**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.

Roll no. – 23UADS4155 % 3 = 0 ( Variance in Batch size and number of Epochs ).

**Model Description:** This model is a fully connected feedforward neural network (FNN). It is built using TensorFlow 1.x and trained using the Adam optimizer with categorical cross-entropy loss.

**Model Architecture:**

1. Input Layer (784 neurons)

Each MNIST image (28x28 pixels) is flattened into a 1D vector of 784 values.

The pixel values are normalized to the range [0,1].

1. Hidden Layer 1 (128 neurons, ReLU activation)

Weights: W1 → Shape: (784, 128)

Biases: b1 → Shape: (128, )

Activation: ReLU (Rectified Linear Unit)

Computes: ReLU(X \* W1 + b1)

1. Hidden Layer 2 (64 neurons, ReLU activation)

Weights: W2 → Shape: (128, 64)

Biases: b2 → Shape: (64, )

Activation: ReLU

Computes: ReLU(Layer1 \* W2 + b2)

1. Output Layer (10 neurons, Softmax activation)

Weights: W\_out → Shape: (64, 10)

Biases: b\_out → Shape: (10, )

Activation: Softmax

Computes: Softmax(Layer2 \* W\_out + b\_out)

Outputs a probability distribution over 10 classes (digits 0-9).

**Python Implementation:**import tensorflow.compat.v1 as tf

import tensorflow\_datasets as tfds

import numpy as np

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

import time

from sklearn.metrics import confusion\_matrix

from openpyxl import Workbook

# Disable TensorFlow v2

tf.disable\_v2\_behavior()

# ✅ Load MNIST Dataset

mnist = tfds.load('mnist', split=['train', 'test'], as\_supervised=True)

train\_data, test\_data = mnist

# ✅ Preprocessing Function

def preprocess(images, labels):

    images = tf.cast(images, tf.float32) / 255.0  # Normalize

    images = tf.reshape(images, [784])  # Flatten

    labels = tf.one\_hot(labels, depth=10)  # One-hot encode

    return images, labels

# ✅ Hyperparameters to Tune

batch\_size\_list = [1,10, 100]

epochs\_list = [10, 50,100]

# ✅ Training Function

def train\_and\_evaluate(batch\_size, epochs):

    print(f"\nTraining with batch\_size={batch\_size}, epochs={epochs}")

    # ✅ Prepare Dataset

    train\_dataset = train\_data.map(preprocess).batch(batch\_size)

    test\_dataset = test\_data.map(preprocess).batch(batch\_size)

    # ✅ Placeholders

    X = tf.placeholder(tf.float32, [None, 784])

    Y = tf.placeholder(tf.float32, [None, 10])

    # ✅ Initialize Weights & Biases

    weights = {

        'h1': tf.Variable(tf.random\_normal([784, 128])),

        'h2': tf.Variable(tf.random\_normal([128, 64])),

        'out': tf.Variable(tf.random\_normal([64, 10]))

    }

    biases = {

        'b1': tf.Variable(tf.random\_normal([128])),

        'b2': tf.Variable(tf.random\_normal([64])),

        'out': tf.Variable(tf.random\_normal([10]))

    }

    # ✅ Neural Network Model

    def neural\_network(x):

        layer1 = tf.nn.relu(tf.add(tf.matmul(x, weights['h1']), biases['b1']))

        layer2 = tf.nn.relu(tf.add(tf.matmul(layer1, weights['h2']), biases['b2']))

        return tf.add(tf.matmul(layer2, weights['out']), biases['out'])

    # ✅ Compute Loss & Optimizer

    logits = neural\_network(X)

    loss = tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits\_v2(logits=logits, labels=Y))

    optimizer = tf.train.AdamOptimizer(learning\_rate=0.01).minimize(loss)

    # ✅ Compute Accuracy

    predictions = tf.nn.softmax(logits)

    correct\_pred = tf.equal(tf.argmax(predictions, 1), tf.argmax(Y, 1))

    accuracy = tf.reduce\_mean(tf.cast(correct\_pred, tf.float32))

    # ✅ Train Model

    loss\_curve, acc\_curve, val\_acc\_curve = [], [], []

    start\_time = time.time()

    with tf.Session() as sess:

        sess.run(tf.global\_variables\_initializer())

        for epoch in range(epochs):

            avg\_loss = 0

            total\_batches = 0

            iterator = tf.compat.v1.data.make\_one\_shot\_iterator(train\_dataset)

            next\_batch = iterator.get\_next()

            while True:

                try:

                    batch\_x, batch\_y = sess.run(next\_batch)

                    \_, c = sess.run([optimizer, loss], feed\_dict={X: batch\_x, Y: batch\_y})

                    avg\_loss += c

                    total\_batches += 1

                except tf.errors.OutOfRangeError:

                    break

            avg\_loss /= total\_batches

            train\_acc = sess.run(accuracy, feed\_dict={X: batch\_x, Y: batch\_y})

            val\_acc = sess.run(accuracy, feed\_dict={X: batch\_x, Y: batch\_y})

            loss\_curve.append(avg\_loss)

            acc\_curve.append(train\_acc)

            val\_acc\_curve.append(val\_acc)

            print(f"Epoch {epoch+1}, Loss: {avg\_loss:.4f}, Train Acc: {train\_acc:.4f}, Val Acc: {val\_acc:.4f}")

        end\_time = time.time()

        execution\_time = end\_time - start\_time

        # ✅ Evaluate on Test Data

        test\_acc = []

        y\_true, y\_pred = [], []

        iterator = tf.compat.v1.data.make\_one\_shot\_iterator(test\_dataset)

        next\_batch = iterator.get\_next()

        while True:

            try:

                batch\_x, batch\_y = sess.run(next\_batch)

                acc, preds = sess.run([accuracy, predictions], feed\_dict={X: batch\_x, Y: batch\_y})

                test\_acc.append(acc)

                y\_pred.extend(np.argmax(preds, axis=1))

                y\_true.extend(np.argmax(batch\_y, axis=1))

            except tf.errors.OutOfRangeError:

                break

        final\_test\_acc = np.mean(test\_acc)

        print(f"Test Accuracy: {final\_test\_acc:.4f}")

        # ✅ Save Confusion Matrix as Image

        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(f"Confusion Matrix (Batch={batch\_size}, Epochs={epochs})")

        cm\_filename = f"confusion\_matrix\_batch{batch\_size}\_epochs{epochs}.png"

        plt.savefig(cm\_filename)

        plt.close()

        # ✅ Save Loss & Accuracy Curves

        plt.figure(figsize=(12, 4))

        # Loss Curve

        plt.subplot(1, 2, 1)

        plt.plot(loss\_curve, label='Train Loss')

        plt.xlabel("Epochs")

        plt.ylabel("Loss")

        plt.legend()

        plt.title("Loss Curve")

        # Accuracy Curve

        plt.subplot(1, 2, 2)

        plt.plot(acc\_curve, label='Train Accuracy')

        plt.plot(val\_acc\_curve, label='Val Accuracy')

        plt.xlabel("Epochs")

        plt.ylabel("Accuracy")

        plt.legend()

        plt.title("Accuracy Curve")

        curve\_filename = f"curves\_batch{batch\_size}\_epochs{epochs}.png"

        plt.savefig(curve\_filename)

        plt.close()

    return batch\_size, epochs, execution\_time, final\_test\_acc, cm\_filename, curve\_filename

# ✅ Run Experiments & Store Results

results = []

for batch\_size in batch\_size\_list:

    for epochs in epochs\_list:

        res = train\_and\_evaluate(batch\_size, epochs)

        results.append(res)

# ✅ Save Results to Excel

df = pd.DataFrame(results, columns=['Batch Size', 'Epochs', 'Execution Time (s)', 'Test Accuracy', 'Confusion Matrix Image', 'Loss/Accuracy Curves'])

df.to\_excel("training\_results.xlsx", index=False)

print("\n✅ All Results Saved in training\_results.xlsx ✅")

**Description of code:**

Experiments are run with different:

* Batch Sizes: [1, 10, 100]
* Epochs: [10, 50, 100]

**Dataset Loading:** The MNIST dataset is loaded using tensorflow\_datasets and split into training and test sets.

**Preprocessing**: Images are normalized to the range [0,1], flattened, and labels are one-hot encoded.

**Model Training:**

A TensorFlow v1 session is used for training.

Training data is processed in batches using tf.data.

The model updates weights using Adam optimization.

Training loss and accuracy are recorded.

**Evaluation:**

The model is tested on unseen test data.

A confusion matrix is generated.

Loss and accuracy curves are plotted.

**Results Storage:**

Results, including execution time and test accuracy, are stored in an Excel file.

Confusion matrix and accuracy/loss curves are saved as images. (Provided in exp4/results directory.)

**Comparative Analysis of Batch Size and Epochs:**

**Impact of Batch Size:**

Larger batch sizes (100) lead to higher accuracy (up to 96.94%) and significantly reduced execution time compared to smaller batch sizes.

Smaller batch sizes (1) result in very low accuracy (~11.34%) and extremely high execution time.

A batch size of 10 provides reasonable performance, but accuracy decreases with higher epochs.

**Impact of Number of Epochs:**

For batch size 100, increasing epochs from 10 to 50 improves accuracy, but going beyond 50 does not significantly improve performance.

For batch size 10, increasing epochs beyond 10 reduces accuracy, likely due to overfitting.

For batch size 1, accuracy remains very low even with increased epochs, making it inefficient.

**Time Complexity Considerations:**

Batch size 1 has the highest execution time, making it impractical.

Batch size 100 is the fastest and achieves the highest accuracy.

Training with a batch size of 10 and high epochs is inefficient as execution time increases without improving accuracy.

**My Comments:**

Implement a Convolutional Neural Network (CNN) for better feature extraction.