Lab2: EEG Motor Imagery Classification

313551055 柯柏旭

Overview

這份報告介紹了 SCCNet 用於腦電波 (EEG) 運動意象分類的實現和評估，重點在於如何克服三個數據集（LOSO、SD 和 LOSO finetune）中的過擬合問題。訓練和測試過程包括數據加載、模型訓練、學習率調整、模型保存和結果可視化。關鍵策略包括調整訓練迭代次數（epochs）、dropout 率和卷積層參數，以提高模型準確率。儘管面臨過擬合問題，但這些調整使模型性能顯著提升，特別是在 LOSO finetune 資料集上。報告強調了數據增強、參數調整和特定結構變更在提高模型性能方面的有效性。

Implementation Details

Details of training and testing code

Training code 按照三種類型的資料集共分成三個部分，分別有 `train_sccnet_LOSO`、`train_sccnet_SD` 和 `train_sccnet_LOSOFT` 三個 function，內容其實大同小異，主要內容如下，以 `train_sccnet_LOSO` 為例:

```
def train_sccnet_LOSO(num_epochs, batch_size, learning_rate,
dropout_rate, model_save_path):
    # dropout = 0.8, lr = 0.001, epoch = 80, acc = 60.07
    # 設置設備 (GPU 或 CPU)
    device = torch.device('cuda' if torch.cuda.is_available() else
'cpu')
    print(device)

    # 加載數據集
    train_dataset = MIBCI2aDataset(mode='LOSO_train')
    train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)

    # 創建模型
    model = SCCNet(numClasses=4, timeSample=438,
 dropoutRate=dropout_rate).to(device)
```

```
    # 定義損失函數和優化器
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=learning_rate,
weight_decay=0.0001)
    scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=50,
gamma=0.9)

    # 訓練模型
    model.train()
    loss_history = []  # 用於存儲每個 epoch 的損失
    for epoch in range(num_epochs):
        running_loss = 0.0
        for i, (inputs, labels) in enumerate(train_loader):
            inputs, labels = inputs.to(device), labels.to(device)

            # 梯度歸零
            optimizer.zero_grad()

            # 前向傳播
            outputs = model(inputs)

            # 計算損失
            loss = criterion(outputs, labels)

            # 反向傳播
            loss.backward()

            # 更新權重
            optimizer.step()

            # 累加損失
            running_loss += loss.item()

        avg_loss = running_loss / len(train_loader)
        loss_history.append(avg_loss)  # 記錄當前 epoch 的平均損失
```

```
        # 打印每個 epoch 的損失和學習率
        current_lr = optimizer.param_groups[0]['lr']
        print(f'Epoch [{epoch + 1}/{num_epochs}], Loss:
{avg_loss:.4f}, Learning Rate: {current_lr}')

        # 更新學習率
        scheduler.step()

    # 保存模型權重
    torch.save(model.state_dict(), model_save_path)
    print(f'Model saved to {model_save_path}')

    # 繪製損失曲線
    plot_loss_curve(loss_history, title='Training Loss Curve')
```

與另外兩個 function 不同的地方只有 leanring rate scheduler 的參數和加載 dataset 的 mode 而已。

這個 training function 包含了數據加載、模型訓練、學習率調整、模型保存和訓練過程可視化。

Testing code 如下:

```
def test_sccnet(model_path, batch_size, mode):
    # 設置設備 (GPU 或 CPU)
    device = torch.device('cuda' if torch.cuda.is_available() else
'cpu')

    # 加載數據集
    if mode == 'LOSO' or mode == 'FT':
        test_dataset = MIBCI2aDataset(mode='LOSO_test')
    elif mode == 'SD':
        test_dataset = MIBCI2aDataset(mode='SD_test')

    test_loader = DataLoader(test_dataset, batch_size=batch_size,
shuffle=False)

    # 創建模型
```

```
    model = SCCNet(numClasses=4, timeSample=438,
dropoutRate=0.5).to(device)

    # 加載模型權重
    load_model(model, model_path)

    # 設置模型為評估模式
    model.eval()

    # 初始化變量來計算準確率
    correct = 0
    total = 0
    all_preds = []
    all_labels = []

    # 不需要計算梯度
    with torch.no_grad():
        for inputs, labels in test_loader:
            inputs, labels = inputs.to(device), labels.to(device)

            # 前向傳播
            outputs = model(inputs)
            outputs = nn.functional.softmax(outputs, dim=1)
            _, predicted = torch.max(outputs.data, 1)

            # 累計正確預測和總樣本數
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            all_preds.extend(predicted.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())

    # 計算準確率
    accuracy = correct / total * 100
    print(f'Accuracy of the model on the test dataset:
{accuracy:.2f}%')
```

```
    # 繪製混淆矩陣
    plot_confusion_matrix(all_labels, all_preds, classes=['Left
Hand', 'Right Hand', 'Feet', 'Tongue'], normalize=True)

    return accuracy
```

三份資料集都使用這個 function 測試，根據模式 (LOSO、FT 或 SD) 加載相應的測試數據集，並使用 DataLoader 進行批次處理。

Detail of the SCCNet

```
class SCCNet(nn.Module):
    def __init__(self, numClasses=4, timeSample=438, Nu=44, Nc=22,
dropoutRate=0.5):
        super(SCCNet, self).__init__()

        # First convolutional block: Spatial component analysis
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=Nu,
kernel_size=(Nc, 16), stride=1, padding=0)
        self.bn1 = nn.BatchNorm2d(Nu)

        # Second convolutional block: Spatio-temporal filtering
        self.conv2 = nn.Conv2d(in_channels=Nu, out_channels=20,
kernel_size=(1, 12), stride=1, padding=(0, 11))
        self.bn2 = nn.BatchNorm2d(20)
        self.square2 = SquareLayer()
        self.dropout2 = nn.Dropout(dropoutRate)

        # Pooling layer: Temporal smoothing
        self.pool = nn.AvgPool2d(kernel_size=(1, 62), stride=(1, 12))

        # Fully connected layer
        pooled_time = (timeSample - 62) // 12 + 1
        self.fc = nn.Linear(in_features=20 * pooled_time,
out_features=numClasses)

    def forward(self, x):
        # Add a channel dimension to the input (加上 batch dimension)
```

```
        x = x.unsqueeze(1)
        # print(f'After unsqueeze: {x.shape}')  # 調試輸出形狀

        # First convolutional block
        x = self.conv1(x)
        x = self.bn1(x)

        # Second convolutional block
        x = self.conv2(x)
        # print(f'After conv2: {x.shape}')  # 調試輸出形狀
        x = self.bn2(x)
        x = self.square2(x)
        x = self.dropout2(x)

        # Pooling layer
        x = self.pool(x)
        # print(f'After pool: {x.shape}')  # 調試輸出形狀

        # Flatten the tensor
        x = x.view(x.size(0), -1)
        # print(f'After flatten: {x.shape}')  # 調試輸出形狀

        # Fully connected layer
        x = self.fc(x)
        # print(f'After fc: {x.shape}')  # 調試輸出形狀

        return x
```

```
SCCNet(
  (conv1): Conv2d(1, 22, kernel_size=(22, 16), stride=(1, 1))
  (bn1): BatchNorm2d(22, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (conv2): Conv2d(22, 20, kernel_size=(1, 12), stride=(1, 1),
padding=(0, 11))
  (bn2): BatchNorm2d(20, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (square2): SquareLayer()
  (dropout2): Dropout(p=0.5, inplace=False)
  (pool): AvgPool2d(kernel_size=(1, 62), stride=(1, 12), padding=0)
  (fc): Linear(in_features=640, out_features=4, bias=True)
)
Output shape: torch.Size([32, 4])
```

- **第一卷積塊**：進行 Spatial component analysis，使用卷積層（conv1）和批量歸一化層（bn1）。
- **第二卷積塊**：進行 Spatio-temporal filtering，使用卷積層（conv2）、批量歸一化層（bn2）、自定義平方層（square2）和 dropout 層（dropout2）。
- **池化層**：使用平均池化層（pool）進行時間平滑，減少特徵圖的尺寸。
- **全連接層**：將展平的特徵圖輸入全連接層（fc）進行最終分類。

至於文章中提到最後的 softmax 層，為了確保能精確在訓練過程計算 loss ，因此是在 testing 做分類算 acc 的時候才把它加上去。

Anything you want to mention

之後針對三種不同資料集，除了調整一些訓練的超參數外，SCCNet 的 conv 層的參數也有做調整 (Ex: stride, Nu, Nt...)，但基本上架構都是長得一樣的 。例如上一小節的 SCCNet 為針對 LOSO dataset 的架構。

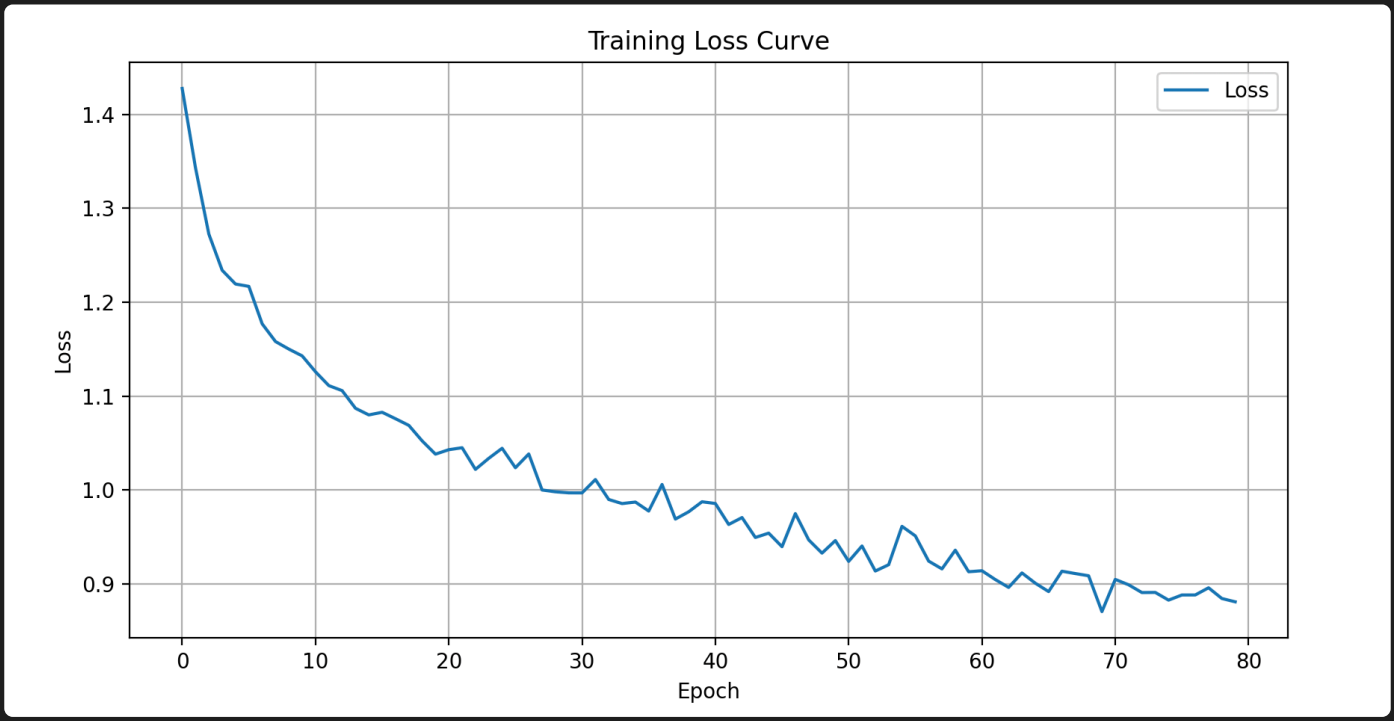
Analyze on the experiment results

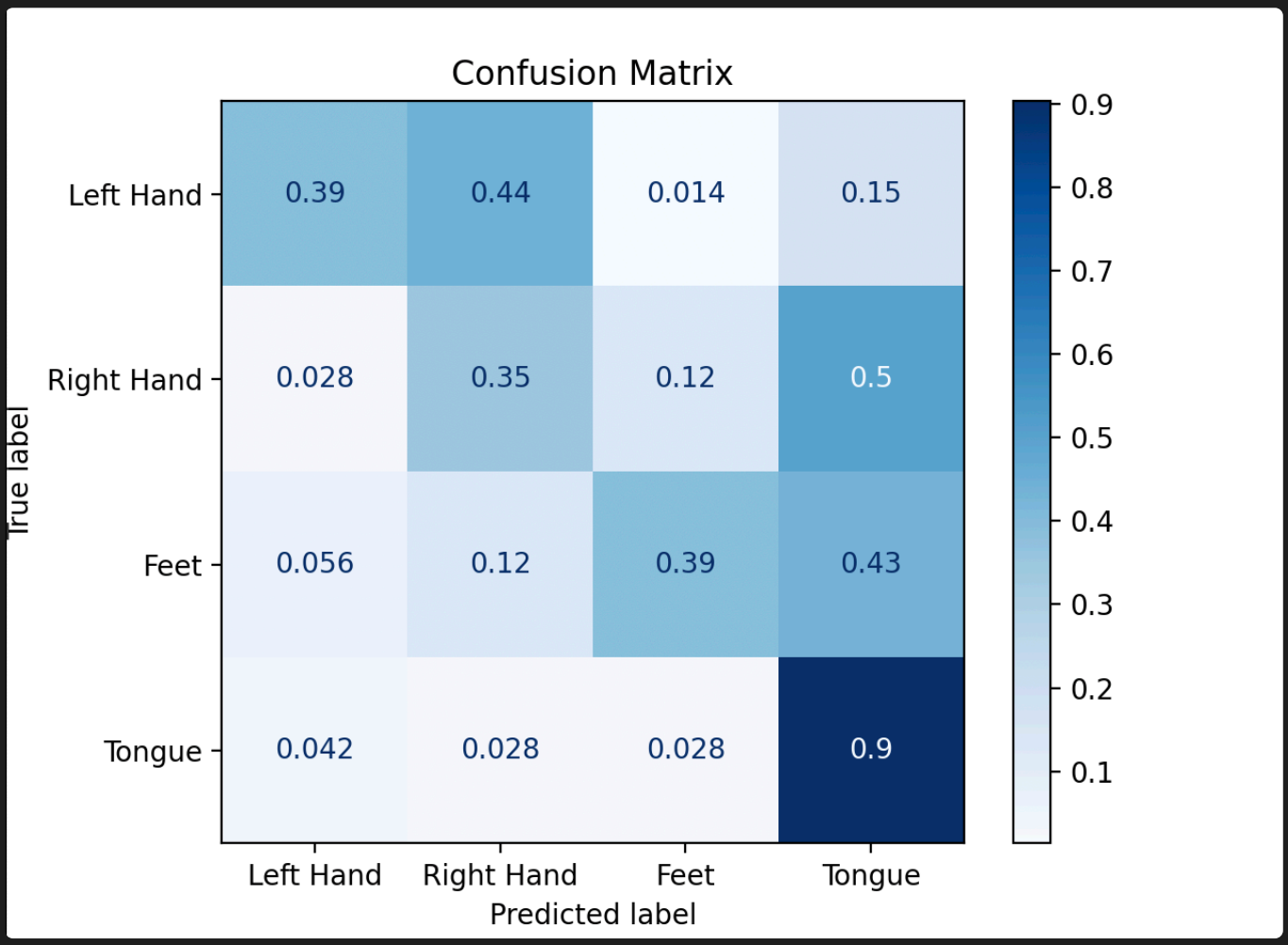
Discover during the training process

- LOSO dataset

Max accuracy = 60.07%, dropout rate = 0.8, training epochs = 80, batch_size = 64, learning rate = 0.001

我發現 LOSO dataset 之所以比其他資料集還要難以訓練，是因為它太容易 overfitting 了，幾乎是隨便訓練都 overfitting。當然其他資料集也存在這個問題，但是 LOSO 很明顯就是在少 epochs 的時候 testing 準確率會比多 epochs 的時候還高，而且表現得非常浮動.....，也所以我對於這個資料集最不同的訓練方式是把他的 epochs 調很低，只有80，但也因為這樣導致準確率很浮動，能達到 60% 運氣成分佔的還蠻多的。



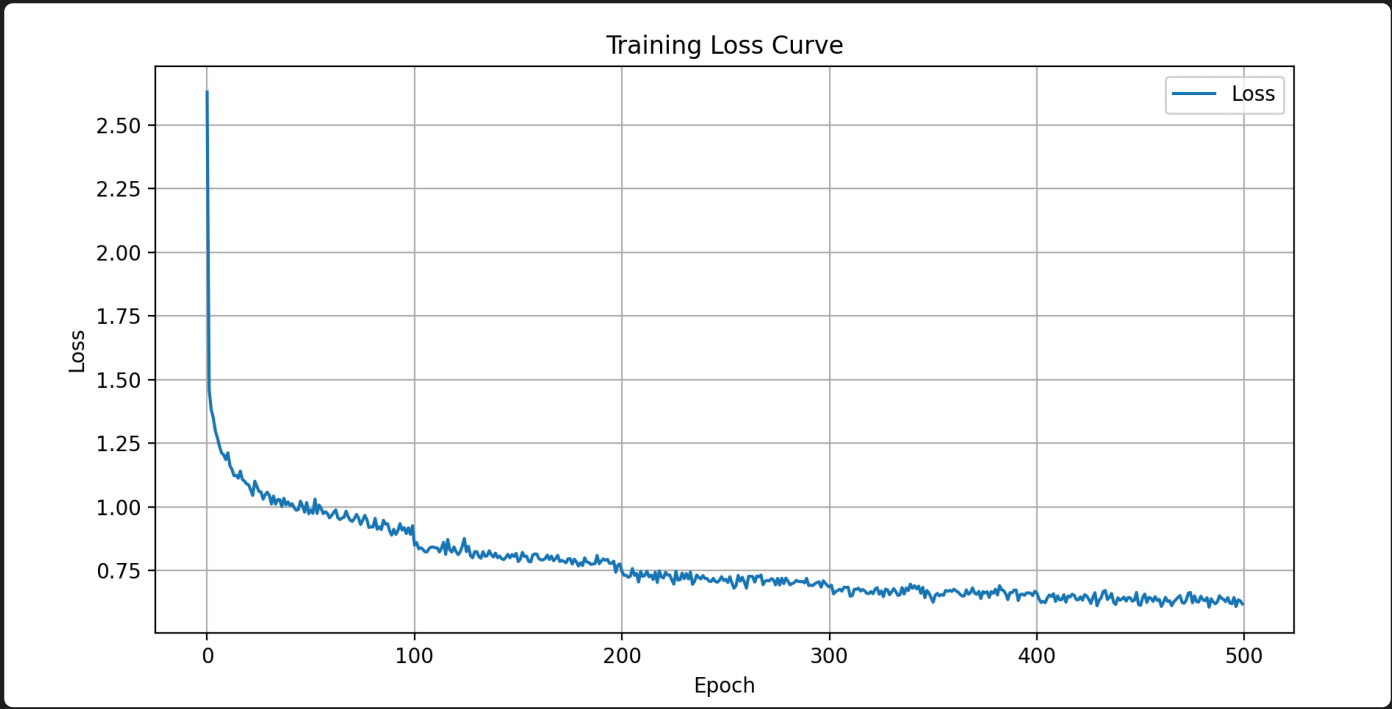


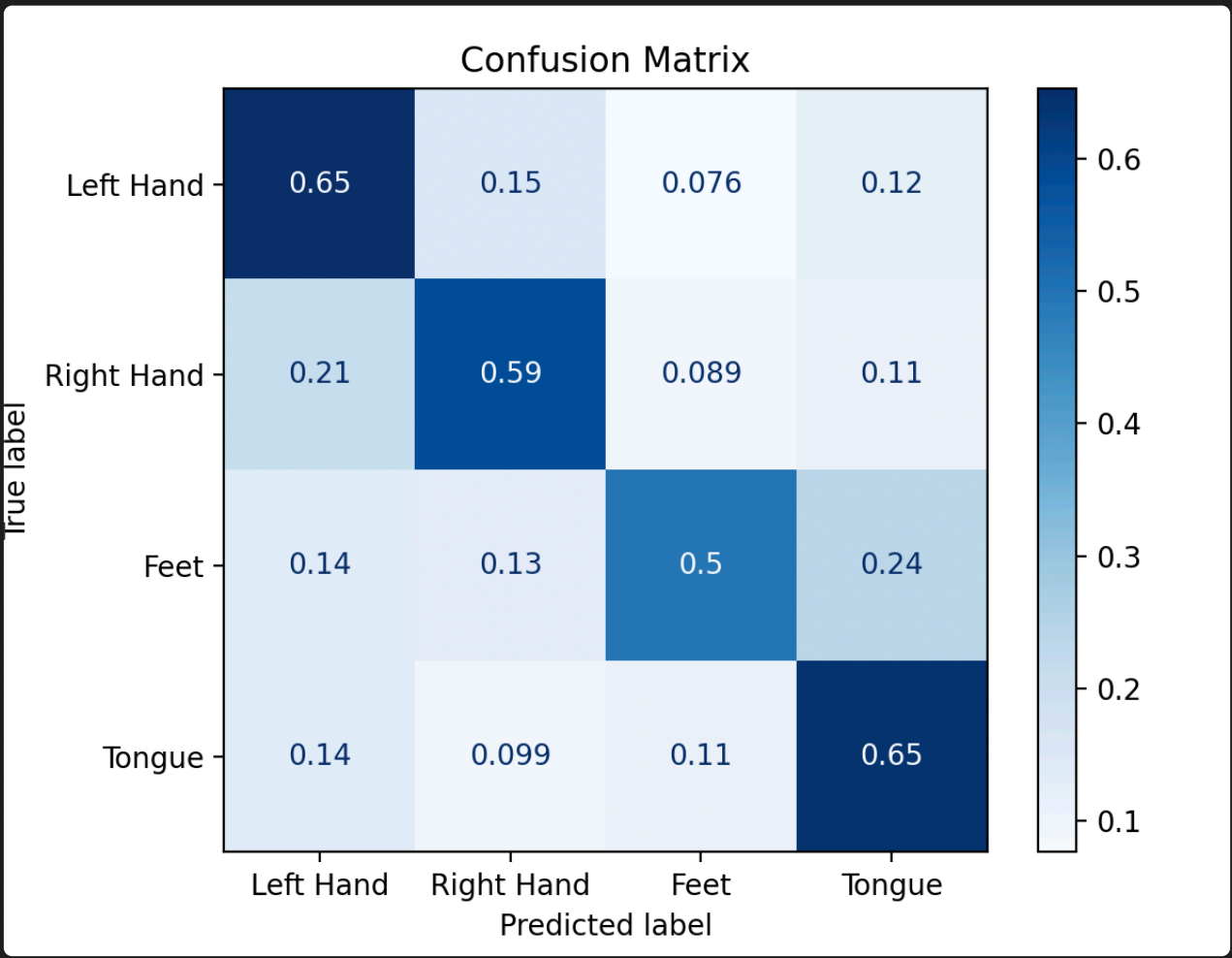
以上為混淆矩陣，從對角線的 entry 來看可以知道舌頭的分辨率比較高，但是模型好像很喜歡猜舌頭，右手和腳幾乎都猜成舌頭了.....。至於左手混淆成右手的機率也蠻高的。

- SD dataset

Max accuracy = 61.02%, dropout rate = 0.8, training epochs = 500, batch_size = 64, learning rate = 0.01

相較於之前的 LOSO 資料集，SD 的準確度會隨著 epochs 的增加穩穩的上升，我推測比起 LOSO，SD 對於 overfitting 的問題比較沒有這麼嚴重（但還是存在，少了 dropout layer 準確率還是差很多），所以比起 LOSO ，SD 的 epochs 我會設的高一些，learning rate 因為有 scheduler，所以一開始也設的蠻大的。



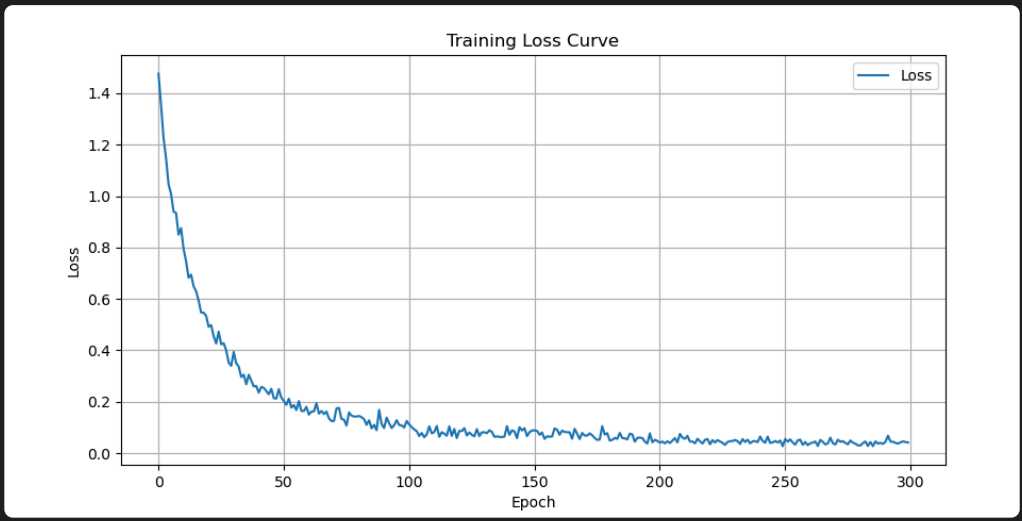


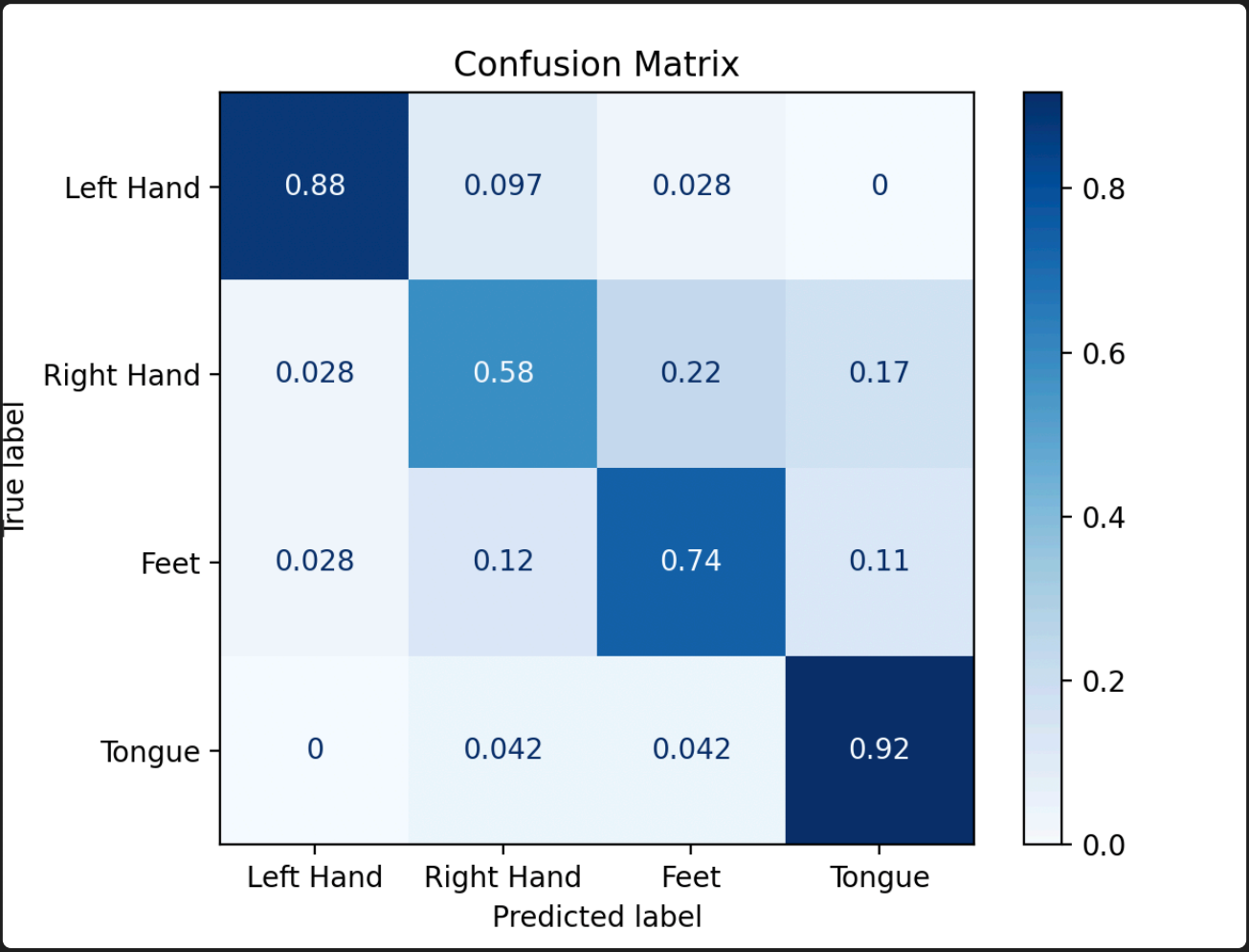
從混淆矩陣來看，比起 LOSO，SD 的分類結果看起來穩定許多(LOSO只有舌頭分得比較好)。

- LOSO finetune dataset

Max accuracy = 80.56%, dropout rate = 0.8, training epochs = 300, batch_size = 64, learning rate = 0.001

LOSO_FT 的資料集真的好訓練許多，推測可能是有經過微調 LOSO 使得特徵變得強之類的，它的 training loss 可以降到 0.0幾之下。





從分類結果可以看到，預測的類別又更加準確了，基本上除了右手以外，模型的表現都比前面兩個還要好許多。

Comparison between the three training methods

- LOSO

先前有提到過，LOSO 的資料集很容易 overfitting，推測原因可能有資料不平衡 (舌頭的樣本很多或是 training 與 testing 的資料集差異過大...)等等.....，所以訓練成果變得有點在吃運氣。但我發現唯一應該去調整的是 epochs 的數量，這也是 LOSO 和另外兩個資料集很大的不同點，LOSO 很明顯會在大約 100 epochs 後隨著 epochs 增加準確率開始下降，因此 LOSO 的 epochs 數量只有設置為 80。至於雖然 overfitting 很嚴重，但我如果把 dropout rate 設在像 paper 一樣的 0.5 的話，準確率不會來的有 0.8 高。

- SD

SD 一樣有 overfitting 的問題，但是並沒有 LOSO 來得嚴重，它的測試準確率會穩穩上升直至平緩。但如果直接像 LOSO 一樣只 train 個 80 epochs，準確率大概只會有 50多%，因此 SD 訓練了大概 300 ~ 500 epochs 左右以避免 underfitting。其餘設置皆和 LOSO 差異不大。

- LOSO finetune

經過微調後的 LOSO 資料集可能達成了數據增強、標籤平衡或是消除了資料中的雜訊等等，總而言之，它變得好訓練許多。然而，要藉由之前的訓練模式調整超參數去達到 80 % 還是相當有難度的。因此，我試著在以不改變模型架構的前提，去調整各層的一些參數。後來發現，第一層的卷積步長(stride)，是我達到 80 % 的關鍵。我將它從 1 調為了 2，準確率便飆升了不少。我推測原因可能有這些: (1) 較大的 stride 會導致每次卷積步驟中跳過更多的像素，這樣每個卷積核會看到更大的輸入範圍。這有利於捕捉整體結構信息，從而提高分類性能。(2) 較大的 stride 減少了特徵圖的尺寸和參數量，從而減少了模型的複雜度。

- Overall comparison

LOSO：需控制 epochs 數量和調整 dropout rate 以防止過擬合。

SD：需增加訓練 epochs 數量，以避免 underfitting，達到穩定的高準確率。

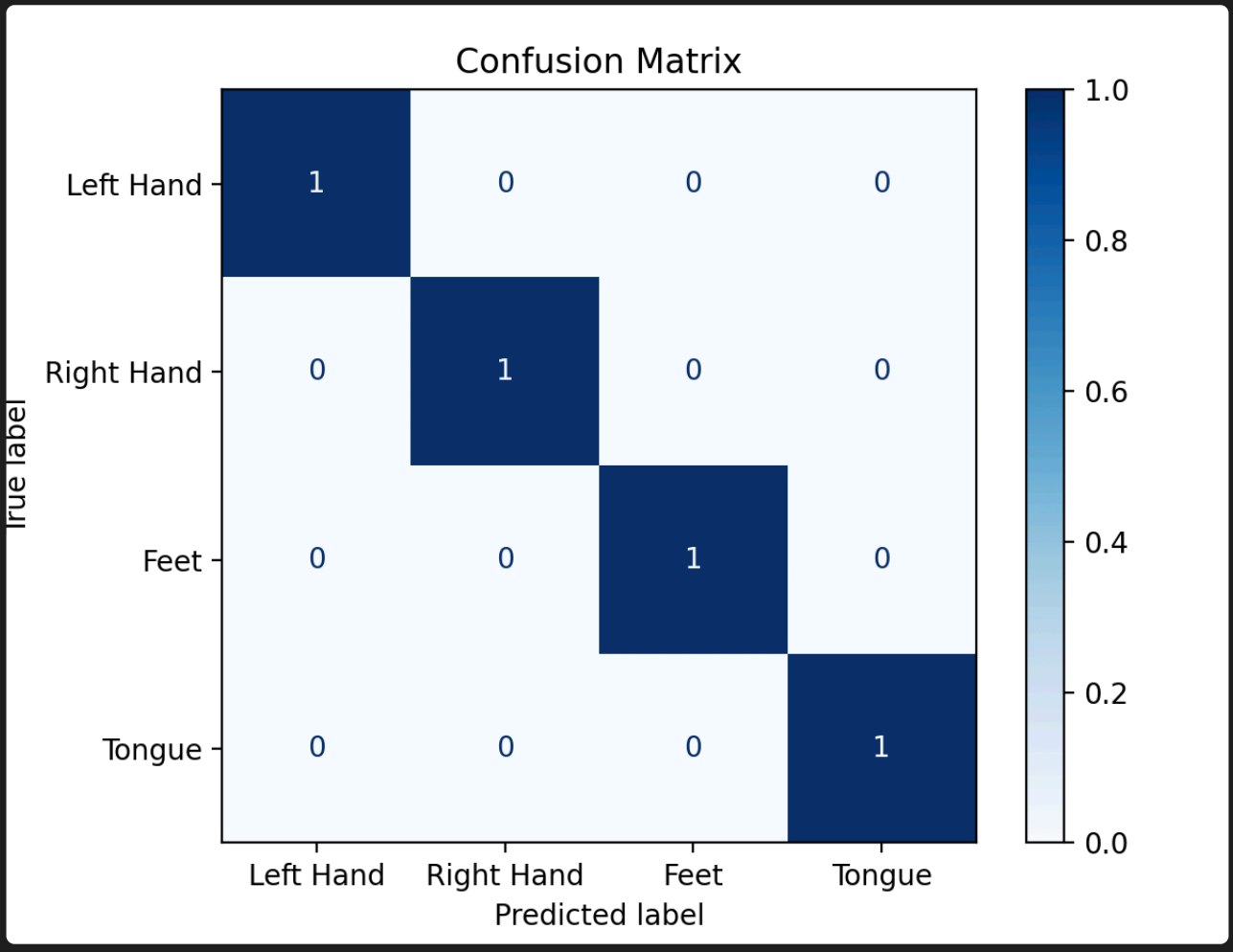
LOSO Finetune：通過調整卷積層參數（特別是 stride），顯著提高準確率。

Discussion

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

針對三筆資料集，總體來說都存在的問題便是 overfitting。之所以會這麼認為，我做了以下兩種實驗: (1) 拿訓練的資料集去做測試、(2) 關掉一些防止 overfitting 的功能。

經過實驗 (1)，我發現不管是 LOSO、SD 還是 LOSO_FT，使用 training dataset 去做測試都幾乎可以達到百分之百的準確率，如下圖:



但 testing dataset 的準確率可差遠了.....。

接著經過實驗 (2)，關掉如 dropout layer、weight decay 等等的功能後，準確率通常也都會大幅下降。

因此我認為導致這三個資料集達到高準確度最根本的原因還是它們太容易 overfitting 了。

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

因此，我主要針對 overfitting 的問題去提高準確度。

首先，造成 overfitting 的可能原因有**資料量不足**，訓練數據集中的樣本數量很少，無法覆蓋數據分佈的多樣性。所以我有試著在訓練之前，對資料做數據增強 (data augmentation)。

```
    def augment(self, feature):
        # 添加隨機噪聲
        if np.random.rand() > 0.5:
            noise = np.random.normal(0, 0.1, feature.shape)
            feature = feature + noise
        # 隨機平移
        if np.random.rand() > 0.5:
```

```
            shift = np.random.randint(-10, 10)
            feature = np.roll(feature, shift, axis=-1)
        # 隨機裁剪並填充
        if np.random.rand() > 0.5:
            crop_size = np.random.randint(400, 438)
            start = np.random.randint(0, feature.shape[-1] -
crop_size)
            cropped_feature = feature[:, start:start+crop_size]
            feature = np.pad(cropped_feature, ((0, 0), (0,
feature.shape[-1] - crop_size)), 'constant')

        return feature
```

但很遺憾的就是，有沒有這個東西準確度好像都沒有顯著的改變.....。

接著，我試著去調整 paper 中本來就存在的防止 overfitting 的功能參數，dropout rate 和 L2 weight decay。後來發現調整 dropout rate 還蠻有用的，每個資料集大概都可以提升個 3 ~ 5%準確度。文章中原本是設置為 0.5 ，但我後來將他調整成 0.8，讓模型在每次訓練中，多丟 30% 的神經元以提高他的穩健性。

最後，我個人認為最有幫助的調整是，改變第一層卷積層的 stride（從 1 提高到 2）。這樣的操作成功讓我的 LOSO_FT 達到 80%的準確率，調和沒調實際上差蠻多的，大概上升了 5 ~ 10 %。我推測這麼做會改善的原因有:

(1) 較大的 stride 減少了特徵圖的尺寸和參數量，從而減少了模型的複雜度。這可以降低 overfitting 的風險，特別是在訓練數據不足的情況下。

(2) 較大的 stride 會導致每次卷積步驟中跳過更多的像素，這樣每個卷積核會看到更大的輸入範圍。這有助捕捉整體結構信息，從而提高分類性能。

不過我也有將 stride 調整成更大過（3 和 4），但這麼做的效果都沒有 2 來得好。

報告結束，謝謝助教批改！