PSTALN_TP01bis_pytorchtensors

December 12, 2021

1 Pytorch tensors

`pytorch` est une librairie de fonctions sur les tenseurs avec la plupart des fonctionnalités de numpy mais une api un peu différente. La documentation de référence est dispo en ligne : https://pytorch.org/docs/.

Cette documentation n'est pas très avenante mais il est souvent utile de s'y référer lorsque l'on veut obtenir des résultats un peu exotiques. Ce notebook présente quelques fonctionnalités pratiques.

Commençons par créer une matrice d'entiers aléatoires de taille (3, 6), et afficher sa transposée :

[1]:
```
import torch
import torch.nn as nn

x = (torch.rand(3, 6) * 20).long()
print(x)
print(x.t())
```

```
tensor([[13,  2, 14, 17,  7, 16],
        [ 8,  1, 15, 17, 14,  4],
        [17,  5, 10, 11,  9, 18]])
tensor([[13,  8, 17],
        [ 2,  1,  5],
        [14, 15, 10],
        [17, 17, 11],
        [ 7, 14,  9],
        [16,  4, 18]])
```

Cette matrice peut être redimensionnée en un tenseur du même nombre d'étélments mais avec d'autres dimensions en utlisant `view()`. Si on remplace une dimension par -1, celle ci est calculée automatiquement. La fonction `transpose()` permet de transposer deux dimensions en particulier; la fonction `size()`, ou `shape` dans les versions récentes de pytorch, donne les dimensions du tenseur.

[2]:
```
y = x.view(3, 2, 3)
print(y)
print(y.view(9, -1))
print(y.transpose(0, 1).shape)
```

```
tensor([[[13,  2, 14],
         [17,  7, 16]],
```

```
        [[ 8,  1, 15],
         [17, 14,  4]],

        [[17,  5, 10],
         [11,  9, 18]]])
tensor([[13,  2],
        [14, 17],
        [ 7, 16],
        [ 8,  1],
        [15, 17],
        [14,  4],
        [17,  5],
        [10, 11],
        [ 9, 18]])
torch.Size([2, 3, 3])
```

La fonction `size(d)` peut aussi donner la taille d'un axe en particulier, et la fonction `numel` renvoie le nombre d'éléments du tenseur.

[3]:
```
print(y.shape[1])
print(y.numel())
```

```
2
18
```

Il est possible de convertir un tenseur vers une représentation numpy et inversement.

[4]:
```
numpy_y = y.numpy()
torch.from_numpy(numpy_y)
```

[4]:
```
tensor([[[13,  2, 14],
         [17,  7, 16]],

        [[ 8,  1, 15],
         [17, 14,  4]],

        [[17,  5, 10],
         [11,  9, 18]]])
```

On peut ajouter des dimensions de taille 1 avec `unsqueeze()` et les supprimer avec `squeeze()`

[5]:
```
y3 = y.unsqueeze(2)
print(y3.size())
print(y3.squeeze().size())
```

```
torch.Size([3, 2, 1, 3])
torch.Size([3, 2, 3])
```

Les fonctions `cat` et `split` permettent de concatèner tenseurs ou subdiviser

un tenseur selon une dimension. La documentation donne les paramètres : https://pytorch.org/docs/stable/torch.html#torch.split

[6]:
```
a = torch.rand(2, 3)
b = torch.zeros(2, 3)
print(a, b)
c = torch.cat([a, b], 1)
print(c)
d, e = torch.split(c, 1, 0)
print(d, e)
```

```
tensor([[0.8500, 0.1563, 0.8163],
        [0.9280, 0.7575, 0.8666]]) tensor([[0., 0., 0.],
        [0., 0., 0.]])
tensor([[0.8500, 0.1563, 0.8163, 0.0000, 0.0000, 0.0000],
        [0.9280, 0.7575, 0.8666, 0.0000, 0.0000, 0.0000]])
tensor([[0.8500, 0.1563, 0.8163, 0.0000, 0.0000, 0.0000]]) tensor([[0.9280,
0.7575, 0.8666, 0.0000, 0.0000, 0.0000]])
```

Comme `numpy`, pytorch permet de broadcaster des tenseurs de dimensions différentes (voir https://pytorch.org/docs/stable/notes/broadcasting.html).

[7]:
```
# broadcasting
x = torch.rand(3, 1)
y = torch.rand(3, 2)
print(x)
print(y)
print(x + y)
```

```
tensor([[0.3197],
        [0.5943],
        [0.0036]])
tensor([[0.1273, 0.6515],
        [0.9275, 0.6294],
        [0.5487, 0.9294]])
tensor([[0.4470, 0.9712],
        [1.5218, 1.2237],
        [0.5523, 0.9330]])
```

On peut faire des multiplications de matrices avec la fonction `matmul`. Lorsque l'on a des tenseurs d'ordre 3 contenant des batches de matrices, on peut faire la multiplication de matrice batch par batch avec la fonction `bmm`.

[8]:
```
x = torch.rand(3, 2)
y = torch.rand(2, 3)
print(x.matmul(y))

x = torch.rand(2, 3, 2)
y = torch.rand(2, 2, 3)
```

```
print(x.bmm(y))
```

```
tensor([[0.0532, 0.5248, 0.6761],
        [0.0555, 0.8164, 1.1968],
        [0.0482, 0.6527, 0.9365]])
tensor([[[1.1692, 1.2118, 0.1688],
         [0.4436, 0.3515, 0.0747],
         [0.0892, 0.0812, 0.0140]],

        [[0.6081, 0.2735, 0.6336],
         [0.8064, 0.6328, 0.6444],
         [0.8184, 0.3926, 0.8350]]])
```

L'utililisation du GPU est simple mais elle n'est pas automatique. Il y a plusieurs méthodes pour transferer un tenseur vers le GPU. On peut soit utiliser les types cuda (`torch.cuda.FloatTensor`), créer un tenseur sur CPU puis le migrer vers le gpu (`y = x.cuda()`) ou alors utiliser `device = torch.device('cuda')` puis migrer le tenseur avec `y = x.to(device)`. Le retour vers cpu se fait grâce à la fonction `x.cpu()`. Lorsqu'on applique une opération sur des tenseurs, ces derniers doivent être sur le même dispositif (cpu ou gpu) et le résultat est généré sur le même dispositif.

Le passage CPU / GPU est coûteux, donc il faut éviter de le faire trop souvent. On peut par exemple copier toutes les données sur GPU en début d'apprentissage, ou alors le faire à chaque batch dans la boucle d'apprentissage.

En plus d'envoyer les données, il faut appeler `model.cuda()` ou `model.to(device)` pour placer également les paramètres du modèle sur GPU.

Pour les systèmes multi-GPU, il est possible de choisir le GPU dans une clause `with` ou d'utiliser plusieurs `torch.device` avec comme paramètre `cuda:0`, `cuda:1`…

[9]:
```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.l1 = nn.Linear(10, 2)
    def forward(self, x):
        return self.l1(x)

model = Model()
model.to(device)

result = model(torch.rand(3, 10))
print(result.data)

numpy_result = result.data.cpu().numpy()

print(numpy_result)
```

```
if torch.cuda.is_available():
    with torch.cuda.device(1):
        x = torch.rand(3, 3).cuda()
        print(x.get_device())
```

```
tensor([[-0.6225, -0.8150],
        [-0.1379, -0.9843],
        [-0.5546, -0.9633]])
[[-0.62250274 -0.815043  ]
 [-0.13787398 -0.98430896]
 [-0.55464745 -0.9633362 ]]
```

pytorch permet de sauvegarder les modèles de deux manières. La première ne sauvegarde que les paramètres du modèle et est donc plus portable (néanmoins ces derniers ne peuvent être chargés que par pytorch). La seconde sauvegarde l'objet python contenant le modèle (un peu comme `pickle`) et le recharge directement. C'est plus pratique mais ne fonctionne que si on est dans le même répertoire avec le même code.

[10]:
```
# approche recommandée
torch.save({'state_dict': model.state_dict()}, "model_weights.pt")

model = Model()
checkpoint = torch.load("model_weights.pt")
model.load_state_dict(checkpoint['state_dict'])

# approche non recommandée
torch.save(model, "full_model.pt")
model = torch.load("full_model.pt")
```

On peut aussi convertir les paramètres du modèle en objet numpy pour utilisation par exemple dans un autre langage. Il faut toutefois réimplémenter les mêmes opérations que dans `pytorch`.

[11]:
```
numpy_params = {name: value.cpu().numpy() for name, value in model.state_dict().
 ↪items()}
for name, param in numpy_params.items():
    print(name, param.shape)
    print(param)

print('total number of parameters:', sum([p.numel() for p in model.
 ↪parameters()]))
```

```
l1.weight (2, 10)
[[-0.23444949  0.15453656 -0.2779888   0.1453526   0.01721119  0.1332549
  -0.17579079  0.23618802 -0.16972671  0.26256993]
 [-0.05555913 -0.28719866  0.14800663 -0.24426146  0.23201643 -0.12804502
  -0.29238114 -0.23149183 -0.2761161  -0.15984961]]
l1.bias (2,)
[-0.22496673 -0.28234628]
```

```
total number of parameters: 22
```

1.1 Exercice 1

Avec l'aide de la documentation pytorch, calculez l'expression suivante :

$$a = 1_{(3\times 2)} b = sin(1 + \sqrt{3I_2} + 5||a||)$$

où a est une matrice de taille (3, 2) contenant des 1, I_2 est la matrice identité de taille (2, 2) et $||\cdot||$ est la norme 2 (norme euclidienne).

[ ]:

1.2 Exercice 2

1. Combien de paramètres a le modèle MLP du notebook précédent ?
2. Assurez-vous de faire tourner le notebook précédent avec une acceleration GPU.

[ ]:

1.3 Exercice 3

Considerons des pseudo-textes de longueur 5, associés chacun à une classe parmis 7. Le vocabulaire est de taille 13, donc les mots sont représentés par des entiers entre 0 et 12. Exemple d'un tel texte : $x = [6, 2, 11, 3, 6]$, $y = 4$.

On souhaite créer un classifieur pour ce type de données. Il prendra en entrée les séquences d'entiers des textes, puis les convertira en vecteurs de taille 3 grâce à une couche d'embedding partagée entre toutes les positions. Enfin, ces embeddings seront concaténés et passés à une couche de décision pour produire la classe associée. Note : utilisez la couche Embedding fournie par pytorch.

$e_i = Embed(x_i)$

$\hat{y} = softmax(W[e_0, e_1, e_2, e_3, e_4] + b)$

1. Générer un tenseur aléatoire X contenant 1000 textes définis selon ces critères, et un tenseur aléatoire Y contenant des étiquettes associées.
2. Écrire une classe dérivant de `nn.Module` pour implémenter le réseau de neurones proposé.
3. Découper les instances en deux et entraînez le modèle sur la première moitié en minimisant l'entropie croisée.
4. Quel est le taux d'erreur obtenu ?

[ ]: