurone formel s'écrit comme :

$$y = \sigma(\sum_i \omega_i x_i)$$

où σ désigne la fonction sigmoïde à valeurs entre dans $[-1, +1]$ (tangente hyperbolique) :

$$\sigma(v) = \frac{1 - e^{-2v}}{1 + e^{-2v}}$$

La fonction dérivée s'écrit très simplement :

$$\sigma'(v) = 1 - \sigma(v)^2$$

La sortie peut être interprétée comme un produit scalaire des entrées avec des poids du neurone, ce qui permet l'écriture vectorielle suivante :

$$y = \sigma(v) = \sigma([\omega_0, \omega_1, \ldots, \omega_n] \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \ldots \\ x_n \end{bmatrix}) = \mathbf{w}^\top \mathbf{x}$$

Assembler m cellules au sein d'une couche conduit au calcul matriciel suivant :

$$\mathbf{v} = [\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_m]^\top \mathbf{x} = \Omega \mathbf{x}$$

où $\Omega = (\omega_{ij})$ désigne la matrice des poids de la couche neuronale. A noter que les indices des entrées commencent à 0, x_0 représentant l'entrée constante unitaire permettant de représenter le seuil des cellules via les poids ω_{i0}.
Enfin, présenter plusieurs exemples en entrée des cellules pour obtenir l'ensemble de sorties correspondant conduit à la généralisation suivante du produit matriciel :

$$\mathbf{V} = \Omega \mathbf{X}$$

où $\mathbf{X}$ et $\mathbf{V}$ représentent respectivement les matrices des entrées en colonne et des sorties avant sigmoïde, également en colonne.
Ces considérations permettent à présent d'envisager l'écriture de fonctions de calcul matriciel des états des cellules d'un réseau. Pour K exemples :

$$\mathbf{V} = \begin{bmatrix} \begin{bmatrix} v_1^1 \\ v_2^1 \\ v_3^1 \\ \vdots \\ v_m^1 \end{bmatrix} \begin{bmatrix} v_1^2 \\ v_2^2 \\ v_3^2 \\ \vdots \\ v_m^2 \end{bmatrix} \ldots \begin{bmatrix} v_1^K \\ v_2^K \\ v_3^K \\ \vdots \\ v_m^K \end{bmatrix} \end{bmatrix} \tag{1}$$

$$= \begin{bmatrix} \omega_{10} & \omega_{11} & \omega_{12} & \ldots & \omega_{1m} \\ \omega_{20} & \omega_{21} & \omega_{22} & \ldots & \omega_{2m} \\ \ldots & & & & \\ \omega_{m0} & \omega_{m1} & \omega_{22} & \ldots & \omega_{2m} \end{bmatrix} . \begin{bmatrix} \begin{bmatrix} 1 \\ x_1^1 \\ x_2^1 \\ \vdots \\ x_n^1 \end{bmatrix} \begin{bmatrix} 1 \\ x_1^2 \\ x_2^2 \\ \vdots \\ x_n^2 \end{bmatrix} \ldots \begin{bmatrix} 1 \\ x_1^K \\ x_2^K \\ \vdots \\ x_n^K \end{bmatrix} \end{bmatrix} \tag{2}$$

Bien noter que la première colonne de la matrice des poids Ω représente les poids ω_0 des cellules, donc les biais. Il s'ensuit que la matrice des entrées ici représentée est la matrice des exemples de la base augmentée d'une entrée x_0 toujours positionnée à 1.

Questions posées

- Quel intérêt y-a-t-il à représenter les seuils θ des cellules par des poids reliés à des entrées constantes à 1 ?
- Quelle structure de réseau peut-on utiliser dans le problème de reconnaissance de chiffres manuscrits posé ?
- Ecrire une fonction `w = mlp1def(n, m)` permettant de générer une matrice de poids aléatoires (distribution uniforme) dont les valeurs sont comprises dans l'intervalle $[-1, +1]$ et de dimension $m \times (n + 1)$ (nombre de cellules $\times$ nombre d'entrées augmenté de 1) ;
- Ecrire une fonction `y = sigmo(v)` permettant de calculer la sortie d'un neurone en fonction de son état `v`, `v` pouvant être un vecteur d'états représentant l'état de toutes les cellules d'une couche de neurones ;
- Ecrire une fonction `y = mlp1run(x, w)` permettant de calculer le vecteur de sortie d'une couche de neurones en fonction de ses entrées `x` (un vecteur colonne de dimension n pour un seul exemple présenté ou une matrice $(n \times K)$ pour K exemples présentés) et de sa matrice de poids `w` (toujours de dimension $m \times (n+1)$). Il revient à la fonction `mlprun()` d'augmenter le vecteur des entrées en ajoutant une ligne à 1 pour représenter les biais.

Pour la programmation il est demandé d'ouvrir un nouveau fichier `neuralnetwork.py` qui contiendra l'ensemble des fonctions écrites pour le TP. Afin de tester ces fonctions, on définira une matrice d'exemples jouets tel que par exemple :

```
# 3 jeux d'entrees pour un reseau a 5 cellules :
x = np.array([[-1, -0.5, 0, 0.5, 1], [-1, -0.5, 0, 0.5, 1], [-1, -0.5, 0, 0.5, 1]])
# rangement des exemples en colonne:
x = np.transpose(x)
# dimensions de la matrice obtenue:
x.shape
> (5, 3)
```

Décision

A ce stade, pour chaque exemple présenté en entrée, on obtient un vecteur de sortie dans $[-1,+1]^m$ où m désigne le nombre de cellules, donc le nombre de sorties du réseau, en concordance avec l'interprétation habituelle d'un réseau de neurones comme un approximateur de fonction :

$$f_\Omega : \Re^n \rightarrow [-1,+1]^m \subset \Re^m, \mathbf{x} \longrightarrow \mathbf{y} = f_\Omega(\mathbf{x})$$

Dans un problème de classification, on cherche à associer à tout exemple en entrée sa classe d'appartenance en sortie. Dans un problème à m classes, on choisit un neurone de la couche de sortie pour représenter une classe, suivant en cela l'algorithme WTA ou *Winner Takes All*. Pour un exemple $\mathbf{x}^k$ de classe C :

$$C = \arg\max_i \{y_i\}$$

Ceci fonctionne dans la mesure où les sorties du réseau sont proportionnelles à la densité de probabilité d'appartenance d'un exemple à une classe, ce qui peut être vérifié expérimentalement à condition d'utiliser des fonctions sigmoïdes normalisées à valeurs dans $[0,1]$ et sommant à 1 sur toutes les sorties (fonctions *softmax*), et une fonction de coût quadratique pour l'apprentissage. Choisir le neurone dont l'activité est la plus élevée revient alors à adopter le *principe du maximum de vraisemblance*.
Il n'est pas demandé dans le TP d'utiliser des fonctions *softmax*. La fonction sigmoïde à sorties dans $[-1,+1]$ peut être utilisée pour la décision WTA.

Questions posées

- Ecrire une fonction `mlpclass(y)` qui, à partir du vecteur (ou de la matrice) des sorties `y` d'un réseau, retourne le label (ou le vecteur des labels) du ou des exemples correspondants.
- Ecrire une fonction `[score rate] = score(Label, LabelD)` qui compare le label (ou le vecteur des labels) calculé par un réseau avec le label (ou les labels) désiré(s), et retourne le nombre de labels correctement estimés ainsi que le taux en pourcentage d'exemples bien classés.

Typiquement, la liste des opérations à effectuer pour calculer les labels des exemples précédents suite à leur présentation en entrée d'un réseau conduira aux lignes de programmation suivantes :

```
# definition d'un reseau a 5 entrees et 2 cellules:
w = mlp1def( 5, 2)
# calcul des sorties du reseau:
y = mlp1run( x, w)
# estimation WTA des labels:
Labels = mlpclass(y)
```

```
# score obtenu:
[score, rate] = score(Labels, array([2, 1, 2]))
print( "Taux de reconnaissance: " + str(rate) )
```

Apprentissage

La mise en oeuvre d'un apprentissage requiert de décider pour une fonction de coût. Nous utiliserons dans la suite la fonction de coût la plus utilisée, dite fonction de *coût quadratique*. Sur un ensemble d'apprentissage comportant K exemples :

$$Q = \frac{1}{2} \sum_k \|\hat{\mathbf{y}}^{(k)} - \mathbf{y}^{(k)}\|^2$$

où ici les sorties du réseau sont désignées par $\hat{\mathbf{y}}$, c'est à dire comme les estimées des sorties désirées $\mathbf{y}$ représentant la classe d'appartenance des exemples. Les sorties désirées sont codées en application de la règle WTA dans le cas optimal. La sortie du neurone représentant la classe se voit imposer la valeur maximale en sortie de sigmoïde, $+1$, tandis que les sorties des autres neurones se voient imposer la valeur minimale admissible en sortie de sigmoïde : -1. Pour exemple la classe de label 2 est codée par le vecteur de sorties désirés suivant :

$$\mathbf{y} = \begin{bmatrix} -1 \\ +1 \\ -1 \\ -1 \\ -1 \end{bmatrix}$$

L'erreur du réseau commise pour un exemple $\mathbf{x}$ présenté en entrée s'écrit :

$$\varepsilon_\Omega(\mathbf{x}) = \hat{\mathbf{y}} - \mathbf{y} = f_\Omega(\mathbf{x}) - \mathbf{y} = \sigma(\Omega\mathbf{x}) - \mathbf{y}$$

ce qui conduit à l'expression suivante du coût quadratique :

$$Q(\Omega) = \frac{1}{2} \sum_i (\sigma(\Omega_i \mathbf{x}) - y_i)^2$$

L'algorithme de descente du gradient permettant de minimiser l'erreur quadratique s'écrit pour une connexion donnée ω_{ij} du réseau :

$$\omega_{ij}^q \longrightarrow \omega_{ij}^{q+1} = \omega_{ij}^q - \lambda \frac{\partial Q}{\partial \omega_{ij}}$$

où λ désigne le *pas d'apprentissage*. Après dérivation on obtient :

$$\frac{\partial Q}{\partial \omega_{ij}} = \varepsilon_i \frac{\partial \sigma(\Omega\mathbf{x})}{\partial \omega_{ij}} = \varepsilon_i \sigma'(v_i) x_j$$

$\varepsilon_i = (\hat{y}_i - y_i)$ désigne l'erreur sur le neurone i. Pour des raisons de commodité lors de la programmation de la rétropropagation sur un réseau MLP, il est souhaitable de faire apparaître le gradient $\delta_i = \frac{\partial Q}{\partial v_i}$ dans l'expression précédente :

$$\frac{\partial Q}{\partial \omega_{ij}} = \delta_i x_j \text{ avec } \delta_i = \varepsilon_i \sigma'(v_i)$$

Pour les poids $[\omega_{i0}, \omega_{i1}, \ldots, \omega_{in}]$ d'une cellule i, le gradient s'obtient par le calcul matriciel suivant :

$$\left[\frac{\partial Q}{\partial \omega_{i0}}, \frac{\partial Q}{\partial \omega_{i1}}, \ldots, \frac{\partial Q}{\partial \omega_{in}}\right] = \delta_i \mathbf{x}^\top$$

Pour toutes les connexions du réseau on obtient :

$$\nabla_\Omega = \begin{bmatrix} \delta_1 \\ \delta_2 \\ \vdots \\ \delta_m \end{bmatrix} \mathbf{x}^\top$$

et pour K exemples rangés à apprendre, en colonne dans la matrice $\mathbf{X}$, on généralise par :

$$\nabla_\Omega = \Delta \mathbf{X}^\top$$

où Δ désigne la matrice des δ ci-dessus rangés en colonne. Avec cette expression, le gradient local pour une connexion ω_{ij} quelconque s'écrit :

$$\frac{\partial Q}{\partial \omega_{ij}} = \sum_k^K \delta_i^k x_j^k$$

Ce qui s'apparent au *gradient total* : les contributions dues à chaque exemple de la base sont sommées avant adaptation des poids.

Questions posées

- Ecrire une fonction `sigmop(v)` implémentant le calcul de la fonction sigmoïde dérivée pour un état, ou un vecteur d'états `v` ;
- Ecrire la fonction `label2target(c)` permettant de générer le vecteur (ou la matrice) de sorties désirées correspondant à la classe (ou au vecteur des classes) `c` présenté en argument ;
- Ecrire la fonction `mlperror(y, target)` où `y` désigne le vecteur de sorties du réseau (ou la matrice des vecteurs de sorties du réseau pour plusieurs exemples) et `target` le vecteur des sorties désirées (ou la matrice des sorties désirées). La fonction retourne le vecteur (ou la matrice) des erreurs ;
- Ecrire la fonction `sqrerror(error)` qui retourne l'erreur quadratique calculée à partir du vecteur d'erreurs (ou de la matrice d'erreurs) donnée en argument de la fonction.
- En reprenant la règle d'apprentissage définie ci-dessus pour un réseau sans couche cachée, écrire la fonction de modification des poids d'un réseau telle que :

```
# x: vecteur ou matrice des exemples a apprendre
# target: vecteur ou matrice des sorties desirees
# w: matrice des poids du reseau
# lr: pas d'apprentissage (learning rate)
# it: nombre d'iterations d'apprentisssage
# nw: nouvelle matrice des poids
# L: erreur quadratique (ou vecteur des erreurs quadratiques lorsque plusieurs
    iterations sont demandees)
[nw, L] = mlp1train(x, target, w, lr, it)
```

- Tester un apprentissage sur 1000 itérations du jeu de données défini plus haut et constater la décroissance de la fonction de coût quadratique en l'affichant :

```
# inclusion des librairies:
import numpy as np
import matplotlib.pyplot as plt
from neuraltework import *
# donnees a apprendre:
x = ...
target = ...
# parametres d'apprentissage:
w =
it = 1000
lr = 0.01
# apprentissage:
[nw, L] = mlp1train(x, target, w, lr, it)
# affichage:
axe_x=np.linspace(1,it,it)
plt.plot(axe_x,L)
plt.ylabel('Cout quadratique')
plt.xlabel("Iterations d'apprentissage")
plt.show()
```

— Relancer l'apprentissage en prenant `lr = 1`. Expliquer le résultat obtenu.
— Effectuer un apprentissage des données de la base d'apprentissage MNIST et donner les scores obtenus en apprentissage et en généralisation

1 Apprentissage par rétropropagation

Les réseaux à couche unique du type précédent ne permettent de séparer des formes non linéairement séparables. Ajouter une couche dite *couche cachée* permet de résoudre le problème des données non linéairement séparables. La difficulté est d'estimer l'erreur des cellules appartenant à la couche cachée (ou aux couches cachées lorsqu'il y en a plusieurs). On y parvient en utilisant l'algorithme de *rétropropagation du gradient*. Dans la suite de TP on souhaite programmer cet algorithme pour des réseaux comportant une couche cachée.
Pour une connexion ω_{ij} quelconque d'une cellule i quelconque du réseau, la règle de modification selon le principe de descente du gradient s'écrit toujours :

$$\omega_{ij}^q \longrightarrow \omega_{ij}^{q+1} = \omega_{ij}^q - \lambda \frac{\partial Q}{\partial \omega_{ij}}$$

On décompose ensuite le gradient en fonction de l'état de la cellule v_i de la cellule i pour faire apparaître l'erreur δ (pour rappel, $v_i = \mathbf{w}^\top \mathbf{x}$ avec $\mathbf{w}_i$ les connexions entrantes de la cellule i et $\mathbf{x}$ l'état des entrées de la cellule) :

$$\frac{\partial Q}{\partial \omega_{ij}} = \frac{\partial Q}{\partial v_i}\frac{\partial v_i}{\partial \omega_{ij}} = \delta_i \frac{\partial v_i}{\partial \omega_{ij}} = \delta_i x_j$$

Le calcul de δ est simple lorsque l'on est sur une cellule de sortie du réseau puisque l'on connaît alors l'erreur. Il prend la même forme que pour un réseau sans couche cachée :

$$\delta_i = \frac{\partial}{\partial v_i}\left(\frac{1}{2}\sum_l (\hat{y}_l - y_l)^2\right) = \frac{1}{2} 2\varepsilon_i \frac{\partial \hat{y}_i}{\partial v_i} = \varepsilon_i \sigma'(v_i)$$

Pour une cellule de la couche cachée, δ doit être décomposé en fonction des états des cellules de la couche suivante :

$$\delta_i^c = \sum_l \delta_l^{c+1} \frac{\partial v_l^{c+1}}{\partial v_i^c}$$

On a :

$$\frac{\partial v_l^{c+1}}{\partial v_i^c} = \frac{\partial}{\partial v_i^c} \mathbf{w}_l^{c+1} \mathbf{x}^{c+1} = \frac{\partial}{\partial v_i^c} \mathbf{w}_l^{c+1} \mathbf{y}^c = \frac{\partial}{\partial v_i^c} \omega_{li}^{c+1} y_i^c = \omega_{li}^{c+1} \sigma'(v_i)$$

d'où :

$$\delta_i^c = \sigma'(v_i) \sum_l \omega_{li}^{c+1} \delta_l^{c+1}$$

connaissant les δ de la couche suivante (couche de sortie), on en déduit la règle d'apprentissage :

$$\frac{\partial Q}{\partial \omega_{ij}} = \sigma'(v_i) \left(\sum_l \omega_{li}^{c+1} \varepsilon_l \sigma'(v_l^{c+1}) \right) x_j$$

Pour la programmation on aura tout intérêt à utiliser la règle de récurrence sur δ précédente plutôt que le calcul de la règle complète ci-dessus :

$$\delta_i^c = \sigma'(v_i) [\omega_{1i}, \omega_{2i}, \ldots, \omega_{mi}] \begin{bmatrix} \delta_1 \\ \delta_2 \\ \vdots \\ \delta_m \end{bmatrix}^{c+1}$$

Soit pour les h cellules de la couche cachée :

$$\begin{bmatrix} \delta_1 \\ \delta_2 \\ \vdots \\ \delta_h \end{bmatrix}^c = \begin{bmatrix} \sigma'(v_1) \\ \sigma'(v_2) \\ \vdots \\ \sigma'(v_h) \end{bmatrix} .* \begin{bmatrix} \omega_{11} & \omega_{21} & \ldots & \omega_{m1} \\ \omega_{12} & \omega_{22} & \ldots, & \omega_{m2} \\ \ldots & & & \\ \omega_{1h} & \omega_{2h} & \ldots & \omega_{mh} \end{bmatrix} \begin{bmatrix} \delta_1 \\ \delta_2 \\ \vdots \\ \delta_m \end{bmatrix}^{c+1} = \begin{bmatrix} \sigma'(v_1) \\ \sigma'(v_2) \\ \vdots \\ \sigma'(v_h) \end{bmatrix} .* \Omega^{*\top} \begin{bmatrix} \delta_1 \\ \delta_2 \\ \vdots \\ \delta_m \end{bmatrix}^{c+1}$$

où l'on voit apparaître la matrice transposée des poids de la couche de sortie, mais sans les poids biais, pour cette raison notée Ω^*. L'opérateur $.*$ désigne la multiplication point à point. Comme pour le réseau sans couche cachée, cette expression se généralise aux K exemples pour réaliser un gradient total.

Questions posées

En utilisant le même principe de programmation que précédemment, écrire les fonctions suivantes de traitement d'un réseau MLP à une couche cachée :

- Définition d'un réseau à une couche cachée : `[w1, w2] = mlp2def(n, c, m)` où `c` désigne le nombre de cellules de la couche cachée ;
- Propagation des entrées pour le calcul des sorties du réseau : `y = mlp2run(x, w1, w2)`
- Modification des poids du réseau par rétropropagation de l'erreur : `[nw1, nw2, L] = mlp2train(x, target, w1, w2, lr, it)`
- Tester l'algorithme sur l'exemple jouet vu plus haut
- Tester l'algorithme sur le jeu de données MNIST et observer la décroissance de la fonction de coût pour différentes valeurs du pas d'apprentissage. Quels sont les taux de reconnaissance obtenus en apprentissage et en généralisation ?

Annexes

Installation de Python pip

pip ou *pip3* est un programme d'installation de packages pour Pyton(3). Il doit être installé par l'administrateur de la machine. Il est normalement installé automatiquement pour les versions de Python supérieures à Python 3.4. Sous linux :

```
sudo apt-get install python3-pip
```

Installation de Python Numpy

Déjà installé de base avec Python3. Sinon :

```
python3 -m pip install -U numpy
```

Installation de Python Matplotlib

```
python3 -m pip install -U matplotlib
%  python3 -m pip --proxy http://<login>:<passwrd>@134.157.107.4:3128 install -U matplotlib
```

1.1 Installation sous windows

Installateur de base pour la distribution de Python et librairies numpy et matplotlib :

```
http://winpython.github.io/
```

Ou (mieux) installation d'Anaconda (gestionnaire de paquets également) :

```
https://www.anaconda.com/download/#download
```