Note du cours 5 de ML : Réseau de neurones

Nicolas Baskiotis
MLIA, LIP6, Sorbonne Université

5 mai 2020

Résumé

Ces notes ne sont malheureusement pas un cours et ne peuvent remplacer complétement le cours magistral. Elles ont été écrites très rapidement, elles sont destinées à vous guider autant que possible dans la lecture des slides en soulignant le message important de chacun. J'ai essayé de donner des références pour tous les points techniques, n'hesitez pas à vous y reporter pour affiner votre compréhension.

1 Résumé des épisodes

Slide 2 Le problème de l'apprentissage supervisé tel que vue dans les cours précédents est formalisé comme la recherche dans un espace de fonctions du minimum d'une fonction de coût donné. Par exemple, dans le cas des modèles linéaires, l'espace des fonctions est l'ensemble des fonctions linéaires paramétré par le vecteur de poids $\mathbf{w}$ (un poids pour chaque dimension), et la fonction de coût peut être les moindres carrés, une fonction logistique ou le coût hinge (généralisation du coût perceptron). La recherche du minimum se fait par descente de gradient sur le vecteur $\mathbf{w}$. On recherche bien sûr la fonction qui minimise le coût sur l'ensemble d'apprentissage (principe de la minimisation empirique du risque).

Slide 3 Les fonctions linéaires ne sont pas très expressives. Pour résoudre des problèmes plus complexes, on a vu dans le cours précédent les approches par noyaux : il s'agit de modifier les données d'entrées en utilisant un noyau qui permet d'avoir une représentation dans un espace de plus grande dimension (noyau polynomial, noyau gaussien, . . .).

Les réseaux de neurones proposent eux de combiner par couche des modèles "simples" - des perceptron, chacun étant un neurone - pour aboutir à des features plus expressifs. Il s'agit là aussi (historiquement du moins) d'un processus de transformation des features mais itératif (par couche) et surtout la transformation est apprise (les couches sont paramétrées et les paramètres sont ajustées - on parle d'apprentissage end-to-end), non pas fixée à l'avance comme pour les noyaux. Le dessin représente par exemple un réseau à deux couches linéaires, une contenant deux neurones (le 1 et le 2) et une deuxième couche avec un neuronne (3). Remarque très importante : il ne suffit pas d'empiler des couches linéaires ! Mettre bout à bout des couches linéaires n'augmentent absolument pas l'expressivité, cela reste une fonction linéaire ! Il faut nécessairement introduire de la non-linéarité entre les couches, c'est-à-dire une fonction (appelée activation) qui transforme non linéairement la sortie de chaque couche. On peut par exemple imaginer la fonction *signe* en sortie de la première couche (donc des neurones 1 et 2) qui va permettre de binariser la décision de cette couche. Avec une telle fonction, il est facile de séparer les nuages de la figure : il suffit que le neurone 1 code pour *à droite* du nuage rouge, le neurone 2 *à gauche* du nuage rouge, en combinant ces deux décisions on peut parfaitement classer le jeu de données. Ceci serait impossible sans linéarité (essayer pour comprendre le problème). Cependant, il ne va pas être possible d'utiliser la fonction signe comme non-linéarité, car tout comme la fcontion de coût 0-1, elle est constante par morceaux : du coup sa dérivée est nulle partout . . . Or si on veut pouvoir utiliser un algorithme de descente de gradient pour apprendre les poids, nous avons besoin que la dérivée soit informative. On utilisera donc des fonctions similaires à celle-ci mais informative, comme la sigmoïde ou la tangente hyperbolique.

2 Réseau à deux couches

Le point technique central dans les réseaux de neurones est ce qu'on appelle la back-propagation : c'est l'algorithme utilisé pour effectuer une descente de gradient et apprendre les poids dans les différentes couches. La vision moderne est de voir le réseau comme une série de fonctions composées (les modules ou les couches que l'on enchaîne les uns derrière les autres). C'est cette vision que l'on développera dans la section 5. Pour bien comprendre le problème, nous allons commencer par illustrer le fonctionnement sur un réseau simple à deux couches avec trois neurones comme celui de l'introduction : le réseau à deux entrées $\mathbf{x}_1, \mathbf{x}_2$ (donc on traite des exemples dans $\mathbb{R}^2$) et une sortie $\hat{y}$. Cette sortie est le résultat d'abord de l'application de deux fonctions linéaires qui produisent les sorties a_1, a_2 ; ces sorties sont transformées non-linéairement par la fonction d'activation $g(x) = tanh(x)$ et produisent les scalaires z_1 et z_2. Le neurone 3 de la dernière couche combine linéairement ces deux scalaires (qui correspondent à la sortie de la première couche et sont donc les entrées de la deuxième couche) pour produire la sortie a_3. Cette sortie est enfin transformée également par une fonction d'activation, dans cet exemple la même que précédemment, $g(x) = tanh(x)$ pour produire $\hat{y}$. La sortie correspond donc au calcul

$$\hat{y} = g(w_{13}g(w_{11}x_1 + w_{21}x_2 + w_{01}) + w_{23}g(w_{12}x_1 + w_{22}x_2 + w_{02}) + w_{03})$$

.

L'inférence (le calcul de la sortie) est donc un processus relativement intuitif : il suffit d'appliquer dans l'ordre les différentes couches et de considérer comme entrée d'une couche les sorties de la couche précédente.

Pour un problème de classification binaire, on peut utiliser une fonction seuil sur la valeur de sortie (par exemple avec la fonction signe) pour déduire le label positif ou négatif. Il est tout à fait approprié d'utiliser dans ce cadre une fonction de coût aux moindres carrés : la sortie de réseau étant comprise entre -1 et 1 grâce à la dernière fonction d'activation tangente hyperbolique (qui rappelons le est une approximation continue de la fonction signe), le problème de divergence de l'erreur qui se pose dans les modèles linéaires n'existe pas dans ce contexte. Nous obtenons donc la formulation usuelle de notre problème d'apprentissage : trouver les paramètres du réseau (de toutes les couches) qui minimise la fonction de coût sur un ensemble d'apprentissage. Cependant, un réseau à plus d'une couche n'est pas une fonction convexe et l'optimiser n'est pas évident du fait des multiples compositions fonctionnelles : changer les poids d'une couche modifient non seulement les sorties de cette couche mais également de toutes les couches suivantes. Une utilisation naïve de l'algorithme de descente de gradient serait donc relativement instable.

L'algorithme de rétro-propagation du gradient (backpropagation) est l'algorithme classique utilisé pour l'apprentissage des réseaux. Il consiste à corriger itérativement les poids du réseau couche par couche en commençant par la fin du réseau et en considérant lorsque l'on traite une couche que toutes les autres couches sont figées. On commence ainsi par la dernière couche : les poids sont mis-à-jour de façon à diminuer l'erreur du coût. Pour cela, il suffit de calculer le gradient du coût en fonction des paramètres/poids de la couche et de les mettre à jour selon l'algorithme de descente du gradient. Pour cette dernière couche, c'est comme si on faisait abstraction de toutes les couches précédentes et qu'on appliquait l'algorithme usuel. Il est par contre illusoire d'arriver à un bon résultat sans optimiser les autres couches. Le slide 8 présente les calculs pour le réseau à 2 couches. Nous avons une composition de fonction : $\hat{y} = g(a_3) = g(w_{23}z_2 + w_{13}z_1 + w_{03}$. Par dérivation en chaîne on obtient le gradient du coût par rapport aux poids w_{03} (le biais), w_{13} et w_{23}. Le gradient fait intervenir sans surprise la dérivée partielle de L par rapport à la sortie a_3 (comment le coût varie en fonction de la sortie de la couche) et la dérivée partielle de a_3 en fonction des poids (comment la sortie de la couche varie en fonction des poids). Il est très important pour comprendre le mécanisme de l'algorithme de se souvenir que le gradient (et les dérivées partielles) indiquent l'effet d'une variation minime des entrées sur la sortie : grossièrement, il nous indique dans quel sens faire varier le poids afin d'augmenter ou de diminuer la sortie. L'enchaînement des dérivées partielles - qui correspond à une multiplication - permet de savoir au final si il faut diminuer ou augmenter la valeur du paramètre pour diminuer ou augmenter la valeur de sortie. Vous pouvez imaginer pour simplifier les choses que ce qui nous intéresse est le signe de la dérivée partielle : si elle est positive, alors pour faire baisser la valeur de sortie il faut augmenter le poids (la fonction est croissante) ; dans le cas inverse, il faut baisser le poids (la fonction est décroissante). En enchaînant les fonctions, ce qui nous intéresse alors est le signe des produits des dérivées partielles pour connaître le changement à appliquer au poids.

Considérons maintenant (slide 9) l'avant dernière couche (dans l'exemple choisit, il s'agit de la première couche). On aimerait également optimiser les poids de façon à minimiser l'erreur en sortie (du réseau, pas de la couche en question). La question est donc de savoir comment corriger ces poids afin de minimiser l'erreur en sortie, sachant que notre sortie z_1 ou z_2 va être utiliser par la dernière couche. Il faut donc prendre en compte cette transformation pour savoir dans quel sens modifier les poids pour minimiser le coût. On va considérer la dernière couche figée, comme si elle ne dépendait pas de poids, et calculer le gradient du coût par rapport aux poids de l'avant dernière couche. Nous avons encore une composition de fonctions : la fonction linéaire de la couche qui donne z_i, le passage par la fonction d'activation g, la fonction linéaire de la dernière couche suivie de la fonction d'activation de la dernière couche (et également les moindres carrés pour calculer l'erreur). Comme pour la dernière couche, on va calculer le produit des dérivées partielles (ce que nous donne la dérivation en chaîne) pour savoir dans quel sens il faut faire varier le poids. On remarque que les dérivées partielles qui interviennent sont celles sur le "chemin de calcul" de l'entrée de la couche à la sortie. Prenons le cas de a_1 du premier neurone : la dérivée du coût par rapport à un poids correspond au produit de la dérivée de L par rapport à la sortie de la couche a_1 et la dérivée de a_1 par rapport au poids. Cette dérivée du coût par rapport à a_1 est elle-même décomposée en la dérivée de L par rapport à a_3 la sortie du réseau, et la dérivée de a_3 par rapport à a_1 : pour savoir dans quel sens faire varier le poids, on regarde l'influence du poids sur a_1, que l'on combine à l'influence de a_1 sur a_3, combinée à l'influence de a_3 sur le coût L. La dérivée partielle du coût L par rapport à a_1 s'apparente à l'erreur à corriger au niveau de la couche considérée. Elle est composée du produit de l'erreur des couches suivantes (dérivée de L par rapport à a_3) et du poids de la connexion entre a_1 et a_3.

3 Apprentissage dans le cas général

Cette partie présente la vision historique des réseaux de neurones, qui couplaient une couche et la fonction d'activation associée. La seule (grosse) subtilité non vu dans les slides précédents concerne le cas générique de la backpropagation (slide 14). Dans l'exemple simple de la section précédente, les neurones de la première couche n'étaient reliés qu'à un seul neurone de la deuxième couche. Dans le cas général, un neurone peut être relié à de multiples neurones de la couche d'après. Le calcul alors de l'influence du coût sur la sortie de la couche doit prendre en compte tous les neurones auquel il est relié. Il faut alors sommé sur toutes les connections sortantes du neurone pour obtenir la dérivée du coût par rapport à la sortie du neurone (slide 18). On y revient plus en détails dans la section 5

4 Exemples

Plusieurs schéma repris de Duda et al. montrant le paysage de l'erreur et la forte non convexité des réseaux de neurones. Le slide 24 vous montre une manière de construire un réseau à 3 couches capable de reconnaître n'importe quelle région. Très important : on peut montrer qu'un réseau à 2 couches est un apprentisseur universel, c'est-à-dire qu'il peut approximer n'importe quel découpage de l'espace.

5 Vision modulaire

C'est la vision actuelle des choses, qui sert en particulier de base à `Pytorch` et `tensorflow`. Plutôt que de considérer une couche comme étant la combinaison d'une couche linéaire et d'une fonction d'activation, on va considérer le réseau comme un graphe dirigé de modules. Cela va permettre d'alléger les notations et de les uniformiser. Un module peut être une fonction quelconque, paramétrée ou non : une fonction linéaire multidimensionelle (paramétrée par une matrice de poids W où un élément w_{ij} représente le poids entre l'entrée i et la sortie j), une fonction d'activation tangeante, sigmoïde, ...(dans ce cas sans paramètres), ou une fonction de coût. Le graphe permet de représenter l'enchaînement des calculs. Le réseau simple présenté au début du cours peut être représenté par l'enchaînement de 5 modules : un module linéaire, une fonction d'activation, un autre module linéaire, une fonction d'activation et enfin la fonction de coût. La backpropagation est expliquée dans ce cas au slide 28. Un module non paramétré n'a pas besoin

d'être optimisé (il n'a pas de paramètres). On aura tout de même besoin de connaître la dérivée de ses sorties par rapport à ses entrées pour rétro-propager l'erreur. Pour un module paramétré, nous avons besoin de connaître la dérivée du coût par rapport aux paramètres. Soit le module M^h de la couche h, les entrées de ce module sont les z_i^{h-1}, les sorties les z_j^h et les paramètres w_{ij}^h. Ce module est relié en sortie à un autre module M^{h+1}. Pour calculer la dérivée de L par rapport à w_{ij}^h, il faut prendre en compte l'effet de z_j^h sur toute la suite des calculs (sur donc toutes les sorties z_k^{h+1} du module M^{h+1}). La règle de la dérivation en chaîne nous donne alors que l'erreur rétro-propagée des couches suivantes $\delta_j^h = \frac{\partial L}{\partial z_j^h}$ est la somme sur tous les sorties k de M^{h+1} de $\frac{\partial L}{\partial z_k^{h+1}}$ (l'erreur rétro-propagée des couches suivantes) pondérées par l'influence de z_j^h sur ces sorties : $\sum_k \frac{\partial L}{\partial z_k^{h+1}} \frac{\partial z_k^{h+1}}{\partial z_j^h}$. Il suffit de multiplier cette quantité par $\frac{\partial z_j^h}{\partial w_{ij}^h}$ pour obtenir $\frac{\partial L}{\partial w_{ij}^h}$.

On s'aperçoit donc que pour calculer toutes les dérivées partielles par rapport aux poids, il suffit de connaître pour chaque module : 1) la dérivée de ses sorties par rapport aux poids $\frac{\partial z_j^h}{\partial w_{ij}^h}$ (ou plus généralement par rapport à ses paramètres), ce qui permet de savoir comment corriger les paramètres lorsque l'on connaît l'erreur en aval δ_j^j (ces dérivées sont bien sûr nulles pour un module sans paramètres) ; 2) la dérivée de ses sortires par rapport à ses entées $\frac{\partial z_j^{h+1}}{\partial z_i^h}$ qui ne sont pas importantes pour sa mise-à-jour, mais nécessaires pour rétro-propager l'erreur aux couches en amont. Et c'est tout ! Le graphe permet ensuite grâce à ces dérivées partielles de calculer toutes les mises-à-jour nécéssaires, sans que les modules n'aient besoin de connaître toute la topologie du réseau. Il suffit pour un module de connaître uniquement son erreur en sortie δ_j^h pour se mettre à jour, c'est-à-dire de connaître uniquement son successeur.

C'est cette vision modulaire qui a permis entre autre le développement des librairies de calculs utilisées à l'heure actuelle, très flexibles. Pour coder un module, il suffit de fournir 3 fonctions : le calcul du module en lui-même, la dérivée partielle par rapport aux entrées et la dérivée partielle par rapport aux paramètres. La puissance de l'autograd (dérivée automatique à partir de graphes de calcul) permet aujourd'hui de coder un module en donnant uniquement la première fonction, les dérivées sont déduites des calculs effectués.

Le reste des slides est à titre informatif, vous pouvez vous réferer à énormément de ressources en ligne pour en apprendre plus sur l'apprentissage profond en attendant l'année prochaine et le M2.

Ressources :

— Pattern Classification de Duda, Hart, Stork. Trouvable en ebook.
— Le livre de Bengio et le site associé.