PyTorch

Introduction à l'apprentissage automatique – GIF-4101 / GIF-7005

Professeur : Christian Gagné

Semaine 10

UNIVERSITÉ LAVAL

10.8 Notions de base sur PyTorch

PyTorch

- Librairie de différentiation automatique pour l'apprentissage profond
- Début octobre 2016 par une équipe de Facebook
- Construit par dessus le moteur C de Torch
- Plus *pythonesque* que TensorFlow
- Très proche de la syntaxe de numpy
- Supporte les calculs par GPU (extrêmement rapide, facteur 10 !)
- Supporte les graphes dynamiques
- Stable et utilisable pour déploiement à grande échelle

Concept de tenseur

- Pytorch est organisé autour d'opérations de manipulation de tenseurs, incluant dérivation automatique
  - Créer un tenseur à partir d'une liste avec `torch.<type>Tensor()`

```
import torch
a = torch.FloatTensor([[1,2,3], [2,3,4]])
```

  - Créer un tenseur aléatoire

```
a = torch.randn(2, 3)
print(a)
```

```
>> tensor([[ 0.0991, -0.8607,  0.8124],
          [ 2.1726,  0.7590, -0.2185]])
```

  - Créer un tenseur à partir d'un array Numpy

```
a = torch.from_numpy(numpy_array)
```

Concept de tenseur

- On peut effectuer toute sorte d'opérations sur les tenseurs

```
a = torch.FloatTensor([[1,2,3], [2,3,4]])
b = torch.FloatTensor([[4,3,3], [5,3,4]])
c = a + b
print(c)
```

```
>> tensor([[5., 5., 6.],
          [7., 6., 8.]])
```

- Liste complète ici : `https://pytorch.org/docs/stable/torch.html`

Dérivation automatique

- Durant l'application des opérations, PyTorch se construit un graphe de calcul
  - Ce graphe permet de suivre toutes les opérations nécessaires au calcul du résultat
- Ensuite facile de calculer automatiquement la dérivée à chaque étape du graphe
  - Pour indiquer le calcul de la dérivée par rapport à un certain tenseur, utiliser paramètre `requires_grad`

```
a = torch.FloatTensor([[1,2,3], [2,3,4]], requires_grad=True)
```

  - Ou bien une fois le tenseur existant

```
a.requires_grad = True
```

Exemple d'une régression linéaire (1/2)

- Déclarer vecteur de poids et un biais aléatoire

```
# 10 dimensions
W = torch.randn(10, requires_grad=True)
b = torch.randn(1, requires_grad=True)
```

- Executer la chaîne d'opération (très proche de numpy)

```
# y_hat est la sortie prédite, x l'entrée
y_hat = W.dot(x) + b
```

- Calculer l'erreur quadratique

```
# y est la sortie désirée
err = 0.5 * (y_hat - y) ** 2
```

Exemple d'une régression linéaire (2/2)

- Dériver l'équation à l'aide de la méthode `backward()`

```
err.backward()
```

- On peut alors récupérer dérivées dans les tenseurs W et b

```
W_grad = W.grad
b_grad = b.grad
```

- Faire un pas dans la bonne direction pour descendre le gradient

```
W = W - eta * W.grad
b = b - eta * b.grad
```

Exécution sur GPU

- Possible de faire facilement toutes les opérations sur les tenseurs sur un GPU
  - PyTorch définit tenseurs `torch.cuda.<type>Tensor` de la même manière que ceux vus préalablement.
  - Pour traduire un tenseur d'un type non-GPU (non-cuda) à un type GPU (cuda) et inversement, il suffit d'utiliser la méthode `to` :

```
a = a.to('cuda')  # vers le GPU
a = a.to('cpu')   # de retour sur le CPU
```

10.9 Définir un réseau

Définir un réseau

- PyTorch offre une manière de déclarer facilement des réseaux
  - Définir réseau avec tenseurs directement serait une tâche ardue
  - Typiquement on utilise le package `torch.nn` et on hérite de `nn.Module`

```
import torch.nn as nn
class MonReseau(nn.Module):
    def __init__(self):
        super().__init__()
        # on définit la structure du réseau ici
        # - les couches
        # - les opérations non-linéaires
        # - les méthodes de régularisation
    def forward(self, x):
        # on effectue l'inférence ici
```

Définir un réseau

- Plusieurs types de couches sont offertes
  - Composition de modules simples pour créer des modules plus complexes
- Exemples de modules de base
  - Linéaire

```
torch.nn.Linear(in_features, out_features, bias=True)
```

  - Convolution 2D

```
torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1,
                padding=0, dilation=1, groups=1, bias=True)
```

  - Dropout

```
torch.nn.Dropout(p=0.5, inplace=False)
```

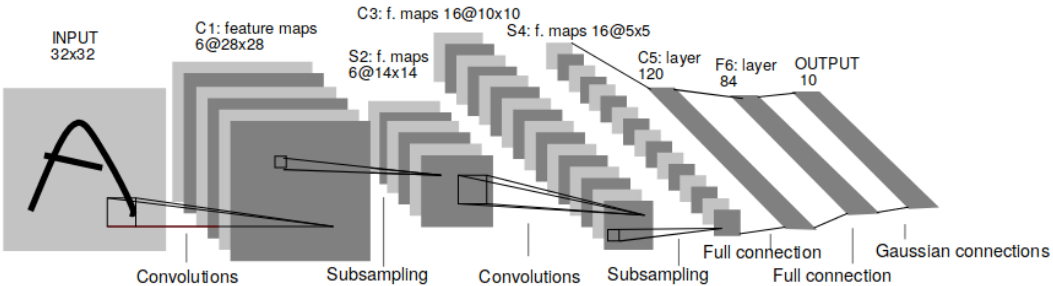
- Voir `https://pytorch.org/docs/stable/nn.html` pour plus de détails

Définir un réseau

- Plupart des couches sont également disponibles en fonctions à partir de `torch.nn.functional`
  - Attention : le module n'enregistre pas ces couches lorsqu'elles sont déclarées directement en fonction
  - Paramètres de ces couches ne sont pas pris en compte dans la liste des paramètres
  - Certaines couches comme `dropout` ou `batchnorm` ont des comportements différents en entraînement et en test, changer mode du réseau change le comportement d'une couche classe, mais pas d'une couche fonction
- Vaut mieux donc utiliser couches fonctions seulement lorsque la couche n'a pas de paramètres à optimiser et/ou même comportement entre entraînement et test (ex. fonction d'activation)

Définir un réseau

Supposons le réseau Lenet-5



Tiré de *Y. LeCun, L. Bottou, Y. Bengio et P. Haffner, Gradient-based learning applied to document recognition, Proceedings of the IEEE, 86(11), 1998.* Accédé en ligne le 6 novembre 2020 au `http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf`.

Définir un réseau

- Implémentation PyTorch du réseau Lenet-5 pour un jeu de données avec image sur un *channel* (tenseur 2D)

```
import torch.nn as nn
import torch.nn.functional as F
class Lenet5(nn.Module):
    def __init__(self):
        super().__init__()
        self.C1 = nn.Conv2d(1, 6, kernel_size=5)
        self.S2 = nn.MaxPool2d(2)
        self.C3 = nn.Conv2d(6, 16, kernel_size=5)
        self.S4 = nn.MaxPool2d(2)
        self.C5 = nn.Linear(16*4*4, 120)
        self.F6 = nn.Linear(120, 64)
        self.output = nn.Linear(64, 10)

        [...]
```

Définir un réseau

```
        [...]

    def forward(self, x):
        y = self.S2(F.relu(self.C1(x)))
        y = self.S4(F.relu(self.C3(y)))
        y = y.view(-1, 16*4*4) # redimensionne
        y = F.relu(self.C5(y))
        y = F.relu(self.F6(y))
        return self.output(y)
```

Définir un réseau

- De la même manière, il est très facile d'envoyer un réseau sur le GPU avec la méthode `to` :

```
model = Lenet5()
model.to('cuda') # vers le GPU
model.to('cpu')  # de retour sur le CPU
```

- Il est également possible de changer le mode du réseau, ce qui changera le comportement de certaines couches, comme ceci :

```
model = Lenet5()
model.train() # en mode entraînement
model.eval()  # en mode test
```

10.10 Manipuler les données

Charger et manipuler des données

- Classe pour gérer les jeux de données :

```
torch.utils.data.Dataset
```

  - Doit définir une méthode `__getitem__(self, index)` pour accéder à une instance
  - Doit définir une méthode `__len__(self)` pour retourner la taille du jeu de données

- Classe pour charger des lots de données :

```
torch.utils.data.DataLoader
```

  - Doit recevoir un objet `Dataset` et une `batch_size`, d'autres arguments permettent des options avancées
  - `DataLoader` est un itérateur python

Charger et manipuler des images

- Sous-package `torchvision` implémente plusieurs fonctions utiles pour vision numérique et traitement d'images
  - `torchvision.datasets` permet de télécharger plusieurs jeux de données populaires tels que MNIST, CIFAR ou encore SVHN
  - `ImageFolder` et `DatasetFolder` permettent de charger facilement un jeu de données organisées en répertoires
  - `torchvision.transforms` implémente des transformations sur les images
  - `ToTensor` permet de convertir en un tenseur PyTorch
  - `Normalize` permet de normaliser un tenseur PyTorch
- Plusieurs autres fonctions disponibles, voir `https://pytorch.org/vision/stable/datasets.html`

Exemple avec MNIST

```python
from torch.utils.data import DataLoader
from torchvision.datasets import MNIST
from torchvision.transforms import ToTensor

batch_size = 64

# télécharge dans 'path/to/data'
train_set = MNIST('path/to/data', train=True, transform=ToTensor(), download=True)
train_loader = DataLoader(train_set, batch_size=batch_size, shuffle=True)
```

10.11 Entraîner un réseau

Entraîner un réseau

- Une fois les données chargées, il faut un optimiseur et fonction d'erreur pour faire un entraînement
  - Optimiseurs dans `torch.optim`
  - Fonctions d'erreurs dans `torch.nn`, comme les couches
- Par exemple, pour effectuer du classement à plusieurs classes, nous pourrions utiliser :
  - Optimiseur par descente du gradient stochastique `torch.optim.SGD`
  - Entropie croisée `torch.nn.CrossEntropyLoss`

Entraîner un réseau

- Entraînement de LeNet-5 en classification

```
nb_epoch = 10
batch_size = 64
learning_rate = 0.01
momentum = 0.9
# télécharge dans 'path/to/data'
train_set = MNIST('path/to/data', train=True, transform=ToTensor(),
                  download=True)
train_loader = DataLoader(train_set, batch_size=batch_size, shuffle=True)
model = Lenet5()
model.train() # mettre en mode entraînement
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate,
                            momentum=momentum)
criterion = torch.nn.CrossEntropyLoss()

[...]
```

Entraîner un réseau

```
[...]

for i_epoch in range(nb_epoch):
    for i_batch, batch in enumerate(train_loader):
        X, y = batch
        optimizer.zero_grad()        # important! remet les gradients à 0
        y_hat = model(X)             # calcule la prédiction
        loss = criterion(y_hat, y)   # calcule l'erreur
        loss.backward()              # dérive le graphe
        optimizer.step()             # effectue une étape d'optimisation
```

Utiliser un réseau pré-entraîné

- Possible de sauvegarder un réseau via son dictionnaire d'état (`state_dict`) et fonction `torch.save`

```
state = model.state_dict()
torch.save(state, 'path/to/model')
```

- De la même manière, il est possible de charger un modèle pré-entraîné avec la fonction `torch.load` et la méthode `load_state_dict`

```
state = torch.load('path/to/model')
model.load_state_dict(state)
```

- Il est prudent de charger un réseau avec une indication sur la destination pour être sûr qu'il se retrouve premièrement sur le CPU

```
state = torch.load('path/to/model', map_location=lambda storage, loc: storage)
```

- Plus de détails : `https://bit.ly/2Pu0Ibm`

Utiliser un réseau pré-entraîné

- Sous-package `torchvision.models` implémente plusieurs modèles utiles aux tâches de vision.
- Peuvent être chargés avec des poids pré-entraînés sur l'énorme jeu de données d'images naturelles ImageNet
- Par exemple, il est possible de charger un ResNet-18 avec les poids pré-entraîné comme suit :

```
from torchvision.models import resnet18
model = resnet18(pretrained=True)
```

Utiliser un réseau pré-entraîné

- Accès aux paramètres du réseau avec leur nom de couche avec la méthode `named_parameters()`
  - Ainsi, il est possible d'analyser le réseau

```
for name, param in model.named_parameters():
    print(name)
    print(param.grad)
```

  - De modifier un réseau

```
model.nom_de_couche = NouvelleCouche()
```

  - Et de geler des couches :

```
for name, param in model.named_parameters():
    if name == nom_de_couche_a_geler:
        param.requires_grad = False
```

Utiliser un réseau pré-entraîné

- Si vous geler des couches, il est alors important de seulement donner les paramètres devant être optimisés à l'optimiseur

```
params = filter(lambda x: x.requires_grad, model.parameters())
optimizer = torch.optim.SGD(params, lr=learning_rate, momentum=momentum)
```