EEE598: Deep Learning Assignment 2

Chih-Hao Tsai

Arizona State University

September 15, 2024

1 Problem 1: Perceptron

1.1 Perform perceptron algorithm by hand

In this section, I've drawn multiple hyperplanes as demonstration of my perceptron algorithm. My approach to the bias term is: $b_n = b_{n-1} + 0.5(y)$, and I also set the maximum iteration to 20 times. When performing the algorithm with this bias term, 4 iterations occurred with 6 updates in weight and bias.

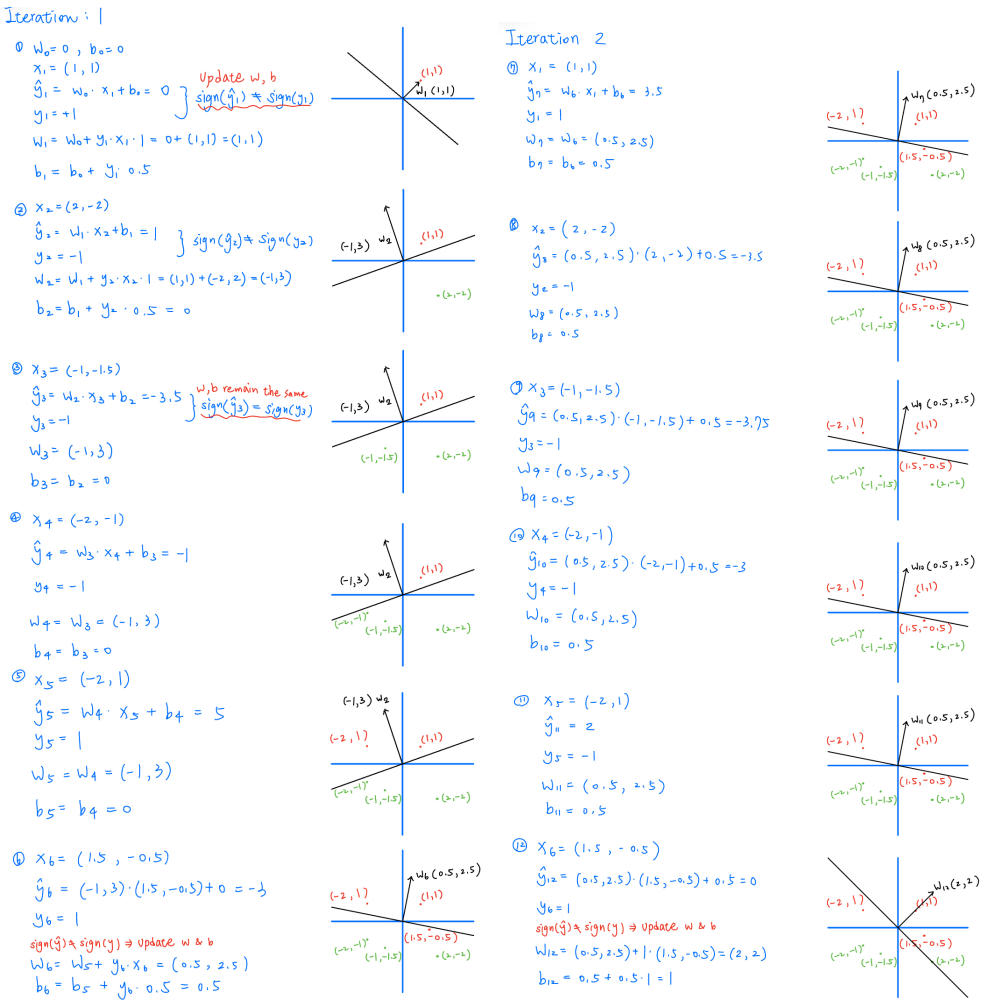

Fig. 1: First two iterations of the algorithm

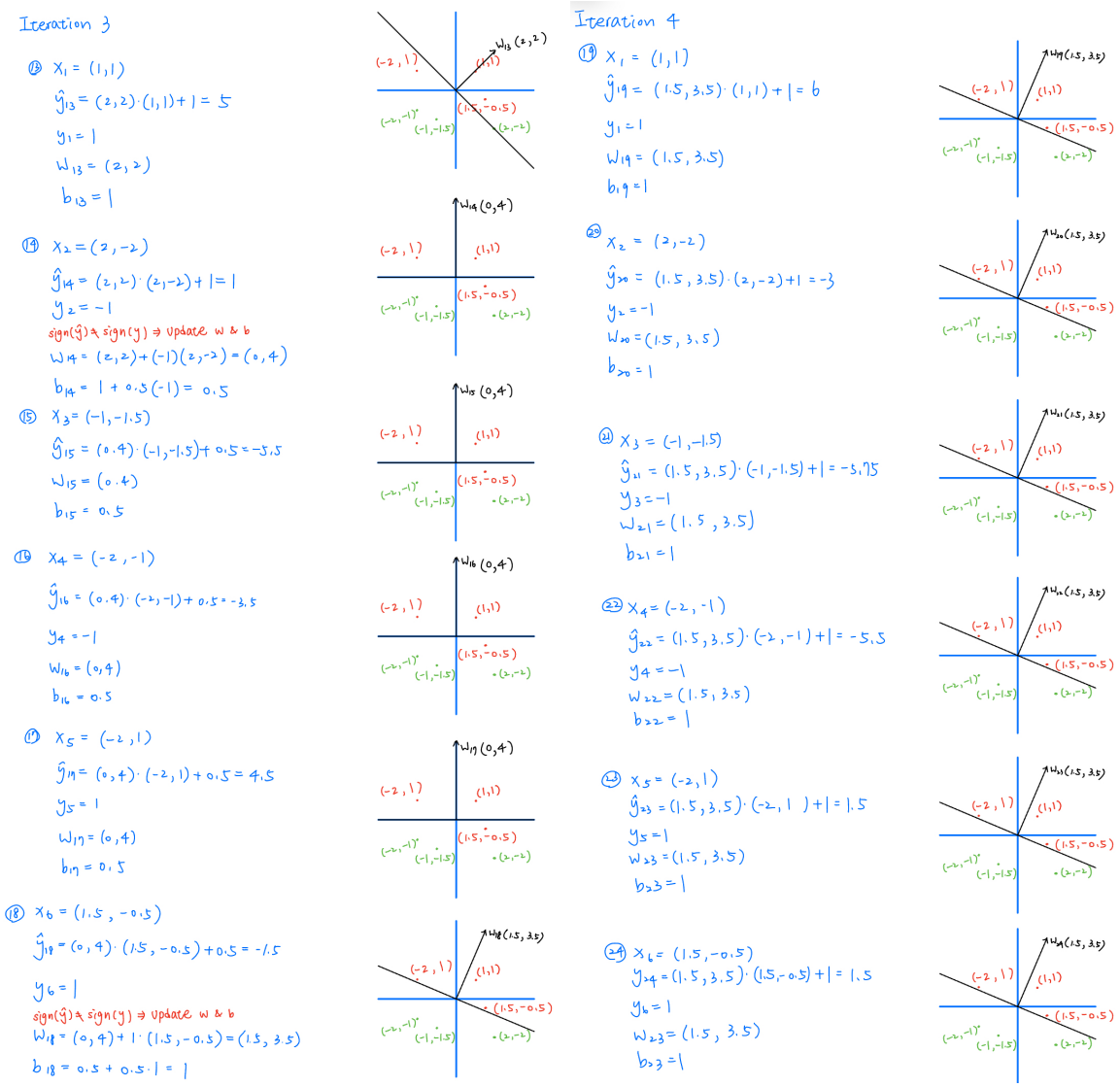

Fig. 2: Last two iterations of the algorithm

1.2 A code to verify the result

Based on the original perceptron algorithm and my modified version of the update of bias term, my perceptron algorithm looks like this:

Algorithm 1 Perceptron Algorithm

function PERCEPTRON ALGORITHM
  $w^{(0)} \leftarrow 0$
  for $t = 1, \ldots, T$ **do**
    Receive $x^{(t)}$
    Receive $y^{(t)}$
    $\hat{y}_A^{(t)} = \text{sign}\left(\langle w^{(t-1)}, x^{(t)} \rangle\right)$
    if $\hat{y}_A^{(t)} * y^{(t)} <= 0$ **then**
      $w_n^{(t)} = w_n^{(t-1)} + y^{(t)} \cdot x_n^{(t)} \cdot \mathbf{1}[y^{(t)} \neq \hat{y}^{(t)}]$
      $b_n^{(t)} = b_{n-1}^{(t)} + 0.5 * y^{(t)}$
    end if
  end for
end function

And I implemented the algorithm with Python. The result corresponds to the result of my hand-drawn hyperplanes as can be seen in the figures below.

```python
# Initialize weight, bias
w = [0, 0]
b = 0

# Define maximum iteration and step
max_iter = 20
step = 0

for iter in range(max_iter):
    # Initialize error count
    error_cnt = 0
    print(f"Iteration: {iter+1}")
    for item in D:

        # Count and printout the steps of the algorithm
        step += 1
        print(f'step: {step}')

        # Get the data from D, x as feature and y as label
        x, y = item[0], item[1]

        # Calculate y-hat
        a = np.dot(w, x) + b

        # Update w and b if y-hat and y has different signs
        if y*a <= 0:
            w = w + y * np.array(x)
            b = b + y * 0.5
            error_cnt += 1

        # Print the y-hat, w, b for every step
        print(f'y_hat = {a}, w = {w}, b = {b}\n')

    # Stops the algorithm if not update is needed (error count = 0)
    if error_cnt == 0:
        print(f"Total iterations : {iter + 1}")
        break
    elif iter == (max_iter-1):
        print('The dataset cannot converge!!')
```

Fig. 3: My Python algorithm

```
Iteration: 1
step: 1
y_hat = 0, w = [1 1], b = 0.5

step: 2
y_hat = 0.5, w = [-1  3], b = 0.0

step: 3
y_hat = -3.5, w = [-1  3], b = 0.0

step: 4
y_hat = -1.0, w = [-1  3], b = 0.0

step: 5
y_hat = 5.0, w = [-1  3], b = 0.0

step: 6
y_hat = -3.0, w = [0.5 2.5], b = 0.5

Iteration: 2
step: 7
y_hat = 3.5, w = [0.5 2.5], b = 0.5

step: 8
y_hat = -3.5, w = [0.5 2.5], b = 0.5

step: 9
y_hat = -3.75, w = [0.5 2.5], b = 0.5

step: 10
y_hat = -3.0, w = [0.5 2.5], b = 0.5

step: 11
y_hat = 2.0, w = [0.5 2.5], b = 0.5

step: 12
y_hat = 0.0, w = [2. 2.], b = 1.0

Iteration: 3
step: 13
y_hat = 5.0, w = [2. 2.], b = 1.0

step: 14
y_hat = 1.0, w = [0. 4.], b = 0.5

step: 15
y_hat = -5.5, w = [0. 4.], b = 0.5

step: 16
y_hat = -3.5, w = [0. 4.], b = 0.5

step: 17
y_hat = 4.5, w = [0. 4.], b = 0.5

step: 18
y_hat = -1.5, w = [1.5 3.5], b = 1.0

Iteration: 4
step: 19
y_hat = 6.0, w = [1.5 3.5], b = 1.0

step: 20
y_hat = -3.0, w = [1.5 3.5], b = 1.0

step: 21
y_hat = -5.75, w = [1.5 3.5], b = 1.0

step: 22
y_hat = -5.5, w = [1.5 3.5], b = 1.0

step: 23
y_hat = 1.5, w = [1.5 3.5], b = 1.0

step: 24
y_hat = 1.5, w = [1.5 3.5], b = 1.0

Total iterations : 4
```

Fig. 4: These are the result of the algorithm, as can be seen, total of 4 iterations are implemented

1.3 Add a point to make the dataset linear inseparable

To make the dataset linear inseparable, I add a point $(x_1, x_2) = (1, 2)$ with label -1 to the original dataset. Making the dataset look like this:

$$D = \begin{bmatrix} (1,1), & 1 \\ (2,-2), & -1 \\ (-1,-1.5), & -1 \\ (-2,-1), & -1 \\ (-2,1), & 1 \\ (1.5,-0.5), & 1 \\ (1,2), & -1 \end{bmatrix}$$

To make the algorithm runs nearly forever and can better show the result, I modified the maximum iteration to 2000 times. And then, I plotted all the possible hyperplanes and all data points as with Python package Matplotlib. Also, I keep track on the difference between w_n, w_{n-1}, which is $||w_n - w_{n-1}||$, so that I can know if w is converging to 0.

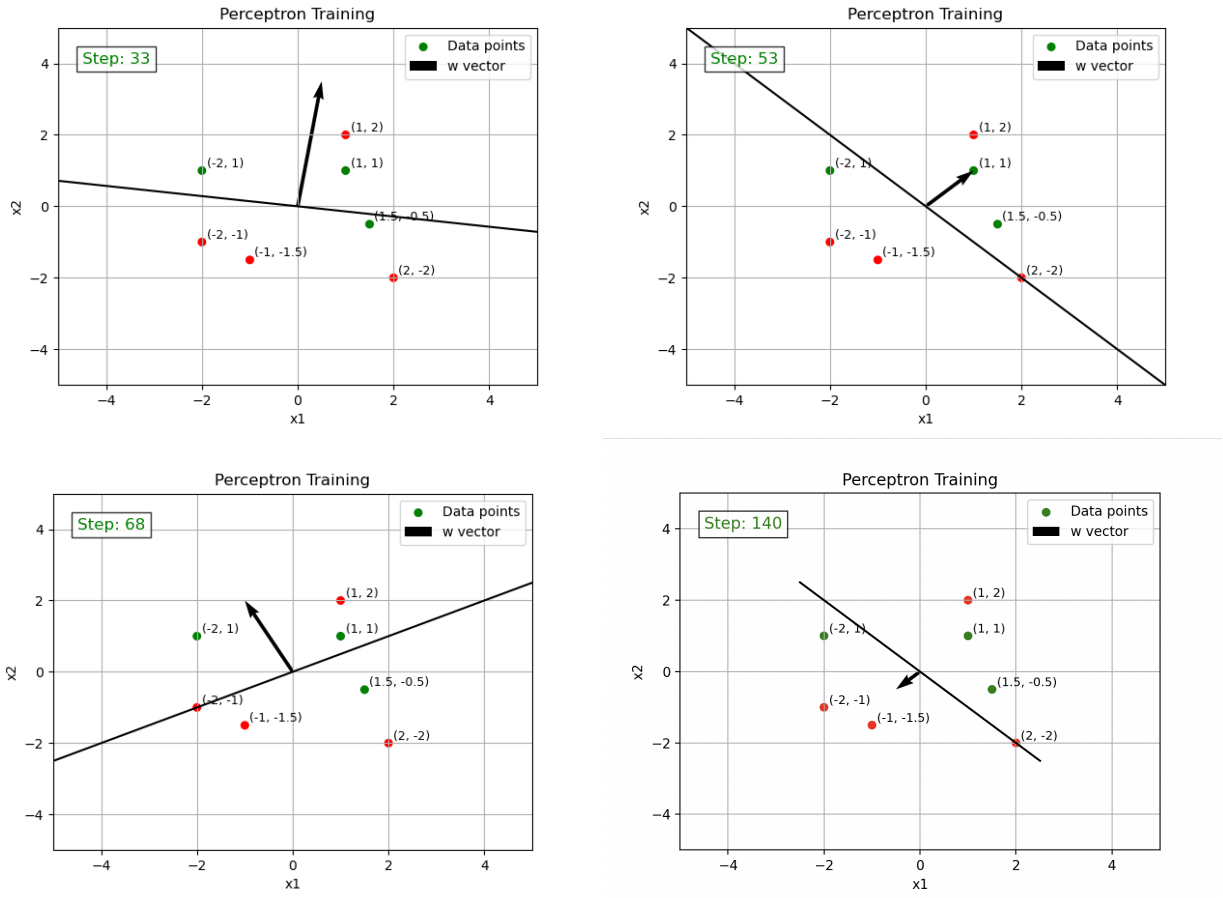

Fig. 5: These hyperplanes show that this new dataset is impossible to be linear separable

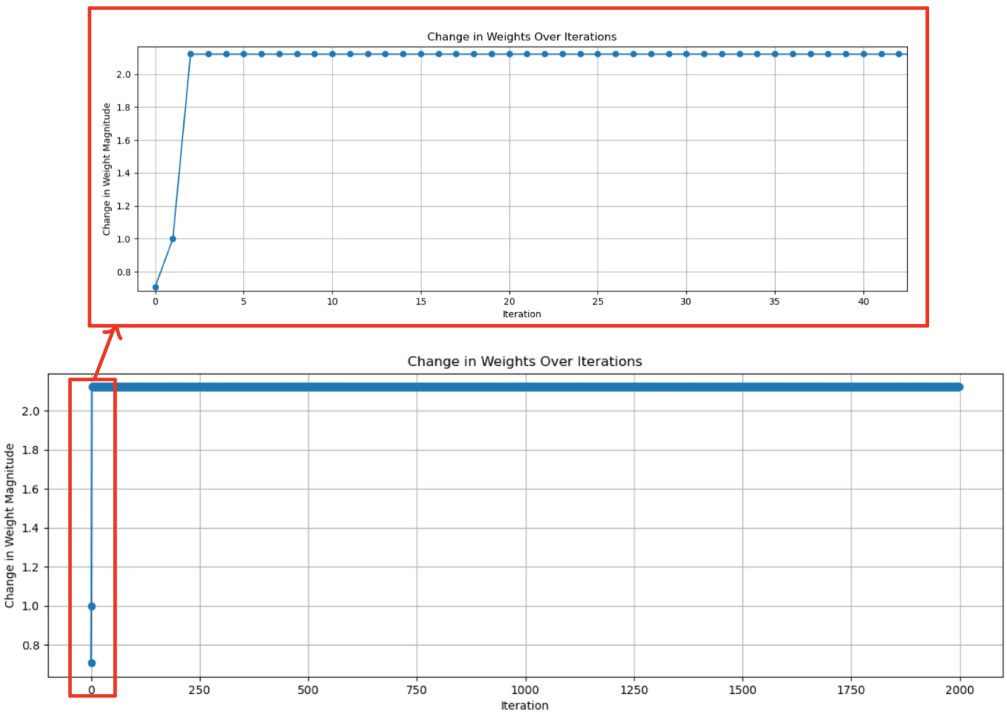

Fig. 6: Until the algorithm meets the maximum iteration, the change in w still cannot converge. It shows that $||w_n - w_{n-1}||$ never converges to 0, which means this dataset is linear inseparable.

2 Problem 2: Backpropagation

In this problem, we have to implement backpropagation to a ANN, which has an input layer, an output layer, and 5 hidden layers (K=5). Each layer have 6 neurons.

- Define a function that forward pass through the ANN:
  The first TODO lines in the attached file is to define the function *compute_network_output*, which is meant to calculate pre-activations and activations at the k^{th} hidden layer ($k = 0, ..., K-1$) and keep the pre-activations and activations in separate list (all_f and all_h). To do so, I implement two equations in the for-loop:

$$f_k = \beta_k + \Omega_k \cdot h_k \tag{1}$$

$$h_k = ReLU(f_{k-1}) \tag{2}$$

  Then, outside of the for-loop, we have to calculate the pre-activations from the last hidden layer. As the result, I implement equation(2) again.

```
for layer in range(K):
    # TODO Update preactivations and activations at this layer
    all_f[layer] = all_biases[layer] + np.dot(all_weights[layer], all_h[layer])
    all_h[layer+1] = ReLU(all_f[layer])

# TODO Compute the output from the last hidden layer
all_f[K] = all_biases[K] + np.dot(all_weights[K], all_h[K])
```

Fig. 7: In the for-loop, equation(1) and equation(2) are both i,plemented

- Define $lossfunction$ and $\frac{\partial l_i}{\partial f_K}$ function
  I define loss function as:

$$lossfunction = \sum_{i=0}^{K} (net\_output - y_i)^2$$

  Therefore, $\frac{\partial l_i}{\partial f_K} = 2 \cdot (net\_output - y)$

```
# TODO
def least_squares_loss(net_output, y):
  return sum(pow((net_output-y), 2))

def d_loss_d_output(net_output, y):
    return 2*(net_output-y)
```

Fig. 8: Defining loss function and $\frac{\partial l_i}{\partial f_K}$

- Define main backward pass routine
  In this step, I follow the equations on the reference book (7.4.1 Backpropagation algorithm summary).
  - Derivatives of loss with respect to biases

$$\frac{\partial l_i}{\partial \beta_k} = \frac{\partial l_i}{\partial f_k}, k \in \{K, K-1, ..., 1\}$$

  - Derivatives of loss with respect to weights

$$\frac{\partial l_i}{\partial \Omega_k} = \frac{\partial l_i}{\partial f_k} h_k^T, k \in \{K, K-1, ..., 1\}$$

  - Derivatives of loss with respect to activations(3) and pre-activations(4)

$$\Omega_k^T \frac{\partial l_i}{\partial f_k}, k \in \{K, K-1, ..., 1\} \tag{3}$$

$$\frac{\partial \ell_i}{\partial f_{k-1}} = \mathbb{I}[f_{k-1} > 0] \odot \left(\Omega_k^T \frac{\partial \ell_i}{\partial f_k} \right), k \in \{K, K-1, ..., 1\} \tag{4}$$

```python
# Now work backwards through the network
for layer in range(K,-1,-1):
  # TODO Calculate the derivatives of the loss with respect to the biases at layer from all_dl_df[layer].
  # NOTE!  To take a copy of matrix X, use Z=np.array(X)
  all_dl_dbiases[layer] = np.array(all_dl_df[layer])

  # TODO Calculate the derivatives of the loss with respect to the weights at layer from all_dl_df[layer] and all_h[layer]
  all_dl_dweights[layer] = np.outer(all_dl_df[layer], all_h[layer])

  # TODO: calculate the derivatives of the loss with respect to the activations from weight and derivatives of next preactivations
  all_dl_dh[layer] = np.dot(all_weights[layer].T, all_dl_df[layer])

  if layer > 0:
    # TODO Calculate the derivatives of the loss with respect to the pre-activation f
    all_dl_df[layer-1] = all_dl_dh[layer] * indicator_function(all_f[layer-1])
```

Fig. 9: This is the code I implement the equations based on the book.

```
-----------------------------------------------
Bias 0, derivatives from backprop:
[[-0.   ]
 [93.236]
 [16.894]
 [ 0.   ]
 [ 0.   ]
 [ 0.   ]]
Bias 0, derivatives from finite differences
[[ 0.   ]
 [93.236]
 [16.894]
 [ 0.   ]
 [ 0.   ]
 [ 0.   ]]
Success!  Derivatives match.
-----------------------------------------------
Bias 1, derivatives from backprop:
[[  0.   ]
 [ -0.   ]
 [ 24.744]
 [ -0.   ]
 [140.369]
 [ 20.416]]
Bias 1, derivatives from finite differences
[[  0.   ]
 [  0.   ]
 [ 24.744]
 [  0.   ]
 [140.369]
 [ 20.416]]
Success!  Derivatives match.
-----------------------------------------------
Bias 2, derivatives from backprop:
[[-69.507]
 [  0.   ]
 [  0.   ]
 [  0.   ]
 [  0.   ]
 [ 49.537]]
Bias 2, derivatives from finite differences
[[-69.507]
 [  0.   ]
 [  0.   ]
 [  0.   ]
 [  0.   ]
 [ 49.537]]
Success!  Derivatives match.
```

```
-----------------------------------------------
Bias 3, derivatives from backprop:
[[  0.   ]
 [-49.716]
 [ 54.032]
 [  0.   ]
 [ 21.745]
 [  0.   ]]
Bias 3, derivatives from finite differences
[[  0.   ]
 [-49.716]
 [ 54.032]
 [  0.   ]
 [ 21.745]
 [  0.   ]]
Success!  Derivatives match.
-----------------------------------------------
Bias 4, derivatives from backprop:
[[-0.   ]
 [-5.417]
 [ 0.   ]
 [-0.   ]
 [59.583]
 [11.152]]
Bias 4, derivatives from finite differences
[[ 0.   ]
 [-5.417]
 [ 0.   ]
 [ 0.   ]
 [59.583]
 [11.152]]
Success!  Derivatives match.
-----------------------------------------------
Weight 0, derivatives from backprop:
[[ -0.   ]
 [111.883]
 [ 20.273]
 [  0.   ]
 [  0.   ]
 [  0.   ]]
Weight 0, derivatives from finite differences
[[  0.   ]
 [111.883]
 [ 20.273]
 [  0.   ]
 [  0.   ]
 [  0.   ]]
Success!  Derivatives match.
```

Fig. 10: These are the result of my code.

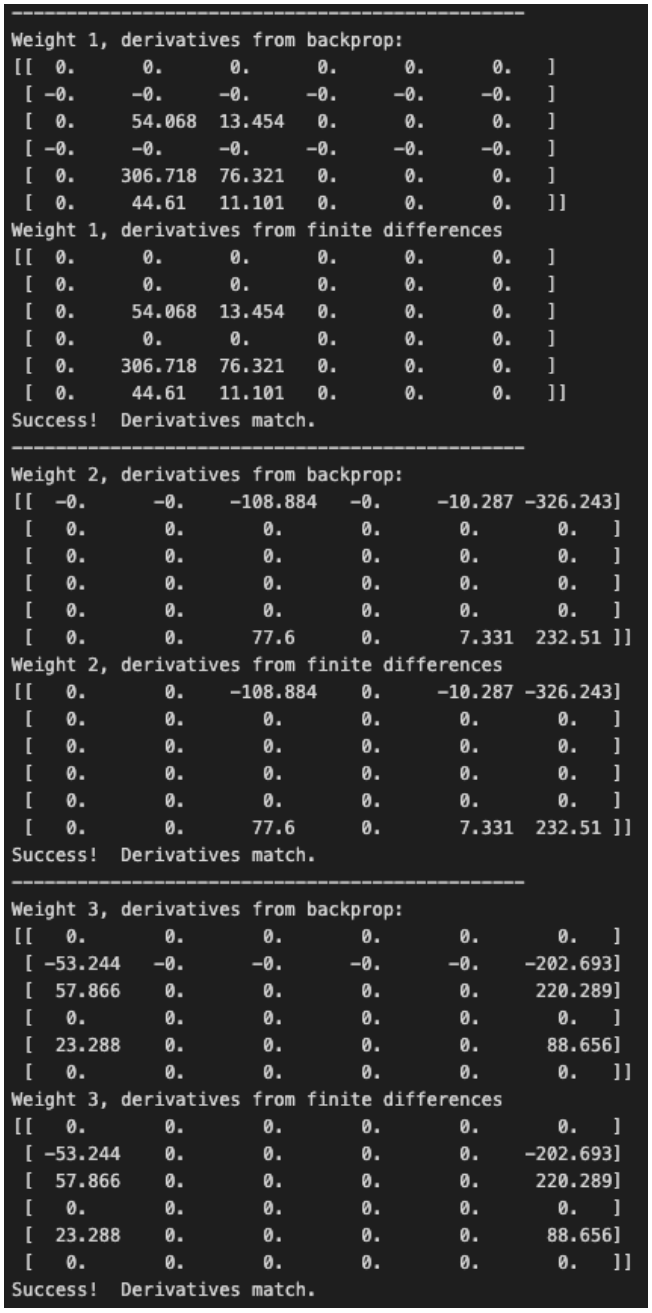

Fig. 11: The output of weights 1 to 3

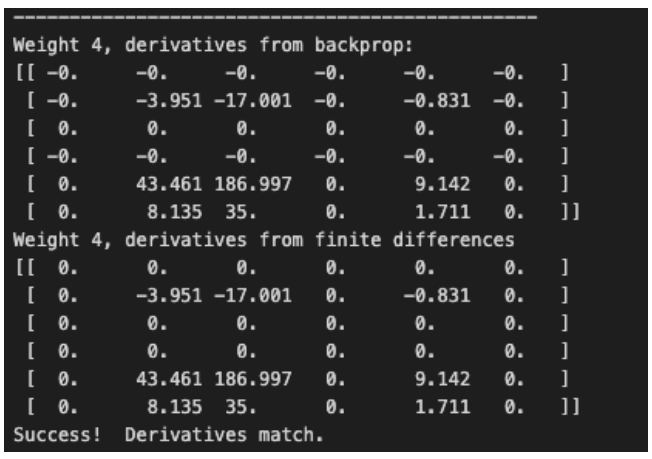

Fig. 12: The output of weight 4

3 Problem 3: MLP v.s. KAN

3.1 Training on MLP

In this sub-problem, I create a MLP with 5 features as input, 2 hidden layers (20, 5 nodes for each layer), and one node for output. The structure can be seen in figure 13. The parameters I choose including: 0.1 as learning rate, Adam as optimizer. The reason why I use these parameters is that these parameters seems to be optimal for this structure after multiple attempts on different parameters (such as adjusting learning rate or use SGD as optimizer). Afterwards, I split the dataset into 70-20-10 (train-test-val) and run the training session. The result of the MLP shows in figure 3.

```
NeuralNetwork(
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=5, out_features=20, bias=True)
    (1): ReLU()
    (2): Linear(in_features=20, out_features=5, bias=True)
    (3): ReLU()
    (4): Linear(in_features=5, out_features=1, bias=True)
  )
)
```

Fig. 13: The structure of my MLP.

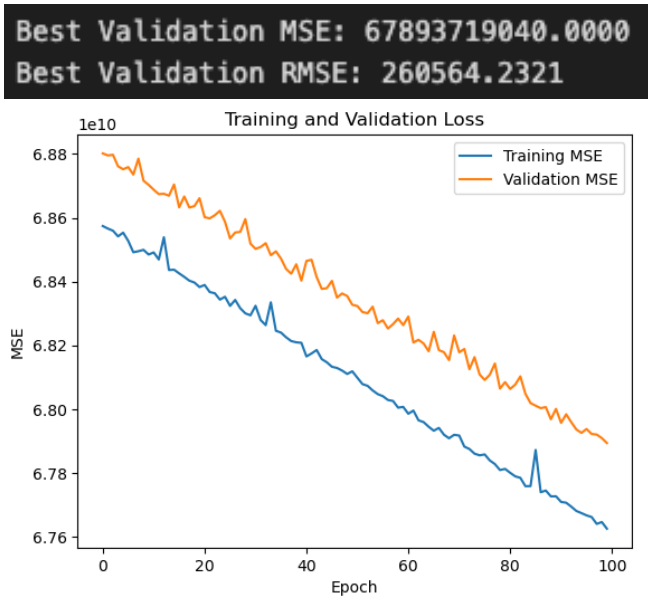

Fig. 14: The result of MLP training.

3.2 Training on KAN

In my KAN model, I use the same learning rate and the same optimizer in order to highlight the influence on loss in terms of using different models. I built my KAN structure with also 3 like figure 15. And the output surprise me with only a little improvement on loss. (figure 16)

```
MultKAN(
  (act_fun): ModuleList(
    (0-2): 3 x KANLayer(
      (base_fun): SiLU()
    )
  )
  (base_fun): SiLU()
  (symbolic_fun): ModuleList(
    (0-2): 3 x Symbolic_KANLayer()
  )
)
```

Fig. 15: My KAN structure

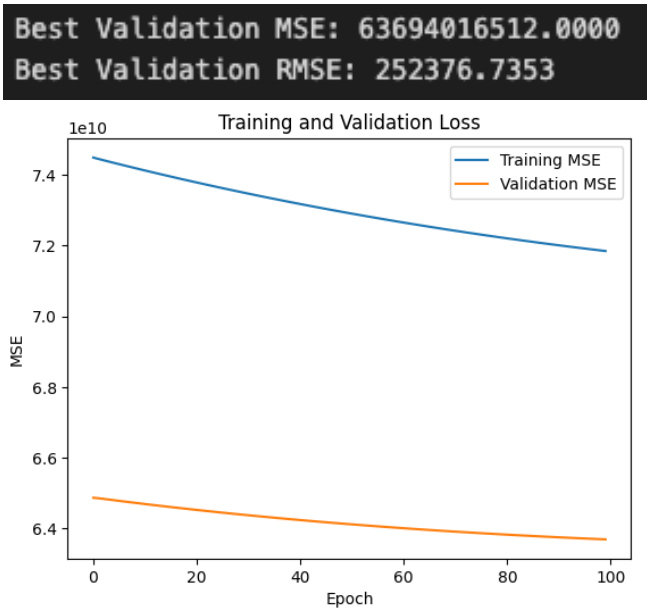

Fig. 16: The output of my KAN.

3.3 Analyse both models

Theoretically, KAN would have better performance than classic MLP because of learnable activation function that KAN uses, avoiding the limitaion of fixed activation function (like MLP), and capturing features more efficiently. However, my KAN structure was not as good as I expected. Only minor improvement on training accuracy and validation accuracy.

Another feature of KAN that worth mention is the training time, training time of KAN model is much higher than classic MLP model in my case. I have MLP training session completed in 4 minutes while KAN model took me almost 45 minutes to get the result.