DL HW2

1. Introduction

In this assignment, we should classify the EEG on BCI competition dataset. There are two models, which are EEGNet and DeepConvNet can accomplish this task. The architecture are provided by the TA. We should implement both with pytorch and use three different activation.

2. Experiment set up

A. The detail of your model

- Load data

  I use *torch.utils.data.DataLoader* to load my data. It can randomly choose the batch of step each epoch.

```
dataset = SignalLoader('train')
loader = DataLoader(dataset, batch_size=batch_size,shuffle=True)
dataset_test = SignalLoader('test')
loader_test = DataLoader(dataset_test, batch_size=batch_size,shuffle=False)
class SignalLoader(data.Dataset):
    def __init__(self, mode):
        # self.train_data, self.train_label, self.test_data, self.test_label = read_bci_data()
        if mode == 'train':
            self.data, self.label, _, _= read_bci_data()
        else:
            _, _, self.data, self.label= read_bci_data()

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

    def __getitem__(self, index):
        return torch.from_numpy(self.data[index]), torch.from_numpy(np.array(self.label[index]))
```

- Batch size: 64
- Optimizer: Adam
- Epoch: 10000

```
optimizer_EEG = torch.optim.Adam(EEG.parameters(), lr=lr)
```

- Initial weights

  The initial method is *torch.nn.init.orthogonal_.* I have tried the other initial method: *init.normal_, init.xavier_normal_, init.kaiming_normal_,* and the accuracy of *init.orthogonal_* is the highest.
  init_gain = 0.01 also the best value among other I have tried.

```
EEG = init_weights(EEGNet(), init_type='orthogonal', init_gain=0.01)
```

```
def init_weights(net, init_type='normal', init_gain=0.02):
    def init_func(m):  # define the initialization function
        if isinstance(m, nn.Conv2d):
            if init_type == 'normal':
                init.normal_(m.weight.data, 0.0, init_gain)
            elif init_type == 'xavier':
                init.xavier_normal_(m.weight.data, gain=init_gain)
            elif init_type == 'kaiming':
                init.kaiming_normal_(m.weight.data, a=0, mode='fan_in')
            elif init_type == 'orthogonal':
                init.orthogonal_(m.weight.data, gain=init_gain)
            else:
                raise NotImplementedError('initialization method [%s] is not implemented' % init_type)

    print('initialize network with %s' % init_type)
    net.apply(init_func)  # apply the initialization function <init_func>

    return net
```

- Loss

  Loss function is *torch.nn.CrossEntropyLoss()* . This function is suitable for the classification task.

```
entroy=nn.CrossEntropyLoss()
 loss = entroy(result.float().cuda(GPUID[0]), label.long().cuda(GPUID[0]))
```

- Learning rate

  Initial learning rate is 10e-4. There are two kinds of decay.

  1. decay 10^(1/5) each 50 epoch before 250 epoch and fix

```
if epoch < 250 and (epoch+1)%50==0:
    lr /= (10**(1/5))
```

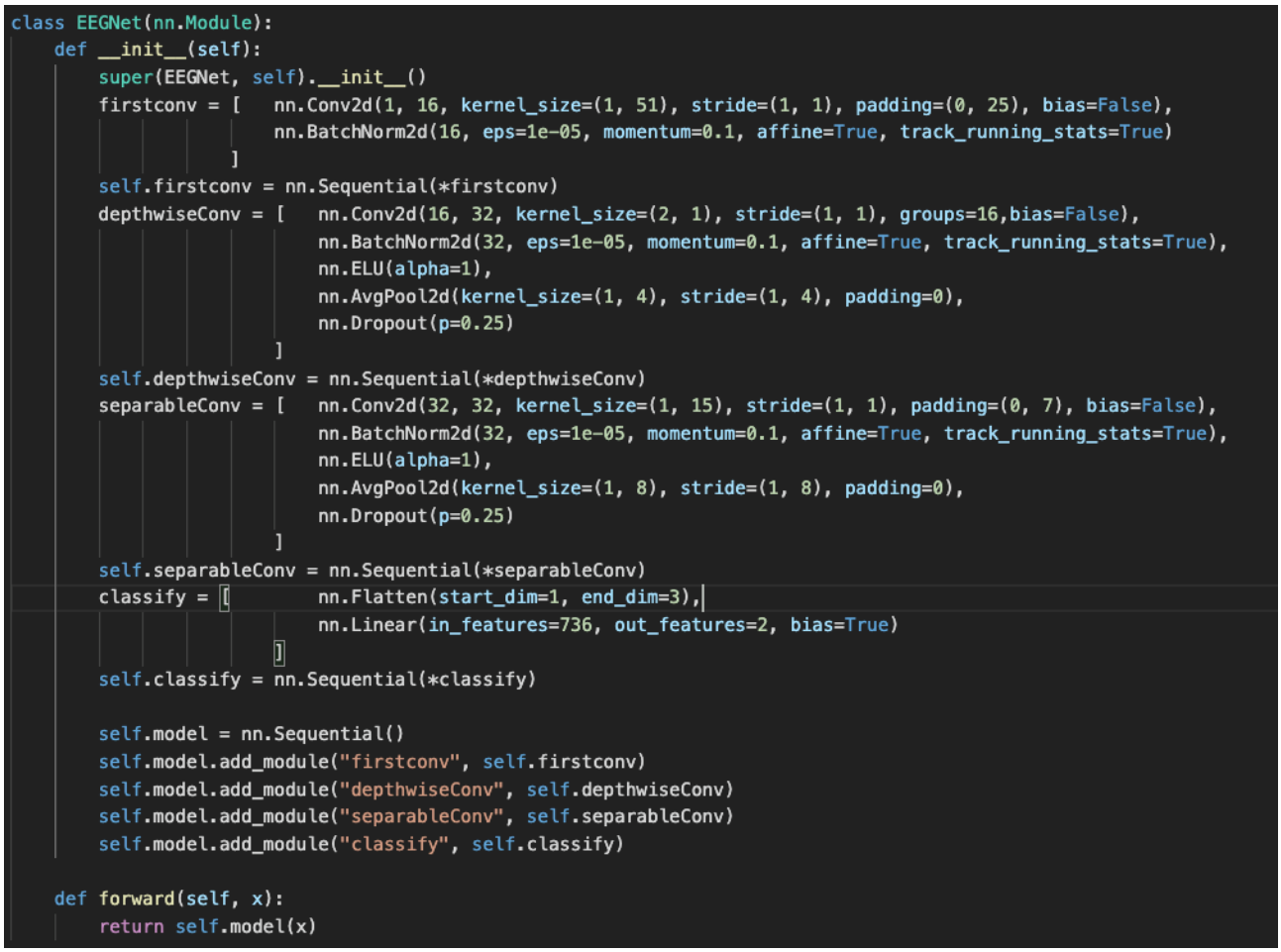
  2. decay 10^(1/5) each 50 epoch before 250 epoch and decay 10^(1/5) each 100 for rest epoch

```
if epoch < 250 and (epoch+1)%50==0:
    lr /= (10**(1/5))
elif (epoch+1)%100==0:
    lr /= (10**(1/5))
```

  EEG with ELU uses 2. , and the other 5 experiment use 1

- EEGNet

  The architecture is all the same as the spec except the layer *nn.Flatten* in ‘classify’. It is the extract layer I add in the network. Without this layer, the channels between previous layer *nn.Dropout(p=0.25)* and last layer *nn.Linear(in_features=736, out_features=2, bias=True)* are not consistent.The other settings to the parameters are all the same as the spec. I search torch.nn and find the corresponding functions to use.
  I buid the blocks of layers ‘firstconv’, ‘depthwiseConv’, ‘separableConv’, ‘classify' respectively. Then, combine them together.



```
class EEGNet(nn.Module):
    def __init__(self):
        super(EEGNet, self).__init__()
        firstconv = [   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.firstconv = nn.Sequential(*firstconv)
        depthwiseConv = [   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),
                            nn.ELU(alpha=1),
                            nn.AvgPool2d(kernel_size=(1, 4), stride=(1, 4), padding=0),
                            nn.Dropout(p=0.25)
                        ]
        self.depthwiseConv = nn.Sequential(*depthwiseConv)
        separableConv = [   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),
                            nn.ELU(alpha=1),
                            nn.AvgPool2d(kernel_size=(1, 8), stride=(1, 8), padding=0),
                            nn.Dropout(p=0.25)
                        ]
        self.separableConv = nn.Sequential(*separableConv)
        classify = [        nn.Flatten(start_dim=1, end_dim=3),
                            nn.Linear(in_features=736, out_features=2, bias=True)
                    ]
        self.classify = nn.Sequential(*classify)

        self.model = nn.Sequential()
        self.model.add_module("firstconv", self.firstconv)
        self.model.add_module("depthwiseConv", self.depthwiseConv)
        self.model.add_module("separableConv", self.separableConv)
        self.model.add_module("classify", self.classify)

    def forward(self, x):
        return self.model(x)
```

- DeepConvNet

  The architecture is all the same as the spec. I take *#filters* as the reference to the parameter *in_channels* and *out_channels.*

```
class DeepCovNet(nn.Module):
    def __init__(self):
        super(DeepCovNet, self).__init__()
        first = [   nn.Conv2d(1, 25, kernel_size=(1, 5), bias=False),
                    nn.Conv2d(25, 25, kernel_size=(2, 1), bias=True),
                    nn.BatchNorm2d(25, eps=1e-05, momentum=0.1),
                    nn.ELU(alpha=1),
                    nn.MaxPool2d(kernel_size=(1, 2)),
                    nn.Dropout(p=0.5)
                ]
        self.first = nn.Sequential(*first)
        second = [
                    nn.Conv2d(25, 50, kernel_size=(1, 5), bias=True),
                    nn.BatchNorm2d(50, eps=1e-05, momentum=0.1),
                    nn.ELU(alpha=1),
                    nn.MaxPool2d(kernel_size=(1, 2)),
                    nn.Dropout(p=0.5)
                ]
        self.second = nn.Sequential(*second)
        third = [
                    nn.Conv2d(50, 100, kernel_size=(1, 5), bias=True),
                    nn.BatchNorm2d(100, eps=1e-05, momentum=0.1),
                    nn.ELU(alpha=1),
                    nn.MaxPool2d(kernel_size=(1, 2)),
                    nn.Dropout(p=0.5)
                ]
        self.third = nn.Sequential(*third)
        forth = [
                    nn.Conv2d(100, 200, kernel_size=(1, 5), bias=True),
                    nn.BatchNorm2d(200, eps=1e-05, momentum=0.1),
                    nn.ELU(alpha=1),
                    nn.MaxPool2d(kernel_size=(1, 2)),
                    nn.Dropout(p=0.5)
                ]
        self.forth = nn.Sequential(*forth)
        classify = [    nn.Flatten(start_dim=1, end_dim=3),
                        nn.Linear(in_features=8600, out_features=2, bias=True)
                    ]
        self.classify = nn.Sequential(*classify)

        self.model = nn.Sequential()
        self.model.add_module("first", self.first)
        self.model.add_module("second", self.second)
        self.model.add_module("third", self.third)
        self.model.add_module("forth", self.forth)
        self.model.add_module("classify", self.classify)


    def forward(self, x):
        return self.model(x)
```

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

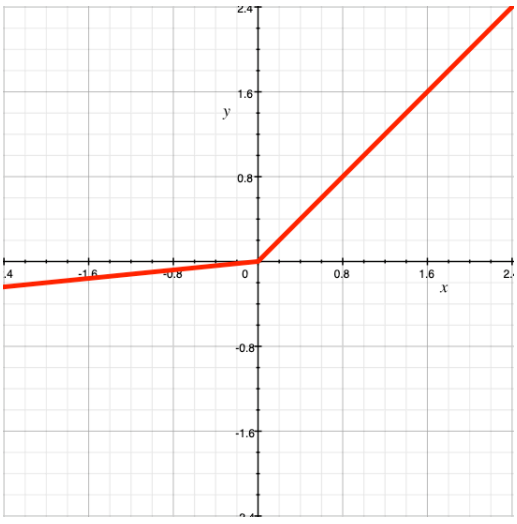
- ReLU

$$R(x) = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases}$$

Simoid function in the assignment 1 limits the value in 0 and 1. When the value is much big or small, the gradient becomes zero.This phenomenon is called gradient vanish.
ReLU can solve the problem above. This why Relu is different from the Sigmoid or tanh. However, Relu may cause another problem. It set the negative value to zero. That means those neurons becomes dead and never activate.

- LeakyReLU

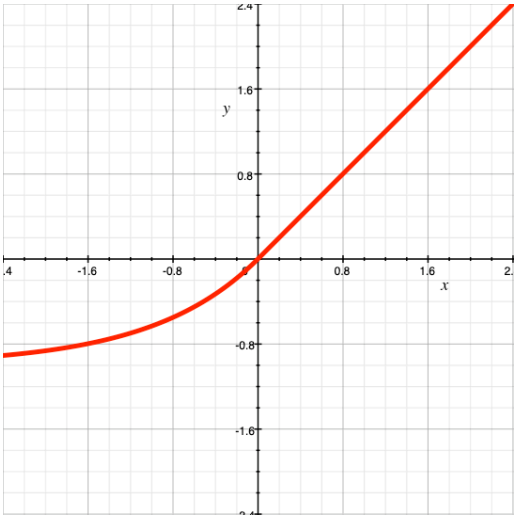
$$R(x) = \begin{cases} x & x > 0 \\ \alpha x & x \leq 0 \end{cases}$$



LeakyRelu set the negative value be the very small negative value. This can give chance to dead neurons activate again.

- ELU

$$R(x) = \begin{cases} x & x > 0 \\ \alpha(e^x - 1) & x \leq 0 \end{cases}$$



This also solve the dead neurons problem. The negative part is the exponential cure. It's differ from ELU.

3. **Experimental results** (30%)

A. The highest testing accuracy

- EEGNet

ReLU	LeakyReLU	ELU
87.59%	**87.5%**	**83.24%**

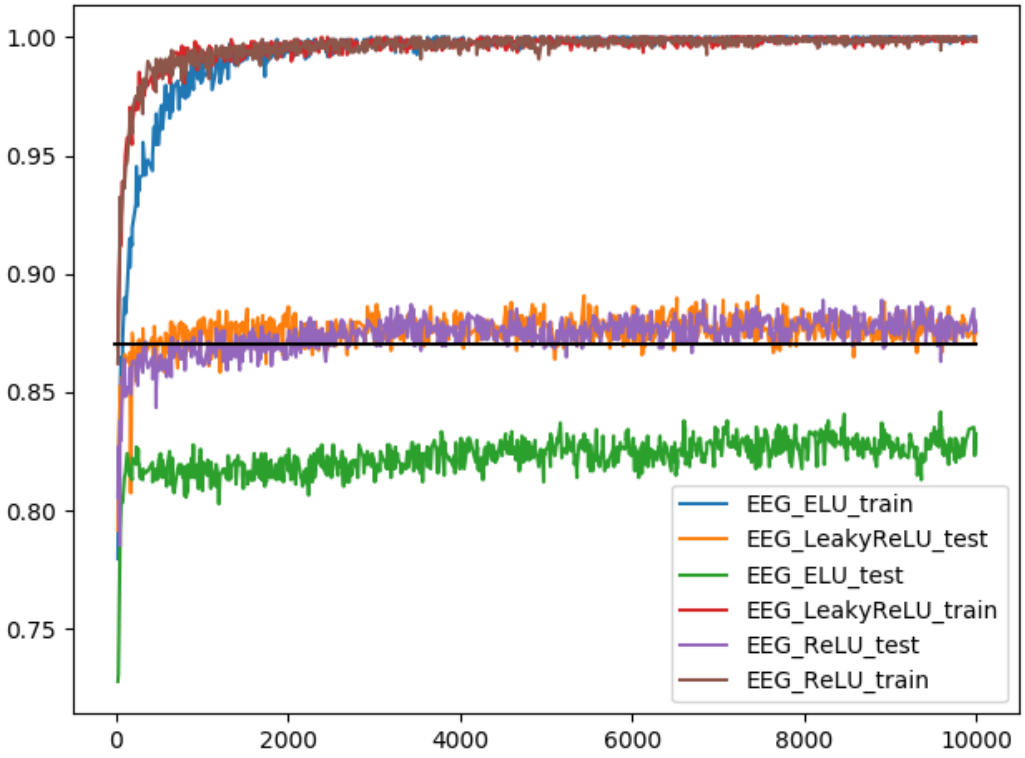
- DeepConvNet

ReLU	LeakyReLU	ELU
81.3%	81.76%	79.26%

B. Comparison figures

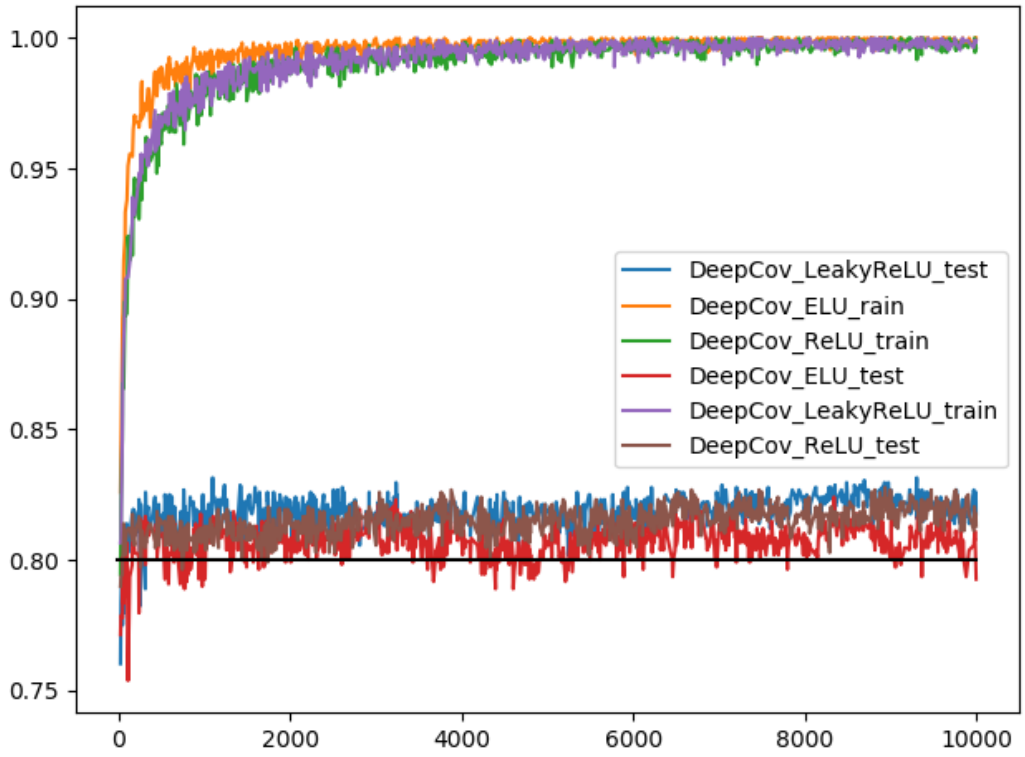
- EEGNet

  The black line is 0.87.



- DeepConvNet

  The black line is 0.80.



4. **Discussion** (20%)

The assignment needs to record the accuracy of train and test each epoch. I test each epoch when I train the model. I use *model.eval()* in the testing part. but I forgot to set *model.train()* in the training part. The accuracy becomes 0.5 when learning rate is small, whatever training part or testing part. This is very strange, so I find this bug for a long time. When I fix it, the result becomes more reasonable.

However, the learning rate is too high and the model overfitting when model works. (the following image)

Thus, I try to set little learning rate and set decay to solve the problem.