Master Mathématiques appliquées, statistique - parcours Data Science (AMU)
Cours apprentissage statistique et réseaux de neurones (Frédéric Richard).
Contrôle de connaissances n°2, 21/03/2025 (durée 1h).

Partie 1. On définit deux réseaux de neurones en vue de les appliquer pour la classification des images de chiffres manuscrits de la base de données MNIST.

[ ]:
```
import torch
import torch.nn as nn
```

[ ]:
```
# Définition du réseau de neurones n°1.
class RN_1(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(28 * 28, 512)
        self.relu1 = nn.ReLU()
        self.fc2 = nn.Linear(512, 256)
        self.relu2 = nn.ReLU()
        self.fc3 = nn.Linear(256, 128)
        self.relu3 = nn.ReLU()
        self.fc4 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.relu1(x)
        x = self.fc2(x)
        x = self.relu2(x)
        x = self.fc3(x)
        x = self.relu3(x)
        x = self.fc4(x)
        return x

# Définition du réseau de neurones n°2.
class RN_2(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(64 * 7 * 7, 128)
```

```
        self.relu3 = nn.ReLU()
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu1(x)
        x = self.pool1(x)

        x = self.conv2(x)
        x = self.relu2(x)
        x = self.pool2(x)

        x = self.flatten(x)
        x = self.fc1(x)
        x = self.relu3(x)
        x = self.fc2(x)
        return x
```

Séries de questions I.

1. Remplir les tableaux en annexe 1 décrivant les différentes couches des deux réseaux de neurones.
2. Qu'est-ce qui différencie fondamentalement les réseaux de neurones n°1 et n°2 en termes de structure ? Préciser votre réponse en donnant la définition mathématiques des couches qui diffèrent.
3. Qu'est ce que la couche ReLU que l'on retrouve dans les trois réseaux ?
4. Dans le réseau de neurones n°2, que font les couches pool1 et pool2 ? Quel est leur intérêt ?
5. Lequel des réseaux de neurones n°1 et n°2 est le plus adapté pour la classification des images de chiffres manuscrits et pourquoi ?
6. En cas de sur-apprentissage, comment pourrait-on modifier le réseau de neurones n°2 pour le rendre moins complexe ?
7. Qu'est ce que la batch normalisation et le dropout ? Quel est leur intérêt ?

Partie 2. On continue le programme.

```
[ ]: import torch.optim as optim
     import torchvision
     import torchvision.transforms as transforms
     from torch.utils.data import DataLoader

     # Etape 1
     transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.
       1307,), (0.3081,))])
     train_dataset = torchvision.datasets.MNIST(root='./data', train=True,
       transform=transform, download=True)
     test_dataset = torchvision.datasets.MNIST(root='./data', train=False,
       transform=transform, download=True)
     train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
```


max pool
cross entropy

```
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

# Etape 2
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = RN_1().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

num_epochs = 5
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    print(f"Époque {epoch+1}/{num_epochs}, Perte: {running_loss/
  len(train_loader):.4f}")

# Etape 3
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Résultat: {100 * correct / total:.2f}%')
```

Série de questions II

1. Que font les trois étapes du programme ?
2. Dans ce programme pytorch, comment les données sont-elles manipulées ?
3. Dans l'étape 2, quel est le critère qui est minimisée (préciser sa définition mathématique) ?
4. Dans l'étape 2, quelle est la signification de la variable *running_loss* affichée à la fin d'une itération sur *epoch* ?
5. Dans l'étape 2, pourquoi a-t-on deux boucles *for* ?
6. Dans l'étape 3, qu'affiche-t-on dans "Résultat" ?
7. En utilisant ce programme, comment peut-on comparer les performances des réseaux 1 et 2 ?

Annexe 1 (à remettre)

Nom : LEGENDRE
Prénom : Antoine

Nom	Type de couche	Spécifications	Dimension de la sortie	Nombre de paramètres
Réseau de neurones n°1 (RN_1)				
flatten	applatissement			/
fc1	régression linéaire	Entrée: Image 28×28 Sortie: 512 canaux	512	28×28×512
relu1	activation			/
fc2	régression linéaire	Entrée: 512 canaux Sortie: 256	256	512×256
relu2	activation			/
fc3	régression linéaire	Entrée: 256 canaux Sortie: 128	128	256×128
relu3	activation			/
fc4	régression linéaire	Entrée: 128 canaux Sortie: 10 classes	10	128×10
Réseau de neurones n°2 (RN_2)				
conv1	convolution	1 canal d'entrée, 32 filtres de sortie de taille 3×3	32	1×32×3×3+32=320
relu1	activation			/
pool1	max pooling			/
conv2	convolution	Entrée: 32 ; Sortie: 64 filtres 3×3	64	32×64×3×3+64
relu2	activation			/
pool2	max pooling			/
flatten	applatissement			/
fc1	couche entièrement connectée	Entrée: 64 canaux d'images réduites en taille 7×7 Sortie: 128	128	64×7×7×128
relu3	activation			/
fc2	couche entièrement connectée	Entrée: 128 canaux ; Sortie: 10 classes	10	128×10

4

1,5