TASK 3

Comparison between CNN and YOLO in terms of key characteristics and application areas:

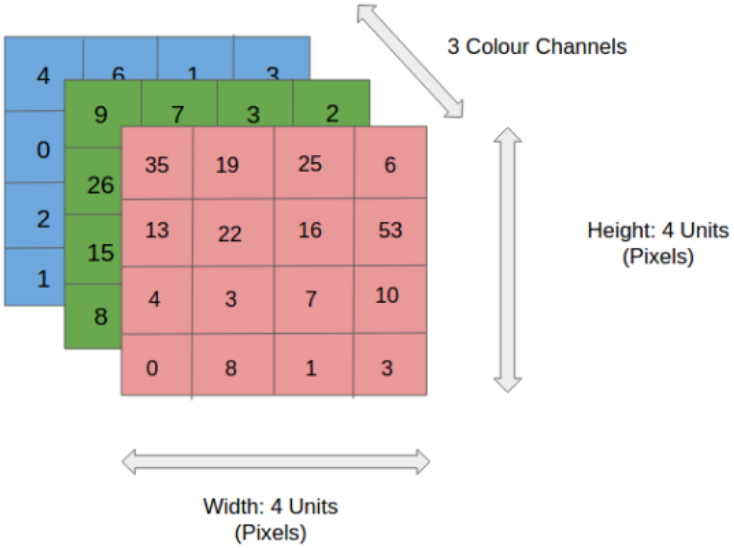
1. **Object Detection Approach:**
    - CNN: CNNs are primarily used for image classification tasks, where the goal is to classify the entire image into different predefined categories. However, CNNs can also be used for object detection by applying additional techniques like region proposal methods (e.g., R-CNN, Fast R-CNN) or sliding window approaches.
    - YOLO: YOLO is specifically designed for object detection. It directly predicts bounding box coordinates and class probabilities for multiple objects within an image. YOLO operates in a single pass through the network, making it faster and more efficient compared to CNN-based object detection methods.
2. **Inference Speed:**
    - CNN: Traditional CNN-based object detection methods, such as R-CNN and its variants, require multiple passes through the network for region proposals, feature extraction, and classification. This makes them slower in terms of inference speed, limiting their real-time application.
    - YOLO: YOLO is known for its real-time object detection capabilities. By processing images in a single pass, YOLO achieves faster inference speeds, making it suitable for applications that require real-time or near real-time performance.
3. **Accuracy:**
    - CNN: CNN-based object detection methods generally achieve high accuracy, especially when combined with region proposal techniques. By leveraging the power of deep learning and multi-scale feature representations, CNNs can effectively capture complex object patterns and achieve state-of-the-art accuracy.
    - YOLO: YOLO offers a good trade-off between accuracy and speed. While YOLO may have slightly lower accuracy compared to some other object detection methods, it still performs well in detecting and localizing objects accurately. YOLO's strength lies in its ability to provide real-time object detection with reasonable accuracy.
4. **Localization Precision:**

- CNN: CNNs excel in precise localization of objects. By utilizing various convolutional layers and feature hierarchies, CNNs can accurately predict object bounding boxes with fine-grained details, making them suitable for applications that require precise object localization.
- YOLO: YOLO performs well in localizing objects accurately, even for small or densely packed objects. However, due to the fixed grid and limited spatial resolution, YOLO may struggle with detecting very small objects or objects with low contrast.

5. **Application Areas:**
    - CNN: CNNs are widely used for image classification tasks, such as recognizing objects in images, face recognition, medical imaging analysis, and natural language processing tasks like sentiment analysis on images.
    - YOLO: YOLO is well-suited for real-time object detection applications, including video surveillance, autonomous driving, robotics, and any application that requires fast and accurate object detection in real-time or near real-time scenarios.

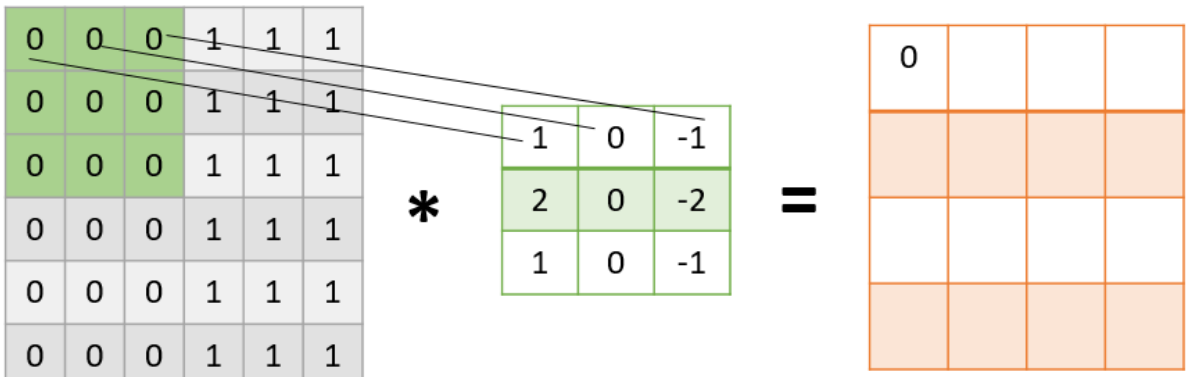
Different layers of CNN:

- **Input Layer:**
  It represents the input image and can be Grayscale or RGB. Every image is made up of pixels that range from 0 to 255. We need to normalize them i.e convert the range between 0 to 1 before passing it to the model.

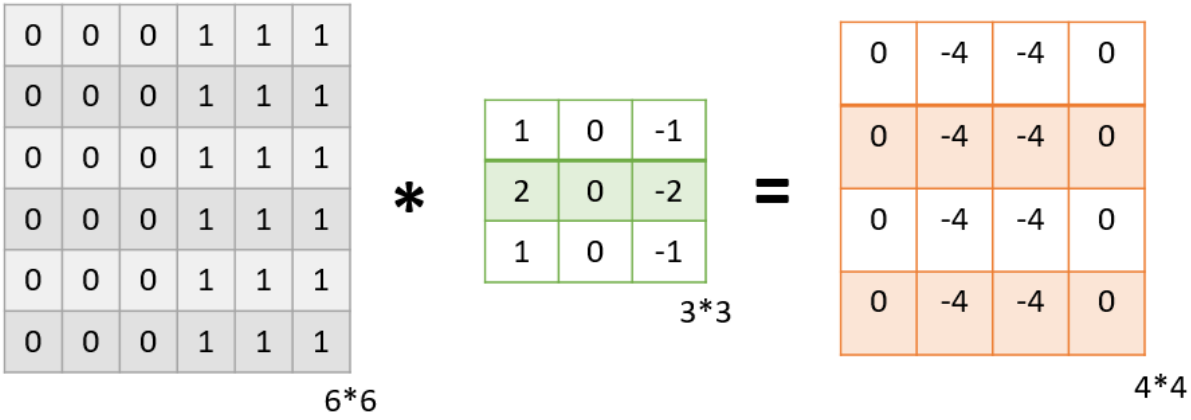


- **Convolution Layer:**
  The convolution layer is the layer where the filter is applied to our input image to extract or detect its features. A filter is applied to the image multiple times and creates a feature map which helps in classifying the input image.

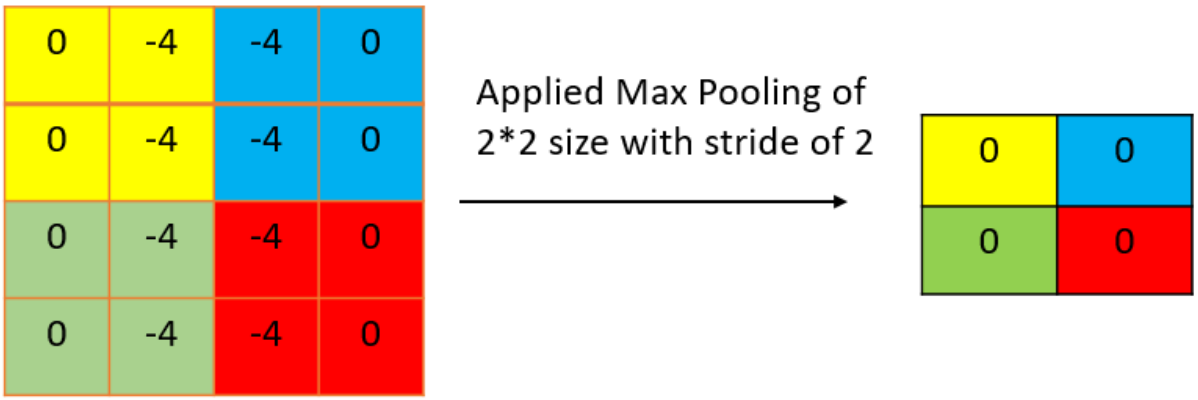
  In the below figure, we have an input image of size 6*6 and applied a filter of 3*3 on it to detect some features. In this example, we have applied only one filter but in practice, many such filters are applied to extract information from the image.



Calculation:
0*1 + 0*0 + 0*-1 +
0*2 + 0*0 + 0*-2 +
0*1 + 0*0 + 0*-1

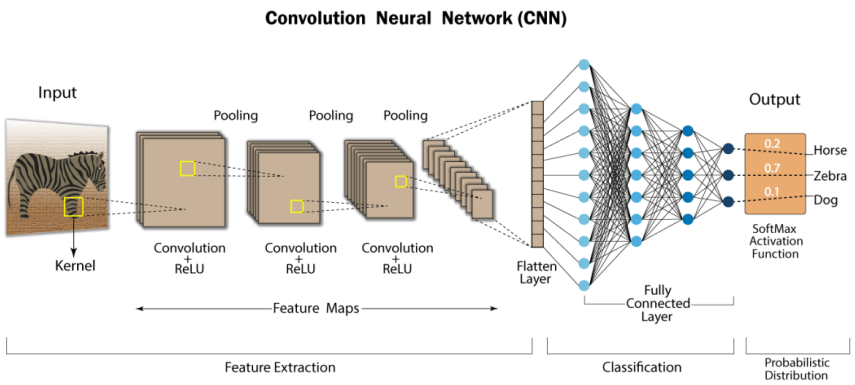


- **Pooling Layers:**
  The pooling layer is applied after the Convolutional layer and is used to reduce the dimensions of the feature map which helps in preserving the important information or features of the input image and reduces the computation time. The most common types of Pooling are Max Pooling and Average Pooling.



- **Fully Connected Layer:**
  The Fully connected layer is used for classifying the input image into a label. This layer connects the information extracted from the previous steps (i.e Convolution layer and Pooling layers) to the output layer and eventually classifies the input into the desired label. ReLU activations functions are applied to them for non-linearity.

The complete process of a CNN model can be seen in the below image.



Implementation of CNN:

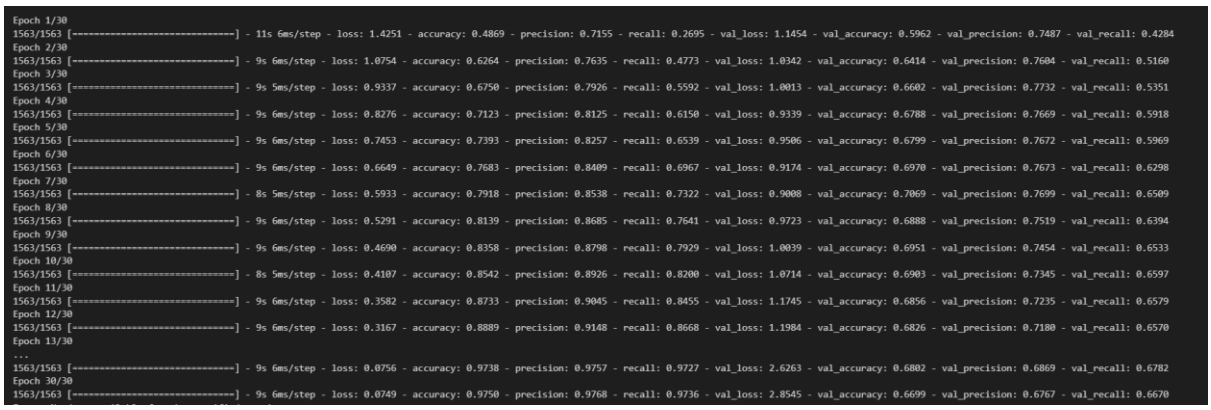
The below code uses tensorflow for training the model. For the dataset, the built in CIFAR10 dataset from keras has been used.

```
import tensorflow as tf
import numpy as np
from keras.datasets import cifar10 as cf10
from keras.utils import to_categorical
from keras import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from keras.metrics import Precision, Recall
import matplotlib.pyplot as plt

(train_images, train_labels), (test_images, test_labels) = cf10.load_data()

def show_images(train_images,
                class_names,
                train_labels,
                nb_samples = 12, nb_row = 4):
  plt.figure(figsize=(12, 12))
  for i in range(nb_samples):
    plt.subplot(nb_row, nb_row, i + 1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[train_labels[i][0]])
  plt.show()

class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog',
'horse', 'ship', 'truck']
show_images(train_images, class_names, train_labels)

# Normalizing
train_images = train_images / 255
test_images = test_images / 255

# Convert labels to numerical format
train_labels = to_categorical(train_labels, len(class_names))
test_labels = to_categorical(test_labels, len(class_names))

# Variables
INPUT_SHAPE = (32, 32, 3)
FILTER1_SIZE = 32
FILTER2_SIZE = 64
FILTER_SHAPE = (3, 3)
POOL_SHAPE = (2, 2)
FULLY_CONNECT_NUM = 128
```

```
NUM_CLASSES = len(class_names)

BATCH_SIZE = 32
EPOCHS = 30
METRICS = metrics=['accuracy', Precision(name='precision'),
Recall(name='recall')]

# Model architecture implementation
model = Sequential()
model.add(Conv2D(FILTER1_SIZE, FILTER_SHAPE, activation='relu',
input_shape=INPUT_SHAPE))
model.add(MaxPooling2D(POOL_SHAPE))
model.add(Conv2D(FILTER2_SIZE, FILTER_SHAPE, activation='relu'))
model.add(MaxPooling2D(POOL_SHAPE))
model.add(Flatten())
model.add(Dense(FULLY_CONNECT_NUM, activation='relu'))
model.add(Dense(NUM_CLASSES, activation='softmax'))

# Model Training

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics =
METRICS)
training_history = model.fit(train_images, train_labels,
                  epochs=EPOCHS, batch_size=BATCH_SIZE,
                  validation_data=(test_images, test_labels))


# Model evaluation by comparing both training and testing dataset to get the
optimal value

def show_performance_curve(training_result, metric, metric_label):
  train_perf = training_result.history[str(metric)]
  validation_perf = training_result.history['val_'+str(metric)]
  i = np.argwhere(np.isclose(train_perf, validation_perf, atol=1e-
2)).flatten()
  if len(i) > 0:
    intersection_idx = i[0]
    intersection_value = train_perf[intersection_idx]

    plt.plot(train_perf, label=metric_label)
    plt.plot(validation_perf, label = 'val_'+str(metric))
    plt.axvline(x=intersection_idx, color='r', linestyle='--',
label='Intersection')

    plt.annotate(f'Optimal Value: {intersection_value:.4f}',
            xy=(intersection_idx, intersection_value),
            xycoords='data',
            fontsize=10,
            color='green')
```

```
    plt.xlabel('Epoch')
    plt.ylabel(metric_label)
    plt.legend(loc='lower right')
  else:
    print("Error: No data available for the specified metric.")

show_performance_curve(training_history, 'accuracy', 'accuracy')
```

Output(Accuracy):



YOLO Implementation:

The below code uses YOLO from the ultralytics library in python.

The pretrained weights, config and object names can be obtained by downloading them as follows:

```
!wget https://github.com/AlexeyAB/darknet/releases/download/darknet_yolo_v3_optimal/yolov4.weights
!wget https://raw.githubusercontent.com/AlexeyAB/darknet/master/cfg/yolov4.cfg
!wget https://raw.githubusercontent.com/AlexeyAB/darknet/master/cfg/coco.names
```

Use the below code:

```
import cv2
import numpy as np
from google.colab.patches import cv2_imshow

# read input image
image = cv2.imread("/content/sample.jpg")

Width = image.shape[1]
Height = image.shape[0]
scale = 0.00392

# read class names from text file
classes = None
with open('coco.names', 'r') as f:
  classes = [line.strip() for line in f.readlines()]

# generate different colors for different classes
COLORS = np.random.uniform(0, 255, size=(len(classes), 3))

# read pre-trained model and config file
net = cv2.dnn.readNet("yolov4.weights", "yolov4.cfg")

# create input blob
blob = cv2.dnn.blobFromImage(image, scale, (416,416), (0,0,0), True,
crop=False)

# set input blob for the network
net.setInput(blob)


# function to get the output layer names
# in the architecture
def get_output_layers(net):

    layer_names = net.getLayerNames()

    output_layers = [layer_names[i - 1] for i in
net.getUnconnectedOutLayers()]
```

```
    return output_layers

# function to draw bounding box on the detected object with class name
def draw_bounding_box(img, class_id, confidence, x, y, x_plus_w, y_plus_h):

    label = str(classes[class_id])
    color = COLORS[class_id]

    cv2.rectangle(img, (x,y), (x_plus_w,y_plus_h), color, 2)
    cv2.putText(img, label, (x-10,y-10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, color,
2)


# run inference through the network
# and gather predictions from output layers
outs = net.forward(get_output_layers(net))

# initialization
class_ids = []
confidences = []
boxes = []
conf_threshold = 0.5
nms_threshold = 0.4

# for each detetion from each output layer
# get the confidence, class id, bounding box params
# and ignore weak detections (confidence < 0.5)
for out in outs:
    for detection in out:
        scores = detection[5:]
        class_id = np.argmax(scores)
        confidence = scores[class_id]
        if confidence > 0.5:
            center_x = int(detection[0] * Width)
            center_y = int(detection[1] * Height)
            w = int(detection[2] * Width)
            h = int(detection[3] * Height)
            x = center_x - w / 2
            y = center_y - h / 2
            class_ids.append(class_id)
            confidences.append(float(confidence))
            boxes.append([x, y, w, h])


# apply non-max suppression
indices = cv2.dnn.NMSBoxes(boxes, confidences, conf_threshold, nms_threshold)

# go through the detections remaining
# after nms and draw bounding box
```

```
for i in indices:
    # i = i[0]
    box = boxes[i]
    x = box[0]
    y = box[1]
    w = box[2]
    h = box[3]
    draw_bounding_box(image, class_ids[i], confidences[i], round(x), round(y),
round(x+w), round(y+h))

# display output image
cv2_imshow(image)

# wait until any key is pressed
cv2.waitKey()

 # save output image to disk
cv2.imwrite("object-detection.jpg", image)

# release resources
cv2.destroyAllWindows()
```

The result is that the objects are detected and labelled. The below figures show the original image and the output image.

Original Image:

Output Image: