Barbara Flores

ECE 661: Homework #1
Linear Model, Back Propagation and Building a CNN

Hai "Helen" Li

ECE Department, Duke University — Spring 2025

Objectives

Homework #1 covers the contents of Lectures 02 ~ 04. This assignment includes conceptual questions on the linear model, back propagation and convolutions, as well as lab questions involving trying out LMS algorithm and building and observing a CNN model.

The lab questions in this assignment don't involve the training of deep learning models. So you are welcome to use your own computer to finish the labs. Please refer to the **NumPy/PyTorch tutorial** slides on Canvas for the environment setup on your computer.

Warning: You are asked to complete the assignment independently.

This lab has a total of **100** points plus 10 bonus points, yet your final score cannot exceed 100 points. You must submit your report in PDF format and your original codes for the lab questions through **Gradescope** before **11:55:00 pm, February 3**. You need to submit **three individual files** including:
(1) *a self-contained report in PDF format* that provides answers to all the conceptual questions and clearly demonstrates all your lab codes, results and observations (figures and explanations);
(2) *a single code file* used to produce all the results for Lab: LMS algorithms;
(3) *a Jupyter notebook file* for Lab: Simple NN;
Grading will be based solely on the pdf file. Other files will be used for plagiarism checking.

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 image features and typically have fewer parameters.

Ans: True. Convolutional layers exploit local and shift-invariant image features, which enables better performance in image recognition compared to fully-connected layers. By using small, shared filters that capture spatial information, convolution layers significantly reduce the number of parameters while maintaining the ability to detect meaningful local patterns. This approach allows neural networks to more efficiently learn and recognize complex image features with fewer computational resources.

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 the same size.

Ans: False, It is possible to maintain the same output feature map size even when changing the kernel size by appropriately adjusting the padding and stride parameters. The convolution shape rule provides mathematical formulas that allow for compensation of kernel size changes through strategic padding and stride selection, enabling consistent output dimensions across different kernel sizes.

Problem 1.3 (2 pts) Given a learning task that can be perfectly learned by a Madaline model, this model is suitable for different weight initialization.

Ans: False, Even if a learning task can be perfectly learned by a Madaline model, the model is not necessarily suitable for different weight initializations. Madaline's training process is highly sensitive to random weight initialization and selection patterns, and its error-correction rule lacks a theoretical guarantee of convergence. This means that different initial weights can lead to significantly different outcomes, making the model unreliable across varying weight initializations.

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.

Ans: True. The latency of a neural network is not always directly related to its theoretical FLOPS, as factors like memory bandwidth, parallelization efficiency, and hardware-specific optimizations can create bottlenecks. These limitations can prevent a high-FLOPS processor from achieving lower latency in practice.

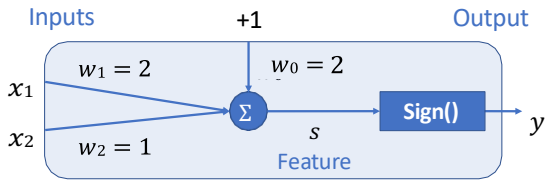
Problem 1.5 (2 pts) The overfitting models can perfectly fit the training data. Theoretically, we should increase the diversity and variability in the training data or prune some of the nodes to improve NN's generalization ability.

Ans: True. Overfitting models fit the training data too well, which reduces their generalization ability. To mitigate this, increasing the diversity and variability of the training data through techniques like data augmentation or collecting more representative samples can be effective. Additionally, reducing model complexity by pruning nodes, applying regularization, or using dropout helps prevent the model from memorizing the data instead of learning general patterns.

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 *w*'s 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?



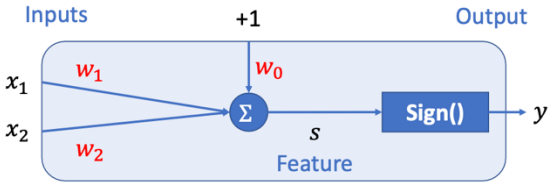
x_1	x_2	s	y
-1	-1		
-1	+1		
+1	-1		
+1	+1		

Figure 1: Problem 2.1.

x_1	x_2	s	y
-1	-1	$w_0*1 + w_1x_1 + w_2x_2 = 2*1 + 2*(-1) + 1*(-1) = -1$	-1
-1	+1	$w_0*1 + w_1x_1 + w_2x_2 = 2*1 + 2*(-1) + 1*(+1) = 1$	+1
+1	-1	$w_0*1 + w_1x_1 + w_2x_2 = 2*1 + 2*(+1) + 1*(-1) = 3$	+1
+1	+1	$w_0*1 + w_1x_1 + w_2x_2 = 2*1 + 2*(+1) + 1*(+1) = 5$	+1

The Adaline is performing the OR logic function, if we consider -1 as F and +1 as True.

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]



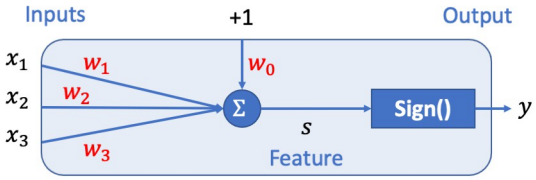
x_1	x_2	s	y
-1	-1		+1
-1	+1		+1
+1	-1		+1
+1	+1		-1

Figure 2: Problem 2.2.

The selected values for the weights are
$w_0 = 1$
$w_1 = -1$
$w_2 = -1$
These values ensure that the Adaline correctly performs the NAND logic function, as demonstrated by the calculations in the table.

x_1	x_2	s	y
-1	-1	$w_0*1 + w_1x_1 + w_2x_2 = 1*1 + (-1)*(-1) + (-1)*(-1) = 3$	+1
-1	+1	$w_0*1 + w_1x_1 + w_2x_2 = 1*1 + (-1)*(-1) + (-1)*(+1) = 1$	+1
+1	-1	$w_0*1 + w_1x_1 + w_2x_2 = 1*1 + (-1)*(+1) + (-1)*(-1) = 1$	+1
+1	+1	$w_0*1 + w_1x_1 + w_2x_2 = 1*1 + (-1)*(+1) + (-1)*(+1) = -1$	-1

Problem 2.3 (4 pts) Propose proper values for weight w_0, w_1, w_2 and w_3 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

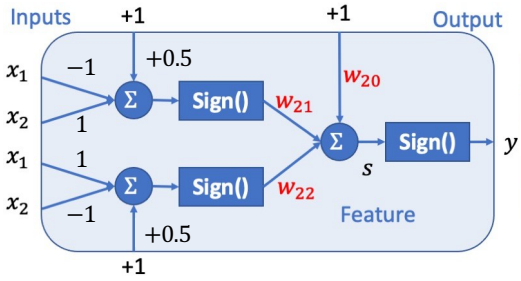
Figure 3: Problem 2.3.

The selected values for the weights are
$w_0 = 0$
$w_1 = 1$
$w_2 = 1$
$w_3 = 1$

These values ensure that the Adaline correctly implements the Majority Vote logic function, as demonstrated by the calculations in the table.

x_1	x_2	x_3	s	y
-1	-1	-1	$w_0 + w_1x_1 + w_2x_2 + w_3x_3 = (0)*1 + (1)*(-1) + (1)*(-1) + (1)*(-1) = -3$	-1
-1	-1	+1	$w_0 + w_1x_1 + w_2x_2 + w_3x_3 = (0)*1 + (1)*(-1) + (1)*(-1) + (1)*(+1) = -1$	-1
-1	+1	-1	$w_0 + w_1x_1 + w_2x_2 + w_3x_3 = (0)*1 + (1)*(-1) + (1)*(+1) + (1)*(-1) = -1$	-1
-1	+1	+1	$w_0 + w_1x_1 + w_2x_2 + w_3x_3 = (0)*1 + (1)*(-1) + (1)*(+1) + (1)*(+1) = +1$	+1
+1	-1	-1	$w_0 + w_1x_1 + w_2x_2 + w_3x_3 = (0)*1 + (1)*(+1) + (1)*(-1) + (1)*(-1) = -1$	-1
+1	-1	+1	$w_0 + w_1x_1 + w_2x_2 + w_3x_3 = (0)*1 + (1)*(+1) + (1)*(-1) + (1)*(+1) = +1$	+1
+1	+1	-1	$w_0 + w_1x_1 + w_2x_2 + w_3x_3 = (0)*1 + (1)*(+1) + (1)*(+1) + (1)*(-1) = +1$	+1
+1	+1	+1	$w_0 + w_1x_1 + w_2x_2 + w_3x_3 = (0)*1 + (1)*(+1) + (1)*(+1) + (1)*(+1) = +3$	+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

Figure 4: Problem 2.4.

The selected values for the weights are
$w_{20} = +1$
$w_{21} = -1$
$w_{22} = -1$

As:

x_1	x_2	s_1	s_2
-1	-1	(0.5)*1 + (-1)*x_1 + (+1)*x_2	(0.5)*1 + (+1)*x_1 + (-1)*x_2
-1	+1	(0.5)*1 + (-1)*x_1 + (+1)*x_2	(0.5)*1 + (+1)*x_1 + (-1)*x_2
+1	-1	(0.5)*1 + (-1)*x_1 + (+1)*x_2	(0.5)*1 + (+1)*x_1 + (-1)*x_2
+1	+1	(0.5)*1 + (-1)*x_1 + (+1)*x_2	(0.5)*1 + (+1)*x_1 + (-1)*x_2

x_1	x_2	s_1	s_2
-1	-1	(0.5)*1 + (-1)*(-1) + (+1)*(-1) = 0.5	(0.5)*1 + (+1)*(-1) + (-1)*(-1) = 0.5
-1	+1	(0.5)*1 + (-1)*(-1) + (+1)*(+1) = 2.5	(0.5)*1 + (+1)*(-1) + (-1)*(+1) = -1.5
+1	-1	(0.5)*1 + (-1)*(+1) + (+1)*(-1) = -1.5	(0.5)*1 + (+1)*(+1) + (-1)*(-1) = 2.5
+1	+1	(0.5)*1 + (-1)*(+1) + (+1)*(+1) = 0.5	(0.5)*1 + (+1)*(+1) + (-1)*(+1) = 0.5

x_1	x_2	$Sign(s_1)$	$Sign(s_2)$	s
-1	-1	Sign(0.5) = +1	Sign(0.5) = +1	W_{20}*1 + w_{21}*(Sign(s_1)) + w_{22}*(Sign(s_1))
-1	+1	Sign(2.5) = +1	Sign(-1.5) = -1	W_{20}*1 + w_{21}*(Sign(s_1)) + w_{22}*(Sign(s_1))
+1	-1	Sign(-1.5) = -1	Sign(2.5) = +1	W_{20}*1 + w_{21}*(Sign(s_1)) + w_{22}*(Sign(s_1))
+1	+1	Sign(0.5) = +1	Sign(0.5) = +1	W_{20}*1 + w_{21}*(Sign(s_1)) + w_{22}*(Sign(s_1))

x_1	x_2	$Sign(s_1)$	$Sign(s_2)$	s	y
-1	-1	Sign(0.5) = +1	Sign(0.5) = +1	(+1)*1 + (-1)*(+1) + (-1)*(+1) = -1	-1
-1	+1	Sign(2.5) = +1	Sign(-1.5) = -1	(+1)*1 + (-1)*(+1) + (-1)*(-1) = +1	+1
+1	-1	Sign(-1.5) = -1	Sign(2.5) = +1	(+1)*1 + (-1)*(-1) + (-1)*(+1) = +1	+1
+1	+1	Sign(0.5) = +1	Sign(0.5) = +1	(+1)*1 + (-1)*(+1) + (-1)*(+1) = -1	-1

3 Back Propagation (15 pts)

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

$$x_2 = \text{Sigmoid}(W_1 x_1 + b_1) \quad (1)$$
$$x_3 = W_2 x_2 + b_2 \quad (2)$$

A L1 loss function $L = \sum_{i=1}^{k} |[t]_i - [x_3]_i|$ is applied in the end, where $t \in \mathcal{R}^{k\times 1}$ 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**.

[**Hint:** Subgradients extend the concept of gradients to functions that are not differentiable everywhere, crucial in optimizing functions like the ReLU activation in neural networks, where the gradient does not exist at zero. For a convex function $f : \mathbb{R}^n \to \mathbb{R}$, a vector g is a subgradient at x if:

$$f(y) \geq f(x) + g^T(y - x)\, \forall y \in \mathbb{R}^n$$

Example: For $f(x) = |x - 1|$, $\partial f(1) = [-1, 1]$ for $x = 1$ (non-differentiable point). Any value of the subgradient can be used in backpropagation.]

First, we have that:

$$L = \sum_{i=1}^{k} |[t]_i - [x_3]_i|$$

$$x_3 = W_2 x_2 + b_2$$

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

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

$$\frac{\partial x_3}{\partial b_2} = I_k$$

Where I_k is the identity matrix of size k x k

- Calculating $\partial L/\partial W_2$

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

$$\frac{\partial L}{\partial W_2} = \left(-sign(t - x_3)\right) {x_2}^T$$

- Calculating $\partial L/\partial b_2$

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

$$\frac{\partial L}{\partial b_2} = \left(-sign(t - x_3)\right) I_k$$

$$\frac{\partial L}{\partial b_2} = -sign(t - x_3)$$

- Calculating $\partial L/\partial W_1$

We have

$$L = \sum_{i=1}^{k} |[t]_i - [x_3]_i|$$

$$x_3 = W_2 x_2 + b_2$$

$$x_2 = Sigmoid(W_1 x_1 + b_1)$$

Let's define z_2 as

$$z = W_1 x_1 + b_1$$

$$x_2 = Sigmoid(z)$$

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

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

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

We know from lecture slide 2, slides 29 and 30 that:

$$\frac{\partial x_2}{\partial z} = Sigmoid(z) \cdot \big(1 - Sigmoid(z)\big)$$

$$\frac{\partial z}{\partial W_1} = {x_1}^T$$

$$\frac{\partial L}{\partial W_1} = -sign(t - x_3) \cdot {W_2}^T \odot Sigmoid(z) \cdot \big(1 - Sigmoid(z)\big) \cdot {x_1}^T$$

$$\frac{\partial L}{\partial W_1} = -sign(t - x_3) \cdot {W_2}^T \odot Sigmoid(W_1 x_1 + b_1) \cdot \big(1 - Sigmoid(W_1 x_1 + b_1)\big) \cdot {x_1}^T$$

- Calculating $\partial L/\partial b_1$

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

$$\frac{\partial z}{\partial b_1} = I_m$$

$$\frac{\partial L}{\partial b_1} = -sign(t - x_3) \cdot {W_2}^T \odot Sigmoid(W_1 x_1 + b_1) \cdot \big(1 - Sigmoid(W_1 x_1 + b_1)\big) \cdot I_m$$

Finally, we have:

$$\frac{\partial L}{\partial W_1} = -sign(t - x_3) \cdot {W_2}^T \odot Sigmoid(W_1 x_1 + b_1) \cdot \big(1 - Sigmoid(W_1 x_1 + b_1)\big) \cdot {x_1}^T$$

$$\frac{\partial L}{\partial W_2} = \big(-sign(t - x_3)\big)\, {x_2}^T$$

$$\frac{\partial L}{\partial b_1} = -sign(t - x_3) \cdot {W_2}^T \odot Sigmoid(W_1 x_1 + b_1) \cdot \big(1 - Sigmoid(W_1 x_1 + b_1)\big)$$

$$\frac{\partial L}{\partial b_2} = -sign(t - x_3)$$

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, $\frac{\partial L}{\partial W_1}$, $\frac{\partial L}{\partial W_2}$, $\frac{\partial L}{\partial b_1}$, $\frac{\partial L}{\partial b_2}$

- Calculating L

$$x_2 = ReLU(W_1 x_1 + b_1)$$

$$x_2 = Relu\left(\begin{bmatrix} 0 & 1 & 1 \\ -2 & 2 & -1 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \end{bmatrix}\right)$$

$$x_2 = Relu\left(\begin{bmatrix} 1 \\ 3 \end{bmatrix}\right)$$

$$x_2 = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$$

$$x_3 = W_2 x_2 + b_2$$

$$x_3 = \begin{bmatrix} 0 & 2 \\ 1 & 1 \end{bmatrix}\begin{bmatrix} 1 \\ 3 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$x_3 = \begin{bmatrix} 7 \\ 5 \end{bmatrix}$$

$$L = \sum_{i=1}^{k} |[t]_i - [x_3]_i|$$

$$L = \sum_{i=1}^{k} \left|\begin{bmatrix} 1 \\ 2 \end{bmatrix} - \begin{bmatrix} 7 \\ 5 \end{bmatrix}\right|$$

$$L = \sum_{i=1}^{k} \begin{bmatrix} |-6| \\ |-3| \end{bmatrix}$$

$$L = 6 + 3$$

$$L = 9$$

- Calculating $\partial L/\partial W_2$

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

$$\frac{\partial L}{\partial W_2} = \left(-sign(t - x_3)\right) {x_2}^T$$

$$\frac{\partial L}{\partial W_2} = \left(-sign\left(\begin{bmatrix}1\\2\end{bmatrix} - \begin{bmatrix}7\\5\end{bmatrix}\right)\right) \begin{bmatrix}1 & 3\end{bmatrix}$$

$$\frac{\partial L}{\partial W_2} = \begin{bmatrix}1\\1\end{bmatrix} \begin{bmatrix}1 & 3\end{bmatrix}$$

$$\frac{\partial L}{\partial W_2} = \begin{bmatrix}1 & 3\\1 & 3\end{bmatrix}$$

- Calculating $\partial L/\partial b_2$

$$\frac{\partial L}{\partial b_2} = \left(-sign(t - x_3)\right)$$

$$\frac{\partial L}{\partial b_2} = \left(-sign\left(\begin{bmatrix}1\\2\end{bmatrix} - \begin{bmatrix}7\\5\end{bmatrix}\right)\right)$$

$$\frac{\partial L}{\partial b_2} = \begin{bmatrix}1\\1\end{bmatrix}$$

- Calculating $\partial L/\partial W_1$

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

$\partial x2/\partial z$ changes because we are now using ReLU instead of sigmoid, so we have:

$$x_2 = Relu(z)$$

$$\frac{\partial x_2}{\partial z} = \begin{cases} 1 & if\ z > 0 \\ 0 & if\ z \leq 0 \end{cases}$$

$$z = (W_1 x_1 + b_1) = \begin{bmatrix}0 & 1 & 1\\-2 & 2 & -1\end{bmatrix} \begin{bmatrix}1\\1\\-1\end{bmatrix} + \begin{bmatrix}1\\2\end{bmatrix} = \begin{bmatrix}1\\3\end{bmatrix} > 0$$

$$\frac{\partial L}{\partial W_1} = {W_2}^T(-sign(t - x_3)) \odot \begin{bmatrix}1\\1\end{bmatrix} {x_1}^T$$

$$\frac{\partial L}{\partial W_1} = \begin{bmatrix}0 & 2\\1 & 1\end{bmatrix}^T \left(-sign\left(\begin{bmatrix}1\\2\end{bmatrix} - \begin{bmatrix}7\\5\end{bmatrix}\right)\right) \odot \begin{bmatrix}1\\1\end{bmatrix} \begin{bmatrix}1 & 1 & -1\end{bmatrix}$$

$$\frac{\partial L}{\partial W_1} = \begin{bmatrix}0 & 1\\2 & 1\end{bmatrix} \cdot \begin{bmatrix}1\\1\end{bmatrix} \cdot \begin{bmatrix}1 & 1 & -1\end{bmatrix}$$

$$\frac{\partial L}{\partial W_1} = \begin{bmatrix} 1 & 1 & -1 \\ 3 & 3 & -3 \end{bmatrix}$$

- Calculating $\partial L/\partial b_1$

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

$$\frac{\partial z}{\partial b_1} = I_m$$

$$\frac{\partial L}{\partial b_1} = -sign(t - x_3) \cdot {W_2}^T \odot \begin{bmatrix} 1 \\ 1 \end{bmatrix} \cdot I_m$$

$$\frac{\partial L}{\partial b_1} = {W_2}^T(-sign(t - x_3)) \odot \begin{bmatrix} 1 \\ 1 \end{bmatrix} I_2$$

$$\frac{\partial L}{\partial b_1} = \begin{bmatrix} 0 & 1 \\ 2 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$\frac{\partial L}{\partial b_1} = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$$

Finally, we have:

$$L = 9$$

$$\frac{\partial L}{\partial W_1} = \begin{bmatrix} 1 & 1 & -1 \\ 3 & 3 & -3 \end{bmatrix}$$

$$\frac{\partial L}{\partial W_2} = \begin{bmatrix} 1 & 3 \\ 1 & 3 \end{bmatrix}$$

$$\frac{\partial L}{\partial b_1} = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$$

$$\frac{\partial L}{\partial b_2} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

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.

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

We have:

Strides = S = 1

K = 3

Padding = P = $\frac{K-1}{2} = 1$

With padding of 1, we add zeros around the matrix, which results in the following padded input matrix to be used in the convolution operation:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & -1 & -1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & -1 & -1 & -1 & -1 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & -1 & -1 & -1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Kernel

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

Example top left corner

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

Convolution result of the top-left corner:

(0)*(0) + (-1/2)*(0) + (0)*(0) + (-1/2)*(0) + (1)*(0) + (-1/2)*(0) + (0)*(0) + (-1/2)*(0) + (0)*(-1) =0

Final Output (you can see the math here:)

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

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

Original Image

Filtered Image with Kernel

A noticeable effect of applying the kernel is that, in general, it reverses the sign of the numbers in the resulting matrix. For example, in the given matrix, we initially had a diamond-like pattern of -1s on the left side and +1s on the right side. After applying the kernel, this pattern tends to invert. However, the edges become more pronounced in terms of contrast.
This effect is even more evident in the image, where the colors appear to be inverted—dark regions tend to become bright, and bright regions tend to become dark.

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 Gradescope.

Lab 1 (15 pts)

To start with, please download the `dataset.mat` file from Canvas and load it into NumPy arrays[a]. 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 corresponds 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^*?

(b) (4pt) Now consider that you can only train with 1 pair of data points and target each time. In such a case, the LMS algorithm should be used to find the optimal weight. Please initialize the weight vector as $W^0 = [0.53, 0.20, 0.10]^T$, 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.

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

(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. Then try further enlarge the learning rate to $r = 1$ and observe how the MSE changes. Based on these observations, comments on how the 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.)

[a] You may refer to `https://docs.scipy.org/doc/scipy/reference/generated/scipy.io.loadmat.html` for loading matrices in .mat file into NumPy arrays.

[b] Please refer to `https://jakevdp.github.io/PythonDataScienceHandbook/04.12-three-dimensional-plotting.html` for plotting 3D plots with Matplotlib.

a) Ans

We have Least square (Wiener) solution for linear model:

$$W^* = (X^T X)^{-1} X^T D$$

After applying the Least Square (Wiener) solution for the linear model, the optimal weight W* is

$$W^* = \begin{bmatrix} 1.0006781 \\ 1.00061145 \\ -2.00031968 \end{bmatrix}$$

Also, we have that individually, the MSE is

$$MSE\ L_k = \frac{1}{2}(d_k - s_k)^2$$

In total, we have that:

$$MSE = \frac{1}{2N}\sum_{k=1}^{N}(d_k - s_k)^2$$

In vectorial form:

$$MSE = \frac{1}{2N}\|D - S\|^2$$

$$MSE = \frac{1}{2N}\|D - XW\|^2$$

$$MSE = \frac{1}{2N}(D - XW)^T(D - XW)$$

The MSE loss of the whole dataset when the weight is set to W* is:

MSE = 0.00005040

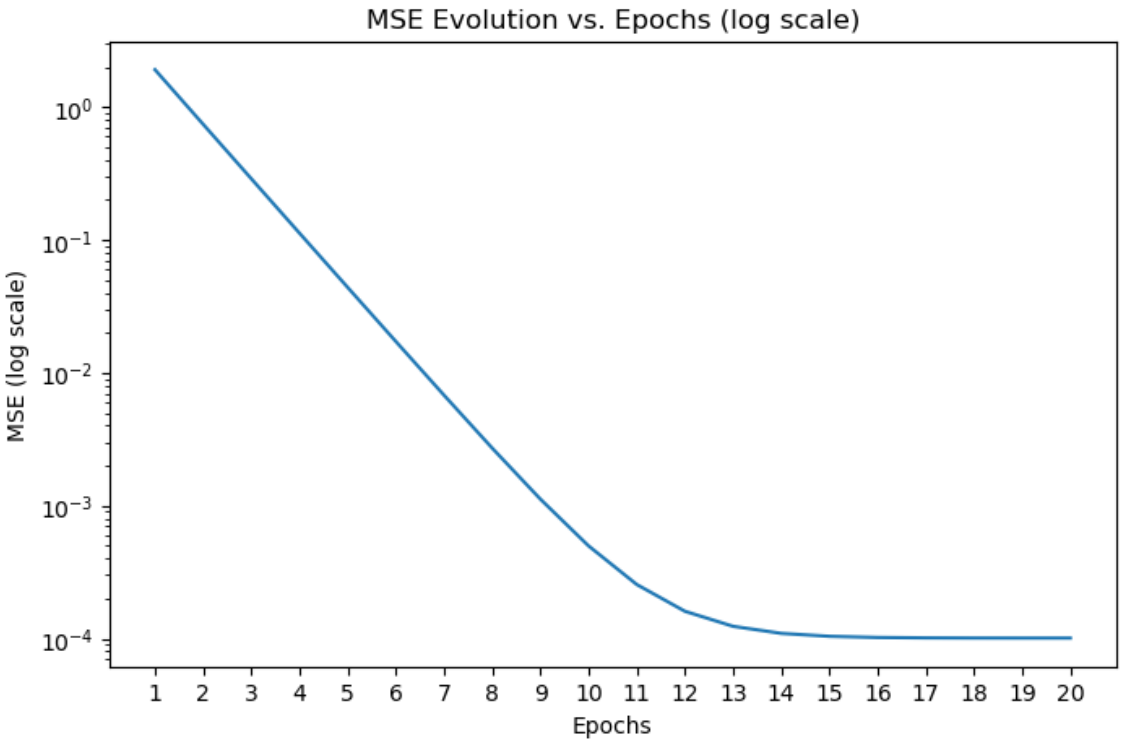
Note: I am using the formula divided by 2, following the slide on page 19 of Lecture 2. Using the classic formula without the division by 2, the result is: MSE = 0.000100799

b) Ans

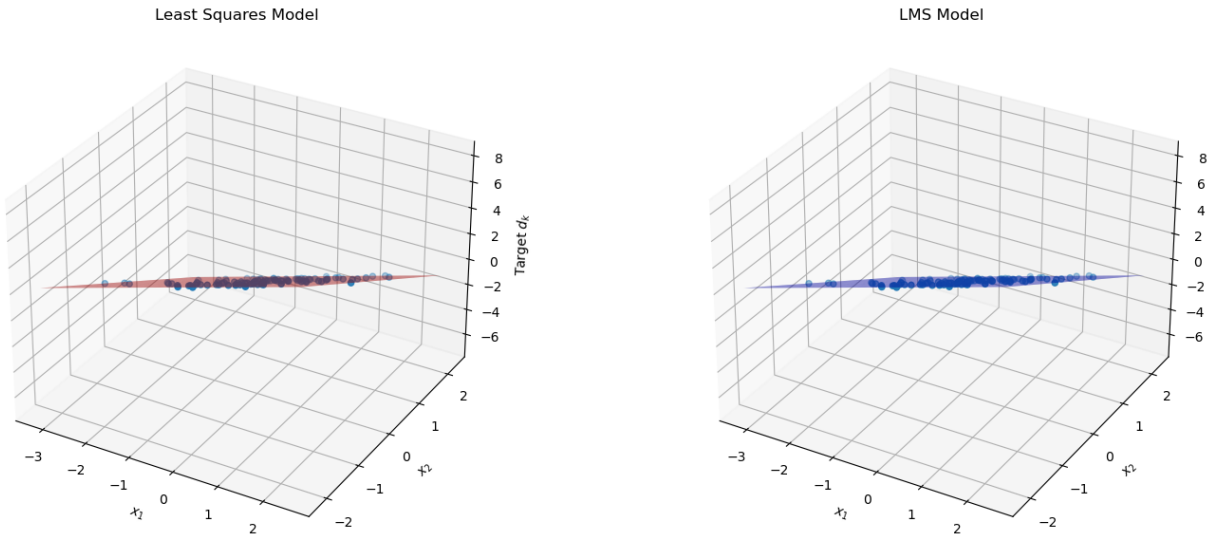
After 20 epochs, the weight vector W, obtained is:

$$W_{epoch\ 20} = \begin{bmatrix} 1.00070598 \\ 1.00059491 \\ -2.00033694 \end{bmatrix}$$

Additionally, we can observe the evolution of the MSE loss in log scale over the 20 epochs. The plot below illustrates how the MSE loss decreases as the algorithm converges, demonstrating the effectiveness of the LMS algorithm in minimizing the error over time.



c) Ans

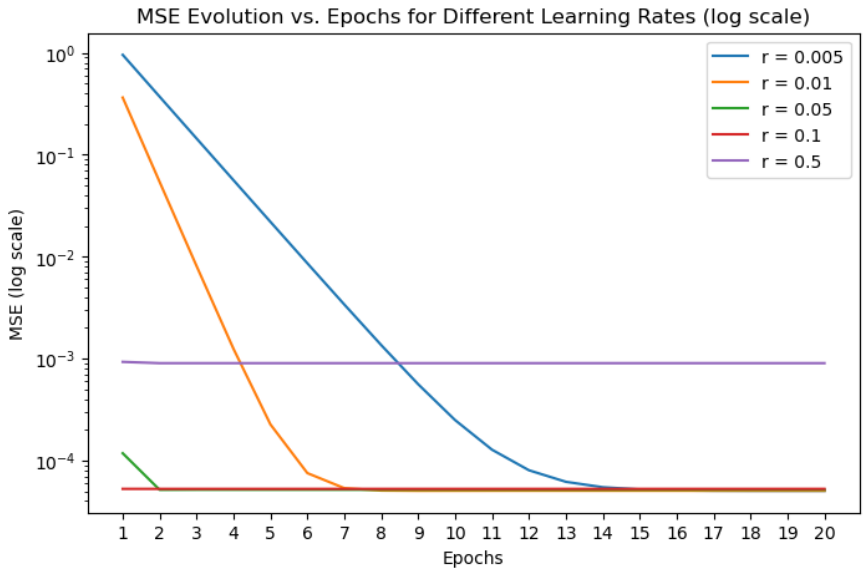


The figure presents the linear models obtained using the least squares method (left) and the LMS algorithm (right). Both models generate similar regression planes that align with the data distribution, suggesting that the relationship between the variables is indeed linear. Since the models closely follow the pattern of the data points, it can be concluded that the linear models fit the data well.

d) Ans

The process from (b) was repeated using learning rates r = 0.01, 0.05, 0.1 and 0.5, along with the original result from r=0.005. The plot shows the MSE evolution over 20 epochs on a log scale. Smaller rates (0.005,0.010.005, 0.010.005,0.01) converge steadily, while medium rates (0.05,0.10.05, 0.10.05,0.1) reach lower MSE faster. However, r=0.5 fails to converge, likely due to instability.

We can observe a trade-off: a larger learning rate leads to faster convergence, but if too large, it can cause instability and prevent convergence, as seen with r=0.5. This problem becomes even more pronounced when **r = 1**, as seen in the following case.



What happens when we try to use a larger learning rate, r=1?

When using **r = 1**, we obtain the results shown below.

Epoch 1: MSE = 14657955294674538496.00000000
Epoch 2: MSE = 95125471798345421701343347522297397248.00000000
Epoch 3: MSE = 617333851573464326395881132062563291706847785467353497600.00000000
….
….
Epoch 16: MSE =
22357466147643364088565502759731845939585806778862764503263437008547488092398145047064967326401870372560913352450470669876822260100784038389331669722151965302789032195665523547264271006186945243815789801252374851135575853453869163608434391074248270855347413774462826293657428973630299452076268502122496.00000000
Epoch 17: MSE = inf
Epoch 18: MSE = inf
Epoch 19: MSE = inf
Epoch 20: MSE = inf

The MSE increases exponentially with each epoch until it overflows, resulting in numerical instability. Eventually, the values become too large to be represented, leading to infinite MSE and complete failure of the learning process.

Finally, the learning rate significantly impacts both the speed and quality of the learning process. A small learning rate (e.g., 0.005, 0.01) ensures stability and steady convergence, but it takes longer to reach an optimal solution. Medium learning rates (e.g., 0.05, 0.1) accelerate convergence while maintaining stability, striking a balance between speed and accuracy. However, a high learning rate (e.g., 0.5 or 1) can introduce instability, causing the MSE to increase uncontrollably instead of decreasing.

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 Gradescope.

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.

Lab 2 (35 points)

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 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 `SimpleNN` class into the report PDF.

(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.

Layer	Input shape	Output shape	Weight shape	# Param	# MAC
Conv 1					
Conv 2					
Conv 3					
FC1					
FC2					

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.

(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.

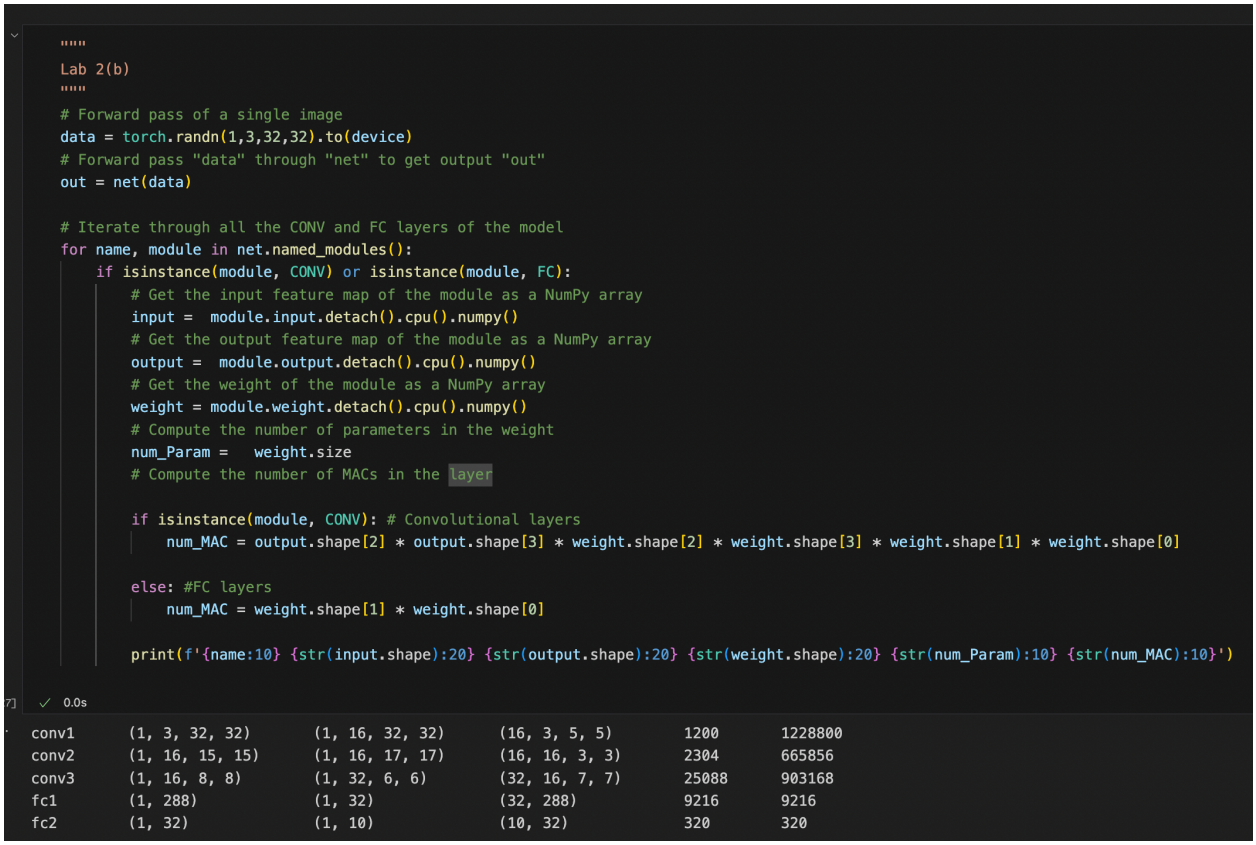
(c) (5pt) In code block 7, finish the code to set all the weights to 0. Perform forward and backward pass again to get the gradients, and plot the histogram of weight elements' gradients in each one of the 5 convolutional and fully-connected layers. Comparing with the histograms you got in (b), are there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

a)

```
    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)
            self.conv2 = CONV(in_channels=16, out_channels=16, kernel_size=3, stride=1, padding=2)
            self.conv3 = CONV(in_channels=16, out_channels=32, kernel_size=7, stride=1, padding=2)

            #in_features = num_channels * height * width
            self.fc1   = FC(in_features = (32)*(3)*(3), out_features=32)
            self.fc2   = FC(in_features= 32, out_features=10)

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

    # Check output shape
    assert(out.detach().cpu().numpy().shape == (5,10))
    print("Forward pass successful")
✓ 0.0s
Run on CPU...
Forward pass successful
```

```
"""
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)
        self.conv2 = CONV(in_channels=16, out_channels=16, kernel_size=3, stride=1, padding=2)
        self.conv3 = CONV(in_channels=16, out_channels=32, kernel_size=7, stride=1, padding=2)

        #in_features = num_channels * height * width
        self.fc1   = FC(in_features = (32)*(3)*(3), out_features=32)
        self.fc2   = FC(in_features= 32, out_features=10)

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

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

b)



```
"""
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)

# 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()
        # Get the output feature map of the module as a NumPy array
        output =  module.output.detach().cpu().numpy()
        # Get the weight of the module as a NumPy array
        weight = module.weight.detach().cpu().numpy()
        # Compute the number of parameters in the weight
        num_Param =   weight.size
        # Compute the number of MACs in the layer

        if isinstance(module, CONV): # Convolutional layers
            num_MAC = output.shape[2] * output.shape[3] * weight.shape[2] * weight.shape[3] * weight.shape[1] * weight.shape[0]

        else: #FC layers
            num_MAC = weight.shape[1] * weight.shape[0]

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

```
✓ 0.0s
conv1      (1, 3, 32, 32)       (1, 16, 32, 32)      (16, 3, 5, 5)        1200       1228800
conv2      (1, 16, 15, 15)      (1, 16, 17, 17)      (16, 16, 3, 3)       2304       665856
conv3      (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
```

"""
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)

# 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()
        # Get the output feature map of the module as a NumPy array
        output = module.output.detach().cpu().numpy()
        # Get the weight of the module as a NumPy array
        weight = module.weight.detach().cpu().numpy()
        # Compute the number of parameters in the weight
        num_Param =   weight.size
        # Compute the number of MACs in the layer

        if isinstance(module, CONV): # Convolutional layers
            num_MAC = output.shape[2] * output.shape[3] * weight.shape[2] * weight.shape[3] * weight.shape[1] * weight.shape[0]

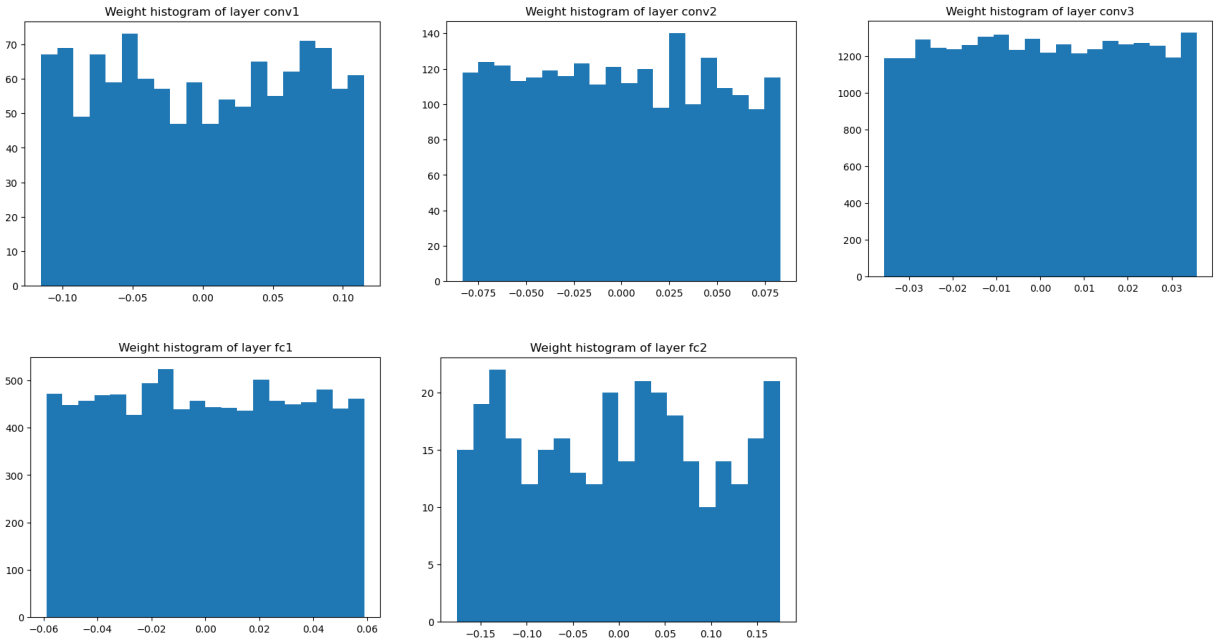
        else: #FC layers
            num_MAC = weight.shape[1] * weight.shape[0]

        print(f'{name:10} {str(input.shape):20} {str(output.shape):20} {str(weight.shape):20} {str(num_Param):10} {str(num_MAC):10}')

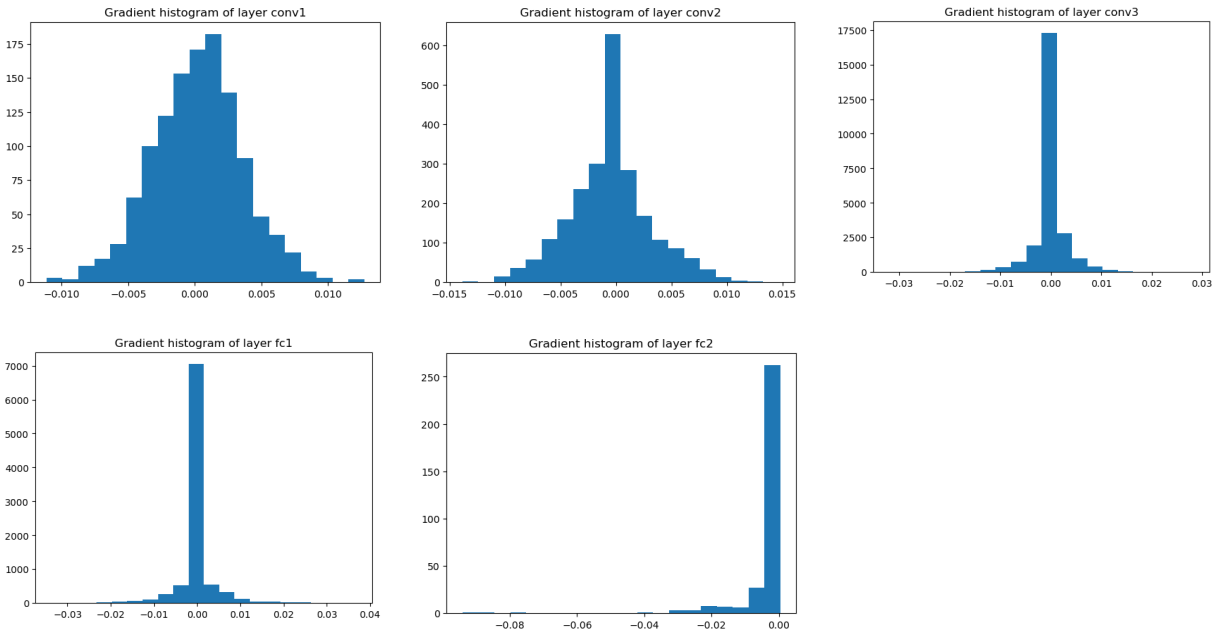
Layer	Input shape	Output Shape	Weight shape	# Param	# MAC
Conv 1	(1, 3, 32, 32)	(1, 16, 32, 32)	(16, 3, 5, 5)	1200	1228800
Conv 2	(1, 16, 15, 15)	(1, 16, 17, 17)	(16, 16, 3, 3)	2304	665856
Conv 3	(1, 16, 8, 8)	(1, 32, 6, 6)	(32, 16, 7, 7)	25088	903168
FC1	(1, 288)	(1, 32)	(32, 288)	9216	9216
FC2	(1,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

Lab 3

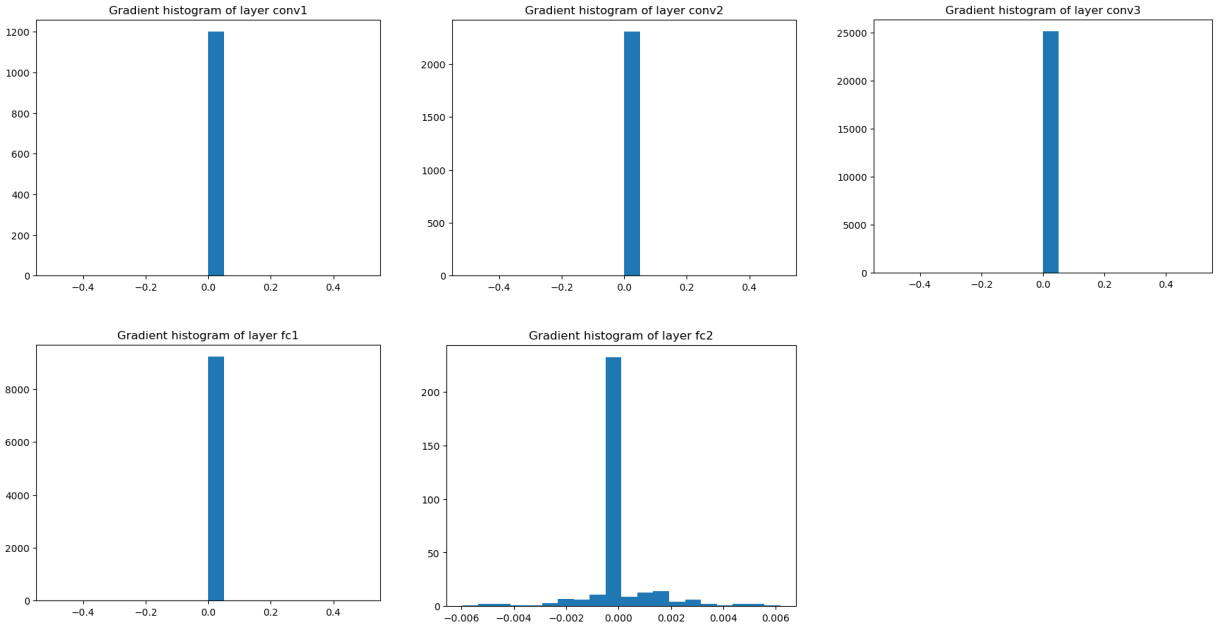
a)



b)



c)



When comparing the histograms obtained with zero-initialized (c) weights versus randomly initialized weights(b), it is evident that the distributions of weights and gradients in the zero-initialized case are much more concentrated, with sharper peaks, whereas the randomly initialized case shows a broader and more dispersed distribution. This happens because when all weights start at zero, the gradients computed during backpropagation are identical for all neurons, preventing them from learning different features. As a result, training is affected by slower convergence and the risk of some neurons remaining inactive, reducing the network's ability to learn complex patterns.