Piscine Machine Learning | Jour 5

Descente de gradient
sur un neurone

Folder Name : ml_d05

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

Langage : Python

Introduction:

Vous savez à présent réaliser des neurones et ce qu'est une fonction de coût. Maintenant, nous devons faire apprendre nos neurones !

Pour ce faire, nous allons voir l'algorithme de descente de gradient appliquée à un neurone.

Reprenons notre formule d'activation du neurone:

Soit x le vecteur d'entrée et w le vecteur de poids (biais compris)

$$h_w(x) = Logistic(\sum_{i=1}^{k} w_i x_i + w_0)$$

et notre fonctions de coût (cross-entropy):

Soit x le vecteur d'entrée, w le vecteur de poids (biais compris), y le vecteur de sortie attendue, i le ième neurone de sortie et $h_w(x)$ la sortie calculée du réseau de neurone:

$$Loss(x, y, w) = -\sum_{i=0}^{k} (y_i \log(h_{wi}(x)) + (1 - y_i) \log(1 - h_{wi}(x)))$$

Hier, nous avons parlé de gradient. Le gradient est le vecteur qui indique dans quelle direction déplacer nos poids synaptiques afin de minimiser l'erreur. Le gradient est représenté par l'ensemble des dérivée partielles de notre fonction de coût par rapport à chacun des poids.

Ainsi le gradient d'un neurone à deux entrées (+ 1 biais) est noté:

$$\nabla = [\frac{\partial Loss(x, y, w)}{\partial w_0}, \frac{\partial Loss(x, y, w)}{\partial w_1}, \frac{\partial Loss(x, y, w)}{\partial w_2}]$$

Cependant, il n'est pas nécessaire de calculer la dérivée partielle de chaque poids car nous pouvons généraliser la dérivée pour tous les poids grâce aux dérivées en chaînes.

Commençons par la dérivée de la somme pondérée entre les poids synaptiques et les entrées:

$$\frac{\partial sp}{\partial w_i} = \frac{\partial(\sum_{t=0}^{k}(w_i x_i) + w_0)}{\partial w_i}$$

$$\frac{\partial sp}{\partial w_i} = x_i \quad \text{pour i > 0 et} \quad \frac{\partial sp}{\partial w_i} = 1 \quad \text{pour i = 0}$$

Etant donnée que l'on dérive par rapport à w_i, toutes les autres pondérations sont considérées comme des constantes et ne sont pas présentent dans la dérivée. Ainsi, la dérivée de $w_i * x_i$ par rapport à w_i est x_i.

Passons à la dérivée de la fonction d'activation par rapport à w_i. Par l'application des dérivées en chaînes:

$$\frac{\partial Logistic(sp)}{\partial w_i} = \frac{\partial Logistic(sp)}{\partial sp}\frac{\partial sp}{\partial w_i}$$

or:

$$\frac{d Logistic(x)}{d x} = Logistic(x)(1 - Logistic(x))$$

donc:

$$\frac{\partial Logistic(sp)}{\partial w_i} = Logistic(sp)(1 - Logistic(sp))\, x_i$$

Et enfin, toujours par l'application des dérivées en chaîne:

$$\frac{\partial Loss(x, y, w)}{\partial w_i} = \frac{\partial Loss(x, y, w)}{\partial Logistic(sp)}\frac{\partial Logistic(sp)}{\partial sp}\frac{\partial sp}{\partial w_i}$$

donc, après plusieurs simplifications:

$$\frac{\partial Loss(x, y, w)}{\partial w_i} = (y - Logistic(sp))\, x_i \quad \text{pour i > 0}$$

$$\frac{\partial Loss(x, y, w)}{\partial w_i} = (y - Logistic(sp)) \quad \text{pour i = 0}$$

Pas trop dur ? ;)

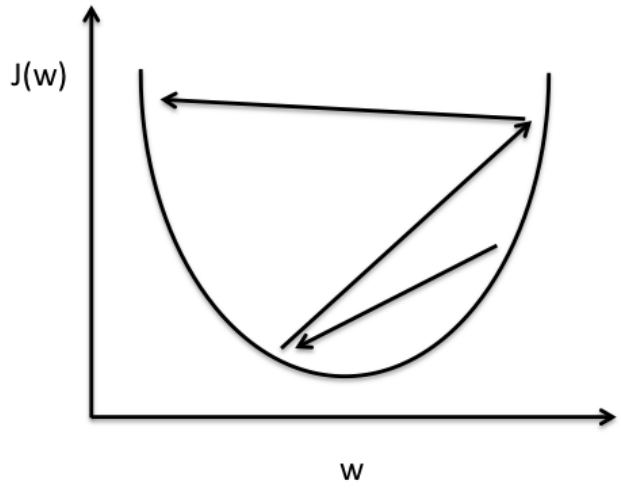
Si vous n'avez pas compris toutes les étapes de dérivation menant à ce résultat, **ce n'est pas grave!** (Mais si vous êtes curieux n'hésitez pas à chercher des explications plus détaillées).

Retenez simplement la formule finale en gardant en tête que c'est la dérivée de la fonction de coût par rapport au paramètre w_i .

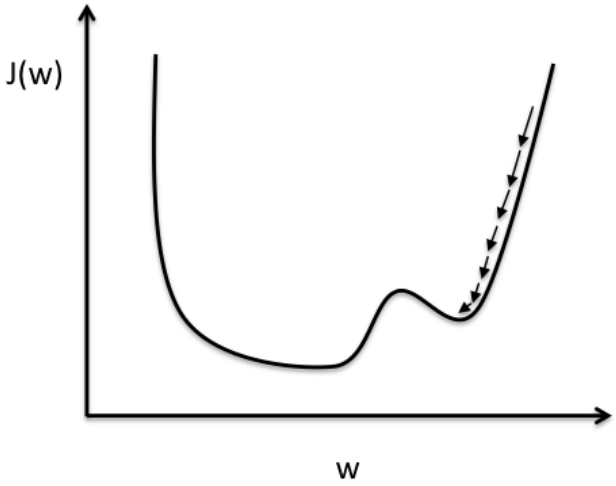
Maintenant que nous avons notre gradient calculé il suffit de l'appliquer à chacun de nos poids synaptiques avec un taux d'apprentissage (learning rate) que l'on note α compris entre 0 et 1:

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

Ce taux représente à quelle vitesse les poids sont ajustés. Il n'y a pas de règle universelle pour le choix de sa valeur. Un taux trop élevé fera diverger l'apprentissage, tandis qu'un taux trop faible mettra trop de temps à converger et augmente les risques de tomber dans un minimum local de notre fonction de coût (voir image ci-dessous). A vous de tester et de trouver un taux d'apprentissage α approprié lors de vos projets.



Large learning rate: Overshooting.



Small learning rate: Many iterations until convergence and trapping in local minima.

Pour éviter de calculer le gradient à chaque fois, on peut isoler ∇ pour le neurone:

$$\nabla = (y - Logistic(sp))$$

et mettre à jour chaque poids de cette manière:

$w_i = w_i + \alpha \nabla$ pour i = 0

$w_i = w_i + \alpha \nabla x_i$ pour i > 0

Il existe plusieurs types de descente de gradient:

- L'iteratif : on itère sur tous les exemples dans l'ordre en faisant une descente de gradient pour chacun. Cette methode fonctionne bien pour des petits dataset

- Le « batch gradient »: On fait la moyenne des gradients sur tous les exemples et on descend ce gradient moyen calculé.

- La descente de gradient stochastique: On choisit un exemple au hasard dans le dataset et exécute une descente de gradient avec. Cette méthode fonctionne bien sur des grands dataset.

- Le « batch mini-lot descent gradient » : Un mélange des deux methodes précédentes. On choisit un lot d'exemple dans le dataset et on fait la moyenne des gradients sur ce lot.

Dans tous les cas, votre descente de gradient sera une boucle dans laquelle vous calculez un gradient utilisé pour ajuster vos poids. La condition d'arrêt de cette boucle peut par exemple être un certain nombre d'itérations ou lorsque le coût passe en dessous d'un certain seuil.

Pour des questions de simplicité, nous vous conseillons de vous concentrer sur les methodes iteratives et stochastiques.

Ex00 :

Fichier: Neuron.py

Ajoutez à la class Neuron une fonction *calculateOutputGradient* qui calcul le gradient du neurone en fonction d'une sortie désirée. Considérez que le neurone à déjà été activé avant l'appel de cette fonction

> **!** Vous verrez demain que le calcul du gradient pour les couches cachées est différent

Ex01 :

Fichier: Neuron.py

Ajoutez à la class Neuron une fonction *applyGradient* qui applique le gradient du neurone à tous les poids synaptiques, qui prend en paramètre le taux d'apprentissage et les entrées.

Ex02 :

Fichier: OrLearn.py

Un dataset a été fourni (or.ds). Créez un script qui fait apprendre ce dataset à un neurone.

Une fois votre seuil d'arrêt dépassé, affichez les prédictions de votre réseau sur le set d'entraînement. Exemple :

0 0 0 : 0

0 1 1 : 1

…

> **!** Vous devez afficher l'évolution du coût de votre neurone en temps réel sur la sortie standard et sur une courbe grâce à MatPlotLib. Inutile de le faire à chaque itérations afin de ne pas trop ralentir l'apprentissage choisissez un nombre judicieux afin d'observer l'évolution de l'apprentissage.

Ex03 :

Fichier: BinaryLearn.py

Un dataset a été fourni (trainBinary.ds). Les entrées des exemples représentent des images de 0 et de 1 codées sur 64 pixels (8 x 8) en niveau de gris (entre 0 et 1).

Créez un script qui fait apprendre un neurone à reconnaître une image de 0 d'une image de 1.

Un deuxième dataset est fourni (testBinary.ds). C'est avec ce dataset que vous devez tester le coût de votre neurone une fois l'apprentissage terminé.
Une fois le seuil d'arrêt dépassé, affichez toutes les paires « Sortie attendue : Prédiction » sur la sortie standard. Exemple:
1 : 1
0 : 0
0 : 1
0 : 0
…

Training set/test set:

Ce dataset, différent du dataset d'entraînement, est appelé le test set. Il permet de tester la bonne généralisation de votre neurone. En effet, même si votre neurone réussit à avoir un coût faible sur le set d'entraînement (training set), cela ne garantit pas qu'il a bien généralisé le problème, et qu'il fonctionnera correctement sur des exemples qu'il n'a jamais rencontré lors de son apprentissage.

Afin de détecter une éventuelle mauvaise généralisation (également appelée trop haute « variance »), nous séparons donc nos données en 2: le training set et le test set. Un bon agent doit avoir à la fois un coût faible sur son training set et sur son test set.

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

Ex04 :

Fichier: NeuralNetwork.py

Reprenez votre fichier NeuralNetwork.py pour y ajouter une fonction *calcLoss* qui prend en paramètre un dataset et calcule la moyenne des coûts des neurones de sortie.

Ex05 :

Fichier: NeuralNetwork.py

Ajoutez une fonction *calculateGradients* à la class NeuralNetwork qui calcule le gradient de tous les neurones de sortie. Cette fonction prend en paramètre les sorties désirées.

> **!** Vous verrez demain comment calculer le gradient des couches cachées

Ex06 :

Fichier: Neuron.py

Ajoutez une fonction *applyGradientWithLayer* à la class Neuron qui prend en paramètres le taux d'apprentissage et une list de neurones correspondant aux entrées du neurone. Cette fonction applique le gradient sur les poids synaptiques.

Ex07 :

Fichier: NeuralNetwork.py

Ajoutez une fonction applyGradients à la class NeuralNetwork qui applique le gradient de tous les neurones. Cette fonction prend en paramètre le taux d'apprentissage.

Ex8 :

Fichier: DigitsLearn.py

Un dataset a été fourni (trainDigits.ds). Les entrées des exemples représentent des images de chiffres manuscrits (0 à 10) codées sur 64 pixels (8 x 8) en niveau de gris (entre 0 et 1). Il y a 10 sorties par exemple, chacun représentant la probabilité que l'image soit un chiffre.

Créez un script qui fait apprendre un réseau de neurones à reconnaître un chiffre à partir d'image de chiffre manuscrit.

Encore une fois, un deuxième dataset est fourni (testDigits.ds). C'est avec ce dataset que vous devez tester le coût de votre réseau de neurones.
Une fois le seuil d'arrêt dépassé, affichez toutes les paires « Sortie attendue : Prédiction » sur la sortie standard. Exemple:
3 : 3
6 : 0
8 : 8
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.