HW2: EEG classification

Ting-Hsuan, Wang (0756045)

April 7, 2019

1 Introduction

EEG(eletroencephalography) is an electrophysiological monitoring method to record electrical activity of the brain. EEG is usually used to diagnose epilepsy, monitor brain activities, or detect one's sleep stage. However, raw EEG data is so noisy that it has to undergo a preprocessing stages to get the meaningful information from it. Before deep learning comes out, scientists have to handcraft the preprocessing methods, which requiers a prior knowledge about the desired EEG signals. Moreover, in handcrafting preprocessing stages, meaningful information that humen might not aware of may be trimmed off from analysis. As a result, people are figuring out ways to solve this, and currently, the solution seems to be deep learning.
Previously, some models have be proposed to solve different kinds of EEG diagrams, such as using Deep Belief Networks(DBNs) to do sleep stages detection, or using CNNs to predict and monitor epilepsy. Nevertheless, those models are designed specifically for a certain task of EGG. EGGNet, which is proposed in 2018, published a general model to preprocess and make prediction based on raw EGG data. EGGNet is able to not only generalize the tasks, but also trained with limited data, and can produce neurophysiologically interpretable features.
In this homework, we're going to implement EGGNet and a deep convolution network to do classification on different EGG data. Also, we're going to do some parameters tunning as well as do experiments on different activation functions to see which gets the highest accuracy. At the end of the report, we'll also give some discussion on the network architectures.

2 Experiment set up

2.1 Network Architecture

For both EEGNet and DeepConvNet we implement here are slightly modified versions of the original ones provided by TA. Further discussions about network architecture modification and parameters tuning are introduced in the "Discussion" section.

```
EEGNet(
  (firstconv): Sequential(
    (0): Conv2d(1, 16, kernel_size=(1, 12), stride=(1, 1), padding=(0, 6))
    (1): Conv2d(16, 16, kernel_size=(1, 25), stride=(1, 1), padding=(0, 12))
    (2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (3): ELU(alpha=0.9)
  )
  (depthwiseConv): Sequential(
    (0): Conv2d(16, 16, kernel_size=(1, 1), stride=(1, 1))
    (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ELU(alpha=0.9)
    (3): Conv2d(16, 32, kernel_size=(2, 1), stride=(1, 1), groups=16, bias=False)
    (4): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ELU(alpha=0.9)
    (6): AvgPool2d(kernel_size=(1, 4), stride=(1, 4), padding=0)
    (7): Dropout(p=0.2)
  )
  (separableConv): Sequential(
    (0): Conv2d(32, 32, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3))
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ELU(alpha=0.9)
    (3): AvgPool2d(kernel_size=(1, 8), stride=(1, 8), padding=0)
    (4): Dropout(p=0.2)
  )
  (classify): Sequential(
    (0): Linear(in_features=736, out_features=2, bias=True)
  )
)
```

Figure 1: EEGNet Model

```
DeepConvNet(
  (conv1): Sequential(
    (0): Conv2d(1, 25, kernel_size=(1, 5), stride=(1, 1), padding=(0, 2))
    (1): BatchNorm2d(25, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.04, inplace)
    (3): Conv2d(25, 25, kernel_size=(2, 1), stride=(1, 1), padding=(1, 0))
    (4): BatchNorm2d(25, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): LeakyReLU(negative_slope=0.04, inplace)
    (6): MaxPool2d(kernel_size=(1, 2), stride=(1, 2), padding=0, dilation=1, ceil_mode=False)
    (7): Dropout(p=0.15)
  )
  (conv2): Sequential(
    (0): Conv2d(25, 50, kernel_size=(1, 5), stride=(1, 1), padding=(0, 2))
    (1): BatchNorm2d(50, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.04, inplace)
    (3): MaxPool2d(kernel_size=(1, 2), stride=(1, 2), padding=0, dilation=1, ceil_mode=False)
    (4): Dropout(p=0.15)
  )
  (conv3): Sequential(
    (0): Conv2d(50, 100, kernel_size=(1, 5), stride=(1, 1), padding=(0, 2))
    (1): BatchNorm2d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.04, inplace)
    (3): MaxPool2d(kernel_size=(1, 2), stride=(1, 2), padding=0, dilation=1, ceil_mode=False)
    (4): Dropout(p=0.15)
  )
  (conv4): Sequential(
    (0): Conv2d(100, 200, kernel_size=(1, 5), stride=(1, 1), padding=(0, 2))
    (1): BatchNorm2d(200, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.04, inplace)
    (3): MaxPool2d(kernel_size=(1, 2), stride=(1, 2), padding=0, dilation=1, ceil_mode=False)
    (4): Dropout(p=0.15)
  )
  (linear): Sequential(
    (0): Linear(in_features=27600, out_features=2, bias=True)
  )
)
```

Figure 2: DeepConvNet Model

2.2 Activation function

2.2.1 ReLU

The function of ReLU looks like this:

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

ReLU is the most commonly used activation function in neural networks. The simple mathematical formula of ReLU makes the model easy and fast to calculate from both forward and backward pass. Also, ReLU converges faster because ReLU is a linear function when the values are positive. Thus, ReLU won't saturate when the positive input value gets larger. Some activation functions such as sigmoid, suffered from such saturated problems. However, due to the fact that ReLU won't update when the input is less than or equal to zero. To see that, assume our for some input x_n is

$$z_n = \sum_{i=0}^{k} w_i a_i^n \tag{2}$$

We get:

$$error = ReLU - y = max(0, z_n) \tag{3}$$

And the error term with respect to the output z_n looks like this:

$$\frac{\partial error}{\partial z_n} = \delta_n = 1 \; if \; z_n \geq 0, \; otherwise \, 0 \tag{4}$$

So when we calculate the gradient, it becomes:

$$\frac{\partial error}{\partial w_j} = \frac{\partial error}{\partial z_n} \frac{\partial z_n}{\partial w_j} = \delta_n \times a_j^n = a_j^n \, if z_n \geq 0, \; otherwise \, 0 \tag{5}$$

In some unfortunate case, our weights might stop updating due to the zero gradient problem caused by dead ReLU problems. Luckily, by just slightly modifying the zero-slop part on the original ReLU term, our weights can start updating again. There're are several variants of ReLU, and we're going to introduce two of them, which are Leaky ReLU and ELU respectively.

2.2.2 Leaky ReLU

The function of Leaky ReLU looks like this:

$$x \, when \, x \geq 0 \, , otherwise \; 0.01x \tag{6}$$

Leaky ReLU output small slope instead of 0 for negative input. In this way, we can still have gradients on negative input. There's a more general form of Leaky ReLU called Parametric ReLU(PReLU) ($y = ax$, $when \, x0$), which learns the slope of nagative term by the neural network.

2.2.3 ELU

As for ELU(Exponential Linear), it looks like this:

$$y = a(\exp(x) - 1),\ when\, x0,\ otherwise\, x \tag{7}$$

ELU combines ReLU and Leaky ReLu. However, ELU saturates for large negative values.

3 Experiment results

3.1 Testing accuracy

```
ELU: alpha=0.9(train): 100.0
ELU: alpha=0.9(test): 87.96296296296296
ReLu(train): 100.0
ReLu(test): 85.64814814814815
LeakyReLu: neg_slope=0.03(train): 100.0
LeakyReLu: neg_slope=0.03(test): 88.33333333333333
```

Figure 3: EEGNet Accuracy

```
ELU: alpha=0.9(train): 100.0
ELU: alpha=0.9(test): 82.5
ReLu(train): 100.0
ReLu(test): 83.7037037037037
LeakyReLu: neg_slope=0.04(train): 100.0
LeakyReLu: neg_slope=0.04(test): 82.31481481481482
```

Figure 4: DeepConvNet Accuracy

3.2 Comparison figures

Different network architecture pairs with different activation functions will results in different results. Some network architecture with LeakyReLu get better accuracies, while other activation functions lead to worse results. We simply choose the architecture that performs the best with any one of the activation functions with tuned activation function's parameters.

With regard to the performance of EEGNet and DeepConvNet, it is anticipated that EEGNet's accuracy will outperform DeepConvNet's by some percentages, since EEGNet is designed specially for general EEG data.

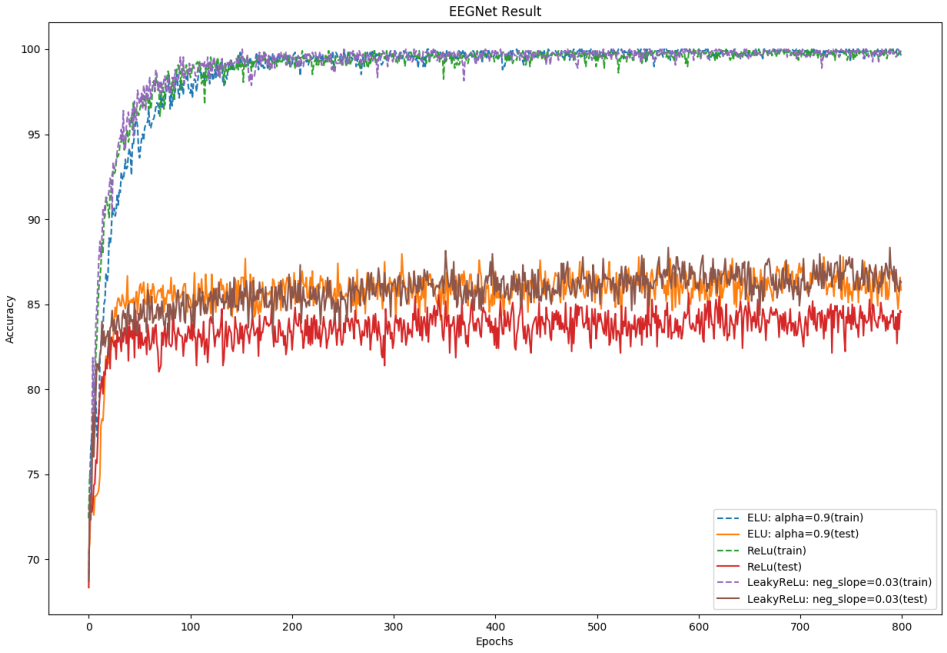

Figure 5: EEGNet Result

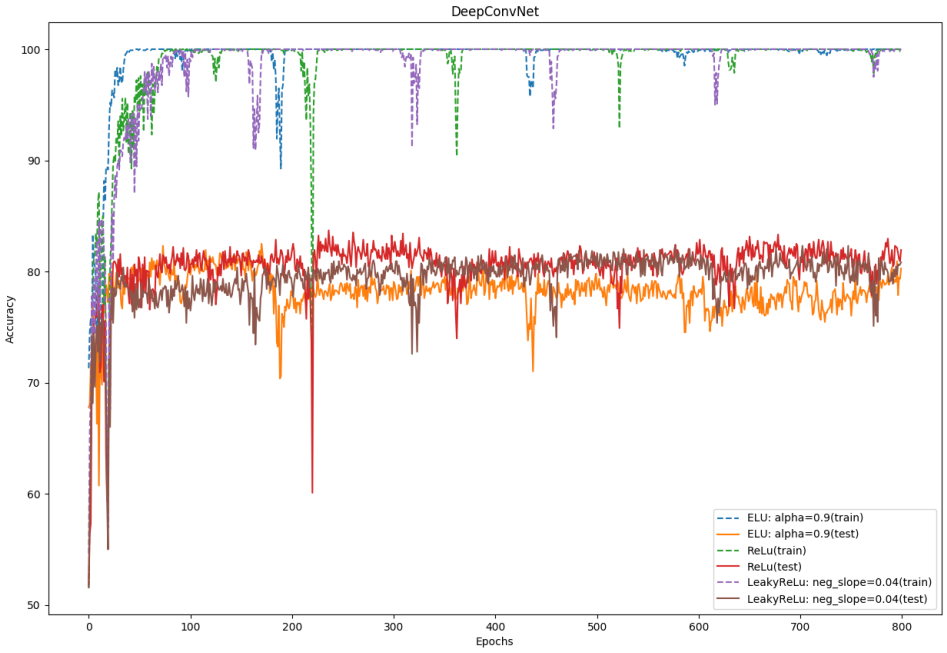

Figure 6: DeepConvNet Result

4 Discussion

To further get better results, we first modify model architectures by adding more layers, then we fine tune the parameters using grid search.

4.1 Model architecture

One of the most intuitive way to achieve higher accuracy is adding more parameters for the model to learn. For both EEGNet and DeepConvNet, we added some extra convolution layers, batch normalization layers and activation layers by shrinking the kernel and padding size, which indeed gave rise to the accuracy.
We also tried different kinds of pooling methods, including max pooling and average pooling. At first, we thought max pooling in DeepConvNet would result in lower accuracy, since max pooling is more like a "brutal method" to shrink model sizes. However, changing max pooling to average pooling even lower the accuracy. As for EEGNet, we've already expected changing the pooling method would get worse result, since this network architecture is specifically designed for EEG data. And the observed results were the same as we thought before.
Furthermore, we do experiments on adding and not adding dropout. Dropout is used as a way to regularize the model, yet some claim that adding dropout often lead to worse model. We thus tried adding and not adding dropout layers. Finally, the results told us that in this dataset, adding dropout gave us better model predictions.
From all the above experiments, we learned that there's no rule of thumb when it comes to designing neural network architecture. The results vary from case to case. Maybe there's a good way to start out the architecture at the beginning, but there's never a definite rule.

4.2 Parameters tuning

After roughly decided what our model looks like, we went into next step to fine tune the parameters to get better results. We first tried different parameter sets on activation functions, ELU and LeakyReLu (and no ReLu, since it did not have parameter to tune), respectively. This led to relativly huge improvements compare to tuning other parameters such as momentum and eps in batch normalization layers. In addition, adding bias term on each layer slightly increase the accuracy.

5 Reference

1. What is the “dying ReLU” problem in neural networks?
2. A practical guide to relu