TP4. Réseaux de neurones convolutionnels.

Frédéric Richard

Cours apprentissage statistique et réseaux de neurones
Master mathématiques appliquées, statistique,
Parcours Data Science.

2025, AMU

L'objectif de ce TP est d'apprendre à mettre en oeuvre des réseaux de neurones convolutionnels pour effectuer des taches de classifi cation. Dans un premier temps, on s'intéresse à la classification des chiffres manuscrits de la base de données MNIST. Puis, on travaille sur un problème de détection de fissures sur des images de matériaux.

Déclaration des librairies et méthodes utiles au TP:

[ ]:
```
# Méthodes de pytorch.
from torch.utils.data import DataLoader, Dataset
from torchvision.datasets import MNIST
from torch.nn import Module, Linear, CrossEntropyLoss, BCEWithLogitsLoss
from torch.optim import SGD, Adam
from torch import Tensor, no_grad
from torch import flatten, max
from torch import manual_seed
from torch import device as torch_device
from torch.backends.cudnn import deterministic
from torch.cuda import is_available as cuda_is_available

# Méthodes de torchvision.
from torchvision import transforms

# Méthodes système.
import glob

# Méthodes numpy.
from numpy import array

# Méthodes de scikit-learn.
from sklearn.model_selection import train_test_split

# Méthodes de matplotlib
from matplotlib import pyplot as plt

# Méthodes I/O images
```

```
from imageio.v3 import imread
```

0.1 1. Classification des chiffres manuscrits.

0.1.1 1.1. Préparation des données.

On commence par charger la base de données MNIST et à la préparer pour le traitement. Les images de la base sont normalisées. On définit une générateur de batchs (mini-lots) pour l'apprentissage et le test, qui intégre des transformations sur les images.

[ ]:
```
def load_mnist(batch_size=64):
    # Définition des transformations : conversion en tenseur et normalisation
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,)) # Normalisation pour MNIST
    ])

    # Chargement des ensembles d'entraînement et de test
    train_dataset = MNIST(root='./data', train=True, download=True,␣
 ↪transform=transform)
    test_dataset = MNIST(root='./data', train=False, download=True,␣
 ↪transform=transform)

    # Création des DataLoaders
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
    test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

    return train_loader, test_loader


train_loader, test_loader = load_mnist()

# Affichage d'un batch pour vérification
images, labels = next(iter(train_loader))
print(f"Taille du batch : {images.shape}")  # Devrait afficher (batch_size, 1,␣
 ↪28, 28)
print(f"Labels : {labels[:10]}")
```

0.1.2 1.2. Régression multinomiale.

Tout d'abord, on effectue la classification des images mnist à partir d'une régression multinomiale.

Définition du réseau : On définit le réseau de neurones qui permet de faire de la régression multinomiale sur la base MNIST.

[ ]:
```
class MultinomialRegression(Module):
    def __init__(self):
        super(MultinomialRegression, self).__init__()
        self.linear = Linear(28 * 28, 10)
```

```
    def forward(self, x):
        x = flatten(x, 1)  # Vectorisation des images
        return self.linear(x)
```

Apprentissage du réseau : On minimise un critère d'entropie croisée avec un algorithme de gradient stochastique par mini-lots.

On prépare l'apprentissage.

[ ]:
```
# Définition d'une valeur de départ aléatoire (seed).
random_seed = 12
# Initialisation du seed pour le générateur de nombres aléatoires de PyTorch.
manual_seed(random_seed)
# Définition du drapeau "deterministic" pour le backend CUDA de PyTorch à True.
deterministic = True
# Détermine le périphérique à utiliser pour les calculs (GPU ou CPU).
device = torch_device("cuda" if cuda_is_available() else "cpu")
# Affiche le périphérique sélectionné (soit "cuda" ou "cpu").
device
```

On définit le modèle, la fonction de perte et l'optimiseur.

[ ]:
```
# Instanciation du modèle.
model = MultinomialRegression().to(device)
# fonction de perte: entropie croisée.
criterion = CrossEntropyLoss()
# optimiseur: algorithme de gradient stochastique par mini-lots.
optimizer = SGD(model.parameters(), lr=0.01)
nb_epoques = 3  # Nombre d'éqoques.

for epoch in range(nb_epoques):
    running_loss = 0.0
    for i, (X, Y) in enumerate(train_loader):
        # Réinitialisation des gradients
        optimizer.zero_grad()
        # Passage avant
        outputs = model(X)
        loss = criterion(outputs, Y)
        # Rétropropagation et optimisation
        loss.backward()
        optimizer.step()
        # Calcul de la perte moyenne.
        running_loss += loss.item()
    # Affichage des statistiques de perte
    print(f"[{epoch + 1}] loss = {running_loss / len(train_loader):.3f}")
```

Evaluation. On évalue le modèle sur la base de test en calculant la précision du modèle (pourcentage d'images correctement classées).

On définit une fonction pour calculer une erreur de classification (*accuracy*).

[ ]:
```python
def evaluate(model, data_loader):
    model.eval()
    correct = 0
    total = 0
    with no_grad():
        for images, labels in data_loader:
            outputs = model(images)
            _, predicted = max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    return correct / total * 100
```

[ ]:
```python
print(f"Précision (apprentissage) : {evaluate(model, train_loader):.1f} %")
print(f"Précision (test) : {evaluate(model, test_loader):.1f} %")
```

0.1.3 1.3. Réseaux de neurones convolutionnels.

1. En vous inspirant du modèle de régression logistique, créer le réseau de neurones convolutionnel (CNN 1) suivant:

Couche	**Spécifications**	**Dimension de la sortie**	**Nombre de paramètres**
Input	-	(28, 28)	-
Convolution 2D	32 filtres de taille 3 x 3, activation ReLU	.....	.....
Max Pooling	Taille de fenêtre 2 x 2 et sous-échantillonage de pas 2	.....	.....
Vectorisation	-	.....	.....
Dense	10 cellules	.....	.....

2. Faire l'apprentissage du CNN 1.
3. Comparer le CNN 1 au modèle de régression logistique en termes de complexité (nombre de paramètres) et de précision.
4. Mêmes questions avec le réseau de neurones convolutionel (CNN 2):

Couche	**Spécifications**	**Dimension de la sortie**	**Nombre de paramètres**
Input	-	(28, 28)	-
Convolution 2D n°1	32 filtres de taille 3 x 3, activation ReLU	.....	.....
Max Pooling	Taille de fenêtre 2 x 2 et sous-échantillonage de pas 2	.....	.....

Couche	Spécifications	Dimension de la sortie	Nombre de paramètres
Convolution 2D n°2	64 filtres de taille 3 x 3, activation ReLU	.....	.....
Max Pooling	Taille de fenêtre 2 x 2 et sous-échantillonage de pas 2	.....	.....
Vectorisation	-	.....	.....
Dense n°1	128 cellules	.....	.....
Dense n°2	10 cellules	.....	.....

0.2 1. Détection des fissures sur des images de matériaux.

0.2.1 1.1. Présentation des données.

Les données se trouvent sur le site surface crack detection de Kaggle. Elles sont constituées d'images de matériaux dont certains comportent des fissures et d'autres non. L'objectif est de construire un réseau de neurones qui permet de détecter ces fissures.

0.2.2 1.2. Préparation des données.

On commence par mettre à disposition les données. Plusieurs solutions sont possibles. Lorsqu'on travaille sur Kaggle, on peut mettre à disposition les données dans le répertoire de travail */kaggle/working/* en utilisant les outils dédiés de kaggle.

[ ]:
```
image_dir = '/kaggle/working/'
!rm -fr image_dir
# Télécharge le dataset "Surface Crack Detection".
!kaggle datasets download -d arunrk7/surface-crack-detection
# Décompresse le fichier archive.
!unzip -z surface-crack-detection.zip
```

On fait ensuite la liste des accès physiques vers les images en distiguant celles qui ont des fissures (dans le répertoire Positive) de celles qui n'en ont pas (dans le répertoire Negative). On crée également un vecteur d'étiquettes pour associer la classe à chaque image.

[ ]:
```
pos_images = glob.glob(image_dir + 'Positive/*.jpg')
neg_images = glob.glob(image_dir + 'Negative/*.jpg')
images = pos_images + neg_images
labels = array([1] * len(pos_images) + [0] * len(neg_images))
```

On peut visualiser quelques images pour se rendre compte des données.

[ ]:
```
def Affiche_Image(path, cnt, titre):
    img = imread(path)
    plt.subplot(2, 5, cnt)
    plt.imshow(img, cmap='gray')
    plt.axis('off')
    plt.title(titre)
```

```
# Afficher les images
plt.figure(figsize=(10, 10))
cnt = 1
for i  in range(5):
    path = neg_images[i]
    titre = "negative:" + path[-9:-4]
    Affiche_Image(path, cnt, titre)
    cnt += 1
for i  in range(5):
    path = pos_images[i]
    titre = "positive:" + path[-11:-4]
    Affiche_Image(path, cnt, titre)
    cnt += 1

plt.tight_layout()
plt.show
print(pos_images[0])
```

Ensuite, on répartie les images en trois sous-ensembles des sous-ensembles qui seront utilisé pour l'entraînement, la validation et le test. Pour cela, on peut utiliser une méthode de scikit-learn qui permet de constituer des sous-ensembles équilibrés entre les deux classes (positif et négatif).

[ ]:
```
# On sépare les données de test du reste des données en retenant au hasard 20%␣
 ↪des données.
images_reste, images_test, y_reste, labels_test = train_test_split(images,␣
 ↪labels,
                                                                    test_size=0.2,␣
 ↪random_state=12, shuffle=True)

images_train, images_val, labels_train, labels_val =␣
 ↪train_test_split(images_reste, y_reste,
                                                                         test_size=0.25,␣
 ↪random_state=12, shuffle=True)

print(f"Taille de la base d'entraînement: {len(images_train)}")
print(f"Taille de la base de validation: {len(images_val)}")
print(f"Taille de la base de test: {len(images_test)}")
```

Puis on personalise une classe Dataset pour gérer les données en pytorch et de manipuler les mini-lots.

[ ]:
```
class Dataset_fissures(Dataset):
    def __init__(self, img_path, img_labels):
        # Liste des chemins d'accès aux images.
        self.img_path = img_path
        # Liste des classes.
        self.img_labels = Tensor(img_labels)
```

```
        # Transformées à appliquer aux images.
        self.transforms = transforms.Compose([
            transforms.Grayscale(),  # Conversion des couleurs en niveaux de␣
↪gris.
            transforms.Normalize(mean=[0.5], std=[0.5]) # Normalisation de␣
↪l'image.
            transforms.ToTensor(),  # Conversion en Tensor.
        ])


    def __getitem__(self, index):
        # Charge une image.
        cur_img = imread(self.img_path[index])
        # Applique une suite de transformations à l'image.
        cur_img = self.transforms(cur_img)
        # Renvoie l'image et sa classe.
        return (cur_img, self.img_labels[index])

    def __len__(self):
        return len(self.img_path)
```

On spécifie ensuite les trois sous-ensembles (apprentissage, validation et test).

[ ]:
```
train_dataset = Dataset_fissures(images_train, labels_train)
val_dataset = Dataset_fissures(images_val, labels_val)
test_dataset = Dataset_fissures(images_test, labels_test)
```

Pour finir, on crée les générateurs de patchs sur les ensembles de données qui seront utilisés pour l'apprentissage, la validation et le test.

[ ]:
```
batch_size =  100
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

Pour l'apprentissage, on utilisera comme fonction de perte, l'entropie croisée binaire, adaptée pour le cas où il y a deux classes. Celle-ci s'appelle BCEWithLogitsLoss. Ce critère nécessite que la dernière couche du réseau ne comporte qu'une seule cellule. Comme optimiseur, on utilisera l'algorithme *Adam* en spécifiant un learning_rate à 1e-3.

Exercice 2.

1. Construire un réseau de neurones convolutionnel pour classer les images, en indiquant sa complexité.
2. Faire l'apprentissage de ce réseau en l'évaluant sur des données de validation à chaque époque.
3. Evaluer la précision de ce réseau sur des données de test.
4. Conclure.