DLP_LAB2

— 313551133 陳軒宇

Overview

這次實驗需要使用 Deep Learning 技術對 Brain-Computer Interface(BCI) 中的運動想像(Motor Imagery)任務進行分類，我們需要實現並訓練一個名為 SCCNet(Spatial Component-wise Convolutional Network) 的深度學習模型，用於分類四種不同的運動想像任務：左手、右手、雙腳和舌頭。

使用的 dataset 是 `BCI Competition IV 2a` ，該資料集包含多個受試者的腦電圖(EEG)信號。每個受試者有兩個 session 的資料，每個 session 包含 288 次試驗，其中每種運動想像類型各有72個試驗。數據已經過預處理，包括去除眼電圖(EOG)信號、帶通濾波(bandpass filter)、歸一化(normalization)和降低採樣率(downsampling)。

本實驗需要實現三種不同的訓練方法，三種方式的差別會在後續介紹。我們需要分別實現這三種方法，訓練對應的模型，並比較它們的性能。

Implementation Details

Details of training and testing code

訓練部分 (trainer.py (http://trainer.py))

1. 定義訓練函數 `train` ，用於訓練 SCCNet 模型。其傳入參數如下：
    - `epochs`
    - `batch_size`
    - `learning_rate`
    - `dropout_rate`
    - `target` ：目標準確率，用於判斷模型是否已經超過目標準確率
    - `optimizer` ：最佳化器，預設為 Adam
    - `scheduler` ：學習率調度器，預設為 StepLR
    - `train_dataset_mode` ：使用的訓練數據集。
    - `test_dataset_mode` ：使用的測試數據集。
    - `base_model_path` ：微調模型的路徑，在 finetune 訓練方法中使用，預設為 `None`
    - `model_path` ：保存訓練過程中模型的路徑
    - `final_model_path` ：最終模型的路徑

```
def train(epochs=1000,
        batch_size=32,
        learning_rate=0.01,
        dropout_rate=0.5,
        target=70,
        optimizer=optim.Adam,
        scheduler=optim.lr_scheduler.StepLR,
        train_dataset_mode=None, test_dataset_mode=None,
        base_model_path=None, # for finetune
        model_path='./model_weight/sd.pt',
        final_model_path='./model_weight/sd_final.pt'):
```

2. 載入資料集：
    - 使用在 Dataloader.py (http://Dataloader.py) 中定義的 `MIBCI2aDataset` 類別加載訓練數據。
    - 通過 DataLoader 將數據批量化，並在訓練時進行 shuffle。

```
train_dataset = MIBCI2aDataset(mode=train_dataset_mode)
train_loader = DataLoader(train_dataset,
                          batch_size=batch_size,
                          shuffle=True)
```

3. 模型初始化:
    - 創建 SCCNet 模型的 instance，並將其移動到可用的設備(GPU 或 CPU)上。

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = SCCNet(dropoutRate=dropout_rate).to(device)
```

4. 定義 loss function, optimizer, scheduler ：
    - 使用 `CrossEntropyLoss` 作為 loss function。
    - 使用 Adam 優化器，並設置學習率為傳入的 `learning_rate` 參數，並根據論文，將 `weight_decay` 設置為 0.0001 (l_2 regularization)
    - 使用 `StepLR` 學習率調度器，每 100 個 epoch 將學習率降低 50。

```
criterion = nn.CrossEntropyLoss()
optimizer = optimizer(model.parameters(), lr=learning_rate,
weight_decay=0.0001)
scheduler = scheduler(optimizer, step_size=100, gamma=0.5)
```

5. 訓練過程：
    - 將模型設置為訓練模式。
    - 在每個 epoch 中，遍歷所有批次數據，進行前向傳播、反向傳播和參數更新。
    - 計算並記錄每個 epoch 的平均損失，並更新學習率。

```
model.train() # set model to train mode

for epoch in range(1, epochs + 1):
    running_loss = 0.0
    for _, (inputs, labels) in enumerate(train_loader):
        inputs, labels = inputs.to(device), labels.to(device)

        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass
        outputs = model(inputs)
        # compute the loss
        loss = criterion(outputs, labels)
        # backpropagation
        loss.backward()
        # update the parameters
        optimizer.step()
        # accumulate the loss
        running_loss += loss.item()

    # compute the average loss
    avg_loss = running_loss / len(train_loader)
    losses.append(avg_loss)
    current_lr = optimizer.param_groups[0]['lr']
    # update the learning rate
    scheduler.step()
```

6. 模型保存和驗證:
    - 保存每個 epoch 的模型，並使用 tester.py (http://tester.py) 中的函數在 validation set 上評估模型性能。
        - 註：為了節省空間且避免發生意外，這裡不保存所有 epoch 的模型，而是在評估模型性能後達到目標準確率或超過目前最高的準確率時，才保存作為最終模型。否則僅保存最後一個 epoch 的模型。
    - 如果達到目標準確率，則提前終止訓練。

```
# save the model
torch.save(model.state_dict(), model_path)
# save the loss history
with open(f"{model_path}.pkl", 'wb') as f:
    pickle.dump(losses, f)

# test the model
acc = sccnet_test(batch_size, test_dataset_mode, model_path)
if acc > max_acc:
    # update the max accuracy
    max_acc = acc
    # save the final model
    torch.save(model.state_dict(), final_model_path)
# if the max accuracy is reached, stop training
if acc > target:
    break
```

7. 微調(finetune)：
    - 提供了加載預訓練模型進行 finetune 的功能，用於 LOSO+FT 訓練方法。

```
if train_dataset_mode == 'finetune':
    # if the base model path does not exist
    if not os.path.exists(base_model_path):
        # raise an error
        raise ValueError(f'Base model does not exist')
    # load the base model
    model.load_state_dict(torch.load(base_model_path))
```

測試部分 (tester.py (http://tester.py))

1. 定義測試函數 `sccnet_test` ，用於測試 SCCNet 模型。其傳入參數如下：
    - `batch_size` ：測試批次大小
    - `mode` ：使用的測試資料集，可以是 `sd_test` 或 `loso_test`
    - `model_path` ：使用的模型檔案路徑
    - `paras` ：使用的模型參數，為了應對參數可能有變化的情況。

```
def sccnet_test(batch_size=32,
                mode="sd_test",
                model_path='./model_weight/model.pt',
                paras={}):
```

2. 載入資料集：

    - 使用在 Dataloader.py (http://Dataloader.py) 中定義的 `MIBCI2aDataset` 類別加載訓練數據。
    - 通過 DataLoader 將數據批量化，但在測試時不需要隨機打亂數據。

```
test_dataset = MIBCI2aDataset(mode=mode)
test_loader = DataLoader(test_dataset,
                         batch_size=batch_size,
                         shuffle=False)
```

3. 載入模型參數：

    - 檢查模型檔案是否存在，如果不存在則 raise 一個錯誤。
    - 由於在函數的傳入參數中已經包含了模型的 instance，所以不需要再次定義模型，只需要將模型參數加載到模型中即可。
    - 將模型設置為評估模式。

```
if not os.path.exists(model_path):
    raise FileNotFoundError(f"Model path not found")

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model.to(device)
model.load_state_dict(torch.load(model_path))
model.eval() # set model to evaluation mode
```

4. 評估模型：

    - 在測試資料集上運行模型，累加資料數量以及正確數量，並保存預測結果與正確結果。
    - 返回正確率。

```
    tot = cnt = 0
    all_preds = []
    all_labels = []

    with torch.no_grad():  # disable gradient calculation
        for inputs, labels in test_loader:
            inputs, labels = inputs.to(device), labels.to(device)

            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)

            tot += labels.size()[0]
            cnt += (predicted == labels).sum().item()

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

    acc = cnt / tot * 100

    return acc
```

Details of the SCCNet

```
SCCNet(
  (conv1): Sequential(
    (0): Conv2d(1, 44, kernel_size=(22, 2), stride=(1, 1))
    (1): Permute2d()
    (2): BatchNorm2d(1, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  )
  (conv2): Sequential(
    (0): Conv2d(1, 20, kernel_size=(44, 12), stride=(1, 1), padding=(0, 6))
    (1): BatchNorm2d(20, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  )
  (square): SquareLayer()
  (dropout): Dropout(p=0.5, inplace=False)
  (avgpool): AvgPool2d(kernel_size=(1, 62), stride=(1, 12), padding=0)
  (fc): Linear(in_features=640, out_features=4, bias=True)
  (softmax): Softmax(dim=1)
)
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 44, 1, 437]           1,980
         Permute2d-2           [-1, 1, 44, 437]               0
       BatchNorm2d-3           [-1, 1, 44, 437]               2
            Conv2d-4           [-1, 20, 1, 438]          10,580
       BatchNorm2d-5           [-1, 20, 1, 438]              40
       SquareLayer-6           [-1, 20, 1, 438]               0
           Dropout-7           [-1, 20, 1, 438]               0
         AvgPool2d-8            [-1, 20, 1, 32]               0
            Linear-9                   [-1, 4]           2,564
         Softmax-10                   [-1, 4]               0
================================================================
Total params: 15,166
Trainable params: 15,166
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.04
Forward/backward pass size (MB): 0.71
Params size (MB): 0.06
Estimated Total Size (MB): 0.81
----------------------------------------------------------------
```

SCCNet (Spatial Component-wise Convolutional Network) 是一個專門為運動想像EEG分類設計的卷積神經網絡模型。其主要特點在於初始卷積層的設計，模仿了傳統EEG分析中常用的空間濾波技術。SCCNet的架構主要包含以下幾個部分：

1. 空間成分分析層 (Spatial Component Analysis Layer)：
    - 使用2D卷積層，輸入通道數為 1，輸出通道數為 Nu (預設為44)
    - 卷積核大小為 (C, N_t)，其中 C 為 EEG 通道數(22)，N_t 為時間卷積核大小(預設為1)
    - 此層的作用類似於傳統EEG分析中的空間濾波，將原始EEG信號轉換為空間成分。
    - 為了適應後續的 BatchNorm2d 操作，在 Conv2d 層之後加入 Permute2d 層，以保持輸入形狀為 $(batch\_size, 1, C, time\_sample)$ 。

```
self.conv1 = nn.Sequential(
    nn.Conv2d(in_channels=1, out_channels=Nu, kernel_size=(C, Nt),
stride=1, padding=0),
    Permute2d((0, 2, 1, 3)),
    nn.BatchNorm2d(1)
)
```

2. 時空卷積層 (Spatio-Temporal Convolution Layer)：

   - 使用2D卷積層，輸入通道數為 1，輸出通道數為 N_c (預設為 20)
   - 卷積核大小為 $(N_u, 12)$，其中 12 對應於 0.1 秒的時間窗口
   - 此層對空間成分進行進一步的時空特徵提取

```
self.conv2 = nn.Sequential(
    nn.Conv2d(in_channels=1, out_channels=Nc, kernel_size=(Nu, 12),
stride=1, padding=(0, 6)),
    nn.BatchNorm2d(Nc),
)
```

3. 平方層 (Square Layer):

   - 對前一層的輸出進行平方運算
   - 這一步驟目的是提取信號的能量特徵，因為頻譜能量變化是運動想像EEG的主要特徵

```
self.square = SquareLayer()
```

4. Dropout層:

   - 使用 dropout 率為 0.5 (可調整)，用於防止 overfitting

```
self.dropout = nn.Dropout(dropoutRate)
```

5. 平均池化層 (Average Pooling Layer):

   - kernel_size 為 $(1, 62)$，stride 為 $(1, 12)$ ，其中 62 對應於 0.5 秒的時間窗口，12 對應於 0.1 秒的時間窗口
   - 此層用於在時間維度上進行平滑處理和降維

```
self.avgpool = nn.AvgPool2d(kernel_size=(1, 62), stride=(1, 12))
```

6. 全連接層 (Fully Connected Layer):

   - 輸入特徵數為 $N_c * (\lfloor \frac{\text{timeSample}-62}{12} \rfloor + 1)$

- 輸出特徵數 4 (對應左手、右手、雙腳和舌頭的運動想像)

```
fc_inSize = Nc * ((timeSample - 62) // 12 + 1)
self.fc = nn.Linear(in_features=fc_inSize, out_features=numClasses,
bias=True)
```

7. Softmax層:

   - 用於將全連接層的輸出轉換為機率分佈

```
self.softmax = nn.Softmax(dim=1)
```

(Optional) Anything you want to mention

Analyze on the experiment results

嘗試了一些不同參數，最好的實驗結果如下(finetune 時使用相同的參數，除了 step_size)：

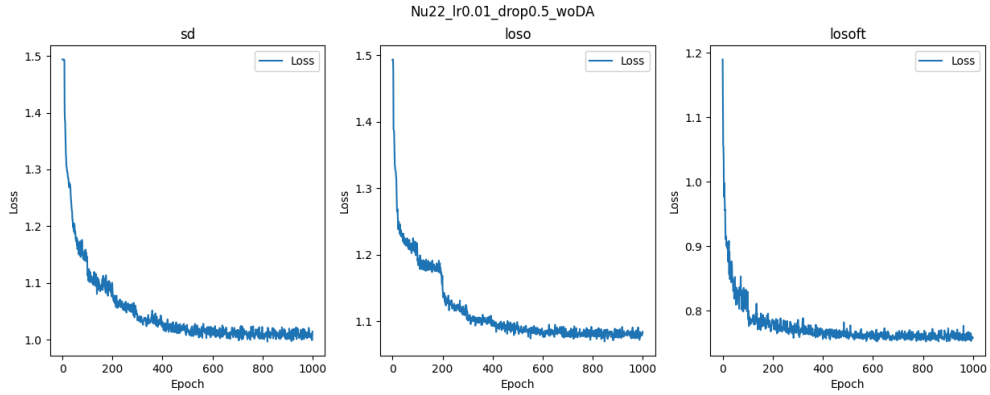
```
>> python .\tester.py
Task: SD_63.11_Nu44_lr0.002_drop0.7_woDA                accuracy: 63.11%
Task: SD_63.76_Nu44_lr0.001_drop0.8_wDA                 accuracy: 63.76%
Task: LOSO_60.76_Nu44_lr0.002_drop0.7_woDA              accuracy: 60.76%
Task: LOSO_60.76_Nu44_lr0.001_drop0.8_wDA               accuracy: 60.76%
Task: LOSO+FT_77.43_Nu22_lr0.01_drop0.5_wDA             accuracy: 77.43%
Task: LOSO+FT_77.78_Nu44_lr0.001_drop0.8_wDA            accuracy: 77.78%
Task: LOSO+FT_80.21_Nu44_lr0.001_drop0.8_wDA_Step500    accuracy: 80.21%
```

Discover during the training process

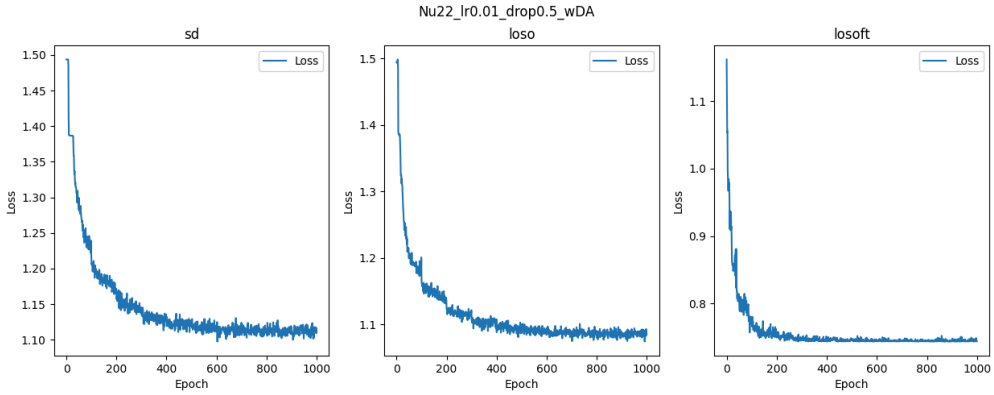
在實驗過程中，發現不管使用哪組參數，loss 到了一定階段，都會出現瓶頸無法再降低，呈現劇烈震動的情況。

使用的部分參數與其 loss curve 如下，其中的 DA 表示有無使用 Data Augmentation：

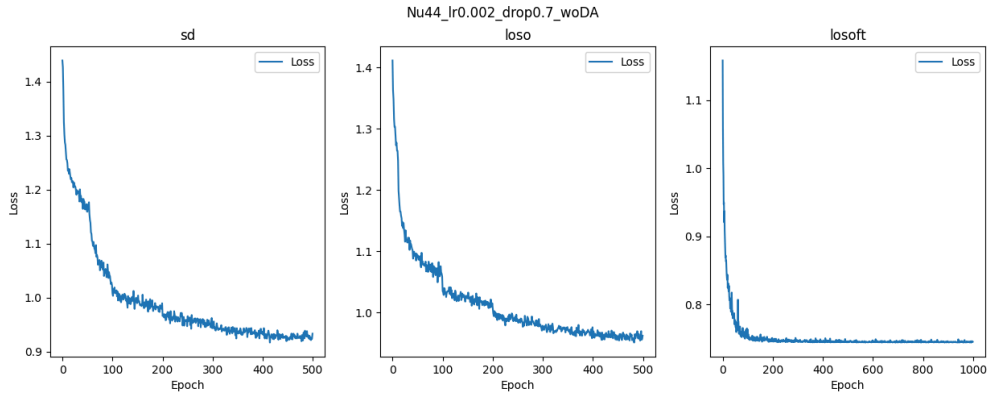
- Nu22_lr0.01_drop0.5_woDA: 60.07%/54.86%/76.39%



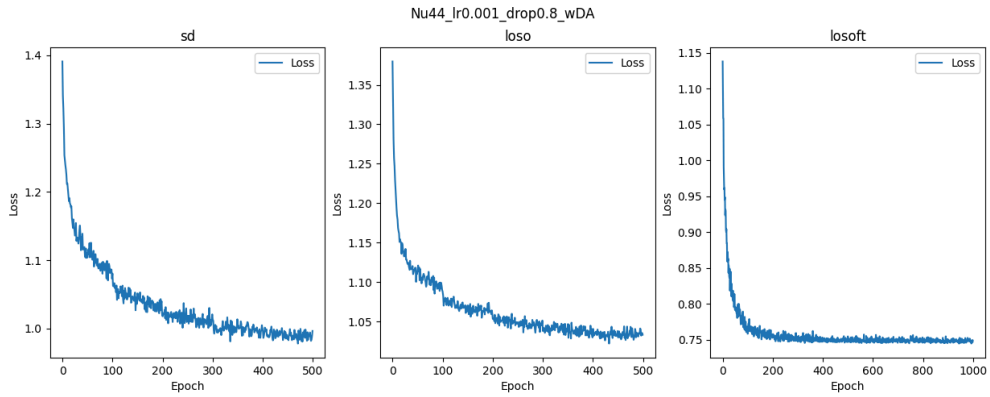
- Nu22_lr0.01_drop0.5_wDA: 55.21%/57.99%/77.43%



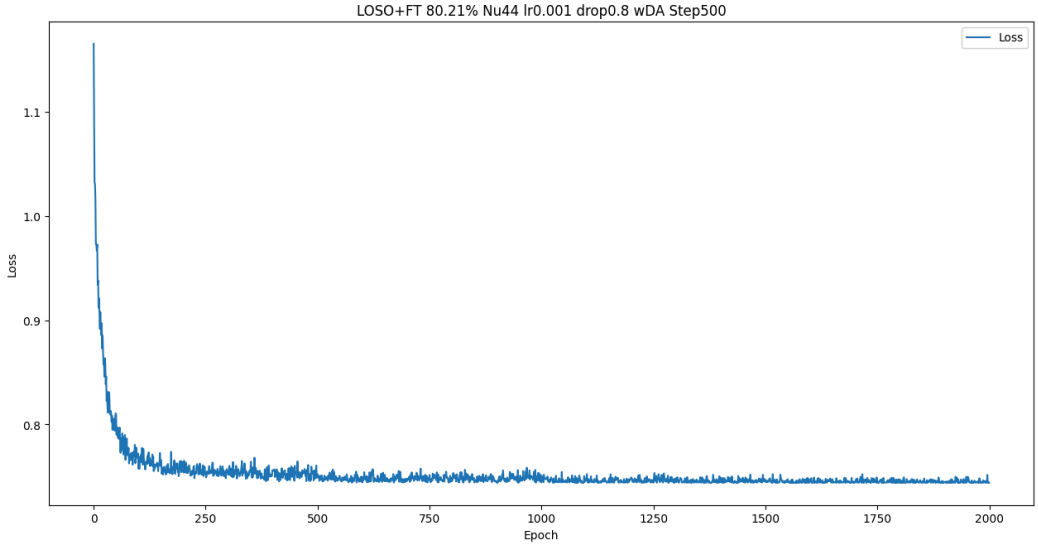
- Nu44_lr0.002_drop0.7_woDA: 63.11%/60.76%/74.31%



- Nu44_lr0.001_drop0.8_wDA: 63.76%/60.76%/77.78%



此外，在 finetune 時嘗試使用了 step_size 為 500 的 scheduler，在參數同樣為 `Nu44_lr0.001_drop0.8_wDA` 的情況下，能夠在第 430 個 epoch 時就達到 80.21 的準確率，其 loss curve 如下：



Comparison between the three training methods

三種訓練方法的差別如下：

1. 受試者相關(Subject Dependent)
    - 此部分的訓練方法和論文中的方法有所不同，論文中是除了其中一個受試者的第 2 個 session 作為 validation set，其餘的所有 session 都做為 training set；而本次實驗中，需要使用所有受試者的第 1 個 session 作為 training set，第 2 個 session 作為 validation set。
2. 留一受試者交叉驗證(Leave-One-Subject-Out, LOSO)
    - 這種方式需要將其中一個受試者的第 2 個 session 作為 validation set ，並使用其餘受試者的所有 session 作為 training set，被選中受試者的第一個 session 將不予使用。
3. LOSO加微調(Fine-tuning)
    - 在 LOSO 的結果上，使用被選中受試者的第一個 session 作為 training set 來做 fine-tuning，並同樣使用該受試者的第 2 個 session 作為 validation set。

從結果來看，使用所有受試者的第 1 個 session 作為 training set，第 2 個 session 作為 validation set 會導致模型性能下降；而使用被選中受試者的第一個 session 作為 training set 來做 fine-tuning 則會導致模型性能提高。這可能是因為 EEG 訊號在受試

者之間甚至在單個受試者內部都存在強烈的 *可變性(variability)*。

(Optional) Anything you want to mention

Discussion

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

這可能是因為 EEG 訊號在受試者之間甚至在單個受試者內部都存在強烈的 *可變性(variability)*，所以將不同受試者的資料合併訓練，可能不會使模型性能受益，從實驗結果中也能看出，SD 和 SI 方案的性能都遠輸於使用受試者的第一個 session 作為 training set 來做 fine-tuning 的方案。如果沒有適當的訓練方案，增加數據可能不一定可以提高模型性能。這也體現了使用個人資料來訓練個性化 BCI 模型的重要性。

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

一些可能提高正確率的方法：

- 透過調整 N_u 來增加模型的參數數量
  - 在實驗中將 N_u 從 22 增加到 44 以增加模型的參數數量。
- 提高 dropout rate 的值，以增加模型的 Generalization 能力
  - 在實驗中使用了 0.7 和 0.8 的 dropout rate 來增加模型的 Generalization 能力。
- 使用 Data Augmentation 技術來增加訓練數據
  - 在實驗中嘗試使用了一些 Data Augmentation 技術，例如 Add Gaussian noise、Time shifting 等。
  - 但同時也注意到，可能並不是所有的 Data Augmentation 技術適合用在 EEG Signal 上，例如通道置換(Channel shuffle)就不適合用在 EEG Signal 上，因為 EEG 的訊號通道是代表電極放置的不同位置。
- 使用 step_size 參數來控制 scheduler 的 step size
  - 將 step_size 調高，讓模型持續維持較大的步伐，使其能夠盡可能的能走到正確率較高的地方。

(Optional) Anything you want to mention

實作的 Data Augmentation 以及其程式碼如下，但主要只使用了 Add Gaussian noise 和 Time shifting 兩種技術：

- 添加高斯噪聲 Add Gaussian noise

```
def gaussian_noise(eeg, std=0.1):
    """
    Add Gaussian noise to the input signal.

    Args:
        eeg (numpy.ndarray): Input signal, shape (n_channels,
n_timepoints) or (n_batch, n_channels, n_timepoints)
        std (float): Standard deviation of the Gaussian noise
    Returns:
        numpy.ndarray: Noised signal with the same shape as input
    """
    noise = np.random.normal(0, std, eeg.shape) # (mean, std, size)
    return eeg + noise
```

- 時間偏移 Time shift

```
def time_shift(eeg, shift_max=10):
    """
    Shift the input signal in time.

    Args:
        eeg (numpy.ndarray): Input signal, shape (n_channels,
n_timepoints) or (n_batch, n_channels, n_timepoints)
        shift_max (int): Maximum shift in time
    Returns:
        numpy.ndarray: Shifted signal with the same shape as input
    """
    shift = np.random.randint(-shift_max, shift_max+1) # [-10, 10]
    return np.roll(eeg, shift, axis=-1) # roll at time axis
```

- 振幅縮放 Amplitude scale

```
def amplitude_scale(eeg, scale_range=(0.8, 1.2)):
    """
    Scale the amplitude of the input signal.

    Args:
        eeg (numpy.ndarray): Input signal, shape (n_channels,
n_timepoints) or (n_batch, n_channels, n_timepoints)
        scale_range (tuple): Range of the scale factor
    Returns:
        numpy.ndarray: Scaled signal with the same shape as input
    """
    scale = np.random.uniform(*scale_range)
    return eeg * scale
```

- 通道置換 Channel shuffle

```
def channel_shuffle(eeg, n_swaps=2):
    """
    Shuffle channels in the EEG signal.

    Args:
        eeg (numpy.ndarray): Input EEG signal, shape (n_channels,
n_timepoints) or (n_batch, n_channels, n_timepoints)
        n_swaps (int): Number of channel swaps to perform
    Returns:
        numpy.ndarray: EEG signal with shuffled channels, same shape as
input
    """
    eeg = np.copy(eeg)

    # 檢查輸入維度
    if eeg.ndim == 2:
        n_channels, n_timepoints = eeg.shape
        n_batch = 1
        eeg = eeg.reshape(1, n_channels, n_timepoints)
    elif eeg.ndim == 3:
        n_batch, n_channels, n_timepoints = eeg.shape
    else:
        raise ValueError("Input EEG must be 2D or 3D array")

    for b in range(n_batch):
        for _ in range(n_swaps):
            # 隨機選擇兩個不同的通道
            ch1, ch2 = np.random.choice(n_channels, 2, replace=False)
            # 交換這兩個通道
            eeg[b, ch1], eeg[b, ch2] = eeg[b, ch2].copy(), eeg[b,
ch1].copy()

    # 如果原始輸入是2D，則壓縮輸出
    if eeg.shape[0] == 1:
        eeg = eeg.squeeze(0)

    return eeg
```

- 頻帶濾波 Bandpass filter

```
def bandpass_filter(eeg, fs, order=5):
    """
    Apply bandpass filter to EEG signal.

    Args:
        eeg (numpy.ndarray): Input EEG signal, shape (n_channels,
n_timepoints) or (n_batch, n_channels, n_timepoints)
        fs (float): Sampling frequency
        order (int): Order of the filter

    Returns:
        numpy.ndarray: Filtered EEG signal, same shape as input
    """
    eeg = np.copy(eeg)

    # 檢查輸入維度
    if eeg.ndim == 2:
        n_channels, n_timepoints = eeg.shape
        n_batch = 1
        eeg = eeg.reshape(1, n_channels, n_timepoints)
    elif eeg.ndim == 3:
        n_batch, n_channels, n_timepoints = eeg.shape
    else:
        raise ValueError("Input EEG must be 2D or 3D array")

    # 設計濾波器
    lowcut = np.random.uniform(0.5, 4)  # 隨機選擇低頻截止
    highcut = np.random.uniform(30, 50)  # 隨機選擇高頻截止
    nyq = 0.5 * fs
    low = lowcut / nyq
    high = highcut / nyq
    b, a = signal.butter(order, [low, high], btype='band')

    # 應用濾波器
    for i in range(n_batch):
        for j in range(n_channels):
            eeg[i, j] = signal.filtfilt(b, a, eeg[i, j])

    # 如果原始輸入是2D，則壓縮輸出
    if n_batch == 1:
        eeg = eeg.squeeze(0)

    return eeg
```

- 傅立葉變換替代 Fourier-transform (FT) surrogate

```
def ft_surrogate(eeg): # Fourier Transform surrogate
    """
    Generate a Fourier Transform surrogate of the input signal.

    Args:
        eeg (numpy.ndarray): Input signal, shape (n_channels,
n_timepoints) or (n_batch, n_channels, n_timepoints)

    Returns:
        numpy.ndarray: Surrogate signal with the same shape as input
    """
    eeg = np.copy(eeg)

    # 檢查輸入維度
    if eeg.ndim == 2:
        n_channels, n_timepoints = eeg.shape
        n_batch = 1
    elif eeg.ndim == 3:
        n_batch, n_channels, n_timepoints = eeg.shape
    else:
        raise ValueError("Input signal must be 2D or 3D array")

    # 重塑信號以統一處理
    eeg = eeg.reshape(-1, n_channels, n_timepoints)

    for b in range(n_batch):
        for ch in range(n_channels):
            # 進行傅立葉變換
            fft = np.fft.fft(eeg[b, ch])

            # 提取幅度和相位
            magnitudes = np.abs(fft)

            # 隨機化相位
            random_phases = np.random.uniform(0, 2*np.pi, len(fft))

            # 保持第一個（直流）分量的相位不變
            random_phases[0] = 0

            # 確保共軛對稱性（對於實值信號）
            random_phases[-len(fft)//2+1:] = -
random_phases[1:len(fft)//2][::-1]

            # 重建信號
            new_fft = magnitudes * np.exp(1j * random_phases)

            # 逆傅立葉變換
            eeg[b, ch] = np.real(np.fft.ifft(new_fft))

    # 如果原始輸入是2D，則壓縮輸出
    if eeg.shape[0] == 1:
        eeg = eeg.squeeze(0)
```

```
return eeg
```