Deep Learning Lab 3

311552004 李明倫

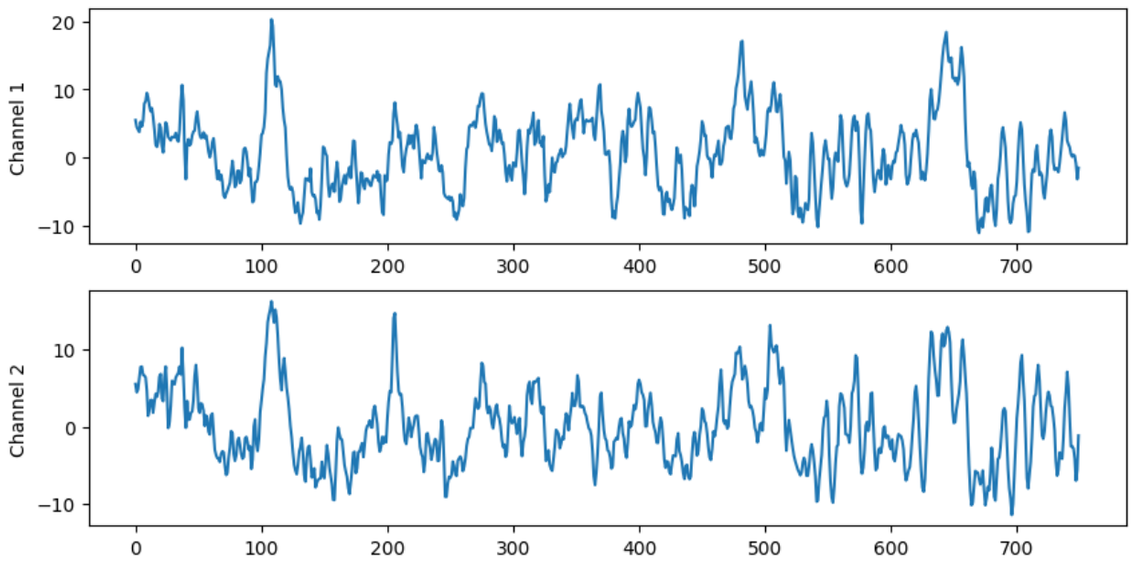
1. Introduction

In this lab, you are required to implement simple EEG classification models: EEGNet and DeepConvNet with BCI competition dataset.

Also, you need to try 3 kinds of activation functions: ReLU, Leaky ReLU, and ELU.

Dataset

BCI Competition III -IIIb Cued motor imagery with 2 classes (left hand, right hand) from 3 subjects.



2. Experiment Set Up

Detail of model

- EEGNet

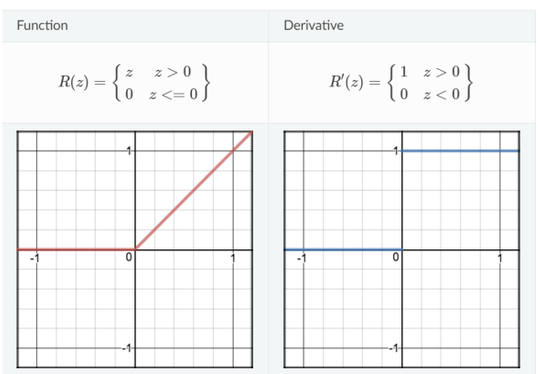
```
EEGNet(
  (firstConv): Sequential(
    (0): Conv2d(1, 16, kernel_size=(1, 51), stride=(1, 1), padding=(0, 25), bias=False)
    (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (depthwiseConv): Sequential(
    (0): Conv2d(16, 32, kernel_size=(2, 1), stride=(1, 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)
  )
  (classify): Sequential(
    (0): Linear(in_features=736, out_features=2, bias=True)
  )
)
```

- DeepConvNet

```
DeepConvNet(
  (block1): Sequential(
    (0): Conv2d(1, 25, kernel_size=(1, 5), stride=(1, 1))
    (1): Conv2d(25, 25, kernel_size=(2, 1), stride=(1, 1))
    (2): BatchNorm2d(25, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (3): ELU(alpha=1.0)
    (4): MaxPool2d(kernel_size=(1, 2), stride=(1, 2), padding=0, dilation=1, ceil_mode=False)
    (5): Dropout(p=0.5, inplace=False)
  )
  (block2): 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)
  )
  (block3): 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)
  )
  (block4): 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)
  )
  (fc): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=8600, out_features=2, bias=True)
  )
)
```

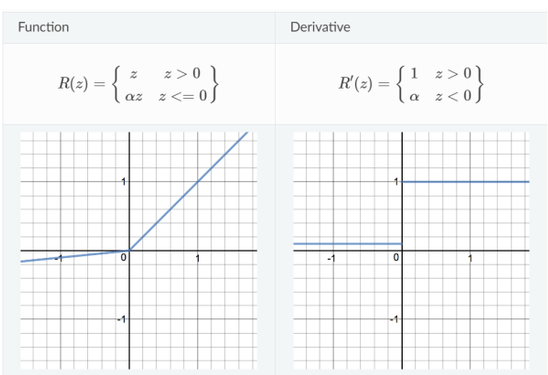
Explain the activation function

ReLU



- Stands for Rectified Linear Units
- 能夠解決梯度消失的問題 (不像 sigmoid 會把值壓在 0 到 1 之間，如果 network 比較深的時候，計算出來的 gradient 就會趨近於 0，就會導致梯度消失的問題，讓 model 沒辦法有效的更新)
- 但如果輸入的值小於 0 的時候，經過 ReLU 的計算會變成 0，就會導致算出來的 gradient 等於 0，也會讓 model 沒有辦法做有效的更新→ Dying ReLU problem

LeakyReLU

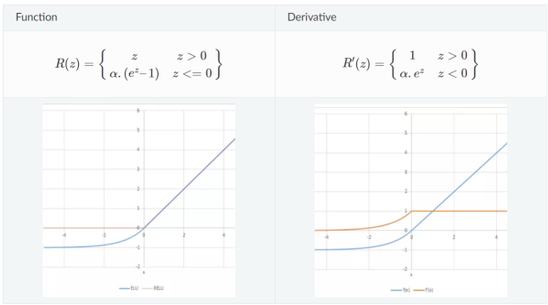


- ReLU 的其中一種變形，目的是希望當輸入 z < 0 的時候，還能夠有一個不等於0的值，避免 dying ReLU 的情況發生

ELU

- Stands for Exponential Linear Unit

- 跟 LeakyReLU 類似，也是希望在輸入 z < 0 時能夠不要是 0，因此當 z < 0 時這裡採用 exponential 的值，能夠平滑一點的有負的值



3. Experimental Results

The highest testing accuracy

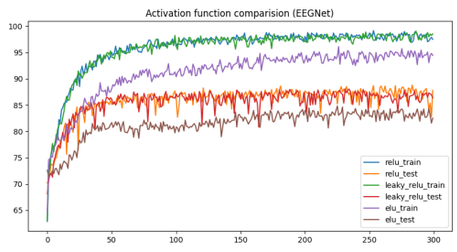
	ReLU	Leaky ReLU	ELU
EEGNet	88.80%	88.06%	84.81%
DeepConvNet	85.28%	83.52%	81.48%

Hyper-parameters: *EEGNet & DeepConvNet*

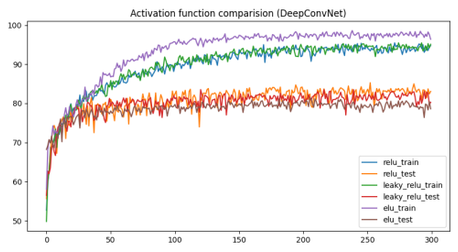
Batch size	Learning rate	Epochs	Optimizer	Loss function
64	0.001	300	Adam	CrossEntropy

Comparison figures

- EEGNet



- DeepConvNet



4. Discussion

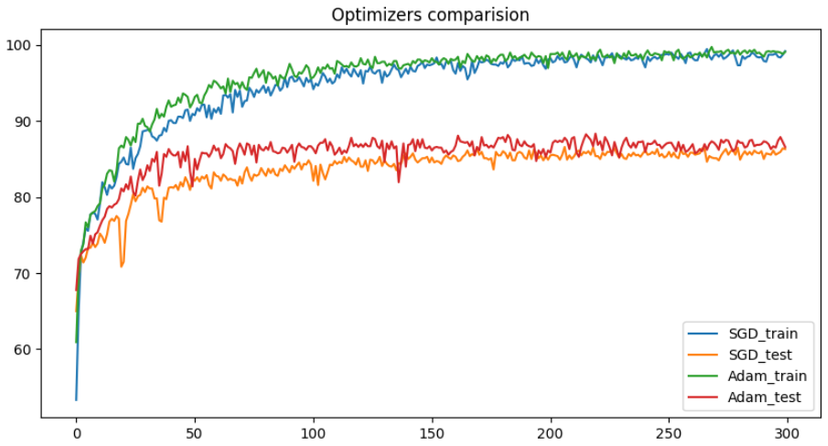
Different optimizers

在這個部分，我用了常見的 SGD + momentum 與 Adam 這兩個 optimizer 去做比較，並且採用大家常用的參數設定下去做測試，看看他們在 EEGNet 的表現上如何

Adam 是大家比較常用的 optimizer，特點就是收斂速度快，而且能夠解決大部分 gradient optimization 的問題，但也有一些文章提出 Adam 在 testing 的 generalization 上會比 training 的時候還要差，因此仍有一部份的經典模型仍然採用 SGD + momentum 的配置下去做 optimization

而在這次的實驗中，得到的結果還是用 Adam 的 optimization 效果比較好

	lr	momentum	*Test Acc 1*	*Test Acc 2*	*Test Acc 3*
SGD	0.01	0.9	86.85%	86.57%	87.78%
Adam	0.001	-	88.15%	88.33%	88.06%



Effect of weight decay

在這個部分，我想去探討一下 weight decay 也就是大家常說的 L2 normalization，對訓練帶來的幫助與降低 overfitting 的表現上，他的影響為何

因此我這邊就用 [0.0005, 0.005, 0.05] 這三個值分別對 SGD 與 Adam 下去做測試

從結果來看會發現說，適當的 weight_decay (wd=0.005) 能夠對 model 的 performance 有所提升，但如果設得太大 (wd=0.05)，反而會讓 penalty 的效果太重，導致 model 的表現更差

- SGD

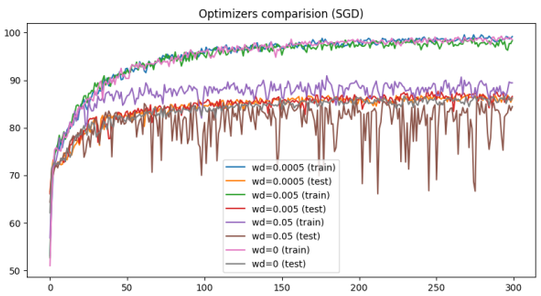
  Without weight decay: 86.67%

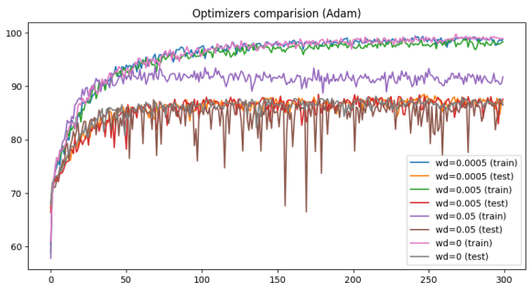
weight_decay	*Test Acc*
0.0005	86.94%
0.005	87.96%
0.05	86.57%

- Adam

  Without weight decay: 88.15%

weight_decay	*Test Acc*
0.0005	88.52%
0.005	88.52%
0.05	87.96%





5. Extra

在這個部分，我自己寫了一個 model 叫 *MyNet*，由主要三個 convolution 的 block 以及最後的 fully connected block 組成。為了符合資料的特性，因此在 convolution filter shape (kernel_size) 的選擇上，採用了 (1, x) 的大小去做卷積。

在每一個 block 中，我有另外再加入 batch normalization layer 去幫助訓練以及減少 overfitting，在倒數兩個 block 裡面也有加入 dropout 去抑制 overfitting，讓 model 可以更 generalize 在 testing data 上。

- Hyper-parameters

Batch size	Lr	Epochs	Activation	Optimizer
128	0.005	200	ELU	Adam, weight_decay=0.001

- MyNet

```
import torch
import torch.nn as nn

torch.manual_seed(0)

class MyNet(nn.Module):
    def __init__(self, num_channel=6):
        super(MyNet, self).__init__()
        self.block1 = nn.Sequential(
            nn.Conv2d(1, num_channel, kernel_size=(1, 5), padding=2),
            nn.BatchNorm2d(num_channel),
            nn.ReLU(),
            nn.AvgPool2d(kernel_size=(1, 4)),
        )

        self.block2 = nn.Sequential(
            nn.Conv2d(num_channel, num_channel, kernel_size=(1, 5), padding=0),
            nn.BatchNorm2d(num_channel),
            nn.ReLU(),
            nn.AvgPool2d(kernel_size=(1, 4)),
        )

        self.block3 = nn.Sequential(
            nn.Conv2d(num_channel, num_channel*2, kernel_size=(1, 5), padding=0),
            nn.BatchNorm2d(num_channel*2),
            nn.ReLU(),
            nn.AvgPool2d(kernel_size=(1, 4)),
            nn.Dropout(0.5),
        )

        self.fc = nn.Sequential(
            nn.Linear(num_channel * 40 * 3, 64),
            nn.BatchNorm1d(64),
            nn.ReLU(),
            nn.Dropout(0.4),
            nn.Linear(64, 2),
        )


    def forward(self, x):
        x = self.block1(x)
        x = self.block2(x)
        x = self.block3(x)

        x = torch.flatten(x, 1)
        x = self.fc(x)
        return x
```

- Training and testing result

```
----- start training -----
Epoch: [1/200] loss: 0.66591, acc: 60.19%, test_acc: 68.80%
Epoch: [11/200] loss: 0.48983, acc: 76.85%, test_acc: 78.52%
Epoch: [21/200] loss: 0.40338, acc: 81.11%, test_acc: 81.67%
Epoch: [31/200] loss: 0.35548, acc: 83.89%, test_acc: 83.98%
Epoch: [41/200] loss: 0.33171, acc: 85.46%, test_acc: 83.15%
Epoch: [51/200] loss: 0.29891, acc: 87.78%, test_acc: 85.93%
Epoch: [61/200] loss: 0.32573, acc: 86.02%, test_acc: 85.37%
Epoch: [71/200] loss: 0.28457, acc: 86.67%, test_acc: 84.72%
Epoch: [81/200] loss: 0.29326, acc: 90.09%, test_acc: 84.35%
Epoch: [91/200] loss: 0.27345, acc: 89.35%, test_acc: 84.72%
Epoch: [101/200] loss: 0.26776, acc: 87.50%, test_acc: 85.46%
Epoch: [111/200] loss: 0.25296, acc: 90.65%, test_acc: 84.54%
Epoch: [121/200] loss: 0.23986, acc: 90.46%, test_acc: 84.17%
Epoch: [131/200] loss: 0.25024, acc: 89.26%, test_acc: 85.56%
Epoch: [141/200] loss: 0.26247, acc: 88.89%, test_acc: 86.30%
Epoch: [151/200] loss: 0.23480, acc: 90.83%, test_acc: 86.85%
Epoch: [161/200] loss: 0.22992, acc: 91.02%, test_acc: 83.80%
Epoch: [171/200] loss: 0.24315, acc: 90.28%, test_acc: 85.09%
Epoch: [181/200] loss: 0.22988, acc: 89.91%, test_acc: 85.74%
Epoch: [191/200] loss: 0.20284, acc: 91.85%, test_acc: 86.02%
Epoch: [200/200] loss: 0.21670, acc: 91.30%, test_acc: 86.48%
-- Best acc: 87.31%
```