Lab3 report

學號：311512049 姓名：陳緯翰

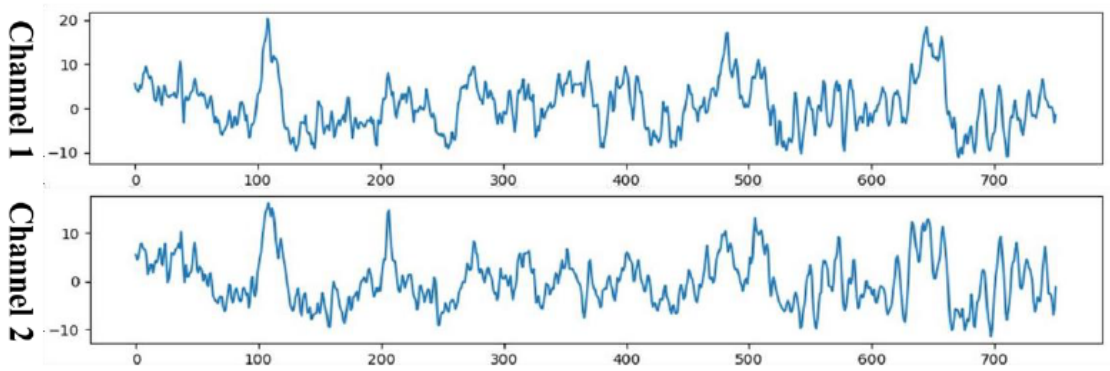
Introduction (20%)

a. Lab Objective

- 建構 EEGNET 以及 DeepConvNet 來實現 EEG 訊號分類模型
- 利用不同的 Activation function (Relu,Leaky Relu,ELY)等並觀察結果
- 將結果可視化並秀出最高準確率之結果(>87%)

b. Dataset

- 此資料集有兩個通道，每個通道各有 750 個 data point ，兩個 label 輸出分別代表球掉落到左手邊與右手邊
- 資料集已經被整理成[S4b_train.npz, X11b_train.npz] and [S4b_test.npz, X11b_test.npz]，並透過 dataloader.py 存取進來



Experiment set up (20%)

A. The detail of your model

- Data

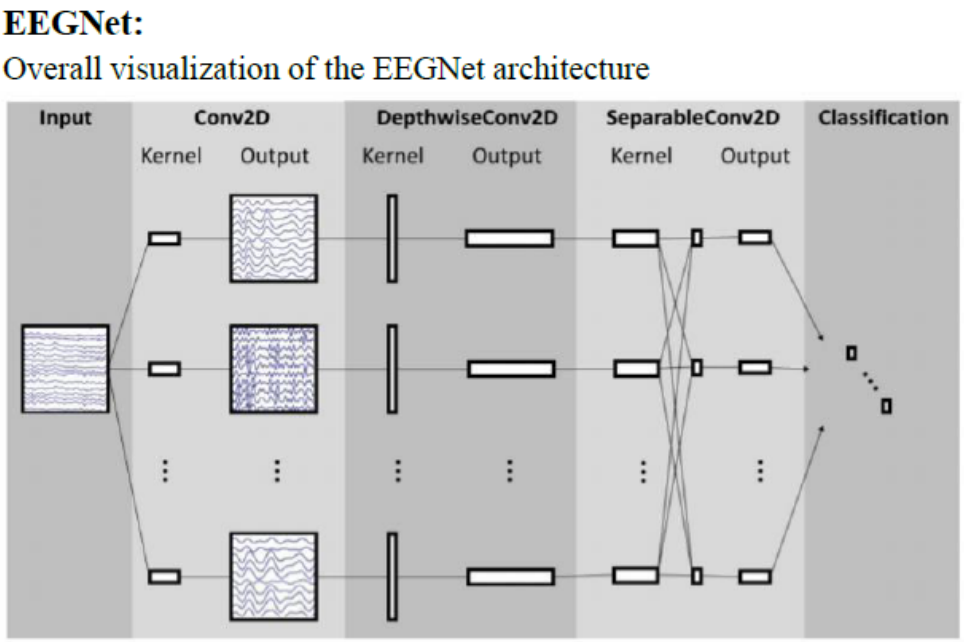
  將 `train_data` 轉換為 `torch.Tensor` 類型的數據，接著使用 `.float()` 方法將數據類型轉換為浮點數。然後，使用 `.cuda()` 方法將 `train_data` 從 CPU 移動到 GPU 上進行加速計算。

```
# import data
train_data, train_label, test_data, test_label = dataloader.read_bci_data()

# convert numpy to 'cuda' torch tensor
train_data = torch.from_numpy(train_data).float().cuda()   # torch.Size([1080, 1, 2, 750])
train_label = torch.from_numpy(train_label).long().cuda()
test_data = torch.from_numpy(test_data).float().cuda()     # torch.Size([1080, 1, 2, 750])
test_label = torch.from_numpy(test_label).long().cuda()
```

接著對訓練數據進行shuffle，以增加訓練效果的隨機性，以此來得到更好的準確率。

```
# shuffle
permutation = torch.randperm(train_data.size()[0])
train_data = train_data[permutation]
train_label = train_label[permutation]
```

- EEGNet



最初的 Conv2D 用來提取訊號的特徵，DepthwiseConv2D 將 multi channel 整理成一個 channel ，最後 SeperableConv2D 從 DepthwiseConv2D 裡面提取特徵

EEGNet implementation details:

```
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)
  )
  (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)
  )
  (classify): Sequential(
    (0): Linear(in_features=736, out_features=2, bias=True)
  )
)
```

程式實現：

```
class EEGNet(nn.Module):
    def __init__(self, mode):
        super(EEGNet, self).__init__()
        if mode == "ELU":
            activate_func = nn.ELU(alpha=1.0, inplace=True)
        elif mode == "ReLU":
            activate_func = nn.ReLU(inplace=True)
        elif mode == "LeakyReLU":
            activate_func = nn.LeakyReLU(negative_slope=0.01, inplace=True)

        self.conv1 = 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.conv2 = nn.Sequential(
            nn.Conv2d(16, 32, (2, 1), stride=(1, 1), bias=False),
            nn.BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True),
            activate_func,
            nn.AvgPool2d(kernel_size=(1, 4), stride=(1, 4), padding=0),
            nn.Dropout(p=0.25)
        )
        self.conv3 = 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),
            activate_func,
            nn.AvgPool2d(kernel_size=(1, 8), stride=(1, 8), padding=0),
            nn.Dropout(p=0.25)
        )
        self.fc1 = nn.Sequential(
            nn.Linear(in_features=736, out_features=2, bias=True)
        )
    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)
        x = x.view(x.size(0), -1)
```

```
        x = self.fc1(x)
        return x
```

- DeepConvNet

DeepConvNet:

You need to implement the DeepConvNet architecture by using the following table, where C = 2, T = 750 and N = 2. **The max norm term is ignorable.**

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

程式實現：

```
class DeepConvNet(nn.Module):
    def __init__(self, mode):
        super(DeepConvNet, self).__init__()
        if mode == "ELU":
            activate_func = nn.ELU(alpha=1.0, inplace=True)
        elif mode == "ReLU":
            activate_func = nn.ReLU(inplace=True)
        elif mode == "LeakyReLU":
            activate_func = nn.LeakyReLU(negative_slope=0.01, inplace=True)

        self.conv1 = nn.Sequential(
            nn.Conv2d(1, 25, (1, 5), stride=(1, 1), padding=(0, 2), bias=True),  # 因為table上有bias，所以這邊也要有
            # second convolutional layer
            nn.Conv2d(25, 25, (2, 1), stride=(1, 1), padding=(0, 0), bias=True),
            nn.BatchNorm2d(25, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True),
            activate_func,
            nn.AvgPool2d(kernel_size=(1, 2), stride=(1, 2), padding=0),
            nn.Dropout(p=0.5)
        )
        self.conv2 = nn.Sequential(
```

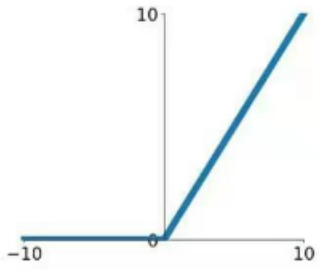
```
            nn.Conv2d(25, 50, (1, 5), stride=(1, 1), padding=(0, 2), bias=True),
            nn.BatchNorm2d(50, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True),
            activate_func,
            nn.AvgPool2d(kernel_size=(1, 2), stride=(1, 2), padding=0),
            nn.Dropout(p=0.5)
        )
        self.conv3 = nn.Sequential(
            nn.Conv2d(50, 100, (1, 5), stride=(1, 1), padding=(0, 2), bias=True),
            nn.BatchNorm2d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True),
            activate_func,
            nn.AvgPool2d(kernel_size=(1, 2), stride=(1, 2), padding=0),
            nn.Dropout(p=0.5)
        )
        self.conv4 = nn.Sequential(
            nn.Conv2d(100, 200, (1, 5), stride=(1, 1), padding=(0, 2), bias=True),
            nn.BatchNorm2d(200, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True),
            activate_func,
            nn.AvgPool2d(kernel_size=(1, 2), stride=(1, 2), padding=0),
            nn.Dropout(p=0.5)
        )
        self.fc = nn.Sequential(
            nn.Linear(in_features=9200, out_features=2, bias=True)
        )

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)
        x = self.conv4(x)
        x = x.view(x.size(0), -1)
        # print(x.size())
        output = self.fc(x)
        return output
```

B. Explain the activation function (ReLU, Leaky ReLU, ELU)

- ReLU(Rectified Linear Unit)

  ReLU函數將小於零的值設置為零，大於等於零的值保持不變。

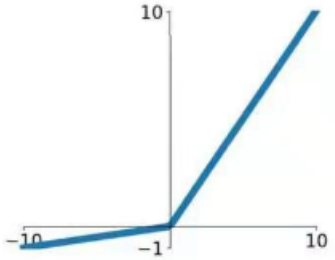
  $f(x) = max(0, x)$



- Leaky ReLU

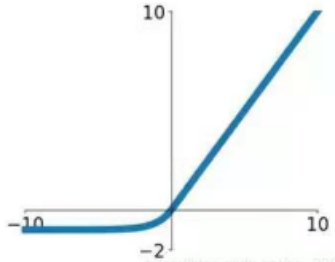
  Leaky ReLU函數是對ReLU函數的改進，當輸入小於零時不再是硬性將其截斷為零，而是對輸入進行輕微的線性修正。

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



- ELU (Exponential Linear Unit)
  ELU函數將小於零的值設置為一個接近於零的值，以解決ReLU函數在輸入小於零時導致的稀疏激活問題。

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



在下面的表格中，我們總結了這三種激活函數的優點和缺點：

激活函數	**優點**	**缺點**
ReLU	實現簡單、計算快速	在輸入小於零時，導致稀疏激活問題，導致部分神經元死亡。
Leaky ReLU	解決了ReLU的死亡神經元問題	在輸入小於零時仍然可能導致部分神經元死亡。
ELU	在輸入小於零時能夠生成非零輸出，不會導致死亡神經元問題。	實現相對複雜，計算較慢。

Experimental results (30%)

A. The highest testing accuracy

	ELU	ReLU	Leaky ReLU
EEGNet	84.26%	**87.31%**	86.39%
DeepConvNet	76.85%	79.72%	80.56%

```
pp037@ec037:~/2023-DL-lesson/lab3$ python3 EEGnet.py
Running on the GPU
NVIDIA GeForce GTX 1060 6GB
Best ELU accuracy : 84.26 %
ELU finish
Best ReLU accuracy : 87.31 %
ReLU finish
Best LeakyReLU accuracy : 86.39 %
LeakyReLU finish
Finish EEGNet all
```

```
pp037@ec037:~/2023-DL-lesson/lab3$ python3 DeepConvNet.py
Running on the GPU
NVIDIA GeForce GTX 1060 6GB
Best ELU accuracy : 76.85 %
ELU finish
Best ReLU accuracy : 79.72 %
ReLU finish
Best LeakyReLU accuracy : 80.56 %
LeakyReLU finish
Finish DeepConvNet all
```

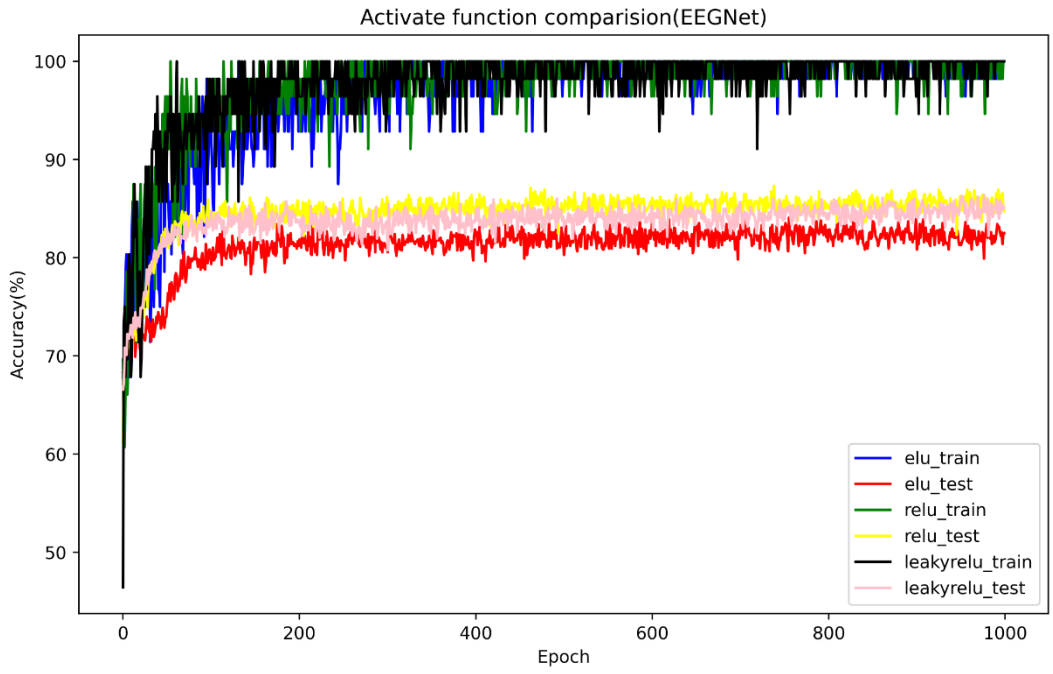
- Screenshot with two models

  見上方

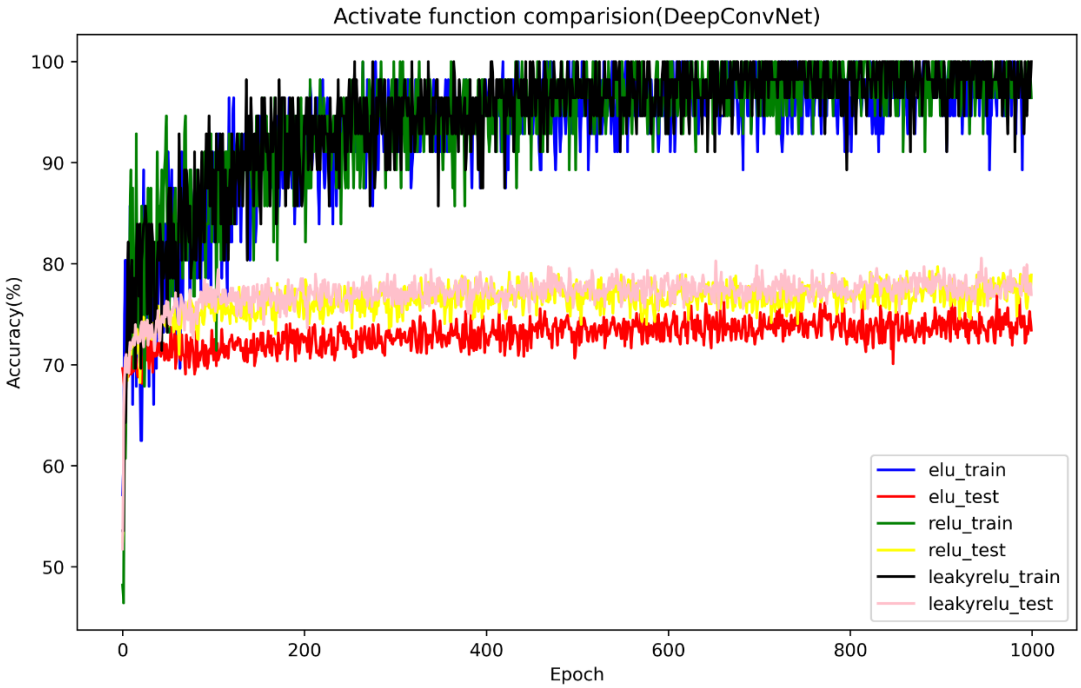
B. Comparison figures

皆用epoch=1000, batch_size=256, learning_rate=1e-03來完成以下結果

- EEGNet



- DeepConvNet



Discussion (30%)

Anything you want to share

Batch size大小對訓練的影響

Batch size 的大小對模型的訓練有很大的影響，主要體現在以下幾個方面：

1. 訓練速度：較大的 batch size 通常可以加快訓練速度，因為每個 epoch 中可以處理更多的數據。但是，過大的 batch size 可能會導致記憶體不足而無法進行訓練。
2. 記憶體佔用：較大的 batch size 需要更多的記憶體進行存儲和計算，因此可能會導致記憶體不足的問題。在選擇 batch size 時，需要考慮硬件設備的記憶體限制。
3. 模型的收斂速度：較大的 batch size 可以使模型更快地收斂，但在一些情況下，較小的 batch size 可能會更好地幫助模型達到最優解，特別是對於小數據集或具有噪聲數據的情況。
4. 模型的泛化能力：較小的 batch size 可以增強模型的泛化能力，因為模型可以看到更多不同的樣本。然而，較小的 batch size 也可能會導致模型過擬合，因為模型更容易記住訓練集中的噪聲數據。

batch size太小可能會造成訓練時震盪太嚴重，從下圖可以看出在16或32時震盪劇烈，批次大小太小可能會導致模型訓練時不穩定，也就是說，模型無法在小批次的數據中學習到足夠的特徵，因此導致損失函數不穩定和波動。

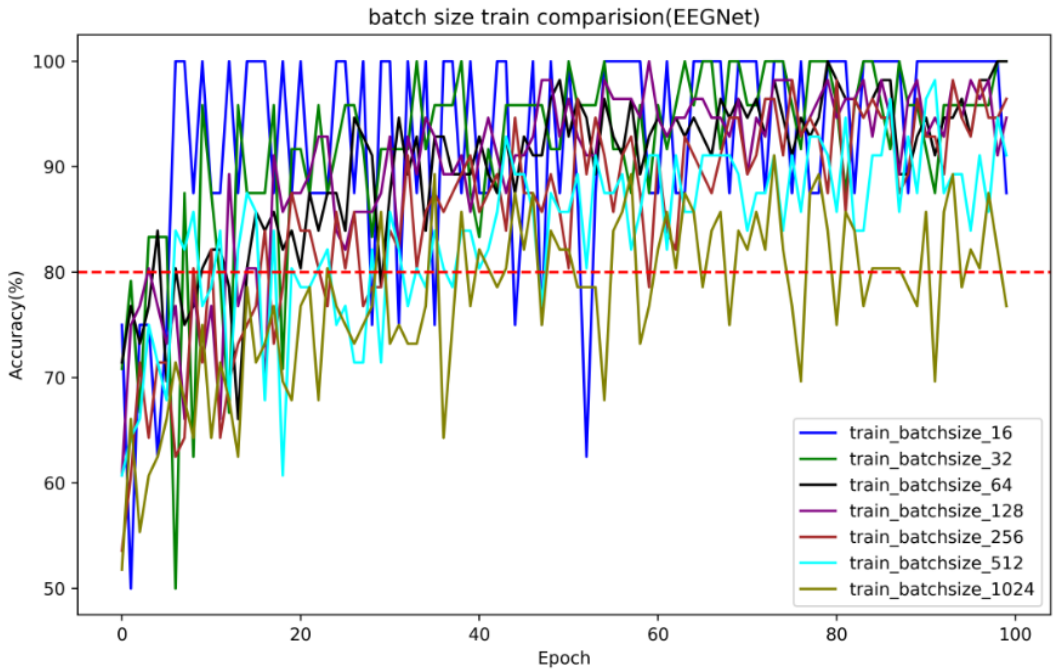
從下圖可以看出batch size在64, 128, 256表現較好, 若batch size在更大甚至超過輸入的大小的話，可能會導致以下問題：

1. 沒有足夠的數據可供使用進行訓練，因此無法更新參數。
2. 無法確保每個樣本都在一個批次中出現，這將導致某些樣本永遠不會被用於訓練。

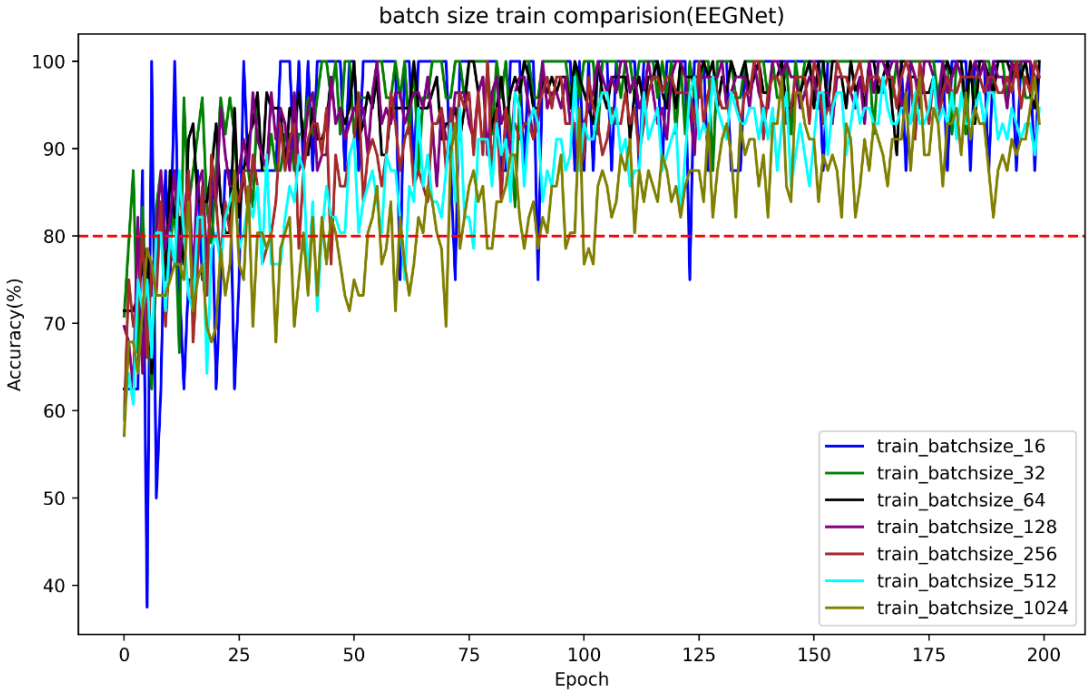
3. 批次大小超過數據集的大小可能會浪費內存，增加訓練時間和計算成本。

所以最好是將batch size設置在64, 128或256

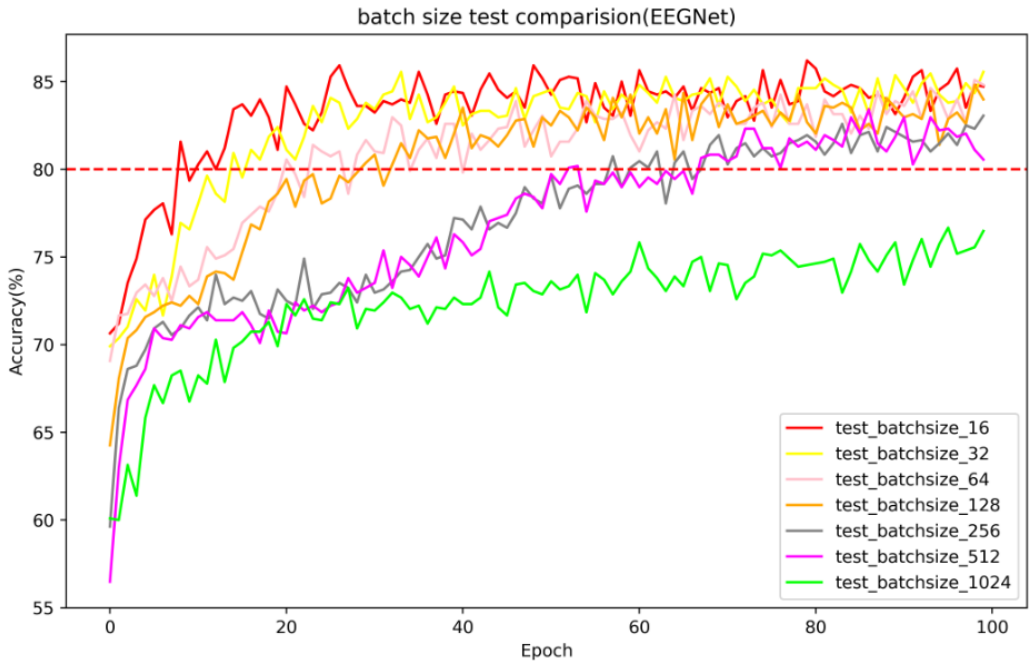
- epoch = 100的train data



- epoch = 200 的train data



- 在進一步從test data中觀察batch size太大確實會產生問題



learning rate 大小對訓練的影響

訓練速度：較大的學習率可以加快模型的訓練速度，因為參數更新得更快。但是，學習率過大可能會導致模型無法收斂。

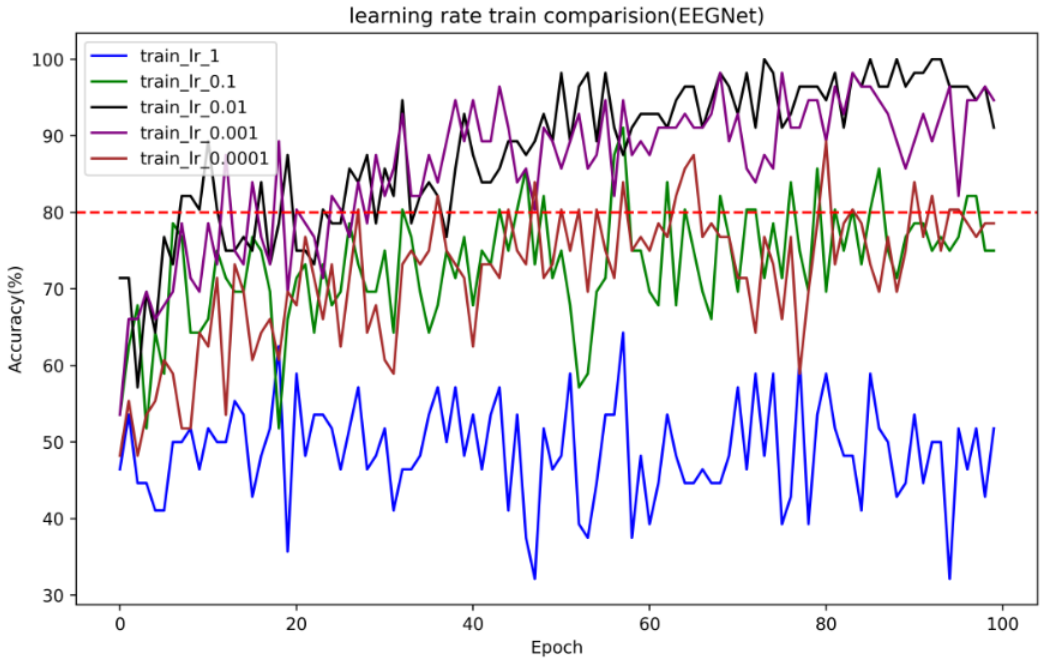
模型性能：較小的學習率可以使模型更容易收斂，而較大的學習率可能會導致模型不收斂或者在最優點附近震盪。因此，選擇合適的學習率可以使模型達到更好的性能。

模型穩定性：過大的學習率可能會導致模型發生梯度爆炸，過小的學習率可能會導致模型發生梯度消失。選擇合適的學習率可以使模型更穩定地訓練。

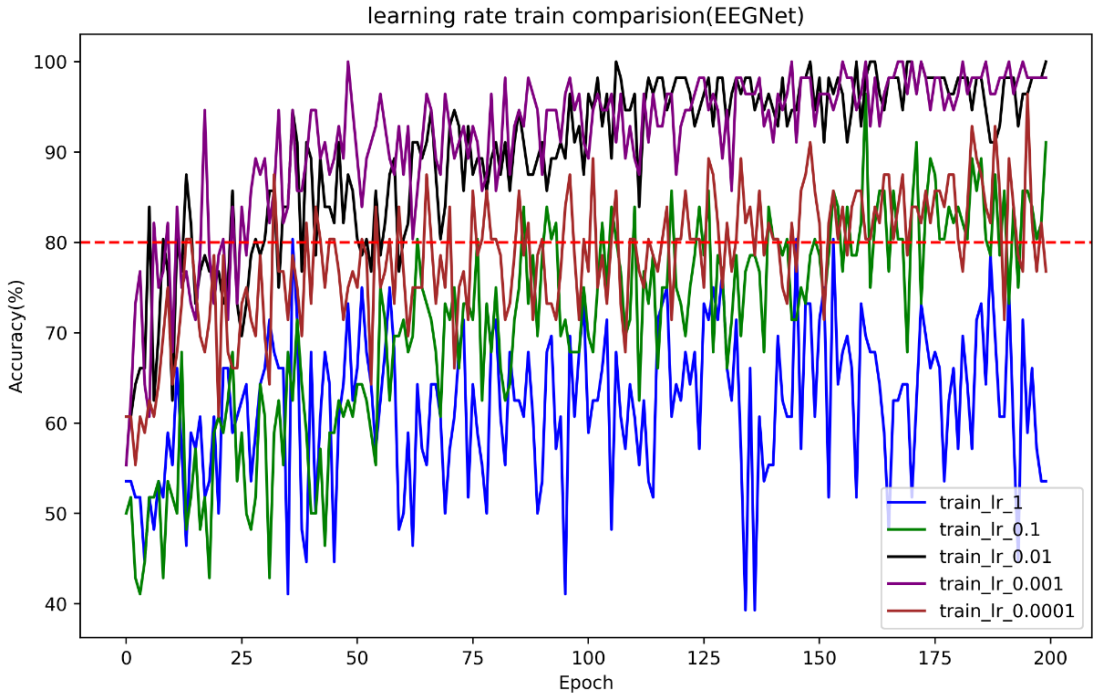
收斂結果：選擇不同的學習率大小可能會導致模型收斂到不同的局部最優解或全局最優解。因此，選擇合適的學習率大小可以使模型收斂到更優的解。

從下圖可以看出learning rate 越低其準確率明顯好，除了 lr=1 會產生劇烈震盪以外，當聚焦到細部，lr=0.001 及0.01 則不會差太多在變更低也就不會有太大的變化

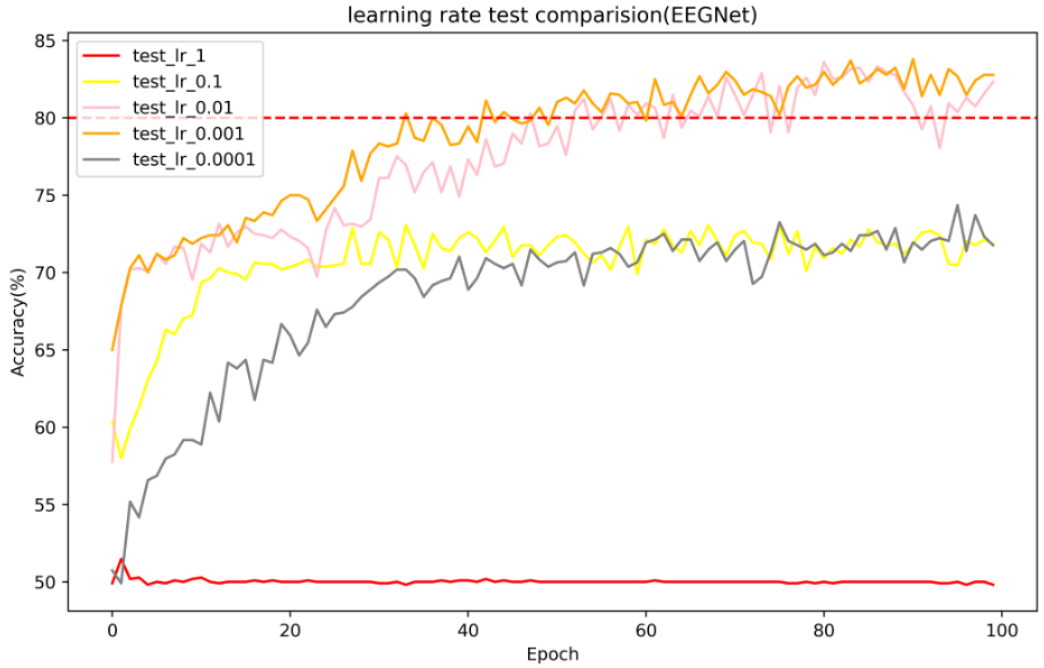
- epoch = 100的train data



- epoch = 200的train data



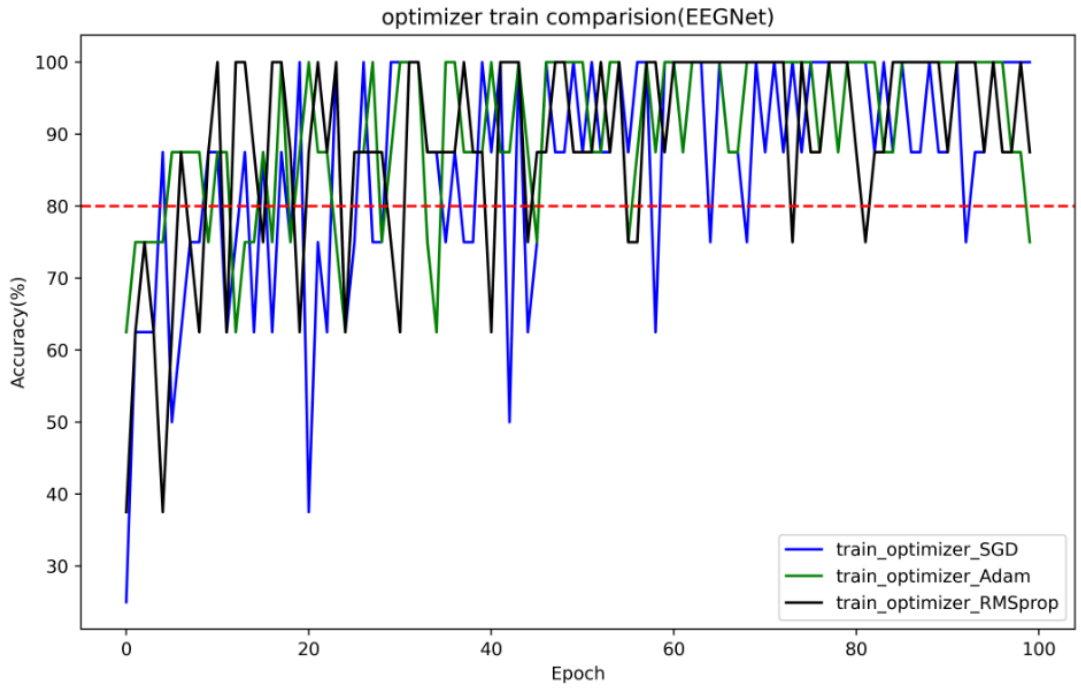
- 在進一步從test data中觀察lr = 1無法收斂而太小也會發生問題(可能發生梯度消失收斂在local min導致訓練不出來)



Optimizer的不同對訓練的影響

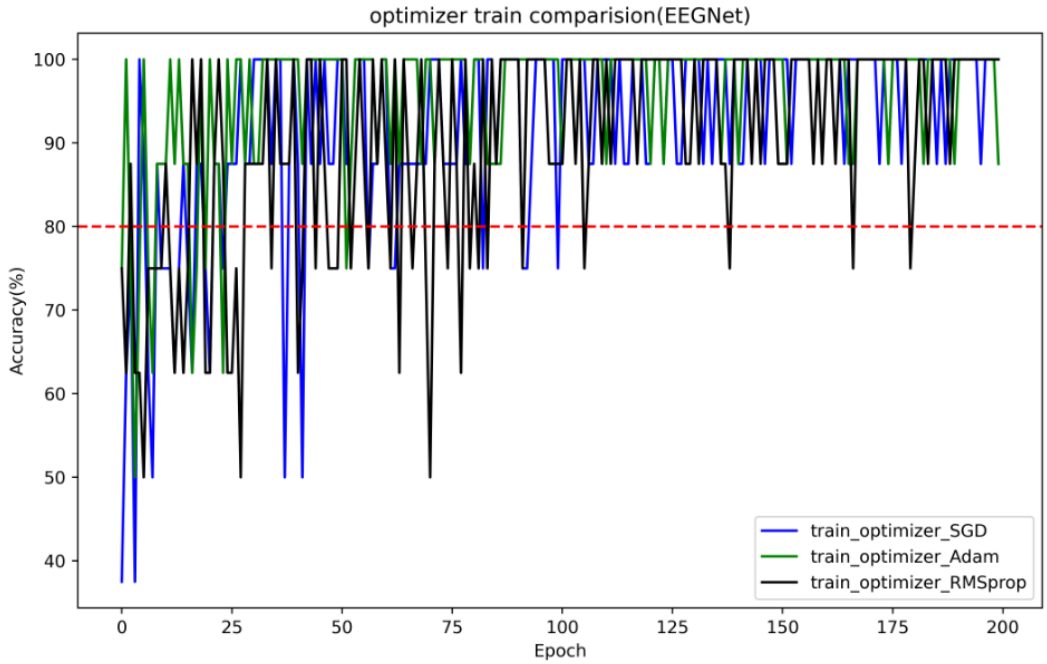
- SGD：隨機梯度下降是一種簡單的優化器，根據梯度更新模型的參數。在每個訓練批次中，SGD 只使用一個樣本或一個小批量的樣本進行更新，因此其更新很嘈雜，訓練過程中會有很大的變化。在大型數據集上，SGD 的收斂速度很慢。
- Adam：自適應矩估計（Adaptive Moment Estimation）是一種結合了隨機梯度下降和動量概念的優化器。Adam 調整每個參數的學習率，對不同參數進行個別的調整，這樣可以在訓練期間更快地收斂到局部最優解。Adam 可以有效地處理稀疏梯度和噪聲數據。
- RMSprop：RMSprop 是另一種自適應優化器，它通過除以梯度平方的移動平均值來調整學習率，這樣可以減少梯度的方差。與 Adam 一樣，RMSprop 可以有效地處理稀疏梯度和噪聲數據，但是在一些情況下，它的性能可能比 Adam 差一些。
- epoch = 100 的train data

  看不出明顯的變化，因此可以說是各有優劣



- epoch = 200 的train data

  Adam 和 SGD可能較為穩定



- 在進一步從test data中觀察使用adam會產生最好的結果