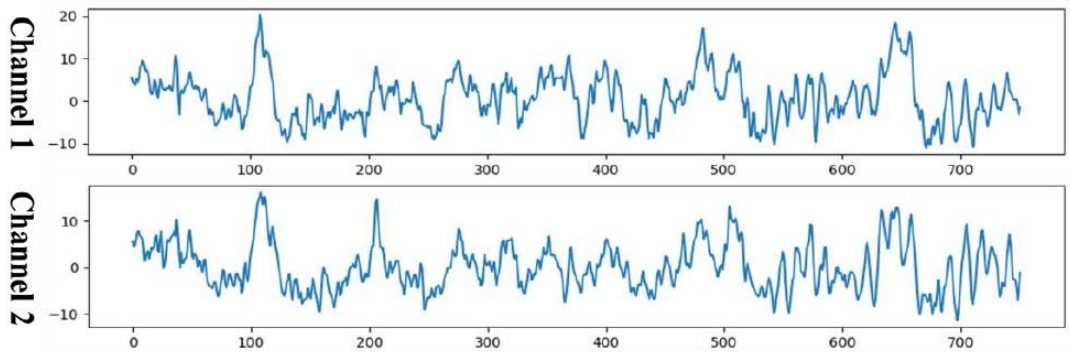
311551069 余忠旻 Lab3 : EEG classification

1. Introduction



上圖是腦波圖，這次作業我們要用 CNN 來做腦波圖形分類問題。會使用 Pytorch 來建構 EEGNet 和 DeepConvNet 來實現 EEG classification，在 PyTorch 中，要使用 torch.nn 套件，此套件包含建立 CNN 的模組、可擴充類別和所有必要元件。建立完 model 後，只需要自己定義 forward 函式，PyTorch 會自行進行 backpropagate，而這次作業會在 forward 過程中使用不同的 activation function (包含 ReLU, LeakyReLU, ELU) 來觀察對結果有甚麼影響。

2. Experiment set up:

A. The detail of your 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)
  )
  (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)
  )
)
```

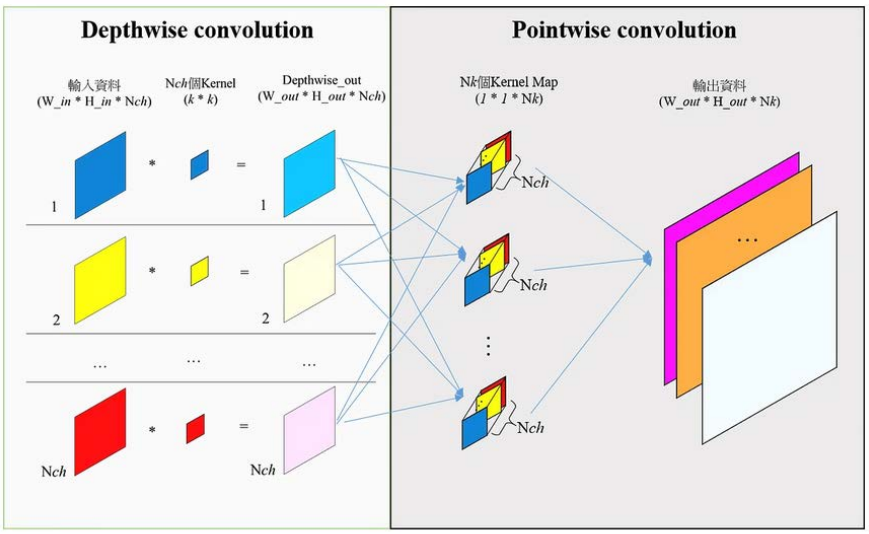
根據圖片上助教給的架構實作就好了

```
class EEGNet(nn.Module):
    def __init__(self, activation_func, device):
        super(EEGNet, self).__init__()
        self.device = device
        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-5, 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 = 16, bias = False),
            nn.BatchNorm2d(32, eps = 1e-5, momentum = 0.1, affine = True, track_running_stats = True),
            activation_func, #ELU(alpha=1.0)
            nn.AvgPool2d(kernel_size = (1, 4), stride = (1, 4), padding = 0),
            nn.Dropout(p = 0.25)
        )
        self.seperableConv = nn.Sequential(
            nn.Conv2d(32, 32, kernel_size = (1, 15), stride = (1, 1), padding = (0, 7), bias = False),
            nn.BatchNorm2d(32, eps = 1e-5, momentum = 0.1, affine = True, track_running_stats = True),
            activation_func, #ELU(alpha=1.0)
            nn.AvgPool2d(kernel_size = (1, 8), stride = (1, 8), padding = 0),
            nn.Dropout(p = 0.25)
        )
        self.classify = nn.Linear(736, 2) #in_features=736, out_features=2, bias=True

    def forward(self, X):
        out = self.firstconv(X.to(self.device))
        out = self.depthwiseConv(out)
        out = self.seperableConv(out)
        out = out.view(out.shape[0], -1)  #flatten/resize
        out = self.classify(out)
        return out
```

其中，我把 activation function 和 device 設定為可傳入的參數，這樣在執行可以較比方便使用不同的 activation function (ReLU, LeakyReLU, ELU)，device 也是可以較方便設定要用 GPU 還是 CPU 來同步資料存取的位置。

- Sequential(): 會依序執行括號內的內容。
- Conv2d(): 可以設定 kernel size, stride 大小等等，是 CNN 的核心，可用來偵測影像中的各個 feature，並將輸出通道作為下一層的輸入。
- BatchNorm2d(): 圖層會在輸入上套用正規化，也就是將資料利用資料的特性轉換到正規化空間，使其平均數是 0，標準差是 1，來提高網路精確度。
- MaxPool(): Pooling 目的是將圖片資料量減少並保留重要資訊的方法，Max Pooling 把原本的資料做 max out 的計算，可協助我們確保影像中的物件位置不會影響神經網路偵測其特定功能的能力。
- Dropout(p = 0.25): 這裡的 0.25 是指該層的神經元在每次迭代訓練時會有 25%的機率停止運作，每次迭代停止運作的 Neurons 會是隨機的，每次都不一樣，因此透過 Dropout 可以讓每次迭代有看似「不同結構」的 Neural Network，來避免出現 overfitting。
- Linear(): 是網路中的最後一層，作為 fully connected layer，會計算每個類別的分數。

而 EEGNet 有用到一個技術是 Depthwise Separable Convolution。一般的卷積運算是每個 Kernel Map 會和所有 channel 去做 convolution，而 Depthwise Separable Convolution 則是會將輸入資料的每一個 Channel 去建立一個 k*k 的 Kernel，然後每一個 Channel 針對對應的 Kernel 都各自分開做卷積，下圖為流程示意圖:



Depthwise separable convolution

由上圖流程示意圖，我們可以估算 Depthwise Separable Convolution 使用的計算量相較於一般卷積的計算量，如下面所計算，它能大幅降低 CNN 的計算量，當 Kernel Map 越大和 Kernel Map 數量越多，Depthwise separable convolution 可以節省越多計算量。在 PyTorch 中，用 groups 參數來實現。

輸入資料: W_in * H_in * Nch，Kernel Map: k * k * Nk
輸出資料: W_out * H_out * Nk
一般卷積 計算量為 W_in * H_in * Nch* k * k * Nk

$$\frac{\text{Depthwise separable convolution 計算量}}{\text{一般卷積計算量}}$$

$$= \frac{W_{in} * H_{in} * Nch * k * k + Nch * Nk * W\_in * H\_in}{W\_in * H\_in * Nch * k * k * Nk}$$

$$= \frac{1}{Nk} + \frac{1}{k * k}$$

Ref: https://chih-sheng-huang821.medium.com/%E6%B7%B1%E5%BA%A6%E5%AD%B8%E7%BF%92-mobilenet-depthwise-separable-convolution-f1ed016b3467

DeepConvNet

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

根據圖片上助教給的架構實作就好了，跟上面一樣，我把 activation function 和 device 設定為可傳入的參數，方便操作。其中要比較注意的事是，因為 loss function 我使用 nn.CrossEntropyLoss, 所以不需要加 softmax after final fc layer，因為 this criterion combines LogSoftMax and NLLLoss in one single class.

```
class DeepConvNet(nn.Module):
    def __init__(self, activation_func, device):
        super(DeepConvNet, self).__init__()
        self.device = device
        self.conv0 = nn.Conv2d(1, 25, kernel_size = (1, 5))
        self.conv1 = nn.Sequential(
            nn.Conv2d(25, 25, kernel_size = (2, 1)),
            nn.BatchNorm2d(25, eps = 1e-5, momentum = 0.1),
            activation_func,
            nn.MaxPool2d(kernel_size = (1, 2)),
            nn.Dropout(p = 0.5)
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(25, 50, kernel_size = (1, 5)),
            nn.BatchNorm2d(50, eps = 1e-5, momentum = 0.1),
            activation_func,
            nn.MaxPool2d(kernel_size = (1, 2)),
            nn.Dropout(p = 0.5)
        )
        self.conv3 = nn.Sequential(
            nn.Conv2d(50, 100, kernel_size = (1, 5)),
            nn.BatchNorm2d(100, eps = 1e-5, momentum = 0.1),
            activation_func,
            nn.MaxPool2d(kernel_size = (1, 2)),
            nn.Dropout(p = 0.5)
        )
        self.conv4 = nn.Sequential(
            nn.Conv2d(100, 200, kernel_size = (1, 5)),
            nn.BatchNorm2d(200, eps = 1e-5, momentum = 0.1),
            activation_func,
            nn.MaxPool2d(kernel_size = (1, 2)),
            nn.Dropout(p = 0.5)
        )

        self.classify = nn.Linear(8600, 2) #in_features=8600, out_features=2

    def forward(self, X):
        out = self.conv0(X.to(self.device))
        out = self.conv1(out)
        out = self.conv2(out)
        out = self.conv3(out)
        out = self.conv4(out)
        out = out.view(out.shape[0], -1)  #flatten/resize
        out = self.classify(out)
        #out = nn.functional.softmax(out, dim = 0)
        return out
```

而細看 LogSoftMax and NLLLoss，公式如下:

nn.LogSoftmax

$$\text{LogSoftmax}(x_i) = \log\left(\frac{\exp(x_i)}{\sum_j \exp(x_j)}\right)$$

nn.NLLLoss

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_{y_n} x_{n,y_n}, \quad w_c = \text{weight}[c] \cdot 1\{c \neq \text{ignore\_index}\},$$

再細看 nn.CrossEntropyLoss, 就會發現其實可以看作是 nn.LogSoftmax 和 nn.NLLLoss 的结合，所以上述實作才不需要加 softmax after final fc layer，CrossEntropyLoss 公式和敘述如下:

CLASS `torch.nn.CrossEntropyLoss(weight=None, size_average=None, ignore_index=-100, reduce=None, reduction='mean', label_smoothing=0.0)` [SOURCE]

This criterion computes the cross entropy loss between input and target.

It is useful when training a classification problem with *C* classes. If provided, the optional argument `weight` should be a 1D *Tensor* assigning weight to each of the classes. This is particularly useful when you have an unbalanced training set.

The *input* is expected to contain raw, unnormalized scores for each class. *input* has to be a Tensor of size either $(minibatch, C)$ or $(minibatch, C, d_1, d_2, ..., d_K)$ with $K \geq 1$ for the *K*-dimensional case. The latter is useful for higher dimension inputs, such as computing cross entropy loss per-pixel for 2D images.

The *target* that this criterion expects should contain either:

- Class indices in the range $[0, C-1]$ where C is the number of classes; if *ignore_index* is specified, this loss also accepts this class index (this index may not necessarily be in the class range). The unreduced (i.e. with `reduction` set to `'none'`) loss for this case can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_{y_n} \log \frac{\exp(x_{n,y_n})}{\sum_{c=1}^{C} \exp(x_{n,c})} \cdot 1\{y_n \neq \text{ignore\_index}\}$$

where x is the input, y is the target, w is the weight, C is the number of classes, and N spans the minibatch dimension as well as $d_1, ..., d_k$ for the *K*-dimensional case. If `reduction` is not `'none'` (default `'mean'`), then

$$\ell(x, y) = \begin{cases} \sum_{n=1}^{N} \frac{1}{\sum_{n=1}^{N} w_{y_n} \cdot 1\{y_n \neq \text{ignore\_index}\}} l_n, & \text{if reduction} = \text{'mean'}; \\ \sum_{n=1}^{N} l_n, & \text{if reduction} = \text{'sum'}. \end{cases}$$

Note that this case is equivalent to the combination of `LogSoftmax` and `NLLLoss`.

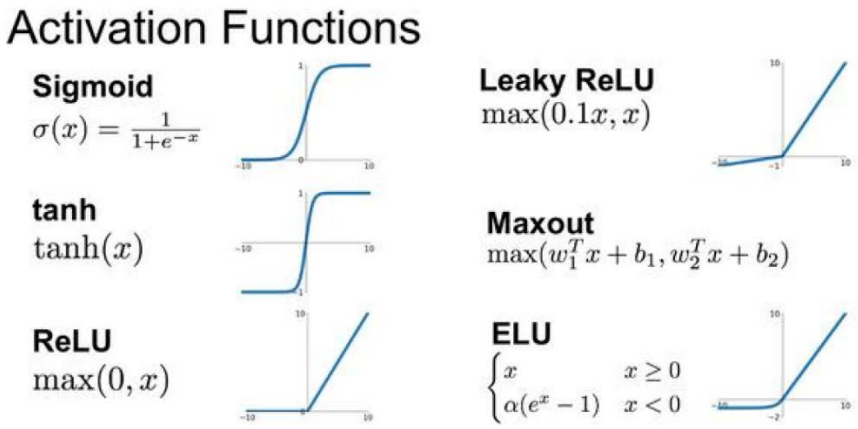
訓練部分，我使用 Adam optimizer 進行 gradient descent update。其中參數部分我將 learning rate 設為 0.001，weight decay 設為 0.01，weight decay 是指在更新參數時加上一個懲罰項，來避免 overfitting，而 Loss function 我使用 Cross Entropy，epochs 則是設為 500 個。
訓練的過程就是重複清空 optimizer 梯度、Forwarding、計算 loss、呼叫 backward 計算 gradients、呼叫 step 進行 gradient descent update 等步驟。
而 testing 也是一樣，只是不用計算 gradient 及 update 而已。

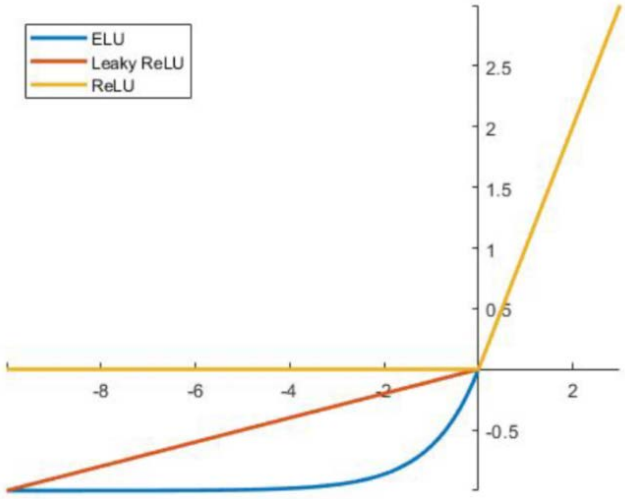
```
# setting device, optimizer, loss function, number of epochs
model.to(device)
optimizer = torch.optim.Adam(model.parameters(), lr = 0.001, weight_decay = 0.01)
criterion = nn.CrossEntropyLoss()
```

```
# training process
total_loss = 0
total_train = 0
correct_train = 0
model.train()
for i, (data, label) in enumerate(train_loader):
    data = data.to(device, dtype = torch.float)
    label = label.to(device, dtype = torch.long)

    # clear gradient
    optimizer.zero_grad()

    # forward propagation
    output = model(data)

    # calculate cross entropy (loss function)
    loss = criterion(output, label)
    total_loss += loss

    # get predictions from the maximum value
    prediction = torch.max(output.data, 1)[1]

    # total number of labels
    total_train += len(label)

    # total correct predictions
    correct_train += (prediction == label).float().sum()

    # Calculate gradients
    loss.backward()

    # Update parameters
    optimizer.step()
```

```
# testing process
total_test = 0
correct_test = 0
model.eval()
for i, (data, label) in enumerate(test_loader):
    data = data.to(device, dtype = torch.float)
    label = label.to(device, dtype = torch.long)

    # no need to calculate gradient and loss function

    # forward propagation
    output = model(data)

    # get predictions from the maximum value
    prediction = torch.max(output.data, 1)[1]

    # total number of labels
    total_test += len(label)

    # total correct predictions
    correct_test += (prediction == label).float().sum()
```

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





上圖是各種 activation function 的圖形以及(ReLU, Leaky ReLU, ELU)三種比較圖形。可以看見 ELU 和 Leaky ReLU 相較於 ReLU 的差異在於，當 value 小於 0 時前者依舊有梯度存在，而後者的梯度會直接為 0，

因此 ReLU 會衍生當某個神經元輸出為 0 後，就難以再度輸出的問題，極端狀況下就是假如輸入 ReLU 的所有值都是負數，則 ReLU activated 之後都為 0，也就是 dying ReLU problem。產生這種現象通常為兩個原因: 參數初始化不合理或者 learning rate 太高導致在訓練過程中參數更新太大。

LeakyReLU 當 value 大於 0 時梯度和 ReLU 一樣，但當 value 小於 0 時 LeakyReLU 的梯度會是一個非零斜率，ReLU 的梯度則是 0，因此 LeakyReLU 能解決上述 dying ReLU problem。

ELU 是一種傾向於將數值更快地收斂到零並產生更準確結果的 activation function，也可以解決上述 dying ReLU problem，ELU 跟 LeakyReLU 相比，計算量較大之外， LeakyReLU do not ensure a noise-robust deactivation state. ELUs saturate to a negative value with smaller inputs and thereby decrease the forward propagated variation and information.

3. Experimental results

A. The highest testing accuracy

learning rate = 0.001

weight decay = 0.01

epochs = 500

EEGNet

```
ReLU has max accuracy  88.24073791503906 %
LeakyReLU has max accuracy  87.77777862548828 %
ELU has max accuracy  82.87036895751953 %
```

DeepConvNet

```
ReLU has max accuracy  84.81481170654297 %
LeakyReLU has max accuracy  85.46296691894531 %
ELU has max accuracy  82.77777099609375 %
```

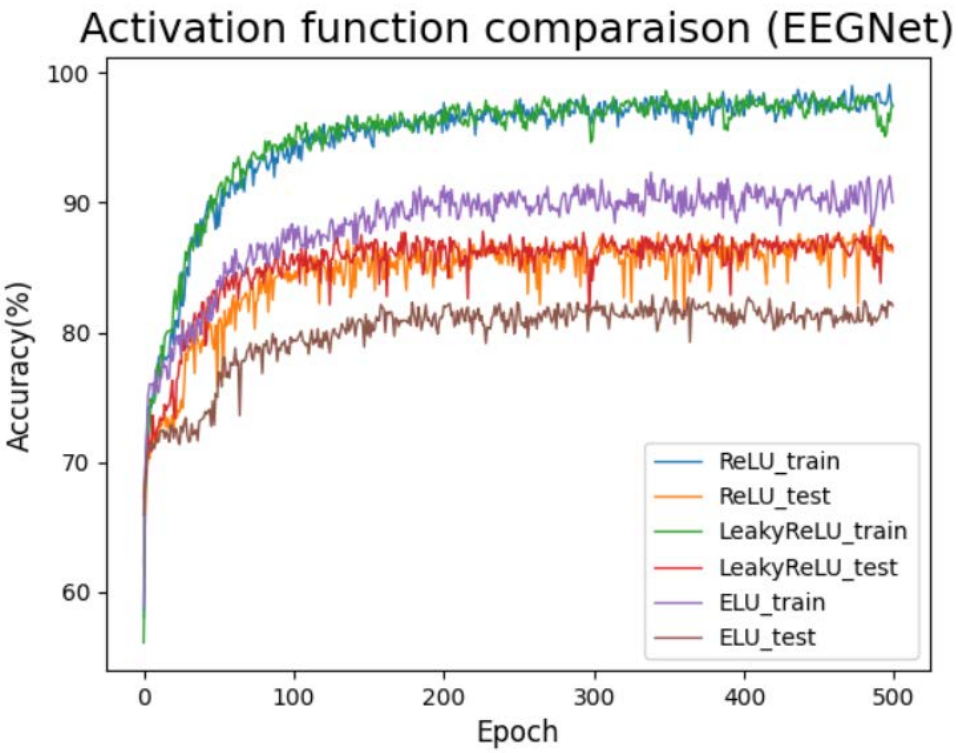
Testing accuracy comparison:

	ReLU	**LeakyReLU**	**ELU**
EEGNet	88.24%	87.77%	82.87%
DeepConvNet	84.81%	85.46%	82.77%

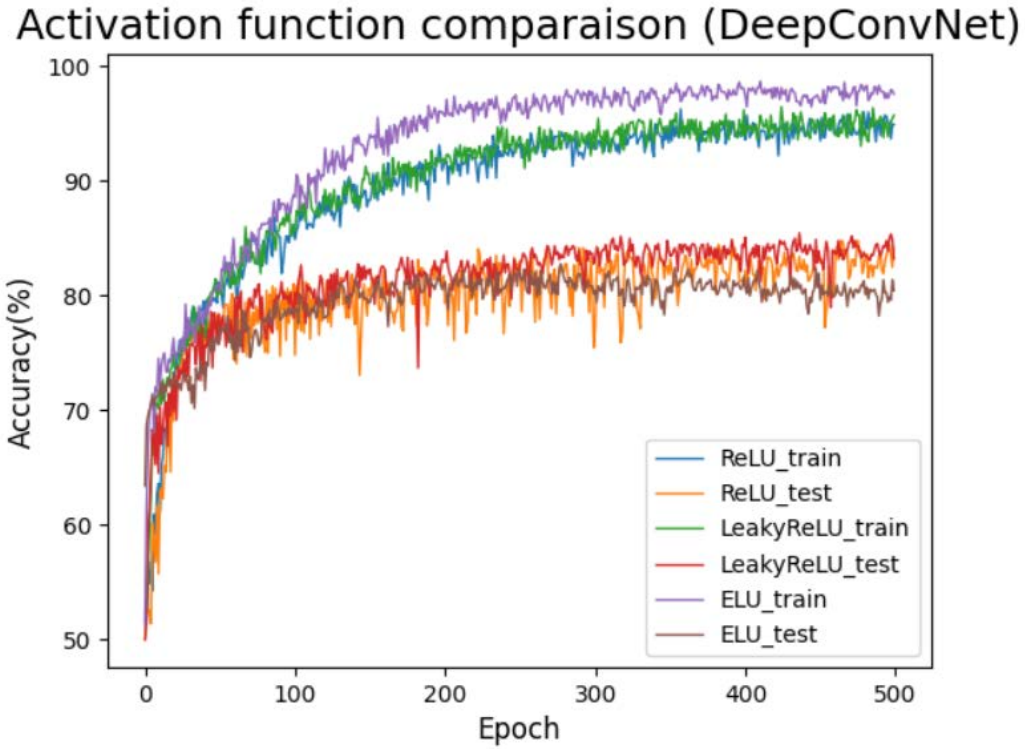
使用 EEGNet + ReLU 和 EEGNet + LeakyReLU 都讓 testing accuracy 超過 87%

B. Comparison figures

EEGNet

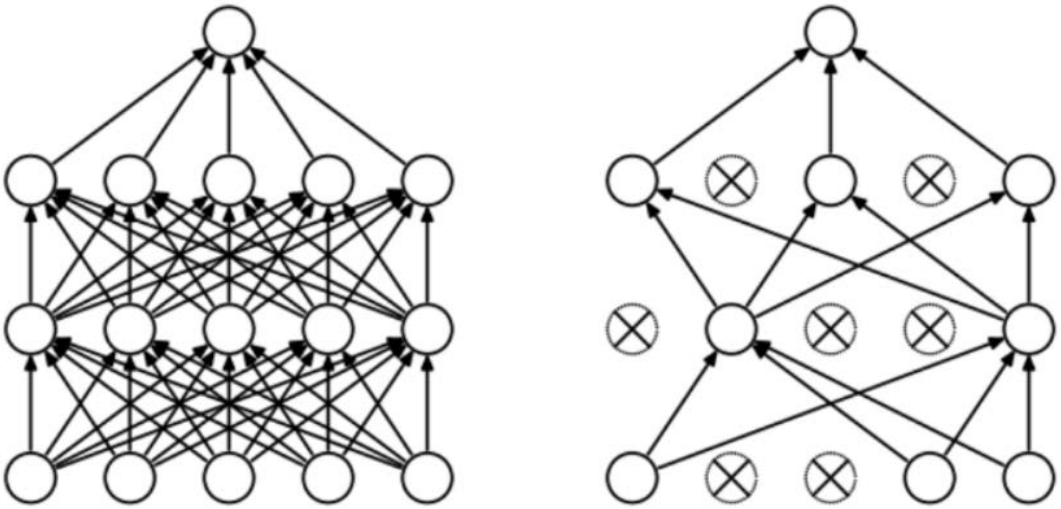


DeepConvNet



4. Discussion

A. Dropout



(a) Standard Neural Net　　(b) After applying dropout.

上面在 Experiment set up 有提到 Dropout，這裡再詳細說明一下。

如果我們在訓練過程中使用了 Dropout Layer，Neural Network 會由上圖(a)變成(b)。在初始化 Dropout Layer 時會傳入一個「機率」，表示每個 Neuron 有多大的機率會「停止運作」。所謂的「停止運作」指的是不管我們輸入什麼到 Neural Network 中，這些 Neuron 的輸出 (Activation) 永遠都是 0。當 Neuron 的輸出為 0，不管連接的 Weight 有多大，相乘以後的結果一定都是 0。此外，在 Backpropagation 的過程中，如果某一個 Weight 其 Input (上一層的 Activation) 為 0，將會導致 Gradient 亦為 0，使得這一個 Weight 無法被更新。 因此，在上圖(b)中，我們以（✕）來表示停止運作的 Neuron，並將其連接的線段 (Weight) 去除，整個 Neural Network 就像是沒有這些 Neuron 的存在。當我們要輸入下一個 Input Batch 到 Neural Network 時，Dropout Layer 會將剛剛停止運作的 Neuron 恢復正常，再重新選擇新的一批 Neuron 來停止運作。就這樣重複進行相同的操作，在每一個 Iteration 中，都有一部份的 Neuron 不會對 Neuron Network 的輸出有所貢獻，連接他們的 Weight 當然也不會被更新。

為甚麼能解決 overfitting 的問題，則是 Dropout 機制就像是一種 Ensemble 的概念來解決 overfitting，在 Ensemble Learning 中，會訓練多個模型，並一起考慮多個模型的預測結果，作為最終的預測結果，方式像是透過取平均或是多數決的概念，來提升模型整體的表現。而 Dropout 在每一個 Iteration 中，我們都會「停止運作」某些 Neuron，就像是從 Neural Network 將這些 Neuron 去除一樣，因此每一個 Iteration 我們都可以得到一個「不同結構」的 Neural Network。最終，在整個 Neural Network 的訓練過程，表面上看起來是在訓練單一個模型，但是實際上是綜合了各種不同

結構的模型。因此，Neural Network 的輸出也不單單只是「一個」Neural Network 的輸出，而是「多個」Neural Network 輸出的綜合結果。除此之外，Dropout 能避免 Neuron 只對單一特徵過度敏感，這樣即使輸入的所有特徵中包含了一些瑕疵，也不會受到太大的影響，當然會使得整個模型的 Generalization 能力更棒。

B. Weight Decay

weight decay 是一種 regularization 的方法，weight decay 會將所有參數的平方乘以另一個較小的數字 (λ) 添加到 loss function 中，而 loss function 和 weights update using gradient descent 就會看起來如下:

$$w_i^{t+1} \longleftarrow w_i^t - \eta \cdot \frac{\partial L'}{\partial w_i} \text{ , where } L'(\boldsymbol{W}) = L(\boldsymbol{W}) + \lambda \cdot \sum_{n=1}^{N} (w_n^t)^2$$

$$\Longrightarrow w_i^{t+1} \longleftarrow w_i^t - \eta \cdot (\frac{\partial L}{\partial w_i} + 2\lambda w_i^t)$$

從上面我們可以看到，進行參數更新時會多減 $\eta \cdot 2\lambda w_i^t$，其實也就是在更新參數時加上一個懲罰項，不讓模型擬合時過度依賴某一些權重，進而達到正則化的效果，來強制約束權重(weight)，因此可以用來避免 overfitting。

C. Execution time of GPU vs. CPU

我使用 torch.cuda.Event() 來記錄 model 分別在 GPU 和 CPU 的執行時間

EEGNet + LeakyReLU (GPU)

```
epoch  490 :
trainig accuracy:  tensor(96.4815, device='cuda:0')   loss:  tensor(0.5170, device='cuda:0', grad_fn=<AddBackward0>)
testing accuracy:  tensor(84.8148, device='cuda:0')

epoch  500 :
trainig accuracy:  tensor(97.2222, device='cuda:0')   loss:  tensor(0.5208, device='cuda:0', grad_fn=<AddBackward0>)
testing accuracy:  tensor(86.7593, device='cuda:0')

LeakyReLU has max accuracy 87.03703308105469% at epoch 314
execution time: 57.598015625s
```

EEGNet + LeakyReLU (CPU)

```
epoch  490 :
trainig accuracy:  tensor(96.3889)   loss:  tensor(0.5567, grad_fn=<AddBackward0>)
testing accuracy:  tensor(84.2593)

epoch  500 :
trainig accuracy:  tensor(97.4074)   loss:  tensor(0.5503, grad_fn=<AddBackward0>)
testing accuracy:  tensor(84.9074)

LeakyReLU has max accuracy 87.03704071044922% at epoch 213
execution time: 905.8438125s
```

DeepConvNet + LeakyReLU (GPU)

```
epoch  490 :
trainig accuracy:  tensor(94.9074, device='cuda:0')   loss:  tensor(0.8554, device='cuda:0', grad_fn=<AddBackward0>)
testing accuracy:  tensor(83.7037, device='cuda:0')

epoch  500 :
trainig accuracy:  tensor(95.1852, device='cuda:0')   loss:  tensor(0.7013, device='cuda:0', grad_fn=<AddBackward0>)
testing accuracy:  tensor(82.4074, device='cuda:0')

LeakyReLU has max accuracy 84.44444274902344% at epoch 334
execution time: 103.2381640625s
```

DeepConvNet + LeakyReLU (CPU)

```
epoch  490 :
trainig accuracy:  tensor(94.7222)   loss:  tensor(0.6814, grad_fn=<AddBackward0>)
testing accuracy:  tensor(84.1667)

epoch  500 :
trainig accuracy:  tensor(94.4444)   loss:  tensor(0.7221, grad_fn=<AddBackward0>)
testing accuracy:  tensor(84.7222)

LeakyReLU has max accuracy 85.18518829345703% at epoch 392
execution time: 2097.917s
```

從上面的比較圖可以發現有沒有使用 GPU，model execution time 會相差很多倍，可見 GPU 的平行運算在 CNN 的加速效果是非常明顯的。

5. Extra

A. Implement any other classification model

相較於 DeepConvNet，有一種較為簡單的 model 能讓 EEG classification 達到接近的準確度，並且計算量少很多，就是 ShallowConvNet。

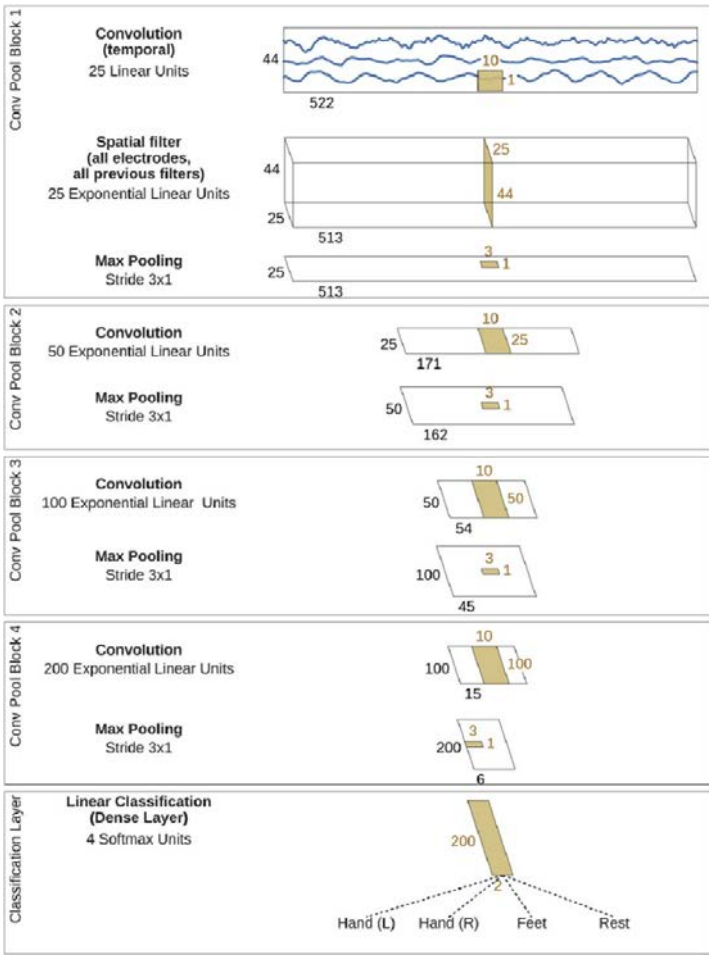
DeepConvNet 是五層的 convolution layer 加上各層的 normalization 和 pooling 和 dropout layer，而 ShallowConvNet 則是只有兩層的 convolutional layers 加上 normalization 和 pooling layer，可見計算量少很多。

下面是 DeepConvNet 和 ShallowConvNet 的比較圖

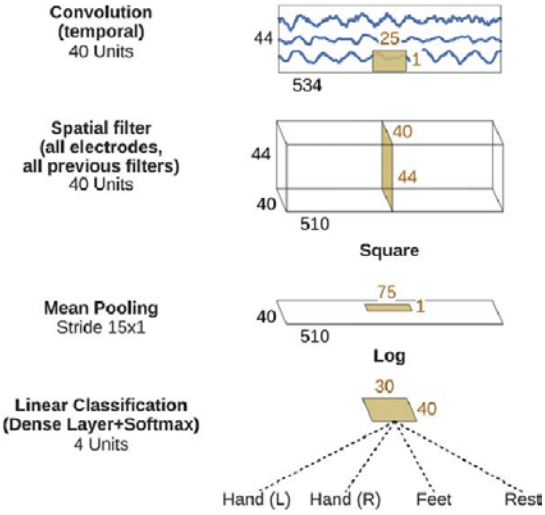
	DeepConvNet	ShallowConvNet
Activation functions	ELU	Square, ReLU
Pooling mode	Max	Mean
Regularization and intermediate normalization	Dropout + batch normalization + a new tied loss function (explanations see text)	Only batch normalization, only dropout, neither of both, nor tied loss
Factorized temporal convolutions	One 10 × 1 convolution per convolutional layer	Two 6 × 1 convolutions per convolutional layer
Splitted vs one-step convolution	Splitted convolution in first layer (see the section "Deep ConvNet for raw EEG signals")	One-step convolution in first layer

下面是 DeepConvNet 和 ShallowConvNet 的示意圖

DeepConvNet



ShallowConvNet

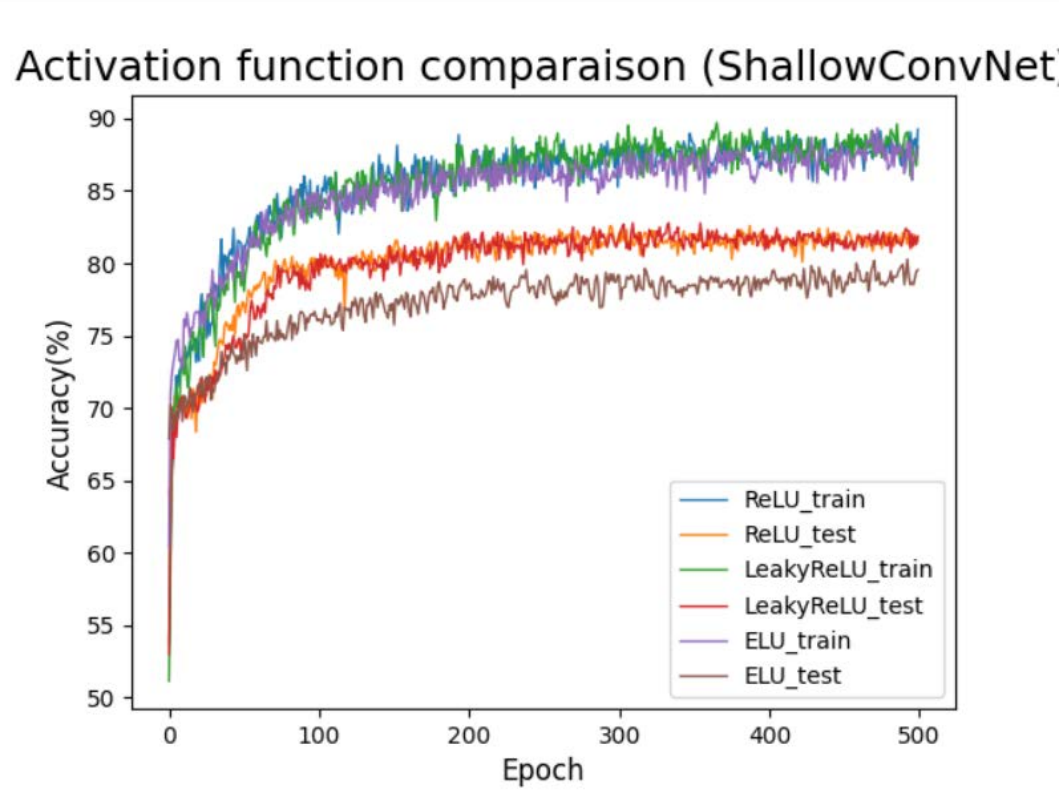


而下面是 ShallowConvNet 的 Comparison figures:

learning rate = 0.001

weight decay = 0.01

epochs = 500



下面是我跑我跑 EEG classification 在 DeepConvNet 和 ShallowConvNet 比較:

DeepConvNet

```
ReLU has max accuracy  84.81481170654297 %
LeakyReLU has max accuracy  85.46296691894531 %
ELU has max accuracy  82.77777099609375 %
```

ShallowConvNet

```
ReLU has max accuracy  82.59259033203125 %
LeakyReLU has max accuracy  82.77777099609375 %
ELU has max accuracy  81.20370483398438 %
```

Testing accuracy comparison:

	ReLU	**LeakyReLU**	**ELU**
DeepConvNet	84.81%	85.46%	82.77%
ShallowConvNet	82.59%	82.77%	81.20%

DeepConvNet + LeakyReLU (GPU)

```
epoch  490 :
trainig accuracy:  tensor(94.9074, device='cuda:0')   loss:  tensor(0.8554, device='cuda:0', grad_fn=<AddBackward0>)
testing accuracy:  tensor(83.7037, device='cuda:0')

epoch  500 :
trainig accuracy:  tensor(95.1852, device='cuda:0')   loss:  tensor(0.7013, device='cuda:0', grad_fn=<AddBackward0>)
testing accuracy:  tensor(82.4074, device='cuda:0')

LeakyReLU has max accuracy 84.4444274902344% at epoch 334
execution time: 103.2381640625s
```

ShallowConvNet + LeakyReLU (GPU)

```
epoch  490 :
trainig accuracy:  tensor(88.0555, device='cuda:0')   loss:  tensor(1.5497, device='cuda:0', grad_fn=<AddBackward0>)
testing accuracy:  tensor(81.2037, device='cuda:0')

epoch  500 :
trainig accuracy:  tensor(89.4444, device='cuda:0')   loss:  tensor(1.3983, device='cuda:0', grad_fn=<AddBackward0>)
testing accuracy:  tensor(81.1111, device='cuda:0')

LeakyReLU has max accuracy 82.77777099609375% at epoch 428
execution time: 50.65135546875s
```

Model execution time comparison:

	ReLU	**LeakyReLU**	**ELU**
DeepConvNet	105.95s	103.23s	103.25s
ShallowConvNet	53.55s	50.65s	50.72s

從上面比較圖，可以看見 ShallowConvNet 雖然沒有比 DeepConvNet 準確度高，但準確度很接近(大概相差 2%-3%)，並且 model execution time 減少了一倍多，代表 ShallowConvNet 能在有限時間內訓練出不錯的 model，適合運用在 real-time 的狀況下。

Reference: Deep learning with convolutional neural networks for EEG decoding and visualization (**https://doi.org/10.1002/hbm.23730**)