SWE-532: Deep Learning
Lab Practical File

Submitted towards the partial fulfilment
of the requirements of the award of the degree of

Master of Technology
In
Software Engineering

Submitted To:-

Dr. Sonika Dahiya
Assistant Professor
Department of Software Engineering

Submitted By:-

Aman Chauhan (24/SWE/08)
II SEM, I YEAR

Delhi Technological University
(FORMERLY Delhi College of Engineering)
SHAHBAD DAULATPUR, BAWANA ROAD,DELHI – 110042

April, 2025

INDEX

S.No.	Experiment Name	Date	Signature
1	Implementation of ANN model on MNIST dataset without hidden layer.		
2	Implementation of ANN model on MNIST dataset with hidden layer.		
3	Implementation of CNN model on MNIST dataset		
4	Read about the CIFAR-10 dataset, implementing a simple CNN with 2 hidden layers. Show the accuracy using confusion matrix.		
5	Write a program to implement k-Nearest Neighbour algorithm to classify any dataset of your choice. Use cross validation and following metrics for performance evaluation: a. Accuracy b. Misclassification Rate c. Sensitivity d. Specificity e. Precision f. F-Score		
6	Write a python code for the following 5 activation functions: a) Step b. Sigmoid c. tanh d. ReLU (rectified linear unit) e. Leaky ReLU		
7	Implement a sliding window approach to classify regions of an image using a pre-trained CNN on CIFAR-10 dataset.		
8	Use Faster R-CNN with pre-trained weights for object detection on simple datasets like Pascal VOC.		
9	To perform fast and accurate object detection using the YOLOv5 model. Run inference on sample images and videos.		
10	To extract named entities from text using a pre-trained BERT model fine-tuned for NER.		

EXPERIMENT- 1

Aim - Implementation of ANN model on MNIST dataset without hidden layer.

Introduction - ANN is a computational framework modeled after the human brain's structure of interconnected neurons. It is built to identify patterns and relationships within data by learning from examples, mimicking the way humans learn.Training an ANN model involves fine-tuning the network's weights and biases to reduce prediction errors through a process called backpropagation, guided by a loss function (such as cross-entropy for classification tasks.This model directly maps the input features to the output classes using only a single weight matrix and a softmax activation function at the output layer. Despite its simplicity, such a model provides a valuable baseline to understand the fundamentals of supervised learning, linear classification, and the limitations of shallow networks. By training the ANN on the MNIST training set and evaluating it on the test set, we can observe how well the model performs in distinguishing digit patterns without the complexity of deeper layers. This experiment helps reinforce foundational concepts like forward propagation, loss calculation using cross-entropy, and weight updates through gradient descent, all within the context of a minimal neural architecture.

Description –

The MNIST dataset (Modified National Institute of Standards and Technology) is a benchmark dataset in the field of machine learning, composed of 70,000 grayscale images of handwritten digits ranging from 0 to 9. Each image is 28x28 pixels, flattened into a 784-dimensional input vector. In this experiment, we implement a basic Artificial Neural Network (ANN) model without any hidden layers, essentially creating a single-layer perceptron.

Code:

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
# Load the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
# Preprocess the data
x_train = x_train.reshape(-1, 28 * 28).astype("float32") / 255.0  # Flatten
and normalize
x_test = x_test.reshape(-1, 28 * 28).astype("float32") / 255.0    # Flatten
and normalize
y_train = to_categorical(y_train, 10)  # One-hot encode the labels
y_test = to_categorical(y_test, 10)
# Build the ANN model
model = Sequential([
   Dense(10, input_shape=(28 * 28,), activation="softmax")  # Output layer
with 10 units (no hidden layer)
])
# Compile the model
model.compile(optimizer=Adam(),
              loss="categorical_crossentropy",
              metrics=["accuracy"])
model.summary()
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 10)	7,850

```
Total params: 7,850 (30.66 KB)
Trainable params: 7,850 (30.66 KB)
Non-trainable params: 0 (0.00 B)
```

#

Train the model

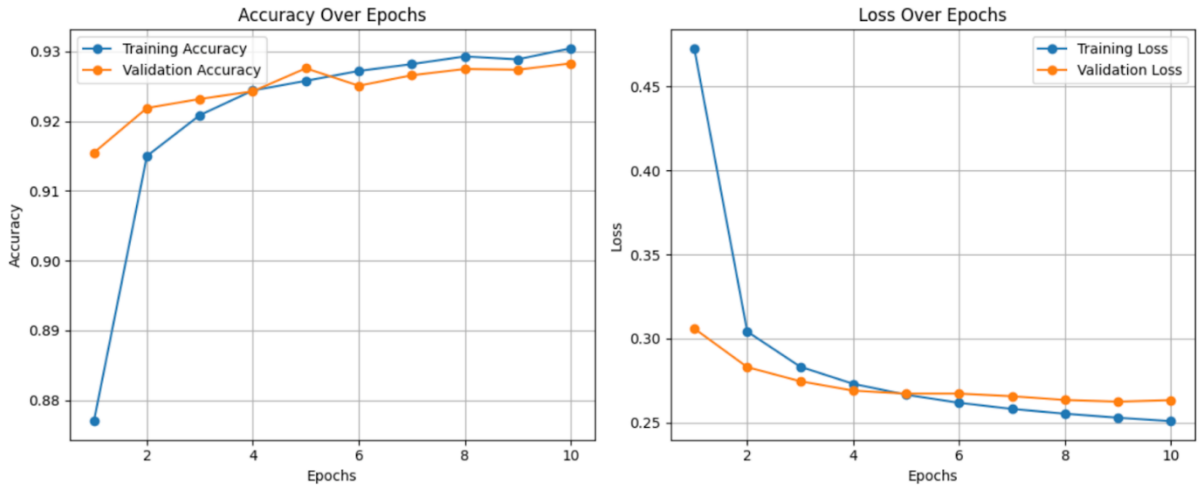
```
history = model.fit(x_train, y_train, epochs=10, batch_size=32,
validation_data=(x_test, y_test))
```

```
Epoch 1/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 3ms/step - accuracy: 0.8113 - loss: 0.7308 - val_accuracy: 0.9155 - val_loss: 0.3060
Epoch 2/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9122 - loss: 0.3141 - val_accuracy: 0.9219 - val_loss: 0.2830
Epoch 3/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - accuracy: 0.9210 - loss: 0.2841 - val_accuracy: 0.9232 - val_loss: 0.2746
Epoch 4/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9247 - loss: 0.2729 - val_accuracy: 0.9243 - val_loss: 0.2691
Epoch 5/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9275 - loss: 0.2607 - val_accuracy: 0.9276 - val_loss: 0.2673
Epoch 6/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9282 - loss: 0.2579 - val_accuracy: 0.9251 - val_loss: 0.2673
Epoch 7/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - accuracy: 0.9284 - loss: 0.2542 - val_accuracy: 0.9266 - val_loss: 0.2657
Epoch 8/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9284 - loss: 0.2624 - val_accuracy: 0.9275 - val_loss: 0.2635
Epoch 9/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9306 - loss: 0.2450 - val_accuracy: 0.9274 - val_loss: 0.2625
Epoch 10/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9304 - loss: 0.2465 - val_accuracy: 0.9283 - val_loss: 0.2634
```

```
# Evaluate the model
test_loss, test_accuracy = model.evaluate(x_test, y_test)
print(f"Test Accuracy: {test_accuracy:.4f}")
import matplotlib.pyplot as plt
def plot_training_curves(history):
    # Extract data from history object
    accuracy = history.history['accuracy']
    val_accuracy = history.history.get('val_accuracy', None)
    loss = history.history['loss']
    val_loss = history.history.get('val_loss', None)

    epochs = range(1, len(accuracy) + 1)

    # Plot training and validation accuracy
    plt.figure(figsize=(12, 5))

    # Subplot for accuracy
    plt.subplot(1, 2, 1)
    plt.plot(epochs, accuracy, marker='o', label='Training Accuracy')
    if val_accuracy is not None:
        plt.plot(epochs, val_accuracy, marker='o', label='Validation
Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.title('Accuracy Over Epochs')
    plt.legend()
    plt.grid(True)

    # Subplot for loss
    plt.subplot(1, 2, 2)
    plt.plot(epochs, loss, marker='o', label='Training Loss')
    if val_loss is not None:
        plt.plot(epochs, val_loss, marker='o', label='Validation Loss')
    plt.xlabel('Epochs')
```

```
    plt.ylabel('Loss')
    plt.title('Loss Over Epochs')
    plt.legend()
    plt.grid(True)

    # Show the plots
    plt.tight_layout()
    plt.show()
# Example usage:
plot_training_curves(history)
```

Output:



EXPERIMENT- 2

Aim- ANN on MNIST with Hidden Layer

Introduction – In general, An Artificial Neural Network (ANN) is a computer model based on the human brain's neuronal network. It is set up to notice patterns and relationships between information by being taught examples, similar to the way humans learn.

Fundamentally, an ANN is a set of interconnected nodes (neurons) placed in layers:

- Input layer: Handles raw input data (e.g., 784 values for 28×28 MNIST images).
- Hidden layer(s): Has multiple neurons and Each neuron calculates weighted sum of inputs, adds bias, and applies a non-linear activation function(e.g., ReLU, tanh). It also Allows the model to learn complex, non-linear mappings.
- Output layer: Produces the final prediction or classification. Often employs softmax activation for multi-class classification and Outputs a probability distribution over all classes.

Code:

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Dropout # Import
Dropout here
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
# Load the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
# Preprocess the data
x_train = x_train.reshape(-1, 28 * 28).astype("float32") / 255.0
# Flatten and normalize
x_test = x_test.reshape(-1, 28 * 28).astype("float32") / 255.0
# Flatten and normalize
y_train = to_categorical(y_train, 10)  # One-hot encode the labels
y_test = to_categorical(y_test, 10)
# Build an ANN with hidden layer
# Define the model
model = Sequential([
   Dense(512, activation="relu", input_shape=(28 * 28,)),# Input layer
   Dropout(0.2),  # Dropout for regularization
   Dense(256, activation="relu"),  # Hidden layer
   Dropout(0.2),
   Dense(128, activation="relu"),  # Hidden layer
   Dense(10, activation="softmax")  # Output layer (10 classes)
])
# Compile the model
model.compile(optimizer="adam",
              loss="categorical_crossentropy",
              metrics=["accuracy"])
model.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 512)	401,920
dropout (Dropout)	(None, 512)	0
dense_2 (Dense)	(None, 256)	131,328
dropout_1 (Dropout)	(None, 256)	0
dense_3 (Dense)	(None, 128)	32,896
dense_4 (Dense)	(None, 10)	1,290

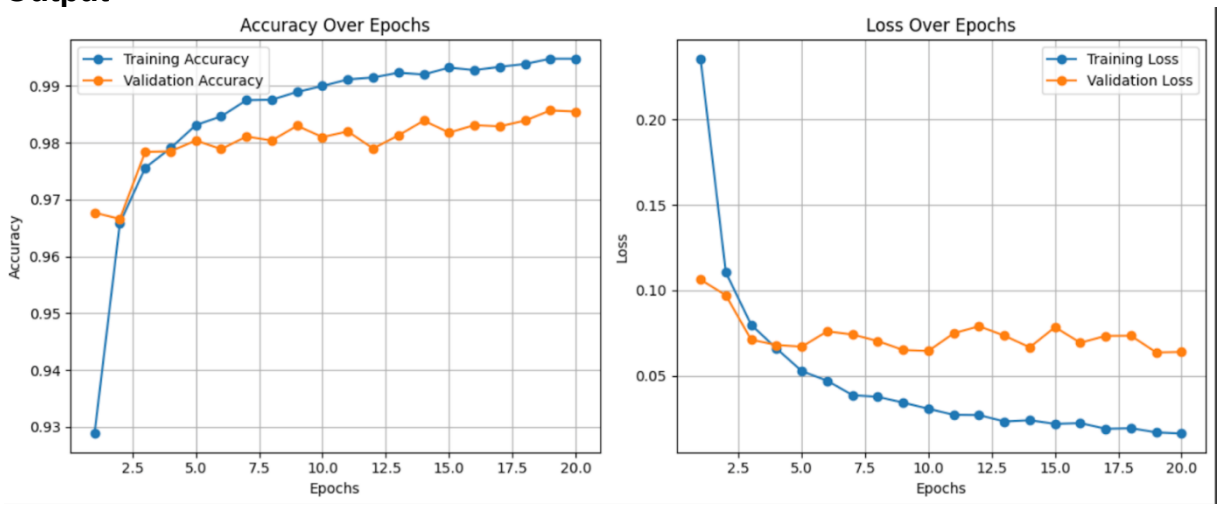
Total params: 567,434 (2.16 MB)
Trainable params: 567,434 (2.16 MB)
Non-trainable params: 0 (0.00 B)

```
# Train the model
history = model.fit(x_train, y_train, epochs=20, batch_size=128, validation_data=(x_test, y_test))
```

```
469/469 ━━━━━━━━━━━━━━━━━━━━ 8s 16ms/step - accuracy: 0.9848 - loss: 0.0476 - val_accuracy: 0.9804 -
Epoch 6/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 10s 16ms/step - accuracy: 0.9853 - loss: 0.0455 - val_accuracy: 0.9789 - val_loss: 0.0759
Epoch 7/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 11s 18ms/step - accuracy: 0.9883 - loss: 0.0357 - val_accuracy: 0.9811 - val_loss: 0.0741
Epoch 8/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 11s 19ms/step - accuracy: 0.9893 - loss: 0.0324 - val_accuracy: 0.9804 - val_loss: 0.0702
Epoch 9/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 9s 19ms/step - accuracy: 0.9892 - loss: 0.0322 - val_accuracy: 0.9830 - val_loss: 0.0650
Epoch 10/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 9s 17ms/step - accuracy: 0.9914 - loss: 0.0266 - val_accuracy: 0.9810 - val_loss: 0.0645
Epoch 11/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 11s 19ms/step - accuracy: 0.9921 - loss: 0.0250 - val_accuracy: 0.9820 - val_loss: 0.0748
Epoch 12/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 10s 19ms/step - accuracy: 0.9919 - loss: 0.0253 - val_accuracy: 0.9790 - val_loss: 0.0789
Epoch 13/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 9s 20ms/step - accuracy: 0.9912 - loss: 0.0246 - val_accuracy: 0.9813 - val_loss: 0.0734
Epoch 14/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 9s 19ms/step - accuracy: 0.9925 - loss: 0.0224 - val_accuracy: 0.9839 - val_loss: 0.0666
Epoch 15/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 9s 17ms/step - accuracy: 0.9944 - loss: 0.0180 - val_accuracy: 0.9818 - val_loss: 0.0784
Epoch 16/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 10s 17ms/step - accuracy: 0.9935 - loss: 0.0200 - val_accuracy: 0.9831 - val_loss: 0.0693
Epoch 17/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 9s 19ms/step - accuracy: 0.9937 - loss: 0.0183 - val_accuracy: 0.9829 - val_loss: 0.0732
Epoch 18/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 10s 19ms/step - accuracy: 0.9936 - loss: 0.0188 - val_accuracy: 0.9839 - val_loss: 0.0733
Epoch 19/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 9s 17ms/step - accuracy: 0.9953 - loss: 0.0145 - val_accuracy: 0.9857 - val_loss: 0.0635
Epoch 20/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 9s 19ms/step - accuracy: 0.9958 - loss: 0.0136 - val_accuracy: 0.9855 - val_loss: 0.0639
```

```
# Evaluate the model
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Test accuracy: {test_acc:.4f}")
plot_training_curves(history)
```

Output-



EXPERIMENT- 3

Aim - Implementation of CNN

Introduction- CNNs are the foundation of most modern computer vision applications and have achieved state-of-the-art performance in tasks like,

- Image classification (e.g., MNIST, CIFAR-10, ImageNet).
- Object detection (e.g., YOLO, Faster R-CNN).
- Image segmentation (e.g., U-Net, Mask R-CNN).
- Facial recognition, medical imaging, and more.

Description- Usually, Standard CNN includes the following major layers:

1. Convolutional Layer – It uses a group of learnable filters (kernels) on the input. Here, Each filter moves over the input image (or feature map) and does element-wise multiplication followed by addition. It also picks up spatial patterns like edges, textures, and shapes. The output is a feature map (also referred to as activation map).
2. Activation Function – It is typically used in ReLU (Rectified Linear Unit). It introduces non-linearity, assisting the model to learn intricate patterns.
3. Pooling Layer (Subsampling) – It reduces the spatial dimensions of the feature maps (width × height). It also assists in minimizing computation and preventing overfitting. The Max Pooling is typically used for obtaining the maximum value within each region.
4. Fully Connected Layer - Following some convolutional and pooling layers, high-level feature maps are flattened and input into one or more dense layers. Final output typically goes through a softmax layer for classification.
5. Output Layer – Here, the outputs prediction probabilities for every class. Example - For MNIST (0–9 digits), output layer consists of 10 units with softmax activation.

Code-

```
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
# Dividing the dataset into training and testing set
(x_train, y_train), (x_test, y_test) = mnist.load_data()
# Preprocess the data
x_train = x_train.reshape(-1, 28 * 28).astype("float32") / 255.0  # Flatten
and normalize
x_test = x_test.reshape(-1, 28 * 28).astype("float32") / 255.0    # Flatten
and normalize
y_train = to_categorical(y_train, 10)  # One-hot encode the labels
y_test = to_categorical(y_test, 10)
# Design a CNN
model = tf.keras.models.Sequential([
   tf.keras.layers.Reshape((28, 28, 1), input_shape=(784,)), #Input layer

   # Feature Extraction
   tf.keras.layers.Conv2D(filters=32,                    kernel_size=(3,3),
activation="relu"),
   tf.keras.layers.MaxPooling2D(pool_size=(2,2)),
   tf.keras.layers.Conv2D(filters=64,                    kernel_size=(3,3),
activation="relu"),
   tf.keras.layers.MaxPooling2D(pool_size=(2,2)),

   # Classification Head
   tf.keras.layers.Flatten(),
   tf.keras.layers.Dense(128, activation="relu"),
   tf.keras.layers.Dense(10, activation="softmax"),
```

```
])

model.compile(
    optimizer="adam",
    loss="categorical_crossentropy",
    metrics=['accuracy']
)
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
reshape (Reshape)	(None, 28, 28, 1)	0
conv2d (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_1 (Conv2D)	(None, 11, 11, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
flatten (Flatten)	(None, 1600)	0
dense (Dense)	(None, 128)	204,928
dense_1 (Dense)	(None, 10)	1,290

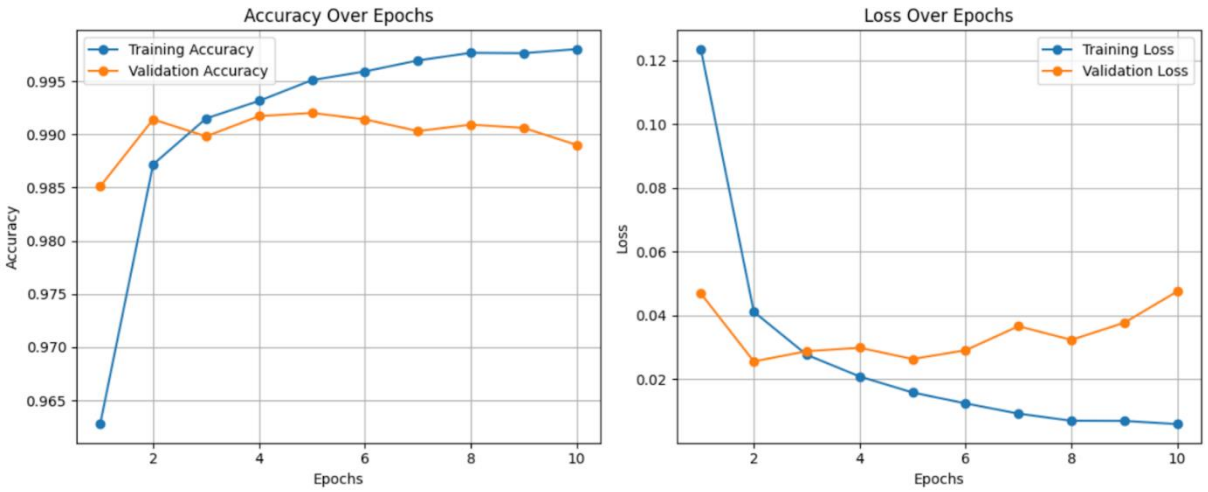
Total params: 225,034 (879.04 KB)
Trainable params: 225,034 (879.04 KB)
Non-trainable params: 0 (0.00 B)

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

```
Epoch 1/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 11s 4ms/step - accuracy: 0.9148 - loss: 0.2817 - val_accuracy: 0.9851 - val_loss: 0.0469
Epoch 2/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9870 - loss: 0.0414 - val_accuracy: 0.9914 - val_loss: 0.0256
Epoch 3/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9920 - loss: 0.0253 - val_accuracy: 0.9898 - val_loss: 0.0288
Epoch 4/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9936 - loss: 0.0193 - val_accuracy: 0.9917 - val_loss: 0.0299
Epoch 5/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - accuracy: 0.9962 - loss: 0.0134 - val_accuracy: 0.9920 - val_loss: 0.0264
Epoch 6/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.9965 - loss: 0.0104 - val_accuracy: 0.9914 - val_loss: 0.0291
Epoch 7/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 12s 5ms/step - accuracy: 0.9977 - loss: 0.0067 - val_accuracy: 0.9903 - val_loss: 0.0367
Epoch 8/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9984 - loss: 0.0052 - val_accuracy: 0.9909 - val_loss: 0.0324
Epoch 9/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - accuracy: 0.9981 - loss: 0.0057 - val_accuracy: 0.9906 - val_loss: 0.0378
Epoch 10/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - accuracy: 0.9983 - loss: 0.0049 - val_accuracy: 0.9890 - val_loss: 0.0476
```

```
# Evaluate the model
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Test accuracy: {test_acc:.4f}")
plot_training_curves(history)
```

Output -


Accuracy Over Epochs
Training Accuracy
Validation Accuracy
Accuracy
Epochs
0.965
0.970
0.975
0.980
0.985
0.990
0.995
2
4
6
8
10
Loss Over Epochs
Training Loss
Validation Loss
Loss
Epochs
0.02
0.04
0.06
0.08
0.10
0.12
2
4
6
8
10

EXPERIMENT- 4

Aim - Read about the CIFAR-10 dataset, implementing a simple CNN with 2 hidden layers. Show the accuracy using the confusion matrix.

Introduction -

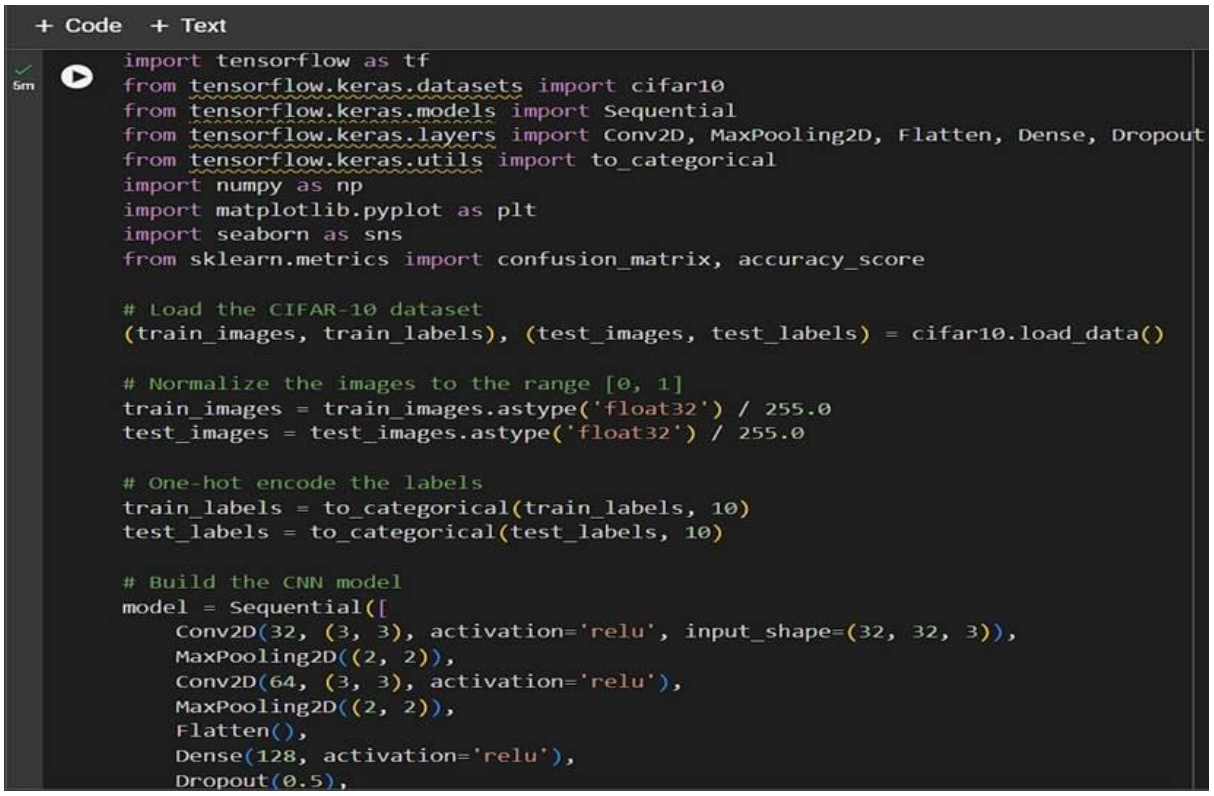
In this experiment, a Convolutional Neural Network (CNN) is implemented to classify these images. CNNs are a type of deep learning architecture designed specifically for processing grid-like data such as images. The model used in this experiment includes two hidden convolutional layers followed by pooling and fully connected layers. These convolutional layers help the model automatically extract hierarchical features from raw image pixels, capturing edges, textures, and complex patterns essential for accurate classification. After training the CNN on the CIFAR-10 training set, the model's performance is evaluated on the test set using a confusion matrix, which provides detailed insights into classification accuracy for each class. The confusion matrix reveals not just the overall accuracy but also highlights specific classes that are often misclassified, offering a deeper understanding of the model's strengths and weaknesses. This experiment demonstrates how even a relatively simple CNN can achieve reasonable performance on image classification tasks and serves as a foundation for understanding more complex architectures and datasets.

Dataset Description -

CIFAR-10: A Classic Dataset for Image Recognition

The CIFAR-10 dataset is a widely used benchmark in the field of machine learning and computer vision, consisting of 60,000 color images divided into 10 distinct classes such as airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. Each image is of 32x32 pixels and is labeled with one of the 10 categories. The dataset is split into 50,000 training images and 10,000 test images, enabling robust training and evaluation of models.

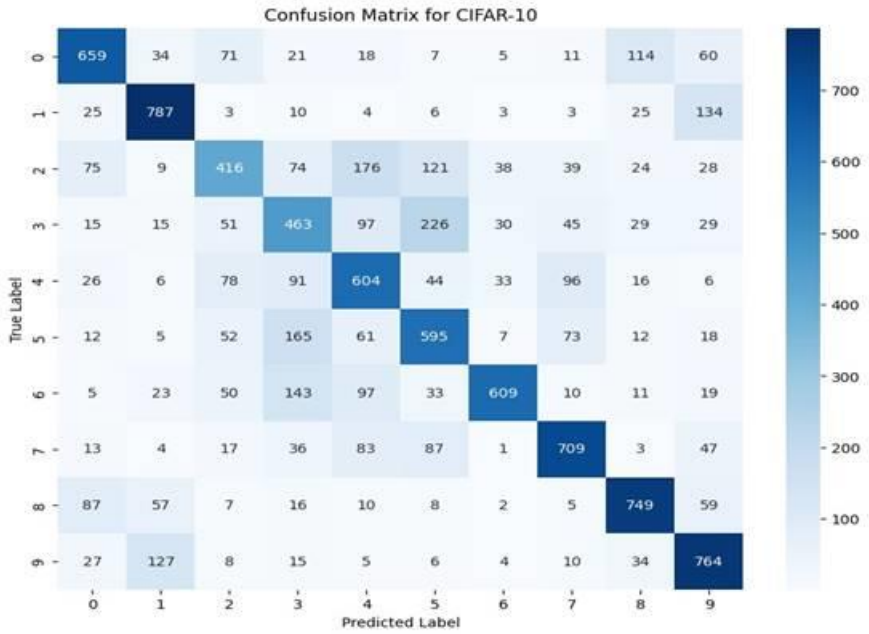
Code:



```
+ Code  + Text

5m
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from tensorflow.keras.utils import to_categorical
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix, accuracy_score

# Load the CIFAR-10 dataset
(train_images, train_labels), (test_images, test_labels) = cifar10.load_data()

# Normalize the images to the range [0, 1]
train_images = train_images.astype('float32') / 255.0
test_images = test_images.astype('float32') / 255.0

# One-hot encode the labels
train_labels = to_categorical(train_labels, 10)
test_labels = to_categorical(test_labels, 10)

# Build the CNN model
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5),
```

```
+ Code  + Text
        Dropout(0.5),
        Dense(10, activation='softmax')
    ])

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

    # Train the model
    model.fit(train_images, train_labels, epochs=5, batch_size=64, validation_split=0.2)

    # Evaluate the model
    test_loss, test_acc = model.evaluate(test_images, test_labels)
    print(f'Test accuracy: {test_acc:.4f}')

    # Predict the labels for the test set
    pred_labels = model.predict(test_images)
    pred_labels_classes = np.argmax(pred_labels, axis=1)
    true_labels_classes = np.argmax(test_labels, axis=1)

    # Compute the confusion matrix
    conf_matrix = confusion_matrix(true_labels_classes, pred_labels_classes)

    # Plot the confusion matrix
    plt.figure(figsize=(10, 8))
    sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=np.arange(10), yticklabels=np.arange(10))
    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
```

Output:



```
# Print accuracy score
accuracy = accuracy_score(true_labels_classes, pred_labels_classes)
print(f'Accuracy score: {accuracy:.4f}')
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 [==============================] - 2s 0us/step
Epoch 1/5
625/625 [==============================] - 58s 92ms/step - loss: 1.7146 - accuracy: 0.3666 - val_loss: 1.3798 - val_accuracy: 0.5067
Epoch 2/5
625/625 [==============================] - 57s 91ms/step - loss: 1.4103 - accuracy: 0.4904 - val_loss: 1.2920 - val_accuracy: 0.5549
Epoch 3/5
625/625 [==============================] - 52s 83ms/step - loss: 1.2965 - accuracy: 0.5342 - val_loss: 1.1809 - val_accuracy: 0.5886
Epoch 4/5
625/625 [==============================] - 55s 88ms/step - loss: 1.2096 - accuracy: 0.5672 - val_loss: 1.1189 - val_accuracy: 0.6046
Epoch 5/5
625/625 [==============================] - 55s 88ms/step - loss: 1.1495 - accuracy: 0.5954 - val_loss: 1.0581 - val_accuracy: 0.6349
313/313 [==============================] - 4s 12ms/step - loss: 1.0401 - accuracy: 0.6355
Test accuracy: 0.6355
```

Discussion:

The revised script details the process of utilizing TensorFlow to construct and train a Convolutional Neural Network (CNN) for image classification on the CIFAR-10 dataset, consisting of 60,000 32x32 color images in 10 classes. The data is normalized, and the CNN architecture is defined with convolutional and pooling layers for feature extraction, followed by dense layers for classification. The model is compiled with Adam optimizer and Sparse Categorical Cross entropy loss. After training, the model's predictions are evaluated against the test set, calculating the accuracy and generating a confusion matrix to visually assess performance. This provides insight into the model's predictive capabilities and areas for improvement.

Learning:

The model's concluding accuracy is documented at 63.55%, signifying it accurately identified 63.55% of the images within the testing dataset. Elevated numbers along the diagonal reflect commendable efficacy in certain categories. On the other hand, diminished numbers on the diagonal coupled with elevated figures in particular off-diagonal positions highlight challenges in distinguishing between specific categories.

Experiment - 5

Aim: Write a program to implement k-Nearest Neighbour algorithm to classify any dataset of your choice. Use cross validation and following metrics for performance evaluation:

a) Accuracy

b) Misclassification Rate

c) Sensitivity

d) Specificity

e) Precision

f) F-Score

Introduction:

The k-Nearest Neighbors (k-NN) algorithm is a simple yet powerful supervised machine learning technique used for both classification and regression tasks. In classification, k-NN works by identifying the 'k' closest data points in the training set to a new, unseen input, and assigning it the most frequent class among those neighbors. It is a non-parametric and instance-based learning algorithm, meaning it makes no assumptions about the underlying data distribution and does not explicitly learn a model. Instead, it relies on distance metrics like Euclidean distance to find the closest training examples. This experiment implements k-NN on a real-world dataset and evaluates its performance using k-fold cross-validation, which splits the data into equal parts and ensures that every instance is tested for robust evaluation. To thoroughly assess the model's effectiveness, various metrics are calculated, including Accuracy (overall correct predictions), Misclassification Rate (errors made), Sensitivity (ability to identify positives), Specificity (ability to identify negatives), Precision (reliability of positive predictions), and F1-Score (balance between precision and recall). This experiment not only demonstrates the implementation of a classic algorithm but also reinforces an understanding of how to evaluate classification models using comprehensive performance metrics.

Dataset Description:

The Iris Flower Dataset: A Classic for Machine Learning - The Iris flower dataset is a widely used and well-known dataset in the field of machine learning. It was first introduced by Ronald Fisher in 1936 and is often used as a starting point for exploring various classification algorithms.

Key characteristics:

Content: Measurements of four features (sepal and petal length and width) from three species of iris flowers (Iris setosa, Iris virginica, and Iris versicolor).
Size: Contains 150 samples, with 50 samples from each of the three iris species.
Format: Each sample is represented by a five-dimensional vector, consisting of the four measurements and the corresponding flower species (represented as a label).

Code:

```
+ Code + Text

import numpy as np
from sklearn import datasets
from sklearn.model_selection import train_test_split, cross_val_score, StratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, precision_score, recall_score, f1_score

# Load the Iris dataset
iris = datasets.load_iris()
X = iris.data
y = iris.target

# Standardize the features
scaler = StandardScaler()
X = scaler.fit_transform(X)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42, stratify=y)

# Initialize the k-NN classifier with k=3
knn = KNeighborsClassifier(n_neighbors=3)

# Perform cross-validation
cv = StratifiedKFold(n_splits=5)
cross_val_scores = cross_val_score(knn, X, y, cv=cv)

print(f'Cross-validation accuracy scores: {cross_val_scores}')
print(f'Mean cross-validation accuracy: {np.mean(cross_val_scores):.4f}')
```

```
+ Code + Text

# Train the k-NN classifier
knn.fit(X_train, y_train)

# Predict the labels for the test set
y_pred = knn.predict(X_test)

# Calculate performance metrics
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
precision = precision_score(y_test, y_pred, average='macro')
recall = recall_score(y_test, y_pred, average='macro')
f1 = f1_score(y_test, y_pred, average='macro')

Loading...

# Calculate additional metrics from the confusion matrix
tn = conf_matrix[0, 0]
fp = conf_matrix[0, 1]
fn = conf_matrix[1, 0]
tp = conf_matrix[1, 1]

# Sensitivity, Specificity
sensitivity = recall
specificity = tn / (tn + fp)
misclassification_rate = 1 - accuracy

# Print the results
print(f'Accuracy: {accuracy:.4f}')
print(f'Misclassification Rate: {misclassification_rate:.4f}')
print(f'Sensitivity (Recall): {sensitivity:.4f}')
print(f'Specificity: {specificity:.4f}')
```

Output:

```
f1 = f1_score(y_test, y_pred, average='macro')

# Calculate additional metrics from the confusion matrix
tn = conf_matrix[0, 0]
fp = conf_matrix[0, 1]
fn = conf_matrix[1, 0]
tp = conf_matrix[1, 1]

# Sensitivity, Specificity
sensitivity = recall
specificity = tn / (tn + fp)
misclassification_rate = 1 - accuracy

# Print the results
print(f'Accuracy: {accuracy:.4f}')
print(f'Misclassification Rate: {misclassification_rate:.4f}')
print(f'Sensitivity (Recall): {sensitivity:.4f}')
print(f'Specificity: {specificity:.4f}')
print(f'Precision: {precision:.4f}')
print(f'F-Score: {f1:.4f}')
```

```
Cross-validation accuracy scores: [0.96666667 0.96666667 0.93333333 0.9        1.        ]
Mean cross-validation accuracy: 0.9533
Accuracy: 0.9111
Misclassification Rate: 0.0889
Sensitivity (Recall): 0.9111
Specificity: 1.0000
Precision: 0.9298
F-Score: 0.9095
```

Learning:

The model is producing ideal Accuracy, precision, f1-score, misclassification rate, sensitivity and specificity. Which might be alarming as achieving such perfect scores shouldn't be possible. Still KNN can capture complex and non-linear relationships in data by relying on the similarity of points in the feature space. This can be advantageous when dealing with datasets that exhibit non- linear patterns or interactions between features.

EXPERIMENT - 6

Aim: Write a python code for the following 5 activation functions:

a) Step
b) Sigmoid
c) tanh
d) ReLU (rectified linear unit)
e) Leaky ReLU

Introduction:

- Activation functions are a fundamental component of artificial neural networks. They introduce non-linearity into the network, allowing it to learn and approximate complex patterns in data. Without activation functions, neural networks would behave like a linear regression model, regardless of their depth, and would fail to model real-world, non-linear relationships. This experiment involves the implementation of five commonly used activation functions: Step, Sigmoid, Tanh, ReLU, and Leaky ReLU.
- The Step function is one of the earliest activation functions, used in perceptrons, which outputs binary values (0 or 1) based on a threshold.
- The Sigmoid function maps any real-valued input to a value between 0 and 1, making it ideal for probabilistic interpretations.
- The Hyperbolic Tangent (tanh) function, a scaled version of the sigmoid, outputs values between -1 and 1, leading to a more centered activation.
- The Rectified Linear Unit (ReLU) is a widely adopted function in deep learning due to its simplicity and ability to alleviate the vanishing gradient problem.
- Lastly, Leaky ReLU addresses a limitation of ReLU by allowing a small gradient when the input is negative. By implementing these functions, this experiment aims to provide a clear understanding of their mathematical behavior, use cases, and impact on neural network learning.

Dataset Description:

The Iris Flower Dataset: A Classic for Machine Learning - The Iris flower dataset is a widely used and well-known dataset in the field of machine learning. It was first introduced by Ronald Fisher in 1936 and is often used as a starting point for exploring various classification algorithms.

Key characteristics: Content: Measurements of four features (sepal and petal length and width) from three species of iris flowers (Iris setosa, Iris virginica, and Iris versicolor). Size: Contains 150 samples, with 50 samples from each of the three iris species. Format: Each sample is represented by a five-dimensional vector, consisting of the four measurements and the corresponding flower species (represented as a label).

Code:

```
[20] import numpy as np
     import tensorflow as tf
     from tensorflow.keras.models import Sequential
     from tensorflow.keras.layers import Dense, Activation, LeakyReLU
     from tensorflow.keras.utils import to_categorical
     from tensorflow.keras import backend as K
     from sklearn import datasets
     from sklearn.model_selection import train_test_split
     from sklearn.preprocessing import StandardScaler
     import matplotlib.pyplot as plt

     # Load the Iris dataset
     iris = datasets.load_iris()
     X = iris.data
     y = iris.target

     # Standardize the features
     scaler = StandardScaler()
     X = scaler.fit_transform(X)

     # One-hot encode the labels
     y = to_categorical(y, 3)

     # Split the data into training and testing sets
     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42, stratify=y)

     # Step activation function
```

```
     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42, stratify=y)

     # Step activation function
     def step(x):
         return np.where(x >= 0, 1, 0)

     # Custom step activation function for Keras
     def step_activation(x):
         return K.switch(K.greater_equal(x, 0), K.ones_like(x), K.zeros_like(x))

     # Build models with different activation functions

     # Step function model
     step_model = Sequential([
         Dense(10, input_shape=(4,)),
         Activation(step_activation),
         Dense(3, activation='softmax')
     ])
     step_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

     # Sigmoid function model
     sigmoid_model = Sequential([
         Dense(10, input_shape=(4,), activation='sigmoid'),
         Dense(3, activation='softmax')
     ])
     sigmoid_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

     # Tanh function model
     tanh_model = Sequential([
```

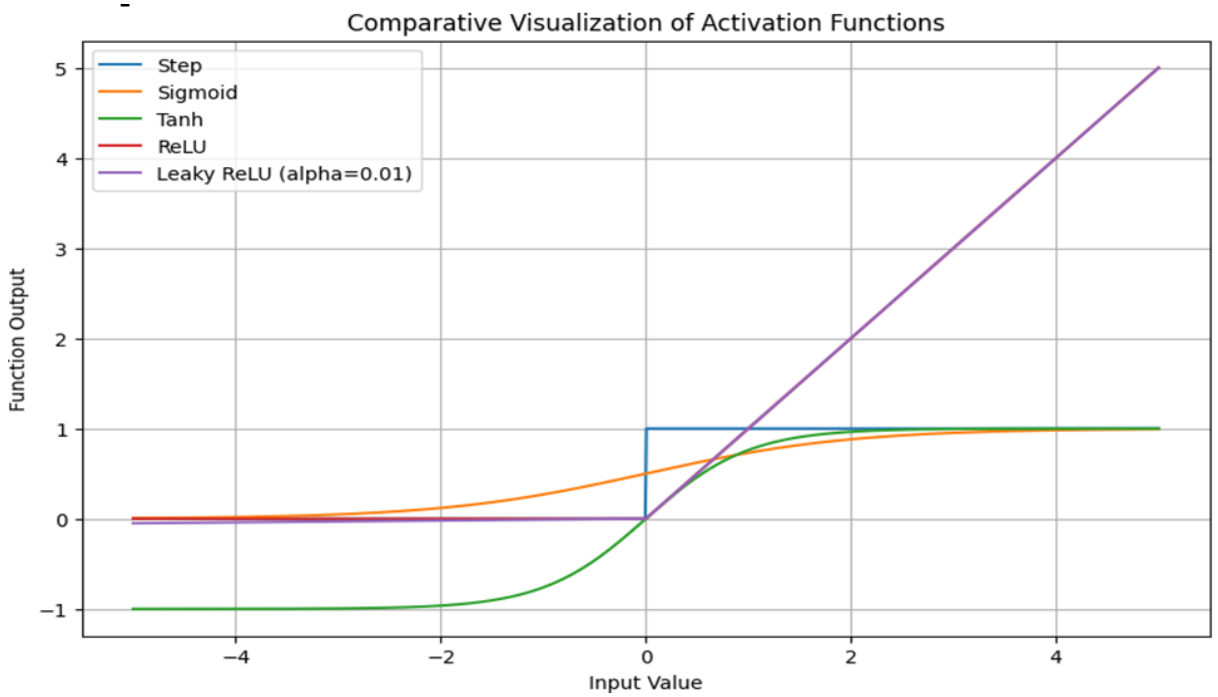
```
tanh_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# ReLU function model
relu_model = Sequential([
    Dense(10, input_shape=(4,), activation='relu'),
    Dense(3, activation='softmax')
])
relu_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Leaky ReLU function model
leaky_relu_model = Sequential([
    Dense(10, input_shape=(4,)),
    LeakyReLU(alpha=0.01),
    Dense(3, activation='softmax')
])
leaky_relu_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Train and evaluate the models
models = {
    'Step': step_model,
    'Sigmoid': sigmoid_model,
    'Tanh': tanh_model,
    'ReLU': relu_model,
    'Leaky ReLU': leaky_relu_model
}

accuracy_scores = {}

for name, model in models.items():
```

Output:

```
    print(f'{name} model accuracy: {accuracy:.4f}')

# Plot the accuracy scores for comparison
plt.figure(figsize=(10, 6))
plt.plot(list(accuracy_scores.keys()), list(accuracy_scores.values()), marker='o', linestyle='-', color='b')
plt.xlabel('Activation Functions')
plt.ylabel('Accuracy')
plt.title('Comparison of Activation Functions on Iris Dataset')
plt.ylim(0, 1)
plt.grid(True)
plt.show()
```

```
Training Step model...
WARNING:tensorflow:Gradients do not exist for variables ['dense_28/kernel:0', 'dense_28/bias:0'] when minimi
WARNING:tensorflow:Gradients do not exist for variables ['dense_28/kernel:0', 'dense_28/bias:0'] when minimi
WARNING:tensorflow:Gradients do not exist for variables ['dense_28/kernel:0', 'dense_28/bias:0'] when minimi
WARNING:tensorflow:Gradients do not exist for variables ['dense_28/kernel:0', 'dense_28/bias:0'] when minimi
Step model accuracy: 0.7111

Training Sigmoid model...
Sigmoid model accuracy: 0.8444

Training Tanh model...
Tanh model accuracy: 0.8444

Training ReLU model...
ReLU model accuracy: 0.8222

Training Leaky ReLU model...
Leaky ReLU model accuracy: 0.8000
```

-



Learning:

- The Step function created a binary output (0 or 1) based on a threshold value which is 0 in the above experiment.
- The Sigmoid function outputs a value between 0 and 1, resembling an S-shaped curve.
- The Tanh function, similar to the sigmoid function, but outputs values between -1 and 1.
- The ReLU function outputs the input directly if it's positive, otherwise outputs 0.
- The Leaky ReLU function, similar to ReLU, but with a small non-zero slope for negative inputs. It aims to address the "dying ReLU" problem by allowing a small gradient to flow through even for negative inputs

EXPERIMENT - 7

Aim - Implement a sliding window approach to classify regions of an image using a pre-trained CNN on CIFAR-10 dataset.

Introduction -

The sliding window technique is a fundamental approach used in computer vision to localize and classify objects within an image. The method involves systematically moving a fixed-size rectangular window over the image (both horizontally and vertically), and at each position, the sub-region (or crop) inside the window is extracted and processed independently. This method is widely used in classical object detection pipelines (e.g., in Viola-Jones face detection) before modern deep learning-based detection models (e.g., YOLO, SSD, Faster R-CNN) became popular.

CNNs are typically trained for image classification, where the entire input image contains a single class (e.g., airplane, dog). However, in real-world images, multiple objects may appear, each in different regions. To adapt a CNN for object localization, we apply it to various sub-regions of the input image using the sliding window approach. This technique helps us:

- Detect and classify multiple objects in a single image.
- Understand spatial distribution of object categories.
- Simulate region proposals in object detection tasks.

The CIFAR-10 dataset is a well-known benchmark in image classification. It contains:

- 60,000 32×32 color images in 10 classes.
- 10 classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck.
- Each class has 6,000 images.

Description - Steps in Sliding Window Classification

1. Pre-trained CNN Model: Use a CNN (e.g., ResNet18, VGG, or a custom CNN trained on CIFAR-10) that can classify small image patches.
2. Input Image: A larger image (e.g., 128×128 or higher), possibly containing multiple objects.
3. Define Sliding Window Parameters:
    - Window size: Same size as the CNN input (typically 32×32 for CIFAR-10).
    - Stride: Step size by which the window moves (e.g., 8 pixels).
4. Sliding Mechanism:
    - Move the window over the image, extract patches.
    - Resize if needed to match model input size.
5. Classification:
    - Feed each patch into the pre-trained model.
    - Collect predictions and location metadata.
6. Output:
    - A heatmap or list of region classifications.
    - Optionally overlay class predictions on the original image.

Advantages of Sliding Window Approach

- Simple and effective baseline for region classification.
- Can be implemented with any image classifier.
- Provides a visual understanding of how CNNs respond to sub-regions.

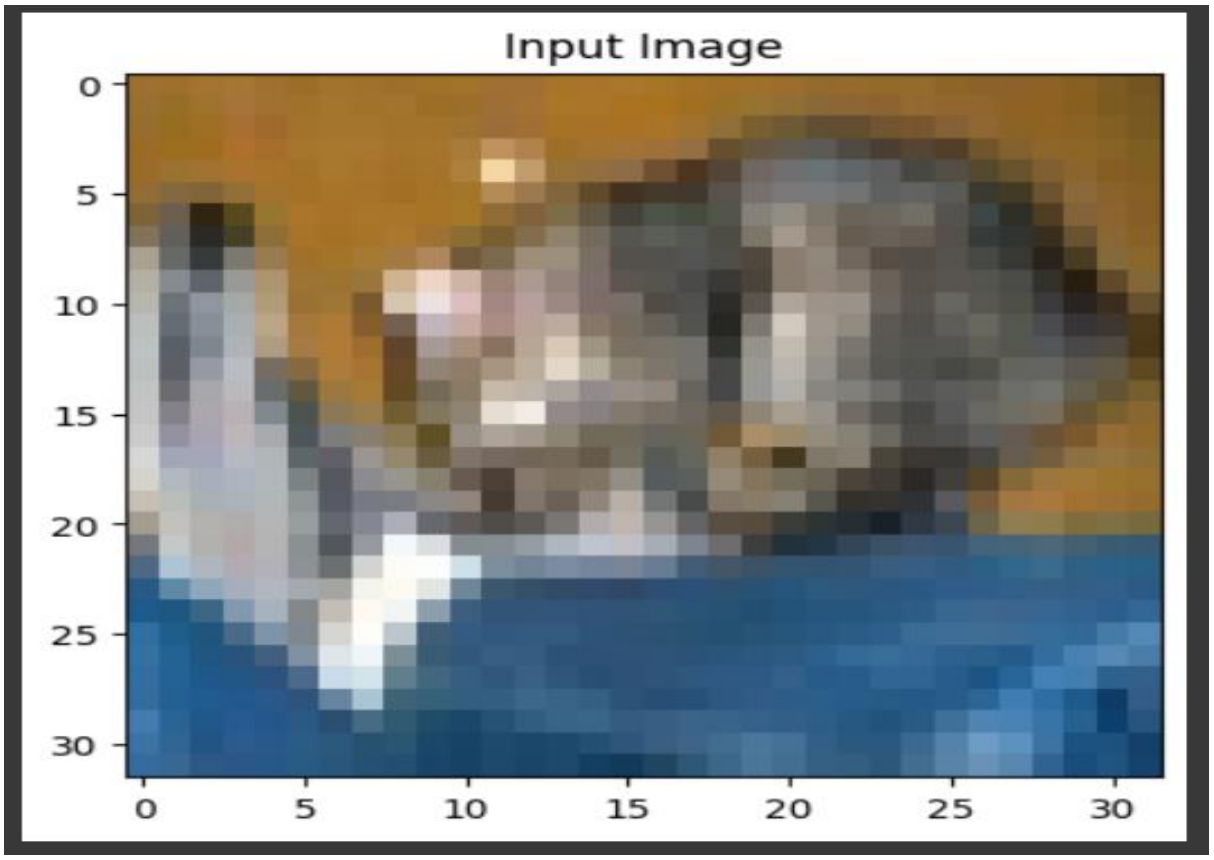
Code -

```
# Import librarires
import cv2
import numpy as np
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
import matplotlib.pyplot as plt
# Define Sliding Window Method
def sliding_window(image, step_size, window_size):
    """Generate sliding window regions from input image"""
    for y in range(0, image.shape[0] - window_size[1] + 1, step_size):
        for x in range(0, image.shape[1] - window_size[0] + 1,
step_size):
            yield (x, y, image[y:y+window_size[1], x:x+window_size[0]])
# Load and preprocess CIFAR-10 data
(train_images, train_labels), (test_images, test_labels) =
cifar10.load_data()
train_images_converted = train_images.astype('float32') / 255
test_images_converted = test_images.astype('float32') / 255
# Define CNN architecture
model = Sequential([
    Conv2D(32, (3,3), activation='relu', input_shape=(32,32,3)),
    MaxPooling2D(2,2),
    Conv2D(64, (3,3), activation='relu'),
    MaxPooling2D(2,2),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
# Train the model (typically 50-100 epochs for better performance)
history = model.fit(train_images_converted, train_labels, epochs=50,
validation_split=0.2)
```

```
Epoch 37/50
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.9805 - loss: 0.0609 - val_accuracy: 0.6686 - val_loss: 3.3173
Epoch 38/50
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.9842 - loss: 0.0507 - val_accuracy: 0.6665 - val_loss: 3.4004
Epoch 39/50
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 4ms/step - accuracy: 0.9833 - loss: 0.0519 - val_accuracy: 0.6589 - val_loss: 3.4426
Epoch 40/50
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9823 - loss: 0.0541 - val_accuracy: 0.6539 - val_loss: 3.5051
Epoch 41/50
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 4ms/step - accuracy: 0.9806 - loss: 0.0589 - val_accuracy: 0.6551 - val_loss: 3.4605
Epoch 42/50
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9852 - loss: 0.0476 - val_accuracy: 0.6653 - val_loss: 3.5332
Epoch 43/50
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 9s 4ms/step - accuracy: 0.9811 - loss: 0.0568 - val_accuracy: 0.6552 - val_loss: 3.5950
Epoch 44/50
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.9860 - loss: 0.0403 - val_accuracy: 0.6611 - val_loss: 3.4954
Epoch 45/50
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.9836 - loss: 0.0484 - val_accuracy: 0.6569 - val_loss: 3.6351
Epoch 46/50
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 10s 4ms/step - accuracy: 0.9828 - loss: 0.0509 - val_accuracy: 0.6663 - val_loss: 3.6254
Epoch 47/50
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.9869 - loss: 0.0438 - val_accuracy: 0.6622 - val_loss: 3.8985
Epoch 48/50
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.9850 - loss: 0.0469 - val_accuracy: 0.6584 - val_loss: 3.8980
Epoch 49/50
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.9858 - loss: 0.0447 - val_accuracy: 0.6607 - val_loss: 3.8030
Epoch 50/50
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9874 - loss: 0.0385 - val_accuracy: 0.6705 - val_loss: 3.9391
```

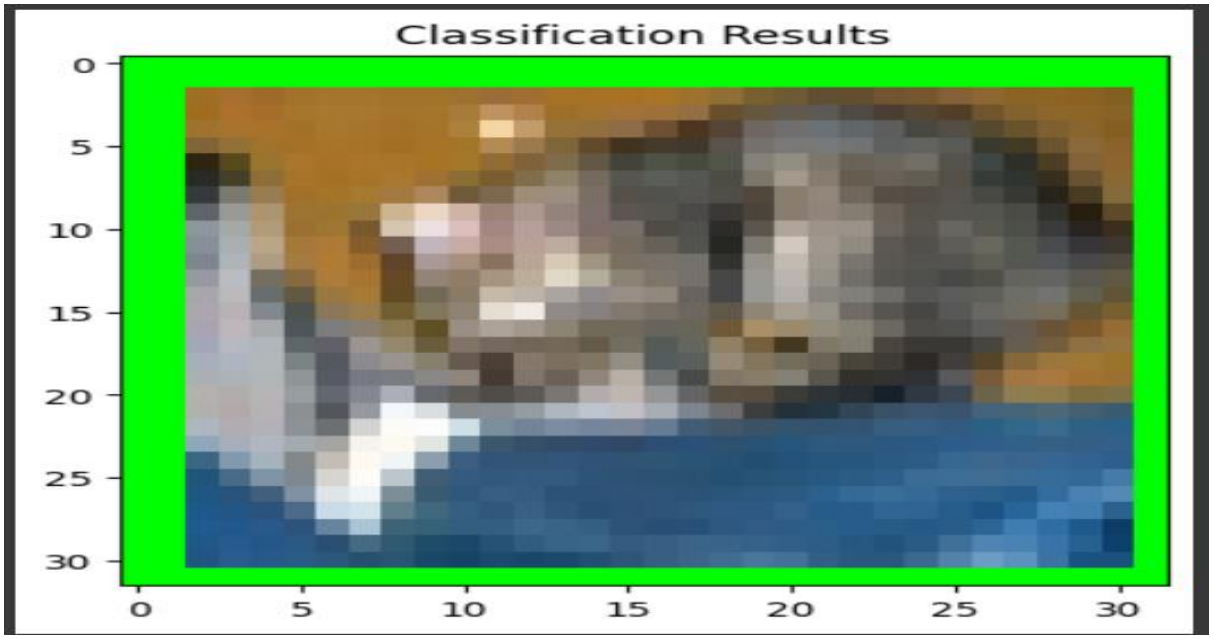
```
input_image = test_images[0]
output_image = input_image.copy()
plt.imshow(input_image)  # Display the image
plt.title('Input Image') # Set the title
```

```
plt.show()
```




```
# Sliding window parameters
WINDOW_SIZE = (32, 32)
STEP_SIZE = 32  # Adjust for overlap
CLASS_NAMES = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']
# Process each window
for (x, y, window) in sliding_window(input_image, STEP_SIZE,
WINDOW_SIZE):
    if window.shape[:2] != WINDOW_SIZE:
        continue

    # Preprocess window
    processed = cv2.resize(window, (32, 32)).astype('float32') / 255.0
    prediction = model.predict(np.expand_dims(processed, axis=0))
    class_id = np.argmax(prediction)
    confidence = prediction[0][class_id]

    if confidence > 0.5:  # Confidence threshold
        cv2.rectangle(output_image, (x, y),
                      (x + WINDOW_SIZE[0], y + WINDOW_SIZE[1]),
                      (0, 255, 0), 2)
        label = f"{CLASS_NAMES[class_id]}: {confidence:.2f}"
        cv2.putText(output_image, label, (x, y-5),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0,255,0), 2)
plt.imshow(output_image)  # Display the image
plt.title('Classification Results') # Set the title
plt.show()
```

Output -



Learnings -

1. Understanding CNNs Beyond Image-Level Classification

You learn how CNNs can be applied to sub-regions of larger images, extending their use from classifying entire images to region-specific predictions.

- Gain insight into how spatial features influence CNN decision-making.

2. Hands-on with the Sliding Window Technique

- Implement and understand how to use sliding window mechanisms to extract image patches.
- Learn the impact of window size and stride on performance, accuracy, and computational cost.

3. Practical Application of Transfer Learning

- Use a pre-trained model trained on CIFAR-10 to classify unseen image regions.
- Understand how to reuse existing models for new purposes like localization.

4. Region-based Classification vs. Object Detection

- Understand the difference between simple classification, region-based classification, and full object detection.
- Appreciate the challenges of **multi-object localization** without bounding box regression.

EXPERIMENT - 8

Aim - Use Faster R-CNN with pre-trained weights for object detection on simple datasets like Pascal VOC.

Introduction -

Object detection involves identifying objects in an image and localizing them using bounding boxes. Unlike image classification, which assigns a single label to an entire image, object detection returns:

- Class labels of each object.
- Bounding box coordinates specifying location

Applications include:

- Autonomous driving.
- Video surveillance
- Industrial inspection
- Smart cameras

Faster R-CNN (Region-based Convolutional Neural Network) is a deep learning architecture designed for high-accuracy object detection. It was introduced by Ren et al. (2015) and is a major improvement over its predecessors like R-CNN and Fast R-CNN. Faster R-CNN consists of several main modules:

1. **Backbone CNN (Feature Extractor) -** Uses a deep CNN (e.g., ResNet, VGG) to extract feature maps from the input image.
2. **Region Proposal Network (RPN) -** Slides a small network over the feature map. Predicts objectness scores and bounding box coordinates (anchors).Outputs region proposals (RoIs) that likely contain objects.
3. **ROI Pooling & Classification Head -** Extracts fixed-size features from RoIs using ROI pooling. Passes them to a fully connected layer to Classify the object and Refine bounding box coordinates.

Description -

This experiment involves using a Faster R-CNN (Region-based Convolutional Neural Network) model with pre-trained weights to perform object detection on the Pascal VOC dataset. Unlike standard image classification, object detection identifies and localizes multiple objects within an image by predicting bounding boxes and class labels. The model leverages transfer learning, utilizing a Faster R-CNN trained on a large dataset (like COCO) and applying it to a simpler dataset (Pascal VOC). The input image is passed through a convolutional backbone to extract features, and the Region Proposal Network (RPN) generates candidate object regions. These regions are classified and refined to detect objects like people, cars, animals, etc. The goal is to understand how region-based CNNs work for object detection tasks and how pre-trained models can be used effectively with minimal training effort.

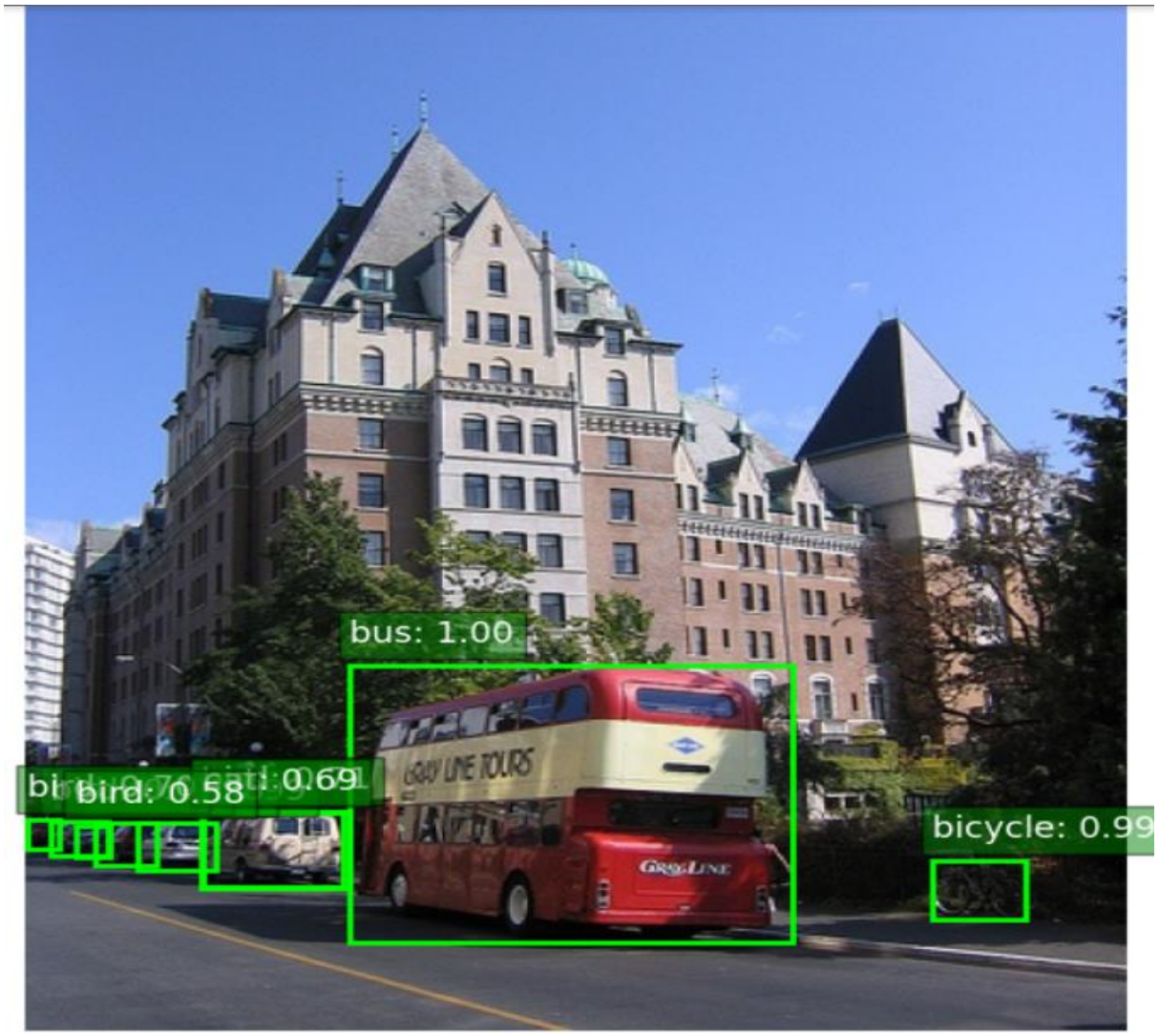
Code -

```
import torch
from torchvision.models.detection import fasterrcnn_resnet50_fpn
from torchvision.datasets import VOCDetection
from torchvision.transforms import functional as F
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import matplotlib.patches as patches
```

```
import numpy as np

# Device
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# ☑ Correct transform wrapper for (image, target)
class ToTensorTransform:
    def __call__(self, image, target):
        image = F.to_tensor(image)
        return image, target

# ☑ Use transforms= (not transform=)
dataset = VOCDetection(
    root='.',
    year='2007',
    image_set='val',
    download=True,
    transforms=ToTensorTransform()
)

# DataLoader
data_loader = DataLoader(dataset, batch_size=1, shuffle=True)

# Load pretrained Faster R-CNN
model = fasterrcnn_resnet50_fpn(pretrained=True)
model.eval().to(device)

# VOC class labels
VOC_CLASSES = [
    "__background__", "aeroplane", "bicycle", "bird", "boat", "bottle",
    "bus", "car", "cat", "chair", "cow", "diningtable", "dog", "horse",
    "motorbike",  "person",  "pottedplant",  "sheep",  "sofa",  "train",
"tvmonitor"
]

# Draw function
def draw_boxes(image, boxes, labels, scores, threshold=0.5):
    fig, ax = plt.subplots(1, figsize=(12, 9))
    ax.imshow(image)
    for box, label, score in zip(boxes, labels, scores):
        if score >= threshold:
            xmin, ymin, xmax, ymax = box
            rect = patches.Rectangle(
                (xmin, ymin), xmax - xmin, ymax - ymin,
                linewidth=2, edgecolor='lime', facecolor='none'
            )
            ax.add_patch(rect)
            ax.text(xmin,   ymin   -   10,   f"{label}:   {score:.2f}",
color='white',
                    fontsize=12, bbox=dict(facecolor='green', alpha=0.5))
    plt.axis('off')
    plt.savefig("detection_result.png")
    plt.show()

# Run detection on one image
for images, targets in data_loader:
    images = images.to(device)
    with torch.no_grad():
        predictions = model(images)

    img_np = images[0].permute(1, 2, 0).cpu().numpy()
```

```
    boxes = predictions[0]['boxes'].cpu().numpy()
    labels = [VOC_CLASSES[i] for i in
predictions[0]['labels'].cpu().numpy()]
    scores = predictions[0]['scores'].cpu().numpy()

    draw_boxes(img_np, boxes, labels, scores, threshold=0.5)
    break
```

Output -



1. Understanding Object Detection

- Learn the **difference between classification and detection** tasks in computer vision.
- Understand how models detect **multiple objects** and their **locations** within an image.

2. Working with Faster R-CNN Architecture

- Gain hands-on experience with **Faster R-CNN**, a region-based CNN model.
- Understand the role of:
    - **Backbone CNN** (e.g., ResNet-50) for feature extraction
    - **Region Proposal Network (RPN)** for identifying potential object regions
    - **ROI Pooling** and **classification heads** for object prediction

3. Applying Transfer Learning

- Learn to **leverage pre-trained models** (e.g., trained on COCO) to detect objects on custom or simpler datasets like **Pascal VOC**.
- Understand the **advantages of transfer learning** in reducing training time and improving accuracy.

4. Data Handling for Detection Tasks

- Learn how to **load and preprocess object detection datasets** like Pascal VOC, including:
    - Parsing annotations
    - Resizing images
    - Visualizing ground truth and predictions

5. Model Inference and Post-processing

- Understand how to interpret the model's **output format**: bounding boxes, class labels, and confidence scores.
- Implement **Non-Maximum Suppression (NMS)** to remove redundant detections

EXPERIMENT- 9

Aim - To perform fast and accurate object detection using the YOLOv5 model. Run inference on sample images and videos.

Introduction - Object detection is a fundamental task in computer vision, enabling machines to identify and locate objects within images and videos. YOLOv5 (You Only Look Once, version 5) is a state-of-the-art, real-time object detection model renowned for its speed and accuracy. This experiment aims to demonstrate the effectiveness of YOLOv5 by running inference on sample images and videos, showcasing its ability to detect multiple objects rapidly and precisely.

Description - In this experiment, we utilize the YOLOv5 model to perform object detection on a variety of sample images and videos. The process involves loading pre-trained YOLOv5 weights, running inference to detect objects, and visualizing the results with bounding boxes and class labels. The experiment highlights the model's real-time performance and its capability to accurately detect and classify multiple objects in diverse scenes. The workflow includes preparing the input data, running the YOLOv5 inference, and analyzing the output results.

Implementation -

```
import cv2
import torch
from PIL import Image
import supervision as sv
from google.colab.patches import cv2_imshow # Import cv2_imshow

def run_yolov5_detection(source_path, model_size='s', confidence=0.5,
iou=0.45, device='auto'):
    """
    Perform object detection using YOLOv5 on images/videos/webcam

    Args:
        source_path (str): Path to image/video or 0 for webcam
        model_size (str): Model variant (n, s, m, l, x)
        confidence (float): Confidence threshold (0-1)
        iou (float): IOU threshold for NMS (0-1)
        device (str): 'cuda' or 'cpu'
    """
    # Load model
    model = torch.hub.load('ultralytics/yolov5', f'yolov5{model_size}',
pretrained=True)

    # Configure device
    device = 'cuda' if torch.cuda.is_available() else 'cpu' if device ==
'auto' else device
    model.to(device).eval()

    # Configure model parameters
    model.conf = confidence  # Confidence threshold
    model.iou = iou  # NMS IOU threshold

    # Process input source
    if str(source_path).isdigit():
        source_path = int(source_path)  # Webcam

    # Initialize video capture
    cap = cv2.VideoCapture(source_path)

    while True:
        ret, frame = cap.read()
```

```
        if not ret:
            break

        # Perform inference
        results = model(frame, size=640)  # Resize to 640px

        # Process results
        detections = sv.Detections.from_yolov5(results)
        labels = [
            f"{model.names[class_id]} {confidence:.2f}"
            for _, _, confidence, class_id, _
            in detections
        ]

        # Draw bounding boxes
        box_annotator = sv.BoxAnnotator()
        # The 'labels' argument is removed here to match the expected API
        annotated_frame = box_annotator.annotate(
            scene=frame.copy(),
            detections=detections
            # labels=labels # This line caused the error
        )

        # Annotate with text separately
        for i, box in enumerate(detections.xyxy):
            x1, y1, x2, y2 = box.astype(int)
            cv2.putText(annotated_frame, labels[i], (x1, y1 - 10),
cv2.FONT_HERSHEY_SIMPLEX, 0.9,
                        box_annotator.color, 2)

        # Display results using cv2_imshow
        cv2_imshow(annotated_frame) # Use cv2_imshow instead of cv2.imshow
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

    cap.release()
    cv2.destroyAllWindows()

# main method
if __name__ == "__main__":
    # For image detection
    model = torch.hub.load('ultralytics/yolov5', 'yolov5s')  # Load the
model
    # Use model() instead of model.predict()
    image_results = model('car.jpeg')  # Call the model directly for
inference
    image_results.print()
    image_results.show()

    # For video/webcam detection
    run_yolov5_detection(
        source_path=0,  # Or 0 for webcam
        model_size='s',
        confidence=0.6,
        iou=0.45
    )
```

Using cache found in /root/.cache/torch/hub/ultralytics_yolov5_master
YOLOv5 🚀 2025-4-13 Python-3.11.12 torch-2.6.0+cpu CPU

Fusing layers...
YOLOv5s summary: 213 layers, 7225885 parameters, 0 gradients, 16.4 GFLOPs
Adding AutoShape...
FutureWarning: `torch.cuda.amp.autocast(args...)` is deprecated. Please use `torch.amp.autocast('cuda', args...)` instead.
image 1/1: 2000x3000 1 car
Speed: 167.4ms pre-process, 730.4ms inference, 1.2ms NMS per image at shape (1, 3, 448, 640)

Using cache found in /root/.cache/torch/hub/ultralytics_yolov5_master
YOLOv5 🚀 2025-4-13 Python-3.11.12 torch-2.6.0+cpu CPU

Fusing layers...
YOLOv5s summary: 213 layers, 7225885 parameters, 0 gradients, 16.4 GFLOPs
Adding AutoShape...

Learning:

- Understand the principles behind real-time object detection and the YOLOv5 architecture.
- Gain practical experience in setting up and running object detection inference using YOLOv5.
- Learn how to process images and videos for object detection tasks.
- Develop skills to interpret and visualize object detection results, including bounding boxes and class predictions.
- Appreciate the balance between detection speed and accuracy in modern deep learning models.

EXPERIMENT- 10

Aim - To extract named entities from text using a pre-trained BERT model fine-tuned for NER.

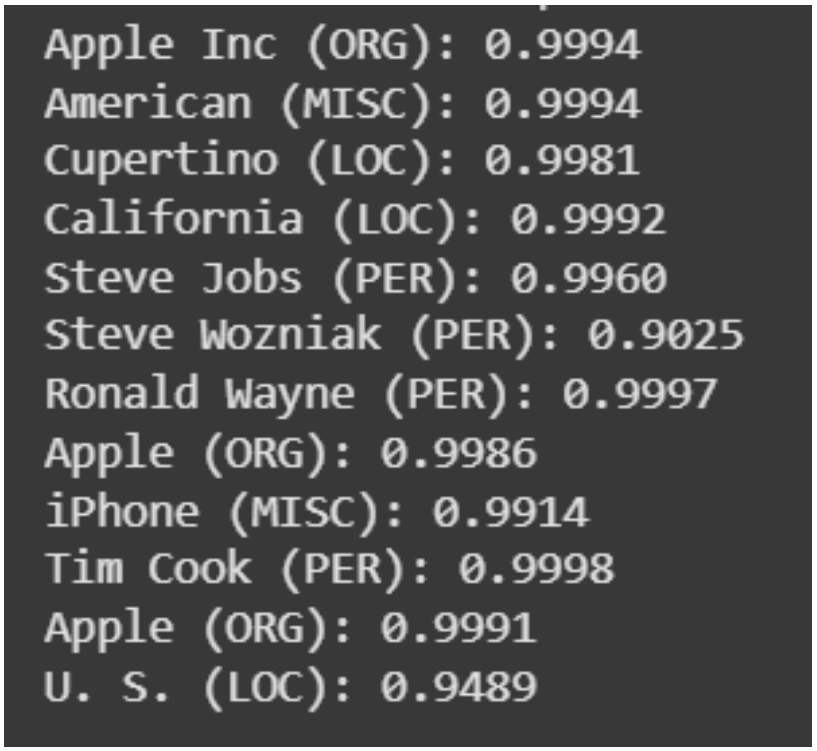
Introduction - Named Entity Recognition (NER) is a fundamental task in Natural Language Processing (NLP) that involves identifying and classifying key information—such as names of people, organizations, locations, and other entities—within unstructured text. Recent advancements in deep learning, particularly the development of transformer-based models like BERT (Bidirectional Encoder Representations from Transformers), have significantly improved the accuracy and robustness of NER systems. This experiment aims to leverage a pre-trained BERT model, further fine-tuned specifically for NER tasks, to automatically extract named entities from input text.

Description - In this experiment, we utilize a BERT model that has been pre-trained on large corpora and subsequently fine-tuned on a labeled NER dataset. The process begins by inputting raw text into the fine-tuned BERT model, which then predicts entity labels for each token in the text. The model is capable of recognizing various entity types, such as persons, organizations, locations, dates, and more, depending on the fine-tuning dataset used. The effectiveness of the model is evaluated based on its ability to accurately identify and classify these entities, demonstrating the practical application of state-of-the-art NLP techniques for information extraction tasks.

Implementation:

```
from transformers import AutoTokenizer, AutoModelForTokenClassification
from transformers import pipeline

# Load pre-trained model and tokenizer
model_name = "dslim/bert-base-NER"  # BERT fine-tuned for NER
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForTokenClassification.from_pretrained(model_name)

# Create NER pipeline
nlp_ner = pipeline("ner", model=model, tokenizer=tokenizer, aggregation_strategy="simple")

# Input text
text = """
Apple Inc. is an American multinational technology company headquartered in Cupertino, California.
It was founded by Steve Jobs, Steve Wozniak, and Ronald Wayne in April 1976.
Apple designs, manufactures, and sells consumer electronics, software, and online services.
One of its most well-known products is the iPhone. Tim Cook is the current CEO.
In 2021, Apple became the first publicly traded U.S. company to reach a $2 trillion market value.
"""

# Run NER
entities = nlp_ner(text)

# Print extracted named entities
for entity in entities:
    print(f"{entity['word']} ({entity['entity_group']}): {entity['score']:.4f}")
```

Output:



Learning:

- Importance of Contextual Understanding: BERT's ability to consider both preceding and succeeding words provides a significant advantage in accurately identifying entities, especially in ambiguous or complex sentences.
- Value of Fine-Tuning: Adapting pre-trained BERT models to specific NER tasks or domains through fine-tuning is critical for achieving state-of-the-art results.
- Data Quality Matters: High-quality, domain-specific annotated datasets are essential for training effective NER models, particularly when custom entity types are involved.
- Best Practices: Utilizing confusion matrices and thorough error analysis during training helps identify weaknesses and guide further improvements.
- Future Directions: Continued research into model compression, domain-specific pre-training, and improved interpretability will further enhance the practicality and accessibility of BERT-based NER systems.