Deep Learning and Practice Lab2: EEG Classification

Zhi-Yi Chin
joycenerd.cs09@nycu.edu.tw

July 26, 2021

1 Introduction

In this lab we are implementing two EEG classification model which are EEGNet and DeepConvNet. Also, we are changing the activation function (ELU, ReLU, LeakyReLU) in the model and see the difference. The dataset of for this lab is from the BCI competition and the shape is (C=1, H=2, W=750).

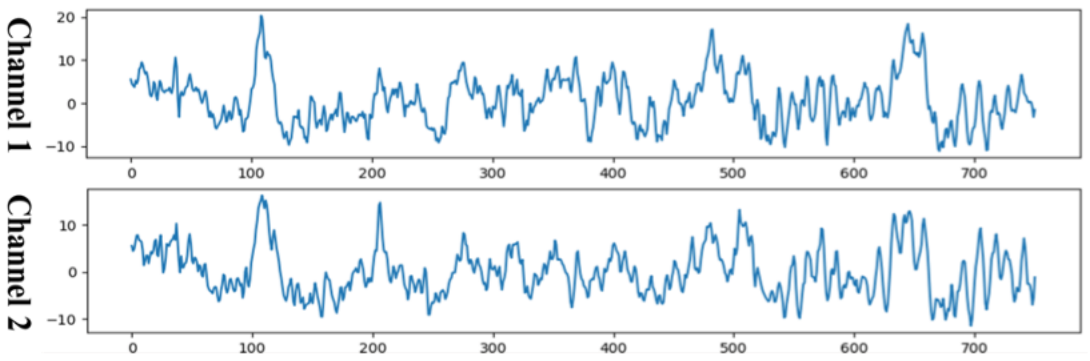

Figure 1: This is the data we are using for EEG classification.

2 Implementation Detail

In this section, we will introduce the method and implementation detail we are using.

2.1 Data Preprocessing

First, we read in the train and test data from the code provided by the TA. Since we will be using Pytorch in this lab, we build an EEGDataset class for the data. This dataset is inherited from the Dataset class in Pytorch. In this dataset class, we read the signal data and the label. Then we first transform it as a NumPy array and then transform it to Tensor. In Pytorch, everything needs to be in Tensor type. The EEGDataset class is shown in Listing 1. Then we use the DataLoader also from Pytorch to wrap our EEGDataset. We set the batch size to 256 and shuffle to True.

```python
class EEGDataset(Dataset):
    # EEG dataset class
    def __init__(self,X,y):
        """
        :param X: data
        :param y: label
        """
        self.X=X
        self.y=y


    def __len__(self):
        return len(self.y)


    def __getitem__(self,index):
        input=np.asarray(self.X[index])
        target=np.asarray(self.y[index])
        return torch.from_numpy(input),torch.from_numpy(target)
```

Listing 1: EEGDataset class.

2.2 Model

2.2.1 EEGNet

EEGNet can mainly separate to five parts.

1. Inputs.
2. Normal convolution.
3. Depthwise convolution.
4. Separable convolution.
5. Classifier

In the normal convolution, the network learns frequency filters. Then in the depthwise convolution, the network connects to each feature map individually to learn frequency-specific spatial filters. The separable convolution combines a depth-wise convolution and learns a temporal summary for each feature map individually, followed by a pointwise convolution, which learns how to mix the feature maps optimally. We flatten the feature maps in the classifier, use one linear layer, and outputs two-class probability.

As for the network structure, the only difference between our network and the original paper is choosing the activation function we want to use. In the original paper, the activation function is `elu`. The network structure of EEGNet is shown in Figure 2 and the code is in Listing 2.

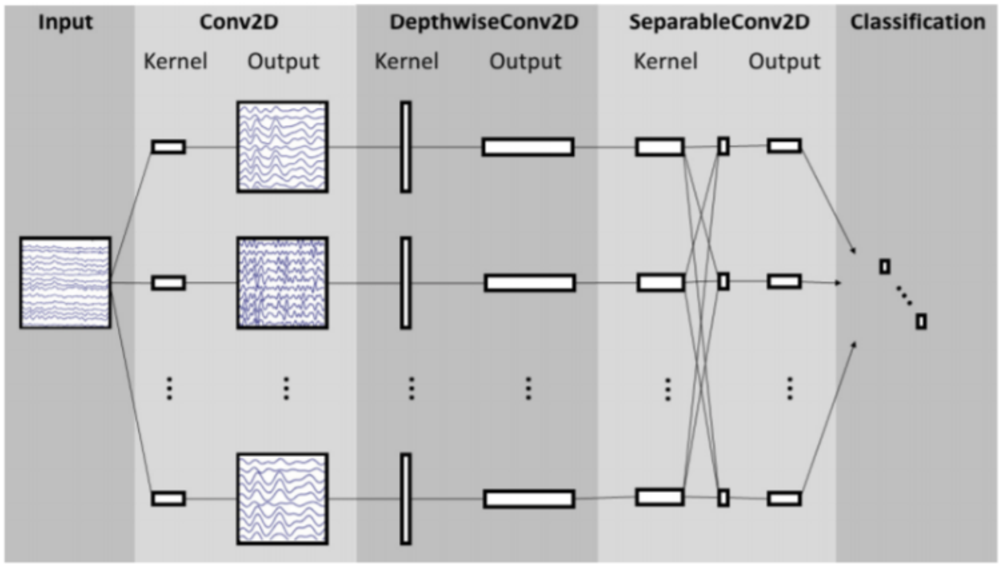

Figure 2: EEGNet architecture

```python
class EEGNet(nn.Module):
    def __init__(self,activation):
        super(EEGNet,self).__init__()
        self.firstconv=nn.Sequential(
            nn.Conv2d(1,16,kernel_size=(1,51),stride=(1,1),padding=(0,25),
                      bias=False),
            nn.BatchNorm2d(16,eps=1e-5,momentum=0.1,affine=True,
                           track_running_stats=True)
        )
        self.depthwiseConv=nn.Sequential(
            nn.Conv2d(16,32,kernel_size=(2,1),stride=(1,1),groups=16,
                      bias=False),
            nn.BatchNorm2d(32,eps=1e-5,momentum=0.1,affine=True,
                           track_running_stats=True),
            activation,
            nn.AvgPool2d(kernel_size=(1,4),stride=(1,4),padding=0),
            nn.Dropout(p=0.25)
        )
        self.separableConv=nn.Sequential(
            nn.Conv2d(32,32,kernel_size=(1,15),stride=(1,1),padding=(0,7),
                      bias=False),
            nn.BatchNorm2d(32,eps=1e-5,momentum=0.1,affine=True,
                           track_running_stats=True),
            activation,
            nn.AvgPool2d(kernel_size=(1,8),stride=(1,8),padding=0),
            nn.Dropout(p=0.25)
        )
        self.classifier=nn.Linear(in_features=736,out_features=2,bias=True)


    def forward(self,x):
        x=self.firstconv(x)
        x=self.depthwiseConv(x)
        x=self.separableConv(x)
        x=x.view(x.shape[0],-1)
        out=self.classifier(x)
        return out
```

Listing 2: EEGNet network.

2.2.2 DeepConvNet

DeepConvNet is used to compared with EEGNet. DeepConvNet contains 6 convolution blocks. Which is a larger network compare to EEGNet. Figure 3 shows the architecture of DeepConvNet and code of DeepConvNet shows in Listing 3.

Layer	**# filters**	**size**	**# params**	**Activation**	**Options**
Input		(C, T)			
Reshape		(1, C, T)			
Conv2D	25	(1, 5)	150	Linear	mode = valid, max norm = 2
Conv2D	25	(C, 1)	25 * 25 * C + 25	Linear	mode = valid, max norm = 2
BatchNorm			2 * 25		epsilon = 1e-05, momentum = 0.1
Activation				ELU	
MaxPool2D		(1, 2)			
Dropout					p = 0.5
Conv2D	50	(1, 5)	25 * 50 * C + 50	Linear	mode = valid, max norm = 2
BatchNorm			2 * 50		epsilon = 1e-05, momentum = 0.1
Activation				ELU	
MaxPool2D		(1, 2)			
Dropout					p = 0.5
Conv2D	100	(1, 5)	50 * 100 * C + 100	Linear	mode = valid, max norm = 2
BatchNorm			2 * 100		epsilon = 1e-05, momentum = 0.1
Activation				ELU	
MaxPool2D		(1, 2)			
Dropout					p = 0.5
Conv2D	200	(1, 5)	100 * 200 * C + 200	Linear	mode = valid, max norm = 2
BatchNorm			2 * 200		epsilon = 1e-05, momentum = 0.1
Activation				ELU	
MaxPool2D		(1, 2)			
Dropout					p = 0.5
Flatten					
Dense	N			softmax	max norm = 0.5

Figure 3: DeepConvNet architecture, where C=2, T=750 and N=2

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

        self.conv1=nn.Conv2d(1,25,kernel_size=(1,5))
        self.first_conv_block=nn.Sequential(
            nn.Conv2d(25,25,kernel_size=(2,1)),
            nn.BatchNorm2d(25,eps=1e-5,momentum=0.1),
            activation,
            nn.MaxPool2d(kernel_size=(1,2)),
            nn.Dropout(p=0.5)
        )

        conv_layers=[]
        channel=25
        for i in range(4):
            conv_layers.append(self._make_conv_block(channel))
            channel*=2
        self.conv_blocks=nn.Sequential(*conv_layers)

        self.classifier=nn.Linear(7600,2)


    def _make_conv_block(self,channel):
        return nn.Sequential(
            nn.Conv2d(channel,channel*2,kernel_size=(1,5)),
            nn.BatchNorm2d(channel*2,eps=1e-5,momentum=0.1),
            self.activation,
            nn.MaxPool2d(kernel_size=(1,2)),
            nn.Dropout(p=0.5)
        )


    def forward(self,x):
        x=self.conv1(x)
        x=self.first_conv_block(x)
        x=self.conv_blocks(x)
        x=x.view(x.shape[0],-1)
        out=self.classifier(x)
        return out
```

Listing 3: DeepConvNet model.

2.2.3 Initialize weights

In Pytorch, we do not need to initialize the model weights if we do not want to; it will help us random initialize it, but sometimes random initialization results in a bad outcome, so we initialize our model weights. If the layer is a convolution layer, we use `kaiming_uniform_` to initialize the weights and initialize the bias to 0. If the layer is a batch norm layer, we set the weights to 1 and bias to 0. At last, if it is a linear layer, then we use `kaiming_uniform_` to initialize the weights and initialize the bias to 0. The code for model initialization is in Listing 4

```
def initialize_weights(m):
    """
    Initialize the model
    :param m: model
    """
    if isinstance(m,nn.Conv2d):
        nn.init.kaiming_uniform_(m.weight.data,nonlinearity="relu")
        if m.bias is not None:
            nn.init.constant_(m.bias.data,0)
    elif isinstance(m,nn.BatchNorm2d):
        nn.init.constant_(m.weight.data,1)
        nn.init.constant_(m.bias.data,0)
    elif isinstance(m,nn.Linear):
        nn.init.kaiming_uniform_(m.weight.data)
        if m.bias is not None:
            nn.init.constant_(m.bias.data,0)
```

Listing 4: Model initialization function.

2.3 Training process

First, we construct our data mention in Section 2.1. Then we initialize the model. Whether it is EEGNet or DeepConvNet, we will pass in the activation function we are using. The weight initialization step is optional afterward. Since our task is a classification task, so we use the cross-entropy loss as our loss function. And we use Adam as our optimizer with `amsgrad` and `weight_decay` as optional. We train our model for 500 epochs, and in every epoch, we do validation. If the validation accuracy is higher than the previous best accuracy, we save the model. We save the model into a checkpoint file at the end of the training.

2.4 Activation function

We choose three activation functions to use in our experiment, which are ELU, ReLU and LeakyReLU.

2.4.1 ELU

The full name of ELU is Exponential Linear Unit. It is a function that tends to converge cost to zero faster and produce more accurate results. Different from other activation functions, ELU has an extra alpha constant which should be a positive number. ELU is similar to ReLU except for negative inputs. On the other hand, ELU becomes smooth slowly until its output is equal to $-\alpha$, whereas ReLU sharply smoothes. The ELU function is as Eq. 1 and the derivative of ELU function is as Eq. 2.

Some of the pros of ELU are: (1) ELU becomes smooth slowly until its output is equal to $-\alpha$ whereas ReLU sharply smoothes. (2) ELU is a solid alternative to ReLU. (3) Unlike ReLU, ELU can produce negative outputs. ELU has one con: when $x > 0$, it can blow up the activation with the output range of [0,inf].

$$R(z) = \begin{cases} z & z > 0 \\ \alpha \cdot (e^z - 1) & z <= 0 \end{cases} \tag{1}$$

$$R'(z) = \begin{cases} 1 & z > 0 \\ \alpha \cdot e^z & z < 0 \end{cases} \tag{2}$$

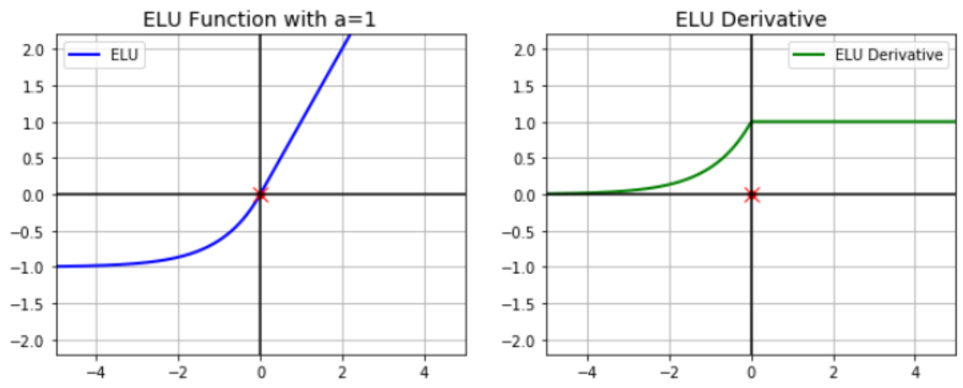

Figure 4: The left hand side is the ELU function and the right hand side is the derivative of the ELU function.

2.4.2 ReLU

ReLU is a recent invention that stands for Rectified Linear Units. The formula is deceptively simple: $\max(0, z)$. Despite its name and appearance, it is not linear and provides the same benefits as Sigmoid but with better performance. Eq. 3 shows the function of ReLU and Eq. 4 shows the derivative of ReLU.

Some of the pros of ReLU are: (1) ReLU avoids and rectifies vanishing gradient problems. (2) ReLU is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations. Moreover, some of the cons of ReLU are: (1) One of its limitations is that it should only be used within hidden layers of a neural network model. (2) ReLU could result in dead neurons (neurons never update). (3) The range of ReLU is [0,inf), which means it can blow up the activation.

$$R(z) = \begin{cases} z & z > 0 \\ 0 & z <= 0 \end{cases} \tag{3}$$

$$R'(z) = \begin{cases} 1 & z > 0 \\ 0 & z < 0 \end{cases} \tag{4}$$

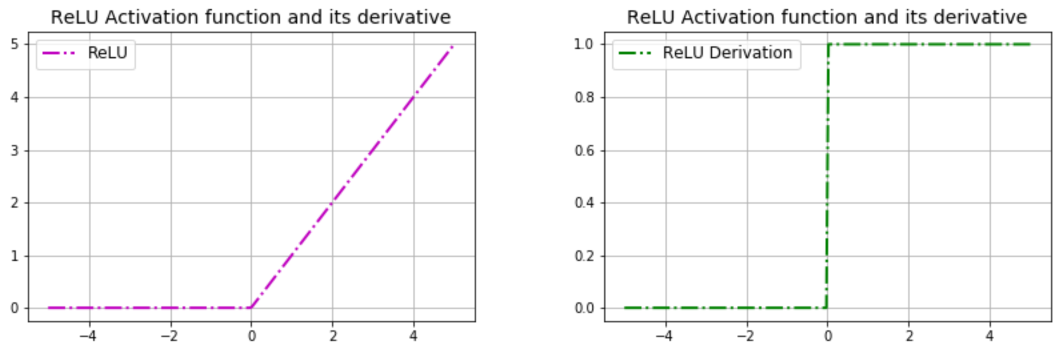

Figure 5: The left hand side is the ReLU function and the right hand side is the derivative of the ReLU function.

2.4.3 LeakyReLU

LeakyReLU is a variant of ReLU. Instead of being 0 when $x < 0$, a LeakyReLU allows a small, non-zero, constant gradient α (normally $\alpha = 0.01$). However, the consistency of the benefit across tasks is presently unclear. Eq. 5 shows the function of LeakyReLU and Eq. 6 shows the derivative of LeakyReLU.

LeakyReLU attempts to fix the "dying ReLU" problem by having a slight negative slope (of 0.01, or so) is the pro of LeakyReLU. However, as LeakyReLU possesses linearity, it cannot be used for complex classification. It lags behind the Sigmoid and Tanh for some of the use cases is the con of LeakyReLU.

$$R(z) = \begin{cases} z & z > 0 \\ \alpha z & z <= 0 \end{cases} \tag{5}$$

$$R'(z) = \begin{cases} 1 & z > 0 \\ \alpha & z < 0 \end{cases} \tag{6}$$

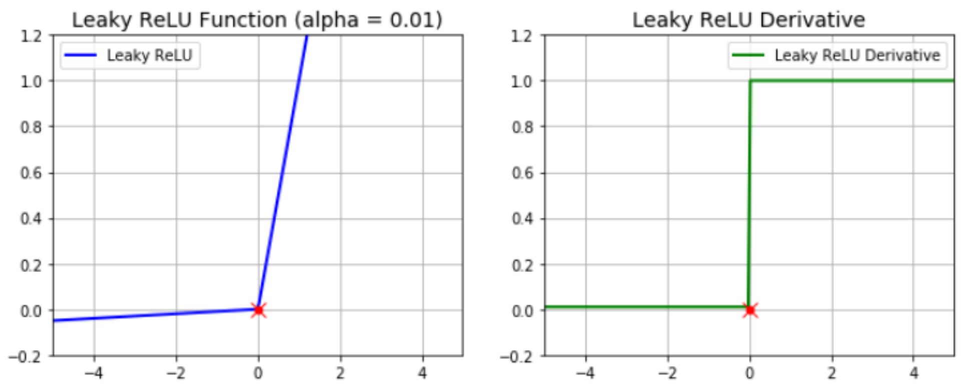

Figure 6: The left hand side if LeakyReLU and the right hand side if the derivative of LeakyReLU

3 Experimental Results

3.1 EEGNet results

	lr	init w	amsgrad	train acc	test acc
ELU	0.005	✗	✓	99.26%	84.07%
ReLU	0.001	✗	✗	99.44%	87.31%
LeakyReLU	0.01	✓	✓	99.44%	**87.87%**

Table 1: This is the best results of EEGNet using three different activation functions. In this table init w means if we have use the weight initialization mention in Section 2.2.3, and amsgrad means whether to set `amsgrad` to `True`.

3.2 DeepConvNet results

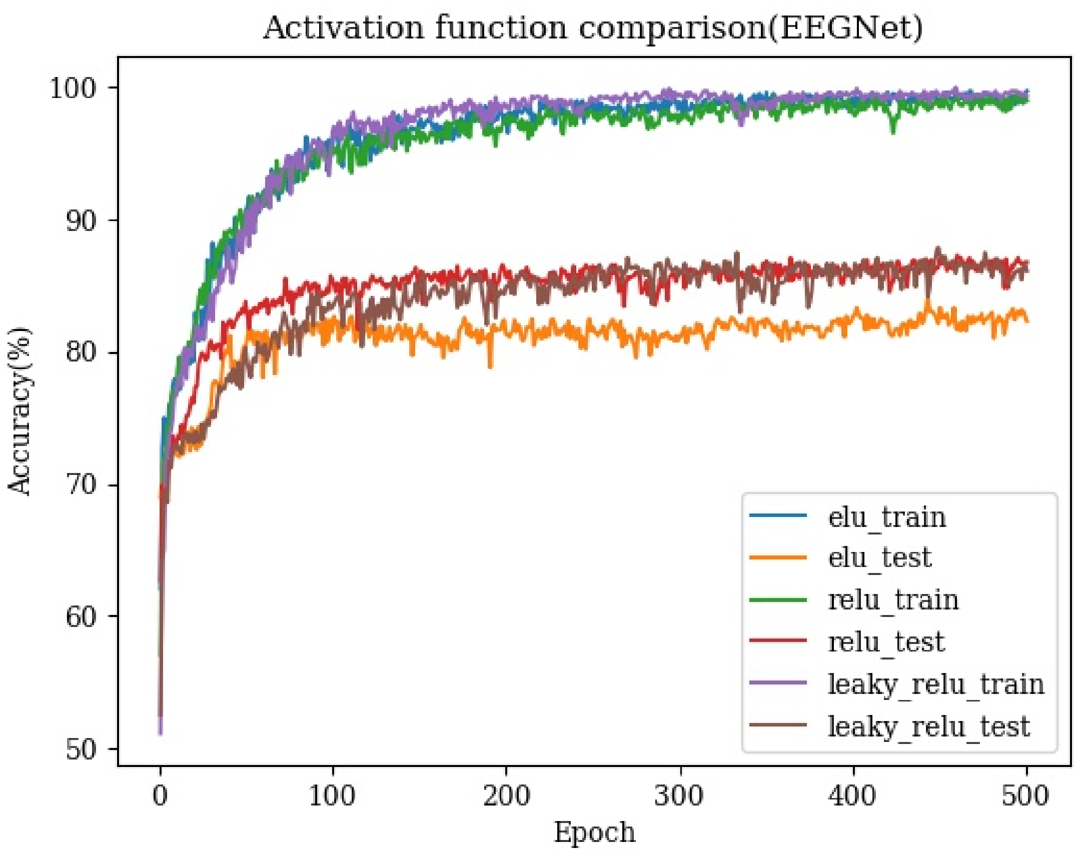

Figure 7: Accuracy curve of EEGNet using different activation functions. We only record the best one in Table 1

	lr	init w	amsgrad	train acc	test acc
ELU	0.001	✗	✓	76.48%	**74.54%**
ReLU	0.01	✗	✗	74.07%	71.02%
LeakyReLU	0.01	✓	✓	72.69%	73.52%

Table 2: This is the best results of DeepConvNet using three different activation functions. In this table init w means if we have use the weight initialization mention in Section 2.2.3, and amsgrad means whether to set `amsgrad` to `True`.

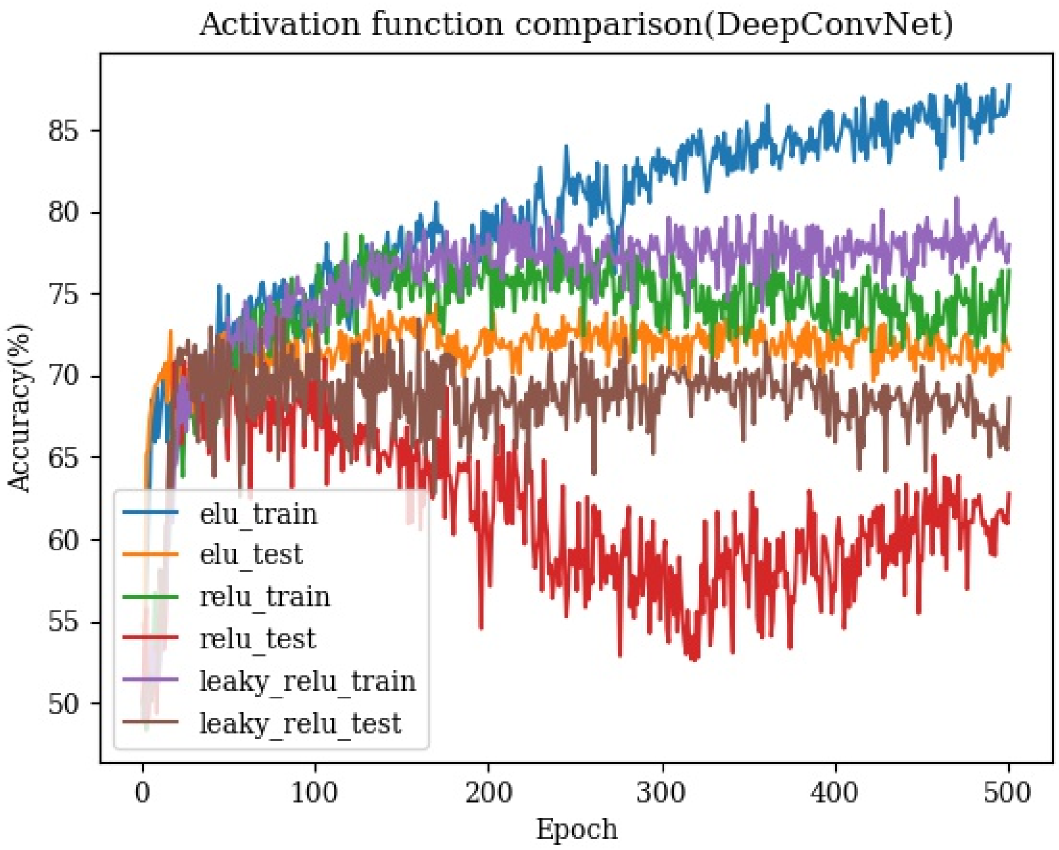

Figure 8: Accuracy curve of DeepConvNet using different activation functions. We only record the best one in Table 2

4 Discussion

From the experimental result in Section 3, we have found a trend. When using ELU as activation function the best results happen when setting `amsgrad` to `True` in the Adam optimizer. When using ReLU as activation function, the best results happen when not initializing our weight mentioned in Section 2.2.3 and set `amsgrad` to `False`. When using LeakyReLU as our activation function, the best results happen when both initializing our weight and set `amsgrad` to `True`. The observation applies to both EEGNet and DeepConvNet, which we find interesting.

In EEGNet, the training accuracy is near 100%, so we think our model is overfitting to our training data, so our testing accuracy cannot be higher than 87%. We add a regularization term to the optimizer, but this move decreases training and testing accuracy. We guess testing data and training data may have a different distribution, so what we learn in training data may not thoroughly apply to the testing data.

We find that DeepConvNet performs much worse compare to EEGNet. The testing accuracy does not exceed 80%. From Figure 8 we can see that training accuracy didn;t exceed 90%. This means the network does not fully learn from the data, so we think DeepConvNet is unsuitable for our data. Also, higher training accuracy does not mean higher testing accuracy in DeepConvNet.