

Qidi Han (6814891757)
EE-569
Homework #5 part 2
April-22-2020

Problem 1: Competition

I. Abstract and Motivation

As the first part of the project, we build the simple CNN with LeNet5, but the simple LeNet5 will have lower accuracy on the test data. The accuracy of the test data is around only 64%. There are many new ways to improve the test accuracy, for example, ResNet and AlexNet. But the model sizes are very big, to learn those new method will help me to better understand the concept and build a new model.

II. Approach and Procedures

After I read more paper about improving LeNet5, I conclude that there are two ways to improve the LeNet5. The first way, I will not add any new layers, but only modify the train data and the initial weight function. The second way, I will add more layers to the model, the filter will be different with LeNet5. I will fully explain the process in the next paragraph.

a. Without adding the layers

There are three things that I modified. The first thing is mean and the standard deviation for the transform. As we learned that zero-centered data and normalized data will give us much better result. Zero-centered mean we subtract the overall mean for each image. Normalized means we divide image with overall standard deviation. I found the mean and standard deviation for the Cifar10 is [0.4914,0.4822,0.4465] and [0.2023,0.1994,0.2010]. I also compute this by using the TensorFlow. Figure 1 shows the mean and std for the Cifar10. The reason of doing zero-centered data and normalized data is because for each train image, we will have different weights and biases to multiple and add. Those weight and biases will cause non-similar range of each features or images. After the zero-centered data and normalized data, those similar range of the features and image will give us a better gradient, it will have less chance to get overfitting issues. And the non-normalized image will give a wide range of the weight for each features and images. Figure 2 shows the feature distribution after zero-centered and normalization.

The second thing that I modified is data augmented. I tried many different ways to make the data augmented. The first way is RandomCrop(), it use for cutting the image with certain range. Since our input image is 32*32, we only can cut the image start 32*32, and I set the padding with 4. The second way is use RandomHorizontalFlip(), it selects random figures to flip at horizontal way. The third way is use RandomRotation(), it selects random number figures to rotation certain degrees, for my experiment of trying many numbers, degree of 10 give me the best answer. The four way is ColorJitter(), it uses for change the color of the image, for some image, the color are very bright, and for some image, the color are very dark. I google the best setting for ColorJitter() of Cifar10, it give me brightness=0.2, contrast=0.2. Those four are the main ways to do the data augmentation. We can random select some of those function to use, but

I find out that RandomCrop() have to use at first position, otherwise, the test accuracy will very low. Figure 3 shows the best pair of data augmentation. Those function will increase the number of train data for our experiment. The test accuracy will definitely increase.

The third thing that I modify is the weight initialization, this modification did not change the result too much, but it helps me to process the program much fast. As we mentioned in the discussion, TA give us zero-weight and small random initialization will cause low test accuracy, because the gradient will vanish, and it will not have good gradient process in backpropagation. Therefore, I use the Xavier's initialization as the TA mentioned.

After I process the data with those three steps, I start to train the model, I use 200 epoch as the total. I use three different learning rates for the process, there are 0.01 at first 150 epoch, and 0.005 at next 60 epoch. Finally, I use 0.001 as the learning rate. Figure 4 shows the test accuracy, train accuracy and plot for CIFAR10 without change layers of LeNet5. I will discussion the result in the explanation part.

b. With adding more layers

After the tried many times without adding more layers, I try to add layers in the process, I borrow the idea from the paper "STRIVING FOR SIMPLICITY: THE ALL CONVOLUTIONAL NET". This paper gives me an idea of adding filter to the process will increase the accuracy, it's because the more filter will give us more information to the images, also he adds 3 additional convolution layers. The paper use 96 filter at first part convolution, and they use 192 filter at second part convolution. Since the model size is also an important part of our computation. I will delete two lays. Those multiple layers help us to extract more features. But more convolutional lay may cause overfitting, we have to keep the balance of doing the number of convolutional lays. From another perspective, adding more layers will add a greater number of weights. It also may cause overfitting, but since we already do the data augmentation for our question, this will help us to get lower chance of overfitting. There are total 7 layers. The first layer still is the same. It's $3 \times 32 \times 32$. It's size of the input image. Since the TA Yao said we put as many as result we can, I did many different models in my experiment. I will put the result on my table in the experimental result section. I will fully explain two of those. The first one has the best accuracy with largest model size. The second one has the lower model size, but slightly lower accuracy.

There are 5 layers in my experiment. The first layer is $32 \times 32 \times 3$ as I mentioned before, the second layer is $96 \times 30 \times 30$ since we use 3×3 filter in our problem. The next layer, I will have $96 \times 28 \times 28$, but the stride is 2. After this layer, there is a max-pooling layer. The mean of the max-pooling layer deletes all the unnecessary part of the image. There are many unnecessary parts of those image, but I don't want to delete too much. Therefore, I use the kernel size=5 and stride=1. Instead of the kernel size 3 and stride=2. I think it will remove too much information. I tried both without and with max pooling, add this additional max-pooling will give me a better result. This is same as the article said, the next layer will be different compare with the author. I will have $96 \times 26 \times 26$, instead of $192 \times 26 \times 26$. The reason of doing this is because I want to reduce the number of layers, therefore, I try 96,64,48 in our problem, the best result is 96. Therefore, I choose the 96. The last convolution layer has $192 \times 1 \times 1$. This is the same as paper, but I delete on layer with $192 \times 26 \times 26$. The reason still the same, I want to reduce the model size. After all the convolution layers, I will do a max pooling with size 6×6 . This is the same step as the paper did. I last output layer is 6912×10 . The 10 represent the information that goes to SoftMax layer. Figure 5 shows the design of my best solution.

The second design is similar with first design. But it has less model size, the model only has 186106 parameters. For this model, I add one more layer to our result, but I will divide last $96*96*26$ layers to $48*48*26$. Also, I will reduce the first two layers to 48 filter as well. I tried 96 and 48, the model size is double, but the reason is similar, therefore, I use 48 for my experiment. The first lay is $32*32*3$. The second layer is $48*30*30$. The third layer have $48*28*28$. The fourth layer has $96*26*26$. The reason of choosing this is because the author has the best answer with $96*26*26$ as his experiment in fourth layers. The next layer is $48*24*24$. The last convolution layer is $192*24*24$. This is very important layer, when I change 192 to 96, the accuracy reduces significantly. Then, I will do a global max pooling as the author did with (6,6) size. the output will have $192*6*6$ and I will reduce it wo 10 as the last layer that goes to SoftMax layers. All the layer that I mentioned before, I use Relu function. It also gives me the best answer. For all the model, I will 0.01 as the first 40 epochs, 0.001 as the next 20 epochs, 0.0001 as the next 10 epochs, and 0.00001 as the last 5 epochs. Figure 5b shows the second design.

Figure 6 shows the test accuracy, train accuracy and plot for CIFAR10 with adding more layers with those two modules. Figure 7 shows all the solution that I have. Figure 8 shows the computer information. To test how robust the model is, I will use my best accuracy model to change 3 different train data. The first test I use $\frac{1}{2}$ of the train image. The second test, I use $\frac{1}{4}$ of the train image, the last test, I use $\frac{1}{8}$ of the train image, Figure 9 shows the test accuracy, train accuracy and plot for different size of train images when I random drop some train images. The running time is not always constant, sometime, I run two model together. I will show couple running time for whole project in Figure 10. Figure 11 shows the inference time and train data time. The inference time is basically the test time for test data. I show 2 examples of doing this, but the inference time in the whole train process for each epoch is almost the same. The train data time is that time spend in training 5k images for one epoch.

III. Experimental Results

```

: import numpy as np

(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()
print(np.mean(train_images[:,:,:0]))
print(np.mean(train_images[:,:,:1]))
print(np.mean(train_images[:,:,:2]))
print(np.std(train_images[:,:,:0]))
print(np.std(train_images[:,:,:1]))
print(np.std(train_images[:,:,:2]))

```

125.306918046875
122.950394140625
113.86538318359375
62.993219278136884
62.08870764001421
66.70489964063091

Figure 1 shows the mean and std for the Cifar10

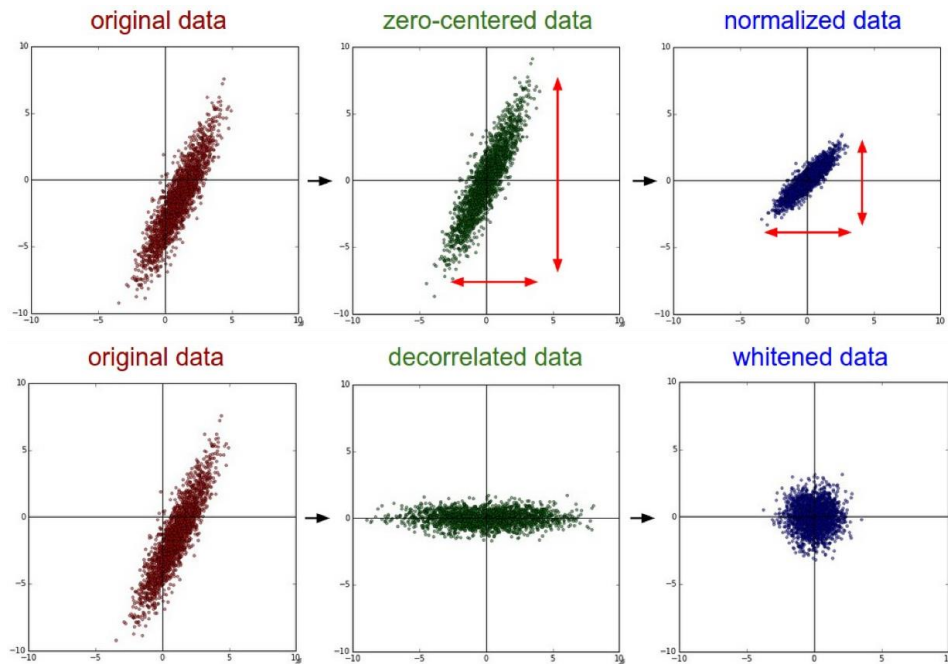


Figure 2 shows the feature distribution after zero-centered and normalization

```
def load_data(batch_size):
    trans1 = transforms.Compose([
        transforms.RandomCrop(32, padding=4),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))])

    trans = transforms.Compose(
        [transforms.ToTensor(),
        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))])
```

Figure 3 shows the best pair of data augmentation

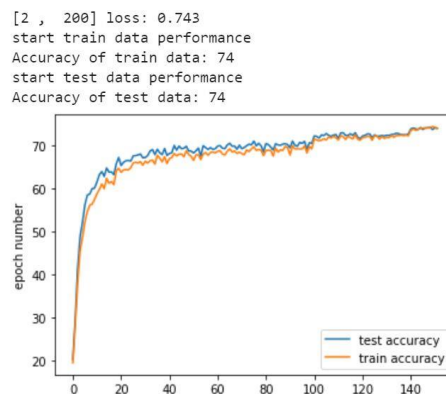


Figure 4 shows the test accuracy, train accuracy and plot for CIFAR10 without change layers of LeNet5.

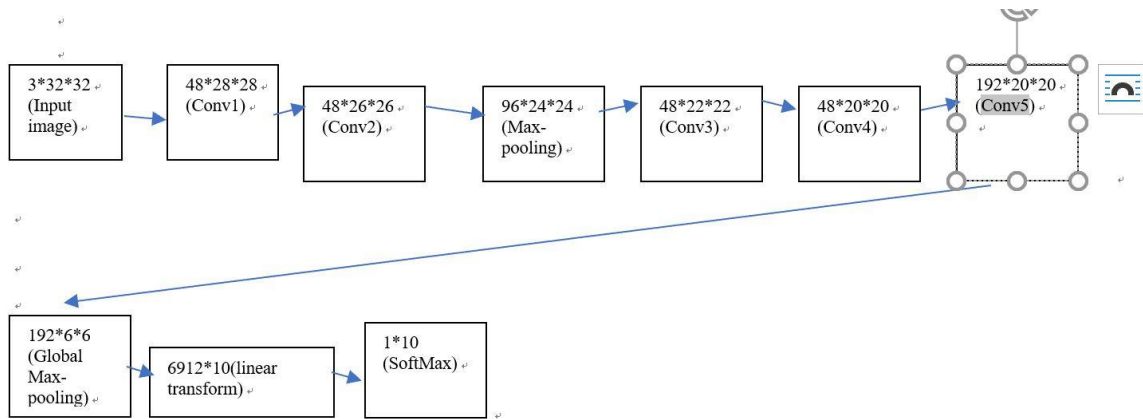


Figure 5 shows the best design

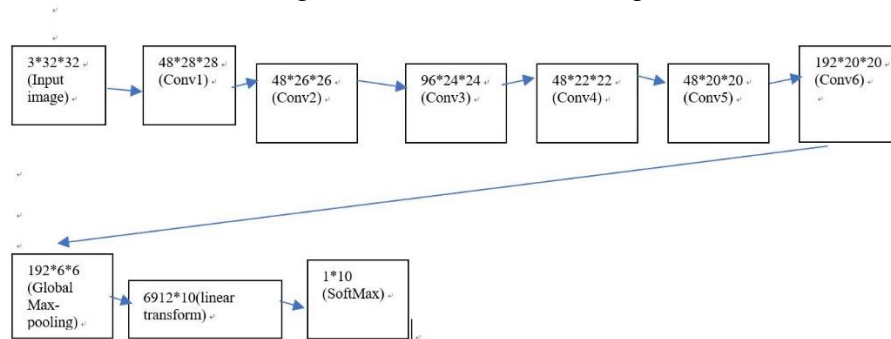


Figure 5b shows my second design

	Best Acc model	Less model size
Model size	339562	186106
plot	<pre> [1, 280] loss: 0.222 start train data performance Accuracy of train data: 92.31 time: 14.44 start test data performance epoch: 0 Accuracy of test data: 86.73 time: 1.75 </pre> <pre>]: print(np.max(train_rate)) print(np.max(test_rate)) 92.292 86.81 </pre>	<pre> Accuracy of train data: 87.02 start test data performance epoch: 4 Accuracy of test data: 84.13 </pre>

Figure 6 shows the test accuracy, train accuracy and plot for CIFAR10 with adding more layers

Model	Best Acc model	Less model size
Design	Figure 5	Figure 6
Model size	339562	186106
Test Accuracy	86.81	84.24
Train Accuracy	90.89	87.29

Model	Model A	Model B
Design	<pre> class Net(nn.Module): def __init__(self): super(Net, self).__init__() # kernel self.conv1 = nn.Conv2d(3, 32, 3).cuda() self.conv2 = nn.Conv2d(32, 32, 3, stride=2).cuda() self.conv3 = nn.Conv2d(32, 32, 3, stride=2).cuda() self.conv4 = nn.Conv2d(32, 192, 1).cuda() self.fc1 = nn.Linear(6912, 10).cuda() def forward(self, x): x=F.relu((self.conv1(x))) x=F.relu((self.conv2(x))) x=F.relu((self.conv3(x))) x=F.relu((self.conv4(x))) x = F.max_pool2d(x, 2).to(device) m=nn.AdaptiveAvgPool2d((6,6)).to(device) x = m(x) x = x.view(-1, self.num_flat_features(x)).to(device) x = self.fc1(x).to(device) return x def num_flat_features(self, x): size = x.size()[1:] # all dimensions except the batch dimension num_features = 1 for s in size: num_features *= s return num_features net = Net() print(net) Net((conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1)) (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(2, 2)) (conv3): Conv2d(32, 32, kernel_size=(3, 3), stride=(2, 2)) (conv4): Conv2d(32, 192, kernel_size=(1, 1), stride=(1, 1)) (fc1): Linear(in_features=6912, out_features=10, bias=True)) </pre>	<pre> class Net(nn.Module): def __init__(self): super(Net, self).__init__() # kernel self.conv1 = nn.Conv2d(3, 64, 3).cuda() self.conv2 = nn.Conv2d(64, 64, 3, stride=2).cuda() self.conv3 = nn.Conv2d(64, 64, 3, stride=2).cuda() self.conv4 = nn.Conv2d(64, 192, 1).cuda() self.fc1 = nn.Linear(6912, 10).cuda() def forward(self, x): x=F.relu((self.conv1(x))) x=F.relu((self.conv2(x))) x=F.relu((self.conv3(x))) x=F.relu((self.conv4(x))) x = F.max_pool2d(x, 2).to(device) m=nn.AdaptiveAvgPool2d((6,6)).to(device) x = m(x) x = x.view(-1, self.num_flat_features(x)).to(device) x = self.fc1(x).to(device) return x def num_flat_features(self, x): size = x.size()[1:] # all dimensions except the batch dimension num_features = 1 for s in size: num_features *= s return num_features net = Net() print(net) Net((conv1): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1)) (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(2, 2)) (conv3): Conv2d(64, 64, kernel_size=(3, 3), stride=(2, 2)) (conv4): Conv2d(64, 192, kernel_size=(1, 1), stride=(1, 1)) (fc1): Linear(in_features=6912, out_features=10, bias=True)) </pre>
Model size	94858	157258
Test Accuracy	79.78	82.31
Train Accuracy	79.212	82.662

Model	Model C	Model D
Design	<pre> class Net(nn.Module): def __init__(self): super(Net, self).__init__() # kernel self.conv1 = nn.Conv2d(3, 64, 3).cuda() self.conv2 = nn.Conv2d(64, 64, 3, stride=2).cuda() self.conv3 = nn.Conv2d(64, 96, 3, stride=2).cuda() self.conv4 = nn.Conv2d(96, 64, 3).cuda() self.conv5 = nn.Conv2d(64, 192, 1).cuda() self.fc1 = nn.Linear(6912, 10).cuda() def forward(self, x): x=F.relu(self.conv1(x)) x=F.relu(self.conv2(x)) x=F.relu(self.conv3(x)) x=F.relu(self.conv4(x)) x=F.relu(self.conv5(x)) x = F.max_pool2d(x, 2).to(device) m=nn.AdaptiveAvgPool2d((6,6)).to(device) x = m(x) x = x.view(-1, self.num_flat_features(x)).to(device) x = self.fc1(x).to(device) return x def num_flat_features(self, x): size = x.size()[1:] # all dimensions except the batch dimension num_features = 1 for s in size: num_features *= s return num_features net = Net() print(net) Net((conv1): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1)) (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(2, 2)) (conv3): Conv2d(64, 96, kernel_size=(3, 3), stride=(2, 2)) (conv4): Conv2d(96, 64, kernel_size=(3, 3), stride=(1, 1)) (conv5): Conv2d(64, 192, kernel_size=(1, 1), stride=(1, 1)) (fc1): Linear(in_features=6912, out_features=10, bias=True)) </pre>	<pre> class Net(nn.Module): def __init__(self): super(Net, self).__init__() # kernel self.conv1 = nn.Conv2d(3, 96, 3).cuda() self.conv2 = nn.Conv2d(96, 96, 3, stride=2).cuda() self.conv3 = nn.Conv2d(96, 96, 3, stride=2).cuda() self.conv4 = nn.Conv2d(96, 96, 3).cuda() self.conv5 = nn.Conv2d(96, 96, 1).cuda() self.conv6 = nn.Conv2d(96, 64, 1).cuda() self.fc1 = nn.Linear(2304, 10).cuda() def forward(self, x): x=F.relu(self.conv1(x)) x=F.relu(self.conv2(x)) x=F.relu(self.conv3(x)) x=F.relu(self.conv4(x)) x=F.relu(self.conv5(x)) x=F.relu(self.conv6(x)) m=nn.AdaptiveAvgPool2d((6,6)).to(device) x = m(x) x = x.view(-1, self.num_flat_features(x)).to(device) x = self.fc1(x).to(device) return x def num_flat_features(self, x): size = x.size()[1:] # all dimensions except the batch dimension num_features = 1 for s in size: num_features *= s return num_features net = Net() print(net) Net((conv1): Conv2d(3, 96, kernel_size=(3, 3), stride=(1, 1)) (conv2): Conv2d(96, 96, kernel_size=(3, 3), stride=(2, 2)) (conv3): Conv2d(96, 96, kernel_size=(3, 3), stride=(2, 2)) (conv4): Conv2d(96, 96, kernel_size=(3, 3), stride=(1, 1)) (conv5): Conv2d(96, 96, kernel_size=(1, 1), stride=(1, 1)) (conv6): Conv2d(96, 64, kernel_size=(1, 1), stride=(1, 1)) (fc1): Linear(in_features=2304, out_features=10, bias=True)) </pre>
Model size	231082	290378
Test Accuracy	84.46	84.81
Train Accuracy	87.59	88.514

Figure 7 shows all the solution that I have.

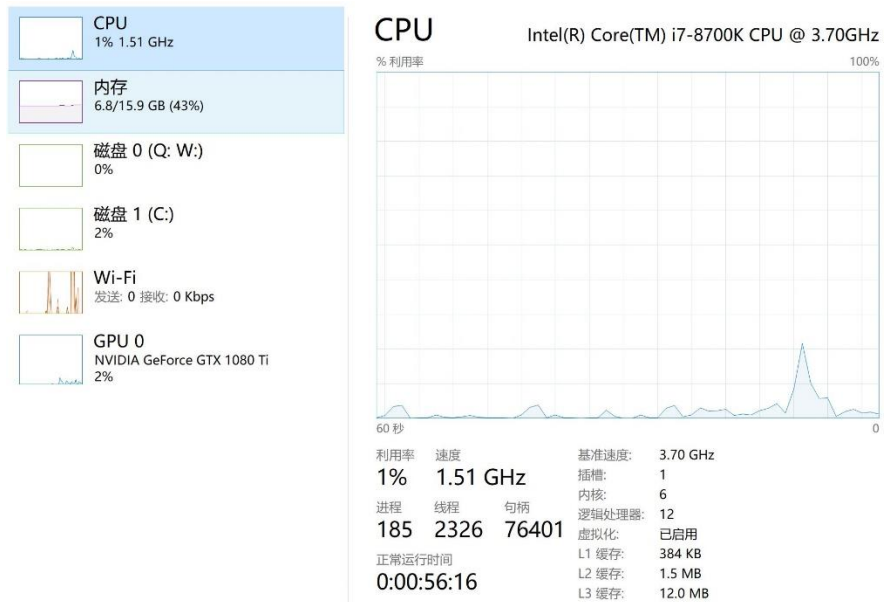
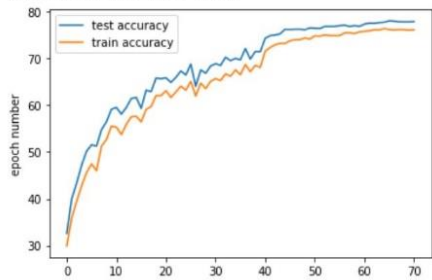
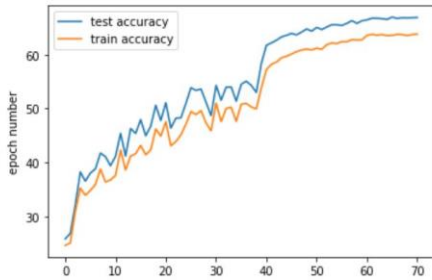


Figure 8 shows the computer information.

Model	Model 1	Model 1
Drop rate	1/2	1/4
Model size	339562	339562
Test Accuracy	84.56	81.64
Train Accuracy	85.474	82.072
Plot	<p>Accuracy of test data: 84.00</p> <p>1121.8523335456848</p> <pre>print(np.max(train_rate)) print(np.max(test_rate))</pre> <p>85.474 84.02</p>	<p>[1, 200] loss: 0.541 start train data performance Accuracy of train data: 81.66 start test data performance epoch: 0 Accuracy of test data: 81.61</p> <p>1072.1203830242157</p>

Model	Model 1	Model 1
-------	---------	---------

Drop rate	1/8	1/16
Model size	339562	339562
Test Accuracy	78.09	63.798
Train Accuracy	76.422	66.96
Plot	<pre>[1, 200] loss: 0.679 start train data performance Accuracy of train data: 76.16 start test data performance epoch: 0 Accuracy of test data: 77.93</pre>  <pre>1063.2795007228851 print(np.max(train_rate)) print(np.max(test_rate)) 76.422 78.09</pre>	<pre>[1, 200] loss: 1.006 start train data performance Accuracy of train data: 63.80 start test data performance epoch: 0 Accuracy of test data: 66.89</pre>  <pre>1017.1360099315643 print(np.max(train_rate)) print(np.max(test_rate)) 63.798 66.96</pre>

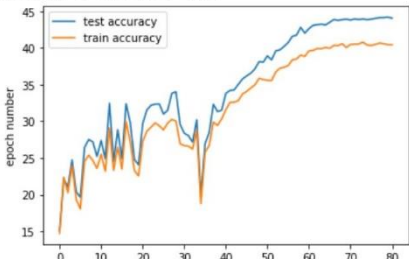

Model	Model 1	Model 1
Drop rate	1/32	1/64
Model size	339562	339562
Test Accuracy	43.94	15.12
Train Accuracy	40.778	15.366
Plot	<pre>start test data performance epoch: 4 Accuracy of test data: 44.05</pre>  <pre>1378.2486641407013 print(np.max(train_rate)) print(np.max(test_rate)) 40.778 43.94</pre>	 <pre>658.0212144851685 print(np.max(train_rate)) print(np.max(test_rate)) 15.366 15.12</pre>

Figure 9 shows the accuracy of different size of train images.

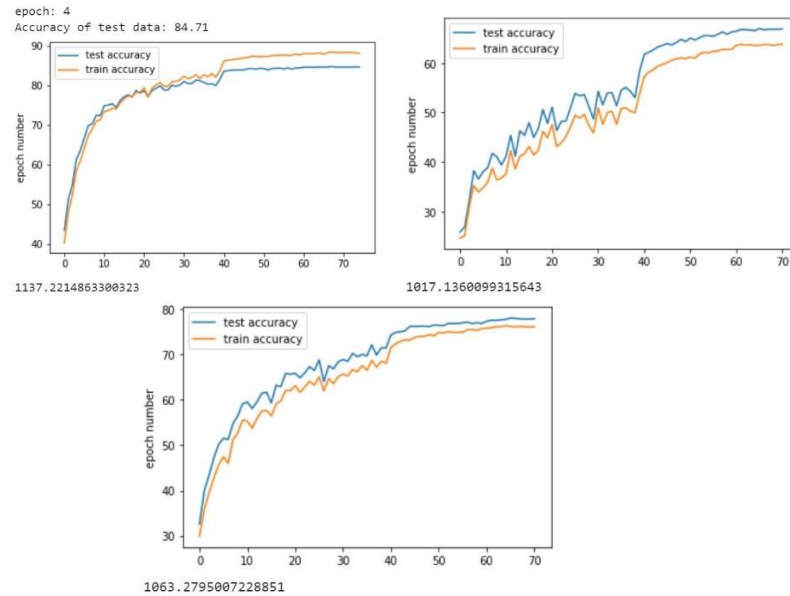


Figure 10 running time for whole process, include plot and everything.

```
[1 , 200] loss: 1.878
start train data performance
Accuracy of train data: 46.62
time: 16.30
start test data performance
epoch: 0
Accuracy of test data: 50.76
time: 1.75
[2 , 200] loss: 1.414
start train data performance
Accuracy of train data: 56.78
time: 15.19
start test data performance
epoch: 1
Accuracy of test data: 60.96
time: 1.74
[3 , 200] loss: 1.225
```

Figure 11 shows the inference time and train data time.

IV. Discussion

1: Motivation and logics behind your design:

I already fully explain the logics behind my design and the way of choosing the parameter, also the motivation of adding more layer, doing the data pre-process, data augmentation and initial weight vector. Also, already explain the advantage and disadvantage of adding more layers. I also make two version of the experiment. Figure 5 shows the design. I already explain the model in the approach section.

Comparison version1 with 1b: I will compare my first without adding layer with question 1b. The performance of my first part of the program is much better than the 1b. It has test

accuracy 74% and 98% train accuracy. The best result for 1b is 64%. I use similar number of epochs. I already explain the different with 1b in last section. Those data pre-process, data augmentation and initial weight vector definitely help us to give a better result, I also already mention the reason of improvement in each section of the approach section.

Comparison version 2 with 1b. The version 2 is built on the version 1. I already said that the benefit and disadvantage for adding more layers. The test accuracy for the test set is around 86.2%, and 91% on train set. And I only use totally 70 epochs. Since, I use the 1080ti GPU, the computing time didn't affect too much when I do the version 2.

2: Accuracy:

1b: test set 64%; train set 96% (40min with 200 epoch)

Version 1: test set 74%; train set 74% (46min with 200 epoch)

Version 2: I already put all the solution in the Figure 7. The best test data result is 86.81%. The best train data result is 92.31%.

1b will have less training time is because, the data is not augmented, also it's the simplest form of the training. The running time is around 17.8-21.2 mins. Figure 10 shows 3 different answers, the time is below the plot. Figure 11 shows the 5k train image process time for one epoch and 1k test image process time in one epoch, it also called inference time. The 5k train image will spend 16s for each epoch. The test data spend 1.75s for each epoch. This the data when I use the GPU. When I use the CPU, the time will increase significant.

Figure 4 and 6 shows performance of version 1 and version2. To test how robust is my model, I random drop some train data. Figure 9 shows the accuracy of different dropping rate. From the model, I think the model is robust. When the number of train data is only 6250, the accuracy still has 78.09%. When the train data is 3125, the accuracy drop to around 44%. It's still not that bad. But when the train data is 782, the accuracy drop to 15%, and then the accuracy only reminds at 10%, it has overfitting issues. But we still can say that the model is robust.

3: Model Size:

Figure 6 and 7 shows the model size. The model size of version 1 is the same as the 1b, it's around 62k. The model size of version 2 is around 330k and 180k.