Piscine Machine Learning | Jour 6

Rétropropagation
du gradient

Nom de repo: ml_d06

Droits: florian.bacho@epitech.eu, voravo_d, girard_z

Langage : Python

Introduction:

Maintenant que vous savez faire une descente de gradient sur une couche de neurone, nous allons voir comment propager le gradient dans les couches cachées. On appelle cette algorithme: rétro-propagation du gradient (gradient backpropagation in english).

Reprenons la formule de mise à jour des paramètres:

$$w_{ij} = w_{ij} + \alpha \frac{\partial Loss(y_t, h_w(x_t))}{\partial w_{ij}}$$

Cependant, il peut être assez fastidieux de calculer les dérivées partielles des paramètres des neurones de chacune des couches. Afin d'éviter cela, nous pouvons utiliser la règle des dérivations en chaînes pour calculer les gradients d'une couche à partir des gradients de la couche suivante.

Petit rappel de la règle des dérivées en chaînes:

$$\frac{\partial f(x)}{\partial x} = \frac{\partial f(x)}{\partial g(x)} \frac{\partial g(x)}{\partial x}$$

où g(x) est un résultat intermédiaire de f(x).

On peut généraliser cette règle si l'on a plusieurs résultats intermédiaires $g_i(x)$:

$$\frac{\partial f(x)}{\partial x} = \sum_i \frac{\partial f(x)}{\partial g_i(x)} \frac{\partial g_i(x)}{\partial x}$$

Maintenant, revenons à la dérivée de la fonction de coût. Nous avons vu hier qu'elle peut s'écrire de cette manière:

$$\frac{\partial Loss}{\partial sp_j} = \frac{\partial Loss}{\partial h(sp_j)} \frac{\partial h(sp_j)}{\partial sp_j}$$

où sp_j est la somme pondérale des poids (entre le neurone j et le neurone i de la couche précédente) et des entrées avant le passage dans la fonction d'activation. La fonction h correspond à la fonction Logistic (pour simplifier la notation).

Ainsi, comme nous l'avons dit précédemment, les dérivées pour neurone d'une couche l peuvent être calculées à partir des dérivées des neurones connectés de la couche l + 1.

$$\frac{\partial Loss}{\partial sp_j} = \left(\sum_k \frac{\partial Loss}{\partial sp_k} \frac{\partial sp_k}{\partial h(sp_j)}\right) \frac{\partial h(sp_j)}{\partial sp_j}$$

on remplace par les dérivées que l'on connaît:

$$\frac{\partial Loss}{\partial sp_j} = \left(\sum_k \frac{\partial Loss}{\partial sp_k} w_{jk}\right) h(sp_j)(1 - h(sp_j))$$

ainsi, pour simplifier la notation:

$$\nabla_j = a_j(1 - a_j) \sum_k \nabla_k w_{jk}$$

où j est une neurone de couche cachée, a_j est l'activation du neurone j, k le $k^{ème}$ neurone de la couche l + 1 et w_{jk} le poid synaptique reliant j à k.

> ! Le calcul du gradient de la couche de sortie reste celui que vous avez vu hier

L'algorithme s'exécute donc en calculant les gradients des neurones de sortie puis en rétro-propageant le gradient sur les couches précédentes jusqu'à la couche d'entrée. (d'où le nom « rétro-propagation du gradient »)

Ensuite, il suffit d'appliquer la règle de mise à jour des poids:

$$w_i = w_i + \alpha \nabla x_i$$

Si vous n'avez pas compris toutes les étapes de dérivation, **ce n'est pas grave!** L'important est de retenir le résultat final, et de comprendre que l'on a pu calculer les dérivées d'une couche à partir des dérivées de la couche supérieure.

Ex00 :

Fichier: Neuron.py

Reprenez votre fichier Neuron.py pour y ajouter une fonction *calculateHiddenGradient* qui prend en paramètre la couche suivante et l'index du neurone courant dans sa couche. Cette fonction calcule le gradient du neurone pour une couche cachée.

Ex01 :

Fichier: NeuralNetwork.py

Reprenez votre fichier NeuralNetwork.py et appliquez les modifications nécéssaires pour exécuter une descente de gradient sur toutes les couches du réseau.

Ex02 :

Fichier: XorLearn.py

Un dataset à été fournit avec le sujet (xor.ds). Créez un script qui fait apprendre ce dataset à un réseau de neurone.

Une fois le seuil d'arrêt dépassé, affichez toutes les paires « Sortie attendue : Prédiction » sur la sortie standard. Exemple:

```
0 0 0 : 0
0 1 1 : 1
...
```

> **!** Vous devez afficher l'évolution du coût de votre réseau en temps réel sur la sortie standard et sur une courbe grâce à MatPlotLib.

Ex03 :

Fichier: TicTacToeLearn.py

Un dataset à été fournit avec le sujet (trainTicTacToe.ds). Créez un script qui fait apprendre ce dataset à un réseau de neurone.

Ce dataset représente des parties gagnées du Tic Tac Toe.

Les entrées:

- 0 lorsque les cases sont vides

- 0.5 pour le joueur O

- 1.0 pour le joueur X

La sortie attendue (sur 1 neurone) est:

- 0 si le joueur O a gagné

- 1 si le joueur X a gagné. Une fois le seuil d'arrêt dépassé, affichez toutes les paires « Sortie attendue : Prédiction » sur la sortie standard. Exemple:

1 : 1
1 : 1
0 : 0
0 : 1
…

> **!** Vous devez afficher l'évolution du coût de votre réseau en temps réel sur la sortie standard **Cependant**, n'affichez la courbe qu'une fois le seuil d'arrêt atteint afin de ne pas ralentir l'apprentissage. En effet, un nombre d'iteration important est nécéssaire pour ce dataset ;)

Conclusion :

Lorsque vous avez crée vos premiers neurones lors du jour 3, nous vous avions demandé d'initialiser aléatoirement leurs poids. Cette étape est indispensable au bon fonctionnement d'un réseau. En effet, si le poids de chaque neurone était initialisé à la même valeur, les neurones de chaque couche calculeraient les mêmes valeurs. Ces valeurs, utilisées durant la rétro-propagation pour calculer les gradients, auraient provoqué une modification identique de chaque poids. Ainsi, les poids n'auraient jamais pu se différencier, et notre programme n'aurait donc pas pu apprendre.

La rétro-propagation est une des notions les plus complexes en apprentissage supervisé. Si tout ne vous semble pas encore parfaitement clair, c'est normal. N'hésitez pas à chercher différents cours et vidéos afin de mieux comprendre le principe. Pour ceux qui se sont un peu découragés lors de cette journée/ces dernières journées, sachez qu'il n'est pas nécessaire de comprendre toutes les formules et démonstrations mathématiques afin de réaliser un projet fonctionnel. Cependant, avoir une bonne intuition de ce que vos neurones font et comment vos poids se mettent à jour vous permettra d'être beaucoup plus efficace. Vous détecterez plus rapidement ce qui ne va pas dans vos implémentations, comment les optimiser, et vous serez capables d'apprendre avec beaucoup plus de facilités des modèles plus poussés (réseaux de neurones récurrents, convolutifs, …).