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

Frankie Willard

September 16, 2022

1 True/False Questions (10 pts)

Problem 1.1 (2 pts) Even if there exist a set of weight for a Madaline model that can perfectly fulfill a learning task, there's no guarantee that this set of weight can be found if we train the Madaline model from a random initialization with the error correction (MD-II) rule.

True. As explained in Lecture 2 Slide 24, a drawback of the error correction rule is that there is no convergence guarantee, as the stochasticity of initialized values and selection of Adalines, as well as the challenge of knowing when it is done training. Thus, with bad initialization values and selection of Adalines, it is possible for the Madaline to never perfectly model a "trivially" fulfillable task.

Problem 1.2 (2 pts) The latency of a neural network measured on a specific processor is not always positively related to its theoretical FLOPS.

True. It is often used as a representative number for neural network latency due to its ease of calculation, but it is only one metric, that has a few flaws. Firstly, observed FLOPs can be different from theoretical FLOPS. Secondly, different types of FLOPS can have different times on different processors (TPUs, GPUs) that may be built to work better with different types of operations.

Problem 1.3 (2 pts) Consider activation function $f(x) = \log(1 + \exp(x))$. The gradient of this function only vanish when $x \to -\infty$, so it is easier to train compared to sigmoid, whose gradients vanish when $x \to$ both $-\infty$ and $+\infty$.

True. The sigmoid function is not used for training, but rather as a final activation to map from the continuous space to the 0-1 space. This is helpful in applications in which we aim to predict some indicator function or classify data in the 0-1 space, but the saturation of values in both the positive and negative direction makes for worse training. Meanwhile, the given log function, shares similiarities to the ReLU activation function with the saturation for negative values but allowing positive values to grow. This allows for the network to better determine the significance of the activation on data, or rather "stronger" features in the data in relation to the neuron. The activation function given is quite similar to ELU, and I would expect it to share some similarities in the benefits of exponentiation and saturation of only negative values, as well as the computational inefficiency required (although not relevant to ease of training but rather time to train)

Problem 1.4 (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.

True. Fully-connected layers learn the relationship between a given feature and the output variable, however, if you shift each feature (e.g. augment an image), it will not be able to recognize it. Meanwhile, convolution takes advantage of spatial information, as it is considers where a pixel is in relation to the pixels around it, just as we do as humans with our visual recognition. This is a much better way to learn to recognize objects, as they may appear in different locations and orientations, such that the input features (pixels) for the same object will be different.

Problem 1.5 (2 pts) Consider two convolutional layers stacked together with a ReLU non-linearity in the middle. The first one has kernel size 5 and stride 2 and the second one has kernel size 3 stride 1. The combined receptive field of the stacked layers is 9.

True. An easy way to explain receptive field is the size input for a model required to produce a 1x1 output. I will be discussing this in terms of the 2D case (although the logic applies regardless). Looking backwards, our second layer will be a convolution of kernel size 3. Thus, we will need an output of size 3x3 from the first layer in order to match the kernel size (we assume no padding in 1.5 as it is not mentioned) so we can perform a convolution in the second layer. In order to produce an output of size 3x3 with the first layer, we must perform 3 convolutions. The first layer has kernel size 5 and stride 2, such that we require size 5x5 to perform one convolution, 7x7 to perform two convolutions (strided over 2 such that it includes 3 from first convolution and 2 from data next to it), and 9x9 to perform three convolutions. Thus, we have a receptive field of 9, as we require input size 9x9 to produce a 1x1 output.

2 Adalines (15 pts)

Problem 2.1 (3 pts) Observe the Adaline shown in Figure **??**, fill in the feature s and output y for each pair of inputs given in the truth table. What logic function is this Adaline performing?

The Adaline is performing the AND logic function.

x_1	x_2	s	y
-1	-1	-5	-1
-1	+1	-3	-1
+1	-1	-1	-1
+1	+1	1	+1

Problem 2.2 (4 pts) Propose proper values for weight w_0, w_1 and w_2 in the Adaline shown in Figure **??** to perform the functionality of a logic **NOR** function. Fill in the feature s for each pair of inputs given in the truth table to prove the functionality is correct.

Using the weights w_0 = -1, w_1 = -1, and w_2 = -1 will cause the Adaline to perform the functionality of the NOR logic function

x_1	x_2	s	y
-1	-1	1	+1
-1	+1	-1	-1
+1	-1	-1	-1
+1	+1	-3	-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 **??** 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.

Using the weights w_0 = 0, w_1 = 1, w_2 = 1, and w_3 = 1 will cause the Adaline to perform the functionality of the Majority Vote function

x_1	x_2	x_3	s	y
-1	-1	-1	-3	-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	+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 **??** 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.

Using the weights w_{20} = -2, w_{21} = 2, and w_{22} = -1 will cause the 2-layer Madaline to perform the functionality of the XOR Vote function

x_1	x_2	s	y
-1	-1	-3	-1
-1	+1	1	+1
+1	-1	1	+1
+1	+1	-1	-1

3 Back Propagation (10 pts)

Problem 3.1 (5 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

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

A MSE loss function $L = \frac{1}{2}(t - x_3)^T(t - x_3)$ is applied in the end, where $t \in \mathcal{R}^{k\times 1}$ 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**.

Note: The ReLU activation function is not continuous on 0. Let us define the subgradient at 0 to be 1/2, such that $RELU'(x) = \frac{sign(x)+1}{2}$. For our case, we add a m x 1 vector of 1's to sign$(W_1x_1+b_1)$ before dividing by 2.

Note: We use the $\circ$ to denote the Hadamard product, or elementwise matrix multiplication of the ReLU output to convert each value of the input to either 0, 1/2, or 1 based on the sign of $W_1x_1+b_1$ (simply used this for demonstrating the effect of the ReLU on the matrices, as each entry of the first FC layer output is ReLU'd and this is a simple, easier, and correct way to multiply the ReLU derivative)

$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial x_2}\frac{\partial x_2}{\partial W_1} = (\frac{\partial L}{\partial x_3}\frac{\partial x_3}{\partial x_2})\frac{\partial x_2}{\partial W_1} = (W_2^T(x_3 - t))(RELU'(W_1x_1 + b_1)x_1^T)$

$= ((W_2^T(x_3 - t)) \circ \frac{sign(W_1x_1+b_1)+1}{2})(x_1^T)$

$\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial x_2}\frac{\partial x_2}{\partial b_1} = (\frac{\partial L}{\partial x_3}\frac{\partial x_3}{\partial x_2})\frac{\partial x_2}{\partial b_1} = (W_2^T(x_3 - t))(RELU'(W_1x_1 + b_1)(1))$
$= (W_2^T(x_3 - t)) \circ \frac{sign(W_1x_1+b_1)+1}{2})$

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

$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial x_3}\frac{\partial x_3}{\partial b_2} = (x_3 - t)(1) = x_3 - t$

Problem 3.2 (5 pts) Given a data $x_1 = [0, 1, 1]^T$, target value $t = [1, 1]^T$, weights and bias at this iteration are

$$W_1 = \begin{bmatrix} 2 & -1 & 1 \\ -3 & 2 & -1 \end{bmatrix}, b_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \tag{3}$$
$$W_2 = \begin{bmatrix} 1 & -2 \\ -3 & 1 \end{bmatrix}, b_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \tag{4}$$

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

$$
\begin{aligned}
&L = \tfrac{1}{2}(t-x_3)^T(t-x_3) \\
&= \tfrac{1}{2}(\begin{bmatrix}1\\1\end{bmatrix} - \begin{bmatrix}-4\\1\end{bmatrix})^T(\begin{bmatrix}1\\1\end{bmatrix} - \begin{bmatrix}-4\\1\end{bmatrix}) \\
&= \tfrac{1}{2}(\begin{bmatrix}5\\0\end{bmatrix})^T(\begin{bmatrix}5\\0\end{bmatrix}) \\
&= \tfrac{5*5}{2} \\
&= 25/2 = 12.5
\end{aligned}
$$

$$
\begin{aligned}
&\tfrac{\partial L}{\partial W_1} = (((W_2^T(x_3-t)) \circ \tfrac{sign(W_1x_1+b_1)+1}{2})(x_1^T) \\
&= ((\begin{bmatrix}1 & -3\\ -2 & 1\end{bmatrix}(\begin{bmatrix}-4\\1\end{bmatrix} - \begin{bmatrix}1\\1\end{bmatrix})) \circ \frac{sign(\begin{bmatrix}2 & -1 & 1\\ -3 & 2 & -1\end{bmatrix}\begin{bmatrix}0\\1\\1\end{bmatrix} + \begin{bmatrix}1\\2\end{bmatrix}) + \begin{bmatrix}1\\1\end{bmatrix}}{2})(\begin{bmatrix}0 & 1 & 1\end{bmatrix}) \\
&= ((\begin{bmatrix}1 & -3\\ -2 & 1\end{bmatrix}\begin{bmatrix}-5\\0\end{bmatrix}) \circ \frac{sign(\begin{bmatrix}1\\3\end{bmatrix}) + \begin{bmatrix}1\\1\end{bmatrix}}{2})(\begin{bmatrix}0 & 1 & 1\end{bmatrix}) \\
&= (\begin{bmatrix}-5\\10\end{bmatrix} \circ \frac{\begin{bmatrix}1\\1\end{bmatrix} + \begin{bmatrix}1\\1\end{bmatrix}}{2})(\begin{bmatrix}0 & 1 & 1\end{bmatrix}) \\
&= (\begin{bmatrix}-5\\10\end{bmatrix} \circ \begin{bmatrix}1\\1\end{bmatrix})(\begin{bmatrix}0 & 1 & 1\end{bmatrix}) \\
&= \begin{bmatrix}-5\\10\end{bmatrix}\begin{bmatrix}0 & 1 & 1\end{bmatrix} \\
&= \begin{bmatrix}0 & -5 & -5\\ 0 & 10 & 10\end{bmatrix}
\end{aligned}
$$

$$
\begin{aligned}
&\tfrac{\partial L}{\partial W_2} = (x_3-t)x_2^T = (\begin{bmatrix}-4\\1\end{bmatrix} - \begin{bmatrix}1\\1\end{bmatrix})\begin{bmatrix}1\\3\end{bmatrix}^T = \begin{bmatrix}-5\\0\end{bmatrix}\begin{bmatrix}1 & 3\end{bmatrix} \\
&= \begin{bmatrix}-5 & -15\\ 0 & 0\end{bmatrix}
\end{aligned}
$$

$$
\begin{aligned}
&\tfrac{\partial L}{\partial b_1} = ((W_2^T(x_3-t)) \circ \tfrac{sign(W_1x_1+b_1)+1}{2})(1) \\
&= ((\begin{bmatrix}1 & -3\\ -2 & 1\end{bmatrix}(\begin{bmatrix}-4\\1\end{bmatrix} - \begin{bmatrix}1\\1\end{bmatrix})) \circ \frac{sign(\begin{bmatrix}2 & -1 & 1\\ -3 & 2 & -1\end{bmatrix}\begin{bmatrix}0\\1\\1\end{bmatrix} + \begin{bmatrix}1\\2\end{bmatrix}) + \begin{bmatrix}1\\1\end{bmatrix}}{2}) \\
&= ((\begin{bmatrix}1 & -3\\ -2 & 1\end{bmatrix}\begin{bmatrix}-5\\0\end{bmatrix}) \circ \frac{sign(\begin{bmatrix}1\\3\end{bmatrix}) + \begin{bmatrix}1\\1\end{bmatrix}}{2}) \\
&= (\begin{bmatrix}-5\\10\end{bmatrix} \circ \frac{\begin{bmatrix}1\\1\end{bmatrix} + \begin{bmatrix}1\\1\end{bmatrix}}{2}) \\
&= (\begin{bmatrix}-5\\10\end{bmatrix} \circ \begin{bmatrix}1\\1\end{bmatrix}) \\
&= \begin{bmatrix}-5\\10\end{bmatrix}
\end{aligned}
$$

$$\frac{\partial L}{\partial b_2} = x_3 - t = \begin{bmatrix} -4 \\ 1 \end{bmatrix} - \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} -5 \\ 0 \end{bmatrix}$$

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.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 1/32 & 1/8 & 1/32 \\ 1/8 & 1/2 & 1/8 \\ 1/32 & 1/8 & 1/32 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & \frac{1}{32} & \frac{5}{32} & \frac{5}{16} & \frac{11}{16} & \frac{5}{16} & \frac{5}{32} & \frac{1}{32} & 0 \\ \frac{5}{32} & \frac{5}{16} & \frac{13}{16} & \frac{31}{32} & \frac{17}{16} & \frac{31}{32} & \frac{13}{16} & \frac{5}{16} & \frac{5}{32} \\ \frac{5}{8} & \frac{13}{16} & \frac{17}{16} & \frac{9}{8} & \frac{9}{8} & \frac{9}{8} & \frac{17}{16} & \frac{13}{16} & \frac{5}{8} \\ \frac{5}{32} & \frac{5}{16} & \frac{13}{16} & \frac{31}{32} & \frac{17}{16} & \frac{31}{32} & \frac{13}{16} & \frac{5}{16} & \frac{5}{32} \\ 0 & \frac{1}{32} & \frac{5}{32} & \frac{5}{16} & \frac{11}{16} & \frac{5}{16} & \frac{5}{32} & \frac{1}{32} & 0 \end{bmatrix}$$

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)

The kernel is a smoothing kernel that takes a weighted combination of the pixel with its neighbors, weighting the innermost pixels most. It takes information from more pixels to the left and right than above (as the matrix has n > m), and generally places more emphasis on them. The slowly reducing magnitude of each pixel's weight causes the smoothing effect (with an emphasis on horizontal smoothing with some vertical), focusing on inner pixels but introducing some information from outer pixels to introduce a blur, reducing the pixelwise noise.

5 Lab: LMS Algorithm (15 pts)

(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^*?

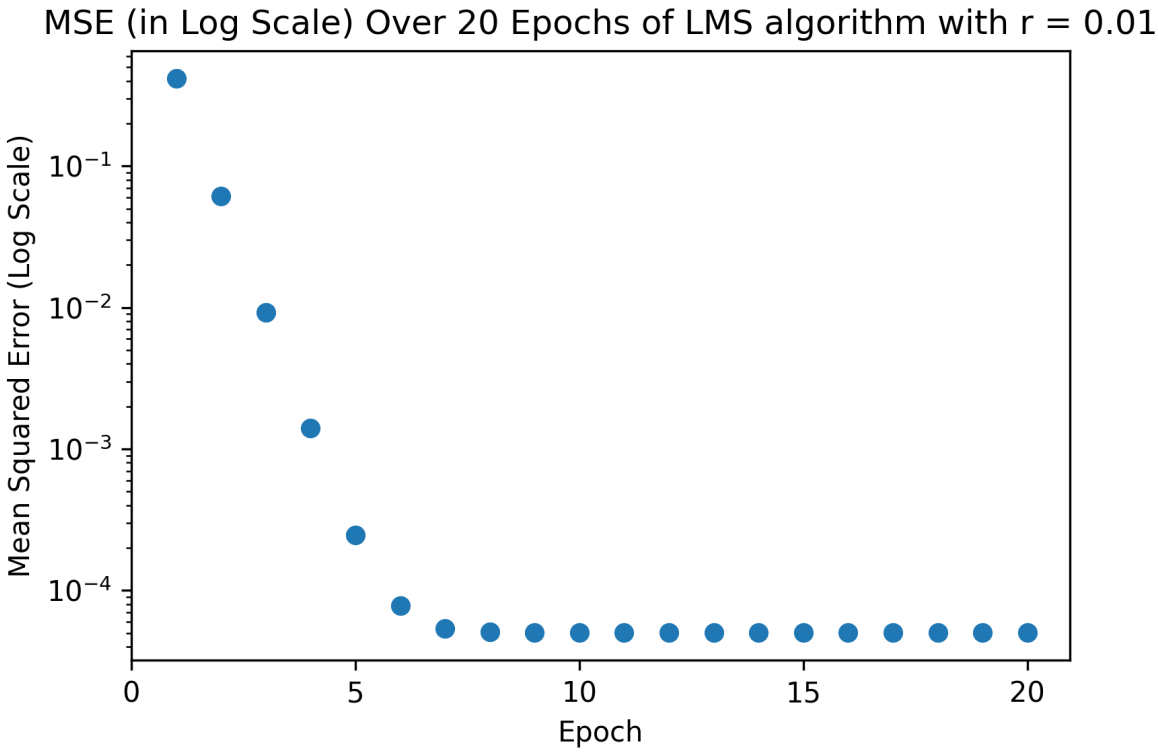
The optimal weight W^* is $\begin{bmatrix} 1.0006781 \\ 1.00061145 \\ -2.00031968 \end{bmatrix}$

The MSE loss of the whole dataset when the weight is set to W^*? is $5.039951565868384e-05$, using the MSE loss defined in the slides (1/2 MSE).

(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 $r = 0.01$, report the weight you get in the end and plot the MSE loss *in log scale* vs. Epochs.

The final weight W calculated after running the LMS algorithm for 20 epochs is $\begin{bmatrix} 1.0006781 \\ 1.00061145 \\ -2.00031968 \end{bmatrix}$

Plot:

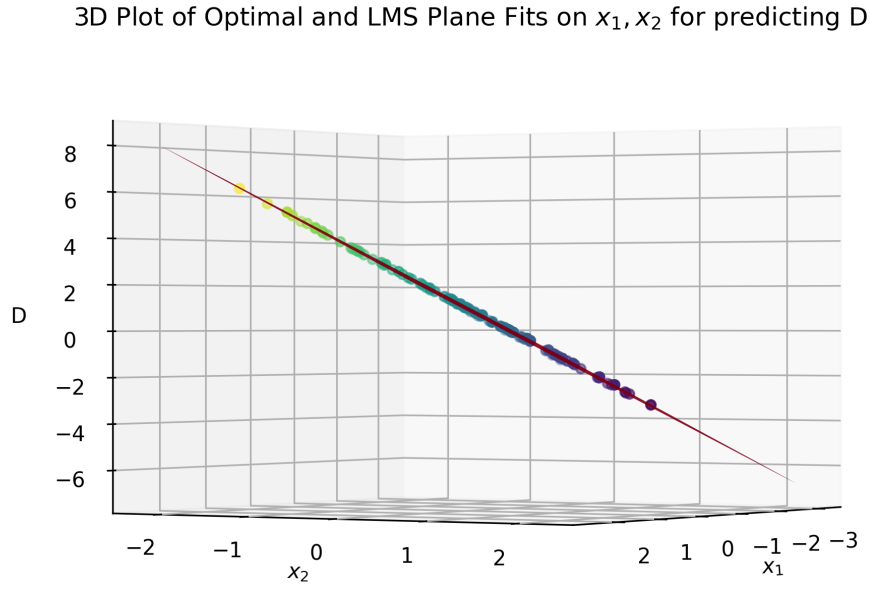


Code output for each epoch:

Epoch 1: MSE = 0.41597664653012956 for Weights: [0.64100742 0.60691145 -1.22256919]
Epoch 2: MSE = 0.06167050055591204 for Weights: [0.87171709 0.84503269 -1.69846199]
Epoch 3: MSE = 0.009183523021172 39 for Weights: [0.95461479 0.93896791 -1.88341134]
Epoch 4: MSE = 0.0013998347202824953 for Weights: [0.98432452 0.97618831 -1.9552063]
Epoch 5: MSE = 0.00024727995925744576 for Weights: [0.99493404 0.99098913 -1.98305376]
Epoch 6: MSE = 7.804454112754207e-05 for Weights: [0.9987048 0.9968916 -1.99384948]
Epoch 7: MSE = 5.386768621940005e-05 for Weights: [1.00003676 0.99925083 -1.99803351]
Epoch 8: MSE = 5.069780261823408e-05 for Weights: [1.00050357 1.00019552 -1.99965493]
Epoch 9: MSE = 5.040083419088486e-05 for Weights: [1.00066553 1.00057433 -2.00028332]
Epoch 10: MSE = 5.042636577765125e-05 for Weights: [1.000721 1.0007264 -2.0005269]
Epoch 11: MSE = 5.045765692029332e-05 for Weights: [1.00073967 1.0007875 -2.00062135]
Epoch 12: MSE = 5.047309380988353 6e-05 for Weights: [1.00074581 1.00081206 -2.00065799]
Epoch 13: MSE = 5.047960899587332e-05 for Weights: [1.00074776 1.00082194 -2.00067221]
Epoch 14: MSE = 5.048222790356126 6e-05 for Weights: [1.00074835 1.00082592 -2.00067773]
Epoch 15: MSE = 5.048326269858834e-05 for Weights: [1.00074851 1.00082752 -2.00067987]
Epoch 16: MSE = 5.048366911150139e-05 for Weights: [1.00074855 1.00082816 -2.00068071]
Epoch 17: MSE = 5.048382843585573e-05 for Weights: [1.00074856 1.00082842 -2.00068103]
Epoch 18: MSE = 5.048389088166683e-05 for Weights: [1.00074855 1.00082853 -2.00068116]
Epoch 19: MSE = 5.048391536703642 6e-05 for Weights: [1.00074855 1.00082857 -2.00068121]
Epoch 20: MSE = 5.048392497431242e-05 for Weights: [1.00074855 1.00082859 -2.00068123]

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

Plot:



In general, it seems as though the line from the linear models in (a) and (b) fit the data well, as evidenced by the fact that the points hover on the line. There is almost no error at all among the points with the lines.

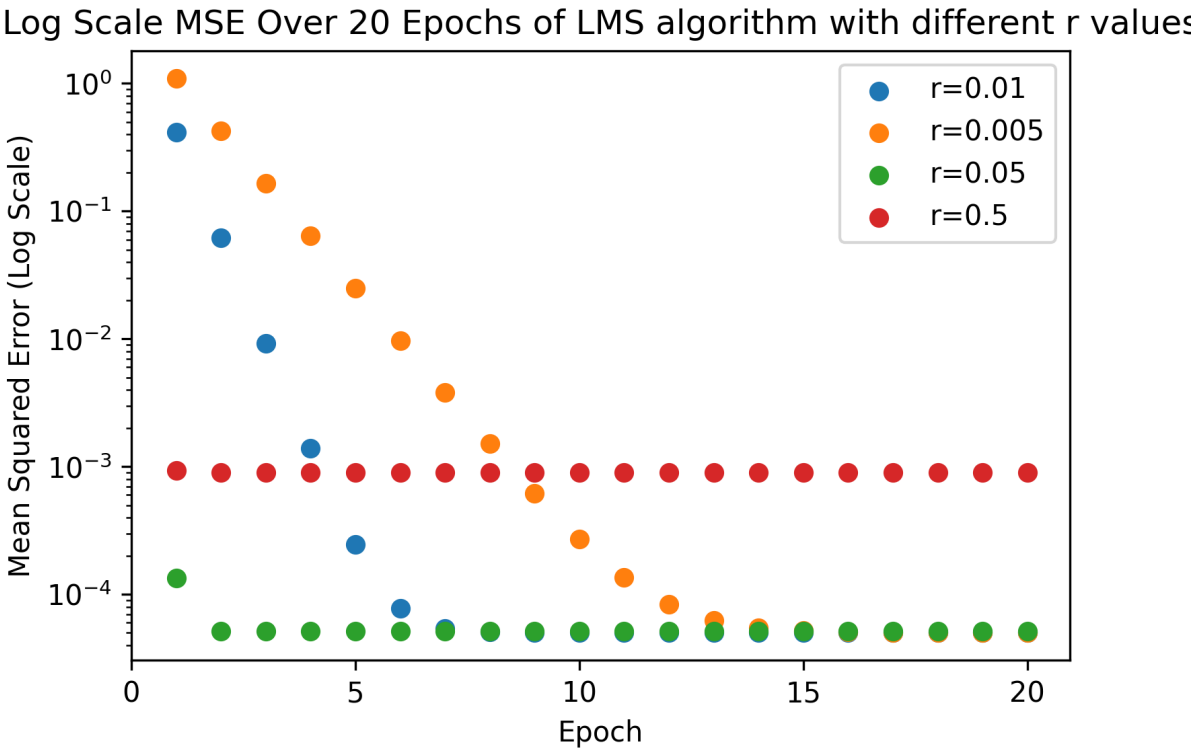
(d) (5pt) Learning rate r is an important hyperparameter for the LMS algorithm, as well as for CNN optimization. Here, try repeat the process in (b) with r set to 0.005, 0.05 and 0.5 respectively. Together with the result you got in (b), plot the MSE losses of the 4 sets of experiments in log scale vs. Epochs in one figure. Then try further enlarge the learning rate to $r = 1$ and observe how the MSE changes. Base on these observations, comment on how learning rate affects the speed and quality of the learning process. (Note: The learning rate tuning for the CNN optimization will be introduced in Lecture 7.)

Based on these runs, the learning rate seems to clearly have great affect on both speed and quality of the learning process. In general, learning rates should likely be somewhat small (at least when further along within epochs), as that helps improve the speed and quality of the learning process. This is because in the cases of the 0.01 and 0.005 and 0.05 learning rates, they were both able to reduce the loss to $10^-4 within 20 epochs, as their small learning rates allowed them not to overstep the local minimum. This seems to be the$

MSE for r= 1

Epoch 1: MSE = 6.221473538428372e+18 for Weights: [2.44853746e+09 -7.82004386e+08 -2.54835788e+09]

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



Epoch 2: MSE = 4.037539197035097e+37 for Weights: [6.23762276e+18 -1.99213110e+18 -6.49190139e+18]

Epoch 3: MSE = 2.6202336516852733e+56 for Weights: [1.58902504e+28 -5.07492407e+27 -1.65380214e+28]

Epoch 4: MSE = 1.7004477366967381e+75 for Weights: [4.04801743e+37 -1.29282929e+37 -4.21303612e+37]

Epoch 5: MSE = 1.103536130595577e+94 for Weights: [1.03122637e+47 -3.29346322e+46 -1.07326463e+47]

Epoch 6: MSE = 7.1615961211223955e+112 for Weights: [2.62703369e+56 -8.39004812e+55 -2.73412554e+56]

Epoch 7: MSE = 4.647646559102238e+131 for Weights: [6.69232890e+65 -2.13735217e+65 -6.96514377e+65]

Epoch 8: MSE = 3.016173793245054e+150 for Weights: [1.70486074e+75 -5.44487257e+74 -1.77435992e+75]

Epoch 9: MSE = 1.957400210057157e+169 for Weights: [4.34310716e+84 -1.38707312e+84 -4.52015528e+84]

Epoch 10: MSE = 1.2702900578582652e+188 for Weights: [1.10640003e+94 -3.53354795e+93 -1.15150277e+94]

Epoch 11: MSE = 8.243775712307943e+206 for Weights: [2.81853747e+103 -9.00166036e+102 -2.93343603e+103]

Epoch 12: MSE = 5.349946461001627e+225 for Weights: [7.18018192e+112 -2.29315947e+112 -7.47288427e+112]

Epoch 13: MSE = 3.4719439410325566e+244 for Weights: [1.82914057e+122 -5.84178933e+121 -1.90370605e+122]

Epoch 14: MSE = 2.2531804416257442e+263 for Weights: [4.65970815e+131 -1.48818706e+131 -4.84966260e+131]

Epoch 15: MSE = 1.4622419568833683e+282 for Weights: [1.18705366e+141 -3.79113421e+140 -1.23544427e+141]

Epoch 16: MSE = 9.489482071517212e+300 for Weights: [3.02400138e+150 -9.65785751e+149 -3.14727574e+150]

Epoch 17: MSE = inf for Weights: [7.70359813e+159 -2.46032471e+159 -8.01763770e+159]

Epoch 18: MSE = inf for Weights: [1.96248006e+169 -6.26764027e+168 -2.04248117e+169]

Epoch 19: MSE = inf for Weights: [4.99938850e+178 -1.59667195e+178 -5.20319013e+178]

Epoch 20: MSE = inf for Weights: [1.27358672e+188 -4.06749784e+187 -1.32550488e+188]

6 Lab: Simple NN (40 pts)

Name	Type	Kernel size	depth/units	Activation	Strides
Conv 1	Convolution	5	32	ReLU	1
MaxPool	MaxPool	3	N/A	N/A	2
Conv 2	Convolution	5	32	ReLU	1
MaxPool	MaxPool	3	N/A	N/A	2
Conv 3	Convolution	5	64	ReLU	1
MaxPool	MaxPool	3	N/A	N/A	2
FC1	Fully-connected	N/A	64	ReLU	N/A
FC2	Fully-connected	N/A	10	ReLU	N/A

Table 1: The padding for all three convolution layers is 2. 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 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.

Lab 6 Code:

```
"""
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 = 32, kernel_size = 5, stride=1,
        padding=2)    #Your code here
        self.conv2 = CONV(in_channels = 32, out_channels = 32, kernel_size = 5, stride=1,
        padding=2)    #Your code here
        self.conv3 = CONV(in_channels = 32, out_channels = 64, kernel_size = 5, stride=1,
        padding=2)    #Your code here
```

```
        self.fc1   = FC(in_features = 576, out_features = 64)     #Your code here
        self.fc2   = FC(in_features = 64, out_features = 10)     #Your code here

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

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

(b) (30pt) 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.

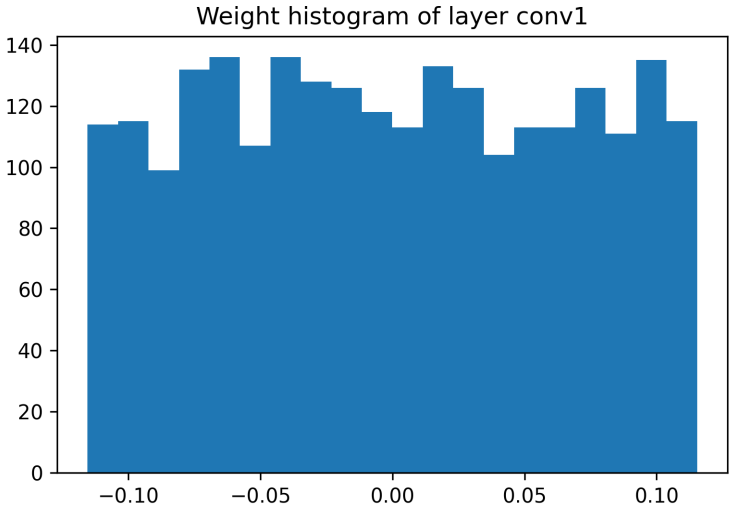
Layer	Input shape	Output shape	Weight shape	# Param	# MAC
Conv 1	(1, 3, 32, 32)	(1, 32, 32, 32)	(32, 3, 5, 5)	2400	2457600
Conv 2	(1, 32, 15, 15)	(1, 32, 15, 15)	(32, 32, 5, 5)	25600	5760000
Conv 3	(1, 32, 7, 7)	(1, 64, 7, 7)	(64, 32, 5, 5)	51200	2508800
FC1	(1, 576)	(1, 64)	(64, 576)	36864	36864
FC2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

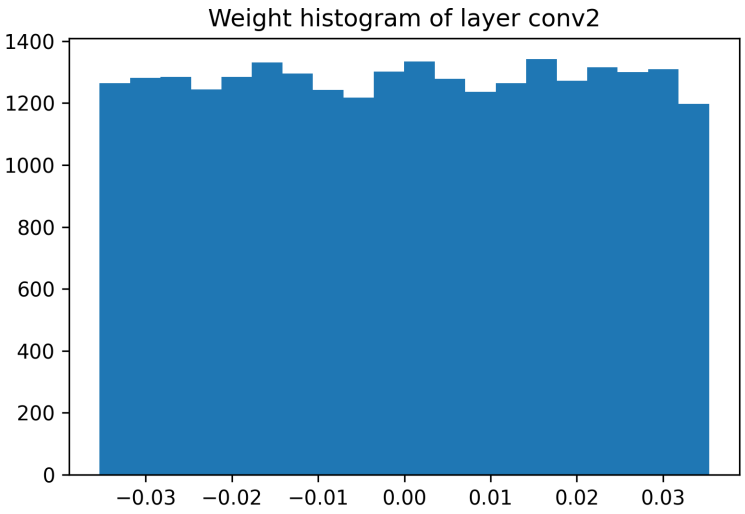
Table 2: Results of Lab 2(b).

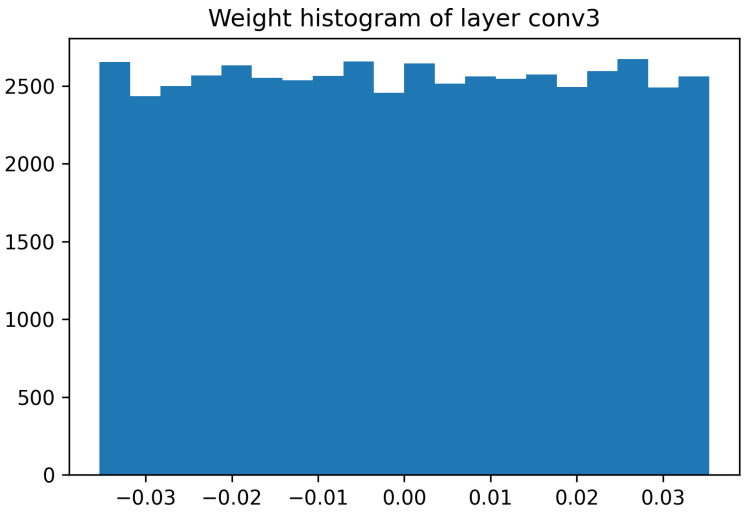
6.1 Lab 3 (Bonus 10 points)

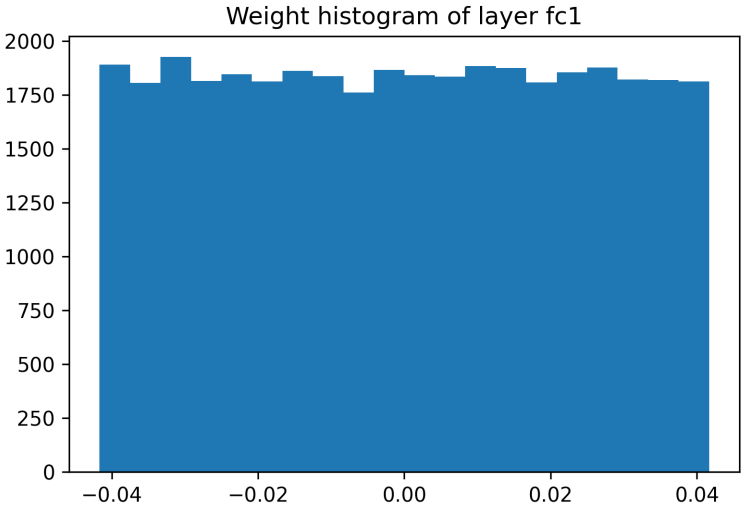
(a) (2pt) Complete the for-loop in code block 5 to plot the histogram of weight elements in each one of the 5 convolutional and fully-connected layers.

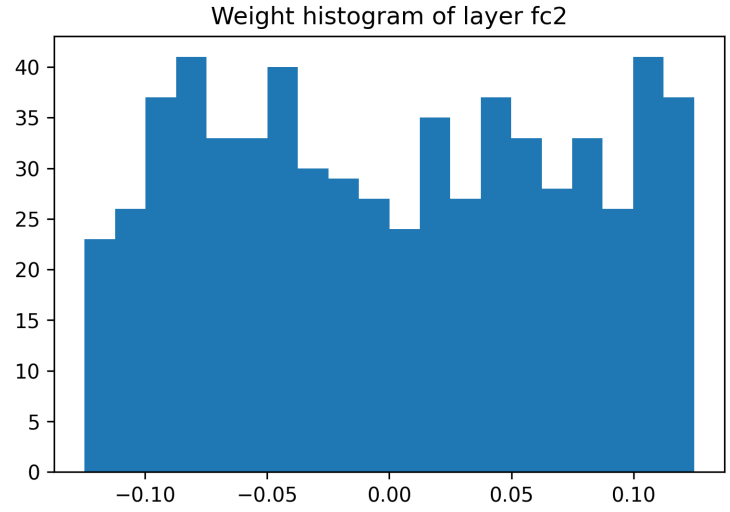
Output Plots:



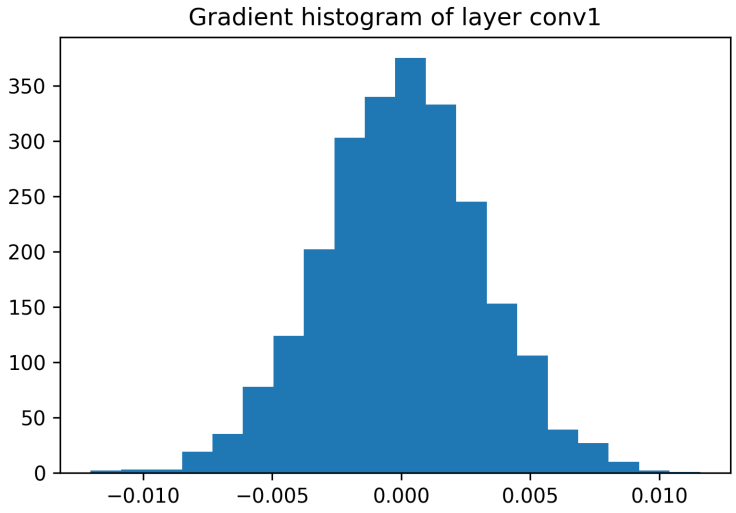

Weight histogram of layer conv2
1400
1200
1000
800
600
400
200
0
−0.03
−0.02
−0.01
0.00
0.01
0.02
0.03

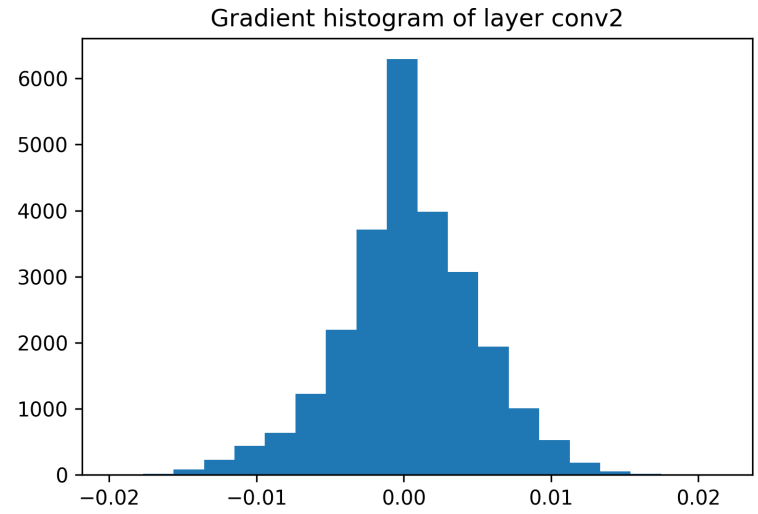

Weight histogram of layer conv3
2500
2000
1500
1000
500
0
−0.03
−0.02
−0.01
0.00
0.01
0.02
0.03

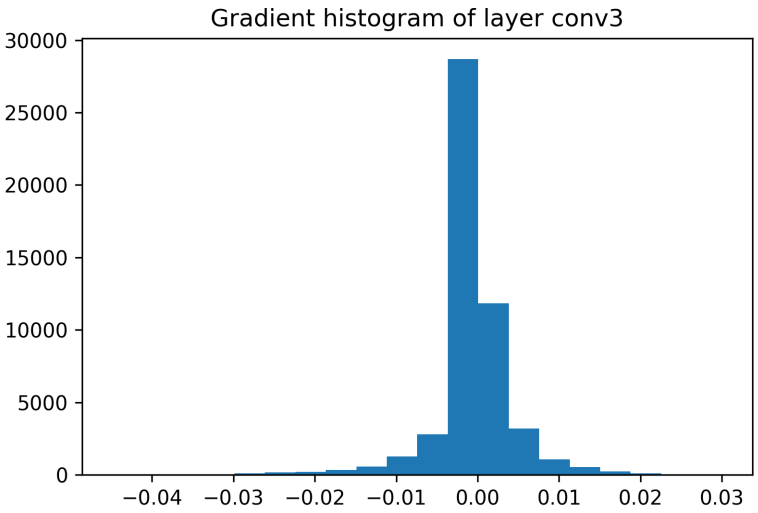

Weight histogram of layer fc1
2000
1750
1500
1250
1000
750
500
250
0
−0.04
−0.02
0.00
0.02
0.04

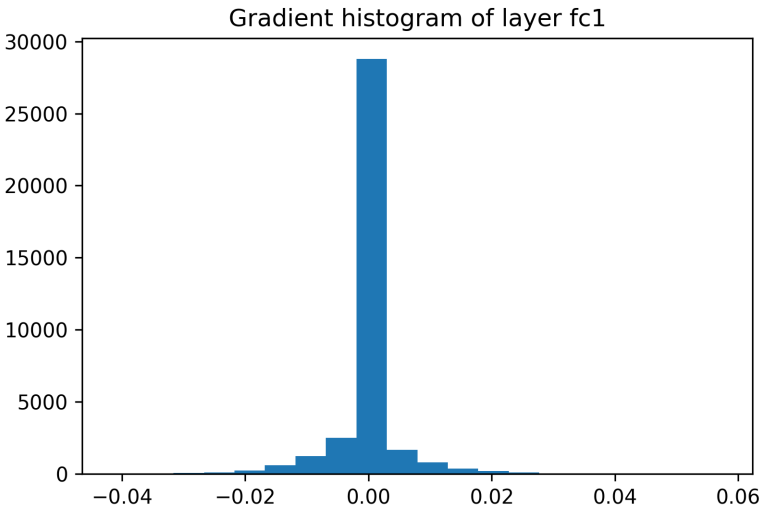


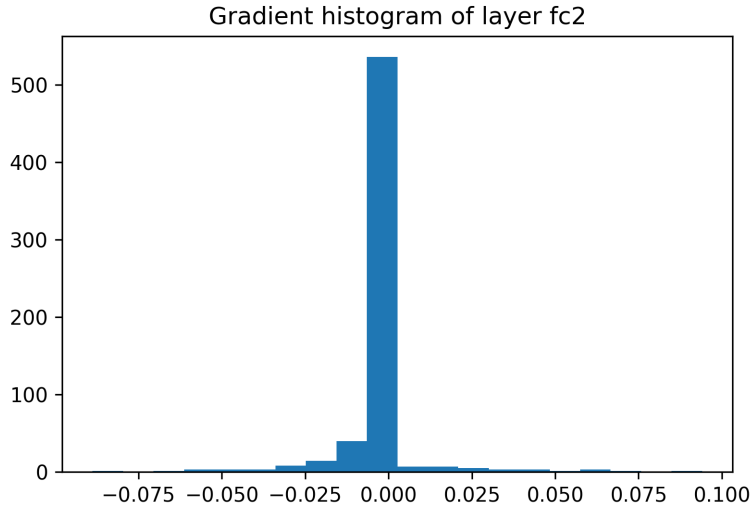
(b) (3pt) In code block 6, complete the code for backward pass, then complete the for-loop to plot the histogram of weight elements' gradients in each one of the 5 convolutional and fully-connected layers. Output Plots:




Gradient histogram of layer conv2
6000
5000
4000
3000
2000
1000
0
−0.02
−0.01
0.00
0.01
0.02


Gradient histogram of layer conv3
30000
25000
20000
15000
10000
5000
0
−0.04
−0.03
−0.02
−0.01
0.00
0.01
0.02
0.03

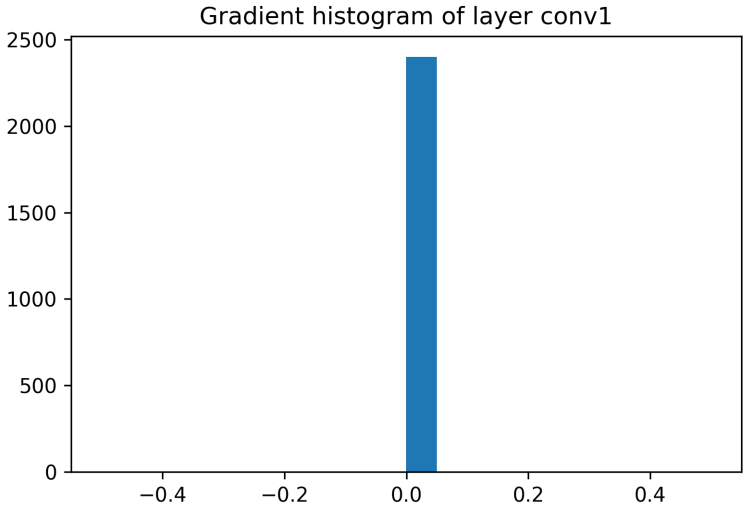

Gradient histogram of layer fc1
30000
25000
20000
15000
10000
5000
0
−0.04
−0.02
0.00
0.02
0.04
0.06

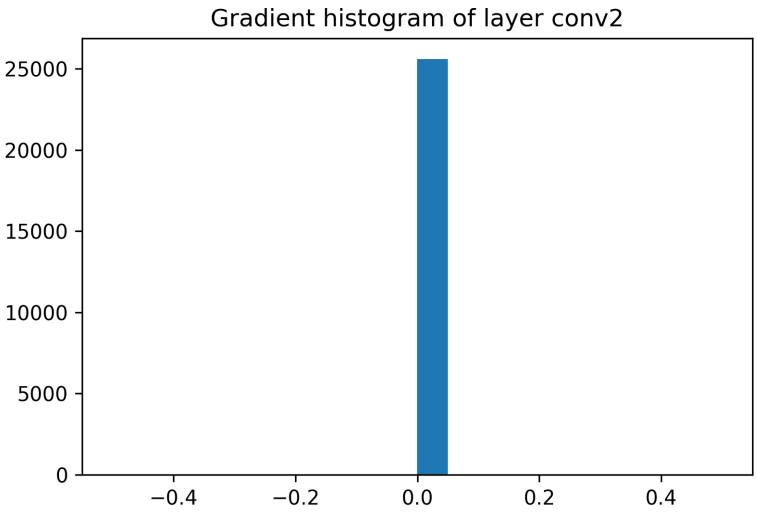


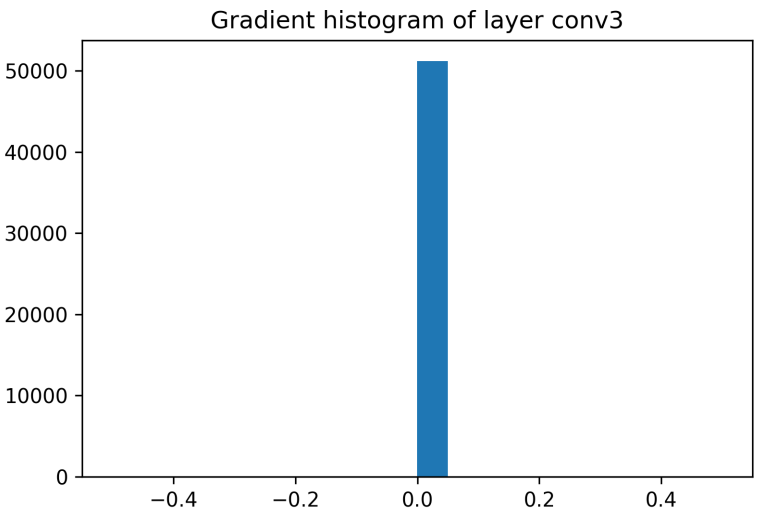
(c) (5pt) In code block 7, finish the code to set all the weights to 0. Perform forward and backward pass again to get the gradients, and plot the histogram of weight elements' gradients in each one of the 5 convolutional and fully-connected layers. Comparing with the histograms you got in (b), are there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

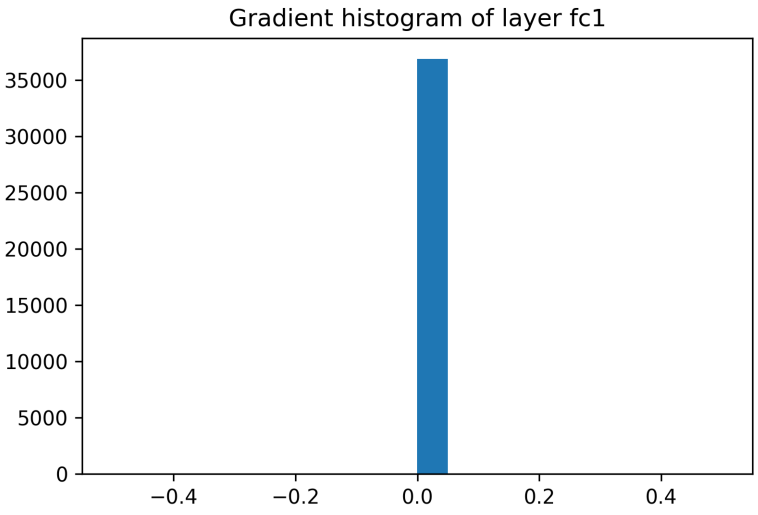
The histograms for the gradients for 0 initialization are quite different from the previous gradients. These histograms have every gradient in exactly one bin (0 bin). Meanwhile, the gradients in the histograms had much more scatter (higher standard deviation). This is because 0 initialization leads to the input becoming 0 and the gradient just passing through as 0 through each layer, as the ReLU activation will keep it at 0. Thus, the value is just zeroed out, and there is no gradient to propagate backwards. As slides 5 put it, the neuron is dead, such that the weights will just be stuck at 0 no matter what, regardless of the incoming data. Meanwhile, the other histograms are scattered as they are able to slowly refine their weights based on their initialization (PyTorch's random initialization-default is Lecun initialization). The Conv layers will are approximately normal, and the FC layers are unskewed, unimodal with a huge spike at 0

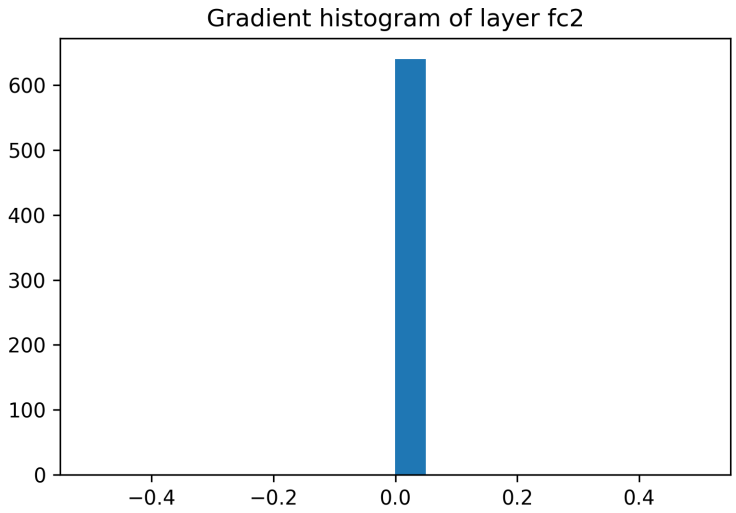
Output Plots:




Gradient histogram of layer conv2
25000
20000
15000
10000
5000
0
−0.4
−0.2
0.0
0.2
0.4


Gradient histogram of layer conv3
50000
40000
30000
20000
10000
0
−0.4
−0.2
0.0
0.2
0.4


Gradient histogram of layer fc1
35000
30000
25000
20000
15000
10000
5000
0
−0.4
−0.2
0.0
0.2
0.4


Gradient histogram of layer fc2
600
500
400
300
200
100
0
−0.4
−0.2
0.0
0.2
0.4