I. Introduction (20%)

In Lab2, we implemented two models with three activation functions for classifying EEG signals. The EEG signals got from BCI competition dataset were pre-processed to shape of (N x 1 x C x T), e.g., (1080 x 1 x 2 x 750), in advance. Then we built EEGNet and DeepConvNet models in which we took ReLU, Leaky ReLU, and ELU as three kinds of different activation functions. The output of two models are either class 0 or class 1, therefore, the number of output features at the last fully connected layer is 2. In order to compare the performance of two models with three kinds of activation functions, we used accuracy results as our evaluated function. The architecture of Lab 2 is shown in Fig. 1.

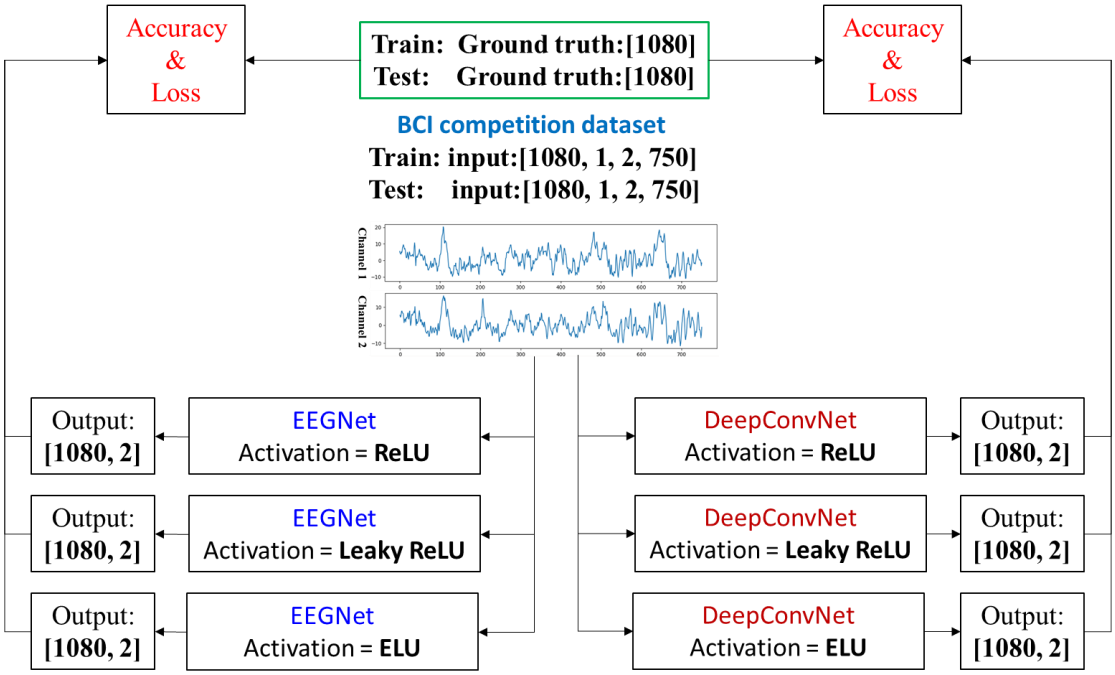


Fig. 1 The architecture of Lab 2

II. Experiment setup (30%)

A. The detail of two models

Before introducing the models, we may discuss convolution first. Because we can design the number of filters, the shape of one filter in convolution stage, and how to convolute data and filters, we need to input these parameters in Conv2D function.

```
self.conv1 = nn.Conv2d(1, 16, kernel_size=(1, 51), stride=(1, 1), padding=(0, 25), bias=False)
```

For example, in the above code, 1 means input channel, 16 means output channels or filter numbers, kernel_size = (1, 51) means the shape of filter is (1, 51), bias means it constructed a bias term in convolution stage if bias

were set True, and stride, and padding are parameters about how to convolute, e.g., convolutional steps and the process in edge positions. Following picture is the formula to calculate the output shape of convolution operations.

Shape:

- Input: $(N, C_{in}, H_{in}, W_{in})$
- Output: $(N, C_{out}, H_{out}, W_{out})$ where

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

Variables:

- **weight** (*Tensor*) – the learnable weights of the module of shape (out_channels, in_channels, kernel_size[0], kernel_size[1])
- **bias** (*Tensor*) – the learnable bias of the module of shape (out_channels)

Examples:

```
>>> # With square kernels and equal stride
>>> m = nn.Conv2d(16, 33, 3, stride=2)
>>> # non-square kernels and unequal stride and with padding
>>> m = nn.Conv2d(16, 33, (3, 5), stride=(2, 1), padding=(4, 2))
>>> # non-square kernels and unequal stride and with padding and dilation
>>> m = nn.Conv2d(16, 33, (3, 5), stride=(2, 1), padding=(4, 2), dilation=(3, 1))
>>> input = torch.randn(20, 16, 50, 100)
>>> output = m(input)
```


In addition, “Depthwise Separable Convolution” is another basis we need to understand in advance. Fig. 2 shows one example of the general convolution with 256x(3x5x5)x(8x8) multiplications.

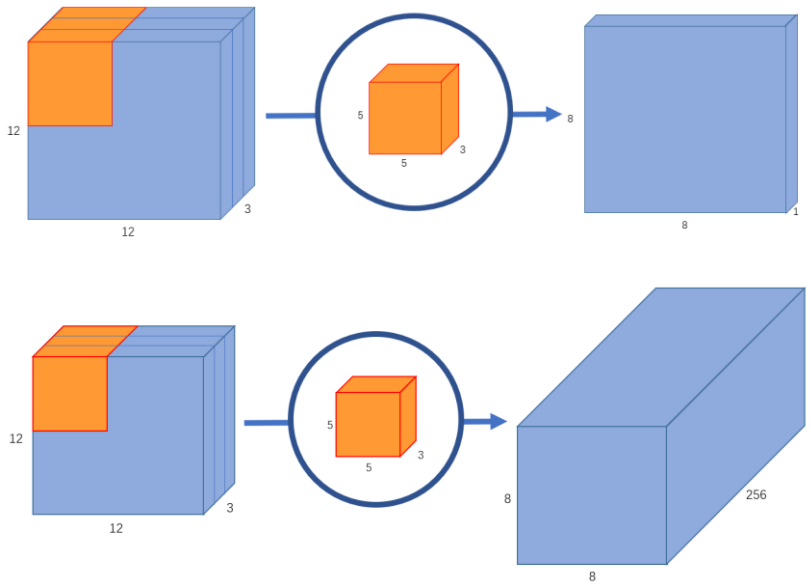


Fig. 2 A general convolution example

Fig. 3 shows one example of the depthwise separable convolution with (3x5x5)x(8x8) + 256x(1x1x3)x(8x8) multiplications. Obviously, depthwise separable convolution is more fast than the general convolution.

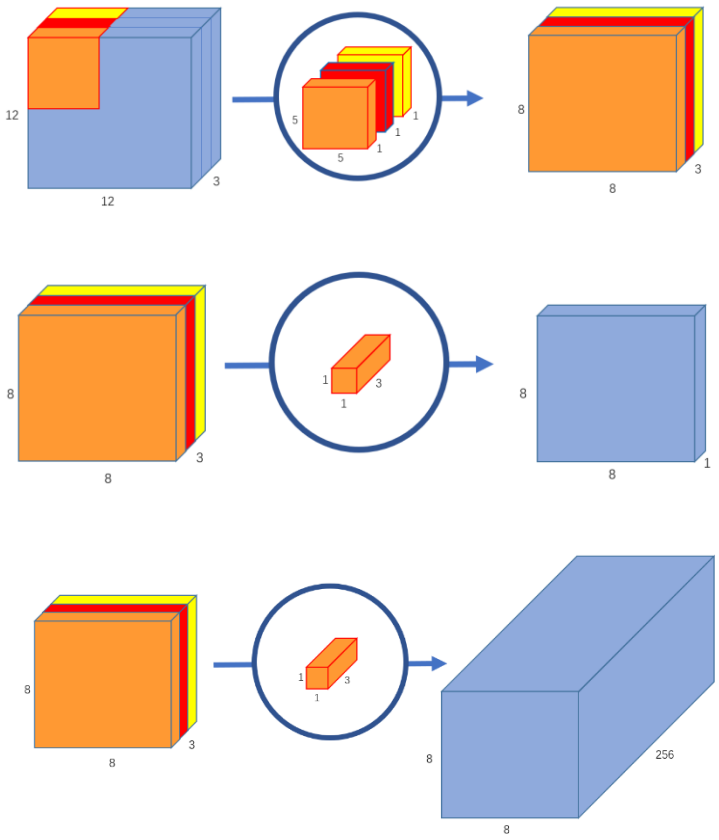


Fig. 3 Depthwise separable convolution

■ EEGNet

```
EEGNet(
  (firstconv): Sequential(
    (0): Conv2d(1, 16, kernel_size=(1, 51), stride=(1, 1), padding=(0, 25), bias=False)
    (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (depthwiseConv): Sequential(
    (0): Conv2d(16, 32, kernel_size=(2, 1), stride=(1, 1), groups=16, bias=False)
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ELU(alpha=1.0)
    (3): AvgPool2d(kernel_size=(1, 4), stride=(1, 4), padding=0)
    (4): Dropout(p=0.25)
  )
  (separableConv): Sequential(
    (0): Conv2d(32, 32, kernel_size=(1, 15), stride=(1, 1), padding=(0, 7), bias=False)
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ELU(alpha=1.0)
    (3): AvgPool2d(kernel_size=(1, 8), stride=(1, 8), padding=0)
    (4): Dropout(p=0.25)
  )
  (classify): Sequential(
    (0): Linear(in_features=736, out_features=2, bias=True)
  )
)
```

Fig. 4 EEGNet architecture

Fig. 4 shows the EEGNet implemented architecture proposed by TA. There

are four packaged layers in EEGNet. The ***firstconv*** layer includes one Conv2D layer and BatchNorm2D layer. The ***deptwiseConv*** layer and ***separableConv*** layer are depthwise separable convolution implemetation we introduced before. The last layer is ***classifier*** implemented by fully connected neural network without softmax. Due to the use of "*CrossEntropyLoss*", we cannot use softmax after fully connected layer.

In Lab2, we replaced the activation function of EEGNET by *ReLU*, *Leaky ReLU*, and *ELU*, and used "*Adam*" as our optimizer. Finally, we used "*summary*(net, [input.shape])" in pytorch to show the output shape of EEGNet (Fig. 5) in Lab2.

```
----------------------------------------------------------------
      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
          ELU-5          [-1, 32, 1, 750]               0
     AvgPool2d-6          [-1, 32, 1, 187]               0
     Dropout-7            [-1, 32, 1, 187]               0
        Conv2d-8          [-1, 32, 1, 187]          15,360
   BatchNorm2d-9          [-1, 32, 1, 187]              64
         ELU-10          [-1, 32, 1, 187]               0
    AvgPool2d-11           [-1, 32, 1, 23]               0
    Dropout-12             [-1, 32, 1, 23]               0
       Linear-13                 [-1, 2]           1,474
================================================================
Total params: 17,874
Trainable params: 17,874
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 1.16
Params size (MB): 0.07
Estimated Total Size (MB): 1.23
----------------------------------------------------------------
```

Fig. 5 Output shape of EEGNet in Lab2

- 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

Fig. 6 DeepConvNet architecture

Fig. 6 shows the DeepConvNet architecture proposed by TA. There are five packaged layers in DeepConvNet. The ***first layer*** is a sequence of {two Conv2D layers, BatchNorm2D, Activation, MaxPool2D, Dropout}. ***The following three layers*** are sequences of {Conv2D layers, BatchNorm2D, Activation, MaxPool2D, Dropout}. The last layer is ***classifier*** implemented by fully connected neural network without softmax. Due to the use of "*CrossEntropyLoss*", we cannot use softmax after fully connected layer.

In Lab2, we replaced the activation function of DeepConvNet by *ReLU*, *Leaky ReLU*, and *ELU*, and used "*Adam*" as our optimizer. Finally, we used "*summary*(net, [input.shape])" in pytorch to show the output shape of DeepConvNet (Fig. 7) in Lab2.

```
----------------------------------------------------------------
      Layer (type)               Output Shape         Param #
================================================================
          Conv2d-1          [-1, 25, 2, 750]             125
          Conv2d-2          [-1, 25, 1, 750]           1,250
     BatchNorm2d-3          [-1, 25, 1, 750]              50
            ELU-4          [-1, 25, 1, 750]               0
       MaxPool2d-5          [-1, 25, 1, 375]               0
         Dropout-6          [-1, 25, 1, 375]               0
          Conv2d-7          [-1, 50, 1, 375]           6,250
     BatchNorm2d-8          [-1, 50, 1, 375]             100
            ELU-9          [-1, 50, 1, 375]               0
      MaxPool2d-10          [-1, 50, 1, 187]               0
        Dropout-11          [-1, 50, 1, 187]               0
         Conv2d-12          [-1, 100, 1, 187]          25,000
    BatchNorm2d-13          [-1, 100, 1, 187]             200
           ELU-14          [-1, 100, 1, 187]               0
      MaxPool2d-15          [-1, 100, 1, 93]               0
        Dropout-16          [-1, 100, 1, 93]               0
         Conv2d-17          [-1, 200, 1, 93]         100,000
    BatchNorm2d-18          [-1, 200, 1, 93]             400
           ELU-19          [-1, 200, 1, 93]               0
      MaxPool2d-20          [-1, 200, 1, 46]               0
        Dropout-21          [-1, 200, 1, 46]               0
         Linear-22                [-1, 2]          18,402
================================================================
Total params: 151,777
Trainable params: 151,777
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 2.57
Params size (MB): 0.58
Estimated Total Size (MB): 3.15
----------------------------------------------------------------
```

Fig. 7 Output shape of DeepConvNet in Lab2

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

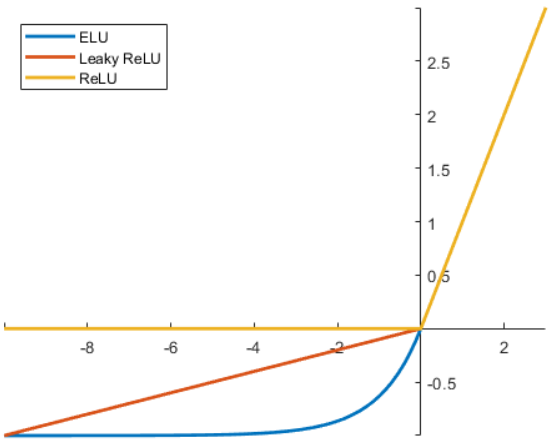


Fig. 8 Output of three activation functions

The formulas of three activation functions are list below and their outputs are shown in Fig. 8.

$$\text{ReLU}: \mathrm{f(x)} = \max(0, \mathrm{x})$$

$$\text{Leaky ReLU}: \mathrm{f(x)} = \begin{cases} x, & if \ \ x \geq 0 \\ negslope \times x, & otherwise \end{cases}$$

$$, \text{whrer} \ \ negslope = 0.01(\text{default})$$

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

In positive area, i.e., x>0, these three activation functions avoided gradient vanishing. However, the output of ReLU is not zero-centered and there exists dead ReLU problem, i.e., some nodes in network may un-active in whole training stage. Leaky ReLU is one method for solving dead ReLU problem due to taking the small factor, or call negative slope, multiplying negative values as output. Similarly, ELU can avoid dead ReLU problem and furthermore the output of ELU is zero-centered.

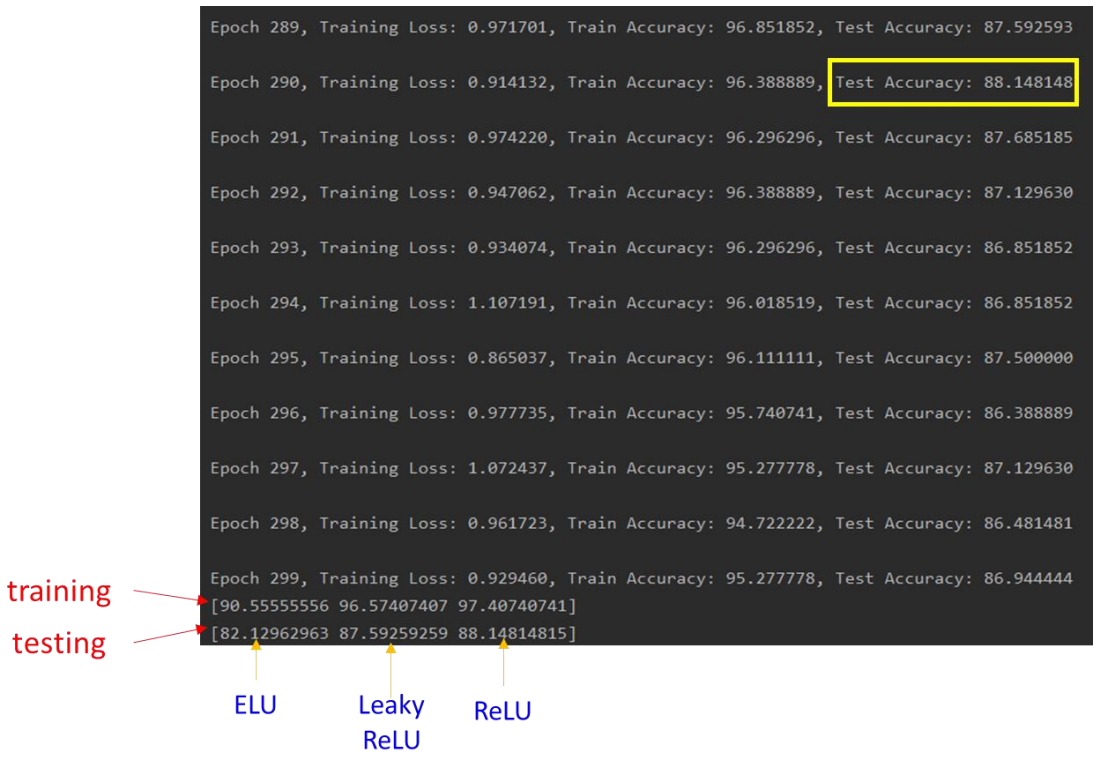
III. Experimental results (30 %)

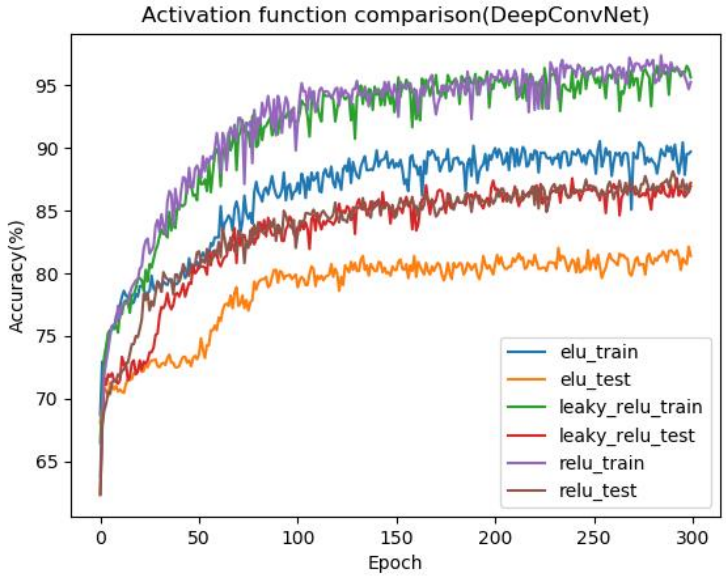
A. The highest testing accuracy

- Screenshot with two models

(1) EEGNet

Batch size = 256 / Learning rate = 1e-3 / weight_decay = 0.01 / Epoch = 300





(2) DeepConvNet

Learning rate = 1e-3 / weight_decay = 0.1 / Epoch = 300

Batch size = 128 @ ELU

Batch size = 256 @ Leaky ReLU

Batch size = 512 @ ReLU

```
Epoch 237, Training Loss: 1.563105, Train Accuracy: 90.740741, Test Accuracy: 82.777778
Epoch 238, Training Loss: 1.472260, Train Accuracy: 91.851852, Test Accuracy: 83.518519
Epoch 239, Training Loss: 1.562791, Train Accuracy: 94.074074, Test Accuracy: 85.000000
Epoch 240, Training Loss: 1.485088, Train Accuracy: 92.777778, Test Accuracy: 83.703704
Epoch 241, Training Loss: 1.466216, Train Accuracy: 90.000000, Test Accuracy: 81.666667
```

⋮

```
Epoch 293, Training Loss: 1.674608, Train Accuracy: 82.407407, Test Accuracy: 77.777778
Epoch 294, Training Loss: 1.581998, Train Accuracy: 88.796296, Test Accuracy: 83.055556
Epoch 295, Training Loss: 1.661035, Train Accuracy: 86.944444, Test Accuracy: 80.370370
Epoch 296, Training Loss: 1.673483, Train Accuracy: 91.759259, Test Accuracy: 82.777778
Epoch 297, Training Loss: 1.517070, Train Accuracy: 91.388889, Test Accuracy: 83.888889
Epoch 298, Training Loss: 1.441400, Train Accuracy: 91.481481, Test Accuracy: 84.074074
Epoch 299, Training Loss: 1.461475, Train Accuracy: 90.185185, Test Accuracy: 83.611111
[98.61111111 93.24074074 94.81481481]
[83.05555556 84.16666667 85.        ]
```

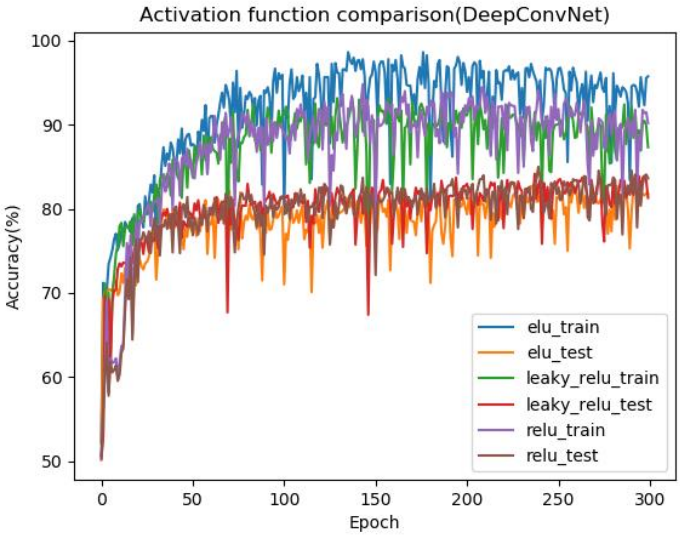
training

testing

ELU

Leaky ReLU

ReLU



- Anything you want to present

  To reach higher test accuracy, I tri some methods list below.

  Unfortunately, there just regularization and dropout worked in Lab 2.

  (1) **Data standardize**: (data – mean) / standard deviation

  (2) **Initial Weight**: xavier_normal, kaiming_normal

  (3) **Dynamic learning rate**: StepLR(optimizer, step_size=100, gamma=0.5)

  (4) **Regularization**: L2

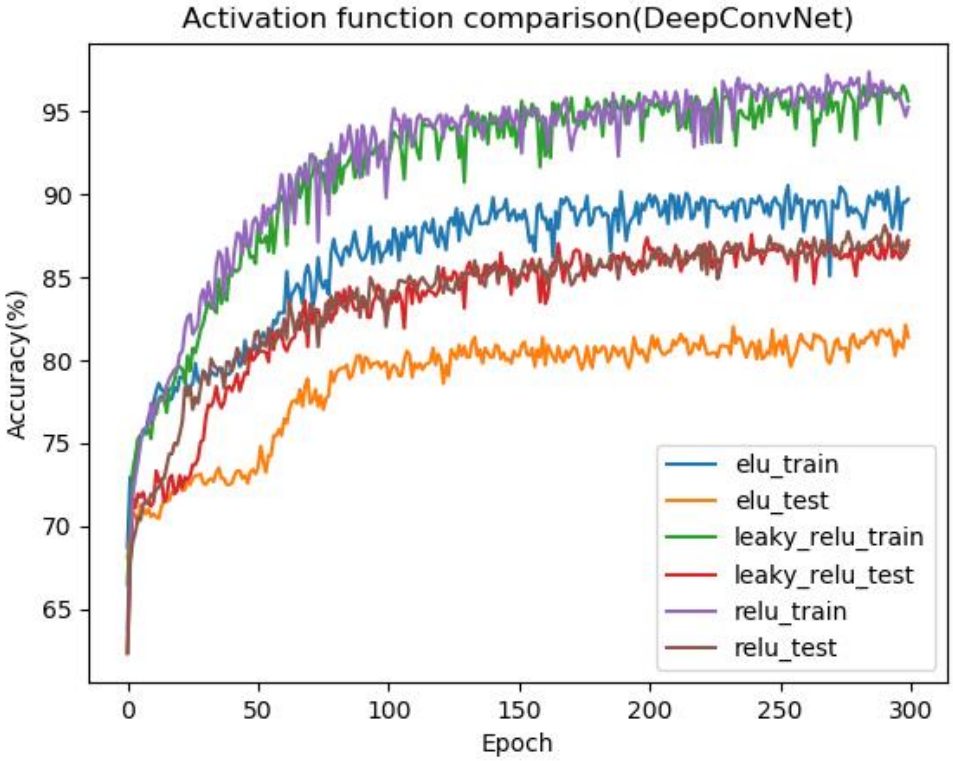
  (5) **Dropout**: 0.5

B. Comparison figures

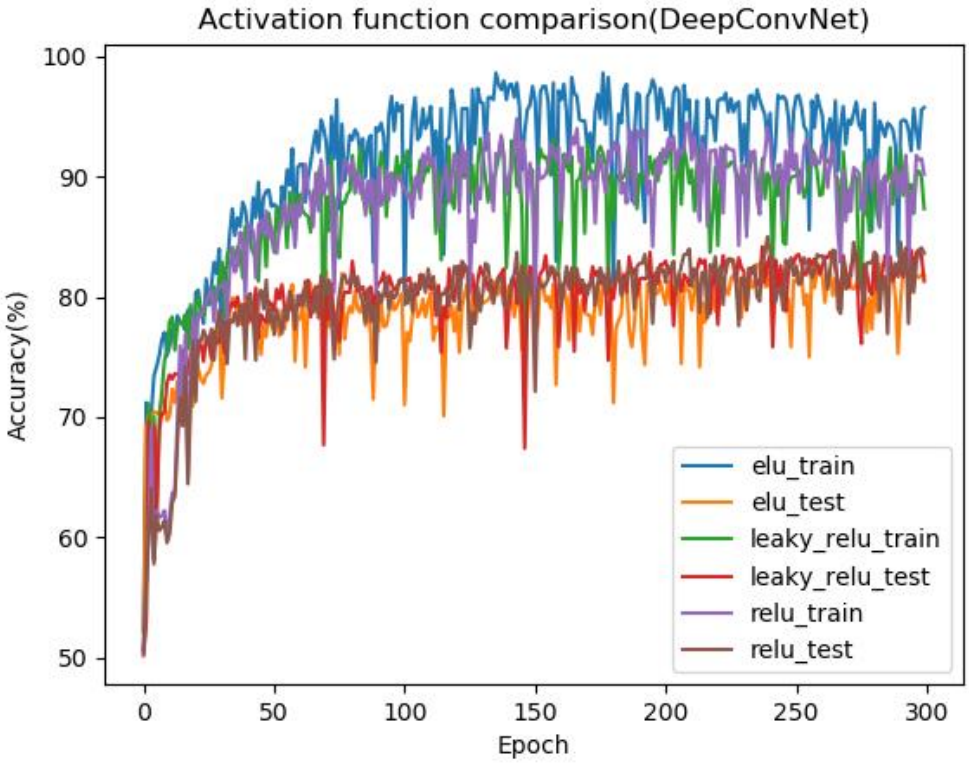
Table 1. The highest accuracy of two architectures with three kinds of activation functions

	ReLU	Leaky ReLU	ELU
EEGNet	**88.15** %	**87.59** %	82.13 %
DeepConvNet	**85.00** %	84.17 %	83.01 %

- EEGNet



■ DeepConvNet



IV. Discussion (20%)

A. Anything you want to share

It's very difficult to tune the hyper parameters by manual. Therefore, I wrote 4 layer for-loop code of activation, learning rate, batch size, and weight decay to get the parameters with the best result.

Activation: [0, 1, 2]

learning rate: [1e-5, 1e-4, 1e-3, 1e-2]

batch size: [32, 64, 128, 256, 512]

weight decay: [1e-4, 1e-3, 1e-2, 1e-1]

We reduce the range of parameters by accuracy after the first training in EEGNet.

Activation: [1, 2]

learning rate: [1e-3, 1e-2]

batch size: [64, 128, 256, 512]

weight decay: [1e-4, 1e-3, 1e-2, 1e-1]

[81.91287878787878, 81.72348484848484, 79.26136363636364, 84.08203125, 83.69140625, 82.6171875, 82.8125, 84.1796875, 84.47265625, 83.59375, 84.9609375, 84.66796875, 83.49609375, 84.375, 84.47265625, 81.72348484848484, 73.57954545454545, 70.83333333333333, 79.8828125, 79.8828125, 71.97265625, 83.203125, 81.34765625, 71.97265625, 83.49609375, 83.203125, 77.44140625, 83.984375, 83.203125, 81.0546875, 84.0909090909091, 83.33333333333333, 79.16666666666667, 83.59375, 84.27734375, 81.73828125, 83.984375, 84.08203125, 84.1796875, 83.49609375, 84.765625, 84.5703125, 83.984375, 84.08203125, 84.66796875, 78.4090909090909, 72.91666666666667, 71.11742424242425, 79.6875, 75.48828125, 72.16796875, 80.95703125, 81.0546875, 73.33984375, 82.6171875, 84.5703125, 79.1015625, 84.1796875, 84.1796875, 80.76171875]

Leak_ReLU : Lr = 1e-3, batch = 256, w_decay = 0.01

ReLU : Lr = 1e-3, batch = 256, w_decay = 0.01

Then, we get the optimal parameters from the accuracy results.

Optimal parameters:

Leak_ReLU:

learning rate = 1e-3, batch size = 256, weight decay = 0.01

ReLU :

learning rate = 1e-3, batch size = 256, weight decay = 0.01

Similarly, we reduce the range of parameters by accuracy after the first training in DeepConvNet.

Activation: [0, 1, 2]

learning rate: [1e-3, 1e-2]

batch size: [64, 128, 256, 512]

weight decay: [1e-5, 1e-4, 1e-3, 1e-2, 1e-1]

[78.97727272727273, 79.92424242424242, 80.11363636363636, 74.90530303030303, 80.37109375, 81.15234375, 80.859375, 79.39453125, 81.4453125, 80.76171875, 80.95703125, 82.32421875, 80.6640625, 79.58984375, 81.4453125, 82.03125, 79.19921875, 77.5390625, 78.80859375, 81.73828125; 80.01893939393939, 78.03030303030303, 74.14772727272727, 68.84469696969697, 80.078125, 80.95703125, 80.2734375, 68.359375, 80.76171875, 76.5625, 80.6640625, 79.58984375, 75.0, 75.390625, 79.1015625, 79.6875, 73.14453125, 74.90234375, 76.5625, 80.6640625; 80.3030303030303, 79.64015151515152, 79.64015151515152, 76.32575757575758, 79.98046875, 80.95703125, 81.34765625, 78.61328125, 80.2734375, 80.6640625, 82.6171875, 82.51953125, 79.00390625, 76.171875, 81.25, 82.8125, 78.7109375, 75.48828125, 80.46875, 80.95703125; 80.3030303030303, 78.03030303030303, 73.86363636363636, 50.85227272727273, 79.6875, 79.58984375, 78.41796875, 64.35546875, 77.5390625, 79.296875, 79.1015625, 75.0, 55.46875, 75.5859375, 79.19921875, 77.83203125, 72.4609375, 65.8203125, 79.00390625, 78.3203125; 80.01893939393939, 79.26136363636364, 79.54545454545455, 76.32575757575758, 80.2734375, 80.76171875, 81.8359375, 78.02734375, 81.15234375, 81.15234375, 81.8359375, 81.8359375, 78.41796875, 79.296875, 81.15234375, 82.421875, 77.9296875, 77.5390625, 74.31640625, 83.0078125; 79.73484848484848, 79.64015151515152, 75.9469696969697, 51.609848484848484, 78.22265625, 80.6640625, 77.9296875, 50.29296875, 76.5625, 78.61328125, 79.6875, 55.17578125, 71.09375, 78.515625, 78.125, 77.24609375, 55.56640625, 71.875, 77.05078125, 78.02734375]

ReLU : Lr = 1e-3, batch = 512, w_decay = 0.1

ELU : Lr = 1e-3, batch = 128, w_decay = 0.1

Leaky ReLU : Lr = 1e-3, batch = 256, w_decay = 0.1

Then, we get the optimal parameters from the accuracy results.

Optimal parameters:

ELU:

learning rate = 1e-3, batch size = 128, weight decay = 0.1

Leak_ReLU:

learning rate = 1e-3, batch size = 256, weight decay = 0.1

ReLU :

learning rate = 1e-3, batch size = 512, weight decay = 0.1