AMAL - TP 2
Prise en main complète de `pyTorch`

Nicolas Baskiotis - Stéphane Rivaud - Benjamin Piwowarski - Laure Soulier

2025-2026

Dans ce TME, vous allez exploiter toute la puissance de pyTorch pour la définition, l'apprentissage et l'évaluation de réseaux de neurones. Nous aborderons en particulier :

— l'utilisation de l'optimiseur de PyTorch
— la définition de réseaux à l'aide de modules
— la gestion des données d'apprentissage
— l'utilisation de GPU (**attention** : cela ne marchera que si vous avez une carte GPU)
— l'utilisation de checkpointing (reprendre un apprentissage après une interruption volontaire ou non)

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.

Pré-requis Vous devez avoir déjà implémenté une boucle de descente de gradient pour l'optimisation d'une régression linéaire telle que vue au TME précédent avec l'utilisation de l'autograd. Il n'est pas nécessaire d'avoir fait l'implémentation à la main des fonctions et de la back-propagation.

1 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))
```

```
## on optimise selon w et b, lr : pas de gradient
optim = torch.optim.SGD(params=[w,b],lr=EPS)
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` que l'on verra ci-dessous afin de reconnaître automatiquement les paramètres des autres entrées/constantes. Son utilisation est optionnelle dans le bout de code précédent, il fonctionne également avec un tenseur classique (et le flag `requires_grad` à vrai).

Question 1

Modifiez votre code pour inclure l'optimiseur plutôt que de faire les mise-à-jour à la main. Comparer votre version, celle avec l'optimiseur SGD et une dernière version avec l'optimiseur Adam.

2 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, une classe module est munie :

— 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 rétro-propagation peut être gérée par la différenciation automatique. Le constructeur permet d'initialiser les différents composants du module.

Ci-dessous un exemple de classe module ; il ne faut surtout pas oublier de le faire hériter de la classe abstraite `nn.Module` (nécessaire en particulier pour bien enregistrer les paramètres des sous-modules dans le module créé).

```
class MonModuleLineaire(torch.nn.Module):
    def __init__(self):
        super(MonModuleLineaire,self).__init__()
        # deux sous-modules linéaires
        self.un = torch.nn.Linear(10,100)
        self.deux = torch.nn.Linear(100,1)
    def forward(self,x):
        # définir le comportement forward du module
        return self.deux(self.un(x))
m = MonModuleLineaire()
x = torch.randn(100,10)
# Les paramètres du module (à passer à l'optimiseur)
print(list(m.parameters()))
# calcul du forward soit par  appel explicite de la fonction,
# soit par appel du module (plus court)
y = m.forward(x)
y = m(x)
```

Question 2

Utilisez les modules `torch.nn.Linear`, `torch.nn.Tanh` et `torch.nn.MSELoss` pour implémenter un réseau à deux couches : `lineaire` → `tanh` → `lineaire` → `MSE`. Implémentez la boucle de descente de gradient avec l'optimiseur.
Utilisez maintenant un conteneur - par exemple le module `torch.nn.Sequential` - pour implémenter le même réseau. Parcourir 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 ?
Utilisez enfin une classe pour implémenter le même réseau.

3 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
```

Un 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 peut se faire à la volée lorsque le i-ème exemple est requêté. 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 3

Implémentez 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`). Testez 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 sklearn.datasets import fetch_openml
#fixer DATA_PATH au répertoire où les données seront téléchargées.
x,y = fetch_openml('mnist_784',return_X_y=True,as_frame=False,data_home=DATA_PATH)
```

4 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* pour les tenseurs - pour les modules c'est bien le cas - 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 = ...
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('cpu') # si on veut remettre sur cpu
```

5 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`.

Un exemple d'utilisation ci-dessous :

```
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():  # si le fichier existe déjà
    with savepath.open("rb") as fp:
        state = torch.load(fp)        #on recommence depuis le modele sauvegarde
else:
    model = ...
    model = model.to(device)
    optim = ...
    state = State(model,optim)

for epoch in range(state.epoch, ITERATIONS):
    for x,y in train_loader:
        state.optim.zero_grad()
        x = x.to(device)
        xhat = state.model(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)
```

6 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 4

Implémentez 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 ?).

Question 5

Faire une campagne d'expériences pour l'autoencodeur sur MNIST et également pour des architectures à 2 ou 3 couches cachées 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 affichez 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 6

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