2019 Deep Learning and Practice
Lab 2 : EEG classification

0756110 李東霖

April 10, 2019

1 Introduction

In this experiment, I implemented EEGNet and DeepConvNet with three activation functions including ReLU, Leaky ReLU and ELU to classify signals of brain into two classes.

The objectives of the experiment are as follows:

- To show the highest accuracy of two models with three activation functions.
- To visualize the accuracy trend.

1.1 Dataset

This dataset has two channel of 750 data points each. The label of two classes in dataset are left-handed and right-handed.

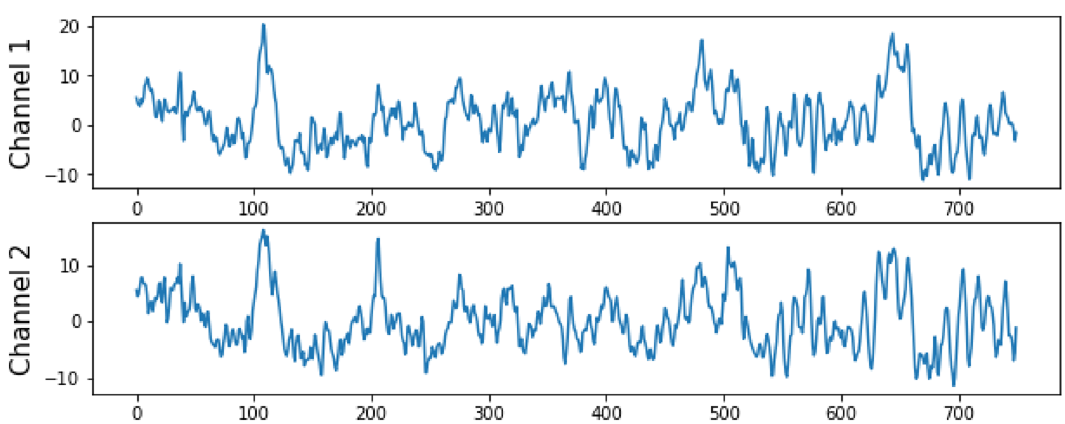

Figure 1: 2 bipolar EEG signals

2 Experiment setup

2.1 Convert Data to Tensor

To leverage `DataLoader` from `pytorch`, I convert `numpy` array to `TensorDataset` and make `TensorDataset` as `DataLoader`, So I can easily train my model with specific batch size.

```python
import dataloader
import torch
from torch.utils.data import TensorDataset, DataLoader

def gen_dataset(train_x, train_y, test_x, test_y):
    datasets = []
    for x, y in [(train_x, train_y), (test_x, test_y)]:
        x = torch.stack(
            [torch.Tensor(x[i]) for i in range(x.shape[0])]
        )
        y = torch.stack(
            [torch.Tensor(y[i:i+1]) for i in range(y.shape[0])]
        )
        datasets += [TensorDataset(x, y)]

    return datasets

train_dataset, test_dataset = gen_dataset(*dataloader.read_bci_data())
train_loader = DataLoader(train_dataset, batch_size=batch_size)
test_loader = DataLoader(test_dataset, len(test_dataset))
```

2.2 EEGNet

Reference from EEGNet: A Compact Convolutional Neural Network for EEG-based Brain-Computer Interfaces.

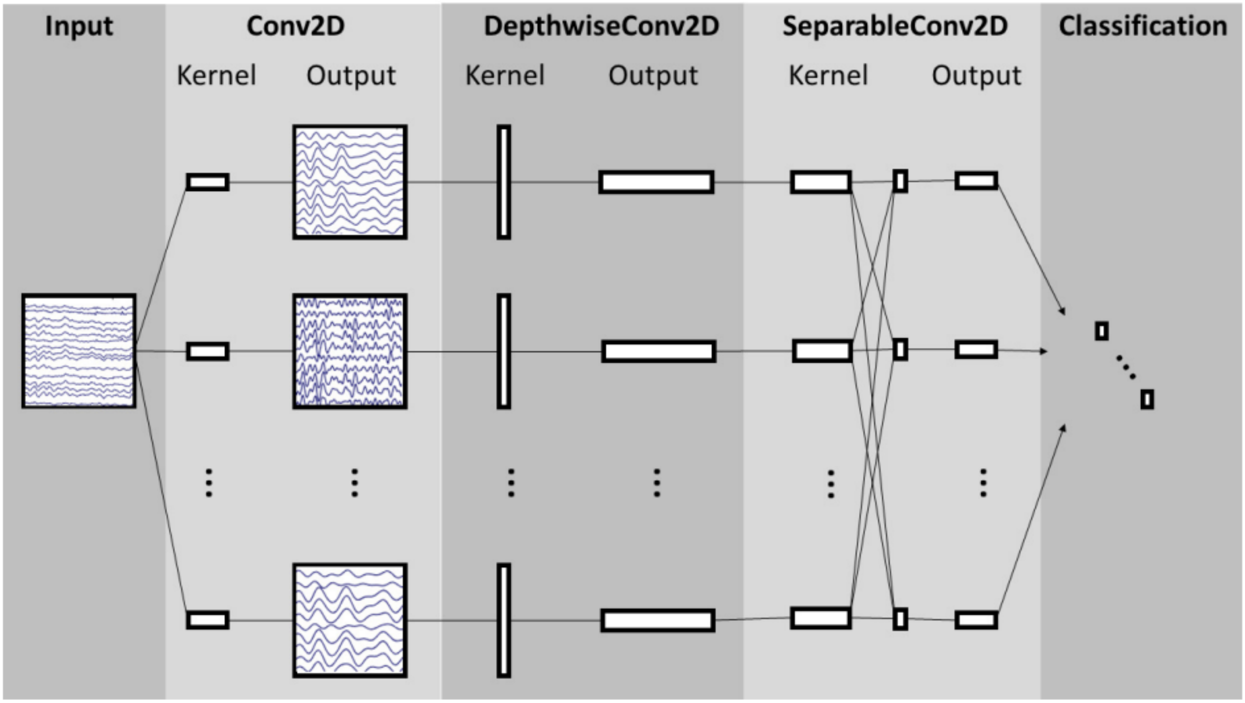

Figure 2: Overall visualization of the EEGNet architecture

The first Conv2D learns how to extract features from signals, then DepthwiseConv2D learns how to combine multiple channel signal into one for each input. The DepthwiseConv2D is different from normal convolution layers in it doesn't fully connect between input and output. Finally SeparableConv2D learns how to extract features from output of DepthwiseConv2D.

```
class EEGNet(nn.Module):
    def __init__(self, activation=None, dropout=0.25):
        super(EEGNet, self).__init__()

        if not activation:
            activation = nn.ELU

        self.firstconv = nn.Sequential(
            nn.Conv2d(
                1, 16, kernel_size=(1, 51),
                stride=(1,1), padding=(0,25), bias=False
            ),
            nn.BatchNorm2d(16)
        )
        self.depthwiseConv = nn.Sequential(
            nn.Conv2d(
```

```
            16, 32, kernel_size=(2,1),
            stride=(1,1), groups=16, bias=False
        ),
        nn.BatchNorm2d(32),
        activation(),
        nn.AvgPool2d(kernel_size=(1, 4), stride=(1, 4), padding=0),
        nn.Dropout(p=dropout)
    )
    self.separableConv = nn.Sequential(
        nn.Conv2d(
            32, 32, kernel_size=(1, 15),
            stride=(1,1), padding=(0, 7), bias=False
        ),
        nn.BatchNorm2d(32),
        activation(),
        nn.AvgPool2d(kernel_size=(1, 8), stride=(1, 8), padding=0),
        nn.Dropout(p=dropout)
    )
    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)
    # flatten
    x = x.view(-1, self.classify[0].in_features)
    x = self.classify(x)
    return x
```

2.3 DeepConvNet

This model architecture has multiple convolution layers. Just like normal CNN. It is comparison to EEGNet in the paper.

To easily modify number of layers and number of kernels, I use one array to present how to build DeepConvNet. e.g. [25, 50] means first layer has 25 output channels and second layer has 50 output channels.

```
from functools import reduce
class DeepConvNet(nn.Module):
    def __init__(self, activation=None, deepconv=[25,50,100,200], dropout=0.5):
        super(DeepConvNet, self).__init__()
```

```
        if not activation:
            activation = nn.ELU

        self.deepconv = deepconv
        self.conv0 = nn.Sequential(
            nn.Conv2d(
                1, deepconv[0], kernel_size=(1, 5),
                stride=(1,1), padding=(0,0), bias=True
            ),
            nn.Conv2d(
                deepconv[0], deepconv[0], kernel_size=(2,1),
                stride=(1,1), padding=(0,0), bias=True
            ),
            nn.BatchNorm2d(deepconv[0]),
            activation(),
            nn.MaxPool2d(kernel_size=(1,2)),
            nn.Dropout(p=dropout)
        )

        for idx in range(1, len(deepconv)):
            setattr(self, 'conv'+str(idx), nn.Sequential(
                nn.Conv2d(
                    deepconv[idx-1], deepconv[idx], kernel_size=(1,5),
                    stride=(1,1), padding=(0,0), bias=True
                ),
                nn.BatchNorm2d(deepconv[idx]),
                activation(),
                nn.MaxPool2d(kernel_size=(1, 2)),
                nn.Dropout(p=dropout)
            ))


        flatten_size =  deepconv[-1] * reduce(
            lambda x,_: round((x-4)/2), deepconv, 750)
        self.classify = nn.Sequential(
            nn.Linear(flatten_size, 2, bias=True),
        )

    def forward(self, x):
        for i in range(len(self.deepconv)):
            x = getattr(self, 'conv'+str(i))(x)
        # flatten
        x = x.view(-1, self.classify[0].in_features)
        x = self.classify(x)
```

```
        return x
```

2.4 Activation functions

Use three kinds of activation functions and compare their outputs.

- ReLU

$$ReLU(x) = max(0, x) \tag{1}$$

- Leaky ReLU

$$LeakyReLU(x) = \begin{cases} x, & \text{if x} > 0 \\ \text{negative\_slope} \times x, & \text{otherwise} \end{cases} \tag{2}$$

- ELU

$$ELU(x) = max(0, x) + min(0, \alpha * (exp(x) - 1)) \tag{3}$$

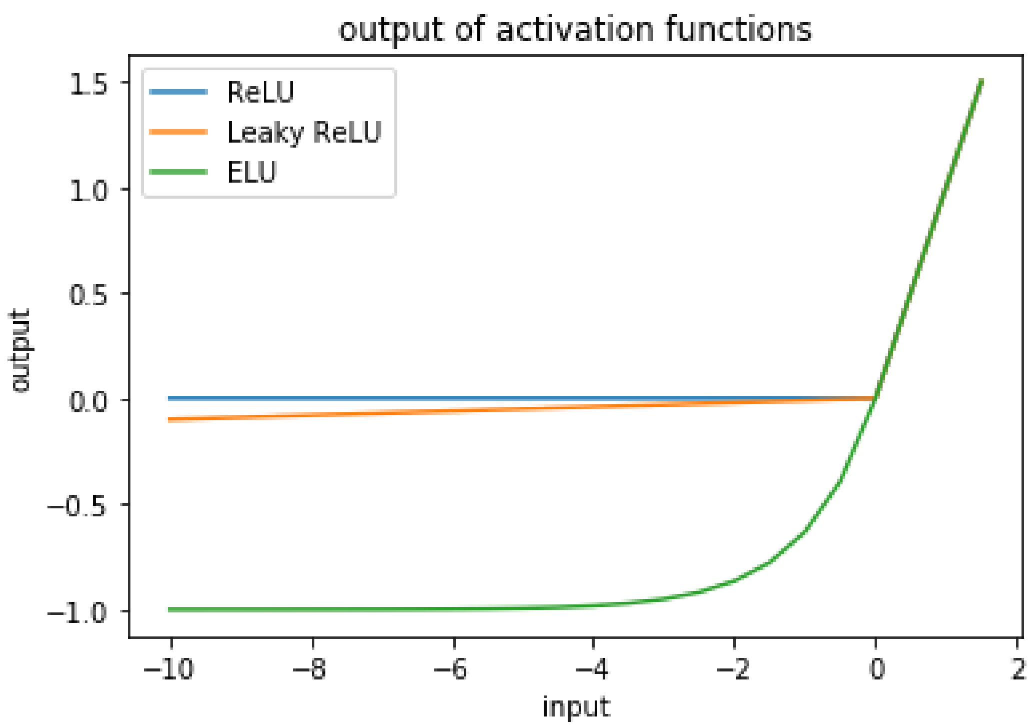

Figure 3: output of activation functions

The big difference between them is output from negative input value. It also means different gradient from negative input value. Let take look at gradient output. To see their difference.

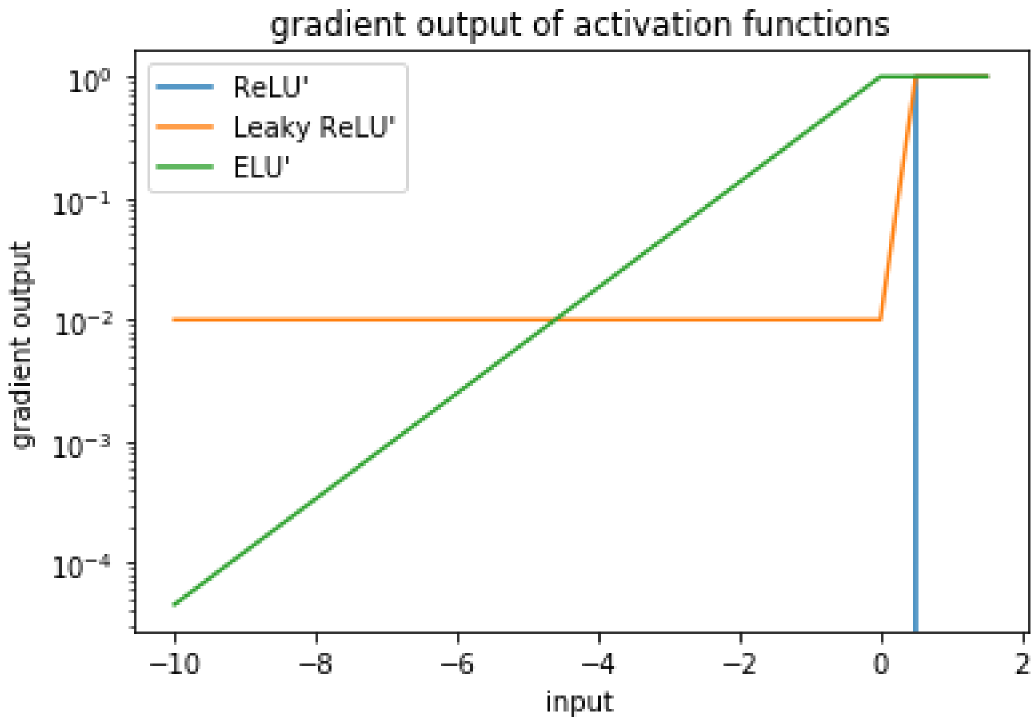

Figure 4: gradient output of activation functions

The ReLU always get zero gradient (log axis can't show zero value) from negative input value that will cause vanishing gradient problem. To avoid vanishing gradient problem, the Leaky ReLU and ELU design different equation at negative input value. In addition, computing volumes of ELU is bigger than Leaky ReLU.

3 Experimental results

Common hyper parameters :

- optimizer : Adam
- criterion (loss) : CrossEntropy
- epoch size : 300
- batch size : 64
- learning rate for EEGNet : 0.01
- learning rate for DeepConvNet : 0.001

3.1 The highest testing accuracy

	ReLU	Leaky ReLU	ELU
EEGNet	78.7962962962963	80.55555555555556	81.01851851851852
EEGNet drop=0.50	80.18518518518519	78.05555555555556	78.33333333333333
DeepConvNet	77.22222222222223	76.29629629629629	77.31481481481481
DeepConvNet deepconv=[50 150 300]	77.31481481481481	75.83333333333333	76.66666666666667

3.2 Comparison figures

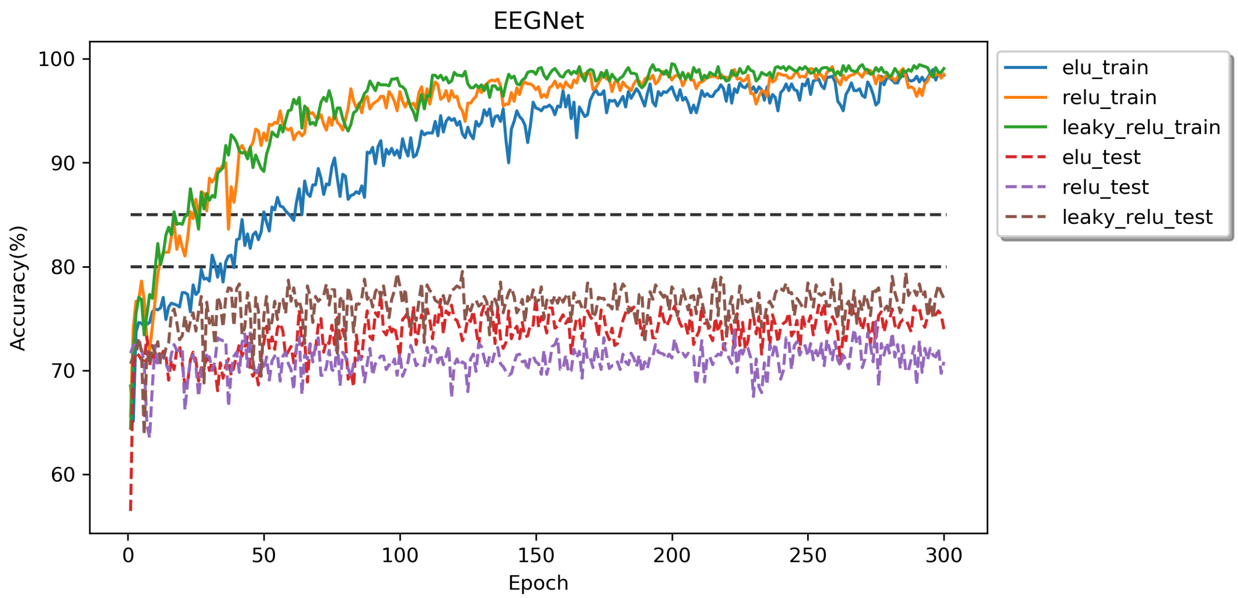

Figure 5: Default EEGNet

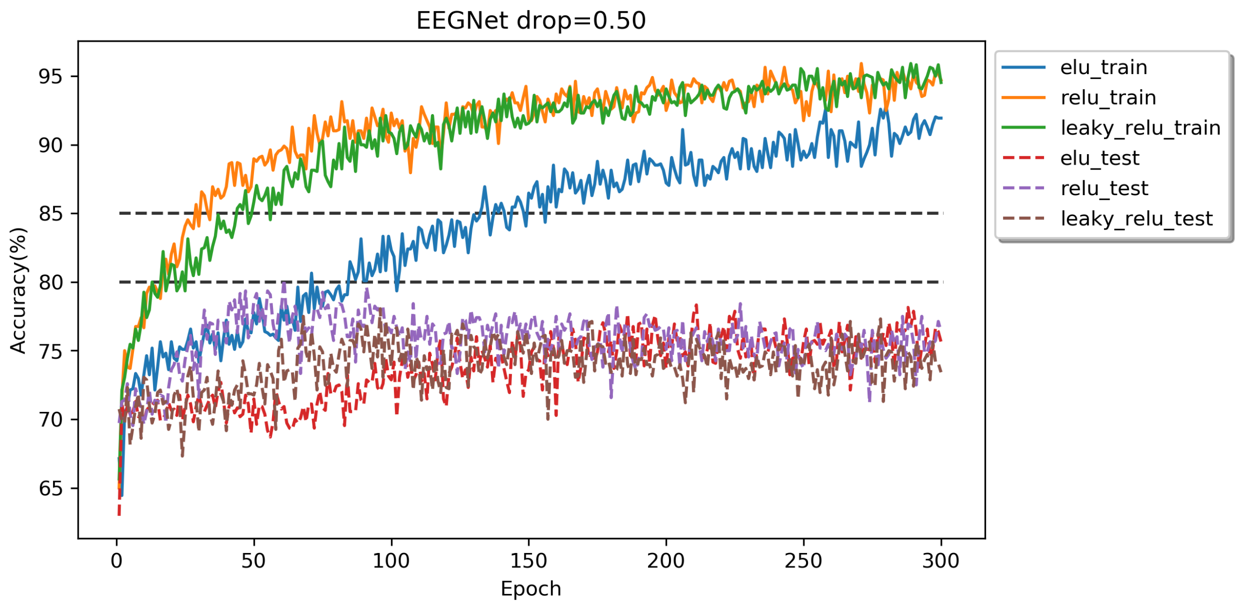

Figure 6: Modified EEGNet dropout = 0.5

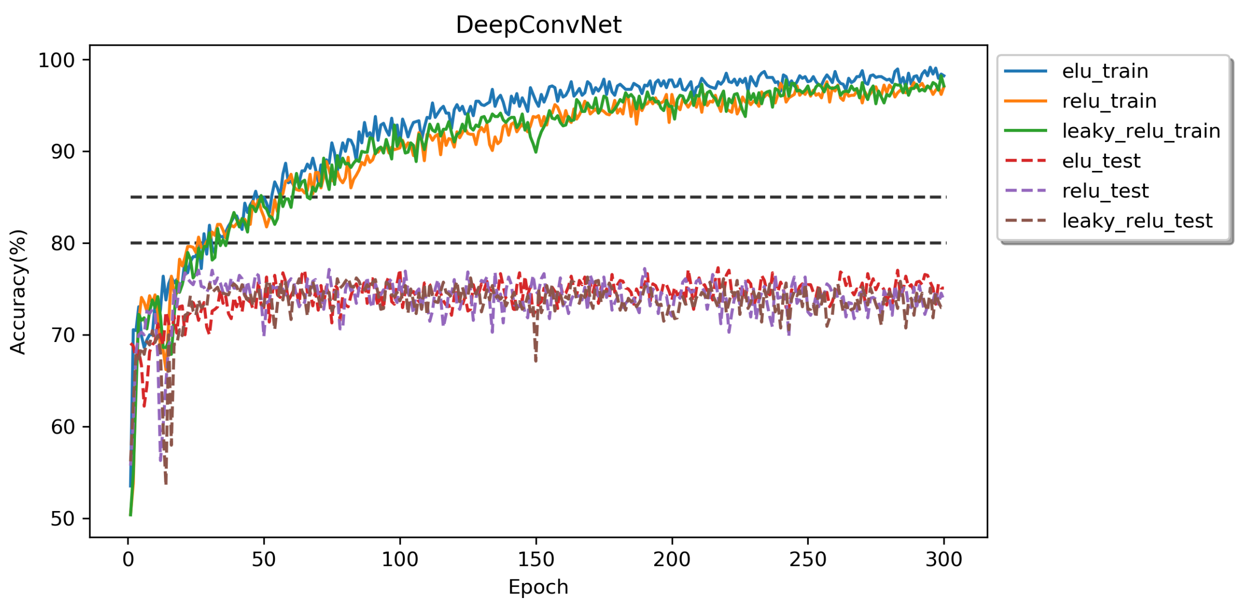

Figure 7: Default DeepConvNet

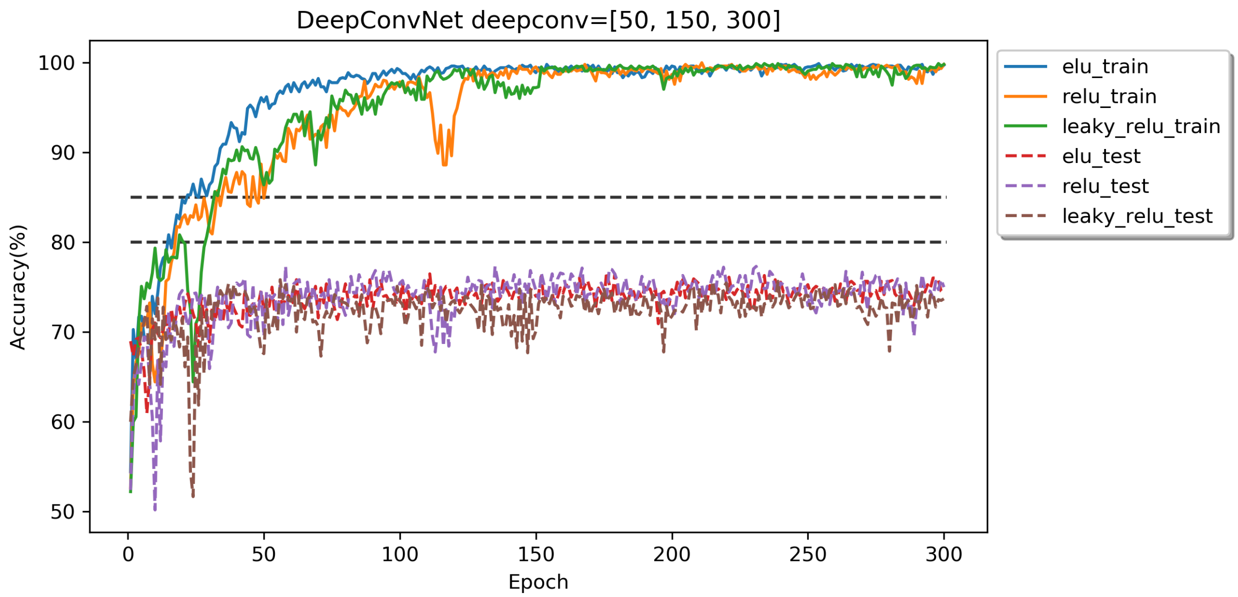

Figure 8: Modified DeepConvNet layers = [50, 150, 300]

4 Discussion

4.1 Dropout and BatchNorm layers with evaluation mode

Because of dropout layer, outputs of model aren't always the same. To solve this problem and leverage all features from testing dataset and that increase accuracy, I set model as evaluation mode when testing. The dropout layer doesn't drop any input value when model is evaluation mode.

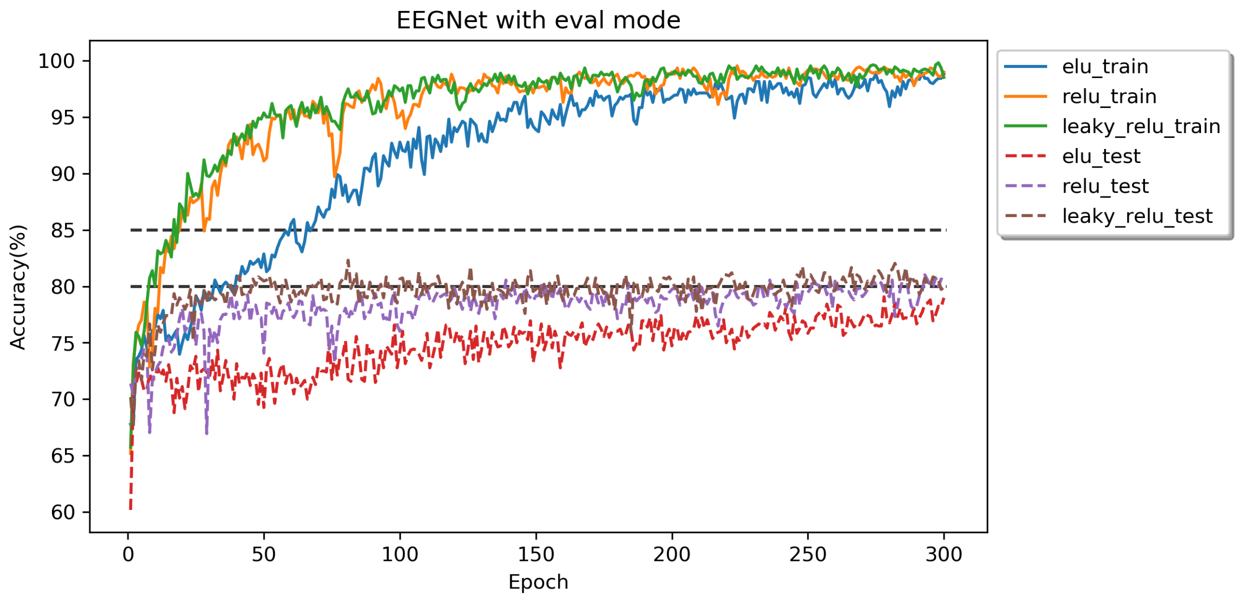

Figure 9: EEGNet with evaluation mode

Then you can find the testing accuracy are more stable than model without evaluation mode.

The evaluation mode also impact BatchNorm layer. BatchNorm layer estimate mean and variance from input values. And evaluation mode can avoid the layer estimating testing input.

4.2 Dropout influence

The dropout layer is used in both models. Hyper parameter of dropout layer is dropout rate that decide how many chance input value becomes zero. If dropout rate is bigger, more input values will become zero, vice versa. The dropout layer causes unstable in output accuracy of model but it can solve overfitting problem. Because model can't learn all feature from training data when training (some input become zero). That means a pattern exist in data even if some features are dropped out, and model need to learn that pattern.

To observe the influence from dropout rate, I tried to modify dropout rate in both models and shown the result as follows.

- dropout rate = 0.9

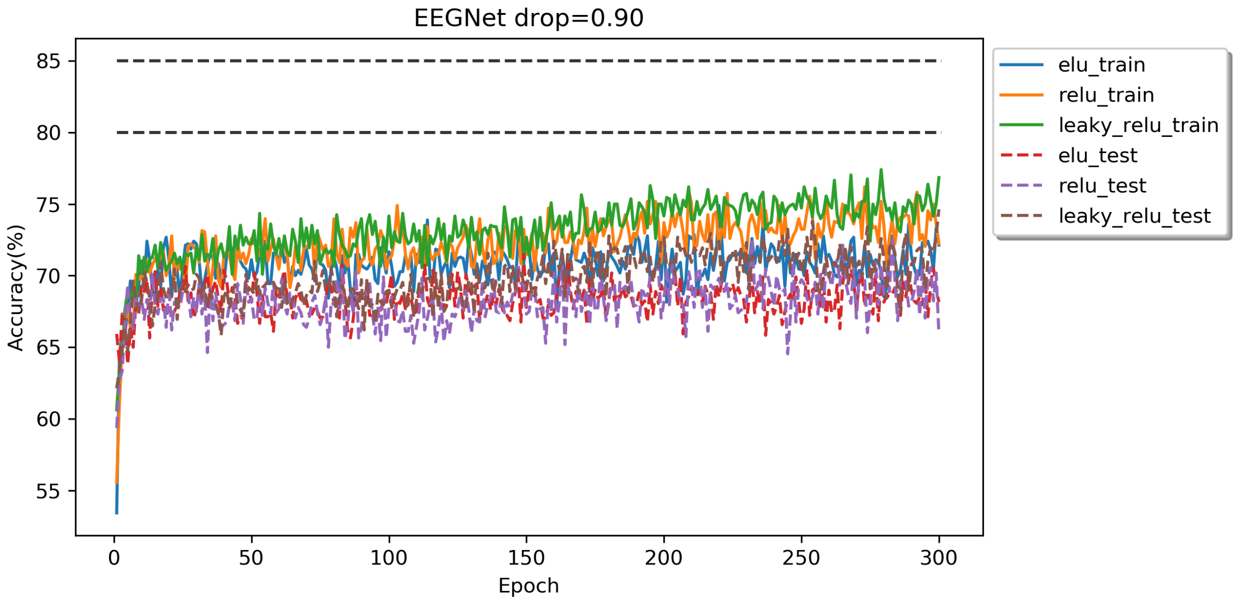

Figure 10: EEGNet dropout rate = 0.9

- dropout rate = 0.5

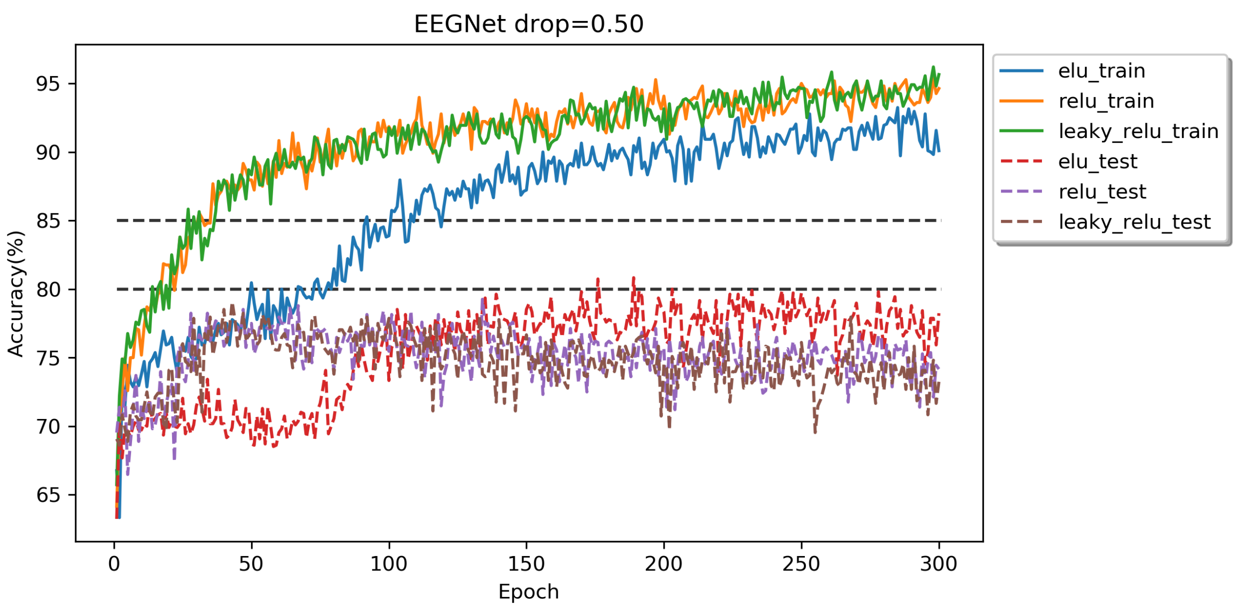

Figure 11: EEGNet dropout rate = 0.5

- dropout rate = 0.1

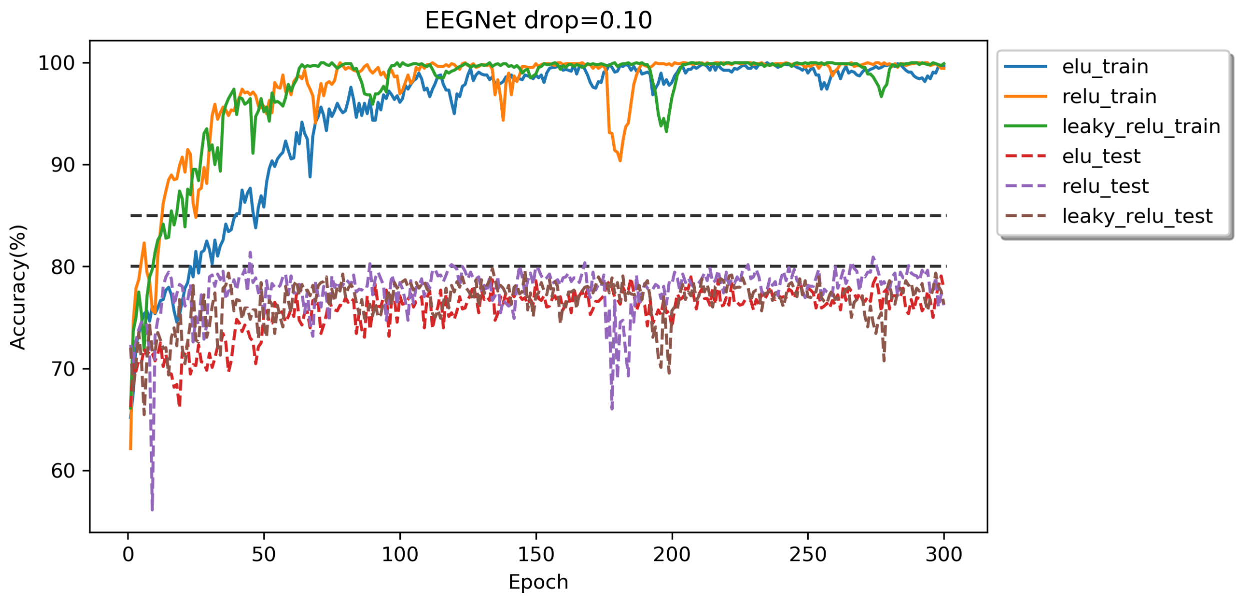

Figure 12: EEGNet dropout rate = 0.1

When dropout rate is high, we can find training accuracy and distance between training and testing accuracy are low. So we need to trade-off dropout rate.