Réseau de Neurones (MLP) avec sortie Softmax

Théorie

Un réseau de neurones multi-couches (MLP - Multi-Layer Perceptron) est un modèle d'apprentissage supervisé basé sur des couches de neurones artificiels. Il est particulièrement efficace pour la classification non linéaire.

Dans notre cas, nous utilisons une couche de sortie Softmax, qui permet de normaliser les sorties du réseau en probabilités pour une classification multiclasses.

Hyperparamètre utilisé

Nous allons optimiser:

• Nombre d'époques (epochs) : sélectionné en fonction de la précision sur l'ensemble de validation.

Nous utilisons également : - **Optimiseur Adam** avec un taux d'apprentissage adaptatif. - **Taille du batch (batch_size)** fixé à 32.

Métriques d'évaluation

Nous afficherons:

- Matrice de confusion : montrant les erreurs de classification sur l'échantillon de test.
- Taux de bien classés sur l'échantillon de validation avec le meilleur nombre d'époques.
- Taux de bien classés sur l'échantillon de test avec ce même hyperparamètre.
- Taux de bien classés par classe sur l'échantillon de test pour observer la précision sur chaque classe.

Recherche du meilleur nombre d'époques et évaluation

```
import os
import torch
```

```
import torch.nn as nn
import torch.optim as optim
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import warnings
from sklearn.metrics import confusion_matrix, accuracy_score
from sklearn.preprocessing import StandardScaler
from torch.utils.data import TensorDataset, DataLoader
# Suppression des avertissements inutiles
warnings.filterwarnings("ignore", category=UserWarning)
# Utilisation du CPU uniquement (désactivation GPU)
device = torch.device("cpu")
torch.backends.cudnn.enabled = False
os.environ["CUDA_VISIBLE_DEVICES"] = "-1"
# Chargement des ensembles de données
train_data = pd.read_csv('../data/covertype_train.csv')
val_data = pd.read_csv('../data/covertype_val.csv')
test_data = pd.read_csv('../data/covertype_test.csv')
# Préparation des données
X_train, y_train = train_data.drop('Cover_Type', axis=1).values,
X_val, y_val = val_data.drop('Cover_Type', axis=1).values,

    val data['Cover Type'].values - 1

X_test, y_test = test_data.drop('Cover_Type', axis=1).values,

    test_data['Cover_Type'].values - 1

# Normalisation des données
scaler = StandardScaler()
X_train, X_val, X_test = scaler.fit_transform(X_train),

    scaler.transform(X_val), scaler.transform(X_test)

# Conversion en tenseurs PyTorch
X_train_torch = torch.tensor(X_train, dtype=torch.float32,
→ device=device)
y_train_torch = torch.tensor(y_train, dtype=torch.long,

→ device=device)

X_val_torch = torch.tensor(X_val, dtype=torch.float32,
→ device=device)
y_val_torch = torch.tensor(y_val, dtype=torch.long,

→ device=device)
```

```
X_test_torch = torch.tensor(X_test, dtype=torch.float32,

→ device=device)

y_test_torch = torch.tensor(y_test, dtype=torch.long,
→ device=device)
# Création des DataLoaders
batch size = 32
train_loader = DataLoader(TensorDataset(X_train_torch,
y_train_torch), batch_size=batch_size, shuffle=True)
val_loader = DataLoader(TensorDataset(X_val_torch, y_val_torch),
⇔ batch_size=batch_size, shuffle=False)
test_loader = DataLoader(TensorDataset(X_test_torch,

y_test_torch), batch_size=batch_size, shuffle=False)

# Définition du modèle PyTorch (MLP)
class MLP(nn.Module):
    def __init__(self, input_dim, num_classes):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_dim, 128)
        self.relu1 = nn.ReLU()
        self.fc2 = nn.Linear(128, 64)
        self.relu2 = nn.ReLU()
        self.fc3 = nn.Linear(64, num_classes)
    def forward(self, x):
       x = self.relu1(self.fc1(x))
       x = self.relu2(self.fc2(x))
       return self.fc3(x) # Pas de softmax ici (inclus dans

→ CrossEntropyLoss)

# Initialisation du modèle
num_features, num_classes = X_train.shape[1], len(set(y_train))
model = MLP(num_features, num_classes).to(device)
# Optimiseur et fonction de perte
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters())
# Entraînement du modèle avec sélection du meilleur nombre

→ d'époques

num epochs = 100
train_acc_list, val_acc_list = [], []
best_val_acc, best_epoch = 0, 0
for epoch in range(num_epochs):
    # Mode entraînement
```

```
model.train()
    correct_train, total_train = 0, 0
    for X_batch, y_batch in train_loader:
        optimizer.zero_grad()
        outputs = model(X_batch)
        loss = criterion(outputs, y_batch)
        loss.backward()
        optimizer.step()
        # Calcul de l'accuracy sur l'entraînement
        _, predicted = torch.max(outputs, 1)
        correct_train += (predicted == y_batch).sum().item()
        total_train += y_batch.size(0)
   train_acc_list.append(correct_train / total_train)
    # Mode validation
   model.eval()
    correct_val, total_val = 0, 0
    with torch.no_grad():
        for X_batch, y_batch in val_loader:
            outputs = model(X_batch)
            _, predicted = torch.max(outputs, 1)
            correct val += (predicted == y batch).sum().item()
            total_val += y_batch.size(0)
    val_accuracy = correct_val / total_val
    val_acc_list.append(val_accuracy)
    # Sauvegarde de la meilleure époque
    if val_accuracy > best_val_acc:
       best_val_acc, best_epoch = val_accuracy, epoch + 1
# Affichage des courbes d'entraînement
plt.figure(figsize=(8, 6))
plt.plot(range(1, num_epochs + 1), train_acc_list, label='Train
→ Accuracy')
plt.plot(range(1, num_epochs + 1), val_acc_list,

→ label='Validation Accuracy')
plt.axvline(best_epoch, color='r', linestyle='--', label=f'Best

    Epoch: {best_epoch}')

plt.xlabel("Époques")
plt.ylabel("Taux de bonnes prédictions")
```

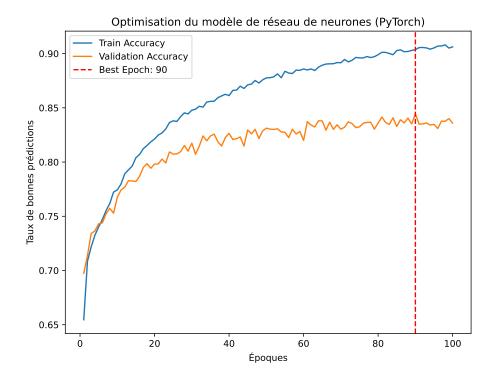
```
plt.title("Optimisation du modèle de réseau de neurones
plt.legend()
plt.show()
# Ré-entraîner le modèle avec le meilleur nombre d'époques
model.train()
for epoch in range(best_epoch):
    for X_batch, y_batch in train_loader:
        optimizer.zero_grad()
        outputs = model(X_batch)
       loss = criterion(outputs, y_batch)
        loss.backward()
        optimizer.step()
# Évaluation sur l'ensemble de test
model.eval()
correct_test, total_test = 0, 0
y_test_pred_classes = []
with torch.no_grad():
    for X_batch, y_batch in test_loader:
        outputs = model(X_batch)
        _, predicted = torch.max(outputs, 1)
       y_test_pred_classes.extend(predicted.cpu().numpy())
        correct test += (predicted == y batch).sum().item()
        total_test += y_batch.size(0)
test_accuracy = correct_test / total_test
# Affichage de la matrice de confusion et des métriques finales
conf_matrix = confusion_matrix(y_test, y_test_pred_classes)
print(f"\n Meilleure époque : {best_epoch} avec une précision de

    validation de {best_val_acc:.2%}")

print("\n Matrice de confusion :")
print(conf_matrix)
print("\n Taux de bien classés par classe :")
class_accuracies = conf_matrix.diagonal() /

    conf_matrix.sum(axis=1)

for i, acc in enumerate(class_accuracies, start=1):
    print(f"Classe {i} : {acc:.2%}")
print(f"\nTaux de bien classés total : {test_accuracy:.2%}")
```



Meilleure époque : 90 avec une précision de validation de 84.45%

Matrice de confusion					:		
[[1740		307	2	0	9	1	60]
[340	2366	26	0	63	34	4]
[1	12	1268	22	13	114	0]
[0	0	16	88	0	6	0]
[7	68	7	0	295	3	0]
[2	21	104	4	4	559	0]
Γ	45	6	0	0	0	0	770]]

Taux de bien classés par classe :

Classe 1 : 82.11% Classe 2 : 83.52% Classe 3 : 88.67% Classe 4 : 80.00% Classe 5 : 77.63% Classe 6 : 80.55% Classe 7 : 93.79%

Taux de bien classés total : 84.49%