examen_apprentissage_2024

January 13, 2024

1 Classification des EEG :

Dans ce projet, nous allons mettre en place deux modèles et tester différentes fonctions d'activation dont *ReLU*, *LeakyReLU* et *ELU* pour classer des signaux cérébraux en deux classes. Ce projet est vise à reproduire en partie les travaux de l'article ***EEGNet: a compact convolutional neural network for EEG-based brain–computer interfaces*** disponible ici.

Les objectifs sont les suivants :

- Déclarer les deux modèles.
- Montrer la plus grande précision des deux modèles en testant les trois fonctions d'activation.
- Visualiser les résultats.

[ ]:
```
import numpy as np
import pandas as pd
import os
import matplotlib.pyplot as plt
import torch
from torch.utils.data import TensorDataset, DataLoader
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from functools import reduce
from IPython.display import clear_output
```

1.0.1 Les données à télécharger

Nous allons utiliser les deux jeux de données ***train_dataset*** et ***test_dataset*** après avoir téléchargé et lu les deux fichiers train et test comme suit

[ ]:
```
train_dataset = torch.load('./train.pt')
test_dataset = torch.load('./test.pt')
print(len(train_dataset))
print(len(test_dataset))
```

Pour afficher un EEG, nous avons besoin de la fonction suivante

[ ]:
```
def plot_eeg(data):
    if len(data.shape) == 3:
        data = data[0]

    if len(data.shape) != 2:
        raise AttributeError("shape no ok")
        return

    plt.figure(figsize=(10,4))
    for i in range(data.shape[0]):
        plt.subplot(2,1, i+1)
        plt.ylabel("Channel "+str(i+1), fontsize=15)
        plt.plot(np.array(data[i, :]))
    plt.show()
```

[ ]:
```
plot_eeg(train_dataset[0][0])
```

1.0.2 Trois fonctions d'activation à tester

Nous allons tester trois fonctions d'activations qu'on peut représenter comme suit :

[ ]:
```
x = torch.arange(-10, 2, 0.5, dtype=torch.float, requires_grad=True)

y = nn.ReLU()(x)
plt.plot(x.data.numpy(), y.data.numpy(), label="ReLU")

y = nn.LeakyReLU()(x)
plt.plot(x.data.numpy(), y.data.numpy(), label="Leaky ReLU")

y = nn.ELU()(x)
plt.plot(x.data.numpy(), y.data.numpy(), label="ELU")

plt.legend()
plt.title('output of activation functions')
plt.ylabel('output')
plt.xlabel('input')
plt.show()
```

Il est difficile de déterminer a priori la meilleure fonction d'activation à utiliser dans un modèle de réseau de neurones profond. Nous allons donc comparer les performances de test de nos modèles en faisant varier le type de fonction d'activation. L'article précédent propose plusieurs modèles à tester, nous allons tester les deux modèles principaux dans ce projet.

1.0.3 Modèle 1

Ce modèle est composé de

- Une couche d'**entrée** simple.

- Une première couche **conv2dNormale** qui apprend à extraire les caractéristiques des signaux.
- Une seconde couche **conv2dProfonde** qui apprend à combiner les signaux de plusieurs canaux en un seul pour chaque entrée. La couche conv2dProfonde est différente des couches de convolution normales car elle ne relie pas complètement l'entrée et la sortie.
- Une troisième **conv2dSeparable** apprend à extraire les caractéristiques de la sortie de la couche conv2dProfonde.
- Une couche de **classification**.

[ ]:
```
class Mod1(nn.Module):
    def __init__(self, activation=None, dropout=0.25):
        super(Mod1, self).__init__()

        if not activation:
            activation = nn.ELU ## si aucune fonction d'activation n'est␣
 ↪déclaré, alors on prend ELU par défaut.

        self.conv2dNormale = ## à compléter par une séquence

 ##    - Une couche Conv2d avec
 ##      un canal d'entrée,
 ##      16 canaux de sorties,
 ##      un noyau de taille (1,51)
 ##      un stride de taille (1,1)
 ##      et un padding de taille (0,25)
 ##      Pas de biais
 ##
 ##    - Une couche BatchNorm2d qui prend en entrées les canaux de sortie de␣
 ↪la couche précédente.
 ##    tous les autres paramètres prennent les valeurs par défaut.


        self.conv2dProfonde = ## à compléter par une séquence de

 ##    - Une couche Conv2d avec un nombre de canaux en entrée correspondant
 ##      au nombre de canaux de sortie de la couche précédente
 ##      32 canaux de sorties
 ##      un noyau de taille (2,1)
 ##      un stride de taille (1,1)
 ##      16 canaux bloqués entre les canaux d'entrée et sortie
 ##      pas de biais
 ##
 ##    - Une couche BatchNorm2d qui prend en entrées les canaux de sortie de␣
 ↪la couche précédente.
 ##    tous les autres paramètres prennent les valeurs par défaut.
 ##
```

```
 ##    - Une activation "activation()" (dépend de l'argument à␣
↪l'instanciation).
 ##
 ##    - Une couche d'average pooling 2D avec
 ##      un noyau de taille (1, 4)
 ##      un stride de taille (1, 4)
 ##      un padding de 0
 ##
 ##    - Une couche de dropout avec une proportion "dropout" (dépend de␣
↪l'argument à l'instanciation).


      self.conv2dSeparable = ## à compléter par une séquence
 ##    - Une couche Conv2d avec un nombre de canaux en entrée correspondant
 ##      au nombre de canaux de sortie de la couche précédente,
 ##      32 canaux de sorties,
 ##      un noyau de taille (1,15),
 ##      un stride de taille (1,1)
 ##      un padding de taille (0,7)
 ##      pas de biais.
 ##
 ##    - Une couche BatchNorm2d qui prend en entrées les canaux de sortie de␣
↪la couche précédente.
 ##    tous les autres paramètres prennent les valeurs par défaut.
 ##
 ##    - Une activation "activation()" (dépend de l'argument à␣
↪l'instanciation).
 ##
 ##    - Une couche d'average pooling 2D avec
 ##      un noyau de taille (1, 8),
 ##      un stride de taille (1, 8)
 ##      un padding de 0.
 ##
 ##    - Une couche de dropout avec une proportion "dropout" (dépend de␣
↪l'argument à l'instanciation).


      self.classification = ## à compléter par
 ##    - Une couche linéaire avec 736 entrées
 ##      2 sorties.

  def forward(self, x):
      x = self.conv2dNormale(x)
      x = self.conv2dProfonde(x)
      x = self.conv2dSeparable(x)
      # applatissement
      x = x.view(-1, self.classification[0].in_features)
```

```
        x = self.classification(x)
        return x
```

1.0.4 Modèle 2

Ce modèle comporte plus de couches de convolution que le modèle précédent.

[ ]:
```
class Mod2(nn.Module):
    def __init__(self, activation=None, convProfond=[25,50,100,200], dropout=0.
↪5):
        super(Mod2, self).__init__()

        if not activation:
            activation = nn.ELU

        self.convProfond = convProfond
        self.couche0 =  ## à compléter par une séquence
 ##    - Une couche Conv2d avec un canal d'entrée
 ##      convProfond[0] canaux de sorties
 ##      un noyau de taille (1,5)
 ##      un stride de taille (1,1)
 ##      un padding (0,0)
 ##      un biais
 ##
 ##    - Une couche Conv2d avec
 ##      convProfond[0] canaux en entrée
 ##      convProfond[0] canaux de sorties
 ##      un noyau de taille (2,1)
 ##      un stride de taille (1,1)
 ##      un padding (0,0)
 ##      un biais
 ##
 ##    - Une couche BatchNorm2d qui prend en entrées les canaux de sortie de␣
↪la couche précédente.
 ##      tous les autres paramètres prennent les valeurs par défaut.
 ##
 ##    - Une activation "activation()" (dépend de l'argument à␣
↪l'instanciation).
 ##
 ##    - Une couche de max pooling 2D avec
 ##      un noyau de taille (1, 2),
 ##
 ##    - Une couche de dropout avec une proportion "dropout" (dépend de␣
↪l'argument à l'instanciation).

        for idx in range(1, len(convProfond)):
            setattr(self, 'couche'+str(idx), ## à compléter par une séquence
```

```
                    ##    - Une couche Conv2d
                    ##      convProfond[idx-1] canaux en entrée
                    ##      convProfond[idx] canaux de sorties
                    ##      un noyau de taille (1,5)
                    ##      un stride de taille (1,1)
                    ##      un padding (0,0)
                    ##      un biais
                    ##
                    ##    - Une couche BatchNorm2d qui prend en entrées les␣
↪canaux de sortie de la couche précédente.
                    ##      tous les autres paramètres prennent les valeurs par␣
↪défaut.
                    ##
                    ##    - Une activation "activation()" (dépend de l'argument␣
↪à l'instanciation).
                    ##
                    ##    - Une couche de max pooling 2D avec
                    ##      un noyau de taille (1, 2),
                    ##
                    ##    - Une couche de dropout avec une proportion "dropout"␣
↪(dépend de l'argument à l'instanciation).
                  )


        flatten_size =  convProfond[-1] * reduce(lambda x,_: round((x-4)/2),␣
↪convProfond, 750)
        self.classification = nn.Sequential(
            nn.Linear(flatten_size, 2, bias=True),
        )

    def forward(self, x):
        for i in range(len(self.convProfond)):
            x = getattr(self, 'couche'+str(i))(x)
        # flatten
        x = x.view(-1, self.classification[0].in_features)
        x = self.classification(x)
        return x
```

1.1 Partie 1.

Compléter soigneusement les deux modèles **Mod1** et **Mod2** en fonction des indications mises en commentaires dans le code.

1.2 Partie 2.

Pour l'entrainement de nos modèles, nous allons fixer le nombre d'epoch à 300 et la taille des minilots de données à 64. Calculer les performances de test des scénarios suivants :

- **Mod1** : avec un learning rate de 10^{-2} et tous les paramètres du réseau par défaut, calculer le taux de bon classement de test pour chacune des trois fonctions d'activation.
- **Mod1** : avec un learning rate de 10^{-2} et un **dropout_p = 0.55** et tous les autres paramètres du réseau par défaut, calculer le taux de bon classement de test pour chacune des trois fonctions d'activation.
- **Mod2** : avec un learning rate de 10^{-3} et tous les paramètres du réseau par défaut, calculer le taux de bon classement de test pour chacune des trois fonctions d'activation.
- **Mod2** : avec un learning rate de 10^{-3} et **convProfond = [50, 150, 300]** et tous les autres paramètres du réseau par défaut, calculer le taux de bon classement de test pour chacune des trois fonctions d'activation.
- **Mod1** : avec un learning rate de 10^{-2} et un **dropout_p = 0.1** et tous les autres paramètres du réseau par défaut, calculer le taux de bon classement de test pour chacune des trois fonctions d'activation.
- **Mod2** : avec un learning rate de 10^{-3} et **dropout_p = 0.1** et tous les autres paramètres du réseau par défaut, calculer le taux de bon classement de test pour chacune des trois fonctions d'activation.

1.2.1 Quelques précautions et directives

- Attention : Au fil des epochs, calculer la performance de test et la stocker au fur et à mesure qu'elle s'améliore sinon conserver la meilleure valeur obtenu sur tout le cycle d'entrainement.
- Utiliser les fonctions fournies en cours si besoin d'afficher des animations.
- Vous pouvez utiliser google Collab pour gagner en temps de calcul en mode TPU.
- Placer les deux instructions suivantes avant et après chaque entrainement d'un modèle pour bien utiliser la carte graphique (si disponible).

[ ]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")  ## pour␣
 ↪lancer un calcul sur GPU si disposible
torch.cuda.empty_cache() ## libérer la mémoire de la GPU à la fin du calcul.
```