TP 4 - Perceptron multicouche (MLP) en NumPy

Introduction à l'apprentissage profond
M2 informatique, Université Paris Cité, 2024-2025

L'objectif de ce mini-projet est de programmer un perceptron multicouche (MLP) en NumPy. Les données dans `tp4_data.pkl` contiennent `10 000` échantillons avec chacun `10` caractéristiques en entrée, et votre tâche est d'entraîner un MLP qui classe ces échantillons en `4` classes, en partant du code dans `tp4_starter.py`.

Instructions pour le rendu (IMPORTANT)

Votre rendu doit être une archive compressée nommée `TP4_NOM_PRENOM_NUMEROETUDIANT.tar.gz`, contenant **exactement trois fichiers** sans structure de répertoires :

- `tp4.py` : script qui entraîne un perceptron multicouche sur les données d'entrée.
- `tp4_presentation.ipynb` : notebook présentant vos travaux de manière lisible, incluant les résultats d'un entraînement et des visualisations (voir question 16).
- `README.md` : fichier expliquant comment utiliser votre programme, avec les dépendances nécessaires.

Le rendu se fera via Moodle : https://moodle.u-paris.fr/course/view.php?id=28594.

Date limite : lundi 3 février 2025, 23h59.

Notes importantes :

- Une partie significative de votre note sera basée sur la présentation dans le fichier `tp4_presentation.ipynb`. Bien que les suggestions pour ce fichier soient en question 16, nous vous conseillons à y prendre des notes au fur et à mesure.
- Ne pas inclure le fichier de données dans l'archive.
- Ne pas importer PyTorch ou d'autres frameworks de deep learning; l'objectif de ce TP est justement que vous reproduisez certaines fonctionnalités de PyTorch « à la main ».

Données

1. Chargez les données avec la fonction `load_data`.

(a) Vérifiez que vous avez deux tableaux NumPy : `X` de forme `(10000, 10)` et `y` de forme `(10000)`.

(b) Créez un graphique qui montre un nuage de points des deux premières coordonnées des données `X`, en visualisant les points de chaque classe avec une couleur différente.

(c) Divisez vos données en trois parties : un jeu `test` qui contient 10% des données, un jeu `validation` qui contient 20% des données, et un jeu `train` qui contient le reste des données.

Architecture du modèle

Le fichier `tp4_starter.py` définit une API simplifiée, inspirée de `pytorch.nn`. Le modèle est encapsulé dans une classe `MLPClassifier` avec trois attributs :
- `net` : le réseau.
- `lr` : le taux d'apprentissage.
- `num_classes` : le nombre de classes de sortie.

Chaque classe (dans ce TP, `ReLU`, `Linear`, `Dropout`, `Sequential` et `MLPClassifier`) possède trois fonctions :
- `forward` : passe des données à travers la couche.
- `backward` : propage le gradient en arrière.
- `step` : met à jour les paramètres.

Vous devez implémenter ces fonctions pour chaque couche.

Remarque. Dans PyTorch, l'API est un peu plus complexe : il y a une classe `optimizer` séparée, qui exécute la fonctionnalité `step`. Nous n'implémentons ici que la descente de gradient stochastique; on parlera de l'intérêt des optimizers plus tard dans le cours.

Couche ReLU

2. Implémentez `forward` et `backward` dans la classe `ReLU`. *Astuce :* Dans la passe forward, sauvegardez les indices où `x > 0` comme attribut. Réutilisez-les dans la passe backward.

3. Pourquoi `step` dans `ReLU` ne fait-il rien ?

4. Comme vu dans le TP 1, la dérivée partielle d'une fonction h peut être approximée ainsi :

$$\frac{\partial h(x, \mathbf{y})}{\partial x}(a, \mathbf{b}) \approx \frac{h(a+\epsilon, b) - h(a-\epsilon, b)}{2\epsilon}$$

   Implémentez `check_gradient` pour comparer le gradient analytique et celui calculé dans `backward`.

Couche Linéaire

5. Initialisez `W` (matrice des poids) et `b` (vecteur biais) avec des valeurs tirées d'une distribution normale avec variance $2/n$ où n est le nombre d'entrées de la couche. (On appelle cette méthode *Kaiming initialisation* ou *He initialisation.*)

6. Implémentez `forward` dans `Linear`. *Astuce :* Sauvegardez les entrées pour la passe backward.

7. (a) Implémentez `backward`, en calculant `grad_W`, `grad_b` et le gradient de l'entrée. (b) Utilisez `check_gradient` pour vérifier votre calcul.

8. Implémentez `step`.

Module Sequential

9. Implémentez `backward` pour la classe `Sequential` : appliquez `backward` à chaque couche en ordre inverse.

Couche Dropout

Une couche `Dropout` choisit aléatoirement une fraction des connexions du réseau à abandonner à chaque passe forward. La fraction de connexions à abandonner est stockée dans la variable `dropout`.

10. Implémentez `forward` dans `Dropout`, en sauvegardant les indices supprimés.

11. (a) Implémentez `backward` dans `Dropout`. (b) Vérifiez la correction de votre implémentation avec `check_gradient`. (c) Comme ReLU, `Dropout` ne fait rien dans `step`. Pourquoi pas ?

Entraînement et Validation

On vous donne la classe `MLPClassifier`, qui devrait maintenant être presque utilisable. Lisez sa définition. La seule fonction restante à définir est `one_hot`, qui est utilisée dans `backward` pour soustraire `1` à la probabilité de la bonne classe (voir aussi TP 2 et TP 3).

12. Implémentez `one_hot` : pour un vecteur `y` de `b` labels entre `0` et `num_classes - 1`, renvoyez une matrice `b × num_classes` avec des `1` aux indices des labels.

13. Lisez le code de la fonction `predict` dans `MLPClassifier`. Que fait `inference_mode` ?

14. Implémentez `validate` pour retourner le ratio de prédictions correctes.

15. Écrivez une boucle d'entraînement. Dans nos tests, nous avons atteint 85 % de précision sur le jeu de données test, et une perte moyenne $\approx$ `0.4`. (Cela après moins d'une minute d'entraînement sur un ordinateur portable acheté en 2019.)

Visualisation et présentation

16. Dans un notebook `tp4_presentation.ipynb`, présentez votre travail. Voici une liste non exhaustive d'idées que vous pourriez développer. Vous n'êtes pas obligé de tout inclure, et vous pouvez ajouter d'autres éléments qui vous semblent pertinents.

- Affichez la progression de l'entraînement de votre modèle, par exemple en affichant la perte, la précision sur les données de validation, et le temps pour chaque époque.
- Créez les courbes de perte et de la précision sur les données de validation de votre modèle.
- Commentez l'efficacité de votre modèle : combien de temps prend une époque d'entraînement ? Dans nos expériences, une époque prenait moins de 5 secondes. Si votre code est moins efficace, essayez de trouver le bottleneck dans votre programme, et vectorisez les opérations pour rendre l'entraînement plus efficace.

- Visualisez les frontières de décision de votre modèle : en partant du graphique des données créé à la question 1, visualisez les régions prédites par votre modèle. Vous pouvez utiliser la fonction `plot_decision_boundaries` fournie à la fin du fichier de départ.
- Rédigez un court texte (environ 100 mots) décrivant le modèle et ses hyperparamètres (taux d'apprentissage, nombre d'époques) : comment les avez-vous choisis ?