Aim: Write a Python program to implement CNN object detection. Discuss numerous performance evaluation metrics for evaluating the object detecting algorithm performance.

Software requirement for Python: Jupyter Notebook

Theory:

Object detection is a computer vision technique whose aim is to detect objects such as cars, buildings, and human beings, just to mention a few. The objects can generally be identified from either pictures or video feeds.

Object detection has been applied widely in video surveillance, self-driving cars, and object/people tracking. In this piece, we'll look at the basics of object detection and review some of the most commonly-used algorithms and a few brand new approaches, as well.

How Object Detection Works

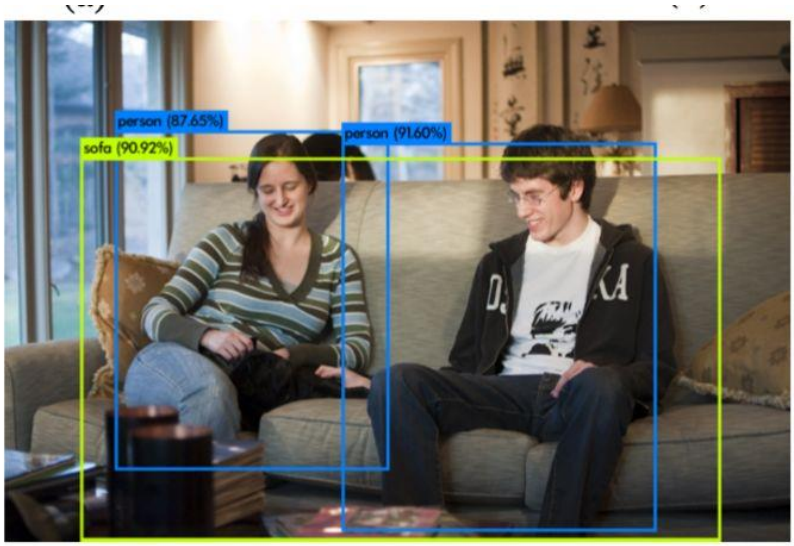
Object detection locates the presence of an object in an image and draws a bounding box around that object. This usually involves two processes; classifying and object's type, and then drawing a box around that object. Some of the common model architectures used for object detection:

1. R-CNN
2. Fast R-CNN
3. Faster R-CNN
4. Mask R-CNN
5. SSD (Single Shot MultiBox Defender)
6. YOLO (You Only Look Once)
7. Objects as Points
8. Data Augmentation Strategies for Object Detection

Once you have trained your first object detector, the next step is to know its performance. Sure enough, you can see the model finds all the objects in the pictures you feed it. Great! But how do you quantify that? How should we decide which model is better?

Since the classification task only evaluates the probability of the class object appearing in the image, it is a straightforward task for a classifier to identify correct predictions from incorrect ones. However, the object detection task localizes the object further with a bounding box associated with its corresponding confidence score to report how certain the bounding box of the object class is detected.

A detector outcome is commonly composed of a list of bounding boxes, confidence levels and classes, as seen in the following Figure:



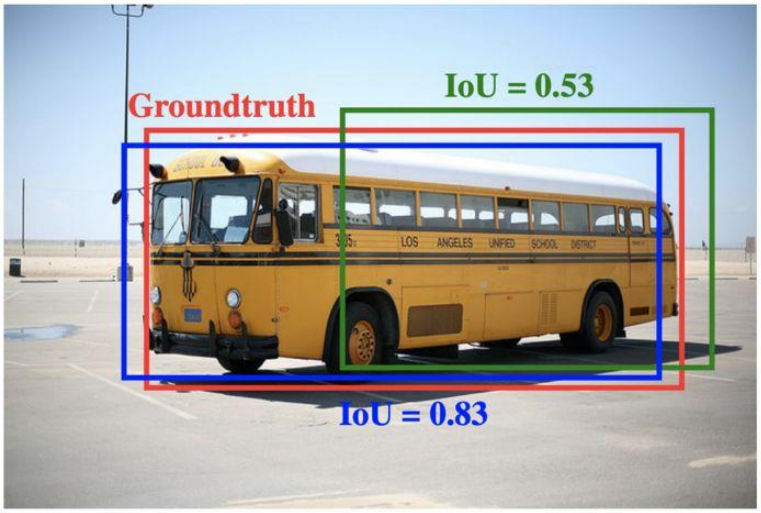
Object detection metrics serve as a measure to assess how well the model performs on an object detection task. It also enables us to compare multiple detection systems objectively or compare them to a benchmark. In most competitions, the average precision (AP) and its derivations are the metrics adopted to assess the detections and thus rank the teams.

Understanding the various metric:

IoU: Guiding principle in all state-of-the-art metrics is the so-called Intersection-over-Union (IoU) overlap measure. It is quite literally defined as the intersection over union of the detection bounding box and the ground truth bounding box.

Dividing the area of overlap between predicted bounding box and ground truth by the area of their union yields the **Intersection over Union (IoU).**

An Intersection over Union score > 0.5 is normally considered a “good” prediction.



IoU metric determines how many objects were detected correctly and how many false positives were generated (will be discussed below).

True Positives [TP]

Number of detections with IoU>0.5

False Positives [FP]

Number of detections with IoU<=0.5 or detected more than once

False Negatives [FN]

Number of objects that not detected or detected with IoU<=0.5

Precision

Precision measures how accurate your predictions are. i.e. the percentage of your predictions that are correct.

Precision = True positive / (True positive + False positive)

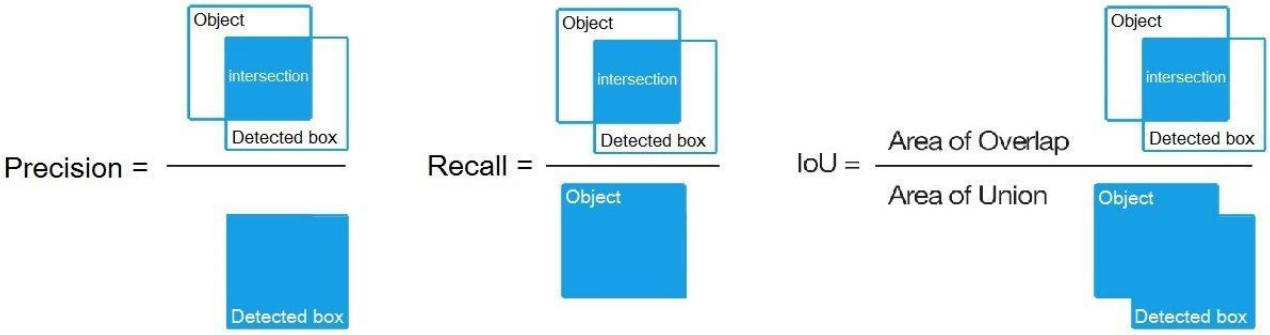
Recall

Recall measures how good you find all the positives.

Recall = True positive / (True positive + False negative)

F1 Score

F1 score is HM (Harmonic Mean) of precision and recall.

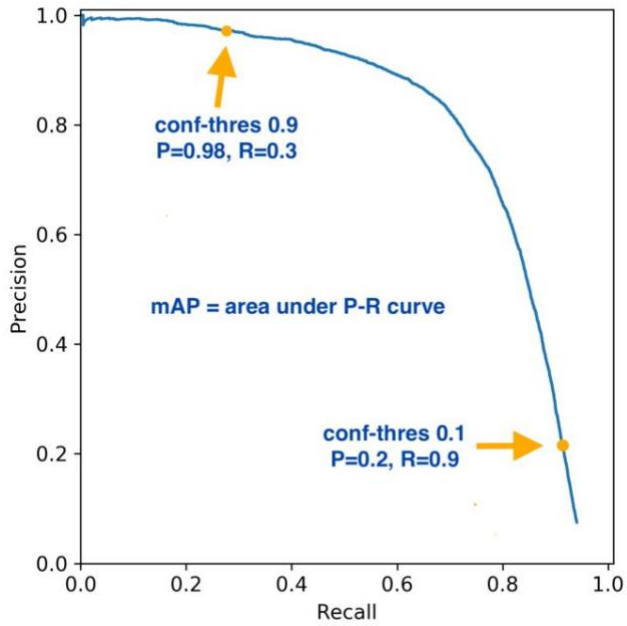


Average Precision (AP)

The general definition for the Average Precision (AP) is finding the area under the precision-recall curve.

Map

The mAP for object detection is the average of the AP calculated for all the classes. mAP@0.5 means that it is the mAP calculated at IOU threshold 0.5.



Procedure:

This code defines and trains a Convolutional Neural Network (CNN) model using the CIFAR-10 dataset. The CNN architecture consists of several convolutional layers, followed by max-pooling and dense layers. The model is trained to classify images as either containing an object of interest (in this case, a specific type of object labeled as 2) or not.

The code first imports necessary libraries such as NumPy, TensorFlow, and its Keras API for defining the CNN model architecture. Then, it defines the CNN model using the Keras functional API, which includes convolutional, max-pooling, and dense layers. The dataset (CIFAR-10) is loaded and preprocessed by normalizing the pixel values and transforming the labels to binary values (1 for the object of interest and 0 otherwise).

The model is then compiled with Adam optimizer, binary cross-entropy loss function, and several metrics such as accuracy, precision, and recall. Finally, the model is trained for a fixed number of epochs using the training data and evaluated using the test data. The results are then printed, which include the test loss, accuracy, precision, and recall of the model.

Conclusion: Thus, we have studied to implement CNN object detection.

Program Code and Output

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import Model
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Dense, Flatten, Input

# Define the CNN model
def create_model():
    input_tensor = Input(shape=(32, 32, 3))
    x = Conv2D(32, (3, 3), activation='relu', padding='same')(input_tensor)
    x = MaxPooling2D((2, 2))(x)
    x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D((2, 2))(x)
    x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D((2, 2))(x)
    x = Flatten()(x)
    x = Dense(256, activation='relu')(x)
    x = Dense(1, activation='sigmoid')(x)
    model = Model(inputs=input_tensor, outputs=x)
    return model

# Load the dataset
(X_train, y_train), (X_test, y_test) = tf.keras.datasets.cifar10.load_data()

# Normalize the images
X_train = X_train / 255.0
X_test = X_test / 255.0
```

```
# Define the object detection labels
y_train = (y_train == 2).astype(int)
y_test = (y_test == 2).astype(int)

# Create the CNN model
model = create_model()

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

# Train the model
model.fit(X_train, y_train, epochs=10, batch_size=64, validation_data=(X_test, y_test))

# Evaluate the model
results = model.evaluate(X_test, y_test, batch_size=64)
print("Test loss, Test accuracy, Test precision, Test recall:", results)
```

OUTPUT OF THE CODE

Epoch 1/10

782/782 [==============================] - 43s 53ms/step - loss: 0.2696 - accuracy: 0.9019 - precision: 0.6091 - recall: 0.0536 - val_loss: 0.2391 - val_accuracy: 0.9099 - val_precision: 0.6334 - val_recall: 0.2350

Epoch 2/10

782/782 [==============================] - 45s 58ms/step - loss: 0.2278 - accuracy: 0.9136 - precision: 0.6854 - recall: 0.2506 - val_loss: 0.2237 - val_accuracy: 0.9151 - val_precision: 0.6982 - val_recall: 0.2660

Epoch 3/10

782/782 [==============================] - 52s 67ms/step - loss: 0.2027 - accuracy: 0.9244 - precision: 0.7276 - recall: 0.3894 - val_loss: 0.2016 - val_accuracy: 0.9233 - val_precision: 0.7271 - val_recall: 0.3730

Epoch 4/10

782/782 [==============================] - 70s 89ms/step - loss: 0.1809 - accuracy: 0.9326 - precision: 0.7587 - recall: 0.4774 - val_loss: 0.2005 - val_accuracy: 0.9262 - val_precision: 0.7937 - val_recall: 0.3540

Epoch 5/10

782/782 [==============================] - 74s 95ms/step - loss: 0.1639 - accuracy: 0.9380 - precision: 0.7781 - recall: 0.5322 - val_loss: 0.1867 - val_accuracy: 0.9308 - val_precision: 0.7541 - val_recall: 0.4570

Epoch 6/10

782/782 [==============================] - 65s 83ms/step - loss: 0.1448 - accuracy: 0.9439 - precision: 0.7917 - recall: 0.5952 - val_loss: 0.1924 - val_accuracy: 0.9286 - val_precision: 0.7449 - val_recall: 0.4350

Epoch 7/10

782/782 [==============================] - 67s 86ms/step - loss: 0.1265 - accuracy: 0.9518 - precision: 0.8201 - recall: 0.6636 - val_loss: 0.2067 - val_accuracy: 0.9225 - val_precision: 0.6144 - val_recall: 0.6040

Epoch 8/10

782/782 [==============================] - 67s 85ms/step - loss: 0.1073 - accuracy: 0.9590 - precision: 0.8454 - recall: 0.7220 - val_loss: 0.2137 - val_accuracy: 0.9204 - val_precision: 0.5951 - val_recall: 0.6380

Epoch 9/10

782/782 [==============================] - 67s 85ms/step - loss: 0.0900 - accuracy: 0.9657 - precision: 0.8710 - recall: 0.7708 - val_loss: 0.2167 - val_accuracy: 0.9264 - val_precision: 0.6507 - val_recall: 0.5700

Epoch 10/10

782/782 [==============================] - 65s 83ms/step - loss: 0.0746 - accuracy: 0.9715 - precision: 0.8864 - recall: 0.8196 - val_loss: 0.2465 - val_accuracy: 0.9316 - val_precision: 0.7416 - val_recall: 0.4850

157/157 [==============================] - 4s 24ms/step - loss: 0.2465 - accuracy: 0.9316 - precision: 0.7416 - recall: 0.4850

Test loss, Test accuracy, Test precision, Test recall: [0.2465059608221054, 0.9315999746322632, 0.7415902018547058, 0.48500001430511475]