Background and Method Introduction: -

2-Layer Neural Network: - In the field of machine learning we seen a substantial transformation thanks to neural networks, especially in image classification applications. Three layers make up a 2-layer neural network, sometimes referred to as a shallow neural network the layers are namely input, output, and one hidden layer. Accurate predictions are made possible by this architecture's ability to effectively extract meaningful representations from the input data and helps predict the image data.

Regarding image classification, every pixel in an image is considered a distinct feature. The network predicts the class label of an image by learning to extract pertinent features through its hidden layers. Because it adds nonlinearity to the model, the activation function also referred to as the Rectified Linear Unit, or ReLU—is essential to this procedure, we are using CIFAR-10 dataset as it has a very nice sample size of 60000 images, and it is very useful to help train a neural network to achieve the best accuracy.

Application of 2-layer Neural network in image classification: In image classification tasks, two-layer neural networks are essential because they are very good at predicting and extracting features. Pixels are arranged in grids to represent images, and each pixel acts as an input feature for the network. The hidden layer of the network gains the ability to extract pertinent features from the input images through forward propagation. By introducing non-linearity, the activation function—typically ReLU—allows the network to capture intricate relationships between features and class labels. By enabling parameter updates based on prediction errors, backpropagation improves the network's accuracy in image classification. Neurons in the output layer are paired with class labels, enabling the network to generate classification probabilities for every picture.

Dataset and Tasks Description: -

CIFAR-10 Dataset: A famous dataset for picture sorting is CIFAR-10. It has ten groups. Each group have six thousand pictures. This dataset has sixty thousand colour pictures that are 32x32 size. We show things like cars, trucks, boats, frogs, dogs, cats, deer, and planes. The dataset is in two parts. One part is for training and other is for testing. People use the training part to teach the model. After that, we use the testing part to check the model. There are 10000 images in test set and 50000 in training set.

Classification Tasks:

In the CIFAR-10 dataset, there is a job to sort 32x32 colour pictures into ten different groups. Every picture shows different things like planes, cars, birds, cats, deer, dogs, frogs, horses, boats, and lorries. The aim is making a neural network that can put these pictures in the right groups by looking at what we show.

Algorithms Used: -

1.In-built PY-torch packages and libraries: - In this we have used the Inbuilt PY-torch packages and ran the 2-layer neural network to predict losses and accuracy at each epoch.

Custom linear classifier with Forward and Backward computation

1) **Data preprocessing: -**

In the first step we load the CIFAR-10 dataset and then perform

Normalization: Here we divide each pixel value by 255. This makes values from 0 to 1. It helps so the features are similar in size. It can make training better. Then we do

One-Hot Encoding: We change the class labels from 0 to 9. We become one-hot encoded vectors. This change is important. It is for tasks with many classes. Each class gets its own binary vector.

Step 1: Load and preprocess CIFAR-10 dataset

```
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

# Normalize the data
X_train = X_train.astype('float32') / 255
X_test = X_test.astype('float32') / 255

# Flatten the images
X_train_flat = X_train.reshape(X_train.shape[0], -1)
X_test_flat = X_test.reshape(X_test.shape[0], -1)

# Convert labels to one-hot encoding
num_classes = 10
y_train_onehot = np.eye(num_classes)[y_train.flatten()]
y_test_onehot = np.eye(num_classes)[y_test.flatten()]

# Normalize the data
mean = np.mean(X_train_flat, axis=0)
std = np.std(X_train_flat, axis=0)
X_train_normalized = (X_train_flat - mean) / (std + 1e-8)
X_test_normalized = (X_test_flat - mean) / (std + 1e-8)
```

2) **Forward Pass: -**

The forward pass computes the scores for each class using the current weights and biases. It involves matrix multiplication between the input data and the weight matrices, followed by the addition of bias terms.

ReLU Activation: The ReLU activation function is applied after the first linear transformation (Z1) to introduce non-linearity into the model. ReLU replaces negative values with zero, effectively allowing the network to learn complex mappings from input to hidden layer activations. Here the class is represented as a binary vector.

```
def forward_pass(X, W1, b1, W2, b2):
    Z1 = np.dot(X, W1) + b1
    A1 = np.maximum(0, Z1)  # ReLU activation
    scores = np.dot(A1, W2) + b2
    return scores, A1
```

3) **Backward Pass: -**

The backward pass finds out the gradients of the loss function with respects to the weights and biases. We use these gradients to update the network parameters by gradient descent.

Chain Rule: We send gradients back through the network with the chain rule from calculus, this make the computation of the gradient for each parameter efficient.

ReLU Backpropagation: We figure out the gradient of the ReLU activation function in backpropagation. This ensures the gradient only moves through neurons that activate.

```
def backward_pass(X, scores, A1, y_onehot, W1, b1, W2, b2, reg_strength):
    num_examples = X.shape[0]
    dscores = (scores - y_onehot) / num_examples
    dW2 = np.dot(A1.T, dscores)
    db2 = np.sum(dscores, axis=0)
    dA1 = np.dot(dscores, W2.T)
    dZ1 = dA1 * (A1 > 0)  # Backprop ReLU
    dW1 = np.dot(X.T, dZ1)
    db1 = np.sum(dZ1, axis=0)

    # Add regularization gradient
    dW2 += reg_strength * W2
    dW1 += reg_strength * W1

    return dW1, db1, dW2, db2
```

```
def predict(X, W1, b1, W2, b2):
    scores, _ = forward_pass(X, W1, b1, W2, b2)
    return np.argmax(scores, axis=1)
```

4) Training:-

Mini-Batch Gradient Descent: we train the network with mini-batch gradient descent. we split the dataset into small batches. This way we need less memory and makes the training faster than big batch gradient descent.

Loss Computation: we use the cross-entropy loss to compute loss. It compares the difference between the predicted and actual class probabilities.

Regularization: we use L2 regularization on the weights. It helps to stop overfitting by adding a penalty for big weight values. We add this penalty to the loss function when we are training.

```
# Mini-batch gradient descent
total_loss = 0

for i in range(num_batches):
    start = i * batch_size
    end = start + batch_size
    X_batch = X_shuffled[start:end]
    y_batch = y_shuffled[start:end]

    # Forward pass
    scores, A1 = forward_pass(X_batch, W1, b1, W2, b2)
```

```
# Computing loss using cross entropy loss(loss= data loss + regularization loss)
epsilon = 1e-8
scores = np.maximum(scores, epsilon)
correct_logprobs = -np.log(scores[range(X_batch.shape[0]), y_batch.argmax(axis=1)])
data_loss = np.sum(correct_logprobs)
reg_loss = 0.5 * reg_strength * (np.sum(W1 * W1) + np.sum(W2 * W2))
loss = data_loss + reg_loss
total_loss += loss
```

Results: -

Base case (Using In-built PY-torch functions): Target percentage = 51%

```
Files already downloaded and verified
Files already downloaded and verified
Epoch 1, Loss: 1.880
Epoch 2, Loss: 1.656
Epoch 3, Loss: 1.572
Epoch 4, Loss: 1.514
Epoch 5, Loss: 1.469
Epoch 6, Loss: 1.428
Epoch 7, Loss: 1.393
Epoch 8, Loss: 1.358
Epoch 9, Loss: 1.327
Epoch 10, Loss: 1.298
Finished Training
Accuracy of the network on the 10000 test images: 51 %
```

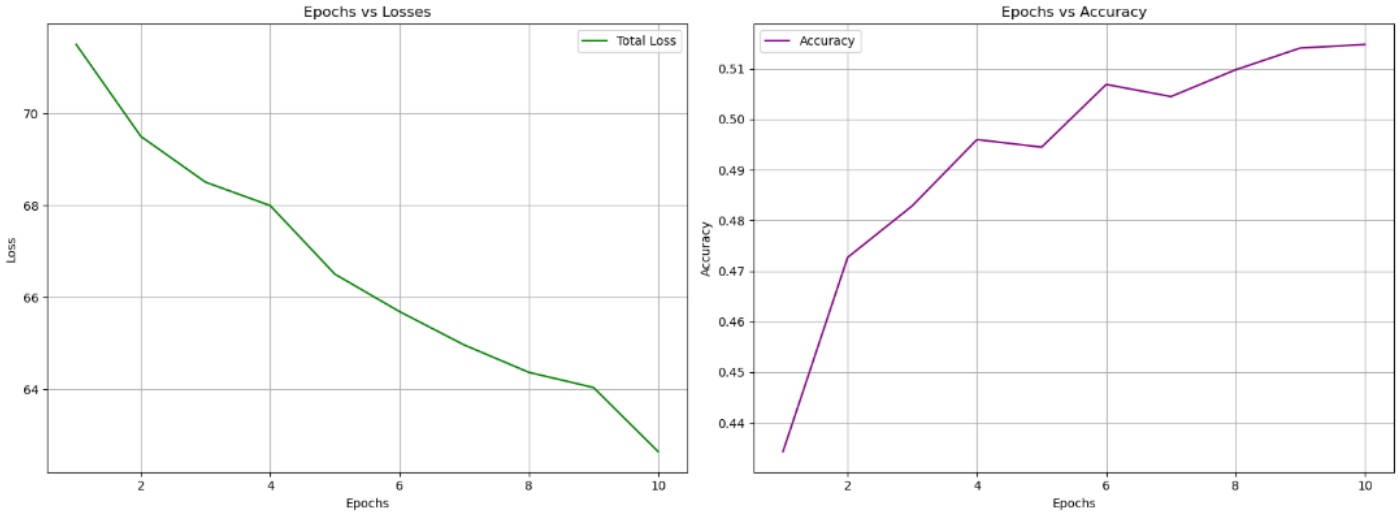
```
print(f'\n losses={running_loss}\n')
```

```
 losses=1014.7645643353462
```

Target percentage= 51%

<u>Custom Linear classifier with Forward and Backward computation</u>: Reached accuracy 51.48%

Accuracy after doing Hyper parameter tuning, regularization, and mini batch gradient descent.



Best Accuracy obtained = 51.48 % which is close to target percentage.

<u>Methods of Improvements:</u> -

1) Hyperparameter Tuning: (using different number of nodes / neurons in the hidden layers, iterating learning rates, regularization strengths to find the best set for maximum accuracy and min loss)

We have considered and compared various hyper parameters like Learning rate, Regularization strength and Number of neurons in the hidden layer. Below are the details why each of these are significant in the 2-layer neural networks.

Learning Rate: The rate at which optimization updates the weights. While a lower learning rate may result in a slower convergence, a greater learning rate may lead to a faster convergence but run the risk of overshooting the ideal solution.

Regularization Strength: The variable that regulates how regularization affects the model. Because regularization penalizes large weights, it helps prevent overfitting. Simpler solutions are encouraged by a bigger penalty on complicated models imposed by a higher regularization strength.

Number of Neurons in the Hidden Layer: The size of the hidden layer. The model can learn more intricate representations with a bigger hidden layer size, but there is a greater chance of overfitting, particularly when there is a lack of training data.

We used Grid Search to identify the best hyperparameters such as learning rate, regularization strength, and hidden layer size where each parameter is used systematically for tuning the model using grid search. This approach exhaustively searches through a predefined set of hyperparameters to identify the combination that yields the best performance.

```
best_accuracy = 0
best_parameters = {}

learning_rates = [1e-2, 1e-3] #tried various different values [1e-4, 1e-5,....] got the best accuracy value at 1e-2
reg_strengths = [1e-2, 1e-3] #tried various different values [1e-4, 1e-5,....] got the best accuracy value at 1e-3
hidden_sizes = [10, 50, 100] #tried various different values [10,20,....50,60,....100,200,....] got the best accuracy value at 1

for lr in learning_rates:
    for reg in reg_strengths:
        for hidden_size in hidden_sizes:
            print(f'\nTraining with learning rate={lr}, regularization strength={reg}, hidden size={hidden_size}\n')
            accuracies_per_epoch, losses_per_epoch = train(X_train_normalized, y_train_onehot, num_epochs=10, batch_size=32, lear
            print(f'Final Accuracy: {accuracies_per_epoch[-1]}')
            print(f'Final Loss: {losses_per_epoch[-1]}')
            print('------------------------------------------------------------')
            # Update best parameters if necessary
            if accuracies_per_epoch[-1] > best_accuracy:
                best_accuracy = accuracies_per_epoch[-1]
                best_parameters = {'learning_rate': lr, 'reg_strength': reg, 'hidden_size': hidden_size}
```

```
print(f'\nBest hyperparameters: {best_parameters}')
best_learning_rate = best_parameters['learning_rate']
best_reg_strength = best_parameters['reg_strength']
best_hidden_size = best_parameters['hidden_size']

# Train the network with the best hyperparameters
best_accuracies_per_epoch, best_losses_per_epoch = train(X_train_normalized, y_train_onehot, num_epochs=10, batch_size=32,
                                                         learning_rate=best_learning_rate,
                                                         reg_strength=best_reg_strength,
                                                         hidden_size=best_hidden_size)
```

```
Best hyperparameters: {'learning_rate': 0.01, 'reg_strength': 0.001, 'hidden_size': 100}
```

2) Normalization:

Data normalization is a preprocessing step that standardizes the scale of input features, normalization helps gradient descent algorithms converge faster and prevents certain features from dominating others. It also makes the optimization process more stable by ensuring that the gradients are within a similar range across all features. Without normalization, training a neural network can become challenging, especially when dealing with features with vastly different scales.

Batch Normalization: We can normalize not just data but also the outputs of each layer. This method helps training go faster and makes the model more stable. This is useful to simplify the model while training and is essential when we are working with a massive dataset.

3) Regularization:

Here we had used the L2 regularization to help model avoid overfitting and improve the accuracy and reduce the regularization losses.

Dropout: In dropout, we randomly ignore some neurons while we are training. This stops the neurons from depending too much on each other and helps stop overfitting. The model then performs better on new data hence we used this technique.

Early Stopping: We used early stopping to stop overfitting by watching the validation loss. If the validation loss does not get better, we stopped the training. This keeps the model from learning the random noise in the data.

Conclusions: -

We Observe that with careful regularization, hyperparameter tuning, and data normalization, the model attains competitive accuracy on test data that hasn't been seen before. The network can identify non-linear relationships in the data and recognize complex patterns thanks to the hidden layer's use of ReLU activation.

Optimal hyperparameters are found by methodically experimenting with different configurations, which improves the model's generalization to new examples. Regularization techniques are employed to reduce overfitting and guarantee consistent performance in a variety of image categories. The model's performance demonstrates how well deep learning methods work for tasks involving image classification. All things considered, the 2-layer neural network is an effective tool for deriving relevant features and producing precise predictions from intricate picture data.

References: -

https://ljvmiranda921.github.io/notebook/2017/02/17/artificial-neural-networks/

Build your own Neural Network for CIFAR-10 using PyTorch | by Shreekanya K | Becoming Human: Artificial Intelligence Magazine

Neural network for CIFAR-10 — NeoML documentation

How to Develop a CNN From Scratch for CIFAR-10 Photo Classification - MachineLearningMastery.com

https://betterprogramming.pub/how-to-build-2-layer-neural-network-from-scratch-in-python-4dd44a13ebba

I have also used AI tool for reference where relevant according to policy.