Experiment 1

Date:

Aim: To write a python program to demonstrate the working of CNN architecture to classify images.

Algorithm:

1. Start
2. In google-collab edit option select notebook settings and select GPU as hardware accelerator to make training time less.
3. Load the CIFAR-10 dataset.
4. Normalize pixel values in the datasets.
5. Define a CNN architecture using tf.keras.Sequential() and add the following layers: Conv2D, MaxPooling2D, and Dropout.
6. Compile the model using model.compile() with the Adam optimizer, sparse categorical cross entropy loss function, and accuracy metric.
7. Train the model using model.fit().
8. Evaluate the model on the test dataset using model.evaluate()
9. Plot the predicted image
10. Stop

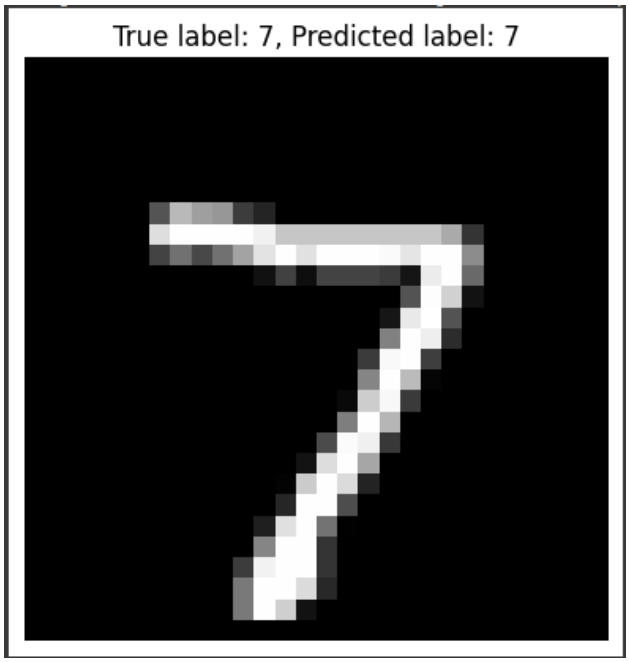
Code:

```
import tensorflow as tf
from tensorflow import keras
import matplotlib.pyplot as plt
import numpy as np

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

# Preprocess the data by reshaping it into a 4D tensor
x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
```

```
# One-hot encode the labels
y_train = keras.utils.to_categorical(y_train, 10)
y_test = keras.utils.to_categorical(y_test, 10)

# Build the model
model = keras.Sequential()
model.add(keras.layers.Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(keras.layers.MaxPooling2D(pool_size=(2, 2)))
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(64, activation='relu'))
model.add(keras.layers.Dense(10, activation='softmax'))

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

# Train the model
model.fit(x_train, y_train, batch_size=128, epochs=10, validation_split=0.1)

# Evaluate the model on the test data
test_loss, test_acc = model.evaluate(x_test, y_test)
print('Test accuracy:', test_acc)

# Predict the class of an image
predictions = model.predict(x_test[:1])

# Get the class with the highest probability
predicted_label = np.argmax(predictions[0])

# Plot the image and its predicted label
plt.imshow(x_test[0].reshape(28, 28), cmap='gray')
plt.title("True label: %d, Predicted label: %d" % (np.argmax(y_test[0]), predicted_label))
plt.axis('off')
plt.show()
```

Output:

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 [==============================] - 8s 0us/step

Epoch 1/10
1407/1407 [==============================] - 20s 6ms/step - loss: 1.6606 - accuracy: 0.3821 - val_loss: 1.2607 - val_accuracy: 0.5526
……..
Epoch 10/10
1407/1407 [==============================] - 8s 6ms/step - loss: 0.9173 - accuracy: 0.6780 - val_loss: 0.8777 - val_accuracy: 0.6942
<keras.callbacks.History at 0x7fa165e46670>

Test accuracy: 0.9843000173568726
1/1 [==============================] - 0s 99ms/step
```



Result:

Thus the python program to demonstrate the working of CNN architecture to classify images was executed successfully and output is verified.

Experiment 2

Date:

Aim: To write a python program to build a simple CNN model for image segmentation.

Algorithm:

1. Start
2. In google-collab edit option select notebook settings and select GPU as hardware accelerator to make training time less.
3. Load the MNIST dataset using Keras.
4. Preprocess the images by reshaping and normalizing them.
5. Define the CNN architecture using the Sequential model from Keras.
6. Compile the model using the compile() function from Keras.
7. Convert the labels to categorical using the to_categorical() function from Keras.
8. Train the model using the fit() function from Keras.
9. Evaluate the model using the evaluate() function from Keras.
10. Select a random image from the test set and segment the image using the trained model.
11. Plot the original and segmented images using the imshow() function from matplotlib.
12. Stop

Code:

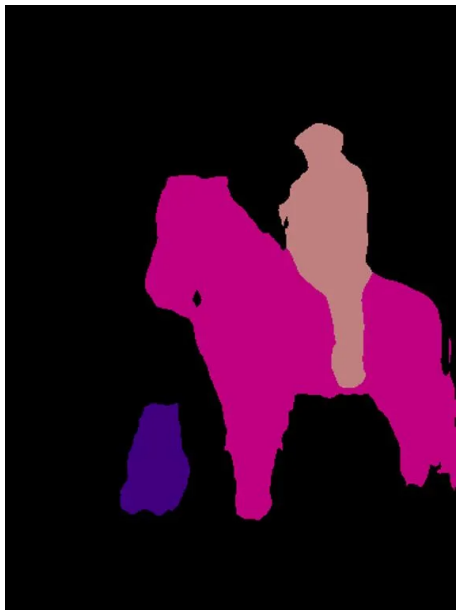
```
import numpy as np
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Dropout, Flatten, Dense
import matplotlib.pyplot as plt

import pixellib
from pixellib.semantic import semantic_segmentation


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

```
x_train = x_train.reshape(x_train.shape[0], x_train.shape[1], x_train.shape[2], 1)
x_test = x_test.reshape(x_test.shape[0], x_test.shape[1], x_test.shape[2], 1)

# Normalize pixel values between 0 and 1
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.

# Define number of labels
num_labels = 10

# Define input shape of the images
input_shape = x_train.shape[1:]

# Define the CNN
model = Sequential([
    Conv2D(32, 3, activation='relu', input_shape=input_shape),
    MaxPooling2D(2),
    Conv2D(64, 3, activation='relu'),
    MaxPooling2D(2),
    Dropout(0.25),
    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(num_labels, activation='softmax')
])

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

# Convert labels to categorical
y_train = tf.keras.utils.to_categorical(y_train, num_labels)
y_test = tf.keras.utils.to_categorical(y_test, num_labels)

# Train the model
model.fit(x_train, y_train, batch_size=128, epochs=10, validation_data=(x_test,
y_test))

# Evaluate the model
loss, accuracy = model.evaluate(x_test, y_test)
```

```
print('Test loss:', loss)
print('Test accuracy:', accuracy)

segment_image = semantic_segmentation()
segment_image.load_pascalvoc_model("deeplabv3_xception_tf_dim_ordering_tf
_kernels.h5")
segment_image.segmentAsPascalvoc("sample1.jpg", output_image_name =
"image_new.jpg")
```

Output:

Original Image



Semantic Image

Result:

Thus the python program to build a simple CNN model for image segmentation. was executed successfully and output is verified.

Experiment 3

Date:

Aim: To write a python program to build and train a CNN model for face recognition.

Algorithm:

1. Start
2. Load the face dataset, split it into training, validation, and test sets, and preprocess the images using ImageDataGenerator from Keras.
3. Define the CNN architecture using the Sequential model from Keras and compile it.
4. Train the model using the fit() function from Keras with the training data and validate it with the validation data.
5. Evaluate the model using the evaluate() function from Keras with the test data.
6. Use the predict() function from Keras to predict the class of each test image and convert the predicted classes to class names.
7. Display a random test image along with its predicted class and corresponding class name.
8. End the program.

Code:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import numpy as np
import os

# Set the directories for the dataset
train_dir = "path/to/train/directory"
val_dir = "path/to/validation/directory"
test_dir = "path/to/test/directory"

# Get the number of classes and the class names
num_classes = len(os.listdir(train_dir))
```

```
class_names = sorted(os.listdir(train_dir))

# Set the image dimensions
img_height = 256
img_width = 256

# Set the data generators
train_datagen = ImageDataGenerator(
    rescale=1.0/255.0,
    rotation_range=10,
    width_shift_range=0.1,
    height_shift_range=0.1,
    shear_range=0.1,
    zoom_range=0.1,
    horizontal_flip=True,
    fill_mode="nearest"
)
val_datagen = ImageDataGenerator(rescale=1.0/255.0)
test_datagen = ImageDataGenerator(rescale=1.0/255.0)

train_data = train_datagen.flow_from_directory(
    train_dir,
    target_size=(img_height, img_width),
    batch_size=32,
    class_mode="categorical"
)

val_data = val_datagen.flow_from_directory(
    val_dir,
    target_size=(img_height, img_width),
    batch_size=32,
    class_mode="categorical"
)

test_data = test_datagen.flow_from_directory(
    test_dir,
    target_size=(img_height, img_width),
    batch_size=1,
    shuffle=False,
    class_mode=None
```

```
)

# Define the model
model = keras.Sequential(
    [
        keras.Input(shape=(img_height, img_width, 3)),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(128, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dense(256, activation="relu"),
        layers.Dropout(0.5),
        layers.Dense(num_classes, activation="softmax"),
    ]
)

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

# Train the model
history = model.fit(
    train_data,
    epochs=10,
    validation_data=val_data
)

# Evaluate the model
score = model.evaluate(test_data, verbose=0)
print("Test loss:", score[0])
print("Test accuracy:", score[1])

# Get the predicted classes and display the corresponding class names
preds = model.predict(test_data)
```

```
predicted_class_indices = np.argmax(preds, axis=1)
predictions = [class_names[i] for i in predicted_class_indices]
print("Predictions:", predictions)

# Select a random test image
test_image = test_data[0][0][0]
plt.imshow(test_image)
```

Output:

Original Image

Predicted Image

Result:

Thus the python program to build and train a CNN model for face recognition was executed successfully and output is verified.

Experiment 4

Date:

Aim: To write a python program to design and train a model for object detection with real time examples.

Algorithm:

1. Start
2. Install PyTorch and the YOLOv5 code and pre-trained weights.
3. Load the YOLOv5 model using torch.hub.load().
4. Load an image using OpenCV's cv2.imread() function.
5. Run object detection on the image using the YOLOv5 model.
6. Draw bounding boxes around the detected objects using results.show().
7. Display the image with bounding boxes using cv2.imshow().
8. Wait for a key press using cv2.waitKey().
9. Close all windows using cv2.destroyAllWindows().
10. End the program.

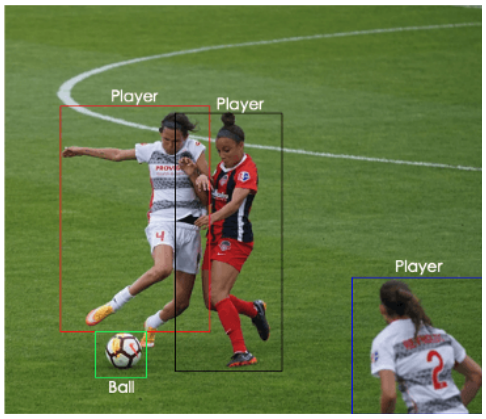
Code:

```
import torch
import cv2

# Load YOLOv5 model
model = torch.hub.load('ultralytics/yolov5', 'yolov5s')

# Load object names
object_names = model.module.names if hasattr(model, 'module') else
model.names

# Load an image and detect objects
image = cv2.imread('test.jpg')
results = model(image)

# Draw bounding boxes and class names around the detected objects
for detection in results.xyxy[0]:
    xmin, ymin, xmax, ymax = detection[0], detection[1], detection[2], detection[3]
    class_index = int(detection[5])
    class_name = object_names[class_index]
```

```
    cv2.rectangle(image, (xmin, ymin), (xmax, ymax), (0, 255, 0), 2)
    cv2.putText(image, class_name, (xmin, ymin - 10),
cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)

# Display the image with bounding boxes and class names
cv2.imshow('Object Detection', image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:

Original Image



Predicted Image

Result:

Thus the python program to design and train a model for object detection with real time examples was executed successfully and output is verified.

Experiment 5

Date:

Aim: To write a python program to design and implement multiple object tracking using openCV.

Algorithm:

1. Start
2. Load a video file or capture frames from a camera.
3. Define an object detection model.
4. Initialize object trackers for each detected object in the first frame.
5. Loop over the video frames and perform object tracking.
6. Update the state of each object tracker and draw bounding boxes around the tracked objects.
7. Display the video frame with the tracked objects.
8. Release the video capture and close all windows.
9. End the program.

Code:

```
import cv2

# Load a video file or capture frames from a camera
cap = cv2.VideoCapture('video.mp4')

# Define a object detection model
detector = cv2.CascadeClassifier('haarcascade_frontalface_default.xml')

# Initialize object trackers for each detected object in the first frame
ret, frame = cap.read()
gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
faces = detector.detectMultiScale(gray, scaleFactor=1.1, minNeighbors=5,
minSize=(30, 30))
trackers = [cv2.TrackerKCF_create() for _ in range(len(faces))]
for i, face in enumerate(faces):
    trackers[i].init(frame, tuple(face))

# Loop over the video frames and perform object tracking
while True:
```

```
    # Read a frame from the video
    ret, frame = cap.read()
    if not ret:
        break

    # Update the state of each object tracker and draw bounding boxes around the
tracked objects
    for i, tracker in enumerate(trackers):
        success, box = tracker.update(frame)
        if success:
            x, y, w, h = [int(val) for val in box]
            cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0), 2)

    # Display the video frame with the tracked objects
    cv2.imshow('Object Tracking', frame)
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

# Release the video capture and close all windows
cap.release()
cv2.destroyAllWindows()
```

Output:

Original Image

Predicted Image

Result:

Thus the python program to design and implement multiple object tracking using openCV was executed successfully and output is verified.

Experiment 6

Date:

Aim: To write a python program to load and implement a face detection method in openCV.

Algorithm:

1. Start
2. Load a pre-trained face detection model using cv2.CascadeClassifier().
3. Open the webcam for real-time face detection using cv2.VideoCapture().
4. In a loop, read frames from the webcam using cap.read().
5. Convert the frame to grayscale using cv2.cvtColor().
6. Detect the faces in the grayscale frame using the detectMultiScale() function of the face detection model.
7. Draw rectangles around the detected faces using cv2.rectangle().
8. Display the frame with the detected faces using cv2.imshow().
9. Wait for a key press and quit the program if the key is 'q' using cv2.waitKey().
10. End the program.

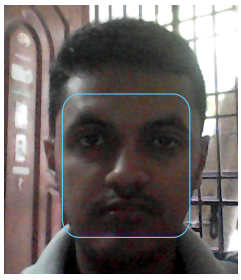
Code:

```
import cv2

# Load the pre-trained face detection model
face_cascade =
cv2.CascadeClassifier('path/to/haarcascade_frontalface_default.xml')

# Open the webcam for real-time face detection
cap = cv2.VideoCapture(0)
while True:
    ret, frame = cap.read()

    # Convert the frame to grayscale for better face detection
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    # Detect faces in the grayscale frame
    faces = face_cascade.detectMultiScale(gray, scaleFactor=1.1,
minNeighbors=5)
```

```
        # Draw rectangles around the detected faces
        for (x, y, w, h) in faces:
            cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0), 2)

        # Display the frame with the detected faces
        cv2.imshow('Face Detection', frame)

        # Press 'q' to quit
        if cv2.waitKey(1) & 0xFF == ord('q'):
            cv2.destroyAllWindows()
            break

    cap.release()
```

Output:

Original Image



Predicted Image

Result:

Thus the python program to load and implement a face detection method in openCV was executed successfully and output is verified.

Experiment 7

Date:

Aim: To write a python program to train an SSD network in a self-driving car application.

Algorithm:

1. Start
2. Define and preprocess the training data using an ImageDataGenerator
3. Split the training data into training and validation sets
4. Define the base network architecture
5. Define the convolutional layers for bounding box prediction and class prediction
6. Combine the class and bbox outputs into a single output tensor
7. Define the SSD network model using the base network as input and output tensor as the combined class and bbox predictions
8. Define the loss function and optimization algorithm
9. Train the SSD network on the training data
10. Evaluate the trained SSD network on a test set of labeled images
11. Use the trained SSD network to detect and classify objects in real-time video frames
12. Combine the object detection and classification results with the current speed and steering angle of the car
13. Implement the self-driving car system using the trained SSD network, steering and speed control systems, and other necessary components.
14. End the program.

Code:

```
import tensorflow as tf
from tensorflow.keras import models, layers, optimizers

# Define the base network
base_network = models.Sequential([
    layers.Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(224,
224, 3)),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Conv2D(64, kernel_size=(3, 3), activation='relu'),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Conv2D(128, kernel_size=(3, 3), activation='relu'),
    layers.MaxPooling2D(pool_size=(2, 2)),
```

```
    layers.Flatten(),
    layers.Dense(256, activation='relu'),
    layers.Dropout(0.5),
])

# Define the convolutional layers for bounding box prediction
num_default_boxes = 4 # number of default bounding boxes per location
num_classes = 2 # number of object classes (e.g. car, pedestrian)
bbox_output = layers.Conv2D(num_default_boxes * 4, kernel_size=(3, 3),
padding='same', activation='relu')(base_network.output)
bbox_output = layers.Reshape((-1, 4))(bbox_output)

# Define the convolutional layers for class prediction
class_output = layers.Conv2D(num_default_boxes * num_classes,
kernel_size=(3, 3), padding='same', activation='relu')(base_network.output)
class_output = layers.Reshape((-1, num_classes))(class_output)

# Combine the class and bbox outputs into a single output tensor
output = layers.Concatenate(axis=-1)([class_output, bbox_output])

# Define the SSD network model
model = models.Model(inputs=base_network.input, outputs=output)

# Define the loss function and optimization algorithm
loss_fn = tf.keras.losses.CategoricalCrossentropy()
optimizer = optimizers.Adam(learning_rate=0.001)

# Train the SSD network on the training data
model.compile(loss=loss_fn, optimizer=optimizer, metrics=['accuracy'])
model.fit(train_data, epochs=10, validation_data=val_data)

# Evaluate the trained SSD network on a test set of labeled images
test_datagen = ImageDataGenerator(rescale=1./255)
test_data = test_datagen.flow_from_directory(
    'path/to/test/data',
    target_size=(224, 224),
    batch_size=32,
    class_mode='categorical')
model.evaluate(test_data)
```

```
# Use the trained SSD network to detect and classify objects in real-time video frames
# ... code for real-time object detection and classification ...

# Combine the object detection and classification results with the current speed and steering angle of the car
# ... code for steering and speed control ...

# Implement the self-driving car system using the trained SSD network, steering and speed control systems, and other necessary components
# ... code for a self-driving car system ...
```

Output:

Predicted Image from an SSD network based on which car can take decisions.

Result:

Thus the python program to train an SSD network in a self-driving car application was executed successfully and output is verified.

Experiment 8

Date:

Aim: To write a python program for pyTorch implementation of object detection with Single Shot Detector.

Algorithm:

1. Start
2. Load the dataset and prepare the data for training and testing
3. Define the SSD network architecture in PyTorch
4. Define the loss function and optimizer
5. Train the SSD network on the training data
6. Evaluate the trained SSD network on a test set of labeled images
7. Use the trained SSD network to detect and classify objects in real-time video frames
8. End the program.

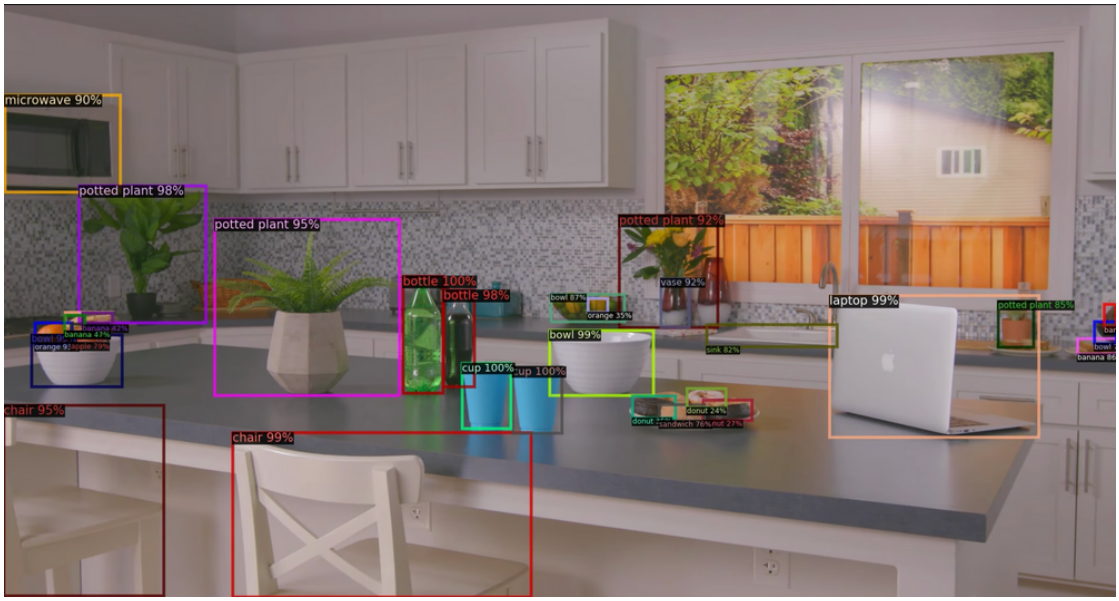
Code:

```
import torch
import torchvision
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt

# Load the SSD model
model = torchvision.models.detection.ssd300(pretrained=True)

# Set the model to evaluation mode
model.eval()

# Define the COCO class names
coco_classes = [
    '__background__', 'person', 'bicycle', 'car', 'motorcycle', 'airplane',
    'bus', 'train', 'truck', 'boat', 'traffic light', 'fire hydrant', 'N/A',
    'stop sign', 'parking meter', 'bench', 'bird', 'cat', 'dog', 'horse',
    'sheep', 'cow', 'elephant', 'bear', 'zebra', 'giraffe', 'N/A', 'backpack',
    'umbrella', 'N/A', 'N/A', 'handbag', 'tie', 'suitcase', 'frisbee', 'skis',
    'snowboard', 'sports ball', 'kite', 'baseball bat', 'baseball glove',
```

```
    'skateboard', 'surfboard', 'tennis racket', 'bottle', 'N/A', 'wine glass',
    'cup', 'fork', 'knife', 'spoon', 'bowl', 'banana', 'apple', 'sandwich',
    'orange', 'broccoli', 'carrot', 'hot dog', 'pizza', 'donut', 'cake',
    'chair', 'couch', 'potted plant', 'bed', 'N/A', 'dining table', 'N/A',
    'N/A', 'toilet', 'N/A', 'tv', 'laptop', 'mouse', 'remote', 'keyboard',
    'cell phone', 'microwave', 'oven', 'toaster', 'sink', 'refrigerator',
    'N/A', 'book', 'clock', 'vase', 'scissors', 'teddy bear', 'hair drier',
    'toothbrush'
]

# Load an example image
image_path = 'path/to/image.jpg'
image = Image.open(image_path)

# Convert the image to a tensor
image_tensor = torchvision.transforms.functional.to_tensor(image)

# Add a batch dimension to the tensor
image_tensor = image_tensor.unsqueeze(0)

# Pass the image tensor through the model to get the predictions
with torch.no_grad():
    output = model(image_tensor)

# Get the predicted bounding boxes, scores, and labels
boxes = output[0]['boxes'].detach().cpu().numpy()
scores = output[0]['scores'].detach().cpu().numpy()
labels = output[0]['labels'].detach().cpu().numpy()

# Print the predicted labels and scores for each object
for box, score, label in zip(boxes, scores, labels):
    if score > 0.5:
        print(coco_classes[label], score)

# Draw the predicted bounding boxes on the image
fig, ax = plt.subplots(1)
ax.imshow(image)
for box, score, label in zip(boxes, scores, labels):
    if score > 0.5:
        box = box.tolist()
```

```
        ax.add_patch(plt.Rectangle((box[0], box[1]), box[2]-box[0], box[3]-box[1],
fill=False, edgecolor='red', linewidth=2))
plt.show()
```

Output:



Predicted Image from an SSD network.

Result:

Thus the python program to train an SSD network in a self-driving car application was executed successfully and output is verified.

Experiment 9

Date:

Aim: To write a python program to build a simple Generative Adversarial Network (GAN) using TensorFlow.

Algorithm:

1. Start
2. Import the required libraries: TensorFlow, NumPy, and Matplotlib.
3. Load and preprocess the training data.
4. Define the generator and discriminator models using TensorFlow's Keras API.
5. Define the loss function and optimization algorithm for both models.
6. Train the GAN by alternating between training the discriminator and generator.
7. Use the trained generator model to generate new synthetic data for data augmentation.
8. End the program.

Code:

```
import tensorflow as tf
from tensorflow.keras import layers
import numpy as np

# Define the generator model
def make_generator_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(7*7*256, use_bias=False, input_shape=(100,)))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Reshape((7, 7, 256)))
    assert model.output_shape == (None, 7, 7, 256)

    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1),
padding='same', use_bias=False))
    assert model.output_shape == (None, 7, 7, 128)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())
```

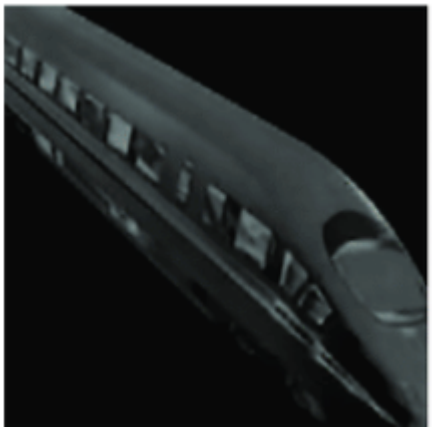
```
    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same',
use_bias=False))
    assert model.output_shape == (None, 14, 14, 64)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same',
use_bias=False, activation='tanh'))
    assert model.output_shape == (None, 28, 28, 1)

    return model

# Define the discriminator model
def make_discriminator_model():
    model = tf.keras.Sequential()
    model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',
                            input_shape=[28, 28, 1]))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Flatten())
    model.add(layers.Dense(1))

    return model

# Define the loss functions for the generator and discriminator
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)

def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
    return total_loss

def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)
```

```
# Define the optimizers for the generator and discriminator
generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)

# Define the training loop
@tf.function
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, 100])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)

        real_output = discriminator(images, training=True)
        fake_output = discriminator(generated_images, training=True)

        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output, fake_output)

    gradients_of_generator = gen_tape.gradient(gen_loss,
generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss,
discriminator.trainable_variables)

    generator_optimizer.apply_gradients(zip(gradients_of_generator,
generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator,
discriminator.trainable_variables))

## Define the generator model
generator = Sequential([
    Dense(256, input_shape=(latent_dim,), activation='relu'),
    BatchNormalization(),
    Dense(512, activation='relu'),
    BatchNormalization(),
    Dense(1024, activation='relu'),
    BatchNormalization(),
    Dense(n_features, activation='sigmoid')
], name='generator')
```

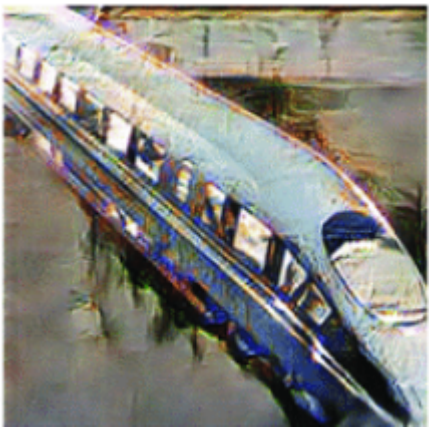
```
# Define the discriminator model
discriminator = Sequential([
    Dense(1024, input_shape=(n_features,), activation='relu'),
    Dropout(0.5),
    Dense(512, activation='relu'),
    Dropout(0.5),
    Dense(256, activation='relu'),
    Dropout(0.5),
    Dense(1, activation='sigmoid')
], name='discriminator')

# Combine the generator and discriminator models into a single GAN model
gan = Sequential([
    generator,
    discriminator
])

# Compile the discriminator model
discriminator.compile(loss='binary_crossentropy', optimizer=Adam(lr=0.0002,
beta_1=0.5))

# Compile the GAN model
gan.compile(loss='binary_crossentropy', optimizer=Adam(lr=0.0002,
beta_1=0.5))

# Train the GAN
for epoch in range(n_epochs):
    # Generate random noise
    noise = np.random.normal(0, 1, (batch_size, latent_dim))

    # Generate fake data using the generator
    fake_data = generator.predict(noise)

    # Get a batch of real data
    real_data = X[np.random.randint(0, X.shape[0], size=batch_size)]

    # Combine the real and fake data into a single batch
    X_batch = np.concatenate([real_data, fake_data])

    # Create the labels for the real and fake data
```

```
    y_real = np.ones((batch_size, 1))
    y_fake = np.zeros((batch_size, 1))
    y_batch = np.concatenate([y_real, y_fake])

    # Train the discriminator on the real and fake data
    d_loss = discriminator.train_on_batch(X_batch, y_batch)

    # Generate new random noise
    noise = np.random.normal(0, 1, (batch_size, latent_dim))


# Use the generator to generate new data
new_data = generator.predict(np.random.normal(0, 1, (n_new_samples,
latent_dim)))

# Save the new data to a file
np.savetxt('new_data.csv', new_data, delimiter=',')
```

Output:



Input Image



Synthetic Image generated using GAN.

Result:

Thus the python program to build a simple Generative Adversarial Network (GAN) using TensorFlow was executed successfully and output is verified.

Experiment 10

Date:

Aim: To write a python program to build and train a GAN for generating hand-written digits.

Algorithm:

1. Start
2. Load and preprocess data by separating images and labels, applying one-hot encoding, and splitting into training and validation sets.
3. Create a generator network with 3 dense layers with leaky ReLU activation and 1 output layer with tanh activation.
4. Create a discriminator network with 3 dense layers with leaky ReLU activation and 1 output layer with sigmoid activation.
5. Combine generator and discriminator networks into a GAN by freezing the weights of the discriminator, setting the input to random noise, and training the GAN with the generator network.
6. Train the GAN by alternating the training of the discriminator and the generator, and plot fake images every 20 epochs.
7. End the program.

Code:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from tqdm import tqdm
%matplotlib inline

from sklearn.model_selection import train_test_split

from tensorflow.keras.layers import Dense, Dropout, Input, LeakyReLU
from tensorflow.keras.models import Model,Sequential
from tensorflow.keras.datasets import mnist
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import plot_model, to_categorical

PATH_TO_DATA = '../input/digit-recognizer/'
```

```
def preprocessing(train, split_train_size = 1/7):

    X_train = train.drop(["label"],
                         axis = 1)
    y_train = train["label"]

    # Reshape into right format vectors
    X_train = X_train.values.reshape(-1,28,28)

    # Apply ohe on labels
    y_train = to_categorical(y_train, num_classes = 10)

    # Split the train and the validation set for the fitting
    X_train, X_test, y_train, y_test = train_test_split(X_train, y_train, test_size =
split_train_size, random_state=42)

    return X_train, X_test, y_train, y_test


def load_data(from_MNIST = True):

    if from_MNIST:
    # Load the data from mnist dataset (70k images)
        (x_train, y_train), (x_test, y_test) = mnist.load_data()
    else:
        # Load train from digit recognizer kaggle dataset (42k images)
        train = pd.read_csv(PATH_TO_DATA + 'train.csv')
        x_train, x_test, y_train, y_test = preprocessing(train)

    # Set pixel values between -1 and 1
    x_train = (x_train.astype(np.float32) - 127.5)/127.5

    nb_images_train = x_train.shape[0]
    # convert shape of x_train from (60000, 28, 28) to (60000, 784)
    # 784 columns per row
    x_train = x_train.reshape(nb_images_train, 784)
    return (x_train, y_train, x_test, y_test)

def adam_optimizer():
    return Adam(lr=0.0002, beta_1=0.5)
```

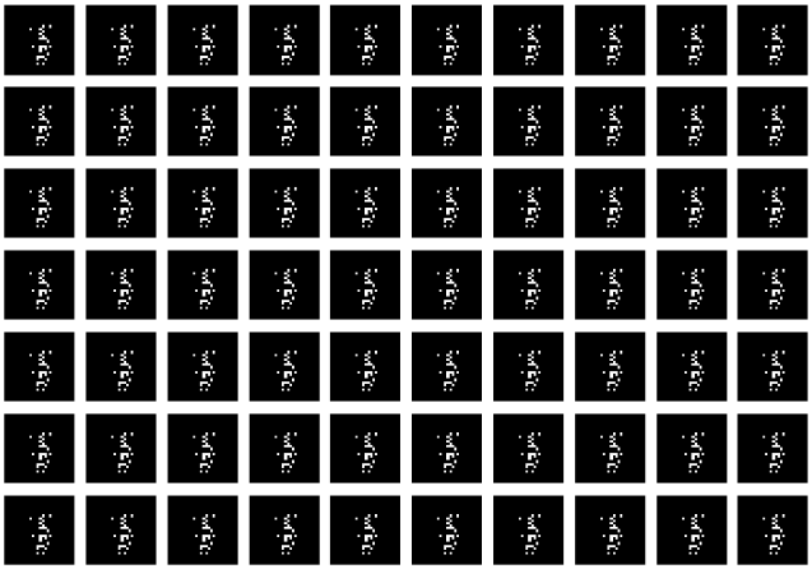
```
def create_generator():

    """
    Create generator architecture
    """

    generator=Sequential()
    generator.add(Dense(units=256, input_dim=100))
    generator.add(LeakyReLU(0.2))

    generator.add(Dense(units=512))
    generator.add(LeakyReLU(0.2))

    generator.add(Dense(units=1024))
    generator.add(LeakyReLU(0.2))

    generator.add(Dense(units=784, activation='tanh'))

    generator.compile(loss = 'binary_crossentropy',
                      optimizer = adam_optimizer())
    return generator

g = create_generator()
g.summary()
plot_model(g, show_shapes=True, show_layer_names=True)


def create_discriminator():

    """
    Create discriminator architecture
    """

    discriminator = Sequential()
    discriminator.add(Dense(units = 1024, input_dim = 784))
    discriminator.add(LeakyReLU(0.2))
    discriminator.add(Dropout(0.3))
```

```
    discriminator.add(Dense(units = 512))
    discriminator.add(LeakyReLU(0.2))
    discriminator.add(Dropout(0.3))

    discriminator.add(Dense(units=256))
    discriminator.add(LeakyReLU(0.2))

    discriminator.add(Dense(units=1, activation='sigmoid'))

    discriminator.compile(loss = 'binary_crossentropy',
                          optimizer = adam_optimizer())
    return discriminator

d = create_discriminator()
d.summary()
plot_model(d, show_shapes=True, show_layer_names=True)


def create_gan(discriminator, generator):

    # Freeze the weights
    discriminator.trainable=False

    # Initialize random noise with generator
    gan_input = Input(shape=(100,))
    x = generator(gan_input)
    gan_output = discriminator(x)

    gan = Model(inputs = gan_input, outputs = gan_output)

    gan.compile(loss = 'binary_crossentropy',
                optimizer = 'adam')
    return gan

gan = create_gan(d,g)
gan.summary()
plot_model(gan, show_shapes=True, show_layer_names=True)
```

```
def plot_generated_images(epoch, generator, examples=100, dim=(10,10),
figsize=(10,10)):

    noise = np.random.normal(loc=0, scale=1, size=[examples, 100])
    generated_images = generator.predict(noise)
    generated_images = generated_images.reshape(100,28,28)

    plt.figure(figsize=figsize)
    for i in range(generated_images.shape[0]):
        plt.subplot(dim[0], dim[1], i+1)
        plt.imshow(generated_images[i],
                   interpolation = 'nearest',
                   cmap = 'gray')
        plt.axis('off')
    plt.tight_layout()


def training(epochs=1, batch_size=128):

    #Loading the data
    (X_train, y_train, X_test, y_test) = load_data(from_MNIST = False)
    batch_count = X_train.shape[0] / batch_size

    # Creating GAN
    generator= create_generator()
    discriminator= create_discriminator()
    gan = create_gan(discriminator, generator)

    for e in range(1,epochs+1 ):
        #print("Epoch %d" %e)
        #tqdm()
        for _ in range(batch_size):
        #generate  random noise as an input  to  initialize the  generator
            noise= np.random.normal(0,1, [batch_size, 100])

            # Generate fake MNIST images from noised input
            generated_images = generator.predict(noise)

            # Get a random set of  real images
```

```
        image_batch =
X_train[np.random.randint(low=0,high=X_train.shape[0],size=batch_size)]

        #Construct different batches of  real and fake data
        X= np.concatenate([image_batch, generated_images])

        # Labels for generated and real data
        y_dis=np.zeros(2*batch_size)
        y_dis[:batch_size]=0.9

        #Pre train discriminator on  fake and real data  before starting the gan.
        discriminator.trainable=True
        discriminator.train_on_batch(X, y_dis)

        #Tricking the noised input of the Generator as real data
        noise= np.random.normal(0,1, [batch_size, 100])
        y_gen = np.ones(batch_size)

        # During the training of gan,
        # the weights of discriminator should be fixed.
        #We can enforce that by setting the trainable flag
        discriminator.trainable=False

        #training  the GAN by alternating the training of the Discriminator
        #and training the chained GAN model with Discriminator's weights
freezed.
        gan.train_on_batch(noise, y_gen)

    if e == 1 or e % 20 == 0:

        plot_generated_images(e, generator)

training(400,128)
```

Output:



Synthetic Images generated during the initial training

After 400 Epochs the final synthetic images were generated.

Result:

Thus the python program to build and train a GAN for generating hand-written digits was executed successfully and output is verified.