TP perceptron multi-couches

Objectifs du TP :
- découvrir l'apprentissage par perceptron multi-couches
- utiliser scikit-learn pour déterminer une architecture MLP performante pour un jeu de données

1 Apprentissage par perceptron multi-couches sous sklearn

La plate-forme `sklearn`, depuis sa version `0.18.1`, fournit quelques fonctionnalités pour l'apprentissage à partir de perceptron multi-couches, en classification (classe `MLPClassifier`) et en régression (classe `MLPRegressor`).

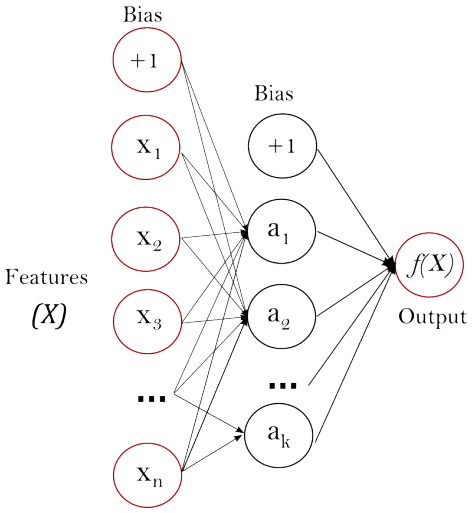

FIGURE 1 – Un perceptron à une couche cachée (source : documentation de `sklearn`)

1.1 MLP sous sklearn

Dans le cas d'un échantillon d'apprentissage $S = \{(s_1, y_1), \cdots (s_m, y_m)\}$ où $x_i \in \mathbb{R}^N$ et $y_i \in \{0, 1\}$ (classification binaire), un MLP a une couche cachée apprend la fonction $f(x) = W_2 g(W_1^T x + b_1) + b_2$ où $W_1 \in \mathbb{R}^{n \times k}$, $W_2 \in \mathbb{R}^{k \times 1}\}$, et $b_1 \in \mathbb{R}^k, b_2 \in \mathbb{R}$, sont les paramètres appris du modèle (figure 1). La fonction d'activation $g(.) : \mathbb{R} \to \mathbb{R}$ est la fonction d'activation de la couche cachée, et la fonction logistique est prise par défaut en sortie. Par défault, g est la tangente hyperbolique :

$$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

— En classification binaire g est la fonction logistique $g(z) = 1/(1 + e^{-z})$ pour obtenir des valeurs entre 0 et 1 : un seuil à 0.5 permet de choisir la classe assignée pour chaque exemple.
— En apprentissage multi-classes ($y_i \in \mathbb{N}^k$), la sortie de $f(x)$ est un vecteur de classes : la fonction d'activation est alors la fonction *softmax* :

$$\text{softmax}(z)_i = \frac{e_i^z}{\sum_{c=1}^{k} e_c^z}$$

où z_i est le $i$ème élément de l'entrée de softmax, qui correspond à la classe i. On obtient ainsi un vecteur contenant les probabilités que x appartienne à chaque classe : la sortie finale est donc la classe qui maximise cette probabilité pour x.

— En régression, g reste la tangente hyperbolique, mais la fonction d'activation en sortie doit être la fonction identité.

Plusieurs fonctions de pertes (minimisées lors de l'apprentissage) sont proposées selon le problème :

— En classification, la fonction d'entroprie croisée est considérée,

$$\ell(\hat{y}, y, W) = -y\ln\hat{y} - (1-y)\ln(1-\hat{y}) + \alpha||W||_2^2$$

— En régression, c'est la fonction du carré d'erreur

$$\ell(\hat{y}, y, W) = \frac{1}{2}||\hat{y} - y||_2^2 + \alpha||W||_2^2$$

Dans les deux cas, le terme $\alpha||W||_2^2$ permet de régulariser la solution en pénalisant la complexité du modèle appris (avec l'importance α). Cela permet notamment d'éviter au mieux le sur-apprentissage.

1.2 La classe `MLPClassifier` : mise en pratique

Ce modèle optimise la fonction entropie croisée en utilisant le solveur LBFGS ou la descente de gradient stochastique. Ses paramètres principaux sont les suivants :

— `hidden_layer_sizes` est un tuple qui spécifie le nombre de neurones de chaque couche cachée, de l'entrée (non comprise) vers la sortie (non comprise). Par exemple, une couche cachée de 55 neurones, `hidden_layer_sizes`=(55) ; pour trois couches cachées de taille respectivement 50, 12 et 100 neurones, `hidden_layer_sizes`=(50,12,100).
— `activation` définit la fonction d'activation pour les couches cachées
— `solver` spécifie l'algorithme utilisé pour minimiser la fonction de perte en sortie
— `alpha` est la magnitude de la régularisation L2
— `max_iter` indique le nombre max d'itérations du solveur
— `tol` est un facteur de tolérance qui permet d'arrêter le solveur précocément lorsque qu'il n'y a pas d'amélioration d'au moins ce paramètre lors de deux itérations successives
— `learning_rate_init` permet de contrôler la taille du pas

Les fonctions usuelles d'apprentissage et de test de sklearn d'un modèle sont alors applicables (fit, predict, score, predict_proba, etc.). La composante `coefs_` du modèle contient les matrices de paramètres qui constituent le modèle, et `intercepts_` contient les poids des biais dans les couches cachées et de sortie.

1.2.1 Prise en main du modèle

1. Soit le jeu de données $S = \{([0,0],0),([0,1],1)\}$. Nous cherchons à apprendre un modèle $f(x) : \mathbb{R}^2 \to \{0,1\}$ qui, à chaque exemple, associe une sortie booléenne. Créez ce jeu de données (`X` et `y`) en utilisant les tableaux de numpy.
2. Créez un modèle MLP de classification avec une couche cachée de 4 neurones : dessinez ce modèle sur feuille, créez le avec `MLPClassifier`, puis apprenez le avec S, et testez le avec les entrées `[2.,2.]`, puis `[-1.,-2.]`. A l'aide des coefficients appris, complétez le dessin du MLP et vérifiez la sortie manuellement.
3. Complétez S en ajoutant les exemples $([1,1],0)$ et $([1,0],1)$. Il s'agit du problème du XOR. Apprenez les deux réseaux dont l'architecture a été présentée en cours avec des perceptrons linéaires à seuil. Les prédictions des réseaux obtenus sont-elles correctes ? Quels sont les coefficients et biais obtenus ?
4. Soit $S' = \{([0,0],[0,1]),([1,1],[1,1])\}$. Nous cherchons à apprendre un modèle $f(x) : \mathbb{R}^2 \to \{0,1\}^2$ qui, à chaque exemple, associe deux sorties booléennes distinctes. Au vu de S', quelles sont les deux fonctions booléennes que nous cherchons à apprendre ? Créez ce jeu de données (`X` et `y`) en utilisant les tableaux de numpy.

5. Déterminez une architecture MLP la plus simple possible pour l'apprentissage conjoint de ces deux fonctions (i.e. combien de neurones dans l'unique couche cachée ?). Le réseau correspondant devra être appris et testé sur S' lui-même.

   Quelle architecture obtenez-vous : sauvegardez ses paramètres dans un fichier. On utilisera pour cela les composantes du modèle que sont `n_layers_`, `n_outputs_`, `out_activation_`, `coefs_`, `intercepts_`, `classes_`, `loss_`. En combien d'itérations le solveur converge-t-il (cf. composante `n_iter_`) ?

1.2.2 Travail sur le jeu de données Iris

Nous avons déjà utilisé le jeu de données Iris lors de précédents TP. Nous utiliserons aussi la fonction `time()` du package `time` pour mesurer le temps d'apprentissage d'un modèle :

```
start_time = time.time()
-- Apprentissage
end_time = time.time()
print('temps ecoule = '+str(end_time - start_time)
```

1. Charger ce jeu de données. Toutes les expériences à venir devront être faites en train/test split [1].
2. Apprendre cinq modèles de classification des données Iris, avec des réseaux qui ont respectivement de 1 à 5 couches cachées, et des tailles de couches entre 10 et 300 au choix. Quelles sont les performances en taux de bonne classification et en temps d'apprentissage obtenus pour chaque modèle ?
3. Comparer les résultats avec une classifieur à noyau de type SVM avec noyaux polynomial : `clsvm = svm.SVC(kernel='poly')`.

Déterminez, pour chaque attribut d'entrées, sa valeur moyenne et l'écart-type de sa distribution.
Une astuce classique lorsque l'on programme des réseaux de neurones, est de précéder l'apprentissage du modèle par la normalisation des valeurs d'entrées. Le principe est de ramener toutes les entrées dans un même intervalle fermé, tout en respectant les distributions de probabilités de ces entrées. Ce processus de normalisation permet de considérer toutes ces valeurs à égale importance. Attention, la normalisation doit se faire autant sur les données d'apprentissage que sur les données de test.

1. Normaliser les données vers l'intervalle réel [0.,1.]. Cela peut se faire par quelques lignes de programmation, colonne par colonne de la matrice d'entrée, ou en utilisant la classe `StandardScaler` du package `preprocessing`. Voici un exemple d'utilisation issu de la documentation de sklearn :

   ```
   from sklearn.preprocessing import StandardScaler
   scaler = StandardScaler()
   scaler.fit(X_train)
   X_train = scaler.transform(X_train)
   X_test = scaler.transform(X_test)
   ```

2. Re-apprendre et tester les cinq modèles MLP et le SVM *après* avoir normalisé les données en entrées. Observez-vous des améliorations ?

1. Attention, avec la nouvelle version de sklearn, tout ce qui concerne la validation croisée, le train-test split du jeu de données, etc., se situe maintenant dans le package appelé `model_selection`.

Pour terminer, nous allons maintenant étudier la convergence des algorithmes d'optimisation disponibles : L-BFGS, SGD et Adam. L'algorithme SGD permet notamment d'adapter le pas d'apprentissage lors de la descente du gradient (paramètre ν dans la formulation mathématique). On retrouve cette particularité avec le solveur Adam. Dans ces deux méthodes, la donnée du pas initial peut alors s'avérer importante : `learning_rate_init`.
L'algorithme L-BFGS est, quant à lui, réputé pour converger rapidement vers une bonne solution quand le jeu de données est relativement petit.

1. Pour chacun des cinq modèles précédemment appris après normalisation, et pour chaque solveur disponible, indiquez : le temps de convergence, le nombre d'itérations pour converger, et les performances du modèle appris. Résumez ces informations au sein d'un tableau récapitulatif.
2. Choisissez le modèle qui propose de meilleurs résultats, et tentez d'améliorer ces résultats en faisant varier la magnitude de la régularisation L2 (paramètre α). Parvenez-vous à battre le SVM ? Si oui, avec quels hyper-paramètres finaux de votre MLP ?

1.3 Une fonction particulière

Réaliser un perceptron multi-couche permettant de détecter si un point (x, y) se trouve à l'intérieur ou à l'extérieur d'un triangle équilatéral centré en $(0, 0)$ et inscrit dans le cercle unité.

1. Choisir l'architecture du réseau (nombre d'unités, connections, fonctions d'activation, etc.).
2. Générer des exemples positifs (points à l'intérieur) et négatifs (points à l'extérieur).
3. Apprendre le modèle et tester sur d'autres points générés comme ci-dessus.

2 Travail sur le jeu de données MNIST

MNIST a été développé par les précurseurs du *deep learning*, Y. LeCun et Y. Bengio, en 1998. Il contient des données d'écriture manuelle des chiffres de 0 à 9. Il mène généralement à un problème de classification multi-classes à 10 classes. Dans sa forme initiale, l'échantillon d'apprentissage comporte 60000 exemples, et 10000 en test.

Un exemple en entrée est une image de taille fixe 28×28, chaque pixel étant blanc (0) ou noir (1). Le chiffre est centré dans l'image (fig. 2).

FIGURE 2 – Quelques exemples de chiffres manuscrits de MNIST. Un exemple est donc un vecteur de 28*28=784 composantes correspondant à un niveau de gris pour chacun des 784 pixels.

Il est possible de le charger sous `sklearn` avec l'instruction `mns = fetch_mldata("MNIST original")` du package `datasets`. Par la suite, les apprentissages se feront sur les 60000 premiers exemples, et les tests se feront sur les exemples restants dans l'échantillon chargé. Il s'agit du protocole utilisé par tous les chercheurs testant eurs algorithmes sur MNIST.

En faisant varier les différents hyper-paramètres du modèle, avec ou sans pre-processing, et en utilisant une GridSearch, parvenez-vous à battre les quelques références ci-après ? Indiquez le temps d'apprentissage du modèle et tous ses hyper-paramètres et paramètres.

Approche	Preprocessing	ACC	Ref
perceptron	aucun	12.0	LeCun 1998
perceptron	oui	8.4	LeCun 1998
MLP 1-HL-300	aucun	3.6	LeCun 1998
MLP 1-HL-300	oui	1.6	LeCun 1998
MLP 2-HL-500-300 cross-ent, softmax	aucun	1.53	Hinton 2005