TP n°3: Réseaux de neurones

Le compte-rendu doit être rédigé en Jupyter-Notebook, Markdown ou LaTeX. Il doit comporter à la fois les codes mais également **et surtout** *des commentaires et interprétations clairs et pertinents concernant (i) vos choix, (ii) vos résultats en rapport avec les notions vues en cours.*

1 Descriptif du TP

Dans ce troisième TP, nous allons nous pencher sur la conception de réseaux de neurones de type "perceptrons" à une ou plusieurs couches, afin de classer les données de la classe `MNIST`. Les résultats obtenus seront **soigneusement discutés et comparés** aux résultats obtenus dans les TP1 et TP2. Le compte-rendu, qui rassemblera toutes ces informations, formera la **note de CC1**.

Dans le détail, nous allons successivement nous pencher sur:

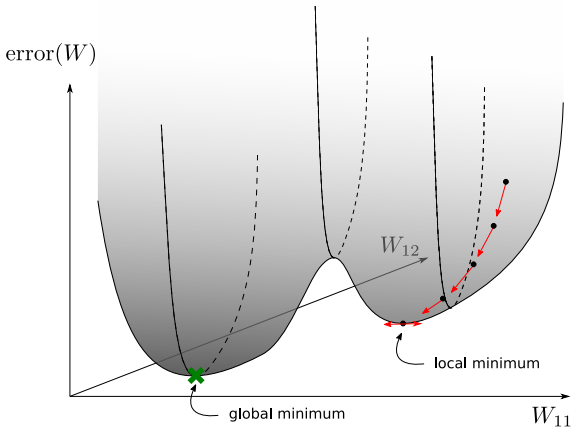
1. la conception d'un réseau de neurones à une couche cachée
2. l'extension à plusieurs couches
3. l'utilisation ou non de représentations telles qu'HOG en entrée du réseau.

Contrairement aux TP1 et TP2, la partie "implémentation" du TP3 est plutôt rapide et simple à traiter si les TP1 et TP2 ont été soigneusement traités. Il est donc surtout attendu que vous travailliez au maximum sur (i) l'interprétation et l'analyse ainsi que sur (ii) des représentations visuelles élégantes et parlantes de vos résultats.

2 Perceptron mono-couche

La librairie `sklearn` donne accès au paquet `neural_network` qui contient la fonction `MLPClassifier` permettant de créer un classifieur de type perceptron multi-couches. De nombreux paramètres permettent de le configurer: il s'agit en effet de choisir avant tout le nombre de neurones par couches via l'option `hidden_layer_sizes`, la fonction d'activation via l'option `activation`, mais également un ensemble de paramètres propres à la façon d'entrainer le réseau.

Rappelons en effet que le réseau de neurones résoud un problème d'optimisation *non convexe* au moyen d'algorithmes adaptés aux problèmes *convexes* (les seuls qu'on sache vraiment résoudre): toutes ces méthodes découlent d'une manière ou d'une autre de l'algorithme de "descente de gradient" (*gradient descent* en anglais). Cette méthode cherche les paramètres de poids des couches $W_{11}, W_{12}, \ldots$ du réseau qui minimisent l'erreur d'apprentissage en (i) commençant par des $W_{11}, W_{12}, \ldots$ pris au hasard, puis (ii) en "descendant" le long de l'opposé de la dérivée (ce qu'on appelle le "gradient") de la fonction d'erreur jusqu'à trouver un minimum local, comme décrit dans la représentation ci-dessous.



Mais la descente de gradient est délicate à contrôler: l'idée est de "sauter" de point en point dans l'espace des $(W_{11}, W_{12}, \ldots)$ pour atteindre un minimum le plus vite possible (en faisant donc de grands pas!) mais tout en essayant de ne pas "rater" le minimum ou de sauter trop loin dans une autre vallée sans jamais s'arrêter (on ne peut donc pas faire de si grands pas!). Un exemple typique de configuration `MLPClassifier` est le suivant:

```
from sklearn.neural_network import MLPClassifier
classifier = MLPClassifier(activation='relu', hidden_layer_sizes
    =(50), solver='sgd', max_iter=100,learning_rate='constant',
    learning_rate_init=.1, verbose=10)
```

Exécutez la commande précédente et appliquez le classifieur (grâce à la fonction `fit()` vue aux TPs précédents) aux données d'entrainement de la base `MNIST`. Répétez l'opération pour différents choix des paramètres (en gardant `verbose=10` qui permet de suivre l'évolution de l'apprentissage). En vous basant sur l'explication précédente, tentez d'expliquer ce que la fonction `fit()` vous retourne et trouvez alors un jeu de paramètres qui vous semble "convenable" (il faut bien sûr justifier!).

Une fois que vous avez trouvé les "bons" paramètres, évaluez les performances de votre perceptron à la fois sur les données d'entrainement et les données de validation (test) de la base `MNIST` (comme dans les TPs précédents, grâce à la fonction `score()`). Si besoin, adaptez à nouveau vos paramètres d'apprentissage (notamment le nombre de neurones) afin d'obtenir un score de validation correct.

3 Généralisation au cas multi-couches

3.1 Données `MNIST` brutes

Nous passons maintenant à la généralisation à un perceptron multi-couches. Pour cela, il suffit de paramétrer la variable `hidden_layer_sizes` de la fonction `MLPClassifier` par une liste contenant l'ensemble des tailles des couches au lieu d'une liste ne contenant qu'une seule valeur comme précédemment.

L'avantage d'une approche multi-couches est qu'elle permet en général de drastiquement réduire le nombre de neurones que l'on utiliserait avec un réseau mono-couche pour atteindre le même niveau de performances. Confirmez cette affirmation en pratique en étudiant les performances atteintes par des réseaux simple-couche aux performances atteintes par des réseaux multi-couches. D'un point de vue théorique, pourquoi peut-on affirmer cela (que gagne-t-on en effet à utiliser plusieurs couches plutôt qu'une?)?

Après avoir validé votre réseau de neurones, en utilisant un logiciel de dessin vectoriel (vous pouvez utiliser Inkscape par exemple), dessinez le réseau de neurones ainsi obtenu en indiquant:

- la taille de chaque couche cachée de neurones (cachée = excluant la couche d'entrée et la couche de sortie)
- la taille des entrées et la taille des sorties
- les tailles des matrices de connexion entre couches.

Ces informations sont en fait stockées dans le paramètre `classifier.coefs_` sous la forme d'une liste de matrices. En utilisant la fonction `shape()` de la librairie `numpy`, vérifiez que (i) la taille de `classifier.coefs_` est correcte et que (ii) la taille de chaque matrice stockée dans `classifier.coefs_` est aussi correcte. En déduire le nombre total de paramètres W_{ij} à entrainer dans le réseau dans le cas simple versus multi-couches.

3.2 Représentation HOG

Effectuez le même travail que précédemment, mais cette fois-ci en prenant en entrée, non plus les données `MNIST` brutes, mais leurs représentations HOG comme vu dans le TP précédent.

Parvient-on, grâce à cette représentation, à réduire la taille du réseau de neurones? Que peut-on en déduire?

4 Rassemblons les résultats

Afin de fournir un rapport propre, clair et facilement réutilisable des travaux menés durant les TP1, TP2 et TP3, nous allons maintenant synthétiser l'ensemble des codes Python développés jusqu'ici en un seul fichier `.ipynb` (ou `.md` ou `.tex`). Ce fichier pourra prendre une forme similaire à celle qui suit mais n'hésitez surtout pas à adopter la structure de code et de commentaires qui vous parait la plus adaptée:

```
# Libraries:
import ... as ...
...
import ... as ...

# Options:
data    = 'mnist_784'    # 'mnist_784','Fashion-MNIST'
classif = 'mlp'          # 'svm','kNN','mlp'
hog_on  = 0              # 0,1
classes = 10             # 2,10

# Load data, create 'train', 'test':
X,y = ...
...

# Build classifier:
if classif == 'svm':
  classifier = ...
elif classif == 'kNN':
  classifier = ...
...

# Fit to data:
classifier.fit(...)

# Score:
print("Training set score: ",classifier.score(...))
print("Testing set score: ",classifier.score(...))
...
```

Tentez de rendre votre compte-rendu le plus complet (commentaires, explications) et le plus "visuel" possible (graphes, images, dessins, tableaux, etc.) afin qu'un non-expert puisse très facilement comprendre et réutiliser votre travail. Votre note de CC1 en dépend fortement!