AS - TP 1
Introduction à `pytorch`

Nicolas Baskiotis - Benjamin Piwowarski
(fortement inspiré du petit guide de la rétro-propagation de L. Denoyer)

2018-2019

Notations, définitions et rappels

Les notations suivantes seront utilisées dans la plupart des TPs :
— un espace de représentation $\mathbb{R}^d$ à d dimensions
— un espace de sortie $\mathbb{R}^c$ à c dimensions
— un exemple $\mathbf{x} = (x_1, \ldots, x_d) \in \mathbb{R}^d$, son étiquette $\mathbf{y} = (y_1, \ldots, y_c) \in \mathbb{R}^c$
— un ensemble d'apprentissage $\mathcal{X} = \{(\mathbf{x}^i, \mathbf{y}^i) \in \mathbb{R}^d \times \mathbb{R}^c\}_{i=1}^N$ de N exemples
— une fonction de coût $\Delta : \mathbb{R}^c \times \mathbb{R}^c \to \mathbb{R}$
— une fonction $f_{\mathbf{w}} : \mathbb{R}^d \to \mathbb{R}^c$ paramétrée par $\mathbf{w}$.

La fonction $f_{\mathbf{w}}$ correspond au prédicteur (ou classifieur) que l'on souhaite apprendre, i.e. trouver le paramètre $\mathbf{w}$ qui minimise le risque d'erreur sur les prédictions. Pour un exemple $\mathbf{x}$ donné, la sortie du prédicteur $\hat{\mathbf{y}} = f_{\mathbf{w}}(\mathbf{x})$ est comparée à la sortie attendu $\mathbf{y}$ grâce à la fonction de coût $\Delta(\hat{\mathbf{y}}, \mathbf{y})$ qui permet de quantifier l'erreur. Pour un paramètre $\mathbf{w}$, le coût associé à l'ensemble d'apprentissage est

$$L(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^{N} \Delta(f_{\mathbf{w}}(\mathbf{x}^i), \mathbf{y}^i)$$

.

Dans le contexte de la minimisation du risque empirique, la formalisation du problème d'apprentissage est la suivante : trouver le paramètre optimal $\mathbf{w}^*$ qui minimise le coût de prédiction sur l'ensemble des données d'apprentissage

$$\mathbf{w}^* = argmin_{\mathbf{w}} L(\mathbf{w}) = argmin_{\mathbf{w}} \frac{1}{N} \sum_{i=1}^{N} \Delta(f_{\mathbf{w}}(\mathbf{x}^i), \mathbf{y}^i)$$

.

Un algorithme d'apprentissage classique pour optimiser le paramètre est l'algorithme de descente de gradient. Il consiste à mettre à jour itérativement le paramètre $\mathbf{w}$ selon la formule :

$$\mathbf{w} \to \mathbf{w} - \epsilon \nabla_{\mathbf{w}} L(\mathbf{w})$$

avec ϵ un paramètre appelé le pas de gradient.

Trois variantes sont très utilisées :
— gradient **batch** : le gradient est calculé sur la somme des coûts de tous les exemples d'apprentissage, i.e. $L(\mathbf{w})$;

— gradient **stochastique** : à chaque itération, un exemple est tiré aléatoirement ; le coût et le gradient du coût n'est calculé que pour cet exemple et la mise-à-jour du paramètre est fait selon ce gradient ;
— gradient **mini-batch** : l'ensemble d'apprentissage est partitionné en petits sous-ensembles appelés *mini-batch* ; à chaque itération le coût et le gradient est calculé sur un des sous-ensembles, et le paramètre est mis-à-jour selon ce gradient. La taille des mini-batchs est un paramètre supplémentaire, généralement quelques dizaines d'exemples.

1 TP 1 - Modèle linéaire

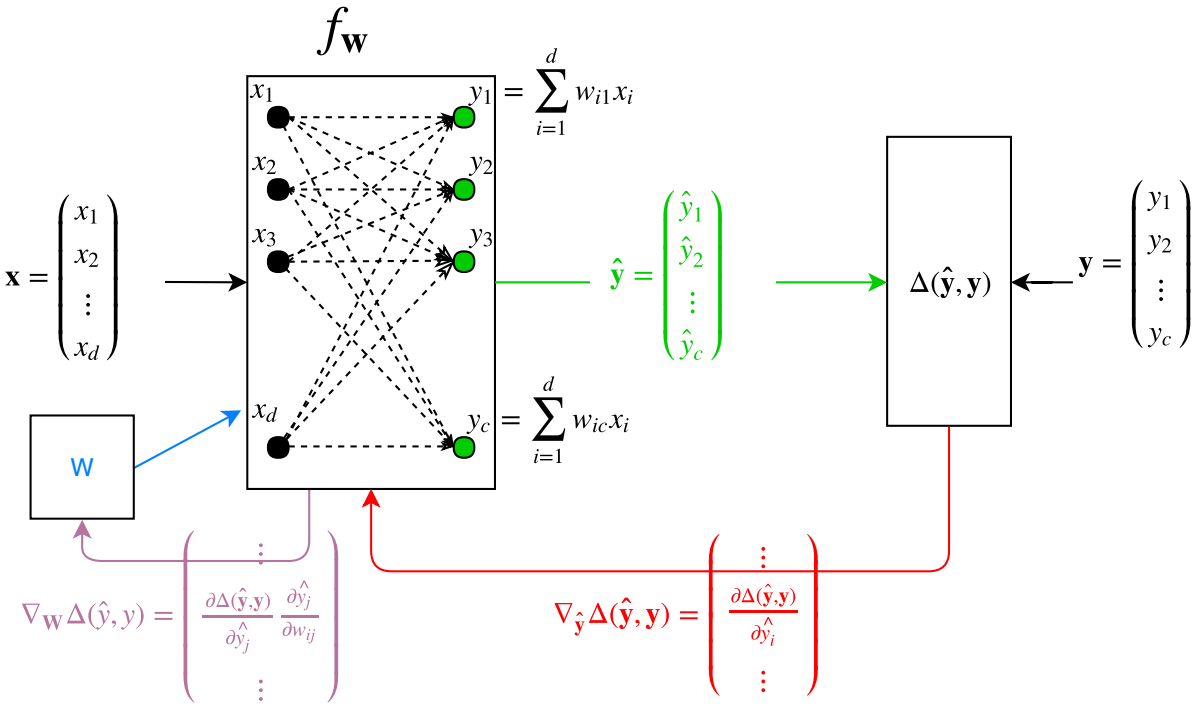
Dans cette partie, nous allons nous focaliser sur l'implémentation en `pytorch` de modèles linéaires.

1.1 Régression linéaire

Le modèle de régression linéaire peut être vu comme la composition de deux éléments :

— un module linéaire, responsable du calcul de $f_{\mathbf{w}}(\mathbf{x}) = \begin{pmatrix} \sum_i w_{i,1} x_i \\ \vdots \\ \sum_i w_{i,c} x_i \end{pmatrix} = \mathbf{w}^T \mathbf{x}$

— un coût aux moindres carrés (MSE) : $\Delta(\hat{\mathbf{y}}, \mathbf{y}) = \|\hat{\mathbf{y}} - \mathbf{y}\|^2$



Calcul du gradient La mise à jour des paramètres du modèle nécessite le calcul de $\nabla_{\mathbf{w}} \Delta(f_{\mathbf{w}}(\mathbf{x}), \mathbf{y})$. En notant $\hat{y}_j$ la j-ème composante du vecteur de sortie $\hat{\mathbf{y}} = f_{\mathbf{w}}(\mathbf{x})$, pour chaque paramètre $w_{i,j}$:

$$\frac{\partial \Delta(f_{\mathbf{w}}(\mathbf{x}), \mathbf{y})}{\partial w_{i,j}} = \frac{\partial \Delta(f_{\mathbf{w}}(\mathbf{x}), \mathbf{y})}{\partial \hat{y}_j} \frac{\partial \hat{y}_j}{\partial w_{i,j}}$$

En notant $\delta_j = \frac{\partial \Delta(f_{\mathbf{w}}(\mathbf{x}),\mathbf{y})}{\partial \hat{y}_j}$, on obtient :

$$\frac{\partial \Delta(f_{\mathbf{w}}(\mathbf{x}),\mathbf{y})}{\partial w_{i,j}} = \delta_j \frac{\partial \hat{y}_j}{\partial w_{i,j}} = \delta_j \frac{\partial f_{\mathbf{w}}(\mathbf{x})}{\partial w_{i,j}}$$

Nous avons donc besoin de calculer le gradient du coût et le gradient du module linéaire par rapport à ses paramètres.

Implémentation Deux classes abstraites seront donc utiles pour l'implémentation : la classe `Module` et la classe `Loss`. Le code ci-dessous vous donne le squelette que devra suivre votre implémentation (selon les conventions `pytorch`).

```
class Module:
    def forward(self,x):
        ## Calcul la sortie du module
        pass
    def backward_update_gradient(self,x,delta):
        ## Gradient du module par rapport aux parametres
        ## et mise a jour du gradient
        pass
    def update_parameters(self,epsilon):
        ## Mise a jour des parametres
        pass
    def backward_delta(self,x,delta):
        ## Retourne le gradient du module par rapport aux entrees
        pass
    def zero_grad(self):
        ## Remise a zero du gradient
        pass
    def initialize_parameters(self):
        ### Initialisation des parametres
        pass
```

```
class Loss:
    def forward(self,y,ypred):
        ## Calcule l'erreur
        pass
    def backward(self,y,ypred):
        ## Gradient du cout
        pass
```

```
class Lineaire(Module):
    def __init__(self,in,out):
        #in: dimension de l'entree
        #out: dimension de la sortie
        pass
```

```
class MSE(Loss):
    pass
```

```
### Exemple de boucle principal pour la descente de gradient
model = Lineaire(nin,nout)
model.initialize_parameters()
loss = MSE()
while True:
    x,y = ...
    model.zero_grad()
    yhat = model.forward(x)
    err = loss.forward(y,yhat)
    delta = loss.backward(y,yhat)
    model.backward_update_gradient(x,delta)
    model.update_parameters(epsilon)
```

En détails :

- les méthodes `forward` permettent de calculer la sortie du module et le coût ;
- la méthode `backward` de `Loss` permet de renvoyer le gradient du coût (les quantités notées δ_i précédemment)
- un module contient une variable interne `gradient` qui permet d'accumuler le gradient (l'accumulation peut être utile par exemple lors des itérations sur un même batch). Cette variable peut être remise à zéro grâce à la fonction `zero_grad`.
- la fonction `backward_update_gradient(x,delta)` permet de calculer le gradient de l'erreur par rapport aux paramètres $\mathbf{w}$ au point $\mathbf{x}$ en utilisant `delta`, le gradient calculé avec la méthode `backward` de `Loss`. Au lieu de renvoyer la valeur du gradient, cette fonction met à jour directement la variable gradient stockée en interne du module.

— enfin, la fonction `update_parameters(epsilon)` est la fonction de mise-à-jour des paramètres : lorsque les gradients ont été calculés sur tous les exemples du mini-batch considéré, cette fonction permet de calculer les nouvelles valeurs des paramètres du module en fonction de la variable gradient stockée en interne du module en suivant l'algorithme de descente du gradient.

1. Implémenter en `pytorch` les classes nécessaires pour la régression linéaire : un module linéaire et un coût MSE. Utiliser bien les outils propres à `pytorch`, en particulier des `Tensor` plutôt que des matrices `numpy`, de telle manière que vos fonctions prennent en entrée des batchs d'exemples (matrice 2D) et non un seul exemple (vecteur). N'hésiter pas à prendre un exemple et de déterminer les dimensions des différentes matrices en jeu.
2. Tester l'ensemble de vos fonctions, puis implémenter l'algorithme de descente de gradient stochastique.
3. Tester votre implémentation avec les données de Boston Housing. Tracer la courbe du coût en apprentissage et celle en test.
4. Implémenter une descente de gradient batch et une mini-batch. Comparer la vitesse de convergence et les résultats obtenus.

Doc officielle tenseur `pytorch` : `https://pytorch.org/docs/stable/tensors.html`

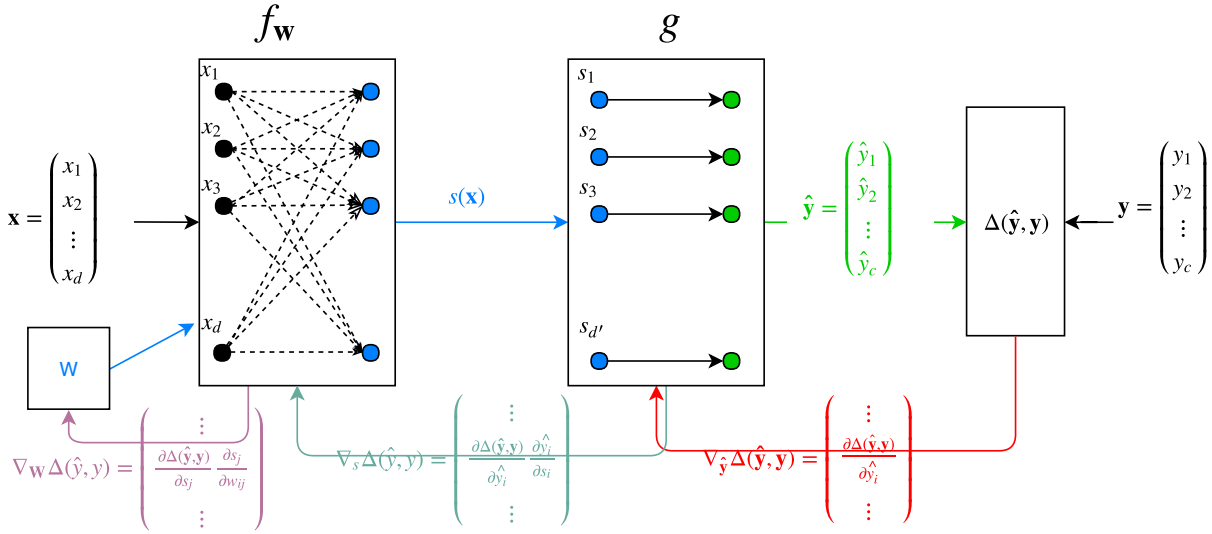
1.2 Perceptron

Pour obtenir un perceptron, il suffit de remplacer le coût MSE par le coût perceptron : $\Delta(\hat{y}, y) = max(0, -y f_{\mathbf{w}}(\mathbf{x}))$. Implémenter un perceptron et tester vos algorithmes sur les données MNIST (en classification binaire et multiclasse).

1.3 Régression logistique et fonction d'activation

Une fonction d'activation peut être vu comme un module sans paramètre. Elles sont utiles pour introduire la non-linéarité dans un réseau de neurones. On peut également utiliser une fonction d'activation pour borner les sorties du réseau : par exemple, la fonction *tanh* permet de forcer les valeurs de sorties à être entre -1 et 1 ; la sigmoïde entre 0 et 1.

Le schéma suivant illustre le cas d'un module linéaire suivi d'une fonction d'activation g : $\hat{\mathbf{y}} = g(f_{\mathbf{w}}(\mathbf{x}))$.



Comme ce module n'a pas de paramètres, il n'y a bien sûr pas de descente de gradient à effectuer sur lui. Par contre, afin de pouvoir mettre à jour les paramètres du module linéaire, il est nécessaire comme auparavant de calculer $\nabla_{\mathbf{w}}\Delta(g(f_{\mathbf{w}}(\mathbf{x})), \mathbf{y})$, soit pour chaque paramètre $w_{i,j}$ et en notant $s(\mathbf{x}) = f_{\mathbf{w}}(\mathbf{x})$:

$$\frac{\partial\Delta(g(f_{\mathbf{w}}(\mathbf{x})), \mathbf{y})}{\partial w_{i,j}} = \frac{\partial\Delta(g(f_{\mathbf{w}}(\mathbf{x})), \mathbf{y})}{\partial g_j(f_{\mathbf{w}}(\mathbf{x}))}\frac{\partial g_j(f_{\mathbf{w}}(\mathbf{x}))}{\partial w_{i,j}} = \frac{\partial\Delta(g(s(\mathbf{x})), \mathbf{y})}{\partial g_j(s(\mathbf{x}))}\frac{\partial g_j(s(\mathbf{x}))}{\partial s_j(\mathbf{x})}\frac{\partial s_j(\mathbf{x})}{\partial w_{i,j}}$$

En notant $\delta_j = \frac{\partial\Delta(g(s(\mathbf{x}), \mathbf{y}))}{\partial s_j(\mathbf{x})}$ et $\delta_i^{'} = \frac{\partial g_i(s(\mathbf{x}))}{\partial s_i(\mathbf{x})}$, on obtient :

$$\frac{\partial\Delta(g(f_{\mathbf{w}}(\mathbf{x}), \mathbf{y}))}{\partial w_{i,j}} = \delta_j\delta_j^{'}\frac{\partial f_{\mathbf{w}}(\mathbf{x})}{\partial w_{i,j}}$$

Nous avons donc besoin des dérivées partielles du module g par rapport à ses entrées. Dans l'implémentation proposée ci-dessus, c'est la fonction `backward_delta` qui est en charge de ce calcul.

Régression logistique La régression logistique peut être vue comme l'enchaînement d'un module linéaire et d'une fonction d'activation sigmoïde

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

avec un coût de cross-entropie :

$$\Delta(\hat{\mathbf{y}}, \mathbf{y}) = \sum_{i=1}^{c} y_i \log \hat{y}_i - (1 - y_i)\log(1 - \hat{y}_i)$$

.

Implémenter la régression logistique et tester votre implémentation.