Petit Guide Pratique pour l'implémentation des Deep Neural Networks en torch v0.1 Support du cours AS - Master DAC http://dac.lip6.fr/master

Ludovic Denoyer

21 septembre 2015

1 Définitions

On va considérer que l'on souhaite apprendre une fonction $f_\theta : \mathbb{R}^X \rightarrow \mathbb{R}^Y$ où X est la dimension de l'espace d'entrée et Y est la dimension de l'espace de sortie. Cette fonction sera apprise sur un ensemble $\{(x^i, y^i)\}$ d'exemples d'apprentissage tels que $x^i \in \mathbb{R}^X$ et $y^i \in \mathbb{R}^Y$, avec $i \in [1..N]$, N étant le nombre d'exemples d'apprentissage. La fonction f_θ est aussi appelée prédicteur car, à une entrée $x \in \mathbb{R}^X$, elle est capable d'associer une sortie (ou prédiction) $f_\theta(x)$.

Fonction de coût : On définit une fonction de coût $\Delta : \mathbb{R}^Y \times \mathbb{R}^Y \rightarrow \mathbb{R}$ qui permet de mesurer la qualité d'une prédiction $f_\theta(x)$ par rapport à une sortie désirée y.

Problème d'apprentissage : Le problème d'apprentissage classique consiste à trouver la valeur optimale des paramètres θ^* qui minimise le coùut de prédiction sur l'ensemble des données d'apprentissage :

$$\theta^* = argmin \frac{1}{N} \sum_{i=1}^{N} \Delta(f_\theta(x^i), y^i) \tag{1}$$

Algorithme d'apprentissage : Nous allons utiliser un algorithme d'apprentissage de type descente de gradient (et ses trois variantes "classiques". Cette algorithme consiste à itérativement mettre à jour les paramètres de la manière suivante :

$$\theta \leftarrow \theta - \epsilon \nabla_\theta L(\theta) \tag{2}$$

où $L(\theta)$ est le coût à minimiser :

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} \Delta(f_\theta(x^i), y^i) \tag{3}$$

Les trois variantes de l'algorithme du gradient sont :

Algorithm 1 Algorithme du gradient (batch)

procedure GradientDescent($\{(x^i, y^i)\}, K$) ▷ K est le nombre d'itérations
  $\theta \leftarrow$ random
  $it \leftarrow 0$ ▷ compteur d'itération
  while $it < K$ **do**
    $G \leftarrow 0$
    for $i \in [1..N]$ **do**
      $G \leftarrow G + \nabla_\theta \Delta(f_\theta(x^i), y^i)$
    end for
    $\theta \leftarrow \theta - \epsilon \times G$
  end while
end procedure

Algorithm 2 Algorithme du gradient (stochastique)

procedure GradientDescent($\{(x^i, y^i)\}, K$) ▷ K est le nombre d'itérations
  $\theta \leftarrow$ random
  $it \leftarrow 0$ ▷ compteur d'itération
  while $it < K$ **do**
    for $i \in [1..N]$ **do**
      $j \leftarrow random(1, N)$
      $G \leftarrow \nabla_\theta \Delta(f_\theta(x^j), y^j)$
      $\theta \leftarrow \theta - \epsilon \times G$
    end for
  end while
end procedure

gradient classique : Le gradient de la somme de l'erreur sur tous les exemples est calculé, puis les paramètres sont mis à jour

gradient stochastique : A chaque fois, un exemple est tiré aléatoirement. Le gradient est calculé **pour cet exemple uniquement** et les paramètres sont mis à jour immédiatemment

gradient mini-batch : Le gradient est calculé sur un sous-ensemble aléatoire des données d'apprentissage, puis les paramètres sont mis à jour

2 Implémentation

2.1 Loss :

Un *Loss* correspond à une fonction de coût de type Δ. Elle est définie de la manière suivante :

```
local Criterion = torch.class('nn.Criterion')

function Criterion:__init()
```

Algorithm 3 Algorithme du gradient (minibatch)

procedure GRADIENTDESCENT($\{(x^i, y^i)\}, K, B$) ▷ K est le nombre d'itérations,$B \leq N$ est la taille du minibatch
  $\theta \leftarrow$ random
  $it \leftarrow 0$ ▷ compteur d'itération
  while $it < K$ **do**
    $G \leftarrow 0$
    for $i \in [1..B]$ **do**
      $j \leftarrow random(1, N)$
      $G \leftarrow G + \nabla_\theta \Delta(f_\theta(x^i), y^i)$
    end for
    $\theta \leftarrow \theta - \epsilon \times G$
  end while
end procedure

```
    self.gradInput = torch.Tensor()
    self.output = 0
end

function Criterion:updateOutput(input, target)
 end

function Criterion:forward(input, target)
    return self:updateOutput(input, target)
end
```

La méthode *forward(self,ypredit,y)* correspond au calcul de $\Delta(ypredit, y)$ où y_predit est une prédiction, et y est une valeur désirée.

Exemple : SquareLoss Le square loss est un coût des moindre carrés. Soit $\Delta^{sq} : \mathbb{R}^Y \times \mathbb{R}^Y \to \mathbb{R}$, il se calcule ainsi :

$$\Delta^{sq}(\hat{y}, y) = \frac{1}{Y} \sum_{k=1}^{Y} (\hat{y}_k - y_k)^2 \tag{4}$$

où y_k est la valeur de la k-ème dimension du vecteur y (de dimension Y).

– **Exercice :** Implémentez ce coût en LUA

Plusieurs autres losses existent :

square loss : $\Delta^{sq}(\hat{y}, y) = \frac{1}{Y} \sum_{k=1}^{Y} (\hat{y}_k - y_k)^2$

hinge loss (1) : $\Delta^{hl1}(\hat{y}, y) = \frac{1}{Y} \sum_{k=1}^{Y} max(0, -\hat{y}_k y_k)$

hinge loss (usuel) : $\Delta^{hl}(\hat{y}, y) = \frac{1}{Y} \sum_{k=1}^{Y} max(0, 1 - \hat{y}_k y_k)$

2.2 Module :

Un module correspond à une fonction de type $f_\theta : \mathbb{R}^X \to \mathbb{R}^Y$. Sa définition est la suivante :

```
local Module = torch.class('nn.Module')

function Module:__init()
   self.gradInput = torch.Tensor()
   self.output = torch.Tensor()
end

function Module:parameters()
   --- renvoie les parametres du module
end

function Module:updateOutput(input)
   --- met a jour self.output
end

function Module:forward(input)
   return self:updateOutput(input)
end
```

La méthode *forward(self,x)* est la fonction qui calcule $f_\theta(x)$.

Exemple : Le module linéaire Le module linéaire est le module classique des réseaux de neurones (sans fonction d'activation). Il effectue le calcul suivant :

$$f_\theta \begin{pmatrix} x_1 \\ \vdots \\ x_X \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^{X} \theta_{i,1} x_i \\ \vdots \\ \sum_{i=1}^{X} \theta_{i,Y} x_i \end{pmatrix} \tag{5}$$

C'est une fonction qui correspond à un produit matriciel de type $f_\theta(x) = \theta.x$ où θ est la matrice des $\theta_{i,j}$.

– **Exercice :** Implémentez le module linéaire en LUA

Exemple : Le module tanh Le module *tanh* correspond à un module non-linéaire permettant le calcul de la tangente hyperbolique d'un vecteur. Il s'écrit de la manière suivante :

$$f_\theta \begin{pmatrix} x_1 \\ \vdots \\ x_X \end{pmatrix} = \begin{pmatrix} tanh(x_1) \\ \vdots \\ tanh(x_X) \end{pmatrix} \tag{6}$$

A noter : La dimension d'entrée et la dimension de sortie est la même.

– **Exercice :** Implémentez le module TanH en LUA

3 Un premier réseau de neurone

Nous allons maintenant nous intéresser à l'implémentation des méthodes permettant l'apprentissage des paramètres. Nous allons considérer le cas simple où les données sont transformées par un module $f_\theta(x)$, puis le coût Δ est calculé.

3.1 Rétro-propagation du gradient (partie 1)

Afin de pouvoir mettre à jour les paramètres θ, nous devons calculer $\nabla_\theta \Delta(f_\theta(x), y)$. Intéressons nous à la valeur de ce gradient pour un paramètre $\theta_{i,j}$ (NB : nous prenons pour exemple le cas où les paramètres sont stockés dans une matrice. Vous verrez que pour certains modules, les paramètres sont des vecteurs uniquement, mais ca ne change rien, c'est juste une histoire d'index). Nous souhaitons être capable de calculer :

$$\frac{\partial \Delta(f_\theta(x), y)}{\partial \theta_{i,j}} \tag{7}$$

Notons $\hat{y}$ la sortie $f\theta(x)$ - i.e la sortie du module. La dérivée précédente s'écrit (par la *chain rule*) :

$$\frac{\partial \Delta(f_\theta(x), y)}{\partial \theta_{i,j}} = \frac{\partial \Delta(f_\theta(x), y)}{\partial \hat{y}_j} \frac{\partial \hat{y}_j}{\partial \theta_{i,j}} \tag{8}$$

Si l'on pose $\frac{\partial \Delta(f_\theta(x), y)}{\partial \hat{y}_j} = \delta_j$, on obtient :

$$\frac{\partial \Delta(f_\theta(x), y)}{\partial \theta_{i,j}} = \delta_j \frac{\partial \hat{y}_j}{\partial \theta_{i,j}} \tag{9}$$

Ainsi, le calcul du gradient nécessite :
- De connaître la valeur des δ_j. NB : δ est un vecteur de taille Y
- De savoir calculer : $\frac{\partial \hat{y}_j}{\partial \theta_{i,j}}$

3.1.1 Implémentation

Dans le cas présent, nous allons considérer l'implémentation suivante : le vecteur des δ_j sera calculé **par le Loss**, il sera ensuite transmis au module qui s'occupera de calculer son gradient. Le schéma est donc le suivant :

1. Le module calcule $\hat{y} = f_\theta(x)$ (par l'utilisation de la méthode *forward*)
2. Le loss calcul le vecteur δ par l'utilisation de la méthode *updateGradInput*
3. Le module calcule son gradient par l'utilisation de la méthode *accGradParameters* qui prend en paramètre l'entrée x, mais aussi la valeur de δ qui a été calculée à l'étape 2

Tricks : Au lieu de calculer uniquement le gradient dans la fonction *accGradParameters*, nous allons plutôt faire une mise à jour (par addition) du gradient qui est stocké en mémoire au niveau du module. Cela nous permettra de faire la somme des gradients sur plusieurs exemples d'apprentissage, et donc d'implémenter les différentes variantes des algorithmes de descente de gradient précédemment décrits.

Ce calcul va donc s'effectuer en rajoutant les fonctions suivantes aux classes précédemment décrites :

```
function Criterion:backward(input, target)
   return self:updateGradInput(input, target)
end

function Criterion:updateGradInput(input, target)
end
```

```
function Module:updateGradInput(input, gradOutput)
   return self.gradInput
end

function Module:accGradParameters(input, gradOutput, scale)
end


function Module:backward(input, gradOutput, scale)
   scale = scale or 1
   self:updateGradInput(input, gradOutput)
   self:accGradParameters(input, gradOutput, scale)
   return self.gradInput
end
```

3.1.2 Exercices :

- Calculez δ dans le cas du square loss
- Calculez l'équation 9 dans le cas d'un module linéaire
- Implémentez le tout !

3.1.3 Algorithme du gradient stochastique (en python)

Avec les méthodes implémentées, faire une étape de descente de gradient sur un exemple (x, y) s'écrit :

```
module:zeroGradParameters()
output=module:forward(x)
loss=criterion:forward(output,y)
delta=criterion:backward(output,y)
module:backward(x,delta)
module:updateParameters(learning_rate)
```

4 BackPropagation : le retour

Maintenant, on va considérer l'architecture suivante où deux modules g et f seront mis en série - c'est à dire que les données seront transformées par g puis par f ensuite. La fonction de prédiction devient ici : $f_\theta(g_\gamma(x))$ où γ sont

les paramètres du module g. Nous avons vu comment nous pouvions calculer le gradient sur θ. Mais comment calculer le gradient sur les paramètres du module g ?

Revenons à l'équation décrite précedemment, mas instancieé pour la fonction g :

$$\frac{\partial \Delta(f_\theta(g_\gamma(x)), y)}{\partial \gamma i, j} = \frac{\partial \Delta(f_\theta(g_\gamma(x)), y)}{\partial \hat{y}_j} \frac{\partial \hat{y}_j}{\partial \gamma_{i,j}} \tag{10}$$

où $\hat{y}$ est la sortie calculée par le module g. On voit ici encore que, pour calculer son gradient, le module g nécessite les valeurs :

$$\begin{aligned} \delta_j &= \frac{\partial \Delta(f_\theta(g_\gamma(x)), y)}{\partial \hat{y}_j} \\ &= \frac{\partial \Delta(f_\theta(\hat{y}), y)}{\partial \hat{y}_j} \end{aligned} \tag{11}$$

On va considérer (comme tout à l'heure) que le module g calculera son gradient, mais que les δ seront calculés par le module "suivant" f. Cela suppose donc que le module f est capable de calculer le gradient de l'erreur **par rapport à ses entrées**. Ce calcul est effectué dans la méthode *backward_delta(self,input,delta)*.

Reprenons notre module f_θ dont l'entrée [1] est x et la sortie est notée $\hat{y}$. Nous devons calculer :

$$\frac{\partial \Delta(f_\theta(x), y)}{\partial x_i} = \sum_{k=1}^{Y} \frac{\partial \Delta(f_\theta(x), y)}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial x_i} \tag{12}$$

Ce terme peut se ré-écrire :

$$\frac{\partial \Delta(f_\theta(x), y)}{\partial x_i} = \sum_{k=1}^{Y} \delta_k \frac{\partial \hat{y}_k}{\partial x_i} \tag{13}$$

ATTENTION : Ici, les δ_k correspondent aux δ_k dont nous avons parlé en section 3.1, c'est-à-dire, les δ_k calculés par le module ou le loss qui est "après" le module f. Ces δ_k nous sont "transmis", et nous n'avons pas à les calculer. La valeur $\frac{\partial \hat{y}_k}{\partial x_i}$ dépend de la forme de la fonction f_θ et chaque "tpye" de module devra implémenter sa propre dérivée.

L'algorithme d'apprentissage (SGD) dans le cas à deux modules en série g puis f s'écrit :

```
g=nn.Linear(5,3)
f=nn.Linear(3,1)

g:zeroGradParameters()
f:zeroGradParameters()

output_g=g:forward(x)
```

1. ATTENTION : Ici, on reprend les notations utilisées précédemment, et on oublie le module g qui ne sert qu'à illustrer la rétro-propagation. D'où le léger changement de notations..

```
output_f=f:forward(output_g)

loss=criterion:forward(output_g,y)

delta_g=criterion:backward(output_g,y)
delta_f=g:backward(x,delta_g)

f:updateParameters(learning_rate)
g:updateParameters(learning_rate)
```

Notez que cette algorithme peut aisément se généraliser à n'importe quelle séquence de modules. La fonction *backward* du *loss* sert en fait à initialiser la rétro-propagation.

5 Pour résumer

- **Un loss :**
  - Peut calculer sa valeur à partir de son entrée et de la valeur désirée
  - Il peut calculer sa dérivée par rapport à son entrée
- **Un module :**
  - Peut calculer sa sortie par rapport à son entrée (forward)
  - Peut mettre à jour son gradient à l'aide de la valeur de son entrée, et de la valeur du delta - qui est la dérivée de l'erreur par rapport à sa sortie - qui est calculé/transmis par le module (ou le loss) suivant
  - Peut calculer son delta, c'est-à-dire la dérivée de l'erreur par rapport à son entrée

6 Modules et Fonctions complexes

Ici, nous allons nous intéresser à des modules plus complexes permettant d'implémenter des fonctions plus compliquées.

6.1 Modules n-aires

Nous avons considéré pour l'instant qu'un module prenait en entrée un vecteur et renvoyait un vecteur. Cependant, nous pouvons imaginer des modules qui prennent en entrée des ensembles de vecteurs (ou tenseur) et produisent des ensembles de vecteurs en sortie.

6.1.1 Module Produit Scalaire

Considérons le module **produit_scalaire**. C'est un module défini ainsi :

$$f : \mathbb{R}^N \times \mathbb{R}^N \to \mathbb{R}^1 \quad f(x, x') = \langle x, x' \rangle \tag{14}$$

Exercice : Implémentez le module **produit scalaire**. Dans ce cas, la fonction *foward* prend en entrée un tableau de 2 vecteurs. La méthode *backward* produit quant à elle un tableau de 2 vecteurs correspondant à la dérivée de l'erreur par rapport à x et par rapport à x'

6.1.2 Module Somme

Le module **somme** est un module permettant de calculer la somme de plusieurs vecteurs. Il est donc d'arité variable en entrée :

$$f(x, x', x'', ...) = x + x' + x'' + ... \tag{15}$$

Exercice : Implémentez le module **somme**.

6.2 Module z

Le module **z** est un module paramétrique tel que :

$$f_\theta \rightarrow \theta \tag{16}$$

C'est un module qui ne prend pas d'entrée et renvoie en sortie le vecteur de ses paramètres.

Exercice : Implémentez le module **z**