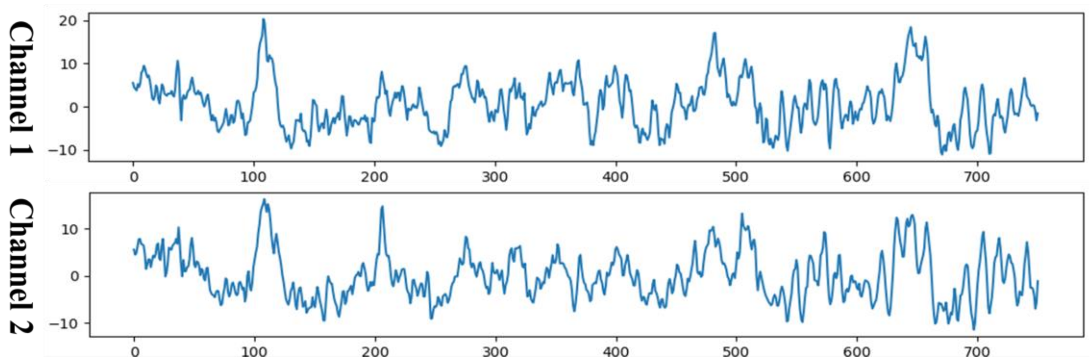
Summer 2022 Deep Learning
Report of Lab #2

Student name: 劉子齊 Jonathan
Student ID: 311605004

Part 1: Introduction

In this lab, my goal is to implement EEGNet and DeepConvNet with three kinds of activation functions, which are ReLU, Leaky ReLU, and ELU, and classify the provided EEG signals into two classes. Also, we should show the highest accuracy of two models with three kinds of activation functions and plot the accuracy curve to show the learning result.

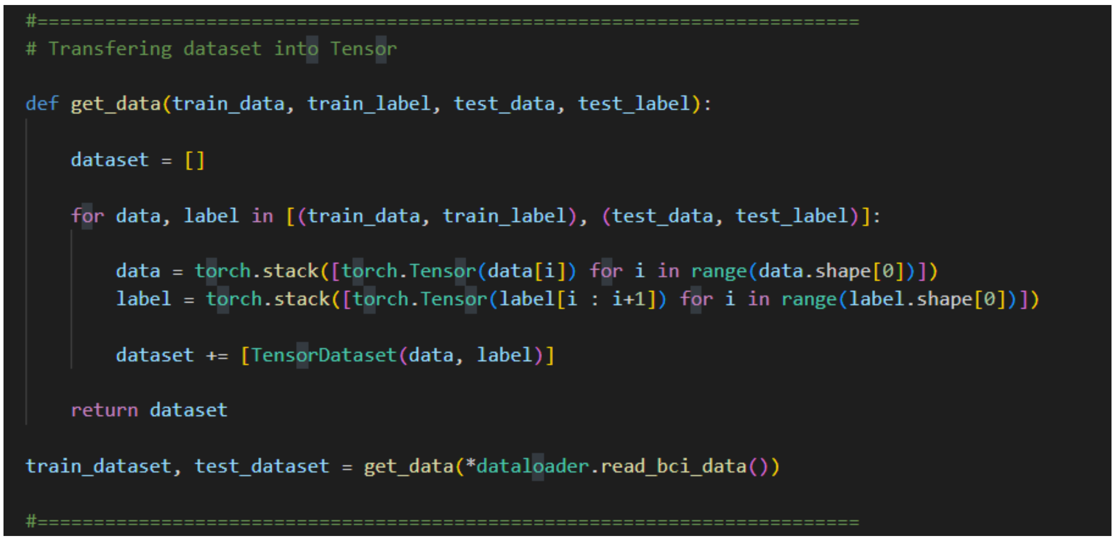
Before moving on to the next part, I would to talk about the dataset I made use of in this lab. The following is how the dataset looks. The dataset got 2 channels with 750 datapoints each. These datapoints can be classified into two labels, which are left-handed and right-handed.



Part 2: Experiment set up

Part 2-0: Data Setup

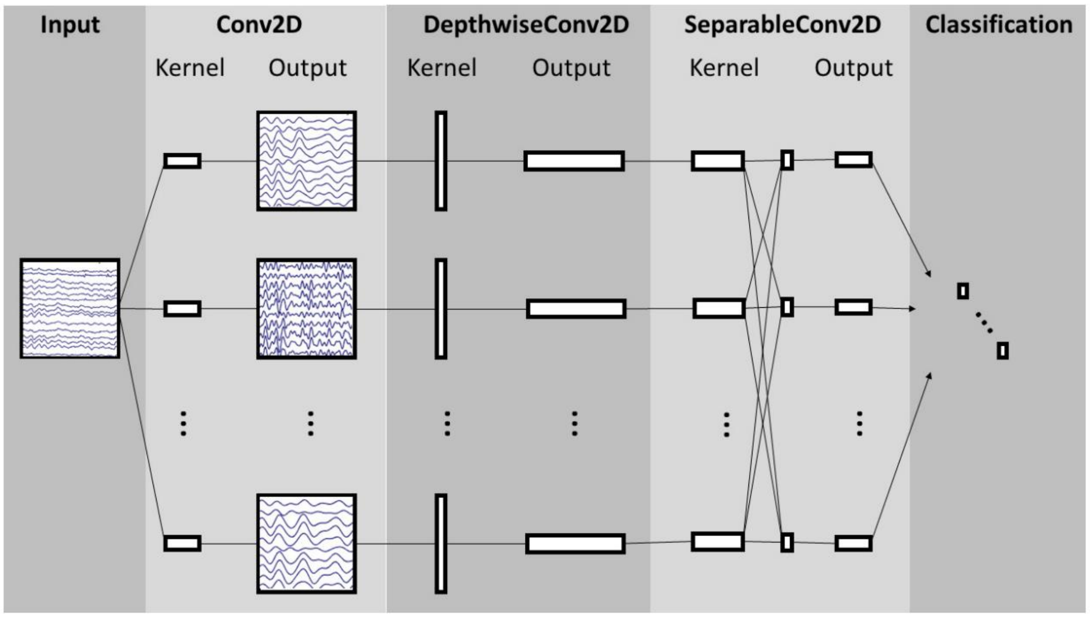
To apply DataLoader from “torch.utils.data”, I convert the original dataset obtained through the “dataloader.py”, which is in the form of numpy array, into the form of TensorDataset. Converting data to TensorDataset allows me to train the model with specific batch size easily. The difference between a NumPy array and a tensor is that the tensors are backed by the accelerator memory like GPU and they are immutable, unlike NumPy arrays. Thus, every tensor can be represented as a multidimensional array or vector, but not every vector can be represented as tensors.



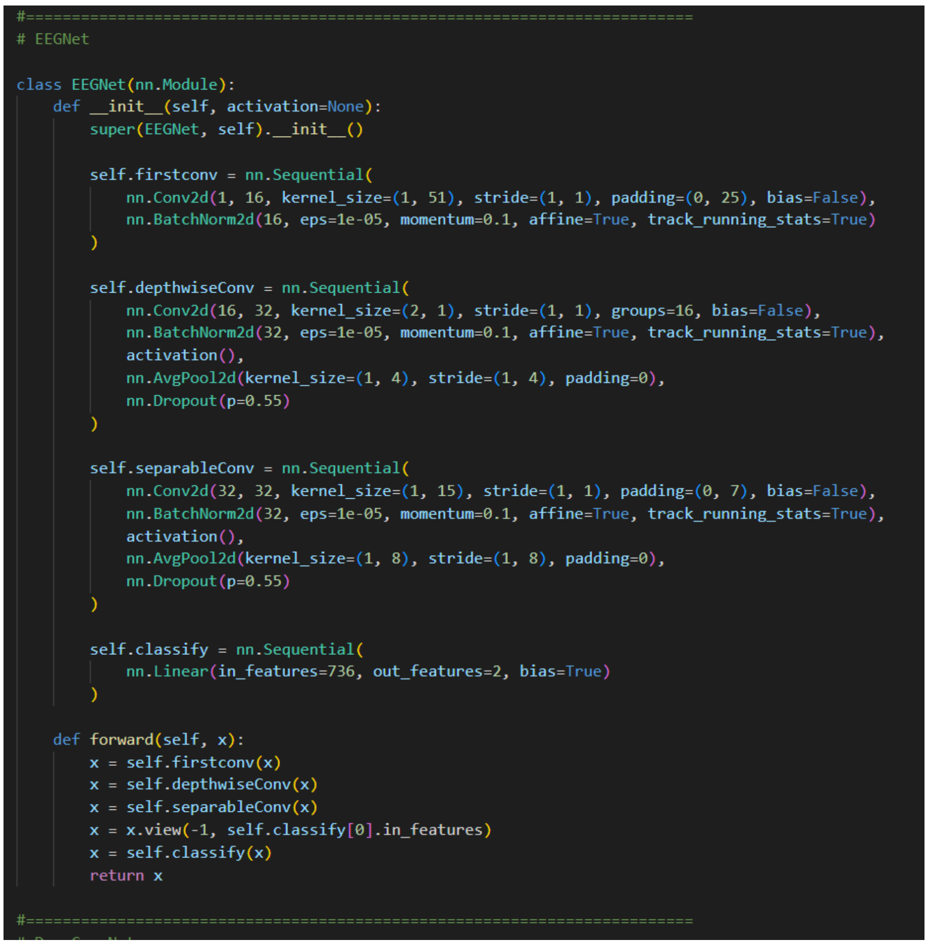
```python
#=========================================================================
# Transfering dataset into Tensor

def get_data(train_data, train_label, test_data, test_label):

    dataset = []

    for data, label in [(train_data, train_label), (test_data, test_label)]:

        data = torch.stack([torch.Tensor(data[i]) for i in range(data.shape[0])])
        label = torch.stack([torch.Tensor(label[i : i+1]) for i in range(label.shape[0])])

        dataset += [TensorDataset(data, label)]

    return dataset

train_dataset, test_dataset = get_data(*dataloader.read_bci_data())

#=========================================================================
```

Part 2-A: Details of the two models

Part 2-A-1: EEGNet



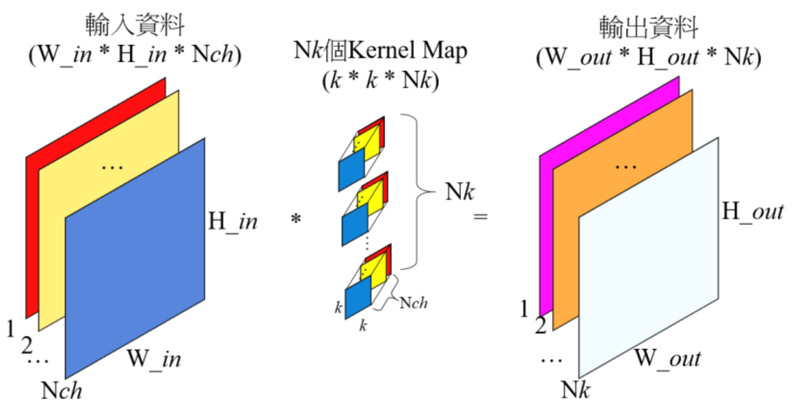
The figure above is the overall visualization of the EEGNet architecture from the reference “EEGNet: A Compact Convolutional Neural Network for EEG-based BrainComputer Interfaces.”, which is also where the EEGNet was born. The following figure is how I implemented the EEGNet architecture. In the first block, the model learns how to extract features from the EEG signals, and will output a feature map containing the EEG signals at different band-pass frequencies.



```
#==========================================================================
# EEGNet

class EEGNet(nn.Module):
    def __init__(self, activation=None):
        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-05, 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-05, momentum=0.1, affine=True, track_running_stats=True),
            activation(),
            nn.AvgPool2d(kernel_size=(1, 4), stride=(1, 4), padding=0),
            nn.Dropout(p=0.55)
        )

        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-05, momentum=0.1, affine=True, track_running_stats=True),
            activation(),
            nn.AvgPool2d(kernel_size=(1, 8), stride=(1, 8), padding=0),
            nn.Dropout(p=0.55)
        )

        self.classify = nn.Sequential(
            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(-1, self.classify[0].in_features)
        x = self.classify(x)
        return x

#==========================================================================
```

In the second block, which is the Depthwise Convolution, the main purpose of this block is to learn a spatial filter. This operation provides a direct way to learn spatial filters for each temporal filter through the not fully-connected property, thus enabling the efficient extraction of frequency-specific spatial filters. I also applied Batch Normalization here before the activation function to improve the learning rate of a neural network before applying the activation function nonlinearity. After that, I applied a 2D average pooling layer, to reduce the sampling rate of the signal. Last, I used the dropout technique and set the dropout probability to help prevent over-fitting when training on small sample sizes

In the third block, we use a Separable Convolution, which is actually a Depthwise Convolution with different size. There are two main goals here. First is to reduce the number of parameters to fit. Second is to decouple the relationship within and across feature maps through the property of Depthwise and Pointwise Convolutions, which can be seen in the figure below.



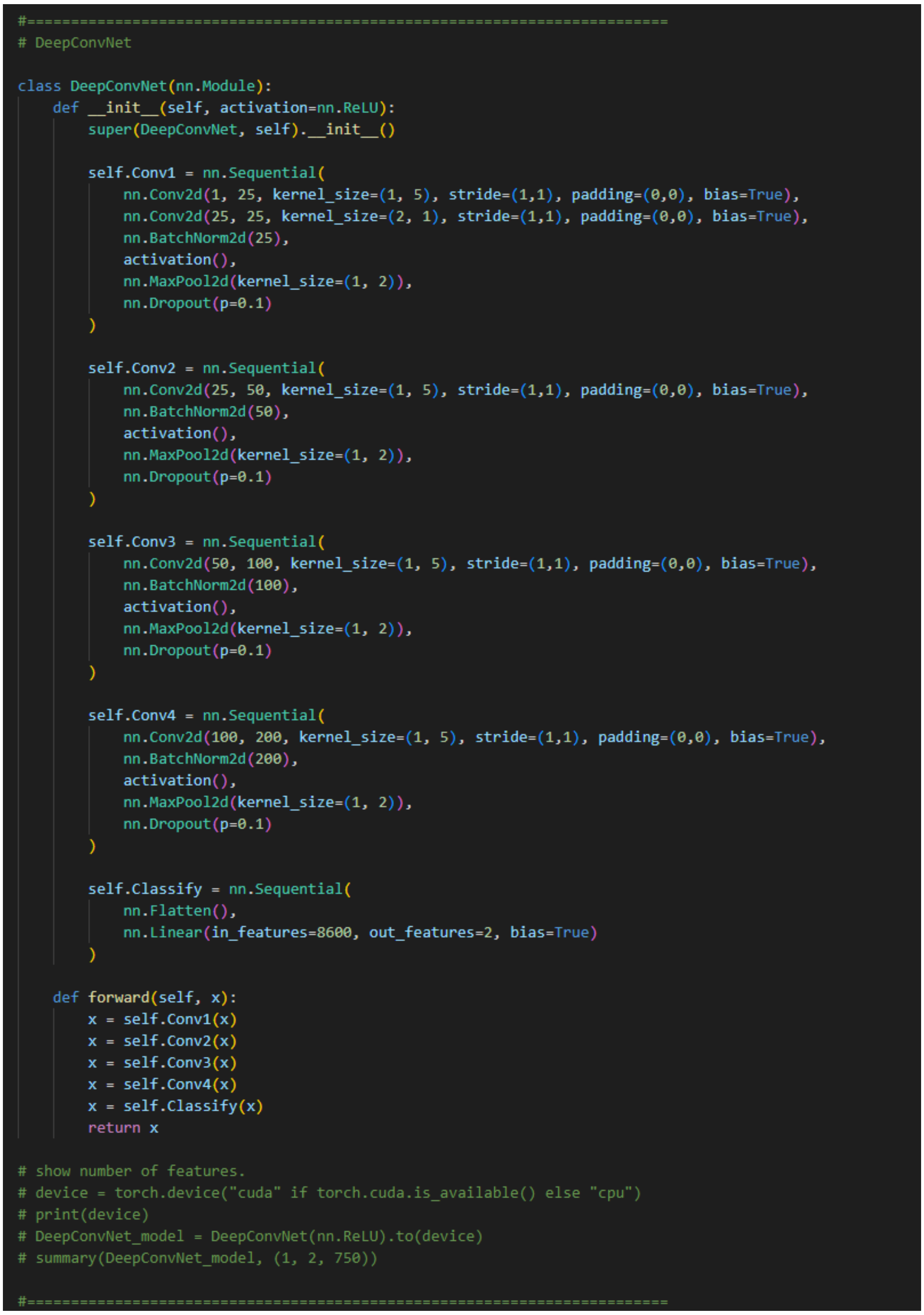
In the last block, which is the classification block, the features were send to the SoftMax classification layer. Then we will get a dictionary named “accuracy” with model names and accuracy results in it.

Part 2-A-2: DeepConvNet

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

The figure above is the of the architecture of the DeepConvNet from the reference “EEGNet: A Compact Convolutional Neural Network for EEG-based BrainComputer Interfaces.” In the paper, we can find some comparison between DeepConvNet and EGGNet. However, we can consider DeepConvNet as a normal

CNN. DeepConvNet is designed to be a general-purpose architecture, which is not restricted to specific feature types. The figure below is how I implemented the DeepConvNet according to the architecture provided by the paper above.



```python
#=========================================================================
# DeepConvNet

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

        self.Conv1 = nn.Sequential(
            nn.Conv2d(1, 25, kernel_size=(1, 5), stride=(1,1), padding=(0,0), bias=True),
            nn.Conv2d(25, 25, kernel_size=(2, 1), stride=(1,1), padding=(0,0), bias=True),
            nn.BatchNorm2d(25),
            activation(),
            nn.MaxPool2d(kernel_size=(1, 2)),
            nn.Dropout(p=0.1)
        )

        self.Conv2 = nn.Sequential(
            nn.Conv2d(25, 50, kernel_size=(1, 5), stride=(1,1), padding=(0,0), bias=True),
            nn.BatchNorm2d(50),
            activation(),
            nn.MaxPool2d(kernel_size=(1, 2)),
            nn.Dropout(p=0.1)
        )

        self.Conv3 = nn.Sequential(
            nn.Conv2d(50, 100, kernel_size=(1, 5), stride=(1,1), padding=(0,0), bias=True),
            nn.BatchNorm2d(100),
            activation(),
            nn.MaxPool2d(kernel_size=(1, 2)),
            nn.Dropout(p=0.1)
        )

        self.Conv4 = nn.Sequential(
            nn.Conv2d(100, 200, kernel_size=(1, 5), stride=(1,1), padding=(0,0), bias=True),
            nn.BatchNorm2d(200),
            activation(),
            nn.MaxPool2d(kernel_size=(1, 2)),
            nn.Dropout(p=0.1)
        )

        self.Classify = nn.Sequential(
            nn.Flatten(),
            nn.Linear(in_features=8600, out_features=2, bias=True)
        )

    def forward(self, x):
        x = self.Conv1(x)
        x = self.Conv2(x)
        x = self.Conv3(x)
        x = self.Conv4(x)
        x = self.Classify(x)
        return x

# show number of features.
# device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# print(device)
# DeepConvNet_model = DeepConvNet(nn.ReLU).to(device)
# summary(DeepConvNet_model, (1, 2, 750))

#=========================================================================
```

Part 2-B. Explain the activation function

The following is the mathematical representation of the three activation functions:

- **Rectified Linear Units (ReLU)**

$$R(z) = \begin{Bmatrix} z & z > 0 \\ 0 & z <= 0 \end{Bmatrix} \qquad R'(z) = \begin{Bmatrix} 1 & z > 0 \\ 0 & z < 0 \end{Bmatrix}$$

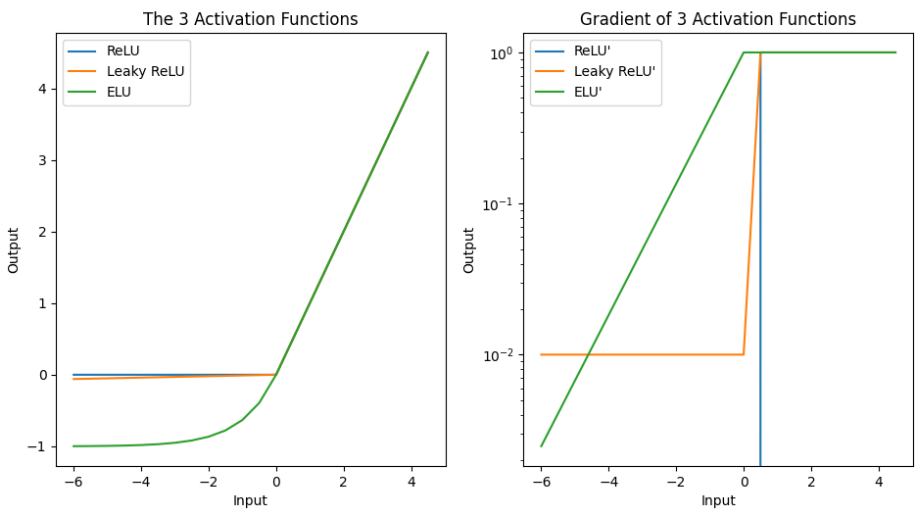
- **Leaky Rectified Linear Units (Leaky ReLU)**

$$R(z) = \begin{Bmatrix} z & z > 0 \\ \alpha z & z <= 0 \end{Bmatrix} \qquad R'(z) = \begin{Bmatrix} 1 & z > 0 \\ \alpha & z < 0 \end{Bmatrix}$$

- **Exponential Linear Unit (ELU)**

$$R(z) = \begin{Bmatrix} z & z > 0 \\ \alpha.(e^z - 1) & z <= 0 \end{Bmatrix} \qquad R'(z) = \begin{Bmatrix} 1 & z > 0 \\ \alpha.e^z & z < 0 \end{Bmatrix}$$

The figure below which I plotted the three activations in, we can easily compare them through it. We can tell that the three activation functions actually act nearly the same from the positive input value. Through the subplot on the right, which is the gradient of the functions, we can tell that ReLU gets 0 gradient whenever the input is negative, and this can probably cause serious vanishing problem to the model. Hence, Leaky ReLU and ELU were designed to avoid such problem. Both of them got values but not zero when negative input, and from the figure we can tell that ELU git greater computation volume than Leaky ReLU.



Part 3: Experimental results

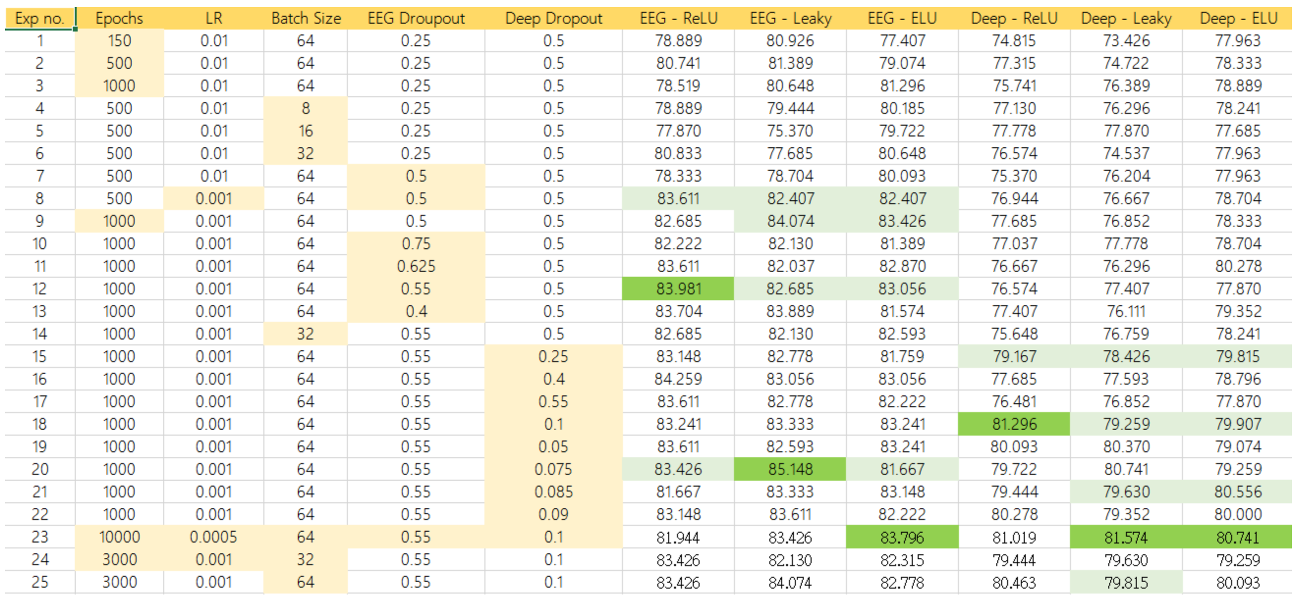
Part 3-A: Highest testing accuracy

For Lab 2, I totally conducted 25 experiments, the following chart is the parameters and the results of all the experiments.

The blocks that are filled with “light orange”, like this 8, are the parameters I modified in that experiment.

The blocks filled with “light green”, like this 83.611, are the result that had improved in comparison to the current experiment to the previous ones.

The most important, the blocks filled with “eye-catching green”, like this 85.148, are the blocks with the best accuracy result of the model and its activation function, which is noted on top of the column.

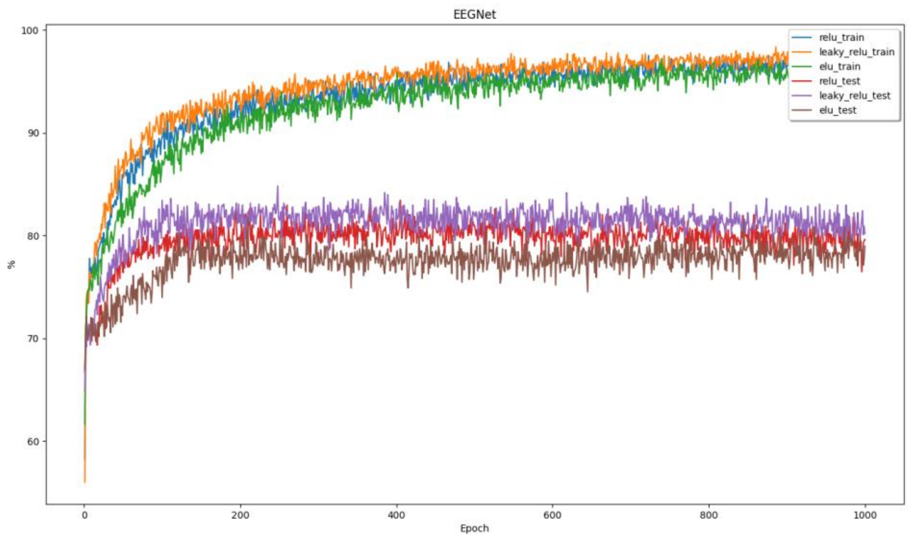


Exp no.	Epochs	LR	Batch Size	EEG Droupout	Deep Dropout	EEG - ReLU	EEG - Leaky	EEG - ELU	Deep - ReLU	Deep - Leaky	Deep - ELU
1	150	0.01	64	0.25	0.5	78.889	80.926	77.407	74.815	73.426	77.963
2	500	0.01	64	0.25	0.5	80.741	81.389	79.074	77.315	74.722	78.333
3	1000	0.01	64	0.25	0.5	78.519	80.648	81.296	75.741	76.389	78.889
4	500	0.01	8	0.25	0.5	78.889	79.444	80.185	77.130	76.296	78.241
5	500	0.01	16	0.25	0.5	77.870	75.370	79.722	77.778	77.870	77.685
6	500	0.01	32	0.25	0.5	80.833	77.685	80.648	76.574	74.537	77.963
7	500	0.01	64	0.5	0.5	78.333	78.704	80.093	75.370	76.204	77.963
8	500	0.001	64	0.5	0.5	83.611	82.407	82.407	76.944	76.667	78.704
9	1000	0.001	64	0.5	0.5	82.685	84.074	83.426	77.685	76.852	78.333
10	1000	0.001	64	0.75	0.5	82.222	82.130	81.389	77.037	77.778	78.704
11	1000	0.001	64	0.625	0.5	83.611	82.037	82.870	76.667	76.296	80.278
12	1000	0.001	64	0.55	0.5	83.981	82.685	83.056	76.574	77.407	77.870
13	1000	0.001	64	0.4	0.5	83.704	83.889	81.574	77.407	76.111	79.352
14	1000	0.001	32	0.55	0.5	82.685	82.130	82.593	75.648	76.759	78.241
15	1000	0.001	64	0.55	0.25	83.148	82.778	81.759	79.167	78.426	79.815
16	1000	0.001	64	0.55	0.4	84.259	83.056	83.056	77.685	77.593	78.796
17	1000	0.001	64	0.55	0.55	83.611	82.778	82.222	76.481	76.852	77.870
18	1000	0.001	64	0.55	0.1	83.241	83.333	83.241	81.296	79.259	79.907
19	1000	0.001	64	0.55	0.05	83.611	82.593	83.241	80.093	80.370	79.074
20	1000	0.001	64	0.55	0.075	83.426	85.148	81.667	79.722	80.741	79.259
21	1000	0.001	64	0.55	0.085	81.667	83.333	83.148	79.444	79.630	80.556
22	1000	0.001	64	0.55	0.09	83.148	83.611	82.222	80.278	79.352	80.000
23	10000	0.0005	64	0.55	0.1	81.944	83.426	83.796	81.019	81.574	80.741
24	3000	0.001	32	0.55	0.1	83.426	82.130	82.315	79.444	79.630	79.259
25	3000	0.001	64	0.55	0.1	83.426	84.074	82.778	80.463	79.815	80.093

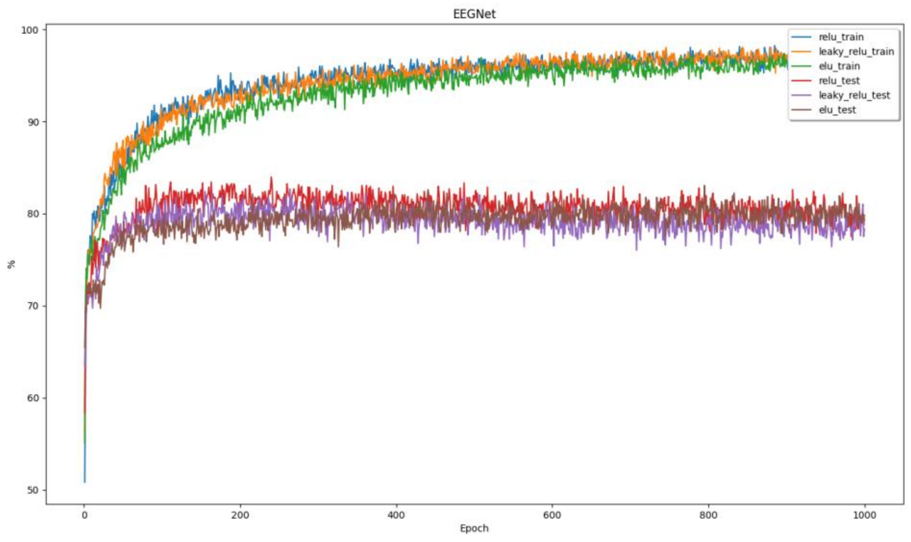
Part 3-B: Comparison figures

In this part, I'll show the comparison figures of the experiments which occurred the best accuracy results I mentioned above.

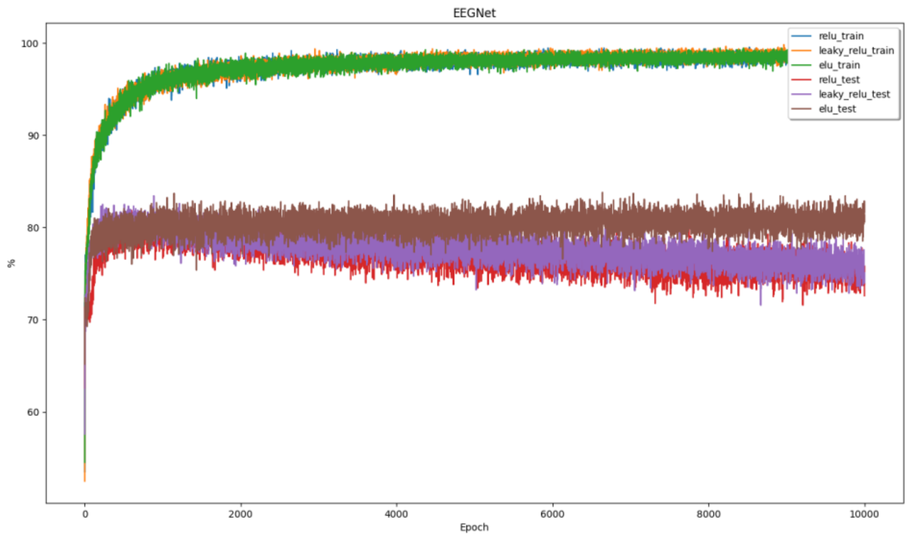
Exp_No.	Epochs	LR	BatchSize	EEG_Droupout	Deep_Dropout	EEG_Leaky
20	1000	0.001	64	0.55	0.075	85.148 %

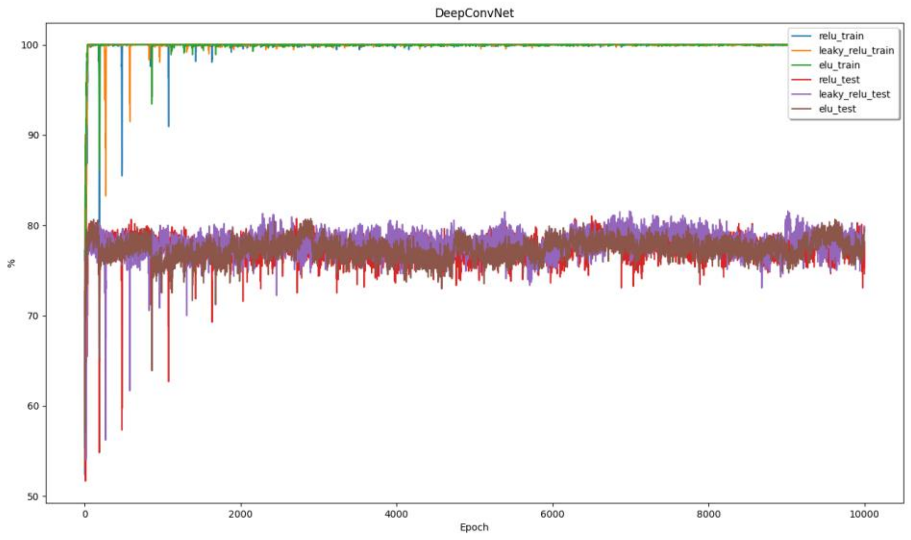


Exp_No.	Epochs	LR	BatchSize	EEG_Droupout	Deep_Dropout	EEG_ReLU
12	1000	0.001	64	0.55	0.5	83.981 %

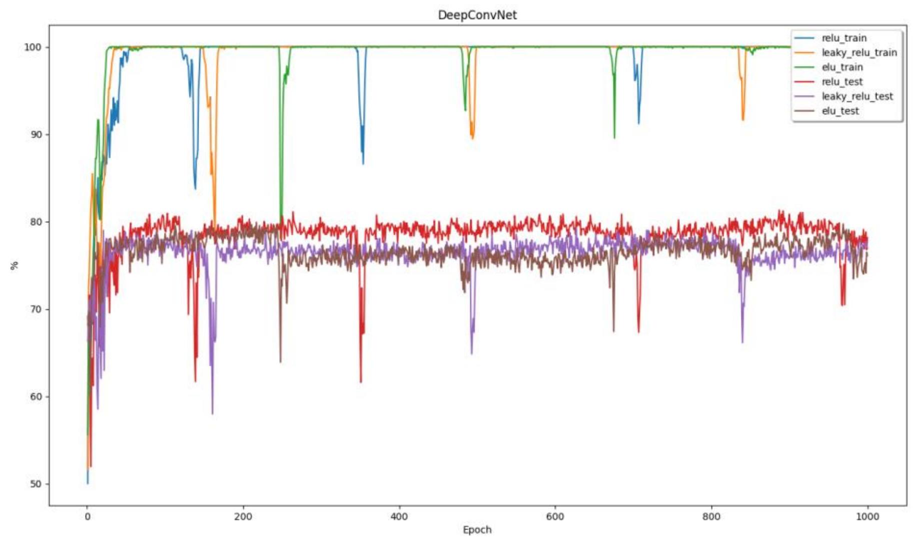


Exp_No.	Epochs	LR	BatchSize	EEG_Droupout	Deep_Dropout	Model Name
23	10000	0.0005	64	0.55	0.1	83.796 % (EEG_ELU)
23	10000	0.0005	64	0.55	0.1	81.574 % (Deep_Leaky)
23	10000	0.0005	64	0.55	0.1	80.741 % (Deep_ELU)





Exp_No.	Epochs	LR	BatchSize	EEG_Droupout	Deep_Dropout	Deep_ReLU
18	1000	0.001	64	0.55	0.1	81.296 %



Part 4: Discussion

Part 4-A: Learning Rate

Through the experiments above, we can easily find the epochs required for the neural networks to reach the loss goal increases as we lower the learning rate, which proves that higher learning rate requires less training time in the same background conditions.

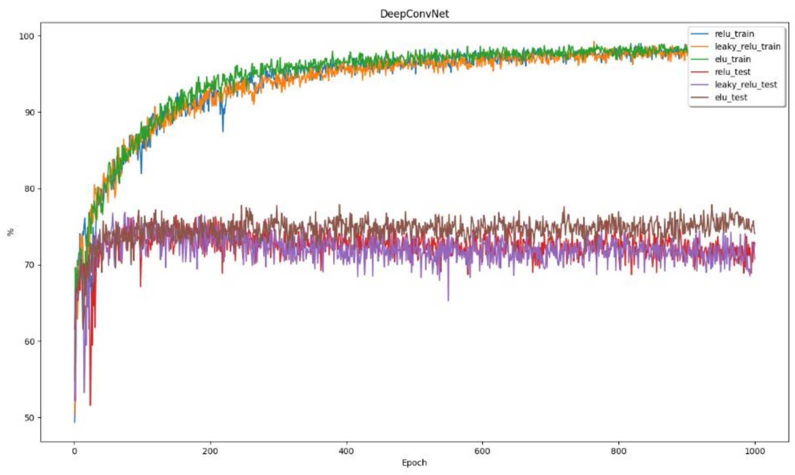
However, if we lower the learning rate to low-low value, such as the learning rate which is 0.0005 in experiment no. 23, also pulling the total epoch number to 10000, we might still get some better results, but might also overfit some of the models.

Part 4-B: Dropouts

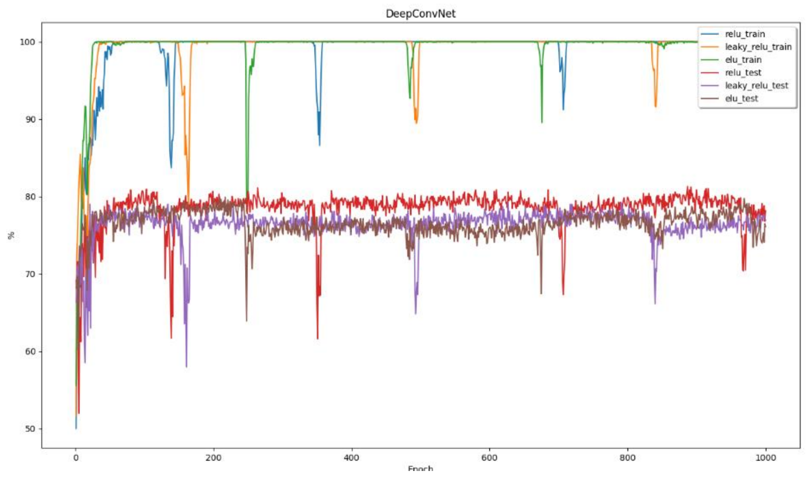
First, I would like to talk about what Dropout layers do. Dropout layers actually simply ignore neurons with the probability of “1-p” either or “p” randomly during the training phase of certain set of neurons which is chosen at random. In the ignoring process, Dropout will not ignore those neurons in part of the forward or backward pass.

The following are the accuracy curves of the experiments with different dropout probability of EEGNet (all the other parameters are the same). From the figures below, we find that when we lower the probability of dropout, the accuracy curve seems to be less stable. Since individual nodes are either dropped out of the net with probability 1−p or kept with probability p, if the probability is too low or too high, it is more likely that the model loses control.

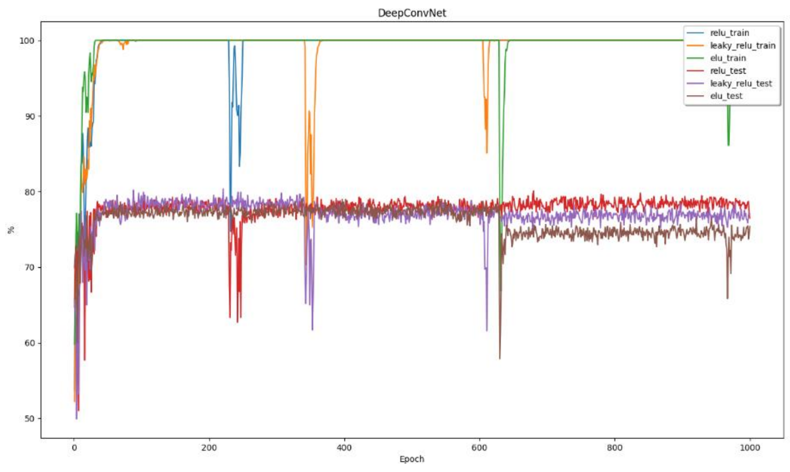
- P = 0.55



- P = 0.1



- P = 0.05


DeepConvNet
relu_train
leaky_relu_train
elu_train
relu_test
leaky_relu_test
elu_test
Epoch