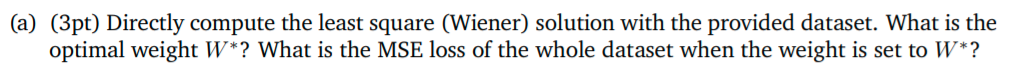
**5 Lab: LMS Algorithm (15 pts)**



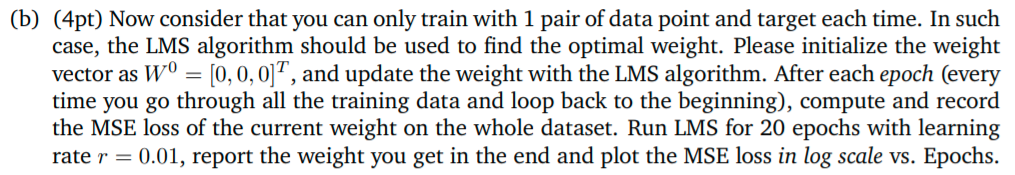
Solution: The optimal weight W\_star is

[[ 0.99769073]

[-2.00001451]

[ 2.99870453]]

Given W\_star, the MSE loss of the whole dataset is (6.145138742034196e-05).



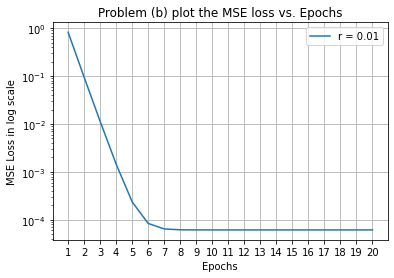
Solution: The weight I get after 20 epochs (W\_20\_Epochs in code) is

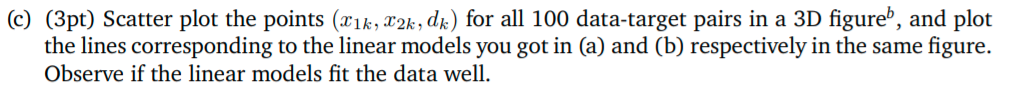
[[ 0.99761689]

[-1.99979502]

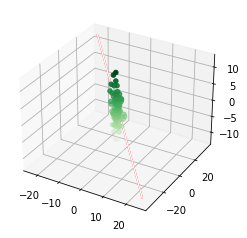
[ 2.99898621]]

Plot the MSE loss in log scale vs. Epochs

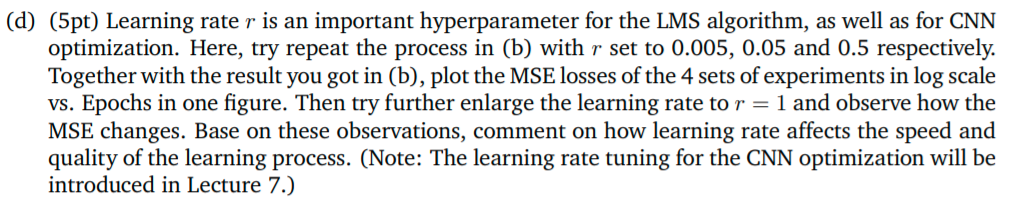




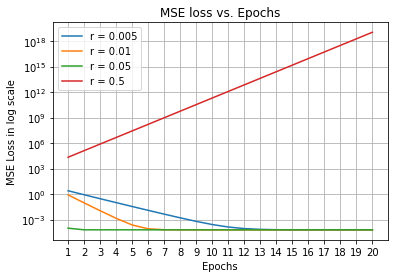
Solution: The 3D scatter plot along with two linear models is shown below.



In this 3D figure, a group of green points belong to all 100 data-target pairs, the linear model I got in (a) (W\_star) is the red line, and the linear model I got in (b) (W\_20\_Epochs) is the white line. It could be seen that the two linear models are very close to each other, and both models fit the 100 data-target pairs relatively well.



Solution: Plot the MSE losses of the 5 sets of experiments in log scale vs. Epochs in one figure.



Enlarge the learning rate to r = 1 and observe how the MSE changes. Shown below are MSE values after each Epoch.

MSE after Epoch 1: 3.6798545341445733e+31

MSE after Epoch 2: 5.402768778759977e+60

MSE after Epoch 3: 7.932354446216017e+89

MSE after Epoch 4: 1.164629648926877e+119

MSE after Epoch 5: 1.7099112607185237e+148

MSE after Epoch 6: 2.5104946643133094e+177

MSE after Epoch 7: 3.685912599287265e+206

MSE after Epoch 8: 5.411663240200822e+235

MSE after Epoch 9: 7.945413309855397e+264

MSE after Epoch 10: 1.1665469535403863e+294

MSE after Epoch 11: inf

MSE after Epoch 12: inf

MSE after Epoch 13: inf

MSE after Epoch 14: inf

MSE after Epoch 15: inf

MSE after Epoch 16: inf

MSE after Epoch 17: inf

MSE after Epoch 18: inf

MSE after Epoch 19: inf

MSE after Epoch 20: inf

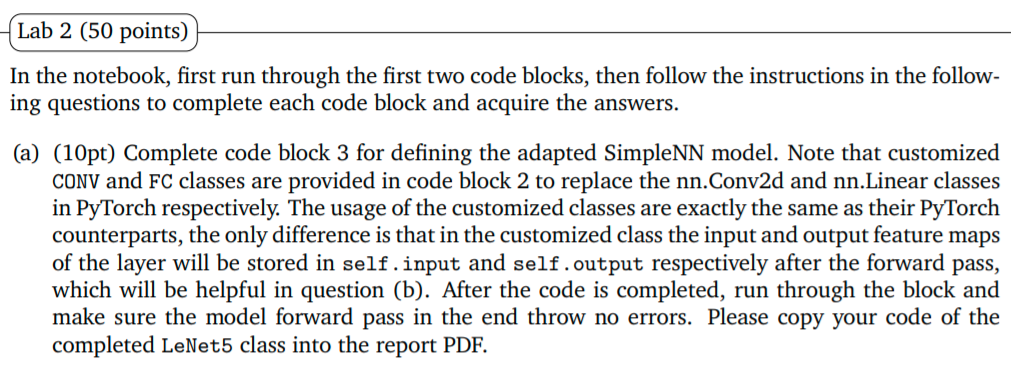
Given learning rate r = 1 and its MSE losses shown above, the MSE loss gradually becomes so large that finally leads to numerical difficulty, making the LMS model training fail.

Comment: Based on all 6 observations, if the learning rate is too small, it would take a very long time for model training to achieve convergence (finding the optimal solution), although the model accuracy could be relatively good.

Although increasing the learning rate could speed up model training, if the learning rate is too large, then the model training might fail (training loss becomes so large that the model fails to converge), which means the model accuracy would be really bad.

Therefore, it is very important to maintain the trade-off between a small learning rate and a large learning rate, through either empirical knowledge or many experiments.

**6 Lab: Simple NN (40 pts)**



Solution: My code of the completed LeNet5 class is shown below.

# Create the neural network module: LeNet-5

class SimpleNN(nn.Module):

def \_\_init\_\_(self):

super(SimpleNN, self).\_\_init\_\_()

# Layer definition

self.conv1 = CONV(in\_channels = 3,

out\_channels = 32,

kernel\_size = 5,

stride = 1,

padding = 2) #Your code here

self.conv2 = CONV(in\_channels = 32,

out\_channels = 32,

kernel\_size = 5,

stride = 1,

padding = 2) #Your code here

self.conv3 = CONV(in\_channels = 32,

out\_channels = 64,

kernel\_size = 5,

stride = 1,

padding = 2) #Your code here

self.fc1 = FC(in\_features = (64\*3\*3), out\_features = 64) #Your code here

self.fc2 = FC(in\_features = 64, out\_features = 10) #Your code here

def forward(self, x):

# Forward pass computation

# Conv 1

#Your code here

#print("dim of x = " + str(x.shape))

out = self.conv1(x)

out = F.relu(out)

#print("dim after CONV1 = " + str(out.shape))

# MaxPool

#Your code here

out = F.max\_pool2d(out, kernel\_size = 3, stride = 2)

#print("dim after MaxPool1 = " + str(out.shape))

# Conv 2

#Your code here

out = self.conv2(out)

out = F.relu(out)

#print("dim after CONV2 = " + str(out.shape))

# MaxPool

#Your code here

out = F.max\_pool2d(out, kernel\_size = 3, stride = 2)

#print("dim after MaxPool2 = " + str(out.shape))

# Conv 3

#Your code here

out = self.conv3(out)

out = F.relu(out)

#print("dim after CONV3 = " + str(out.shape))

# MaxPool

#Your code here

out = F.max\_pool2d(out, kernel\_size = 3, stride = 2)

#print("dim after MaxPool3 = " + str(out.shape))

# Flatten

#Your code here

out = out.view(out.size(0), -1)

#print("dim after Flatten = " + str(out.shape))

# FC 1

#Your code here

out = self.fc1(out)

out = F.relu(out)

#print("dim after FC1 = " + str(out.shape))

# FC 2

#Your code here

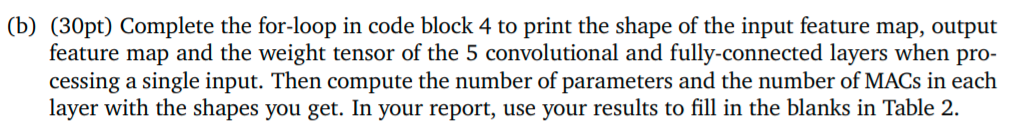
out = self.fc2(out)

out = F.relu(out)

#print("dim after FC2 = " + str(out.shape))

# return out

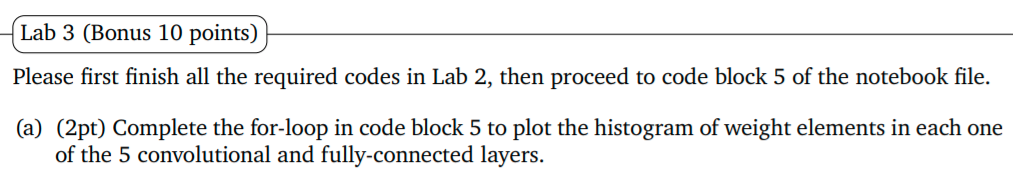
return out



Solution: Table 2 is completed below.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Layer | Input shape | Output shape | Weight shape | # Param | # MAC |
| Conv 1 | (1, 3, 32, 32) | (1, 32, 32, 32) | (32, 3, 5, 5) | 2400 | 2457600 |
| Conv 2 | (1, 32, 15, 15) | (1, 32, 15, 15) | (32, 32, 5, 5) | 25600 | 5760000 |
| Conv 3 | (1, 32, 7, 7) | (1, 64, 7, 7) | (64, 32, 5, 5) | 51200 | 2508800 |
| FC1 | (1, 576) | (1, 64) | (64, 576) | 36864 | 36864 |
| FC2 | (1, 64) | (1, 10) | (10, 64) | 640 | 640 |

Table 2: Results of Lab 2(b).



Solution: Histograms of weight elements in each one of the 5 convolutional and fully-connected layers are shown below.

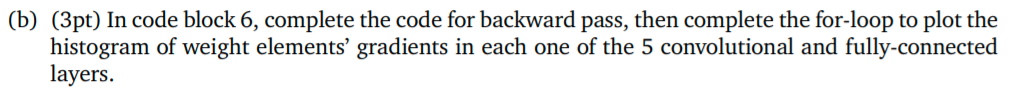




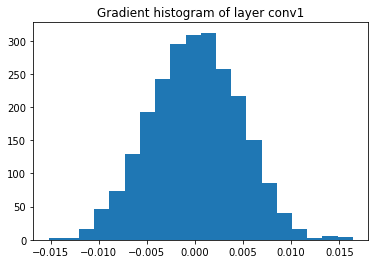


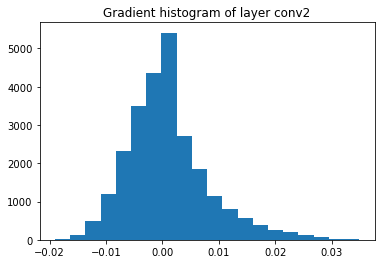


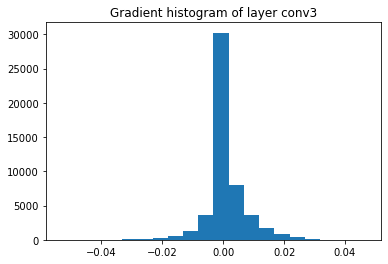


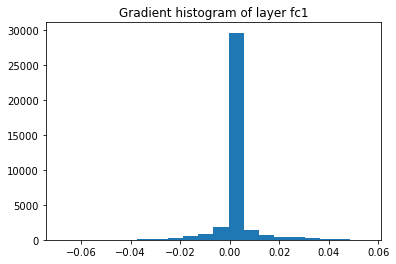


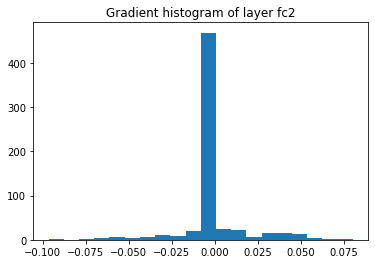
Solution: Histograms of weight elements’ gradients in each one of the 5 convolutional and fully-connected layers are shown below.

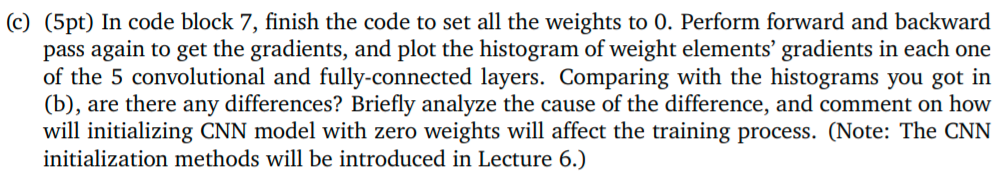




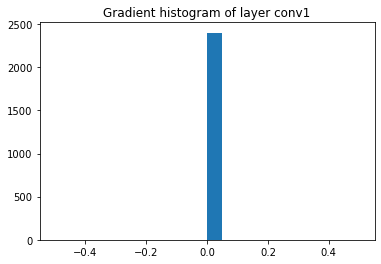


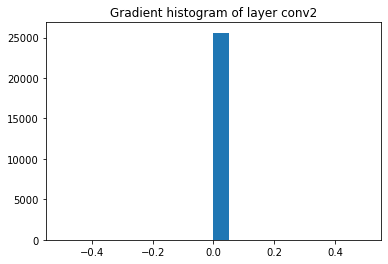


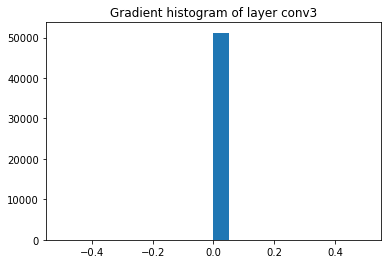


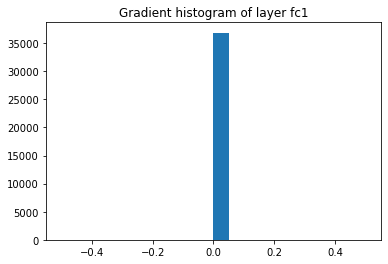


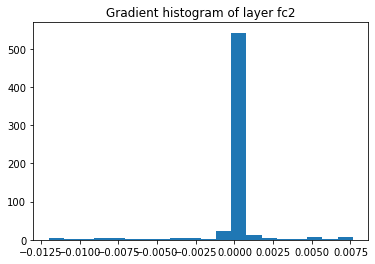
Solution: Histograms of weight elements’ gradients in each one of the 5 convolutional and fully-connected layers are shown below.











Analysis and Comment: Gradient Histograms in (b) generally tend to indicate the characteristic of Normal Distribution, whereas Gradient Histograms in (c) are monolithic, single and isolated.

The cause of difference is exactly weight zero initialization. For a CNN model, if all weights are initialized to be 0, the gradient descent of backpropagation will yield the same derivative for each element in weight matrixes. Therefore, all convolutional layers will extract the same features from training samples from every epoch of model training. Finally, the CNN model will either fail during training or have very bad quality and accuracy.