Apprentissage Statistique

Équations de rétro-propagation

Vincent CHAMBRIN

Abstract

On se propose dans ce qui suit d'énoncer les équations de la rétropropagation et de les démontrer dans le cas d'un réseau ayant une seule couche cachée.

Cas d'une seule couche cachée

Nous allons d'abord commencer par introduire quelques notations et par définir les éléments de notre réseau de neurones à une couche cachée.

On se place dans le cas d'un problème de classification à K classes. Notre réseau de neurones possède donc K neurones sur sa couche de sortie, chacun correspondant à l'une des classes. Pour obtenir une probabilité en sortie, la fonction d'activation utilisée sur cette couche est la fonction SOFTMAX. On note $W^{(O)}$ la matrice des poids et $b^{(O)}$ le vecteur des biais associés ; si bien que l'on peut écrire la pré-activation de cette couche par :

$$z^{(O)} = W^{(O)}a^{(H)} + b^{(O)}$$

où $a^{(H)}$ est l'activation de la couche cachée (H pour *hidden*).

La couche d'entrée de notre réseau est constituée de n neurones correspondant aux *features* utilisées pour la classification. Enfin, la couche cachée est constituée de H neurones et utilise une fonction d'activation g quelconque (différentiable presque partout). La encore, on note $W^{(H)}$ et $b^{(H)}$ la matrice des poids et le vecteur des biais. Pour une entreé vectorielle x, on note $x_1, \cdots, x_n$ ses composantes.

Objet	Dimensions
x	(n)
$W^{(H)}$	(H, n)
$b^{(H)}$	(H)
$W^{(O)}$	(K, H)
$b^{(O)}$	(K)

L'activation de la couche cachée s'écrit,

$$a^{(H)} = g(W^{(H)}x + b^{(H)})$$

et la sortie du réseau vaut :

$$f(x) = a^{(O)} = \text{SOFTMAX}(z^{(O)})$$

Pour chaque entrée x est attendue une sortie y. Le but de l'apprentissage des coefficients du réseau va être de minimiser une fonction de perte $l(f(x), y)$. Pour cela, on peut utiliser un algorithme de descente de gradient. Cela nécessite cependant de calculer les quantités suivantes :

$$\frac{\partial l(f(x),y)}{\partial W_{ij}^{(O)}} \qquad \frac{\partial l(f(x),y)}{\partial b_i^{(O)}}$$

$$\frac{\partial l(f(x),y)}{\partial W_{ij}^{(H)}} \qquad \frac{\partial l(f(x),y)}{\partial b_i^{(H)}}$$

Le but des équations de retro-propagation est de fournir un algorithme permettant de calculer ces quantités efficacement.

La première étape de l'algorithme consiste à calculer le gradient de la perte par rapport à la sortie $f(x)$. On rappelle que $f(x) = a^{(O)} = \text{SOFTMAX}(z^{(O)})$ est un vecteur de taille K et que, pour tout $i \in [\![1, K]\!]$,

$$\text{SOFTMAX}(z^{(O)})_i = \frac{\exp(z_i^{(O)})}{\sum_{j=1}^{K} \exp(z_j^{(O)})}$$

On suppose donc que l'on est en mesure de calculer les réels

$$\frac{\partial l(f(x),y)}{\partial \text{SOFTMAX}(z^{(O)})_i} = \frac{\partial l(f(x),y)}{\partial a_i^{(O)}}$$

On va ensuite utiliser la règle de la chaîne pour calculer les autres quantités. Pour cela, il nous faut d'abord calculer les dérivées partielles du SOFTMAX par rapport aux $z_i^{(O)}$.

$$\frac{\partial a_i^{(O)}}{\partial z_j^{(O)}} = \begin{cases} a_i^{(O)}(1 - a_i^{(O)}) \text{ si } i = j \\ -a_i^{(O)} a_j^{(O)} \text{ sinon.} \end{cases}$$

On peut maintenant commencer à appliquer la règle de la chaîne.

$$\frac{\partial l(f(x),y)}{\partial z_i^{(O)}} = \sum_{j=1}^{K} \frac{\partial l(f(x),y)}{\partial a_j^{(O)}} \frac{\partial a_j^{(O)}}{\partial z_i^{(O)}}$$

Les quantités dans la somme sont toutes calculables.

En utilisant le fait que

$$\frac{\partial z_i^{(O)}}{\partial b_j^{(O)}} = \mathbb{1}_{i=j},$$

et en appliquant à nouveau la règle de la chaîne, on obtient :

$$\frac{\partial l(f(x),y)}{\partial b_i^{(O)}} = \frac{\partial l(f(x),y)}{\partial z_i^{(O)}}$$

Calculons maintenant la dérivée du coûts par rapport aux poids de la couche de sortie.

$$\frac{\partial l(f(x),y)}{\partial W_{ij}^{(O)}} = \sum_{k=1}^{K} \frac{\partial l(f(x),y)}{\partial z_k^{(O)}} \frac{\partial z_k^{(O)}}{\partial W_{ij}^{(O)}}$$

Évidemment, la pré-activation du neurone k à la sortie ne dépend pas des lignes autres que la ligne k de la matrice $W^{(O)}$, i.e.

$$\forall i \neq k, \quad \frac{\partial z_k^{(O)}}{\partial W_{ij}^{(O)}} = 0.$$

Au final,

$$\begin{aligned} \frac{\partial l(f(x),y)}{\partial W_{ij}^{(O)}} &= \frac{\partial l(f(x),y)}{\partial z_i^{(O)}} \frac{\partial z_i^{(O)}}{\partial W_{ij}^{(O)}} \\ &= \frac{\partial l(f(x),y)}{\partial z_i^{(O)}} a_j^{(H)} \end{aligned}$$

Passons maintenant à la couche cachée.

$$\begin{aligned} \frac{\partial l(f(x),y)}{\partial a_i^{(H)}} &= \sum_{j=1}^{K} \frac{\partial l(f(x),y)}{\partial z_j^{(O)}} \frac{\partial z_j^{(O)}}{\partial a_i^{(H)}} \\ &= \sum_{j=1}^{K} \frac{\partial l(f(x),y)}{\partial z_j^{(O)}} W_{ji}^{(O)} \end{aligned}$$

L'équation précédente peut être réécrite sous forme matricielle :

$$\nabla_{a^{(H)}} l(f(x), y) = \left(W^{(O)}\right)' \nabla_{z^{(O)}} l(f(x), y)$$

On remonte à la pré-activation en utilisant la relation

$$a^{(H)} = g(z^H)$$

Pour tout $i \in [\![1, H]\!]$,

$$\frac{\partial l(f(x), y)}{\partial z_i^H} = \sum_{j=1}^{H} \frac{\partial l(f(x), y)}{\partial a_j^{(H)}} \frac{\partial a_j^{(H)}}{\partial z_i^H}$$

Si la fonction d'activation g s'applique composante par composante (e.g. sigmoïde, RELU), l'équation précédente se simplifie en,

$$\frac{\partial l(f(x), y)}{\partial z_i^H} = \frac{\partial l(f(x), y)}{\partial a_i^{(H)}} g'(z_i^H),$$

et l'on a l'écriture vectorielle

$$\nabla_{z^{(H)}} l(f(x), y) = \nabla_{a^{(H)}} l(f(x), y) \odot \nabla_{z^H} g$$

ou encore

$$\nabla_{z^{(H)}} l(f(x), y) = \left(W^{(O)}\right)' \nabla_{z^{(O)}} l(f(x), y) \odot \nabla_{z^H} g$$

En reprenant les calculs effectués pour la couche O, on obtient :

$$\frac{\partial l(f(x), y)}{\partial b_i^{(H)}} = \frac{\partial l(f(x), y)}{\partial z_i^{(H)}}$$

$$\frac{\partial l(f(x), y)}{\partial W_{ij}^{(H)}} = \frac{\partial l(f(x), y)}{\partial z_i^{(H)}} x_j$$

Perte *categorical cross-entropy*

Dans le cas de la perte *categorical cross-entropy* définie par

$$l(f(x), y) = -\sum_{i=1}^{K} y_i \log(f(x)_i) = -\log(f(x)_y)$$

où y est le numéro de la classe ; les équations au niveau de la dernière couche se simplifient considérablement. En effet, on a alors

$$\frac{\partial l(f(x), y)}{\partial a_i^{(O)}} = \frac{-\mathbb{1}_{i=y}}{f(x)_y}$$

et

$$\frac{\partial l(f(x), y)}{\partial z_i^{(O)}} = \sum_{j=1}^{K} \frac{\partial l(f(x), y)}{\partial a_j^{(O)}} \frac{\partial a_j^{(O)}}{\partial z_i^{(O)}} = \frac{-1}{f(x)_y} \frac{\partial a_y^{(O)}}{\partial z_i^{(O)}}$$

En distinguant les cas $i = y$ et $i \neq y$, on obtient :

$$\frac{\partial l(f(x), y)}{\partial z_i^{(O)}} = \begin{cases} a_y^{(O)} - 1 \text{ si } i = y \\ a_i^{(O)} \text{ sinon.} \end{cases}$$

Cas général

On note cette fois L le nombre de couches et

$$\delta^{(l)} = \frac{\partial l(f(x), y)}{\partial z_i^{(l)}}$$

l'erreur au niveau de la couche l. On a alors les équations suivantes :

$$\delta^{(L)} = (J^{(L)})' \ \nabla_{a^{(L)}} l(f(x), y)$$

$$\delta^{(l)} = (J^{(l)})' \times (W^{(l+1)})' \delta^{(l+1)}$$

$$\frac{\partial l(f(x), y)}{\partial b_j^{(l)}} = \delta_j^{(l)} \qquad \frac{\partial l(f(x), y)}{\partial W_{jk}^{(l)}} = a_k^{(l-1)} \delta_j^{(l)}$$

où $J^(l)$ est la matrice jacobienne de la fonction d'activation à la couche l, et avec la convention $a^0 = x$ l'entrée du réseau.

Dans le cas où les fonctions d'activation s'appliquent composantes par composantes, la jacobienne est diagonale et cela revient à faire un produit composantes par composantes avec la dérivée de la fonction d'activation aux points considérés.

La démonstration de ces équations peut se faire par récurrence en reprenant ce qui a déjà été fait. D'autres preuves peuvent être facilement trouvées dans [1] et [2] (avec des notations légèrement différentes).

Code Python

Il est facile d'implémenter l'algorithme de rétropropagation en Python avec l'aide de la bibliothèque Numpy.

```
# github.com/RugessNome/mathappli-apst1
def backprop(layers, cost_derivative):
  # Processing last layer
  l = layers[-1]
  l.delta = np.dot(l.J(l.z, l.a).T, cost_derivative)
  l.nabla_b = l.delta
  l.nabla_w = np.outer(l.delta, layers[-2].a)
  # Iterating over the other layers
  for i in range(2, len(layers)):
    l = layers[-i]
    prev_l = layers[-(i-1)]
    next_l = layers[-(i+1)]
    nabla_a = np.dot(prev_l.w.T, prev_l.delta)
    l.delta = np.dot(l.J(l.z, l.a).T, nabla_a)
    l.nabla_b = l.delta
    l.nabla_w = np.outer(l.delta, next_l.a)
```

Associé à un algorithme de descente de gradient stochastique, on peut facilement construire et entraîner son propre réseau de neurones. Néanmoins, il est souvent plus pratique - et recommandé - d'utiliser des bibliothèques toutes prêtes comme Keras, qui permettent en plus de construire des réseaux beaucoup plus complexes et optimisés (e.g. réseaux convolutionnels).

Bibliographie

[1] M. A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[2] WikiStat. Neural networks and introduction to deep learning. http://wikistat.fr/pdf/st-m-hdstat-rnn-deep-learning.pdf.

Exemple

On se place dans un problème de classification où les entrées sont dans $\mathbb{R}^2$. Les trois classes sont définies par :

1. $y < -\frac{1}{4}$;
2. $x < 0$;
3. les points qui ne sont ni dans 1 ni dans 2.

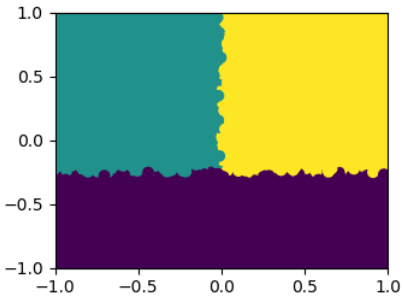

FIGURE 1 – Objectif

On construit un réseau de neurones ayant les couches suivantes :

1. couche d'entrée de 2 neurones ;
2. couche RELU à 4 sorties ;
3. couche SOFTMAX à 3 sorties.

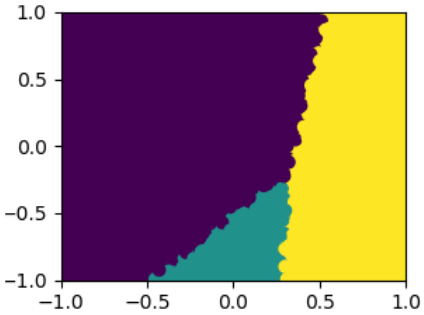

FIGURE 2 – Réseau initialisés avec une $\mathcal{N}(0, 1)$

On entraîne le réseau en utilisant une descente de gradient stochastique :

— 10000 échantillons d'entraînement ;
— 20 *epoch* ;
— *batch* de taille 500 ;
— taux d'apprentissage $\eta = 1$.

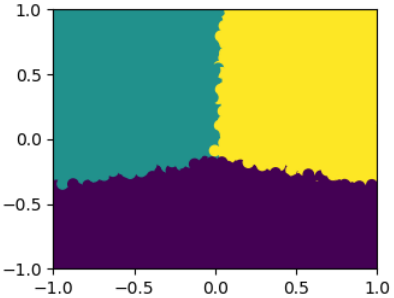

FIGURE 3 – Réseau entraîné.

On obtient un taux de classification correcte de l'ordre de 96%.