Lab3 : EEG Classification

資訊工程學系
學號:0816095
王軒

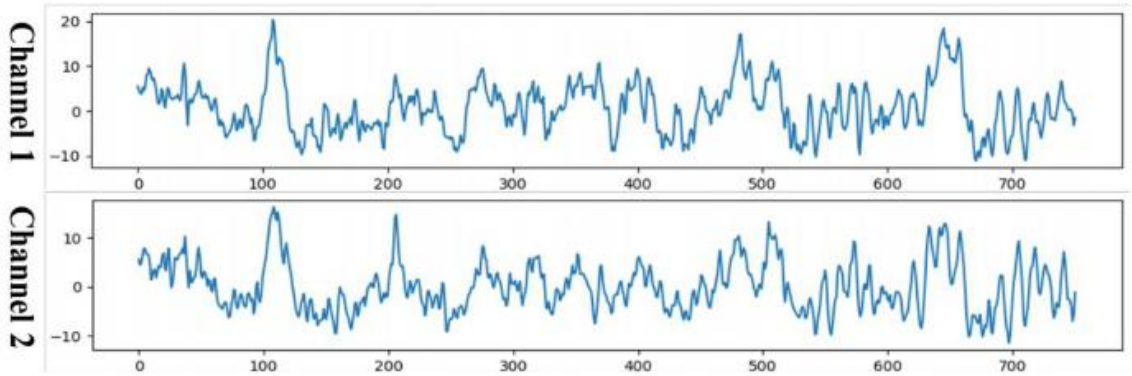
Table of Contents

1. Introduction

A. Problem definition

In this lab, I implemented two types of Model, EEGNet and DeepConvNet and tried a few different activation function and hyper parameters and compare the difference between them.

The goal is to accurately classify the BCI competition dataset, which contains two signals waves



B. Experiment environments and file usage

B.1. Experiment environments

I used Pytorch、Numpy to build the models, Matplotlib for graphing, and some other auxiliary package for this lab.

B.2. File usage

There are a total of 8 .py source code. They are ”dataloader.py”、” EEGNet_training_ELU.py”、” EEGNet_training_ReLU.py”、” EEGNet_training_LeakyReLU.py”、” DeepConvNet_training_ELU.py”、” DeepConvNet_training_ReLU.py”、” DeepConvNet_training_LeakyReLU.py”、”models.py” .

B.2.1. dataloader.py

This is the same code received from the homework. No modification was made to it. It’s purpose is to load BCI dataset.

B.2.2. models.py

All 6 models are defined in this file and loaded to the training files if needed.

B.2.3. All 6 training files

They are all used for training models respective to its name. It should be placed under the same directory as dataloader.py and models.py. When training it will automatically save the best results.

2. Experiment setup

A. Model detail

A.1. EEGNet

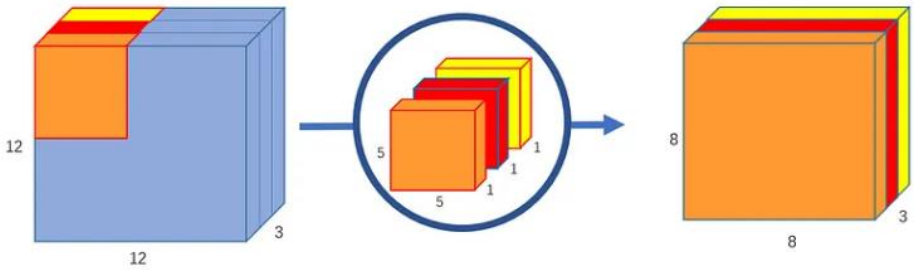
I used Pytorch to build to model. All the implementation details are according to spec, which is a depth wise separable convolution model. There are a total of four layers: first convolution layer, depth wise convolution layer, separable convolution layer, fully connected layer.

First convolution layer:

Simply do a convolution followed with a batch normalization.

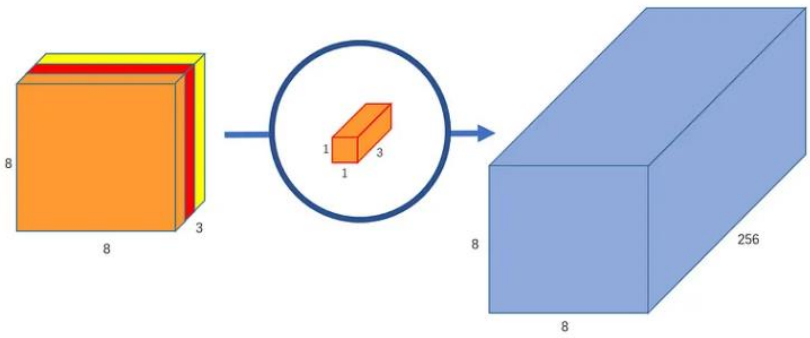
Depth wise convolution:

In depth wise convolution, what we are trying to do is to convolute each channel of the input respectively then combine them together like in the picture below which has a three-channel input (In our case we have a 2 channel input, thus kernel_size=(2, 1)).



Separable convolution layer:

In this layer we combine do a point wise convolution and combine all three convolution outputs.



Fully connected layer:

This is just a simple fc layer that determines the output based on the feature extracted by the previous convolution.

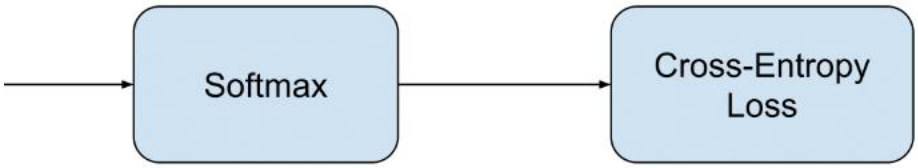
A.2. DeepConvNet

I used Pytorch to build to model. All the implementation details are according to spec. This model is much deeper and has a lot more parameters thus is harder to train than EEGNet.

A.3. Loss function

I chose cross entropy loss as my loss function as it performs better then other loss function that I've tried.

Because I'm using cross entropy loss there is no need to use softmax function after the full connected layer.
torch.nn.functional.CrossEntropyLoss() computes the softmax operation and the cross entropy operation for us.



$$f(s)_i = \frac{e^{s_i}}{\sum_j^C e^{s_j}} \quad CE = -\sum_i^C t_i log(f(s)_i)$$

A.4. Optimizer and scheduler

I chose RMSprop as my optimizer as it performs better then Adam optimizer, and it's great for complex non-convex error surface.

Although RMSprop can adjust learning rate by itself, it still needs a general learning rate. For this I used the MultiStepLR() to customize the learning rate how ever I want.

EEGNets converge much faster and easier than DeepConvNets, and through experiment, I halved the learning rate at 400, 500, 700 epochs, and it worked well.

```
MultiStepLR(optimizer, milestones=[400,500,700], gamma=0.5)
```

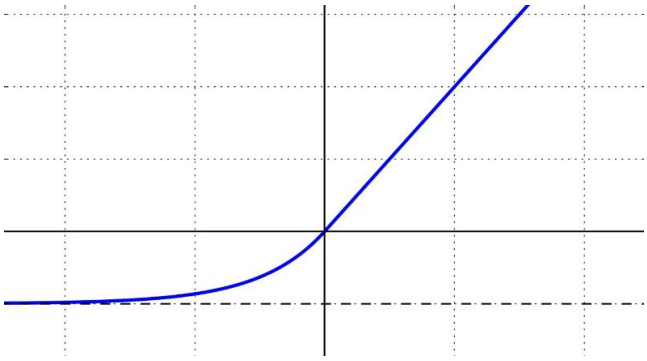
DeepConvNets are a lot harder to converge, I tried many scheduler milestones and eventually settled with 600, 1200.

B. Activation functions

B.1. ELU

Needs to set an alpha value. Gives the model nonlinearity, but still leaves some room for negative values. It's more computationally intensive then ReLU.

$$f(x) = \begin{cases} x, & x \geq 0 \\ \alpha(\exp(x) - 1), & x < 0 \end{cases}$$



B.2. ReLU

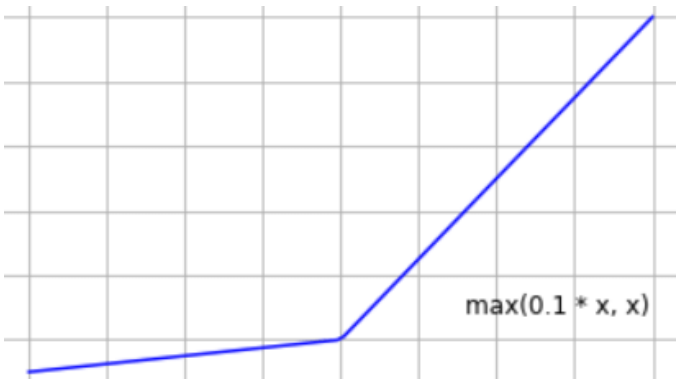
No need to set any parameters. Gives the model nonlinearity. Every thing smaller then 0 will be replaced with 0. Very computationally efficient and widely used.

$$f(x) = \max(0, x)$$

B.3. LeakyReLU

Needs to set the negative slope value. Gives the model nonlinearity. Very similar to ReLU function, instead of zeroing out all the negative value, it "leaks" some value determined by the negative slop, hence the name "LeakyReLU".

$$\text{LeakyReLU}(x) = \max(0, x) + \text{negative\_slope} * \min(0, x)$$



3. Experimental results

A. Highest testing accuracy

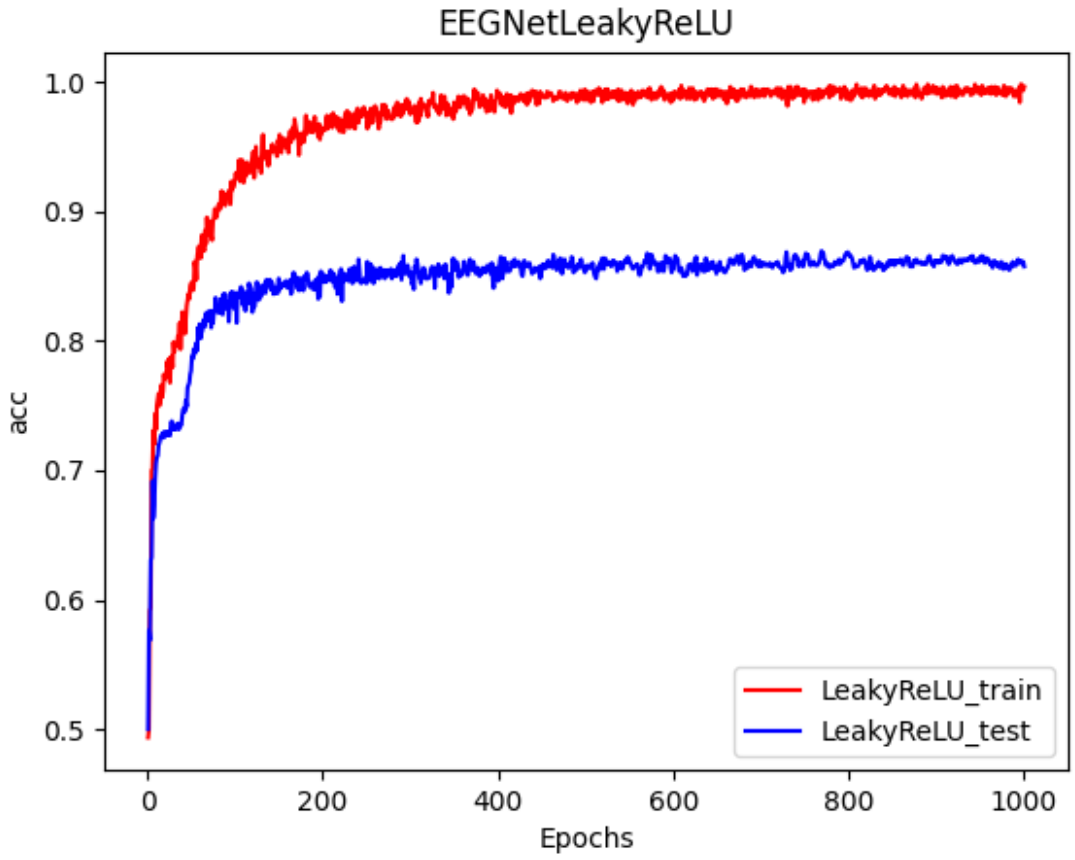
Because the model spec is fixed, I mainly focused on trying different activation functions and hyperparameters to improve the accuracy. Below are the best results that I have obtained.

	Activation function	Loss function	Optimizers	Testing accuracy
EEGNet	LeakyReLU (alpha=0.06)	CrossEntropyLoss	RMSprop (lr=1e-3, momentum=0.6)	87.77%
DeepConvNet	ReLU	CrossEntropyLoss	RMSprop (lr=1e-3, momentum=0.9)	85.73%

A.1. EEGNet screenshots

```
class EEGNetLeakyReLU(nn.Module):
    def __init__(self):
        super(EEGNetLeakyReLU, self).__init__()
        self.firstConv = nn.Sequential(
            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.depthwiseConv = nn.Sequential(
            nn.Conv2d(16, 32, kernel_size=(2,1), stride=(1,1), groups=8,bias=False),
            nn.BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True),
            nn.LeakyReLU(negative_slope=0.06),
            nn.AvgPool2d(kernel_size=(1,4), stride=(1,4),padding=0),
            nn.Dropout(p=0.25)
        )
        self.separableConv = nn.Sequential(
            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.LeakyReLU(negative_slope=0.06),
            nn.AvgPool2d(kernel_size=(1,8), stride=(1,8),padding=0),
            nn.Dropout(p=0.25)
        )
        self.classify = nn.Sequential(
            nn.Flatten(),
            nn.Linear(736,2,bias=True)
        )

    def forward(self, x):
        out = self.firstConv(x)
        out = self.depthwiseConv(out)
        out = self.separableConv(out)
        out = self.classify(out)
        return out
```

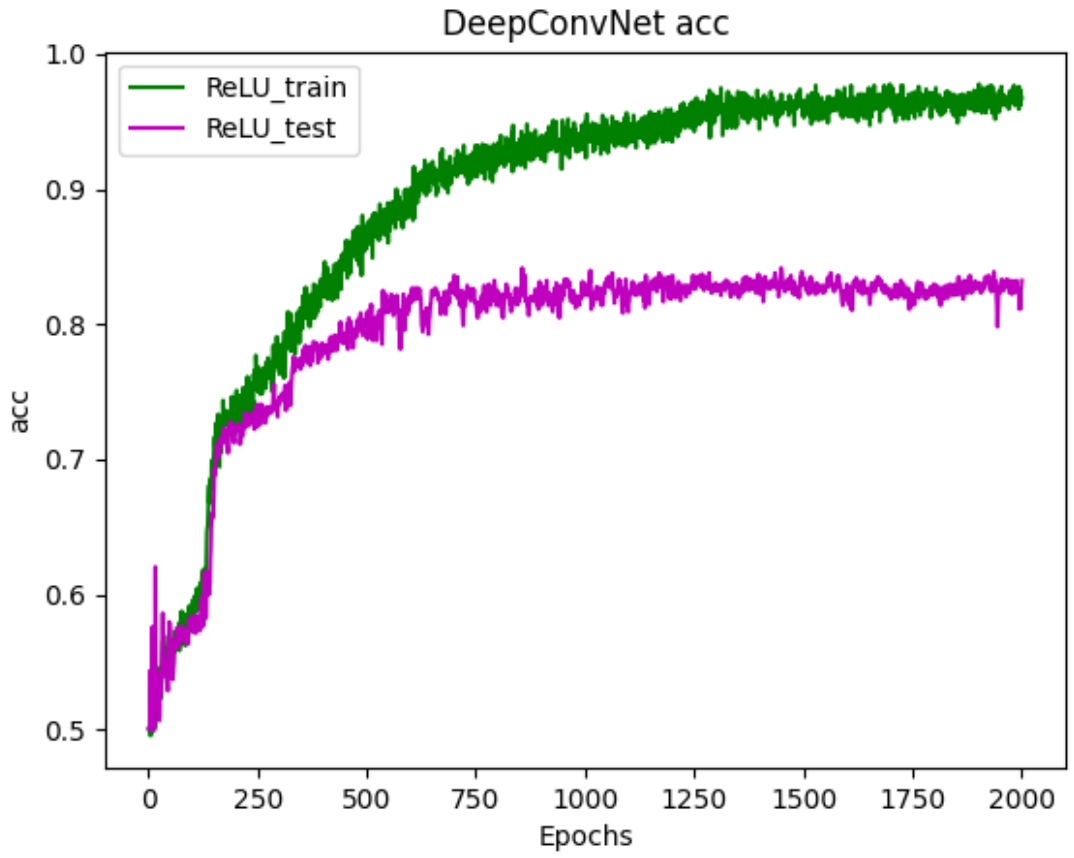
```
epochs = 1000
lr = 1e-3
LeakyReLU_max_val_acc = 0
save_model = True
model = EEGNetLeakyReLU()
criterion = nn.CrossEntropyLoss()
optimizer = optim.RMSprop(model.parameters(),lr = lr, momentum = 0.6)
scheduler = optim.lr_scheduler.MultiStepLR(optimizer, milestones=[400,500,700], gamma=0.5)
```



A.2. DeepConvNet screenshots

```
class DeepConvNetReLU(nn.Module):
    def __init__(self):
        super(DeepConvNetReLU, self).__init__()
        self.model = nn.Sequential(
            nn.Conv2d(1, 25, kernel_size=(1,5), padding='valid'),
            nn.Conv2d(25, 25, kernel_size=(2,1), padding='valid'),
            nn.BatchNorm2d(25, eps=1e-05, momentum=0.1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(1,2)),
            nn.Dropout(p=0.5),

            nn.Conv2d(25, 50, kernel_size=(1,5), padding='valid'),
            nn.BatchNorm2d(50, eps=1e-05, momentum=0.1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(1,2)),
            nn.Dropout(p=0.5),

            nn.Conv2d(50, 100, kernel_size=(1,5), padding='valid'),
            nn.BatchNorm2d(100, eps=1e-05, momentum=0.1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(1,2)),
            nn.Dropout(p=0.5),

            nn.Conv2d(100, 200, kernel_size=(1,5), padding='valid'),
            nn.BatchNorm2d(200, eps=1e-05, momentum=0.1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(1,2)),
            nn.Dropout(p=0.5),

            nn.Flatten(),
            nn.Linear(8600,2,bias=True)
        )

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

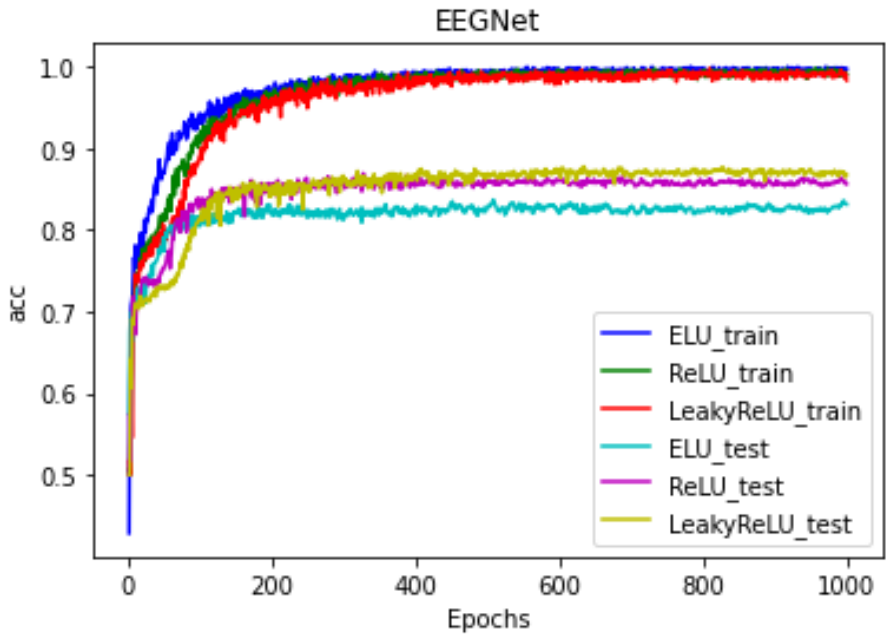
```
epochs = 2000
lr = 1e-3
ReLU_max_val_acc = 0
save_model = True
model = DeepConvNetReLU()
criterion = nn.CrossEntropyLoss()
optimizer = optim.RMSprop(model.parameters(),lr = lr, momentum = 0.9, weight_decay=1e-3)
scheduler = optim.lr_scheduler.MultiStepLR(optimizer, milestones=[600, 1200], gamma=0.5)
```



B. Comparison figures

B.1. EEGNet with different activation function

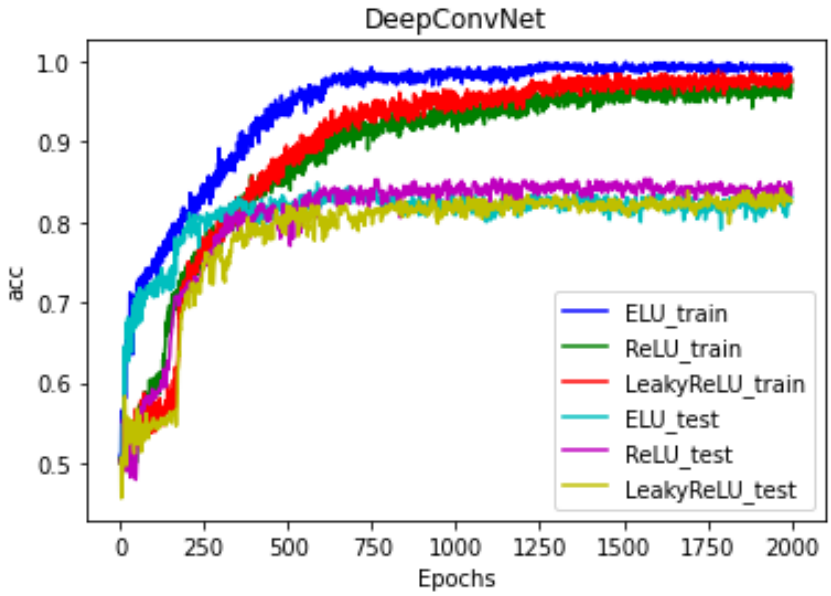
From the figure we can see that although with ELU activation function it has the best training accuracy, testing accuracy ELU is the worst out of the three which means that it overfits more easily. LeakyReLU on the other hand has the lowest training accuracy, but performs best when testing.



Activation function	Testing accuracy
ELU	83.70%
ReLU	86.94%
LeakyReLU	87.77%

B.2. DeepConvNet with different activation function

From the figure we can see that same as EEGNet, ELU activation function tends to overfit. But this time LeakyReLU performs worst when testing, and ReLU performs the best.



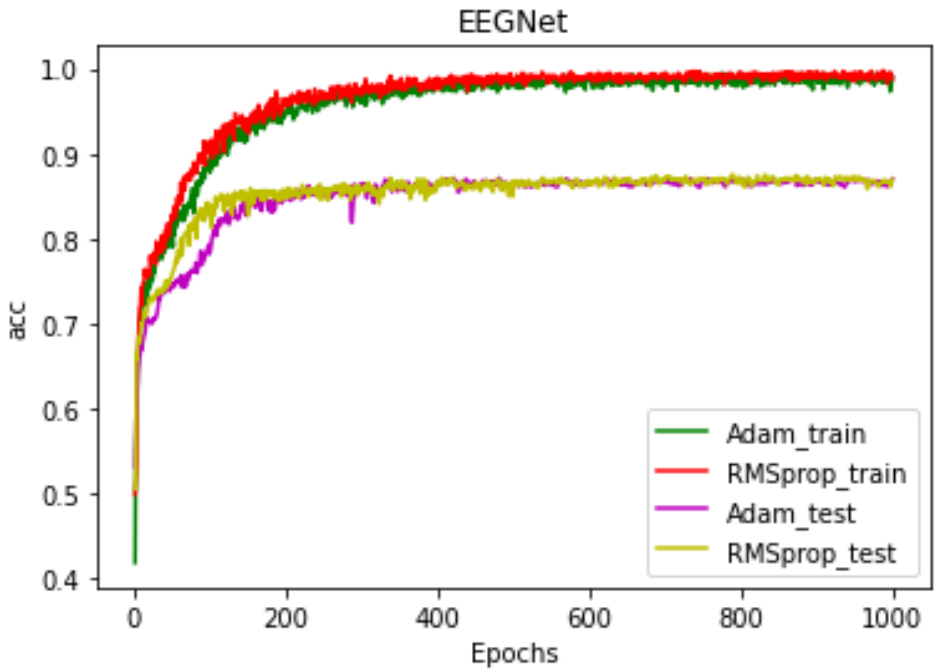
Activation function	Testing accuracy
ELU	84.90%
ReLU	85.73%
LeakyReLU	84.16%

4. Discussion

In this section I am going to compare the effects of different hyperparameters using the best performing model: EEGNet with LeakyReLU.

A. Discussion between optimizers

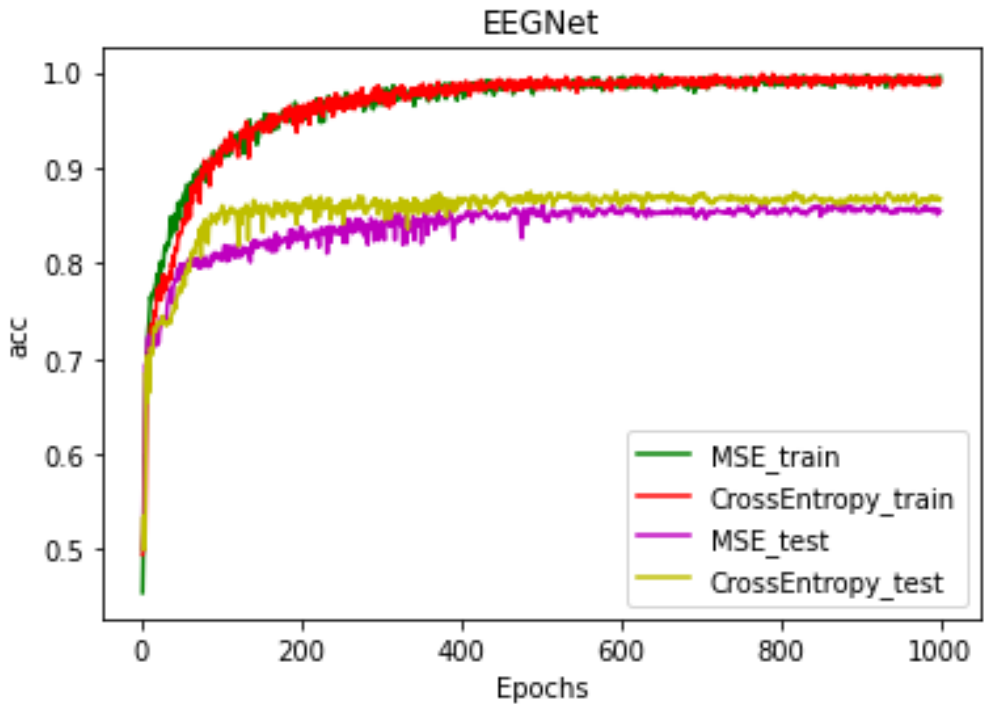
We can see that RMSprop performs slightly better than Adam, thus I chose RMSprop as my optimizer.



Optimizers	Testing accuracy
Adam	87.12%
RMSprop	87.50%

B. Discussion between loss functions

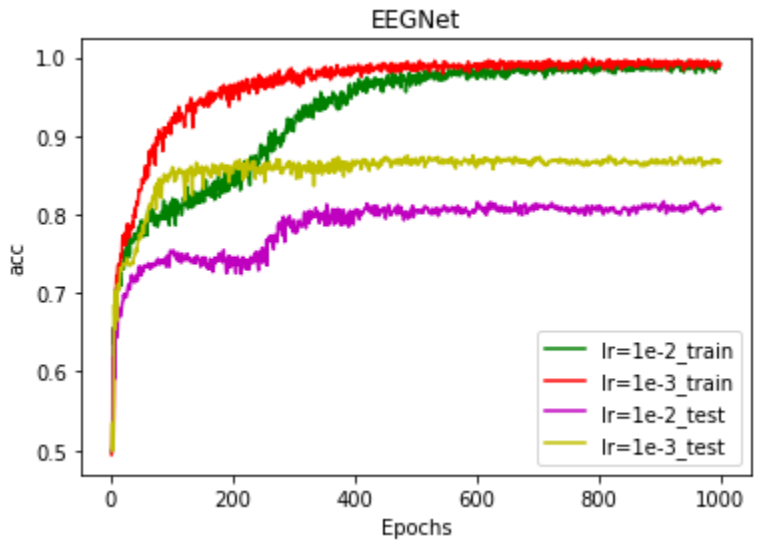
Because of the way MSE calculates loss, I must change the shape of the target from (1080,) to one hot encoding of shape (1080, 2). And from the figure we can see that Cross entropy are superior for classification problems like this.



Loss function	Testing accuracy
MSE	86.11%
CrossEntropy	87.50%

C. Discussion between learning rates

As we can see in the figure, bigger learning rate doesn't mean faster learning. Learning rate of 1e-3 vastly out performs 1e-2, as learning rate of 1e-2 finds it hard to converge.



Learning rate	Testing accuracy
1e-2	81.66%
1e-3	87.50%