Lab3: EEG classification

b06b01073

April 2023

1 Introduction

In this lab, we are going to classify EEG signal by using EEGNet. EEGNet has two major components, which are depth-wise convolution and depth-wise separable convolution. In depth-wise convolution, each channel of the input data is processed by a unique kernel. This method uses less parameters and avoids the risk of overfitting. The depth-wise separable convolution is composed of depth-wise convolution and point-wise convolution, on the other hand. Point-wise convolution summarize the information along the depth dimension.

DeepConvNet and ShallowConvNet were also implemented. The result shows that EEGNet outperforms both DeepConvNet and ShallowConvNet. Notably, the fact that EEGNet beats DeepConvNet in the classification task indicates that stronger network(more parameters) easily suffers from overfitting.

2 Experiment set up

2.1 The detail of your model

2.1.1 EEGNet

The implementation of the EEGNet is the same as the one shown in the spec. The source code is show in Source Code 1.

Source Code 1: The source code of EEGNet

```
class EEGNet(nn.Module):
def __init__(self, activation='relu'):
    super().__init__()
    self.activation = get_activation(activation)
    print(f'Initializing model {self}...')
    print(f'Using activation function
     ↪  {self.activation}')

    self.firstconv = nn.Sequential(
```

```
            nn.Conv2d(in_channels=1, out_channels=16,
            ↪  kernel_size=(1, 51), stride=(1, 1),
            ↪  padding=(0, 25), bias=False),
            nn.BatchNorm2d(num_features=16, eps=1e-5,
            ↪  momentum=0.1, affine=True,
            ↪  track_running_stats=True)
        )

        self.depthwiseConv = nn.Sequential(
            nn.Conv2d(in_channels=16, out_channels=32,
            ↪  kernel_size=(2, 1), stride=(1, 1),
            ↪  groups=16, bias=False),
            nn.BatchNorm2d(num_features=32, eps=1e-5,
            ↪  momentum=0.1, affine=True,
            ↪  track_running_stats=True),
            self.activation,
            nn.AvgPool2d(kernel_size=(1, 4), stride=(1, 4),
            ↪  padding=0),
            nn.Dropout(p=0.25)
        )

        self.separableConv = nn.Sequential(
            nn.Conv2d(in_channels=32, out_channels=32,
            ↪  kernel_size=(1, 15), stride=(1, 1),
            ↪  padding=(0, 7), bias=False),
            nn.BatchNorm2d(num_features=32, eps=1e-5,
            ↪  momentum=0.1, affine=True,
            ↪  track_running_stats=True),
            self.activation,
            nn.AvgPool2d(kernel_size=(1, 8), stride=(1, 8),
            ↪  padding=0),
            nn.Dropout(p=0.25)
        )

        self.classify = nn.Sequential(
            nn.Linear(736, 2, bias=True),
        )

    def forward(self, x):
```

```
        x = self.firstconv(x)
        x = self.depthwiseConv(x)
        x = self.separableConv(x)
        x = torch.flatten(x, 1)
        x = self.classify(x)

        return x
```

2.1.2 DeepConvNet

The implementation of the DeepConvNet is the same as the one shown in the spec. The Source Code is show in Source Code 2.

Source Code 2: The source code of EEGNet

```
class DeepConvNet(nn.Module):
    def __init__(self, activation='relu'):
        super().__init__()

        self.activation = get_activation(activation)
        print(f'Initializing model {self}...')
        print(f'Using activation function
        ↪  {self.activation}')

        self.header = nn.Conv2d(in_channels=1,
        ↪  out_channels=25, kernel_size=(1, 5))

        self.hidden_dims = [25, 25, 50, 100, 200]
        self.hidden_kernel_size = [(2, 1), (1, 5), (1,
        ↪  5), (1, 5)]
        self.pool_kernel_size = (1, 2)

        self.conv_layers1 = nn.Sequential(
            nn.Conv2d(in_channels=self.hidden_dims[0],
            ↪  out_channels=self.hidden_dims[1],
            ↪  kernel_size=self.hidden_kernel_size[0]),
            nn.BatchNorm2d(self.hidden_dims[1]),
            self.activation,

            ↪  nn.MaxPool2d(kernel_size=self.pool_kernel_size),
            nn.Dropout(p=0.5),
```

```
        )
        self.conv_layers2 = nn.Sequential(
            nn.Conv2d(in_channels=self.hidden_dims[1],
             ↪  out_channels=self.hidden_dims[2],
             ↪  kernel_size=self.hidden_kernel_size[1]),
            nn.BatchNorm2d(self.hidden_dims[2]),
            self.activation,

             ↪  nn.MaxPool2d(kernel_size=self.pool_kernel_size),
            nn.Dropout(p=0.5),
        )
        self.conv_layers3 = nn.Sequential(
            nn.Conv2d(in_channels=self.hidden_dims[2],
             ↪  out_channels=self.hidden_dims[3],
             ↪  kernel_size=self.hidden_kernel_size[2]),
            nn.BatchNorm2d(self.hidden_dims[3]),
            self.activation,

             ↪  nn.MaxPool2d(kernel_size=self.pool_kernel_size),
            nn.Dropout(p=0.5),
        )
        self.conv_layers4 = nn.Sequential(
            nn.Conv2d(in_channels=self.hidden_dims[3],
             ↪  out_channels=self.hidden_dims[4],
             ↪  kernel_size=self.hidden_kernel_size[3]),
            nn.BatchNorm2d(self.hidden_dims[4]),
            self.activation,

             ↪  nn.MaxPool2d(kernel_size=self.pool_kernel_size),
            nn.Dropout(p=0.5),
        )

        self.fc = nn.Sequential(
            nn.Linear(in_features=8600,
             ↪  out_features=2),
        )

    def forward(self, x):
        x = self.header(x)
```

```
        x = self.conv_layers1(x)
        x = self.conv_layers2(x)
        x = self.conv_layers3(x)
        x = self.conv_layers4(x)

        x = torch.flatten(x, 1)
        x = self.fc(x)

        return x
```

2.2 Explain the activation function

The activation function provides non-linearity in the model. In this lab, the activation functions are ReLU, LeakyReLU and ELU

2.2.1 ReLU

ReLU provides non-linearity by setting the output to 0 if the input is less than 0. The formula of ReLU is

$$ReLU(x) = \begin{cases} x & \text{, if } x \geq 0 \\ 0 & \text{, otherwise} \end{cases}$$

.

2.2.2 LeakyReLU

The LeakyReLU function is

$$LeakyReLU(x; s) = \begin{cases} x & \text{, if } x \geq 0 \\ xs & \text{, otherwise} \end{cases}$$

, where s is a fixed parameter. Note ReLU is a special case of LeakyReLU where s is set to 0.

2.2.3 ELU

The ELU function is

$$ELU(x) = \begin{cases} x & \text{, if } x \geq 0 \\ \alpha(e^x - 1) & \text{, otherwise} \end{cases}$$

. The ELU function doesn't have a kink at $x = 0$. Thus, it is a more smooth activation function.

3 Experimental results

3.1 The highest testing accuracy

The highest testing accuracy of the EEGNet is shown in Figure 1. The highest testing accuracy of the DeepConvNet is shown in Figure 2.

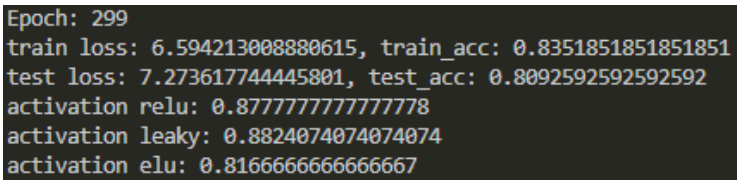

Figure 1: The highest testing accuracy of EEGNet

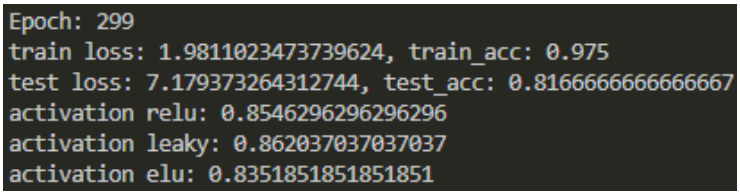

Figure 2: The highest testing accuracy of DeepConvNet

3.2 Comparison figures

The comparison figure of EEGNet and DeepConvNet are shown in Figure 3 and Figure 4, respectively.

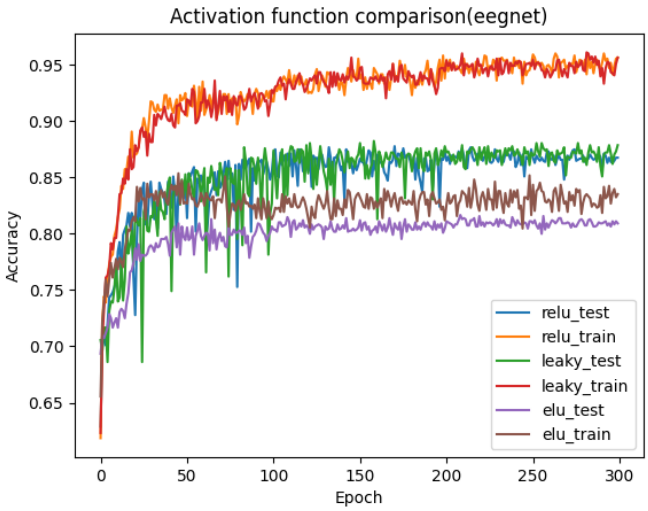

Figure 3: The comparison figure of EEGNet

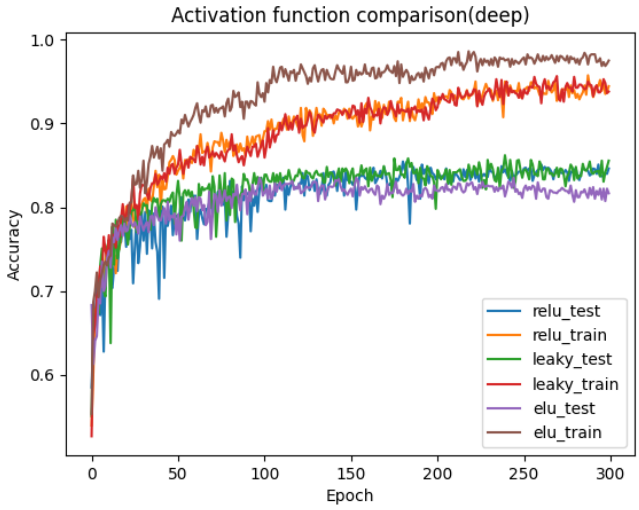

Figure 4: The comparison figure of DeepConvNet

3.3 Discussion

In this lab, one of the major issue is the problem of overfitting. EEGNet utilize dropout function to prevent overfitting. However, the problem is still quite severe if L2-regulariztion wasn't added in my implementation. Multiple techniques

were also implemented to fight against overfitting, such as data augmentation(by adding Gaussian noise) and min-max normalization, but failed to work.

3.4 Extra

I also implemented the ShallowConvNet specified in the EEGNet paper as well. The result are show in Figure 5 and Figure 6.

```
Epoch: 299
train loss: 3.3472752571105957, train_acc: 0.9212962962962963
test loss: 7.797775745391846, test_acc: 0.825
activation relu: 0.8333333333333334
activation leaky: 0.8388888888888889
activation elu: 0.8351851851851851
```

Figure 5: The highest testing accuracy of ShallowConNet

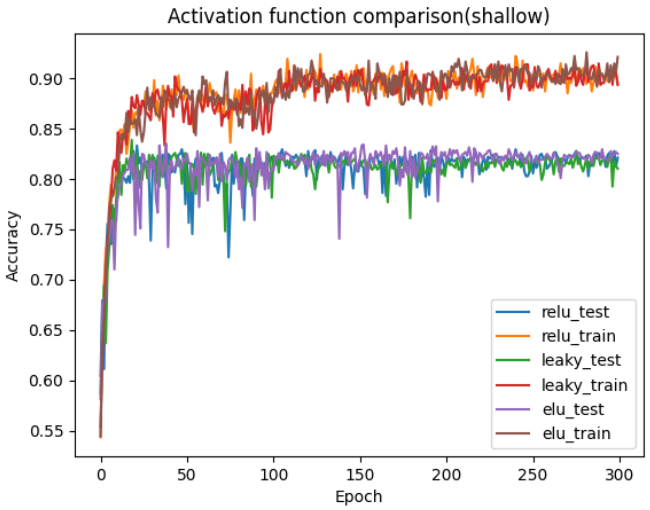

Figure 6: The comparison figure of ShallowConvNet