NYCU DL Lab2 EEG Motor Imagery Classification

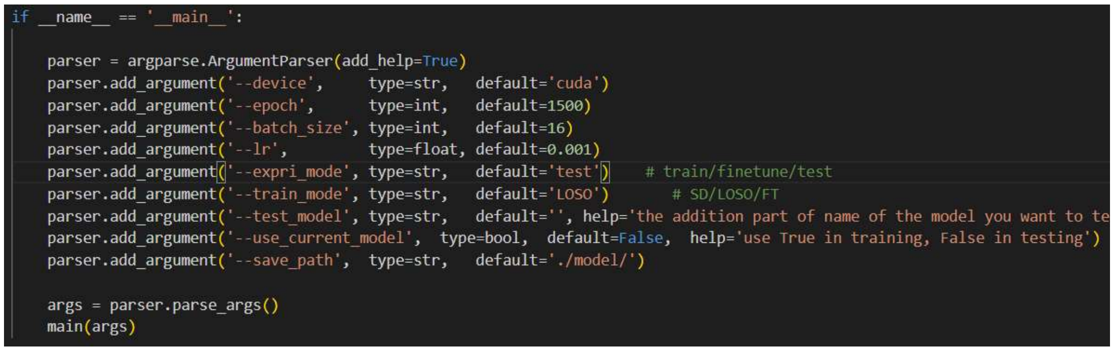
313551073 顏琦恩

I. Overview

In this lab, I implemented the SCCNet from the paper Spatial Component-wise Convolutional Network (SCCNet) for Motor-Imagery EEG Classification, it is an EEG classification network.

In the training section, I'd trained three models using different training strategies: subject dependent, leave-one-subject-out (LOSO), and LOSO with fine tuning. I use the dataset provided by TAs, where the training data of SD includes 8 sections of EEG dataset, LOSO includes 14 sections, and LOSO with fine tuning includes 1 section.

Moreover, I'd implemented a plot function in file utils.py to plot the training loss and accuracy curve. The main function is also in this file, too. To train or test a model, please execute the file utils.py, one can also make some settings there, such as adjusting the learning rate, defining whether you want to train, finetune or test a model.



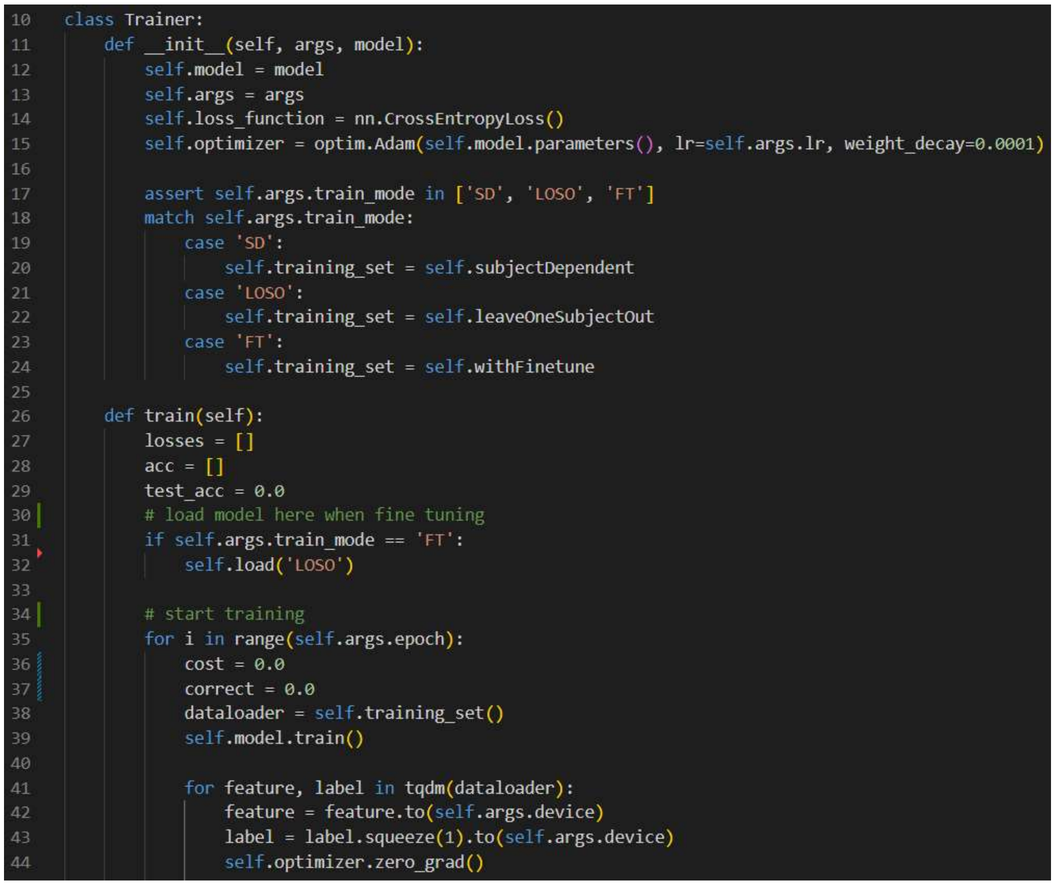
```
if __name__ == '__main__':

    parser = argparse.ArgumentParser(add_help=True)
    parser.add_argument('--device',     type=str,   default='cuda')
    parser.add_argument('--epoch',      type=int,   default=1500)
    parser.add_argument('--batch_size', type=int,   default=16)
    parser.add_argument('--lr',         type=float, default=0.001)
    parser.add_argument('--expri_mode', type=str,   default='test')    # train/finetune/test
    parser.add_argument('--train_mode', type=str,   default='LOSO')       # SD/LOSO/FT
    parser.add_argument('--test_model', type=str,   default='', help='the addition part of name of the model you want to te
    parser.add_argument('--use_current_model',  type=bool,  default=False,  help='use True in training, False in testing')
    parser.add_argument('--save_path',  type=str,   default='./model/')

    args = parser.parse_args()
    main(args)
```

Fig. 1 Arguments can be set in utils.py

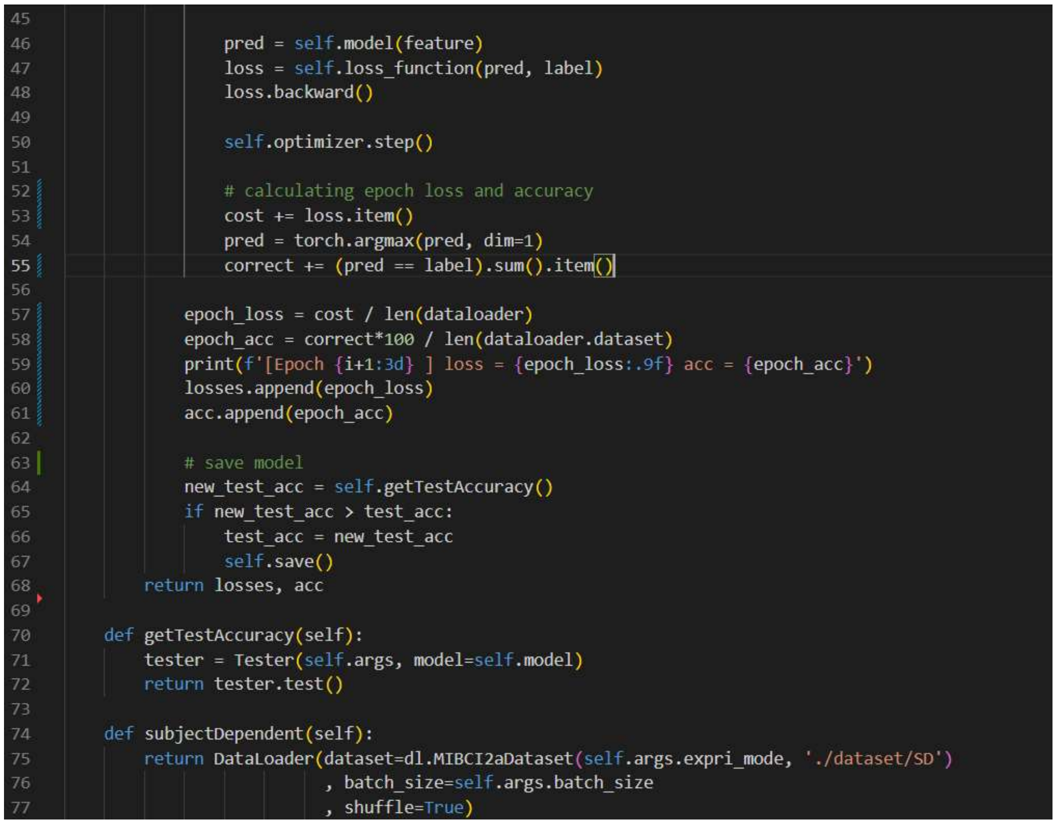
II. Implementation Details

A. Details of training and testing code

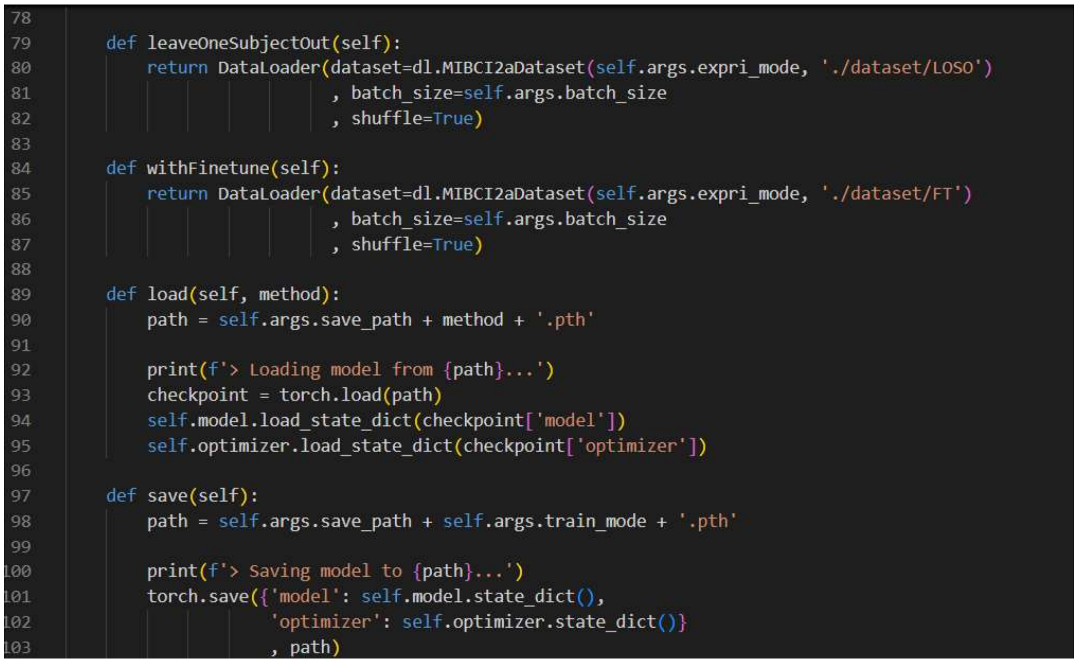
In my training section, I applied cross-entropy as my loss function and Adam with a weight decay of 0.0001, which is mentioned in the paper as the ℓ_2 regularization, as my optimizer. I use variable training_set to get the specific dataset the training process needs. Also, my code checks if it is fine tuning or training before the whole training process. If the current task is fine tuning, I will load my pretrained LOSO model before starting training. During training, I check if the prediction of the test dataset has a better performance, I will save the model which has the highest accuracy in the process.



```python
class Trainer:
    def __init__(self, args, model):
        self.model = model
        self.args = args
        self.loss_function = nn.CrossEntropyLoss()
        self.optimizer = optim.Adam(self.model.parameters(), lr=self.args.lr, weight_decay=0.0001)

        assert self.args.train_mode in ['SD', 'LOSO', 'FT']
        match self.args.train_mode:
            case 'SD':
                self.training_set = self.subjectDependent
            case 'LOSO':
                self.training_set = self.leaveOneSubjectOut
            case 'FT':
                self.training_set = self.withFinetune

    def train(self):
        losses = []
        acc = []
        test_acc = 0.0
        # load model here when fine tuning
        if self.args.train_mode == 'FT':
            self.load('LOSO')

        # start training
        for i in range(self.args.epoch):
            cost = 0.0
            correct = 0.0
            dataloader = self.training_set()
            self.model.train()

            for feature, label in tqdm(dataloader):
                feature = feature.to(self.args.device)
                label = label.squeeze(1).to(self.args.device)
                self.optimizer.zero_grad()
```



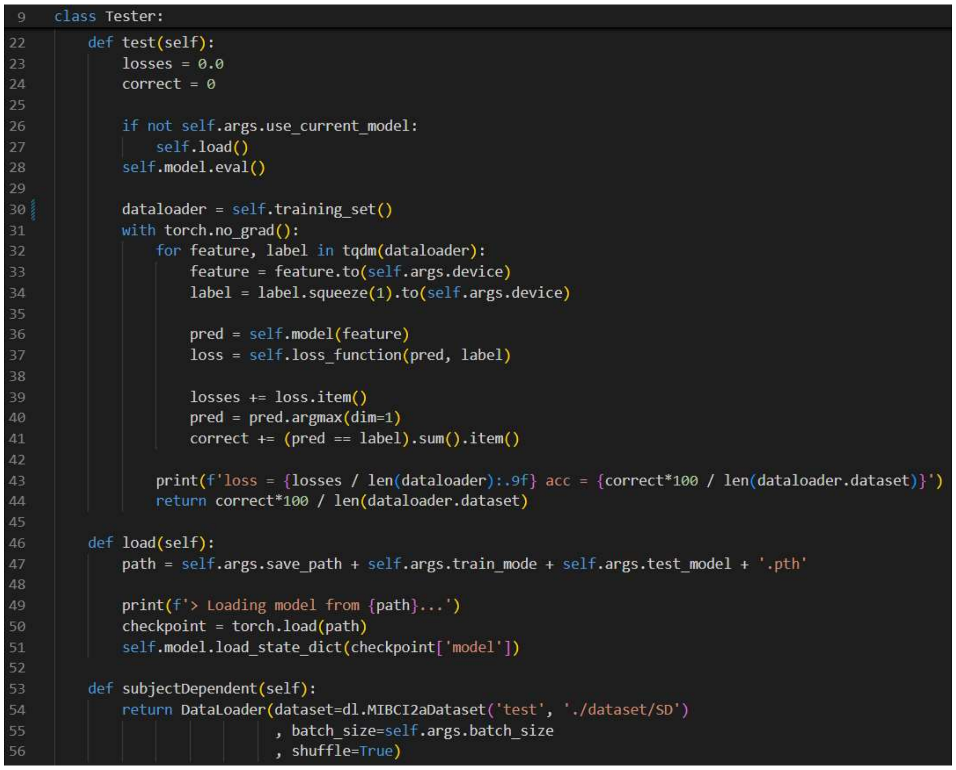
```python

                pred = self.model(feature)
                loss = self.loss_function(pred, label)
                loss.backward()

                self.optimizer.step()

                # calculating epoch loss and accuracy
                cost += loss.item()
                pred = torch.argmax(pred, dim=1)
                correct += (pred == label).sum().item()

            epoch_loss = cost / len(dataloader)
            epoch_acc = correct*100 / len(dataloader.dataset)
            print(f'[Epoch {i+1:3d} ] loss = {epoch_loss:.9f} acc = {epoch_acc}')
            losses.append(epoch_loss)
            acc.append(epoch_acc)

            # save model
            new_test_acc = self.getTestAccuracy()
            if new_test_acc > test_acc:
                test_acc = new_test_acc
                self.save()
        return losses, acc

    def getTestAccuracy(self):
        tester = Tester(self.args, model=self.model)
        return tester.test()

    def subjectDependent(self):
        return DataLoader(dataset=dl.MIBCI2aDataset(self.args.expri_mode, './dataset/SD')
                          , batch_size=self.args.batch_size
                          , shuffle=True)
```



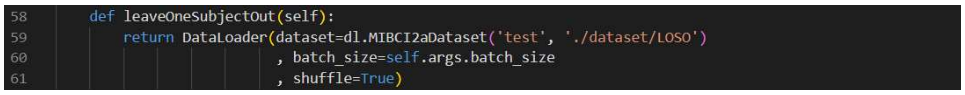
```python

    def leaveOneSubjectOut(self):
        return DataLoader(dataset=dl.MIBCI2aDataset(self.args.expri_mode, './dataset/LOSO')
                          , batch_size=self.args.batch_size
                          , shuffle=True)

    def withFinetune(self):
        return DataLoader(dataset=dl.MIBCI2aDataset(self.args.expri_mode, './dataset/FT')
                          , batch_size=self.args.batch_size
                          , shuffle=True)

    def load(self, method):
        path = self.args.save_path + method + '.pth'

        print(f'> Loading model from {path}...')
        checkpoint = torch.load(path)
        self.model.load_state_dict(checkpoint['model'])
        self.optimizer.load_state_dict(checkpoint['optimizer'])

    def save(self):
        path = self.args.save_path + self.args.train_mode + '.pth'

        print(f'> Saving model to {path}...')
        torch.save({'model': self.model.state_dict(),
                    'optimizer': self.optimizer.state_dict()}
                    , path)
```

Fig. 2 Details of training code

The structure of testing code is familiar with the training code. The difference between them is that we don't need an optimizer in the testing section, so I take this part of my testing code. Furthermore, since the training part has to check the accuracy of the testing dataset to save the model, I applied an if function to check if the testing section has to load a pretrained model before testing.



```python
class Tester:
    def test(self):
        losses = 0.0
        correct = 0

        if not self.args.use_current_model:
            self.load()
        self.model.eval()

        dataloader = self.training_set()
        with torch.no_grad():
            for feature, label in tqdm(dataloader):
                feature = feature.to(self.args.device)
                label = label.squeeze(1).to(self.args.device)

                pred = self.model(feature)
                loss = self.loss_function(pred, label)

                losses += loss.item()
                pred = pred.argmax(dim=1)
                correct += (pred == label).sum().item()

            print(f'loss = {losses / len(dataloader):.9f} acc = {correct*100 / len(dataloader.dataset)}')
            return correct*100 / len(dataloader.dataset)

    def load(self):
        path = self.args.save_path + self.args.train_mode + self.args.test_model + '.pth'

        print(f'> Loading model from {path}...')
        checkpoint = torch.load(path)
        self.model.load_state_dict(checkpoint['model'])

    def subjectDependent(self):
        return DataLoader(dataset=dl.MIBCI2aDataset('test', './dataset/SD')
                          , batch_size=self.args.batch_size
                          , shuffle=True)
```



```
    def leaveOneSubjectOut(self):
        return DataLoader(dataset=dl.MIBCI2aDataset('test', './dataset/LOSO')
                          , batch_size=self.args.batch_size
                          , shuffle=True)
```

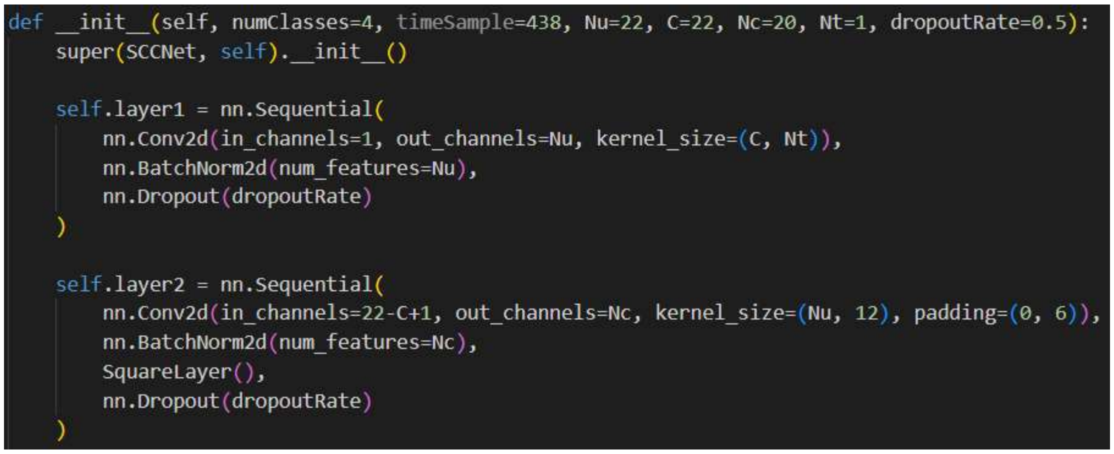
Fig. 3 Details of testing code

B. Details of the SCCNet

The SCCNet is composed of two 2d convolution layers, an average pooling layer, and a softmax classification layer.

In the first convolutional block, I use 1 as the value of in channel, Nu as the out channel, thus the batch normalization has Nu for its input number of features. The paper mentioned that it applied zero padding to this layer, but since I set my kernel size of the convolution layer to (C, Nt), where Nt equals to 1, the padding here will be (0, 0).

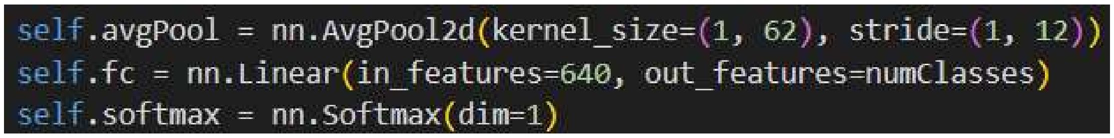
The structure of the second convolutional block is almost the same as the first layer. The differences between two blocks are the square layer and the padding in the convolution layer. In the second block, padding is applied since the kernel size is (Nu, 12), so the padding size to this layer is (0, 6). At the end of these two layers, I applied dropout with a rate of 0.5, just as the paper mentioned.



```
def __init__(self, numClasses=4, timeSample=438, Nu=22, C=22, Nc=20, Nt=1, dropoutRate=0.5):
    super(SCCNet, self).__init__()

    self.layer1 = nn.Sequential(
        nn.Conv2d(in_channels=1, out_channels=Nu, kernel_size=(C, Nt)),
        nn.BatchNorm2d(num_features=Nu),
        nn.Dropout(dropoutRate)
    )

    self.layer2 = nn.Sequential(
        nn.Conv2d(in_channels=22-C+1, out_channels=Nc, kernel_size=(Nu, 12), padding=(0, 6)),
        nn.BatchNorm2d(num_features=Nc),
        SquareLayer(),
        nn.Dropout(dropoutRate)
    )
```

Fig. 4 The first and second layer of SCCNet

After the two convolutional blocks is an average pooling layer with kernel size (1, 62) and stride (1, 12). The last layer performs a softmax classification which includes four classes of out features: left hand, right hand, feet, and tongue.



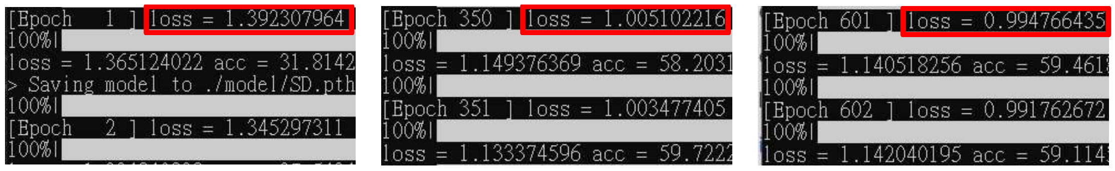
```
self.avgPool = nn.AvgPool2d(kernel_size=(1, 62), stride=(1, 12))
self.fc = nn.Linear(in_features=640, out_features=numClasses)
self.softmax = nn.Softmax(dim=1)
```

Fig. 5 The remain layer of SCCNet

III. Analyze on the experiment results

A. Discover during the training process

In the experiment, I observed that losses of the three methods converge very fast at the beginning of training, but they almost stop converging at about epoch 600. Furthermore, the best model saved during training is usually under epoch 500. I suppose that this result may be due to the variety of the dataset. In the beginning of training, the network learns the larger pattern of the EEG dataset, but after finishing learning these patterns, it can hardly learn the small features of each data. The following figures present the observation here.



```
[Epoch    1 ] loss = 1.392307964
100%|
loss = 1.365124022 acc = 31.8142
> Saving model to ./model/SD.pth
100%|
[Epoch    2 ] loss = 1.345297311
100%|
```

(a)

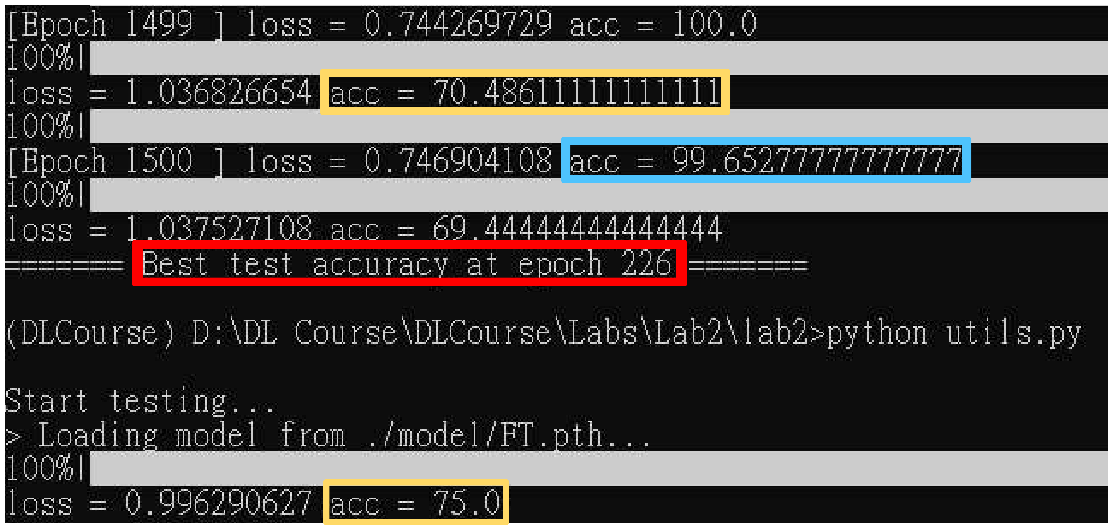
```
[Epoch 350 ] loss = 1.005102216
100%|
loss = 1.149376369 acc = 58.2031
100%|
[Epoch 351 ] loss = 1.003477405
100%|
loss = 1.133374596 acc = 59.7222
```

(b)

```
[Epoch 601 ] loss = 0.994766435
100%|
loss = 1.140518256 acc = 59.461
100%|
[Epoch 602 ] loss = 0.991762672
100%|
loss = 1.142040195 acc = 59.114
```

(c)

Fig. 6 Take SD for example, in the beginning of training (a), training loss is about 1.4; after epoch 350, training loss converges to 1.0; after epoch 600, although the loss seems to continue converging, it is actually just oscillating around 1.0.



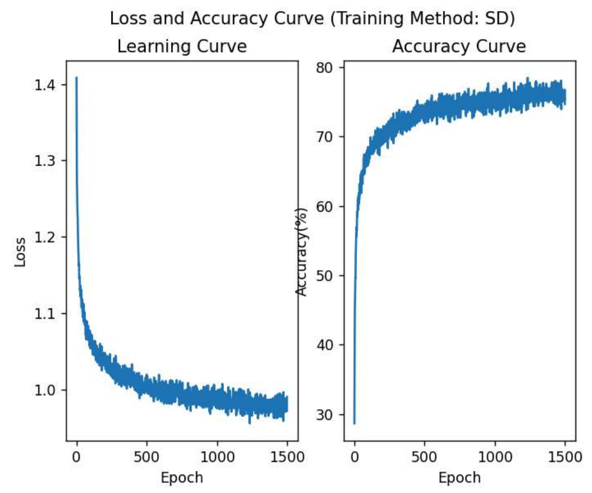
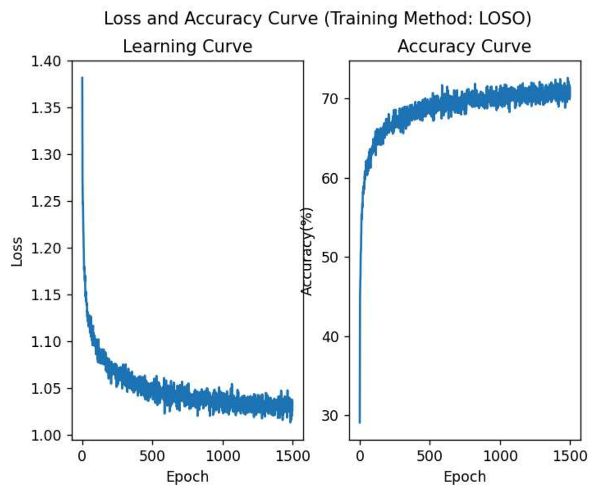
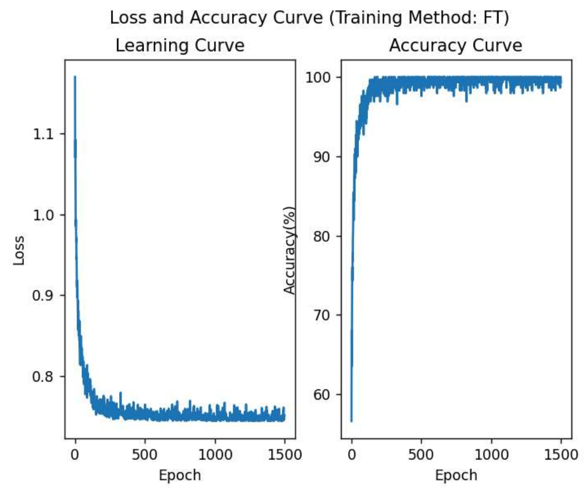
```
[Epoch 1499 ] loss = 0.744269729 acc = 100.0
100%|
loss = 1.036826654 acc = 70.48611111111111
100%|
[Epoch 1500 ] loss = 0.746904108 acc = 99.65277777777777
100%|
loss = 1.037527108 acc = 69.44444444444444
======= Best test accuracy at epoch 226 =======

(DLCourse) D:\DL Course\DLCourse\Labs\Lab2\lab2>python utils.py

Start testing...
> Loading model from ./model/FT.pth...
100%|
loss = 0.996290627 acc = 75.0
```

Fig. 7 Save model at epoch 226 though there were 1500 epochs in this training (the result is from the FT method)

The second observation is that although the accuracy of training can achieve 70% in SD and LOSO, even 100% in FT (Fig.7 the training accuracy is circled by a blue square), but the accuracy of the test dataset is always much lower than these (Fig.7 the testing accuracy is circled by a yellow square). This is probably because of the variety of the dataset, too.

B. Comparison between the three training methods

We can see that though the dataset of LOSO for training is the largest, the accuracy of the model after fine tuning is the highest, the second one is the model using subject dependent, and the last one is the model using leave-one-subject-out. This may be due to the dataset variety and the simplistic structure of the SCCNet.

Methods	Loss and Accuracy Curve	Best Acc for testing
Subject Dependent		63.19%
Leave One Subject Out		60.07%
LOSO with fine tuning		75.69%

IV. Discussion

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

EEG data have strong variability across subjects, and even within a single subject. On the other hand, the network only includes two 2d convolutions, which may be too simple to learn such a complicated feature, so it is difficult to make the task to achieve high accuracy.

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

To improve the accuracy, we can train individualized models for each subject or section of the dataset, this can decrease the variety of the dataset, thus may have a better performance on each model. We can also modify the model structure to a deeper network, this can increase the flexibility of the network.