TME 7 - Réseau de neurones : DIY

L'objectif de ce TME est d'implémenter un réseau de neurones. L'implémentation est inspirée des anciennes versions de `Lua/torch` (avant l'autograd) et des implémentations analogues qui permettent d'avoir des réseaux génériques très modulaires. Chaque couche du réseau est vu comme un module et un réseau est constitué ainsi d'un ensemble de modules. En particulier, les fonctions d'activation sont aussi considérées comme des modules.
Notons $M^h(\mathbf{z}, \mathbf{W})$ le module de la couche h de paramètre $\mathbf{W}$, $\mathbf{z}^h = M^h(\mathbf{z}^{h-1}, \mathbf{W}^h)$ l'entrée de la couche $h+1$ (ou la sortie de la couche h) et $L(\mathbf{y}, \hat{\mathbf{y}})$ la fonction de coût. Pour pouvoir calculer la mise-à-jour des paramètres de chaque module h, on a besoin de calculer $\nabla_{\mathbf{W}^h} L$.
On pose par ailleurs $\delta_j^h = \frac{\partial L}{\partial z_j^h}$. Alors

$$\frac{\partial L}{\partial w_{i,j}^h} = \sum_k \frac{\partial L}{\partial z_k^h} \frac{\partial z_k^h}{\partial w_{i,j}^h} = \sum_k \delta_k^h \frac{\partial z_k^h}{\partial w_{i,j}^h} = \delta_j \frac{\partial z_j^h}{\partial w_{i,j}} = \delta_j \frac{\partial M_j^h(\mathbf{z}^{h-1}, \mathbf{W}^h)}{\partial w_{i,j}} \tag{1}$$

$$\delta_j^{h-1} = \frac{\partial L}{\partial z_j^{h-1}} = \sum_k \frac{\partial L}{\partial z_k^h} \frac{\partial z_k^h}{\partial z_j^{h-1}} = \sum_k \delta_j^h \frac{\partial M^h(\mathbf{z}^{h-1}, \mathbf{W}^h)}{\partial z_j^{h-1}} \tag{2}$$

Ainsi, pour pouvoir utiliser la backpropagation, il suffit que chaque module puisse calculer sa dérivée par rapport à ses paramètres (eq. 1) et sa dérivée par rapport à ses entrées (eq. 2).
Une classe module contient :

- une variable `parameters` qui stocke les paramètres du module (la matrice de poids par exemple pour un module linéaire) ;
- une méthode `forward(data)` qui permet de calculer les sorties du module pour les entrées passées en paramètre ;
- une variable `gradient` qui permet d'accumuler le gradient calculé par exemple ;
- une méthode `zero_grad()` qui permet de réinitialiser à 0 le gradient ;
- une méthode `backward_update_gradient(input,delta)` qui permet de calculer le gradient du coût par rapport aux paramètres et l'additionner à la variable `gradient`, en fonction de l'entrée `input` et des deltas de la couche suivante `delta` ;
- une méthode `backward_delta(input,delta)` qui permet de calculer le gradient du coût par rapport aux entrées en fonction de l'entrée `input` et des deltas de la couche suivante `delta` ;
- une méthode `update_parameters(gradient_step)` qui met à jour les paramètres du module selon le gradient accumulé jusqu'à son appel avec un pas de `gradient_step`.

Lorsque plusieurs modules sont mis en série, il suffit ainsi pour la passe forward d'appeller successivement les fonctions `forward` de chaque module avec comme entrée la sortie du précédent. Pour la passe backward, le dernier module calcule le gradient par rapport à ses paramètres et les deltas qu'il doit rétropropager (à partir des deltas du loss) ; puis en parcourant en sens inverse le réseau, chaque module répète la même opération : le calcul de la mise à jour de son gradient (`backward_update_gradient`) et le delta qu'il doit transmettre à la couche précédente (`backward_delta`).
La classe abstraite d'un module et d'un loss sont les suivantes :

```
class Loss(object):
    def forward(self,y,yhat):
        #calcul le cout
    def backward(self,y,yhat):
        calcul le gradient du cout
class Module(object):
    def __init__(self):
        self._parameters = None
```

```
        self._gradient = None
    def zero_grad(self):
        ## Annule gradient
    def forward(self,X):
        ## Calcule la passe forward
    def update_parameters(self,gradient_step):
        ## Calcule la mise a jour des parametres selon le gradient calcule et
        ## le pas de gradient_step
    def backward_update_gradient(self,input, delta):
        ## Met a jour la valeur du gradient
    def backward_delta(self,input, delta):
        ## Calcul la derivee de l'erreur
```

- Implémenter un module linéaire, une fonction d'activation sigmoïde ou tanh, une fonction de coût MSE.
- Tester "à la main" votre implémentation en créant un réseau à une couche cachée.
- Implémenter une classe `Sequentiel` qui permet d'ajouter des modules en série et qui automatise les procédures de forward et backward quel que soit le nombre de modules. Stocker bien les résultats des différents appels aux forward afin de ne pas gaspiller du temps de calcul.
- Tester sur les données USPS.