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.**

# 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.

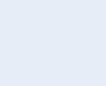
**Ans: True.** Overfitting models fit the training data too well, which reduces their generalization ability. To mitigate this, increasing the diversity and variability of the training data through techniques like data augmentation or collecting more representative samples can be effective. Additionally,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

# 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](#_bookmark0), 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?

Inputs +1



𝑤1 = 2

Σ

𝑤2 = 1

𝑠

Feature

**Sign()**

𝑤0 = 2

𝑥1

𝑤0 = −2

𝑥2

Output

𝑦

4

|  |  |  |  |
| --- | --- | --- | --- |
| 𝑥1 | 𝑥2 | 𝑠 | 𝑦 |
| -1 | -1 | | |
| -1 | +1 | | |
| +1 | -1 | | |
| +1 | +1 | | |

Figure 1: Problem 2.1.

|  |  |  |  |
| --- | --- | --- | --- |
| *x1* | *x2* | *s* | *y* |
| -1 | -1 | w0\*1 + w1x1 + w2x2 = 2\*1 + 2\*(-1) + 1\*(-1) = -1 | -1 |
| -1 | +1 | w0\*1 + w1x1 + w2x2 = 2\*1 + 2\*(-1) + 1\*(+1) = 1 | +1 |
| +1 | -1 | w0\*1 + w1x1 + w2x2 = 2\*1 + 2\*(+1) + 1\*(-1) = 3 | +1 |
| +1 | +1 | w0\*1 + w1x1 + w2x2 = 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](#_bookmark1) 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>]

A diagram of a function

Description automatically generated

The selected values for the weights are

w0 = 1

w1 = -1

w2 = -1

These values ensure that the Adaline correctly performs the NAND logic function, as demonstrated by the calculations in the table.

|  |  |  |  |
| --- | --- | --- | --- |
| *x1* | *x2* | *s* | *y* |
| -1 | -1 | w0\*1 + w1x1 + w2x2 = 1\*1 + (-1)\*(-1) + (-1)\*(-1) = 3 | +1 |
| -1 | +1 | w0\*1 + w1x1 + w2x2 = 1\*1 + (-1)\*(-1) + (-1)\*(+1) = 1 | +1 |
| +1 | -1 | w0\*1 + w1x1 + w2x2 = 1\*1 + (-1)\*(+1) + (-1)\*(-1) = 1 | +1 |
| +1 | +1 | w0\*1 + w1x1 + w2x2 = 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](#_bookmark2) 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>]

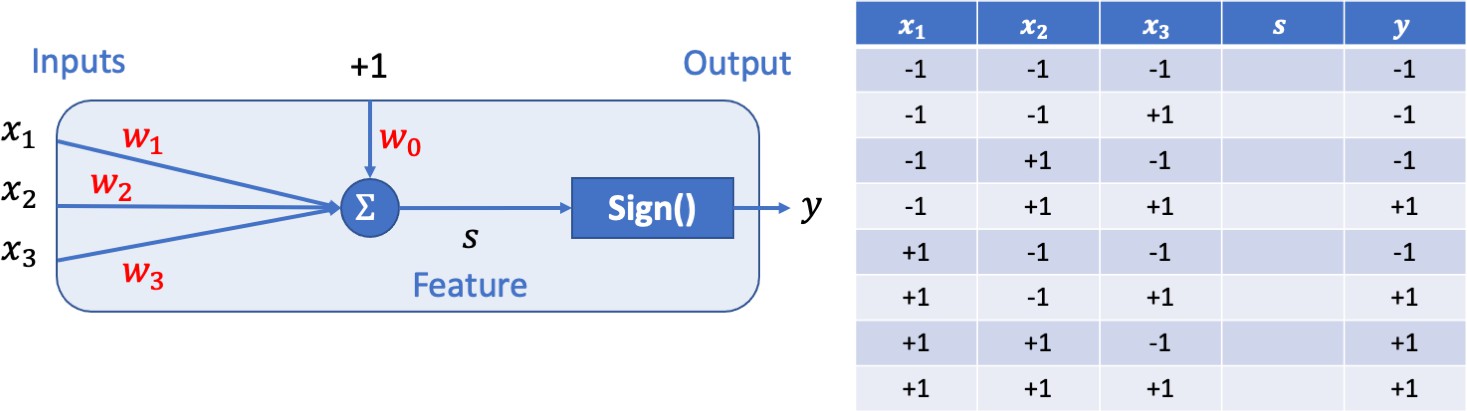


Figure 3: Problem 2.3.

The selected values for the weights are

w0 = 0

w1 = 1

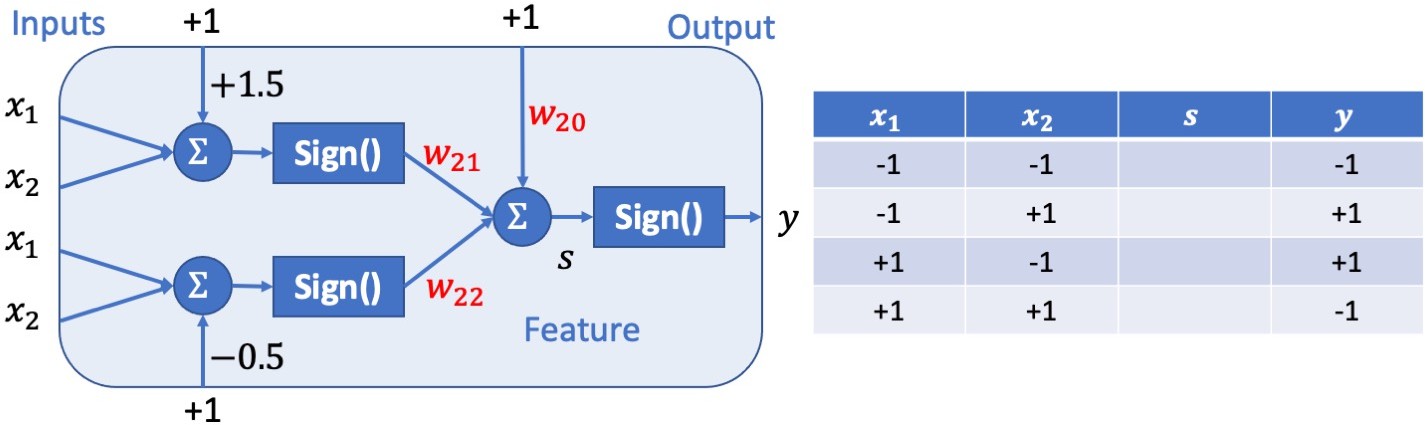
w2 = 1

w3 = 1

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *x1* | *x2* | *x3* | *s* | *y* |
| -1 | -1 | -1 | w0 + w1x1 + w2x2 + w3x3 = (0)\*1 + (1)\*(-1) + (1)\*(-1) + (1)\*(-1) = -3 | -1 |
| -1 | -1 | +1 | w0 + w1x1 + w2x2 + w3x3 = (0)\*1 + (1)\*(-1) + (1)\*(-1) + (1)\*(+1) = -1 | -1 |
| -1 | +1 | -1 | w0 + w1x1 + w2x2 + w3x3 = (0)\*1 + (1)\*(-1) + (1)\*(+1) + (1)\*(-1) = -1 | -1 |
| -1 | +1 | +1 | w0 + w1x1 + w2x2 + w3x3 = (0)\*1 + (1)\*(-1) + (1)\*(+1) + (1)\*(+1) = +1 | +1 |
| +1 | -1 | -1 | w0 + w1x1 + w2x2 + w3x3 = (0)\*1 + (1)\*(+1) + (1)\*(-1) + (1)\*(-1) = -1 | -1 |
| +1 | -1 | +1 | w0 + w1x1 + w2x2 + w3x3 = (0)\*1 + (1)\*(+1) + (1)\*(-1) + (1)\*(+1) = +1 | +1 |
| +1 | +1 | -1 | w0 + w1x1 + w2x2 + w3x3 = (0)\*1 + (1)\*(+1) + (1)\*(+1) + (1)\*(-1) = +1 | +1 |
| +1 | +1 | +1 | w0 + w1x1 + w2x2 + w3x3 = (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](#_bookmark3) 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.



−1

+0.5

1

1

−1

+0.5

Figure 4: Problem 2.4.

The selected values for the weights are

w20 = +1

w21 = -1

w22 = -1

As:

|  |  |  |  |
| --- | --- | --- | --- |
| *x1* | *x2* | *s1* | *s2* |
| -1 | -1 | (0.5)\*1 + (-1)\*x1 + (+1)\*x2 | (0.5)\*1 + (+1)\*x1 + (-1)\*x2 |
| -1 | +1 | (0.5)\*1 + (-1)\*x1 + (+1)\*x2 | (0.5)\*1 + (+1)\*x1 + (-1)\*x2 |
| +1 | -1 | (0.5)\*1 + (-1)\*x1 + (+1)\*x2 | (0.5)\*1 + (+1)\*x1 + (-1)\*x2 |
| +1 | +1 | (0.5)\*1 + (-1)\*x1 + (+1)\*x2 | (0.5)\*1 + (+1)\*x1 + (-1)\*x2 |

|  |  |  |  |
| --- | --- | --- | --- |
| *x1* | *x2* | *s1* | *s2* |
| -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 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *x1* | *x2* | *Sign(s1 )* | *Sign(s2 )* | *s* |
| -1 | -1 | Sign(0.5) = +1 | Sign(0.5) = +1 | W20\*1 + w21\*(Sign(s1 ))+ w22\*(Sign(s1 )) |
| -1 | +1 | Sign(2.5) = +1 | Sign(-1.5) = -1 | W20\*1 + w21\*(Sign(s1 ))+ w22\*(Sign(s1 )) |
| +1 | -1 | Sign(-1.5) = -1 | Sign(2.5) = +1 | W20\*1 + w21\*(Sign(s1 ))+ w22\*(Sign(s1 )) |
| +1 | +1 | Sign(0.5) = +1 | Sign(0.5) = +1 | W20\*1 + w21\*(Sign(s1 ))+ w22\*(Sign(s1 )) |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *x1* | *x2* | *Sign(s1 )* | *Sign(s2 )* | *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 |

1. **Back Propagation (15 pts)**

# A math equations and formulas Description automatically generated with medium confidence

# First, we have that:

# Where Ik is the identity matrix of size k x k

# Calculating ∂L/∂W2

# Calculating ∂L/∂b2

# Calculating ∂L/∂W1

# We have

# Let’s define z2 as

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

# Calculating ∂L/∂b1

# Finally, we have:

# A black and white text with black numbers Description automatically generated

# Calculating L

# Calculating ∂L/∂W2

# Calculating ∂L/∂b2

# Calculating ∂L/∂W1

# ∂x2/∂z changes because we are now using ReLU instead of sigmoid, so we have:

# Calculating ∂L/∂b1

# Finally, we have:

# A black and white text Description automatically generated

We have:

Strides = S = 1

K = 3

Padding = P = = 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:

# Kernel

# Example top left corner

# 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:)](https://docs.google.com/spreadsheets/d/1sH-CrE1iY1uzaZZMwrovozI65bm8eRyzhmtsrr2F3lo/edit?usp=sharing)

# 

# 

|  |  |
| --- | --- |
| Original ImageA person standing on a dock over water  Description automatically generated | Filtered Image with KernelA long shot of a dock  Description automatically generated |

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.

**A screenshot of a document

Description automatically generated**

# Ans

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

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

# 

# Also, we have that individually, the MSE is

# In total, we have that:

# In vectorial form:

# 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

# Ans

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

# 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.

**A graph of a graph

Description automatically generated with medium confidence**

1. Ans

**A graph of a graph of a graph

Description automatically generated with medium confidence**

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.

# 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.

**A graph of different colored lines

Description automatically generated**

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.

A paper with text and numbers

Description automatically generated

a)

A screen shot of a computer program

Description automatically generated

"""

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)

A computer screen shot of a program code

Description automatically generated

"""

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)

|  |  |  |
| --- | --- | --- |
| A graph of a weight  Description automatically generated with medium confidence | A graph of a weight  Description automatically generated | A graph of a weight  Description automatically generated |
| A graph of a weight  Description automatically generated | A graph of a weight  Description automatically generated with medium confidence |  |

b)

|  |  |  |
| --- | --- | --- |
| A graph of a blue pyramid  Description automatically generated with medium confidence | A graph of a blue pyramid  Description automatically generated with medium confidence | A graph of a blue column  Description automatically generated |
| A graph of a blue column  Description automatically generated | A graph with blue and white bars  Description automatically generated |  |

c)

|  |  |  |
| --- | --- | --- |
| A graph of a blue bar  Description automatically generated | A graph of a blue bar  Description automatically generated | A graph with a blue bar  Description automatically generated |
| A graph with a blue bar  Description automatically generated | A graph of a graph  Description automatically generated |  |

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.