AS - TP 3
Implémentation de module, Gestion de données, Checkpointing et GPU

Nicolas Baskiotis - Benjamin Piwowarski

2019-2020

`Dataset` et `Dataloader`

Les classes `Dataset` et `Dataloader` permettent de faciliter la gestion des données sous l`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)` : renvoie l'exemple à l'index spécifié ;

— `__len__(self,)` : renvoie le nombre d'exemples du jeu de données.

Un des principaux avantages (mis à part le pré-traitement possible), il n'est pas nécessaire de charger tout le jeu de données en mémoire, le chargement se fait à la volée.

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

```
class MonDataset(Dataset):
    def __init__(self,...):    pass
    def __getitem__(self,index): pass #retourne un couple exemple,label
    def __len__(self): pass
data = DataLoader(MonDataset(...), shuffle=True, batch_size=BATCH_SIZE)
for x,y in data: #x est un batch de taille BATCH_SIZE, melange
```

Implémenter un dataset pour le jeu de données MNIST qui renvoie une image sous la forme d'un vecteur et le label associé. 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 :

```
from datamaestro import prepare_dataset
ds = prepare_dataset("com.lecun.mnist");
train_images, train_labels = ds.files["train/images"].data(), ds.files["train/labels"].data()
test_images, test_labels = ds.files["test/images"].data(), ds.files["test/labels"].data()
```

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 - $f(x) \approx x$ - 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.

`Implémenter une classe autoencodeur (héritée de Module) selon l'architecture suivante :` *linéaire* → *Relu* pour la partie encodage et *linéaire* → *sigmoïde* pour la partie décodage. Les poids du décodeur correspondent usuellement à la transposée des poids de l'encodeur (quel est l'avantage ?).

GPU, Checkpointing

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 charger sur le GPU (ou CPU) spécifié. 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 sur device
x = x.to(device) # charge les donnees sur device
x = x.to(device='cpu') # si on veut remettre sur cpu
```

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. La même fonction permet également de connaître 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`.

```
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)
```

Faire une campagne d'expériences pour l'autoencodeur sur MNIST en utilisant tous les outils présentés (GPU, checkpointing, tensorboard en particulier).
Implémenter le Highway network (pour la semaine suivante).