DLP
LAB 2

謝侑哲

112550069

Github:

https://github.com/youzhe0305/NYCU-DLP

1. Overview

本次 lab 為，復刻 Spatial Component-wise Convolutional Network (SCCNet)，用來做 Motor-Imagery(MI) EEG 的分類。

MI EEG 為運動心像腦電圖，也就是會在受測者的頭部貼上電極片，用來偵測腦波的釋放，然後讓受測者想像做某些動作，用來觀察人類在執行某些動作前的腦部狀態。

SCCNet 則是針對 MI EEG 分類所提出的，基於 CNN 的模型，以其 spatial filtering 達到良好的分類效果。

此次 lab 中，也使用了三種不同的訓練方式，用來檢測模型的訓練成果，分別是 subject-dependent(SD), leave-one-subject-out(LOSO)以及基於 LOSO 的 Fine-tuning

2. Implementation Details

A. Details of training and testing code

(1) Training Code

Step1 初始化，建立工具:

```
elif os.path.exists('model_weight/model.pt'):
    model = torch.load('model_weight/model.pt') # if model not be trained competely, load model
else:
    model = SCCNet(numClasses=4, device=device, Nu=22, C=22, Nt=1).to(device)  # build a new model, make

dataset = MIBCI2aDataset('train') # get the training dataset
dataloader = DataLoader(dataset, batch_size=hyper_paramaters['batch_size'], shuffle=True) # use dataloade

model.train() # open train mode, make drop layer, normalization layer work
# Adam Optimizer, which decrease learning rate after weight be update
optimizer = torch.optim.Adam(model.parameters(), lr=hyper_paramaters['learning rate'], weight_decay=1e-4)
```

首先，引入在 SCCNet.py 建立好的模型，並且根據輸入的數據給予參數，設定成 train 模式，讓 Dropout Layer, Normalization Layer 能夠正確的運行、更新。如果已經有訓練到一半的資料，則載入繼續訓練。

接著把 Dataloader.py 建立好的 Dataset 引入，並設定為 train 模式，以取得訓練用的 features, labels，得到 Dataset 之後，使用 Dataloader

幫助我們能分 batch 讀取資料，並且做到 Dataset 的隨機化，讓他每次訓練的資料組合不一樣，增加 Robustness。

最後引入 optimizer，這裡使用的是 Adam，是一個結合了 Momentum 跟 Adaptive Grad 的更新方式。Momentum 的算法中，當梯度在做多次接近同向的梯度的更新時，他會前幾次的梯度累積下來，再進行更新，也就是如果都是更新同個方向，他會越更新越快。Adaptive Grad 則是會隨著梯度的更新，調整學習率，在算法中，會根據已經更新的梯度的 L2 Norm 為分母，讓 learning rate 倍數下降。Adam 則是兩個都採用，變成梯度會同時乘上 momentum 並除以累積梯度的 L2 Norm。

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\frac{\partial L_t}{\partial W_t}$$

$$v_t = \beta_1 v_{t-1} + (1-\beta_2)(\frac{\partial L_t}{\partial W_t})^2$$

(m 為 momentum, v 為計算累積梯度的平方，兩者都有參數可以調整累積的速度)

$$W \leftarrow W - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

(乘上 momentum，除以根號 v，即可得到 Adam)

Step2 訓練過程

```python
n_epoch = hyper_paramaters['n_epoch']
loss_log = [] # record training loss for loss plot
log_point = [] # record time point for loss plot
max_accuracy = 0 # for model to find the max acc
for epoch in range(n_epoch):
    epoch_loss = [] # each batch will generate loss, epoch_loss record them for compute average value
    for idx, (batch_features, batch_labels) in enumerate(dataloader):
        batch_features = batch_features.to(device) # make data tensor on chose device (normaly GPU)
        batch_labels = batch_labels.to(device)
        output = model(batch_features) # get output for loss caculate

        loss = model.get_loss(output, batch_labels) # compute cross-entropy loss
        epoch_loss.append(loss.item())
        loss.backward() # backpropagation from loss tensor
        optimizer.step() # update parameter tensor by the .grad attribute get in backpropagation
        optimizer.zero_grad() # clear .grad attribute to avoid accumulation

    avg_epoch_loss = sum(epoch_loss) / len(epoch_loss) # compute average loss of exch batch in a epoch
    if (epoch+1)%1 == 0 or epoch==0:
        print(f'epoch: {epoch+1}, loss: {avg_epoch_loss}') # log
        acc = accuracy_explore_tool(model)
        if acc > max_accuracy: # save the best accuracy model
            max_accuracy = acc
            torch.save(model, 'model_weight/model.pt')
```

訓練過程中，首先從 hyperparameter 中，取得訓練的 epoch 數量，並且用 for 迴圈重複此數量的訓練，接著 enumerate dataloader，每次取得一個 batch 的資料，再把這些資料丟進 model，觸發__call__() function，進而呼叫 forward() function，得到神經網路的輸出。

拿 label 跟輸出做比較，使用 model 的 get_loss function，其中包含了 Cross Entropy 的計算，取得 Loss tensor 之後，可以從 loss 開始做 backpropagation，把梯度累積在每個參數的.grad attribute 上，接著 optimizer 會把 model 的參數都做更新，利用前面提到的 Adam 方法，把.grad 作為這次更新的梯度使用，之後把梯度清空，避免一次更新的梯度累加到下一次，影響結果。

最後，做 validation 測試每個 epoch 的 accuracy，把 model 中，accuracy 最好的參數保留下來。經過多次重複，訓練就完成了。

(2) Testing Code

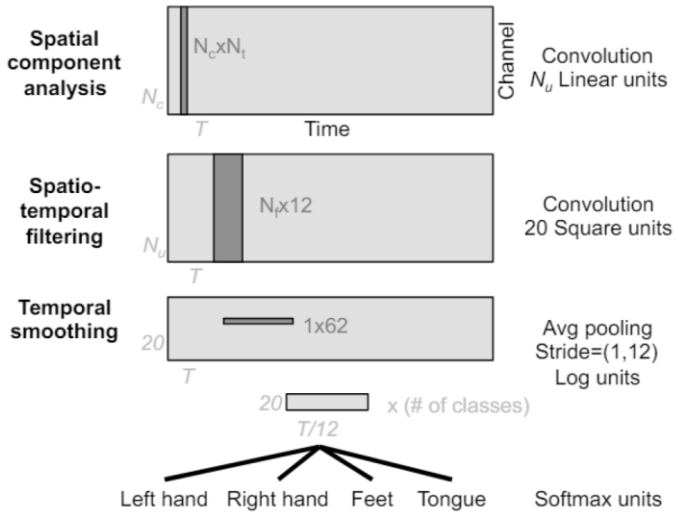
Testing Code 的部分，大致上跟 Training Code 的部份一樣，初始化的部分只差在把 dataset 改成 test，以及不會重新建一個 model，而是直接載入參數。因此我這裡只針對對於資料與模型的運作闡述

```
for i, (features, labels) in enumerate(dataloader):
    idx += 1
    features = features.to(device)
    labels = labels.to(device)

    print(f'subject {idx}')
    output = model.forward(features)
    class_prob = F.softmax(output, dim=1) # dim to provide the dim softmax should process (may be
    prediction = torch.argmax(class_prob, dim=1)

    loss = model.get_loss(output, labels)
    all_loss.append(loss)

    correct_case = torch.sum( (prediction == labels).type(torch.int) ).item()
    subject_acc = correct_case / labels.shape[0]

    all_correct_case += correct_case
    all_sample += labels.shape[0]

print(f'test accuracy: {all_correct_case/all_sample * 100}%, loss:{sum(all_loss)/len(all_loss)}')
```

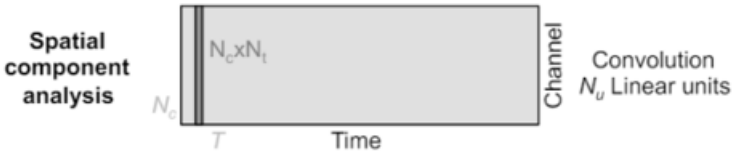
首先，透過 enumerate dataloader，取得一個 batch 的資料，接著把資料丟進 model 做 forward propagation，然後把取得的資料丟進用來計算多種類分類問題的機率的 softmax function，並把 class prediction 設為機率最高的那個。接著比對跟 label 的差異，把每個 batch 的正確數量加在一起，除以總數，即可得到 accuracy。

B. Details of the SCCNet

SCNNet 主要是用來處理 MI EEG 腦電圖的神經網路，是基於 CNN 的神經網路，網路架構如下，channel 為電極通道，Time 為 EEG 的攝影時間，可以整理成一張類似於二維圖的矩陣。網路部分則主要分成四個部分：



第一部分 Spatial component analysis:



這部分主要做電極分布的捲積，使用 spatial filter 作為捲積核，並使用了 Nu 個捲積核，這個 filter 的特色是，他的高度 N_c 正好跟 channel 的高度相同，也就是會考慮一段時間內，每一個電極的特徵。另外捲積核的寬度 N_t 則是代表考慮多長時間段的 EEG 資料，不過在後文的實驗結果中有提到，N_t 的調整在統計方法 ANOVA 中，並未通過 alpha=0.05 的假設檢定，因此 N_t 直接設成 1 就行，減少整體的參數量。也因為 N_t 不影響準度，所以這一層基本上就是只做電極分布狀況的特徵提取。實作如下:

```
self.first_block = nn.Sequential(
    nn.Conv2d(in_channels=1, out_channels=Nu, kernel_size=(C, Nt), padding_mode='zeros', padding=Nt-1)
    nn.BatchNorm2d(Nu),
)
```

縱軸為電極，橫軸為時間的 EEG，可以視為一張 feature channel=1 的 2D 圖形，因此這裡設計成 in_channel=1，其他部分則照 paper 的設計，把 out_channel, kernel_size 設計成同樣的大小。

另外，根據下圖 paper 節錄所述，在第一層與第二層，都使用 Zero-padding 與 batch normalization，因此在 Conv Layer 多加上了 padding。並加上了 batch normalization layer

nents. Zero-padding and batch normalization were applied to both the first and second convolutions, as well as ℓ_2

第二部分 Spatial-temporal filtering:

這部分如其名，就是多考慮了時間特徵的捲積，因此這裡的 kernel 跟上一層的主要區別在於，寬度設成了 12，也就是 0.1 秒。透過考慮了時間特徵的捲積，用來找出不同時間的電極分布的關係(inter-component correlation)，又稱為 spectral kernel。另外在該層後面，也使用了 square layer，來提取資料的功率(power)，因為 power 的變化是 EEG 資料裡面，最顯著的特徵之。該部分的實作如下:

```
self.secood_block = nn.Sequential(
    nn.Conv2d(in_channels=1, out_channels=20, kernel_size=(Nu, 12), padding_mode='zeros', padding=(0
    nn.BatchNorm2d(20),
    SquareLayer(),
    nn.Dropout(dropoutRate)
)
```

第一層使用了如上文所述的捲積，並用了 20 個捲積核及 zero-padding，接著用 square layer 提取 power 特徵，最後補上 Dropout layer，用來增加 robustness，dropout rate 則如下圖論文節錄所示，使用 0.5

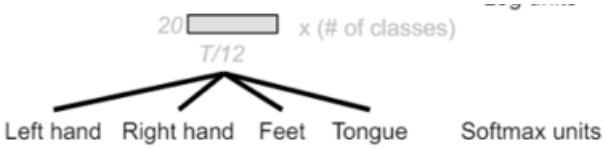
Next, we applied dropout with a rate of 0.5 to prevent over-fitting.

第三部分 Temporal smoothing:

這部分主要是用來把各個時間段的數據平滑化，並且降維，特徵能夠以更平滑的方式被提取出來，並且只保留重要的特徵，而方法就是用 size=(1,62), stride=(1,12)的 Average Pooling，也就是每 0.5 秒平均成一段，做完一次前進 0.1 秒。換句話說，0~0.5, 0.1~0.6, 0.2~0.7...等時段，會被平均變成下一層的 feature。在 pooling 後面也接上了一層 log layer，除了數據變得更加平滑之外，EEG 的功率通常是呈對數分布，這樣可以更好的提取特徵(ex: 常有 micrometer 比之 millimeter 的程度差異，這時候用 log 就可以更好的分析出差別)。實作如下，跟前文敘述同:

```
self.third_block = nn.Sequential(
    nn.AvgPool2d(kernel_size=(1,62), stride=(1,12)),
    LogLayer()
)
```

第四部份 Classifier:



這部分在 paper 中沒有寫得很明確，所以我預設他是按照一般 CNN 的方法，在最後接上一層 fully connection layer，並將 output features 設為 class 的數量，之後再利用多種類分類常用的 softmax 作為 output unit，計算每個 class 的機率，進一步選擇機率最高的 class 作為給出的預測。實作如下:

```
self.fc = nn.Sequential(
    nn.Linear(in_features= 20*int((438-62)/12 + 1), out_features=4),
)
```

在 forward 的部分，只放了 fully connection layer，因為 Pytorch 的 Cross Entropy 的輸入要是 classifier 的輸出，而不是經過 SoftMax 的輸出，因此只放入一層 Linear Layer，SoftMax 則是在 test 的過程中，另外去做。

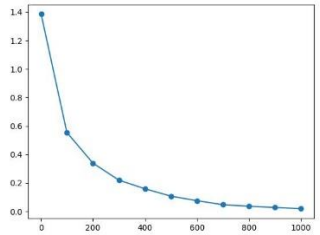
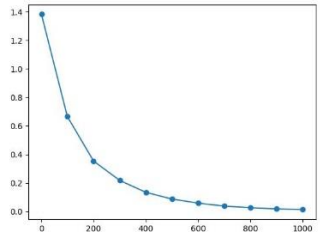
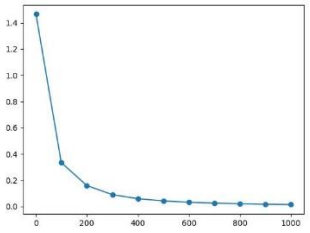
3. Analyze on the experiment results

A. Discover during the training process

在訓練過程中，有許多地方可以討論，以下我將分 n 個部分，分別討論(1) training loss 與 accuracy 的關係、(2)參數調整與 accuracy 關係

(1) training loss 與 accuracy 的關係:

首先看到以下三種訓練的訓練結果圖，三張圖分別訓練了 1000 epoch，並且每 100 個 epoch 記錄一個點

LOSO training loss	SD training loss	Fine-tune training loss
		

圖中可以發現三張圖的 training loss 是逐步的下降的，所以我預期得到的 accuracy，應該要是越來越高的，因此一開始都是用 train 到最後的結果來觀察訓練成效。

但是訓練出來的效果遲遲上不去，後來在每一次的 epoch 後面都做一次 validation 之後，發現這個資料集很容易發生 overfitting 的問題，因此我練出來的最高的 accuracy 通常會落在 100~200 epoch 之間，但是不太穩定的出現在某一個位置。

發現這個問題之後，我就改變了訓練的方式，首先是把訓練的 epoch 調低，以便更快的看出成果，其次就是把訓練的參數紀錄，設成條件式，每當 validation 出現了更高的 accuracy 時，就將其存起來，而不是只保存訓練到最後的 training loss 最低的參數，避免得到的是 overfitting 的解。

(2) 參數調整與 accuracy 關係

訓練過程中，我多次嘗試調校各種參數，嘗試不同參數的效果，主要調校 learning rate, batch size, dropout rate, regularization

- Learning rate: 最佳解: 0.001
  - 在 learning rate 的調校上，我嘗試了 0.1~0.00001 的範圍，以輪替除 2 除 5 的方式嘗試，並針對各個 model 去做訓練，最後訓練出來的結果表示，0.001 在使用 Adam 訓練時 SCCNet 的表現是最好的，這或許也解釋了為甚麼 pytorch API 預設的 lr 會是 0.001
- Batch size: 最佳解 288
  - 因為我的 dataset 儲存資料的方式，是把所有 subject 的每個 trial 放在一起，成為一個 3 維的 ndarray，所以一個 epoch 的 sample 數量會是 288*16(8 個 subject 時)。其中我嘗試了 8~288*4 的 batch size，以每次乘 2 倍，發現導致最好 accuracy 的參數是 288。在 batch size 的選用上，過小的 batch size 會導致算出的梯度隨機性更高，波動更大，但相對在經過足夠的訓練之後，generalization 的能力會上升，但隨機的波動會導致 accuracy 收斂時，比較不穩定，當 global best 的坑比較小的，小 batch size 就會比較難收斂進去。大的 batch size 則相反，計算更加平滑，但 generalization 的能力會下降，卻更容易收斂到 global best。基於上述原因，我嘗試出來的 batch size 在 288 這個不太大也不太小的值訓練的最好。
- dropout rate: 最佳解 0.5
  - 我嘗試了 0, 0.3, 0.5, 0.7, 0.8, 0.9 的 dropout rate，發現在 0.5 表現得最好，就跟 paper 中提到的一樣，另外令人意外的是，在 0.8 的表

現是第二好的。關於這樣的結果，我不是很確定原因，0.5 的 dropout rate 表現得最好，可能是因為剛好可以達到不錯的 generalizations 能力，又不會在傳播的過程中損失太多資訊。至於 0.8 是次好的原因，我推測是因為高的 dropout rate 會讓模型變得更不穩定，但在我的測試結果中 0.8 的 dropout rate 剛好擲骰子擲到了比較好的 accuracy，但實際上跟 0.7 應該是差不多甚至更低的。

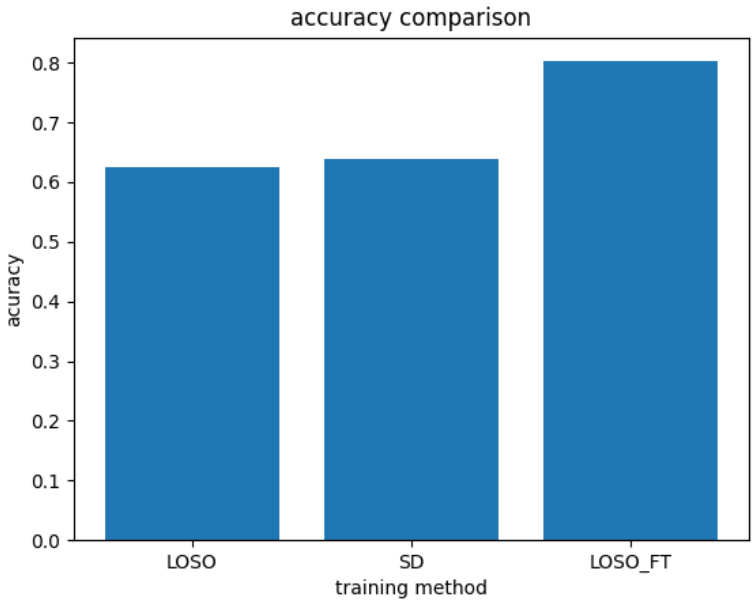
- regularization: 最佳解 weight decay 1e-4
  - 我嘗試了 1, 1e-2, 1e-4, 1e-6 等 regularization，得到的結果中是 1e-4 最好，跟 paper 中提到的一樣，1e-4 是個能減少 overfitting 卻又不會讓模型過於簡單的解。不過有趣的是，我有試著讓模型在 1 的狀況下多跑幾遍，因為此時的模型過於簡單，導致得到的 accuracy 有很大的改變，會讓出來的結果變得像擲骰子一樣，有時候還會意外的擲出很高的數值。

B. Comparison between the three training methods

首先針對三個訓練方法分別簡介:

- LOSO: leave-one-subject-out, 9 個 subject 中，拿其中 8 個做訓練，剩下的一個用其第二個 session 做測試。
- SD: subject-dependent, 跟 LOSO 差不多，但是把 test subject 的第一個 session 用來做訓練
- FT: 把 LOSO 訓練出來最好的結果，拿來對於 test subject 的第一個 session 做 fine tuning

三者在訓練及參數調整之後的 accuracy bar chart 如下:



數值為: LOSO(0.625), SD(0.6393), FT(0.8021)

排序是 LOSO_FT > SD > LOSO。

首先，關於 LOSO，算出來的結果大致與 paper 的結果符合，在 0.6 附近，作為基準值來看其他訓練方法應該是合理的

LOSO_FT 最高是因為經過 LOSO 訓練之後，model 已經有基本提取出 EEG 特徵的能力，之後在針對 test subject 的其中一個 session 做單一足量的訓練，可以讓出來的 model 提取該 subject 特徵的能力被特化，儘管泛化性會下降，但給出了 80.21%的高表現。

SD 的表現比預期的要差，我本來以為他會比 LOSO 高出很多，因為訓練過程中，有把 test subject 的一個 session 納入訓練資料中，但看起來結果並沒有高出太多，雖然 paper 中的數據也沒有高出太多，但只高出了 0.0143，這甚至可以被認為是誤差了。關於 SD 低於 FT 高於 LOSO 的原因很明顯，就是多了針對 test subject 的特化，但是因為是在一般訓練過程中，跟其他資料一起訓練的，因此特化的作用被稀釋，得出的 accuracy 就比較低。不過只高出一點點，我猜測可能的原因是，我在做參數的調整時，調到的最佳參數跟 paper 中的不太一樣，所以導致 LOSO 偏高 SD 偏低，就導致了雖然 paper 中的差距小，但我做出來的數據更小的狀況。

4. Discussion

A. What is the reason to make the task hard to achieve high accuracy?

在訓練的過程中，讓我最難優化模型的 accuracy 的原因，主要是訓練出來的 model 很容易 overfitting，也因此在 3-A 中我討論的參數，大多跟模型會不會 overfitting 或 generalization 的能力有關。我認為會造成容易 overfitting 的原因有幾個:

- 資料量不足:
  - 因為人腦的神經元數量有上百億個，放出的電波的也非常多樣，上文中也有提到，出現的波長可能是微米級與毫米級的差距。相較變化如此大的 EGG，我們能用來訓練的資料量卻只有 4000 多個，並且局限於 9 個受測者身上，這對於模型要去很好的泛化到所有人的 EEG 是非常困難的，資料量不足也就成為容易 overfitting 的主因之一。
- 參數不對:
  - 如上文所提，我調整的參數大多跟避免 overfitting 有關，但是我調整參數的方式並沒有什麼特別的根據，就是憑感覺照某個倍數去調，因此可能會發生我調參數的過程中，忽略掉某些可以很好避免 overfitting 的解。另外，我調參數的方式也是一個一個調，並沒有嘗試一次調一整組，類似於 Grid Search 的方式。
- Non-linear 能力不足
  - 在 SCCNet 中，可以發現網路中並沒有任何的 activation function 作為 hidden unit，這可能會導致 model 做出的結果不夠 non-linear，這雖然不太會導致 overfitting，卻可能導致 model 在 SD, FT 等已經放入 test subject 的訓練中，特化能力下降，導致 accuracy 沒辦法進一步提升。

B. What can you do to improve the accuracy of this task?

針對 4-B 提到的部分，這裡再分述有何解決辦法:

- 資料量不足:
  - 針對這個問題，我認為可以嘗試使用 data augmentation 的方法，像是小幅度在時間軸上平移資料之類的，增加模型的 generalization 能力。
- 參數不對:
  - 針對這個問題，我認為可以引入一些自動大量的調整參數的方法，諸如 GridSearch, Bayesian optimiazation 等等方法，讓電腦能夠自己去運算出最好的參數解
- Non-linear 能力不足
  - 針對這個問題，我認為可以在 layer 之間引入不同的 activation function 嘗試，像是 Leaky ReLU, PReLU 等等。