Use 1 free late-day, remain 4 free late-days

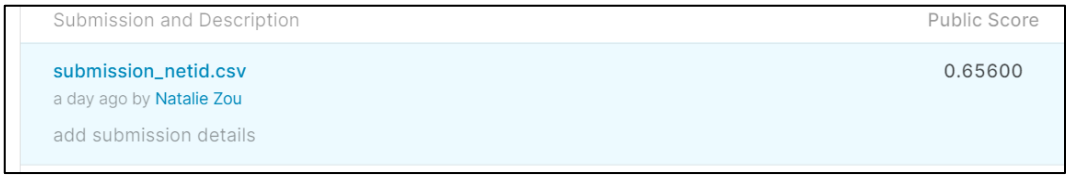
Project 2

Huiyi Zou 301355563

Part 1

My Kaggle account is Natalie Zou.
The best accuracy of the training data is 65%, and the accuracy on Kaggle is 65.6%.

```
Accuracy of the network on the val images: 64 %
[48] loss: 0.198
Accuracy of the network on the val images: 64 %
[49] loss: 0.190
Accuracy of the network on the val images: 65 %
[50] loss: 0.192
Accuracy of the network on the val images: 65 %
Finished Training
```



In the Loading CIFAR100_SFU_CV part, I add the randomCrop and RandomHorizontalFlip and Normalize, since the accuracy increases after applying them. Here I normalize the data with mean 0.5 and std 0.5 which makes the network has a better accuracy than with mean 0 and std 1.
I referred to two websites: https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html https://www.aiworkbox.com/lessons/augment-the-cifar10-dataset-using-the-randomhorizontalflip-and-randomcrop-transforms

```
train_transform = transforms.Compose(
        [transforms.RandomCrop(32, 4),
          transforms.RandomHorizontalFlip(),
          transforms.ToTensor(),
          transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
          ])
test_transform = transforms.Compose(
        [transforms.ToTensor(),
          transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
          ])
```

Network Architecture

The network consists of six convolutional modules (conv-batchnorm-relu or conv-batchnorm-relu-maxpool) and three linear layers.

The architecture is input -> [conv->batchnorm->relu]*3 -> maxpool -> [conv->batchnorm->relu]*2 -> maxpool ->[conv->batchnorm->relu] -> [linear->batchnorm->relu]*2 -> linear.

Layer No.	Layer Type	Kernel Size	Input | Output Dimension	Input | Output Channels
1	conv2d	3	32 | 32	3 | 6
2	batchnorm2d	-	32 | 32	-
3	Relu	-	32 | 32	-
4	conv2d	3	32 | 30	6 | 16
5	batchnorm2d	-	30 | 30	-
6	Relu	-	30 | 30	-
7	conv2d	3	30 | 28	16 | 48
8	batchnorm2d	-	28 | 28	-
9	Relu	-	28 | 28	-
10	maxpool2d	2	28 | 14	-
11	conv2d	3	14 | 12	48 | 100
12	batchnorm2d	-	12 | 12	-
13	Relu	-	12 | 12	-
14	conv2d	3	12 | 12	100 | 300
15	batchnorm2d	-	12 | 12	-
16	Relu	-	12 | 12	-
17	maxpool2d	2	12 | 6	-
20	conv2d	3	6 | 4	300 | 600
21	batchnorm2d	-	4 | 4	-
22	Relu	-	4 | 4	-
23	Linear	-	9600 | 2000	-
24	batchnorm1d	-	2000 | 2000	-
25	Relu	-	2000 | 2000	-
26	Linear	-	2000 | 800	-
27	batchnorm1d	-	800 | 800	-
28	Relu	-	800 | 800	-
29	Linear	-	800 | 100	-

For the network structure, I inspired from VGGNet. I set the kernel of Conv layer to 3. The network only contains two max pool layers to avoid too much loss. In some Conv layers, I use padding to overcome the corner information loss. And it can keep the same size for the output that the network can be deeper with more layers. I make the output channels be between 2 and 3 times of the input channels in Conv layers, because our images are size of 3*32*32 that using the same parameters of VGGNet is not suitable. And I found setting a decreased output size in each linear layer has a higher accuracy than keeping the output size unchanged in the first two linear layers. Without changing other things, when normalization layers inserted before Relu layers, the accuracy is 65%, while it is only 62% when normalization layers after Relu layers. Thus, I put batchnorm before Relu.

```
self.conv1 = nn.Conv2d(3, 6, 3, padding=1)
self.Norm1 = nn.BatchNorm2d(6)
self.conv2 = nn.Conv2d(6, 16, 3)
self.Norm2 = nn.BatchNorm2d(16)
self.conv3 = nn.Conv2d(16, 48, 3)
self.Norm3 = nn.BatchNorm2d(48)
self.conv4 = nn.Conv2d(48, 100, 3)
self.Norm4 = nn.BatchNorm2d(100)
self.conv5 = nn.Conv2d(100, 300, 3, padding=1)
self.Norm5 = nn.BatchNorm2d(300)
self.conv6 = nn.Conv2d(300, 600, 3)
self.Norm6 = nn.BatchNorm2d(600)

self.pool = nn.MaxPool2d(2, 2)
```

```
self.fc_net = nn.Sequential(
        nn.Linear(600 * 4 * 4, 2000),
        nn.BatchNorm1d(2000),
        nn.ReLU(inplace=True),

        nn.Linear(2000, 800),
        nn.BatchNorm1d(800),
        nn.ReLU(inplace=True),

        nn.Linear(800, TOTAL_CLASSES)
)
```

```
def forward(self, x):

        # <<TODO#3&#4>> Based on the above edits, you'll have
        # to edit the forward pass description here.

        x = F.relu(self.Norm1(self.conv1(x)))
        # Output size = 32 * 32
        x = F.relu(self.Norm2(self.conv2(x)))
        # Output size = 30 * 30
        x = self.pool(F.relu(self.Norm3(self.conv3(x))))
        # Output size = 28//2 * 28//2 = 14 * 14
        x = F.relu(self.Norm4(self.conv4(x)))
        # Output size = 12 * 12
        x = self.pool(F.relu(self.Norm5(self.conv5(x))))
        # Output size = 12//2 * 12//2 = 6 * 6
        x = F.relu(self.Norm6(self.conv6(x)))
        # Output size = 4 * 4

        # See the CS231 link to understand why this is 16*5*5!
        # This will help you design your own deeper network
        x = x.view(-1, 600 * 4 * 4)
        x = self.fc_net(x)

        # No softmax is needed as the loss function in step 3
        # takes care of that

        return x
```

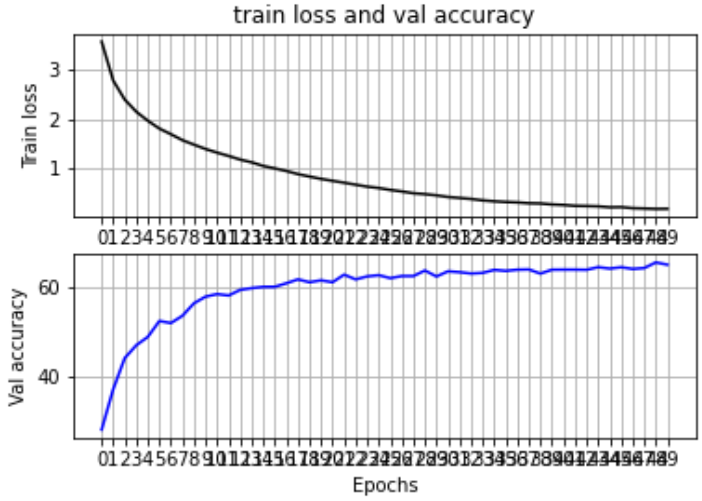
I let the learning rate be 0.001, 0.002, 0.004, 0.005, 0.006, 0.008, 0.01. The 0.006 learning rate makes the best accuracy. The momentum is useful. The optimizer with the momentum makes the model increase 2% validation accuracy.

```
optimizer = optim.SGD(net.parameters(), lr=0.006, momentum=0.9)
```

I tried epochs with 15, 25, 40, 50, 60, 80. The model is underfitting with litter epochs, and overfitting with larger epochs. When having 40 or 50 epochs, the model has a better convergence. And 50 epochs have a better validation accuracy and a better testing accuracy on Kaggle.

```
EPOCHS = 50
```

This is the plot of training loss and the validation accuracy.

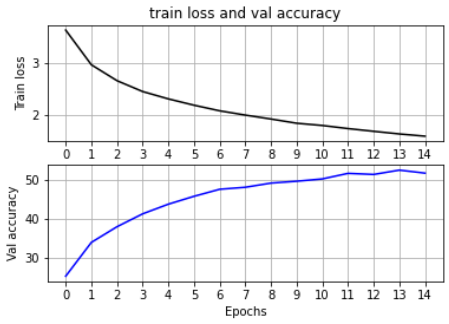


Ablation Study

submission_netid.csv 2 days ago by Natalie Zou add submission details	0.63100
submission_netid.csv 3 days ago by Natalie Zou add submission details	0.49700

The previous version: Got 51% accuracy for training data, and 49.7% accuracy for testing on Kaggle

```
[1] loss: 3.629
Accuracy of the network on the val images: 25 %
[2] loss: 2.965
Accuracy of the network on the val images: 33 %
[3] loss: 2.661
Accuracy of the network on the val images: 37 %
[4] loss: 2.453
Accuracy of the network on the val images: 41 %
[5] loss: 2.313
Accuracy of the network on the val images: 43 %
[6] loss: 2.192
Accuracy of the network on the val images: 45 %
[7] loss: 2.085
Accuracy of the network on the val images: 47 %
[8] loss: 2.003
Accuracy of the network on the val images: 48 %
[9] loss: 1.927
Accuracy of the network on the val images: 49 %
[10] loss: 1.847
Accuracy of the network on the val images: 49 %
[11] loss: 1.804
Accuracy of the network on the val images: 50 %
[12] loss: 1.745
Accuracy of the network on the val images: 51 %
[13] loss: 1.693
Accuracy of the network on the val images: 51 %
[14] loss: 1.642
Accuracy of the network on the val images: 52 %
[15] loss: 1.599
Accuracy of the network on the val images: 51 %
Finished Training
```



- The value of epochs was 15 and the learning rate was 0.005. (same as the original file)
- Added randomCrop and RandomHorizontalFilp and Normalize with mean 0 and std 1.

```
train_transform = transforms.Compose(
        [transforms.RandomCrop(32, 4),
          transforms.RandomHorizontalFlip(),
          transforms.ToTensor(),
          transforms.Normalize((0, 0, 0), (1, 1, 1))
          ])
test_transform = transforms.Compose(
        [transforms.ToTensor(),
          transforms.Normalize((0, 0, 0), (1, 1, 1))
          ])
```

- There were five convolution layer modules and three linear layers.
  The network architecture was:
  input -> [[conv->batchnorm->relu]*2 -> maxpool]*2 -> [conv->batchnorm->relu] -> [linear->batchnorm->relu]*2 -> linear

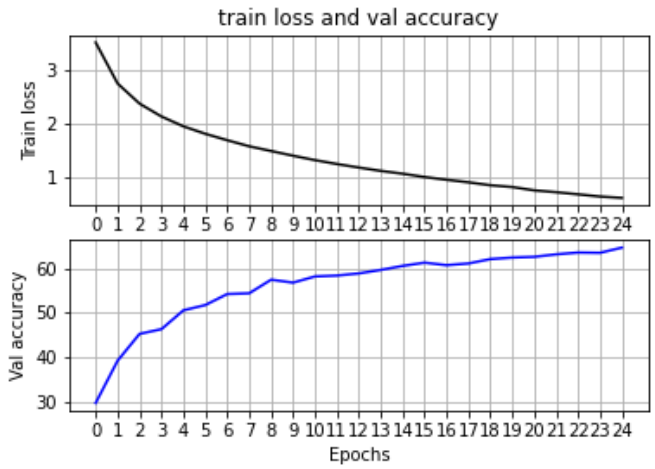
Layer No.	Layer Type	Kernel size	Input | Output Dimension	Input | Output Channels
1	conv2d	3	32 | 30	3 | 6
2	batchnorm2d	-	30 | 30	-
3	Relu	-	30 | 30	-
4	conv2d	3	30 | 28	6 | 16
5	batchnorm2d	-	28 | 28	-
6	Relu	-	28 | 28	-
7	maxpool2d	2	28 | 14	-
8	conv2d	3	14 | 12	16 | 48
9	batchnorm2d	-	12 | 12	-
10	Relu	-	12 | 12	-
11	conv2d	3	12 | 10	48 | 100
12	batchnorm2d	-	10 | 10	-
13	Relu	-	10 | 10	-
14	maxpool2d	2	10 | 5	-
15	conv2d	3	5 | 3	100 | 300
16	batchnorm2d	-	3 | 3	-
17	Relu	-	3 | 3	-
20	Linear	-	2700 | 1000	-
21	batchnorm1d	-	1000 | 1000	-
22	Relu	-	1000 | 1000	-
23	Linear	-	1000 | 400	-
24	batchnorm1d	-	400 | 400	-
25	Relu	-	400 | 400	-
26	Linear	-	400 | 100	-

```
self.conv1  =  nn.Conv2d(3,  6,  3)
self.Norm1  =  nn.BatchNorm2d(6)
self.conv2  =  nn.Conv2d(6,  16,  3)
self.Norm2  =  nn.BatchNorm2d(16)
self.conv3  =  nn.Conv2d(16,  48,  3)
self.Norm3  =  nn.BatchNorm2d(48)
self.conv4  =  nn.Conv2d(48,  100,  3)
self.Norm4  =  nn.BatchNorm2d(100)
self.conv5  =  nn.Conv2d(100,  300,  3)
self.Norm5  =  nn.BatchNorm2d(300)

self.pool1  =  nn.MaxPool2d(2,  2)

self.fc_net  =  nn.Sequential(
        nn.Linear(300  *  3  *  3,  1000),
        nn.BatchNorm1d(1000),
        nn.ReLU(inplace=True),

        nn.Linear(1000,  400),
        nn.BatchNorm1d(400),
        nn.ReLU(inplace=True),

        nn.Linear(400,  TOTAL_CLASSES)
)
```

```
def  forward(self,  x):

        x  =  F.relu(self.Norm1(self.conv1(x)))
        x  =  self.pool1(F.relu(self.Norm2(self.conv2(x))))
        x  =  F.relu(self.Norm3(self.conv3(x)))
        x  =  self.pool1(F.relu(self.Norm4(self.conv4(x))))
        x  =  F.relu(self.Norm5(self.conv5(x)))

        x  =  x.view(-1,  300  *  3  *  3)
        x  =  self.fc_net(x)

        return  x
```

The edited version: Got 64% accuracy for training data, and 63.1% accuracy for testing on Kaggle

```
[1] loss: 3.522
Accuracy of the network on the val images: 29 %
[2] loss: 2.757
Accuracy of the network on the val images: 39 %
[3] loss: 2.382
Accuracy of the network on the val images: 45 %
[4] loss: 2.142
Accuracy of the network on the val images: 46 %
[5] loss: 1.958
Accuracy of the network on the val images: 50 %
[6] loss: 1.819
Accuracy of the network on the val images: 51 %
[7] loss: 1.699
Accuracy of the network on the val images: 54 %
[8] loss: 1.586
Accuracy of the network on the val images: 54 %
[9] loss: 1.498
Accuracy of the network on the val images: 57 %
[10] loss: 1.410
Accuracy of the network on the val images: 56 %
[11] loss: 1.328
Accuracy of the network on the val images: 58 %
[12] loss: 1.256
Accuracy of the network on the val images: 58 %
[13] loss: 1.190
Accuracy of the network on the val images: 58 %
[14] loss: 1.127
Accuracy of the network on the val images: 59 %
[15] loss: 1.074
Accuracy of the network on the val images: 60 %
[16] loss: 1.013
Accuracy of the network on the val images: 61 %
[17] loss: 0.962
Accuracy of the network on the val images: 60 %
[18] loss: 0.914
Accuracy of the network on the val images: 61 %
[19] loss: 0.859
Accuracy of the network on the val images: 62 %
[20] loss: 0.826
Accuracy of the network on the val images: 62 %
[21] loss: 0.764
Accuracy of the network on the val images: 62 %
[22] loss: 0.730
Accuracy of the network on the val images: 63 %
[23] loss: 0.690
Accuracy of the network on the val images: 63 %
[24] loss: 0.649
Accuracy of the network on the val images: 63 %
[25] loss: 0.623
Accuracy of the network on the val images: 64 %
Finished Training
```



- The learning rate was still 0.005. I added the value of epochs to 25, since 15 epochs did not converge enough.

```
EPOCHS = 25
```

- I kept data normalization and data augmentation same as the previous one.

- I changed the network architecture. I set padding for some layers and added one more convolution layer module. I also adjusted the input and output size of linear layers. Then, there were six convolution layer modules and three linear layers. This network architecture also is used in the final version:
  input -> [conv->batchnorm->relu]*3 -> maxpool -> [conv->batchnorm->relu]*2 -> maxpool ->[conv->batchnorm->relu] -> [linear->batchnorm->relu]*2 -> linear

Layer No.	Layer Type	Kernel Size	Input | Output Dimension	Input | Output Channels
1	conv2d	3	32 | 32	3 | 6
2	batchnorm2d	-	32 | 32	-
3	Relu	-	32 | 32	-
4	conv2d	3	32 | 30	6 | 16
5	batchnorm2d	-	30 | 30	-
6	Relu	-	30 | 30	-
7	conv2d	3	30 | 28	16 | 48
8	batchnorm2d	-	28 | 28	-
9	Relu	-	28 | 28	-
10	maxpool2d	2	28 | 14	-
11	conv2d	3	14 | 12	48 | 100
12	batchnorm2d	-	12 | 12	-
13	Relu	-	12 | 12	-
14	conv2d	3	12 | 12	100 | 300
15	batchnorm2d	-	12 | 12	-
16	Relu	-	12 | 12	-
17	maxpool2d	2	12 | 6	-
20	conv2d	3	6 | 4	300 | 600
21	batchnorm2d	-	4 | 4	-
22	Relu	-	4 | 4	-
23	Linear	-	9600 | 2000	-
24	batchnorm1d	-	2000 | 2000	-
25	Relu	-	2000 | 2000	-
26	Linear	-	2000 | 800	-
27	batchnorm1d	-	800 | 800	-
28	Relu	-	800 | 800	-
29	Linear	-	800 | 100	-

```
self.conv1 = nn.Conv2d(3, 6, 3, padding=1)
self.Norm1 = nn.BatchNorm2d(6)
self.conv2 = nn.Conv2d(6, 16, 3)
self.Norm2 = nn.BatchNorm2d(16)
self.conv3 = nn.Conv2d(16, 48, 3)
self.Norm3 = nn.BatchNorm2d(48)
self.conv4 = nn.Conv2d(48, 100, 3)
self.Norm4 = nn.BatchNorm2d(100)
self.conv5 = nn.Conv2d(100, 300, 3, padding=1)
self.Norm5 = nn.BatchNorm2d(300)
self.conv6 = nn.Conv2d(300, 600, 3)
self.Norm6 = nn.BatchNorm2d(600)

self.pool = nn.MaxPool2d(2, 2)

self.fc_net = nn.Sequential(
        nn.Linear(600 * 4 * 4, 2000),
        nn.BatchNorm1d(2000),
        nn.ReLU(inplace=True),

        nn.Linear(2000, 800),
        nn.BatchNorm1d(800),
        nn.ReLU(inplace=True),

        nn.Linear(800, TOTAL_CLASSES)
)
```

```
def forward(self, x):

        x = F.relu(self.Norm1(self.conv1(x)))
        x = F.relu(self.Norm2(self.conv2(x)))
        x = self.pool(F.relu(self.Norm3(self.conv3(x))))
        x = F.relu(self.Norm4(self.conv4(x)))
        x = self.pool(F.relu(self.Norm5(self.conv5(x))))
        x = F.relu(self.Norm6(self.conv6(x)))
        x = x.view(-1, 600 * 4 * 4)
        x = self.fc_net(x)

        return x
```

After this editing, it is obvious that the validation accuracy increased from 51% to 64%, and the testing accuracy increased from 49.7% to 63.1%.

Part 2

Using ResNet as a fixed feature

The train accuracy is 0.4933, the test accuracy is 0.4257. Here, resnet_last_only is True.

```
TRAINING Epoch 1/50 Loss 0.1745 Accuracy 0.0077
TRAINING Epoch 2/50 Loss 0.1647 Accuracy 0.0163
TRAINING Epoch 3/50 Loss 0.1563 Accuracy 0.0363
TRAINING Epoch 4/50 Loss 0.1484 Accuracy 0.0697
TRAINING Epoch 5/50 Loss 0.1413 Accuracy 0.0913
TRAINING Epoch 6/50 Loss 0.1346 Accuracy 0.1240
TRAINING Epoch 7/50 Loss 0.1291 Accuracy 0.1603
TRAINING Epoch 8/50 Loss 0.1243 Accuracy 0.1747
TRAINING Epoch 9/50 Loss 0.1198 Accuracy 0.2073
TRAINING Epoch 10/50 Loss 0.1162 Accuracy 0.2353
TRAINING Epoch 11/50 Loss 0.1127 Accuracy 0.2400
TRAINING Epoch 12/50 Loss 0.1091 Accuracy 0.2600
TRAINING Epoch 13/50 Loss 0.1076 Accuracy 0.2780
TRAINING Epoch 14/50 Loss 0.1028 Accuracy 0.3140
TRAINING Epoch 15/50 Loss 0.1026 Accuracy 0.2993
TRAINING Epoch 16/50 Loss 0.0986 Accuracy 0.3287
TRAINING Epoch 17/50 Loss 0.0972 Accuracy 0.3327
TRAINING Epoch 18/50 Loss 0.0949 Accuracy 0.3477
TRAINING Epoch 19/50 Loss 0.0933 Accuracy 0.3510
TRAINING Epoch 20/50 Loss 0.0908 Accuracy 0.3713
TRAINING Epoch 21/50 Loss 0.0889 Accuracy 0.3893
TRAINING Epoch 22/50 Loss 0.0883 Accuracy 0.3943
TRAINING Epoch 23/50 Loss 0.0857 Accuracy 0.4083
TRAINING Epoch 24/50 Loss 0.0859 Accuracy 0.4003
TRAINING Epoch 25/50 Loss 0.0848 Accuracy 0.4177
TRAINING Epoch 26/50 Loss 0.0826 Accuracy 0.4227
TRAINING Epoch 27/50 Loss 0.0822 Accuracy 0.4183
TRAINING Epoch 28/50 Loss 0.0823 Accuracy 0.4187
TRAINING Epoch 29/50 Loss 0.0790 Accuracy 0.4430
TRAINING Epoch 30/50 Loss 0.0799 Accuracy 0.4337
TRAINING Epoch 31/50 Loss 0.0783 Accuracy 0.4527
TRAINING Epoch 32/50 Loss 0.0785 Accuracy 0.4473
TRAINING Epoch 33/50 Loss 0.0764 Accuracy 0.4477
TRAINING Epoch 34/50 Loss 0.0759 Accuracy 0.4657
TRAINING Epoch 35/50 Loss 0.0750 Accuracy 0.4663
TRAINING Epoch 36/50 Loss 0.0751 Accuracy 0.4670
TRAINING Epoch 37/50 Loss 0.0761 Accuracy 0.4527
TRAINING Epoch 38/50 Loss 0.0735 Accuracy 0.4807
TRAINING Epoch 39/50 Loss 0.0728 Accuracy 0.4673
TRAINING Epoch 40/50 Loss 0.0739 Accuracy 0.4633
TRAINING Epoch 41/50 Loss 0.0716 Accuracy 0.4813
TRAINING Epoch 42/50 Loss 0.0715 Accuracy 0.4777
TRAINING Epoch 43/50 Loss 0.0715 Accuracy 0.4750
TRAINING Epoch 44/50 Loss 0.0711 Accuracy 0.4863
TRAINING Epoch 45/50 Loss 0.0693 Accuracy 0.4923
TRAINING Epoch 46/50 Loss 0.0691 Accuracy 0.4960
TRAINING Epoch 47/50 Loss 0.0696 Accuracy 0.4897
TRAINING Epoch 48/50 Loss 0.0689 Accuracy 0.4947
TRAINING Epoch 49/50 Loss 0.0687 Accuracy 0.4897
TRAINING Epoch 50/50 Loss 0.0685 Accuracy 0.4933
Finished Training
```

```
Test Loss: 0.0772 Test Accuracy 0.4257
```

Fine-tuning:

The train accuracy is 0.8413, the test accuracy is 0.5562. Here, resnet_last_only is False.

```
TRAINING Epoch 1/50 Loss 0.1724 Accuracy 0.0083
TRAINING Epoch 2/50 Loss 0.1533 Accuracy 0.0443
TRAINING Epoch 3/50 Loss 0.1349 Accuracy 0.0997
TRAINING Epoch 4/50 Loss 0.1203 Accuracy 0.1633
TRAINING Epoch 5/50 Loss 0.1076 Accuracy 0.2467
TRAINING Epoch 6/50 Loss 0.0987 Accuracy 0.3077
TRAINING Epoch 7/50 Loss 0.0910 Accuracy 0.3523
TRAINING Epoch 8/50 Loss 0.0835 Accuracy 0.4047
TRAINING Epoch 9/50 Loss 0.0783 Accuracy 0.4343
TRAINING Epoch 10/50 Loss 0.0722 Accuracy 0.4790
TRAINING Epoch 11/50 Loss 0.0691 Accuracy 0.5177
TRAINING Epoch 12/50 Loss 0.0644 Accuracy 0.5277
TRAINING Epoch 13/50 Loss 0.0614 Accuracy 0.5640
TRAINING Epoch 14/50 Loss 0.0569 Accuracy 0.5950
TRAINING Epoch 15/50 Loss 0.0563 Accuracy 0.5950
TRAINING Epoch 16/50 Loss 0.0525 Accuracy 0.6170
TRAINING Epoch 17/50 Loss 0.0503 Accuracy 0.6387
TRAINING Epoch 18/50 Loss 0.0482 Accuracy 0.6540
TRAINING Epoch 19/50 Loss 0.0467 Accuracy 0.6667
TRAINING Epoch 20/50 Loss 0.0446 Accuracy 0.6853
TRAINING Epoch 21/50 Loss 0.0427 Accuracy 0.6987
TRAINING Epoch 22/50 Loss 0.0423 Accuracy 0.6953
TRAINING Epoch 23/50 Loss 0.0397 Accuracy 0.7147
TRAINING Epoch 24/50 Loss 0.0400 Accuracy 0.7120
TRAINING Epoch 25/50 Loss 0.0382 Accuracy 0.7303
TRAINING Epoch 26/50 Loss 0.0362 Accuracy 0.7390
TRAINING Epoch 27/50 Loss 0.0355 Accuracy 0.7520
TRAINING Epoch 28/50 Loss 0.0355 Accuracy 0.7500
TRAINING Epoch 29/50 Loss 0.0343 Accuracy 0.7533
TRAINING Epoch 30/50 Loss 0.0335 Accuracy 0.7607
TRAINING Epoch 31/50 Loss 0.0326 Accuracy 0.7657
TRAINING Epoch 32/50 Loss 0.0309 Accuracy 0.7837
TRAINING Epoch 33/50 Loss 0.0307 Accuracy 0.7823
TRAINING Epoch 34/50 Loss 0.0294 Accuracy 0.7917
TRAINING Epoch 35/50 Loss 0.0282 Accuracy 0.8030
TRAINING Epoch 36/50 Loss 0.0292 Accuracy 0.7970
TRAINING Epoch 37/50 Loss 0.0285 Accuracy 0.7997
TRAINING Epoch 38/50 Loss 0.0265 Accuracy 0.8110
TRAINING Epoch 39/50 Loss 0.0261 Accuracy 0.8197
TRAINING Epoch 40/50 Loss 0.0268 Accuracy 0.8103
TRAINING Epoch 41/50 Loss 0.0241 Accuracy 0.8367
TRAINING Epoch 42/50 Loss 0.0251 Accuracy 0.8293
TRAINING Epoch 43/50 Loss 0.0235 Accuracy 0.8317
TRAINING Epoch 44/50 Loss 0.0243 Accuracy 0.8323
TRAINING Epoch 45/50 Loss 0.0237 Accuracy 0.8393
TRAINING Epoch 46/50 Loss 0.0225 Accuracy 0.8473
TRAINING Epoch 47/50 Loss 0.0223 Accuracy 0.8467
TRAINING Epoch 48/50 Loss 0.0218 Accuracy 0.8527
TRAINING Epoch 49/50 Loss 0.0227 Accuracy 0.8427
TRAINING Epoch 50/50 Loss 0.0229 Accuracy 0.8413
Finished Training
```

```
Test Loss: 0.0561 Test Accuracy 0.5562
```

The num_epochs is 50.
The learning_rate is 0.0015.
The batch_size is 32.
I set the resnet_last_only to False for fine-tuning.

I set RandomResizedCrop() for the training data. And to keep the suitable size, I resize the testing data. I use RandomHorizontalFlip(), set weight_decay and add a dropout layer to reduce the overfitting.

```
#TODO1: Load pre-trained ResNet Model
self.resnet18 = models.resnet18(pretrained=True)
```

```
#TODO2: Replace fc layer in resnet to a
self.resnet18.fc = nn.Sequential(
        nn.Dropout(),
        nn.Linear(num_feats, num_classes)
        )
```

```
#TODO3: Forward pass x throug
x = self.resnet18(x)
```

```
NUM_EPOCHS = 50
LEARNING_RATE = 0.0015
BATCH_SIZE = 32
RESNET_LAST_ONLY = False
```

```
data_transforms = {
    'train': transforms.Compose([
        transforms.Resize(256),
        #TODO: Transforms.RandomResizedCrop() instead of Ce
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        #TODO: Transforms.Normalize()
        transforms.Normalize((0, 0, 0), (1, 1, 1))
    ]),
    'test': transforms.Compose([
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
        #TODO: Transforms.Normalize()
        transforms.Normalize((0, 0, 0), (1, 1, 1))
    ]),
}
```

```
#Setting the optimizer and loss criterion
optimizer = optim.SGD(model.parameters(), lr=LEARNING_RATE, momentum=0.9, weight_decay = 0.0005)
```