ECE 661 - Homework 1

Michael Li (zl310)

September 18, 2023

I have adhered to the Duke Community Standard in completing this assignment.

Contents

List of Figures

1 True/False Questions (10 pts)

Problem 1.1 (2 pts) The overfitting models can perfectly fit the training data. We should increase the noise in the training data and the number of parameters of the model to improve NN's generalization ability.

False. It's possible to add zero mean Gaussian noise to the underlying data if the idea is to generate more training data via data augmentation. However, increasing the number of parameters generally decrease the generalisability of a model. Fifteen data points sampled from a third degree polynomial should not be fitted with a fifteen degree polynomial, which would perfectly fit the data but would not represent the underlying trend. Increasing the number of parameters generally cause the model to become more flexible, meaning that the model will fit to whatever noise is present in the training data.
Therefore, it's true that adding noise to training data as a form of data augmentation can reduce overfitting, but the second part of the solution (increase number of parameters) is wrong.

Problem 1.2 (2 pts) Given a learning task that can be perfectly learned by a Madaline model, the same set of weight values will be achieved after training, no matter how the Madaline is initialized.

False. The Madaline model uses the Error-Correction Rule to train the model. This involves flipping random Adalines (the n^{th} iteration in any given layer flips a random combination of n Adalines) and seeing if the model accuracy actually improves. Therefore, the model is very sensitive to its initial state and the randomness of the model and convergence isn't guaranteed.

Problem 1.3 (2 pts) The error surface can be complicated. The direction of steepest descent is not always the direction towards the minimum. Full batch size can keep the direction of steepest descent perpendicular to the contour lines. Thus, we should increase the batch size when the error surface is complicated.

False. The first two sentences are true (I'm assuming that "minimum" means global minimum in the second sentence). For surfaces with a global minimum and local minima, the steepest descent at the current point in a localised region may lead the model towards a local minimum instead of the global minimum. The third sentence is also true as covered in lecture (slide 42, lecture 2). However, the last sentence is false. Following the steepest descent (which happens when the full batch size is used) is not a good design choice when the error surface is complicated because the direction of steepest descent can easily lead to a local minimum. Using a small batch size allows the direction of travel to zig-zag in a semi-random fashion on the surface, meaning that the direction of travel isn't always perpendicular to the contour lines. When there is a lot of local minima, the zig-zag pattern could help the model reach the global minimum. Therefore, decreasing batch size would help when the error surface is complicated.

Problem 1.4 (2 pts) In the following code, "If-else" splits the modified Adalines model into two parts. Each part is differentiable. The backpropagation algorithm can be applied to the training of the entire model.

Algorithm 1 A modified Adalines with branches

Require: w_1, w_2, x_1, x_2, n
Ensure: $n \neq 0$
Ensure: $(x_1 w_1 + x_2 w_2) \neq 0$
  if $n > 0$ **then**
    $y \leftarrow Sign(x_1 w_1 + x_2 w_2)$
  else
    $y \leftarrow Sign(x_1 w_1 + x_2 w_2) + 5$
  end if

Figure 1: Algorithm 1.

False. The modified Adaline cannot support backpropagation because each branch of the Adaline still uses the signum function with $\vec{x}_1 w_1 + \vec{x}_2 w_2$ as its argument. Since the if statement does not guarantee the sign of $\vec{x}_1 w_1 + \vec{x}_2 w_2$, the input to the signum function within each branch can either be negative or positive. Since the signum function is not differentiable given an input with unrestricted sign, each part of the modified Adaline still isn't differentiable.

Problem 1.5 (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 will always lead to a larger output feature map size.

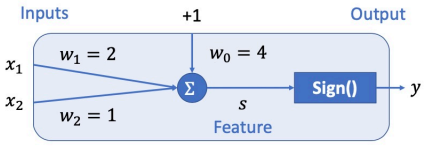
False. The convolution shape rule is

$$W_2 = \lfloor \frac{W_1 - K + 2P}{S} \rfloor + 1, H_2 = \lfloor \frac{H_1 - K + 2P}{S} \rfloor + 1$$

for an input feature map with shape $H_1 \times W_1$ where K is the convolution kernel size, S is the size of each stride, P is the border padding, resulting in an output feature map with shape $H_2 \times W_2$. Using the above equation, if the kernel size increases (larger K) and W_1, H_1, P, S stay constant, the output feature map will be smaller.

2 Adalines (15 pts)

Problem 2.1 (3 pts) Observe the Adaline shown in Figure 2, 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?

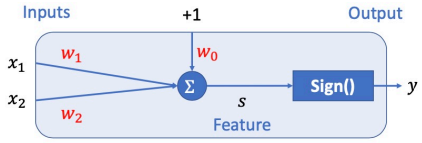


x_1	x_2	s	y
-1	-1	1	1
-1	+1	3	1
+1	-1	5	1
+1	+1	7	1

Figure 2: Problem 2.1.

Figure 2 shows the filled-out table with feature s and output y using $s = w_0 + \vec{x}_1 w_1 + \vec{x}_2 w_2, y = \text{sgn}(s)$. Since the output y is one independent of the values of $\vec{x}_1, \vec{x}_2$, the $\boxed{identity}$ logic function is performed following the naming convention of this link.

Problem 2.2 (4 pts) Propose proper values for weight w_0, w_1 and w_2 in the Adaline shown in Figure 3 to perform the functionality of a logic NAND function. Fill in the feature s for each pair of inputs given in the truth table to prove the functionality is correct.

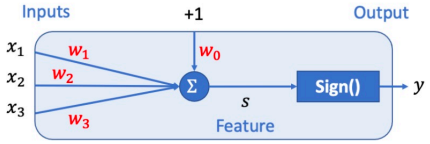


x_1	x_2	s	y
-1	-1	1.5	+1
-1	+1	0.5	+1
+1	-1	0.5	+1
+1	+1	-0.5	-1

Figure 3: Problem 2.2.

I used scikit-learn's Linear Regression package to solve this problem. The input is the 4×2 matrix as given in $\vec{x}_1, \vec{x}_2$ in Figure 3 and the target is the column vector y. The weights are: $\boxed{w_0 = 0.5, w_1 = w_2 = -0.5}$. The calculations involve: $s = w_0 + \vec{x}_1 w_1 + \vec{x}_2 w_2, y = \text{sgn}(s)$. See Figure 3 for the values of s.

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 4 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.

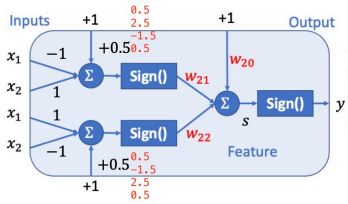


x_1	x_2	x_3	s	y
-1	-1	-1	-1.5	-1
-1	-1	+1	-0.5	-1
-1	+1	-1	-0.5	-1
-1	+1	+1	0.5	+1
+1	-1	-1	-0.5	-1
+1	-1	+1	0.5	+1
+1	+1	-1	0.5	+1
+1	+1	+1	1.5	+1

Figure 4: Problem 2.3.

Again, I used scikit-learn. The input data is the 8×3 matrix that corresponds to $\vec{x}_1, \vec{x}_2, \vec{x}_3$ and the target vector is y. One set of weights that works is $\boxed{w_0 = 0, w_1 = w_2 = w_3 = 0.5}$. The feature s is filled out as shown in Figure 4.

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 5a 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
+1	+1	-1	-1

(a) See column 3 for s.

```
before masking (outputs from layer 1):
[[ 0.5  0.5]
 [ 2.5 -1.5]
 [-1.5  2.5]
 [ 0.5  0.5]]
after masking (outputs from layer 1):
[[ 1.  1.]
 [ 1. -1.]
 [-1.  1.]
 [ 1.  1.]]
fitting xor (w20, w21, w22) = (1.0, array([-1., -1.]))
```

(b) Process of fitting.

Figure 5: Problem 2.4.

The raw outputs of the first layer Adalines are shown in the left-hand-side of Figure 5a. The weights are $\boxed{w_{20} = 1, w_{21} = w_{22} = -1}$. The first matrix in Figure 5b shows the outputs from layer 1, with the left and right columns corresponding to the top and bottom Adalines, respectively. The feature s is filled out as shown in Figure 5a.

Please refer to the code at the back for details involving the computation of the weights.

3 Back Propagation (10 pts)

Problem 3.1 (5 pts) Consider a 2-layer fully-connected NN, where we have input $\vec{x}_1 \in \mathbb{R}^{n\times 1}$, hidden feature $\vec{x}_2 \in \mathbb{R}^{m\times 1}$ output $\vec{x}_3 \in \mathbb{R}^{k\times 1}$ and weights and bias $W_1 \in \mathbb{R}^{m\times n}$, $W_2 \in \mathbb{R}^{k\times m}$, $\vec{b}_1 \in \mathbb{R}^{m\times 1}$, $\vec{b}_2 \in \mathbb{R}^{k\times 1}$ of the two layers. The hidden features and outputs are computed as follows

$$\vec{x}_2 = \sigma(W_1\vec{x}_1 + \vec{b}_1) \tag{1}$$

$$\vec{x}_3 = W_2\vec{x}_2 + \vec{b}_2 \tag{2}$$

A MSE loss function $L = 0.5(\vec{t} - \vec{x}_3)^T(\vec{t} - \vec{x}_3)$ is applied in the end, where $\vec{t} \in \mathbb{R}^{k\times 1}$ is the target value. Following the chain rule, derive the gradient $\frac{\partial L}{\partial W_1}$, $\frac{\partial L}{\partial W_2}$, $\frac{\partial L}{\partial \vec{b}_1}$, $\frac{\partial L}{\partial \vec{b}_2}$ in a vectorized format.

The inputs to the loss function can be expressed in terms of their own parameters:

$$L = 0.5\left(\vec{t} - \vec{x}_3\left(W_2, \vec{x}_2(\sigma(W_1, \vec{x}_1, \vec{b}_1)), \vec{b}_2\right)\right)^T\left(\vec{t} - \vec{x}_3\left(W_2, \vec{x}_2(\sigma(W_1, \vec{x}_1, \vec{b}_1)), \vec{b}_2\right)\right) \tag{3}$$

Using the chain rule and the relationships between different variables in Equation 3,

$$\frac{\partial L}{\partial \vec{b}_1} = \frac{\partial L}{\partial \vec{x}_3}\frac{\partial \vec{x}_3}{\partial \vec{x}_2}\frac{\partial \vec{x}_2}{\partial \vec{b}_1} = \frac{\partial L}{\partial \vec{x}_3}\frac{\partial \vec{x}_3}{\partial \vec{x}_2}\left[\frac{\partial\left[\sigma(W_1\vec{x}_1 + \vec{b}_1)\right]}{\partial(W_1\vec{x}_1 + \vec{b}_1)}\frac{\partial(W_1\vec{x}_1 + \vec{b}_1)}{\partial \vec{b}_1}\right] \tag{4}$$

Since we know which variables are vectors, matrices, and scalars, we can use identities defined in `https://en.wikipedia.org/wiki/Matrix_calculus#Identities` and `https://en.wikipedia.org/wiki/Matrix_calculus#Identities`, assuming the "numerator layout" convention. The first and second terms are (Equations 5 and 6):

$$\frac{\partial L}{\partial \vec{x}_3} = -(\vec{t} - \vec{x}_3)^T \in \mathbb{R}^{1\times k} \tag{5}$$

$$\frac{\partial \vec{x}_3}{\partial \vec{x}_2} = \frac{\partial(W_2\vec{x}_2)}{\partial \vec{x}_2} + \frac{\partial \vec{b}_2}{\partial \vec{x}_2} = W_2 + \mathbf{0} \in \mathbb{R}^{k\times m} \tag{6}$$

To calculate the third term $\frac{\partial([\sigma(W_1\vec{x}_1+\vec{b}_1)])}{\partial(W_1\vec{x}_1+\vec{b}_1)}$, let $\vec{y} = W_1\vec{x}_1 + \vec{b}_1 = [y_1, \ldots, y_m]^T$, $\vec{z} = \sigma(W_1\vec{x}_1 + \vec{b}_1) = \sigma(\vec{y}) = [\sigma(y_1), \ldots \sigma(y_m)]^T = [z_1, \ldots, z_m]^T$. Rewriting $\frac{\partial([\sigma(W_1\vec{x}_1+\vec{b}_1)])}{\partial(W_1\vec{x}_1+\vec{b}_1)}$,

$$\begin{aligned}\frac{\partial\left(\left[\sigma(W_1\vec{x}_1 + \vec{b}_1)\right]\right)}{\partial(W_1\vec{x}_1 + \vec{b}_1)} = \frac{\partial \vec{z}}{\partial \vec{y}} &= \begin{bmatrix} \frac{dz_1}{dy_1} & \frac{dz_1}{dy_2} & \cdots & \frac{dz_1}{dy_m} \\ \frac{dz_2}{dy_1} & \frac{dz_2}{dy_2} & \cdots & \frac{dz_2}{dy_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{dz_m}{dy_1} & \frac{dz_m}{dy_2} & \cdots & \frac{dz_m}{dy_m} \end{bmatrix} = \begin{bmatrix} z_1(1-z_1) & 0 & \ldots & 0 \\ 0 & z_2(1-z_2) & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & z_m(1-z_m) \end{bmatrix} \\ &= \text{diag}(\vec{z})\text{diag}(\vec{1} - \vec{z}) = \text{diag}(\vec{x}_2)\text{diag}(\vec{1} - \vec{x}_2) \in \mathbb{R}^{m\times m}\end{aligned} \tag{7}$$

where the diag(v) function sequentially fills the diagonal of a square matrix with the elements of v. In Equation 7, $\frac{\partial z_i}{\partial y_j} = 0, \forall i \neq j$ because $z_i = \sigma(y_i)$ is only a function of y_i and not a function of $y_j, \forall j \neq i$. This is taken from pages 2 and 3 of this document (note: the context is $\vec{z} = W\vec{y}$, but the form of the matrix at the beginning of page 3 isn't predicated on $\vec{z} = W\vec{y}$ being the relationship between $\vec{y}$ and $\vec{z}$ and a non-linear function $f : \mathbb{R}^m \to \mathbb{R}^m$ that maps $\vec{y}$ onto $\vec{z}$ would also have $\frac{\partial \vec{z}}{\partial \vec{y}}$ of the same form.). The fourth term in Equation 4 is as follows:

$$\frac{\partial(W_1\vec{x}_1 + \vec{b}_1)}{\partial \vec{b}_1} = \frac{\partial(W_1\vec{x}_1)}{\partial \vec{b}_1} + \frac{\partial \vec{b}_1}{\partial \vec{b}_1} = \mathbf{0} + \mathbf{I} \in \mathbb{R}^{m\times m} \tag{8}$$

Putting everything together,

$$\frac{\partial L}{\partial \vec{b}_1} = \frac{\partial L}{\partial \vec{x}_3}\frac{\partial \vec{x}_3}{\partial \vec{x}_2}\frac{\partial \vec{x}_2}{\partial \vec{b}_1} = \frac{\partial L}{\partial \vec{x}_3}\frac{\partial \vec{x}_3}{\partial \vec{x}_2}\frac{\partial \vec{x}_2}{\partial(W_1\vec{x}_1 + \vec{b}_1)}\frac{\partial(W_1\vec{x}_1 + \vec{b}_1)}{\partial \vec{b}_1} = \boxed{-(\vec{t} - \vec{x}_3)^T W_2\text{diag}(\vec{x}_2)\text{diag}(\vec{1} - \vec{x}_2)} \in \mathbb{R}^{1\times m} \tag{9}$$

The dimensions in Equation 9 work out correctly.

Similarly to Equation 4, using the chain rule and the loss function in Equation 3,

$$\frac{\partial L}{\partial \vec{b}_2} = \frac{\partial L}{\partial \vec{x}_3}\frac{\partial \vec{x}_3}{\partial \vec{b}_2} \tag{10}$$

Here, we have less layers of multiplications because we're only interested in the two connections between L and $\vec{b}_2$. The result is therefore

$$\frac{\partial L}{\partial \vec{b}_2} = \frac{\partial L}{\partial \vec{x}_3}\frac{\partial \vec{x}_3}{\partial \vec{b}_2} = \boxed{-(\vec{t}-\vec{x}_3)^T} \in \mathbb{R}^{1\times k} \tag{11}$$

where $\frac{\partial \vec{x}_3}{\partial \vec{b}_2} = \frac{\partial (W_2\vec{x}_2)}{\partial \vec{b}_2} + \frac{\partial \vec{b}_2}{\partial \vec{b}_2} = \mathbf{I} \in \mathbb{R}^{k\times k}$.

Using the chain rule and the loss function in Equation 3,

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial \vec{x}_3}\frac{\partial \vec{x}_3}{\partial \vec{x}_2}\frac{\partial \vec{x}_2}{\partial W_1} = \frac{\partial L}{\partial \vec{x}_3}\frac{\partial \vec{x}_3}{\partial \vec{x}_2}\left[\frac{\partial(\left[\sigma(W_1\vec{x}_1+\vec{b}_1)\right])}{\partial(W_1\vec{x}_1+\vec{b}_1)}\frac{\partial(W_1\vec{x}_1+\vec{b}_1)}{\partial W_1}\right] \tag{12}$$

Using Equation 9 and seeing matching terms in Equations 4 and 12, we know that the first three terms of the expanded form of Equation 12 is

$$\frac{\partial L}{\partial \vec{x}_3}\frac{\partial \vec{x}_3}{\partial \vec{x}_2}\frac{\partial \vec{x}_2}{\partial(W_1\vec{x}_1+\vec{b}_1)} = -(t-\vec{x}_3)^T W_2 \text{diag}(\vec{x}_2)\text{diag}(\vec{1}-\vec{x}_2) \in \mathbb{R}^{1\times m} \tag{13}$$

Getting the last term $\frac{\partial(W_1\vec{x}_1+\vec{b}_1)}{\partial W_1} = \frac{\partial(W_1\vec{x}_1)}{\partial W_1} + \mathbf{0}$ is more difficult. This section of the article in Wikipedia suggests that vector-by-matrix partial derivatives are not well defined. Another link suggests that

$$\frac{\partial(Wx)}{\partial(\text{vec}(W))} = \vec{x}^T \otimes \mathbf{I}_{m\times m} \in \mathbb{R}^{m\times(m\cdot n)} \tag{14}$$

given matrix $W \in \mathbb{R}^{m\times n}$ and vector $x \in \mathbb{R}^{n\times 1}$. Note that the vectorisation operator in Equation 14 is defined as, for $A = [\vec{a}_1 \dots \vec{a}_n] \in \mathbb{R}^{m\times n}$:

$$\text{vec}(A) = \begin{bmatrix}\vec{a}_1 \\ \vdots \\ \vec{a}_n\end{bmatrix} \in \mathbb{R}^{(m\cdot n)\times 1} \tag{15}$$

See this page for the definition of the Kronecker Product denoted by $\otimes$ in 14. Yet another link suggests that the partial derivative $\frac{\partial(Wx)}{\partial W}$ (without vectorisation this time) is a third order tensor.

Consider the derivative with respect of L to $\text{vec}(W_1)$, adapted from Equation 12:

$$\begin{aligned}\frac{\partial L}{\partial \vec{x}_3}\frac{\partial \vec{x}_3}{\partial \vec{x}_2}\left(\frac{\partial \vec{x}_2}{\partial(W_1\vec{x}_1+\vec{b}_1)}\frac{\partial(W_1\vec{x}_1+\vec{b}_1)}{\partial\left[\text{vec}(W_1)\right]}\right) &= -(t-\vec{x}_3)^T W_2 \text{diag}(\vec{x}_2)\text{diag}(\vec{1}-\vec{x}_2)\frac{\partial(W_1\vec{x}_1+\vec{b}_1)}{\partial\left[\text{vec}(W_1)\right]} \\ &= -(t-\vec{x}_3)^T W_2 \text{diag}(\vec{x}_2)\text{diag}(\vec{1}-\vec{x}_2)(\vec{x}_1^T \otimes \mathbf{I}_{m\times m})\end{aligned} \tag{16}$$

So, the derivative of L with respect to the vectorised version of matrix W is

$$\frac{\partial L}{\partial\left[\text{vec}(W_1)\right]} = \boxed{-(\vec{t}-\vec{x}_3)^T W_2 \text{diag}(\vec{x}_2)\text{diag}(\vec{1}-\vec{x}_2)(\vec{x}_1^T \otimes \mathbf{I}_{m\times m})} \in \mathbb{R}^{1\times(m\cdot n)} \tag{17}$$

The shape of the result in Equation 17 is expected, since $\frac{\partial L}{\partial W_1}$ would have a shape of $\mathbb{R}^{n\times m}$ (note that $W_1 \in \mathbb{R}^{m\times n}$ and the "numerator layout" convention indicates that the shape of $\frac{\partial \vec{a}}{\partial A}$ has the same shape as A^T). The vectorisation process stacks columns of height m for n repetitions, so to find $\frac{\partial L}{\partial W_1}$ we can un-stack the vector to create a $\mathbb{R}^{n\times m}$ (note to self: $\frac{\partial L}{\partial W_1} \notin \mathbb{R}^{m\times n}$) matrix. In NumPy notation, the derivative would be:

$$\frac{\partial L}{\partial W_1} = \boxed{\frac{\partial L}{\partial \left[\text{vec}(W_1)\right]}.\text{reshape}(n,m)} \in \mathbb{R}^{n\times m}, \text{ for } W_1 \in \mathbb{R}^{m\times n} \tag{18}$$

Similarly,

$$\frac{\partial L}{\partial \left[\text{vec}(W_2)\right]} = \frac{\partial L}{\partial \vec{x}_3}\frac{\partial \vec{x}_3}{\partial \left[\text{vec}(W_2)\right]} = \frac{\partial L}{\partial \vec{x}_3}\frac{\partial \left[W_2\vec{x}_2 + \vec{b}_2\right]}{\partial \left[\text{vec}(W_2)\right]} = \boxed{-(\vec{t}-\vec{x}_3)^T\left(\vec{x}_2^T \otimes \mathbf{I}_{k\times k}\right) \in \mathbb{R}^{1\times(k\cdot m)}} \tag{19}$$

Since $W_2 \in \mathbb{R}^{k\times m}$, it follows from Equation 14 that $\vec{x}_2^T \otimes \mathbf{I}_{k\times k} \in \mathbb{R}^{k\times(k\cdot m)}$. Again, un-stacking $\frac{\partial L}{\partial[\text{vec}(W_2)]}$ to produce a matrix with shape $\mathbb{R}^{k\times m}$ (which is the shape of W_2^T) gives the desired derivative:

$$\frac{\partial L}{\partial W_2} = \boxed{\frac{\partial L}{\partial \left[\text{vec}(W_2)\right]}.\text{reshape}(m,k)} \in \mathbb{R}^{m\times k}, \text{ for } W_2 \in \mathbb{R}^{k\times m} \tag{20}$$

Note that my solutions can be transposed to match PyTorch's solutions. ***Please see the attached code for more information. I demonstrate there that my algorithms for calculating the vectorised derivatives reach the same results as the ones used by PyTorch.***

Problem 3.2 (5 pts) Replace the Sigmoid function with ReLU function. Given a data $\vec{x}_1 = [0,1,2]^T$, target value $\vec{t} = [1,2]^T$, weights and bias at this iteration are

$$W_1 = \begin{bmatrix} 3 & -1 & 1 \\ -5 & 2 & -1 \end{bmatrix}, \vec{b}_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, W_2 = \begin{bmatrix} 1 & -2 \\ -3 & 1 \end{bmatrix}, \vec{b}_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \tag{21}$$

Following the results of Problem 3.1, calculate the values of L, $\frac{\partial L}{\partial W_1}$, $\frac{\partial L}{\partial W_2}$, $\frac{\partial L}{\partial \vec{b}_1}$, $\frac{\partial L}{\partial \vec{b}_2}$.

To calculate L, first calculate $\vec{x}_3$:

$$\vec{x}_3 = W_2\vec{x}_2 + \vec{b}_2 = W_2\text{ReLU}(W_1\vec{x}_1 + \vec{b}_1) + \vec{b}_2 = [-1,-3]^T \tag{22}$$

So, $L = 0.5(\vec{t}-\vec{x}_3)^T(\vec{t}-\vec{x}_3) = \boxed{14.5}$.
The symbolic expressions for the derivatives are as follows. Note that $\vec{z} = W_1\vec{x}_1 + \vec{b}_1$ and $\text{ReLU}(\vec{z}) = \vec{x}_2$:

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial \vec{x}_3}\frac{\partial \vec{x}_3}{\partial \vec{x}_2}\left[\frac{\partial([\text{ReLU}(\vec{z})])}{\partial \vec{z}}\frac{\partial \vec{z}}{\partial W_1}\right] = \left[-(\vec{t}-\vec{x}_3)^T W_2 \text{diag}\left(\vec{v}\right)\left(\vec{x}_1^T \otimes \mathbf{I}_{2\times 2}\right)\right].\text{reshape}(3,2) \in \mathbb{R}^{3\times 2} \tag{23}$$

where $\vec{v} = [v_1, v_2]^T$ ($\vec{v}$ has two coordinates since $\vec{x}_2 \in \mathbb{R}^{2\times 1}$) for $v_i = \begin{cases} 1 & z_i > 0 \\ 0 & \text{else} \end{cases}$. Using English, $\frac{\partial([\text{ReLU}(\vec{z})])}{\partial \vec{z}} = \text{diag}(\vec{v})$ is a diagonal matrix where each diagonal value is 1 if the corresponding coordinate in $\vec{z} = W_1\vec{x}_1 + \vec{b}_1$ is greater than 0, and 0 otherwise. Equations 24, 25, and 26 summarise the derivatives symbolically using the above definition for $\vec{v}$.

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial \vec{x}_3}\frac{\partial \vec{x}_3}{\partial W_2} = \left[-(\vec{t}-\vec{x}_3)^T\left(\vec{x}_2^T \otimes \mathbf{I}_{2\times 2}\right)\right].\text{reshape}(2,2) \in \mathbb{R}^{2\times 2} \tag{24}$$

$$\frac{\partial L}{\partial \vec{b}_1} = \frac{\partial L}{\partial \vec{x}_3}\frac{\partial \vec{x}_3}{\partial \vec{x}_2}\frac{\partial\left[\text{ReLU}(\vec{z})\right]}{\partial \vec{z}}\frac{\partial \vec{z}}{\partial \vec{b}_1} = -(\vec{t}-\vec{x}_3)^T W_2 \text{diag}(\vec{v}) \in \mathbb{R}^{1\times 2} \tag{25}$$

$$\frac{\partial L}{\partial \vec{b}_2} = \frac{\partial L}{\partial \vec{x}_3}\frac{\partial \vec{x}_3}{\partial \vec{b}_2} = -(\vec{t} - \vec{x}_3)^T \in \mathbb{R}^{1\times 2} \tag{26}$$

The components that are needed and they are listed below (Equations 27 to 31).

$$\frac{\partial L}{\partial \vec{x}_3} = -(\vec{t} - \vec{x}_3)^T = \begin{bmatrix} -2 & -5 \end{bmatrix} \tag{27}$$

$$\begin{aligned} &\frac{\partial \vec{x}_3}{\partial W_2} = \frac{\partial \left(W_2\vec{x}_2 + \vec{b}_2\right)}{\partial W_1} = \left(\vec{x}_2^T \otimes \mathbf{I}_{2\times 2}\right) = \begin{bmatrix} 2 & 0 & 2 & 0 \\ 0 & 2 & 0 & 2 \end{bmatrix} \\ &\text{similarly, } \frac{\partial \vec{z}}{\partial W_1} = \frac{\partial \left(W_1\vec{x}_1 + \vec{b}_1\right)}{\partial W_1} = \left(\vec{x}_1^T \otimes \mathbf{I}_{2\times 2}\right) = \begin{bmatrix} 0 & 0 & 1 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 & 0 & 2 \end{bmatrix} \end{aligned} \tag{28}$$

$$\frac{\partial \vec{x}_3}{\partial \vec{x}_2} = W_2 = \begin{bmatrix} 1 & -2 \\ -3 & 1 \end{bmatrix} \tag{29}$$

$$\frac{\partial \left[\text{ReLU}(\vec{z})\right]}{\partial \vec{z}} = \frac{\partial \left[\text{ReLU}\left(W_1\vec{x}_1 + \vec{b}_1\right)\right]}{\partial \left(W_1\vec{x}_1 + \vec{b}_1\right)} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \text{ given that } W_1\vec{x}_1 + \vec{b}_1 = \begin{bmatrix} 2 \\ 2 \end{bmatrix} \tag{30}$$

$$\frac{\partial \vec{z}}{\partial \vec{b}_1} = \frac{\partial \left(W_1\vec{x}_1 + \vec{b}_1\right)}{\partial \vec{b}_1} = \mathbf{I}_{2\times 2}; \text{ similarly, } \frac{\partial \vec{x}_3}{\partial \vec{b}_2} = \frac{\partial \left(W_2\vec{x}_2 + \vec{b}_2\right)}{\partial \vec{b}_2} = \mathbf{I}_{2\times 2} \tag{31}$$

Here is the intermediate output:

$$\vec{x}_2 = \text{ReLU}\left(\begin{bmatrix} 2 \\ 2 \end{bmatrix}\right) = \begin{bmatrix} 2 \\ 2 \end{bmatrix} \tag{32}$$

Using the intermediate derivatives in Equations 27 to 31 and the symbolic expressions in Equations 23 to 26, we can calculate everything:

$$\boxed{\frac{\partial L}{\partial W_1} = \begin{bmatrix} 0 & 0 \\ 13 & -1 \\ 26 & -2 \end{bmatrix}, \frac{\partial L}{\partial W_2} = \begin{bmatrix} -4 & -10 \\ -4 & -10 \end{bmatrix}, \frac{\partial L}{\partial \vec{b}_1} = \begin{bmatrix} 13 & -1 \end{bmatrix}, \frac{\partial L}{\partial \vec{b}_2} = \begin{bmatrix} -2 & -5 \end{bmatrix}} \tag{33}$$

Since we used the "numerator layout convention", we can transpose the expressions to get the derivatives expressed using the "denominator layout convention" which is what PyTorch uses.

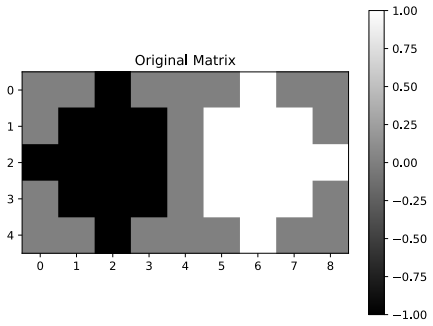
Please see the code attached for more information. I used the expressions above to compute the derivatives and checked my results with PyTorch.

4 2D Convolution (10 pts)

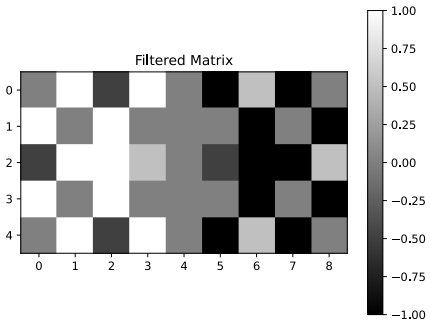
Problem 4.1 (5 pts) Derive the 2D convolution results of the following 5×9 input matrix and the 3×3 kernel. Consider 0s are padded around the input and the stride is 1, so that the output should also have shape 5×9.

$$\begin{bmatrix} 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & -1 & -1 & -1 & 0 & 1 & 1 & 1 & 0 \\ -1 & -1 & -1 & -1 & 0 & 1 & 1 & 1 & 1 \\ 0 & -1 & -1 & -1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \circledast \begin{bmatrix} 0.0 & -0.5 & 0.0 \\ -0.5 & 1.0 & -0.5 \\ 0.0 & -0.5 & 0.0 \end{bmatrix} = \boxed{\begin{bmatrix} 0 & 1 & -0.5 & 1 & 0 & -1 & 0.5 & -1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & -1 \\ -0.5 & 1 & 1 & 0.5 & 0 & -0.5 & -1 & -1 & 0.5 \\ 1 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & -1 \\ 0 & 1 & -0.5 & 1 & 0 & -1 & 0.5 & -1 & 0 \end{bmatrix}} \quad (34)$$

The padding is $P = 1$ for the output to be $\mathbb{R}^{5\times 9}$. The associated plots are shown in Figures 6a and 6b.



(a) Input matrix.



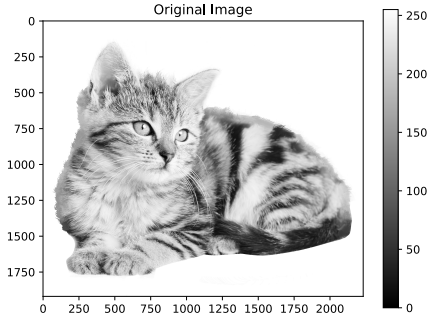
(b) Result of convolution.

Figure 6: Problem 4.1, filtering.

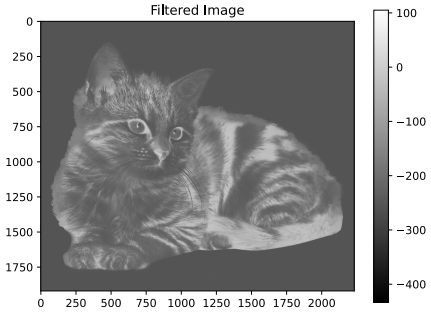
Please see the code attached for details on calculation and plotting.

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.

The output in Figure 6b seems to be partially inverted and blurred compared to the original matrix (Figure 6a). It is not a full inversion, however. The kernel turns a region that generally has positive values into a region that generally has negative values.



(a) Input image.



(b) Output image after filtering.

Figure 7: Problem 4.2, effects of the kernel.

Figures 7a and 7b demonstrate the effects of the filter. The filtered image has its pixel values inverted and has become slightly blurry due to the convolution. However, the filter doesn't fully invert the image.

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 Sakai.

To start with, please download the dataset.mat file from Sakai and load it into NumPy arraysa. There are two variables in the file: data $X \in \mathbb{R}^{100\times 3}$ and target $\vec{D} \in \mathbb{R}^{100\times 1}$. Each individual pair of data and target is composed into X and $\vec{D}$ following the same way as discussed on Lecture 2 Page 8. Specifically, each row in X correspond to the transpose of a data point, with the first element as constant 1 and the other two as the two input features x_{1k} and x_{2k}. The goal of the learning task is finding the weight vector $\vec{W} \in \mathbb{R}^{3\times 1}$ for the linear model that can minimize the MSE loss, which is also formulated on Lecture 2 Page 7.

Lab 1 (a) (3pt) Directly compute the least square (Wiener) solution with the provided dataset. What is the optimal weight $\vec{W}^*$? What is the MSE loss of the whole dataset when the weight is set to $\vec{W}^*$?

The MSE loss of the whole dataset using the optimal weight is $\boxed{5.03995157 \cdot 10^{-5}}$ using the definition $MSE = \frac{1}{2K}(\vec{D} - X\vec{W})^T(\vec{D} - X\vec{W})$, where $K =$ the number of data points. The optimal weight is

$$\vec{W}^* = \left(X^T X\right)^{-1} X^T \vec{D} = \boxed{\begin{bmatrix} 1.00067810 \\ 1.00061145 \\ -2.00031968 \end{bmatrix}}$$

Lab 1 (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 $\vec{W}_0 = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T$, 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 weight vector using the LMS algorithm after 20 epochs with $r = 0.01$ is

$$\vec{W}_{\text{LMS}} = \boxed{\begin{bmatrix} 1.00074855 \\ 1.00082859 \\ -2.00068123 \end{bmatrix}}$$

See Figure 8 for the plot of MSE vs epoch number when $r = 0.01$.

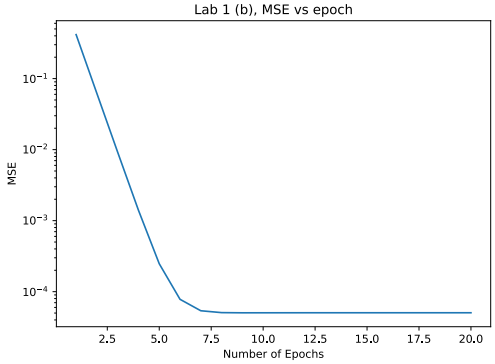


Figure 8: Lab 1 b) Plot of MSE (log scale) vs epoch, $r = 0.01$.

Lab 1 (c) (3pt) Scatter plot the points (x_{1k}, x_{2k}, d_k) for all 100 data-target pairs in a 3D figure, and plot the lines corresponding to the linear models you got in **(a)** and **(b)** respectively in the same figure. Observe if the linear models fit the data well.

Please see Figure 9 for the required plot. The yellow dashed line uses the Wiener solution's weights; the blue line uses the LMS algorithm's weights. Both linear model fit the data well (at least from this 3D perspective) since the two lines are surrounded by data points.

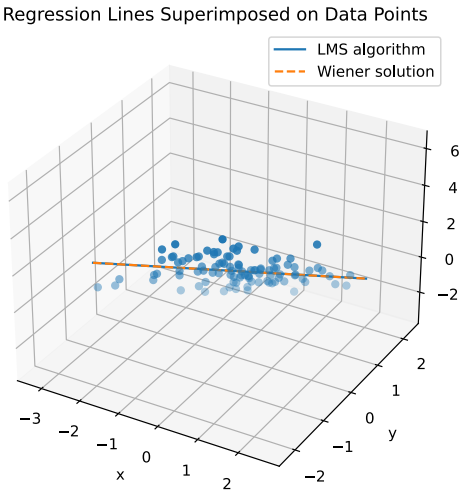


Figure 9: Lab 1 c) 3D plot of regression lines, superimposed on data.

Lab 1 (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.
Please see Figure 10 for the plots illustrating the relationship between MSE and epochs.

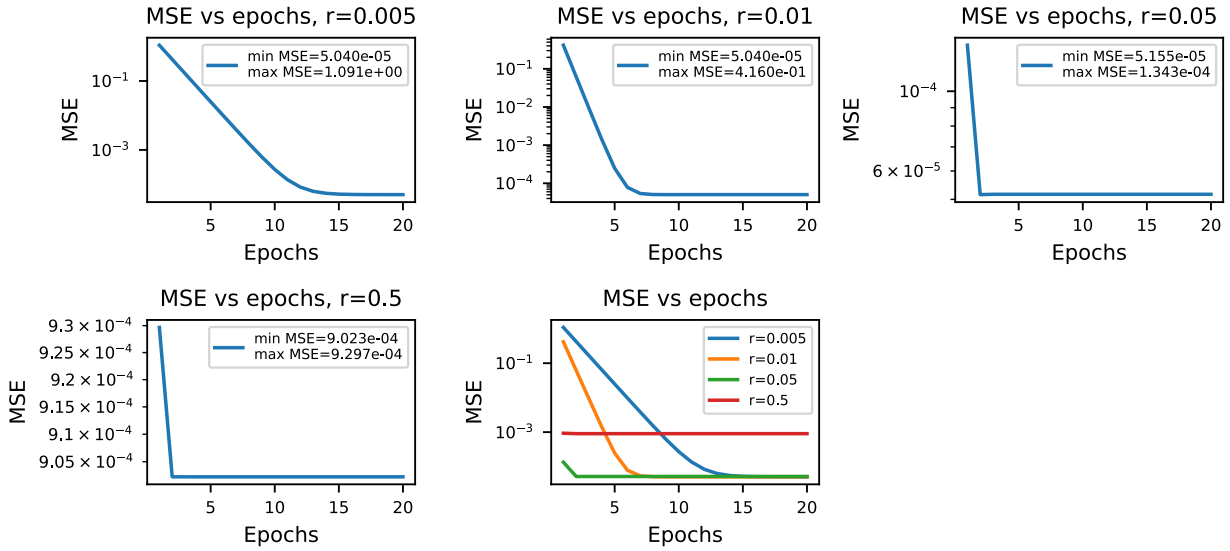


Figure 10: Lab 1 d) MSE versus epoch for $r = \{0.005, 0.01, 0.05, 0.5\}$.

When learning rate is too low (0.005), the algorithm approaches the best solution very slowly since the weights vector can only change a little bit each time a data point is fed into the algorithm. We can see that the MSE after the first epoch is much higher for $r = 0.005$ than it is for $r = 0.5$. Although we get a pretty small MSE after the first epoch when learning rate is high, the LMS algorithm is unable to further decrease the MSE by much after the first couple epochs. This results in a suboptimal final weight when r is too high, since the high learning rate causes the algorithm to overshoot and oscillate around the optimal solution. When learning rate is very high (e.g., $r = 0.7$ in Figure 11), the algorithm completely misses the global minimum and the MSE shoots off to infinity. So, when the learning rate is too small, the training speed is too low, and when the learning rate is too large, the training quality decreases (i.e., higher MSE). When learning rate was $r = 1$, NumPy complained because matrix multiplications overflowed due to the LMS algorithm overshooting the minimum. If the algorithm had run for more than 20 epochs, the run with $r = 0.7$ would also overflow at some point.

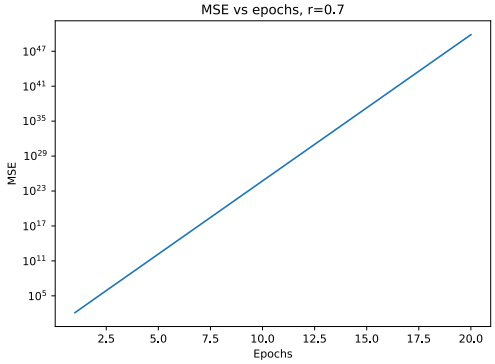


Figure 11: Lab 1 d) MSE versus epoch for $r = 0.7$, demonstrate overshooting the minimum.

6 Lab: Simple NN (40 pts)

Lab 2 (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.

```
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, dilation=1, groups=1,
                          bias=False, padding_mode='zeros')
        self.conv2 = CONV(in_channels=16, out_channels=16, kernel_size=3, stride=1,
                          padding=2, dilation=1, groups=1,
                          bias=False, padding_mode='zeros')
        self.conv3 = CONV(in_channels=16, out_channels=32, kernel_size=7, stride=1,
                          padding=2, dilation=1, groups=1,
                          bias=False, padding_mode='zeros')
        self.fc1   = FC(in_features=288, out_features=32, bias=True)
        self.fc2   = FC(in_features=32, out_features=10, bias=True)

    def forward(self, x):
        #print(x.size())
        # Forward pass computation
        # Conv 1
        out = F.relu(self.conv1(x))#; print(out.size())
        # MaxPool
        out = F.max_pool2d(out, 4, stride=2)#; print(out.size())
        # Conv 2
        out = F.relu(self.conv2(out)); print(out.size())
        # MaxPool
        out = F.max_pool2d(out, 3, stride=2)#; print(out.size())
        # Conv 3
        out = F.relu(self.conv3(out)); print(out.size())
        # MaxPool
        out = F.max_pool2d(out, 2, stride=2)#; print(out.size())
        # Flatten
        out = out.view(out.size(0), -1)#; print(out.size())
        # FC 1
        out = F.relu(self.fc1(out))#; print(out.size())
        # FC 2
        out = F.relu(self.fc2(out))#; print(out.size())
        return out
```

Lab 2 (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 1.

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

Table 1: Lab 2 b) Required information about each layer.

Lab 3 (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.

Please see Figure 12 for the required plot.

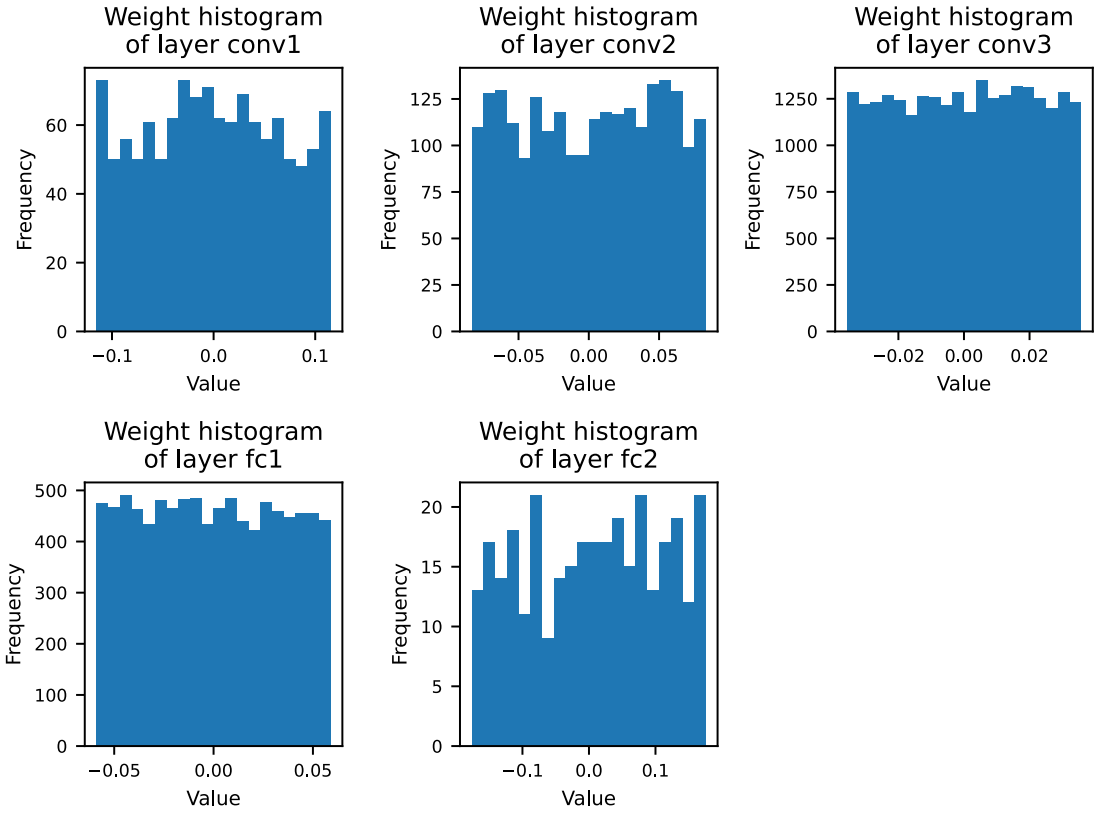


Figure 12: Lab 3 a) histogram of weight elements.

Lab 3 (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.

Please see Figure 13 for the required plots.

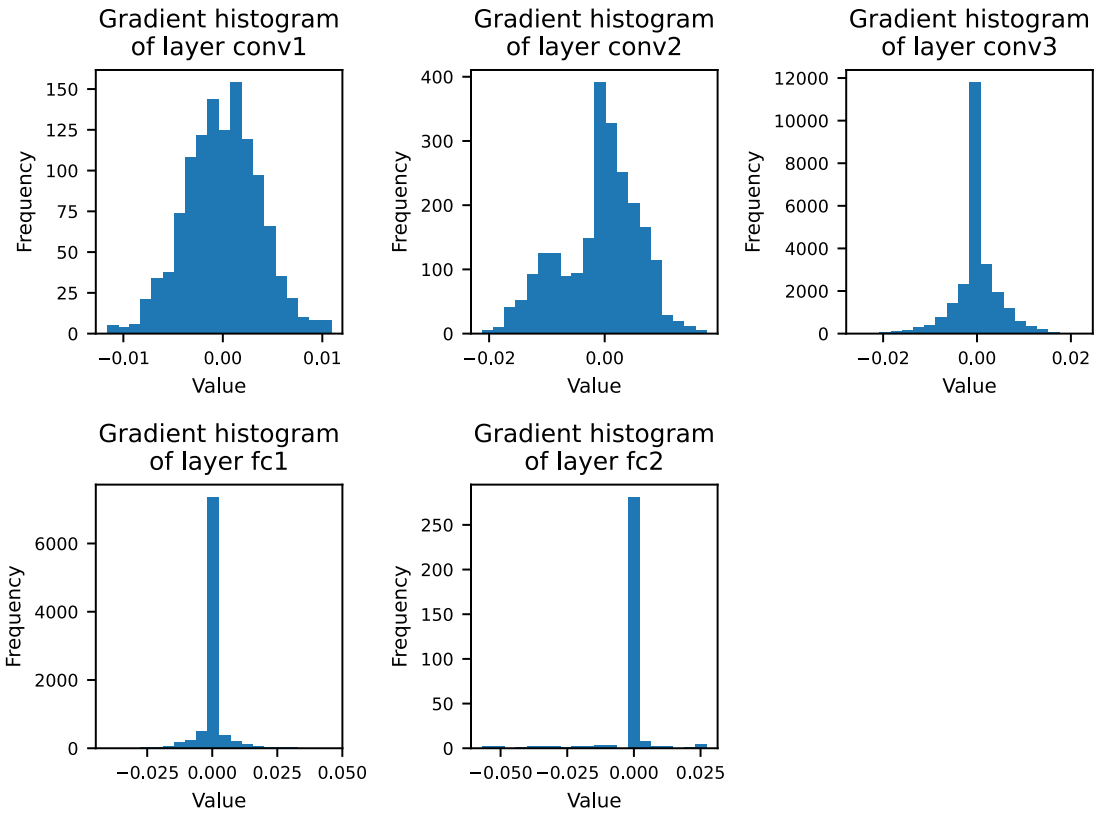


Figure 13: Lab 3 b) histogram of weight elements' gradients.

Lab 3 (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.

Please see Figure 14 for the required plots.

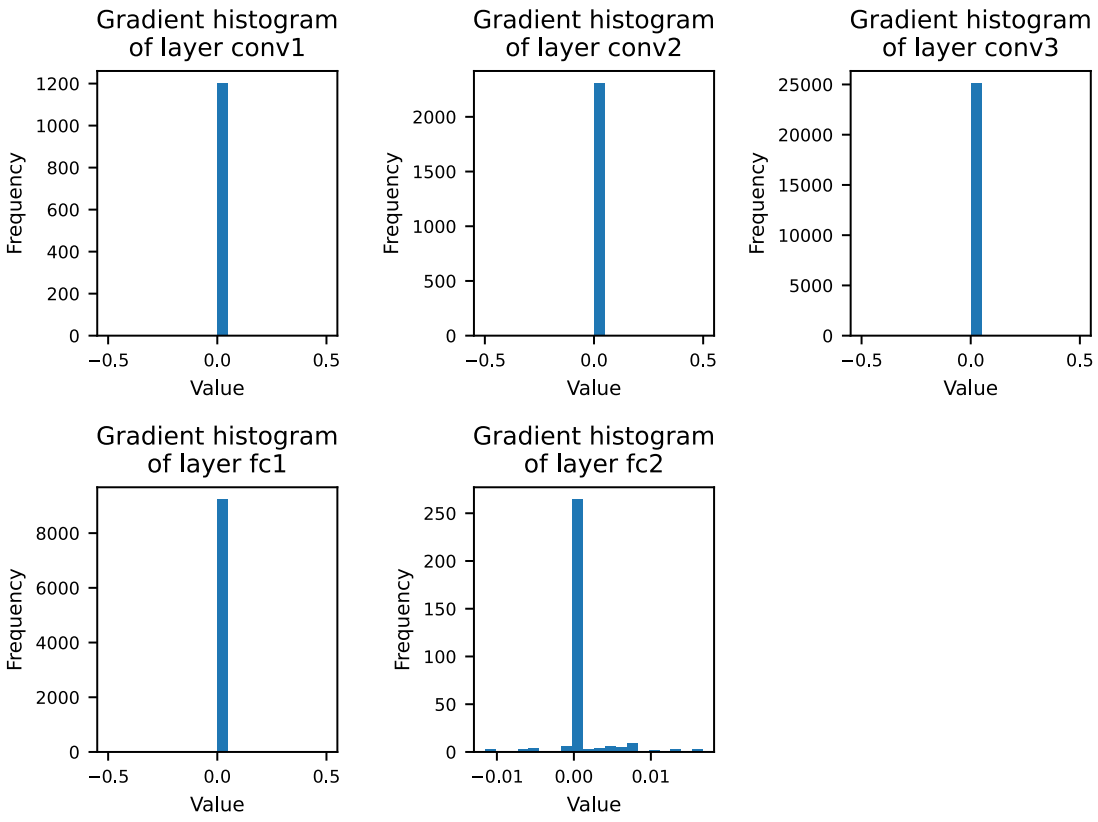


Figure 14: Lab 3 c) histogram of weight elements' gradients when weights are initialised to be identically zero.

The weights for all the convolution layers and the first FC layer are zero compared to lab 3b), which are approximately centred at zero but not identically zero. When using ReLU activation, the gradient with respect to the weights tensor is zero if every input to the ReLU module is zero. When the initial weights tensor is zero, the convolution of the input feature map with the weights tensor is also zero when the bias is zero. Since the identically-zero result of the convolution is fed into the ReLU function, the output of the ReLU function will also be zero. When using the chain rule to compute the derivative of loss with respect to the weights tensor, one of the intermediate gradients that appear is the gradient of the ReLU function with respect to its input, which becomes identically zero if the inputs to the ReLU function are all zero. This is seen in Question 3 in Equation 23, where the derivative $\frac{\partial[\text{ReLU}(\vec{z})]}{\partial \vec{z}}$ is the zero matrix when every element of $\vec{z}$ is smaller than or equal to 0 since regardless of how $\vec{z} < \vec{0}$ changes, as long as the expression $\vec{z} < \vec{0}$ is true, the output $\text{ReLU}(\vec{z})$ is zero. Therefore, the gradient $\frac{\partial[\text{ReLU}(\vec{z})]}{\partial \vec{z}}$ is the zero matrix. Multiplication by a zero tensor causes the entire gradient to be zero as well. We didn't initialise the weights vector in lab 3 b) as identically zero, so the model doesn't have gradients that are identically zero for the convolutional layers.

When using ReLU activation, initialising all the weight vectors to zero would cause all neurons to be dead since the gradient of loss with respect to a weight tensor is zero. This means that the weights will never be updated using an iterative algorithm since the algorithm doesn't know which direction leads to lower loss. Therefore, training doesn't produce much improvement.

section2

September 17, 2023

[6]:
```
import numpy as np
from sklearn.linear_model import LinearRegression
```

[7]:
```
def fit_and_return_coef(x, y):
    model = LinearRegression()
    model.fit(x, y)  # x is (n_samples, n_features), y is (n_samples, n_targets)
    # print(model.score(x, y))
    return model.intercept_, model.coef_


def check_fitting(x, intercept, weights):
    return np.matmul(x, weights) + intercept


# q2.2:
x_2 = np.array([[-1, -1], [-1, +1], [+1, -1], [+1, +1]])
y_2 = np.array([+1, +1, +1, -1])
w0, coefs = fit_and_return_coef(x_2, y_2)

print(check_fitting(x_2, w0, coefs))
print(w0, coefs)
```

```
[ 1.5  0.5  0.5 -0.5]
0.5 [-0.5 -0.5]
```

[8]:
```
# q2.3:
x_3 = np.array([[-1, -1, -1],
                [-1, -1, +1],
                [-1, +1, -1],
                [-1, +1, +1],
                [+1, -1, -1],
                [+1, -1, +1],
                [+1, +1, -1],
                [+1, +1, +1]])
y_3 = np.array([-1, -1, -1, +1, -1, +1, +1, +1])
w0, coefs = fit_and_return_coef(x_3, y_3)

print(check_fitting(x_3, w0, coefs))
```

```
print(w0, coefs)
```

```
[-1.5 -0.5 -0.5  0.5 -0.5  0.5  0.5  1.5]
0.0 [0.5 0.5 0.5]
```

[9]:
```
# q2.4 A xor B = (A nand B) and (A or B)
x_4 = np.array([[-1, -1], [-1, +1], [+1, -1], [+1, +1]])
y_41 = np.matmul(x_4, np.array([-1, +1])) + 0.5
y_42 = np.matmul(x_4, np.array([+1, -1])) + 0.5
x_4final = np.concatenate((y_41, y_42)).reshape(2, 4).T

print('before masking (outputs from layer 1):\n' + str(x_4final))

x_4final[x_4final > 0] = +1
x_4final[x_4final < 0] = -1
y_4final = np.array([-1, +1, +1, -1])
# print(y_41, y_42)
print('after masking (outputs from layer 1):\n'  + str(x_4final))
intercept, arr = fit_and_return_coef(x_4final, y_4final)

# print('testing ' + str(np.matmul(x_4final, np.array(arr)) + intercept))

print('fitting xor (w20, w21, w22) = ' + str(fit_and_return_coef(x_4final,␣
 ↪y_4final)))
print(str(check_fitting(x_4final, *fit_and_return_coef(x_4final, y_4final))))

# print('fit xor ' + str(fit_and_return_coef(x_4_part_2, y_4final)))
# print('check xor ' + str(check_fitting(x_4_part_2,␣
 ↪*fit_and_return_coef(x_4_part_2, y_4final))))
```

```
before masking (outputs from layer 1):
[[ 0.5  0.5]
 [ 2.5 -1.5]
 [-1.5  2.5]
 [ 0.5  0.5]]
after masking (outputs from layer 1):
[[ 1.  1.]
 [ 1. -1.]
 [-1.  1.]
 [ 1.  1.]]
fitting xor (w20, w21, w22) = (1.0, array([-1., -1.]))
[-1.  1.  1. -1.]
```

section3

September 17, 2023

[146]:
```
import numpy as np
import torch
import torch.functional as F
import array_to_latex as a2l

'''Solving Q3 using the chain rule and PyTorch, good for sigmoid and ReLU'''

def my_relu(vec):
    return np.array([max(0, v) for v in vec.flatten()]).reshape(-1, 1)

def a2l_short(a):  # prints out nice latex
    return a2l.to_ltx(a, frmt='{:.2f}', arraytype='bmatrix', print_out=False)

def my_relu_derivative(vec):
    ret = np.zeros((len(vec), len(vec)))  # square matrix
    relu_vec = my_relu(vec)
    # if ReLu(vec_i) > 0 then ret_{i,i} = 1, ret_{i,i} = 0 otherwise
    relu_vec[relu_vec > 0] = 1
    relu_vec[relu_vec <= 0] = 0
    np.fill_diagonal(a=ret, val=relu_vec)
    return ret

def my_sigmoid_derivative(vec):  # computes diag(vec)diag(1-vec)
    ret1 = np.zeros((len(vec), len(vec)))
    ret2 = np.zeros((len(vec), len(vec)))
    np.fill_diagonal(a=ret1, val=vec)
    np.fill_diagonal(a=ret2, val=1-vec)
    return np.matmul(ret1, ret2)

def my_run(relu_or_sigmoid, wrt, verbose=False, rand=False):
    if not rand:
        x1 = np.array([0., 1., 2.]).reshape(-1, 1)
        W1 = np.array([[ 3., -1,  1],
                       [-5.,  2, -1]])
        b1 = np.array([1., 2]).reshape(-1, 1)

        W2 = np.array([[ 1., -2],
```

```
                      [-3.,  1]])
        b2 = np.array([1., 1]).reshape(-1, 1)
        t = np.array([1., 2]).reshape(-1, 1)
    else:
        shape1 = np.random.randint(2, 12)
        shape2 = np.random.randint(2, 12)
        x1 = np.random.rand(1, shape1).reshape(-1, 1) * np.random.
↪randint(low=-5, high=5)
        W1 = np.random.rand(shape2, shape1) * np.random.randint(low=-5, high=5)
        b1 = np.random.rand(1, shape2).reshape(-1, 1) * np.random.
↪randint(low=-5, high=5)

        shape3 = np.random.randint(2, 12)
        W2 = np.random.rand(shape3, shape2) * np.random.randint(low=-5, high=5)
        b2 = np.random.rand(1, shape3).reshape(-1, 1) * np.random.
↪randint(low=-5, high=5)
        t = np.random.rand(1, shape3).reshape(-1, 1) * np.random.
↪randint(low=-5, high=5)
    if verbose:
        print(x1.shape, W1.shape, b1.shape, W2.shape, b2.shape, t.shape)
    if relu_or_sigmoid == 'relu':
        # print('here', (np.matmul(W1, x1) + b1).shape)
        x2 = my_relu(np.matmul(W1, x1) + b1)
    elif relu_or_sigmoid == 'sigmoid':
        x2 = 1/(1 + np.exp(-(np.matmul(W1, x1) + b1)))
    else:
        raise KeyError('relu or sigmoid')
    x3 = np.matmul(W2, x2) + b2

    mse_here = 0.5 * np.linalg.norm(t - x3) ** 2
    print('mse=' + str(mse_here))

    if relu_or_sigmoid == 'relu':
        step_f_x2_wrt_z = my_relu_derivative(x2)  # = diag(v)
        # dReLU(z)/dz as matrix
    if relu_or_sigmoid == 'sigmoid':
        step_f_x2_wrt_z = my_sigmoid_derivative(x2)
        # dSigma(z)/dz as matrix
    if wrt == 'W1':  # see equation 23
        dL_dW1 = np.matmul(np.matmul(-(t - x3).T, W2),
                           np.matmul(step_f_x2_wrt_z, np.kron(x1.T, np.
↪identity(W1.shape[0]))))
        print('dL/d' + wrt + ', res before re-shaping\n', a2l_short(dL_dW1))
        print('res after re-shaping\n', a2l_short(dL_dW1.reshape(W1.shape[1],␣
↪W1.shape[0])))
    if wrt == 'W2':  # see equation 24
```

```
        print('dL/d' + wrt + ', res before re-shaping\n',
              a2l_short(np.matmul(-(t - x3).T, np.kron(x2.T, np.identity(W2.
 ↪shape[0])))))
        print('res after re-shaping\n',
              a2l_short(np.matmul(-(t - x3).T,
                        np.kron(x2.T, np.identity(W2.shape[0])))).reshape(W2.
 ↪shape[1], W2.shape[0]))
    if wrt == 'b1':  # see equation 25
        print('dL/d' + wrt, a2l_short(np.matmul(np.matmul(-(t - x3).T, W2),␣
 ↪step_f_x2_wrt_z)))
    if wrt == 'b2':  # see equation 26
        print('dL/d' + wrt, a2l_short(-(t - x3).T))

    if verbose:  # prints out intermediate steps
        print('dL/dx3 = -(t - x3)^T=\n', a2l_short(-(t - x3).T))
        print('dx3/dW2=x2^T kron I_2x2=\n', a2l_short(np.kron(x2.T, W2.
 ↪shape[0])))
        print('d(z)/dW1=d(W1x1+b1)/dW1=x1^T kron I_2x2=\n', a2l_short(np.
 ↪kron(x1.T, W1.shape[0])))
        print('dx3/dx2\n', a2l_short(W2))
        print('dReLU(z)/dz=dReLU(W1x1+b1)/dW1=\n', a2l_short(step_f_x2_wrt_z))
        print('z=W1x1+b1\n', a2l_short(np.matmul(W1, x1) + b1))
        print('x2=ReLU(z)=\n', a2l_short(x2))
        print('x3=\n', a2l_short(x3))
    return x1, W1, b1, W2, b2, t


def compute_grad_pytorch(relu_or_sigmoid, wrt,
                         x1=None, W1=None, b1=None, W2=None, b2=None, t=None,
                         rand=False):
    if not rand:  # uses whatever is given in Q3.2
        x1_ = np.array([0., 1., 2.]).reshape(-1, 1)
        b1_ = np.array([1., 2]).reshape(-1, 1)
        W1_ = np.array([[ 3., -1,  1],
                        [-5.,  2, -1]])

        b2_ = np.array([1., 1]).reshape(-1, 1)
        W2_ = np.array([[ 1., -2],
                        [-3.,  1]])
        t_ = np.array([1., 2]).reshape(-1, 1)
    else:  # these vars come from my_run
        x1_ = x1
        W1_ = W1
        b1_ = b1

        W2_ = W2
        b2_ = b2
```

```
        t_ = t

    x1n = torch.tensor(x1_, requires_grad=True)
    b1n = torch.tensor(b1_, requires_grad=True)
    b2n = torch.tensor(b2_, requires_grad=True)
    W1n = torch.tensor(W1_, requires_grad=True)
    W2n = torch.tensor(W2_, requires_grad=True)

    x1_after = torch.matmul(W1n, x1n) + b1n

    if relu_or_sigmoid == 'relu':
        x2n = torch.relu(x1_after)
    else:
        x2n = torch.sigmoid(x1_after)
    x3n = torch.matmul(W2n, x2n) + b2n
    tn = torch.tensor(t_, requires_grad=True)
    mse = 0.5 * F.norm(tn - x3n)**2
    print(mse)
    which_param = {'W1': W1n, 'W2': W2n, 'b1': b1n, 'b2': b2n}
    mse.backward(inputs=which_param[wrt])
    print(relu_or_sigmoid + ' dL/d' + wrt + '\n', which_param[wrt].grad.T, '\n')
    return


my_run(relu_or_sigmoid='relu', wrt='W1')
compute_grad_pytorch(relu_or_sigmoid='relu', wrt='W1')

my_run(relu_or_sigmoid='relu', wrt='W2')
compute_grad_pytorch(relu_or_sigmoid='relu', wrt='W2')

my_run(relu_or_sigmoid='relu', wrt='b1')
compute_grad_pytorch(relu_or_sigmoid='relu', wrt='b1')

my_run(relu_or_sigmoid='relu', wrt='b2', verbose=True)
compute_grad_pytorch(relu_or_sigmoid='relu', wrt='b2')
```

```
mse=14.499999999999998
dL/dW1, res before re-shaping
 \begin{bmatrix}
  0.00 &  0.00 &  13.00 & -1.00 &  26.00 & -2.00
\end{bmatrix}
res after re-shaping
 \begin{bmatrix}
  0.00 &  0.00\\
  13.00 & -1.00\\
  26.00 & -2.00
\end{bmatrix}
```

```
tensor(14.5000, dtype=torch.float64, grad_fn=<MulBackward0>)
relu dL/dW1
 tensor([[ 0.0000,  0.0000],
        [13.0000, -1.0000],
        [26.0000, -2.0000]], dtype=torch.float64)

mse=14.499999999999998
dL/dW2, res before re-shaping
 \begin{bmatrix}
 -4.00 & -10.00 & -4.00 & -10.00
\end{bmatrix}
res after re-shaping
 \begin{bmatrix}
 -4.00 & -10.00\\
 -4.00 & -10.00
\end{bmatrix}
tensor(14.5000, dtype=torch.float64, grad_fn=<MulBackward0>)
relu dL/dW2
 tensor([[ -4.0000, -10.0000],
        [ -4.0000, -10.0000]], dtype=torch.float64)

mse=14.499999999999998
dL/db1 \begin{bmatrix}
  13.00 & -1.00
\end{bmatrix}
tensor(14.5000, dtype=torch.float64, grad_fn=<MulBackward0>)
relu dL/db1
 tensor([[13.0000, -1.0000]], dtype=torch.float64)

(3, 1) (2, 3) (2, 1) (2, 2) (2, 1) (2, 1)
mse=14.499999999999998
dL/db2 \begin{bmatrix}
 -2.00 & -5.00
\end{bmatrix}
dL/dx3 = -(t - x3)^T=
 \begin{bmatrix}
 -2.00 & -5.00
\end{bmatrix}
dx3/dW2=x2^T kron I_2x2=
 \begin{bmatrix}
  4.00 &  4.00
\end{bmatrix}
d(z)/dW1=d(W1x1+b1)/dW1=x1^T kron I_2x2=
 \begin{bmatrix}
  0.00 &  2.00 &  4.00
\end{bmatrix}
dx3/dx2
 \begin{bmatrix}
```

```
  1.00 & -2.00\\
 -3.00 &  1.00
\end{bmatrix}
dReLU(z)/dz=dReLU(W1x1+b1)/dW1=
 \begin{bmatrix}
  1.00 &  0.00\\
  0.00 &  1.00
\end{bmatrix}
z=W1x1+b1
 \begin{bmatrix}
  2.00\\
  2.00
\end{bmatrix}
x2=ReLU(z)=
 \begin{bmatrix}
  2.00\\
  2.00
\end{bmatrix}
x3=
 \begin{bmatrix}
 -1.00\\
 -3.00
\end{bmatrix}
tensor(14.5000, dtype=torch.float64, grad_fn=<MulBackward0>)
relu dL/db2
 tensor([[-2.0000, -5.0000]], dtype=torch.float64)
```

[147]:
```
my_run(relu_or_sigmoid='sigmoid', wrt='W1')
compute_grad_pytorch(relu_or_sigmoid='sigmoid', wrt='W1')

my_run(relu_or_sigmoid='sigmoid', wrt='W2')
compute_grad_pytorch(relu_or_sigmoid='sigmoid', wrt='W2')

my_run(relu_or_sigmoid='sigmoid', wrt='b1')
compute_grad_pytorch(relu_or_sigmoid='sigmoid', wrt='b1')

my_run(relu_or_sigmoid='sigmoid', wrt='b2')
compute_grad_pytorch(relu_or_sigmoid='sigmoid', wrt='b2')
```

```
mse=4.201102887391705
dL/dW1, res before re-shaping
 \begin{bmatrix}
  0.00 &  0.00 &  0.78 & -0.10 &  1.55 & -0.21
\end{bmatrix}
res after re-shaping
 \begin{bmatrix}
  0.00 &  0.00\\
```

```
  0.78 & -0.10\\
  1.55 & -0.21
\end{bmatrix}
tensor(4.2011, dtype=torch.float64, grad_fn=<MulBackward0>)
sigmoid dL/dW1
 tensor([[ 0.0000,  0.0000],
        [ 0.7774, -0.1050],
        [ 1.5547, -0.2100]], dtype=torch.float64)

mse=4.201102887391705
dL/dW2, res before re-shaping
 \begin{bmatrix}
 -0.78 & -2.43 & -0.78 & -2.43
\end{bmatrix}
res after re-shaping
 \begin{bmatrix}
 -0.78 & -2.43\\
 -0.78 & -2.43
\end{bmatrix}
tensor(4.2011, dtype=torch.float64, grad_fn=<MulBackward0>)
sigmoid dL/dW2
 tensor([[-0.7758, -2.4324],
        [-0.7758, -2.4324]], dtype=torch.float64)

mse=4.201102887391705
dL/db1 \begin{bmatrix}
  0.78 & -0.10
\end{bmatrix}
tensor(4.2011, dtype=torch.float64, grad_fn=<MulBackward0>)
sigmoid dL/db1
 tensor([[ 0.7774, -0.1050]], dtype=torch.float64)

mse=4.201102887391705
dL/db2 \begin{bmatrix}
 -0.88 & -2.76
\end{bmatrix}
tensor(4.2011, dtype=torch.float64, grad_fn=<MulBackward0>)
sigmoid dL/db2
 tensor([[-0.8808, -2.7616]], dtype=torch.float64)
```

[148]:
```
# testing with random inputs
x1, W1, b1, W2, b2, t = my_run(relu_or_sigmoid='relu', wrt='W1', rand=True)
compute_grad_pytorch(relu_or_sigmoid='relu', wrt='W1', rand=True,
                     x1=x1, W1=W1, b1=b1, W2=W2, b2=b2, t=t)

x1, W1, b1, W2, b2, t = my_run(relu_or_sigmoid='relu', wrt='W2', rand=True)
```

```
compute_grad_pytorch(relu_or_sigmoid='relu', wrt='W2', rand=True,
                     x1=x1, W1=W1, b1=b1, W2=W2, b2=b2, t=t)

x1, W1, b1, W2, b2, t = my_run(relu_or_sigmoid='relu', wrt='b1', rand=True)
compute_grad_pytorch(relu_or_sigmoid='relu', wrt='b1', rand=True,
                     x1=x1, W1=W1, b1=b1, W2=W2, b2=b2, t=t)
```

```
mse=4284.1927808565515
dL/dW1, res before re-shaping
 \begin{bmatrix}
 -59.28 & -75.72 & -209.31 & -290.40 & -370.93 & -1025.27 & -358.42 & -457.81 &
-1265.43 & -80.43 & -102.73 & -283.96 & -430.61 & -550.02 & -1520.29 & -321.76 &
-410.99 & -1136.00
\end{bmatrix}
res after re-shaping
 \begin{bmatrix}
 -59.28 & -75.72 & -209.31\\
 -290.40 & -370.93 & -1025.27\\
 -358.42 & -457.81 & -1265.43\\
 -80.43 & -102.73 & -283.96\\
 -430.61 & -550.02 & -1520.29\\
 -321.76 & -410.99 & -1136.00
\end{bmatrix}
tensor(4284.1928, dtype=torch.float64, grad_fn=<MulBackward0>)
relu dL/dW1
 tensor([[  -59.2843,   -75.7240,  -209.3059],
        [ -290.3996,  -370.9283, -1025.2691],
        [ -358.4225,  -457.8142, -1265.4272],
        [  -80.4290,  -102.7323,  -283.9585],
        [ -430.6100,  -550.0196, -1520.2886],
        [ -321.7636,  -410.9897, -1136.0013]], dtype=torch.float64)

mse=32.47118572498797
dL/dW2, res before re-shaping
 \begin{bmatrix}
  0.00 &  0.00 &  0.00 &  0.00 &  0.00 &  0.00 &  0.00 &  0.00 &  0.00 &  0.00 &
0.00 &  0.00 &  0.00 &  0.00 &  0.00 &  0.00 &  0.00 &  0.00 &  0.00 &  0.00 &
0.00 &  0.00 &  0.00 &  0.00 &  0.00 &  0.00 &  0.00 &  0.00 &  0.00 &  0.00 &
0.00 &  0.00
\end{bmatrix}
res after re-shaping
 \begin{bmatrix}
  0.00 &  0.00 &  0.00 &  0.00\\
  0.00 &  0.00 &  0.00 &  0.00\\
  0.00 &  0.00 &  0.00 &  0.00\\
  0.00 &  0.00 &  0.00 &  0.00\\
  0.00 &  0.00 &  0.00 &  0.00\\
```

```
  0.00 &  0.00 &  0.00 &  0.00\\
  0.00 &  0.00 &  0.00 &  0.00\\
  0.00 &  0.00 &  0.00 &  0.00
\end{bmatrix}
tensor(32.4712, dtype=torch.float64, grad_fn=<MulBackward0>)
relu dL/dW2
 tensor([[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]], dtype=torch.float64)

mse=23.41035200715275
dL/db1 \begin{bmatrix}
  0.00 &  0.00 &  0.00 &  0.00 &  0.00 &  0.00 &  0.00
\end{bmatrix}
tensor(23.4104, dtype=torch.float64, grad_fn=<MulBackward0>)
relu dL/db1
 tensor([[0., 0., 0., 0., 0., 0., 0.]], dtype=torch.float64)
```

[148]:

section4

September 17, 2023

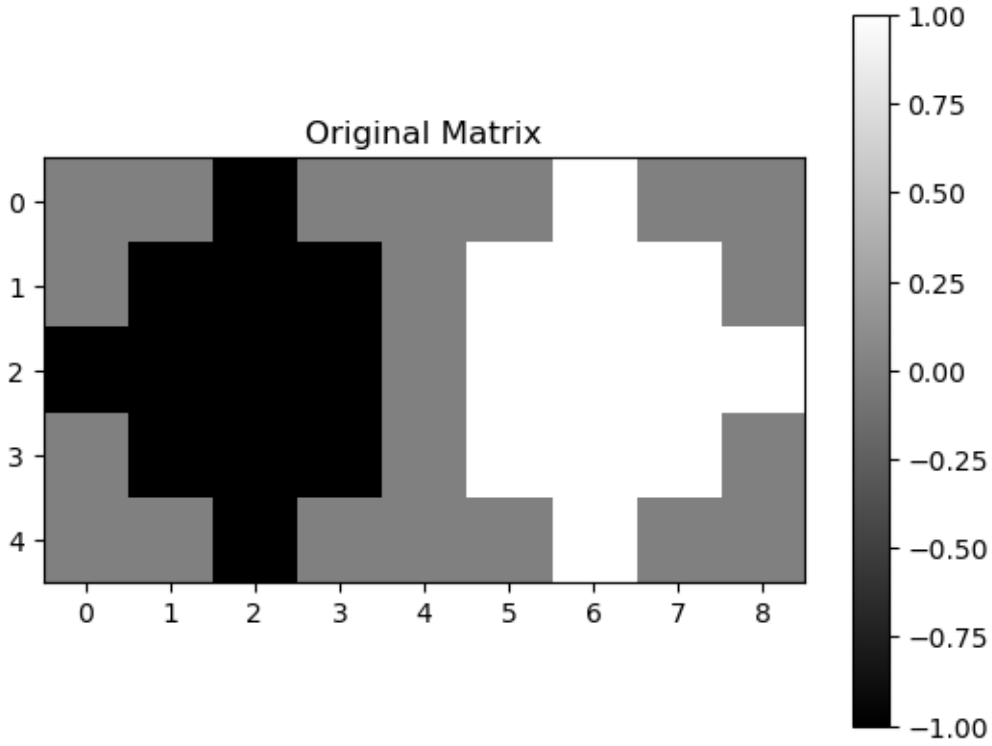
[69]:
```
from torch.nn.functional import conv2d
import torch
import numpy as np
import array_to_latex as a2l
import matplotlib.pyplot as plt
```

[70]:
```
data = np.array([[+0, +0, -1, +0, +0, +0, +1, +0, +0],
                 [+0, -1, -1, -1, +0, +1, +1, +1, +0],
                 [-1, -1, -1, -1, +0, +1, +1, +1, +1],
                 [+0, -1, -1, -1, +0, +1, +1, +1, +0],
                 [+0, +0, -1, +0, +0, +0, +1, +0, +0]])
a2l.to_ltx(data, frmt='{:d}', arraytype='bmatrix')
mat_in = torch.tensor(data)
mat_in = torch.unsqueeze(torch.unsqueeze(mat_in, 0), 0).float()
kernel = np.array(([  +0, -0.5,   +0],
                       [-0.5,   +1, -0.5],
                       [  +0, -0.5,   +0]))
a2l.to_ltx(kernel, frmt='{:.1f}', arraytype='bmatrix')
kernel = torch.tensor(kernel)
kernel = torch.unsqueeze(torch.unsqueeze(kernel, 0), 0).float()
kernel = kernel
# print(kernel)
# print(mat_in)
fig, ax = plt.subplots(1, 1)
mat_in = mat_in.squeeze()
p1 = ax.imshow(mat_in.squeeze(), cmap='gray')
ax.set_title('Original Matrix')
fig.colorbar(p1, ax=ax)
plt.savefig('q4_1_orig.pdf', dpi=500, bbox_inches='tight')

print('total energy=', sum(mat_in.flatten() ** 2))
```

```
\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}
  0.0 & -0.5 &  0.0\\
 -0.5 &  1.0 & -0.5\\
  0.0 & -0.5 &  0.0
\end{bmatrix}
total energy= tensor(24.)
```

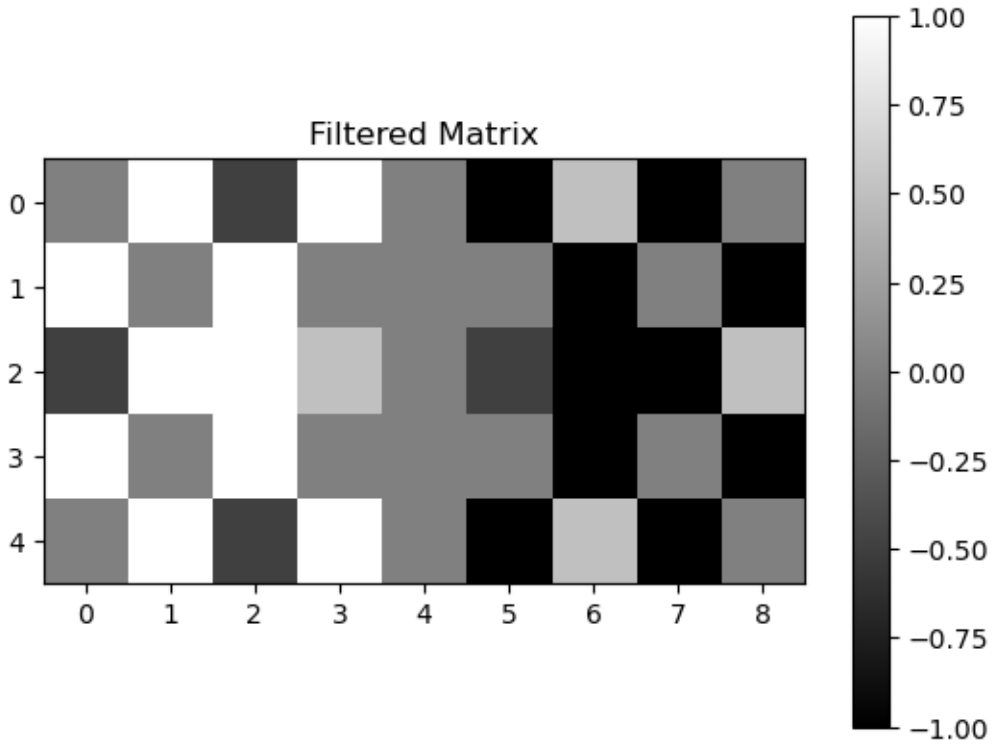


[71]:
```
def conv_res_to_latex(mat, k):
    rrr = np.array(conv2d(mat, k, stride=1, padding=1, dilation=1, groups=1).
 ↪squeeze())
    a2l.to_ltx(rrr, frmt='{:.2f}', arraytype='bmatrix')
    return rrr


mat_in = mat_in.unsqueeze(0)
mat_filtered = conv_res_to_latex(mat=mat_in, k=kernel).squeeze()
fig, ax = plt.subplots(1, 1)
p1 = ax.imshow(mat_filtered.squeeze(), cmap='gray')
ax.set_title('Filtered Matrix')
fig.colorbar(p1, ax=ax)
plt.savefig('q4_1_filtered.pdf', dpi=500, bbox_inches='tight')
```

```
print('total energy=', sum(mat_filtered.flatten() ** 2))
```

```
\begin{bmatrix}
  0.00 &  1.00 & -0.50 &  1.00 &  0.00 & -1.00 &  0.50 & -1.00 &  0.00\\
  1.00 &  0.00 &  1.00 &  0.00 &  0.00 &  0.00 & -1.00 &  0.00 & -1.00\\
 -0.50 &  1.00 &  1.00 &  0.50 &  0.00 & -0.50 & -1.00 & -1.00 &  0.50\\
  1.00 &  0.00 &  1.00 &  0.00 &  0.00 &  0.00 & -1.00 &  0.00 & -1.00\\
  0.00 &  1.00 & -0.50 &  1.00 &  0.00 & -1.00 &  0.50 & -1.00 &  0.00
\end{bmatrix}
total energy= 22.0
```

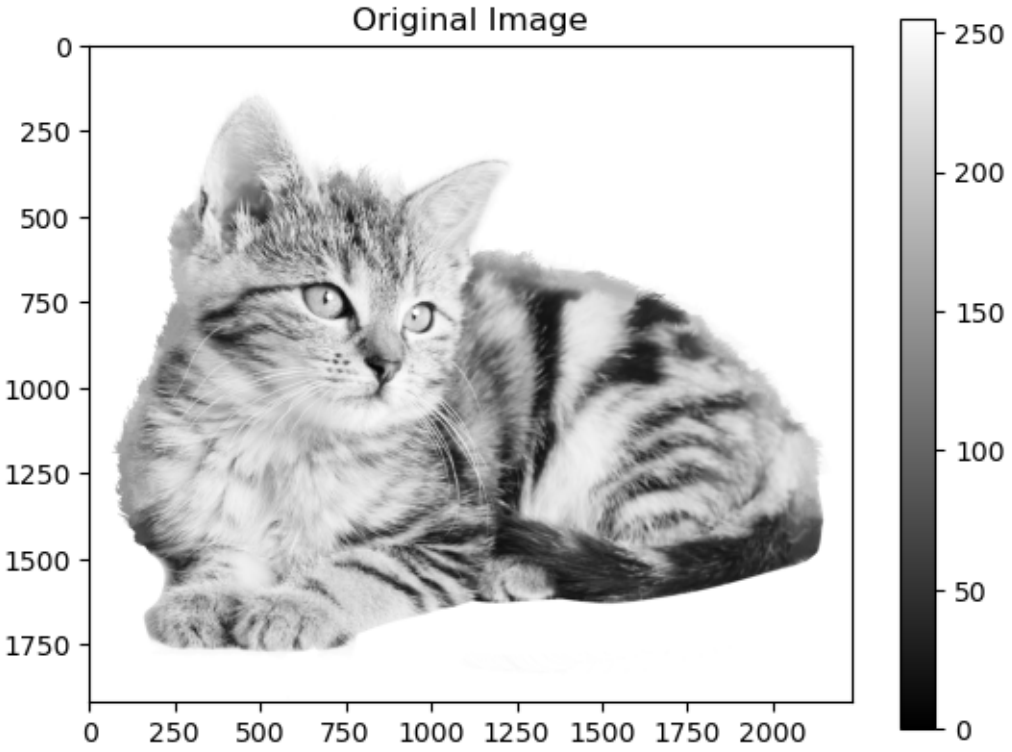


[72]:
```
import torchvision.transforms as transforms
import PIL.Image as Image
cat = Image.open('cat.png')
cat = cat.convert('L')

fig, ax = plt.subplots(1, 1)
p1 = ax.imshow(cat, cmap='gray')
ax.set_title('Original Image')
fig.colorbar(p1, ax=ax)
plt.savefig('q4_2_orig.pdf', dpi=500, bbox_inches='tight')

print('total energy=', sum(np.array(cat).flatten() ** 2))
```

```
total energy= 248152700
```

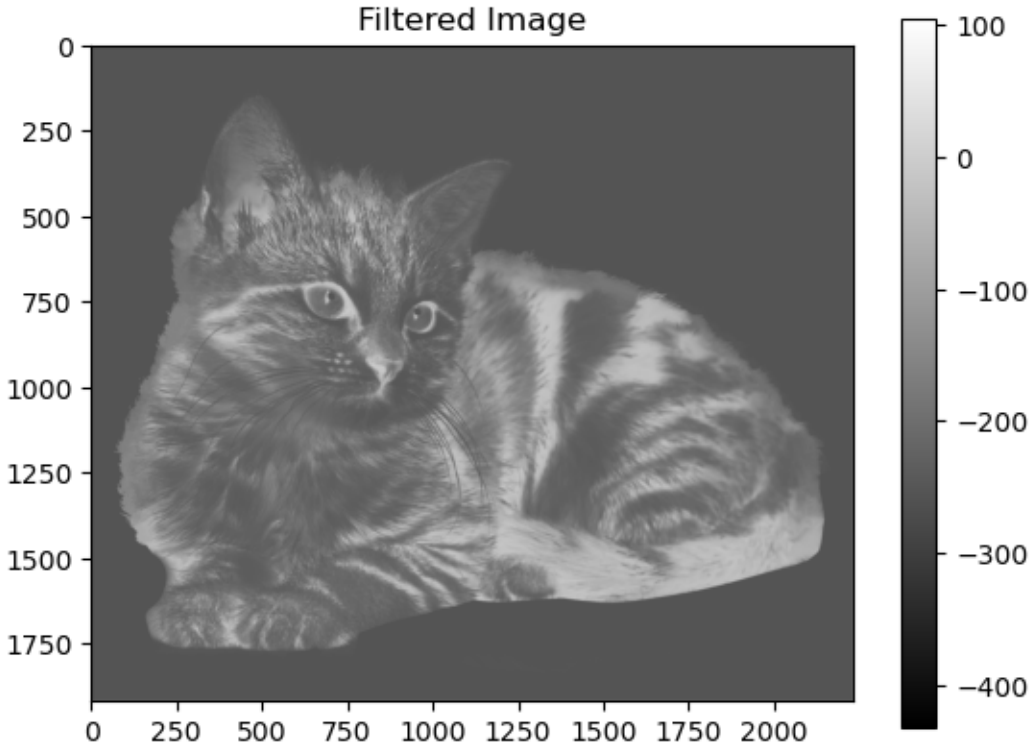


[73]:
```
transform = transforms.Compose([
    transforms.PILToTensor()
])
img_tensor = transform(cat)
img_tensor = img_tensor.unsqueeze(0)
print(img_tensor.shape)
kernel = torch.tensor([[  +0, -0.5,   +0],
                       [-0.5,   +1, -0.5],
                       [  +0, -0.5,   +0]])
kernel = kernel.repeat(1, 1, 1, 1)
print(kernel.shape)
img_conv = np.array(conv2d(img_tensor.float(), kernel, stride=1, padding=1,␣
 ↪dilation=1, groups=1).squeeze())

fig, ax = plt.subplots(1, 1)
p1 = ax.imshow(img_conv, cmap='gray')
ax.set_title('Filtered Image')
fig.colorbar(p1, ax=ax)
plt.savefig('q4_2_filtered.pdf', dpi=500, bbox_inches='tight')

print('total energy=', sum(img_conv.flatten() ** 2))
```

```
torch.Size([1, 1, 1920, 2232])
torch.Size([1, 1, 3, 3])
total energy= 203831801017.75
```



section5

September 18, 2023

[ ]:
```
from scipy.io import loadmat
import numpy as np
import array_to_latex as a2l

dataset = loadmat('dataset.mat')
x = np.array(dataset['X'])
d = np.array(dataset['D'])
```

[ ]:
```
"""part a"""

k = len(x[:, 0])
wiener_W = np.matmul(np.matmul(np.linalg.inv(np.matmul(x.T, x)), x.T), d)
print('Wiener weight\n' + a2l.to_ltx(wiener_W, frmt='{:.8f}',␣
 ↪arraytype='bmatrix', print_out=False),
      '\nshape=' + str(wiener_W.shape))
mse_wiener = np.matmul((d - np.matmul(x, wiener_W)).T, (d - np.matmul(x,␣
 ↪wiener_W))) / (2 * k)
print('mse_wiener=', mse_wiener)
```

[ ]:
```
"""part b"""

%matplotlib inline
import matplotlib.pyplot as plt
W_0 = np.array([0, 0, 0]).reshape(-1, 1)
# W_0 = torch.tensor(W_0, dtype=torch.float64, requires_grad=True)

epochs = 20


def update_weight(w_k, r, x_k, target_k):
    # print(w_k.flatten().shape, x_k.shape)
    predicted_k = np.dot(w_k.flatten(), x_k)

    w_next = w_k.flatten() + r * x_k * (target_k.flatten() - predicted_k)
    return w_next
```

```
def run_lms(W0, e, k_, lr):
    # W0: init weights; e: epochs; k_: size of dataset
    w_k_p = W0
    mse_list = []
    for epoch in range(e):
        for idx in range(k_):
            w_k = update_weight(w_k_p, r=lr, x_k=x[idx, :], target_k=d[idx])
            w_k_p = w_k
        mse_lms = np.matmul((d.flatten() - np.matmul(x, w_k_p).T).T,
                            (d.flatten() - np.matmul(x, w_k_p).T)) / (2 * k)
        mse_list.append(mse_lms)
    return w_k_p, mse_list

w_k_prev, mse_lms_list = run_lms(W0=W_0, e=epochs, k_=k, lr=0.01)
print('LMS weight\n' + a2l.to_ltx(w_k_prev.reshape(3, 1), frmt='{:.8f}',␣
 ↪arraytype='bmatrix', print_out=False))
# print(mse_lms_list[0].shape)
fig = plt.figure(dpi=400)
plt.semilogy(np.linspace(1, epochs, num=20), mse_lms_list)
# plt.plot(np.linspace(1, epochs, num=20), np.log10(mse_lms_list))
plt.title('Lab 1 (b), MSE vs epoch')
plt.xlabel('Number of Epochs')
plt.ylabel('MSE')
fig.tight_layout()
plt.savefig('lab1b.pdf', dpi=700, bbox_inches='tight')
```

[ ]:
```
"""part c"""

xy_len = 16
param_end, param_start = -2, 2

param = np.linspace(param_end, param_start, num=xy_len)
ones = np.ones(xy_len)
xy = np.vstack((ones, param, param))
print(xy)

fig = plt.figure(dpi=400)
ax = plt.axes(projection='3d')
x_points = x[:, 1]
y_points = x[:, 2]
z_points = d.flatten()

z_line_iter = np.matmul(w_k_prev, xy)
print(w_k_prev.shape, wiener_W.shape)
z_line_wiener = np.matmul(wiener_W.flatten(), xy)

ax.scatter3D(x_points, y_points, z_points)
```

```
ax.plot3D(param, param, z_line_iter, label='LMS algorithm')
ax.plot3D(param, param, z_line_wiener, linestyle='dashed', label='Wiener␣
 ↪solution')
ax.legend()
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.set_title('Regression Lines Superimposed on Data Points')
fig.tight_layout()
plt.savefig('lab1c.pdf', dpi=700, bbox_inches='tight')
```

The linear model fits the data quite well. The weight vector generated using the Wiener solution is very similar to the weight vector produced using the LMS algorithm.

[ ]:
```
"""part d"""

fig, ax = plt.subplots(2, 3, dpi=300, figsize=(15/2, 7/2))
ax_new = ax.flatten()
ax_new[-1].axis('off')
epochs = 20
W_init = np.array([0, 0, 0]).reshape(-1, 1)
r_list = [0.005, 0.01, 0.05, 0.5]
for idx, r in enumerate(r_list):
    w_k_prev, mse_list = run_lms(W0=W_init, e=epochs, k_=k, lr=r)
    ax_new[idx].semilogy(np.linspace(1, epochs, num=20), mse_list,
                         label='min MSE={:.3e}\nmax MSE={:.3e}'.
 ↪format(min(mse_list), max(mse_list)))
    ax_new[idx].set_title('MSE vs epochs, r={:g}'.format(r), fontsize=9.5)
    ax_new[idx].set_xlabel('Epochs', fontsize=9)
    ax_new[idx].set_ylabel('MSE', fontsize=9)
    ax_new[idx].legend(fontsize=6)
    ax_new[idx].tick_params(axis='x', which='both', labelsize=7)
    ax_new[idx].tick_params(axis='y', which='both', labelsize=7)

    ax_new[-2].semilogy(np.linspace(1, epochs, num=20), mse_list, label='r={:
 ↪g}'.format(r))
ax_new[-2].set_title('MSE vs epochs', fontsize=9.5)
ax_new[-2].set_xlabel('Epochs', fontsize=9)
ax_new[-2].set_ylabel('MSE', fontsize=9)
ax_new[-2].legend(fontsize=6)
ax_new[-2].tick_params(axis='x', which='both', labelsize=7)
ax_new[-2].tick_params(axis='y', which='both', labelsize=7)
# fig.suptitle('MSE vs Epochs for Different Values of r')
fig.tight_layout()
plt.savefig('lab1d_four.pdf', dpi=700, bbox_inches='tight')
```

[ ]:
```
r = 0.7
fig, ax = plt.subplots(1, 1, dpi=300)
w_k_prev, mse_list = run_lms(W0=W_init, e=epochs, k_=k, lr=r)
ax.semilogy(np.linspace(1, epochs, num=20), mse_list)
ax.set_title('MSE vs epochs, r={:g}'.format(r))
ax.set_xlabel('Epochs')
ax.set_ylabel('MSE')
fig.tight_layout()
plt.savefig('lab1d_inf.pdf', dpi=700, bbox_inches='tight')
```

When learning rate is too low (0.005), the algorithm approaches the best solution very slowly since the weights vector can only change a little bit each time a data point is fed into the algorithm. We can see that the MSE after the first epoch is much higher for r=0.005 than it is for r=0.5. Although we get a pretty small MSE after the first epoch when learning rate is high, the LMS algorithm is unable to further decrease the MSE by much after the first couple epochs. This results in a suboptimal final weight when r is too high, since the high learning rate causes the algorithm to overshoot and oscillate around the optimal solution. When learning rate is very high (e.g., lr = 0.7), the algorithm completely misses the global minimum and the MSE shoots off to infinity. So, when lr is too small, the training speed is too low, and when lr is too large, the training quality decreases (i.e., higher MSE).

[ ]:
```

```

SimpleNN

September 18, 2023

0.0.1 Code block 1: Package initialization

Import required packages, do not change.

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

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

0.0.2 Code block 2: Useful classes

Customized implementation of nn.Conv2d and nn.Linear, do not change.

[ ]:
```
class CONV(nn.Conv2d):
    def __init__(self, in_channels, out_channels, kernel_size, stride=1,
                 padding=0, dilation=1, groups=1,
                 bias=False, padding_mode='zeros'):
        self.input = None
        self.output = None
        kernel_size = _pair(kernel_size)
        stride = _pair(stride)
        padding = _pair(padding)
        dilation = _pair(dilation)
        super(CONV, self).__init__(
            in_channels, out_channels, kernel_size, stride, padding, dilation,
            groups, bias, padding_mode)

    def forward(self, input):
        self.input = input
```

```
        # 'pytorch 2.0'
        self.output = super().forward(input)
        return self.output

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

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

0.1 Lab 2

0.1.1 Code block 3: SimpleNN implementation

Please follow the instructions in Lab 2(a) and fill in the code in the lines marked **Your code here**

[ ]:
```
"""
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, dilation=1, groups=1,
                          bias=False, padding_mode='zeros')
        self.conv2 = CONV(in_channels=16, out_channels=16, kernel_size=3,␣
 ↪stride=1,
                          padding=2, dilation=1, groups=1,
                          bias=False, padding_mode='zeros')
        self.conv3 = CONV(in_channels=16, out_channels=32, kernel_size=7,␣
 ↪stride=1,
                          padding=2, dilation=1, groups=1,
                          bias=False, padding_mode='zeros')
        self.fc1   = FC(in_features=288, out_features=32, bias=True)
        self.fc2   = FC(in_features=32, out_features=10, bias=True)

    def forward(self, x):
```

```
        #print(x.size())
        # Forward pass computation
        # Conv 1
        out = F.relu(self.conv1(x))#; print(out.size())
        # MaxPool
        out = F.max_pool2d(out, 4, stride=2)#; print(out.size())
        # Conv 2
        out = F.relu(self.conv2(out)); print(out.size())
        # MaxPool
        out = F.max_pool2d(out, 3, stride=2)#; print(out.size())
        # Conv 3
        out = F.relu(self.conv3(out)); print(out.size())
        # MaxPool
        out = F.max_pool2d(out, 2, stride=2)#; print(out.size())
        # Flatten
        out = out.view(out.size(0), -1)#; print(out.size())
        # FC 1
        out = F.relu(self.fc1(out))#; print(out.size())
        # FC 2
        out = F.relu(self.fc2(out))#; print(out.size())
        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")
```

0.1.2 Code block 4: Shape observation

Please follow the instructions in Lab 2(a) and fill in the code in the lines marked **Your code here**. Gather the printed results in Table 2 in your report.

```
[ ]: """
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.forward(data)    #Your code here

# Iterate through all the CONV and FC layers of the model
for name, module in net.named_modules():
    if isinstance(module, CONV) or isinstance(module, FC):
        # Get the input feature map of the module as a NumPy array
        input = module.input.detach().cpu().numpy()     #Your code here
        # Get the output feature map of the module as a NumPy array
        output = module.output.detach().cpu().numpy()     #Your code here
        # Get the weight of the module as a NumPy array
        weight = module.weight.detach().cpu().numpy()    #Your code here
        # Compute the number of parameters in the weight
        num_Param = np.prod([i for i in weight.shape])    #Your code here
        # Compute the number of MACs in the layer
        if isinstance(module, CONV):
            num_MAC = input.shape[1] * (np.array(module.kernel_size[0]) ** 2) *␣
 ↪\
                      np.prod(output.shape)   #Your code here
            # C1 * K*k * C2 * H2 * W2
        else:  # isinstance(module, FC)
            num_MAC = len(input.flatten()) * len(output.flatten())   #Your code␣
 ↪here

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

0.2 Lab 3 (Bonus)

0.2.1 Code block 5: Initial weight histogram

Please follow the instructions in Lab 3(a) and fill in the code in the lines marked **Your code here**. Copy the output figures into your report.

```
[ ]: """
Lab 3(a)
"""
# # my addition:
# size_title = 10
# size_label = 8
# size_ticks = 7
#
# fig, ax = plt.subplots(2, 3, dpi=300)
```

```
# ax_new = ax.flatten()
# plot_idx = 0
# ax_new[-1].axis('off')
# # end of my addition


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

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

#         # my addition:
#         weight = weight.reshape(-1)
#         ax_new[plot_idx].hist(weight, bins=20)
#         ax_new[plot_idx].set_title("Weight histogram\nof layer "+name,␣
 ↪fontsize=size_title)
#         ax_new[plot_idx].set_xlabel("Value", fontsize=size_label)
#         ax_new[plot_idx].set_ylabel("Frequency", fontsize=size_label)
#         ax_new[plot_idx].tick_params(axis='x', which='both',␣
 ↪labelsize=size_ticks)
#         ax_new[plot_idx].tick_params(axis='y', which='both',␣
 ↪labelsize=size_ticks)
#         plot_idx += 1
# fig.tight_layout()
# plt.savefig('lab3a.pdf', dpi=700, bbox_inches='tight')
# end of my addition
```

0.2.2 Code block 6: Gradient histogram

Please follow the instructions in Lab 3(b) and fill in the code in the lines marked **Your code here**. Copy the output figures into your report.

[ ]:
```
'''
Lab 3(b)
'''
# Loss definition
criterion = nn.MSELoss()
# Random target
target = torch.randn(1, 10).to(device)

# Loss computation, using out defined in Lab 2(b)
```

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

# # my addition:
# fig, ax = plt.subplots(2, 3, dpi=300)
# ax_new = ax.flatten()
# plot_idx = 0
# ax_new[-1].axis('off')
# # end of my addition

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

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

#         # my addition:
#         gradient = gradient.reshape(-1)
#         ax_new[plot_idx].hist(gradient, bins=20)
#         ax_new[plot_idx].set_title("Gradient histogram\nof layer "+name,␣
 ↪fontsize=size_title)
#         ax_new[plot_idx].set_xlabel("Value", fontsize=size_label)
#         ax_new[plot_idx].set_ylabel("Frequency", fontsize=size_label)
#         ax_new[plot_idx].tick_params(axis='x', which='both',␣
 ↪labelsize=size_ticks)
#         ax_new[plot_idx].tick_params(axis='y', which='both',␣
 ↪labelsize=size_ticks)
#         plot_idx += 1
# fig.tight_layout()
# plt.savefig('lab3b.pdf', dpi=700, bbox_inches='tight')
# end of my addition
```

0.2.3 Code block 7: Zero initialization?

Please follow the instructions in Lab 3(c) and fill in the code in the lines marked **Your code here**. Copy the output figures into your report.

[ ]:
```
'''
Lab 3(c)
'''
```

```python
# Set model weights to zero
for name, module in net.named_modules():
    if isinstance(module, CONV) or isinstance(module, FC):
        # Set the weight of each module to all zero
        module.weight.data.fill_(0)

# Reset gradients
net.zero_grad()

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

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

# # my addition:
# fig, ax = plt.subplots(2, 3, dpi=300)
# ax_new = ax.flatten()
# plot_idx = 0
# ax_new[-1].axis('off')
# # end of my addition


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

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

#         # my addition:
#         print(sum(np.abs(gradient).flatten()))
#         ax_new[plot_idx].hist(gradient, bins=20)
#         ax_new[plot_idx].set_title("Gradient histogram\nof layer "+name,␣
 ↪fontsize=size_title)
#         ax_new[plot_idx].set_xlabel("Value", fontsize=size_label)
#         ax_new[plot_idx].set_ylabel("Frequency", fontsize=size_label)
```

```
#         ax_new[plot_idx].tick_params(axis='x', which='both',␣
 ↪labelsize=size_ticks)
#         ax_new[plot_idx].tick_params(axis='y', which='both',␣
 ↪labelsize=size_ticks)
#         plot_idx += 1
# fig.tight_layout()
# plt.savefig('lab3c.pdf', dpi=700, bbox_inches='tight')
# # end of my addition
```

The weights for all the convolution layers and the first FC layer are zero compared to lab 3b), which are approximately centred at zero but not identically zero. When using ReLU activation, the gradient with respect to the weights tensor is zero if every input to the ReLU module is zero. When the initial weights tensor is zero, the convolution of the input feature map with the weights tensor is also zero when the bias is zero. Since the identically-zero result of the convolution is fed into the ReLU function, the output of the ReLU function will also be zero. When using the chain rule to compute the derivative of loss with respect to the weights tensor, one of the intermediate gradients that appear is the gradient of the ReLU function with respect to its input, which becomes identically zero if the inputs to the ReLU function are all zero. Multiplication by a zero tensor causes the entire gradient to be zero as well. We didn't initialise the weights vector in lab 3b) as identically zero, so the model doesn't have gradients that are identically zero for the convolutional layers.

[ ]: