School of Computer Science, UPES, Dehradun.

A

LABORATORY FILE

On

Deep Learning Lab

B.TECH. – 5th Semester

AUG. – NOV.- 2025.

Submitted to:

Dr. Rakesh Ranjan

Assistant Professor, UPES.

Submitted by:

Name: Aayush Grover

Batch: 12

SAP ID: 500119337

Roll No.: R2142230006

INDEX

Sr. No.	Experiment Name	Page No.	Signature

Experiment 2:
Implementing Neural Network Components from Scratch

AIM:
To develop a deep, fundamental understanding of neural network mechanics by building and training basic architecture from the ground up, without relying on high-level deep learning libraries.

Tasks:
1. **Single Neuron for a Linearly Separable Problem:**
- Implement a single neuron using Python.
- Define its components: weighted inputs, a bias term, and a step activation function.
- Train the neuron to function as an **AND gate** and verify its output against the complete truth table.

2. **Feedforward Neural Network (FFNN) for a Non-Linear Problem:**
- Extend the single neuron concept to a multi-neuron, multi-layer architecture (e.g., 2 input neurons, 2 hidden neurons, 1 output neuron).
- Implement the forward propagation mechanism to pass inputs through the network.
- Use pre-trained weights and biases to demonstrate that this network can solve the **XOR problem**, which is not linearly separable.

3. **Full Multilayer Perceptron (MLP) with Backpropagation:**
- Implement a complete MLP architecture with an input layer, at least one hidden layer, and an output layer.
- Implement the **backpropagation algorithm** to calculate gradients and update the network's weights and biases.
- Incorporate different activation functions (e.g., Sigmoid or ReLU) and a loss function (e.g., Mean Squared Error).
- Train the MLP on a simple dataset (e.g., the Iris dataset or a synthetic dataset) to perform classification.

CODES USED:

SINGLE NEURON FOR A LINEARLY SEPERABLE PROBLEM

[2]
```
import numpy as np
```

```
class SingleNeuron:
    def __init__(self, num_inputs, learning_rate=0.01):
        self.weights = np.random.rand(num_inputs)
        self.bias = np.random.rand()
        self.learning_rate = learning_rate

    def activate(self, x):
        return 1 if x >= 0 else 0  # Step activation function

    def predict(self, inputs):
        weighted_sum = np.dot(inputs, self.weights) + self.bias
        return self.activate(weighted_sum)

    def train(self, training_data, epochs):
        for epoch in range(epochs):
            for inputs, target in training_data:
                prediction = self.predict(inputs)
                error = target - prediction
                self.weights += self.learning_rate * error * inputs
                self.bias += self.learning_rate * error
```

[4]
```
# AND gate training data
and_data = [
    (np.array([0, 0]), 0),
    (np.array([0, 1]), 0),
    (np.array([1, 0]), 0),
    (np.array([1, 1]), 1)
]
```

[5]
```
# Train the neuron
neuron = SingleNeuron(num_inputs=2)
neuron.train(and_data, epochs=100)
```

[6]
```
# Verify the output against the AND gate truth table
print("AND Gate Verification:")
for inputs, target in and_data:
    prediction = neuron.predict(inputs)
    print(f"Input: {inputs}, Expected: {target}, Predicted: {prediction}")
```

```
AND Gate Verification:
Input: [0 0], Expected: 0, Predicted: 0
Input: [0 1], Expected: 0, Predicted: 0
Input: [1 0], Expected: 0, Predicted: 0
Input: [1 1], Expected: 1, Predicted: 1
```

FEEDFORWARD NEURAL NETWORK (FFNN) FOR A NON LINEAR PROBLEM

```python
import numpy as np

class FFNN:
    def __init__(self, input_size, hidden_size, output_size):
        # Initialize weights and biases for the hidden layer
        self.weights_hidden = np.random.rand(input_size, hidden_size)
        self.bias_hidden = np.random.rand(hidden_size)

        # Initialize weights and biases for the output layer
        self.weights_output = np.random.rand(hidden_size, output_size)
        self.bias_output = np.random.rand(output_size)

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def forward(self, inputs):
        # Calculate the output of the hidden layer
        hidden_layer_input = np.dot(inputs, self.weights_hidden) + self.bias_hidden
        hidden_layer_output = self.sigmoid(hidden_layer_input)

        # Calculate the output of the output layer
        output_layer_input = np.dot(hidden_layer_output, self.weights_output) + self.bias_output
        output = self.sigmoid(output_layer_input)

        return output
```

```python
# XOR gate data
xor_data = [
    (np.array([0, 0]), np.array([0])),
    (np.array([0, 1]), np.array([1])),
    (np.array([1, 0]), np.array([1])),
    (np.array([1, 1]), np.array([0]))
]
```

```python
# Create an FFNN with 2 input neurons, 2 hidden neurons, and 1 output neuron
ffnn = FFNN(input_size=2, hidden_size=2, output_size=1)
```

```python
# Pre-trained weights and biases that solve the XOR problem
# These values are often found through training, but for this step, we use pre-trained ones
ffnn.weights_hidden = np.array([[10, -10], [10, -10]])
ffnn.bias_hidden = np.array([-5, 15])
ffnn.weights_output = np.array([[10], [10]])
ffnn.bias_output = np.array([-15])
```

```python
# Verify the output against the XOR gate truth table
print("XOR Gate Verification:")
for inputs, target in xor_data:
    prediction = ffnn.forward(inputs)
    # Apply a threshold to the output to get a binary prediction (0 or 1)
    binary_prediction = 1 if prediction >= 0.5 else 0
    print(f"Input: {inputs}, Expected: {target[0]}, Predicted: {binary_prediction}")
```

```
XOR Gate Verification:
Input: [0 0], Expected: 0, Predicted: 0
Input: [0 1], Expected: 1, Predicted: 1
Input: [1 0], Expected: 1, Predicted: 1
Input: [1 1], Expected: 0, Predicted: 0
```

FULL MULTILAYER PERCEPTRON (MLP) WITH BACKPROPAGATION

2s

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_classification
```

0s

[13]

```
class MLP:
    def __init__(self, input_size, hidden_size, output_size, learning_rate=0.01):
        # Initialize weights and biases for the hidden layer
        self.weights_hidden = np.random.rand(input_size, hidden_size) * 0.01
        self.bias_hidden = np.zeros((1, hidden_size))

        # Initialize weights and biases for the output layer
        self.weights_output = np.random.rand(hidden_size, output_size) * 0.01
        self.bias_output = np.zeros((1, output_size))

        self.learning_rate = learning_rate

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

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

    def relu(self, x):
        return np.maximum(0, x)

    def relu_derivative(self, x):
        return (x > 0).astype(float)

    def mse_loss(self, y_true, y_pred):
        return np.mean((y_true - y_pred)**2)

    def mse_loss_derivative(self, y_true, y_pred):
        return 2 * (y_pred - y_true) / y_true.shape[0]
```

```
    def forward(self, inputs, activation='sigmoid'):
        # Hidden layer
        self.hidden_layer_input = np.dot(inputs, self.weights_hidden) + self.bias_hidden
        if activation == 'sigmoid':
            self.hidden_layer_output = self.sigmoid(self.hidden_layer_input)
        elif activation == 'relu':
            self.hidden_layer_output = self.relu(self.hidden_layer_input)
        else:
            raise ValueError("Invalid activation function")

        # Output layer
        self.output_layer_input = np.dot(self.hidden_layer_output, self.weights_output) + self.bias_output
        output = self.sigmoid(self.output_layer_input) # Sigmoid for output layer in this example

        return output

    def backward(self, inputs, y_true, y_pred, activation='sigmoid'):
        # Calculate the error for the output layer
        output_error = self.mse_loss_derivative(y_true, y_pred)
        output_delta = output_error * self.sigmoid_derivative(y_pred) # Assuming sigmoid in output layer

        # Calculate the error for the hidden layer
        hidden_layer_error = np.dot(output_delta, self.weights_output.T)
        if activation == 'sigmoid':
            hidden_layer_delta = hidden_layer_error * self.sigmoid_derivative(self.hidden_layer_output)
        elif activation == 'relu':
            hidden_layer_delta = hidden_layer_error * self.relu_derivative(self.hidden_layer_output)
        else:
             raise ValueError("Invalid activation function")

        # Update weights and biases
        self.weights_output -= self.learning_rate * np.dot(self.hidden_layer_output.T, output_delta)
        self.bias_output -= self.learning_rate * np.sum(output_delta, axis=0, keepdims=True)
        self.weights_hidden -= self.learning_rate * np.dot(inputs.T, hidden_layer_delta)
        self.bias_hidden -= self.learning_rate * np.sum(hidden_layer_delta, axis=0, keepdims=True)
```

```
    def train(self, X_train, y_train, epochs, activation='sigmoid'):
        for epoch in range(epochs):
            # Forward pass
            y_pred = self.forward(X_train, activation)

            # Backward pass
            self.backward(X_train, y_train, y_pred, activation)

            # Print loss every 1000 epochs
            if epoch % 1000 == 0:
                loss = self.mse_loss(y_train, y_pred)
                print(f"Epoch {epoch}, Loss: {loss}")
```

```
[14] # Generate a synthetic dataset
     X, y = make_classification(n_samples=100, n_features=2, n_informative=2, n_redundant=0, random_state=42)
     y = y.reshape(-1, 1) # Reshape y to be a column vector
```

```
[15] # Split data 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)
```

```
[16] # Create an MLP
     input_size = X_train.shape[1]
     hidden_size = 4 # You can adjust the hidden layer size
     output_size = y_train.shape[1]
     mlp = MLP(input_size, hidden_size, output_size, learning_rate=0.1)
```

```
[17] # Train the MLP
     mlp.train(X_train, y_train, epochs=10000, activation='relu')
```

```
Epoch 0, Loss: 0.24997663303281947
Epoch 1000, Loss: 0.022095361897587682
Epoch 2000, Loss: 0.014330059441492899
Epoch 3000, Loss: 0.012301431957095373
Epoch 4000, Loss: 0.011255235810793068
```

RESULT:

```
Epoch 4000, Loss: 0.011255335810793068
Epoch 5000, Loss: 0.010482429723992468
Epoch 6000, Loss: 0.009852360813976929
Epoch 7000, Loss: 0.009319303301216598
Epoch 8000, Loss: 0.008855306426027642
Epoch 9000, Loss: 0.00844618163289316
```

```
[18] # Evaluate the MLP on the test set
     y_pred_test = mlp.forward(X_test, activation='relu')
     binary_predictions_test = (y_pred_test >= 0.5).astype(int)

     print("\nTest Set Evaluation:")
     for i in range(len(X_test)):
         print(f"Input: {X_test[i]}, Expected: {y_test[i][0]}, Predicted: {binary_predictions_test[i][0]}")
```

```
Test Set Evaluation:
Input: [-2.00347738 -2.39955005], Expected: 0, Predicted: 0
Input: [1.09885263 1.25291095], Expected: 1, Predicted: 1
Input: [1.57233676 1.4991983 ], Expected: 1, Predicted: 1
Input: [-1.40210053 -1.72067112], Expected: 0, Predicted: 0
Input: [ 1.08266027 -0.98021491], Expected: 1, Predicted: 1
Input: [-0.38566776  0.01722979], Expected: 0, Predicted: 0
Input: [-1.4117586 -1.5332749], Expected: 0, Predicted: 0
Input: [-1.54446032 -1.51042899], Expected: 0, Predicted: 0
Input: [2.59123946 0.24472415], Expected: 1, Predicted: 1
Input: [0.55942643 2.38869353], Expected: 0, Predicted: 0
Input: [ 1.50575249 -0.38919817], Expected: 1, Predicted: 1
Input: [-0.05319823  1.85605469], Expected: 0, Predicted: 0
Input: [1.83991037 2.30450019], Expected: 1, Predicted: 1
Input: [-1.65830375 -1.57127256], Expected: 0, Predicted: 0
Input: [-0.78284083  1.20852705], Expected: 0, Predicted: 0
Input: [ 1.56070438 -0.42795824], Expected: 1, Predicted: 1
Input: [ 1.46830827 -0.48949935], Expected: 1, Predicted: 1
Input: [2.36867367 2.25661188], Expected: 1, Predicted: 1
Input: [-1.17762637 -1.20592943], Expected: 0, Predicted: 0
Input: [-2.05832072 -2.52343407], Expected: 0, Predicted: 0
```

Experiment 3
Application of a DL Framework for Classification

AIM:
To apply a high-level deep learning framework to solve a practical, application-specific classification problem, utilizing best practices for data handling, model training, and evaluation.

Tasks:
1. **Framework and Dataset Selection:**
o Choose a deep learning framework (e.g., PyTorch or TensorFlow/Keras) based on the findings from Experiment 1.
o Select a standard benchmark dataset, such as **MNIST** (for handwritten digits) or **Fashion-MNIST** (for clothing articles).
2. **Data Preprocessing:**
o Load the dataset and explore its properties.
o Preprocess the data: normalize pixel values to a range (e.g., 0 to 1), and one-hot encode the labels.
o Split the dataset into training, validation, and testing sets.
3. **Model Building and Training:**
o Design and build a sequential neural network model suitable for the dataset.
o Compile the model by defining a loss function (e.g., CategoricalCrossentropy), an optimizer (e.g., Adam), and evaluation metrics (e.g., accuracy).
o Train the model using the training data and validate its performance on the validation set. Plot the training/validation accuracy and loss curves.
4. **Evaluation:**
o Evaluate the final trained model on the unseen test set to measure its generalization performance.
o Generate a classification report and a confusion matrix to analyze the model's predictions for each class.

CODES USED:

```
[1] import tensorflow as tf
    from tensorflow.keras.datasets import mnist
    import numpy as np
    import matplotlib.pyplot as plt
    import torch
    import torch.nn as nn
    import torch.optim as optim
    import torchvision
    import torchvision.transforms as transforms
    import matplotlib.pyplot as plt
    import seaborn as sns
    from sklearn.metrics import confusion_matrix, classification_report
```

DATASET LOADING

```
[2] transform = transforms.Compose([transforms.ToTensor(),
                                    transforms.Normalize((0.5,), (0.5,))])

    train_dataset = torchvision.datasets.MNIST(root='./data', train=True, download=True, transform=transform)
    test_dataset = torchvision.datasets.MNIST(root='./data', train=False, download=True, transform=transform)

    train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=64, shuffle=True)
    test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=64, shuffle=False)
```

```
100%|██████████| 9.91M/9.91M [00:00<00:00, 55.4MB/s]
100%|██████████| 28.9k/28.9k [00:00<00:00, 1.64MB/s]
100%|██████████| 1.65M/1.65M [00:00<00:00, 14.2MB/s]
100%|██████████| 4.54k/4.54k [00:00<00:00, 3.30MB/s]
```

```
[3] # Load MNIST dataset
    (x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

PREPROCESSING

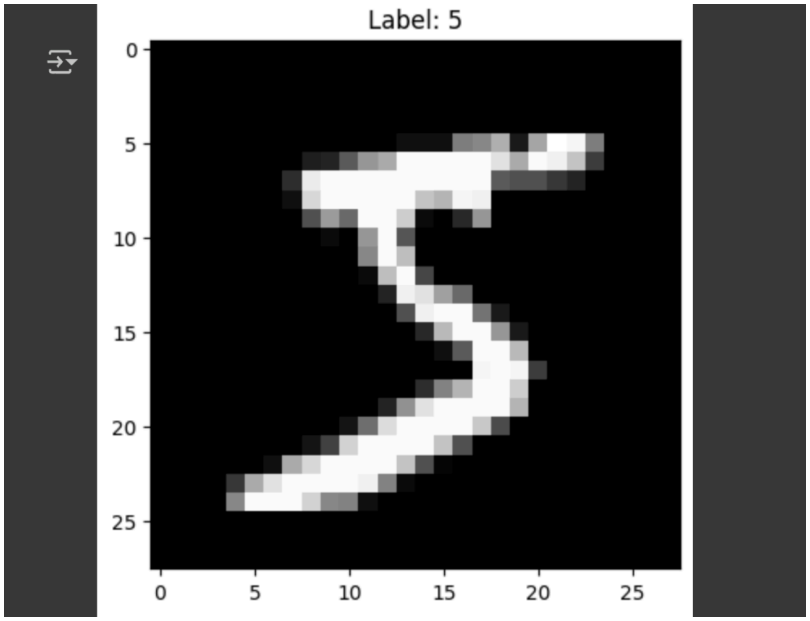
```
#Normalize pixel values + flatten images
x_train = x_train.reshape(-1, 784) / 255.0
x_test  = x_test.reshape(-1, 784)  / 255.0
```

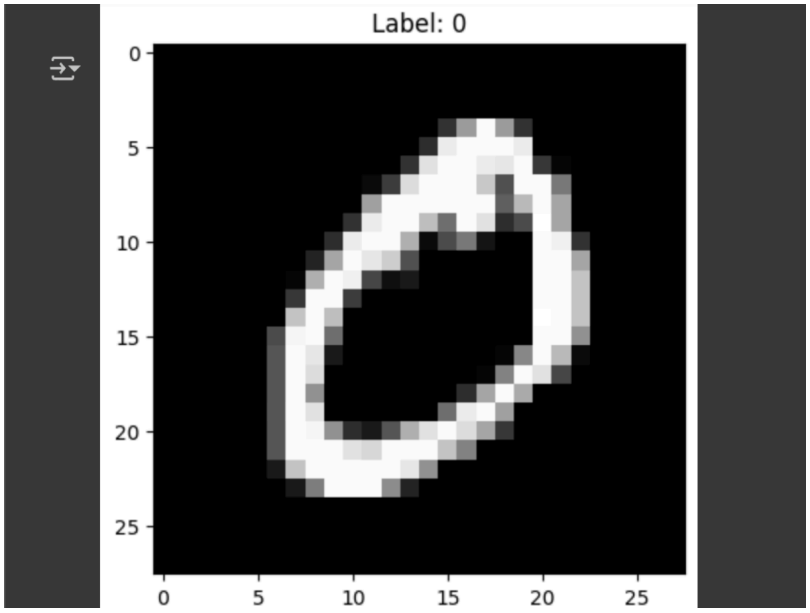
```
[5] # One-hot encode the labels (targets)
    y_train_onehot = tf.one_hot(y_train, depth=10)
    y_test_onehot  = tf.one_hot(y_test,  depth=10)
```

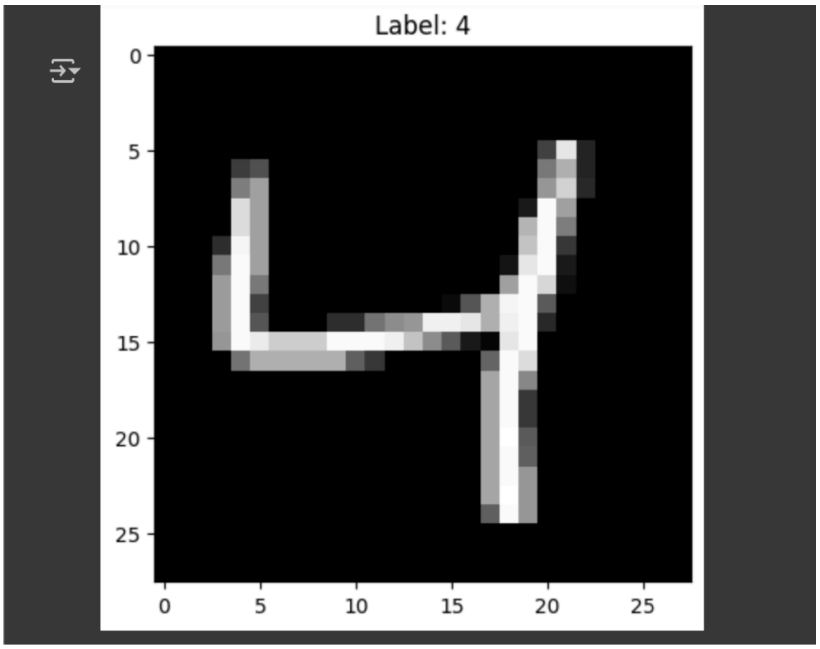
```
[6] #Sanity-check the feature matrix shapes
    print( x_train.shape)
    print(x_test.shape)
```

```
(60000, 784)
(10000, 784)
```

```
[7] for i in range(3):
        plt.imshow(x_train[i].reshape(28,28), cmap="gray")
        plt.title(f"Label: {y_train[i]}")
        plt.show()
```


Label: 5
0
5
10
15
20
25
0
5
10
15
20
25


Label: 0
0
5
10
15
20
25
0
5
10
15
20
25


Label: 4
0
5
10
15
20
25
0
5
10
15
20
25

TRAIN THE MODEL

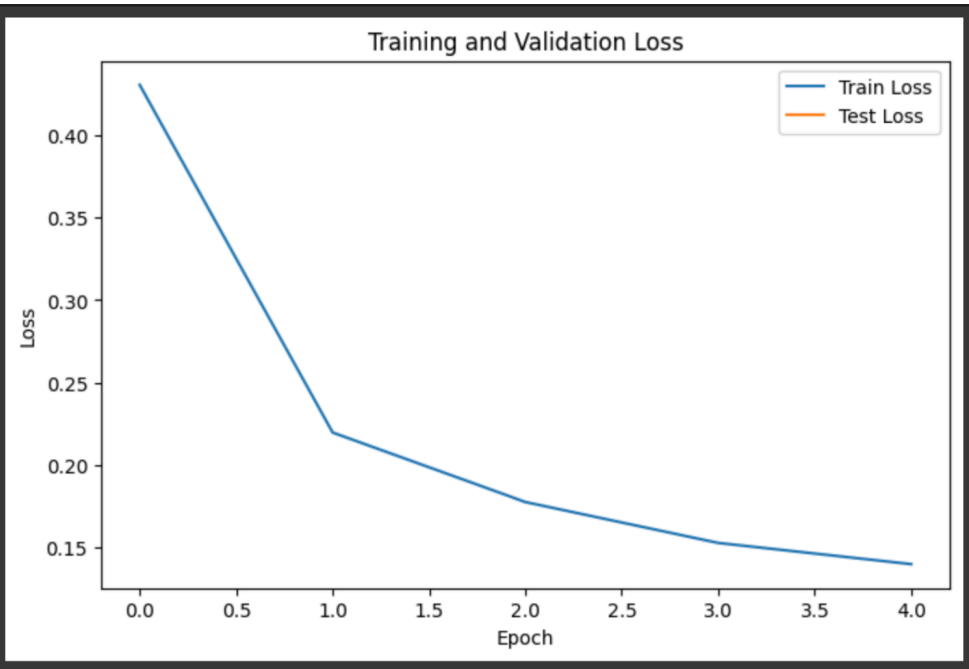
```
[8]  class NeuralNet(nn.Module):
         def __init__(self):
             super(NeuralNet, self).__init__()
             self.fc1 = nn.Linear(28*28, 128)
             self.fc2 = nn.Linear(128, 64)
             self.fc3 = nn.Linear(64, 10)
             self.relu = nn.ReLU()
             self.dropout = nn.Dropout(0.2)

         def forward(self, x):
             x = x.view(-1, 28*28)
             x = self.relu(self.fc1(x))
             x = self.dropout(x)
             x = self.relu(self.fc2(x))
             x = self.fc3(x)
             return x

     model = NeuralNet()
```

```
     criterion = nn.CrossEntropyLoss()
     optimizer = optim.Adam(model.parameters(), lr=0.001)
```

```
[10] num_epochs = 5
     train_losses, test_losses = [], []

     for epoch in range(num_epochs):
         model.train()
         running_loss = 0.0
         for images, labels in train_loader:
             outputs = model(images)
             loss = criterion(outputs, labels)

             optimizer.zero_grad()
             loss.backward()
             optimizer.step()

             running_loss += loss.item()
         train_losses.append(running_loss/len(train_loader))
```

PLOT LOSS CURVES

```
[11] plt.figure(figsize=(8,5))
     plt.plot(train_losses, label='Train Loss')
     plt.plot(test_losses, label='Test Loss')
     plt.xlabel('Epoch')
     plt.ylabel('Loss')
     plt.legend()
     plt.title('Training and Validation Loss')
     plt.show()
```



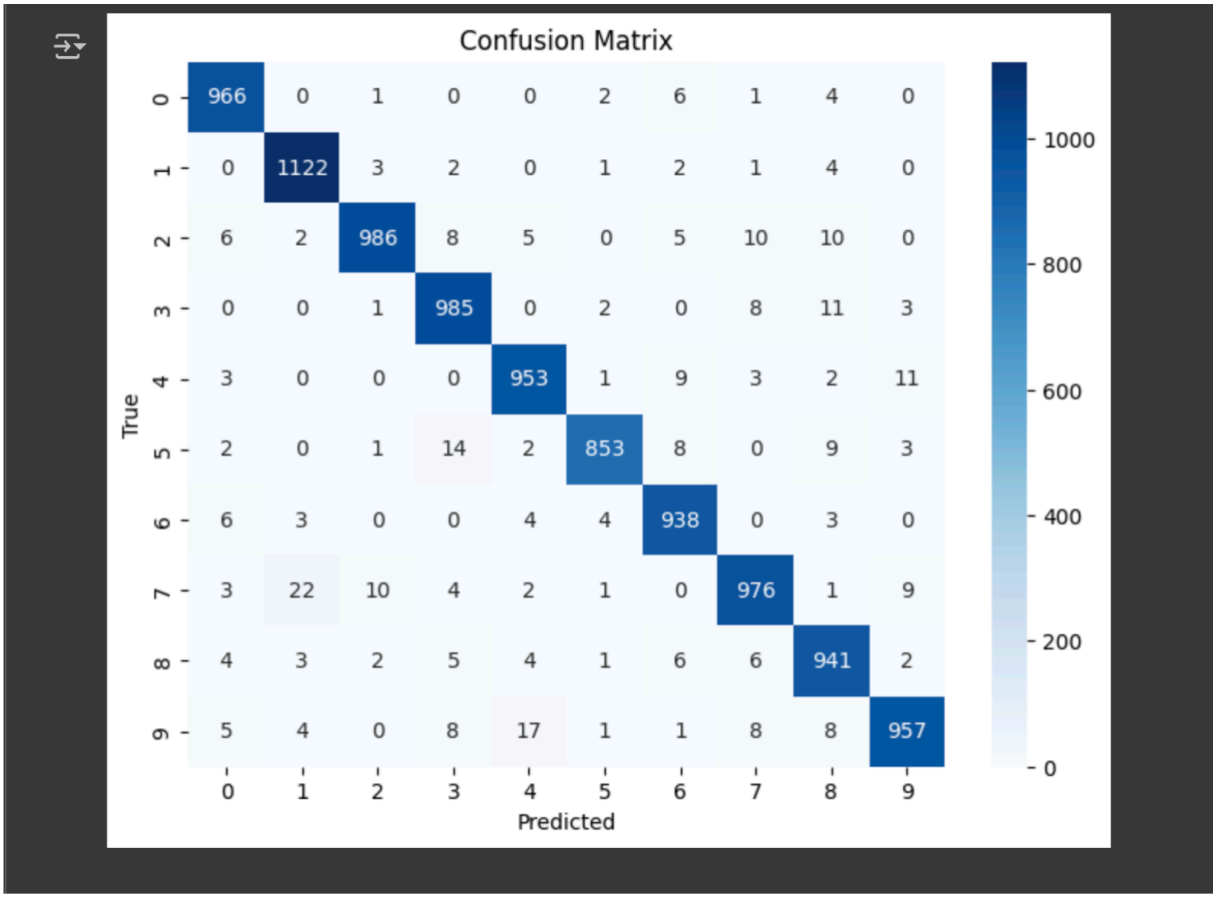
ACCURACY

```
[12] all_preds, all_labels = [], []
     model.eval()
     with torch.no_grad():
         correct, total = 0, 0
         for images, labels in test_loader:
             outputs = model(images)
             _, predicted = torch.max(outputs.data, 1)
             total += labels.size(0)
             correct += (predicted == labels).sum().item()
             all_preds.extend(predicted.numpy())
             all_labels.extend(labels.numpy())

     print(f"Test Accuracy: {100 * correct / total:.2f}%")
```

```
Test Accuracy: 96.77%
```

CONFUSION MATRIX

```
cm = confusion_matrix(all_labels, all_preds)
plt.figure(figsize=(8,6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```



RESULT:

PERFORMANCE METRICS

```
print(classification_report(all_labels, all_preds))
```

```
              precision    recall  f1-score   support

           0       0.97      0.99      0.98       980
           1       0.97      0.99      0.98      1135
           2       0.98      0.96      0.97      1032
           3       0.96      0.98      0.97      1010
           4       0.97      0.97      0.97       982
           5       0.98      0.96      0.97       892
           6       0.96      0.98      0.97       958
           7       0.96      0.95      0.96      1028
           8       0.95      0.97      0.96       974
           9       0.97      0.95      0.96      1009

    accuracy                           0.97     10000
   macro avg       0.97      0.97      0.97     10000
weighted avg       0.97      0.97      0.97     10000
```

Experiment 4:
Image Classification using Pretrained Models and Transfer Learning

AIM:
To leverage the power of **transfer learning** by using a state-of-the-art, pretrained convolutional neural network to perform a complex image classification task. This experiment will demonstrate how to adapt a model trained on a large-scale dataset (like ImageNet) to a new, specific dataset, thereby achieving high accuracy with significantly less training time and data.

Tasks:
1. **Understand Transfer Learning Concepts:**
   - Research and summarize the core ideas behind transfer learning.
   - Differentiate between the two main strategies:
     - **Feature Extraction:** Using the pretrained model's convolutional base to extract features from new images and training only a new, smaller classifier on top of it. This is fast and effective for smaller datasets.
     - **Fine-Tuning:** Freezing the initial layers of the pretrained model (which learns general features like edges and textures) and retraining the later, more specialized layers along with a new classifier. This can yield higher accuracy if you have a larger dataset.
2. **Select a Pretrained Model and Dataset:**
   - Choose a well-known pretrained model from a deep learning framework's library (e.g., Keras Applications). Popular choices include EfficientNet, Inception, Xception, **ResNet50**, or **MobileNetV2**.
   - Select an image dataset for your classification task. A good choice would be the "Cats vs. Dogs" dataset or the "CIFAR-10" dataset, as they are complex enough to benefit from transfer learning but manageable for a lab experiment.
3. **Implementation: Feature Extraction:**
   - Load your chosen pretrained model, making sure to instantiate it **without its final classification layer** (include_top=False) and with weights pretrained on **ImageNet**.
   - Freeze the layers of the loaded convolutional base to prevent their weights from being updated during training. This is a crucial step in feature extraction.
   - Add a new classification head to the model. This typically consists of a Flatten layer followed by one or more Dense layers, with a final Dense layer using a softmax activation function for multi-class classification or sigmoid for binary classification.
   - Compile and train your model, but **only the new classification layers should be trained**.
4. **Implementation: Fine-Tuning (Optional but Recommended):**
   - After the initial training of the classifier head, **unfreeze** some of the later layers of the pretrained base model.
   - Re-compile the model with a very low learning rate to prevent catastrophic forgetting (i.e., erasing the learned features of the pretrained model).
   - Continue training the model. This will subtly adjust the higher-level features in the pretrained model to be more specific to your new dataset.

5. **Evaluation and Analysis:**

- Evaluate the final model on your test dataset.
- Report the final accuracy and loss.
- Write a brief report comparing the results of the feature extraction approach (and fine-tuning, if performed) to the performance of the simple CNN you might have built from scratch in a previous experiment. Discuss why transfer learning is such an effective and widely used technique in computer vision.

CODES USED:

```
[5]  # Use ImageDataGenerator for data loading and augmentation
     # Rescale pixel values to be between 0 and 1
     train_datagen = ImageDataGenerator(rescale=1./255, validation_split=0.2)
```

```
[6]  # Load training data
     train_ds = train_datagen.flow_from_directory(
         extract_path,
         target_size=(img_height, img_width),
         batch_size=batch_size,
         class_mode='categorical',
         subset='training')
```

```
Found 2400 images belonging to 1 classes.
```

```
[7]  # Load validation data
     val_ds = train_datagen.flow_from_directory(
         extract_path,
         target_size=(img_height, img_width),
         batch_size=batch_size,
         class_mode='categorical',
         subset='validation')

     print("Data loaded and preprocessed successfully.")
```

```
Found 600 images belonging to 1 classes.
Data loaded and preprocessed successfully.
```

SELECT A PRETRAINED MODEL AND DATASET

```
[8] from tensorflow.keras.applications import MobileNetV2
    from tensorflow.keras.layers import Dense, GlobalAveragePooling2D
    from tensorflow.keras.models import Model
```

```
[9] # Load the MobileNetV2 model pretrained on ImageNet, without the top classification layer
    base_model = MobileNetV2(weights='imagenet', include_top=False, input_shape=(img_height, img_width, 3))
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/mobilenet_v2/mobilenet
9406464/9406464 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

```
# Add a global average pooling layer
x = base_model.output
x = GlobalAveragePooling2D()(x)
```

```
[11] # Add a dense layer for classification (assuming binary classification for now, adjust if needed)
     # You need to determine the number of classes in your dataset
     num_classes = len(train_ds.class_indices) # Get the number of classes from the training data generator
     predictions = Dense(num_classes, activation='softmax')(x)
```

```
[12] # Create the new model
     model = Model(inputs=base_model.input, outputs=predictions)
```

```
# Freeze the layers of the base model
for layer in base_model.layers:
    layer.trainable = False
```

```
# Compile the model
model.compile(optimizer='adam',
              loss='categorical_crossentropy', # Use categorical_crossentropy for one-hot encoded labels
              metrics=['accuracy'])

model.summary()
```

Model: "functional"

Layer (type)	Output Shape	Param #	Connected to
input_layer (InputLayer)	(None, 128, 128, 3)	0	-
Conv1 (Conv2D)	(None, 64, 64, 32)	864	input_layer[0][0]
bn_Conv1 (BatchNormalizatio…	(None, 64, 64, 32)	128	Conv1[0][0]
Conv1_relu (ReLU)	(None, 64, 64, 32)	0	bn_Conv1[0][0]
expanded_conv_dept… (DepthwiseConv2D)	(None, 64, 64, 32)	288	Conv1_relu[0][0]
expanded_conv_dept… (BatchNormalizatio…	(None, 64, 64, 32)	128	expanded_conv_de…
expanded_conv_dept… (ReLU)	(None, 64, 64, 32)	0	expanded_conv_de…
expanded_conv_proj… (Conv2D)	(None, 64, 64, 16)	512	expanded_conv_de…

block_16_expand_BN (BatchNormalizatio…	(None, 4, 4, 960)	3,840	block_16_expand[…
block_16_expand_re… (ReLU)	(None, 4, 4, 960)	0	block_16_expand_…
block_16_depthwise (DepthwiseConv2D)	(None, 4, 4, 960)	8,640	block_16_expand_…
block_16_depthwise… (BatchNormalizatio…	(None, 4, 4, 960)	3,840	block_16_depthwi…
block_16_depthwise… (ReLU)	(None, 4, 4, 960)	0	block_16_depthwi…
block_16_project (Conv2D)	(None, 4, 4, 320)	307,200	block_16_depthwi…
block_16_project_BN (BatchNormalizatio…	(None, 4, 4, 320)	1,280	block_16_project…
Conv_1 (Conv2D)	(None, 4, 4, 1280)	409,600	block_16_project…
Conv_1_bn (BatchNormalizatio…	(None, 4, 4, 1280)	5,120	Conv_1[0][0]
out_relu (ReLU)	(None, 4, 4, 1280)	0	Conv_1_bn[0][0]
global_average_poo… (GlobalAveragePool…	(None, 1280)	0	out_relu[0][0]
dense (Dense)	(None, 1)	1,281	global_average_p…

Total params: 2,259,265 (8.62 MB)
Trainable params: 1,281 (5.00 KB)
Non-trainable params: 2,257,984 (8.61 MB)

IMPLIMENTATAION : FEATURE EXTRACTION

```
[16] # Train the model
     epochs = 5 # You can adjust the number of epochs
     history = model.fit(
         train_ds,
         epochs=epochs,
         validation_data=val_ds
     )

     print("Feature extraction model trained successfully.")
```

```
Epoch 1/5
75/75 ━━━━━━━━━━━━━━━━━━━━ 33s 440ms/step - accuracy: 1.0000 - loss: 0.0000e+00 - val_accuracy: 1.0000 - val_loss: 0.0000e+00
Epoch 2/5
75/75 ━━━━━━━━━━━━━━━━━━━━ 34s 455ms/step - accuracy: 1.0000 - loss: 0.0000e+00 - val_accuracy: 1.0000 - val_loss: 0.0000e+00
Epoch 3/5
75/75 ━━━━━━━━━━━━━━━━━━━━ 34s 458ms/step - accuracy: 1.0000 - loss: 0.0000e+00 - val_accuracy: 1.0000 - val_loss: 0.0000e+00
Epoch 4/5
75/75 ━━━━━━━━━━━━━━━━━━━━ 33s 436ms/step - accuracy: 1.0000 - loss: 0.0000e+00 - val_accuracy: 1.0000 - val_loss: 0.0000e+00
Epoch 5/5
75/75 ━━━━━━━━━━━━━━━━━━━━ 33s 446ms/step - accuracy: 1.0000 - loss: 0.0000e+00 - val_accuracy: 1.0000 - val_loss: 0.0000e+00
Feature extraction model trained successfully.
```

IMPLIMENTATAION : FINE-TUNING

17s

```
# Unfreeze some layers of the base model for fine-tuning
# It's common to unfreeze the later layers
# Let's unfreeze the last few blocks of MobileNetV2
# You might need to experiment to find the optimal number of layers to unfreeze
for layer in base_model.layers[-30:]: # Unfreeze the last 30 layers as an example
    layer.trainable = True

# Re-compile the model with a lower learning rate
from tensorflow.keras.optimizers import Adam

model.compile(optimizer=Adam(learning_rate=0.0001), # Use a low learning rate
              loss='categorical_crossentropy',
              metrics=['accuracy'])

model.summary()
```

Model: "functional"

Layer (type)	Output Shape	Param #	Connected to
input_layer (InputLayer)	(None, 128, 128, 3)	0	-
Conv1 (Conv2D)	(None, 64, 64, 32)	864	input_layer[0][0]
bn_Conv1 (BatchNormalizatio…	(None, 64, 64, 32)	128	Conv1[0][0]
Conv1_relu (ReLU)	(None, 64, 64, 32)	0	bn_Conv1[0][0]
expanded_conv_dept… (DepthwiseConv2D)	(None, 64, 64, 32)	288	Conv1_relu[0][0]
expanded_conv_dept… (BatchNormalizatio	(None, 64, 64, 32)	128	expanded_conv_de…

block_16_expand (Conv2D)	(None, 4, 4, 960)	153,600	block_15_add[0][…
block_16_expand_BN (BatchNormalizatio…	(None, 4, 4, 960)	3,840	block_16_expand[…
block_16_expand_re… (ReLU)	(None, 4, 4, 960)	0	block_16_expand_…
block_16_depthwise (DepthwiseConv2D)	(None, 4, 4, 960)	8,640	block_16_expand_…
block_16_depthwise… (BatchNormalizatio…	(None, 4, 4, 960)	3,840	block_16_depthwi…
block_16_depthwise… (ReLU)	(None, 4, 4, 960)	0	block_16_depthwi…
block_16_project (Conv2D)	(None, 4, 4, 320)	307,200	block_16_depthwi…
block_16_project_BN (BatchNormalizatio…	(None, 4, 4, 320)	1,280	block_16_project…
Conv_1 (Conv2D)	(None, 4, 4, 1280)	409,600	block_16_project…
Conv_1_bn (BatchNormalizatio…	(None, 4, 4, 1280)	5,120	Conv_1[0][0]
out_relu (ReLU)	(None, 4, 4, 1280)	0	Conv_1_bn[0][0]
global_average_poo… (GlobalAveragePool…	(None, 1280)	0	out_relu[0][0]
dense (Dense)	(None, 1)	1,281	global_average_p…

Total params: 2,259,265 (8.62 MB)
Trainable params: 1,527,681 (5.83 MB)
Non-trainable params: 731,584 (2.79 MB)

RESULT:

EVALUATION AND ANALYSIS

```
[18] # Evaluate the fine-tuned model on the validation data
     loss, accuracy = model.evaluate(val_ds)

     print(f"Validation Loss: {loss:.4f}")
     print(f"Validation Accuracy: {accuracy:.4f}")
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/ops/nn.py:944: UserWarning: You are using a softmax over axis -1 of a tensor of shape
  warnings.warn(
/usr/local/lib/python3.12/dist-packages/keras/src/losses/losses.py:33: SyntaxWarning: In loss categorical_crossentropy, expected y_pred.
  return self.fn(y_true, y_pred, **self._fn_kwargs)
19/19 ━━━━━━━━━━━━━━━━━━━━ 9s 325ms/step - accuracy: 1.0000 - loss: 0.0000e+00
Validation Loss: 0.0000
Validation Accuracy: 1.0000
```