ML : Gradient descent

TP6 Apprentissage profond

Descente de gradient

Jérôme Buisine
jerome.buisine@univ-littoral.fr

8 décembre 2023

Durée : 3h

L'objectif de ce TP est la prise en main de la librairie `micrograd` pour la compréhension du mécanisme de propagation du gradient.

1 Ressources

Voici un ensemble de ressources qui peuvent vous être utiles durant ce TP :

— 1. Documentation officielle de l'outil `Git` ;
— 2. Gitlab : interface web pour la gestion de projets versionnés sous `Git` ;
— 3. pyenv : permet la gestion d'environnement Python.
— 4. matplotlib : librairie Python qui permet un affichage rapide de données.
— 5. jupyter : un outil de travail permettant une interaction rapide et visuelle avec une console Python.
— 6. micrograd : librairie deep learning minimaliste en Python.

2 Configuration de l'environnement

2.1 Installation des dépendances

Nous allons utiliser quelques librairies pour ce projet. Tout d'abord, mettez à jour `pip`. Puis ajoutez les dépendances suivantes dans un fichier nommé `requirements.txt` et installez-les :

```
numpy==1.23.2
pandas==1.4.4
matplotlib==3.5.3
jupyterlab==3.4.5
```

Créez un dossier à la racine de votre projet nommé `tp6-gradient_descent`. Dans ce dossier et depuis votre terminal, lancez la commandes suivantes (la première étant optionnelle) :

```
# (optionnel) spécifie l'environnement virtuel Python à Jupyter
ipython kernel install --user --name=ml-venv
# lance l'application Jupyter
jupyter-lab
```

3 Description de la librairie micrograd

La librairie micrograd permet de créer des réseaux de neurones MLP (Multi-Layer Perceptron) et d'utiliser le mécanisme propagation du gradient (backprogation). Pour cela, elle est composée de :

— `micrograd.engine.Value` : permet de stocker un scalaire et le gradient qui va lui être associé ;

— `micrograd.nn.Neuron` : spécifie le comportement d'un Neurone (nombre de poids associés au nombre de données d'entrée ainsi qu'un biais) ;

— `micrograd.nn.MLP` : permet de spécifier des réseaux de neurones multicouches. Un neurone de sortie est dit non-linéaire s'il est dans une couche intermédiaire (avec par défaut une fonction d'action de type `ReLU`). Le neurone de sortie est lui linéaire.

Le mécanisme de la chain rule est déjà intégré au sein de la librairie. Lorsqu'une erreur est calculée et propagée (méthode `backward` d'une instance de `Value`), chaque opération rencontrée permettra de calculer le gradient à chaque poids de notre modèle (autres scalaires de type `micrograd.engine.Value`). L'arbre des gradients, lié cette propagation est ensuite dressé.

Pour plus d'informations vous pouvez visualiser le code de propagation de l'erreur dans engine.py, ainsi que la manière dont le gradient peut-être calculé pour chaque opérateur d'un scalaire de type `Value`.

4 Compréhension du calcul du gradient

Créez un nouveau notebook nommé « gradient.ipynb ». Dans ce notebook nous allons prendre en main la librairie `micrograd`.

Tout d'abord, installez les dépendances suivantes qui seront utiles pour la suite du TP :

```
pip install micrograd graphviz scikit-learn
```

— La librairie `graphviz` nous permettra d'afficher le graphe des gradients ;

— La librairie `scikit-learn` sera utilisée pour générer des données synthétique.

Pour la suite du TP, proposez de fixer les graines aléatoires afin d'obtenir des résultats similaires à ceux illustrés dans celui-ci :

```
1 import numpy as np
2 import random
3 np.random.seed(42)
4 random.seed(42)
```

Tâche 1 : Proposez un premier modèle MLP, composé d'un seul Neurone qui lui prend deux données en entrée. Le modèle sera au final un perceptron. Afficher ensuite les paramètres initialisés de ce modèle.

Indications :

→ la classe MLP est présente dans le module `micrograd.nn` ;
→ Vous pouvez vérifier que le MLP propose une sortie similaire à un Neurone classique.

Vous devriez avoir un modèle composé de 3 paramètres :

```
MLP of [Layer of [LinearNeuron(2)]]
Model has: 3 parameters
Model parameters are: [
        Value(data=0.2788535969157675, grad=0),
        Value(data=-0.9499784895546661, grad=0),
        Value(data=0, grad=0)
]
```

Question : à quoi correspond le dernier paramètre de ce modèle ?

Tâche 2 : Définir la fonction d'activation sigmoïde définit telle que : $\sigma(x) = \frac{1}{1+e^{-x}}$

Indications :

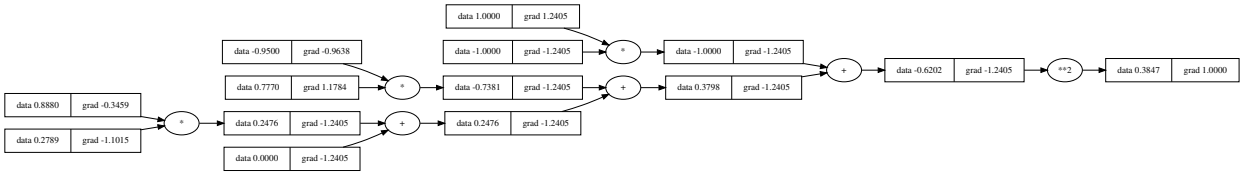
→ elle sera utilisée en sortie du modèle actuel (comme un post-traitement) ;
→ attention il faudra bien affecter la nouvelle valeur à l'attribut `data` de la sortie du modèle.

Tâche 3 : La librairie `micrograd` permet de visualiser l'arbre du gradient. Proposez des données d'entrée au modèle et calculer l'erreur quadrique par rapport à la sortie du modèle.

Indications :

→ proposez les données d'entrée suivante `[0.888, 0.777]` et une sortie attendue de `[1]` ;
→ la fonction sigmoïd sera ensuite utilisée en sortie ;
→ il faudra propager l'erreur calculée pour calculer le gradient ;
→ inspirez-vous du notebook proposé par l'auteur de la librairie pour afficher l'arbre des gradients.

Voici un aperçu d'un graphe proposant la manière dont le gradient est propagé :



Tâche 4 : Visualisez le graphe obtenu et proposez des interprétations quant à son contenu. Pouvez-vous expliquer comment est calculer le gradient ?

Indications :

→ toutes les opérations sont présentes (même la différence avec l'attendue) ;
→ il faudra bien identifier l'erreur de prédiction ;
→ il faudra bien calculer la dérivée de l'erreur utilisée ;
→ il faudra également cibler les poids associés aux neurones ;

Remarque : pour calculer la différence avec l'attendue en sortie, micrograd propose de multiplier par `-1` puis de réaliser la somme entre les deux scalaires.

Tâche 5 : Proposez maintenant la mise à jour des poids du neurone relativement aux gradients associés à ces poids.

Indications :

→ définir une fonction binaire qui permet de définir le label de sortie. La prédiction vaut 1 si la sortie est supérieure à 0.5, sinon elle vaudra 0 ;

→ avant de propager toute nouvelle valeur d'erreur (loss), il faudra réinitialiser à 0 les gradients de votre modèle.

Note : votre modèle doit maintenant être en capacité de prédire la sortie attendue pour les données proposées. Il est possible toutefois qu'il ait fallu propager le gradient plus d'une fois.

5 Modèle de séparation non-linéaire

Nous allons maintenant utiliser un modèle MLP avec une couche intermédiaire pour proposer un modèle de séparation plus précis sur un problème plus complet.

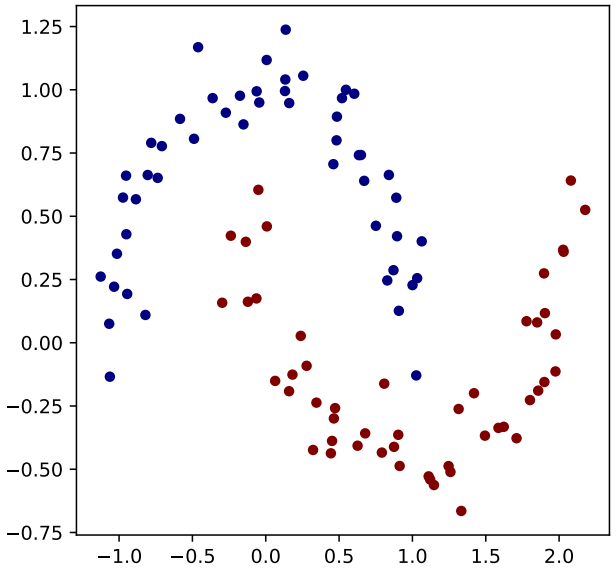
Tâche 6 : Générez des données synthétiques à l'aide de la fonction `make_moon` de la librairie `scikit-learn` disponible dans le module `datasets`.

Indications :

→ proposez 100 échantillons pour ce dataset ;

→ un bruit fixé à 0.1.

Vous devriez obtenir visuellement les données suivantes :



Tâche 7 : Proposez une fonction `model_loss` qui va calculer l'erreur moyenne que peut faire un modèle sur des données d'entrée.

Indications :

→ elle prendra 3 paramètres : le modèle, les descripteurs et les labels ;

→ une activation sigmoïde en sortie du modèle ;
→ une activation binaire pour connaître le label prédit ;
→ la fonction d'erreur calculée sera la MSE (l'erreur quadratique moyenne) ;
→ la méthode retournera l'erreur moyenne ainsi que le pourcentage de bonnes prédictions (précision) du modèle.

Tâche 8 : Créer un nouveau modèle MLP. Ce modèle sera composé de deux neurones à 2 entrées et un neurone de sortie. Visualisez de nouveau le graphe des gradients obtenu sur des données synthétiques. Que pouvez-vous remarquer ?

Tâche 9 : Nous allons maintenant entraîner ce modèle MLP sur les nouvelles données. La fonction `model_loss` précédemment créée nous permettra de connaître l'erreur du modèle et sa performance.

Indications :

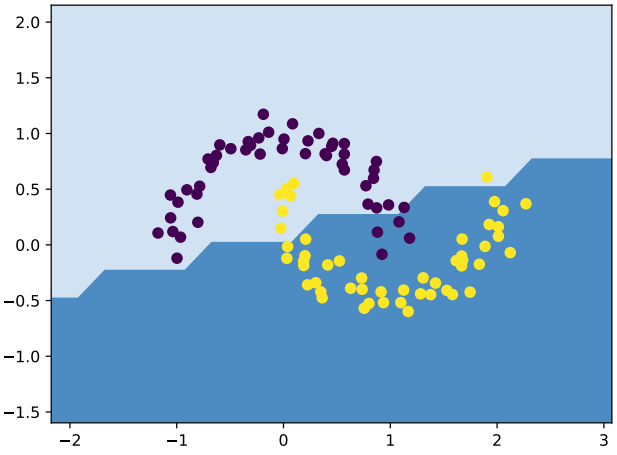
→ ce modèle sera composé de deux neurones à 2 entrées et un neurone de sortie ;
→ l'apprentissage sera fera sur 10 itérations ;
→ la mise à jour des poids sera réalisée avec un taux d'apprentissage dégressif relativement au nombre d'itérations. Le taux d'apprentissage sera initialisé à 1 et décroît à hauteur de 10% par itération ;
→ à chaque itération, la valeur moyenne d'erreur et la performance du modèle devront être affichée.

Note : le taux d'apprentissage est une variable importante qui permet de spécifier le comportement de la descente de gradient sur les poids du réseau. On parlera ici de descente de gradient stochastique.

Vous devriez obtenir un apprentissage proche de celui-ci :

```
[step 0] loss 0.24932434841760936, accuracy 50.0%
[step 1] loss 0.22098897218974045, accuracy 83.0%
[step 2] loss 0.1963744546993089, accuracy 86.0%
[step 3] loss 0.1736782052148665, accuracy 86.0%
[step 4] loss 0.1550638946177969, accuracy 84.0%
[step 5] loss 0.14091364956889652, accuracy 85.0%
[step 6] loss 0.12992480209746776, accuracy 84.0%
[step 7] loss 0.12150999835687959, accuracy 86.0%
[step 8] loss 0.11498508333360369, accuracy 87.0%
[step 9] loss 0.10970825565014697, accuracy 87.0%
```

Vous devriez obtenir une séparation du modèle proposé proche de celle-ci :



Tâche 10 : Proposez maintenant un nouveau modèle MLP avec 16 neurones cachés. Entraînez-le et observez l'évolution de ses performances.

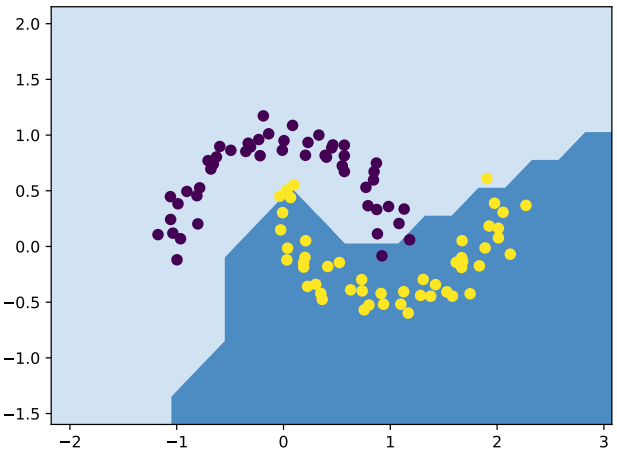
Indications :

→ cette fois-ci l'apprentissage sera fera sur 100 itérations ;

→ la mise à jour des poids sera réalisée avec un taux d'apprentissage dégressif relativement au nombre d'itérations. Le taux d'apprentissage sera initialisé à 1 et décroît à hauteur de 10% par itération ;

→ à chaque itération, la valeur moyenne d'erreur et la performance du modèle devront être affichée.

Vous devriez obtenir une séparation améliorer avec ce nouveau modèle proposé proche de celle-ci :



6 Remise des travaux

N'oubliez pas de réaliser un commit de vos travaux. Taguez également votre projet avec le tag « `tp6` » et soumettez-le sur le serveur Gitlab. Il fera office de rendu.