AS TME 1 : Rapport Types de Gradient

Seurin Mathieu

September 26, 2016

Article

But de la descente de gradient : Optimiser une fonction, dans le cadre des réseaux de neurones, minimiser la fonction de coût, qui définit à quel point le modèle est proche des données ou non.

Pour cela, la descente de gradient est un processus itératif qui va chercher à chaque étape à diminuer un l'erreur commise par le modèle, jusqu'à l'atteinte du critère d'arrêt (convergence ou un certain nombre d'itérations)

Pour contrôler l'effet de la mis à jour, on utilise un paramètre η (learning rate) qui va définir si on effectue de petites mises à jour du gradient à chaque fois (convergence lente, mais sur) ou au contraire de grosses mises à jour (convergence plus rapide, mais pas sur, risque de divergence)

La différence entre les trois types de descente de gradient tient juste à la quantité de données que l'on utilise pour calculer l'erreur.

1 Batch gradient descent

Pour le batch gradient descent, on utilise toute la base disponible pour calculer l'erreur et ajuster les paramètres du modèle. Le problème est qu'il faut que l'ensemble de la base tiennent en mémoire vive et chaque itération est longue et couteuse. De plus, on ne peut l'utiliser pour de l'apprentissage *online*.

2 Stochastic gradient descent

Cette version, au lieu d'utiliser toute la base, utilise seulement un exemple pour mettre à jour le gradient. L'avantage est que chaque itération est très rapide et on peut apprendre online. Par contre, cela implique de fortes variations au niveau de la fonction de coût, ceci peut être vu comme un avantage car il peut ainsi atteindre des minimums locaux potentiellement meilleurs que ce dans lesquels l'algorithme est placé à l'initialisation.

3 Mini-batch gradient descent

Enfin, cette version hérite des deux premières, au lieu de prendre toute la base ou un seul exemple, on va prendre N (compris généralement entre 50 et 256) exemples. Ceci permet de réduire la variance de la version stochastique et permet d'aller plus vite que la version batch.

Les deux principaux problèmes posés sont :

1. Comment choisir le learning rate et quels sont les raffinements possibles ? (learning rate adaptatif etc...)

2. Les problèmes sont souvent hautement non-convexes, comment éviter de rester bloquer dans des minimus locaux ?

4 Raffinement de la descente de gradient

4.1 Ajouter de l'inertie

Le principe pour ajouter de l'inertie est de tenir compte de la mise à jour précédente. Si la mise jour de paramètres juste avant était très importante, la suivante va conserver 'l'inertie' et être également relativement importante. Cela ajoute de la stabilité, et accélère le processus de convergence. (On peut imaginer une balle qui descend une colline, et qui gagne de la vitesse au fur et à mesure, cela va plus vite que faire des petits bonds simples à chaque itération)

4.2 Nesterov accelerated gradient

L'inertie est une bonne chose lorsque l'on est en phase de descente, seulement, quand on arrive à l'optimum, il ne faut pas que le gradient ai trop d'inertie et 'remonte' la pente (risque de divergence)

La solution est de, au lieu de faire la mise à jour selon le gradient actuel, d'utiliser celui ci pour estimer les prochaines valeurs du vecteur de paramètre θ. Ainsi, le gradient peut anticiper la prochaine valeur et donc s'adapter si la pente se réduit voir remonte.

5 Adagrad

Un autre problème de la descente de gradient : Pour les données où il y a beaucoup de différences de fréquences des paramètres (dans le texte par exemple) et que la plupart des paramètres sont à zéro, les paramètres rares seront très rarement mis à jour, et quand ils le seront, ce sera autant que les paramètres fréquents. On aimerait que les paramètres rares aient des mises à jour plus importantes et à l'inverse les paramètres fréquents aient des mises à jour plus petites.

C'est ce que Adagrad a pour objectif. Il va choisir différents learning rate η pour chaque θ_i. Pour cela, il réduit η_i à chaque fois que le paramètre associé θ_i est mis à jour. Donc si le paramètre est mis à jour très souvent, le learning rate sera de plus en plus petit, à l'inverse, quand il est peu modifié, le learning rate sera important et donc la mise à jour aussi.

6 Adadelta

Adadelta a pour but de fixer un défaut de Adagrad, le fait que le learning rate, à partir d'un certain nombre d'itérations, est tellement petit qu'il ne fait quasiment plus de mise à jour et donc n'apprend plus.

Pour cela, au lieu de diminuer le learning rate à chaque itération, depuis l'itération 1, Adagrad tient seulement compte des W dernières mises à jour

pour ajuster le learning rate, ainsi cela permet de continuer l'apprentissage sur des nouveaux exemples, tout en tenant compte de la fréquence des paramètres θ

De plus, l'avantage d'Adadelta est que l'on a pas besoin de définir un η par défaut.

Les algorithmes RMSprop et Adam sont très similaires à l'Adadelta, mais cherche à réduire encore la diminution très importantes du learning rate.