Overview

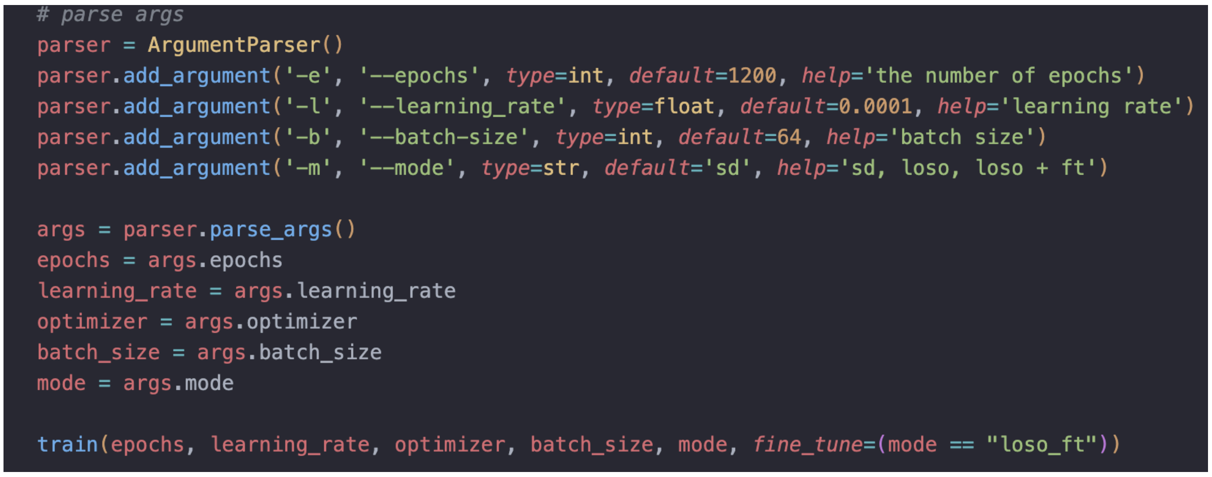
- This assignment involves implementing and evaluating various training methods for a brain-computer interaction (BCI) model, specifically the SCCNet architecture.
- The training methods assessed include Subject Dependent (SD), Leave-One-Subject-Out (LOSO), and LOSO with Fine Tuning (LOSO+FT).
- A major challenge identified was severe training loss fluctuation, which was mitigated by employing larger batch sizes and smaller learning rates.
- The impact of data shift was significant in BCI tasks due to the variability in brain signals, leading to the recommendation of data augmentation to improve model generalization.
- Analysis of the models' performance under different training methods revealed that LOSO+FT achieved the highest accuracy by effectively balancing generalization and individualized tuning.

Implementation Details

Details of training and testing code

Training code

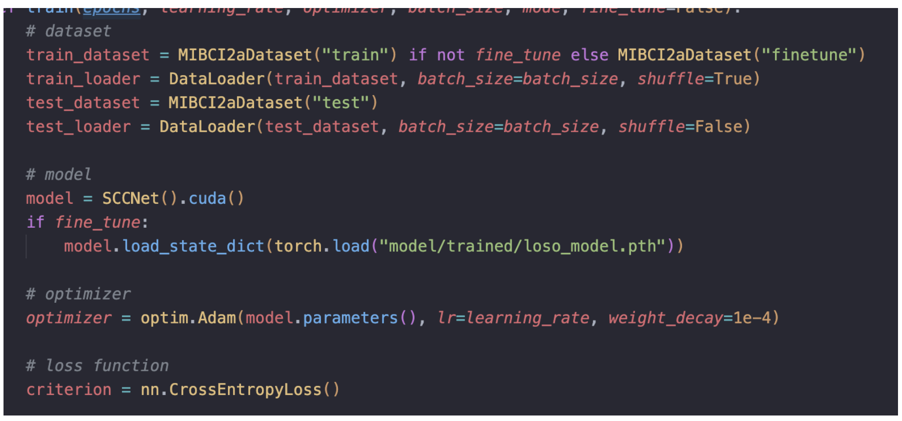
Before beginning the training process, the training parameters are first extracted from the command line arguments. These parameters are then passed to the training function.



```python
# parse args
parser = ArgumentParser()
parser.add_argument('-e', '--epochs', type=int, default=1200, help='the number of epochs')
parser.add_argument('-l', '--learning_rate', type=float, default=0.0001, help='learning rate')
parser.add_argument('-b', '--batch-size', type=int, default=64, help='batch size')
parser.add_argument('-m', '--mode', type=str, default='sd', help='sd, loso, loso + ft')

args = parser.parse_args()
epochs = args.epochs
learning_rate = args.learning_rate
optimizer = args.optimizer
batch_size = args.batch_size
mode = args.mode

train(epochs, learning_rate, optimizer, batch_size, mode, fine_tune=(mode == "loso_ft"))
```

At the start of the training function, the datasets, model, optimizer, and loss function are each initialized respectively.

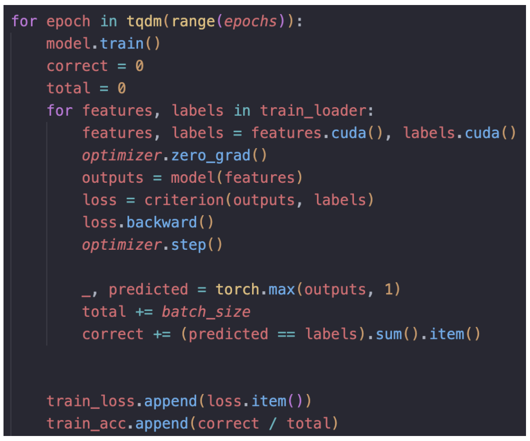
- If the training mode is "LOSO + FT", the pretrained LOSO model is selected, and the "finetune" dataset is used for training.
- L2 regularization is applied with a coefficient of 0.0001 by setting the `weight_decay` parameter in the optimizer's configuration.



```
# dataset
train_dataset = MIBCI2aDataset("train") if not fine_tune else MIBCI2aDataset("finetune")
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_dataset = MIBCI2aDataset("test")
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

# model
model = SCCNet().cuda()
if fine_tune:
    model.load_state_dict(torch.load("model/trained/loso_model.pth"))

# optimizer
optimizer = optim.Adam(model.parameters(), lr=learning_rate, weight_decay=1e-4)

# loss function
criterion = nn.CrossEntropyLoss()
```

Within the training loop (line 41-47), the following things are performed sequentially to optimize the model weights in each epoch:

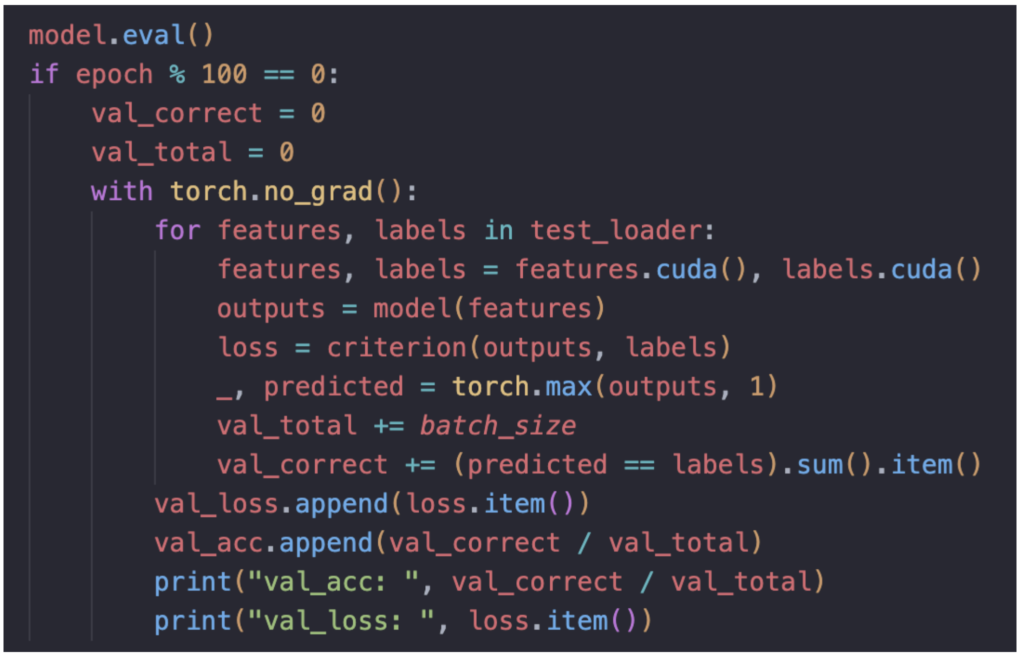
- `model.train()`: switches the model to training mode, enabling gradient calculation and back propagation.
- `optimizer.zero_grad()`: resets the gradients of the model parameters.
- `loss = criterion(outputs, labels)`: computes the loss between the model's predictions `outputs` and the actual labels `labels`.
- `loss.backward()`: performs backpropagation, calculates the gradients, and stores the calculated gradients in the neural network.
- `optimizer.step()`: updates the model's weights using the calculated gradients and the given learning rate.
- `_, predicted = torch.max(outputs, 1)` retrieves the index of the highest value in the model's output for each input sample.
- `(predicted == labels).sum().item()` counts the number of correct predictions by comparing the predicted labels to the actual labels and then sums the total number of correct matches.



```
for epoch in tqdm(range(epochs)):
    model.train()
    correct = 0
    total = 0
    for features, labels in train_loader:
        features, labels = features.cuda(), labels.cuda()
        optimizer.zero_grad()
        outputs = model(features)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

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


    train_loss.append(loss.item())
    train_acc.append(correct / total)
```

Although the following lines do not actually modify the model's weights, tracking the accuracy and loss trends on the test dataset can help understand how well the model is learning and identify any potential issues.

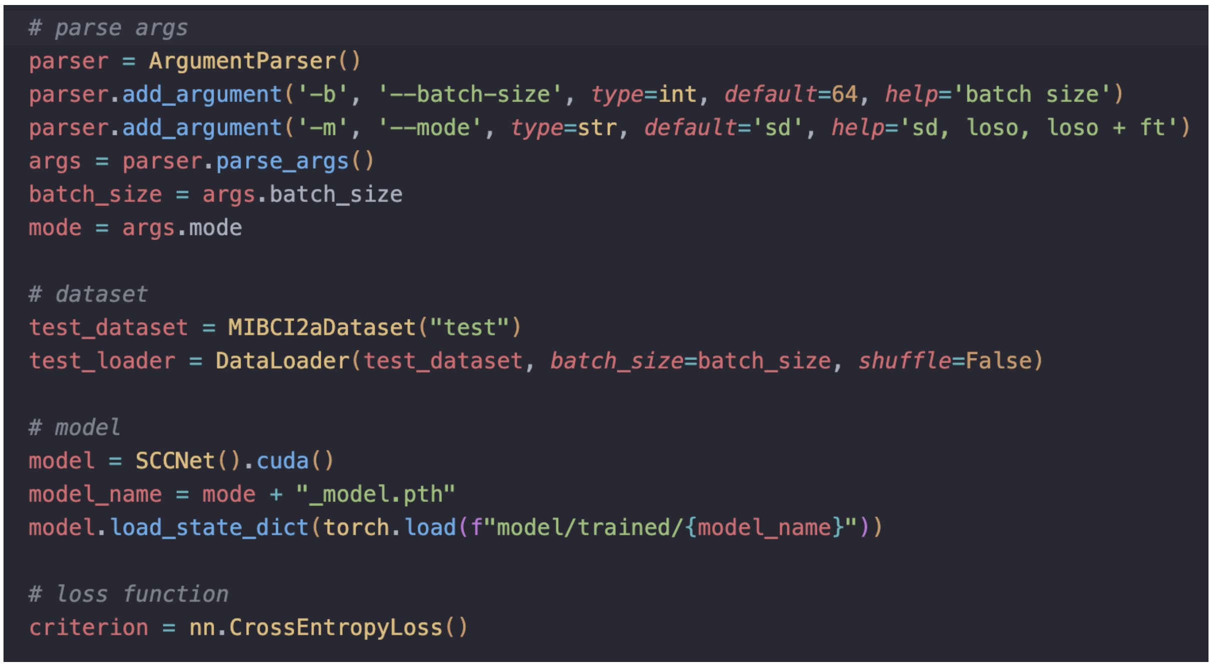
- `model.eval()`: switches the model to evaluation mode, disabling certain layers like dropout.
- `with torch.no_grad()`: disables gradient calculation, reducing memory usage and speeding up computations during inference.



```python
model.eval()
if epoch % 100 == 0:
    val_correct = 0
    val_total = 0
    with torch.no_grad():
        for features, labels in test_loader:
            features, labels = features.cuda(), labels.cuda()
            outputs = model(features)
            loss = criterion(outputs, labels)
            _, predicted = torch.max(outputs, 1)
            val_total += batch_size
            val_correct += (predicted == labels).sum().item()
        val_loss.append(loss.item())
        val_acc.append(val_correct / val_total)
        print("val_acc: ", val_correct / val_total)
        print("val_loss: ", loss.item())
```

Testing code

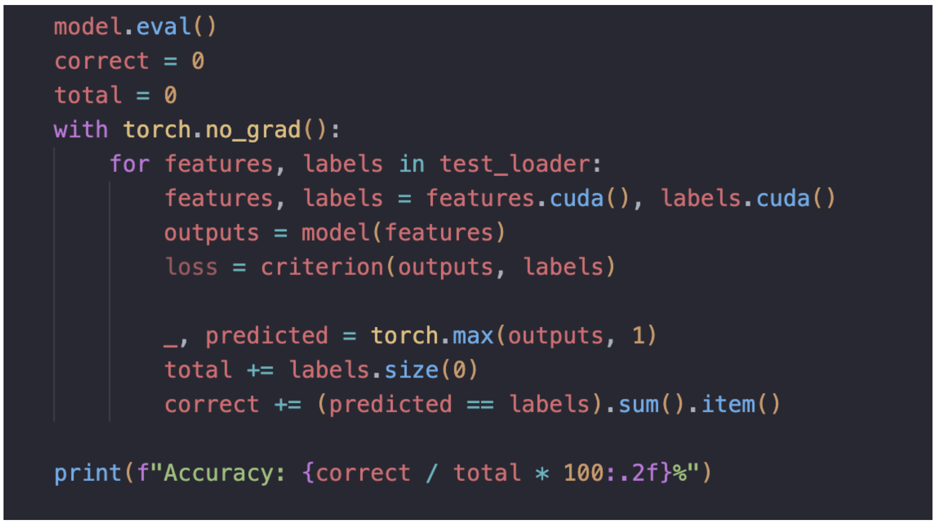
Similar to the training code, the arguments are parsed at the beginning of the testing code. Once parsed into testing parameters, the dataset, model, and loss function are set up accordingly.



```python
# parse args
parser = ArgumentParser()
parser.add_argument('-b', '--batch-size', type=int, default=64, help='batch size')
parser.add_argument('-m', '--mode', type=str, default='sd', help='sd, loso, loso + ft')
args = parser.parse_args()
batch_size = args.batch_size
mode = args.mode

# dataset
test_dataset = MIBCI2aDataset("test")
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

# model
model = SCCNet().cuda()
model_name = mode + "_model.pth"
model.load_state_dict(torch.load(f"model/trained/{model_name}"))

# loss function
criterion = nn.CrossEntropyLoss()
```

The following code is the main part of the testing code.

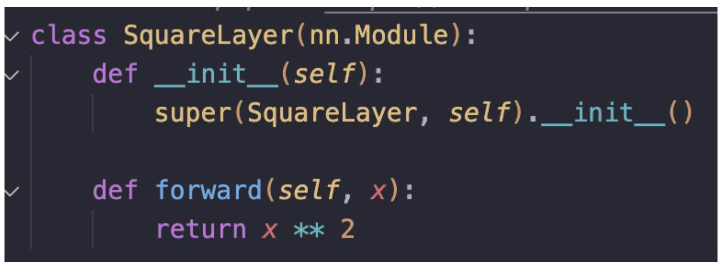
- `model.eval()`: switches the model to evaluation mode, disabling certain layers like dropout.
- `with torch.no_grad()`: disables gradient calculation, reducing memory usage and speeding up computations during inference.
- `outputs = model(features)`: retrieves the predicted outputs from the model for the given input features.
- `_, predicted = torch.max(outputs, 1)`: retrieves the index of the highest value in the model's output for each input sample.
- `(predicted == labels).sum().item()`: counts the number of correct predictions.



```python
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for features, labels in test_loader:
        features, labels = features.cuda(), labels.cuda()
        outputs = model(features)
        loss = criterion(outputs, labels)

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

print(f"Accuracy: {correct / total * 100:.2f}%")
```

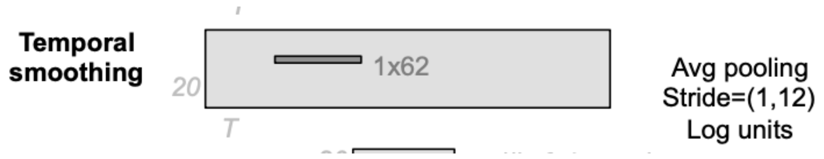
Details of the SCCNet

- `super(SCCNet, self).__init__()`: calls the parent class's constructor (`nn.Module`’s constructor).
- `nn.Conv2d(1, Nu, (C, Nt), padding=0)`: first convolutional layer with 1 input channel, Nu output channels, kernel size (C, Nt), and no padding.
- `nn.Conv2d(Nu, Nc, (1, 12), padding=(0, 6))`: second convolutional layer with Nu input channels, Nc output channels, kernel size (1, 12), and padding (0, 6).
- `self.squareLayer = SquareLayer()`: layer that squares its input. The definition of this layer is shown below.

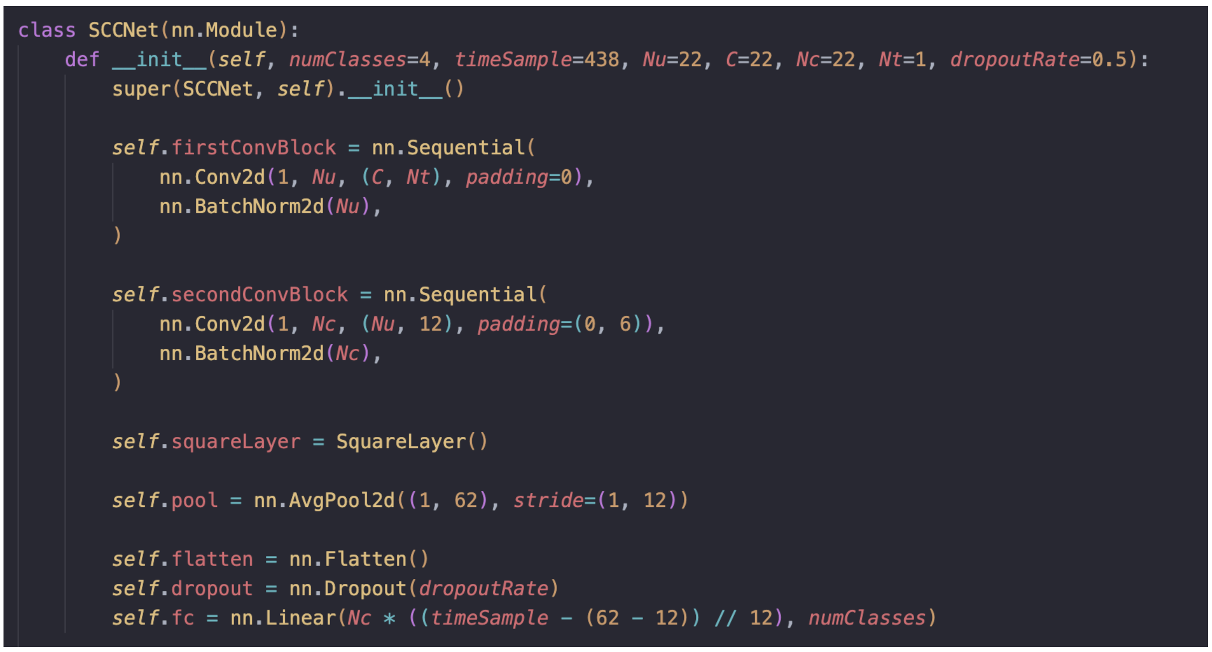


```python
class SquareLayer(nn.Module):
    def __init__(self):
        super(SquareLayer, self).__init__()

    def forward(self, x):
        return x ** 2
```

- `self.pool = nn.AvgPool2d((1, 62), stride=(1, 12))`: average pooling layer with kernel size (1, 62) and stride (1, 12). The kernel size and stride are set according to the given paper.



- `self.flatten = nn.Flatten()`: flattens the input to 1D.
- `self.dropout = nn.Dropout(dropoutRate)`: applies dropout with the specified dropout rate to avoid overfitting according to the given paper.
- `self.fc = nn.Linear(Nc * ((timeSample - (62 - 12)) // 12), numClasses)`: fully connected layer transforming the flattened input into 4 classes.



```python
class SCCNet(nn.Module):
    def __init__(self, numClasses=4, timeSample=438, Nu=22, C=22, Nc=22, Nt=1, dropoutRate=0.5):
        super(SCCNet, self).__init__()

        self.firstConvBlock = nn.Sequential(
            nn.Conv2d(1, Nu, (C, Nt), padding=0),
            nn.BatchNorm2d(Nu),
        )

        self.secondConvBlock = nn.Sequential(
            nn.Conv2d(1, Nc, (Nu, 12), padding=(0, 6)),
            nn.BatchNorm2d(Nc),
        )

        self.squareLayer = SquareLayer()

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

        self.flatten = nn.Flatten()
        self.dropout = nn.Dropout(dropoutRate)
        self.fc = nn.Linear(Nc * ((timeSample - (62 - 12)) // 12), numClasses)
```

Analyze on the experiment results

Discover during the training process

The main issue I faced in this assignment was the severe fluctuation in training loss. According to several related articles, potential solutions include **using a larger batch size** and **reducing the learning rate**.

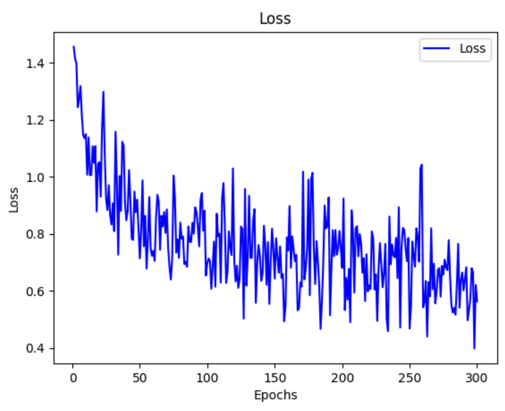
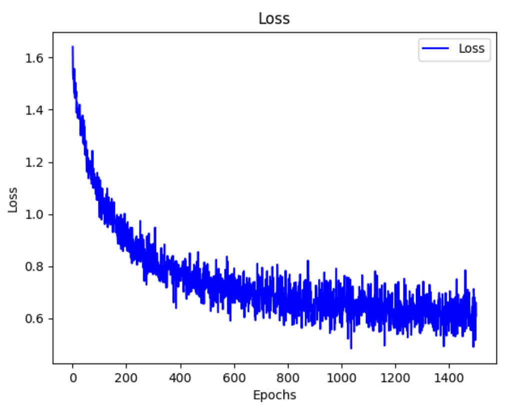
Larger Batch Size

Increasing the batch size reduces the variance in the gradient, resulting in a less noisy training process and more stable updates to the model weights. This can help achieve a more consistent decrease in training loss.

Smaller Learning Rate

Lowering the learning rate helps the model avoid exceeding the optimal values during training. This leads to more gradual and stable convergence, reducing fluctuations in training loss.

By applying these methods, the fluctuation problem was significantly alleviated. The results are shown below.

	
Epochs: 300; batch size: 64; learning rate: 0.001.	Epochs: 1500; batch size: 256; learning rate: 0.0001.

Comparison between the three training methods

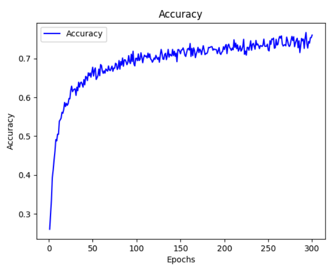
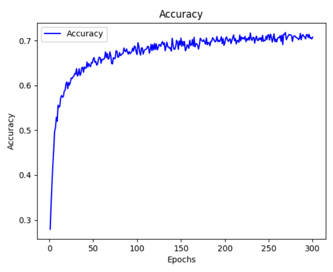
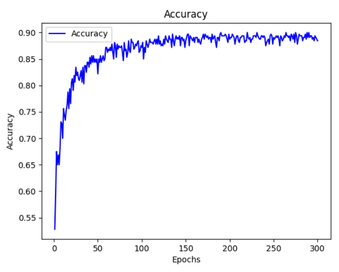
Brief explanation of each training methods

- Subject Dependent (SD): The SD model is trained using data from both two sessions of all subjects, including the first session of the test subject.
- Leave-One-Subject-Out (LOSO): The model is trained on data from all subjects except the test subject, hence the name Leave-One-Subject-Out.
- Leave-One-Subject-Out + Finetune (LOSO+FT): This training method involves two phases. First, an LOSO model is trained using data from all subjects except the test subject as above. In the second phase, the model is fine-tuned using data from the first session of the test subject.

Performance

All models for these methods are trained with the following parameters:

- Epochs: 300
- Batch size: 64
- Learning rate: 0.001

SD	LOSO	LOSO+FT
		
Accuracy: 63.67%	Accuracy: 53.82%	Accuracy: 75.00%

LOSO vs. SD

The LOSO method offers better **generalizability** than the SD method by training on data from all subjects except the test subject. Although SD often performs better because it includes test subject data, this results in worse generalizability due to **potential overfitting**. LOSO ensures the model is exposed only to unseen data during testing, balancing performance and generalizability.

LOSO+FT vs. SD

The model trained with the LOSO+FT method performs better than that of the SD method. This is because it initially trains the model on data from a broader range of subjects (excluding the test subject), which enhances the model's **generalization** capabilities. In the second phase, it fine-tunes the model with data from the test subject, leveraging individualized traits. On the other hand, the SD method pools all data, treating the data from the test subject the same as the training data from other subjects, leading to potential overfitting and reduced generalizability.

Discussion

The reason why the task is hard to achieve high accuracy

Achieving high accuracy is difficult mainly due to the data shift problem in the dataset.

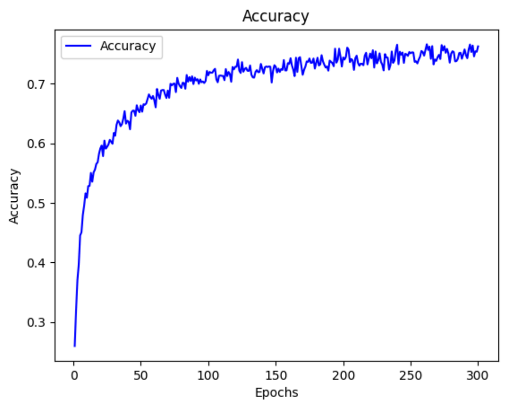
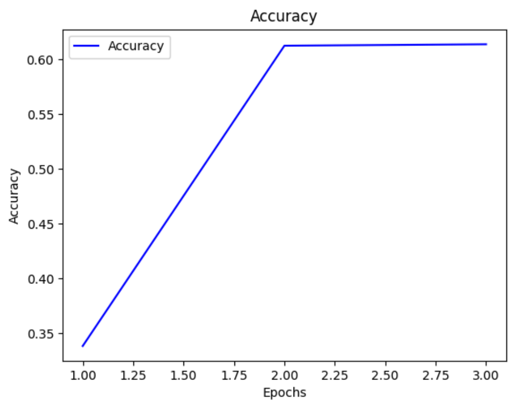
Data shift

Data shift occurs when training and test data distributions differ, causing the model hard to generalize and perform poorly on the test dataset after training.

Why data shift is considered

Data shift is typically identified by a significant drop in accuracy from training to testing. The primary reason this task is challenging is evident from experiment results, where test accuracy consistently drops by about 15% compared to training accuracy, regardless of the training methods applied.

As shown below, the training accuracy can reach approximately 75%, but the same model achieves only approximately 60% accuracy on the test dataset.

Training accuracy chart	Test accuracy chart
	
Epochs: 300; batch size: 64; learning rate: 0.001.	batch size: 64.

Additionally, BCIs often involve highly variable and non-stationary data. Factors such as user fatigue, emotional state, and electrode placement cause brain signals to change, leading to larger potential differences between the training and test distributions. Thus, the impact of data shift is particularly prominent in BCI applications.

What to do to improve the accuracy of this task

Data augmentation, by increasing the diversity of the training data, helps the model generalize better to unseen data. This involves techniques such as rotation, scaling, flipping, and adding noise, which can simulate the variations present in the test data.