Rakib SHEIKH
Asano - DC Paris M2 Tech Lead - Machine Learning for Information Access
noobzik@pm.me

TD/TP 1 : MNIST et Convolutional Neural Network

Le but de ce TP est de prendre en main :
- la bibliothèque Torch pour réaliser des tâches de deep learning sur le traitement de l'image et la détection d'objet. Nous commencerons par MNIST.
- initier à la notion de déploiements des modèles en production avec FastAPI.
- Développer un front-end application pour intérargir avec le modèle déployé sur FastAPI via Streamlit.
- Packager l'ensemble du projet sous la forme d'un livrable docker.

Durée du TP estimé : 1.5 jours.

> **Note**
>
> Le rendu sera réalisé sous la forme d'un lien Git de l'hebergeur de votre choix.

1. Pré-requis :

Nous allons commencer par attaquer les tâches de l'apprentissage profond (deep learning). Comme vous avez vu dans le cours de Data Science & Machine Learning, les tâches du deep learning sont utiliser pour traiter de très grande volumétrie de données qui ne sont pas réalisable dans un modèle de machine learning.

Pour cela il est obligatoire d'avoir une carte graphique de type NVIDIA et d'installer CUDA

> **Warning**
>
> Si vous n'avez pas de carte graphique Nvidia, je ne peux rien faire pour vous ! Vous êtes livré à vous-même sur votre processeur. Je vous invite donc à préparer votre chaise Quechua / Décatlhon de camping pendant la phase de l'entraînement.

> **Note**
>
> Pour ceux qui sont sous **Linux** : Vous pouvez installer toute la stack du deep learning avec une ligne de commande :
>
> ```
> wget -nv -O- https://lambdalabs.com/install-lambda-stack.sh | sh -
> sudo reboot
> ```

2. Initialisation du dossier de travail

Pour être en parfait accord avec les règle du MLOPS niveau 2 disponible ici, il faudrait que nos codes soient rédigés dans un scipt python, et organisé de la même manière que dans les 5 TP des taxi jaunes de la ville de New-York. Nous allons donc créer des dossiers ayant l'achitecture suivante :

```
├── notebook          <- Fichiers notebooks pour la partie exploratoire
├── src               <- Stockage des fichiers sources pythons, prêtes à être industrialisés
│   ├── model         <- Stockage du code permettant de lancer l'apprentissage du code
│   └── app           <- Stockage du code FastAPI et du front end
│
├── model             <- Stockage des modèles entrainés, qu'ils soient sous forme de
                         checkpoints ou sous la forme finalisée
```

> **Note**
>
> À partir de maintenant, vous devez respecter cette structure de dossier pour le reste de la semaine

3. Notre premier notebook pour le deep learning.

3. 1. Initialisation

Nous allons commencer par charger les pré-requis :

```
# Cellule 1
```

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import datasets, transforms

import matplotlib.pyplot as plt

# Cellule 2
if torch.cuda.is_available()
  device = "cuda"
else if torch.backends.mps.is_available()
  device = "mps"
else
  device = "cpu"
```

Comme vous l'avez remarqué, nous testons si la carte graphique est disponible, dans le cas contraire, nous utiliserons le processeur.

3. 2. Chargement du dataset MNIST

Nous allons charger un dataset nommé **MNIST**. C'est une base de donnée qui comprend des images de chiffre écrit à la main, de 0 à 9. Notre but est donc de développer un modèle permettant la reconnaissance des chiffres écrits à la main.

1. Pour charger notre dataset, nous allons appeler `dataset` de la librairie `torchvision` de la manière suivante :

```
# Ne pas executer ce code pour le moment !
train_loader = torch.utils.data.DataLoader(datasets.MNIST("../data/raw", download=True, train=True),
batch_size=64, shuffle=True)
test_load = torch.utils.data.DataLoader(datasets.MNIST("../data/raw", download=True, train=False), batch_size=64,
shuffle=True)
```

2. Nous devons appliquer des transformations aléatoires afin de mieux détecter les différentes écritures sur lequel on peut être confrontés. Pour cela, nous allons définir une série de transformation. Le premier consiste à transformer les images sous la forme de `Tensor` afin de le préparer aux calcules du GPU. Le deuxième consiste à normaliser les valeurs des images.

```
tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.1307,), (0.3081,))])
```

3. Nous pouvons donc appliquer ces transformations dès le chargement du dataset en ajoutant `transform=tf` !

```
train_loader = torch.utils.data.DataLoader(datasets.MNIST("../data/raw", download=True, train=True, transform=tf),
batch_size=64, shuffle=True)

test_load = torch.utils.data.DataLoader(datasets.MNIST("../data/raw", download=True, train=False, transform=tf),
batch_size=64, shuffle=True)
```

4. Notre dataset étant chargé, trouvez un moyen d'afficher les 5 premiers images à l'aide de matplotlib.

```
# Début du code
batch = next(iter(train_loader))
x = batch[0][:10]
y = batch[1][:10]

# Ici vous rédigerez un code pour afficher les 5 premiers images
```

3. 3. Construction d'un modèle de convolution neural network

Nous allons construire le modèle suivante :

- Conv2D : in_channels=1, out_channels=n_kernel, kernel_size=5
- ReLu
- MaxPool : kernel_size:2
- Conv2D : in_channels=n_kernel, out_channels=n_kernel, kernel_size=5
- ReLu
- MaxPool : kernel_size:2
- Flatten
- Linear : in_features=n*kernel*4*4, out_features=50
- Linear: in_features=50, out_features=50

Pour cela,nous devons définir une classe ayant le nom de notre modèle, donc `ConvNet` qui va prendre d'entrée un `nn.Module` Et nous allons définir deux méthodes, à savoir la fonction d'initalisation `__init__` et la fonction `forward`:

```
class ConvNet(nn.Module):
```

```
def __init__(self, input_size, n_kernels, output_size):
    super().__init__()
    self.net = nn.Sequential(

    )

def forward(self, x):
    return
```

1. Complétez la méthode `__init__` en fonction des graphes qui vous est fourni.

Vous devez utiliser uniquement les méthodes suivantes : `nn.Conv2d`, `nn.ReLU`, `nn.MaxPool2d`, `nn.Flatten`, `nn.Linar`.

2. Complétez la méthode forward.

3. 4. Construction du train et du test.

Maintenant, nous devons créer une méthode `train` pour réaliser l'entraînement, et une méthode `test` qui va pouvoir calculer le nombre de prédictions correcte sur les images.

3. 4. 1. train

Pour rappel, l'entraînement d'un modèle consiste à écrire une boucle qui va calculer la descente du gradient en fonction du résultat par la (les) fonction d'activation(s) définit dans la classe du modèle en question.

1. Rédigez une fonction `train`, permettant de:
   - prendre en paramètre un modèle, une `perm` avec une valeur par défaut fixée à `torch.arange(0, 784).long()` et le nombre d'epoch (`n_epoch`) avec une valeur par défaut à 1.
   - lancer l'entraînement du modèle à l'aide de `model.train()`
   - de fixer un optimizer qui va ajuster automatiquement la valeur γ qui est notre learning rate : `torch.optim.AdamW(model.parameters())`
   - Rédiger une boucle `for` représentant le nombre d'itération (epoch)
     - Rédiger une `for` imbriqué représentant l'itération `i` qui définit la i-ème image du train_loader et un couple (data, target) représentant les valeurs de la matrice et la cible attendu.
       - Envoyer les données vers la carte graphique : `data, targets = data.to(device), target.to(device)`
       - Appliquer les permutations de pixels par la matrice circulaire de Toeplitz. Pour cela, nous aurons trois ligne tel que

         ```
         data = data.view(-1, 28*28)
         data = data[:, perm]
         data = data.view(-1, 1, 28, 28)
         ```

       - Appliquer le step en appelant sur l'`optimizer` la méthode `zero_grad()` suivit d'une méthode lançant une prédiction sur `data` dont le résultat sera stocké dans la variable `logits` .
       - Calculer le loss à l'aide de `F.cross_entropy()`, elle prends deux arguments.
       - Appliquer la mise à jour des poids sur le `loss` avec la méthode `backward()`
       - Appliquer la méthode `step()` sur l'`optimizer` pour finaliser l'itération d'un cycle d'apprentissage.
       - Afficher tout les 100-itération le numéro de l'itération, le numéro du step (correspondant au numéro de l'image du train-loader) et la valeur du loss du modèle.

3. 4. 2. test

La phase de test va reprendre essentiellement le contenu du `train` mais pas en sa totalité.

1. Rédigez une fonction `test` permettant de :
   - D'initialiser deux variables `test_loss` et `correct` à 0.
   - Appeler la méthode `model.eval()` afin de préparer le modèle à l'inférence.
   - Rédiger une `for` imbriqué représentant l'itération `i` qui définit la i-ème image du train_loader et un couple (data, target) représentant les valeurs de la matrice et la cible attendu.
     - Envoyer les données vers la carte graphique : `data, targets = data.to(device), target.to(device)`
     - Appliquer les permutations de pixels par la matrice circulaire de Toeplitz.
     - Lancer une prédiction à l'aide de `model(date)` dont le résultat sera stocké dans la variable `logits`.
     - Calculer les métriques :
       - La variable `test_loss` va réaliser un algorithme de type `map-reduce` avec les éléments de `logits` par rapport au `targets` associé à une réduction de somme. La fonction permettant cette tâche provient de `F.cross_entropy()` qui prend trois paramètres.
       - La variable `pred` va renvoyer la prédiction finale par la probabilité la plus haute parmis les 10 éléments de sorties possible. Nous utiliserons la fonction `torch.argmax()` qui prends en paramètre le `logits` que l'on a définit, et une `dim=1` pour préciser que le tableau de `logits` et en dimension 1.
       - La variable `correct` qui va compter le nombre de prédiction correcte, afin de calculer plus tard, le taux d'accuracy : $\frac{\text{test\_loss}}{\text{len(test\_loader.dataset)}}$.
   - Calculer le loss avec la formule suivante

- Calculer l'accuracy avec la formule suivante : $\frac{\text{Nombre de prédictions correcte}}{\text{Nombre de prédictions totale}}$.
- Affichez l'accuracy et le loss.

3. 5. Lancement de l'entrainement.

Il est temps de lancer l'entrainement et d'afficher les premiers résutlats. Rédigez une fonction `main` dont le code suivant fonctionne

```
n_kernels = 6
convnet = ConvNet(input_size, n_kernels, output_size)
convnet.to(device)
print(f"Parameters={sum(p.numel() for p in convnet.parameters())/1e3}K")

train(convnet)
test(convnet)
```

Vous devriez avoir un affichage ressemblant à celui-ci:

```
Parameters=6.422K
epoch=0, step=0: train loss=2.3233
epoch=0, step=100: train loss=0.4378
epoch=0, step=200: train loss=0.1719
epoch=0, step=300: train loss=0.2179
epoch=0, step=400: train loss=0.3929
epoch=0, step=500: train loss=0.2728
epoch=0, step=600: train loss=0.1188
epoch=0, step=700: train loss=0.0639
epoch=0, step=800: train loss=0.2077
epoch=0, step=900: train loss=0.1554
test loss=0.1042, accuracy=0.9667
```

3. 6. Nouveau modèle : le perceptron multi-couches

1. Définir une nouvelle classe représentant un perceptron multi-couche de 3 `nn.Linear` dont les deux activations intermédiaires est `nn.ReLU`. Vous devez donc définir `__init__`et `forward`.
2. Lancez l'entrainement avec le code suivant :

```
input_size = 28*28
output_size = 10

n_hidden = 8
mlp = MLP(input_size, n_hidden, output_size)
mlp.to(device)
print(f"Parameters={sum(p.numel() for p in mlp.parameters())/1e3}K")

train(mlp)
test(mlp)
```

Discutez des performances entre un Convolutional neural network et un Multilayer Perceptron

3. 7. Application d'un shuffle aléatoire

Si vos analyses s'avèrent correctes, nous allons démontrer dans le cas où les affirmations du convets sont fausses. Maintenant, appliquons une permutation aléatoire en fixant : `perm = torch.randperm(784)`

1. Refaire l'entrainement, cette fois-ci en donnant un paramètre `perm`, pour les deux modèles.
2. **Discutez des performances entre un Convolutional neural network et un Multilayer Perceptron dans ce cas là**

3. 8. Sauvegarde du modèle

Notre but ici est de réutiliser le modèle afin de le mettre en production sous un site internet permettant de détecter en temps réel les chiffres manuscrites à l'aide de `FastAPI`

Pour cela, vous allez besoin de mobiliser l'intégralité de vos connaissance de `docker` à partir de maintenant.

1. Appelez la méthode `torch.save()` sur le modèle convnets que vous venez d'entrainer. Elle prend deux paramètre à savoir le modèle en question grâce à la méthode `state_dict()` et un chemin où l'on va sauvegarder le modèle au format `pt`.

3. 9. Conversion d'un notebook vers des fichiers sources

Il est primordiale de comprendre qu'on ne met jamais en production un fichier notebook. En effet, il est sous la forme binarisée et rend difficile à assurer le suivi des modifications et de version à l'aide de Git. De ce faite, il faudra maintenant convertir vos travaux du fichier notebook en de véritables scripts python.

1. Rédigez un script python permettant de réaliser l'entraînement du modèle.

3. 10. Déploiement de l'API (back-end)

Maintenant que nous avons rédigé un script python qui permet de réaliser l'entraînement du modèle, nous avons en sortie un modèle sauvegardé et versionné. (Notre modèle s'appelle : `mnist-0.0.1.pt`).

Nous allons donc commencer à le mettre en production via FastAPI.

1. Rédigez un programme FastAPI permettant d'exposer un endpoint `api/v1/predict` qui va prendre en paramètre un `body` dont on va récupérer une image.
   - Convertir cette image en type `np.array` dont le contenue de chaque élément sera `np.float32` et appliquer une normalisation entre 0 et 1 en divisant par 255.
   - Rédiger une fonction `predict` qui va prendre en paramètre l'image que vous venez de transformer et le modèle déjà initalisé qui va réaliser la prédiction.
   - Lancer une prédiction.
   - Retournez le résultat de la prédiction sous la forme d'un dictionnaire.

3. 11. Développement front sur le graphe manuscrit en ligne

Nous allons utiliser `streamlit`, une librairie qui permet de réaliser un front-end assez rapidement pour les Data Scientist qui ne veulent pas se casser la tête à le faire à la main. Rappelez-vous que notre API prends en entrée une image.

Afin de vous aiguiller, vous devez avoir dans votre frond-end, deux boxes :
- Le premier box servira de zone de dessin à la main.
- Le deuxième box servira d'une visualisation de l'image qui sera envoyé à notre modèle pour réaliser l'inférence.

Elle dispose également d'un bouton pour envoyer l'image vers notre back-end via FastAPI. Et enfin une zone de texte dynamique permettant d'afficher le résultat de l'inférence par notre back-end.

3. 12. Etape finale : Dockeriser 2 images (front et back)

1. Packager sous la forme d'une image docker le front (streamlit) à l'aide d'un Dockerfile
2. Packager sous la forme d'une image le back (FastAPI) à l'aide d'un Dockerfile
3. Rédiger un docker-compose.yml qui va lancer le front et le back.

> **Note**
>
> Pour aller plus loin, c'est une bonne pratique de bien séparer le front et le back sous deux Git différents, vu qu'ils sont indépendant des uns des autres !

3. 13. Conclusion

Vous venez de finir le niveau 0 des principes du MLOPS selon Google.
- Le Niveau 1 correspond à l'automatisation du déploiement des modèles en production.
- Le Niveau 2 est l'automatisation intégrale de toute la pipeline du projet, c'est à dire de la récupération des données, jusqu'à la mise en production du modèle avec un recyclage automatique en cas de baisse de performance.
  - C'est ce qu'on va voir dans la deuxième partie du cours.