Lab2

鄭余玄

1 Introduction

In this lab, we are going to implement both *EEGNet* and *DeepConvNet*. The training and testing dataset is from the BCI Competition with 2 classes and 2 channels. I will explain my implementation in detail in Section 2. In addition, experiments setup and results are described in Section 3.

2 Method

First, my script will parse arguments of hyper-parameters, such as learning rate, activation function, etc. The activation functions are mapped in a `ModuleDict` for the simplicity of further experiments:

```
Activations = nn.ModuleDict([
    ['relu', nn.ReLU()],
    ['lrelu', nn.LeakyReLU()],
    ['elu', nn.ELU()],
])
```

Then, the model is loaded to specified device (default: `cuda`), and the BCI dataset is loaded by `DataLoader` of *pytorch*. During training, the optimizer must set to be zero for each epoch, and then calculate the loss of model. In addition, it is safe to use `torch.no_grad()` to prevent from updating the model when calculating the accuracy.

In addition, since the loss function is `nn.CrossEntropyLoss` which combines both `nn.LogSoftmax` and `nn.NLLLoss`, there is no need to apply `nn.Softmax` layer in the model. For prediction, the `outputs` are logits , and thus should apply a `softmax` layer.

```
for epoch in range(args.epochs):
  for i, (inputs, labels) in enumerate(train_loader, start=1):
```

```
    # zero the parameter gradients
    optimizer.zero_grad()

    # forward + backward + optimize
    outputs = net(inputs)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()

    # batch training accuracy
    with torch.no_grad():
      predicted = torch.argmax(softmax(outputs), dim=1)
      total += labels.size(0)
      correct += (predicted == labels).sum().item()
```

On the other hand, after the convolution in the forward phase, the hidden tensor is flatten by `x.view(-1, dim)`.

2.1 EEGNet

In *EEGNet*, the output shape of the first convolution layer (`firstConv`) is the same as input size; thus the padding size is adjusted according to the kernel size. In contrast to *tensorflow*, *pytorch* can not specified the padding size of a convolution layer to be `SAME`. Therefore, I wrote a `Conv2dSame` inheriting `Conv2d` such that the padding is automatically adjusted.

We know that the output layer of a convolution layer has size:

$$H_{\text{out}} = \left\lfloor \frac{H_{\text{in}} + 2 \times \text{padding}_H - K_H}{\text{stride}_H} + 1 \right\rfloor$$

where K_H is the "full" space of the kernel $K_H = \text{dilation}_H \times (k_H - 1) + 1$, and k_H is the input kernel shape. As a result, for a `SAME` padding layer ($H_{\text{in}} = H_{\text{out}}$), the padding is calculated as the following:

$$2 \times \text{padding}_H = \left(\left\lfloor \frac{H_{\text{in}} + \text{stride}_H - 1}{\text{stride}_H} \right\rfloor - 1 \right) * \text{stride}_H + K_H - H_{\text{in}}$$

Note that if the padding is odd, I choose to pad right and bottom in my implementation.

2.2 DeepConvNet

In *DeepConvNet*, there are four repeated similar parts of convolution layers in the same pattern. Therefore, it could be simplify as in the form of `nn.ModuleList`:

```
channels = [25, 50, 100, 200]
self.convs = nn.ModuleList([
    nn.Sequential(
        nn.Conv2d(in_channels, out_channels, kernel_size=(1, 5)),
        nn.BatchNorm2d(out_channels, **bn_args),
        Activations[act],
        nn.MaxPool2d(kernel_size=(1, 2), stride=(1, 2)),
        nn.Dropout(p),
    ) for in_channels, out_channels in zip(channels[:-1], channels[1:])
])
```

2.3 Activation Function

These activation functions are used in the following experiments:

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

$$\mathrm{LeakyReLU}(x, \mathrm{neg\_slope} = 0.01) = \begin{cases} x & \text{if } x \geq 0 \\ \mathrm{neg\_slope} \times x & \text{otherwise} \end{cases} \tag{2}$$

$$\mathrm{ELU}(x, \alpha = 1.) = \max(0, x) + min(0, \alpha \times (e^x - 1)) \tag{3}$$

In Figure 1, it shows that the *ReLU* activation function has relatively "hard" clip off the input value, while the *LeakyReLU* preserve some information in the negative part. In comparison, *ELU* utilizes the tail of an exponential function as the negative part of its activation, and thus its derivative is more natural when x is close to 0. All of these activation function could solve the vanishing gradient problem from the `softmax` activation function.

$$\frac{\mathrm{d}}{\mathrm{d}x}\mathrm{ReLU}(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

$$\frac{\mathrm{d}}{\mathrm{d}x}\mathrm{LeakyReLU}(x, \mathrm{neg\_slope} = 0.01) = \begin{cases} 1 & \text{if } x \geq 0 \\ \mathrm{neg\_slope} & \text{otherwise} \end{cases} \tag{5}$$

$$\frac{\mathrm{d}}{\mathrm{d}x}\mathrm{ELU}(x, \alpha = 1.) = \begin{cases} 1 & \text{if } x \geq 0 \\ ELU(x) + \alpha & \text{otherwise} \end{cases} \tag{6}$$

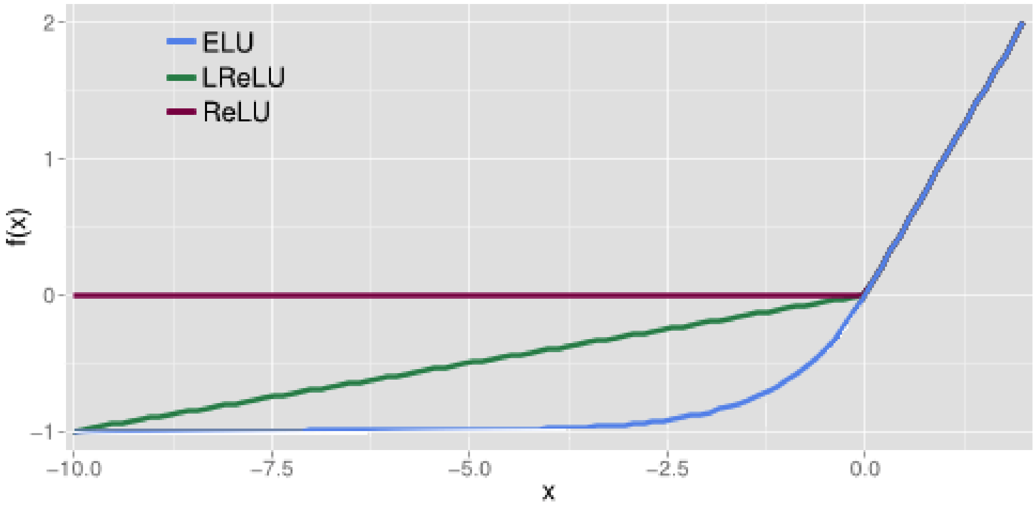


Figure 1: Activation functions

3 Result

	ELU	ReLU	LeakyReLU
EEGNet	87.59%	89.17%	88.06%
DeepConvNet	81.48%	81.67%	81.76%

Table 1: (Best) Test Accuracy

The configuration to reproduce the following experiments are:

```
python3 lab2.py --net EEGNet -lr 0.001 --weight_decay 0.06 --kernel_size 55 --epochs 1200 --batch_size 128 --activation elu
python lab2.py --net DeepConvNet -lr 0.001 --weight_decay 0.06 --batch_size 128 --epochs 300 --activation elu
```

From Table 1, we can conclude that the performance of these three activation functions are similar while `LeakyReLU` performs slightly better. On the other hand, the optimizer used in all experiments is `Adam` which performs the similar as `RMSprop`.

The main efforts on *EEGNet* is (1) the kernel size of the `firstConv` (2) the weight decay of optimizer (3) `BatchNorm2d` parameters. In comparison, the *DeepConvNet* is too easy to over-fit; therefore, it performs relatively worse.

First, in the original paper of *EEGNet*, the kernel size of `firstConv` is 64. However, in this dataset, I figure out that 53 and 55 performs the best while a too large kernel size could not learn well. On the other hand, at first during the training phase, I did not notice the `nn.CrossEntropyLoss` had included the `softmax` function, but I got a acceptable test accuracy. It provides a clue that the weights of the model is too large, and therefore needs a weight decay in the opimizer.

3.1 EEGNet

```
EEGNet(
  (firstConv): Sequential(
    (0): Conv2dSame(1, 16, kernel_size=(1, 55), stride=(1, 1), padding=(None, None), bias=False)
    (1): BatchNorm2d(16, eps=1e-05, momentum=None, affine=True, track_running_stats=False)
  )
  (depthwiseConv): Sequential(
    (0): Conv2d(16, 32, kernel_size=(2, 1), stride=(1, 1), groups=16, bias=False)
    (1): BatchNorm2d(32, eps=1e-05, momentum=None, affine=True, track_running_stats=False)
    (2): ReLU()
    (3): AvgPool2d(kernel_size=(1, 4), stride=(1, 4), padding=0)
    (4): Dropout(p=0.25)
  )
  (separableConv): Sequential(
    (0): Conv2dSame(32, 32, kernel_size=(1, 16), stride=(1, 1), padding=(None, None), bias=False)
    (1): BatchNorm2d(32, eps=1e-05, momentum=None, affine=True, track_running_stats=False)
    (2): ReLU()
    (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)
  )
)
```

3.2 DeepConvNet

```
DeepConvNet(
  (conv0): Sequential(
    (0): Conv2d(1, 25, kernel_size=(1, 5), stride=(1, 1))
    (1): Conv2d(25, 25, kernel_size=(2, 1), stride=(1, 1))
    (2): BatchNorm2d(25, eps=1e-05, momentum=None, affine=True, track_running_stats=False)
    (3): ReLU()
    (4): MaxPool2d(kernel_size=(1, 2), stride=(1, 2), padding=0, dilation=1, ceil_mode=False)
    (5): Dropout(p=0.8)
  )
  (convs): ModuleList(
    (0): Sequential(
      (0): Conv2d(25, 50, kernel_size=(1, 5), stride=(1, 1))
      (1): BatchNorm2d(50, eps=1e-05, momentum=None, affine=True, track_running_stats=False)
      (2): ReLU()
      (3): MaxPool2d(kernel_size=(1, 2), stride=(1, 2), padding=0, dilation=1, ceil_mode=False)
      (4): Dropout(p=0.8)
    )
    (1): Sequential(
      (0): Conv2d(50, 100, kernel_size=(1, 5), stride=(1, 1))
      (1): BatchNorm2d(100, eps=1e-05, momentum=None, affine=True, track_running_stats=False)
      (2): ReLU()
      (3): MaxPool2d(kernel_size=(1, 2), stride=(1, 2), padding=0, dilation=1, ceil_mode=False)
      (4): Dropout(p=0.8)
    )
    (2): Sequential(
      (0): Conv2d(100, 200, kernel_size=(1, 5), stride=(1, 1))
      (1): BatchNorm2d(200, eps=1e-05, momentum=None, affine=True, track_running_stats=False)
      (2): ReLU()
      (3): MaxPool2d(kernel_size=(1, 2), stride=(1, 2), padding=0, dilation=1, ceil_mode=False)
      (4): Dropout(p=0.8)
    )
```

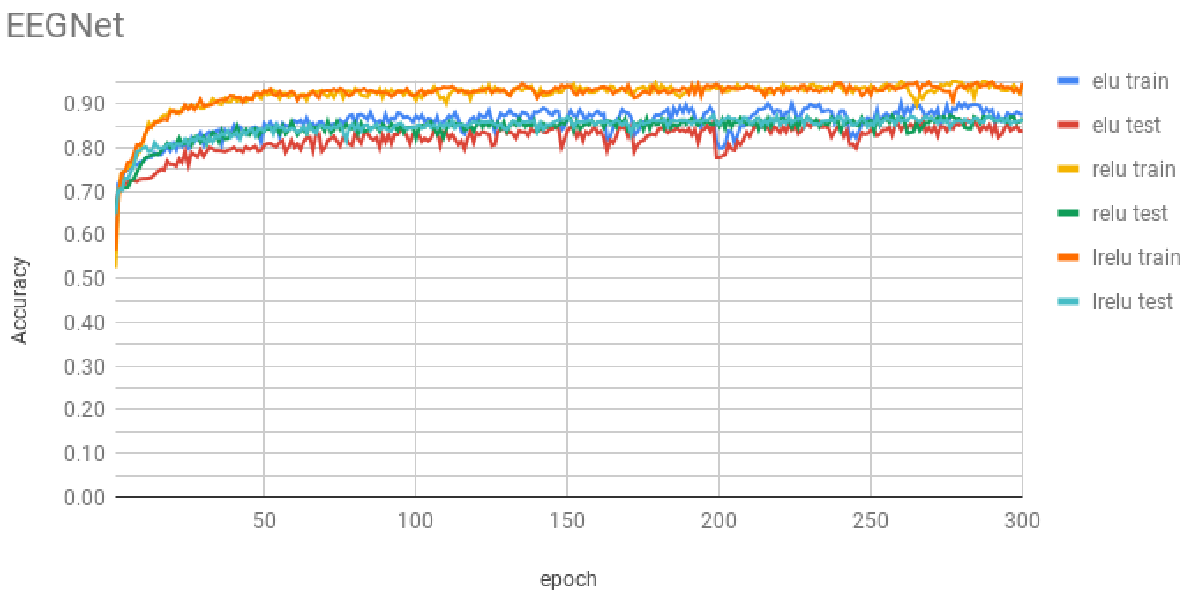
```
  )
  (classify): Sequential(
    (0): Linear(in_features=8600, out_features=2, bias=True)
  )
)
```

3.3 Comparison Figure

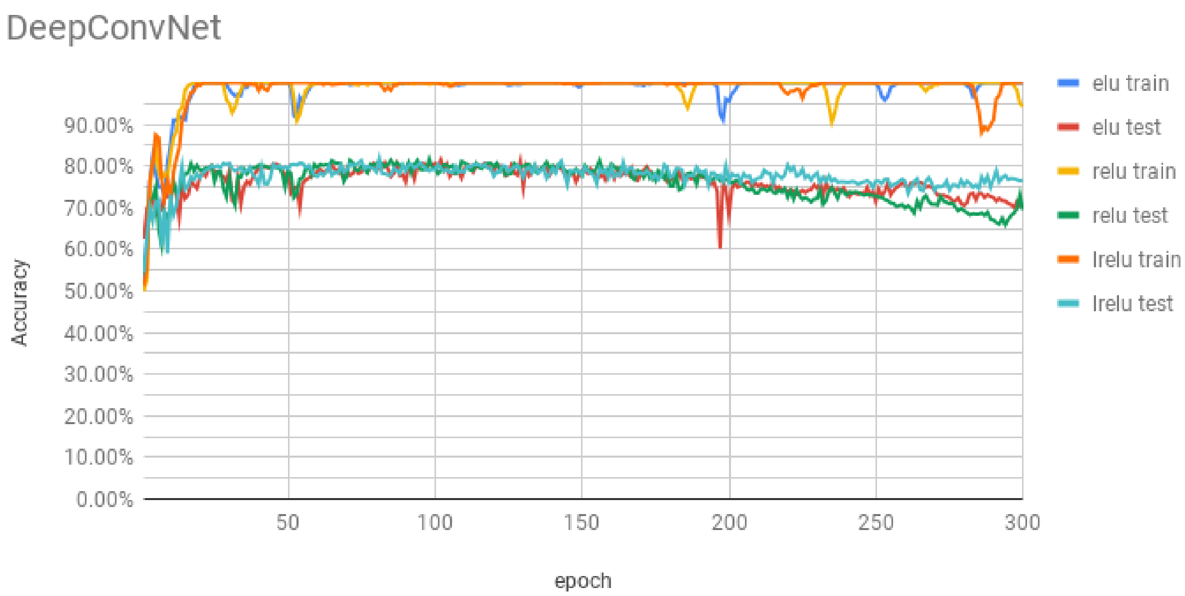
The comparison figure in shown in Figure 2. In both model, the train and test accuracy converges in few epochs. As mentioned above, the training accuracy of *DeepConvNet* converges to 100% very fast, that is overfitting, and the test accuracy is almost not increasing after 100 epochs in contrast to the *EEGNet* which could slightly increase its test accurarcy even at 1200 epochs.

4 Discussion

In this lab, I have tried a lot of skills to improve the performance of models. The most significant result is the *EEGNet* which has a more shallow neural network in comparion to *DeepConvNet*. In contrast, all methods I have had applied on the *DeepConvNet* has little effects and all over-fits in a small epochs, even if the dropout ratio if set to 0.99. These could imply that a shallow neural network would perform better under a limited small size of dataset.



(a) EEGNet Accuracy



(b) DeepConvNet Accuracy

Figure 2: Accuracy