I. Introduction

In this lab, I implement the EEGNet and DeepConvNet by pytorch on python. Then apply these models on BCI dataset. In the experiment, I try to use different activation functions on the models. In the end, I got 82% accuracy on the test set.

II. Experiment set up

A. The detail of your model

EEGNet:

```
1   import torch.nn as nn
2   import torch.nn.functional as F
3
4
5   class EEGnet(nn.Module):
```

First, I import the **torch.nn**. Then, I declare a class and make it inheritance the **nn.Module**.

```
6       def __init__(self, activation = nn.ELU()):
7           super().__init__()
```

In the constructor of it, I use a parameter "**activation**" to decide which activation function should this EEGNet use. Then, I follow the instruction of EEGNet to create it.

```
8           self.firstconv = nn.Sequential(
9               nn.Conv2d(1, 16, kernel_size=(1, 51), stride=(1, 1), padding=(0, 25), bias=False),
10              nn.BatchNorm2d(16)
11          )
12
```

In the first section, I follow the spec, declaring a 2D-convolution layer with 16 filter and a batch normalization layer.

```
13          self.depthwiseConv = nn.Sequential(
14              nn.Conv2d(16, 32, kernel_size=(2, 1), stride=(1, 1), groups=16, bias=False),
15              nn.BatchNorm2d(32),
16              activation,
17              nn.AvgPool2d(kernel_size=(1, 4), stride=(1, 4), padding=0),
18              nn.Dropout(0.25)
19          )
20
```

In the second section, I follow the spec, declaring a 2D-convolution layer with 16 filter and divide to 16 groups. And a batch normalization layer, activation layer, 2D-average pooling layer, and dropout layer with dropout rate 0.25.

```
21          self.separableConv = nn.Sequential(
22              nn.Conv2d(32, 32, kernel_size=(1, 15), stride=(1, 1), padding=(0, 7), bias=False),
23              nn.BatchNorm2d(32),
24              activation,
25              nn.AvgPool2d(kernel_size=(1, 8), stride=(1, 8), padding=0),
26              nn.Dropout(0.25)
27          )
28
```

In the third section, I follow the spec, declaring a 2D-convolution layer with 32 filter. And a batch normalization layer, activation layer, 2D-average pooling layer, and dropout layer with dropout rate 0.25.

```
29          self.flatten = nn.Flatten()
30          self.classify = nn.Linear(736, 2)
31
```

After that, the network need to flatten the output of convolution layer, making it to be able to be fed into the fully connected layer.

```
32      def forward(self, x):
33          out = self.firstconv(x)
34          out = self.depthwiseConv(out)
35          out = self.separableConv(out)
36          out = self.flatten(out)
37          out = self.classify(out)
38          return out
```

After finishing the constructor, it also need to define the function "**forward**" by users. In this part, I just follow the order, put the input data layer by layer.

DeepConvNet:

```
2   import torch.nn as nn
3   import torch.nn.functional as F
4
5   class DeepConvNet(nn.Module):
6       def __init__(self, activation = nn.ELU()):
7           super().__init__()
8
```

First, I import the **torch.nn**. Then, I declare a class and make it inheritance the **nn.Module**. In the constructor of it, I use a parameter "**activation**" to decide which activation function should this EEGNet use. Then, I follow the instruction of EEGNet to create it.

```
        self.conv1 = nn.Sequential(
            nn.Conv2d(1, 25, kernel_size=(1, 5)),
            nn.Conv2d(25, 25, kernel_size=(2, 1)),
            nn.BatchNorm2d(25),
            activation,
            nn.MaxPool2d((1,2)),
            nn.Dropout(0.5)
        )

```

In the first section, I follow the spec, declaring 2 2D-convolution layer with both 25 filters but different kernel size. And a batch normalization layer, activation layer, 2D-average pooling layer, and dropout layer with dropout rate 0.5.

```
        self.conv2 = nn.Sequential(
            nn.Conv2d(25, 50, kernel_size=(1, 2)),
            nn.BatchNorm2d(50),
            activation,
            nn.MaxPool2d((1,2)),
            nn.Dropout(0.5)
        )
```

In the second section, I follow the spec, declaring 2 2D-convolution layer with both 50 filters. And a batch normalization layer, activation layer, 2D-average pooling layer, and dropout layer with dropout rate 0.5.

```
        self.conv3 = nn.Sequential(
            nn.Conv2d(50, 100, kernel_size=(1, 2)),
            nn.BatchNorm2d(100),
            activation,
            nn.MaxPool2d((1,2)),
            nn.Dropout(0.5)
        )
```

In the third section, I follow the spec, declaring 2 2D-convolution layer with both 100 filters. And a batch normalization layer, activation layer, 2D-average pooling layer, and dropout layer with dropout rate 0.5.

```
        self.conv4 = nn.Sequential(
            nn.Conv2d(100, 200, kernel_size=(1, 2)),
            nn.BatchNorm2d(200),
            activation,
            nn.MaxPool2d((1,2)),
            nn.Dropout(0.5)
        )

```

In the forth section, I follow the spec, declaring 2 2D-convolution layer with

both 200 filters. And a batch normalization layer, activation layer, 2D-average pooling layer, and dropout layer with dropout rate 0.5.

```
40          self.flatten = nn.Flatten()
41          self.FC = nn.Linear(9000, 2)
```

After that, the network need to flatten the output of convolution layer, making it to be able to be fed into the fully connected layer.

```
43      def forward(self, x:torch.Tensor):
44          out = x.reshape((x.shape[0], 1, 2, 750))
45          out = self.conv1(out)
46          out = self.conv2(out)
47          out = self.conv3(out)
48          out = self.conv4(out)
49          out = self.flatten(out)
50          out = self.FC(out)
51          return out
```

After finishing the constructor, it also need to define the function "**forward**" by users. In this part, I first use the "**reshape**" function to fit the first convolution layer's shape. Then, put the input data layer by layer.

Training model:

```
train_data, train_label, test_data, test_label = read_bci_data()

train_data = torch.Tensor(train_data).to(device)
train_label = torch.Tensor(train_label).type(torch.LongTensor).to(device)
test_data = torch.Tensor(test_data).to(device)
test_label = torch.Tensor(test_label).type(torch.LongTensor).to(device)

dataset = MyDataset(train_data, train_label)
train_loader = torch.utils.data.DataLoader(dataset, batch_size=64, shuffle=True, drop_last=True)
```

First, I use the "**read_bci_data**" function to read the data. Then I convert them to the torch Tensor datatype and put them on GPU's memory. After that, I use pytorch's **DataLoader** to cut the training data into minibatches.

```
net = EEGnet(nn.ELU()).to(device)
optimizer = torch.optim.Adam(net.parameters(), lr = 0.02)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.5, patience=10,
```

Then, I declare my model and put it on GPU's memory. Besides, I use adam optimizer. And use the learning rate scheduler to reduce learning rate when the test loss start increasing.

```
for epoch in range(1, EPOCH+1):
    total_acc = 0
    total_item = 0
    for (x, y_true) in train_loader:
        y_pred = net(x)
        output = loss(y_pred, y_true)
        net.zero_grad()
        output.backward()
        optimizer.step()
```

In each epoch, I would load the batched data from data loader one by one. For each batch, I would put it into my model to get the predicted output. Then, calculate its loss. After that, use this information to do the backpropagation and gradient descend.

```
with torch.no_grad():
    y_pred = net(test_data)
    output = loss(y_pred, test_label)
```

Then, put the test data to the model, calculate its accuracy and loss.

```
scheduler.step(output.item())
```

And tell the learning rate scheduler whether to reduce the learning rate.

B. Explain the activation function (ReLU, Leaky ReLU, ELU)

ReLU: $f(x) = \max(0, x)$

Leaky ReLU: $f(x) = \max(0.1x, x)$

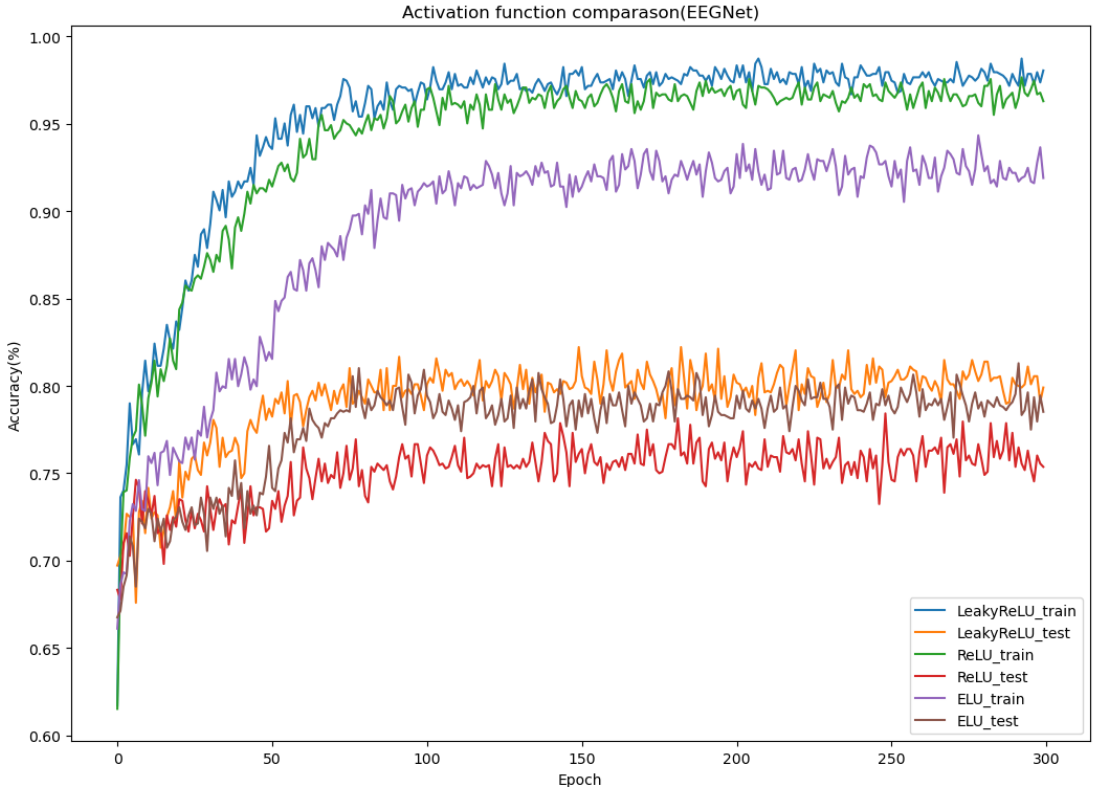
ELU: $f(x) = \max(0, x) + \min(0, \alpha(e^x - 1))$

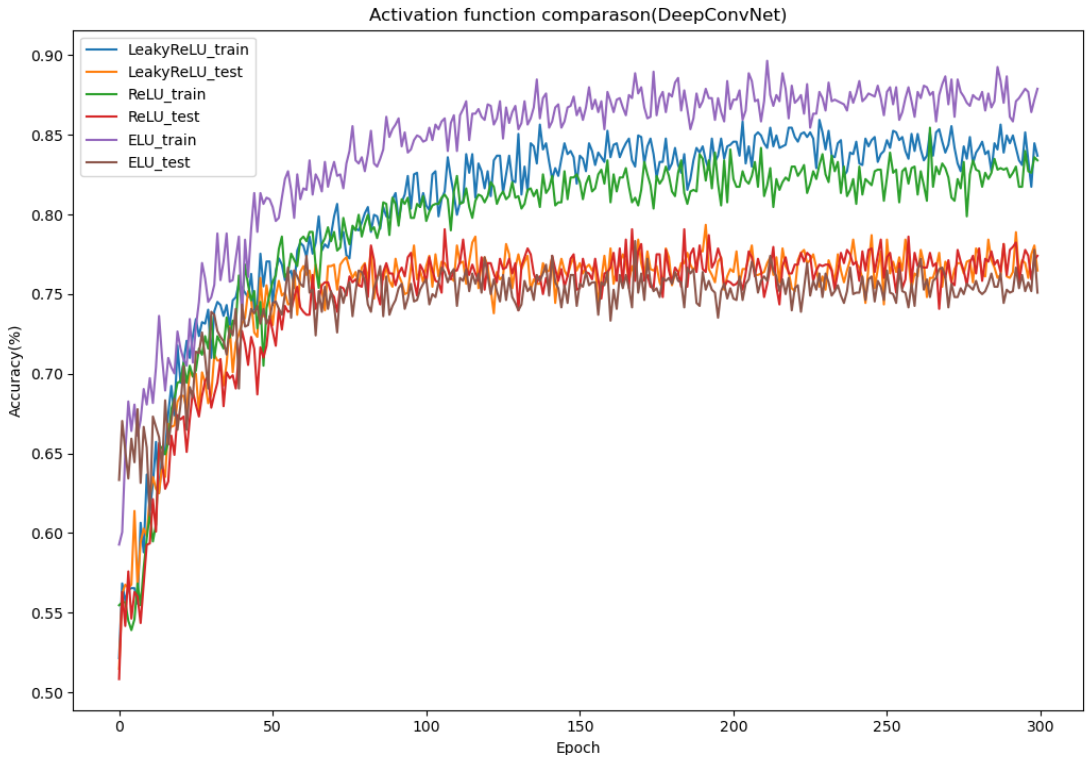
III. Experimental results

A. The highest testing accuracy

	Leaky ReLU	ReLU	ELU
EEGNet	**82.22%**	78.43%	81.3%
DeepConvNet	**79.35%**	**79.07%**	78.33%

B. Comparison figures





IV. Discussion

In the experiment results of EEGNet, we can see that the relu and leaky relu function have a better performance on training set. Their training accuracy grew faster than the ELU one. However, in the test set, all of their performance is very close. I've

done the experiment several times. I found that these 3 activation function all can have the best accuracy. It's hard to say which activation is better in EEGNet.

In the experiment results of DeepConvNet, it had a different result. The ELU activation function had a superior performance toward all the other activation functions. As for the test set, it's like the EEGNet. All these 3 activation function had similar performance.

Comparing these 2 networks, we can see that DeepConvNet had a closer accuracy between training set and test set than EEGNet. I think it's because that the DeepConvNet use a bigger dropout rate than EEGNet did. So that DeepConvNet has smaller overfitting problem than EEGNet.