FACULTY OF ENGINEERING AND TECHNOLOGY (Co-Ed)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

LAB MANUAL

Neural Networks and Deep Learning Laboratory

(22CSL78)

2025-2026

Neural Networks and Deep Learning Laboratory [As per Choice Based Credit System(CBCS)scheme] (Effective from the academic year 2025-2026) SEMESTER – VII			
Course Code	**22CSL78**	**CIE Marks**	**50**
Number of Lecture Hours/Week	**02**	**SEE Marks**	**50**
Total Number of Lecture Hours	**30**	**Exam Hours**	**3**
CREDIT– 01			
Course Objectives : This course will enable students to: • Implement and train various types of neural network models, starting with a basic multilayer perceptron (MLP) for classification tasks. • Apply deep learning to computer vision. • Use deep learning for natural language processing (NLP). • Explore advanced deep learning architectures. • Develop real-world deep learning applications.			
Programs			
1)Create neural network class and initialize those weights and biases.			
2)Implement all activation functions in Neural Network.			
3)Implement Loss function for Neural Network.			
4)Implement Forward propagation and Backward propagation.			
5)Program to train and test a neural network.			
6)Train and test the Convolution neural network using any data set , preprocess it.			
7)Implement the Convolution neural network for image classification.			
8)Train and test the recurrent neural network using any data set , preprocess it.			
9)Implement Facial recognition using neural network.			
10) Implement object detection using neural network.			

COURSE OUTCOMES: At the end of the course the student will be able to:

CO1	Make use of deep learning techniques to implement fundamental concepts of Neural Networks
CO2	Apply convolutional neural networks (CNNs) for computer vision tasks.
CO3	Develop and apply recurrent neural networks (RNNs) for sequential data.
CO4	Apply deep learning techniques for facial recognition & object detection
CO5	Prepare a well organized laboratory report detailing experimental procedure,reports

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO 10	PO 11	PSO 1	PSO 2	PSO 3
CO1	3	3	2	2	2	-	-	-	2	2	-			
CO2	3	3	3	2	2	-	-	1	2	-	-			
CO3	3	3	2	2	2	-	-	-	-	-	-			
CO4	3	3	3	2	3	-	-	-	2	3	2			
Avg	**3.0**	**3**	**2.5**	**2**	**2.2**	**-**	**-**	**1**	**1.5**	**1.2**	**1**			

Prog 1: Create neural network class and initialize those weights and biases.

```
import numpy as np
class NeuralNetwork:
    def __init__(self, layers):
        layers (list): Number of neurons in each layer.
                    Example: [3, 5, 2] means:
                    - 3 neurons in input layer
                    - 5 neurons in hidden layer
                    - 2 neurons in output layer

        self.layers = layers
        self.weights = []
        self.biases = []
        self.initialize_parameters()

    def initialize_parameters(self):
        """Initialize weights and biases with random values."""
        np.random.seed(42)  # for reproducibility

        for i in range(1, len(self.layers)):
            # He initialization for weights
            weight_matrix = np.random.randn(self.layers[i], self.layers[i - 1]) * np.sqrt(2. /
self.layers[i - 1])
            bias_vector = np.zeros((self.layers[i], 1))

            self.weights.append(weight_matrix)
            self.biases.append(bias_vector)

    def summary(self):
```

```
        print("Neural Network Architecture:")
        for i in range(len(self.layers) - 1):
            print(f" Layer {i} -> Layer {i+1}: "
                  f"Weights shape {self.weights[i].shape}, "
                  f"Biases shape {self.biases[i].shape}")
if __name__ == "__main__":
    nn = NeuralNetwork([3, 5, 2])
    nn.summary()
```

Output :

```
Neural Network Architecture:
 Layer 0 -> Layer 1: Weights shape (5, 3), Biases shape (5, 1)
 Layer 1 -> Layer 2: Weights shape (2, 5), Biases shape (2, 1)
```

Prog 2: Implement all activation functions in Neural Network.

```
import numpy as np

class ActivationFunctions:

    @staticmethod
    def sigmoid(x):
        return 1 / (1 + np.exp(-x))
    @staticmethod
    def sigmoid_derivative(x):
        s = ActivationFunctions.sigmoid(x)
        return s * (1 - s)
    @staticmethod
    def tanh(x):
        return np.tanh(x)
    @staticmethod
    def tanh_derivative(x):
        return 1 - np.tanh(x)**2
    @staticmethod
    def relu(x):
        return np.maximum(0, x)
    @staticmethod
    def relu_derivative(x):
        return np.where(x > 0, 1, 0)
    @staticmethod
    def leaky_relu(x, alpha=0.01):
        return np.where(x > 0, x, alpha * x)
```

```
    @staticmethod
    def leaky_relu_derivative(x, alpha=0.01):
        dx = np.ones_like(x)
        dx[x < 0] = alpha
        return dx

    @staticmethod
    def elu(x, alpha=1.0):
        return np.where(x > 0, x, alpha * (np.exp(x) - 1))

    @staticmethod
    def elu_derivative(x, alpha=1.0):
        return np.where(x > 0, 1, ActivationFunctions.elu(x, alpha) + alpha)

    @staticmethod
    def softmax(x):
        exps = np.exp(x - np.max(x, axis=0, keepdims=True))  # stability fix
        return exps / np.sum(exps, axis=0, keepdims=True)

    @staticmethod
    def softmax_derivative(x):
        s = ActivationFunctions.softmax(x)
        return np.diagflat(s) - np.dot(s, s.T)
    @staticmethod
    def linear(x):
        return x
    @staticmethod
    def linear_derivative(x):
        return np.ones_like(x)
if __name__ == "__main__":
```

```
x = np.array([-2.0, -1.0, 0.0, 1.0, 2.0])

print("Input:", x)
print("ReLU:", ActivationFunctions.relu(x))
print("Sigmoid:", ActivationFunctions.sigmoid(x))
print("Tanh:", ActivationFunctions.tanh(x))
print("Leaky ReLU:", ActivationFunctions.leaky_relu(x))
print("ELU:", ActivationFunctions.elu(x))
print("Softmax:", ActivationFunctions.softmax(x))
```

Output:

```
Input: [-2. -1.  0.  1.  2.]
ReLU: [0. 0. 0. 1. 2.]
Sigmoid: [0.11920292 0.26894142 0.5        0.73105858 0.88079708]
Tanh: [-0.96402758 -0.76159416  0.          0.76159416  0.96402758]
Leaky ReLU: [-0.02 -0.01  0.    1.    2.  ]
ELU: [-0.86466472 -0.63212056  0.          1.          2.        ]
Softmax: [0.01165623 0.03168492 0.08612854 0.23412166 0.63640865]
```

Prog 3: Implement Loss function for Neural Network.

```
import numpy as np
class LossFunctions:

    @staticmethod
    def mse(y_true, y_pred):
        """Mean Squared Error loss."""
        return np.mean((y_true - y_pred) ** 2)

    @staticmethod
    def mse_derivative(y_true, y_pred):
        return 2 * (y_pred - y_true) / y_true.size

    @staticmethod
    def mae(y_true, y_pred):
        return np.mean(np.abs(y_true - y_pred))

    @staticmethod
    def mae_derivative(y_true, y_pred):
        return np.sign(y_pred - y_true) / y_true.size

    @staticmethod
    def binary_cross_entropy(y_true, y_pred):
        eps = 1e-15  # avoid log(0)
        y_pred = np.clip(y_pred, eps, 1 - eps)
        return -np.mean(y_true * np.log(y_pred) + (1 - y_true) * np.log(1 - y_pred))

    @staticmethod
```

```
    def binary_cross_entropy_derivative(y_true, y_pred):
        eps = 1e-15
        y_pred = np.clip(y_pred, eps, 1 - eps)
        return (-(y_true / y_pred) + (1 - y_true) / (1 - y_pred)) / y_true.size
    @staticmethod
    def categorical_cross_entropy(y_true, y_pred):
        y_true: one-hot encoded true labels
        y_pred: predicted probabilities (softmax outputs)
        eps = 1e-15
        y_pred = np.clip(y_pred, eps, 1 - eps)
        return -np.mean(np.sum(y_true * np.log(y_pred), axis=0))


    @staticmethod
    def categorical_cross_entropy_derivative(y_true, y_pred):
        eps = 1e-15
        y_pred = np.clip(y_pred, eps, 1 - eps)
        return -(y_true / y_pred) / y_true.shape[0]
if __name__ == "__main__":
    y_true = np.array([[1, 0, 0], [0, 1, 0]]).T  # one-hot encoded (3 classes, 2 samples)
    y_pred = np.array([[0.7, 0.2, 0.1], [0.1, 0.8, 0.1]]).T


    print("Mean Squared Error:", LossFunctions.mse(y_true, y_pred))
    print("Categorical Cross Entropy:", LossFunctions.categorical_cross_entropy(y_true,
y_pred))
```

Output:

Mean Squared Error: 0.03333333333333334

Categorical Cross Entropy: 0.2899092476264711

Prog 4: Implement Forward propagation and Backward propagation.

```
import numpy as np

# Activation function (Sigmoid) and its derivative
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

# Input dataset (4 samples, 2 features)
X = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])

# Output labels (XOR problem)
y = np.array([[0], [1], [1], [0]])

# Set random seed for reproducibility
np.random.seed(42)

# Initialize weights and biases
input_layer_size = 2
hidden_layer_size = 2
output_layer_size = 1
```

```
W1 = np.random.uniform(size=(input_layer_size, hidden_layer_size))
b1 = np.random.uniform(size=(1, hidden_layer_size))
W2 = np.random.uniform(size=(hidden_layer_size, output_layer_size))
b2 = np.random.uniform(size=(1, output_layer_size))

# Training parameters
learning_rate = 0.5
epochs = 10000

# Training loop
for epoch in range(epochs):
    # ---- Forward propagation ----
    hidden_input = np.dot(X, W1) + b1
    hidden_output = sigmoid(hidden_input)

    final_input = np.dot(hidden_output, W2) + b2
    predicted_output = sigmoid(final_input)

    # ---- Backward propagation ----
    error = y - predicted_output
    d_predicted_output = error * sigmoid_derivative(predicted_output)

    error_hidden = d_predicted_output.dot(W2.T)
    d_hidden_layer = error_hidden * sigmoid_derivative(hidden_output)

    # ---- Update weights and biases ----
    W2 += hidden_output.T.dot(d_predicted_output) * learning_rate
    b2 += np.sum(d_predicted_output, axis=0, keepdims=True) * learning_rate
    W1 += X.T.dot(d_hidden_layer) * learning_rate
    b1 += np.sum(d_hidden_layer, axis=0, keepdims=True) * learning_rate
```

```
    # Print progress occasionally
    if epoch % 2000 == 0:
        loss = np.mean(np.square(error))
        print(f"Epoch {epoch} - Loss: {loss:.6f}")

# Final output
print("\n Final predicted output after training:")
print(predicted_output)
```

Output :

```
Epoch 0 - Loss: 0.324659
Epoch 2000 - Loss: 0.002546
Epoch 4000 - Loss: 0.000902
Epoch 6000 - Loss: 0.000537
Epoch 8000 - Loss: 0.000380

Final predicted output after training:
[[0.01890475]
 [0.98371361]
 [0.98369334]
 [0.01686123]]
```

Prog 5 : Program to train and test a neural network.

```
import numpy as np
# Activation + derivative
def sigmoid(x): return 1 / (1 + np.exp(-x))
def sigmoid_deriv(a): return a * (1 - a)

# Neural Network
class NeuralNet:
    def __init__(self, lr=0.1):
        np.random.seed(42)
        self.W1 = np.random.randn(4, 2)   # 2→4
        self.b1 = np.zeros((4, 1))
        self.W2 = np.random.randn(1, 4)   # 4→1
        self.b2 = np.zeros((1, 1))
        self.lr = lr

    def forward(self, X):
        self.Z1 = np.dot(self.W1, X) + self.b1
        self.A1 = np.maximum(0, self.Z1)          # ReLU
        self.Z2 = np.dot(self.W2, self.A1) + self.b2
        self.A2 = sigmoid(self.Z2)                # Sigmoid
        return self.A2

    def backward(self, X, Y):
        m = Y.shape[1]
        dZ2 = (self.A2 - Y) * sigmoid_deriv(self.A2)
        dW2 = np.dot(dZ2, self.A1.T) / m
        dB2 = np.sum(dZ2, axis=1, keepdims=True) / m
        dA1 = np.dot(self.W2.T, dZ2)
```

```
        dZ1 = dA1 * (self.A1 > 0)
        dW1 = np.dot(dZ1, X.T) / m
        dB1 = np.sum(dZ1, axis=1, keepdims=True) / m
        # Update
        self.W1 -= self.lr * dW1; self.b1 -= self.lr * dB1
        self.W2 -= self.lr * dW2; self.b2 -= self.lr * dB2

    def train(self, X, Y, epochs=2000):
        for i in range(epochs + 1):
            out = self.forward(X)
            loss = np.mean((Y - out)**2)
            self.backward(X, Y)
            if i % 200 == 0:
                print(f"Epoch {i:4d} | Loss: {loss:.5f}")

# --- Train & Test ---
X = np.array([[0,0,1,1],[0,1,0,1]])   # inputs
Y = np.array([[0,1,1,0]])             # outputs

nn = NeuralNet(lr=0.1)
nn.train(X, Y, epochs=4000)

preds = nn.forward(X)
print("\nPredictions:\n", preds.round(3))
```

Output :

```
Epoch    0 | Loss: 0.26242
Epoch  200 | Loss: 0.24559
Epoch  400 | Loss: 0.22857
Epoch  600 | Loss: 0.19455
```

Epoch 800 | Loss: 0.15112

Epoch 1000 | Loss: 0.12031

Epoch 1200 | Loss: 0.09432

Epoch 1400 | Loss: 0.07337

Epoch 1600 | Loss: 0.05717

Epoch 1800 | Loss: 0.04514

Epoch 2000 | Loss: 0.03632

Epoch 2200 | Loss: 0.02974

Epoch 2400 | Loss: 0.02476

Epoch 2600 | Loss: 0.02096

Epoch 2800 | Loss: 0.01801

Epoch 3000 | Loss: 0.01569

Epoch 3200 | Loss: 0.01379

Epoch 3400 | Loss: 0.01227

Epoch 3600 | Loss: 0.01099

Epoch 3800 | Loss: 0.00993

Epoch 4000 | Loss: 0.00903

Predictions:

[[0.143 0.923 0.923 0.063]]

Prog 6: Train and test the Convolution neural network using any data set , preprocess it.

```
# Import libraries
import tensorflow as tf
from tensorflow.keras import layers, models

# Load and preprocess data
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

# Normalize pixel values (0–1) and reshape for CNN (samples, height, width, channels)
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
x_train = x_train[..., None]  # add channel dimension
x_test = x_test[..., None]

# Build CNN model
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')
])

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

```
# Train the model
print("Training CNN on MNIST dataset...")
model.fit(x_train, y_train, epochs=3, batch_size=64, validation_split=0.1, verbose=1)

# Evaluate on test set
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=0)
print(f"\nTest accuracy: {test_acc * 100:.2f}%")
```

Output:

```
Epoch 1/3
844/844 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 10s 11ms/step - accuracy: 0.8800
- loss: 0.4101 - val_accuracy: 0.9817 - val_loss: 0.0681
Epoch 2/3
844/844 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 9s 10ms/step - accuracy: 0.9816 -
loss: 0.0602 - val_accuracy: 0.9867 - val_loss: 0.0462
Epoch 3/3
844/844 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 8s 10ms/step - accuracy: 0.9864 -
loss: 0.0425 - val_accuracy: 0.9887 - val_loss: 0.0397

 Test accuracy: 98.90%
```

Prog 7: Implement the Convolution neural network for image classification.

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import cifar10

# Load and preprocess the dataset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Normalize pixel values (0–1)
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
# Convert labels to flat array
y_train = y_train.flatten()
y_test = y_test.flatten()
# Build CNN model
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')  # 10 classes
])


#  Compile model
model.compile(optimizer='adam',
```

```
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy'])

#  Train model
print(" Training CNN on CIFAR-10 dataset...")
model.fit(x_train, y_train, epochs=5, batch_size=64, validation_split=0.1, verbose=1)

#  Evaluate model
print("\n Evaluating on test data...")
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=1)
print(f"\n Test accuracy: {test_acc * 100:.2f}%")

# Save model
# model.save("cnn_cifar10_model.h5")
```

Output :

704/704 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **11s** 14ms/step - accuracy: 0.3326 - loss: 1.8153 - val_accuracy: 0.5308 - val_loss: 1.2969

Epoch 2/5

704/704 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **9s** 13ms/step - accuracy: 0.5502 - loss: 1.2693 - val_accuracy: 0.5732 - val_loss: 1.1751

Epoch 3/5

704/704 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **9s** 13ms/step - accuracy: 0.6129 - loss: 1.0956 - val_accuracy: 0.6386 - val_loss: 1.0217

Epoch 4/5

704/704 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **9s** 13ms/step - accuracy: 0.6570 - loss: 0.9834 - val_accuracy: 0.6634 - val_loss: 0.9626

Epoch 5/5

704/704 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **9s** 13ms/step - accuracy: 0.6837 - loss: 0.9105 - val_accuracy: 0.6844 - val_loss: 0.8937

313/313 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **1s** 3ms/step - accuracy: 0.6730 - loss: 0.9409

Test accuracy: 67.00%

Prog 8: Train and test the recurrent neural network using any data set , preprocess it.

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import imdb
from tensorflow.keras.preprocessing import sequence

# Load Dataset (IMDB Movie Reviews)
max_words = 10000   # Only keep top 10k most common words
max_len = 200       # Cut reviews after 200 words
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_words)

# Preprocess Data (Pad sequences to same length)
x_train = sequence.pad_sequences(x_train, maxlen=max_len)
x_test = sequence.pad_sequences(x_test, maxlen=max_len)

#  Build RNN model (LSTM)
model = models.Sequential([
    layers.Embedding(input_dim=max_words, output_dim=64, input_length=max_len),
    layers.LSTM(64),
    layers.Dense(1, activation='sigmoid')
])

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

```
#  Train model
print("\n Training RNN (LSTM) on IMDB dataset...")
model.fit(x_train, y_train, epochs=3, batch_size=64, validation_split=0.2, verbose=1)

#  Evaluate on test data
print("\n Evaluating on test data...")
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=1)
print(f"\n Test Accuracy: {test_acc * 100:.2f}%")
```

Output :

Model: "sequential_2"

Layer (type)	Output Shape	Param #
embedding (Embedding)	?	0 (unbuilt)
lstm (LSTM)	?	0 (unbuilt)
dense_4 (Dense)	?	0 (unbuilt)

Total params: 0 (0.00 B)

Trainable params: 0 (0.00 B)

Non-trainable params: 0 (0.00 B)

Training RNN (LSTM) on IMDB dataset...

Epoch 1/3

313/313 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **25s** 74ms/step - accuracy: 0.6952 - loss: 0.5481 - val_accuracy: 0.8476 - val_loss: 0.3683

Epoch 2/3

313/313 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **24s** 75ms/step - accuracy: 0.8951 - loss: 0.2680 - val_accuracy: 0.8500 - val_loss: 0.3421

Epoch 3/3

313/313 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **24s** 75ms/step - accuracy: 0.9297 - loss: 0.1940 - val_accuracy: 0.8508 - val_loss: 0.3468

Evaluating on test data...

782/782 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **14s** 18ms/step - accuracy: 0.8423 - loss: 0.3727

Test Accuracy: 84.36%

Prog 9: Implement Facial recognition using neural network.

```
import tensorflow as tf
from tensorflow.keras import layers, models
from sklearn.datasets import fetch_lfw_people
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
import numpy as np

#  Load the LFW (Labeled Faces in the Wild) dataset
print(" Loading dataset...")
faces = fetch_lfw_people(min_faces_per_person=70, color=True, resize=0.4)
X = faces.images / 255.0          # Normalize pixel values
y = faces.target
target_names = faces.target_names

# Split into training and testing sets
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Reshape data for CNN
x_train = np.expand_dims(x_train, -1) if x_train.ndim == 3 else x_train
x_test = np.expand_dims(x_test, -1) if x_test.ndim == 3 else x_test

# Build CNN model
model = models.Sequential([
    layers.Conv2D(32, (3,3), activation='relu', input_shape=x_train.shape[1:]),
    layers.MaxPooling2D(2,2),
    layers.Conv2D(64, (3,3), activation='relu'),
```

```
    layers.MaxPooling2D(2,2),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(len(target_names), activation='softmax')
])

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

# Show summary
model.summary()

#  Train model
print("\n Training CNN for facial recognition...")
model.fit(x_train, y_train, epochs=5, batch_size=32, validation_split=0.1, verbose=1)

#  Evaluate on test data
print("\n Evaluating model...")
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=1)
print(f"\nTest Accuracy: {test_acc * 100:.2f}%")

#  Predict on one test sample
sample = np.expand_dims(x_test[0], axis=0)
pred = np.argmax(model.predict(sample), axis=1)[0]
print(f"\n Predicted: {target_names[pred]}")
print(f" Actual: {target_names[y_test[0]]}")
```

Output :

Training CNN for facial recognition...

Epoch 1/5

28/28 [==============================] - 6s 170ms/step - loss: 1.3391 - accuracy: 0.5784 - val_loss: 0.7633 - val_accuracy: 0.8023

Epoch 5/5

28/28 [==============================] - 5s 164ms/step - loss: 0.3412 - accuracy: 0.9056 - val_loss: 0.4232 - val_accuracy: 0.8800

Evaluating model...

9/9 [==============================] - 0s 22ms/step - loss: 0.4412 - accuracy: 0.8760

Test Accuracy: 87.60%

Prog 10: Implement object detection using neural network.

```
import tensorflow as tf
import cv2
import numpy as np

# Load a pre-trained object detection model from TensorFlow Hub
import tensorflow_hub as hub

print(" Loading pre-trained MobileNet SSD model...")
detector = hub.load("https://tfhub.dev/tensorflow/ssd_mobilenet_v2/2")

# Load labels for COCO dataset
labels_path = tf.keras.utils.get_file(
    'coco_labels.txt',
    'https://raw.githubusercontent.com/amikelive/coco-labels/master/coco-labels-paper.txt'
)
with open(labels_path, "r") as f:
    labels = [line.strip() for line in f.readlines()]

# Load and preprocess an image
image_path = tf.keras.utils.get_file(
    'dog.jpg',
    'https://storage.googleapis.com/mledu-datasets/cats_and_dogs_filtered/train/dogs/dog.1.jpg'
)
image = cv2.imread(image_path)
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
input_tensor = tf.convert_to_tensor([image_rgb], dtype=tf.uint8)
```

```
# Run detection
print(" Running object detection...")
detections = detector(input_tensor)

# Extract detection results
boxes = detections['detection_boxes'][0].numpy()
classes = detections['detection_classes'][0].numpy().astype(int)
scores = detections['detection_scores'][0].numpy()

# Draw boxes for detections above threshold
for i in range(len(scores)):
    if scores[i] > 0.5:
        ymin, xmin, ymax, xmax = boxes[i]
        h, w, _ = image.shape
        (left, top, right, bottom) = (int(xmin * w), int(ymin * h),
                                      int(xmax * w), int(ymax * h))
        label = labels[classes[i]] if classes[i] < len(labels) else "Unknown"
        cv2.rectangle(image, (left, top), (right, bottom), (0, 255, 0), 2)
        cv2.putText(image, f"{label}: {scores[i]:.2f}", (left, top - 10),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 255, 0), 2)

# Display image with detections
cv2.imshow("Detected Objects", image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output :