AS - TP2 - Mini-Batch, train/test et implémentation de critère

Ludovic Denoyer et Nicolas Baskiotis

27 sept. 2016

Notations

— la base d'apprentissage : $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N) \in \mathbb{R}^d \times \mathcal{Y}\}$, $\mathcal{Y}$ discret ou continue (regression) ;
— le modèle : $f_{\boldsymbol{\theta}} : \mathbb{R}^d \to \mathbb{R}$ de paramètre $\boldsymbol{\theta} \in \mathbb{R}^d$;
— la fonction de coût : $\Delta(\hat{y}, y)$ entre la prédiction et la sortie attendue ;

Préambule - (re)prise en main de Lua/Torch

Torch est particulier comme langage, très optimisé sur les opérations matricielles, ce qui implique quelques inconvénients lorsque l' on est habitué à **matlab** ou à **python**. Parmi quelques difficultés :

— beaucoup d'opérations sont *in-places* : `torch.cmax(t,0)` renvoie bien une nouvelle matrice, mais `t:cmax(0)` remplace les valeurs du tenseur !
— il n'y a pas de copies explicites lors des opérations d'affectations : `u = t` fait pointer `u` vers `t` !
— il est difficile de raisonner sur les indexes : on privilégie des matrices contiguës à des fins d'accélération des calculs ;
— pas de conversion de type explicite : un tenseur de double ne sera pas vu comme un tenseur d'entier, la moyenne n'est pas possible sur un tenseur de booléen, ... ; pensez à convertir les tenseurs avec les fonctions adéquates (par exemple `t:double()`) ;
— et une règle en or : quand vous ne savez pas faire, faites des boucles `for` !! Contrairement à python ou matlab, les boucles sont ultra-optimisées et utilisées à tout bout de champ.

Afin de vérifier vos connaissances, essayez de réaliser le chargement de données décrit ci-dessous. Ne perdez pas trop de temps dessus (15min maximum), une solution se trouve sur le site de l'ue.

Chargement des données

Dans ce TP, nous allons travailler sur le jeu de données classique `mnist` : 60k images de 28*28 pixels qui représentent chacune un chiffre manuscrit. Chaque pixel est codé par un entier, l'intensité lumineuse du pixel. Les données sont stockées dans un tenseur de dimension $60000 \times 28 \times 28$. Le jeu de données est par nature multi-classes. Afin de traiter un problème de classification binaire, nous allons considérer par la suite que deux classes de chiffres (au choix). Proposez un script qui permet de charger les données, les transforme en un tenseur 2d de dimension 60000×784, normalise les données entre 0 et 1, et renvoie dans un ordre aléatoire les exemples et les labels associés appartenant uniquement aux deux classes considérées, en renommant les labels en $\{-1, +1\}$.

Indications :

— pour charger les données : `mnist = require 'mnist'`
— fonctions utiles pour le traitement : `torch.max torch.div torch.reshape torch.size` (n'oubliez pas de convertir les tenseurs en double, sinon vous risquez de faire des divisions entières).
— pour afficher une image (sur la forme 2d de l'exemple) :
`require 'gnuplot'`, `gnuplot.imagesc(mnist.traindataset()[1]`
— pour le slicing (sélection des exemples correspondant à un label donné), utilisez :

— soit une boucle pour compter le nombre d'exemples correspondants, puis créer le tenseur de la bonne taille et copier les lignes qui vous intéressent ;
— soit "à la python" :

1. utilisez `Tensor.eq` qui renvoie un masque binaire, à `true` quand l'élément est égale au paramètre (le label d'intérêt) ;
2. utilisez le masque pour récupérer les indices correspondant (utilisez `torch.linspace` pour générer la liste des indices, puis la notation `indices[mask]` permet de récupérer tous les indices) ;
3. utilisez `t:index(dim,indices)` qui permet de faire le slicing selon la dimension désirée ;
4. générez le tenseur de labels avec des 1 ou -1 ;
5. mélangez le tout avec `torch.randperm`.

Et voilà, votre `np.where` est prêt ![1]

Batch, mini-batch et stochastique

1. Coder une fonction `accuracy(y,out)` qui renvoie la précision (le pourcentage de concordance, de bonne prédiction) entre deux tenseurs 1d `y` et `out`. On supposera que les deux tenseurs sont réels et qu'un nombre négatif conduit à la prédiction -1, un nombre positif à la prédiction $+1$ (utilisez soit une boucle, soit calcul matriciel `torch.sign`, `torch.cmul`, `torch.eq` et `torch.mean`.)
2. Rappeler la différence entre une descente de gradient batch et une descente de gradient stochastique. Ajoutez à votre code du dernier TP un timer pour chronométrer les deux versions. Comparez les temps d'exécution des deux versions pour arriver à une même valeur de coût.
3. Il est possible de passer un tenseur aux fonctions `forward` et `backward` plutôt que de faire une boucle sur tous les exemples, un à la fois. Faites une deuxième version de l'algorithme batch exploitant cette possibilité. Est-ce que le temps de calcul devrait être amélioré ? Remarquez-vous en pratique une différence ?
4. Une troisième version de la descente du gradient est la version mini-batch : à chaque itération d'une nouvelle époque, la base d'exemples est mélangée aléatoirement et divisée en plusieurs blocs de même taille d'exemples ; puis, le calcul du gradient est fait successivement sur chaque bloc, la mise-à-jour du gradient n'intervenant qu'à la fin du traitement d'un bloc. Cette méthode se situe donc entre le stochastique (mini-batch avec N blocs d'un exemple) et le batch (mini-batch avec 1 bloc de N exemples). Implémentez et comparez les temps de calcul. Discutez de cette approche d'un point de vue théorique et pratique.

Indications :
— Timer : `timer = torch.Timer()`, `timer:time().real` pour avoir le temps passé depuis le déclenchement ;
— module logger : permet de logger dans un fichier des informations et de tracer facilement les graphes associés. `logger = optim.Logger('fichier.log')`, `logger:setNames('accuracy', 'loss', 'time')`, `logger:add(acc,loss,timer:time().real)`, `logger.plot()` pour tracer (possible de manière interactif si placé dans une boucle, utilisez `logger.showPlot = false` pour désactiver).

Train/test

1. Le jeu de données `mnist` fournit également un jeu de test en plus du jeu d'apprentissage. Quel est l'intérêt ?
2. Tracez les courbes de précisions en apprentissage et en test durant l'optimisation. Que remarquez vous ?

1. Toute solution plus simple sera grandement appréciée. :)

3. Augmentez le nombre de données en considérant comme positif et/ou négatif un ensemble de chiffres plutôt qu'un seul chiffre. Comment se comportent les erreurs en apprentissage et en test ?
4. (si vous avez le temps) Implémentez une fonction `split(data,label,s)` qui permet de renvoyer `xtrain,ytrain,xtest,ytest` un découpage en train/test d'un jeu de données avec $s\%$ des données dans le train.

Implémentation d'un critère/fonction de coût

En **Torch**, afin de pouvoir coder des réseaux de neurones génériques, le calcul du gradient est découplé en deux parties, l'un qui concerne le module (grossièrement le type de neurone considéré) et l'autre la fonction de coût (ou critère). Ce découpage est une partie du processus de backpropagation que nous verrons ultérieurement en détail.

Soit $f_{\boldsymbol{\theta}}$ le module, Δ le critère et $\hat{y} = f_{\boldsymbol{\theta}}(x)$; afin d'appliquer une descente de gradient, il faut calculer $\frac{\partial \Delta(f_{\boldsymbol{\theta}}(x),y)}{\partial \theta_i}$ pour tous les paramètres θ_i. En utilisant une dérivation en chaîne (*chain rule*), on a :

$$\frac{\partial \Delta(f_{\boldsymbol{\theta}}(x), y)}{\partial \theta_i} = \frac{\partial \Delta(\hat{y}, y)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \theta_i}$$

Le calcul de $\frac{\partial \hat{y}}{\partial \theta_i}$ est effectué par le module, ce que nous verrons dans les prochaines séances. Le calcul de $\frac{\partial \Delta(\hat{y},y)}{\partial \hat{y}}$ est lui effectué par le critère. Voici le code générique d'un critère :

```
local MyCriterion, parent = torch.class('MyCriterion', 'nn.Criterion') -- heritage en torch

function MyCriterion:__init() -- constructeur
  -- equivalent a  parent.__init(self)
  self.gradInput = torch.Tensor()
  self.output = 0
end

function MyCriterion:forward(input, target) -- appel generique pour calculer le cout
  return self:updateOutput(input, target)
end
function MyCriterion:updateOutput(input, target)  -- a completer
end

function Criterion:backward(input, target)  -- appel generique pour calculer le gradient du cout
  return self:updateGradInput(input, target)
end
function MyCriterion:updateGradInput(input, target) -- a completer
   return self.gradInput
end
```

Implémentez et testez le critère suivant (Huber Loss, ou régression robuste, ou *smooth L1* quand $\delta = 1$) :

$$\Delta_\delta(\hat{y}, y) = \begin{cases} \frac{1}{2}(\hat{y} - y)^2 & \text{si } |\hat{y} - y| \leq \delta \\ \delta|\hat{y} - y| - \frac{1}{2}\delta^2 & \text{sinon} \end{cases}$$

A quoi vous fait penser ce critère ? Quel est l'intérêt ? Vous pouvez implémenter également (ou à la place) le critère modifié de Huber, adapté à la classification :

$$\Delta(\hat{y}, y) = \begin{cases} max(0, 1 - \hat{y}y)^2 & \text{si } 1 - \hat{y}y \leq 2 \\ -4\hat{y}y & \text{sinon} \end{cases}$$