Deep Learning – Lab 2

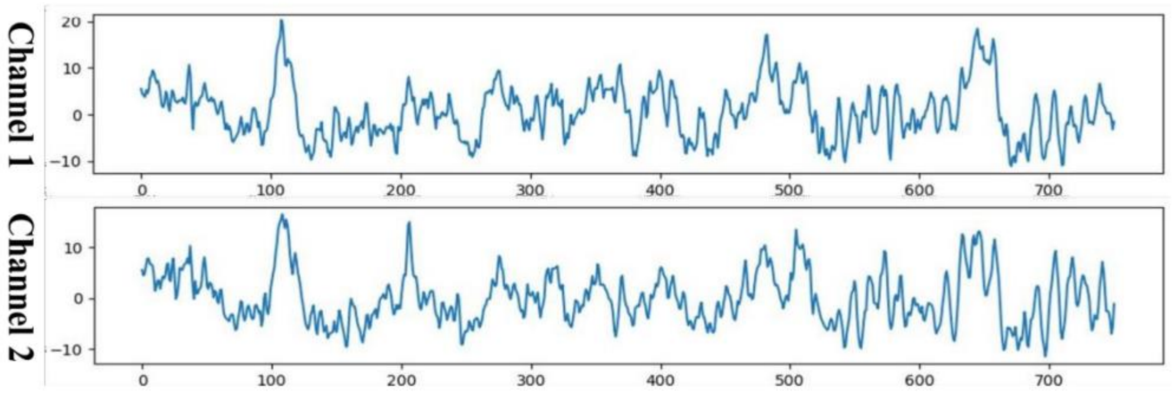
EEG classification

I. Introduction

i. Requirements

- ➢ Implement EEGNet and DeepConvNet to classify EEG Data.
- ➢ Comparison of ReLU, Leaky ReLU, and ELU activation functions.

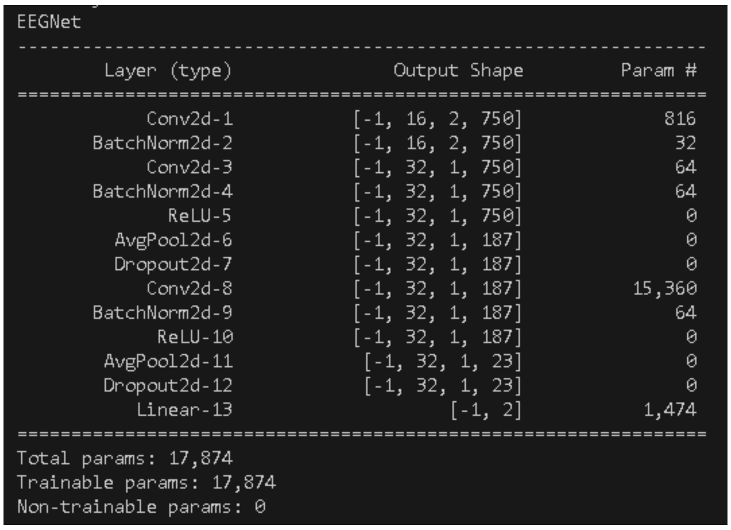
ii. Dataset



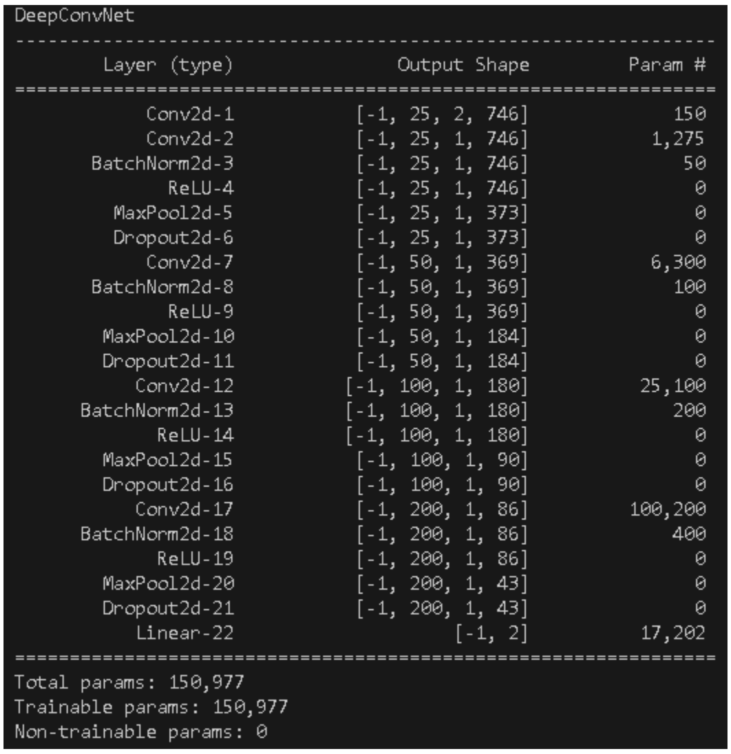
- ➢ The dataset consists of four files: S4b_train.npz, X11b_train.npz, S4b_test.npz and X11b_test.npz. Both S4b and X11b contain data with shape of 540*2*750. Here, 540 represents the number of data samples, 2 is the number of channels, and 750 is the time axis (as shown in the figure).
- ➢ To use CNN as the neural network architecture, we concatenate the two datasets(S4b and X11b) in the dataloader, and expanding them a single dimension into size 1080*1*2*750. The dimension 1 corresponds to the channel number typically used in image data processing (e.g., 3 for RGB images and 1 for grayscale images).
- ➢ Therefore, in this task, we can treat the input data as 1080 samples of grayscale images with a length of 750 and a width of 2. We will use CNN for classification, and the output layer will have two neurons representing the probabilities of the two classes (using Cross Entropy).

II. Experiment Set Up

i. The detail of your model



```
EEGNet
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 16, 2, 750]             816
       BatchNorm2d-2           [-1, 16, 2, 750]              32
            Conv2d-3           [-1, 32, 1, 750]              64
       BatchNorm2d-4           [-1, 32, 1, 750]              64
              ReLU-5           [-1, 32, 1, 750]               0
         AvgPool2d-6           [-1, 32, 1, 187]               0
         Dropout2d-7           [-1, 32, 1, 187]               0
            Conv2d-8           [-1, 32, 1, 187]          15,360
       BatchNorm2d-9           [-1, 32, 1, 187]              64
             ReLU-10           [-1, 32, 1, 187]               0
        AvgPool2d-11            [-1, 32, 1, 23]               0
        Dropout2d-12            [-1, 32, 1, 23]               0
           Linear-13                    [-1, 2]           1,474
================================================================
Total params: 17,874
Trainable params: 17,874
Non-trainable params: 0
```



```
DeepConvNet
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 25, 2, 746]             150
            Conv2d-2           [-1, 25, 1, 746]           1,275
       BatchNorm2d-3           [-1, 25, 1, 746]              50
              ReLU-4           [-1, 25, 1, 746]               0
         MaxPool2d-5           [-1, 25, 1, 373]               0
         Dropout2d-6           [-1, 25, 1, 373]               0
            Conv2d-7           [-1, 50, 1, 369]           6,300
       BatchNorm2d-8           [-1, 50, 1, 369]             100
              ReLU-9           [-1, 50, 1, 369]               0
        MaxPool2d-10           [-1, 50, 1, 184]               0
        Dropout2d-11           [-1, 50, 1, 184]               0
           Conv2d-12          [-1, 100, 1, 180]          25,100
      BatchNorm2d-13          [-1, 100, 1, 180]             200
             ReLU-14          [-1, 100, 1, 180]               0
        MaxPool2d-15           [-1, 100, 1, 90]               0
        Dropout2d-16           [-1, 100, 1, 90]               0
           Conv2d-17           [-1, 200, 1, 86]         100,200
      BatchNorm2d-18           [-1, 200, 1, 86]             400
             ReLU-19           [-1, 200, 1, 86]               0
        MaxPool2d-20           [-1, 200, 1, 43]               0
        Dropout2d-21           [-1, 200, 1, 43]               0
           Linear-22                    [-1, 2]          17,202
================================================================
Total params: 150,977
Trainable params: 150,977
Non-trainable params: 0
```

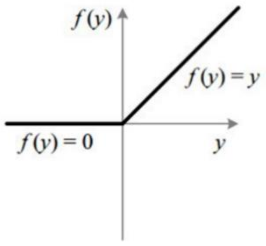
- DeepConvNet: is formed by stacking a large number of convolutional layers to extract signal features using deep convolutions.
- EEGNet: utilizes a three-layer architecture: firstConv, depthWiseConv, and separableConv. This not only significantly reduces parameters and computational complexity but also improves accuracy.
  - First Conv: It is a layer of filters that performs convolution on the input signal to extract signal features.
  - DepthWiseConv: It uses groups to divide the input channels into separate groups, and each group undergoes convolution independently. The number of groups is equal to the number of input channels, which means different input channels do not interact with each other in this convolution layer. While this layer maximizes the reduction in computational complexity through grouping, it cannot capture the interaction between different channels' features.
  - SeparableConv: It utilizes Pointwise Convolution to enable interaction between different channels, addressing the issue of the previous convolution layer's inability to extract inter-channel features.
  - In summary, EEGNet achieves the goal of reducing parameters and computational complexity through the model architecture, while also mitigating overfitting caused by overly complex models, resulting in improved accuracy.

ii. Explain the activation function

- ReLU

$$ReLU(x) = \begin{cases} 0, & x \le 0 \\ x, & x > 0 \end{cases}$$

$$ReLU'(x) = \begin{cases} 0, & x \le 0 \\ 1, & x > 0 \end{cases}$$

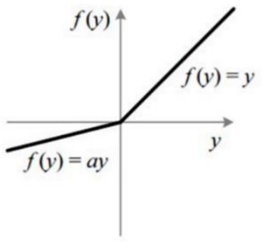


  - The derivative of the Sigmoid function in the saturation region is quite flat, approaching zero. This leads to difficulties in convergence, particularly evident in deep networks. In contrast, the majority of ReLU's derivatives are constant, which facilitates better model convergence
  - Due to the fact that the derivative of ReLU in the negative half region is zero, this is known as one-sided suppression. It causes many neurons in the neural network not to be activated, resulting in network sparsity. This has been proven to be helpful for model training(Occam's razor principle).

- Leaky ReLU

$$LeakyReLU(x) = \begin{cases} 0.1 * \mathrm{x}, & x \le 0 \\ x, & x > 0 \end{cases}$$

$$LeakyReLU'(x) = \begin{cases} 0.1, & x \le 0 \\ 1, & x > 0 \end{cases}$$

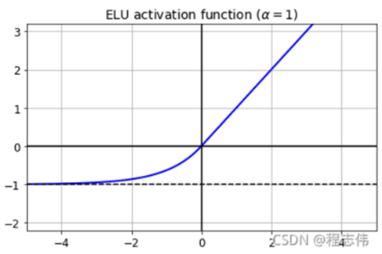


  - ReLU can lead to a problem called "neuron death" because its derivative is always 0 in the negative region, which may prevent neurons from updating in later stages of training.
  - In contrast, Leaky ReLU addresses this issue by introducing a small slope for negative input values, ensuring that the gradient does not become zero. This allows neurons to continue updating during training.

- ELU

$$ELU(x) = \begin{cases} \mathrm{a} * (\mathrm{e}^{x} - 1), x \le 0 \\ x, \quad x > 0 \end{cases}$$

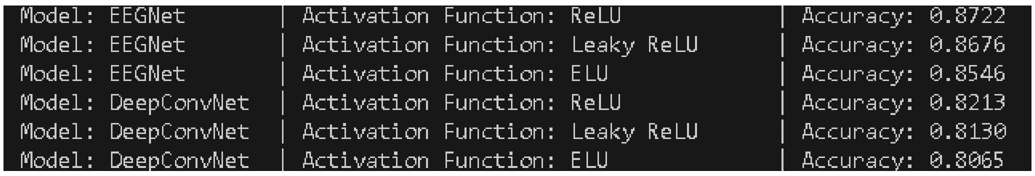
$$ELU(x) = \begin{cases} \mathrm{ELU(x)} + \mathrm{a}, x \le 0 \\ 1, \quad x > 0 \end{cases}$$



  - Compared to Leaky ReLU, the output is smoother.

III.. Experimental Results

i. The highest testing accuracy

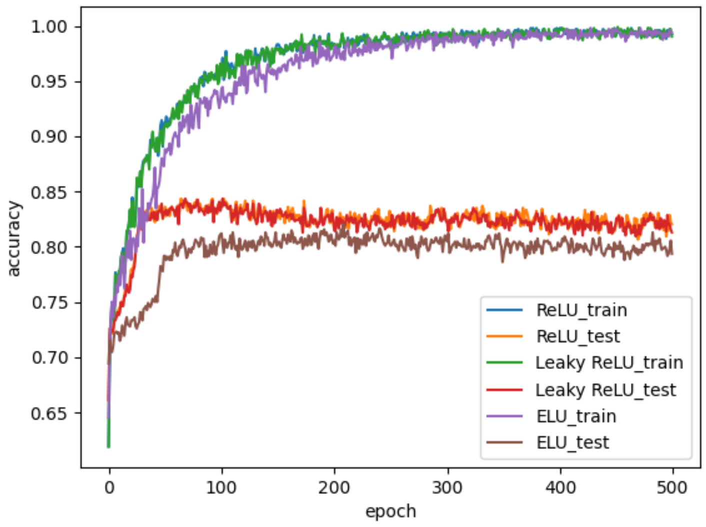


```
Model: EEGNet       | Activation Function: ReLU       | Accuracy: 0.8722
Model: EEGNet       | Activation Function: Leaky ReLU | Accuracy: 0.8676
Model: EEGNet       | Activation Function: ELU        | Accuracy: 0.8546
Model: DeepConvNet  | Activation Function: ReLU       | Accuracy: 0.8213
Model: DeepConvNet  | Activation Function: Leaky ReLU | Accuracy: 0.8130
Model: DeepConvNet  | Activation Function: ELU        | Accuracy: 0.8065
```

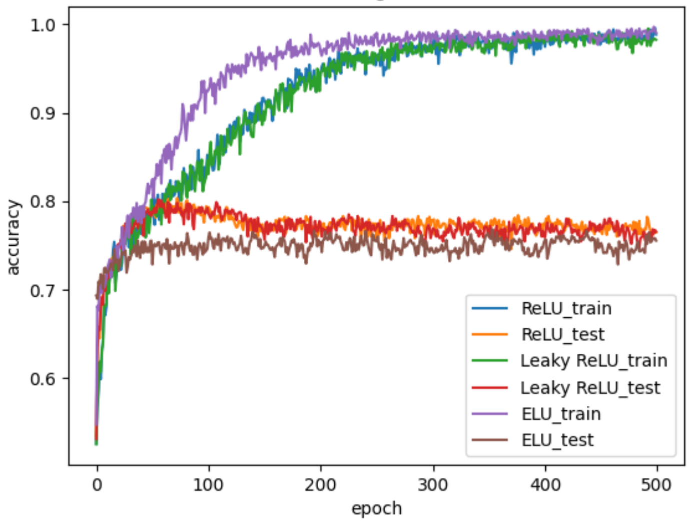
- Highest: EEGNet, ReLU, 0.8722

ii. Comparison figures

EEGNet

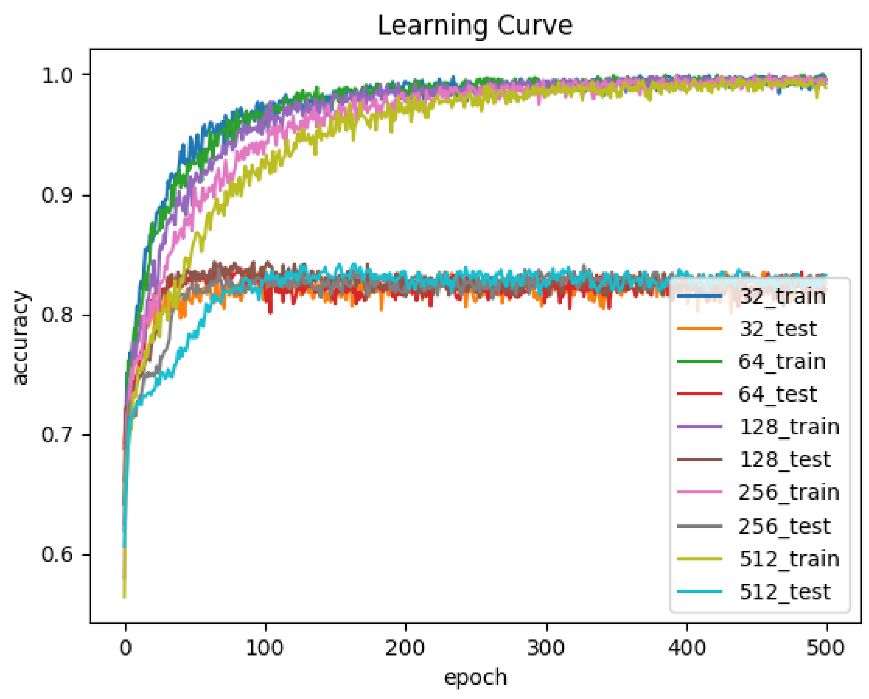


DeepConvNet



IV. Discussion

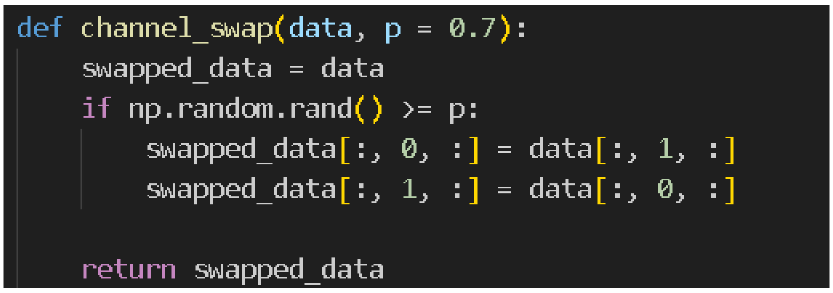
i. Different batch size



- In this experiment, I compared different batch sizes: 32, 64, 128, 256, and 512. From the plots, it can be observed that with larger batch sizes, the model generally exhibits slower convergence initially but demonstrates more stable convergence in the later stages.
- This behavior is because, with a larger batch size, the model processes more training data in one go before computing gradients, allowing it to better capture the overall characteristics of the data. However, it also leads to slower progress and increased memory consumption. Hence, selecting an appropriate batch size is crucial.
- In this task, I achieved the best results using a batch size of 128.

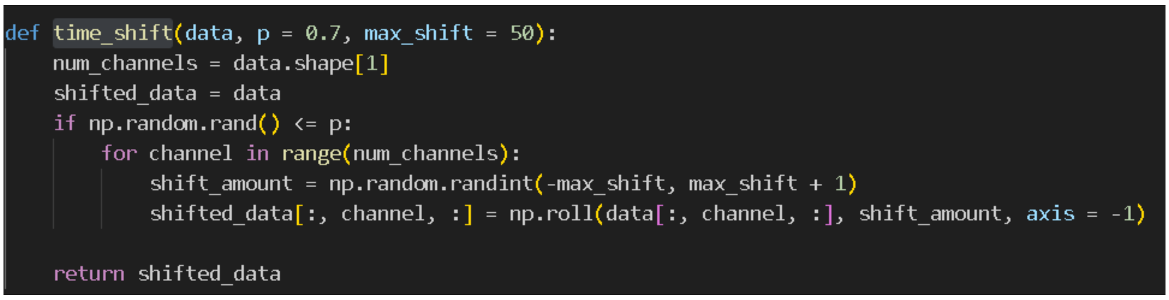
ii. With and without augmentation

- **In this lab, I implement five augmentation to augment the input data.**
- **channel_swap(): swap two channel input.**



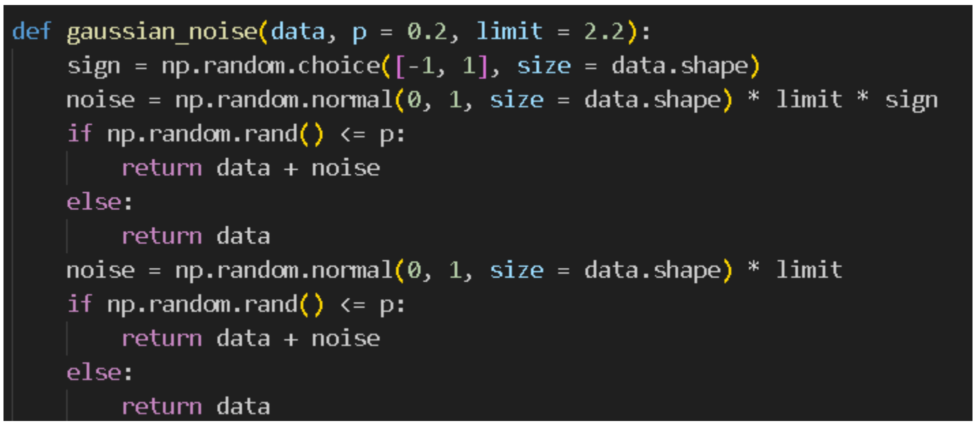
```python
def channel_swap(data, p = 0.7):
    swapped_data = data
    if np.random.rand() >= p:
        swapped_data[:, 0, :] = data[:, 1, :]
        swapped_data[:, 1, :] = data[:, 0, :]

    return swapped_data
```

- **time_shift(): perform data shifting along the time axis.**



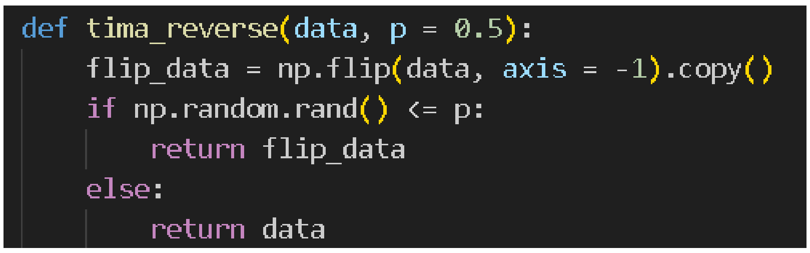
```python
def time_shift(data, p = 0.7, max_shift = 50):
    num_channels = data.shape[1]
    shifted_data = data
    if np.random.rand() <= p:
        for channel in range(num_channels):
            shift_amount = np.random.randint(-max_shift, max_shift + 1)
            shifted_data[:, channel, :] = np.roll(data[:, channel, :], shift_amount, axis = -1)

    return shifted_data
```

- **guassian_noise(): and guassian noise.**



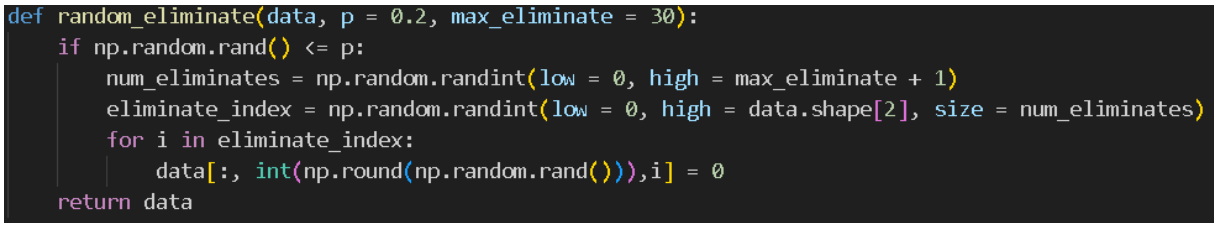
```python
def gaussian_noise(data, p = 0.2, limit = 2.2):
    sign = np.random.choice([-1, 1], size = data.shape)
    noise = np.random.normal(0, 1, size = data.shape) * limit * sign
    if np.random.rand() <= p:
        return data + noise
    else:
        return data
    noise = np.random.normal(0, 1, size = data.shape) * limit
    if np.random.rand() <= p:
        return data + noise
    else:
        return data
```

- **time_reverse(): reverse the data along the time axis.**



```python
def tima_reverse(data, p = 0.5):
    flip_data = np.flip(data, axis = -1).copy()
    if np.random.rand() <= p:
        return flip_data
    else:
        return data
```

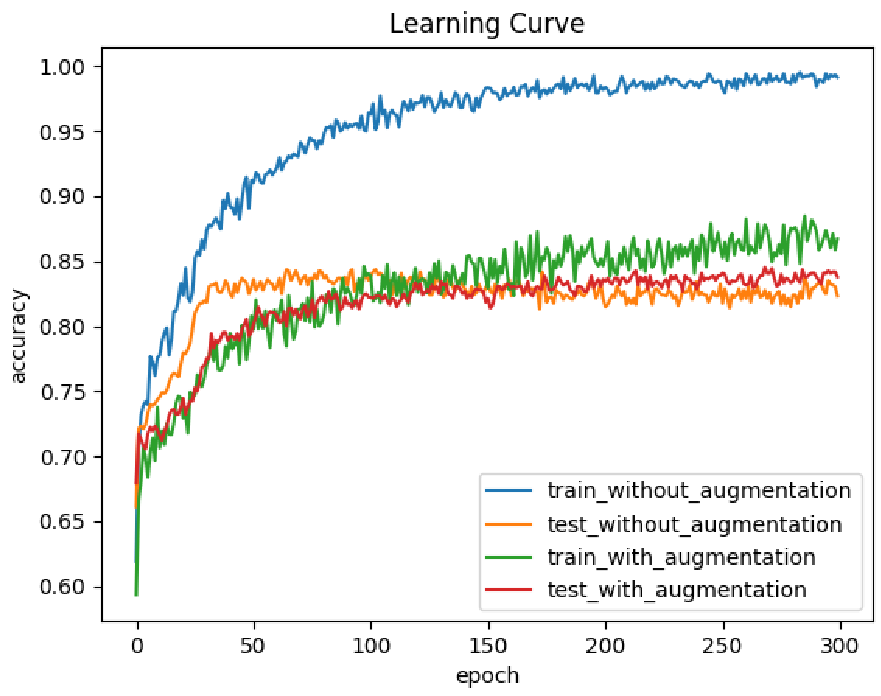
- **Random_eliminate():**



```
def random_eliminate(data, p = 0.2, max_eliminate = 30):
    if np.random.rand() <= p:
        num_eliminates = np.random.randint(low = 0, high = max_eliminate + 1)
        eliminate_index = np.random.randint(low = 0, high = data.shape[2], size = num_eliminates)
        for i in eliminate_index:
            data[:, int(np.round(np.random.rand())),i] = 0
    return data
```

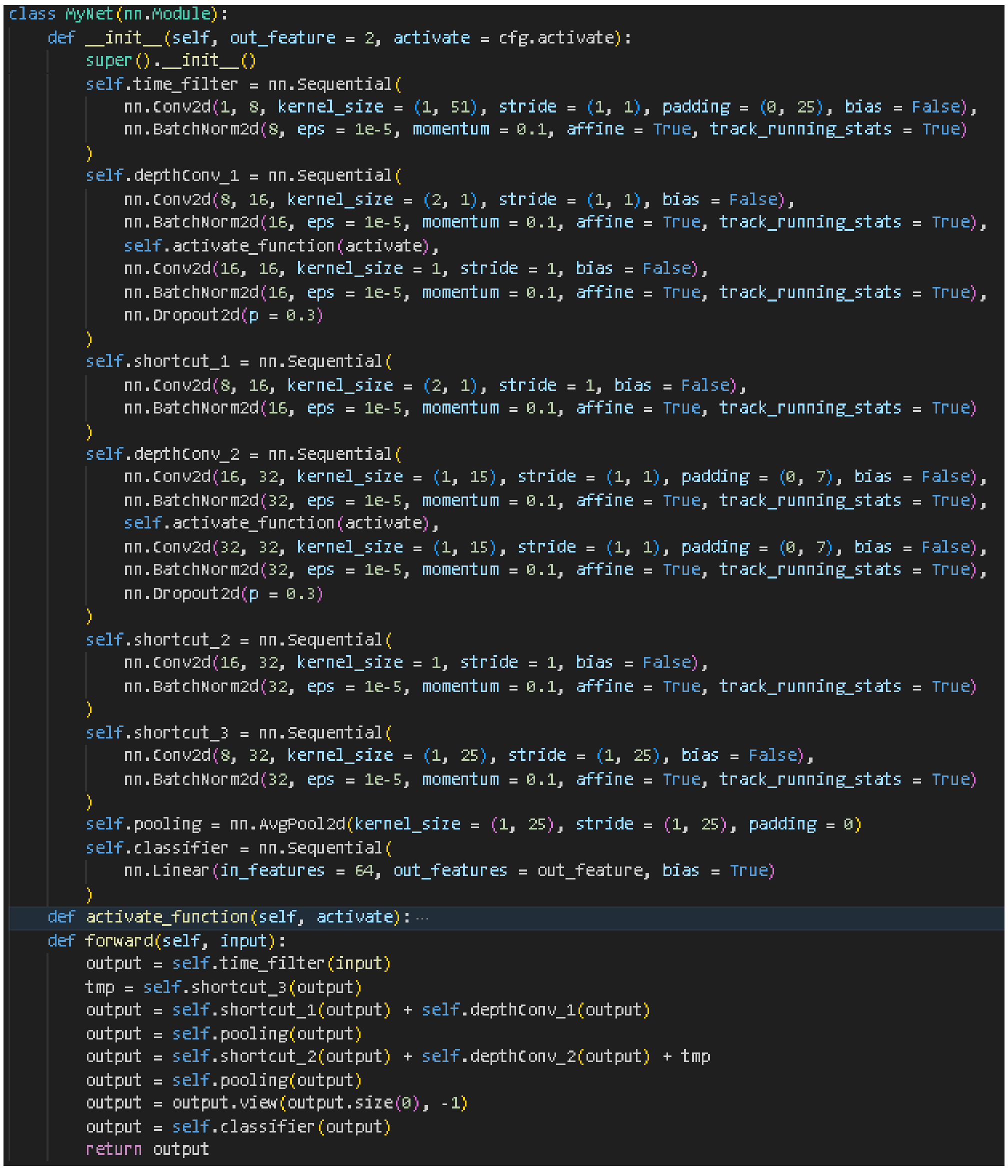
- **Due to severe overfitting observed during training, I addressed this issue by implementing data augmentation and introducing some noise to the data.**

- **This approach helps to increase the model's generalization capability and alleviate overfitting. The comparison between with and without augmentation in the following figure clearly shows that incorporating data augmentation significantly reduces the occurrence of overfitting.**



V. Extra

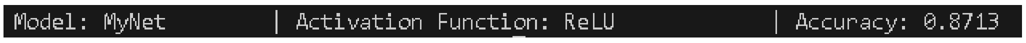
i. Implement any other classification model



```python
class MyNet(nn.Module):
    def __init__(self, out_feature = 2, activate = cfg.activate):
        super().__init__()
        self.time_filter = nn.Sequential(
            nn.Conv2d(1, 8, kernel_size = (1, 51), stride = (1, 1), padding = (0, 25), bias = False),
            nn.BatchNorm2d(8, eps = 1e-5, momentum = 0.1, affine = True, track_running_stats = True)
        )
        self.depthConv_1 = nn.Sequential(
            nn.Conv2d(8, 16, kernel_size = (2, 1), stride = (1, 1), bias = False),
            nn.BatchNorm2d(16, eps = 1e-5, momentum = 0.1, affine = True, track_running_stats = True),
            self.activate_function(activate),
            nn.Conv2d(16, 16, kernel_size = 1, stride = 1, bias = False),
            nn.BatchNorm2d(16, eps = 1e-5, momentum = 0.1, affine = True, track_running_stats = True),
            nn.Dropout2d(p = 0.3)
        )
        self.shortcut_1 = nn.Sequential(
            nn.Conv2d(8, 16, kernel_size = (2, 1), stride = 1, bias = False),
            nn.BatchNorm2d(16, eps = 1e-5, momentum = 0.1, affine = True, track_running_stats = True)
        )
        self.depthConv_2 = nn.Sequential(
            nn.Conv2d(16, 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),
            self.activate_function(activate),
            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),
            nn.Dropout2d(p = 0.3)
        )
        self.shortcut_2 = nn.Sequential(
            nn.Conv2d(16, 32, kernel_size = 1, stride = 1, bias = False),
            nn.BatchNorm2d(32, eps = 1e-5, momentum = 0.1, affine = True, track_running_stats = True)
        )
        self.shortcut_3 = nn.Sequential(
            nn.Conv2d(8, 32, kernel_size = (1, 25), stride = (1, 25), bias = False),
            nn.BatchNorm2d(32, eps = 1e-5, momentum = 0.1, affine = True, track_running_stats = True)
        )
        self.pooling = nn.AvgPool2d(kernel_size = (1, 25), stride = (1, 25), padding = 0)
        self.classifier = nn.Sequential(
            nn.Linear(in_features = 64, out_features = out_feature, bias = True)
        )
    def activate_function(self, activate):···
    def forward(self, input):
        output = self.time_filter(input)
        tmp = self.shortcut_3(output)
        output = self.shortcut_1(output) + self.depthConv_1(output)
        output = self.pooling(output)
        output = self.shortcut_2(output) + self.depthConv_2(output) + tmp
        output = self.pooling(output)
        output = output.view(output.size(0), -1)
        output = self.classifier(output)
        return output
```

ii. I employed residual learning techniques to construct MyNet, following the architecture of DenseNet. In comparison to EEGNet, I reduced the width of the convolutional layers while increasing the model's depth.

iii. As a result, I achieved an accuracy of 0.8713.



```
Model: MyNet          | Activation Function: ReLU          | Accuracy: 0.8713
```