EXPERIMENT 4

OBJECTIVE : WAP to evaluate the performance of implemented three-layer neural network with variations in activation functions, size of hidden layer, learning rate, batch size and number of epochs.

DESCRIPTION OF MODEL(Configuration : Hidden layer , Learning rate)

The model is a fully connected feed-forward neural network designed to classify handwritten digits from the MNIST dataset.

Model follows a structured approach using forward propagation, loss calculation, backpropagation, and optimization to learn the correct classification.

1. Network Architecture 1

   - Input layer : Each MNIST image is 28 × 28 pixels (784 pixels)--> 784 input size
   - Hidden Layer 1 : 256 neurons , Activation Function: ReLU
   - Output Layer: 10 neurons (For each digit from 0 to 9) , No activation because it will be handled by softmax in the loss function.

   Model parameters(hyperparameters)

   - Number of epochs : 50
   - Learning rate : 0.01
   - Batch size : 10
   - Loss function : Softmax Cross-Entropy/Categorical cross-entropy
   - Optimiser : Adam

2. Network Architecture 2 (For 15 configurations)

   - Input layer : Each MNIST image is 28 × 28 pixels (784 pixels)--> 784 input size
   - Hidden Layers (2) : Neuron Configurations- [(160, 100), (100, 100), (100, 160), (60, 60), (100, 60)] , Activation Function: ReLU
   - Output Layer: 10 neurons (For each digit from 0 to 9) , No activation because it will be handled by softmax in the loss function.

   Model parameters(hyperparameters)

   - Number of epochs : 50
   - Learning rates : [0.01 , 0.1 , 1.0]

- Batch size : 10
- Loss function : Softmax Cross-Entropy/Categorical cross-entropy
- Optimiser : Adam

PYTHON IMPLEMENTATION

```
import tensorflow as tf

import tensorflow_datasets as tfds

import numpy as np

import matplotlib.pyplot as plt

import time

from sklearn.metrics import confusion_matrix

import seaborn as sns


# Load and preprocess the MNIST dataset

def preprocess(image, label):

    image = tf.cast(image, tf.float32) / 255.0  # Normalize to [0,1]

    image = tf.reshape(image, [-1])  # Flatten to 784

    label = tf.one_hot(label, depth=10)  # Convert to one-hot encoding

    return image, label


# Load dataset and apply preprocessing

mnist_dataset = tfds.load("mnist", split=["train", "test"], as_supervised=True)

train_data = mnist_dataset[0].map(preprocess).shuffle(10000).batch(10)

test_data = mnist_dataset[1].map(preprocess).batch(10)
```

```
# Define neural network parameters

input_size = 784

hidden_layer1_size = 160

hidden_layer2_size = 100

output_size = 10

learning_rate = 0.01

epochs = 50


# Initialize weights and biases

W1 = tf.Variable(tf.random.normal([input_size, hidden_layer1_size]))

b1 = tf.Variable(tf.zeros([hidden_layer1_size]))

W2 = tf.Variable(tf.random.normal([hidden_layer1_size, hidden_layer2_size]))

b2 = tf.Variable(tf.zeros([hidden_layer2_size]))

W_out = tf.Variable(tf.random.normal([hidden_layer2_size, output_size]))

b_out = tf.Variable(tf.zeros([output_size]))


# Forward pass function

def forward_pass(X):

    hidden_layer1 = tf.nn.relu(tf.matmul(X, W1) + b1)

    hidden_layer2 = tf.nn.relu(tf.matmul(hidden_layer1, W2) + b2)

    output_layer = tf.matmul(hidden_layer2, W_out) + b_out  # No activation (logits)
```

```python
    return output_layer


# Define loss function

def loss_fn(logits, labels):

    return tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits, 
labels=labels))


# Define optimizer

optimizer = tf.optimizers.Adam(learning_rate)


# Training step function

def train_step(X, Y):

    with tf.GradientTape() as tape:

        logits = forward_pass(X)

        loss = loss_fn(logits, Y)

    gradients = tape.gradient(loss, [W1, b1, W2, b2, W_out, b_out])

    optimizer.apply_gradients(zip(gradients, [W1, b1, W2, b2, W_out, b_out]))

    return loss


# Compute accuracy

def compute_accuracy(dataset):

    total_correct = 0

    total_samples = 0
```

```
    for X, Y in dataset:
        logits = forward_pass(X)
        correct_pred = tf.equal(tf.argmax(logits, 1), tf.argmax(Y, 1))
        total_correct += tf.reduce_sum(tf.cast(correct_pred, tf.float32))
        total_samples += X.shape[0]
    return total_correct / total_samples

# Track loss and accuracy
loss_history = []
accuracy_history = []
start_time = time.time()

# Training loop
for epoch in range(epochs):
    avg_loss = 0
    total_batches = 0

    for batch_x, batch_y in train_data:
        loss = train_step(batch_x, batch_y)
        avg_loss += loss
        total_batches += 1
```

```
    avg_loss /= total_batches
    train_acc = compute_accuracy(train_data)
    loss_history.append(avg_loss.numpy())
    accuracy_history.append(train_acc.numpy())
    print(f"Epoch {epoch+1}, Loss: {avg_loss:.4f}, Training Accuracy: {train_acc:.4f}")
end_time = time.time()
execution_time = end_time- start_time

# Test the model
test_acc = compute_accuracy(test_data)
print(f"Test Accuracy: {test_acc:.4f}")

# Plot Loss Curve
plt.figure(figsize=(10, 5))
plt.plot(loss_history, label='Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Loss Curve')
plt.legend()
plt.show()

# Plot Accuracy Curve
```

```
plt.figure(figsize=(10, 5))

plt.plot(accuracy_history, label='Accuracy', color='orange')

plt.xlabel('Epochs')

plt.ylabel('Accuracy')

plt.title('Accuracy Curve')

plt.legend()

plt.show()


# Compute confusion matrix

y_true, y_pred = [], []

for X, Y in test_data:

    logits = forward_pass(X)

    predictions = tf.argmax(logits, axis=1).numpy()

    labels = tf.argmax(Y, axis=1).numpy()

    y_pred.extend(predictions)

    y_true.extend(labels)

conf_matrix = confusion_matrix(y_true, y_pred)

plt.figure(figsize=(8, 6))

sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=range(10),
yticklabels=range(10))

plt.xlabel("Predicted")

plt.ylabel("Actual")

plt.title("Confusion Matrix")
```

```
plt.show()

print(f"Execution Time: {execution_time:.2f} seconds")
```

DESCRIPTION OF CODE

- tensorflow – Used for building and training the neural network.
- tensorflow_datasets – Used to load the MNIST dataset.
- numpy – Used for numerical operations and working on arrays.
- matplotlib.pyplot – Used to plot loss and accuracy curves.
- time – Used to measure the execution time over training loop.
- sklearn.metrics.confusion_matrix – Used to compute the confusion matrix for performance evaluation.
- seaborn – Used to visualise the confusion matrix as heatmap.
- First we will load the mnist dataset.

preprocess() :

- Dataset is loaded and preprocessing is done . Pixel values are normalised between [0,1] .
- Dataset is converted to a 1D array (784).
- One hot encoding is done to convert categorical labels to vector having binary values , e.g. class label 2 –[0 ,0, 1, 0, 0, 0, 0, 0, 0, 0].
- mnist dataset is split into train and test dataset each containing tuple (image , label).
- map(preprocess) : Applies preprocessing to every image in the dataset.
- shuffle(10000) : Randomly shuffles 10,000 images to prevent model learn by order of images.
- batch(10) : Divides dataset into mini-batches of size 10 images for training. For 10 forward passes , backward propagation will be performed once.
- Neural network parameters are defined.

Weights and bias initialisation :

- Initialised using tf.Variable . Weights (W1, W2, W_out) : Randomly initialized
- Biases (b1, b2, b_out) : Initialized as zero

forward_pass() :

- Layer 1 output : A1 = sigmoid(XW1 + b1)
- Layer 2 output : A2 = sigmoid(A1W2 + b2)
- Output Layer (Logits): Output(Z) = (A2W_out + b_out)
- Returns logit values.

loss_fn() :

- Softmax Cross-Entropy Loss: Measures how different the predictions are from true labels.
- Loss = $-\sum (Y_i) \log(\text{softmax}(Z_i))$
- tf.reduce_mean(): Computes average loss within a batch.

- Adam Optimizer : Optimizer adjusts the weights and biases to minimize the loss function during training.

train_step() :

- tf.GradientTape(): Help in resource management . Stores sequence of operations (like matrix multiplication , logits , loss values ,which parameter contributed more in loss) inside the block and calculate gradients automatically.
- tape.gradients() calculate gradient of loss with respect to each weight and bias . Memory is freed up as soon as it is called.
- optimizer.apply_gradients(): Updates weights using the calculated gradients , zip pairs gradient with respective parameter.

compute_accuracy() :

- Helps calculate training and test accuracy (correct predictions/total predictions).
- Prediction - (argmax): Returns the index of the highest probability class.
- equal(): Compares predictions with true labels.

Training loop:

  - Loops over 50 epochs.
  - Calculates the loss and training accuracy . Prints loss and accuracy value for each epoch . These values are added in loss history and accuracy history lists.
  - Execution time is calculated over the training loop.

- Test accuracy is calculated.
- Loss curve , Accuracy curve and Confusion matrix are plotted.

PERFORMANCE EVALUATION

- Performance has been evaluated by plotting Loss curve , Training Accuracy curve and Confusion matrix.
- The training and test accuracy for **Network architecture 1** is Training Accuracy: 33.64% , Test Accuracy: **33.43%** as it just contains a single hidden layer and batch-size is also small(10) , it has executed in **least time(10308.36 seconds**).
- The training and test accuracy for Network architecture 2 having having 15 configurations vary according to the model parameters.

- The training and test accuracy achieved is less for **learning rate 0.01** but maximum train and test accuracy reached are **Training Accuracy**: **91.76%** , **Test Accuracy**: **91.18%** for configuration of **hidden layer (60,60)** .

- For **learning rate values 0.1 and 1.0** the training and test accuracy were nearly same and have just reached **9 – 10**% .

- **Maximum execution time** required (15 combinations) was for configuration having learning rate 1.0 (**14831.84 seconds**) for configuration of **hidden layer (60,60)**.

- **Minimum execution time** required (15 combinations) was for configuration having learning rate 0.01 (**12698.1 seconds**) for configuration of **hidden layer(60,60).**
- The **execution time** required is higher for learning rate 1.0 than for learning rate values 0.01 and 0.1 .

- Execution time ranges from approximately 12000 to 15000 seconds.

My Comments(Limitations and Scope for Improvement)

- Maximum train and test accuracy reached are **Training Accuracy**: **91.76%** , **Test Accuracy**: **91.18%** for configuration of **hidden layer (60,60)** .
- The accuracies achieved are very less in maximum cases which is just about **9-10%.**
- The execution time is high.
- To improve accuracy and decrease execution time , a different activation function can be used , batch size can be increased , the learning rate value can be changed , the number of hidden layers and number of hidden layer neurons need to be changed .