

Introduction

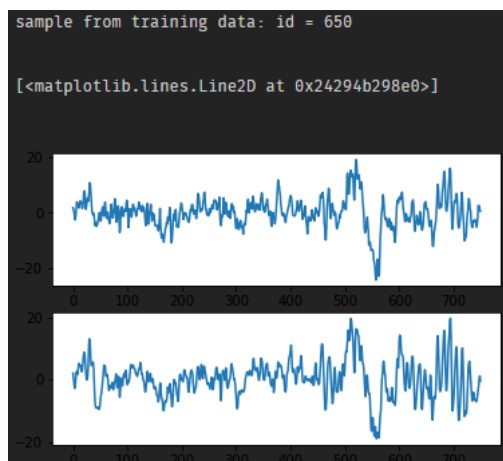
In this lab, I will implement **EEGNet** and **DeepConvNet** on **BCI competition dataset**, to classify the **EEG signals**. In both these 2 models, I will both try 3 different kinds of activation functions: **ELU**, **ReLU** and **LeakyReLU**.

The data are from the BCI competition dataset. Each data observation has **2** bipolar EEG channels, and has a label (either *left hand* or *right hand*). In other words, this task is a **binary classification problem**. They are 2 training data files and 2 testing data files, which can be preprocessed by the function `read_bci_data()` in the given script

dataloader.py (<http://dataloader.py>).

Data introduction

Both the data have the shape of (1080, 1, 2, 750), which means the data sizes are 1080 for both training and testing data. Each observation can be deemed as a **single-channel** image, with a (2×750) resolution. 2 is the number of bipolar EEG channels and 750 is the number of total time steps. A randomly sampled training observation looks like this:



As for the labels (preprocessed), both training and testing labels have the shape (1080,). There are only 2 possible values: **0** and **1**.

```
np.unique(y_train)
```

```
array([0., 1.])
```

Before building any model, we first wrap these data and labels into a **PyTorch DataLoader** via the help of `torch.utils.data.TensorDataset` and `torch.utils.data.DataLoader`. We shuffled the data in training data but not those in testing data.

Hyper Parameters Settings

The **batch sizes** for both training data and testing data are **270**, which is a quarter ($\frac{1}{4}$) of the whole data sizes.

```
1 | batch_size = 270
```

The **learning rate** is **0.001** (10^{-3}).

```
1 | lr = 0.001
```

The **training epochs** are **300**.

```
1 | epochs = 300
```

The **optimizer** is the **AdamW**, which is simply just the *Adam* optimizer with *weight decay*. All the arguments are left as default (i.e. weight decay = 0.01).

```
1 | import torch.optim as optim
2 | optimizer = optim.AdamW(model.parameters(), lr = lr)
```

The **loss function** is the **(binary) cross entropy loss**, since the task is actually a binary classification problem.

```
1 | import torch.nn as nn
2 | loss_fn = nn.CrossEntropyLoss()
```

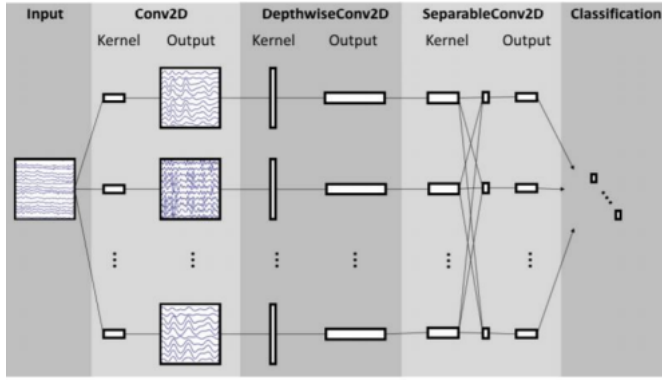
2 models (**EEGNet**, **DeepConvNet**) together with 3 activation functions (**ELU**, **ReLU**, **LeakyReLU**), all 6 combinations have the same hyper parameters settings.

Experiment Setup

A. The detail of models

◆ EEGNet

The model architecture of EEGNet looks like so:



EEGNet takes advantage of the **depthwise separable convolution** technique, which is also used in other image classification network used in *computer vision* realm such as **MobileNet**

(paper link for MobileNet:

<https://arxiv.org/pdf/1704.04861.pdf>

(<https://arxiv.org/pdf/1704.04861.pdf>)).

The advantage of depthwise separable convolution is that it is **computationally more efficient** than standard convolution.

Let W_k, H_k be the width and height of the convolution filters, and W_f, H_f be those of the input frames. Let M be the number of input channels and N be the number of output channels. Also, assume that $stride = 1$ and $padding = 1$. Then the number of operations needed for a standard convolution is:

$$W_k \times H_k \times M \times N \times W_f \times H_f$$

On the other hand, depthwise separable convolution consists of 2 different kinds of convolution steps:

depthwise convolution and **pointwise convolution**.

In the **depthwise convolution**, all M filters only have **1** channel and only correspond to **1** input channel in the input frame. The number of operations needed is:

$$W_k \times H_k \times M \times W_f \times H_f$$

In the **pointwise convolution**, all N filters have a small size of **1x1**, but have M channels. The number of operations needed is:

$$M \times N \times W_f \times H_f$$

So, the total number of operations needed is

$$W_k \times H_k \times M \times W_f \times H_f + M \times N \times W_f \times H_f$$

which is basically less than

$W_k \times H_k \times M \times N \times W_f \times H_f$ of a standard convolution.

The following image shows the structure of my EEGNet.

```
EEGNet(
  (firstconv): Sequential(
    (0): Conv2d(1, 16, kernel_size=(1, 51), stride=(1, 1), padding=(0, 25), bias=False)
    (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (depthwiseConv): Sequential(
    (0): Conv2d(16, 32, kernel_size=(2, 1), stride=(1, 1), groups=16, bias=False)
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ELU(alpha=1.0)
    (3): AvgPool2d(kernel_size=(1, 4), stride=(1, 4), padding=0)
    (4): Dropout(p=0.25, inplace=False)
  )
  (separableConv): Sequential(
    (0): Conv2d(32, 32, kernel_size=(1, 15), stride=(1, 1), padding=(0, 7), bias=False)
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ELU(alpha=1.0)
    (3): AvgPool2d(kernel_size=(1, 8), stride=(1, 8), padding=0)
    (4): Dropout(p=0.25, inplace=False)
  )
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (classify): Linear(in_features=736, out_features=2, bias=True)
)
```

All the arguments settings in different layers are left the same as those described in the lab spec.

Also, the inner structures of `firstconv`, `depthwiseConv`, `separableConv` and `classify` blocks are left unchanged as well.

The only difference is that a *flatten* layer was added between convolution blocks and fully-connected blocks. Finally, the activation function layers are interchangeable. The default one is **ELU**.

◆ DeepConvNet

The DeepConvNet is in fact, just a simple deep convolution neural network (CNN).

This is the structure of DeepConvNet, where

$C = 2, T = 750, N = 2$ in this implementation.

The max norm term is neglected here. The activation

Layer	# filters	size	# params	Activation	Options
Input		(C, T)			
Reshape		(1, C, T)			
Conv2D	25	(1, 5)	150	Linear	mode = valid, max norm = 2
Conv2D	25	(C, 1)	$25 * 25 * C + 25$	Linear	mode = valid, max norm = 2
BatchNorm			$2 * 25$		epsilon = 1e-05, momentum = 0.1
Activation				ELU	
MaxPool2D		(1, 2)			
Dropout					p = 0.5
Conv2D	50	(1, 5)	$25 * 50 * C + 50$	Linear	mode = valid, max norm = 2
BatchNorm			$2 * 50$		epsilon = 1e-05, momentum = 0.1
Activation				ELU	
MaxPool2D		(1, 2)			
Dropout					p = 0.5
Conv2D	100	(1, 5)	$50 * 100 * C + 100$	Linear	mode = valid, max norm = 2
BatchNorm			$2 * 100$		epsilon = 1e-05, momentum = 0.1
Activation				ELU	
MaxPool2D		(1, 2)			
Dropout					p = 0.5
Conv2D	200	(1, 5)	$100 * 200 * C + 200$	Linear	mode = valid, max norm = 2
BatchNorm			$2 * 200$		epsilon = 1e-05, momentum = 0.1
Activation				ELU	
MaxPool2D		(1, 2)			
Dropout					p = 0.5
Flatten					
Dense	N			softmax	max norm = 0.5

The following image shows the architecture of my DeepConvNet.

```

DeepConvNet(
  (conv_in): Conv2d(1, 25, kernel_size=(1, 5), stride=(1, 1))
  (conv_1): Sequential(
    (0): Conv2d(25, 25, kernel_size=(2, 1), stride=(1, 1))
    (1): BatchNorm2d(25, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ELU(alpha=1.0)
    (3): MaxPool2d(kernel_size=(1, 2), stride=(1, 2), padding=0, dilation=1, ceil_mode=False)
    (4): Dropout(p=0.5, inplace=False)
  )
  (conv_2): Sequential(
    (0): Conv2d(25, 50, kernel_size=(1, 5), stride=(1, 1))
    (1): BatchNorm2d(50, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ELU(alpha=1.0)
    (3): MaxPool2d(kernel_size=(1, 2), stride=(1, 2), padding=0, dilation=1, ceil_mode=False)
    (4): Dropout(p=0.5, inplace=False)
  )
  (conv_3): Sequential(
    (0): Conv2d(50, 100, kernel_size=(1, 5), stride=(1, 1))
    (1): BatchNorm2d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ELU(alpha=1.0)
    (3): MaxPool2d(kernel_size=(1, 2), stride=(1, 2), padding=0, dilation=1, ceil_mode=False)
    (4): Dropout(p=0.5, inplace=False)
  )
  (conv_4): Sequential(
    (0): Conv2d(100, 200, kernel_size=(1, 5), stride=(1, 1))
    (1): BatchNorm2d(200, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ELU(alpha=1.0)
    (3): MaxPool2d(kernel_size=(1, 2), stride=(1, 2), padding=0, dilation=1, ceil_mode=False)
    (4): Dropout(p=0.5, inplace=False)
  )
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (dense): Linear(in_features=8600, out_features=2, bias=True)
)

```

All activation function layers are interchangeable. The default one is **ELU**.

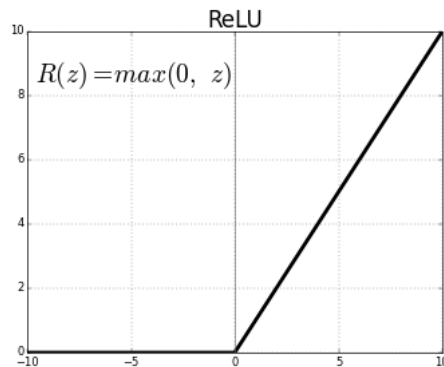
Aside from `conv_in` block, all other convolution blocks from `conv_1` to `conv_4` have the same inner structure. (Of course, they have different arguments settings though.)

Convolution → Batch Normalization → Activation Function → Maximum Pooling → Dropout

B. Explanation of the Activation Function

★ ReLU

The **rectified linear unit** function, or **ReLU** in abbreviation, looks like this

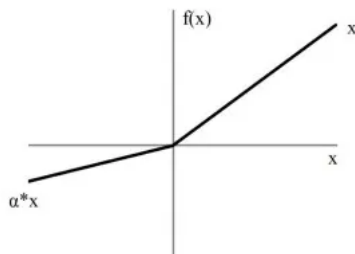


If the input z is greater than 0, the value will be output without any change. Otherwise, 0 will be output. The mathematical representation of ReLU function is:

$$ReLU(z) = \max(0, z)$$

★ LeakyReLU

The **leaky ReLU** function is a variant of ReLU function. It looks like this:

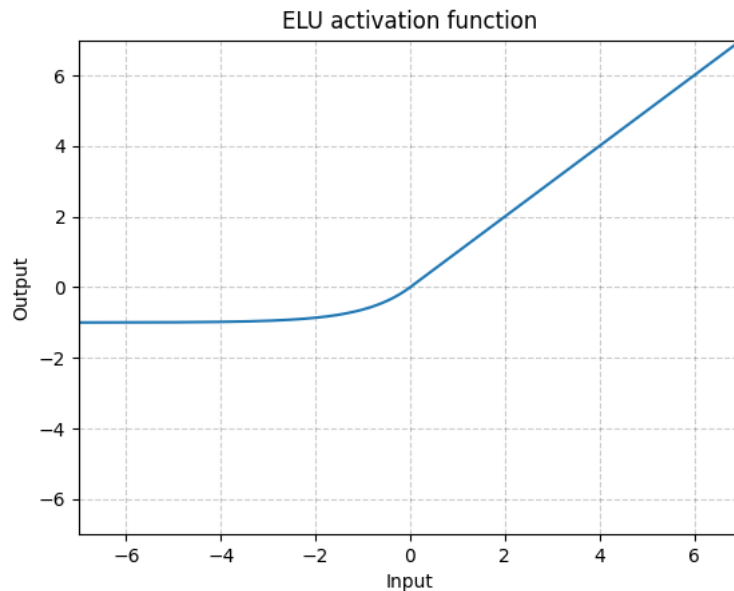


The negative input will be output as α times the input. By default, $\alpha = 0.01$, and we don't change this value. The mathematical representation of Leaky ReLU function is:

$$LeakyReLU(z) = \max(0, z) + \min(0, \alpha z)$$

★ ELU

The **exponential linear unit** function, or **ELU** in short, can also be considered a variant of ReLU function. It looks like this:



The mathematical representation of ELU function is:

$$ELU(z) = \max(0, z) + \min(0, \alpha(e^z - 1))$$

The α value is set to 1.0 by default. The setting is remained the same in my code implementation.

Under the PyTorch framework, simply typing

```
1 import torch.nn as nn
2 nn.ReLU()
3 nn.LeakyReLU()
4 nn.ELU()
```

can conveniently implement these activation functions.

Again, the default activation function in both my EEGNet and DeepConvNet is **ELU**.

Experimental Results

A. The Highest Testing Accuracy

EEGNet

```
The testing accuracy of the model using ELU is: 82.31%
The testing accuracy of the model using LeakyReLU is: 87.31%
The testing accuracy of the model using ReLU is: 86.30%
```

DeepConvNet

```
The testing accuracy of the model using ELU is: 81.57%
The testing accuracy of the model using LeakyReLU is: 81.30%
The testing accuracy of the model using ReLU is: 82.96%
```

Test Acc	ReLU	Leaky ReLU	ELU
EEGNet	86.30%	👑 87.31%	82.31%
DeepConvNet	82.96%	81.30%	81.57%

The above table of test accuracy shows that the best combination which generates the highest test accuracy of **87.31%** is the **EEGNet** using **Leaky ReLU** activation function.

Besides, one can see that no matter what activation function is used, **EEGNet outperforms DeepConvNet** under the same experimental setting.

Anything You Want to Share


For reference, I will also put on the highest training accuracy of these 6 combinations.

EEGNet

```
The maximum training accuracy of the model using ELU is: 98.52%
The maximum testing accuracy of the model using ELU is: 82.31%
The maximum training accuracy of the model using ReLU is: 98.80%
The maximum testing accuracy of the model using ReLU is: 86.30%
The maximum training accuracy of the model using LeakyReLU is: 98.70%
The maximum testing accuracy of the model using LeakyReLU is: 87.31%
```

DeepConvNet

```
The maximum training accuracy of the model using ELU is: 97.96%
The maximum testing accuracy of the model using ELU is: 81.57%
The maximum training accuracy of the model using ReLU is: 94.81%
The maximum testing accuracy of the model using ReLU is: 82.96%
The maximum training accuracy of the model using LeakyReLU is: 94.54%
The maximum testing accuracy of the model using LeakyReLU is: 81.30%
```

Train Acc	ReLU	Leaky ReLU	ELU
EEGNet	 98.80%	98.70%	98.52%
DeepConvNet	94.81%	94.54%	97.96%

During the training session, one can see that EEGNet still defeats DeepConvNet, no matter what activation function is used. However, the differences between them are not that much, compared to the differences in testing session. The best combination is **EEGNet** with **ReLU**, although the accuracy of leaky ReLU is just slightly lower.

Another thing I want to share is about how I store the model weights with the highest accuracy. In the code snippet of the `training()` function, I added this part:

```
1   for e in range(epochs):
2       .....
3       '''testing session'''
4       accuracy = testing(model, test_dataloader, device)
5       .....
6       if accuracy > best_test_acc:
7           best_test_acc = accuracy
8           best_model_weights = copy.deepcopy(model.state_dict())
```


So, in every epoch, I checked whether the current model has the highest test accuracy. If so, update the highest record of test accuracy and the best model weights.

However, originally I wrote this line:

```
1 best_model_weights = model.state_dict()
```

instead of this line:

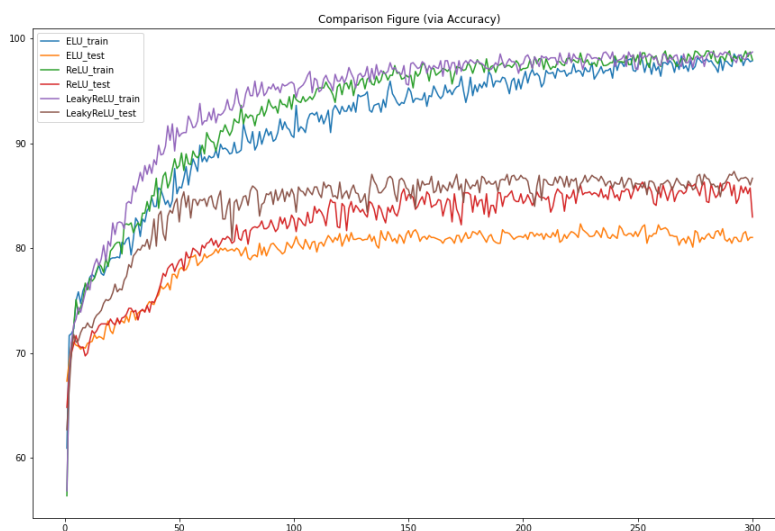
```
1 best_model_weights = copy.deepcopy(model.state_dict())
```

and find out that the model weights stored are **not** the ones with the highest accuracy, but the ones in the last epoch (epoch = 300).

I was so confused and was struggling to debug, until I found out that the weights stored in `best_model_weights` are from the variable `model` which is the **current** model. The model will be updated in every epoch, which indirectly influences the weights stored in `best_model_weights`. This kind of **reference problem** happens a lot in Python. The solution is simply applying **deep copy**, which can be done by calling `deepcopy()` function from the package `copy`.

B. Comparison figures

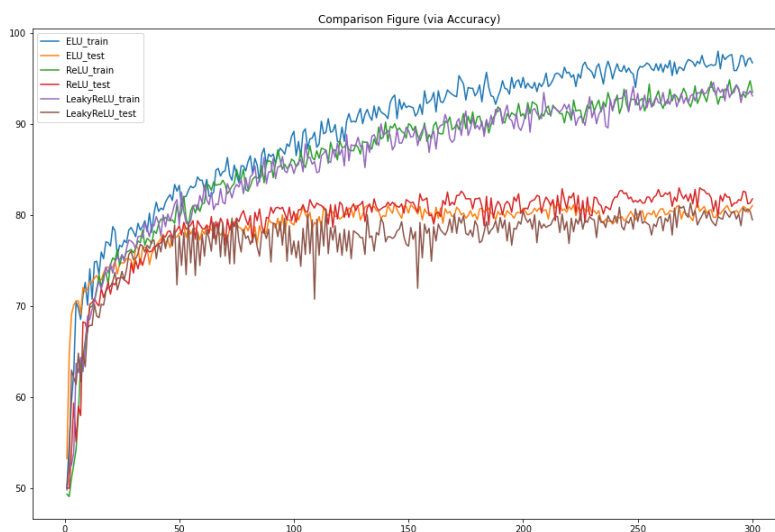
◆ EEGNet



One can see that **Leaky ReLU** is the best activation function to the **EEGNet** model. The model with leaky ReLU gains the best accuracy among the 3 activation functions in both training and testing session.

On the other, the model with **ELU** performs worse than the models with the other activation functions, especially during the testing session.

◆ DeepConvNet



As for the **DeepConvNet** models, interestingly, the best performed models are not the same in training session and in testing session. During the **training** session, the best model is the one with **ELU** function, while during the **testing** session, the best model is the one with **ReLU** function.

Nevertheless, I think we should focus on the testing accuracy, since high training accuracy with low testing accuracy might be a sign of *overfitting*. Thus, I will argue that **ReLU** is more suitable when using the **DeepConvNet** model.

Discussion

The experiment results show that **EEGNet outperforms DeepConvNet**, no matter which one of the 3 activation functions is used.

I believe that the outcomes are on account of the ***Depthwise-separable convolution*** applied in the network structure of EEGNet.

Depthwise convolution and **pointwise convolution** both have their specialties regarding extracting features and summarizing.

Depthwise convolution can ***learn a temporal summary for each feature map individually***.

These summarized features of all feature maps are then passed to the pointwise convolution.

Pointwise convolution can ***learn how to optimally mix the feature maps together***, i.e. how to distribute the weighting for each maps.

Therefore, these kinds of data with several temporal features are especially suitable for **depthwise-separable-convolution-based** CNN models! In my code implementation, I do find EEG models surpass ordinary CNN models (DeepConvNet) under all 3 activation functions, namely *ELU*, *ReLU* and *leaky ReLU*.