AMAL - TP 3
Définition de fonctions en `pyTorch`

Nicolas Baskiotis - Benjamin Piwowarski - Laure Soulier

2020-2021

Dans ce TME, nous finissons notre revue de PyTorch en montrant comment :
— définir des classes permettant de représenter les données d'apprentissage ;
— passer sur GPU (**attention** : cela ne marchera que si vous avez une carte NVidia)
— faire en sorte de pouvoir reprendre un apprentissage après une interruption volontaire ou non (checkpointing)

Nous utiliserons donc dès ce TME toutes les possibilités offertes par PyTorch, et ce cadre de travail restera le même jusqu'à la fin de ce module. Ne pas oublier d'installer `datamaestro` et `datamaestro_image` si ce n'est déjà fait.

1 Gérer les données avec `Dataset` et `Dataloader`

Les classes `Dataset` et `Dataloader` permettent de faciliter la gestion des données sous `PyTorch`. La classe `Dataset` est une classe abstraite qui permet de préciser comment un exemple est chargé, pré-traité, transformé etc et donne accès par l'intermédiaire d'un itérateur à chaque exemple d'un jeu de données. La classe `Dataloader` encapsule un jeu de données et permet de requêter de diverses manières ce jeu de données : spécifier la taille du mini-batch, si l'ordre doit être aléatoire ou non, de quelle manière sont concaténés les exemples, etc.

Pour implémenter un `Dataset`, il suffit de définir deux méthodes `__getitem__(self,index)` et `__len__(self)` :

```
from torch.utils.data import Dataset, Dataloader
class MonDataset(Dataset):
    def __init__(self,...):
        pass
    def __getitem__(self,index):
        """ retourne un couple (exemple,label) correspondant a l'index """
        pass
    def __len__(self):
        """ renvoie la taille du jeu de donnees """
        pass
```

des principaux avantages (mis à part le pré-traitement possible) est qu'il n'est pas nécessaire de charger tout le jeu de données en mémoire, le chargement se fait à la volée. Par ailleurs, la classe pré-existente `TensorDataset` permet de construire un dataset pour une liste de tenseurs passée en argument (le i-ème exemple est un n-uplet composé de la i-ème ligne de chaque tenseur).

Une fois un dataset `MonDataset` implémenté, il suffit de créer un `DataLoader` de la manière suivante par exemple :

```
# Creation du dataloader, en specifiant la taille du batch et ordre aleatoire
data = DataLoader(MonDataset(...), shuffle=True, batch_size=BATCH_SIZE)
for x,y in data:
    #x,y est un batch de taille BATCH_SIZE
```

Question 1

Implémenter un dataset pour le jeu de données MNIST qui renvoie une image sous la forme d'un vecteur normalisé entre 0 et 1, et le label associé (sans utiliser `TensorDataset`). Tester votre dataset avec un dataloader pour différentes tailles de batch et explorer les options du dataloader.
Vous pouvez vous référer à la doc officielle.

Pour charger MNIST :

```
#pip install datamaestro datamaestro_image
from datamaestro import prepare_dataset
ds = prepare_dataset("com.lecun.mnist");
train_img, train_labels = ds.train.images.data(), ds.train.labels.data()
test_img, test_labels =  ds.test.images.data(), ds.test.labels.data()
```

2 Implémentation d'un autoencodeur

Un autoencodeur est un réseau de neuronnes qui permet de projeter (*encoder*) un jeu de données dans un espace de très petite dimension (il *compresse* le jeu de données). La dimension de sortie est la même que l'entrée, il est entrainé de manière à ce que la sortie soit la plus proche possible que l'entrée - $f(\mathbf{x}) \approx \mathbf{x}$ - avec un coût aux moindres carrés par exemple ou un coût de cross entropie. On appelle *décodage* le calcul de la sortie à partir des données projetées.

Question 2

Implémenter une classe autoencodeur (héritée de Module) selon l'architecture suivante : *linéaire* $\rightarrow$ *Relu* pour la partie encodage et *linéaire* $\rightarrow$ *sigmoïde* pour la partie

décodage. Les poids du décodeur correspondent usuellement à la transposée des poids de l'encodeur (quel est l'avantage ?).

3 GPU

Afin de profiter de la puissance de calcul d'un GPU, il faut obligatoirement spécifier à `PyTorch` de charger les tenseurs sur le GPU ainsi que le module (i.e. les paramètres du module). Il n'est pas possible de faire des opérations lorsqu'une partie des tenseurs est sur GPU et l'autre sur CPU (un message d'erreur s'affiche dans ce cas). L'opérateur `to(device)` des tenseurs et des modules permet de les copier sur le GPU (ou CPU) spécifié (attention, l'opération n'est pas *inplace*, il faut sauver le résultat dans une variable qui elle sera sur le bon device). Ci-dessous un exemple :

```
#permet de selectionner le gpu si disponible
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = Autoencodeur(...)
model = model.to(device) #chargement du module (des parametres) sur device
x = x.to(device) # charge les donnees sur device
y = model(x) # calcul gpu
y = y.to(device='cpu') # si on veut remettre sur cpu
```

4 Checkpointing

Il est souvent utile de sauvegarder au fur et à mesure de l'apprentissage le modèle afin par exemple de pouvoir reprendre les calculs en cas d'interruption. `PyTorch` a un mécanisme très pratique pour cela par l'intermédiaire de la fonction `state_dict()` qui permet de renvoyer sous forme de dictionnaire les paramètres importants d'un modèle (les paramètres d'apprentissage). Mais il est souvent nécessaire également de connaître l'état de l'optimiseur utilisé pour reprendre les calculs. Cette même fonction `state_dict()` permet également de sauver les valeurs des paramètres importants pour l'optimiseur utilisé. En pratique, les fonctions haut niveau `torch.save()` et `torch.load()` permettent très facilement de sauvegarder et charger les informations voulues et des informations complémentaires : elles vont utiliser le sérialiseur usuel de python `pickle` pour les structures habituelles et les fonctions `state_dict()` pour les objets de `PyTorch`.

```
from pathlib import Path
savepath = Path("model.pch")
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
class State:
    def __init__(self,model,optim):
        self.model = model
        self.optim = optim
        self.epoch, self.iteration = 0,0
if savepath.is_file():
```

```
    with savepath.open("rb") as fp:
        state = torch.load(fp)          #on recommence depuis le modele sauvegarde
else:
    autoencoder = ...
    autoencoder = autoencoder.to(device)
    optim = ...
    state = State(autoencoder,optim)
for epoch in range(state.epoch, ITERATIONS):
    for x,y in train_loader:
        state.optim.zero_grad()
        x = x.to(device)
        xhat = autoencoder(x)
        l = loss(xhat,x)
        l.backward()
        state.optim.step()
        state.iteration += 1
    with savepath.open("wb") as fp:
        state.epoch = epoch + 1
        torch.save(state,fp)
```

Question 3

Faire une campagne d'expériences pour l'autoencodeur sur MNIST en utilisant tous les outils présentés (GPU, checkpointing, `tensorboard` en particulier).

Pour `tensorboard`, il est possible non seulement de tracer des scalaires en fonction des itérations, mais également des états latents (méthode `add_embedding`), des courbes précision/rappel (méthode `add_pr_curve`) ou des images (méthode `add_image`).

Vous pouvez également visualiser la continuité des `embeddings` obtenus (les représentations dans l'espace latent) : prenez deux images $\mathbf{x}^1$ et $\mathbf{x}^2$ de classes différentes, calculez leur représentation $\mathbf{z}^1$ et $\mathbf{z}^2$, puis afficher les images correspondant au décodage de l'interpolation de leur représentation $\lambda * \mathbf{z}^1 + (1-\lambda)\mathbf{z}^2$ pour quelques valeurs de λ entre 0 et 1.

Question 4

Implémenter le Highway network. Comparer les performances par rapport à d'autres architectures classiques.