Assignment 1 - Answers

September 3, 2020

1 Assignment 1 - Linear Model, Back Propagation and Building a CNN

1.1 1. True/False Questions

For each question, please provide a short explanation to support your judgment.

1.1.1 Problem 1.1

Using a proper setting (e.g., learning rate, initialization etc.), the gradient descent algorithm is always able to yield an optimal solution on convex functions, as the local optimal value is always the global optimal value.

> True, for a convex function, the local minimum which gradient descent will lead you towards is always the global minimum making it the global optimal value for a convex function.

1.1.2 Problem 1.2

For a given task, having more layers in a neural network model always improves its capacity. Thus, it is beneficial to have as many layers as possible to improve NN's generalization ability.

> False, this usually depends on how large the dataset is and the complexity of the problem being solved. Additional layers do not always improve a model and can lead to overfitting. If the model is overfitting on the dataset, this makes the model less generalizable.

1.1.3 Problem 1.3

Given a learning task that can be perfectly learned by a Madaline model, the same set of weight values will be achieved after training, no matter how the Madaline is initialized.

> False, since Madeline uses the error-correction rule, Madeline is sensitive to initialization and as a result there is no theoretical guarantee of convergence. With random initialization, it is unknown how far the model is to the optimal objective, which means that deciding when to stop training is uncertain. This can lead to different weights if the model is stopped earlier or later in relation to the optimal objective than in previous run.

1.1.4 Problem 1.4

Sigmoid neurons are likely to 'die' during the training process, as the gradients in the middle of the 'S-shape' sigmoidal curve may vanish. Thus, these neurons are less likely to be updated. We should use a larger learning rate to train all of the sigmoid neurons.

> False. The sigmoid function squishes the input values between 0 and 1. Looking at the derivative of the sigmoid function we can see that extreme values will lead to a much smaller gradient. Regardless of how large the learning rate is, since it is multiplied by the gradient, the update to the weight will be small.

1.1.5 Problem 1.5

According to the "convolution shape rule," for a convolution operation with fixed kernel size, stride and padding, increasing the height and width of the input feature map will always lead to a larger output feature map size.**

> True, we can use the following formulas to show this:
>
> $$W_2 = \left[\frac{W_1 - K + 2P}{S}\right] + 1$$
>
> $$H_2 = \left[\frac{H_1 - K + 2P}{S}\right] + 1$$
>
> These two formulas describe the input feature map which has a shape of $H_1 \times W_1$ and the output feature map which has a shape $H_2 \times W_2$. An increase in H_1 and W_1 would lead to an increase in W_2 and H_2.

1.2 2. Adaline

In the following problems, you will be asked to derive the output of a given Adaline, or propose proper weight values for the Adaline to mimic the functionality of some simple logic functions. For all problems, please consider $+1$ as True and -1 as False in the inputs and outputs.

1.2.1 Problem 2.1

Observe the Adaline shown in Figure 1, fill in the feature s and output y for each pair of inputs given in the truth table. What logic function is this Adaline performing?

The logic function being performed is the AND function.

[1]:

[1]:

	x1	x2	s	y
0	-1	-1	-3.5	-1
1	-1	1	-1.5	-1
2	1	-1	-1.5	-1
3	1	1	0.5	1

1.2.2 Problem 2.2

Propose proper values for weight w0 , w1 and w2 in the Adaline shown in Figure 2 to perform the functionality of a logic NOR function. Fill in the feature s for each pair of inputs given in the truth table to prove the functionality is correct.

> Proper values for weights w0, w1, and w2 that would give the NOR function are -3, -3, -1, respectivly.

[2]:

[2]:

	x1	x2	s	y
0	-1	-1	1	1
1	-1	1	-1	-1
2	1	-1	-5	-1
3	1	1	-7	-1

1.2.3 Problem 2.3

Propose proper values for weight w0 , w1 , w2 and w3 in the Adaline shown in Figure 3 to perform the functionality of a Majority Vote function. Fill in the feature s for each triplet of inputs given in the truth table to prove the functionality is correct.

[3]:

[3]:

	x1	x2	s	y
0	-1	-1	-5	-1
1	-1	-1	-3	-1
2	-1	1	-1	-1
3	-1	1	1	1
4	1	-1	1	1
5	1	-1	3	1
6	1	1	5	1
7	1	1	7	1

1.2.4 Problem 2.4

As discussed in Lecture 2, the XOR function cannot be represented with a single Adaline, but can be represented with a 2-layer Madaline. Propose proper values for second-layer weight w20,w21 and w22 in the Madaline shown in Figure 4 to perform the functionality of a XOR function. Fill in the feature s for each pair of inputs given in the truth table to prove the functionality is correct.

[4]:

[4]:

	x1	x2	s	y
0	-1	-1	-1	-1
1	-1	1	1	1
2	1	-1	1	1
3	1	1	-1	-1

1.3 3. Back Propagation

1.3.1 Problem 3.1

The feed-forward computation of a logistic neuron could be represented as follows.

Given an input $x_l \epsilon R^{n\times 1}$, the output $x_{l+1} \epsilon R^{m\times 1}$ is calculated as follows:

$y_l = W_l x_l + b_l, x_{l+1} = \sigma(y_l)$

Where $W \epsilon R^{m\times n}$ denotes the weight matrix, $b \epsilon R^{m\times 1}$ denotes the bias term, and $\sigma : R^{m\times 1}$ $R^{m\times 1}$ denotes the Sigmoid activation function. Following the chain rule, derive the gradient $\frac{\partial x_{l+1}}{\partial x_l}, \frac{\partial x_{l+1}}{W_l}, \frac{\partial x_{l+1}}{\partial b_l}$ in a **vectorized format**. (Hint: make sure the gradient has the same shape as the parameter.)

$$\frac{\partial x_{l+1}}{\partial x_l} = W^T \cdot \frac{\partial x_{l+1}}{\partial y_l}$$

$$\boxed{= W^T \cdot x_{l+1} \odot (1 - x_{l+1})}$$

$$\frac{\partial x_{l+1}}{\partial W_l} = \frac{\partial x_{l+1}}{\partial y_l} \cdot x_l^T$$

$$\boxed{= x_{l+1} \odot (1 - x_{l+1}) \cdot x_l^T}$$

$$\frac{\partial x_{l+1}}{\partial b_l} = \frac{\partial x_{l+1}}{\partial y_l} \cdot \frac{\partial y_l}{\partial b_l}$$

$$\boxed{= x_{l+1} \odot (1 - x_{l+1})}$$

1.3.2 Problem 3.2

Consider a 2-layer fully-connected NN composed by the logistic neuron mentioned in Problem 3.1, where we have input $x_1 \epsilon R^{n\times 1}$, hidden feature $x_2 \epsilon R^{m\times 1}$, output $x_3 \epsilon R^{k\times 1}$ and weights and bias $W1, W2, b1, b2$ of the two layers. A MSE loss function $L = \frac{1}{2}(t-x_3)^T(t-x_3)$ is applied in the end. Following the chain rule, derive the gradient $\frac{\delta L}{\delta W_1}$ in a vectorized format.

$$\frac{\partial L}{\partial W_l} = \frac{\partial L}{\partial x_3} \cdot \frac{\partial x_3}{\partial x_2} \cdot \frac{\partial x_2}{\partial y_l} \cdot \frac{\partial y_l}{\partial W_l}$$

$$\frac{\partial L}{\partial x_3} = (x_3 - t)$$

$$\frac{\partial x_3}{\partial x_2} = W_2^T$$

$$\frac{\partial x_2}{\partial y_l} = x_2 \odot (1 - x_2)$$

$$\frac{\partial y_l}{\partial W_l} = x_1^T$$

$$\boxed{\frac{\partial L}{\partial W_l} = W_2^T \cdot (x_3 - t) \cdot (x_2 \odot (1 - x_2)) \cdot x_1^T}$$

1.4 4. 2D Convolution

1.4.1 Problem 4.1

Derive the 2D convolution results of the following 5×9 input matrix and the 3×3 kernel. Consider 0s are padded around the input and the stride is 1, so that the output should also have shape 5×9.

[4]:

```
Output feature map:
 [[ 0.    0.   -0.25 -0.5   1.75 -0.5  -0.25  0.    0.  ]
 [-0.25 -0.5   1.5   1.25  1.    1.25  1.5  -0.5  -0.25]
 [ 1.75  1.5   1.    1.    1.    1.    1.    1.5   1.75]
 [-0.25 -0.5   1.5   1.25  1.    1.25  1.5  -0.5  -0.25]
 [ 0.    0.   -0.25 -0.5   1.75 -0.5  -0.25  0.    0.  ]]
```

1.4.2 Problem 4.2

Compare the output matrix and the input matrix in Problem 4.1, briefly analyze the effect of this 3×3 kernel on the input.

> This type of kernel is used to sharpen images which it does by emphasizing the difference between adjacent numbers in the matrix.

[7]:

```
Original matrix:
[[0 0 0 0 1 0 0 0 0]
 [0 0 1 1 1 1 1 0 0]
 [1 1 1 1 1 1 1 1 1]
 [0 0 1 1 1 1 1 0 0]
 [0 0 0 0 1 0 0 0 0]]

 After the kernel is applied:
[[ 0.    0.   -0.25 -0.5   1.75 -0.5  -0.25  0.    0.  ]
 [-0.25 -0.5   1.5   1.25  1.    1.25  1.5  -0.5  -0.25]
 [ 1.75  1.5   1.    1.    1.    1.    1.    1.5   1.75]
 [-0.25 -0.5   1.5   1.25  1.    1.25  1.5  -0.5  -0.25]
 [ 0.    0.   -0.25 -0.5   1.75 -0.5  -0.25  0.    0.  ]]
```

1.5 5 Lab: LMS Algorithm

In this lab question, you will implement the LMS algorithm with NumPy to learn a linear regression model for the provided dataset. You will also be directed to analyze how the choice of learning rate in the LMS algorithm affect the final result. All the codes generating the results of this lab should be gathered in one file and submit to Sakai.

There are two variables in the file: data $X \in R^{100\times3}$ and target $D \in R^{100\times1}$. Each individual pair of data and target is composed into X and D following the same way as discussed on Lecture 2 Page 8. Specifically, each row in X correspond to the transpose of a data point, with the first element as constant 1 and the other two as the two input features x_{1k} and x_{2k}. The goal of the learning task is finding the weight vector $W \in R^{3\times1}$ for the linear model that can minimize the MSE loss, which is also formulated on Lecture 2 Page 8.

(a) Directly compute the least square (Wiener) solution with the provided dataset. What is the optimal weight W∗? What is the MSE loss of the whole dataset when the weight is set to W∗?

[7]:

```
The MSE loss of the whole dataset when the weight is set to W* is:
5.0399515658684104e-05
```

[11]:

```
Weight after 20 epochs:
[[-0.9993219 ]
 [ 1.00061145]
 [-2.00031968]]
```

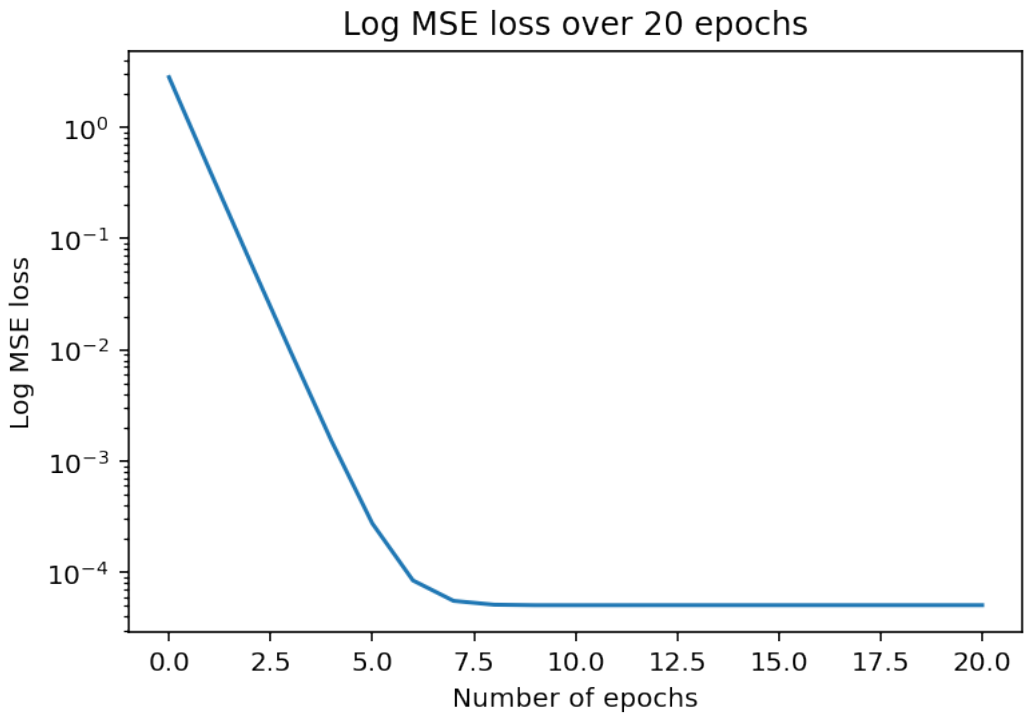
(b)

Now consider that you can only train with 1 pair of data point and target each time. In such case, the LMS algorithm should be used to find the optimal weight. Please initialize the weight vector as $W_0 = [0, 0, 0]^T$, and update the weight with the LMS algorithm. After each epoch (every time you go through all the training data and loop back to the beginning), compute and record the MSE loss of the current weight on the whole dataset. Run LMS for 20 epochs with learning rate r = 0.01, report the weight you get in the end and plot the MSE loss in log scale vs. Epochs

[8]:

```
 Weights after running for 20 epochs:
[[-0.99925145]
 [ 1.00082859]
 [-2.00068123]]
```

[13]:

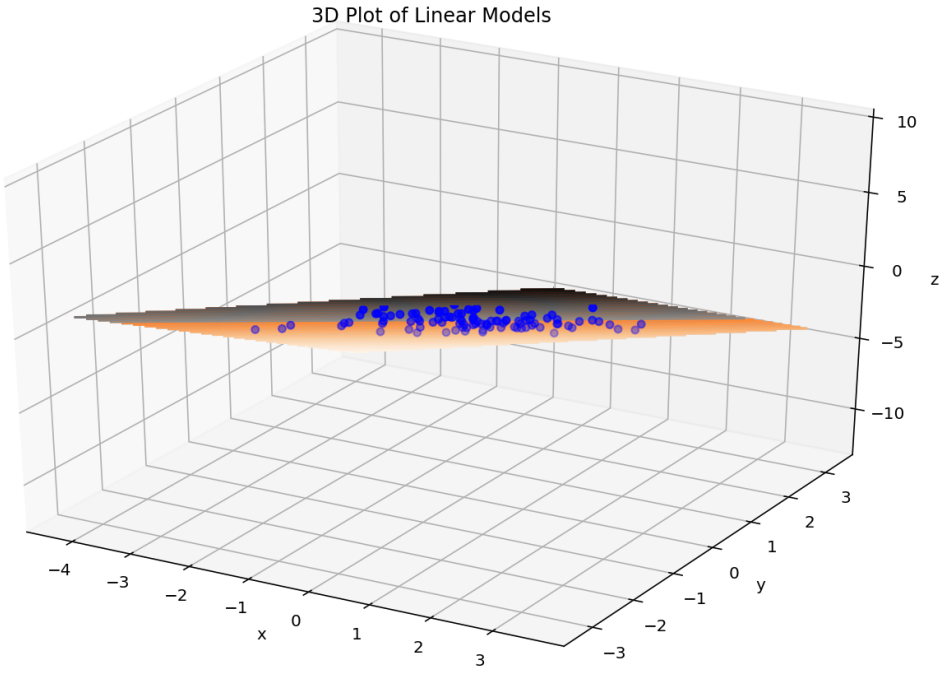


(c)

Scatter plot the points $(x_{1k},\ x_{2k},\ d_k)$ for all 100 data-target pairs in a 3D figure (https://jakevdp.github.io/PythonDataScienceHandbook/04.12-three-dimensional-plotting.html), and plot the lines corresponding to the linear models you got in (a) and (b) respectively in the same figure. Observe if the linear models fit the data well.

> Looking at the plot, we can see that the two planes do overlap but also fit the data well. Looking at the weights, for each of these separate runs, we see that they are extremly close but not exactly the same which is why we see the planes overlap.

[19]:

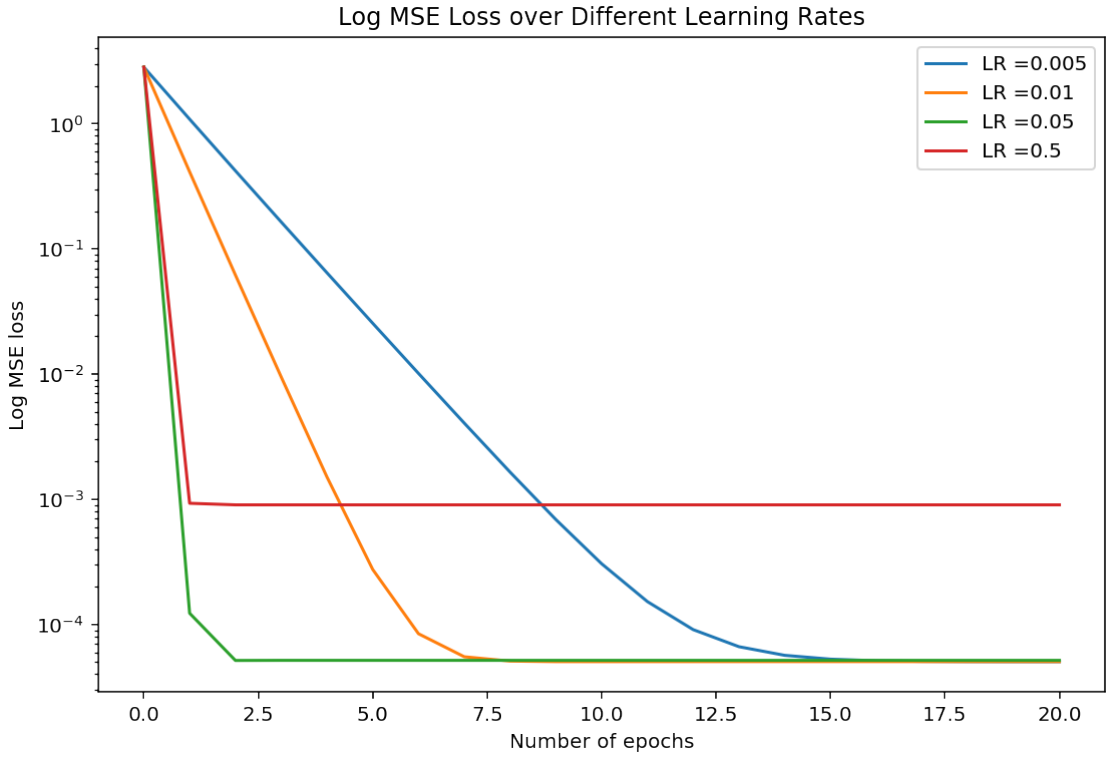


(d)

Learning rate r is an important hyperparameter for the LMS algorithm, as well as for CNN optimization. Here, try repeat the process in (b) with r set to 0.005, 0.05 and 0.5 respectively. Together with the result you got in (b), plot the MSE losses of the 4 sets of experiments in log scale vs. Epochs in one figure. Then try further enlarge the learning rate to r = 1 and observe how the MSE changes. Base on these observations, comment on how learning rate affects the speed and quality of the learning process. (Note: The learning rate tuning for the CNN optimization will be introduced in Lecture 7.)

> If we look at the plots of MSE loss for each over the learning rate over epochs, we can see that increasing the learning rate leads to finding the optimal parameters faster as the MSE loss drops in less epochs. This is only true up to a certain point as once a learning rate of 0.5 is used, causes the model to converge to a suboptimal solution. In the n-dimensional space where the gradient descent algorithm is working to find a local minimum, having a large learning rate causes gradient descent to bounce off valley walls where the optimal solution may lie. When we use a learning rate of 1, the weights explode and gradient descent will cause the loss to increase.

[15]:



[16]:

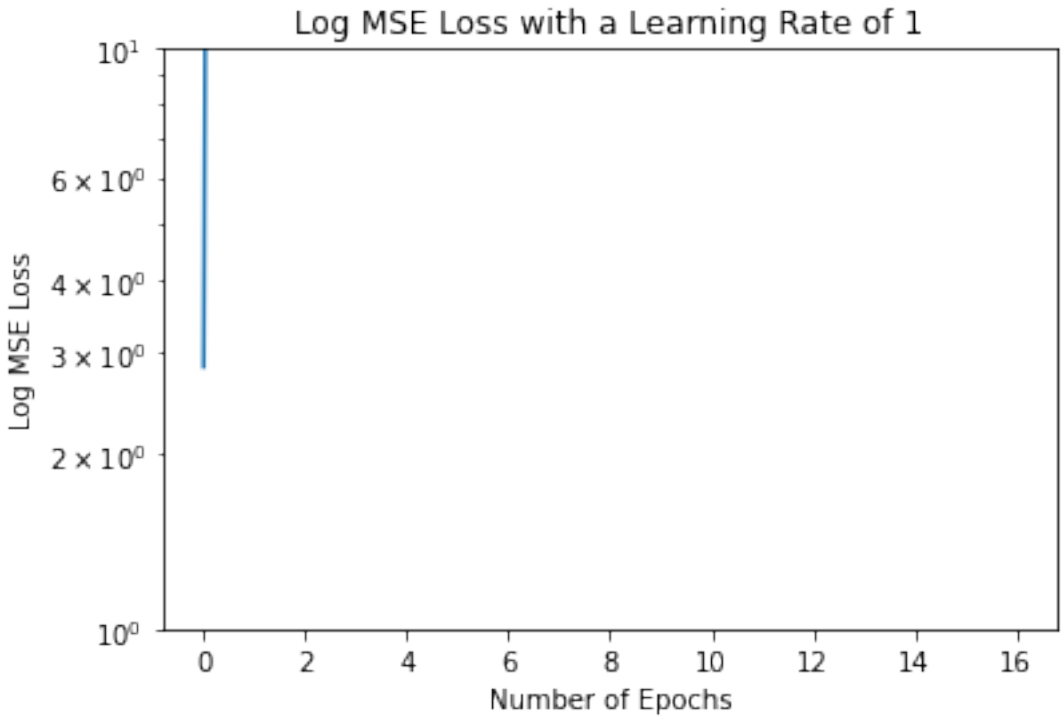
```
Weights for learning rate of 0.005
[[-0.99922727]
 [ 1.00059363]
 [-2.00040422]]

Weights for learning rate of 0.01
[[-0.99925145]
 [ 1.00082859]
 [-2.00068123]]

Weights for learning rate of 0.05
[[-0.99946913]
 [ 1.00163155]
 [-2.00162854]]

Weights for learning rate of 0.5
[[-1.02030504]
 [ 0.98520802]
 [-1.9666911 ]]
```

[13]:



[18]:

```
MSE for LR of 1:
```

[18]:
```
[2.8311433336832783,
 8.190171003880238e+19,
 5.315159923331934e+38,
 3.4493685919950074e+57,
 2.2385297629920696e+76,
 1.452734135583647e+95,
 9.427779355808586e+113,
 6.118326912315607e+132,
 3.9705982494066945e+151,
 2.576791120209086e+170,
 1.6722549248543786e+189,
 1.085239898479929e+208,
 7.042859433380322e+226,
 4.570590250858831e+245,
 2.9661667166370503e+264,
 1.9249472186294156e+283,
 1.2492291056081223e+302,
 inf,
```

```
  inf,
  inf,
  inf]
```

1.6 6 LeNet-5

(a)

Complete code block 3 for defining the adapted LeNet-5model. Note that customized CONV and FC classes are provided in code block 2 to replace the nn.Conv2d and nn.Linear classes in PyTorch respectively. The usage of the customized classes are exactly the same as their PyTorch counterparts, the only difference is that in the customized class the input and output feature maps of the layer will be stored in self.input and self.output respectively after the forward pass, which will be helpful in question (b). After the code is completed, run through the block and make sure the model forward pass in the end throw no errors. Please copy your code of the completed LeNet5 class into the report PDF.

The final code for LeNet-5 is shown below.

[10]:
```
"""
Lab 2(a)
Build the LeNet-5 model by following Table 1
"""

# Create the neural network module: LeNet-5
class LeNet5(nn.Module):
    def __init__(self):
        super(LeNet5, self).__init__()
        # Layer definition
        self.conv1 =     CONV(in_channels=3, out_channels=6, kernel_size=5,␣
↪stride=1)
        self.conv2 =     CONV(in_channels=6, out_channels=16, kernel_size=5,␣
↪stride=1)
        self.fc1   =     FC(in_features = 400, out_features = 120)
        self.fc2   =     FC(120, 84)
        self.fc3   =     FC(84, 10)
        self.maxpool = nn.MaxPool2d(kernel_size=2)
        self.relu = nn.ReLU()

    def forward(self, x):
        # Forward pass computation
        # Conv 1
        x = self.conv1(x)
        x = self.relu(x)
        # MaxPool
        x = self.maxpool(x)
        # Conv 2
        x = self.conv2(x)
```

```
        x = self.relu(x)
        # MaxPool
        x = self.maxpool(x)
        # Flatten
        x = x.view(x.size(0), -1)
        # FC 1
        x = self.fc1(x)
        x = self.relu(x)
        # FC 2
        x = self.fc2(x)
        x = self.relu(x)
        # FC 3
        out = self.fc3(x)
        return out

# GPU check
device = 'cuda' if torch.cuda.is_available() else 'cpu'
if device =='cuda':
    print("Run on GPU...")
else:
    print("Run on CPU...")

# Model Definition
net = LeNet5()
net = net.to(device)

# Test forward pass
data = torch.randn(5,3,32,32)
data = data.to(device)
# Forward pass "data" through "net" to get output "out"
out = net(data)    #Your code here

# Check output shape
assert(out.detach().cpu().numpy().shape == (5,10))
print("Forward pass successful")
```

```
Run on GPU...
Forward pass successful
```

(b)

Complete the for-loop in code block 4 to print the shape of the input feature map, output feature map and the weight tensor of the 5 convolutional and fully-connected layers when processing a single input. Then compute the number of parameters and the number of MACs in each layer with the shapes you get. In your report, use your results to fill in the blanks in Table 2.

Layer	Input Shape	Output Shape	Weight Shape	# of Parameters	# of MAC
conv1	(1, 3, 32, 32)	(1, 6, 28, 28)	(6, 3, 5, 5)	450	352800

Layer	Input Shape	Output Shape	Weight Shape	# of Parameters	# of MAC
conv2	(1, 6, 14, 14)	(1, 16, 10, 10)	(16, 6, 5, 5)	2400	240000
fc1	(1, 400)	(1, 120)	(120, 400)	48000	48000
fc2	(1, 120)	(1, 84)	(84, 120)	10080	10080
fc3	(1, 84)	(1, 10)	(10, 84)	840	840

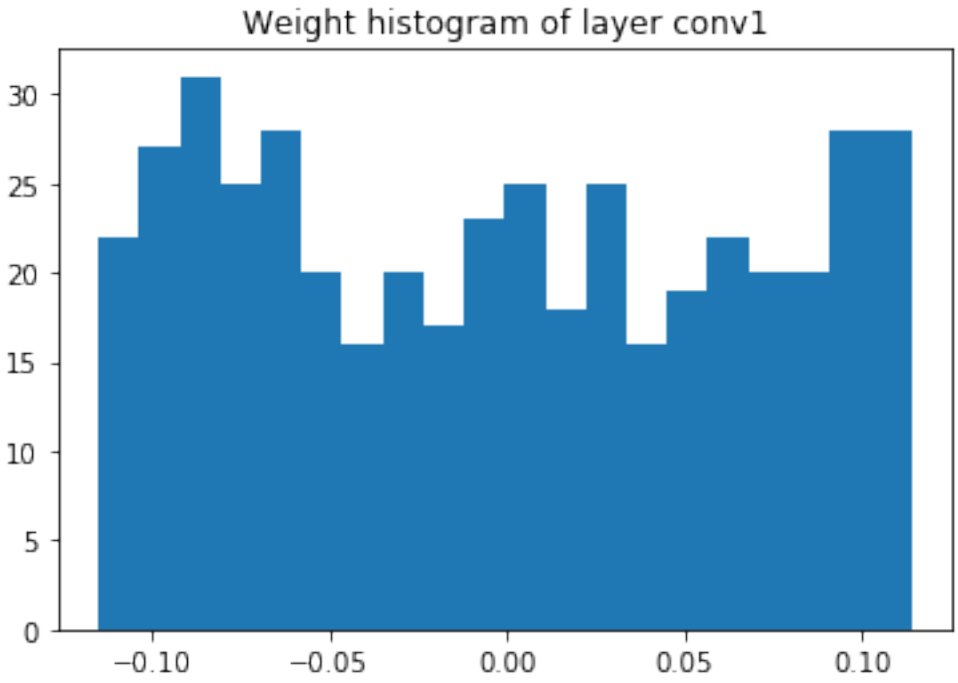
1.7 Lab 3 (Bonus)

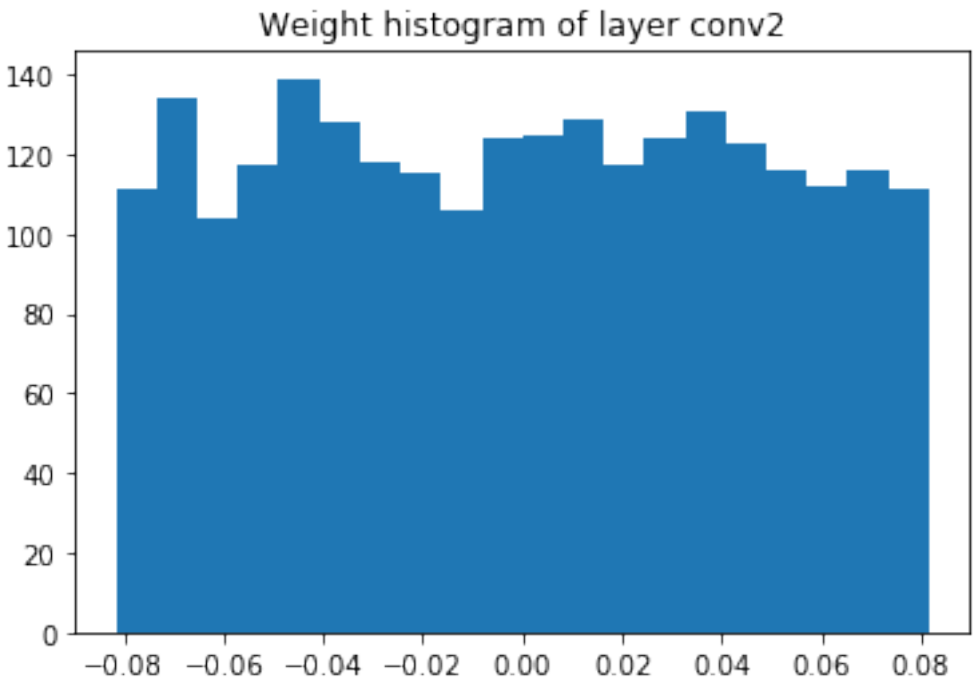
(a)

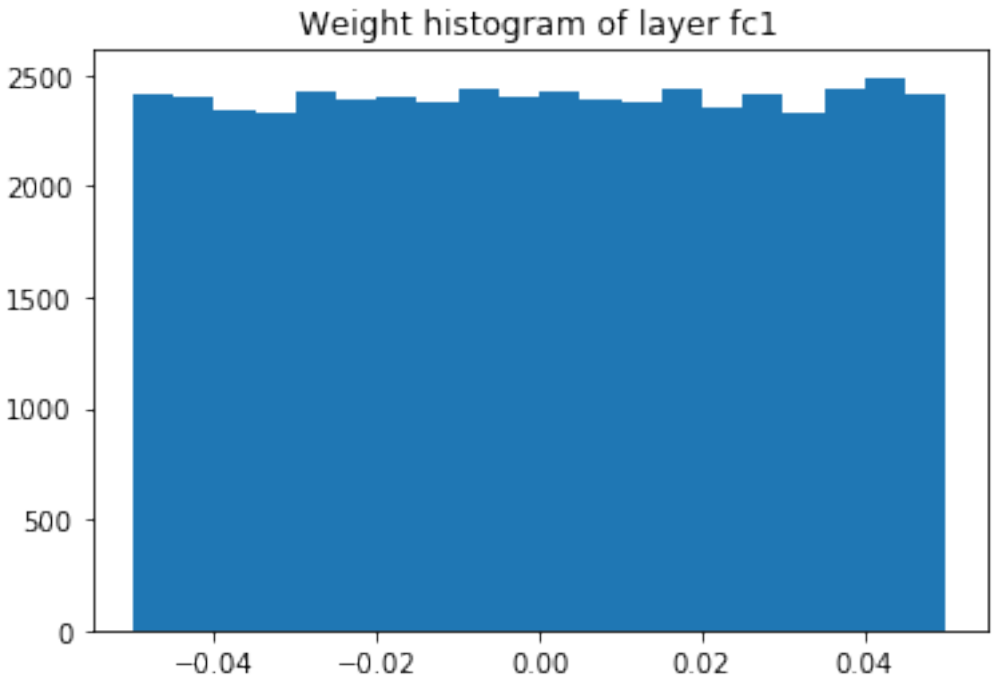
Complete the for-loop in code block 5 to plot the histogram of weight elements in each one of the 5 convolutional and fully-connected layers.

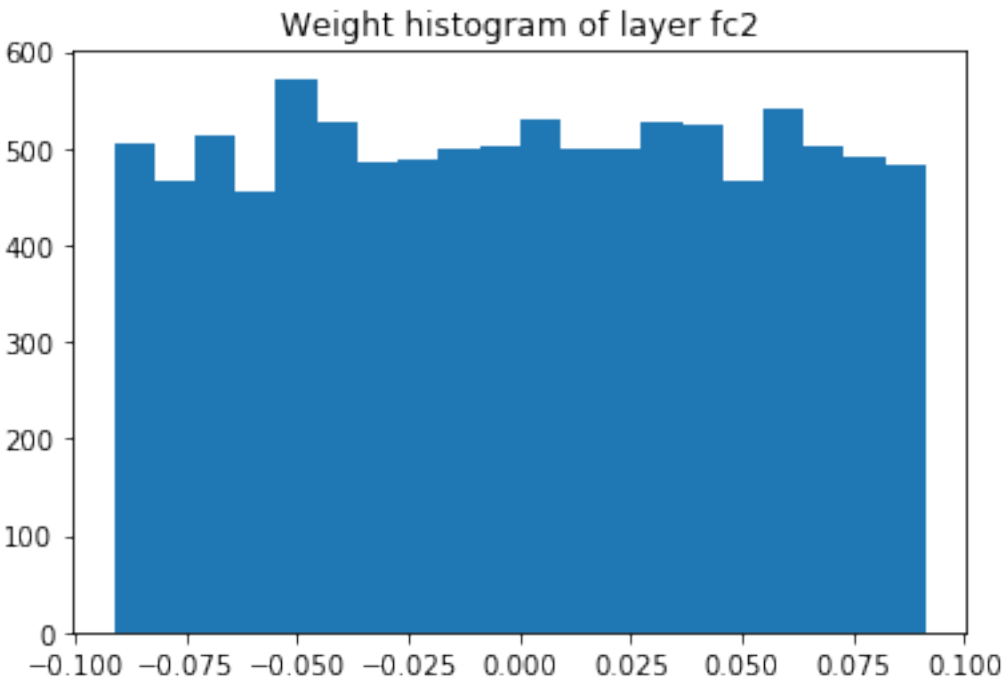
The initial weight histograms are shown below.

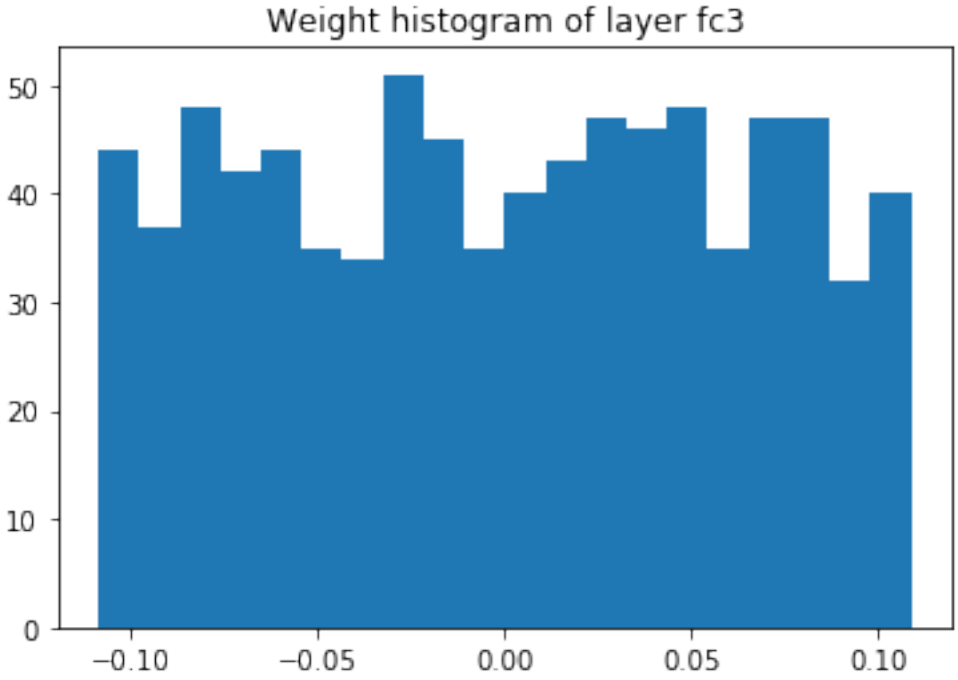
[11]:




Weight histogram of layer conv2
140
120
100
80
60
40
20
0
−0.08
−0.06
−0.04
−0.02
0.00
0.02
0.04
0.06
0.08


Weight histogram of layer fc1
2500
2000
1500
1000
500
0
−0.04
−0.02
0.00
0.02
0.04





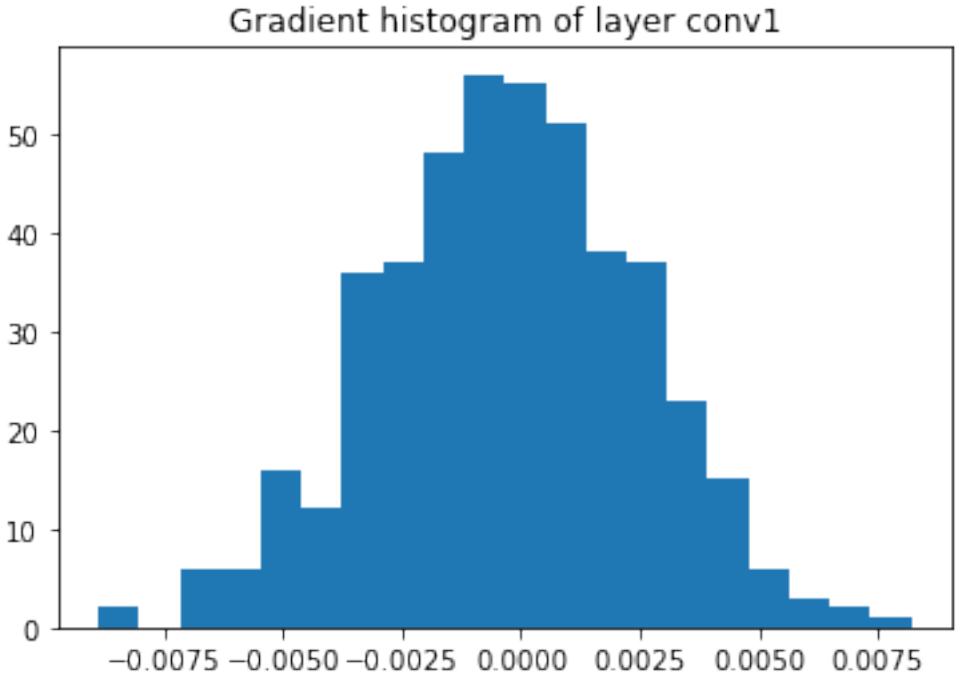
(b)

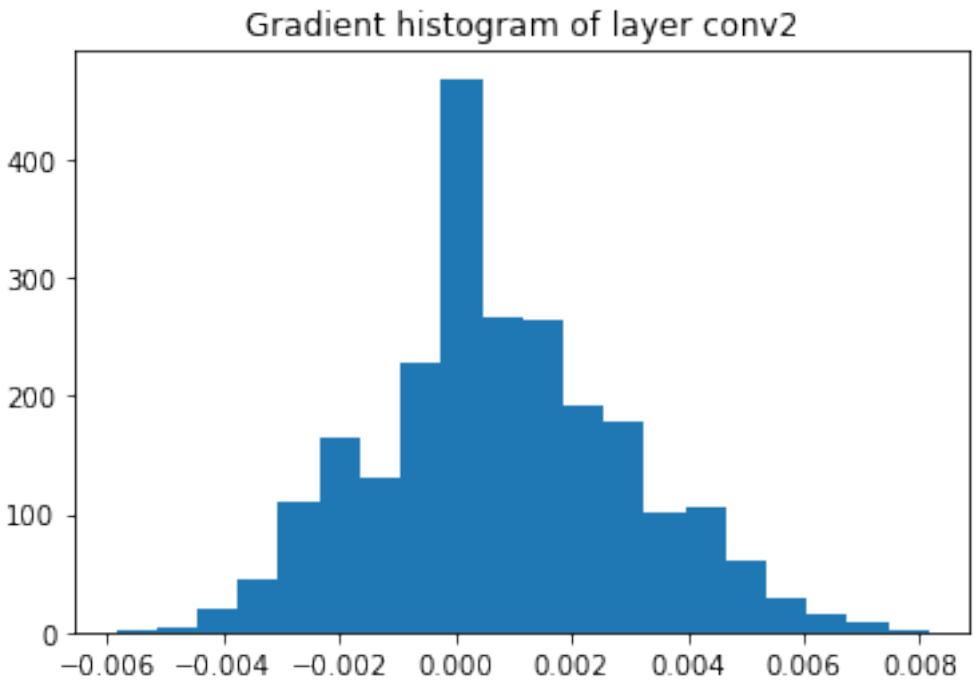
In code block 6, complete the code for backward pass, then complete the for-loop to plot the

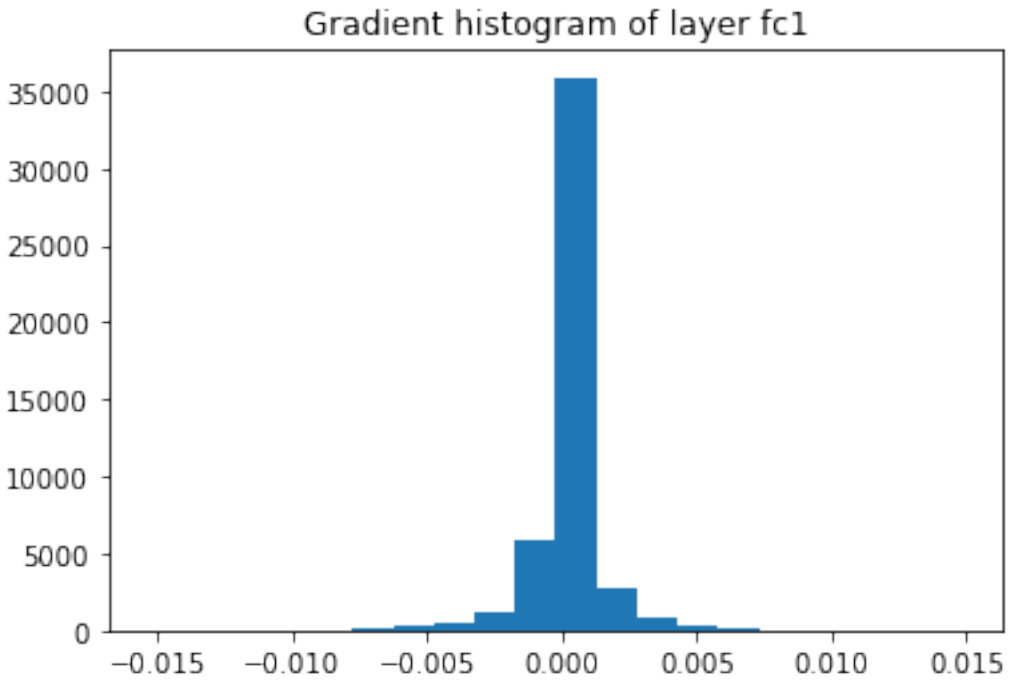
histogram of weight elements' gradients in each one of the 5 convolutional and fully-connected layers.

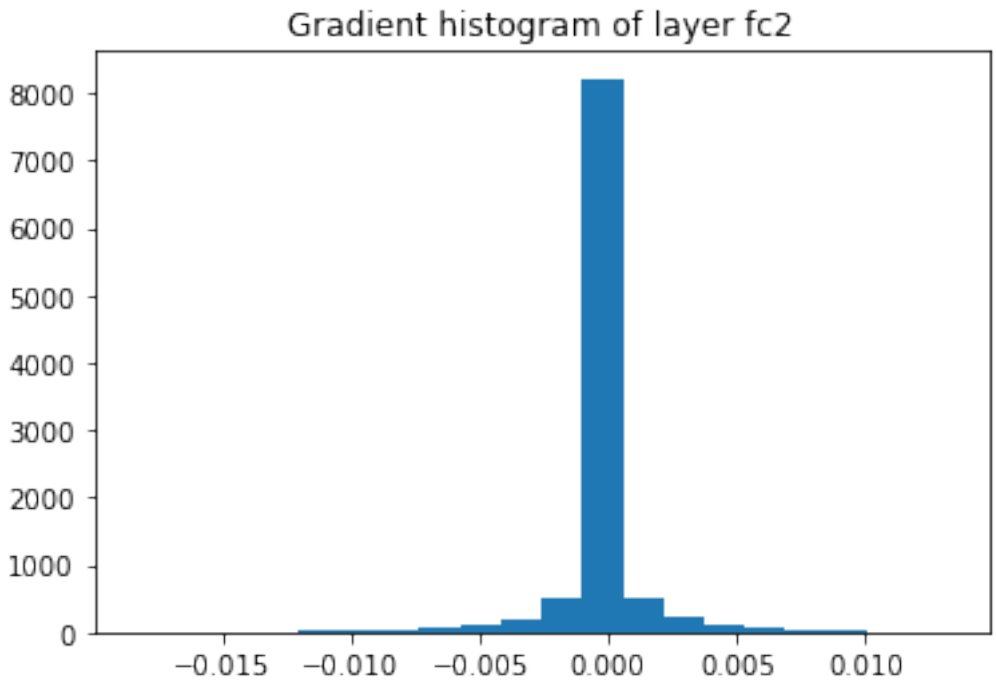
The gradient histograms are shown below.

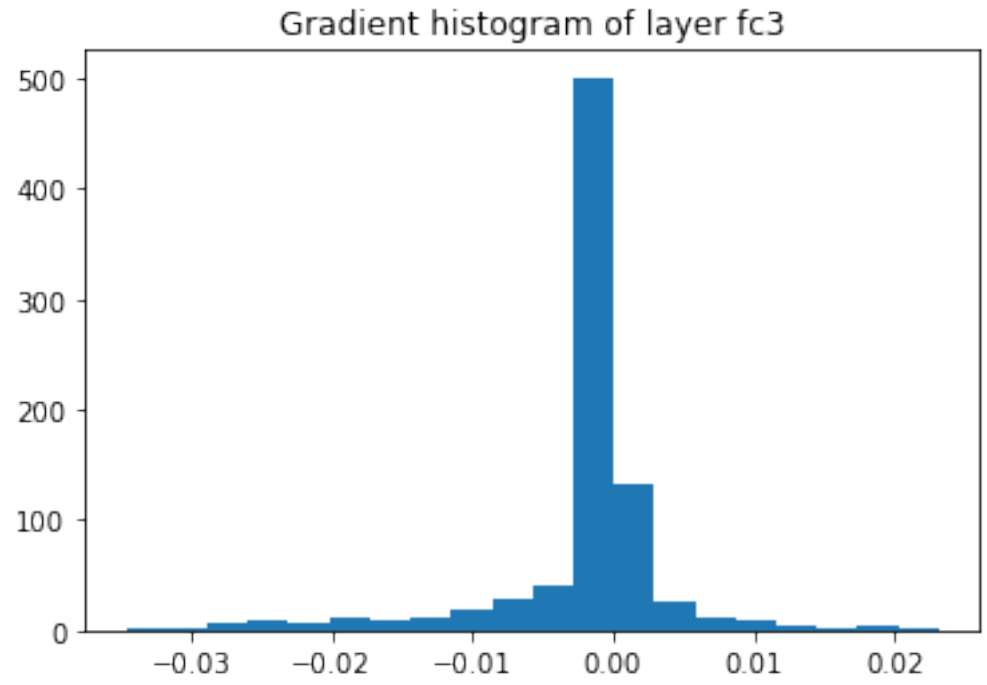
[12]:




Gradient histogram of layer conv2
400
300
200
100
0
−0.006
−0.004
−0.002
0.000
0.002
0.004
0.006
0.008


Gradient histogram of layer fc1
35000
30000
25000
20000
15000
10000
5000
0
−0.015
−0.010
−0.005
0.000
0.005
0.010
0.015





(c)

(In code block 7, finish the code to set all the weights to 0. Perform forward and backward pass

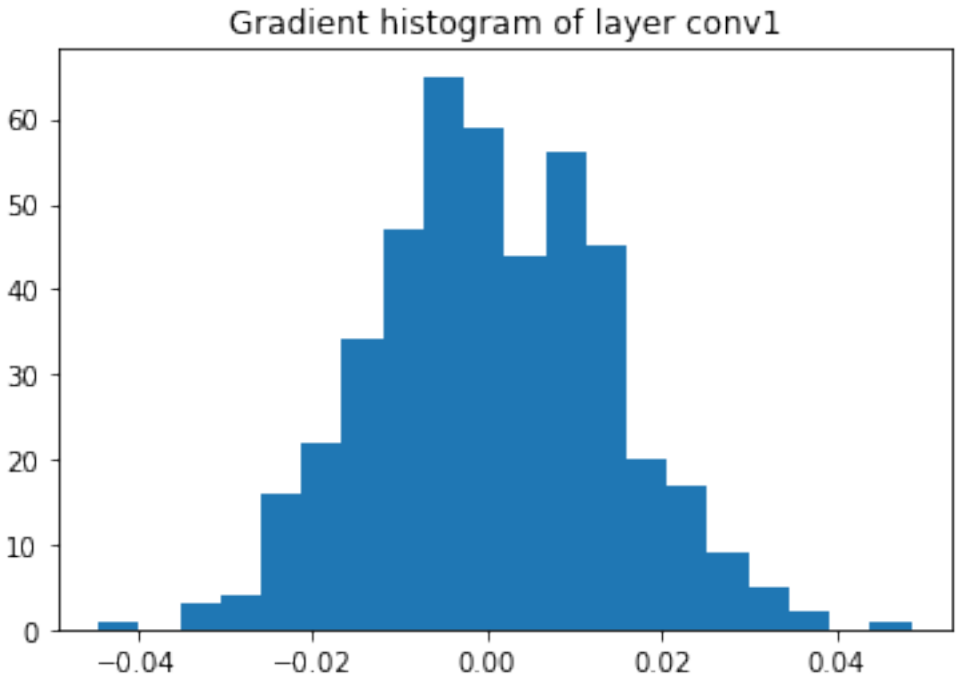
again to get the gradients, and plot the histogram of weight elements' gradients in each one of the 5 convolutional and fully- connected layers. Comparing with the histograms you got in (b), are there any differences? Briefly analyze the cause of the difference, and comment on how will initializing CNN model with zero weights will affect the training process. (Note: The CNN initialization methods will be introduced in Lecture 6.)

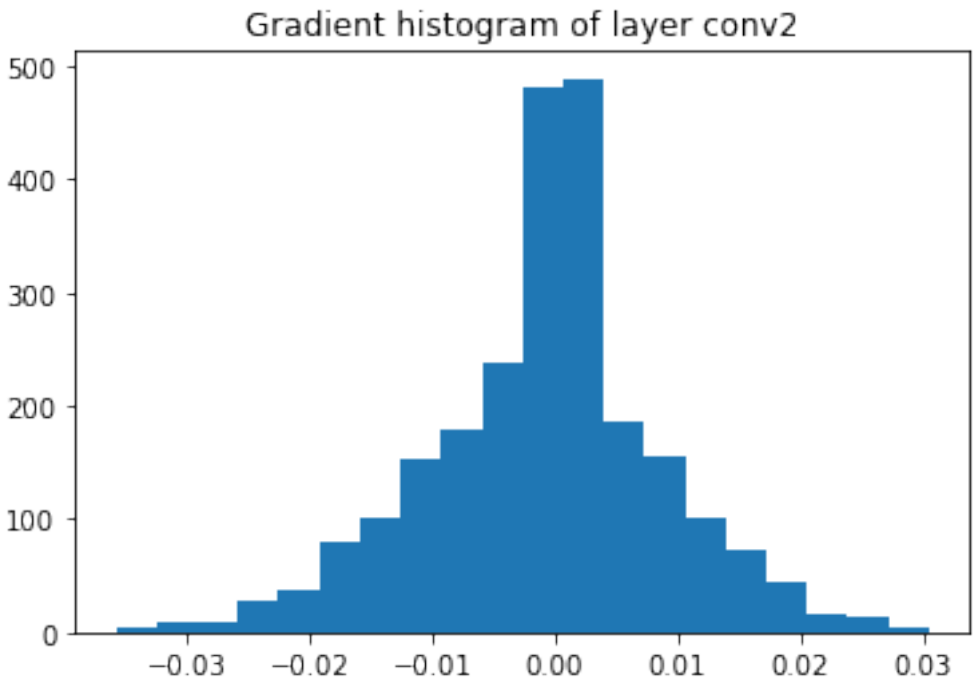
The resuling gradient histograms are shown in the figures below. Compared to those in part b), it is clear that initializing all the weights at 0 is problematic for the training process. Initializing the neurons' weights to 0 makes them useless because they can never be updated, making training inconsequential. The gradients also generally remain nearly identical, which we can see by the lack of distribution of gradient values across all the layers compared to those in part b).

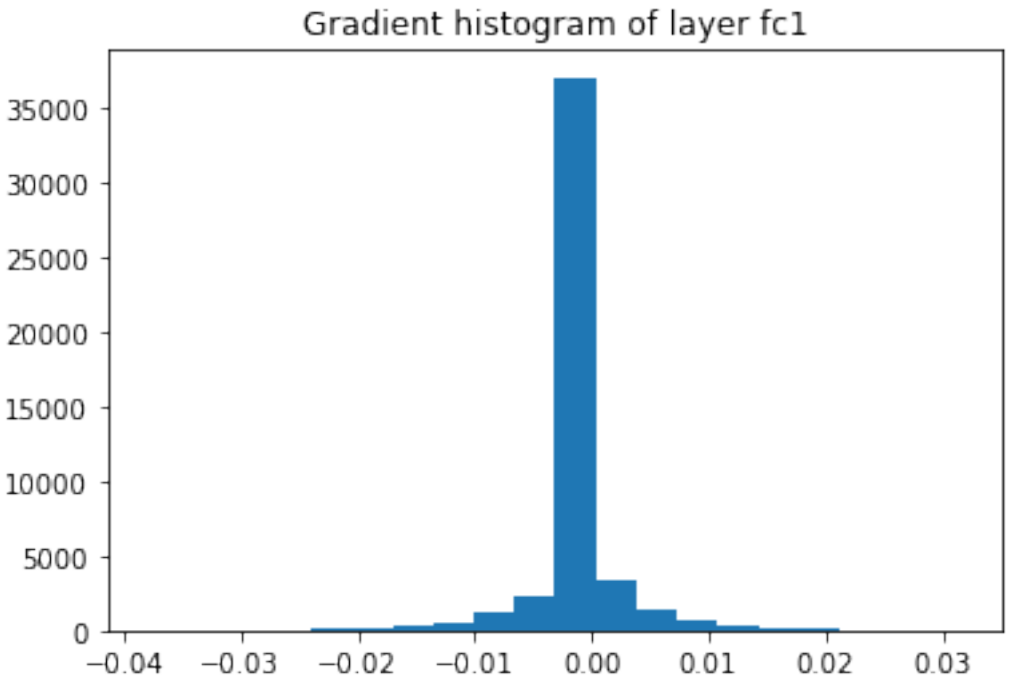
Comparing with the histograms you got in (b), are there any differences? Briefly analyze the cause of the difference, and comment on how will initializing CNN model with zero weights will affect the training process.

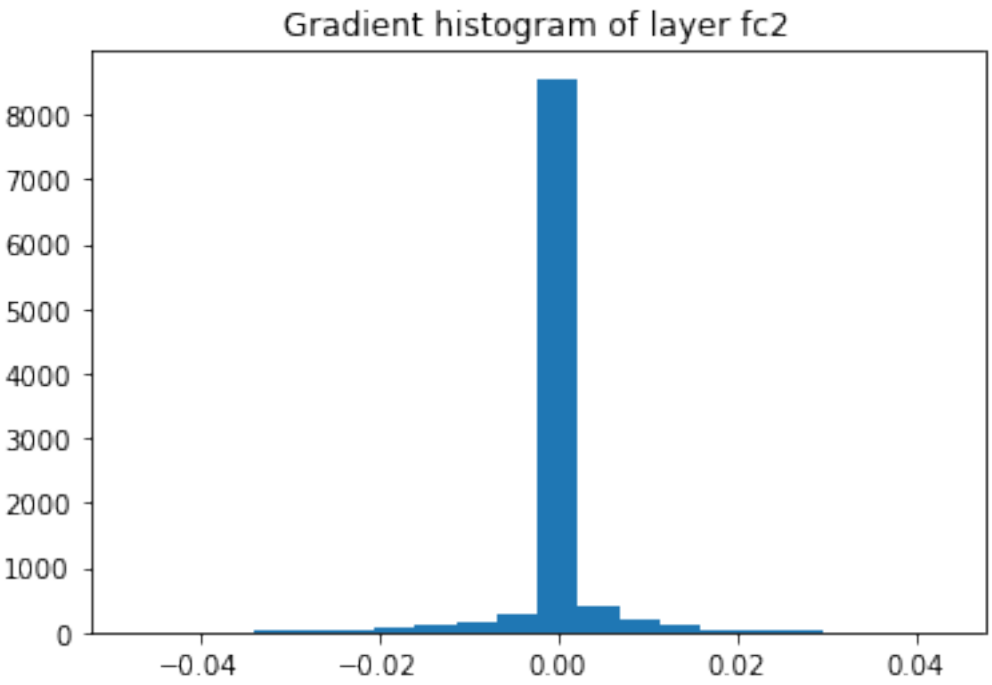
> When the weights were set to zero, the histograms show that the gradient for all layers except the last are 0. This is because when a model is initialized with all weights at zero, this effectivly kills the neurons. In this situation all the neurons are following the same gradient and therefore will all be updated by the same amount making training useless.

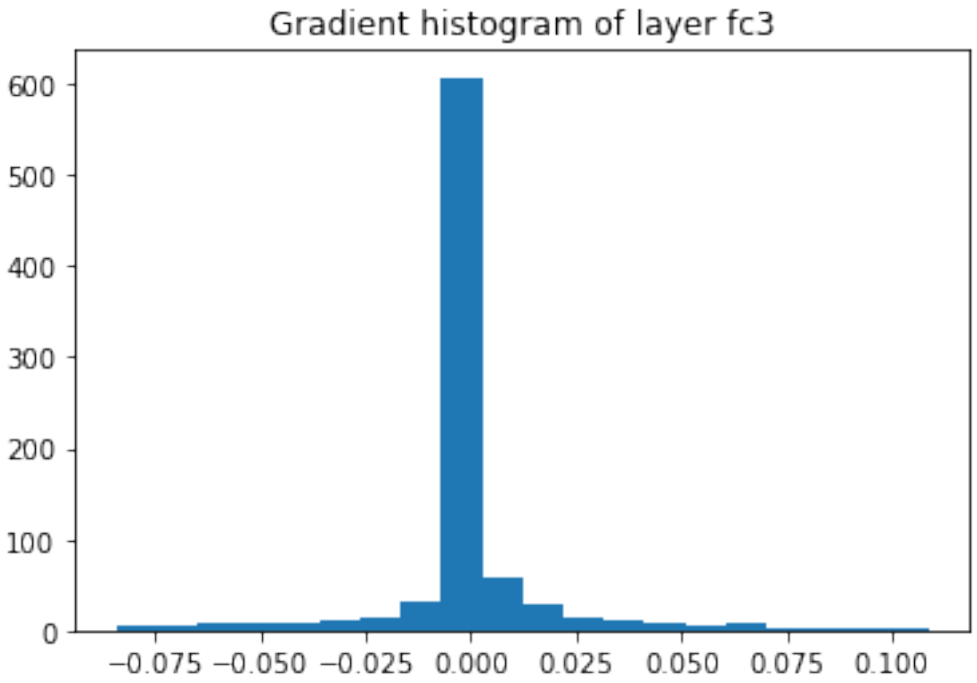
[13]:




Gradient histogram of layer conv2
500
400
300
200
100
0
−0.03
−0.02
−0.01
0.00
0.01
0.02
0.03


Gradient histogram of layer fc1
35000
30000
25000
20000
15000
10000
5000
0
−0.04
−0.03
−0.02
−0.01
0.00
0.01
0.02
0.03


Gradient histogram of layer fc2
8000
7000
6000
5000
4000
3000
2000
1000
0
−0.04
−0.02
0.00
0.02
0.04


Gradient histogram of layer fc3
600
500
400
300
200
100
0
−0.075
−0.050
−0.025
0.000
0.025
0.050
0.075
0.100