

Experiment No 5

Aim: -

To implement the Backpropagation algorithm to train a Deep Neural Network (DNN) with at least two hidden layers using TensorFlow.

Apparatus: -

- Programming Language: **Python 3.x**
- Libraries/Frameworks: **TensorFlow 2.x, NumPy**
- Platform: **Google Colab / Jupyter Notebook**

Theory: -

What is a Deep Neural Network (DNN)

A Deep Neural Network is an artificial neural network with multiple hidden layers between the input and output. It learns complex patterns in data by adjusting weights and biases through training.

Backpropagation Algorithm:

Backpropagation is the main algorithm used to train neural networks. It works in two phases:

1. **Forward Propagation:**
 - Input data passes through layers using weights, biases, and activation functions.
 - Produces a prediction at the output layer.
2. **Backward Propagation:**
 - The prediction is compared with the true label using a **loss function** (Binary Cross-Entropy).
 - The error is propagated backward through the network.
 - **Gradients** are calculated using the **chain rule of calculus**.
 - Weights are updated using the **Stochastic Gradient Descent (SGD)** optimizer.

Activation Functions used:

- **ReLU (Rectified Linear Unit):** Introduces non-linearity in hidden layers.
- **Sigmoid:** Produces values between 0 and 1 for binary classification.

Loss Function (Binary Cross-Entropy):

$$L_{BCE} = -\frac{1}{n} \sum_{i=1}^n (Y_i \cdot \log \hat{Y}_i + (1 - Y_i) \cdot \log (1 - \hat{Y}_i))$$

This ensures the network minimizes the error between predicted output \hat{y} and true label y .

Procedure: -

1. Data Preparation:

- Generated dummy dataset with 1000 samples, each having 20 features.
- Labels assigned: 1 if the sum of features > 0 , else 0.

2. Model Design:

- Input layer with 20 features.
- Hidden Layer 1: 64 neurons, ReLU activation.
- Hidden Layer 2: 32 neurons, ReLU activation.
- Output Layer: 1 neuron, Sigmoid activation.

3. Initialization:

- Weights and biases initialized using **Glorot Uniform initializer**.

4. Training:

- Used **SGD optimizer** with learning rate 0.01.
- Trained for **20 epochs** with a batch size of 32.
- Implemented forward pass, loss computation, backpropagation, and weight update.

5. Visualization:

- Plotted training loss across epochs to verify learning.

Code: -

```
#Implement backpropagation algorithm to train DNN with atleast two
hidden layer
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

# Fix random seed for reproducibility
tf.random.set_seed(42)
np.random.seed(42)

# Generate dummy data
X_train = np.random.randn(1000, 20).astype(np.float32)
y_train = (np.sum(X_train, axis=1) > 0).astype(np.float32).reshape(-1,
1) # Simple binary target

# Network architecture parameters
input_dim = 20
hidden1_units = 64
hidden2_units = 32
output_units = 1
learning_rate = 0.01
epochs = 20
batch_size = 32

# Initialize weights and biases
```

```

initializer = tf.initializers.GlorotUniform()

W1 = tf.Variable(initializer([input_dim, hidden1_units]))
b1 = tf.Variable(tf.zeros([hidden1_units]))

W2 = tf.Variable(initializer([hidden1_units, hidden2_units]))
b2 = tf.Variable(tf.zeros([hidden2_units]))

W3 = tf.Variable(initializer([hidden2_units, output_units]))
b3 = tf.Variable(tf.zeros([output_units]))

# Activation functions
def relu(x):
    return tf.nn.relu(x)

def sigmoid(x):
    return tf.sigmoid(x)

# Forward pass function
def forward_pass(x):
    z1 = tf.matmul(x, W1) + b1
    a1 = relu(z1)

    z2 = tf.matmul(a1, W2) + b2
    a2 = relu(z2)

    z3 = tf.matmul(a2, W3) + b3
    output = sigmoid(z3)
    return output

# Binary cross-entropy loss
def loss_fn(y_true, y_pred):
    # Add small epsilon to avoid log(0)
    epsilon = 1e-7
    y_pred = tf.clip_by_value(y_pred, epsilon, 1 - epsilon)
    return -tf.reduce_mean(y_true * tf.math.log(y_pred) + (1 - y_true)
* tf.math.log(1 - y_pred))

# Optimizer parameters
optimizer = tf.keras.optimizers.SGD(learning_rate=learning_rate)

# Training loop
num_batches = X_train.shape[0] // batch_size
loss_history = []

for epoch in range(epochs):
    epoch_loss = 0
    for i in range(num_batches):

```

```

x_batch = X_train[i*batch_size:(i+1)*batch_size]
y_batch = y_train[i*batch_size:(i+1)*batch_size]

with tf.GradientTape() as tape:
    y_pred = forward_pass(x_batch)
    loss = loss_fn(y_batch, y_pred)

    # Compute gradients
    gradients = tape.gradient(loss, [W1, b1, W2, b2, W3, b3])
    # Update weights and biases
    optimizer.apply_gradients(zip(gradients, [W1, b1, W2, b2, W3,
b3]))

    epoch_loss += loss.numpy()

epoch_loss /= num_batches
loss_history.append(epoch_loss)
print(f"Epoch {epoch+1}/{epochs}, Loss: {epoch_loss:.4f}")

# Plot loss curve
plt.plot(loss_history)
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss over Epochs')
plt.show()

```

Observations: -

- Loss decreased steadily across epochs.
- The training curve showed convergence, confirming that backpropagation updated weights correctly.
- Initial epochs showed higher loss, which reduced as the model learned.

Conclusion: -

The practical successfully demonstrated the **implementation of backpropagation** to train a Deep Neural Network with two hidden layers using TensorFlow. The model's training loss decreased steadily, proving that weights were optimized correctly. This method is fundamental in training all modern deep learning architectures.

Result: -

The DNN was successfully trained using the backpropagation algorithm, and the training loss reduced significantly over 20 epochs, validating the correctness of the implementation.

