ECE 661: Homework #1

Linear Model, Back Propagation and Building a CNN

Name: `Suim Park(NetID:sp699)`

1 True/False Questions (10 pts)

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

Problem 1.1 (2 pts) On image recognition tasks, the convolution layers, compared to fully-connected layers, usually lead to better performance by exploiting shift invariant images features and having more parameters.

> **False**; Convolutional layers lead to better performance in image recognition tasks by efficiently capturing local features and being shift-invariant, all while using fewer parameters than fully-connected layers due to parameter sharing.

Problem 1.2 (2 pts) According to the "convolution shape rule," for a convolution operation with a fixed input feature map, increasing the height and width of kernel size can not lead to the output feature maps in same size.

> **False**; According to the "convolution shape rule," the size of the output feature map depends on several factors, including the input size, kernel size, stride, and padding. If you increase the height and width of the kernel size while keeping the input size, stride, and padding constant, the output feature map will shrink because the convolution operation reduces the spatial dimensions.

Problem 1.3 (2 pts) The overfitting models can perfectly fit the training data. Theoretically, we should increase slightly the noise in the training data or prune some of the nodes to improve NN's generalization ability.

> **True**; When a model is overfitted, it fits the training data perfectly but performs poorly on new data due to a lack of generalization. To improve generalization, slightly increasing noise in the training data can help the model focus on broader patterns rather than memorizing specific details. Pruning, or removing unnecessary neurons or connections in the neural network, is another effective method to reduce model complexity and prevent it from learning irrelevant patterns, thus improving its performance on unseen data. Both approaches enhance the model's ability to generalize.

Problem 1.4 (2 pts) The latency of a neural network measured on a specific processor is not always positively related to its theoretical FLOPS.

> **True**; The latency of a neural network is not always positively related to its theoretical FLOPS. While FLOPS measures a processor's computational capacity, factors like memory bandwidth, data access patterns, parallelism, processor architecture, and communication overheads can affect the actual latency. Even if a processor has high

> FLOPS, inefficiencies in these areas can cause higher latency, meaning the two are not directly correlated.

Problem 1.5 (2 pts) Given a learning task that can be perfectly learned by a Madaline model, this model is suitable for different weight initializations.

> **False**; The Madaline model has complexity due to its structure and training method, which can make it difficult to scale to large models. In addition, the Madaline model is sensitive to the random selection of training patterns and the initial weight values, making it challenging to determine when to stop training. These factors can lead to inconsistent results depending on the initialization, which is why it is not always suitable for different weight initializations.

2 Adalines (15 pts)

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. **The answer of the proposed weight values should be within this set {-1, 0, 1}.**

Problem 2.1 (3 pts) 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?

x_1	x_2	*s*	*y*
-1	-1	2x1 + 2x(-1) + 1x(-1) = **-1**	**-1**
-1	+1	2x1 + 2x(-1) + 1x1 = **1**	**+1**
+1	-1	2x1 + 2x1 + 1x(-1) = **3**	**+1**
+1	+1	2x1 + 2x1 + 1x1 = **5**	**+1**

> This Adaline is performing the **OR** function. It outputs **-1** only when both input values are **-1**. However, if one or both of the inputs is **+1**, the output value is **+1**, which aligns with the behavior of the OR function.

Problem 2.2 (4 pts) Propose proper values for weight w_0, w_1 and w_2 in the Adaline shown in Figure 2 to perform the functionality of a logic **NAND** function. Fill in the feature *s* for each pair of inputs given in the truth table to prove the functionality is correct. [**Hint**: The truth table of NAND function can be found here. https://en.wikipedia.org/wiki/NAND_logic]

x_1	x_2	*s*	*y*
-1	-1		+1
-1	+1		+1
+1	-1		+1
+1	+1		-1

1. $w_0 - w_1 - w_2 > 0$
2. $w_0 - w_1 + w_2 > 0$
3. $w_0 + w_1 - w_2 > 0$
4. $w_0 + w_1 + w_2 < 0$

Based on the 4 inequalities, $w_0 > w_1$, $w_0 > w_2$ and $w_0 > 0$. Therefore, $w_0 = 1$ and $w_1 = 0$ or -1 and $w_2 = 0$ or -1. After applying various values to find the appropriate y for each input, the **resulting weights** are $w_0 = 1$, $w_1 = -1$, and $w_2 = -1$.

x_1	x_2	s	y
-1	-1	1x1 + (-1)x(-1) + (-1)x(-1) = **3**	+1
-1	+1	1x1 + (-1)x(-1) + (-1)x1 = **1**	+1
+1	-1	1x1 + (-1)x1 + (-1)x(-1) = **1**	+1
+1	+1	1x1 + (-1)x1 + (-1)x1 = **-1**	-1

Problem 2.3 (4 pts) Propose proper values for weight w_0, w_1 and w_2 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. [**Hint**: The truth table of Majority Vote function can be found here. https://en.wikichip.org/wiki/boolean_algebra/majority_function]

x_1	x_2	x_3	s	y
-1	-1	-1		-1
-1	-1	+1		-1
-1	+1	-1		-1
-1	+1	+1		+1
+1	-1	-1		-1
+1	-1	+1		+1
+1	+1	-1		+1
+1	+1	+1		+1

1. $w_0 - w_1 - w_2 - w_3 < 0$
2. $w_0 - w_1 - w_2 + w_3 < 0$
3. $w_0 - w_1 + w_2 - w_3 < 0$
4. $w_0 - w_1 + w_2 + w_3 > 0$
5. $w_0 + w_1 - w_2 - w_3 < 0$
6. $w_0 + w_1 - w_2 + w_3 > 0$
7. $w_0 + w_1 + w_2 - w_3 > 0$
8. $w_0 + w_1 + w_2 + w_3 > 0$

Based on the 8 inequalities, they can be organized as follows:

1. $w_0 < w_1, w_0 < w_2, w_0 < w_1 + w_2$
2. $w_0 > -w2, w_0 > -w_3, w_0 > -(w_2 + w_3)$

> Using the organized inequalities above, the following **result** was obtained: $w_0 = 0$, $w_1 = 1$, $w_2 = 1$, $w_3 = 1$.

x_1	x_2	x_3	s	y
-1	-1	-1	0x1 + 1x(-1) + 1x(-1) + 1x(-1) = **-3**	-1
-1	-1	+1	0x1 + 1x(-1) + 1x(-1) + 1x1 = **-1**	-1
-1	+1	-1	0x1 + 1x(-1) + 1x1 + 1x(-1) = **-1**	-1
-1	+1	+1	0x1 + 1x(-1) + 1x1 + 1x1 = **1**	+1
+1	-1	-1	0x1 + 1x1 + 1x(-1) + 1x(-1) = **-1**	-1
+1	-1	+1	0x1 + 1x1 + 1x(-1) + 1x1 = **1**	+1
+1	+1	-1	0x1 + 1x1 + 1x1 + 1x(-1) = **1**	+1
+1	+1	+1	0x1 + 1x1 + 1x1 + 1x1 = **3**	+1

Problem 2.4 (4 pts) 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 w_{20}, w_{21} and w_{22} 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.

x_1	x_2	s	y
-1	-1		**-1**
-1	+1		**+1**
+1	-1		**+1**
+1	+1		**-1**

> The following table shows the first operation involving x_1 and x_2 in this context:

x_1	x_2	s_1	y_1
-1	-1	(0.5)x1 + (-1)x(-1) + 1x(-1) = 0.5	**+1**
-1	+1	(0.5)x1 + (-1)x(-1) + 1x1 = 2.5	**+1**
+1	-1	(0.5)x1 + (-1)x1 + 1x(-1) = -1.5	**-1**
+1	+1	(0.5)x1 + (-1)x1 + 1x1 = 0.5	**+1**

> The following table shows the second operation involving x_1 and x_2 in this context:

x_1	x_2	s_2	y_2
-1	-1	(0.5)x1 + 1x(-1) + (-1)x(-1) = 0.5	**+1**
-1	+1	(0.5)x1 + 1x(-1) + (-1)x1 = -1.5	**-1**
+1	-1	(0.5)x1 + 1x1 + (-1)x(-1) = 2.5	**+1**

x_1	x_2	s_2	y_2
+1	+1	(0.5)x1 + 1x1 + (-1)x1 = 0.5	**+1**

> The final operation involving two operations' results can be shown in this table:

x_1	x_2	x_{21}	x_{22}	s	y
-1	-1	+1	+1		-1
-1	+1	+1	-1		+1
+1	-1	-1	+1		+1
+1	+1	+1	+1		-1

> Based on the table, they can be organized as follows:
>
> 1. w_{20} + w_{21} + w_{22} < 0
> 2. w_{20} + w_{21} - w_{22} > 0
> 3. w_{20} - w_{21} + w_{22} > 0
> 4. w_{20} + w_{21} + w_{22} < 0

> The inequalities above can be summarized as follows: w_{20} < -w_{21} - w_{22} and w_{20} > 0

> Therefore, the values of each weight are as follows: w_{20} **= 1**, w_{21} **= -1** and w_{22} **= -1**

x_1	x_2	x_{21}	x_{22}	s	y
-1	-1	+1	+1	1x1 + (-1)x1 + (-1)x1 = **-1**	-1
-1	+1	+1	-1	1x1 + (-1)x1 + (-1)x(-1) = **+1**	+1
+1	-1	-1	+1	1x1 + (-1)x(-1) + (-1)x1 = **+1**	+1
+1	+1	+1	+1	1x1 + (-1)x1 + (-1)x1 = **-1**	-1

3 Back Propagation (15 pts)

Problem 3.1 (10 pts) Consider a 2-layer fully-connected NN, where we have input $x_1 \in \mathbb{R}^{n\times 1}$, hidden feature $x_2 \in \mathbb{R}^{m\times 1}$, output $x_3 \in \mathbb{R}^{k\times 1}$ and weights and bias $W_1 \in \mathbb{R}^{m\times n}$, $W_2 \in \mathbb{R}^{k\times m}$, $b_1 \in \mathbb{R}^{m\times 1}$, $b_2 \in \mathbb{R}^{k\times 1}$ of the two layers. The hidden features and outputs are computed as follows:

$$x_2 = \text{Sigmoid}(W_1 x_1 + b_1) \quad (1) \tag{1}$$

$$x_3 = W_2 x_2 + b_2 \quad (2) \tag{2}$$

A MSE loss function $L = \frac{1}{2}(t - x_3)^T(t - x_3)$ is applied in the end, where $t \in \mathbb{R}^{k\times 1}$ is the target value. Following the chain rule, derive the gradient $\frac{\partial L}{\partial W_1}$, $\frac{\partial L}{\partial W_2}$, $\frac{\partial L}{\partial b_1}$, $\frac{\partial L}{\partial b_2}$ in a **vectorized format**.

> To get a value $\frac{\partial L}{\partial W_1}$, this chain rule will be used: $\frac{\partial L}{\partial W_1}$ = $\frac{\partial L}{\partial x_3} \cdot \frac{\partial x_3}{\partial x_2} \cdot \frac{\partial x_2}{\partial W_1}$

- $\frac{\partial L}{\partial x_3} = -(t - x_3) \quad \in \mathbb{R}^{k \times 1}$
- $\frac{\partial x_3}{\partial x_2} = W_2 = W_2^T \quad \in \mathbb{R}^{m \times k}$
- $\frac{\partial x_2}{\partial W_1} = (x_2 \odot (1 - x_2))x_1 = (x_2 \odot (1 - x_2))x_1^T \quad \in \mathbb{R}^{1 \times n}$ ($\odot$ is element-wise multiplication; *scalar value*) Therefore, $\frac{\partial L}{\partial W_1}$ is

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial x_3} \cdot \frac{\partial x_3}{\partial x_2} \cdot \frac{\partial x_2}{\partial W_1} = \frac{\partial x_3}{\partial x_2} \cdot \frac{\partial L}{\partial x_3} \cdot \frac{\partial x_2}{\partial W_1}$$

$$= W_2^T \cdot [-(t - x_3)] \cdot [(x_2 \odot (1 - x_2))x_1^T]$$

$$= \left[W_2^T(-(t - x_3)) \odot (x_2 \odot (1 - x_2))\right] x_1^T \quad \in \mathbb{R}^{m \times n}$$

To get a value $\frac{\partial L}{\partial W_2}$, this chain rule will be used: $\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial x_3} \cdot \frac{\partial x_3}{\partial W_2}$

- $\frac{\partial L}{\partial x_3} = -(t - x_3) \quad \in \mathbb{R}^{k \times 1}$
- $\frac{\partial x_3}{\partial W_2} = x_2 = x_2^T \quad \in \mathbb{R}^{1 \times m}$ Therefore, $\frac{\partial L}{\partial W_2}$ is

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial x_3} \cdot \frac{\partial x_3}{\partial W_2}$$

$$= -(t - x_3) \cdot x_2^T \quad \in \mathbb{R}^{k \times m}$$

To get a value $\frac{\partial L}{\partial b_1}$, this chain rule will be used: $\frac{\partial L}{\partial b_1}$ = $\frac{\partial L}{\partial x_3} \cdot \frac{\partial x_3}{\partial x_2} \cdot \frac{\partial x_2}{\partial b_1}$

- $\frac{\partial L}{\partial x_3} = -(t - x_3) \quad \in \mathbb{R}^{k \times 1}$
- $\frac{\partial x_3}{\partial x_2} = W_2 = W_2^T \quad \in \mathbb{R}^{m \times k}$
- $\frac{\partial x_2}{\partial b_1} = (x_2 \odot (1 - x_2))$ ($\odot$ is element-wise multiplication; *scalar value*) Therefore, $\frac{\partial L}{\partial b_1}$ is

$$\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial x_3} \cdot \frac{\partial x_3}{\partial x_2} \cdot \frac{\partial x_2}{\partial b_1} = \frac{\partial x_3}{\partial x_2} \cdot \frac{\partial L}{\partial x_3} \cdot \frac{\partial x_2}{\partial b_1}$$

$$= W_2^T \cdot [-(t - x_3)] \cdot (x_2 \odot (1 - x_2))$$

$$= \left[W_2^T(-(t - x_3)) \odot (x_2 \odot (1 - x_2))\right] \quad \in \mathbb{R}^{m \times 1}$$

To get a value $\frac{\partial L}{\partial b_2}$, this chain rule will be used: $\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial x_3} \cdot \frac{\partial x_3}{\partial b_2}$

- $\frac{\partial L}{\partial x_3} = -(t - x_3) \quad \in \mathbb{R}^{k \times 1}$
- $\frac{\partial x_3}{\partial b_2} = 1 \quad \in \mathbb{I}$ Therefore, $\frac{\partial L}{\partial b_2}$ is

$$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial x_3} \cdot \frac{\partial x_3}{\partial b_2}$$

$$= -(t - x_3) \cdot \mathbb{I} \quad \in \mathbb{R}^{k \times 1}$$

Problem 3.2 (5 pts) Replace the Sigmoid function with ReLU function. Given a data $x_1 = [1, 1, -1]^T$, target value $t = [1, 2]^T$, weights and bias at this iteration are

$$W_1 = \begin{bmatrix} 0 & -1 & 1 \\ 2 & 2 & -1 \end{bmatrix}, b_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \quad (3)$$

$$W_2 = \begin{bmatrix} 0 & 2 \\ 1 & 1 \end{bmatrix}, b_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad (4)$$

Following the results in Problem 3.1, calculate the values of: L, $\frac{\partial L}{\partial W_1}$, $\frac{\partial L}{\partial W_2}$, $\frac{\partial L}{\partial b_1}$, $\frac{\partial L}{\partial b_2}$

Using its value, x_2 and x_3 is this:

$$x_2 = \mathrm{ReLU}(W_1 x_1 + b_1) = \mathrm{ReLU}(\begin{bmatrix} 0 & -1 & 1 \\ 2 & 2 & -1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \end{bmatrix})$$

$$= \mathrm{ReLU}(\begin{bmatrix} -2 \\ 5 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \end{bmatrix}) = \mathrm{ReLU}(\begin{bmatrix} -1 \\ 7 \end{bmatrix}) = \begin{bmatrix} 0 \\ 7 \end{bmatrix}$$

$$x_3 = W_2 x_2 + b_2 = \begin{bmatrix} 0 & 2 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} 14 \\ 7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 15 \\ 8 \end{bmatrix}$$

> To get a value L, this equation is used: $L = \frac{1}{2}(t - x_3)^T(t - x_3)$
>
> - $t - x_3 = \begin{bmatrix} 1 \\ 2 \end{bmatrix} - \begin{bmatrix} 15 \\ 8 \end{bmatrix} = \begin{bmatrix} -14 \\ -6 \end{bmatrix}$ Therefore Loss value is:
>
> $$L = \frac{1}{2}[\,-14 \quad -6\,] \begin{bmatrix} -14 \\ -6 \end{bmatrix}$$
>
> $$= \frac{1}{2}(196 + 36) = \frac{1}{2} \times 232 = 116$$

> To get a value $\frac{\partial L}{\partial W_1}$, this chain rule will be used: $\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial x_3} \cdot \frac{\partial x_3}{\partial x_2} \cdot \frac{\partial x_2}{\partial W_1}$
>
> - $\frac{\partial L}{\partial x_3} = -(t - x_3) = -(\begin{bmatrix} 1 \\ 2 \end{bmatrix} - \begin{bmatrix} 15 \\ 8 \end{bmatrix}) = \begin{bmatrix} 14 \\ 6 \end{bmatrix}$
> - $\frac{\partial x_3}{\partial x_2} = W_2 = W_2^T = \begin{bmatrix} 0 & 1 \\ 2 & 1 \end{bmatrix}$
> - $\mathrm{ReLU}\prime(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases} \; (z = W_1 x_1 + b_1)$
> - $\frac{\partial x_2}{\partial W_1} = \frac{\partial x_2}{\partial (W_1 x_1 + b_1)} \cdot \frac{\partial (W_1 x_1 + b_1)}{\partial W_1} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} x_1^T$ In this chain rule, $\frac{\partial L}{\partial W_1}$ is this:
>
> $$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial x_3} \cdot \frac{\partial x_3}{\partial x_2} \cdot \frac{\partial x_2}{\partial W_1} = \left(W_2^T \frac{\partial L}{\partial x_3} \odot \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right) x_1^T$$
>
> $$= [(\begin{bmatrix} 0 & 1 \\ 2 & 1 \end{bmatrix} \cdot \begin{bmatrix} 14 \\ 6 \end{bmatrix}) \odot \begin{bmatrix} 0 \\ 1 \end{bmatrix}] \cdot [\,1 \quad 1 \quad -1\,]$$
>
> $$= (\begin{bmatrix} 6 \\ 34 \end{bmatrix} \odot \begin{bmatrix} 0 \\ 1 \end{bmatrix}) \cdot [\,1 \quad 1 \quad -1\,]$$

$$= \begin{bmatrix} 0 \\ 34 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & -1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 34 & 34 & -34 \end{bmatrix}$$

To get a value $\frac{\partial L}{\partial W_2}$, this chain rule will be used: $\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial x_3} \cdot \frac{\partial x_3}{\partial W_2}$

- $\frac{\partial L}{\partial x_3} = -(t - x_3) = -(\begin{bmatrix} 1 \\ 2 \end{bmatrix} - \begin{bmatrix} 15 \\ 8 \end{bmatrix}) = \begin{bmatrix} 14 \\ 6 \end{bmatrix}$
- $\frac{\partial x_3}{\partial W_2} = x_2 = x_2^T = \begin{bmatrix} 0 & 7 \end{bmatrix}$ In this chain rule, $\frac{\partial L}{\partial W_2}$ is this:

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial x_3} \cdot \frac{\partial x_3}{\partial W_2} = \frac{\partial x_3}{\partial W_2} \cdot \frac{\partial L}{\partial x_3}$$

$$= \begin{bmatrix} 14 \\ 6 \end{bmatrix} \cdot \begin{bmatrix} 0 & 7 \end{bmatrix} = \begin{bmatrix} 0 & 98 \\ 0 & 42 \end{bmatrix}$$

To get a value $\frac{\partial L}{\partial b_1}$, this chain rule will be used: $\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial x_3} \cdot \frac{\partial x_3}{\partial x_2} \cdot \frac{\partial x_2}{\partial b_1}$

- $\frac{\partial L}{\partial x_3} = -(t - x_3) = -(\begin{bmatrix} 1 \\ 2 \end{bmatrix} - \begin{bmatrix} 15 \\ 8 \end{bmatrix}) = \begin{bmatrix} 14 \\ 6 \end{bmatrix}$
- $\frac{\partial x_3}{\partial x_2} = W_2 = W_2^T = \begin{bmatrix} 0 & 1 \\ 2 & 1 \end{bmatrix}$
- $\mathrm{ReLU}\prime(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases} \ (z = W_1 x_1 + b_1)$
- $\frac{\partial x_2}{\partial b_1} = \frac{\partial x_2}{\partial (W_1 x_1 + b_1)} \cdot \frac{\partial (W_1 x_1 + b_1)}{\partial b_1} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ In this chain rule, $\frac{\partial L}{\partial W_1}$ is this:

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial x_3} \cdot \frac{\partial x_3}{\partial x_2} \cdot \frac{\partial x_2}{\partial W_1}$$

$$= [W_2^T \frac{\partial L}{\partial x_3}] \odot \begin{bmatrix} 0 \\ 1 \end{bmatrix} = (\begin{bmatrix} 0 & 1 \\ 2 & 1 \end{bmatrix} \cdot \begin{bmatrix} 14 \\ 6 \end{bmatrix}) \odot \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} 6 \\ 34 \end{bmatrix} \odot \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 34 \end{bmatrix}$$

To get a value $\frac{\partial L}{\partial b_2}$, this chain rule will be used: $\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial x_3} \cdot \frac{\partial x_3}{\partial b_2}$

- $\frac{\partial L}{\partial x_3} = -(t - x_3) = -(\begin{bmatrix} 1 \\ 2 \end{bmatrix} - \begin{bmatrix} 15 \\ 8 \end{bmatrix}) = \begin{bmatrix} 14 \\ 6 \end{bmatrix}$
- $\frac{\partial x_3}{\partial b_2} = \mathbb{I}_{\mathbb{k}}$ In this chain rule, $\frac{\partial L}{\partial W_2}$ is this:

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial x_3} \cdot \frac{\partial x_3}{\partial b_2}$$

$$= \begin{bmatrix} 14 \\ 6 \end{bmatrix} \cdot \mathbb{I}_{\mathbb{k}} = \begin{bmatrix} 14 \\ 6 \end{bmatrix}$$

$$\therefore L = 166, \frac{\partial L}{\partial W_2} = \begin{bmatrix} 0 & 0 & 0 \\ 34 & 34 & -34 \end{bmatrix}$$

$$\frac{\partial L}{\partial W_2} = \begin{bmatrix} 0 & 98 \\ 0 & 42 \end{bmatrix}$$

$$\frac{\partial L}{\partial W_1} = \begin{bmatrix} 0 \\ 34 \end{bmatrix}$$

$$\frac{\partial L}{\partial b_2} = \begin{bmatrix} 14 \\ 6 \end{bmatrix}$$

4 2D Convolution (10 pts)

Problem 4.1 (5 pts) 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.

$$\begin{bmatrix} 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & -1 & -1 & -1 & 0 & 1 & 1 & 1 & 0 \\ -1 & -1 & -1 & -1 & 0 & 1 & 1 & 1 & 1 \\ 0 & -1 & -1 & -1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 0 & -1/2 & 0 \\ -1/2 & 1 & -1/2 \\ 0 & -1/2 & 0 \end{bmatrix}$$

In [ ]:
```python
import pandas as pd
import numpy as np
from scipy.signal import convolve2d

# input matrix
input_matrix = np.array(
    [
        [0, 0, -1, 0, 0, 0, 1, 0, 0],
        [0, -1, -1, -1, 0, 1, 1, 1, 0],
        [-1, -1, -1, -1, 0, 1, 1, 1, 1],
        [0, -1, -1, -1, 0, 1, 1, 1, 0],
        [0, 0, -1, 0, 0, 0, 1, 0, 0],
    ]
)

# kernel 3x3
kernel = np.array([[0, -1 / 2, 0], [-1 / 2, 1, -1 / 2], [0, -1 / 2, 0]])

# padding
padding = 1
padded_input_matrix = np.pad(input_matrix, padding, mode="constant", constant_values=0)

# convolution
output_matrix = convolve2d(padded_input_matrix, kernel, mode="valid")

print("Padded Input Matrix:")
print(padded_input_matrix)
print("\nConvolution Result (5x9):")
print(output_matrix)
```

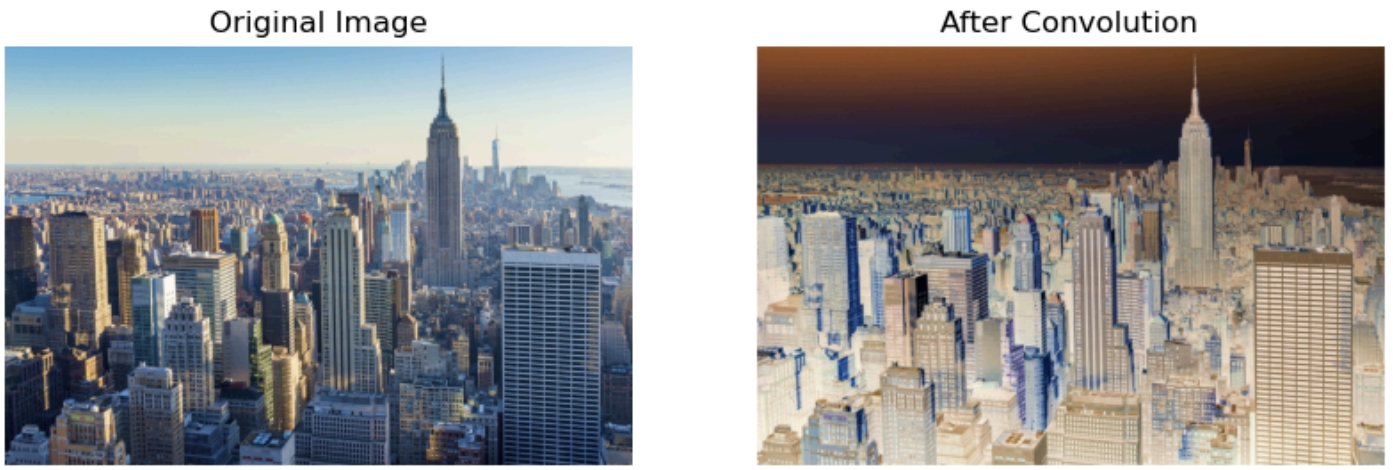
```
Padded Input Matrix:
[[ 0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0 -1  0  0  0  1  0  0  0]
 [ 0  0 -1 -1 -1  0  1  1  1  0  0]
 [ 0 -1 -1 -1 -1  0  1  1  1  1  0]
 [ 0  0 -1 -1 -1  0  1  1  1  0  0]
 [ 0  0  0 -1  0  0  0  1  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0]]

Convolution Result (5x9):
[[ 0.   1.  -0.5  1.   0.  -1.   0.5 -1.   0. ]
 [ 1.   0.   1.   0.   0.   0.  -1.   0.  -1. ]
 [-0.5  1.   1.   0.5  0.  -0.5 -1.  -1.   0.5]
 [ 1.   0.   1.   0.   0.   0.  -1.   0.  -1. ]
 [ 0.   1.  -0.5  1.   0.  -1.   0.5 -1.   0. ]]
```

$$\therefore Output = \begin{bmatrix} 0 & 1 & -0.5 & 1 & 0 & -1 & 0.5 & -1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & -1 \\ -0.5 & 1 & 1 & 0.5 & 0 & -0.5 & -1 & -1 & 0.5 \\ 1 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & -1 \\ 0 & 1 & -0.5 & 1 & 0 & -1 & 0.5 & -1 & 0 \end{bmatrix}$$

Problem 4.2 (5 pts) Compare the output matrix and the input matrix in Problem 4.1, briefly analyze the effect of this 3×3 kernel on the input. (Hint: apply this kernel to an image to see the outputs, attached the image you applied and result image)

In [ ]:
```python
import matplotlib.pyplot as plt
from PIL import Image


# Load and preprocess the image (color version)
def load_image(path):
    img = Image.open(path)
    img = np.array(img)
    return img


# Apply 2D convolution
def apply_convolution(image, kernel):
    # If the image has multiple channels (e.g., RGB), apply convolution to each channel
    if len(image.shape) == 3:  # Check if the image is colored (3 dimensions: H x W x C)
        output = np.zeros_like(image)
        for i in range(3):  # Apply convolution for each channel (R, G, B)
            output[:, :, i] = convolve2d(
                image[:, :, i], kernel, mode="same", boundary="fill", fillvalue=0
            )
        return output
    else:
        # For grayscale images, apply convolution as usual
        return convolve2d(image, kernel, mode="same", boundary="fill", fillvalue=0)


# Plotting the original and result images
def plot_images(original, result):
    fig, ax = plt.subplots(1, 2, figsize=(10, 5))

    ax[0].imshow(original)
    ax[0].set_title("Original Image")
```

```
    ax[0].axis("off")

    ax[1].imshow(result.astype(np.uint8))  # Ensure result is in the correct format
    ax[1].set_title("After Convolution")
    ax[1].axis("off")

    plt.show()


# Main execution
if __name__ == "__main__":
    # Load the image (provide your image path)
    image_path = "/Users/suim/Downloads/24Fall/ECE 661 Computer Engineering Maching Lear
    image = load_image(image_path)

    # Apply the 3x3 kernel
    result_image = apply_convolution(image, kernel)

    # Plot the original and the result image
    plot_images(image, result_image)
```



> The output matrix resulting from the application of the 3×3 kernel significantly alters the input image. The kernel highlights edges and contrast in the image, causing features like buildings to stand out more sharply.

5 Lab: LMS Algorithm (15 pts)

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 Canvas.

Lab 1 (15 pts)

To start with, please download the dataset.mat file from Canvas and load it into NumPy arrays. There are two variables in the file: data $X \in \mathbb{R}^{100\times 3}$ and target $D \in \mathbb{R}^{100\times 1}$. Each individual pair of data and target is composed into X and D following the same way as discussed in Lecture 2. 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 $W \in \mathbb{R}^{3\times 1}$ for the linear model that can minimize the MSE loss, which is also formulated on Lecture 2.

(a) (3pt) 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^*?

> Optimal Weight: $W^* = \begin{bmatrix} 1.0007 \\ 1.0006 \\ -2.0003 \end{bmatrix}$, MSE Loss = 0.0001008

In [ ]:
```
from scipy.io import loadmat

# load the data
file_path = "/Users/suim/Downloads/24Fall/ECE 661 Computer Engineering Maching Learning
data = loadmat(file_path)
print(data.keys())
```

```
dict_keys(['__header__', '__version__', '__globals__', 'D', 'X'])
```

In [ ]:
```
D = data["D"]
X = data["X"]

# compute the least square (Wiener) solution
w_star = np.linalg.inv(X.T @ X) @ X.T @ D
print(w_star.flatten())
```

```
[ 1.0006781   1.00061145 -2.00031968]
```

In [ ]:
```
loss = np.mean((D - X @ w_star) ** 2)
print(loss)
```

```
0.00010079903131736677
```

> Through calculations, the weight was determined using the Wiener solution, and the loss value was obtained based on this weight.

(b) (4pt) 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 $r = 0.005$, report the weight you get in the end and plot the MSE loss in $log\ scale$ vs. Epochs.

> Final weight: $[1.00068274 \quad 1.0006024 \quad -2.00033003]$

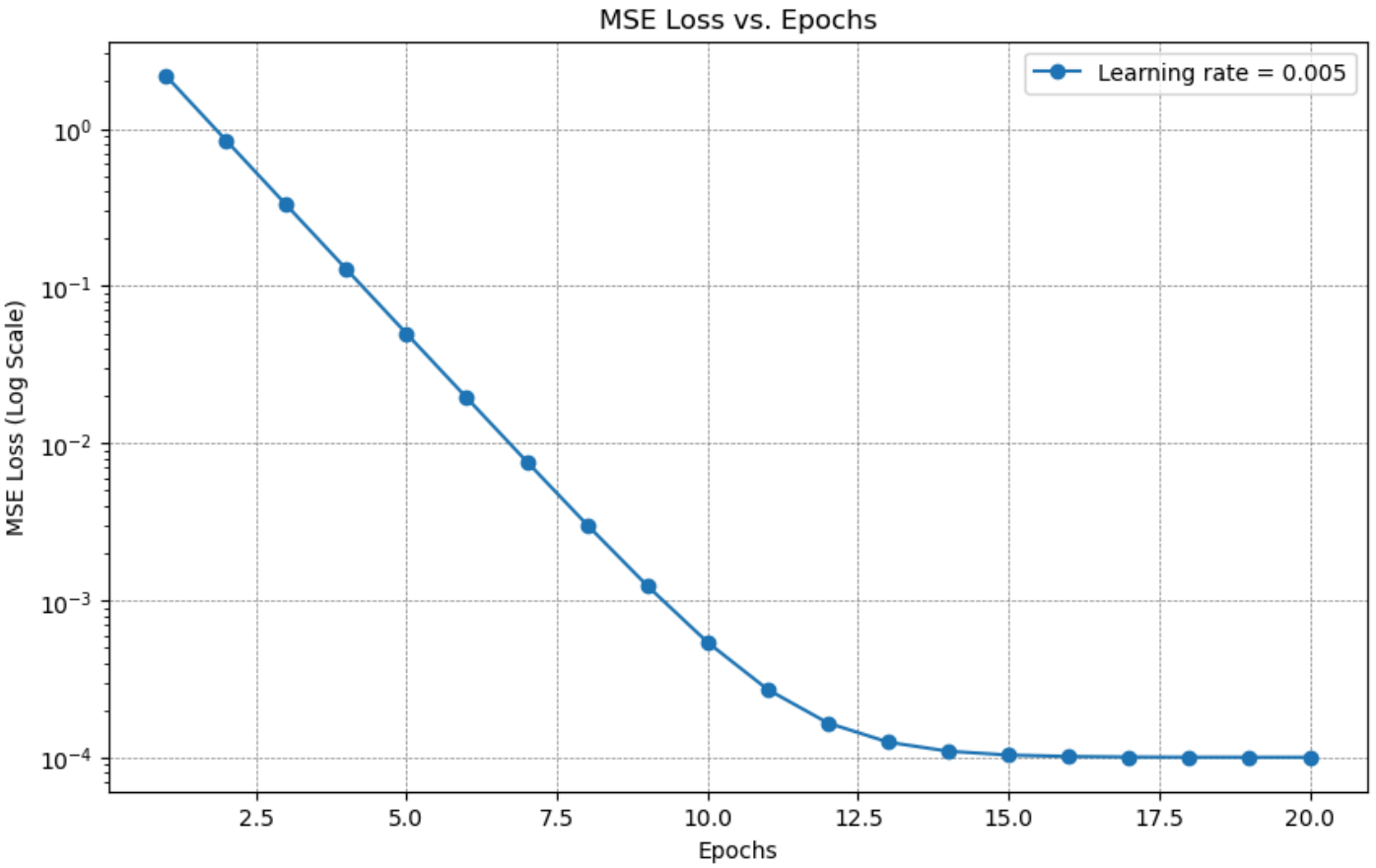
In [ ]:
```
# Initialize the weight matrix W
W = np.zeros((3, 1))  # shape (3, 1)

# Initialize the learning rate, epoch, and MSE list
learning_rate_first = 0.005
epochs = 20
mse_list = []

# Loop through each epoch and data point
for epoch in range(epochs):
    for i in range(X.shape[0]):  # Loop through each data point
```

```
        X_i = X[i].reshape(1, -1)  # shape (1, 3)
        D_i = D[i]  # scalar value

        # Prediction
        y_pred = np.dot(X_i, W)  # shape (1, 1)

        # Update weights using LMS
        W = W + learning_rate_first * (D_i - y_pred) * X_i.T  # shape (3, 1)

    # Calculate MSE for the entire dataset
    y_preds = np.dot(X, W)  # shape (100, 1)
    mse = np.mean((D - y_preds) ** 2)
    mse_list.append(mse)
```

In [ ]:
```
# Report the final weights
W_final = W.flatten()
print("Final weights:", W_final)
```

```
Final weights: [ 1.00068274  1.0006024  -2.00033003]
```

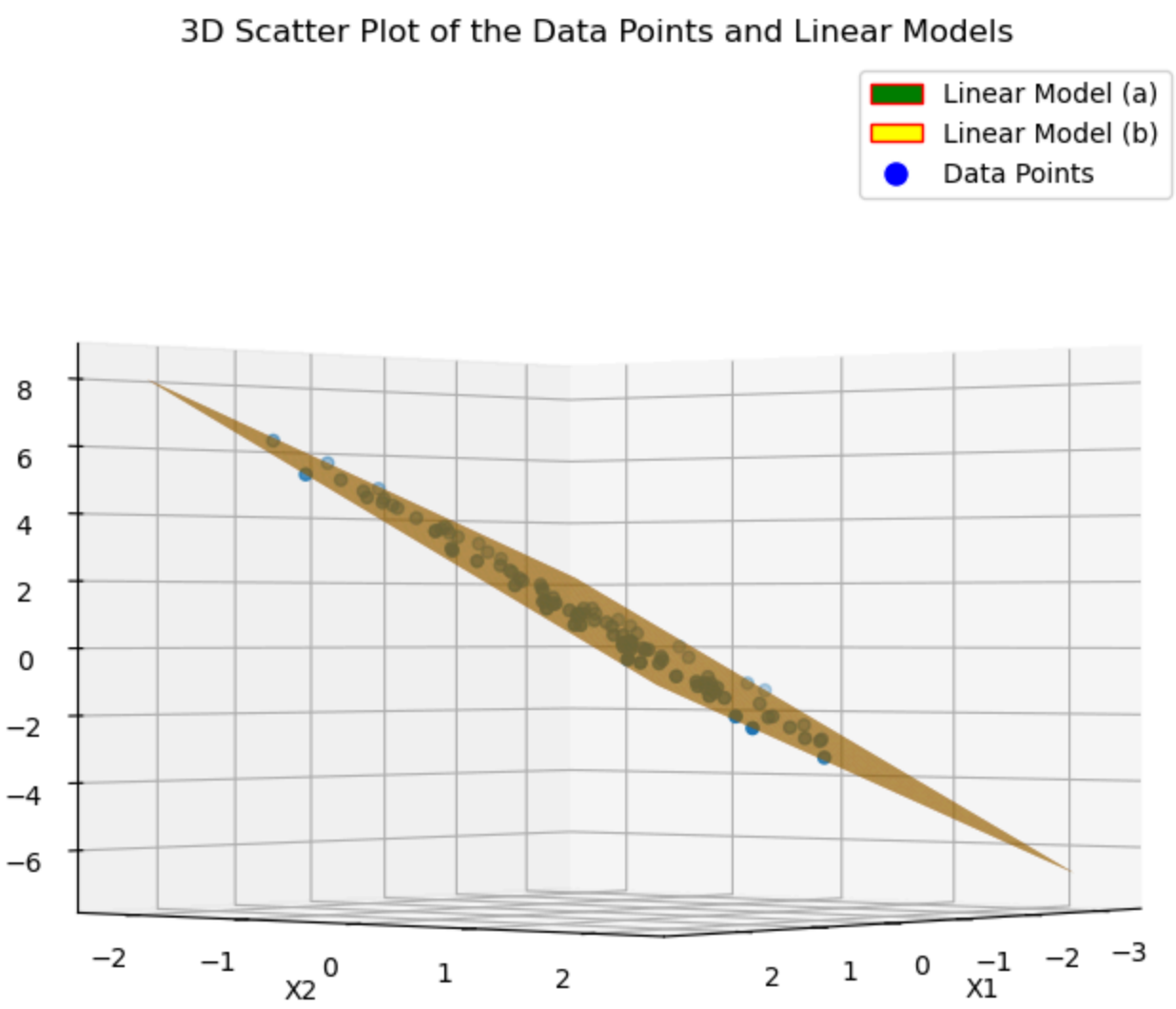
In [ ]:
```
# plot the MSE loss in log scale vs. Epochs
plt.figure(figsize=(10, 6))
plt.plot(range(1, epochs + 1), mse_list, marker="o")
plt.xlabel("Epochs")
plt.ylabel("MSE Loss (Log Scale)")
plt.yscale("log")
plt.title("MSE Loss vs. Epochs")
plt.grid(color="gray", linestyle="--", linewidth=0.5)
plt.legend(["Learning rate = 0.005"])
plt.show()
```



> If the epochs repeat, the weight is updated based on forward propagation and backpropagation, leading to a decrease in the MSE loss.

(c) (3pt) Scatter plot the points (x_{1k}, x_{2k}, d_k) for all 100 data-target pairs in a 3D figure, 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.

In [ ]:
```python
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.lines import Line2D
from matplotlib.patches import Patch

fig = plt.figure(figsize=(12, 8))
ax = fig.add_subplot(projection="3d")

ax.scatter(X[:, 1], X[:, 2], D, marker="o")

# Range for the decision boundary
x1_range = np.linspace(np.min(X[:, 1]), np.max(X[:, 1]), 100)
x2_range = np.linspace(np.min(X[:, 2]), np.max(X[:, 2]), 100)
x1_grid, x2_grid = np.meshgrid(x1_range, x2_range)

# Plot the Wiener solution
D_a = w_star[0] + w_star[1] * x1_grid + w_star[2] * x2_grid
ax.plot_surface(x1_grid, x2_grid, D_a, color="red", alpha=0.5, label="Linear Model (a)")

D_b = W[0] + W[1] * x1_grid + W[2] * x2_grid
ax.plot_surface(
    x1_grid, x2_grid, D_b, color="yellow", alpha=0.5, label="Linear Model (a)"
)

ax.set_title("3D Scatter Plot of the Data Points and Linear Models")
ax.set_xlabel("X1")
ax.set_ylabel("X2")
ax.set_zlabel("D")
ax.view_init(elev=0, azim=40)

# Custom legend for the surfaces
legend_elements = [
    Patch(facecolor="green", edgecolor="r", label="Linear Model (a)"),
    Patch(facecolor="yellow", edgecolor="r", label="Linear Model (b)"),
    Line2D(
        [0],
        [0],
        marker="o",
        color="w",
        markerfacecolor="blue",
        markersize=10,
        label="Data Points",
    ),
]
ax.legend(handles=legend_elements)

plt.show()
```



> It can be observed that the data points align well with the two linear models.

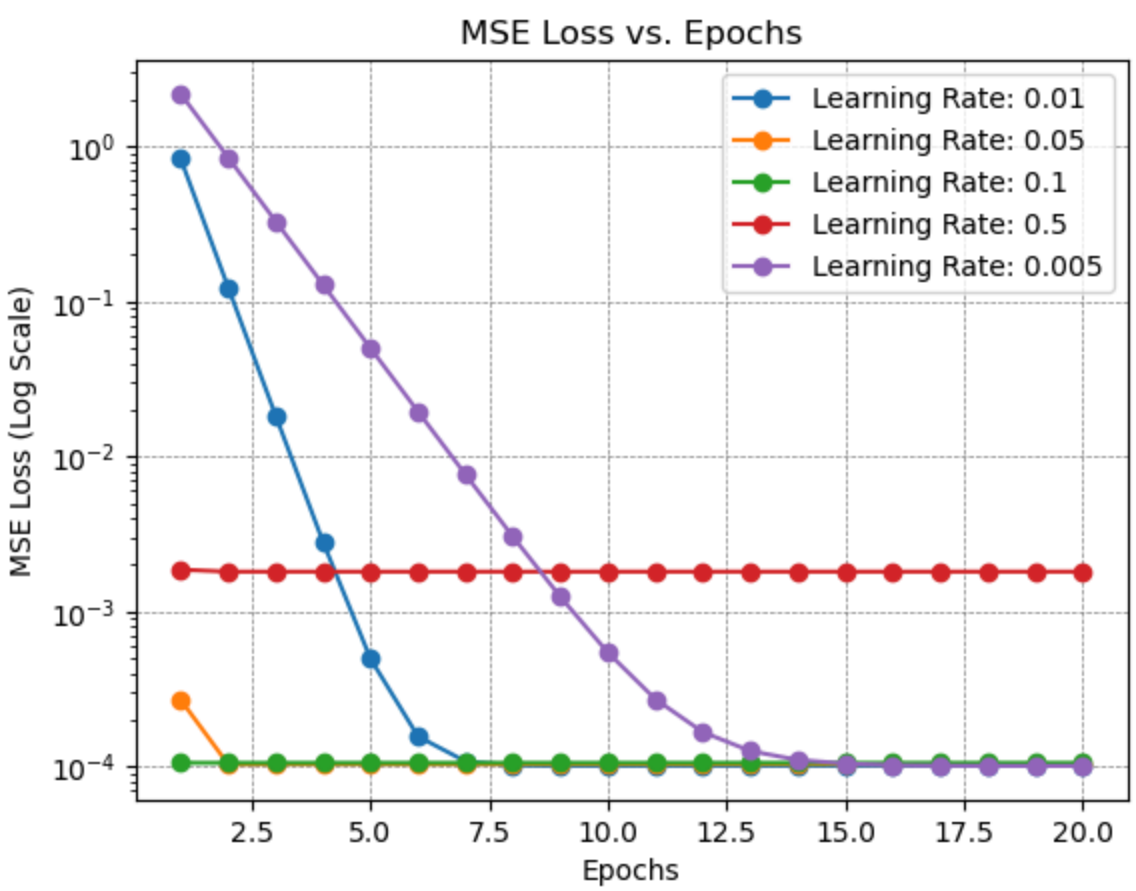
(d) (5pt) Learning rate r is an important hyperparameter for the LMS algorithm, as well as for CNN optimization. Here, try repeat the process in (b) with r set to 0.01, 0.05, 0.1 and 0.5 respectively. Together with the result you got in (b), plot the MSE losses of all sets of experiments in log scale vs. Epochs in one figure.

In [ ]:
```python
# Different learning rate (0.01, 0.05, 0.1, 0.5)
learning_rates = [0.01, 0.05, 0.1, 0.5]

# Initialize the weight matrix W
weights = [np.zeros((3, 1)) for _ in learning_rates]

# Initialize the MSE list
mse_lists = [[] for _ in learning_rates]

# Loop through each epoch and data point
for epoch in range(epochs):
    for i in range(X.shape[0]):  # Loop through each data point
```

```
        X_i = X[i].reshape(1, -1)  # shape (1, 3)
        D_i = D[i]  # scalar value

        # Update weights for each learning rate
        for j, learning_rate in enumerate(learning_rates):
            # Prediction
            y_pred = np.dot(X_i, weights[j])  # shape (1, 1)

            # Update weights using LMS
            weights[j] = (
                weights[j] + learning_rate * (D_i - y_pred) * X_i.T
            )  # shape (3, 1)

    # Calculate MSE for the entire dataset for each learning rate
    for j in range(len(learning_rates)):
        y_preds = np.dot(X, weights[j])  # shape (100, 1)
        mse = np.mean((D - y_preds) ** 2)
        mse_lists[j].append(mse)

# Report the final weights and MSE lists
W_1, W_2, W_3, W_4 = weights
mse_list_1, mse_list_2, mse_list_3, mse_list_4 = mse_lists
```

In [ ]:

```
# Plot the MSE loss in log scale vs. Epochs
for i, learning_rate in enumerate(learning_rates):
    plt.plot(
        range(1, epochs + 1),
        mse_lists[i],
        label=f"Learning Rate: {learning_rate}",
        marker="o",
    )

plt.plot(
    range(1, epochs + 1),
    mse_list,
    label=f"Learning Rate: {learning_rate_first}",
    marker="o",
)
plt.xlabel("Epochs")
plt.ylabel("MSE Loss (Log Scale)")
plt.yscale("log")
plt.title("MSE Loss vs. Epochs")
plt.grid(color="gray", linestyle="--", linewidth=0.5)
plt.legend()
plt.show()
```



> The graph clearly shows that the reduction in MSE Loss varies depending on the learning rate. Specifically, a significant decrease in MSE Loss is observed when the learning rate is set to 0.005 or 0.01. In contrast, with other learning rates, such as 0.1 or 0.5, the changes in loss are less pronounced and remain almost constant. This highlights the critical importance of selecting an appropriate learning rate when building a neural network, as it greatly affects the model's performance and convergence.

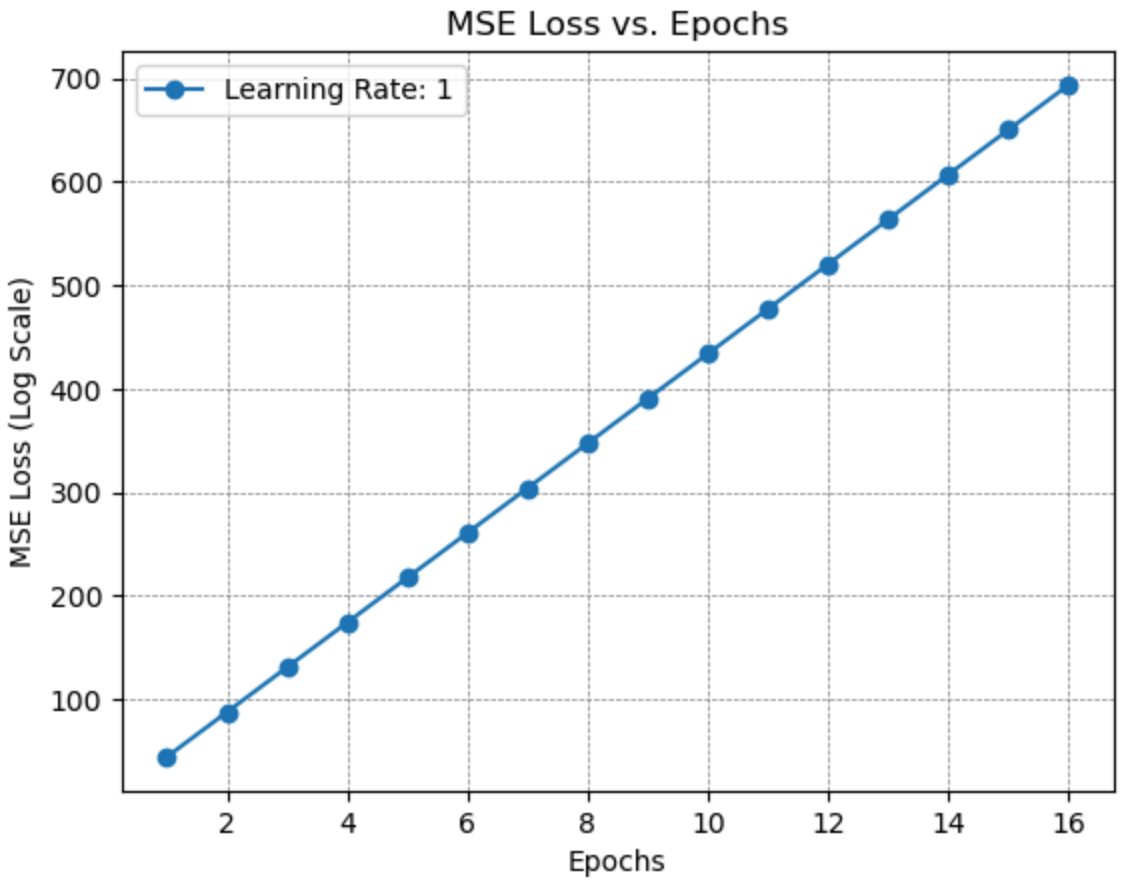
Then try further enlarge the learning rate to $r = 1$ and observe how the MSE changes. Base on these observations, comment on how learning rate affects the speed and quality of the learning process. (Note: The learning rate tuning for the CNN optimization will be introduced in Lecture 7.)

In [ ]:
```python
# Initialize the learning rate (1)
learning_rate_big = 1
mse_list_big = []
W = np.zeros((3, 1))

# Loop through each epoch and data point
for epoch in range(epochs):
    for i in range(X.shape[0]):  # Loop through each data point
        X_i = X[i].reshape(1, -1)  # shape (1, 3)
        D_i = D[i]  # scalar value

        # Prediction
        y_pred = np.dot(X_i, W)  # shape (1, 1)

        # Update weights using LMS
        W = W + learning_rate_big * (D_i - y_pred) * X_i.T  # shape (3, 1)

    # Calculate MSE for the entire dataset
    y_preds = np.dot(X, W)  # shape (100, 1)
```

```
    mse = np.mean((D - y_preds) ** 2)
    mse_list_big.append(mse)
```

```
/var/folders/6s/3xxfn9n94l76v9fzq_1cwkf80000gn/T/ipykernel_94271/4265597254.py:20: Runtim
eWarning: overflow encountered in square
  mse = np.mean((D - y_preds) ** 2)
```

In [ ]:
```
plt.plot(
    range(1, epochs + 1),
    np.log(mse_list_big),
    label=f"Learning Rate: {learning_rate_big}",
    marker="o",
)
plt.xlabel("Epochs")
plt.ylabel("MSE Loss (Log Scale)")
# plt.yscale("log")
plt.title("MSE Loss vs. Epochs")
plt.grid(color="gray", linestyle="--", linewidth=0.5)
plt.legend()
plt.show()
```



> When the learning rate is set to 1, the MSE loss increases rapidly, leading to a potential overflow. In the graph, this is evident as the MSE loss grows significantly with each epoch. By the 16th epoch, the MSE loss becomes too large, causing an overflow that the graph cannot display beyond this point. This rapid increase suggests that the learning rate is too high, preventing the model from converging properly and instead driving the loss values upwards uncontrollably.

6 Lab: Simple NN (35 pts)

For getting started with deep Neural Network model easily, we consider a simple Neural Network model here and details of the model architecture is given in Table 1. This lab question focuses on building the model in PyTorch and observing the shape of each layer's input, weight and output. Please refer to the **NumPy/PyTorch Tutorial slides** on Canvas and the official documentations if you are unfamiliar with PyTorch syntax. Please finish this lab by completing the `SimpleNN.ipynb` notebook file provided on Canvas. The completed notebook file should be submitted to Canvas.

Name	Type	Kernel size	depth/units	Activation	Strides
Conv 1	Convolution	5	16	ReLU	1
MaxPool	MaxPool	4	N/A	N/A	2
Conv 2	Convolution	3	16	ReLU	1
MaxPool	MaxPool	3	N/A	N/A	2
Conv 3	Convolution	7	32	ReLU	1
MaxPool	MaxPool	2	N/A	N/A	2
FC1	Fully-connected	N/A	32	ReLU	N/A
FC2	Fully-connected	N/A	10	ReLU	N/A

Table 1: The padding for all three convolution layers is 2. The padding for all three MaxPool layers is 0. A flatten layer is required before FC1 to reshape the feature.

In the notebook, first run through the first two code blocks, then follow the instructions in the following questions to complete each code block and acquire the answers.

(a) (10pt) Complete code block 3 for defining the adapted SimpleNN model. Note that customized CONV and FC classes are provided in code block 2 to replace the nn.Conv2d and nn.Linear classes inPyTorchrespectively. 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 SimpleNN class into the report PDF.

In [ ]:
```
import argparse
import os, sys
import time
import datetime
import math
import numpy as np
import matplotlib.pyplot as plt

# Import pytorch dependencies
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.nn.modules.utils import _single, _pair, _triple
```

In [ ]:
```
class CONV(nn.Conv2d):
    def __init__(
```

```
        self,
        in_channels,
        out_channels,
        kernel_size,
        stride=1,
        padding=0,
        dilation=1,
        groups=1,
        bias=False,
        padding_mode="zeros",
    ):
        self.input = None
        self.output = None
        kernel_size = _pair(kernel_size)
        stride = _pair(stride)
        padding = _pair(padding)
        dilation = _pair(dilation)
        super(CONV, self).__init__(
            in_channels,
            out_channels,
            kernel_size,
            stride,
            padding,
            dilation,
            groups,
            bias,
            padding_mode,
        )

    def forward(self, input):
        self.input = input
        # 'pytorch 2.0'
        self.output = super().forward(input)
        return self.output


class FC(nn.Linear):
    def __init__(self, in_features, out_features, bias=True):
        self.input = None
        self.output = None
        super(FC, self).__init__(in_features, out_features, bias)

    def forward(self, input):
        self.input = input
        self.output = F.linear(input, self.weight, self.bias)
        return self.output
```

In [ ]:

```
"""
Lab 2(a)
Build the SimpleNN model by following Table 1
"""


# 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=16, kernel_size=5, stride=1, padding=2
        )  # Your code here
```

```
        self.conv2 = CONV(
            in_channels=16, out_channels=16, kernel_size=3, stride=1, padding=2
        )  # Your code here
        self.conv3 = CONV(
            in_channels=16, out_channels=32, kernel_size=7, stride=1, padding=2
        )  # Your code here
        self.fc1 = FC(in_features=32 * 3 * 3, out_features=32)  # Your code here
        self.fc2 = FC(in_features=32, out_features=10)  # Your code here

    def forward(self, x):
        # Forward pass computation
        # Conv 1
        x = self.conv1(x)  # Your code here
        x = F.relu(x)
        # MaxPool
        x = F.max_pool2d(x, kernel_size=4, stride=2)  # Your code here
        # Conv 2
        x = self.conv2(x)  # Your code here
        x = F.relu(x)
        # MaxPool
        x = F.max_pool2d(x, kernel_size=3, stride=2)  # Your code here
        # Conv 3
        x = self.conv3(x)  # Your code here
        x = F.relu(x)
        # MaxPool
        x = F.max_pool2d(x, kernel_size=2, stride=2)  # Your code here
        # Flatten
        x = torch.flatten(x, 1)  # Your code here
        # FC 1
        x = self.fc1(x)
        x = F.relu(x)  # Your code here
        # FC 2
        x = self.fc2(x)  # Your code here
        out = F.relu(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 = SimpleNN()
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 CPU...
Forward pass successful
```

(b) (25pt) 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.

In [ ]:
```
"""
Lab 2(b)
"""

# Forward pass of a single image
data = torch.randn(1, 3, 32, 32).to(device)
# Forward pass "data" through "net" to get output "out"
out = net(data)  # Your code here

# Iterate through all the CONV and FC layers of the model
for name, module in net.named_modules():
    if isinstance(module, CONV) or isinstance(module, FC):
        # Get the input feature map of the module as a NumPy array
        input = module.input.detach().cpu().numpy()  # Your code here
        # Get the output feature map of the module as a NumPy array
        output = module.output.detach().cpu().numpy()  # Your code here
        # Get the weight of the module as a NumPy array
        weight = module.weight.detach().cpu().numpy()  # Your code here
        # Compute the number of parameters in the weight
        num_Param = weight.size  # Your code here
        # Compute the number of MACs in the layer
        if isinstance(module, CONV):
            # For CONV layers: MACs = output height * output width * kernel height * ker
            output_h, output_w = (
                output.shape[2],
                output.shape[3],
            )  # Get the height and width of the output
            kernel_h, kernel_w = weight.shape[2], weight.shape[3]  # Get the kernel size
            input_channels = weight.shape[1]  # Get the number of input channels
            output_channels = weight.shape[0]  # Get the number of output channels
            num_MAC = (
                output_h
                * output_w
                * kernel_h
                * kernel_w
                * input_channels
                * output_channels
            )
        elif isinstance(module, FC):
            # For FC layers: MACs = input features * output features
            num_MAC = module.in_features * module.out_features

        print(
            f"{name:10} {str(input.shape):20} {str(output.shape):20} {str(weight.shape):
        )
```

```
conv1      (1, 3, 32, 32)       (1, 16, 32, 32)      (16, 3, 5, 5)        1200       1228
800
conv2      (1, 16, 15, 15)      (1, 16, 17, 17)      (16, 16, 3, 3)       2304       6658
56
conv3      (1, 16, 8, 8)        (1, 32, 6, 6)        (32, 16, 7, 7)       25088      9031
68
fc1        (1, 288)             (1, 32)              (32, 288)            9216       9216
fc2        (1, 32)              (1, 10)              (10, 32)             320        320
```

Layer	Input shape	Output shape	Weight shape	# Param	# MAC
Conv 1	(1, 3, 32, 32)	(1, 16, 32, 32)	(16, 3, 5, 5)	1200	1228800
Conv 2	(1, 16, 15, 15)	(1, 16, 17, 17)	(16, 16, 3, 3)	2304	665856
Conv 3	(1, 16, 8, 8)	(1, 32, 6, 6)	(32, 16, 7, 7)	25088	903168
FC1	(1, 288)	(1, 32)	(32, 288)	9216	9216
FC2	(1, 32)	(1, 10)	(10, 32)	320	320

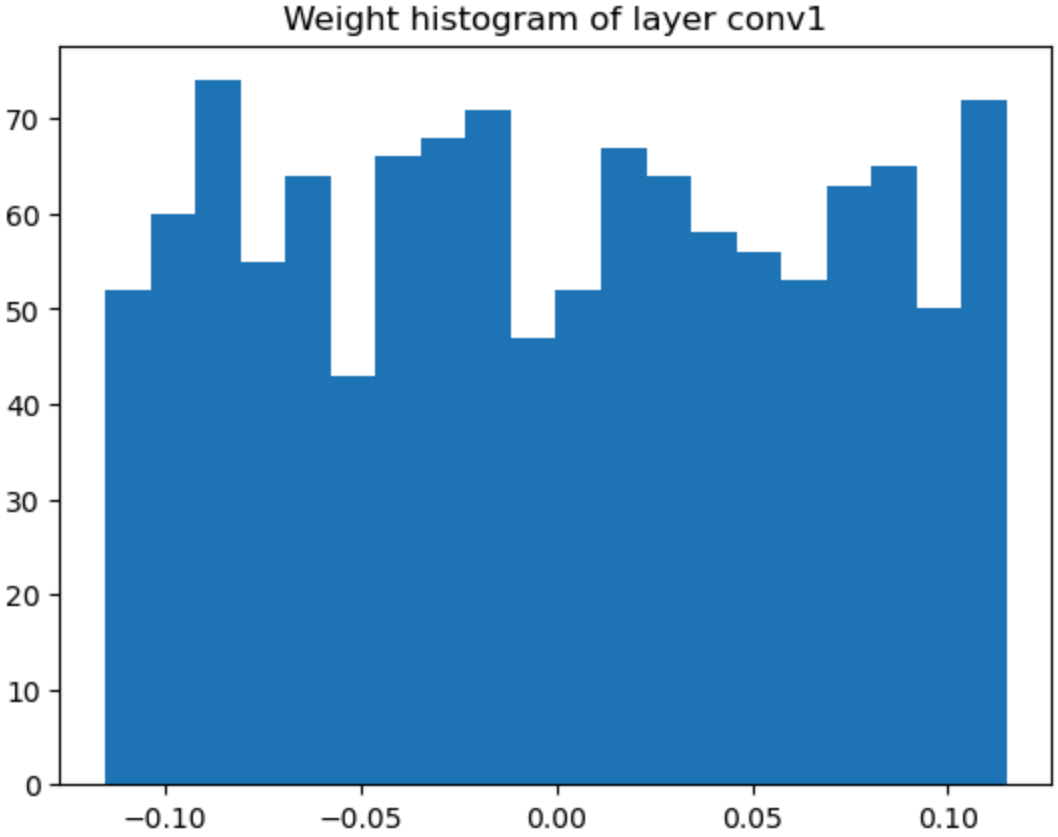
Table 2: Results of Lab 2(b).

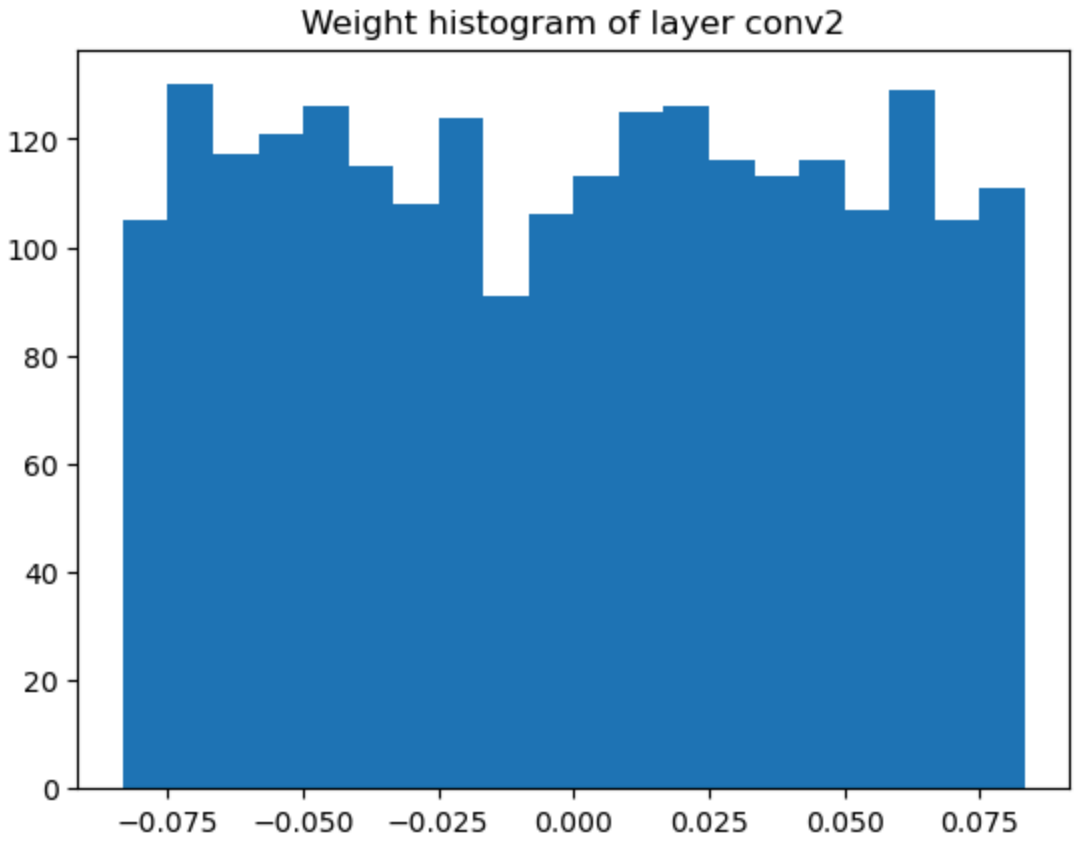
Lab 3 (Bonus 10 points)

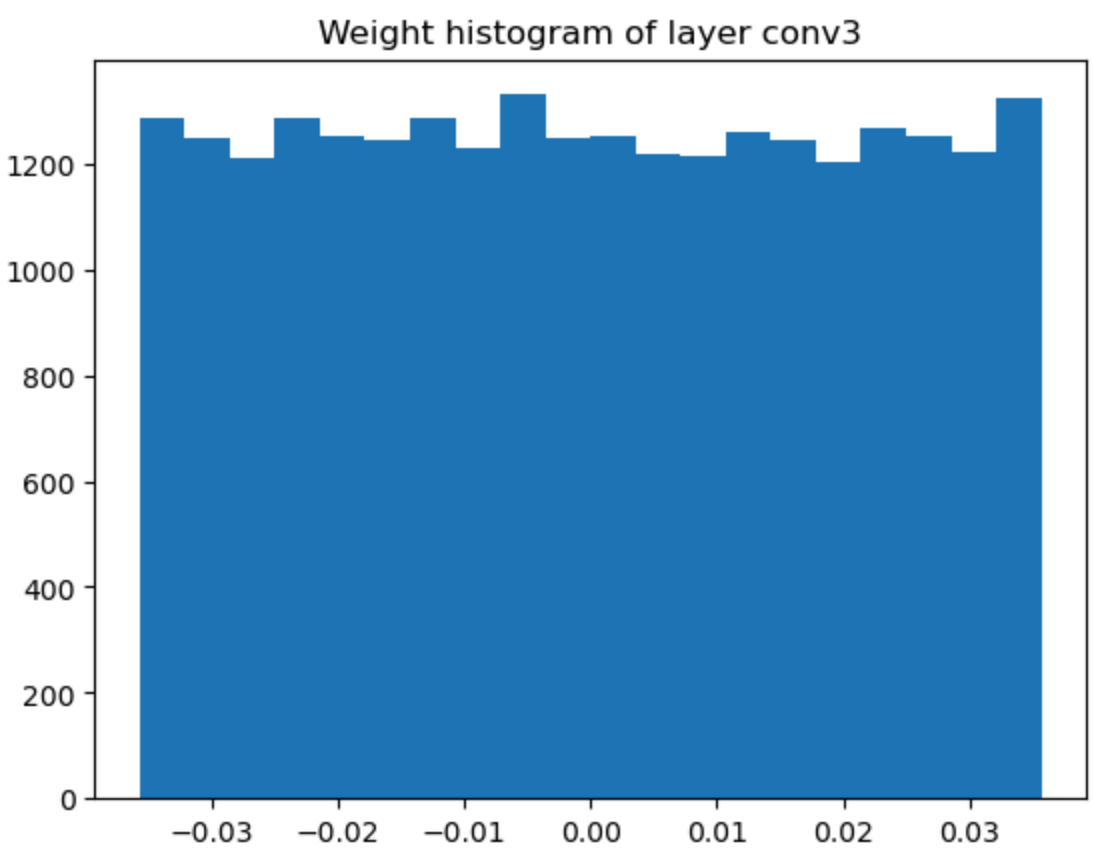
Please first finish all the required codes in Lab 2, then proceed to code block 5 of the notebook file.

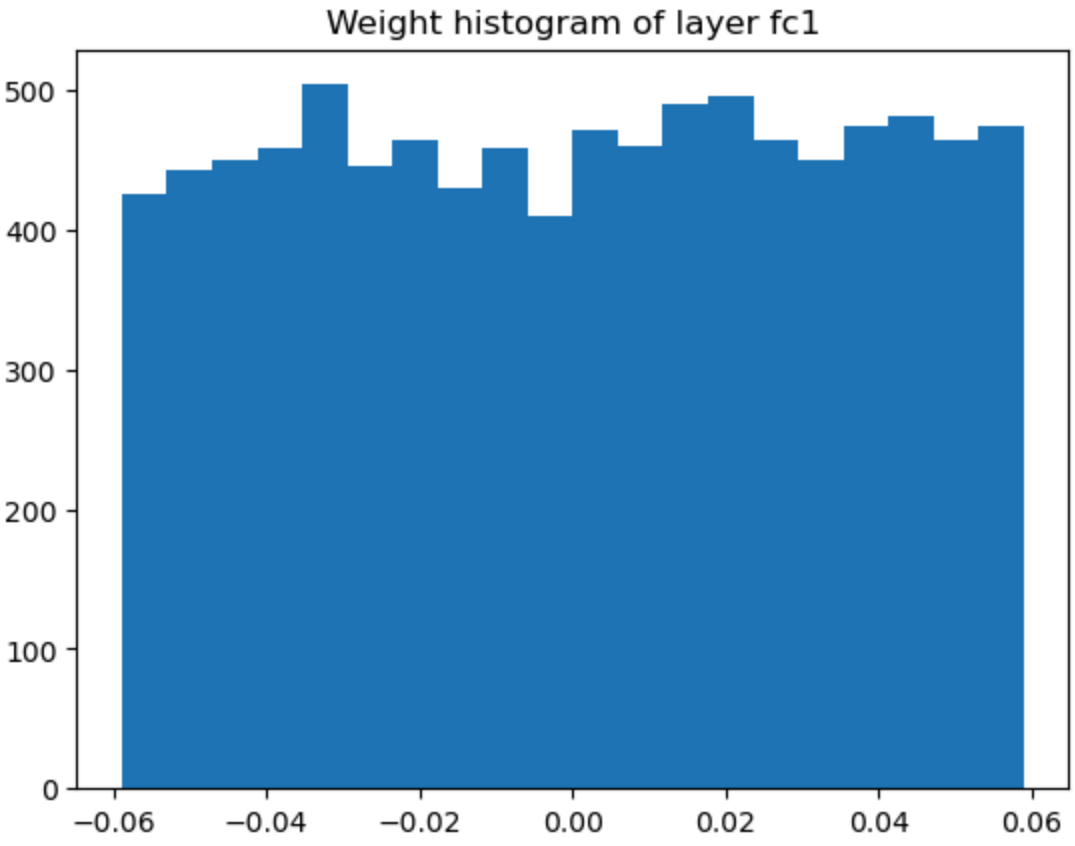
(a) (2pt) Complete the for-loop in code block 5 to plot the histogram of weight elements in each one of the 5 convolutional and fully-connected (FC) layers.

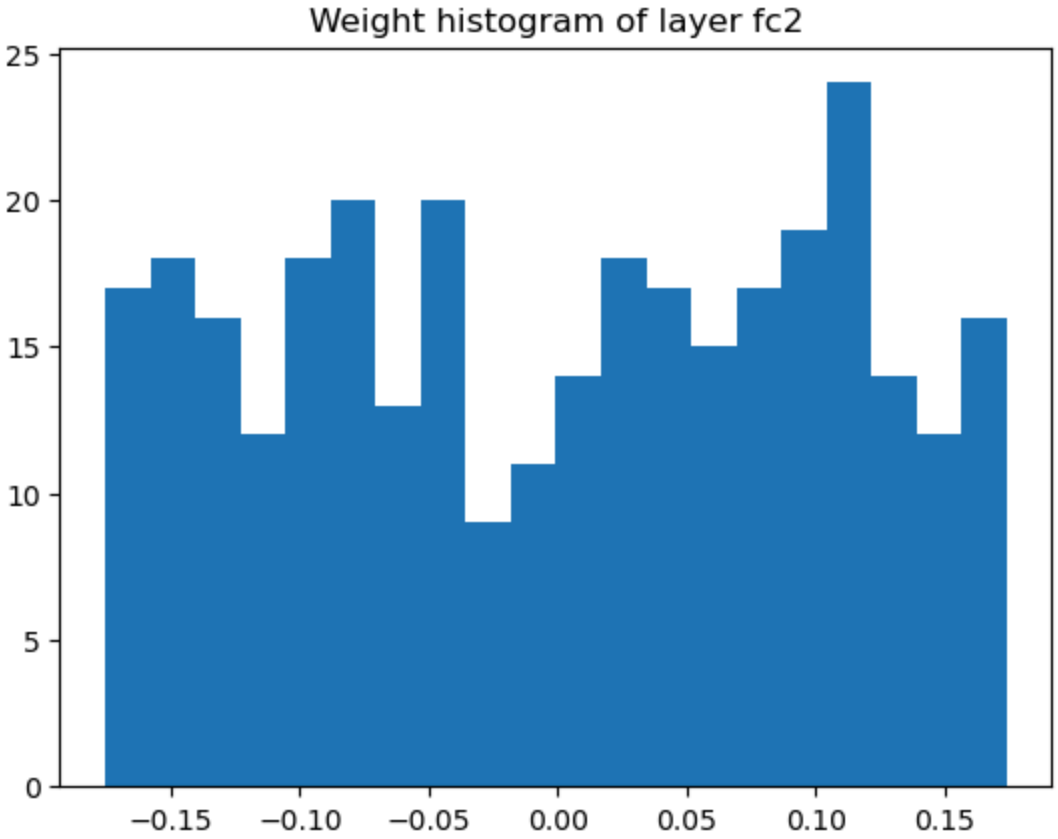
In [ ]:
```
"""
Lab 3(a)
"""

for name, module in net.named_modules():
    if isinstance(module, CONV) or isinstance(module, FC):
        # Get the weight of the module as a NumPy array
        weight = module.weight.detach().cpu().numpy()  # Your code here

        # Reshape for histogram
        weight = weight.reshape(-1)
        _ = plt.hist(weight, bins=20)
        plt.title("Weight histogram of layer " + name)
        plt.show()
```


Weight histogram of layer conv1
70
60
50
40
30
20
10
0
−0.10
−0.05
0.00
0.05
0.10


Weight histogram of layer conv2
120
100
80
60
40
20
0
−0.075
−0.050
−0.025
0.000
0.025
0.050
0.075


Weight histogram of layer conv3
1200
1000
800
600
400
200
0
−0.03
−0.02
−0.01
0.00
0.01
0.02
0.03

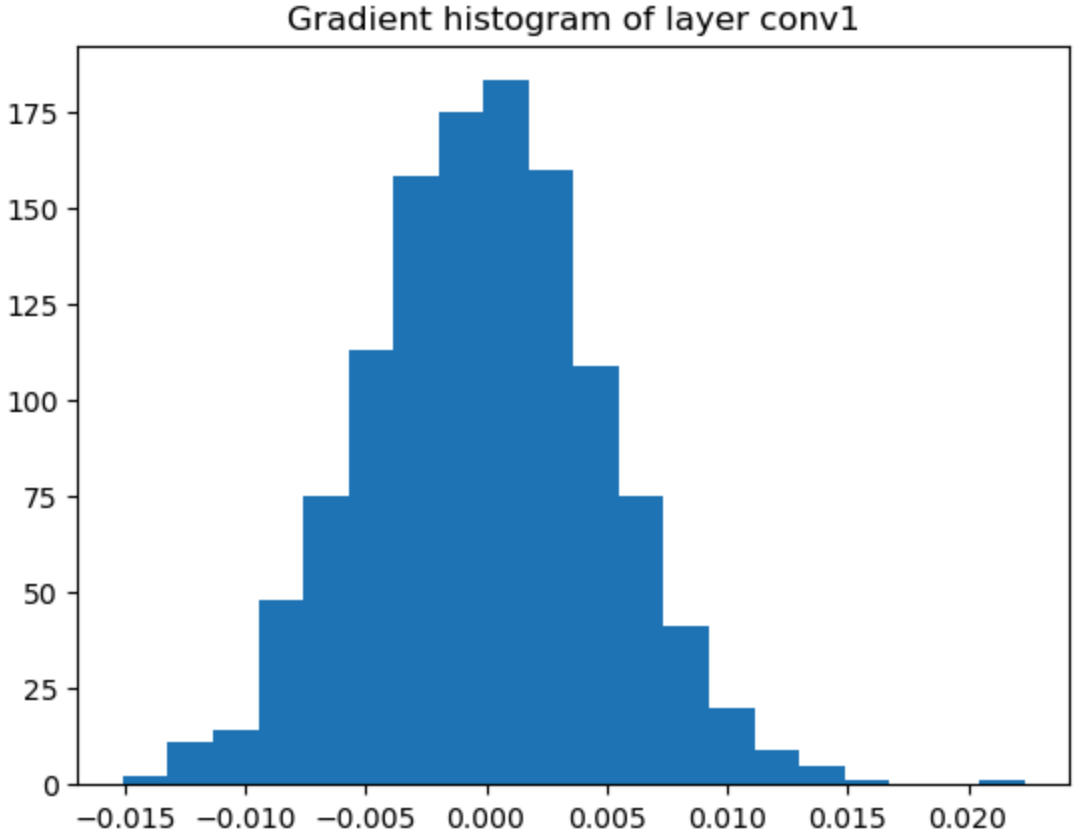

Weight histogram of layer fc1
500
400
300
200
100
0
−0.06
−0.04
−0.02
0.00
0.02
0.04
0.06

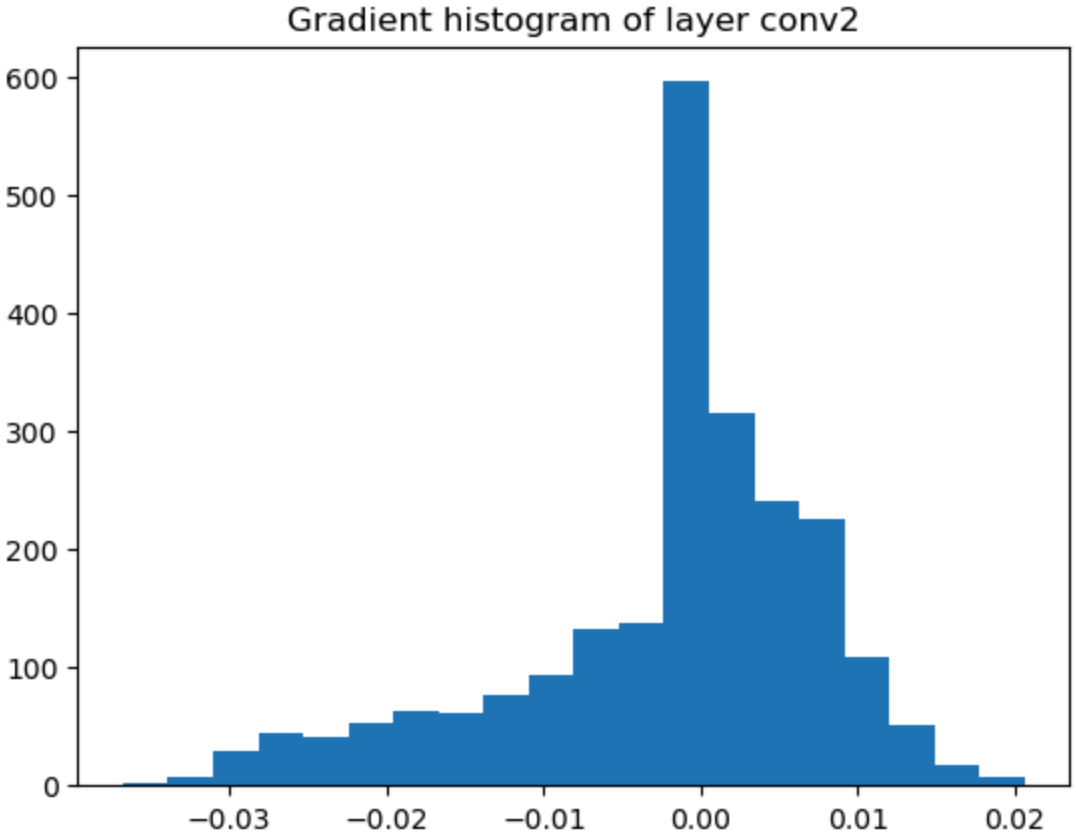


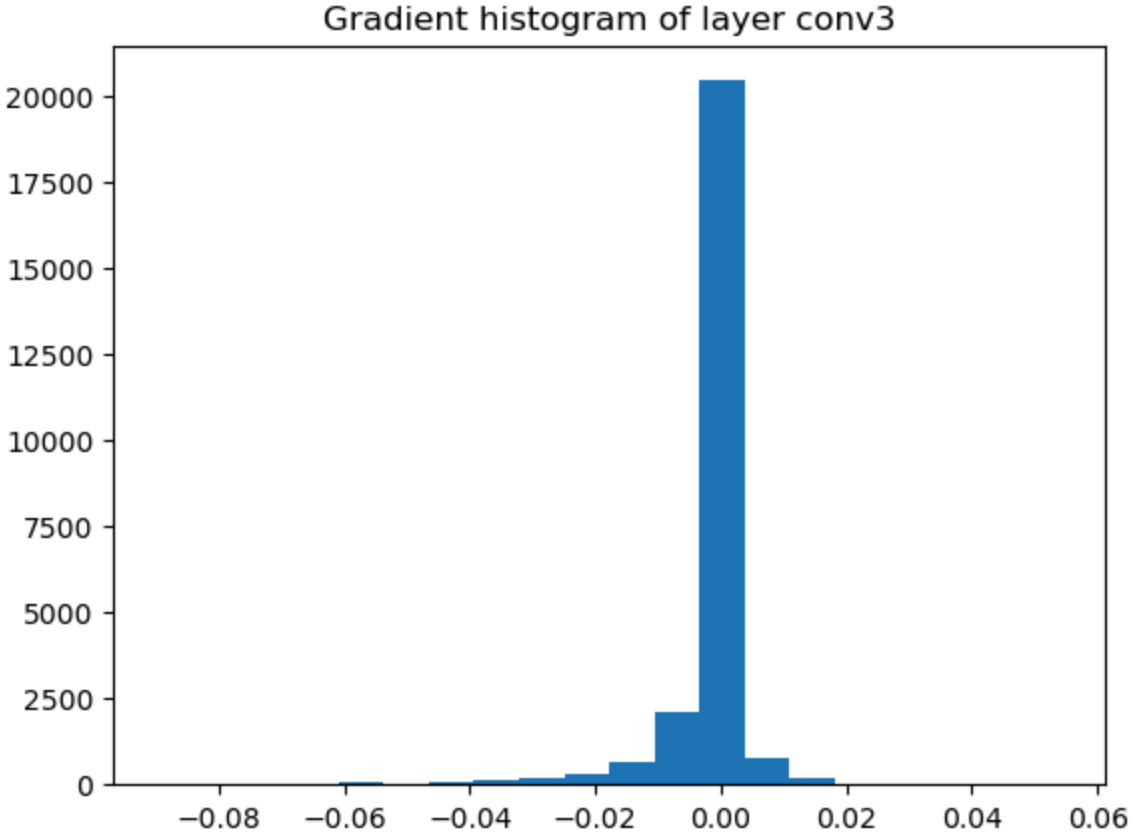
(b) (3pt) In code block 6, complete the code for backward pass, then complete the for-loop to plot the histogram of weight elements' gradients in each one of the 5 convolutional and FC layers.

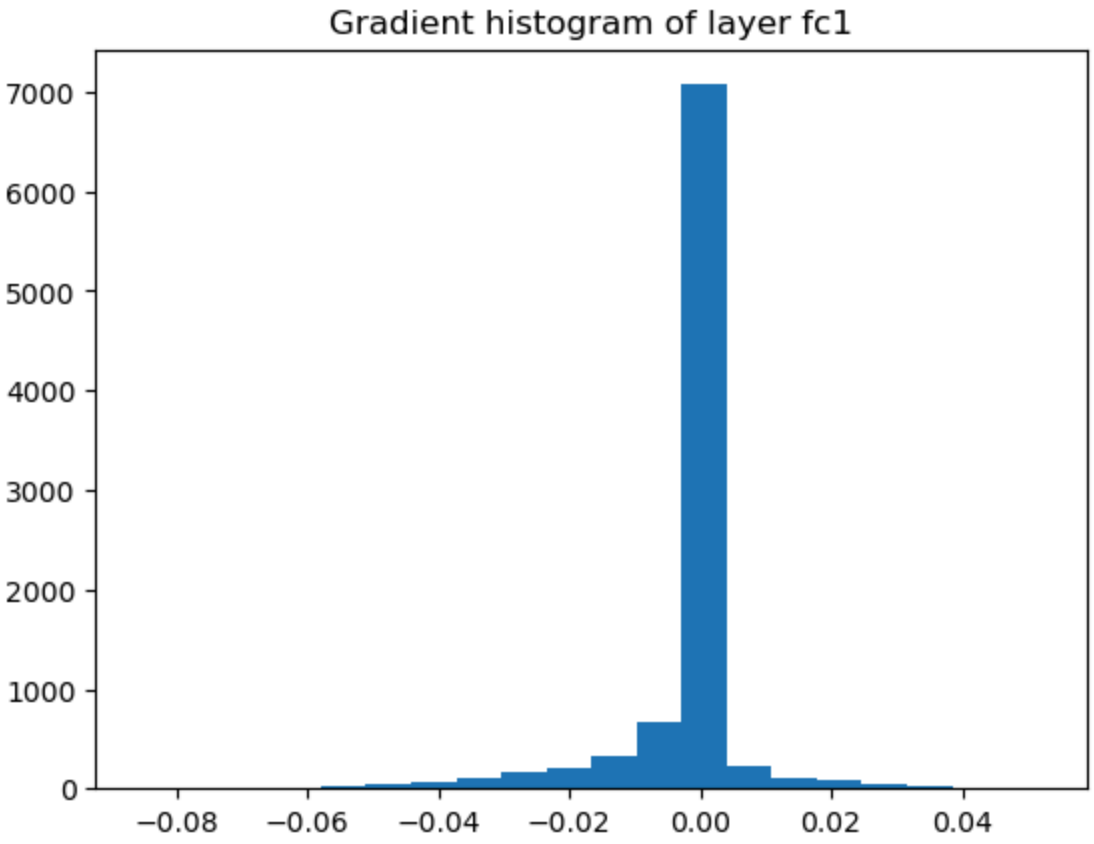
In [ ]:

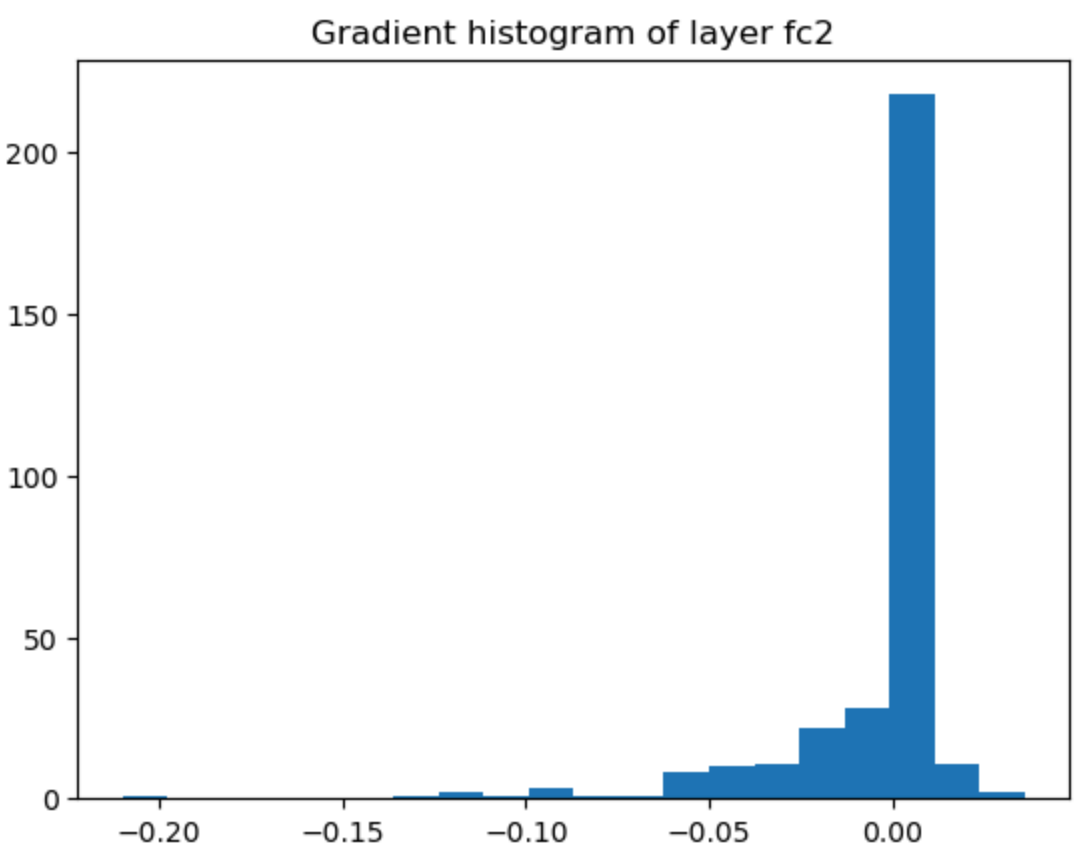
```
"""
Lab 3(b)
"""

# Loss definition
criterion = nn.MSELoss()
# Random target
target = torch.randn(1, 10).to(device)

# Loss computation
loss = criterion(out, target)  # Your code here
# Backward pass for gradients
loss.backward()  # Your code here

for name, module in net.named_modules():
    if isinstance(module, CONV) or isinstance(module, FC):
        # Get the gradient of the module as a NumPy array
        gradient = module.weight.grad.cpu().detach().numpy()  # Your code here

        # Reshape for histogram
        gradient = gradient.reshape(-1)
        _ = plt.hist(gradient, bins=20)
        plt.title("Gradient histogram of layer " + name)
        plt.show()
```


Gradient histogram of layer conv1
175
150
125
100
75
50
25
0
−0.015
−0.010
−0.005
0.000
0.005
0.010
0.015
0.020


Gradient histogram of layer conv2
600
500
400
300
200
100
0
−0.03
−0.02
−0.01
0.00
0.01
0.02


Gradient histogram of layer conv3
20000
17500
15000
12500
10000
7500
5000
2500
0
−0.08
−0.06
−0.04
−0.02
0.00
0.02
0.04
0.06


Gradient histogram of layer fc1
7000
6000
5000
4000
3000
2000
1000
0
−0.08
−0.06
−0.04
−0.02
0.00
0.02
0.04

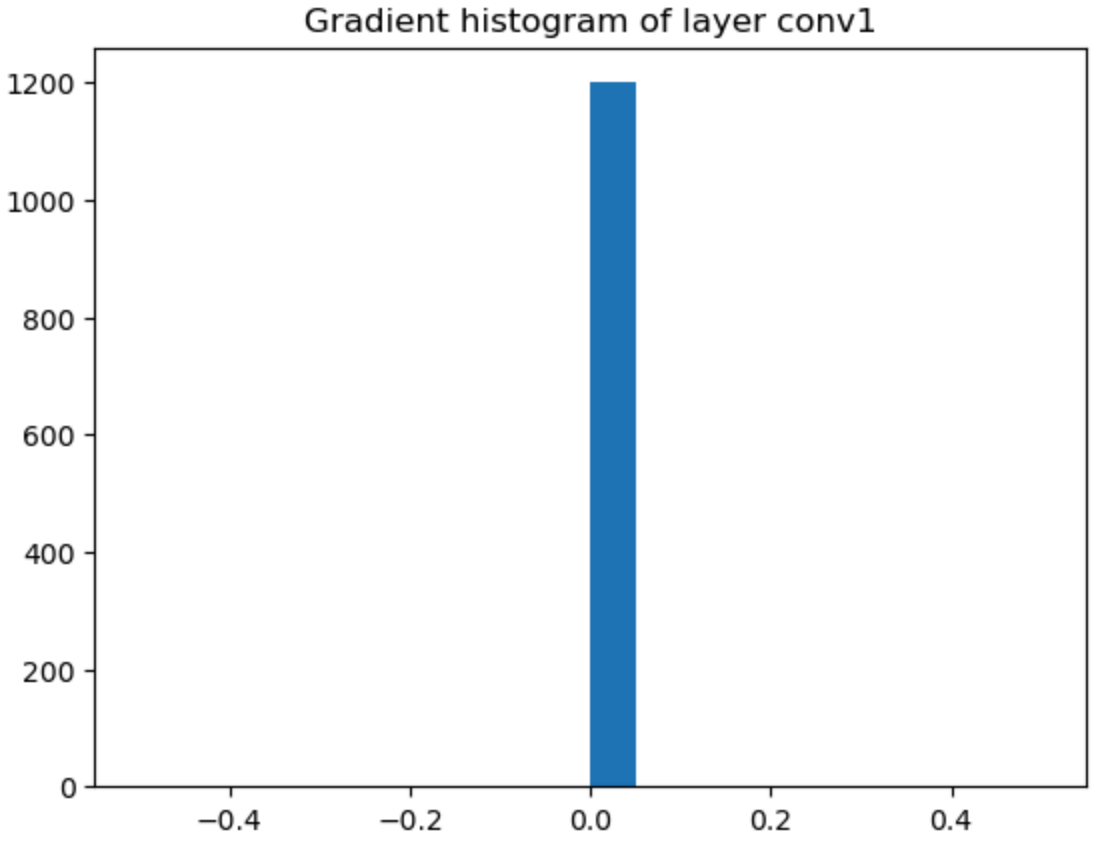


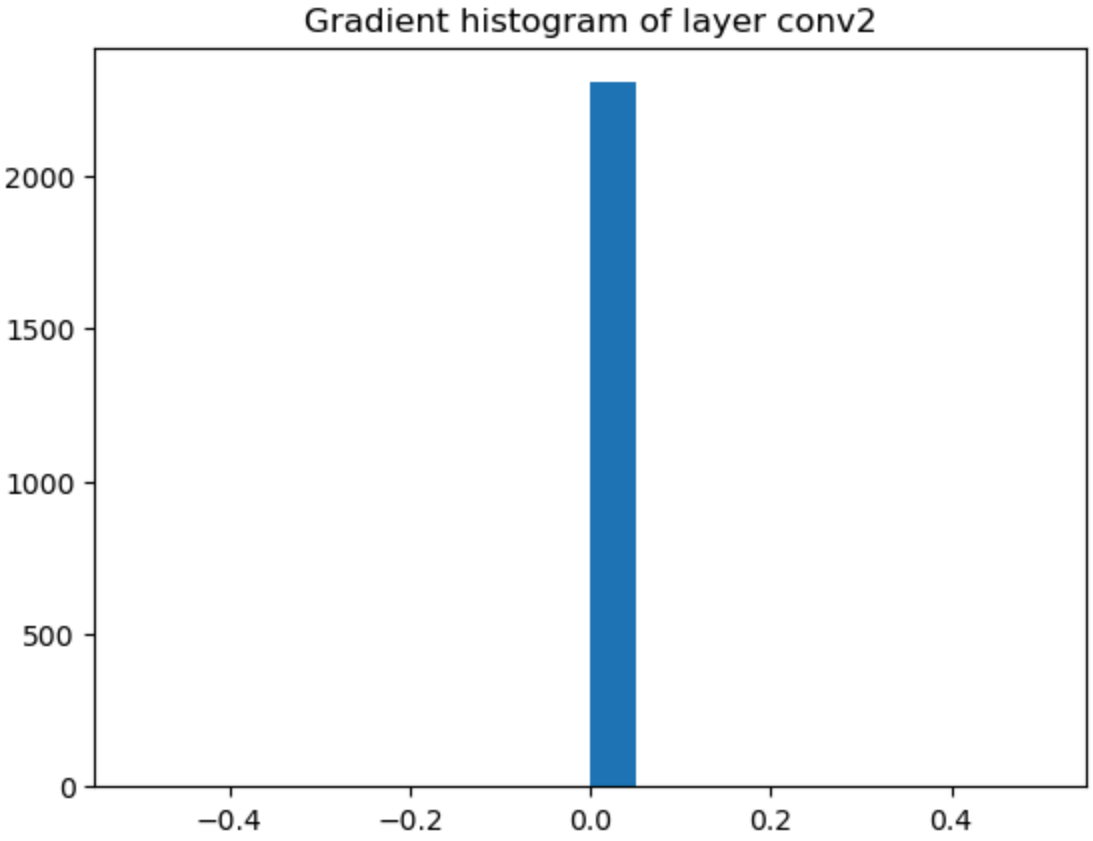
(c) (5pt) In code block 7, finish the code to set all the weights to 0. Perform forward and backward pass again to get the gradients, and plot the histogram of weight elements' gradients in each one of the 5 convolutional and fully-connected layers. Comparing with the histograms you got in (b), are there 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.)

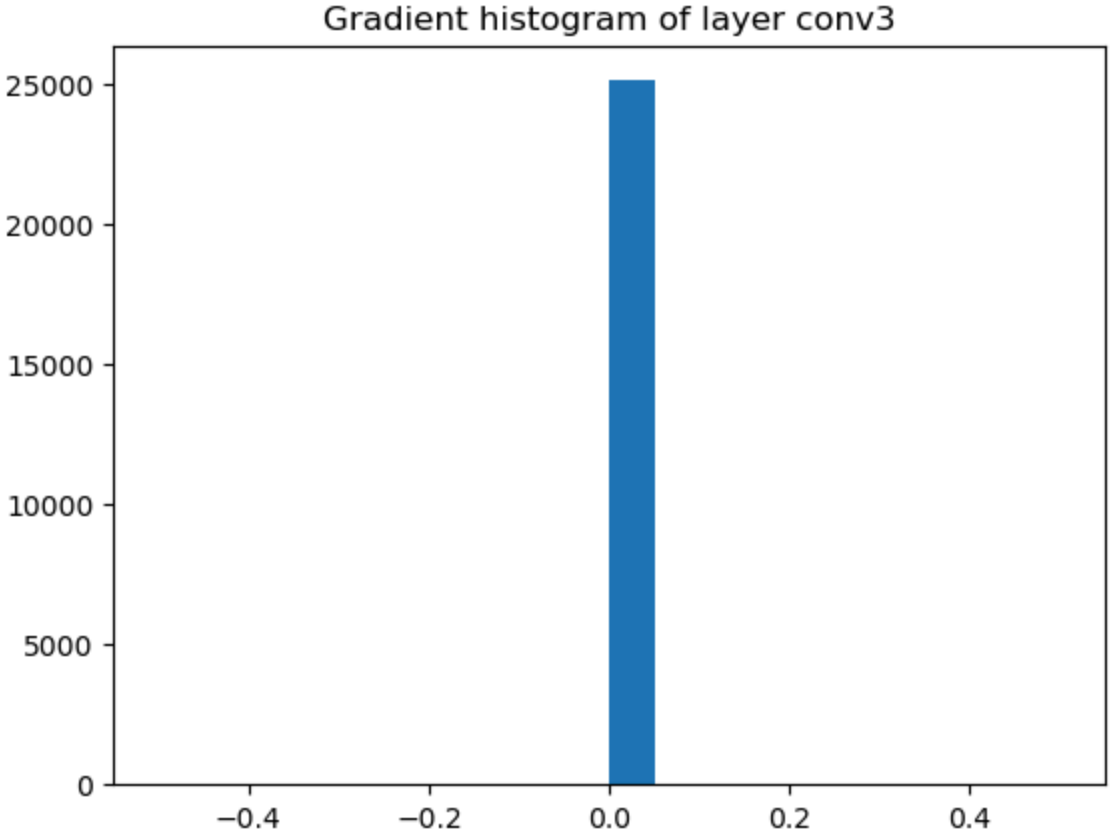
In [ ]:

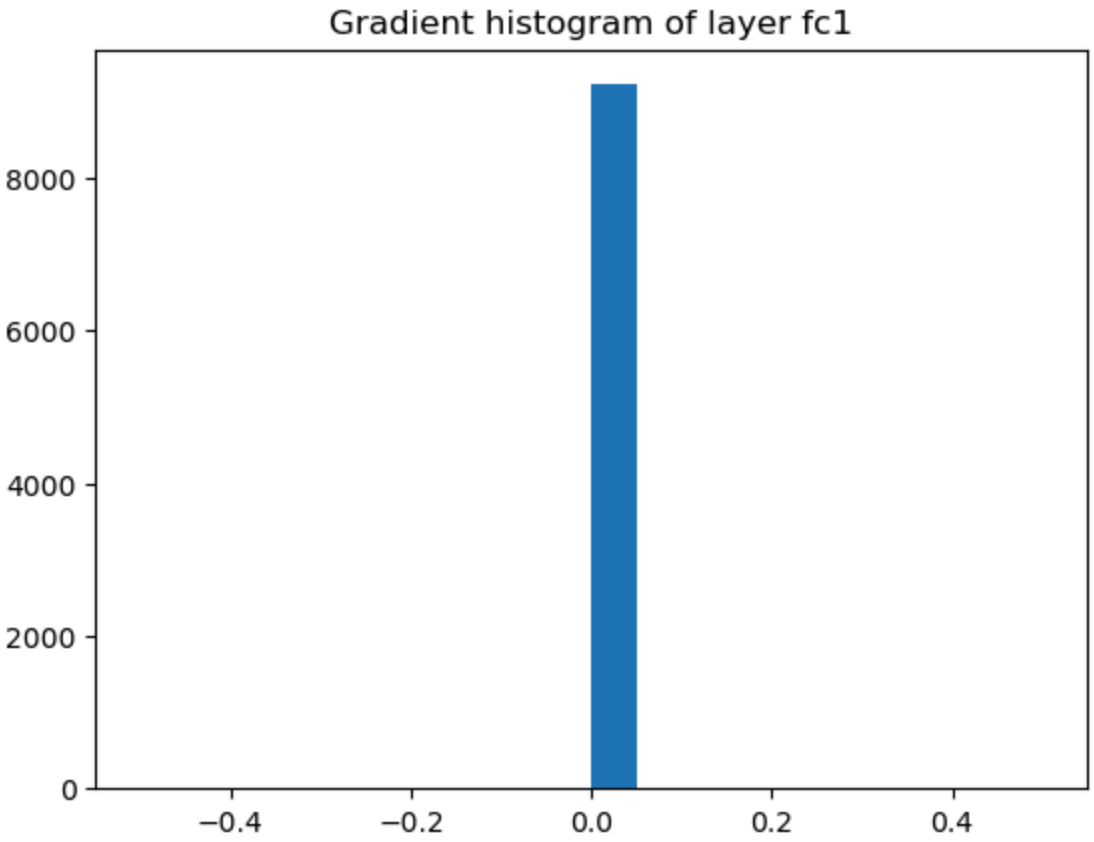
```
"""
Lab 3(c)
"""

# Set model weights to zero
for name, module in net.named_modules():
    if isinstance(module, CONV) or isinstance(module, FC):
        # Set the weight of each module to all zero
        nn.init.zeros_(module.weight)  # Your code here

# Reset gradients
net.zero_grad()

# Forward and backward pass
# Random data and target
data = torch.randn(1, 3, 32, 32).to(device)
target = torch.randn(1, 10).to(device)

# Forward pass
out = net(data)  # Your code here
# Loss computation
loss = criterion(out, target)  # Your code here
# Backward pass
loss.backward()  # Your code here

for name, module in net.named_modules():
```

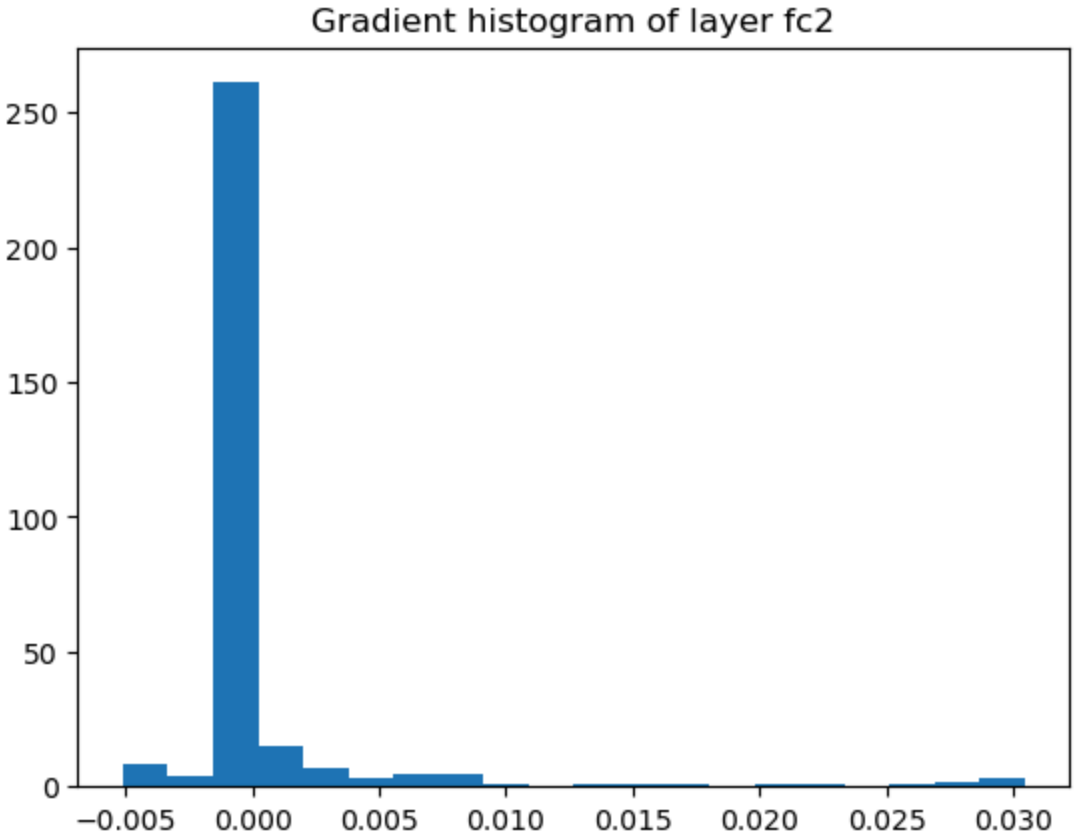
```python
    if isinstance(module, CONV) or isinstance(module, FC):
        # Get the gradient of the module as a NumPy array
        gradient = module.weight.grad.cpu().detach().numpy()  # Your code here

        # Reshape for histogram
        gradient = gradient.reshape(-1)
        _ = plt.hist(gradient, bins=20)
        plt.title("Gradient histogram of layer " + name)
        plt.show()
```






Gradient histogram of layer conv3
25000
20000
15000
10000
5000
0
−0.4
−0.2
0.0
0.2
0.4


Gradient histogram of layer fc1
8000
6000
4000
2000
0
−0.4
−0.2
0.0
0.2
0.4



> Comparing the histograms of gradients from initialized weights versus zero weights, there is a significant difference. With initialized weights, the gradients are more spread out and follow a bell-curve distribution around zero, allowing the network to learn and adjust weights effectively. However, when weights are initialized to zero, the gradient histograms show a strong concentration around zero, indicating that most gradients are either extremely small or zero. This is problematic because it prevents the network from learning effectively. Initializing a CNN model with zero weights can lead to symmetry issues during training, where all neurons in the network are updated in the same way, effectively making them identical. This results in no meaningful learning or differentiation between neurons, ultimately hindering the convergence of the model. Therefore, proper weight initialization is critical for breaking symmetry and ensuring effective gradient flow during backpropagation.