DLP-Lab4-1

ILE 310553043 You-Pin, Chen

April 9, 2022

1 Introduction

In this assignment, we are asked to implement two deep learning model from scratch using Pytorch, which are EEGNet[2] and DeepConvNet[3] respectively. Both of them are compact convolutional neural network for EEG-based BCIs. The dataset for this lab is BCI Competition III dataset that each data consists of two channels of EEG signals. In the experiment results, we have to show the highest accuracy of two architectures with three kinds of activation functions including ReLU, LeakyReLU and ELU. Eventually, we achieved accuracy 88.7% in inference. The code is available in this github[link].

2 Experiment Setup

2.1 Model Structure

2.1.1 EEGNet

EEGNet is a compact CNN architecture for EEG-based BCIs that can be applied across several different BCI paradigms can produce neurophysiologically interpretable features. Most importantly, the architecture are allowed to trained with very limited data. Fig.1 visualize the architecture of EEGNet. The network starts with a temporal convolution (second column) to learn frequency filters, then uses a depthwise convolution (middle column), connected to each feature map individually, to learn frequency-specific spatial filters. The separable convolution (fourth column) is a combination of a depthwise convolution, which learns a temporal summary for each feature map individually, followed by a pointwise convolution, which learns how to optimally mix the feature maps together.

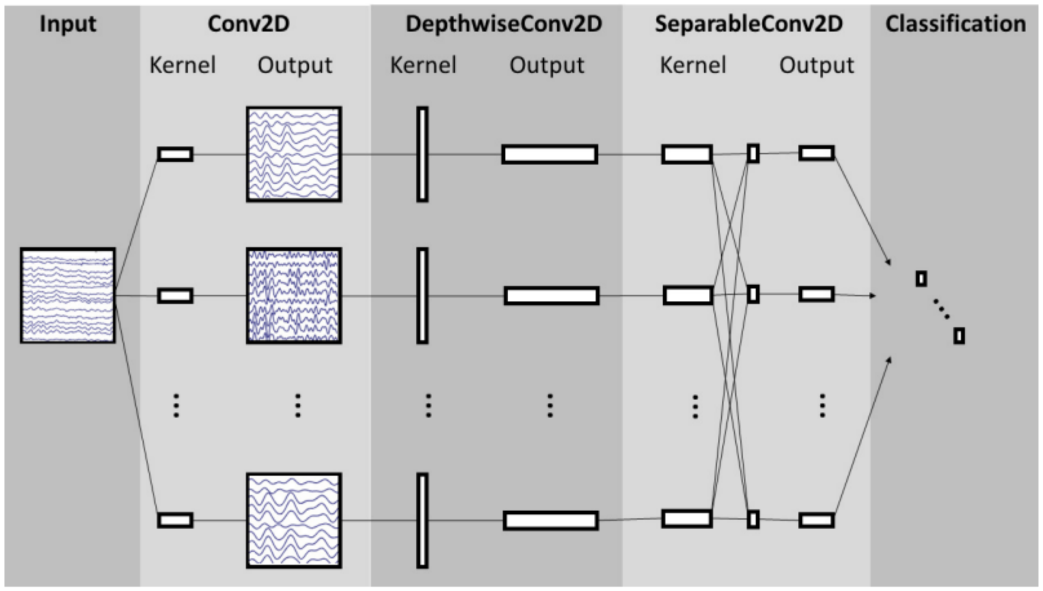


Figure 1: Overall visualization of the EEGNet architecture.

```
'''
The implementation of EEGNet
'''
class EEGNet(nn.Module):
    def __init__(self, activation):
        super(EEGNet, self).__init__()

        self.activation = activation
        self.first_conv = nn.Sequential(
            nn.Conv2d(1, 16, (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.depthwise_conv = nn.Sequential(
            nn.Conv2d(16, 32, (2, 1), stride=(1, 1), groups=16, bias=False),
            nn.BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True),
            self.activation,
            nn.AvgPool2d(kernel_size=(1, 4), stride=(1, 4), padding=0),
            nn.Dropout(p=0.25)
        )
        self.separable_conv = nn.Sequential(
            nn.Conv2d(32, 32, (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),
            self.activation,
            nn.AvgPool2d(kernel_size=(1, 8), stride=(1, 8), padding=0),
            nn.Dropout(p=0.25)
        )
        self.classify = nn.Sequential(
            nn.Linear(in_features=736, out_features=2, bias=True)
        )

    def forward(self, x):
        x = self.first_conv(x)
        x = self.depthwise_conv(x)
        x = self.separable_conv(x)
        x = x.flatten(1)
        x = self.classify(x)
        return x
```

2.1.2 DeepConvNet

DeepConvNet is a CNN model that performs standard convolution operations. As the name implies, it has incredibly deep CNN layers. The architecture is able to extract a wide range of features and is not restricted to specific feature types. DeepConvNet had four convolution-max-pooling blocks, with a special first block designed to handle EEG input, followed by three standard convolution-max-pooling blocks and a dense softmax classification layer.

```
'''
The implementation of DeepConvNet
'''
class DeepConvNet(nn.Module):
    def __init__(self, activation):
        super(DeepConvNet, self).__init__()

        self.activation = activation
        self.cnn_layers = 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),
            self.activation,
            nn.MaxPool2d(kernel_size=(1, 2)),
            nn.Dropout(p=0.5),

            nn.Conv2d(25, 50, kernel_size=(1, 5), stride=(1, 1), padding=(0, 0), bias=True),
            nn.BatchNorm2d(50),
            self.activation,
            nn.MaxPool2d(kernel_size=(1, 2)),
            nn.Dropout(p=0.5),

            nn.Conv2d(50, 100, kernel_size=(1, 5), stride=(1, 1), padding=(0, 0), bias=True),
            nn.BatchNorm2d(100),
            self.activation,
            nn.MaxPool2d(kernel_size=(1, 2)),
            nn.Dropout(p=0.5),

            nn.Conv2d(100, 200, kernel_size=(1, 5), stride=(1, 1), padding=(0, 0), bias=True),
            nn.BatchNorm2d(200),
            self.activation,
            nn.MaxPool2d(kernel_size=(1, 2)),
            nn.Dropout(p=0.5)
        )
        self.classify = nn.Sequential(
            nn.Linear(in_features=43*200, out_features=2, bias=True)
        )

    def forward(self, x):
        x = self.cnn_layers(x)
        x = x.flatten(1)
        x = self.classify(x)
        return x
```

2.2 Activation Function

2.2.1 ReLU

Rectified Linear Unit (Eq.2.2.1) also known as *ReLU*, is one of the popular activation function. It turns the value to 0 if it is negative. Unlike the derivative of sigmoid function will suffer vanishing gradient problem, The derivative of ReLU is 1 for values larger than zero. Because multiplying 1 by itself several times still gives 1, this basically addresses the vanishing gradient problem.

$$ReLU(x) = \begin{cases} x, & if\ x > 0 \\ 0, & if\ x \leq 0 \end{cases} \tag{2.2.1}$$

2.2.2 Leaky ReLU

Nonetheless, the negative component of the ReLU function cannot be discriminated against because it is 0. As a result, the derivative of negative values are simply set to 0. It is not a big deal with forward propagation, but during backpropagation, the gradient is 0. This problem we called *Dead ReLU*. To solve this problem, one variants of ReLU which called *Leaky ReLU* came out. Equation is shown in Eq.2.2.2. Leaky ReLU is defined to address this problem. Instead of defining the ReLU function as 0 for negative values of x, we define it as an extremely small linear component of x.

$$LReLU(x) = \begin{cases} x, & if\ x \geq 0 \\ \alpha \cdot x, & if\ x < 0 \end{cases} \tag{2.2.2}$$

2.2.3 ELU

ELU[1], also know as Exponential Linear Unit is an activation function which is somewhat similar to the ReLU with some differences. Similar to other non-saturating activation functions, ELU does not suffer from the problem of vanishing gradients and exploding gradients. It has proven to be better than ReLU and its variants like Leaky ReLU. Using ELU leads to a lower training times and and higher accuracies in Neural Networks as compared to ReLU, and its variants. ELU activation function is continuous and differentiable at all points. For positive values of input x, the function simply outputs x. Whereas if the input is negative, the output is $\exp(z) - 1$. As input values tend to become more and more negative, the output tends to be closer to 1. The derivative of the ELU function is 1 for all positive values and $\exp(x)$ for all negative values. Mathematically, the ELU activation function can be written in Eq.2.2.3.

$$ELU(x) = \begin{cases} x, & if\ x \geq 0 \\ \alpha \cdot (\exp(x) - 1), & if\ x < 0 \end{cases} \tag{2.2.3}$$

2.3 Hyperparameters Setup

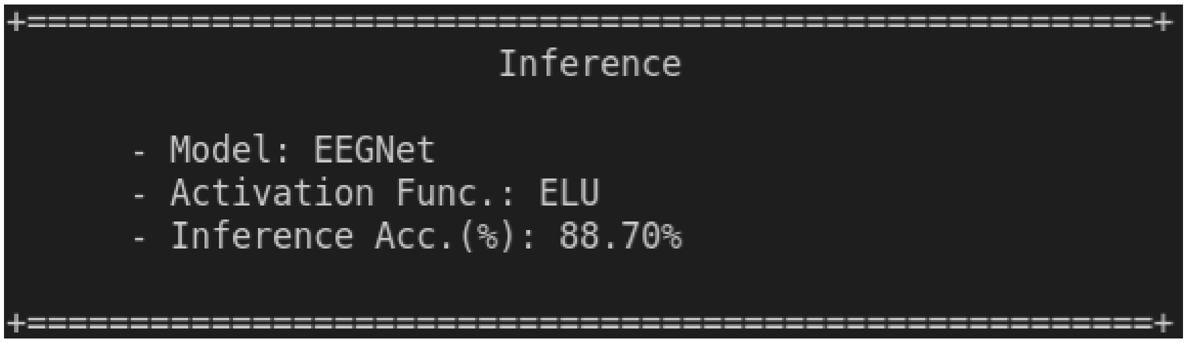
The hyperparameters setup for baseline are as follows:

• *epoch*	500
• *batch size*	128
• *learning rate*	$1e - 3$
• *loss function*	Cross Entropy Loss
• *optimizer*	Adam

3 Experimental Results

3.1 The highest testing accuracy

The setup for our best result is EEGNet with ELU($\alpha = 0.001$). Optimizer choose Adam with weight decay 0.001 and others are same in section 2.3. We achieved accuracy 88.7% in inference.



```
+===========================================================+
                        Inference

     - Model: EEGNet
     - Activation Func.: ELU
     - Inference Acc.(%): 88.70%

+===========================================================+
```

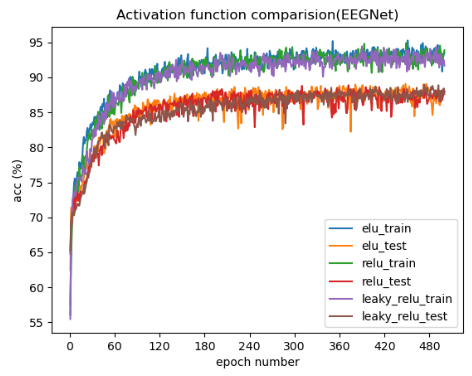
Figure 2: Screenshot for our best result.

	ELU	ReLU	Leaky ReLU
EEGNet	**88.70%**	**87.96%**	**88.61%**
DeepConvNet	83.43%	83.24%	83.43%

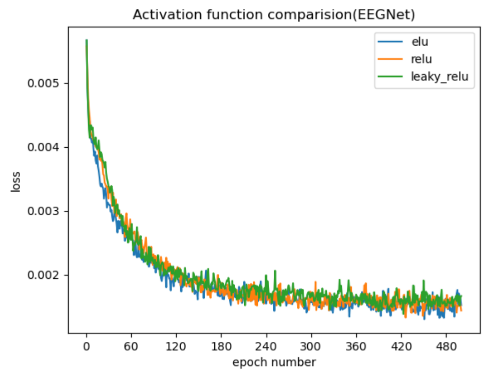
Table 1: Performance Comparison in two architectures with ELU, ReLU and LReLU.

3.2 Comparison figures

3.2.1 EEGNet



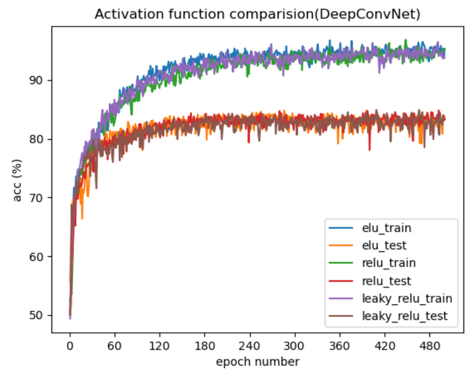
(a) Acc.(%) Rate Curve.



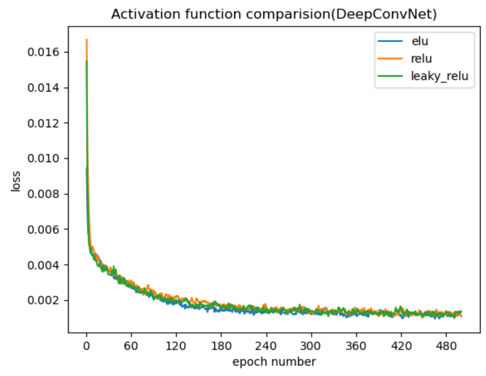
(b) Loss Rate curve

Figure 3: An illustration of the result with EEGNet.

3.2.2 DeepConvNet



(a) Acc.(%) Rate Curve



(b) Loss Rate curve

Figure 4: An illustration of the result with DeepConvNet.

3.2.3 Comparison

Our best result shows in Table.1 which EEGNet get over five percentages accuracy than DeepConvNet. However, the number of parameters of DeepConvNet is about 9 times larger than EEGNet. In Table.2 we can see the comparison of two models. It validates the design of EEGNet is to reduce the computational cost and EEGNet is better for extracting EEG signal features.

	Total Params	Total Size(MB)	Best Acc.(%)
EEGNet	**17,874**	**1.69**	**88.70%** (ELU)
DeepConvNet	150,977	3.62	83.43% (ELU)

Table 2: Comparison in two architectures summary.

4 Discussion

4.1 Regularization

Dropout and *Weight Decay* are both types of regularization methods which Dropout randomly selects some nodes and removes them along with all of their incoming and outgoing connections and weight decay add a small penalty to the loss function to prevent overfitting. First, [2] has already applied Dropout since the parameter p, which means the probability of an element to be zeroed, is 0.25 by default. Since the EEGNet study stated that they set dropout probability to 0.5 because to the large number of data, we attempt to the rules and got our best result. Second, We wanted to see how the penalty affected the loss function when reducing overfitting and pushing the model's performance, we tried many different combinations shows in Table.3.

	ELU(α=0.001)	ReLU	LReLU(α=0.001)	Avg.
$d = 0.25$, $wd = 0$	85.74%	86.94%	86.30%	86.33%
$d = 0.25$, $wd = 0.1$	83.15%	83.80%	83.70%	83.55%
$d = 0.25$, $wd = 0.01$	85.65%	86.57%	86.57%	86.26%
$d = 0.25$, $wd = 0.001$	85.56%	87.69%	86.57%	86.60%
$d = 0.5$, $wd = 0$	86.76%	87.87%	88.43%	**87.69%**
$d = 0.5$, $wd = 0.1$	84.07%	85.28%	83.33%	84.23%
$d = 0.5$, $wd = 0.01$	86.85%	87.87%	88.24%	**87.65%**
$d = 0.5$, $wd = 0.001$	88.70%	87.50%	87.59%	**87.93%**

Table 3: Comparison in different overfitting techniques.

4.2 Alpha value

Then we compared different α value that were mentioned in Sec. 2.2.2 and 2.2.3. The other experiment setup are same as Sec. 2.3. Optimizer using Adam with weight decay($\alpha = 0.001$). Table.4 shows that α value in EEGNet is suitable for setting 0.001 in both ELU and LReLU and is suitable for setting 0.1 in DeepConvNet.

	EEGNet	DeepConvNet	Best
$\alpha_{ELU} = 1.0$	82.59%	78.89%	EEGNet
$\alpha_{ELU} = 0.1$	85.74%	**83.52%**	EEGNet
$\alpha_{ELU} = 0.01$	87.69%	81.11%	EEGNet
$\alpha_{ELU} = 0.001$	**87.78%**	81.94%	EEGNet
$\alpha_{LReLU} = 1.0$	69.44%	77.50%	DeepConvNet
$\alpha_{LReLU} = 0.1$	87.50%	**82.69%**	EEGNet
$\alpha_{LReLU} = 0.01$	87.31%	82.68%	EEGNet
$\alpha_{LReLU} = 0.001$	**88.61%**	80.74%	EEGNet

Table 4: Comparison in different α value.

References

[1] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[2] Vernon J Lawhern, Amelia J Solon, Nicholas R Waytowich, Stephen M Gordon, Chou P Hung, and Brent J Lance. Eegnet: a compact convolutional neural network for eeg-based brain–computer interfaces. *Journal of neural engineering*, 15(5):056013, 2018.

[3] Robin Tibor Schirrmeister, Jost Tobias Springenberg, Lukas Dominique Josef Fiederer, Martin Glasstetter, Katharina Eggensperger, Michael Tangermann, Frank Hutter, Wolfram Burgard, and Tonio Ball. Deep learning with convolutional neural networks for eeg decoding and visualization. *Human brain mapping*, 38(11):5391–5420, 2017.