Lab2: EEG Motor Imagery Classification

1. Overview

This report explores the analysis of EEG data for motor imagery tasks using the BCI Competition IV Dataset 2a, which involves classifying mental representations of limb movements. The preprocessing steps include resampling, bandpass filtering, and epoching to prepare the data for analysis. The primary model, SCCNet, was evaluated under three training methods: Subject Dependent (SD), Leave-One-Subject-Out (LOSO), and Fine-Tune (FT).
Results indicated that while the model performed well on training data, it exhibited overfitting, with training accuracy higher than test accuracy. The Fine-Tune method achieved the highest test accuracy.
To improve accuracy, incorporating RNNs like LSTM or GRU to capture temporal dependencies, applying data augmentation, optimizing hyperparameters.

2. Implementation Details

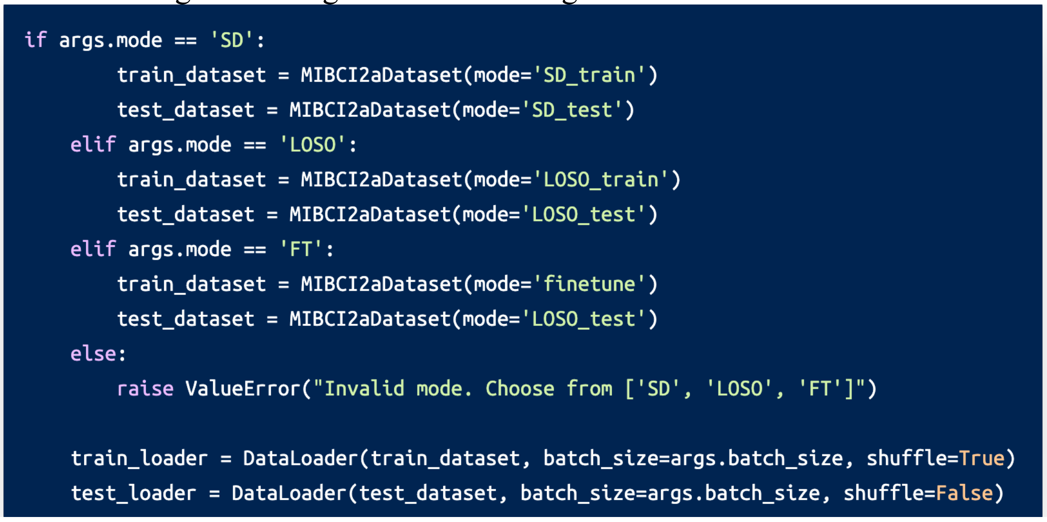
A. Details of training and testing code

1) Data Loading and Preprocessing

Load datasets in different modes (Subject Dependent (SD), Leave-One-Subject-Out (LOSO), Fine-tune (FT)) using the MIBCI2aDataset class.

```python
class MIBCI2aDataset(torch.utils.data.Dataset):
    def __init__(self, mode):
        # remember to change the file path according to different experiments
        assert mode in ['SD_train', 'LOSO_train','finetune','SD_test','LOSO_test']
        if mode == 'SD_train':
            self.features = self._getFeatures(filePath='./dataset/SD_train/features/')
            self.labels = self._getLabels(filePath='./dataset/SD_train/labels/')

        if mode == 'LOSO_train':
            self.features = self._getFeatures(filePath='./dataset/LOSO_train/features/')
            self.labels = self._getLabels(filePath='./dataset/LOSO_train/labels/')

        if mode == 'finetune':
            self.features = self._getFeatures(filePath='./dataset/FT/features/')
            self.labels = self._getLabels(filePath='./dataset/FT/labels/')

        if mode == 'SD_test':
            self.features = self._getFeatures(filePath='./dataset/SD_test/features/')
            self.labels = self._getLabels(filePath='./dataset/SD_test/labels/')

        if mode == 'LOSO_test':
            self.features = self._getFeatures(filePath='./dataset/LOSO_test/features/')
            self.labels = self._getLabels(filePath='./dataset/LOSO_test/labels/')
```

Create training and testing data loaders using DataLoader.



```
if args.mode == 'SD':
        train_dataset = MIBCI2aDataset(mode='SD_train')
        test_dataset = MIBCI2aDataset(mode='SD_test')
    elif args.mode == 'LOSO':
        train_dataset = MIBCI2aDataset(mode='LOSO_train')
        test_dataset = MIBCI2aDataset(mode='LOSO_test')
    elif args.mode == 'FT':
        train_dataset = MIBCI2aDataset(mode='finetune')
        test_dataset = MIBCI2aDataset(mode='LOSO_test')
    else:
        raise ValueError("Invalid mode. Choose from ['SD', 'LOSO', 'FT']")

    train_loader = DataLoader(train_dataset, batch_size=args.batch_size, shuffle=True)
    test_loader = DataLoader(test_dataset, batch_size=args.batch_size, shuffle=False)
```

2) Model Training

Define the train_model function to perform forward propagation, compute loss, backpropagation, and update parameters.
Compute training loss and accuracy at the end of each epoch.

```python
def train_model(model, train_loader, criterion, optimizer, device):
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0

    for features, labels in train_loader:
        features, labels = features.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = model(features)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item() * features.size(0)

        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    epoch_loss = running_loss / len(train_loader.dataset)
    epoch_accuracy = correct / total
    return epoch_loss, epoch_accuracy
```

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    model = SCCNet(timeSample=train_dataset.features.shape[2],C=train_dataset.features.shape[1],Nc=args.Nc, Nt=args.Nt,
Nu=args.Nu, dropoutRate=args.dropoutRate).to(device)

    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(),  lr=args.lr, weight_decay=args.weight_decay)

    num_epochs = args.num_epochs
    best_accuracy = 0.0
    best_model_path = "best_SCCNet_model.pth"

    with open(args.csv_path, mode='w', newline='') as csv_file:
        fieldnames = ['epoch', 'train_loss', 'train_accuracy', 'test_accuracy']
        writer = csv.DictWriter(csv_file, fieldnames=fieldnames)
        writer.writeheader()

        for epoch in range(num_epochs):
            train_loss, train_accuracy = train_model(model, train_loader, criterion, optimizer, device)
            test_accuracy = evaluate_model(model, test_loader, device)

            if test_accuracy > best_accuracy:
                best_accuracy = test_accuracy
                torch.save(model, best_model_path)
                print(f"New best model saved with accuracy: {best_accuracy:.4f}")

            print(f"Epoch {epoch + 1}/{num_epochs}, Loss: {train_loss:.4f}, Train Accuracy: {train_accuracy:.4f}, Test
Accuracy: {test_accuracy:.4f}")

            writer.writerow({'epoch': epoch + 1, 'train_loss': train_loss, 'train_accuracy': train_accuracy,
'test_accuracy': test_accuracy})

    print(f"Best model saved with accuracy: {best_accuracy:.4f}")
```

3) Model Evaluation

Define the evaluate_model function to evaluate the model and calculate testing accuracy.

```
def evaluate_model(model, test_loader, device):
    model.eval()
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for features, labels in test_loader:
            features, labels = features.to(device), labels.to(device)
            outputs = model(features)
            _, preds = torch.max(outputs, 1)
            all_preds.extend(preds.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())
    accuracy = accuracy_score(all_labels, all_preds)
    return accuracy
```

4) Model Saving and Loading

Save the best model during the training process.
Provide the load_model function to load the model from the saved model file.

```
for epoch in range(num_epochs):
            train_loss, train_accuracy = train_model(model, train_loader, criterion, optimizer, device)
            test_accuracy = evaluate_model(model, test_loader, device)

            if test_accuracy > best_accuracy:
                best_accuracy = test_accuracy
                torch.save(model, best_model_path)
                print(f"New best model saved with accuracy: {best_accuracy:.4f}")

            print(f"Epoch {epoch + 1}/{num_epochs}, Loss: {train_loss:.4f}, Train Accuracy: {train_accuracy:.4f}, Test Accuracy: 
{test_accuracy:.4f}")
```

5) Command Line Arguments

Use argparse to parse command line arguments, including mode selection, model hyperparameter settings, and more

```
python trainer.py --mode <mode>  --Nc <Nc>  --Nt <Nt> --Nu <Nu> --dropoutRate <dropoutRate> --batch_size <batch size>  -
-num_epochs <num epochs> --lr <lr> --weight_decay <weight decay> --csv_path <csv path>
```

.

B. Details of the SCCNet (10%)

```python
class SCCNet(nn.Module):
    def __init__(self, numClasses=4, timeSample=438, Nu=22, Nc=22, C=22, Nt=1, dropoutRate=0.7):
        super(SCCNet, self).__init__()

        self.conv1 = nn.Conv2d(1, Nu, (C, Nt), padding=0)
        self.bn1 = nn.BatchNorm2d(Nu)

        padding_width = 12 // 2
        self.conv2 = nn.Conv2d(1, Nc, (Nu, 12), padding=(0, padding_width))
        self.bn2 = nn.BatchNorm2d(Nc)
        self.square = SquareLayer()
        self.dropout = nn.Dropout(dropoutRate)

        self.avg_pool = nn.AvgPool2d((1, 62), stride=(1, 12))

        output_width = ((timeSample - Nt + 1 - 62) // 12 + 1)
        in_features = Nc * output_width
        self.fc = nn.Linear(in_features, numClasses)

    def forward(self, x):
        x = x.unsqueeze(1)
        x = self.conv1(x)
        x = self.bn1(x)
        x = x.permute(0, 2, 1, 3)
        x = self.conv2(x)
        x = self.bn2(x)
        x = self.square(x)
        x = self.dropout(x)
        x = self.avg_pool(x)
        x = torch.flatten(x, 1)
        x = self.fc(x)
        x = F.softmax(x, dim=1)
        return x
```

1) First Convolution Layer:

- **Use `nn.Conv2d` for 2D convolution with a kernel size of (C, Nt) and an output channel number of Nu.**
- **Follow with `nn.BatchNorm2d` for batch normalization.**

2) Second Convolution Layer:

- **Use `nn.Conv2d` for 2D convolution with a kernel size of (Nu, 12), an output channel number of Nc, and padding in the time dimension.**
- **Use `nn.BatchNorm2d` for batch normalization as well.**

3) Activation and Dropout Layers:

- **A custom square activation layer (`SquareLayer`) is used to extract power variations from the data.**
- **Use `nn.Dropout` to prevent overfitting.**

4) Average Pooling Layer:

- **Use `nn.AvgPool2d` for average pooling with a pool size of (1, 62) and a stride of (1, 12).**

5) Fully Connected Layer:

- **Calculate the number of input features for the fully connected layer based on the pooled output size.**
- **Use `nn.Linear` for the fully connected layer, with the output corresponding to 4 classes.**

3. Analyze on the experiment results

- Experiment Parameter

Training method				Dropout Rate	epochs	Learning Rate	Weight decay
SD				0.7	1500	0.001	0.0001
LOSO				0.7	1500	0.001	0.0001
FT				0.9	1500	0.001	0.0001

- Experiment Result

Training method	Training loss	Training Accuracy	Test Accuracy
Subject Dependent	0.8692	0.9601	0.6437
LOSO	0.9279	0.8844	0.6215
LOSO with Finetuning	0.7718	1.0	0.8125

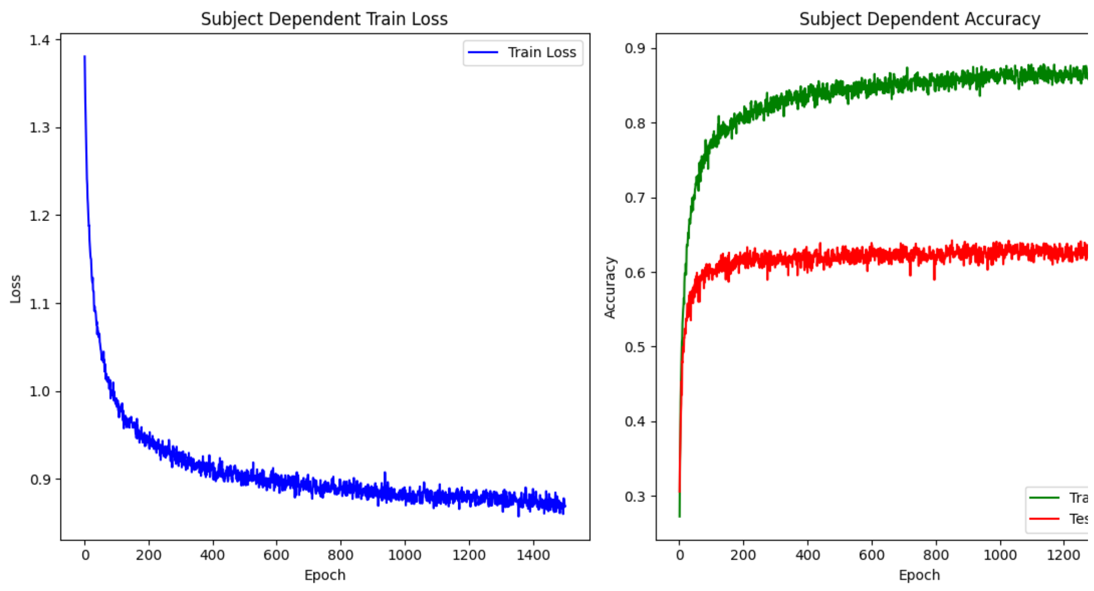


Figure 1 Subject Dependent Train Loss and Accuracy

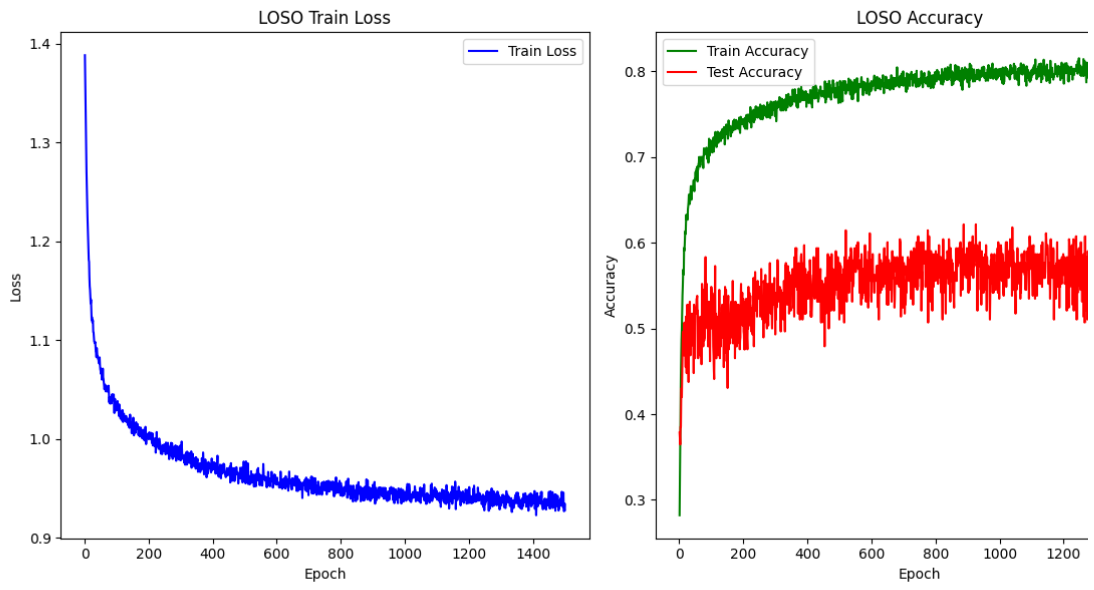


Figure 2 LOSO Train Loss and Accuracy

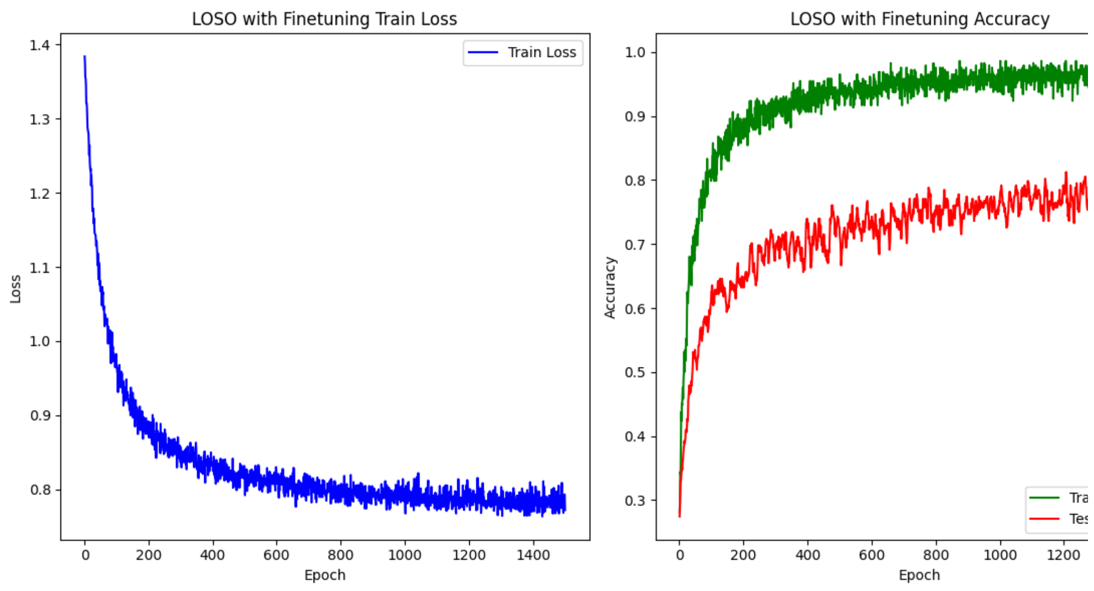


Figure 3 LOSO with Finetuning Train Loss and Accuracy

A. Discover during the training process

1) The Model Easily Overfits

- The model performs very well on the training data but poorly on the test data because it has learned the noise and specific features in the training data rather than the general patterns.
- Overfitting can be mitigated through various methods, including reducing model complexity, using regularization techniques, and introducing dropout layers.

2) Reduce Model Neurons to Improve Generalization When Training Accuracy is Much Higher Than Test Accuracy

- When training accuracy is much higher than test accuracy, it may indicate that the model is too complex and has learned the noise in the training data.
- Reducing the number of neurons or layers in the model can decrease its complexity, helping it to capture the general patterns in the data rather than the features specific to the training data.

3) Dropout Layers Help Improve Model Generalization:

- Using dropout layers during training can randomly drop some neurons, preventing the model from relying too heavily on certain neurons and thereby improving its generalization capability.
- Introducing dropout layers at key points in the model (such as before or between fully connected layers) and appropriately adjusting the dropout rate (e.g., 0.5) can effectively enhance the model's generalization ability.

B. Comparison between the three training methods

1) Subject Dependent (SD) Method

- Training accuracy (0.9601) is higher than testing accuracy (0.6437), indicating that the model performs well on the training data but shows overfitting on the test data.
- Training loss (0.8692) also suggests that the model has lower error on the training data.

2) Leave-One-Subject-Out (LOSO) Method

- Both training accuracy (0.8844) and testing accuracy (0.6215) are lower than in the SD mode, which may be due to the increased challenge of leaving one subject out for testing each time.
- Training loss (0.9279) is slightly higher than in the SD mode, indicating a small increase in error on the training data.

3) Fine-tune (FT) Method

- Training accuracy (1.0) shows that the model achieves perfect accuracy on the training data.
- Testing accuracy (0.8125) is significantly higher than the other two methods, showing that the effectiveness of fine-tuning dataset in improving test accuracy.
- Training loss (0.7718) is lower than in the other two modes, reflecting a smaller error on the training data.

4. Discussion

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

1) Preventing Overfitting

The model had achieved very high accuracy on the training data but had performed poorly on unseen test data. It means the model has learned the noise and specific details of the training data rather than the underlying patterns. We tried different solution to prevent overfitting, such as reducing the number of neurons , increase the dropout rate in dropout layer.

2) Enhancing Robustness

Robustness means the model's resilience to variations and perturbations in the data. To Introducing harder tasks, such as data augmentation, noise addition, or adversarial training, helps in building models that can handle unexpected changes and noise in the input data, making them more reliable in practical applications.

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

1) **Add RNN to the Model**

Incorporate Recurrent Neural Networks (RNNs), such as Long Short-Term Memory (LSTM) or Gated Recurrent Units (GRU), to capture the temporal dependencies in EEG data. RNNs can help the model understand the sequence and timing of brain activity associated with different motor imagery tasks.

2) **Optimize Model Hyperparameters**:

Use techniques such as grid search or random search to find the optimal hyperparameters (e.g., learning rate, batch size, number of layers, number of neurons, dropout rate) for the model.