INDRAPRASTHA INSTITUTE OF INFORMATION TECHNOLOGY
NEW DELHI

Department of Computer Science & Engineering

CSE 343/ECE 363 : Machine Learning

Dr. Jainendra Shukla

Assignment - 3: Perceptron / MLP / SVM

Anant Kumar Kaushal (2022067)

SECTION - A

$x = [1, 2, 3]$
$Y = [3, 4, 5]$ [Given]

let initial weights and biases be : [since these are normalized]

$w_1 = 0.5$, $w_2 = 0.5$

$b_1 = 0.5$, $b_1 = 0.5$

$\eta = 0.01$ (Given)

I <u>Forward Pass</u>

$$\text{INPUT} \ \bigcirc \xrightarrow{w_1} \overset{b_1}{\bigcirc} \xrightarrow{w_2} \overset{b_2}{\bigcirc} \ \text{OUTPUT}$$

(hidden layer)

Output for the first layer $= \text{Relu}([x] \times w_1 + b_1)$

$= \text{Relu}([0.5, 1, 1.5] + 0.5)$

$= \text{Relu}([1, 1.5, 2])$

$= [1, 1.5, 2]$

Output for the second layer (output layer) $= \text{Relu}[h_1 \times w_2 + b_2]$

$= [[1, 1.5, 2] \times 0.5 + 0.5]$

$= [[0.5, 0.75, 1] + 0.5]$

$= [1, 1.25, 1.5]$

II Backpropagation

1 Gradient of loss:

$$L = \frac{1}{2} \sum_{i=1}^{2} (y_i - \hat{y}_i)^2$$

$$L = \frac{1}{2} \sum \left(w^T x_i + b - \hat{y}_i \right)^2$$

$$\frac{\partial L}{\partial b} = \left(w^T x_i + b - \hat{y}_i \right) \times 1$$

And $w^T x + b =$

Now for output layer

$$w^T x_2 + b = y_2 = [1, 1.25, 1.5]$$

$$\hat{y}_2 = [3, 4, 5]$$

~~∴ gradient b_2 =~~

$$\frac{\partial L}{\partial b} = \left([1, 1.25, 1.5] - [3, 4, 5] \right)$$

$$= [-2, -2.75, -3.45]$$

Now for b_2:

$$\text{grad\_}b_2 = \sum \text{gradient}$$

$$= (-2) + (-2.75) + (-3.45)$$

$= -8.25$

Since we want to normalize the above computation so divide the above by gradient by 3.

$\therefore$ grad_b_2 $= \dfrac{-8.25}{3} = 2.75$

Illy grad_w_2 $= \dfrac{\partial L}{\partial b} \times \text{Relu}'\cancel{(\text{input} \cdot w_1 + b_1)}^{\,1} \times \text{input}$

$= \begin{bmatrix} -2 & -2.75 & -3 \end{bmatrix} \begin{bmatrix} 1 \\ 1.5 \\ 2 \end{bmatrix}$

$= (-2) + (-4.125) + (-7)$

Divide above by to normalize it

$= \dfrac{(-2) + (-4.125) + (-7)}{3}$

$= -4.375$

<u>For the first layer:</u>

output $= \dfrac{\partial L}{\partial b} \times w_2$

$= [-2, 2.75, 3.5] \times 0.5$

$= [-1, 1.375, -1.75]$

Illy Grad_b_1 $= \dfrac{1}{3} \sum$ output

$$= \frac{(-1) + (-1.375) + (-1.75)}{3}$$

$$= -1.375$$

~~lly grad-w1 = [-1 -1.375 -1.75] [1~~

lly grad-w_1 = output × ReLU'(input × w_1 + b_1) × input

$$= \begin{bmatrix} -1 & -1.375 & -1.75 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

$$= \frac{-1 - 2.75 - 5.25}{3}$$

$$= -3$$

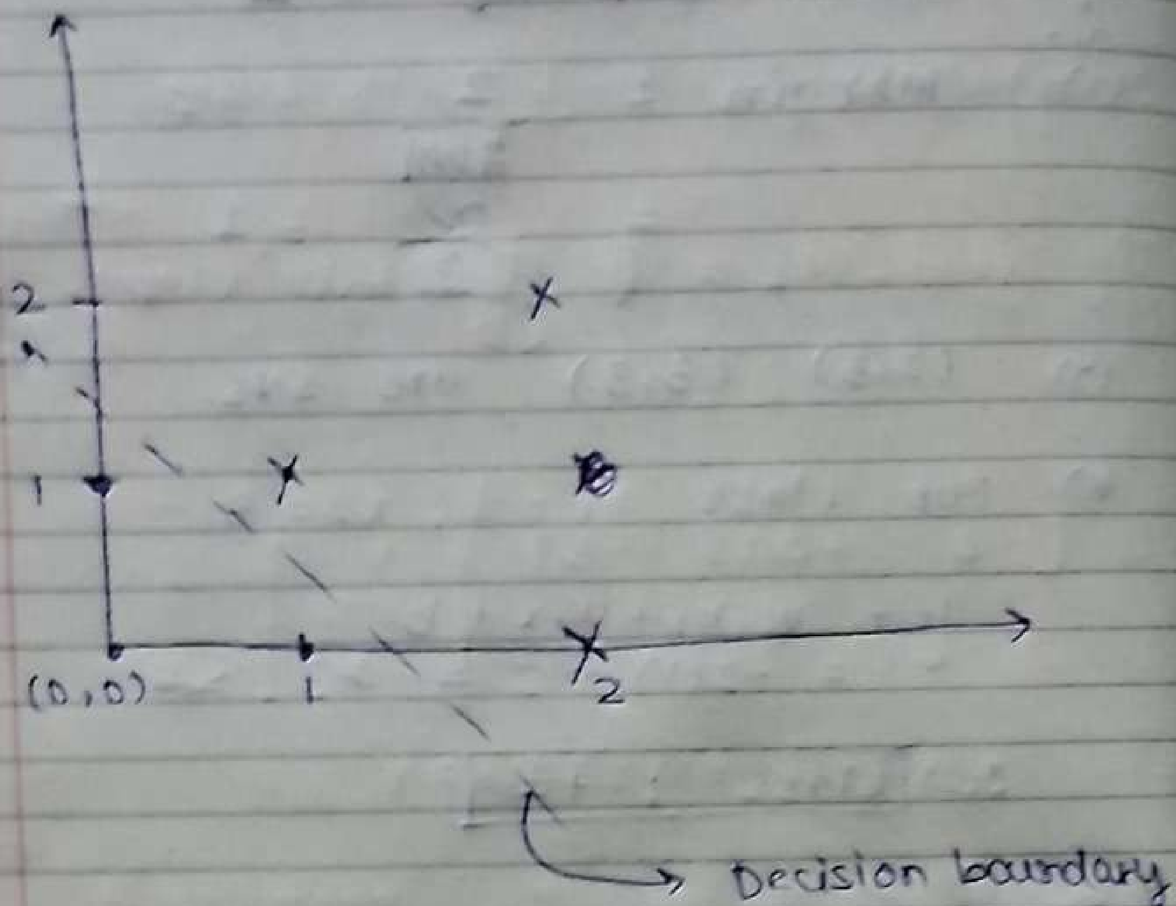
$\therefore$

$$\boxed{\begin{aligned} w_1 &= 0.5 + -\eta(\text{grad-}w_1) = 0.5 + (0.01)(3) &= 0.53 \\ b_1 &= 0.5 - \eta(\text{grad-}b_1) = 0.5 + (0.01)(1.375) &= 0.51375 \\ w_2 &= 0.5 - \eta(\text{grad-}w_2) = 0.5 + (0.01)(4.375) &= 0.54375 \\ b_2 &= 0.5 - \eta(\text{grad-}b_2) = 0.5 + (0.01)(2.75) &= 0.5275 \end{aligned}}$$

A3 w_1 =
w_2 =
b
$\|w\|$
$\therefore$
(a) mar
(b) (2,3
(c) For
y =
$\therefore$ [C

A2

(a)



$\therefore$ Point are linearly seperable as it can be seen from the plot.

(b) For '+' class:

Using $w_1 x_1 + w_2 x_2 + b = y$

$(1,0)$: $w_1 + b = 1$ ①

$(0,1)$: $w_2 + b = 1$ ②

On computing ① and ②, we get $w_1 = w_2$ and $b = 1 - w_1$

For (-) class:

$(1,1)$: $w_1 + w_2 + b = -1$ ③
$(2,0)$: $2w_1 + b = -1$

~~Using w_1~~
Put values of w_2 and b, in ③

$$w_1 + \cancel{w_1} + 1 - \cancel{w_1} = -1$$

$$w_1 = -2$$

and $w_2 = -2$

||ly $b = 3$

∴ ~~Decisi~~ Weight vector : $-2x_1 - 2x_2 = 3$

$$x_1 + x_2 = -\frac{3}{2}$$

SVs : 1. $(1,0)$ & $(2,0)$
2. $(0,1)$ & $(1,1)$

they both have same minimum distance between them.

Q3 $w_1 = -2$

$w_2 = 0$

$b = 5$

$||w|| = \sqrt{w_1^2 + w_2^2} = 2$

$\therefore$

(a) $\text{margin} = \frac{2}{||w||}$

$= \frac{\cancel{2}}{\cancel{2}} = 1$

~~(b) (2,3) ; (3,3) , are SVs~~

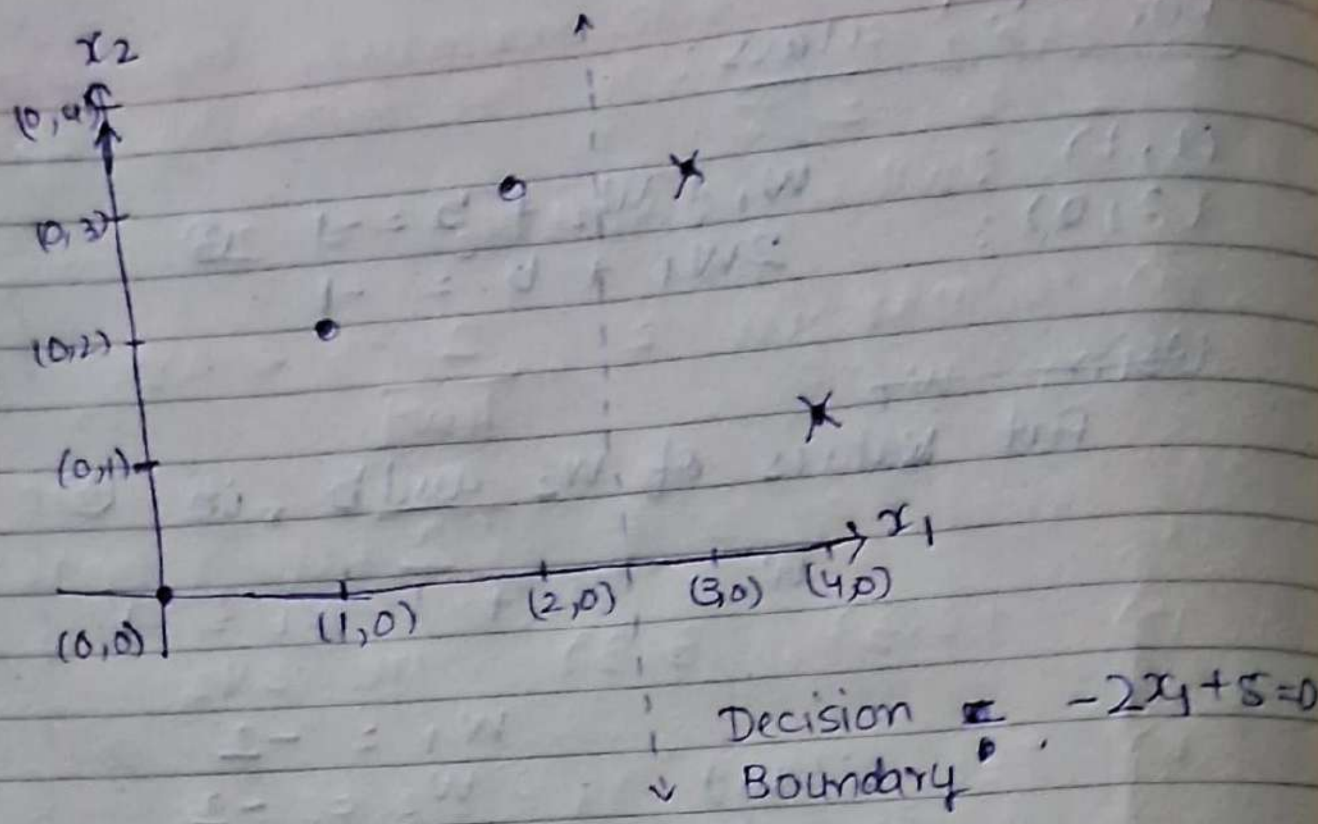
(c) For class $x_1 = 1$, $x2 = 3$

$y = w_1x_1 + w_2x + b$

$= -2(1) + 5 = +3$ $\cancel{>0}$

$\therefore$ $\boxed{\text{Class : } +1}$

(b)



$\therefore$ Hence (2,3) & (3,3) are SVs as they are closest to the desicion boundary can be seen graphically.

SECTION - B

1. Class Neural Network with mentioned parameters and functions is implemented as follows:-

```
class NeuralNetwork:
    def __init__(self, N, layer_sizes, lr, activation_func,weight_init,
epochs=100, batch_size=128):
        self.N = N
        self.layer_sizes = layer_sizes
        self.lr = lr
        self.activation_func = activation_func
        self.weight_init = weight_init
        self.epochs = epochs
        self.batch_size = batch_size
        self.weights = self.create_weights()
        self.biases = [np.zeros((1, layer_sizes[i + 1])) for i in range(N - 1)]
def compute_loss(self, X, Y):
        activations = self.forward(X)
        predictions = activations[-1]
        return -np.mean(Y * np.log(predictions + 1e-8))

    def predict_proba(self, X):
        activations= self.forward(X)
        return activations[-1]

    def predict(self, X):
        proba = self.predict_proba(X)
        return np.argmax(proba, axis=1)

    def score(self, X, Y):
        predictions = self.predict(X)
        true_labels = np.argmax(Y, axis=1)
        return np.mean(predictions == true_labels)
```

2.Activation Functions along with their gradients are implemented as follows:-

```
def sigmoid(self, Z):
        return 1 / (1 + np.exp(-Z))

    def sigmoid_derivative(self, Z):
        s = self.sigmoid(Z)
        return s * (1 - s)

    def tanh(self, Z):
        return np.tanh(Z)

    def tanh_derivative(self, Z):
        return 1 - np.tanh(Z) ** 2

    def relu(self, Z):
        return np.maximum(0, Z)

    def relu_derivative(self, Z):
        return Z > 0

    def leaky_relu(self, Z, alpha=0.01):
        return np.where(Z > 0, Z, alpha * Z)

    def leaky_relu_derivative(self, Z, alpha=0.1):
        dZ = np.ones_like(Z)
        dZ[Z < 0] = alpha
        return dZ

    def softmax(self, Z):
        expZ = np.exp(Z - np.max(Z, axis=1, keepdims=True))
        return expZ / expZ.sum(axis=1, keepdims=True)
```

3.Activation Functions along with their gradients are implemented as follows:-

```
def create_weights(self):
        matrices = []
        for idx in range(1, len(self.layer_sizes)):
            if self.weight_init == 'zero':
                matrix = np.zeros((self.layer_sizes[idx],
self.layer_sizes[idx-1]))
            elif self.weight_init == 'random':
                matrix = np.random.rand(self.layer_sizes[idx],
self.layer_sizes[idx-1]) * 0.01
            elif self.weight_init == 'normal':
                matrix = np.random.randn(self.layer_sizes[idx],
self.layer_sizes[idx-1]) * np.sqrt(2 / self.layer_sizes[idx-1])
            matrices.append(matrix)
        return matrices
```

In this code appropriate scaling factors have been implemented for ‘random’ and ‘normal’ initializations . In ‘random’ initialization weights are scaled by **0.01** to create small random weights, which can help in stabilizing initial gradients. In ‘normal’ initialization weights are scaled by **np.sqrt(2 / self.layer_sizes[idx-1])**. This is known as He initialization, which helps maintain gradient flow in deeper networks by considering the previous layer's size when scaling weights drawn from a normal distribution.

4. Pre processing and Plots Of The Respective Models :-

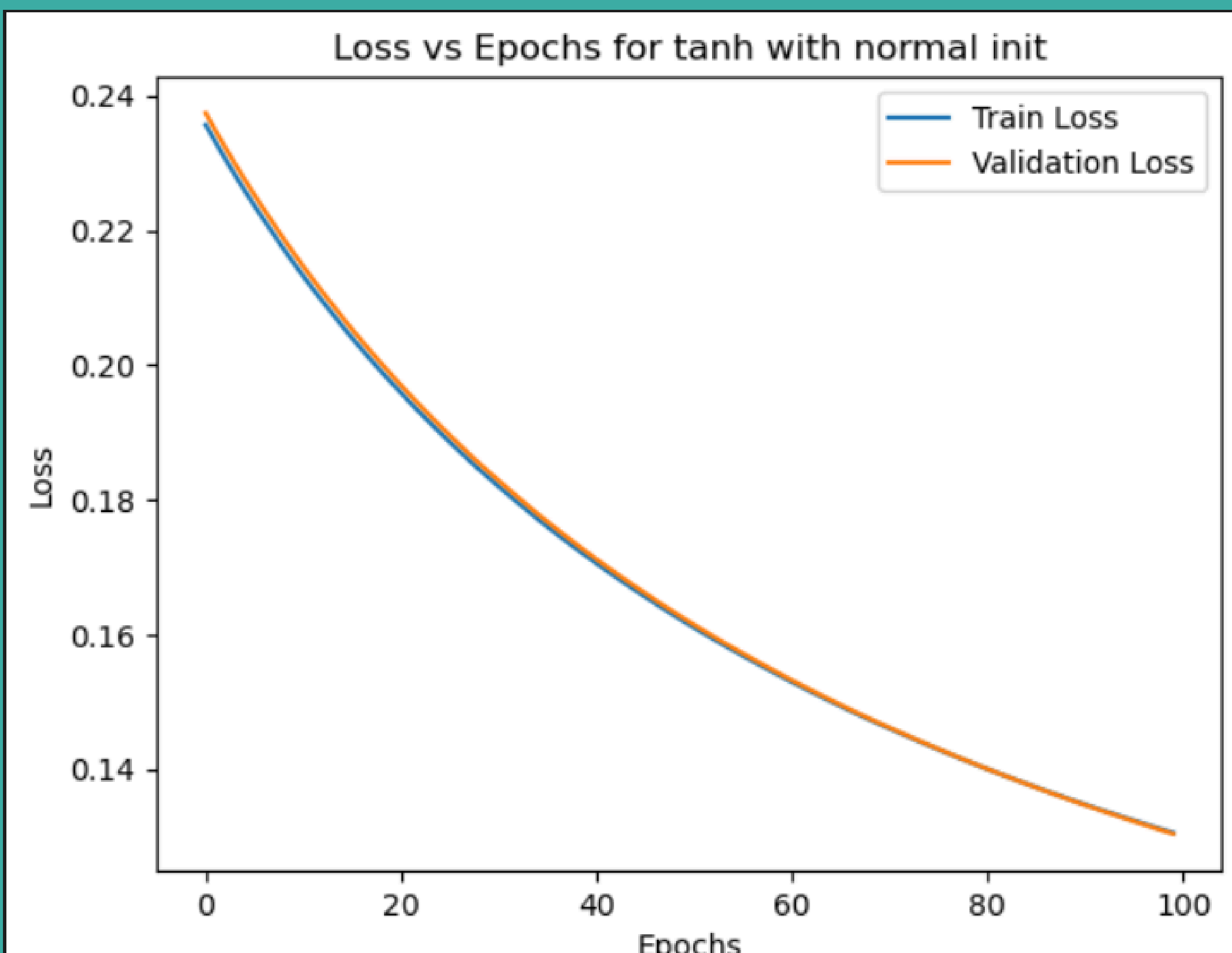
```
def load_images(file_path):
    with open(file_path, 'rb') as file:
        cool, num_images, height, width = struct.unpack(">IIII", file.read(16))
        images = np.fromfile(file, dtype=np.uint8).reshape(num_images, height * 
width) / 255.0
    return images

def load_labels(file_path):
    with open(file_path, 'rb') as file:
        cool, num_labels = struct.unpack(">II", file.read(8))
        labels = np.fromfile(file, dtype=np.uint8)
    return labels

def convert_to_one_hot(labels, classes=10):
    return np.eye(classes)[labels]
```

```
model = NeuralNetwork(
                N=6,
                layer_sizes=[784, 256, 128, 64, 32, 10],
                lr=2e-5,
                activation_func=activation,
                weight_init=init,
                epochs=100,
                batch_size=128,
            )
```

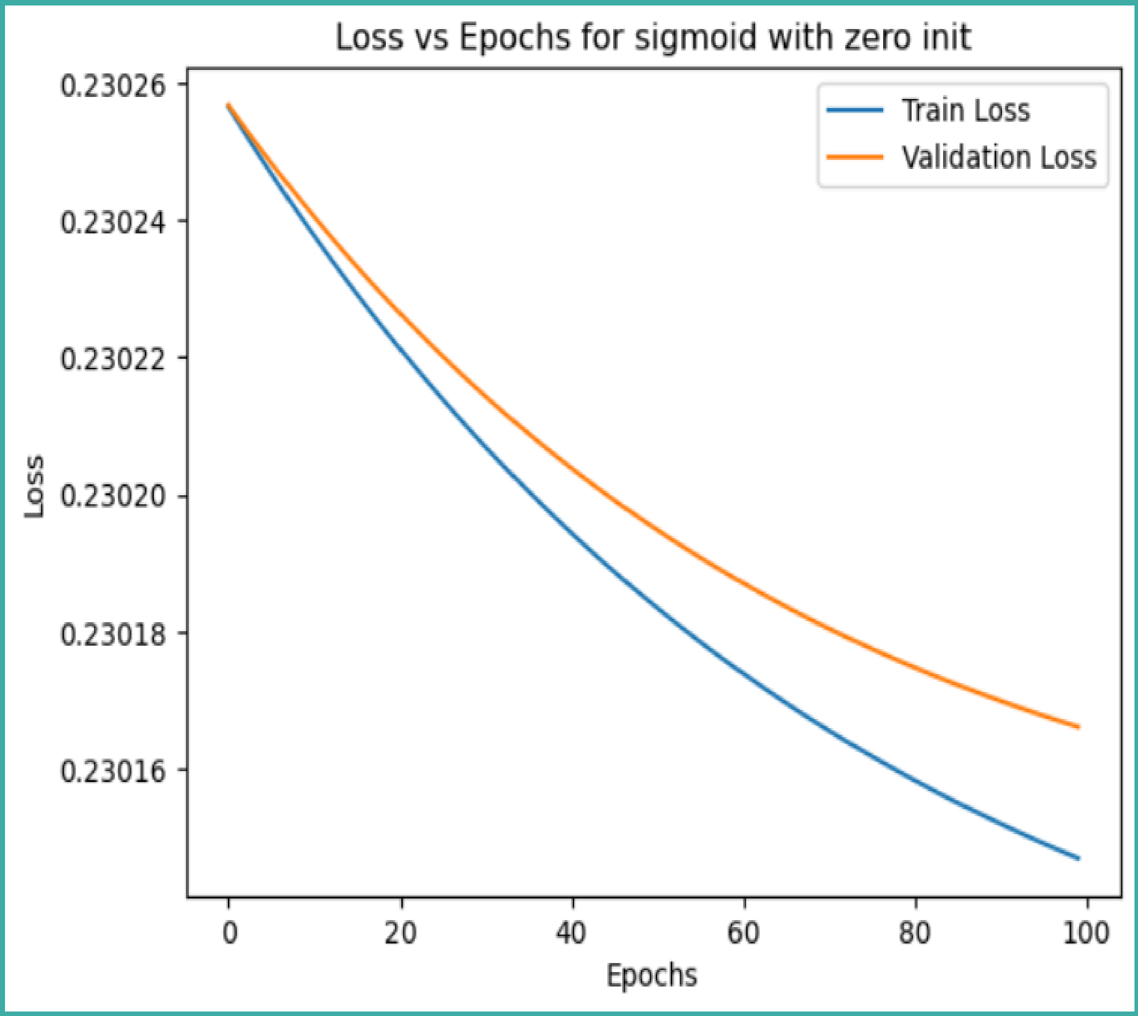
The ***tanh normal initialization*** function stands out as the best-performing method due to its high accuracy on the training set, reaching approximately 63%, which is the highest among other initialization-function combinations tested. This performance indicates that using the hyperbolic tangent (tanh) activation function combined with a normal distribution for weight initialization effectively supports the network's learning process. The training and validation losses for this setup were also closely monitored to ensure the model's generalization, indicating consistent improvement across the epochs without significant overfitting or underfitting.



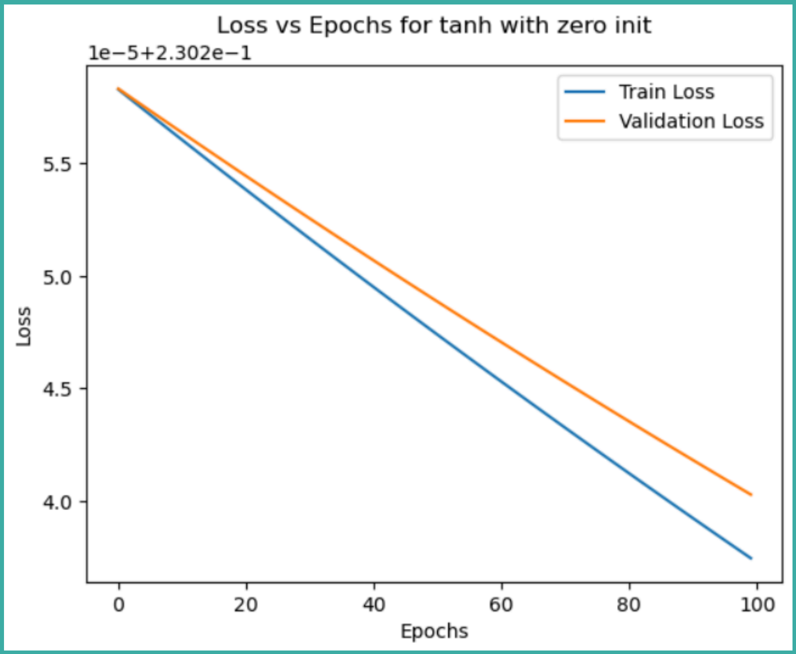

```
Epoch 100/100 - Train Loss: 0.130473
Accuracy on train set: 0.63798828125
Validation Loss: 0.130331
```

All three ***zero initialization*** functions are suboptimal for training neural networks because it initializes all weights to the same value, causing neurons to produce identical outputs and gradients during the forward and backward passes. This symmetry prevents the network from learning effectively, as each neuron in a layer receives the same updates and fails to diversify in function. Plots for the same are shown below :

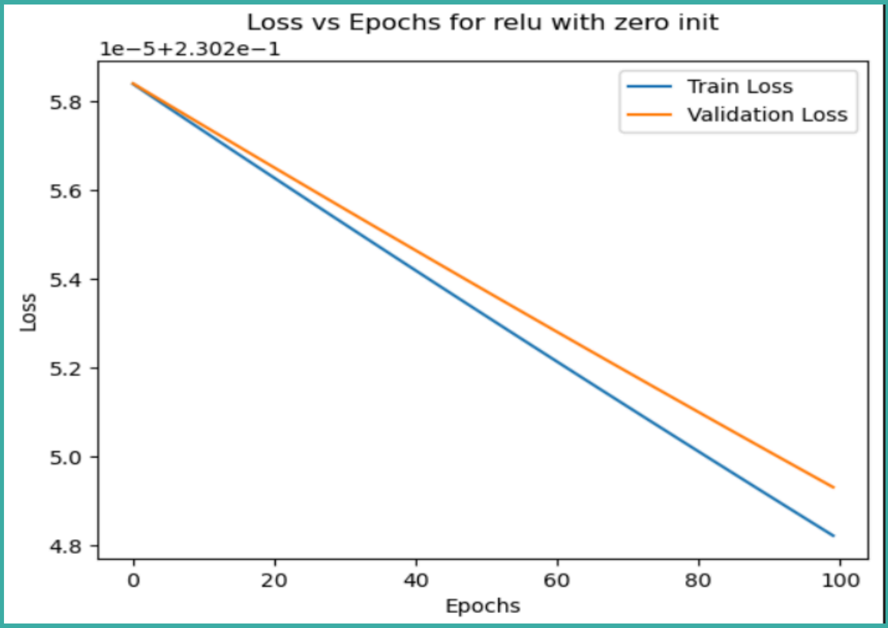
SIGMOID WITH ZERO INIT:



TANH WITH ZERO INIT:



LEAKY WITH ZERO INIT:



RELU_LEAKY WITH ZERO INIT: