# TD - PyTorch & Deep Learning

## Introduction à Pytorch

Lors de votre première année, vous avez été exposés à l'apprentissage automatique au travers de notebooks utilisant scikit-learn.

Bien que ces outils soient utiles et aient leur place, en pratique les notebooks sont majoritairement utilisés pour l'exploration de données et la visualisation, tandis que scikit-learn cache une grande partie de la complexité de l'apprentissage automatique à des fins éducatives.
L'objectif de cette seconde année est d'une part d'approfondir vos connaissances en apprentissage et de vous introduire au framework Pytorch.

**Vous utiliserez durant les TPs de manière prioritaire des scripts python au lieu des notebooks** jupyter auxquels vous avez étés habitués. Ceci dit, si vous pensez qu'un notebook est plus adapté (par exemple pour une phase d'exploration des données), n'hésitez pas à en faire un en plus de votre script python.

> **Info**
>
> Les machines qui vous sont fournies possèdent un environnement `virtualenv` nommé '*envia*' qui possède les packages nécessaires au suivi des TP/TDs. Si vous choisissez d'utiliser une machine personelle, suivez la démarche en annexe pour installer l'environnement.

## Pytorch

Pytorch est l'un des deux grands framework de machine learning utilisés, avec Tensorflow. Leur fonctionnement est globalement le même, avec des syntaxes similaires. Pytorch gagnant de plus en plus de traction, c'est ce framework que nous étudierons.

## Tenseurs et opérations sur tenseurs

Les tenseurs sont une structure de données spécialisée très similaire aux tableaux et aux matrices. Dans PyTorch, nous utilisons des tenseurs pour encoder les entrées et les sorties d'un modèle, ainsi que ses paramètres.

Les tenseurs sont similaires aux ndarrays de NumPy, à la différence que les tenseurs peuvent s'exécuter sur des GPU ou d'autres accélérateurs matériels. Les tenseurs sont également optimisés pour la différenciation automatique (nous en verrons plus à ce sujet plus tard).

### Initialisation d'un Tenseur

Les tenseurs peuvent être créés directement à partir de données. Le type de données est automatiquement déduit.

```
import torch
import numpy as np

data = [[1, 2],[3, 4]]
x_data = torch.tensor(data)
```

#### À partir d'un tableau NumPy

Les tenseurs peuvent être créés à partir de tableaux NumPy.

```
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
```

#### À partir d'un autre tenseur

Le nouveau tenseur conserve les propriétés (forme, type de données) du tenseur argument, sauf si elles sont explicitement modifiées.

```
x_ones = torch.ones_like(x_data)
# conserve les propriétés de x_data
print(f"Tenseur de uns : \n {x_ones} \n")

x_rand       =       torch.rand_like(x_data,
dtype=torch.float)
# remplace le type de données de x_data
print(f"Tenseur aléatoire : \n {x_rand} \n")
```

Sortie :

```
Tenseur de uns :
 tensor([[1, 1],
        [1, 1]])

Tenseur aléatoire :
 tensor([[0.4223, 0.1719],
        [0.3184, 0.2631]])
```

#### Avec des valeurs aléatoires ou constantes

`shape` est un tuple de dimensions du tenseur. Dans les fonctions ci-dessous, il détermine la dimensionnalité du tenseur de sortie.

```
shape = (2,3,)
rand_tensor = torch.rand(shape)
ones_tensor = torch.ones(shape)
zeros_tensor = torch.zeros(shape)

print(f"Tenseur aléatoire : \n {rand_tensor} \n")
print(f"Tenseur de uns : \n {ones_tensor} \n")
print(f"Tenseur de zéros : \n {zeros_tensor}")
```

Sortie :

```
Tenseur aléatoire :
 tensor([[0.1602, 0.6000, 0.4126],
        [0.5558, 0.0912, 0.3004]])

Tenseur de uns :
 tensor([[1., 1., 1.],
[1., 1., 1.]])

Tenseur de zéros :
 tensor([[0., 0., 0.],
        [0., 0., 0.]])
```

## Attributs d'un Tenseur

Les attributs d'un tenseur décrivent sa forme, son type de données et le dispositif sur lequel il est stocké.

```
tensor = torch.rand(3,4)

print(f"Forme du tenseur : {tensor.shape}")
print(f"Type de données du tenseur :
{tensor.dtype}")
print(f"Dispositif sur lequel le tenseur est
stocké : {tensor.device}")
```

Sortie :

```
Forme du tenseur : torch.Size([3, 4])
Type de données du tenseur : torch.float32
Dispositif sur lequel le tenseur est stocké : cpu
```

**? Question**

1. Créez le tenseur suivant `tensor([[0., 0., 0.], [1., 1., 1.]])`
2. Affichez le type de ce tenseur (déterminé en fonction des élements qu'il contient). Transformez ce tenseur en int.
3. Créez ce tenseur, puis affichez sa propriété *'shape'*. A quoi correspond le premier chiffre de shape ?

## Opérations sur les Tenseurs

Plus de 100 opérations sur les tenseurs, y compris l'arithmétique, l'algèbre linéaire, la manipulation de matrices (transposition, indexation, découpage), l'échantillonnage et plus encore sont possibles. Vous trouverez la liste complète sur la documentation Pytorch.

Chacune de ces opérations peut être exécutée sur le GPU (généralement à des vitesses plus élevées que sur un CPU). Par défaut, les tenseurs sont créés sur le CPU.

Nous devons explicitement déplacer les tenseurs vers le GPU en utilisant la méthode .to (après avoir vérifié la disponibilité du GPU). Gardez à l'esprit que la copie de grands tenseurs entre dispositifs peut être coûteuse en termes de temps et de mémoire !

```
# Nous déplaçons notre tenseur vers le GPU s'il
est disponible
if torch.cuda.is_available():
  tensor = tensor.to('cuda')
```

**i Info**

CUDA (Compute Unified Device Architecture) est une plateforme de calcul parallèle développée par NVIDIA pour exploiter la puissance des processeurs graphiques (GPU) dans le cadre du calcul général.

PyTorch intègre nativement le support de CUDA via la bibliothèque `torch.cuda`. Cela permet aux utilisateurs de PyTorch d'allouer et de manipuler facilement des tenseurs sur le matériel NVIDIA.

**Exercice**

Essayez quelques-unes des opérations de la liste ci-dessous.

### Indexation et découpage standard de type numpy

```
tensor = torch.ones(4, 4)
print('Première ligne : ',tensor[0])
print('Première colonne : ', tensor[:, 0])
print('Dernière colonne :', tensor[..., -1])
tensor[:,1] = 0
print(tensor)
```

Sortie :

```
Première ligne :  tensor([1., 1., 1., 1.])
Première colonne :  tensor([1., 1., 1., 1.])
Dernière colonne : tensor([1., 1., 1., 1.])
tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])
```

### Joindre des tenseurs

Vous pouvez utiliser `torch.cat` pour concaténer une séquence de tenseurs le long d'une dimension donnée. Voir aussi `torch.stack`, une autre opération de jonction de tenseurs qui est subtilement différente de `torch.cat`.

```
t1 = torch.cat([tensor, tensor, tensor], dim=1)
print(t1)
```

Sortie :

```
tensor([[1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
       [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
       [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
       [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.]])
```

### Opérations arithmétiques

```
# Ceci calcule le produit matriciel entre deux
tenseurs. y1, y2, y3 auront la même valeur
y1 = tensor @ tensor.T
y2 = tensor.matmul(tensor.T)

# Ceci calcule le produit élément par élément. z1,
z2, z3 auront la même valeur
z1 = tensor * tensor
z2 = tensor.mul(tensor)
```

**Question**

A quoi correspond l'opération `tensor.T` ?

### Tenseurs à élément unique

Si vous avez un tenseur à un seul élément, par exemple en agrégeant toutes les valeurs d'un tenseur en une seule valeur, vous pouvez le convertir en une valeur numérique Python en utilisant `item()` :

```
agg = tensor.sum()
agg_item = agg.item()
print(agg_item, type(agg_item))
```

Sortie :

```
12.0 <class 'float'>
```

## Datasets & DataLoaders

Le code pour traiter les échantillons de données peut devenir désordonné et difficile à maintenir ; idéalement, nous voulons que notre code de dataset soit découplé de notre code d'entraînement du modèle pour une meilleure lisibilité et modularité.

PyTorch fournit deux objects principaux pour gérer les données : `torch.utils.data.DataLoader` et `torch.utils.data.Dataset` qui vous permettent d'utiliser des datasets pré-chargés ainsi que vos propres données.

Dataset stocke les échantillons et leurs étiquettes correspondantes, et DataLoader enveloppe un itérable autour du Dataset pour permettre un accès facile aux échantillons.

### Chargement d'un Dataset

Voici un exemple de chargement d'un dataset.

```
import torch
from torch.utils.data import Dataset, DataLoader

class RandomDataset(Dataset):
    def __init__(self, size=1000, dim=5):
        self.size = size
        self.dim = dim
        self.data = torch.randn(size, dim)
        self.labels = torch.randint(0, 2, (size,))

    def __len__(self):
        return self.size

    def __getitem__(self, idx):
        return self.data[idx], self.labels[idx]
```

**Info**

Les classes Dataset doivent implémenter trois fonctions:

- `__init__`, définie lors de l'instanciation de l'objet,
- `__len__` permettant d'obtenir la taille du jeu de données
- `__getitem__`, qui permet d'aller chercher un point de donnée en particulier.

Nous pouvons indexer les Datasets manuellement comme une liste : `training_data[index]`.

Si vous voulez simplement créer un dataset contenant des tenseurs de variables d'entrées et d'étiquettes, vous pouvez utiliser la classe `TensorDataset` directement.

```
dataset = TensorDataset(input_features, labels)
```

Notez que les variables d'entrées et les étiquettes doivent correspondre sur la première dimension: chaque ligne du jeu de données correspond à une étiquette, et pas chaque colonne.

### Création d'un DataLoader

Le Dataset récupère les caractéristiques et les étiquettes de notre dataset un échantillon à la fois.
Lors de l'entraînement d'un modèle, nous voulons généralement passer des échantillons en "mini-batchs", mélanger les données à chaque époque pour réduire le surajustement du modèle (paramètre `shuffle`).

```
dataset = RandomDataset(size=1000, dim=5)
dataloader = DataLoader(dataset, batch_size=8,
shuffle=True)
```

**Question**

- Rappelez ce qu'est un batch.
- Quelle est son utilité dans le contexte d'un dataloader?
- A quoi correspondrait donc le batch size ?

**Exercice**

- Testez le code en itérant sur un batch.
- Affichez le batch ainsi que les étiquettes correspondantes.

**Info**

Vous pouvez itérer sur un `DataLoader` comme vous le feriez sur une liste ou tout autre itérable avec une boucle `for`.

## Préparation de vos données pour l'entraînement avec DataLoaders

Evidemment, il vous faudra mettre de côté une partie des échantillons pour constituer le jeu de test. En Pytorch, nous utilisons simplement un second `DataLoader` qui itère sur les données de test

```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(training_data,
batch_size=64, shuffle=True)
test_dataloader    =    DataLoader(test_data,
batch_size=64, shuffle=True)
```

**Exercice**

Décrivez ce qu'il se passe au niveau des données lorsque l'on utilise un *k-fold*.
Comment vous y prendriez vous pour créer vos dataloaders ?

## Différentiation automatique avec `torch.autograd`

Lors de l'entraînement de réseaux de neurones, l'algorithme le plus fréquemment utilisé est la rétropropagation. Dans cet algorithme, les paramètres (les poids du modèle) sont ajustés en fonction du gradient de la fonction de perte par rapport aux paramètres donnés.

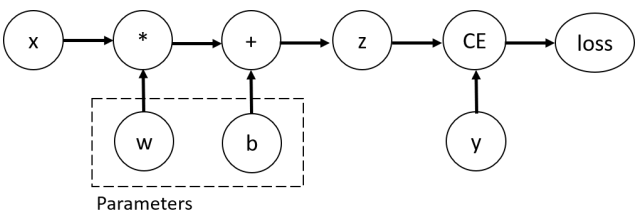
Pour calculer ces gradients, PyTorch dispose d'un moteur de différentiation intégré appelé `torch.autograd`. Il prend en charge le calcul automatique du gradient pour tout graphe computationnel.

Considérons le réseau de neurones le plus simple à une couche, avec une entrée x, des paramètres w et b, et une fonction de perte. Il peut être défini dans PyTorch de la manière suivante :

```
import torch
from        torch.nn.functional        import
binary_cross_entropy_with_logits

x = torch.ones(5)  # tenseur d'entrée
y = torch.zeros(3)  # sortie attendue
w = torch.randn(5, 3, requires_grad=True)
b = torch.randn(3, requires_grad=True)
z = torch.matmul(x, w) + b
loss = binary_cross_entropy_with_logits(z, y)
```

Ce code définit le graphe computationnel suivant :




Dans ce réseau, w et b sont des paramètres que nous devons optimiser. Ainsi, nous devons pouvoir calculer les gradients de la fonction de perte par rapport à ces variables.

**? Question**

- Rappelez ce qu'est une fonction de perte.
- Comment fonctionne la BCE (binary cross entropy) ?
- Pour quel type de tâche cette fonction de loss est-elle adaptée ?

**i Info**

Vous pouvez définir la valeur de `requires_grad` lors de la création d'un tenseur, ou plus tard en utilisant la méthode `x.requires_grad_(True)`.

Par défaut, un tenseur de paramètres est initialisé avec le suivi de gradient activé.

Les fonctions permettant de construire le graphe computationnel sont des objet de la classe `Function` et doivent définir :

- Une méthode `forward(context, *args)` : passe avant, calcule le résultat de la fonction appliquée aux arguments
- Une méthode `backward(context, *args)` : passe arrière, calcule les dérivées partielles par rapport aux entrées. Les arguments de cette méthode correspondent aux valeurs des dérivées suivantes dans le graphe de calcul.

Cet objet sait comment calculer la fonction dans la direction avant, et aussi comment calculer sa dérivée lors de l'étape de rétropropagation. Une référence à la fonction de rétropropagation est stockée dans la propriété `grad_fn` d'un tenseur.

**Exercice**

Affichez la fonction de gradient (`grad_fn`) pour z et pour la perte (loss).
Au vu du graph computationnel, cela vous semble-t-il avoir du sens ?

## Calcul des gradients

Pour optimiser les poids des paramètres du réseau de neurones, nous devons calculer les dérivées de notre fonction de perte par rapport aux paramètres, c'est-à-dire que nous avons besoin de $\frac{\partial \text{loss}}{\partial w}$ et $\partial \frac{\text{loss}}{\partial b}$ pour des valeurs fixes de $x$ et $y$. Pour calculer ces dérivées, nous appelons `loss.backward()`, puis nous récupérons les valeurs de `w.grad` et `b.grad` :

```
loss.backward()
print(w.grad)
print(b.grad)
```

Sortie:

```
tensor([[0.3313, 0.0626, 0.2530],
```

```
        [0.3313, 0.0626, 0.2530],
        [0.3313, 0.0626, 0.2530],
        [0.3313, 0.0626, 0.2530],
        [0.3313, 0.0626, 0.2530]])
tensor([0.3313, 0.0626, 0.2530])
```

> **? Question**
>
> Expliquez les dimensions des gradients en relation avec l'architecture (i.e. le graph computationnel) de votre modèle.

> **⚠ Warning**
>
> Vu les propriétés additives du gradient, l'appel à `backward()` met à jour par addition le gradient attaché à la variable.
>
> Il faut explicitement demander sa remise à zéro lors d'un nouveau calcul, sinon le résultat est additioné à l'ancien. C'est pour ça que vous verrez `optimizer.zero_grad()` à chaque changement de batch dans les boucles d'entrainement.

### Désactivation du suivi des gradients

Par défaut, tous les tenseurs avec requires_grad=True suivent leur historique computationnel et prennent en charge le calcul des gradients.

Cependant, il existe des cas où nous n'avons pas besoin de cela, par exemple, lorsque nous avons entraîné le modèle et que nous voulons simplement l'appliquer à des données d'entrée, c'est-à-dire que nous voulons uniquement effectuer des calculs avant dans le réseau.

Nous pouvons arrêter le suivi des calculs en entourant notre code de calcul avec le bloc `torch.no_grad()` :

```python
z = torch.matmul(x, w) + b
print(z.requires_grad)

with torch.no_grad():
    z = torch.matmul(x, w) + b
print(z.requires_grad)
```

Sortie:

```
True
False
```

## Construction d'un modèle

L'espace de noms `torch.nn` fournit tous les éléments de base dont vous avez besoin pour construire votre propre réseau neuronal. Chaque module dans PyTorch hérite de la classe `nn.Module`.
Un réseau neuronal est lui-même un module qui se compose d'autres modules (couches). Cette structure imbriquée permet de construire et de gérer facilement des architectures complexes.

Dans les sections suivantes, nous allons voir un exemple complet de construction d'un réseau de neurones. Commencons avec les imports:

```python
import os
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
```

### Obtenir l'appareil pour l'entraînement

Nous voulons idéalement pouvoir entraîner notre modèle sur un accélérateur matériel comme le GPU, si disponible. Vérifions si `torch.cuda` est disponible, sinon nous utiliserons le CPU.

```python
device = (
    "cuda"
    if torch.cuda.is_available()
    else "cpu"
)
print(f"Utilisation de l'appareil {device}")
```

Un serveur GPU est en cours de montage pour la formation, pour le moment nous utiliserons les CPUs de vos machines, mais auront l'occasion de tester vos codes sur GPU au cours de l'année.

### Définir la classe

Nous définissons notre réseau neuronal en héritant de `nn.Module`, et initialisons les couches du réseau neuronal dans `__init__`. Chaque sous-classe de `nn.Module` implémente les opérations sur les données d'entrée dans la méthode forward.

```python
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

Nous créons une instance de `NeuralNetwork`, la déplaçons vers le matériel (CPU ou GPU), et affichons sa structure.

```python
model = NeuralNetwork().to(device)
print(model)
```

```
NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=784, out_features=512,
bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512,
bias=True)
```

```
    (3): ReLU()
    (4): Linear(in_features=512, out_features=10,
bias=True)
  )
)
```

> **? Question**
>
> Au vu de l'architecture du modèle, quel type de tâche essayons nous d'effectuer ici ?
> *Indice*: Allez voir la documentation de `torch.flatten()`, et déterminez le nombre de sorties

Pour utiliser le modèle, nous lui passons les données d'entrée. Cela exécute la méthode forward du modèle, ainsi que quelques opérations en arrière-plan.
N'appelez pas directement `model.forward()`, mais faites plutôt comme ci-dessous où l'appel de forward est implicite:

```
X = torch.rand(1, 28, 28, device=device)
logits = model(X)
pred_probab = nn.Softmax(dim=1)(logits)
y_pred = pred_probab.argmax(1)
print(f"Classe prédite : {y_pred}")
```

```
Classe prédite : tensor([7], device='cuda:0')
```

Appeler le modèle sur l'entrée renvoie un tenseur à 2 dimensions avec dim=0 correspondant à chaque sortie de 10 valeurs brutes prédites pour chaque classe, et dim=1 correspondant aux valeurs individuelles de chaque sortie `logits`. Nous obtenons les probabilités de prédiction en le passant à travers une instance du module `nn.Softmax`.

> **? Question**
>
> - Qu'est-ce qu'un logit ?
> - Pourquoi les passer par la fonction `Softmax`?

## Couches du modèle

Décomposons les couches du modèle. Pour l'illustrer, nous allons prendre un mini-lot d'exemple de 3 points de données et voir ce qui lui arrive lorsque nous le faisons passer à travers le réseau.

```
input = torch.rand(3,28,28)
print(input.size())
```

torch.Size([3, 28, 28])

### nn.Flatten

Nous initialisons la couche nn.Flatten pour convertir chaque input de 28x28 en un tableau contigu de 784 valeurs de pixels (la dimension du mini-lot (à dim=0) est maintenue). Cela devrait vous rappeller un certain TP de l'année dernière..

```
flatten = nn.Flatten()
flat_input = flatten(input)
print(flat_input.size())
```

```
torch.Size([3, 784])
```

> **? Question**
>
> - Que fait la fonction `size()` ? Comment se compare-t-elle à la fonction `shape()` ?
> - A quoi correspondent les arguments de la fonction `torch.rand` ?

### nn.Linear

La couche linéaire est un module qui applique une transformation linéaire sur l'entrée en utilisant ses poids et biais stockés.

```
layer1 = nn.Linear(in_features=28*28,
out_features=20)
hidden1 = layer1(flat_input)
print(hidden1.size())
```

```
torch.Size([3, 20])
```

### nn.ReLU

Les activations non linéaires créent les mappages complexes entre les entrées et les sorties du modèle. Elles sont appliquées après les transformations linéaires pour introduire de la non-linéarité, aidant les réseaux neuronaux à apprendre une grande variété de phénomènes.

Dans ce modèle, nous utilisons `nn.ReLU` entre nos couches linéaires, mais il existe d'autres activations pour introduire de la non-linéarité dans votre modèle.

### nn.Sequential

nn.Sequential est un conteneur ordonné de modules. Les données sont passées à travers tous les modules dans le même ordre que défini. Vous pouvez utiliser des conteneurs séquentiels pour assembler rapidement un réseau comme `seq_modules`.

```
seq_modules = nn.Sequential(
    flatten,
    layer1,
    nn.ReLU(),
    nn.Linear(20, 10)
)
input = torch.rand(3,28,28)
logits = seq_modules(input)
```

> **Exercice**
>
> Réecrivez le code de la classe `NeuralNetwork` pour qu'il se passe du conteneur `nn.Sequential()`. Vous adapterez la fonction `forward()`en conséquence.

### nn.Softmax

La dernière couche linéaire du réseau neuronal renvoie des logits - valeurs brutes dans [-∞, ∞] - qui sont passées au module nn.Softmax. Les logits sont mis à l'échelle pour des valeurs [0, 1] représentant les probabilités prédites par le modèle pour chaque classe. Le paramètre dim indique la dimension le long de laquelle les valeurs doivent sommer à 1.

```
softmax = nn.Softmax(dim=1)
pred_probab = softmax(logits)
```

### Paramètres du modèle

De nombreuses couches à l'intérieur d'un réseau neuronal sont paramétrées, c'est-à-dire qu'elles ont des poids et des biais associés qui sont optimisés pendant l'entraînement.

L'héritage de `nn.Module` suit automatiquement tous les champs définis à l'intérieur de votre objet modèle, et rend tous les paramètres accessibles en utilisant les méthodes `parameters()` ou `named_parameters()` de votre modèle.

Dans cet exemple, nous itérons sur chaque paramètre, et imprimons sa taille et un aperçu de ses valeurs.

```
print(f"Structure du modèle : {model}\n\n")

for name, param in model.named_parameters():
      print(f"Couche : {name} | Taille :
{param.size()} | Valeurs : {param[:2]} \n")
```

**Exercice**

- Lancez ce code, puis déterminez le nombre total de paramètres de votre modèle.
- Ecrivez l'équation du nombre de paramètres entraînables en fonction du nombre d'entrées et de sorties d'une couche linéaire.

*Indice*: Commencez par vous rappeller quels paramètres d'un réseau de neurones sont entraînables.

## Optimisation des paramètres du modèle

Maintenant que nous avons un modèle, il est temps d'entraîner, de valider et de tester notre modèle en optimisant ses paramètres sur nos données.
Les données en question proviennent du dataset Fashion-MNIST, facilement chargeable depuis torchvision.

**Question**

Allez rechercher le jeu de données en question:
- Quel est le type de tâche ?
- Est-il plus ou moins dur que le jeu MNIST que vous avez déjà rencontré ? Pourquoi ?

L'entraînement d'un modèle est un processus itératif ; à chaque itération, le modèle fait une estimation de la sortie, calcule l'erreur de son estimation (perte), collecte les dérivées de l'erreur par rapport à ses paramètres (comme nous l'avons vu dans la section précédente), et optimise ces paramètres en utilisant la descente de gradient.

Pour une explication plus détaillée de ce processus, consultez cette vidéo sur la rétropropagation de 3Blue1Brown.

### Code prérequis

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

training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)

test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
)

train_dataloader = DataLoader(training_data,
batch_size=64)
test_dataloader = DataLoader(test_data,
batch_size=64)

class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits

model = NeuralNetwork()
```

### Hyperparamètres

Les hyperparamètres sont des paramètres ajustables qui vous permettent de contrôler le processus d'optimisation du modèle. Différentes valeurs d'hyperparamètres peuvent avoir un impact sur l'entraînement du modèle et les taux de convergence.

Nous définissons les hyperparamètres suivants pour l'entraînement :

- *Nombre d'époques (epoch)* - le nombre de fois que nous parcourons l'ensemble de données
- *Taille du lot (batch size)* - le nombre d'échantillons de données propagés à travers le réseau avant que les paramètres ne soient mis à jour
- *Taux d'apprentissage (learning rate)* - à quel point mettre à jour les paramètres du modèle à chaque lot. Des valeurs plus petites donnent une vitesse d'apprentissage lente, tandis que des valeurs plus grandes peuvent entraîner un comportement imprévisible pendant l'entraînement.

```
learning_rate = 1e-3
batch_size = 64
epochs = 5
```

> **? Question**
>
> Selon vous, choix de la taille des couches cachées est-il un hyperparamètre ?

## Boucle d'optimisation

Une fois que nous avons défini nos hyperparamètres, nous pouvons alors entraîner et optimiser notre modèle avec une boucle d'optimisation.
Chaque itération de la boucle d'optimisation correspond à une époque (epoch): l'ensemble des données à été vu.

Chaque époque se compose de deux parties principales :

1. *La boucle d'entraînement* - itérer sur l'ensemble de données d'entraînement et essayer de converger vers des paramètres optimaux.
2. *La boucle de validation/test* - itérer sur l'ensemble de données de test pour vérifier si les performances du modèle s'améliorent.

Familiarisons-nous brièvement avec certains des concepts utilisés dans la boucle d'entraînement. Passez à la section suivante pour voir l'implémentation complète de la boucle d'optimisation.

### Fonction de perte

Lorsqu'on lui présente des données d'entraînement, notre réseau non entraîné est susceptible de ne pas donner la bonne réponse.
La fonction de perte mesure le degré de dissimilarité entre le prédiction donnée par le modèle et la réponse attendue, et c'est cette fonction de perte que nous voulons minimiser pendant l'entraînement.

Les fonctions de perte courantes comprennent `nn.MSELoss` (Erreur Quadratique Moyenne) pour les tâches de régression, et `nn.CrossEntropyLoss` pour la classification.

Nous passons les logits de sortie de notre modèle à `nn.CrossEntropyLoss`, qui normalisera les logits et calculera l'erreur de prédiction.

```
# Initialisation de la fonction de perte
loss_fn = nn.CrossEntropyLoss()
```

> **Exercice**
>
> Dans le cadre d'une classification binaire utilisant la Cross Entropy (CE), calculez la perte totale pour le batch suivant:
> - Etiquette: 1, Prédiction du modèle: 0.7
> - Etiquette: 0, Prédiction du modèle: 0.3
> - Etiquette: 1, Prédiction du modèle: 0.8
>
> *Formule CE*: $-y * \log(p) - (1 - y) * \log(1 - p)$ .

### Optimiseur

L'optimisation est le processus d'ajustement des paramètres du modèle pour réduire l'erreur du modèle à chaque étape d'entraînement.
Les algorithmes d'optimisation définissent comment ce processus est effectué. Toute la logique d'optimisation est encapsulée dans l'objet `optimizer`. Ici, nous utilisons l'optimiseur SGD ; de plus, il existe de nombreux optimiseurs différents disponibles dans PyTorch tels que ADAM et RMSProp, qui fonctionnent mieux pour différents types de modèles et de données.

Nous initialisons l'optimiseur en enregistrant les paramètres du modèle qui doivent être entraînés, et en passant l'hyperparamètre du taux d'apprentissage.

```
optimizer = torch.optim.SGD(model.parameters(),
lr=learning_rate)
```

À l'intérieur de la boucle d'entraînement, l'optimisation se déroule en trois étapes :

1. Appeler `optimizer.zero_grad()` pour réinitialiser les gradients des paramètres du modèle. Les gradients s'accumulent par défaut ; pour éviter le double comptage, nous les mettons explicitement à zéro à chaque itération.
2. Rétropropager la perte de prédiction avec un appel à `loss.backward()`. PyTorch calcule les gradients en fonction de la perte par rapport à chaque paramètre.
3. Une fois que nous avons nos gradients, nous appelons `optimizer.step()` pour ajuster les paramètres par les gradients collectés dans la passe arrière.

> **i Info**
>
> Entraîner un modèle performant dépend de nombreux facteurs: l'architecture du modèle, le choix de l'optimiseur, le choix des hyperparamètres et bien d'autres. Pour rentrer plus dans les détails, consultez cette excellente ressource.

## Implémentation complète

Nous définissons `train_loop` qui boucle sur notre code d'optimisation, et `test_loop` qui évalue les performances du modèle par rapport à nos données de test.

```
def train_loop(dataloader, model, loss_fn,
optimizer):
    size = len(dataloader.dataset)
    for batch, (X, y) in enumerate(dataloader):
        # Calculer la prédiction et la perte
        pred = model(X)
        loss = loss_fn(pred, y)

        # Rétropropagation
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

        if batch % 100 = 0:
```

```
            loss, current = loss.item(), batch *
len(X)
        print(f"perte: {loss:>7f}  [{current:>5d}/
{size:>5d}]")


def test_loop(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct = 0, 0

    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
                  correct  +=  (pred.argmax(1)  =
y).type(torch.float).sum().item()

    test_loss /= num_batches
    correct /= size
              print(f"Erreur    de    test:    \n
Précision: {(100*correct):>0.1f}%, Perte moyenne:
{test_loss:>8f} \n")

# Initialisation de la fonction de perte et de
l'optimiseur
loss_fn = nn.CrossEntropyLoss()
optimizer  =  torch.optim.SGD(model.parameters(),
lr=learning_rate)

epochs = 10
for t in range(epochs):
                                    print(f"Époque
{t+1}\n-------------------------------")
     train_loop(train_dataloader, model, loss_fn,
optimizer)
    test_loop(test_dataloader, model, loss_fn)
print("Terminé !")
```

Nous initialisons la fonction de perte et l'optimiseur, et les passons à train_loop et test_loop. N'hésitez pas à augmenter le nombre d'époques pour suivre l'amélioration des performances du modèle

**? Question**

- A quoi correspondent X et y dans ce code ?

# Annexe

## Gestion de l'environnement

Si vous souhaitez utiliser votre propre machine, vous pouvez installer un environnement local en utilisant conda.

**Exercice**

Installez **miniconda**, une version de conda minimaliste, puis créez un environnement sous python 3.12. Enfin, **activez-le**.

Conda et Pip sont deux gestionnaires de packages. Pip est spécifique à python tandis que Conda permet d'installer des dépendances dans d'autres langages.

Il est possible d'installer pip dans un environnement conda et de profiter du répertoire de packages disponibles par pip.

Vous pouvez installer un package grâce à

- `conda install <package>`

ou d'un

- `pip install <package>`

si le package n'est pas disponible sous conda.

**Exercice**

Installez les packages suivants:

- pytorch
- pandas
- numpy
- matplotlib
- jupyter
- seaborn (optional)

En pratique, la plupart des installations sont faites avec pip pour des questions de rapidité.

**i Info**

Sur github vous trouverez parfois des fichiers correspondant à l'environnement utilisé par les développeurs du code. Vous trouverez parfois un fichier *environment.yml* qui peut être directement installé en tant qu'environnement conda avec

```
conda env create -n my_env --file ENV.yaml
```

Plus communément, vous trouverez l'équivalent *requirements.txt* sous pip:

```
pip install -r requirements.txt
```