AS - TP 2
Graphe de calcul, autograd et modules

Nicolas Baskiotis - Benjamin Piwowarski

2019-2020

1 Graphes de fonction

Un élément central de PyTorch est le graphe de calcul : lors du calcul d'une variable, l'ensemble des opérations qui ont servi au calcul sont stockées sous la forme d'un graphe de calcul. Ce graphe est *acyclique*. Les noeuds internes du graphe représentent les opérations, le noeud terminal le résultat et les racines les variables d'entrées. Ce graphe sert en particulier à calculer les dérivées partielles de la sortie par rapport aux variables d'entrées – en utilisant les règles de dérivations chainées des fonctions composées.

Graphe de fonctions Dans ces TPs, nous allons considérer que tout réseau de neurone peut s'exprimer sous la forme d'une composition de fonctions de base (transformation – e.g. linéaire. ; activation – e.g. sigmoïde, softmax ; erreur – e.g. erreur quadratique).

Un exemple est donné figure 1 – pour l'instant, nous ne nous intéressons pas à la nature de ce graphe, mais plus à sa formalisation : nous avons une entrée ($\mathbf{x}$), des paramètres (θ) et trois fonctions (linéaire, transposée, erreur quadratique). La sortie est un scalaire $\ell \in \mathbb{R}$.

La fonction objectif est $L(\mathbf{x}, b_1, \theta, b_2)$ – elle dépend donc de quatre tenseurs (ici des scalaires, vecteurs et matrices). Pour apprendre, il faut savoir calculer le gradient du risque $\nabla_\theta L$ par rapport aux paramètres (ici θ, b_1 et b_2).

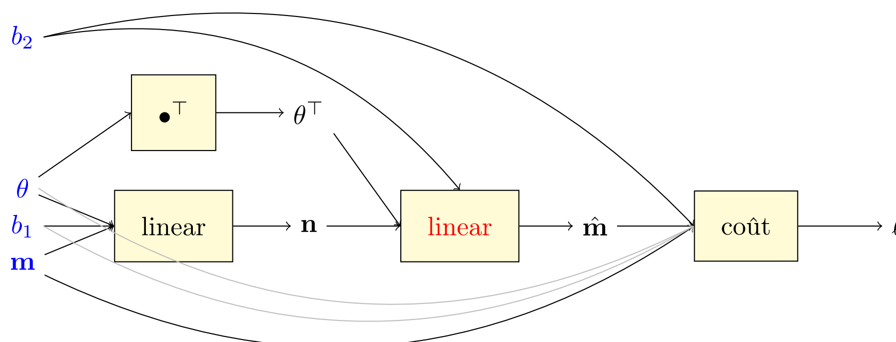

Figure 1 – Un graphe de calcul (auto-encodeur)

Fonction Afin d'abstraire ce graphe de calcul, nous nous plaçons dans le cadre général d'une fonction f comme illustré en figure 2 : pour toute fonction, il y a trois types de paramètres : ceux qui lui sont propres

($\mathbf{x}$), ceux qui sont partagés avec d'autres fonctions du graphe ($\mathbf{y}$) et ceux qui ne pas des entrées de f mais qui interviennent dans le calcul du coût ($\mathbf{z}$).

Toute fonction dans un graphe de calcul peut être vue de cette façon. Par exemple, pour la fonction en rouge de la figure 1, nous avons les correspondances suivantes :

f correspond à $(b_2, \theta, b_1, \mathbf{m}) \mapsto (b_2, \theta^\top, \mathbf{n})$
g correspond à linear
L correspond à coût
$\mathbf{x}$ correspond à $(b_2, \theta, b_1, \mathbf{m})$
$\mathbf{y}$ correspond à $(b_2, \theta^\top, \mathbf{n})$
$\mathbf{z}$ correspond à $\mathbf{m}, \hat{\mathbf{m}}$

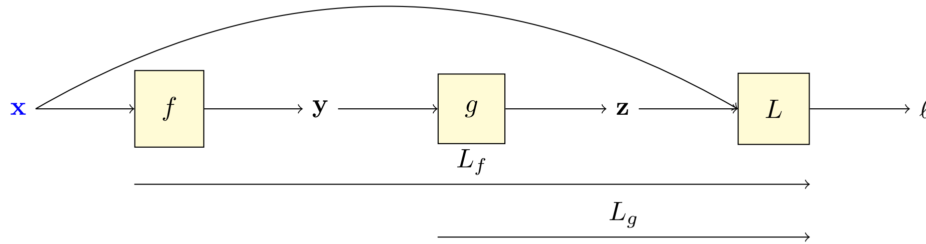

FIGURE 2 – Une fonction g dans le graphe de calcul. f et L représentent l'ensemble des fonctions qui viennent avant/après dans le graphe de calcul.

Afin de comprendre dans la pratique le mécanisme de rétro-propagation, nous nous concentrons sur une fonction g (figure 2). Les entrées $\mathbf{x}$ correspondent aux observations ou bien à des paramètres. Le coût est :

$$L_f(\mathbf{x}) = L(\mathbf{x}, g \circ f(\mathbf{x})) = L_g \circ g(\mathbf{y}) = L(\mathbf{z})$$

Nous notons ∇_a^h la dérivée partielle du coût évalué en L_h, comme par exemple

$$\nabla_{\mathbf{y}_k}^g = \frac{\partial L_g}{\partial \mathbf{y}_k}(\mathbf{y})$$

On suppose que l'on connait $\nabla_{\mathbf{z}_k}^L$ et $\nabla_{\mathbf{x}_k}^L$. En utilisant le théorème de dérivation des fonctions composées, nous avons :

$$\nabla_{\mathbf{x}_k}^f = \nabla_{\mathbf{x}_k}^L + \sum_i \nabla_{\mathbf{y}_i} \times \frac{\partial f_i}{\partial \mathbf{x}_k}(\mathbf{x})$$

$$\nabla_{\mathbf{y}_k} = \sum_i \nabla_{\mathbf{z}_i}^L \times \frac{\partial g_i}{\partial \mathbf{y}_k}(\mathbf{x}, \mathbf{y})$$

En regardant les équations ci-dessus, nous observons que pour calculer les dérivées partielles par rapport à l'ensemble des entrées (en vert), il suffit de remonter l'information depuis les fonctions filles vers leurs parentes et savoir calculer les dérivées partielles des fonctions par rapport à leurs entréees.

En particulier, pour la fonction g, il suffit de connaître $\nabla_{\mathbf{z}_k}^L$ (transmis par la fonction L, en bleu) et la forme analytique des dérivées partielles $\frac{\partial g_i}{\partial \mathbf{y}_k}$ de g par rapport à une de ses entrées. **Autograd** (dans PyTorch) permet de faire cela de manière automatique en enregistrant les fonctions parentes pour chaque calcul effectué.

2 Différenciation automatique : autograd

Comme vu au TP précédent, toute opération sous Pytorch hérite de la classe `torch.nn.Function` et doit définir :

- une méthode forward(*args) : passe avant, calcule le résultat de la fonction appliquée aux arguments
- une méthode backward(*args) : passe arrière, calcule les dérivées partielles par rapport aux entrées . Les arguments de cette méthode correspondent aux valeurs des dérivées suivantes dans le graphe de calcul. En particulier, il y a autant d'arguments à backward que de sorties pour la méthode forward et autant de sorties que d'arguments dans la méthode forward. Le calcul se fait sur les valeurs du dernier appel de forward.

En pratique, ce ne sont pas ces fonctions qui sont directement utilisées, mais un encapsulage de ces fonctions dans les tenseurs. A chaque fois qu'une opération est exécutée sur un tenseur, le graphe de calcul est calculé à la volée et stocké dans le tenseur résultant. La méthode `backward` d'un tenseur permet de rétropropager le calcul du gradient sur toutes les variables qui ont servies à son calcul ; la valeur du gradient pour chaque dérivée partielle se trouve dans l'attribut `grad` de la variable concernée.

Toutefois, comme le graphe de calcul est coûteux en ressource, il faut spécifier manuellement que l'on souhaite le calculer. Ceci est fait par l'intermédiaire de l'attribut booléen `requires_grad` des tenseurs (il suffit de le spécifier pour les variables racines pour que l'information soit propagée). De plus, les gradients intermédiaires ne sont pas conservés par défaut pour des raisons d'optimisation mémoire. Si on souhaite les conserver, il faut appeler la méthode `retain_grad()` sur la variable concernée.

Par ailleurs, il est possible de spécifier que pour un bloc d'exécution donné le gradient n'aura pas à être calculer et ainsi désactiver le graphe de calcul grâce à l'instruction `with torch.no_grad()`.

```
a = torch.rand((1,10),requires_grad=True)
b = torch.rand((1,10),requires_grad=True)
c = a.mm(b.t())
d = 2 * c
c.retain_grad() # on veut conserver le gradient par rapport à c
d.backward()  ## calcul du gradient et retropropagation jusqu'aux feuilles du graphe de calcul
print(d.grad) #Rien : le gradient par rapport à d n'est pas conservé
print(c.grad) # Celui-ci est conservé
print(a.grad) ## gradient de c par rapport à a qui est une feuille
print(b.grad) ## gradient de c par rapport à b qui est une feuille

with torch.no_grad():
    c = a.mm(b.t())
c.backward() ## Erreur
```

Implémentation Implémenter un algorithme du gradient pour la régression linéaire en utilisant les fonctionnalités de la différenciation automatique.

3 Optimiseur

Pytorch inclut une classe très utile pour la descente de gradient, `torch.optim`, qui permet :

- d'économiser quelques lignes de codes
- d'automatiser la mise-à-jour des paramètres
- d'abstraire le type de descente de gradient utilisé (SGD,Adam, rmsprop, . . .)

Une liste de paramètres à optimiser est passée à l'optimiseur lors de l'initialisation. La méthode `zero_grad()` permet de remettre le gradient à zéro et la méthode `step()` permet de faire une mise-à-jour des paramètres. L'exemple ci-dessous permet de mettre à jour les paramètres qu'une fois toutes les 100 itérations (mini-batch de 100).

```
w = torch.nn.Parameter(torch.randn(1,10))
b = torch.nn.Parameter(torch.randn(1))
optim = torch.optim.SGD(params=[w,b],lr=EPS) ## on optimise selon w et b, lr : pas de gradient
optim.zero_grad()
# Reinitialisation du gradient
    for i in range(NB_EPOCH):
    loss = MSE(f(x,w,b),y) #Calcul du cout
    loss.backward()        # Retropropagation
    if i % 100 = 0:
        optim.step()           # Mise-à-jour des paramètres w et b
        optim.zero_grad()      # Reinitialisation du gradient
```

La classe `Parameter` est un wrapper de la classe `Tensor` qui permet entre autre de spécifier automatiquement que le gradient est requis pour ce tenseur et également de noter ce tenseur comme un paramètre à optimiser. Cette différenciation est utilisée essentiellement dans la classe `Module` afin de reconnaître automatiquement les paramètres des autres entrées/constantes.

4 Module

Dans le framework PyTorch (et dans la plupart des frameworks analogues), le module est la brique de base qui permet de construire un réseau de neurones. Il permet de représenter en particulier :

- une couche du réseau (linéaire : `torch.nn.Linear`, convolution : `torch.nn.convXd`, ...)
- une fonction d'activation (tanh : torch.nn.Tanh, sigmoïde : torch.nn.Sigmoid , ReLu : torch.nn.ReLu, ...)
- une fonction de coût (MSE : `torch.nn.MSELoss`, L1 : `torch.nn.L1Loss`, CrossEntropy : `torch.nn.CrossEntropyLoss`, ...)
- mais également des outils de régularisation (BatchNorm : `torch.nn.BatchNorm1d`, Dropout : `torch.nn.Dropout`, ...)
- un ensemble de modules ; en termes informatique, un module est un conteneur abstrait qui peut contenir d'autres conteneurs : plusieurs modules peuvent être mis ensemble afin de former un nouveau module plus complexe.

Le fonctionnement est très proche des fonctions : un module encapsule en fait une fonction héritée de `torch.nn.Function` mais de manière à gérer automatiquement les paramètres à apprendre. La classe `Parameter` est utilisée pour créer un paramètre du module. Le paramètre ainsi créé est automatiquement ajouté à la liste des paramètres du module. La liste des paramètres est ensuite accessible par la méthode `parameters`. Tout comme une fonction, le module est muni :

- d'une méthode `forward` qui permet de calculer la sortie du module à partir des entrées
- d'une méthode `backward` qui permet d'effectuer la rétro-propagation (localement).

Dans le cas où la méthode `forward` ne fait que des calculs à l'aide de fonctions disponibles sous `PyTorch`, la méthode `backward` n'a pas besoin d'être implémentée ! En effet, la rétropropagation peut être gérée par la différenciation automatique.

Implémentation Utiliser les modules `torch.nn.Linear`, `torch.nn.Tanh` et `torch.nn.MSELoss` pour implémenter un réseau à deux couches : `lineaire` → `tanh` → `lineaire` → `MSE`. Implémenter la boucle de descente de gradient avec l'optimiseur.

Utiliser maintenant un conteneur - par exemple le module `torch.nn.Sequential` - pour implémenter le même réseau. Parcourer la doc pour comprendre la différence entre les différents types de conteneurs. Que se passe-t-il pour les paramètres des modules mis ainsi ensemble ?