
Background and Method Introduction: -

Linear classifier: - A linear classifier works by identifying a linear decision boundary in the feature space that distinguishes between classes, it is a machine learning model that is used to do binary and multi-class classification tasks which can be effective and fast. The decision boundary is defined as a linear combination of input features weighted by learned parameters and this is useful for identifying the images with certain level of accuracy. There are three viewpoints in a linear classifier that are Algebraic viewpoint, visual viewpoint, Geometric viewpoint.

Application of Linear classifier in image classification: - Linear classifiers are commonly used as baseline models for image classification because they are simple and easy to understand. While they may not capture complex patterns as well as deep learning models, they are an excellent starting point for classification tasks, particularly with small to medium-sized datasets. In image classification tasks linear classifiers assign labels to the images based on their features and thus help the deep learning models to make further improvements in predictions and also, they are useful to predict a certain image with accuracy.

Dataset and Tasks Description: -

CIFAR-10 Dataset:

The CIFAR-10 dataset is a well-known benchmark dataset for image classification. It consists of 60,000 32x32 colour images in 10 classes, with 6,000 images per class. The classes represent various common objects such as airplanes, automobiles, birds, cats, deer, dogs, frogs, horses, ships, and trucks. The dataset is divided into training and test sets, with the training set used for model training and the test set used for evaluation.

Classification Tasks:

We are using the linear classifier to predict the correct class label for each image in the CIFAR-10 dataset. During training, the custom linear classifier learns the optimal parameters (weights and biases) that define the decision boundary between different classes. The trained classifier is then evaluated on the test set to assess its performance in accurately classifying unseen images. The performance metrics include classification accuracy and loss. We Use forward and backward computations and then we used gradient descent optimization to classify the images based on their labels.

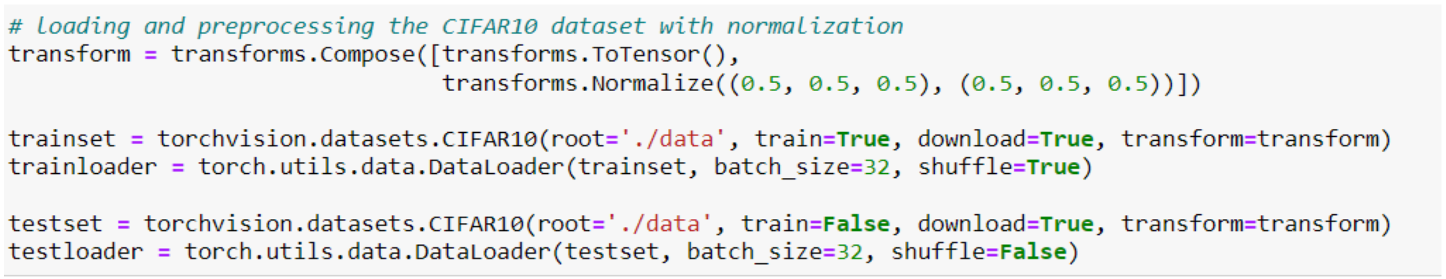
Algorithms Used: -

1.In-built PY-torch packages and libraries: - In this we have used the Inbuilt packages and ran the linear classification to predict losses and accuracy at each pass.

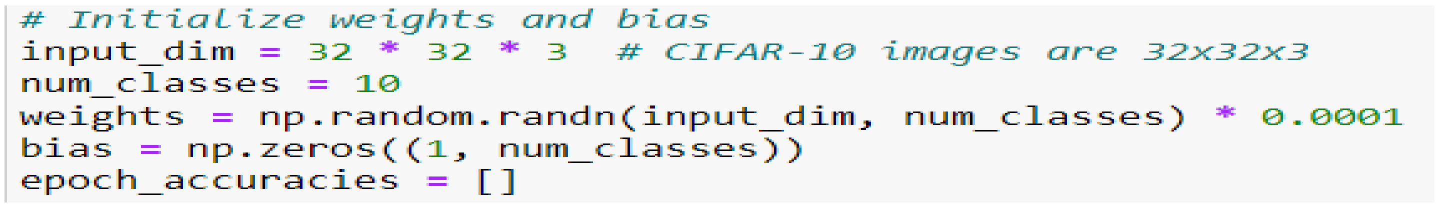
2. Custom linear classifier with Forward and Backward computation: -

The Custom linear classifier uses a forward pass, backward pass, SoftMax function and a loop for gradient descent optimization to carry out multi-class classification. The classifier is composed of weights, biases, activation functions (SoftMax), and input features (image pixel values).

In the first step we load the CIFAR-10 dataset and then initialize the weight and bias matrices for the CIFAR-10 dataset with various factors and initialize the epoch accuracies list to store the accuracy at each epoch.

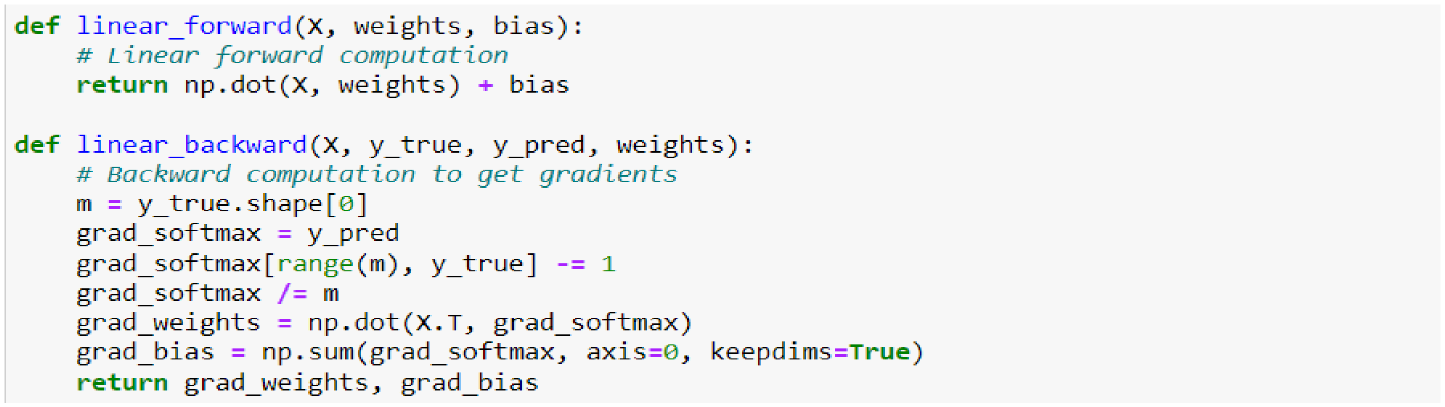


```
# loading and preprocessing the CIFAR10 dataset with normalization
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=32, shuffle=True)

testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=32, shuffle=False)
```



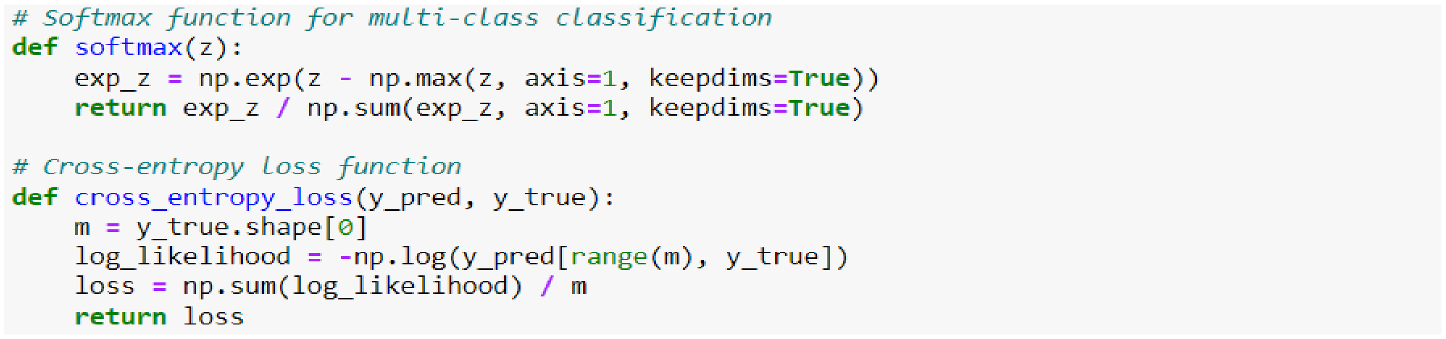
```
# Initialize weights and bias
input_dim = 32 * 32 * 3  # CIFAR-10 images are 32x32x3
num_classes = 10
weights = np.random.randn(input_dim, num_classes) * 0.0001
bias = np.zeros((1, num_classes))
epoch_accuracies = []
```

The second step is to perform linear forward which is computing the dot product of the input features and weights, followed by adding the weights. We then do backward pass for computing the gradient of weights and bias using the SoftMax cross entropy loss, true labels, and input features of the data.



```
def linear_forward(X, weights, bias):
    # Linear forward computation
    return np.dot(X, weights) + bias

def linear_backward(X, y_true, y_pred, weights):
    # Backward computation to get gradients
    m = y_true.shape[0]
    grad_softmax = y_pred
    grad_softmax[range(m), y_true] -= 1
    grad_softmax /= m
    grad_weights = np.dot(X.T, grad_softmax)
    grad_bias = np.sum(grad_softmax, axis=0, keepdims=True)
    return grad_weights, grad_bias
```

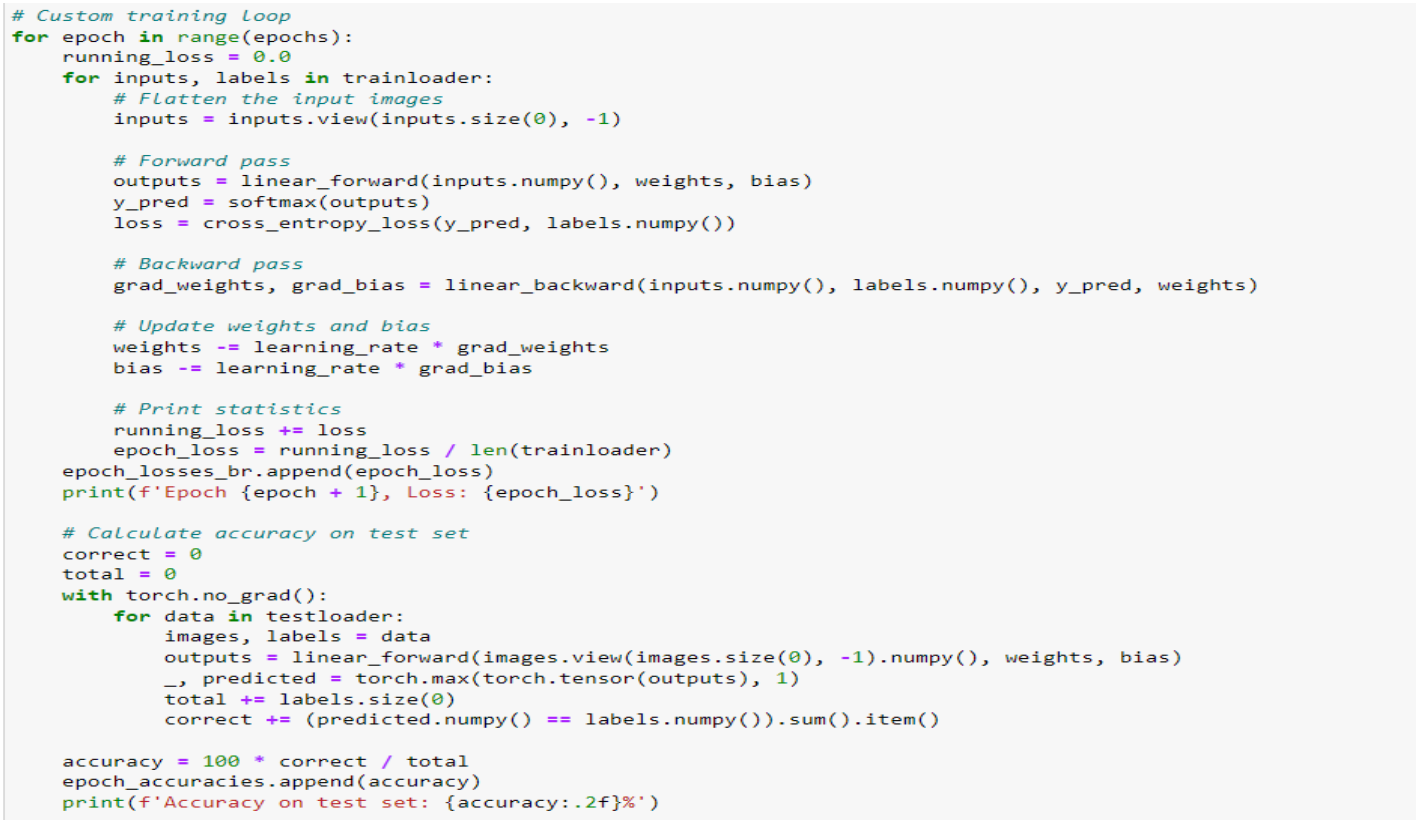
The third step involves SoftMax function and cross-entropy loss which calculates the losses and the SoftMax function calculates the probabilities for multiclass classification and ensures that it adds up to 1. The cross-entropy function calculates the loss between the predicted probabilities and true labels.



```
# Softmax function for multi-class classification
def softmax(z):
    exp_z = np.exp(z - np.max(z, axis=1, keepdims=True))
    return exp_z / np.sum(exp_z, axis=1, keepdims=True)

# Cross-entropy loss function
def cross_entropy_loss(y_pred, y_true):
    m = y_true.shape[0]
    log_likelihood = -np.log(y_pred[range(m), y_true])
    loss = np.sum(log_likelihood) / m
    return loss
```

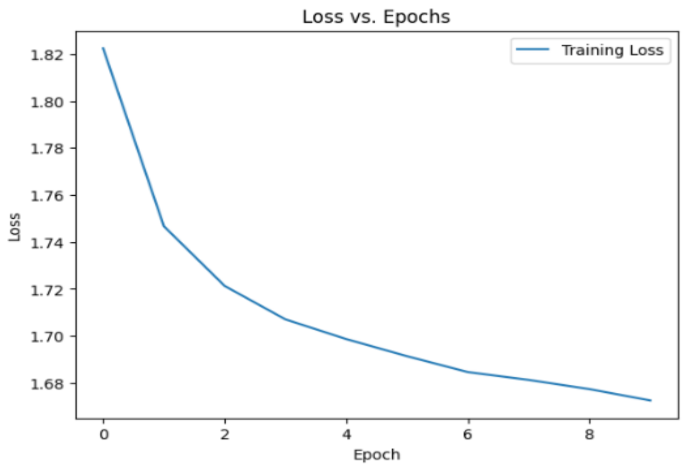
In the fourth step we define training parameters and custom training loop where the code computes predictions, losses, and gradients by making forward and backward passes within each epoch and training batch iteration. We then use gradient descent optimization to update the weights and bias parameters based on the gradients.

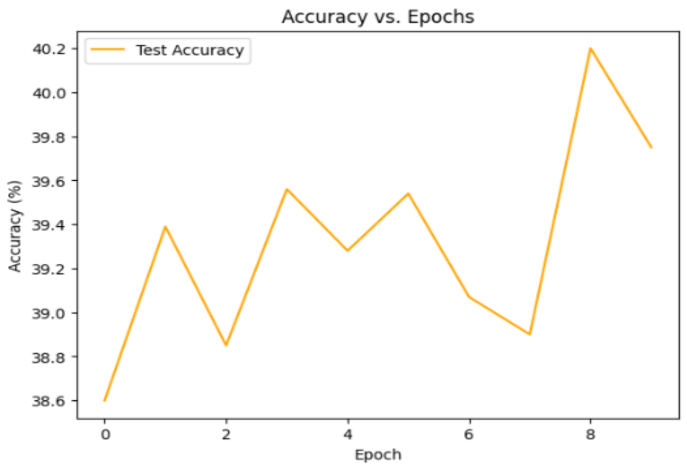
We then calculate and print the average loss for each epoch of the training process so that the model's progress can be monitored. The accuracies are then stored in Epoch accuracies to plot a graph, this iterative process allows the model to learn from the training data and improve its performance over subsequent epochs, which results in improving the classification accuracy on previously unseen test data.



```python
# Custom training loop
for epoch in range(epochs):
    running_loss = 0.0
    for inputs, labels in trainloader:
        # Flatten the input images
        inputs = inputs.view(inputs.size(0), -1)

        # Forward pass
        outputs = linear_forward(inputs.numpy(), weights, bias)
        y_pred = softmax(outputs)
        loss = cross_entropy_loss(y_pred, labels.numpy())

        # Backward pass
        grad_weights, grad_bias = linear_backward(inputs.numpy(), labels.numpy(), y_pred, weights)

        # Update weights and bias
        weights -= learning_rate * grad_weights
        bias -= learning_rate * grad_bias

        # Print statistics
        running_loss += loss
        epoch_loss = running_loss / len(trainloader)
    epoch_losses_br.append(epoch_loss)
    print(f'Epoch {epoch + 1}, Loss: {epoch_loss}')

    # Calculate accuracy on test set
    correct = 0
    total = 0
    with torch.no_grad():
        for data in testloader:
            images, labels = data
            outputs = linear_forward(images.view(images.size(0), -1).numpy(), weights, bias)
            _, predicted = torch.max(torch.tensor(outputs), 1)
            total += labels.size(0)
            correct += (predicted.numpy() == labels.numpy()).sum().item()

    accuracy = 100 * correct / total
    epoch_accuracies.append(accuracy)
    print(f'Accuracy on test set: {accuracy:.2f}%')
```

The last step was to perform plotting of the epochs and losses and accuracies obtained at each pass.





<u>Final results</u>: -

<u>In-built PY-torch functions</u>:

```
Epoch 1, Loss: 1.8266595772116594
Accuracy on test set: 37.83%
Epoch 2, Loss: 1.7476202757482107
Accuracy on test set: 39.70%
Epoch 3, Loss: 1.7235996894781511
Accuracy on test set: 38.98%
Epoch 4, Loss: 1.7083833230586671
Accuracy on test set: 38.75%
Epoch 5, Loss: 1.6991303633667305
Accuracy on test set: 40.29%
Epoch 6, Loss: 1.692570198901708
Accuracy on test set: 40.08%
Epoch 7, Loss: 1.685872100670217
Accuracy on test set: 39.45%
Epoch 8, Loss: 1.6804230991114582
Accuracy on test set: 39.18%
Epoch 9, Loss: 1.6762715080039134
Accuracy on test set: 39.89%
Epoch 10, Loss: 1.6722326811810602
Accuracy on test set: 39.46%
```

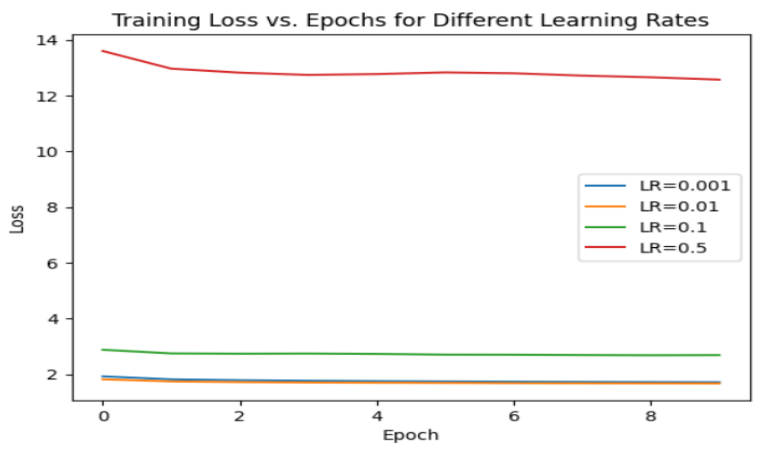
Custom Linear classifier with Forward and Backward computation:

Accuracy (without hyper parameter tuning): Increased accuracy to 40.20% from 36.68%

```
Epoch 1, Loss: 1.8224517470561155
Accuracy on test set: 38.60%
Epoch 2, Loss: 1.7467156130259338
Accuracy on test set: 39.39%
Epoch 3, Loss: 1.7212980540675968
Accuracy on test set: 38.85%
Epoch 4, Loss: 1.7070700090375008
Accuracy on test set: 39.56%
Epoch 5, Loss: 1.6986894754387507
Accuracy on test set: 39.28%
Epoch 6, Loss: 1.6914051335751592
Accuracy on test set: 39.54%
Epoch 7, Loss: 1.6846458117825398
Accuracy on test set: 39.07%
Epoch 8, Loss: 1.6813191540556045
Accuracy on test set: 38.90%
Epoch 9, Loss: 1.6773998729570023
Accuracy on test set: 40.20%
Epoch 10, Loss: 1.6725763857359996
Accuracy on test set: 39.75%
```

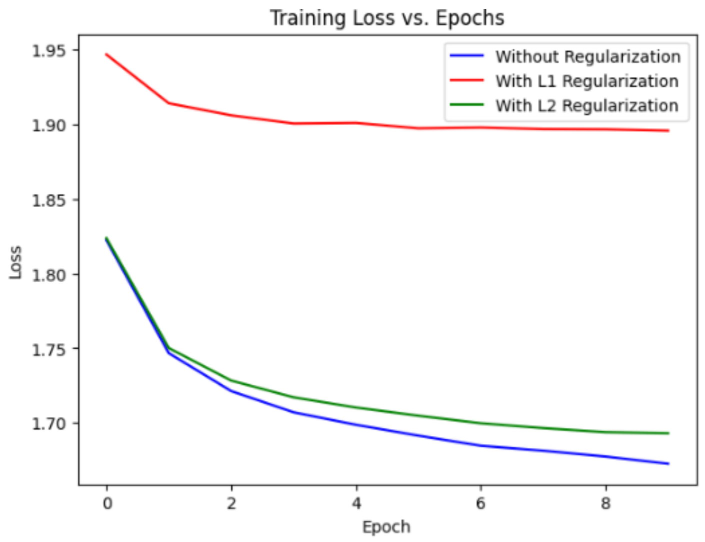
After Hyper parameter tuning:

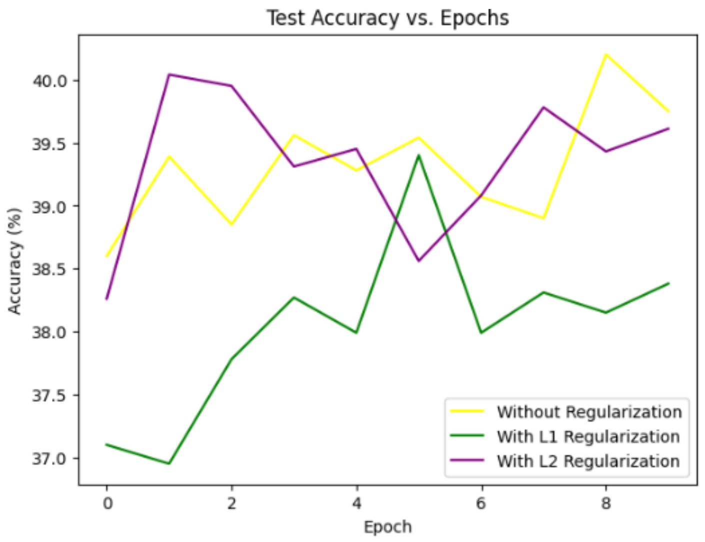
Best Learning rate after hyperparameter tuning: 0.01 as this yielded the lowest losses.



Regularization:

Comparison of Training losses Vs Epochs and Test Accuracy and Epochs for L1, L2 regularization and without regularization:





From the above fig we can conclude that L2 regularization yields better loss Vs epoch results and we also observe that the accuracy is more than results without the regularization and inbuilt packages also yield a similar result when we compare it to the custom linear classifier.

Methods of Improvements: -

Hyperparameter Tuning: (different learning rates)

In Hyperparameter tuning we have opted for using different learning rates and then we are observing the convergence change at each learning rate, we could identify that the learning rate of 0.001 and 0.01 are quite similar and thus we can conclude that this effects the accuracy of the prediction when compared to no hyper parameter tuning.

Different Regularization methods and Regularization strengths:

I have used both L1 and L2 regularization methods for optimizing the machine learning model and in both cases, it helps with not over fitting the machine learning model and regularization strength is useful in finetuning the degree of regularization in the model by influencing the sparsity of the model in L1 and magnitude of its weights in L2 regularizations.

In L1, L2 regularizations the penalty term is proportional to the sum of absolute values and sum of squared values respectively and L1 encourages the sparsity of weighted matrix by replacing some of the weights with zero but L2 encourages to reduce the weights of the existing weight matrix and effectively controls the weights from not getting too large. Which in turn effects the regularization strength.

Conclusions: -

To summarize, the custom linear classifier model, combined with hyperparameter tuning and regularization, provides a versatile approach to developing robust machine learning models. By adjusting learning rates and regularization parameters, we can improve model performance, prevent overfitting, and improve generalization to new data. Regularization techniques such as L1 and L2

----
regularization effectively constrain the model, lowering overfitting and increasing generalization. Hyperparameter tuning enables us to identify the best fitting feature to tune the model to make it more accurate and robust. Overall, this framework enables practitioners to create dependable machine learning systems capable of achieving better performance and robustness.

Reference: -

I have used ChatGPT for reference where relevant according to policy.