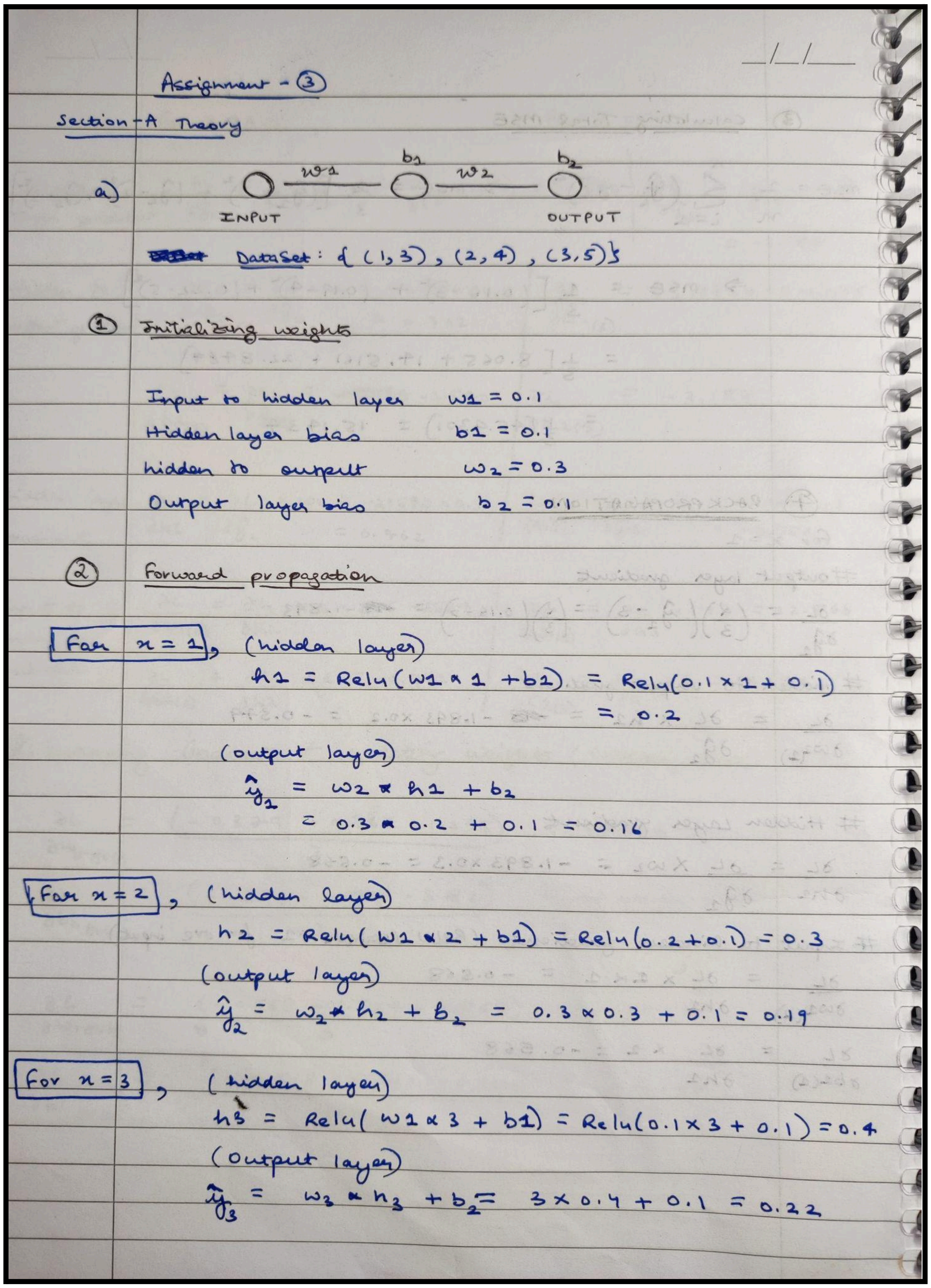
ML Assignment-3
Report

Aditya Sharma (2022038)

Section A (Theoretical)

A)



Assignment - ③

Section-A Theory

a) INPUT ○ —w_1— ○ b_1 —w_2— ○ b_2 OUTPUT

DataSet: {(1,3), (2,4), (3,5)}

① Initializing weights

Input to hidden layer	$w_1 = 0.1$
Hidden layer bias	$b_1 = 0.1$
hidden to output	$w_2 = 0.3$
Output layer bias	$b_2 = 0.1$

② Forward propagation

$\boxed{\text{For } x = 1}$, (hidden layer)

$h_1 = \text{Relu}(w_1 \times 1 + b_1) = \text{Relu}(0.1 \times 1 + 0.1)$
$= 0.2$

(output layer)

$\hat{y}_1 = w_2 \times h_1 + b_2$
$= 0.3 \times 0.2 + 0.1 = 0.16$

$\boxed{\text{For } x = 2}$, (hidden layer)

$h_2 = \text{Relu}(w_1 \times 2 + b_1) = \text{Relu}(0.2 + 0.1) = 0.3$

(output layer)

$\hat{y}_2 = w_2 \times h_2 + b_2 = 0.3 \times 0.3 + 0.1 = 0.19$

$\boxed{\text{For } x = 3}$, (hidden layer)

$h_3 = \text{Relu}(w_1 \times 3 + b_1) = \text{Relu}(0.1 \times 3 + 0.1) = 0.4$

(output layer)

$\hat{y}_3 = w_3 \times h_3 + b_2 = 3 \times 0.4 + 0.1 = 0.22$

__/__/__

③ Calculating Total MSE

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \Rightarrow MSE = \frac{1}{3}\left[(\hat{y}_1 - 3)^2 + (\hat{y}_2 - 2)^2 + (\hat{y}_3 - 3)^2\right]$$

$$\Rightarrow MSE = \frac{1}{3}\left[(0.16-3)^2 + (0.19-4)^2 + (0.22-5)^2\right]$$

$$= \frac{1}{3}\left[8.065 + 14.5161 + 22.8484\right]$$

$$= \frac{1}{3}[45.4301] = 15.1434$$

④ BACKPROPAGATION

For $x = 1$

output layer gradient

$$\frac{\partial L}{\partial \hat{y}_1} = \left(\frac{2}{3}\right)(\hat{y}_1 - 3) = \left(\frac{2}{3}\right)(0.16-3) = -1.893$$

Hidden to output gradient

$$\frac{\partial L}{\partial w2_{(1)}} = \frac{\partial L}{\partial \hat{y}_1} \times h1 = -1.893 \times 0.2 = -0.379$$

Hidden Layer gradient

$$\frac{\partial L}{\partial h1} = \frac{\partial L}{\partial \hat{y}_1} \times w_2 = -1.893 \times 0.3 = -0.568$$

Input to hidden gradient (ReLU derivate = 1 for +ve input)

$$\frac{\partial L}{\partial w1_{(1)}} = \frac{\partial L}{\partial h1} \times 1 \times 1 = -0.568$$

$$\frac{\partial L}{\partial b1_{(1)}} = \frac{\partial L}{\partial h1} \times 1 = -0.568$$

	For $x = 2$,	For $x = 3$
output layer gradient	$\frac{\partial L}{\partial \hat{y}_2} = \left(\frac{2}{3}\right)\left(\hat{y}_2 - 4\right) = \frac{2}{3}\left(~~g~~\, 0.19 - 4\right) = -2.540$	$\frac{\partial L}{\partial \hat{y}_3} = \left(\frac{2}{3}\right)\left(0.22 - 5\right) = -3.187$
Hidden to output gradient	$\frac{\partial L}{\partial w_{2(2)}} = \frac{\partial L}{\partial \hat{y}_2} \times h_2 = -2.540 \times 0.3 = 0.762$	$\frac{\partial L}{\partial w_{2(3)}} = -3.187 \times 0.4 = -1.275$
	$\frac{\partial L}{\partial b_{2(2)}} = \frac{\partial L}{\partial \hat{y}_{(2)}} = \cancel{3.187}\ -2.540$	$\frac{\partial L}{\partial b_{2(3)}} = -3.187$
Hidden Layer Gradient	$\frac{\partial L}{\partial h_2} = \frac{\partial L}{\partial \hat{y}_2} \times w_2 = -2.540 \times 0.3 = 0.762$	$\frac{\partial L}{\partial h_3} = \frac{\partial L}{\partial \hat{y}_3} \times w_2 = -3.187 \times 0.3 = -0.956$
Input to hidden ~~gr~~ gradients	$\frac{\partial L}{\partial w_{1(2)}} = \frac{\partial L}{\partial h_2} \times 1 \times 2 = -1.524$	$\frac{\partial L}{\partial w_{1(3)}} = \frac{\partial L}{\partial h_3} \times 1 \times 3 = -2.568$
	$\frac{\partial L}{\partial b_{1(2)}} = \frac{\partial L}{\partial h_2} \times 1 = -0.762$	$\frac{\partial L}{\partial b_{2(3)}} = \frac{\partial L}{\partial h_3} \times 1 = -0.956$

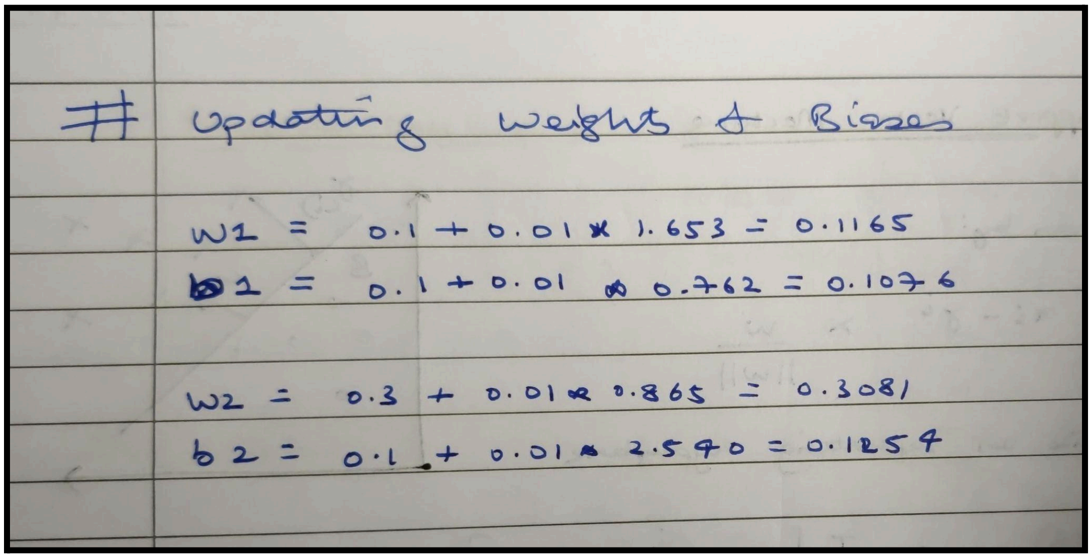
⑤ Averaging Gradient & updating weights (Learning-rate = 0.01)

$$\frac{\partial L}{\partial w_{2\,avg}} = \frac{(-0.379 - 0.762 - 1.275)}{3} = -0.825$$

$$\frac{\partial L}{\partial b_{2\,avg}} = \frac{(-1.893 - 2.540 - 3.187)}{3} = -2.540$$

$$\frac{\partial L}{\partial w_{1\,avg}} = \frac{(-0.568 - 1.524 - 2.868)}{3} = -1.853$$

$$\frac{\partial L}{\partial b_{1\,avg}} = \frac{(-0.568 - 0.762 - 0.956)}{3} = -0.762$$



Updating weights & Biases

$w1 = 0.1 + 0.01 * 1.653 = 0.1165$

$b1 = 0.1 + 0.01 * 0.762 = 0.1076$

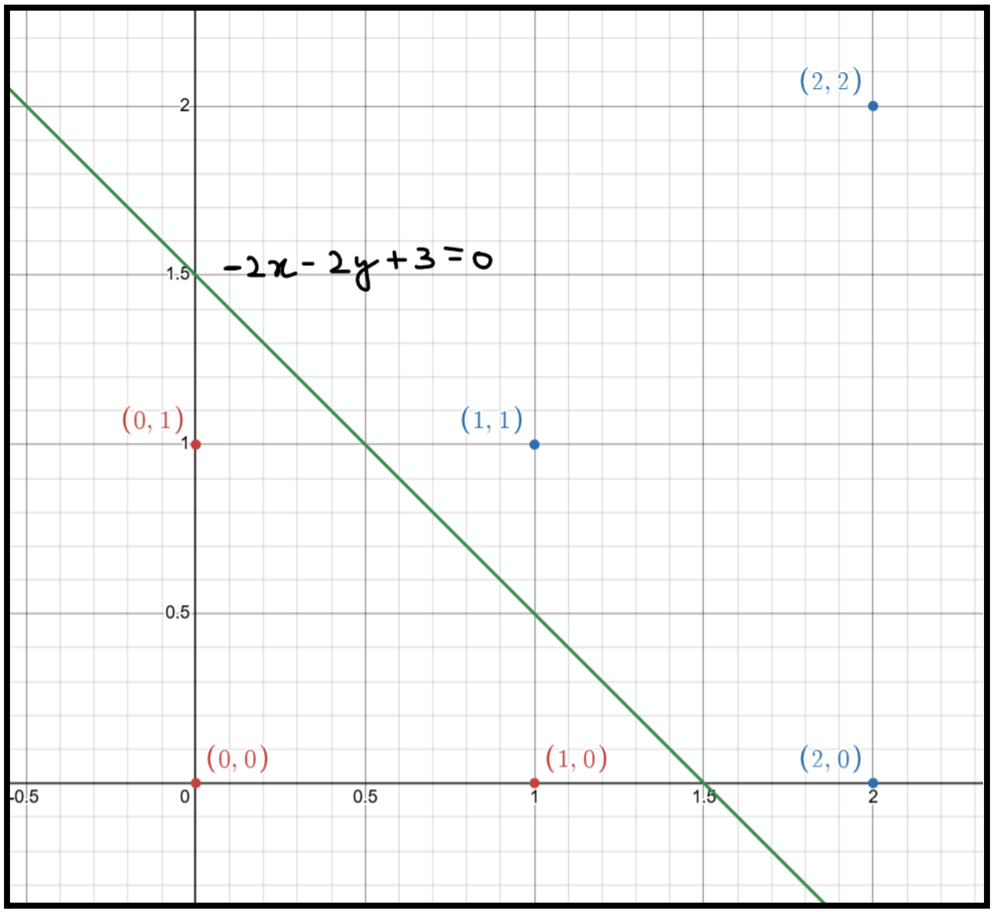
$w2 = 0.3 + 0.01 * 0.865 = 0.3081$

$b2 = 0.1 + 0.01 * 2.540 = 0.1254$

B)

a)

Yes, **the points are linearly separable**! We can draw a straight line that completely separates the positive and negative classes.



b)

Looking at the plot:

1. The **red** points (+) are at coordinates (0,0), (1,0), and (0,1)
2. The **blue** points (-) are at coordinates (1,1), (2,2), and (2,0)

To find the maximum margin hyperplane and support vectors:

1) The hyperplane will be of the form $w_1x_1 + w_2x_2 + b = 0$

2) For support vectors on the margin, we need: $w_1x_1 + w_2x_2 + b = \pm1$ (depending on their class)

3) Looking at the geometry, the points (1,0) [+] and (1,1) [-] appear to be potential support vectors as they're closest to where the separating line would be. The point (2,0) [-] also appears to be a support vector.

4) For these support vectors: $w_1(1) + w_2(0) + b = 1$ (for point (1,0)) $w_1(1) + w_2(1) + b = -1$ (for point (1,1)) $w_1(2) + w_2(0) + b = -1$ (for point (2,0))

5) From the first two equations: $w_1 + b = 1$ $w_1 + w_2 + b = -1$ Subtracting these: $w_2 = -2$

6) From the first and third equations: $w_1 + b = 1$ $2w_1 + b = -1$ Subtracting these: $w_1 = -2$

Therefore:

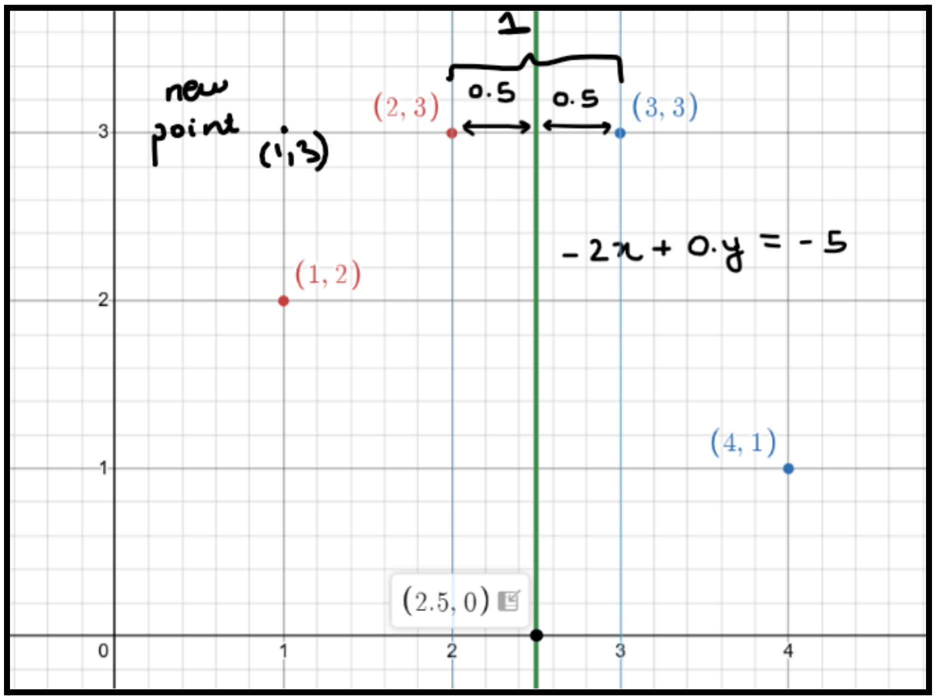
- **Weight vector w = [-2, -2]**
- Bias term b = 3
- **Support vectors are: (1,0), (1,1), (0,1), (2,0)**

The maximum margin hyperplane equation is: $\mathbf{-2x_1 - 2x_2 + 3 = 0}$

You can verify this separates the classes correctly and has maximum margin by checking that:

1) All **positive** points satisfy: $-2x_1 - 2x_2 + 3 \geq 1$
2) All **negative** points satisfy: $-2x_1 - 2x_2 + 3 \leq -1$
3) The support vectors lie exactly on the margins (= ±1)

C)



Let's solve this with w1 = -2, w2 = 0, b = 5.

Let's solve each part:

(a) Calculate the margin of the classifier:

The margin of a linear SVM classifier is given by 1/||w||, where ||w|| is the magnitude of the weight vector.

$||w|| = \sqrt{(w1^2 + w2^2)}$
$||w|| = \sqrt{((-2)^2 + 0^2)}$
$||w|| = \sqrt{4}$
$||w|| = 2$

Therefore, **margin = 1.** I.e distance between the two boundaries

(b) Identify the support vectors:

To find support vectors, we need to find points that satisfy:
|w1x1 + w2x2 + b| = 1

For each point, let's calculate w1x1 + w2x2 + b = -2x1 + 5:

Point 1 (1,2): -2(1) + 5 = 3
Point 2 (2,3): -2(2) + 5 = 1 ✓
Point 3 (3,3): -2(3) + 5 = -1 ✓
Point 4 (4,1): -2(4) + 5 = -3

Therefore, **sample 2 and 3 are support vectors i.e (2,3) and (3,3)**

since they give values of 1 and -1 respectively.

(c) Predict the class of new point (1,3):

f(x) = w1x1 + w2x2 + b
f(x) = -2(1) + 0(3) + 5
f(x) = -2 + 0 + 5
f(x) = 3

Since f(x) > 0, this point is classified as class +1.

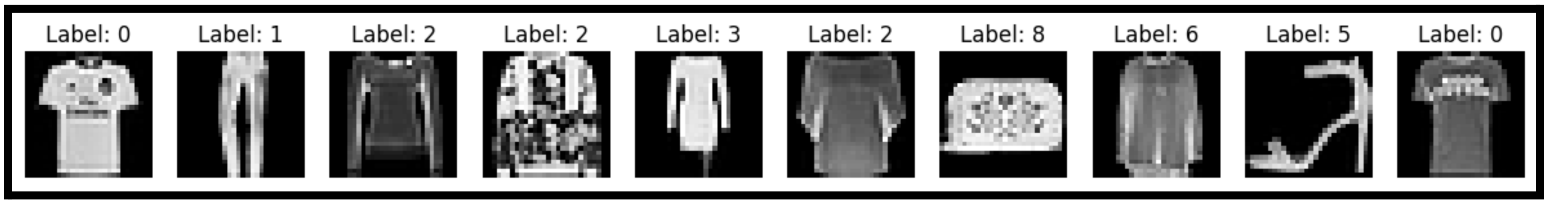
To verify:
- If f(x) > 0, class = +1
- If f(x) < 0, class = -1

Therefore, the **new point** (x1=1, x2=3) is classified as **+1.**

Section C (Algorithm implementation using packages)

1. Data Preprocessing and Visualization (1 Point)

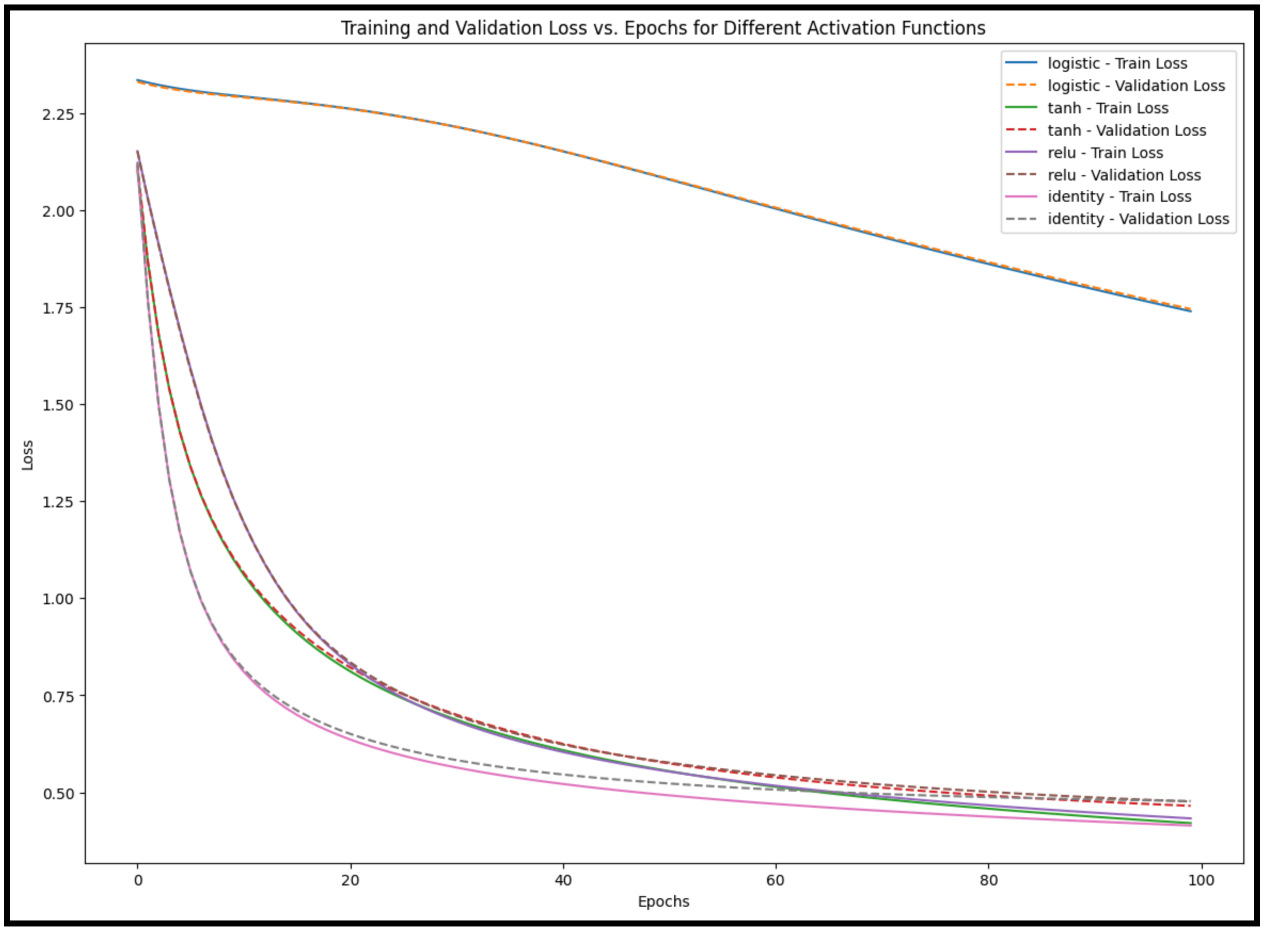
- **Data Splitting and Preprocessing:** For this assignment, I used the first 8000 images from the Fashion-MNIST training set as the training dataset, and the first 2000 images from the test set as the testing dataset. Each image's pixel values were scaled from the original range (0–255) to a normalized range (0–1), improving model convergence during training by ensuring features are on a similar scale.
- **Visualization:** Ten images from the test set were visualized to verify data integrity and observe the nature of samples within the dataset. The grayscale images showed clear patterns, each labeled by category (e.g., shirts, pants), which confirmed the appropriateness of the images for training a classifier.



2. Training MLP Classifier with Different Activation Functions (4 Points)

- **Model Configuration:** An MLP Classifier with three layers of sizes [128, 64, 32] was configured, and it was trained for 100 iterations with a batch size of 128. The 'adam' solver was chosen for its efficiency with large datasets, and the learning rate was set to 2e-5.
- **Activation Functions Tested:** I experimented with four activation functions — Logistic, Tanh, ReLU, and Identity — to observe their effects on training and validation losses.
- **Results and Observations:**
  - **Loss vs. Epochs:** For each activation function, training and validation loss were tracked across epochs to assess model performance and convergence behavior. Plots revealed that Tanh produced the lowest validation and training losses, indicating better overall learning and generalization.

- **Best Activation Function:** Tanh was observed to provide the most stable and lowest loss over 100 epochs. Logistic struggled to converge, and Identity had a linear response, making it less suitable for complex representations. ReLU performed well initially but occasionally plateaued, likely due to its zeroing effect, which can impact gradient flow.

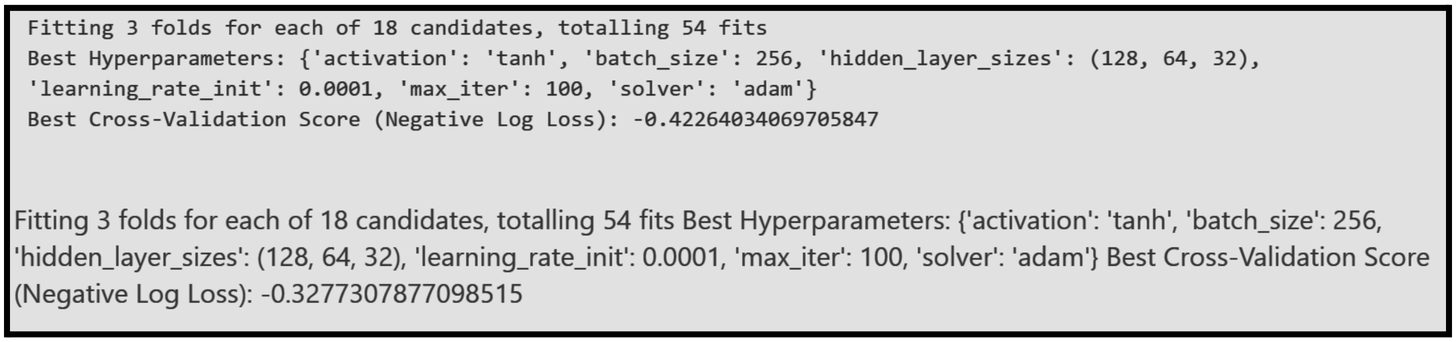


3. Hyperparameter Optimization Using Grid Search (3 Points)

- **Grid Search Objective:** To further optimize the classifier, I performed a grid search using Tanh (the best activation function from part 2) and experimented with variations in solver type (‘adam’ and ‘sgd’), batch size (64, 128, 256), and learning rate (1e-4, 2e-5, 1e-5).
- **Optimal Hyperparameters:** The grid search results indicated that the optimal hyperparameters were:
    - **Solver:** ‘adam’
    - **Batch Size:** 256
    - **Learning Rate:** 1e-4
    - **Cross-Validation Score (Negative Log Loss):** -0.3277
- **Explanation:** The larger batch size and higher learning rate enabled more efficient learning, while the ‘adam’ solver proved most effective at handling complex loss

surfaces. The best parameters were selected based on the lowest negative log loss in cross-validation, ensuring the classifier was well-optimized for the dataset.

-

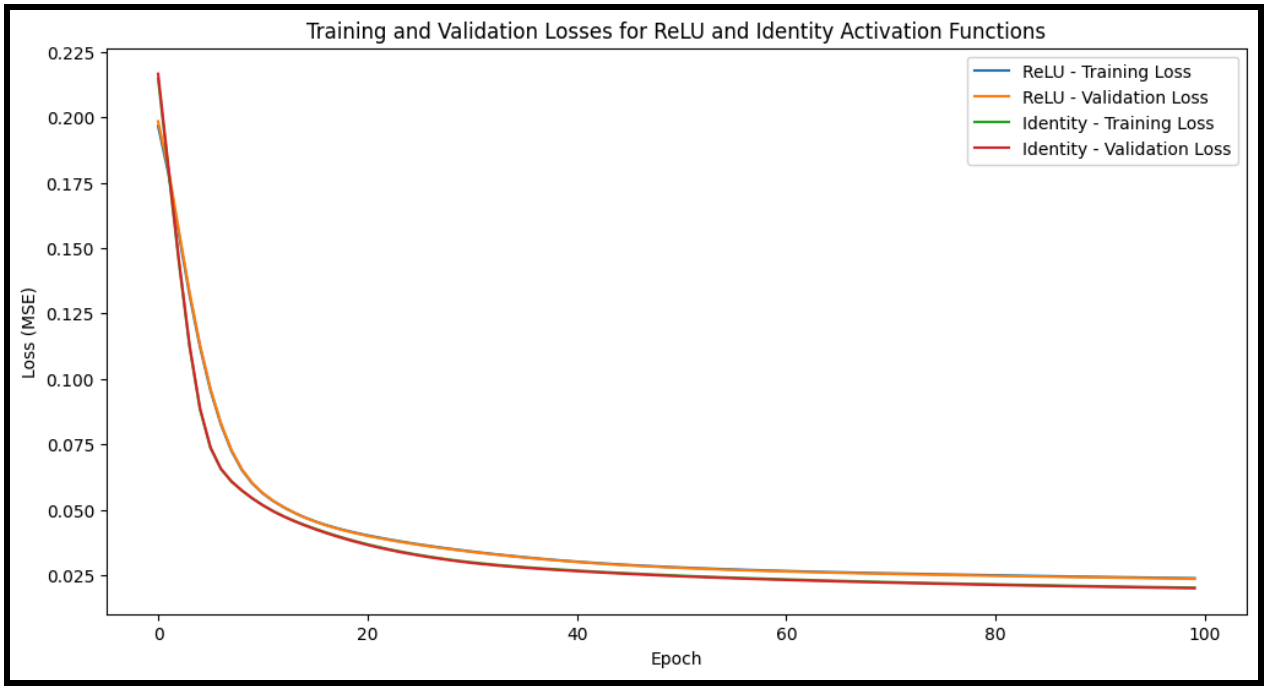


```
Fitting 3 folds for each of 18 candidates, totalling 54 fits
Best Hyperparameters: {'activation': 'tanh', 'batch_size': 256, 'hidden_layer_sizes': (128, 64, 32),
'learning_rate_init': 0.0001, 'max_iter': 100, 'solver': 'adam'}
Best Cross-Validation Score (Negative Log Loss): -0.42264034069705847
```

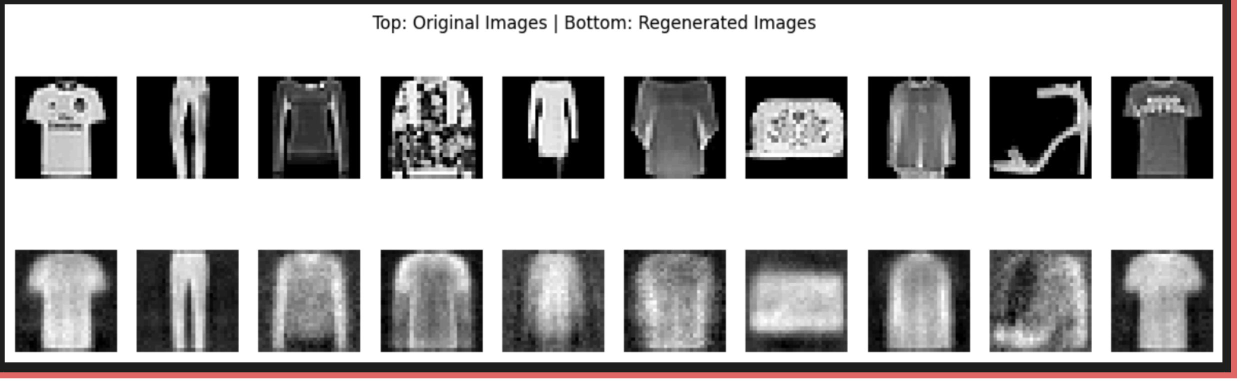
```
Fitting 3 folds for each of 18 candidates, totalling 54 fits Best Hyperparameters: {'activation': 'tanh', 'batch_size': 256,
'hidden_layer_sizes': (128, 64, 32), 'learning_rate_init': 0.0001, 'max_iter': 100, 'solver': 'adam'} Best Cross-Validation Score
(Negative Log Loss): -0.3277307877098515
```

4. MLP Regressor for Image Regeneration (4 Points)

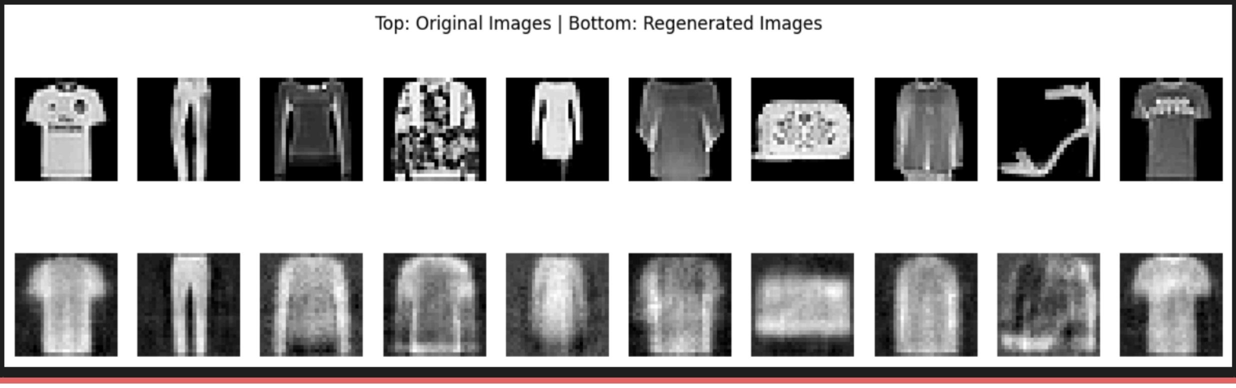
- **Model Configuration:** I designed a 5-layer MLP Regressor with the layer sizes [128, 64, 32, 64, 128] to achieve image regeneration (or autoencoding) — a process where the input is regenerated as the output to learn compressed image representations.
- **Training Process:** Two versions of the MLP Regressor were trained using ReLU and Identity as activation functions, with a constant learning rate of 2e-5. The training ran for 100 epochs, tracking the mean squared error (MSE) between input and output.
- **Results and Image Regeneration Quality:**
    - **Loss Curves:** Both ReLU and Identity activation functions showed consistent reductions in training and validation losses, confirming effective learning.
    - **Image Regeneration Observations:**
        - **ReLU Model:** Generated images retained basic contours but appeared blurred. ReLU’s nature (setting negative values to zero) can cause issues in capturing gradients, resulting in less detailed output.
        - **Identity Model:** Produced slightly clearer images with smoother gradients, thanks to its linear nature, which maintained gradient flow better than ReLU.
    - **Conclusion:** Both models achieved smooth convergence, and the Identity activation was marginally better at preserving fine details, but both struggled with capturing high levels of detail. More complex architectures or different activation functions could improve regeneration quality.



```
ReLU Model Regenerated Images:
```



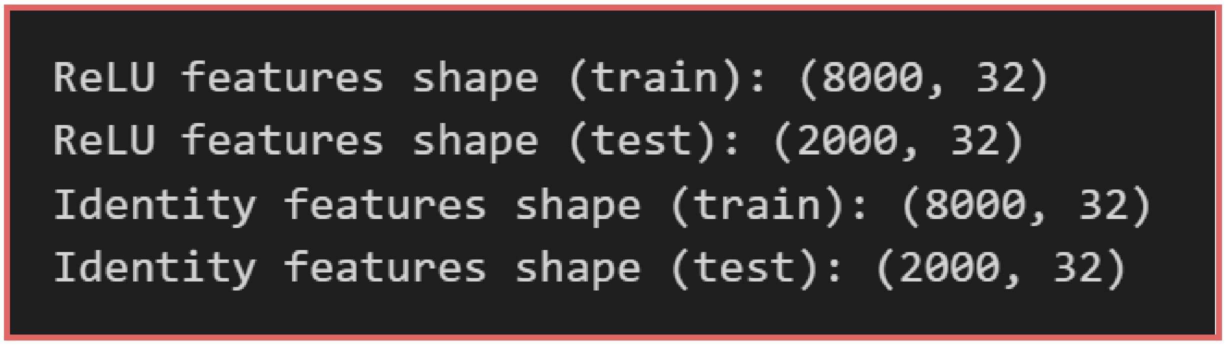
```
Identity Model Regenerated Images:
```



5. Feature Extraction and Secondary Classification (3 Points)

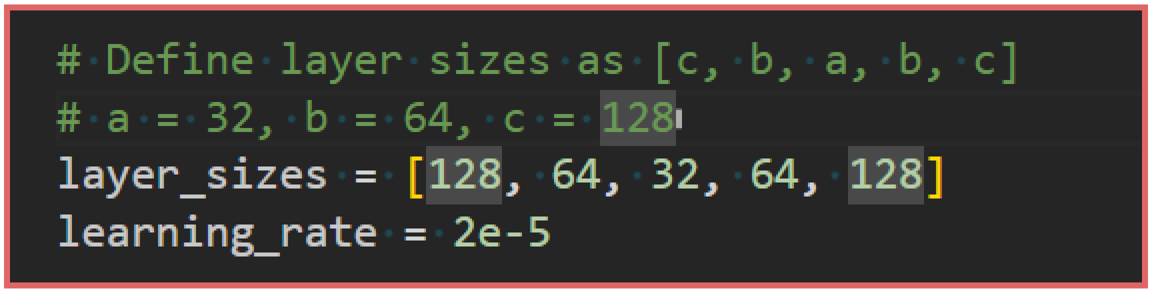
- **Feature Extraction:** Using the trained ReLU and Identity networks, I extracted feature vectors of size ‘a’ (32) from the bottleneck layer for each image. This layer’s output served as a compressed representation of the image, preserving essential information while reducing dimensionality.
- **Secondary Classifier Training:** Using these extracted features as inputs, I trained two smaller MLP classifiers, each with two layers of size ‘a’ and trained them for 200 iterations.
- **Results:**
    - **ReLU Features Classifier Accuracy:** 0.7740
    - **Identity Features Classifier Accuracy:** 0.8115
- **Comparison and Explanation:**
    - The smaller classifiers achieved good accuracy, comparable to the initial model, demonstrating that the compressed feature vectors retained essential visual patterns like edges and shapes.
    - **Why This Approach Works:**
        - **Feature Compression:** The bottleneck layer distilled critical information into a lower-dimensional representation, focusing on key features while discarding irrelevant details.
        - **Dimensionality Reduction:** Reduced dimensions helped the smaller classifiers avoid overfitting, making learning more efficient.
        - **Knowledge Transfer:** The pretrained networks captured high-level patterns, enabling the secondary classifiers to perform well with limited input dimensions, effectively leveraging the pretraining knowledge for robust classification.

Verifying Shapes of features



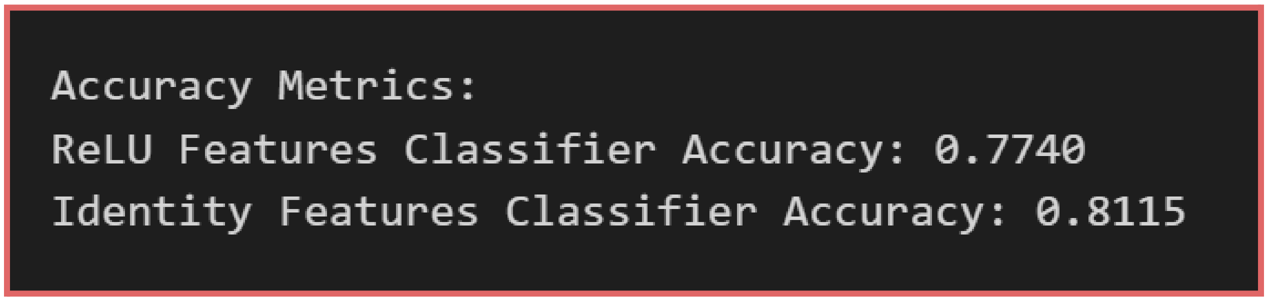
```
ReLU features shape (train): (8000, 32)
ReLU features shape (test): (2000, 32)
Identity features shape (train): (8000, 32)
Identity features shape (test): (2000, 32)
```

Layer Architecture



```
# Define layer sizes as [c, b, a, b, c]
# a = 32, b = 64, c = 128
layer_sizes = [128, 64, 32, 64, 128]
learning_rate = 2e-5
```

Accuracy Metrics



```
Accuracy Metrics:
ReLU Features Classifier Accuracy: 0.7740
Identity Features Classifier Accuracy: 0.8115
```

Section B (Scratch Implementation)

Neural Network Implementation Report for MNIST Classification

(bonus)

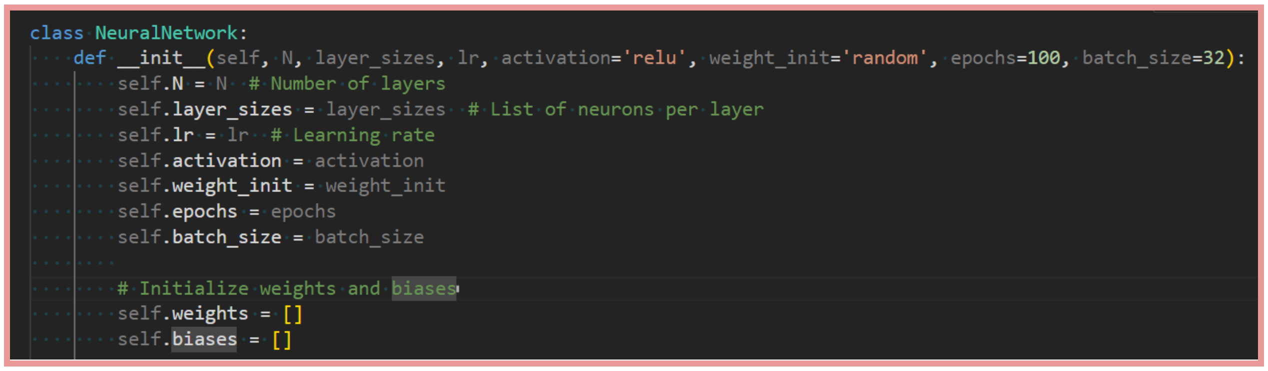
a)

Neural Network Components

1. Initialization
    1. The network takes the number of layers, layer sizes, learning rate, activation function, weight initialization type, number of epochs, and batch size.
    2. Weight Initialization: Weights are initialized using 'He' or 'Xavier' methods based on the activation function.
2. Activation Functions
    1. activation and activation_derivative: Define and differentiate the activation function, critical for forward and backward passes.
3. Training (fit)
    1. Data is shuffled and split into batches.
    2. Forward Pass: Calculates activations.
    3. Backward Pass: Updates weights using gradients.
    4. Early Stopping: Stops training if validation loss doesn't improve for a set patience.
4. Loss Calculation (compute_loss)
    1. Cross-Entropy Loss: Measures prediction accuracy against true labels.
5. Prediction and Scoring
    1. predict: Performs a forward pass and outputs predicted labels.
    2. predict_proba: Returns probabilities for each class.

   3. score: Computes model accuracy by comparing predictions to true labels.

6. Running the Model
   1. Model Creation: Initializes a neural network with three layers.
   2. Training: The fit method trains the model on MNIST data.
   3. Evaluation: score reports test accuracy.



```
class NeuralNetwork:
    def __init__(self, N, layer_sizes, lr, activation='relu', weight_init='random', epochs=100, batch_size=32):
        self.N = N  # Number of layers
        self.layer_sizes = layer_sizes  # List of neurons per layer
        self.lr = lr  # Learning rate
        self.activation = activation
        self.weight_init = weight_init
        self.epochs = epochs
        self.batch_size = batch_size
        
        # Initialize weights and biases
        self.weights = []
        self.biases = []
```

b)

Implemented **Activations Class** with all required Activation functions as static method

```
class Activations:
    @staticmethod
    def sigmoid(x, derivative=False):
        sigmoid_x = 1 / (1 + np.exp(-x))
        if derivative:
            return sigmoid_x * (1 - sigmoid_x)
        return sigmoid_x

    @staticmethod
    def tanh(x, derivative=False):
        tanh_x = np.tanh(x)
        if derivative:
            return 1 - tanh_x ** 2
        return tanh_x

    @staticmethod
    def relu(x, derivative=False):
        if derivative:
            return (x > 0).astype(float)
        return np.maximum(0, x)

    @staticmethod
    def leaky_relu(x, alpha=0.01, derivative=False):
        if derivative:
            dx = np.ones_like(x)
            dx[x < 0] = alpha
            return dx
        return np.where(x > 0, x, alpha * x)

    @staticmethod
    def softmax(x):
        exp_x = np.exp(x - np.max(x, axis=1, keepdims=True))
        return exp_x / np.sum(exp_x, axis=1, keepdims=True)
```

c)

Implemented `WeightInitializer` with all required Activation functions as static method

```
class WeightInitializer:
    @staticmethod
    def zero_init(input_size, output_size):
        weights = np.zeros((input_size, output_size))
        bias = np.zeros((1, output_size))
        return weights, bias

    @staticmethod
    def random_init(input_size, output_size):
        # Using He initialization scaling factor
        scale = np.sqrt(2.0 / input_size)
        weights = np.random.uniform(-scale, scale, (input_size, output_size))
        bias = np.zeros((1, output_size))
        return weights, bias

    @staticmethod
    def normal_init(input_size, output_size):
        # Using Xavier/Glorot initialization scaling factor
        scale = np.sqrt(2.0 / (input_size + output_size))
        weights = np.random.normal(0, scale, (input_size, output_size))
        bias = np.zeros((1, output_size))
        return weights, bias
```

d)

Running the model for 40 iterations as more than that was taking too much time

```
# Initialize model
    model = NeuralNetwork(
        N=6,  # input + 4 hidden + output
        layer_sizes=[784, 256, 128, 64, 32, 10],
        lr=2e-3,
        activation=activation,
        weight_init=weight_init,
        epochs=40,
        batch_size=128
    )
```

Running the network for this configurations

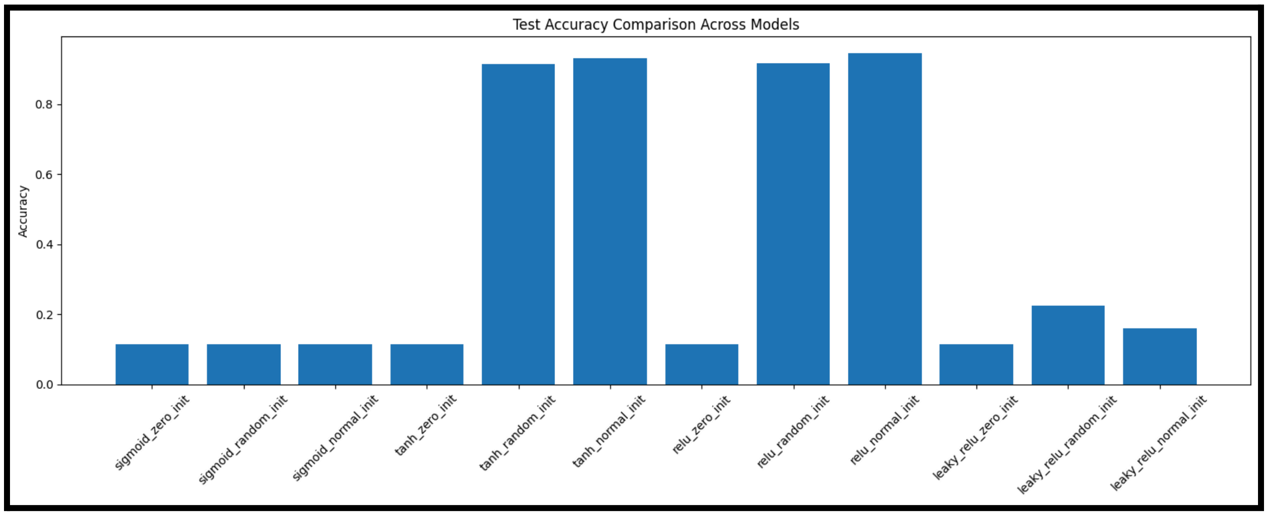
```
configs = [
    ('sigmoid', 'zero_init'),
    ('sigmoid', 'random_init'),
    ('sigmoid', 'normal_init'),
    ('tanh', 'zero_init'),
    ('tanh', 'random_init'),
    ('tanh', 'normal_init'),
    ('relu', 'zero_init'),
    ('relu', 'random_init'),
    ('relu', 'normal_init'),
    ('leaky_relu', 'zero_init'),
    ('leaky_relu', 'random_init'),
    ('leaky_relu', 'normal_init'),
]
```

Also, in back propagation used clipping to that the gradient doesn't overflow float representation

```
delta = np.clip(delta, -1e10, 1e10)
```

Implemented, Early Stop to prevent Overfitting, by having a patience counter

```
                # Early stopping check
                if patience_counter >= patience:
                    print(f"Early stopping at epoch {epoch}")
                    self.weights = best_weights
                    self.biases = best_biases
                    break
```



Test Results

Best Performing Models

- **ReLU + Normal Init**: 94.60% accuracy (Loss: 0.1825)
- Tanh + Normal Init: 93.27% accuracy (Loss: 0.2503)
- ReLU + Random Init: 91.80% accuracy (Loss: 0.2809)

Detailed Analysis

1. Initialization

- In Zero-Init a lot of oscillations are seen in the test-validation graph.
    - When all the weights are initialized to 0, it means that all the neurons in a layer will perform the same computations and learn the same features. This is because the gradients computed during backpropagation will be identical for all the neurons in a layer, leading to the same updates being applied to the weights.
- Tanh and Relu perform identically both giving good performance

2. Activation

Sigmoid:

- The sigmoid activation function performed the worst, with a consistent test accuracy of around 11.35% (random chance).
- The key issue with sigmoid is the **vanishing gradient problem**. As the network gets deeper, the gradients computed during backpropagation become very small, making it difficult for the network to learn effectively.

- Additionally, the sigmoid function outputs values between 0 and 1, **which are not zero-centered.** This can further contribute to the vanishing gradient issue.

Tanh:

- Tanh performed **much better than sigmoid**, with the best model (Tanh + Normal Initialization) achieving a test accuracy of 93.27%.
- Tanh is an improvement over sigmoid because it is zero-centered, which helps alleviate the vanishing gradient problem to some extent.
- The strong gradients provided by tanh, especially in the middle range of the activation, allow the network to learn more effectively compared to sigmoid.

ReLU (Rectified Linear Unit):

- ReLU was the best-performing activation function, with the best model (ReLU + Normal Initialization) achieving a test accuracy of 94.60%.
- ReLU is a non-linear activation that does not suffer from the vanishing gradient problem, as the gradient is either 0 or 1 for negative or positive inputs, respectively.
- ReLU introduces sparsity in the network, as many neurons will output 0 for negative inputs. This helps the network learn more efficient and discriminative features.
- ReLU also has the advantage of being computationally efficient, as the function is simply the maximum of 0 and the input value.

Leaky ReLU:

- Leaky ReLU, a variant of ReLU that allows a small non-zero gradient for negative inputs, performed worse than ReLU.
- The best Leaky ReLU model (Leaky ReLU + Random Initialization) achieved a test accuracy of 22.47%.
- The choice of the small non-zero gradient in Leaky ReLU (typically around 0.01) can be tricky to tune and may not be optimal for all datasets and network architectures.
- The instability caused by the non-zero gradient for negative inputs may have contributed to the poorer performance compared to the standard ReLU.

Softmax:

- Softmax is not an activation function used in the hidden layers, but rather in the output layer to convert the logits into probability distributions.

- The softmax function is crucial for the final classification task, as it allows the network to output probabilities for each class, which can be used for making predictions and calculating loss.

Key Takeaways

Initialization Impact

- Normal initialization provides stable learning.
- Zero initialization prevents learning.
- Random initialization offers good baseline performance.

Activation Choice

- ReLU: Best for deep networks
- Tanh: Strong alternative
- Modern architectures favor ReLU variants.

Best Practices

- Use normal initialization.
- Choose ReLU for deep networks.
- Avoid zero initialization.
- Monitor loss curves for stability.