Classification supervisée

Rémy Degenne

9 février 2022

Ces notes de cours contiennent uniquement un bref résumé de ce qui a été dit en cours. Reportez vous à vos notes pour plus de détails.

1 Cours 4 : réseaux de neurones convolutifs

1.1 Entrainer un réseau de neurones

Cette section liste des questions et problèmes courants qui se posent quand on veut entrainer un réseau de neurones.

Combien de neurones? Considérons le cas d'un perceptron multi-couches à une couche cachée. Cette couche contient un certain nombre de neurones, qu'il faut choisir.

Si on a trop peu de neurones, le réseau se rapproche d'un modèle linéaire, qui ne peut pas séparer plus de deux classes, et ne pourra pas classer correctement des données qui ne sont pas linéairement séparables. (voir les dessins faits en classe)

Conclusion: une couche petite réalise une compression de l'information. Si on a trop peu de neurones, on perd trop d'information pour pouvoir encore classer les données.

Est-ce qu'on peut avoir trop de neurones?

Théorème: un réseau de neurone à une couche cachée peut approximer n'importe quelle fonction avec une précision aussi bonne qu'on veut, si la couche contient assez de neurones.

Est-ce que ça veut dire qu'il suffit d'ajouter des neurones pour avoir un meilleur classeur? Pas vraiment. Ce théorème nous parle de l'erreur d'entrainement: si on a assez de neurones et qu'on trouve de bons poids, on peut avoir une erreur d'entrainement aussi petite qu'on veut. Mais on est intéressé par l'erreur de test, pas l'erreur d'entrainement. Trop de neurones conduit à du sur-apprentissage.

Autre problème: plus on a de neurones, plus on a de poids à apprendre, et plus l'apprentissage est long (en nombre d'époques). Le calcul du gradient et la mise à jour des poids sont également plus coûteux (en temps de calcul).

Minimum local et minimum global Rappel: la descente de gradient ne trouve qu'un minimum local. En pratique, on a observé que tous les minimums locaux des réseaux de neurones ont des valeurs proches. C'est à dire que n'importe quel minimum local est proche d'un minimum global. C'est une observation empirique. On a des pistes théoriques pour commencer à l'expliquer dans certains cas particuliers, mais ce n'est pas encore très clair.

Eviter le sur-apprentissage Comme la majorité des classeurs, un réseau de neurones peut sur-apprendre. On a plusieurs méthodes pour limiter le sur-apprentissage. Une première, qui n'est pas spécifique aux réseaux, est de stopper l'apprentissage plus tôt grâce à la validation.

Méthode 1: régulariser les poids. Inclure dans la perte un terme proportionnel à $\|w\|$, avec un coefficients de proportionnalité α. Encourage des choix plus "simples", où l'algorithme ne compte pas sur une entrée particulière mais est obligé de tout prendre en compte. Les poids obtenus généralisent mieux. Attention: si α est trop grand, le modèle devient trop simple (presque linéaire).

Méthode 2: "dropout". Supprimer aléatoirement des neurones à chaque épisode de l'apprentissage. Oblige le réseau à ne pas compter que sur le résultat d'un neurone particulier (sur un attribut particulier d'une donnée) mais à utiliser toutes les parties du réseau et des données.

1.2 Réseaux de neurones convolutifs

Tâche considérée: reconnaître des objets sur des images.

Quand on utilise une image en entrée d'un perceptron multi-couches, les relations spatiales entre les pixels ne sont pas conservées. Un tel réseau ne sait pas que deux pixels sont côte à côte dans l'image, il considère chaque pixel comme une entrée indépendante. Un humain ne pourrait jamais reconnaître un chat sur une photo si on lui présentait juste un ensemble désordonné de pixels!

On veut un réseau de neurones qui utilise la *structure* des données.

Autre point important: un chat est un chat où qu'il se trouve sur l'image. Ce qu'on veut détecter est invariant par translation. On veut utiliser un réseau de neurones qui a également cette propriété.

Convolution:

$$(f \star g)(n) = \sum_{m=-\infty}^{+\infty} f(n-m)g(m) \, .$$

En 2D, si g n'est positive qu'entre $-M$ et M sur les deux axes,

$$(f \star g)(n) = \sum_{k=-M}^{M} \sum_{m=-M}^{+M} f(n-k, n-m)g(k, m) \, .$$

Voir dessin fait en cours (ou Wikipedia, par exemple).

Un neurone dans une sous-couche de convolution partage ses poids avec tous les autres neurones de la même sous-couche. Chaque "patch" de taille $M \times M$ de

l'image est transformé en une sortie. On obtient une nouvelle image constituée des sorties de tous les neurones de la sous-couche. La structure spatiale est conservée et comme les poids sont les même partout, l'invariance par translation est assurée.

Autre avantage: beaucoup moins de poids!

Autre avantage: tous les calculs effectués par les neurones d'une même sous-couche sont les mêmes (mais sur des bouts de l'image différents). C'est un type de calcul qu'on peut réaliser efficacement avec des GPU (cartes graphiques).

On a défini une couche convolutive. On peut les empiler pour obtenir un réseau de neurones convolutif.

- On peut ne calculer que certains patchs pour réduire la dimension de la sortie.
- On peut, après la couche de convolution, utiliser une couche de "pooling".
- La dernière couche du réseau sera une couche non-convolutive.
- Souvent en pratique: des couches avec des images de plus en plus petites, mais de plus en plus de sous-couches (appelées canaux).

2 Cours 5

2.1 Remarques sur le TP 4

Régularisation Pas assez de régularisation: bonne erreur d'entrainement mais la frontière de décision est très sensible aux exemples et l'erreur de test est moins bonne.

Trop de régularisation: tend vers une frontière linéaire. Grosse erreur.

C'est un hyper-paramètre. On peut l'optimiser en utilisant la validation.

Observer les poids d'un neurone On a une image en entrée et chaque neurone a un poids par pixel. On peut arranger ces poids dans une matrice et l'observer comme une image en noir et blanc. On observe que certains neurones sont sensibles à certaines parties de l'image et pas d'autres. Par exemple, des alternances de bandes noires et blanches montrent une sensibilité à un bord.

2.2 Structure d'un réseau convolutif

Couche convolutive

- Entrée: plusieurs images de même taille
- Sortie: un nombre différent d'images. Chaque image est le résultat de l'application d'un noyau de convolution aux images d'entrée.
- On applique ensuite une fonction d'activation (ReLU, sigmoide...). La fonction la plus utilisée est ReLU.

Voir ici: `http://d2l.ai/chapter_convolutional-neural-networks/conv-layer.html`

Padding: technique qui consiste à ajouter de faux pixels autour de l'image d'entrée pour obtenir la même taille d'image en sortie.

Stride: pour réduire la taille de l'image en sortie et faire moins de calculs, on peut ne calculer la convolution que tous les n pixels (stride de n).

Voir ici: `http://d2l.ai/chapter_convolutional-neural-networks/padding-and-strides.html`.

Couche de "pooling" Cette couche réalise une agrégation de valeurs proches. But: diminuer la taille de l'image en essayant de conserver les infos.

L'agrégation peut être faite de différentes manières. La plus courante est de calculer un maximum.

Pour une image d'entrée de taille $2n * 2m$ et un pooling de taille 2×2, on obtient en sortie une image de taille $n \times m$. Appelons P_e la matrice de pixels de l'image d'entrée et P_s celle de l'image de sortie. Pour chaque $i, j \in \{1, \ldots, n\} \times \{1, \ldots, m\}$, $P_s(i, j) = \max\{P_e(2i, 2j), P_e(2i+1, 2j), P_e(2i, 2j+1), P_e(2i+1, 2j+1)\}$.

Voir ici: `http://d2l.ai/chapter_convolutional-neural-networks/pooling.html`

Couche dense C'est le type de couche d'un perceptron multi-couches. Un neurone est connecté à toutes les sorties de la couche précédente.

Ajouter des couches convolutives après une couche dense n'a pas de sens. On a perdu toute information spatiale.

Un réseau peut se décomposer en deux parties: une première partie convolutive construit une représentation des données, puis une deuxième partie dense (un MLP) classe les données à partir de cette représentation.

En pratique: on peut utiliser la même représentation d'images pour plein de tâches. Entrainer une représentation est complexe $\rightarrow$ on ne le fait qu'une fois. On ne réentraine que le MLP qui se trouve en fin de réseau.

Exemples de réseaux: `http://d2l.ai/chapter_convolutional-neural-networks/lenet.html`

2.3 Pourquoi des cartes graphiques?

Entrainer un réseau de neurones nécessite beaucoup de données et beaucoup de capacité de calcul. Un ordinateur typique contient deux types de composants capables d'effectuer des calculs:

- processeur (CPU): quelques coeurs, très rapides et pouvant calculer n'importe quoi.
- carte graphique (GPU): plein de coeurs, moins rapides et plus spécialisés. Utilisation idéale: faire la même opération sur des données numériques différentes, en parallèle. C'est exactement ce qu'on veut faire dans une couche convolutive !

Les grandes avancées des 20 dernières années dans la puissance des GPU ont permis aux réseaux de neurones d'être utilisés. Un gros réseau nécessite beaucoup de calculs, ce qui consomme beaucoup d'énergie et a un fort impact environnemental. Des chercheurs essaient de réduire la taille des réseaux tout en conservant les performances.

2.4 Augmentation des donnees

Constituer un jeu de données pour l'apprentissage nécessite un important effort humain pour donner une étiquette à chaque donnée. Les personnes qui effectuent ce travail sont souvent peu payés.

Afin d'obtenir un ensemble d'exemples plus gros pour l'apprentissage, on peut essayer de démultiplier les données qu'on a à disposition:

- inverser une image donne une autre image avec la même étiquette.
- zoomer sur une image peut avoir le même effet (en fonction de la tâche de classification que l'on essaie d'apprendre).
- changer légèrement la couleur de l'image ne change pas les objets qu'elle représente.

2.5 Optimiser les hyper-paramètres

"La validation permet de choisir les hyper-paramètres". Oui mais si on a beaucoup d'hyper-paramètres c'est très couteux ! Si on a 8 hyper-paramètres, chacun avec 10 possibilités, on doit essayer 10^8 combinaisons. Si entrainer un réseau prend une minute, l'entrainement de tous ces réseaux les uns après les autres prendra 190 ans.

Comme on n'aura pas le temps d'essayer toutes les possibilités, on a besoin d'une stratégie de recherche qui nous donne aussi vite que possible de bonnes valeurs.

Méthode 1: recherche sur une grille. C'est la méthode naïve qui essaye toutes les combinaisons une par une. Si on a deux hyper-paramètres a et b avec des valeurs $\{a_1, \ldots, a_n\}$, $\{b_1, \ldots, b_m\}$, on essaie d'abord (a_1, b_1), puis (a_1, b_2), (a_1, b_3), ..., (a_1, b_m), (a_2, b_1), (a_2, b_2)...

Problème de cette méthode: imaginons que la valeur de a change beaucoup l'erreur de validation, alors que la valeur de b ne la change presque pas. Alors on perd m essais de valeurs de b pour chaque valeur de a. Le paramètre a est exploré très lentement.

Méthode 2: recherche aléatoire. A chaque étape, on prend une valeur aléatoire parmi les valeurs de la grille pour chaque hyper-paramètre. L'avantage est qu'on explore tous les paramètres en même temps. C'est une bonne méthode en pratique.

Méthode 3 (et plus): des algorithmes existent qui essayent par exemple d'abord des valeurs aléatoires, puis cherchent "autour" des valeurs pour lesquelles les résultats observés étaient bons.

Remarque: pour presque toutes ces méthodes, il faut d'abord définir une liste de valeurs possibles pour chaque hyper-paramètre. Pour certains paramètres on peut choisir des valeurs en progression arithmétique, comme $(11, 12, 13, 14)$. Mais l'erreur peut être peu sensible à certains paramètres, tant que leur ordre de grandeur reste le même (par exemple pour le pas de gradient). Dans ce cas on préférera une grille logarithmique (exemple: $(1, 10, 100, 1000)$).