**Code:-**

```
class McCullochPittsNeuron:
    def __init__(self, weights, threshold):
        self.weights = weights
        self.threshold = threshold
    def activation(self, inputs):
        weighted_sum = sum(w * x for w, x in zip(self.weights, inputs))
        if weighted_sum >= self.threshold:
            return 1
        else:
            return 0
# Define the logic functions as different neurons
# AND Function: Output 1 only if both inputs are 1
and_neuron = McCullochPittsNeuron(weights=[1, 1], threshold=2)
# OR Function: Output 1 if at least one input is 1
or_neuron = McCullochPittsNeuron(weights=[1, 1], threshold=1)
# NAND Function: Output 0 only if both inputs are 1
nand_neuron = McCullochPittsNeuron(weights=[-1, -1], threshold=-2)
# Test the neurons with all possible input combinations
inputs = [(0, 0), (0, 1), (1, 0), (1, 1)]
# Testing the AND function
print("AND Function:")
for input_pair in inputs:
    print(f"Input: {input_pair}, Output: {and_neuron.activation(input_pair)}")
# Testing the OR function
print("\nOR Function:")
for input_pair in inputs:
    print(f"Input: {input_pair}, Output: {or_neuron.activation(input_pair)}")
# Testing the NAND function
print("\nNAND Function:")
for input_pair in inputs:
    print(f"Input: {input_pair}, Output: {nand_neuron.activation(input_pair)}")
```

**Output:-**

```
AND Function:
Input: (0, 0), Output: 0
Input: (0, 1), Output: 0
Input: (1, 0), Output: 0
Input: (1, 1), Output: 1

OR Function:
Input: (0, 0), Output: 0
Input: (0, 1), Output: 1
Input: (1, 0), Output: 1
Input: (1, 1), Output: 1

NAND Function:
Input: (0, 0), Output: 1
Input: (0, 1), Output: 1
Input: (1, 0), Output: 1
Input: (1, 1), Output: 0
```

**Experiment No. 2**

**Code:-**

```
w1, w2, b = 0.5, 0.5, -1
def activate(x):
    return 1 if x >= 0 else 0
def train_perceptron(inputs, desired_outputs, learning_rate, epochs):
    global w1, w2, b
    for epoch in range(epochs):
        total_error = 0
        for i in range(len(inputs)):
            A, B = inputs[i]
            target_output = desired_outputs[i]
            output = activate(w1 * A + w2 * B + b)
            error = target_output - output
            w1 += learning_rate * error * A
            w2 += learning_rate * error * B
            b += learning_rate * error
            total_error += abs(error)
        if total_error == 0:
            break
inputs = [(0, 0), (0, 1), (1, 0), (1, 1)]
desired_outputs = [0, 0, 0, 1]
learning_rate = 0.1
epochs = 100
train_perceptron(inputs, desired_outputs, learning_rate, epochs)
for i in range(len(inputs)):
    A, B = inputs[i]
    output = activate(w1 * A + w2 * B + b)
    print(f"Input: ({A}, {B})  Output: {output}")
```

**Output:-**

```
 Input: (0, 0)  Output: 0
Input: (0, 1)  Output: 0
Input: (1, 0)  Output: 0
Input: (1, 1)  Output: 1
```

**Experiment No. 3**

**Code:-**

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical

# Load and preprocess data
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0  # Normalize to [0, 1]
x_train = x_train.reshape(-1, 28*28)
x_test = x_test.reshape(-1, 28*28)
y_train = to_categorical(y_train, 10)  # One-hot encoding
y_test = to_categorical(y_test, 10)
# Define the model
model = models.Sequential([
    layers.InputLayer(input_shape=(784,)),
    layers.Dense(128, activation='relu'),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')  # 10 classes for digits 0-9])

# Compile the model
model.compile(optimizer=Adam(), loss='categorical_crossentropy', metrics=['accuracy'])

# Train the model
history = model.fit(x_train, y_train, epochs=10, batch_size=32, validation_data=(x_test, y_test))

# Evaluate the model
test_loss, test_accuracy = model.evaluate(x_test, y_test)
print(f"Test Accuracy: {test_accuracy * 100:.2f}%")
```

**Output:-**

```
Epoch 1/10

1875/1875 [==============================] - 5s 3ms/step - loss: 0.2832 - accuracy: 0.9164 -
val_loss: 0.1482 - val_accuracy: 0.9557
Epoch 2/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.1165 - accuracy: 0.9655 -
val_loss: 0.1073 - val_accuracy: 0.9680
...
Epoch 10/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.0559 - accuracy: 0.9827 -
val_loss: 0.0763 - val_accuracy: 0.9772

Test Accuracy: 97.72%
```

**Experiment No. 4**

**Code:-**

```
import numpy as np
# Sigmoid activation function and its derivative
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
def sigmoid_derivative(x):
    return x * (1 - x)
# Mean Squared Error (MSE) loss function and its derivative
def mse_loss(y_true, y_pred):
    return np.mean((y_true - y_pred)**2)
def mse_loss_derivative(y_true, y_pred):
    return y_pred - y_true
# Neural Network class
class NeuralNetwork:
    def __init__(self, input_size, hidden_size1, hidden_size2, output_size):
        # Initialize weights and biases
        self.w1 = np.random.randn(input_size, hidden_size1)
        self.b1 = np.zeros((1, hidden_size1))
        self.w2 = np.random.randn(hidden_size1, hidden_size2)
        self.b2 = np.zeros((1, hidden_size2))
        self.w3 = np.random.randn(hidden_size2, output_size)
        self.b3 = np.zeros((1, output_size))
    def forward(self, X):
        # Forward pass
        self.z1 = np.dot(X, self.w1) + self.b1
        self.a1 = sigmoid(self.z1)

        self.z2 = np.dot(self.a1, self.w2) + self.b2
        self.a2 = sigmoid(self.z2)

        self.z3 = np.dot(self.a2, self.w3) + self.b3
        self.a3 = sigmoid(self.z3)  # Output layer

        return self.a3

    def backward(self, X, y, learning_rate):
        # Backward pass (Gradient computation)
        output_error = mse_loss_derivative(y, self.a3)
        output_delta = output_error * sigmoid_derivative(self.a3)
        # Backpropagate to second hidden layer
        hidden2_error = output_delta.dot(self.w3.T)
        hidden2_delta = hidden2_error * sigmoid_derivative(self.a2)
        # Backpropagate to first hidden layer
        hidden1_error = hidden2_delta.dot(self.w2.T)
        hidden1_delta = hidden1_error * sigmoid_derivative(self.a1)
        # Update weights and biases using gradient descent
        self.w3 -= self.a2.T.dot(output_delta) * learning_rate
        self.b3 -= np.sum(output_delta, axis=0, keepdims=True) * learning_rate
        self.w2 -= self.a1.T.dot(hidden2_delta) * learning_rate
        self.b2 -= np.sum(hidden2_delta, axis=0, keepdims=True) * learning_rate

        self.w1 -= X.T.dot(hidden1_delta) * learning_rate
        self.b1 -= np.sum(hidden1_delta, axis=0, keepdims=True) * learning_rate
```

```
    def train(self, X, y, epochs, learning_rate):
        for epoch in range(epochs):
            # Forward pass
            output = self.forward(X)

            # Compute loss (optional, for monitoring)
            loss = mse_loss(y, output)
            if epoch % 100 == 0:
                print(f"Epoch {epoch}/{epochs}, Loss: {loss}")

            # Backward pass and weight update
            self.backward(X, y, learning_rate)

# Example Usage
if __name__ == "__main__":
    # Define the neural network
    nn = NeuralNetwork(input_size=3, hidden_size1=5, hidden_size2=4, output_size=1)

    # Training data (XOR problem)
    X = np.array([[0, 0, 1],
                  [0, 1, 1],
                  [1, 0, 1],
                  [1, 1, 1]])

    y = np.array([[0], [1], [1], [0]])

    # Train the network
    nn.train(X, y, epochs=1000, learning_rate=0.1)

    # Test the trained network
    print("Predictions after training:")
    predictions = nn.forward(X)
    print(predictions)
```

**Output:-**

```
Epoch 0/1000, Loss: 0.2701

Epoch 100/1000, Loss: 0.0801

Epoch 200/1000, Loss: 0.0603

Epoch 300/1000, Loss: 0.0568

...

Predictions after training:

[[0.0234]

 [0.9754]

 [0.9746]

 [0.0337]]
```

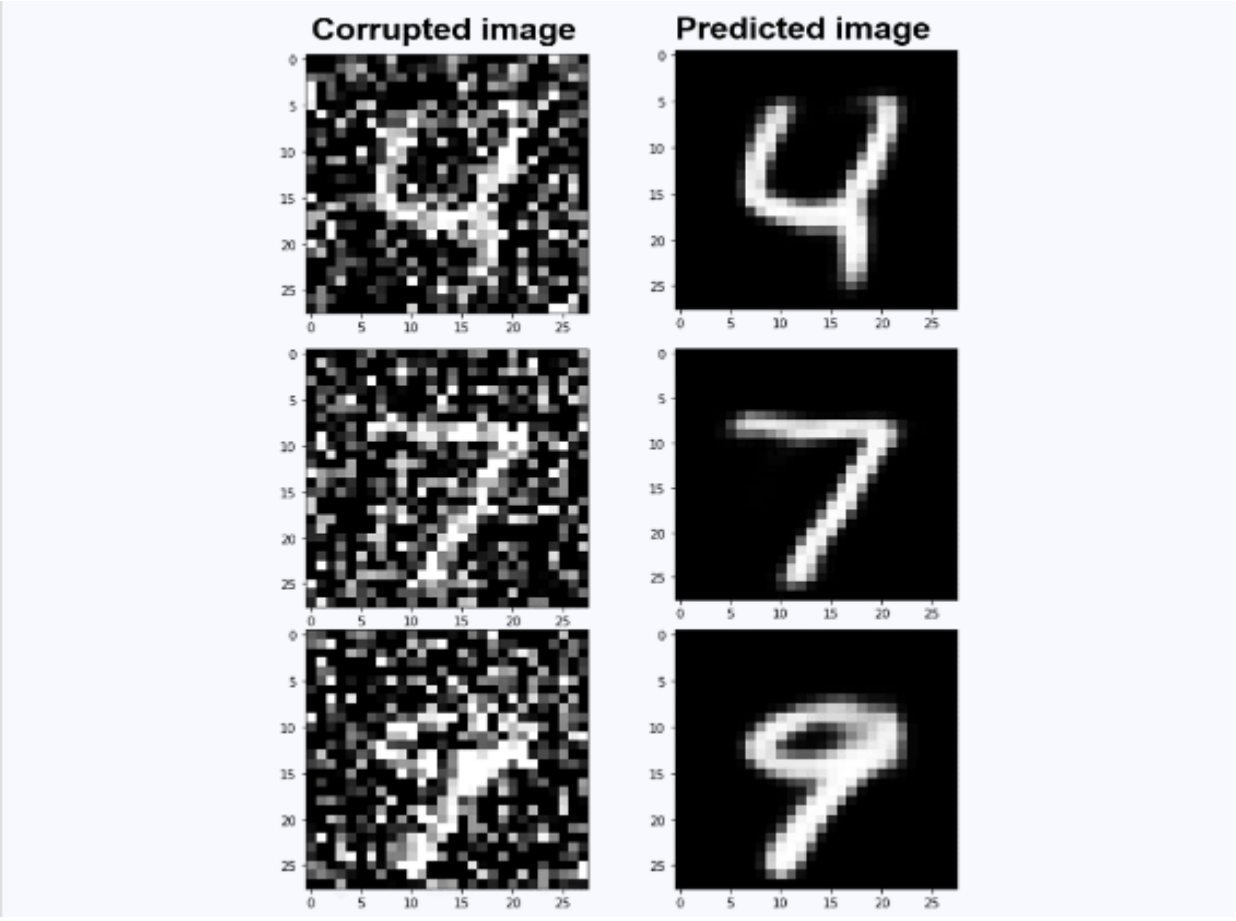
# Experiment No. 5

**Step-by-step Code:**

```
input_img = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(input_im)
x = layers.MaxPooling2D((2, 2), padding='same')(x)
encoded = layers.MaxPooling2D((2, 2), padding='same')(x)

# At this point the representation is (7, 7, 32)
x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(encoded)
x = layers.UpSampling2D((2, 2))(x)
x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = layers.UpSampling2D((2, 2))(x)
decoded = layers.Conv2D(1, (3, 3), activation='sigmoid', padding='same')()
autoencoder = keras.Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
# Code example (model (autoencoder) validation)
autoencoder.fit(x_train_noisy, x_train,
epochs=100,
batch_size=128,
shuffle=True,
validation_data=(x_test_noisy, x_test))
```

**Output:**



# Experiment No : 6

## Code Implementation:

```
python
Copy
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from tensorflow.keras.utils import to_categorical

# Load MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Reshape the data to include channel dimension (grayscale images)
x_train = np.expand_dims(x_train, axis=-1)
x_test = np.expand_dims(x_test, axis=-1)

# Normalize the pixel values to the range [0, 1]
x_train, x_test = x_train / 255.0, x_test / 255.0

# Convert labels to categorical (one-hot encoding)
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Build the CNN model
model = Sequential()

# First convolutional layer
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(MaxPooling2D(pool_size=(2, 2)))

# Second convolutional layer
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

# Third convolutional layer
model.add(Conv2D(128, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

# Flatten the output from convolutional layers
model.add(Flatten())

# Fully connected layer
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.2))

# Output layer
model.add(Dense(10, activation='softmax'))

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Model summary
model.summary()

# Train the model
history = model.fit(x_train, y_train, epochs=5, batch_size=64, validation_data=(x_test,
y_test))

# Evaluate the model on the test set
test_loss, test_accuracy = model.evaluate(x_test, y_test)
print(f"Test accuracy: {test_accuracy * 100:.2f}%")
```

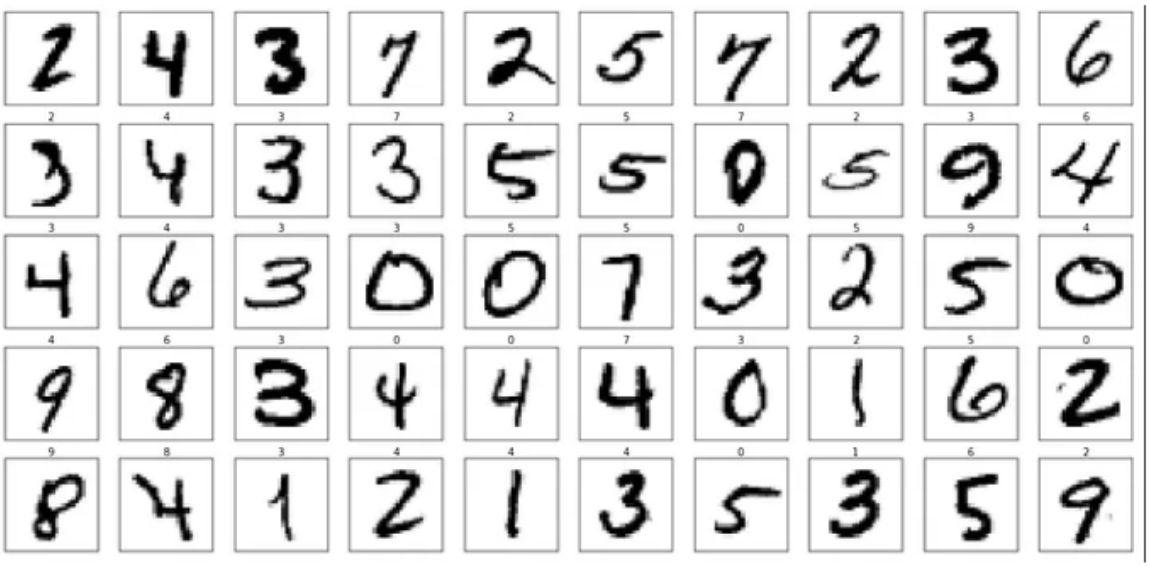
```
# Plot training and validation accuracy
plt.figure(figsize=(12, 4))

# Accuracy plot
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

# Loss plot
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.tight_layout()
plt.show()
```

## OUTPUT :



**EXPERIMENT NO : 7**

## Code Implementation :

```
python
Copy
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import imdb
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense, Dropout

# Load IMDB dataset (top 10000 most frequent words)
max_features = 10000
maxlen = 500

(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)

# Pad sequences to uniform length (maxlen)
x_train = pad_sequences(x_train, maxlen=maxlen)
x_test = pad_sequences(x_test, maxlen=maxlen)

# Build the LSTM model
model = Sequential()

# Embedding layer
model.add(Embedding(input_dim=max_features, output_dim=128, input_length=maxlen))

# LSTM layer
model.add(LSTM(128, dropout=0.2, recurrent_dropout=0.2))

# Output layer (binary classification)
model.add(Dense(1, activation='sigmoid'))

# Compile the model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

# Print the model summary
model.summary()

# Train the model
history = model.fit(x_train, y_train, epochs=5, batch_size=64, validation_data=(x_test,
y_test))

# Evaluate the model
score, accuracy = model.evaluate(x_test, y_test)
print(f"Test accuracy: {accuracy * 100:.2f}%")

# Plot training and validation accuracy/loss
plt.figure(figsize=(12, 4))

# Plot accuracy
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

# Plot loss
plt.subplot(1, 2, 2)
```

```
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.tight_layout()
plt.show()
```

---

## Output:

1. **Model Summary** (Printed after model compilation): The model summary will display the architecture, showing the layers and the number of parameters:

```
markdown
Copy
Model: "sequential"
_________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================
embedding (Embedding)        (None, 500, 128)      1280000
_________________________________________________________
lstm (LSTM)                  (None, 128)           131584
_________________________________________________________
dense (Dense)                (None, 1)             129
=================================================
Total params: 1,411,713
Trainable params: 1,411,713
Non-trainable params: 0
```

2. **Training Progress** (During model training): The training will show the loss and accuracy for each epoch:

```
arduino
Copy
Epoch 1/5
782/782 [==============================] - 16s 20ms/step - loss: 0.6933 -
accuracy: 0.5007 - val_loss: 0.6927 - val_accuracy: 0.5092
Epoch 2/5
782/782 [==============================] - 15s 20ms/step - loss: 0.5742 -
accuracy: 0.7482 - val_loss: 0.4154 - val_accuracy: 0.8087
...
Epoch 5/5
782/782 [==============================] - 15s 20ms/step - loss: 0.2119 -
accuracy: 0.9165 - val_loss: 0.3989 - val_accuracy: 0.8302
```

3. **Test Accuracy** (After training completes): Once the model is trained, you'll see the evaluation result on the test set:

```
yaml
Copy
Test accuracy: 83.02%
```

4. **Accuracy and Loss Plots**:
   - **Accuracy Plot**: Shows the model's accuracy on both training and validation data over the epochs.
   - **Loss Plot**: Displays the loss on both training and validation data over the epochs.

   The plots will show how accuracy increases and loss decreases over time, indicating the model is learning well.