3-Layer NN using Tensorflow

1. **Objective:**

Write a Program to implement a three-layer neural network using Tensor flow library (only, no keras) to classify MNIST handwritten digits dataset. Demonstrate the implementation of feed-forward and back-propagation approaches.

1. **Description of the model:**

This program implements a three-layer fully connected neural network using TensorFlow (without Keras) to classify the MNIST handwritten digits dataset. The model follows a feed-forward neural network architecture with backpropagation for training.

* Input layer (784 neurons), the input consists of 28x28 grayscale images.
* Hidden layer 1 (128 neurons).
* Hidden layer 2 (64 neurons).
* Output layer ( 10 neurons), gives output of numbers between 0-9.

**Tensorflow -** TensorFlow is an open-source machine learning and deep learning library developed by Google. It is widely used for building neural networks and training AI models efficiently.

ReLU is used as an activation function.

1. **Python Implementation:**

import tensorflow as tf

import numpy as np

# Load MNIST dataset

(x\_train, y\_train), (x\_test, y\_test) = tf.keras.datasets.mnist.load\_data()

# Preprocess dataset

x\_train, x\_test = x\_train.reshape(-1, 28\*28).astype(np.float32) / 255.0, x\_test.reshape(-1, 28\*28).astype(np.float32) / 255.0

y\_train, y\_test = tf.one\_hot(y\_train, depth=10), tf.one\_hot(y\_test, depth=10)

# Define network parameters

input\_size = 784  # 28x28 pixels

hidden\_size1 = 128

hidden\_size2 = 64

output\_size = 10  # Digits 0-9

learning\_rate = 0.01

epochs = 20

batch\_size = 128

# Initialize weights and biases

weights = {

    "W1": tf.Variable(tf.random.normal([input\_size, hidden\_size1], stddev=0.1)),

    "W2": tf.Variable(tf.random.normal([hidden\_size1, hidden\_size2], stddev=0.1)),

    "W3": tf.Variable(tf.random.normal([hidden\_size2, output\_size], stddev=0.1)),

}

biases = {

    "b1": tf.Variable(tf.zeros([hidden\_size1])),

    "b2": tf.Variable(tf.zeros([hidden\_size2])),

    "b3": tf.Variable(tf.zeros([output\_size])),

}

# Define feed-forward function

def forward\_propagation(x):

    z1 = tf.matmul(x, weights["W1"]) + biases["b1"]

    a1 = tf.nn.relu(z1)

    z2 = tf.matmul(a1, weights["W2"]) + biases["b2"]

    a2 = tf.nn.relu(z2)

    z3 = tf.matmul(a2, weights["W3"]) + biases["b3"]

    output = tf.nn.softmax(z3)  # Softmax for classification

    return output

# Loss function (Cross-entropy)

def compute\_loss(y\_pred, y\_true):

    return tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits(logits=y\_pred, labels=y\_true))

# Training function with backpropagation

def train\_step(x\_batch, y\_batch):

    with tf.GradientTape() as tape:

        y\_pred = forward\_propagation(x\_batch)

        loss = compute\_loss(y\_pred, y\_batch)

    # Compute gradients

    gradients = tape.gradient(loss, list(weights.values()) + list(biases.values()))

    # Update weights and biases using Gradient Descent

    for i, key in enumerate(weights.keys()):

        weights[key].assign\_sub(learning\_rate \* gradients[i])

    for i, key in enumerate(biases.keys()):

        biases[key].assign\_sub(learning\_rate \* gradients[len(weights) + i])

    return loss

# Training loop

num\_batches = x\_train.shape[0] // batch\_size

for epoch in range(epochs):

    avg\_loss = 0

    for i in range(num\_batches):

        batch\_x = x\_train[i \* batch\_size:(i + 1) \* batch\_size]

        batch\_y = y\_train[i \* batch\_size:(i + 1) \* batch\_size]

        loss = train\_step(batch\_x, batch\_y)

        avg\_loss += loss / num\_batches

    print(f"Epoch {epoch+1}, Loss: {avg\_loss.numpy():.4f}")

# Evaluation

y\_pred\_test = forward\_propagation(x\_test)

correct\_predictions = tf.equal(tf.argmax(y\_pred\_test, axis=1), tf.argmax(y\_test, axis=1))

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

print(f"Test Accuracy: {accuracy.numpy() \* 100:.2f}%")

1. **Description of the code:**

* Importing the libraries (numpy and tensorflow)
* Importing the MNIST dataset which contains images of handwritten numbers.
* Data is reshaped (28x28 matrix in the 1D vector of size 784), normalizing pixel values (0-255 to 0-1), and encoding (one-hot encoding).
* Network parameters like the number of neurons in each layer, learning rate, epochs, and batch size are defined.
* Weights and biases are defined for each layer (W = list of each layer’s weights , B = list of each layer’s biases).
* Froward propagation function to find output of each layer:

Zi = Wi.X + Bi

* Loss function calculates the difference between predicted and actual values.
* Backpropagation is used for updating weights.
* Finally the model is trained and evaluated.

1. **Output:**

Epoch 1, Loss: 2.2877

Epoch 2, Loss: 2.2343

Epoch 3, Loss: 2.1234

Epoch 4, Loss: 1.9871

Epoch 5, Loss: 1.8813

Epoch 6, Loss: 1.8269

Epoch 7, Loss: 1.7605

Epoch 8, Loss: 1.6965

Epoch 9, Loss: 1.6628

Epoch 10, Loss: 1.6416

Epoch 11, Loss: 1.6268

Epoch 12, Loss: 1.6157

Epoch 13, Loss: 1.6071

Epoch 14, Loss: 1.6002

Epoch 15, Loss: 1.5945

Epoch 16, Loss: 1.5896

Epoch 17, Loss: 1.5855

Epoch 18, Loss: 1.5819

Epoch 19, Loss: 1.5788

Epoch 20, Loss: 1.5760

Test Accuracy: 90.98%

1. **Performance:**

* The accuracy of the model is 90%.
* The loss reduces over the 20 epochs.

1. **My comments:**

* The model achieved high classification accuracy.
* It uses a simple three-layer Neural Network with a ReLU activation function.
* It has fully connected layers, requires more parameters, making it less effective and, more memory-intensive.