Neural-Network - Suivi et evaluation des modèles

☰ TOC

1. Introduction

Dans le domaine de l'apprentissage automatique, la création et l'évaluation de modèles efficaces sont essentielles pour résoudre des problèmes complexes et prendre des décisions éclairées. Ce TP vise à vous familiariser avec des outils et des bibliothèques modernes qui facilitent ces processus : PyTorch, Weights & Biases (wandb), et Google Colab.

- **PyTorch** est une bibliothèque open-source largement utilisée pour le calcul tensoriel et l'apprentissage automatique, offrant une grande flexibilité et une accélération matérielle grâce aux GPU. Elle est particulièrement appréciée pour la recherche et le développement de modèles de réseaux de neurones. Il existent des alternatives (tensorflow, cafee, jax, ...)
- **Weights & Biases** est une plateforme qui permet aux chercheurs et aux praticiens de suivre et de visualiser les expériences d'apprentissage automatique. Elle offre des fonctionnalités pour enregistrer les hyperparamètres, les métriques de performance, et même les visualisations des modèles, facilitant ainsi la collaboration et la reproductibilité des expériences.
- **Google Colab**, quant à lui, est un environnement de notebook Jupyter hébergé dans le cloud, qui permet d'écrire et d'exécuter du code Python dans un navigateur web. Il est particulièrement utile pour les projets d'apprentissage automatique car il offre un accès gratuit aux GPU (carte graphique), ce qui accélère considérablement l'entraînement des modèles (que vous n'avez pas en salle de TP).

2. Implementation du modele

Pour implémenter un modèle de réseau de neurones en utilisant PyTorch, nous allons créer une classe qui hérite de `torch.nn.Module`, qui est la classe de base pour tous les modèles de réseaux de neurones dans PyTorch. Nous allons également intégrer `wandb` pour le suivi des expériences. Voici un guide étape par étape pour implémenter le modèle :

Étape 1 : Installation des bibliothèques nécessaires

Avant de commencer, assurez-vous d'avoir installé `pytorch` et `wandb` :

```
!pip install torch torchvision wandb
```

Étape 2 : Importation des bibliothèques

Dans votre notebook Google Colab, commencez par importer les bibliothèques nécessaires :

```
import torch
import torch.nn as nn
import torch.optim as optim
import wandb
```

Étape 3 : Initialisation de wandb

Creez et connectez-vous à votre compte Weights & Biases et initialisez une nouvelle session de suivi :

```
wandb.login()
```

Initialiser une nouvelle session de suivi

```
wandb.init(project="mon-projet-pytorch", entity="mon-compte-wandb")
```

Étape 4 : Définition du modèle

Créez une classe pour votre réseau de neurones. Dans cet exemple, nous allons créer un réseau de neurones simple avec une couche cachée :

```
import torch.nn as nn

# version 1
class MonModele(nn.Module):
    def __init__(self, input_size, hidden_size, output_size=1):
        super(MonModele, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        return out

# version 2
class MonModele(nn.Module):
    def __init__(self, input_size,  hidden_sizes=[5, 8, 9], output_size=1,):
        super(MonModele, self).__init__()

        layers = []
        in_size = input_size
        for h in hidden_sizes:
            layers.append(nn.Linear(in_size, h))
            layers.append(nn.ReLU())
            in_size = h
        layers.append(nn.Linear(in_size, output_size))

        self.model = nn.Sequential(*layers)

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

Étape 5 : Configuration des hyperparamètres

Définissez les hyperparamètres de votre modèle, ce sont des parametres externe aux modeles, qui ne sont pas appris par ce dernier. Puis initialisez le modèle, la fonction de perte et l'optimiseur :

```
# Hyperparamètres
input_size = 10
hidden_size = 50
output_size = 1
learning_rate = 0.001
num_epochs = 100

# Initialisation du modèle
model = MonModele(input_size, hidden_size, output_size)

# Fonction de perte et optimiseur
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```

Étape 6 : Entraînement du modèle

Entraînez votre modèle en utilisant une boucle d'entraînement. Nous allons également enregistrer les métriques avec wandb :

```
for epoch in range(num_epochs):
    # Génération de données d'exemple
    inputs = torch.randn(64, input_size)
    targets = torch.randn(64, output_size)

    # Forward pass
    outputs = model(inputs)
    loss = criterion(outputs, targets)

    # Backward pass et optimisation
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    # Enregistrement des métriques avec Weights & Biases
    wandb.log({"loss": loss.item()})

    if (epoch+1) % 10 == 0:
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')
```

Étape 7 : Fin de la session de suivi

Une fois l'entraînement terminé, terminez la session de suivi avec Weights & Biases :

```
wandb.finish()
```

En suivant ces étapes, vous avez implémenté un modèle de réseau de neurones en utilisant PyTorch et intégré un suivi des expériences. Vous pouvez maintenant entraîner et évaluer votre modèle sur différents jeux de données, analyser l'impact des hyperparamètres sur les performances, et utiliser des techniques de régularisation pour éviter le sur-apprentissage.

3. Entraînement et évaluation de modèles

Pour entraîner et évaluer un modèle sur le jeu de données Flowers102 en utilisant PyTorch et torchvision, nous allons suivre les étapes suivantes. Nous allons charger les données, définir un modèle, puis l'entraîner et l'évaluer sur les ensembles d'entraînement, de validation et de test.

Étape 1 : Installation des bibliothèques nécessaires

Assurez-vous d'avoir installé les bibliothèques nécessaires :

```
pip install torch torchvision wandb
Étape 2 : Importation des bibliothèques
```

Dans votre notebook Google Colab, commencez par importer les bibliothèques nécessaires :

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader, random_split
import wandb
```

Étape 3 : Initialisation de Weights & Biases

Connectez-vous à votre compte Weights & Biases et initialisez une nouvelle session de suivi :

```
wandb.login()
wandb.init(project="flowers102-classification", entity="mon-compte-wandb")
```

Étape 4 : Chargement et préparation des données

Nous allons utiliser le jeu de données Flowers102 disponible via torchvision. Nous allons également diviser les données en ensembles d'entraînement, de validation et de test.

```
# Transformations pour les images
transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
        transform.Flatten()
])

# Egalement possible mais attention : split aleatoire donc modeles non comparables
# Division des données en ensembles d'entraînement, de validation et de test
# full_dataset = torchvision.datasets.Flowers102(root='./data', split='train', download=True, transform=transform)
# train_size = int(0.7 * len(full_dataset))
# val_size = int(0.15 * len(full_dataset))
# test_size = len(full_dataset) - train_size - val_size
```

```
# train_dataset, val_dataset, test_dataset = random_split(full_dataset, [train_size, val_size, test_size])

train_dataset = torchvision.datasets.Flowers102(root='./data', split='train', download=True, transform=transform)
val_dataset = torchvision.datasets.Flowers102(root='./data', split='val', download=True, transform=transform)
test_dataset = torchvision.datasets.Flowers102(root='./data', split='test', download=True, transform=transform)

# Création des DataLoader
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

Étape 5 : Définition du modèle

Nous allons utiliser un modèle pré-entraîné ResNet18 et adapter la dernière couche pour notre tâche de classification.

```
input_size = 256*256
output_size = 102
model = MonModele(input_size, hidden_size, output_size)

# tester aussi avec
# model = torchvision.models.resnet18(pretrained=True)
# num_ftrs = model.fc.in_features
# model.fc = nn.Linear(num_ftrs, output_size)
# Quelle est la taille d'entree pour resnet18 ?
```

Étape 6 : Configuration des hyperparamètres

Définissez les hyperparamètres de votre modèle et initialisez la fonction de perte et l'optimiseur :

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

Étape 7 : Entraînement du modèle

Entraînez votre modèle en utilisant une boucle d'entraînement et enregistrez les métriques avec Weights & Biases :

```
num_epochs = 10
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    for inputs, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    # Validation
```

```
    model.eval()
    val_loss = 0.0
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in val_loader:
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            val_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    # Enregistrement des métriques avec Weights & Biases
    wandb.log({
        "epoch": epoch + 1,
        "train_loss": running_loss / len(train_loader),
        "val_loss": val_loss / len(val_loader),
        "val_accuracy": 100 * correct / total
    })

    print(f'Epoch {epoch + 1}, Train Loss: {running_loss / len(train_loader):.4f}, Val Loss: {val_loss / len(val_loader):
```

Étape 8 : Évaluation du modèle sur l'ensemble de test

Évaluez votre modèle sur l'ensemble de test :

```
model.eval()
test_loss = 0.0
correct = 0
total = 0

# desactive l'enregistrement et le calcule des gradients (plus rapide)
with torch.no_grad():
    for inputs, labels in test_loader:
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        test_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Test Loss: {test_loss / len(test_loader):.4f}, Test Accuracy: {100 * correct / total:.2f}%')
```

Étape 9 : Fin de la session de suivi

Une fois l'entraînement et l'évaluation terminés, terminez la session de suivi avec Weights & Biases :

```
wandb.finish()
```

En suivant ces étapes, vous avez entraîné et évalué un modèle de classification sur le jeu de données Flowers102 en utilisant PyTorch et torchvision. Vous avez également utilisé Weights & Biases pour suivre les métriques d'entraînement et de validation.

4. Analyse des résultats

Une fois que vous avez entraîné et évalué votre modèle, l'analyse des résultats est une étape cruciale pour comprendre les performances de votre modèle et identifier les pistes d'amélioration. Voici comment procéder à cette analyse en utilisant les données enregistrées avec Weights & Biases.

Visualisation des courbes d'apprentissage

Wandb offre une interface intuitive pour visualiser les métriques enregistrées pendant l'entraînement. Vous pouvez accéder à ces visualisations directement depuis votre tableau de bord.

- Courbe de perte d'entraînement : Cette courbe montre comment la perte a évolué au fil des époques pendant l'entraînement. Une courbe de perte qui diminue régulièrement indique que le modèle apprend bien.
- Courbe de perte de validation : Cette courbe montre la perte sur le jeu de données de validation. Si la perte de validation commence à augmenter alors que la perte d'entraînement continue de diminuer, cela peut indiquer un sur-apprentissage.

Comparaison des hyperparamètres

Weights & Biases permet de comparer facilement les performances de différents modèles avec des configurations d'hyperparamètres variées. Vous pouvez lancer plusieurs expériences avec différents taux d'apprentissage, tailles de batch, architectures de réseau, etc., et comparer les résultats.

- Taux d'apprentissage : Un taux d'apprentissage trop élevé peut entraîner une divergence du modèle, tandis qu'un taux trop faible peut ralentir l'apprentissage.
- Taille du batch : Des tailles de batch plus petites peuvent introduire plus de bruit dans le processus d'optimisation, ce qui peut parfois aider à éviter les minima locaux.
- Architecture du réseau : Le nombre de couches et de neurones peut avoir un impact significatif sur les performances du modèle. Des réseaux plus profonds peuvent capturer des caractéristiques plus complexes, mais ils sont aussi plus sujets au sur-apprentissage.
- Techniques de régularisation

Pour éviter le sur-apprentissage, plusieurs techniques de régularisation peuvent être utilisées :

- Dropout : Cette technique consiste à désactiver aléatoirement un certain nombre de neurones pendant l'entraînement, ce qui empêche le modèle de trop dépendre de certains neurones spécifiques.
- Régularisation L1 /L2 : Ces techniques ajoutent une pénalité à la fonction de perte pour décourager les poids de grande amplitude, ce qui peut aider à prévenir le sur-apprentissage.
- Early Stopping : Cette technique consiste à arrêter l'entraînement dès que la perte de validation commence à augmenter, ce qui permet de capturer le modèle à son pic de performance.

Exemple de code pour ajouter du Dropout

```
class MonModele(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, dropout_prob=0.5):
        super(MonModele, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(dropout_prob)
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.dropout(out)
        out = self.fc2(out)
```

```
        return out
```

5. Conclusion de l'analyse

En analysant les courbes d'apprentissage, en comparant les hyperparamètres et en appliquant des techniques de régularisation, vous pouvez obtenir une compréhension approfondie des performances de votre modèle. Utilisez ces informations pour ajuster votre modèle et améliorer ses performances. N'oubliez pas de documenter vos observations et vos ajustements dans votre rapport pour une référence future.

6. Visualisation et analyse des performances