RCNN

RCNN introduces a region suggestion mechanism that efficiently identifies candidate regions likely to contain objects and simplifies the detection process.

```
!curl -L "https://public.roboflow.com/ds/AC3gwMp3FC?key=Zwrb2FUj1d" > roboflow.zip; unzip roboflow.zip; rm roboflow.zip
```

With the help of the above command, we load the data from Roboflow.
After creating the required directories for model training, we implement the following function.

```
def get_iou(bb1, bb2):
    assert bb1['x1'] < bb1['x2']
    assert bb1['y1'] < bb1['y2']
    assert bb2['x1'] < bb2['x2']
    assert bb2['y1'] < bb2['y2']

    x_left = max(bb1['x1'], bb2['x1'])
    y_top = max(bb1['y1'], bb2['y1'])
    x_right = min(bb1['x2'], bb2['x2'])
    y_bottom = min(bb1['y2'], bb2['y2'])

    if x_right < x_left or y_bottom < y_top:
        return 0.0

    intersection_area = (x_right - x_left) * (y_bottom - y_top)

    bb1_area = (bb1['x2'] - bb1['x1']) * (bb1['y2'] - bb1['y1'])
    bb2_area = (bb2['x2'] - bb2['x1']) * (bb2['y2'] - bb2['y1'])

    iou = intersection_area / float(bb1_area + bb2_area - intersection_area)
    assert iou >= 0.0
    assert iou <= 1.0
    return iou
```

This code defines the function "get_iou" which calculates the IoU between two bounded boxes represented by dictionaries ("bb1" and "bb2"). The boxes are assumed to be of the form {'x1': xmin , 'y1': ymin, 'x2': xmax, 'y2': ymax}. The function first makes sure that the coordinates of each boundary box are valid. It then calculates the area of intersection between the two boundary boxes and calculates the IoU using The formula calculates:
IoU = intersection_area / (area_bb1 + area_bb2 - intersection_area(

```
ss = cv2.ximgproc.segmentation.createSelectiveSearchSegmentation()
```

This line of code initializes the selected search segmentation object using the OpenCV library. Selective search is a region proposal algorithm that is commonly used in the object detection pipeline to generate potential regions in an image, and it allows to perform a

selective search on an image and get the region proposal for further processing.

```
# Initialize TensorFlow with GPU support
physical_devices = tf.config.list_physical_devices('GPU')
tf.config.experimental.set_memory_growth(physical_devices[0], True)
```

The provided code initializes TensorFlow with GPU support. It first lists all available physical devices using tf.config.list_physical_devices('GPU') , and then sets the memory growth for the first GPU device to True using tf.config.experimental.set_memory_growth adjusts This is a useful configuration when working with GPUs in TensorFlow.
Setting memory growth to "True" allows TensorFlow to allocate GPU memory dynamically as needed, rather than reserving the entire GPU memory in advance. This can be useful for scenarios where you want to minimize GPU memory consumption and allow multiple TensorFlow processes to share the GPU efficiently. The code assumes that at least one GPU device is available.

```
z = 0

for e, i in enumerate(os.listdir(annot)):
    try:
        if z <= 500:
            if i.startswith("i"):
                filename = os.path.splitext(i)[0]
                filename = filename + ".jpg"
                img = cv2.imread(os.path.join(path, filename))

                df = pd.read_csv(os.path.join(annot, i))
                gtvalues = []

                for row in df.iterrows():
                    x1 = int(row[1]["xmin"])
                    y1 = int(row[1]["ymin"])
                    x2 = int(row[1]["xmax"])
                    y2 = int(row[1]["ymax"])
                    gtvalues.append({"x1": x1, "x2": x2, "y1": y1, "y2": y2})

                ss.setBaseImage(img)
                ss.switchToSelectiveSearchFast()
                ssresults = ss.process()
                imout = img.copy()
                counter = 0
                falsecounter = 0
                flag = 0
                fflag = 0
                bflag = 0

                for e, result in enumerate(ssresults):
                    if e < 2000 and flag == 0:
                        for gtval in gtvalues:
                            x, y, w, h = result
                            iou = get_iou(gtval, {"x1": x, "x2": x + w, "y1": y, "y2": y + h})

                            if counter < 30:
                                if iou > 0.70:
                                    timage = imout[y:y + h, x:x + w]
                                    resized = cv2.resize(timage, (224, 224), interpolation=cv2.INTER_AREA)
                                    train_images.append(resized)
                                    train_labels.append(1)
                                    counter += 1
                            else:
                                fflag = 1

                            if falsecounter < 30:
                                if iou < 0.3:
                                    timage = imout[y:y + h, x:x + w]
                                    resized = cv2.resize(timage, (224, 224), interpolation=cv2.INTER_AREA)
                                    train_images.append(resized)
                                    train_labels.append(0)
                                    falsecounter += 1
                            else:
                                bflag = 1

                        if fflag == 1 and bflag == 1:
                            print("inside")
                            flag = 1

                z = z + 1

            else:
                break
    except Exception as e:
        print(e)
        print("error in " + filename
        continue
```

This Python code is run through a directory containing annotations files for object identification, paired with corresponding image files. Its goal is to create a dataset for a binary classification task that distinguishes between positive samples (containing objects of interest) and negative samples (lacking these objects). For each image, it reads related annotations, performs selective search, region proposal selection, and calculates IoU with real boundary boxes. Then it selects the regions with high IoU as positive samples and the regions with low IoU as negative samples and limits the number of both to 30. The resulting images and labels are added to separate lists for training. This process is repeated for a certain number of images (500).

Then we have to give the proposed regions to our model to extract the features.

```
vggmodel = VGG16(weights='imagenet', include_top=True)
vggmodel.summary()
```

Here we load the VGG16 model with pre-trained ImageNet weights using Keras. The "weights" parameter is set to "imagenet", which indicates that the model should be initialized with weights pre-trained on the ImageNet dataset. The "include_top" parameter is set to "True", which means that the fully connected layer is placed on top of the network responsible for classification.
The VGG16 model has a detailed architecture consisting of multiple convolution and pooling layers followed by fully connected layers.

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelBinarizer

class MyLabelBinarizer(LabelBinarizer):
    def transform(self, y):
        Y = super().transform(y)
        if self.y_type_ == 'binary':
            return np.hstack((Y, 1-Y))
        else:
            return Y
    def inverse_transform(self, Y, threshold=None):
        if self.y_type_ == 'binary':
            return super().inverse_transform(Y[:, 0], threshold)
        else:
            return super().inverse_transform(Y, threshold)
```

Next, we define the "MyLabelBinarizer" class. The purpose of this class is to be able to work with binary and multi-class classification tasks and ensure that the transformed matrix contains the original binary labels and their complements.

```
for layers in (vggmodel.layers)[:15]:
    #print(layers)
    layers.trainable = False

X= vggmodel.layers[-2].output

predictions = Dense(2, activation="softmax")(X)

model_final = Model(inputs = vggmodel.input, outputs = predictions)


opt = Adam(lr=0.0001)

model_final.compile(loss = 'categorical_crossentropy', optimizer = opt, metrics=["accuracy"])

model_final.summary()
```

In the next step, we prepare the pre-trained VGG16 model for transfer learning. We freeze the weights of the first 15 layers of the VGG16 model so that their weights do not change during training and add a new dense layer with two output nodes. Then the obtained model is assembled using Adam optimizer with a learning rate of 0.0001 and categorical-crossentropy as a loss function.

It is common in transfer learning when we take a pre-trained model and tune it for a specific task by adding and training a few additional layers. Freezing the initial layers is often done to preserve the knowledge learned by the model on different datasets (pre-training) while only updating the weights of added layers during training on a new dataset (fine-tuning.(

```
lenc = MyLabelBinarizer()
Y =  lenc.fit_transform(np.array(train_labels))

del train_labels
```

Briefly, this code prepares labels for training using a custom label binarizer.

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Assuming X_train, y_train, X_test, and y_test are your original data arrays

# Training Data Generator without Augmentation
traindata_generator = ImageDataGenerator()
traindata = traindata_generator.flow(x=X_train[0:1000], y=y_train[0:1000], shuffle=False)

# Testing Data Generator without Augmentation
testdata_generator = ImageDataGenerator()
testdata = testdata_generator.flow(x=X_test[0:100], y=y_test[0:100], shuffle=False)
```

The above code snippet In summary, the above code prepares the data generators for training and testing without applying any data augmentation. Data generators are commonly used in machine learning workflows to efficiently manage large datasets and apply real-time data augmentation during model training.

```
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping
checkpoint = ModelCheckpoint("R-CNN.h5", monitor='val_loss', verbose=1, save_best_only=True,
save_weights_only=False, mode='auto')
early = EarlyStopping(monitor='val_loss', min_delta=0, patience=100, verbose=1, mode='auto')
```

This code sets the ModelCheckpoint and EarlyStopping for a Keras model.

- 'ModelCheckpoint' causes the model to be saved during training.

Given the given parameters, it stores only the best model based on the monitored quantity (in this case, val-loss).

- "EarlyStopping" is for when the value of the loss function for the validation data does not change in 100 cycles, the training of the model is stopped.
In short, these are the best models for storage. Whenever there is no improvement in the validation data loss, the model is saved as “R-CNN.h5” and training stops early if the validation loss does not improve for a certain number of epochs.

```
hist = model_final.fit_generator(generator= traindata, steps_per_epoch= 5,
epochs= 100, validation_data= testdata, validation_steps=2, callbacks=
[checkpoint, early])
```

The code is trained using a generator for training data ('traindata') and validation data ('testdata') with "model_final"..

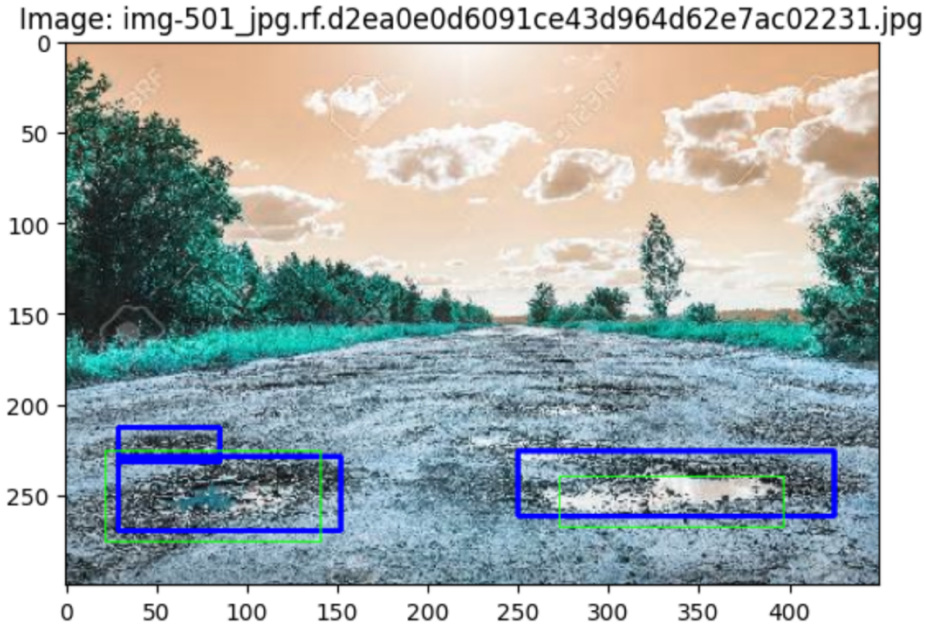
```
annotation_path = "/content/TestInput/Annotations/"

path = "/content/TestInput/Data/Images"
z = 0
for e, i in enumerate(os.listdir(path)):

    if(z<8):  # Add your condition if needed
        z += 1
        img = cv2.imread(os.path.join(path, i))

        # Load ground truth labels
        annotation_file = os.path.join(annotation_path, i.replace('.jpg', '.csv'))
        if os.path.exists(annotation_file):
            df = pd.read_csv(annotation_file)
            for index, row in df.iterrows():
                xmin_gt, ymin_gt, xmax_gt, ymax_gt = int(row['xmin']), int(row['ymin']), int(row['xmax']), int(row['ymax'])
                cv2.rectangle(img, (xmin_gt, ymin_gt), (xmax_gt, ymax_gt), (0, 0, 255), 2)  # Draw ground truth in red

        ss.setBaseImage(img)
        ss.switchToSelectiveSearchFast()
        ssresults = ss.process()
        imout = img.copy()

        detected_boxes = []  # Store detected bounding boxes

        for e, result in enumerate(ssresults):
            if e < 2000:
                x, y, w, h = result
                timage = imout[y:y+h, x:x+w]
                resized = cv2.resize(timage, (224, 224), interpolation=cv2.INTER_AREA)
                img_for_prediction = np.expand_dims(resized, axis=0)

                # Suppress output during prediction
                with open(os.devnull, 'w') as fnull:
                    with contextlib.redirect_stdout(fnull):
                        out = model_final.predict(img_for_prediction)

                if out[0][0] > 0.80:
                    detected_boxes.append((x, y, x+w, y+h))  # Store detected bounding box

        # Merge overlapping bounding boxes
        merged_boxes = cv2.groupRectangles(detected_boxes, 1, 0.2)[0]

        # Draw merged bounding boxes in green for those with IoU > 0.4
        for (x, y, x_plus_w, y_plus_h) in merged_boxes:
            for index, row in df.iterrows():
                xmin_gt, ymin_gt, xmax_gt, ymax_gt = int(row['xmin']), int(row['ymin']), int(row['xmax']), int(row['ymax'])
                intersection_area = max(0, min(x_plus_w, xmax_gt) - max(x, xmin_gt)) * max(0, min(y_plus_h, ymax_gt) - max(y,
ymin_gt))
                union_area = (x_plus_w - x) * (y_plus_h - y) + (xmax_gt - xmin_gt) * (ymax_gt - ymin_gt) - intersection_area
                iou = intersection_area / union_area

                # if iou > 0.3:
                cv2.rectangle(imout, (x, y), (x_plus_w, y_plus_h), (0, 255, 0), 1)
                print(f"IoU for Image {i}: {iou}")

        # Display the image with both ground truth and predicted bounding boxes
        plt.figure()
        plt.imshow(imout)
        plt.title(f"Image: {i}")
        plt.show()

        # break  # Remove this line if you want to process all images
```

This code performs object detection and boundary box drawing by using selective search and a pre-trained model.
This code provides a workflow for detecting objects in images using a pre-trained model. First, the real annotations for each image are read from a CSV file and the real rectangles associated with the objects are displayed in blue in the image. Then, the selective search (SS) algorithm is initialized and run on the image to obtain proposal regions for object detection. After that, proposal regions are used to recognize objects. The size of the extracted regions is changed and as mentioned, a pre-trained model (model_final) is used to predict the presence of objects in each region. Finally, if the prediction probability is greater than a threshold (0.80), the coordinates of the boundary boxes are stored.
Then, the boundary boxes are merged using a function of OpenCV and drawn in green color on the original image. For each merged boundary box, the size of the IoU is calculated with the true boundary boxes and the results are displayed along with the original images with the true boundary boxes and predicted boundary boxes.

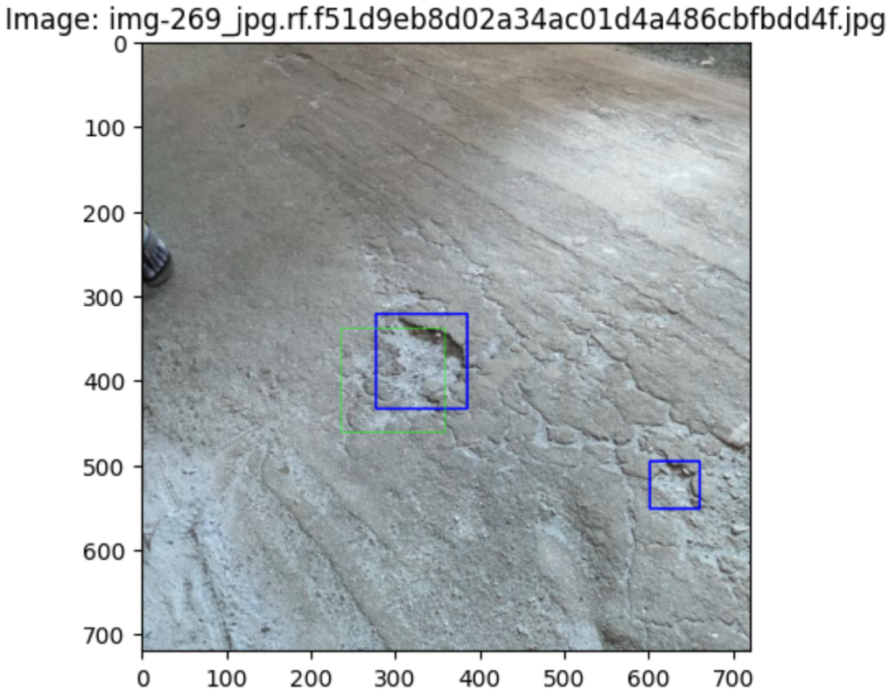
Example of model output for test data:

The green box is the predicted box and the blue box is the labeled data box.

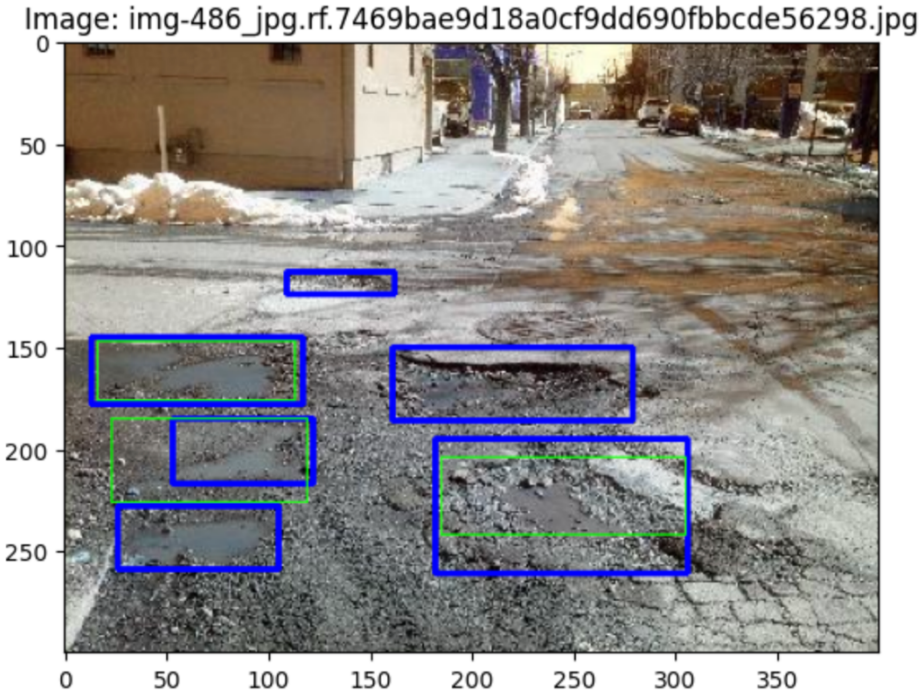
```
IoU for Image img-501_jpg.rf.d2ea0e0d6091ce43d964d62e7ac02231.jpg: 0.38727995457126635
IoU for Image img-501_jpg.rf.d2ea0e0d6091ce43d964d62e7ac02231.jpg: 0.7173879081393533
```



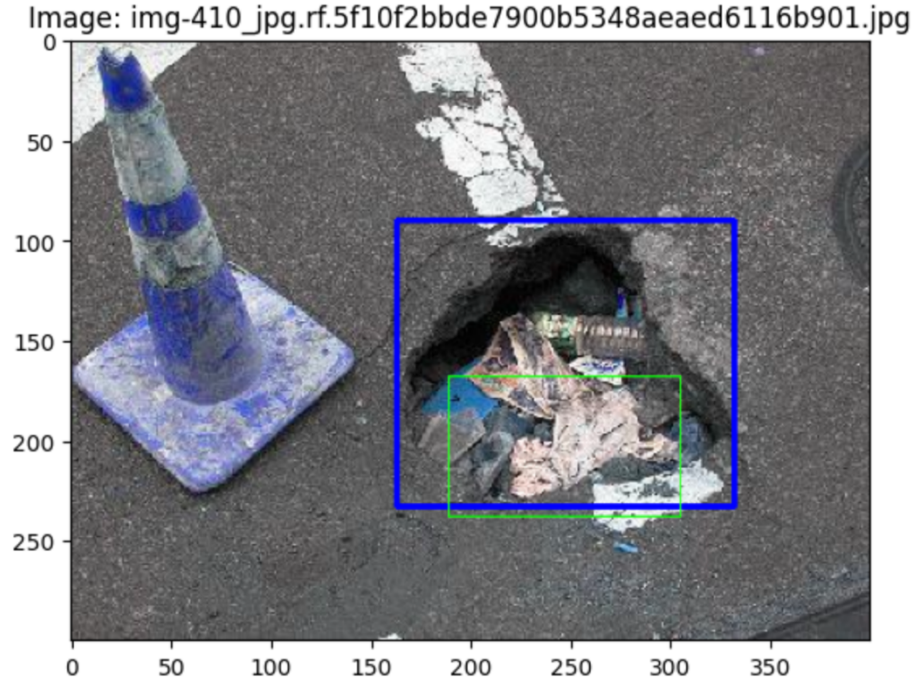

```
IoU for Image img-269_jpg.rf.f51d9eb8d02a34ac01d4a486cbfbdd4f.jpg: 0.400823257010548
```



```
IoU for Image img-486_jpg.rf.7469bae9d18a0cf9dd690fbbcde56298.jpg: 0.8196386946386947
IoU for Image img-486_jpg.rf.7469bae9d18a0cf9dd690fbbcde56298.jpg: 0.5571847507331378
IoU for Image img-486_jpg.rf.7469bae9d18a0cf9dd690fbbcde56298.jpg: 0.5238095238095238
```




```
IoU for Image img-410_jpg.rf.5f10f2bbde7900b5348aeaed6116b901.jpg: 0.30468339596718796
```



Fast-RCNN

Detectron is an open source deep learning framework developed by Facebook Inc. and used to create, train, and evaluate deep learning models in the field of computer vision.
The Fast R-CNN model is one of the models provided by Detectoron to perform object recognition tasks. This model is designed using a deep convolutional neural network (CNN) to extract features from images and a series of object recognition layers (ROI Heads) to recognize the components and objects that exist in the images.
Using a pre-trained model means using a model that has already been trained with big data and can recognize common patterns and features well in the real world. This model can work effectively in recognizing objects and components in new images according to the specific task it is designed to teach. In the following, we will explain and how to use it.

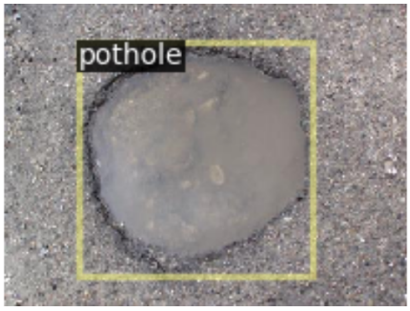
We upload the data as follows.

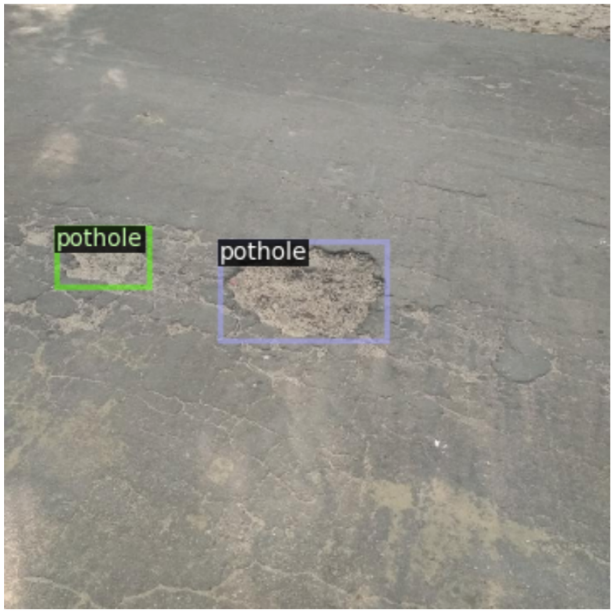
```
!curl -L "https://public.roboflow.com/ds/tARhGryd3k?key=yhHJWCEP5K" > roboflow.zip; unzip roboflow.zip; rm roboflow.zip
```

```
from detectron2.data.datasets import register_coco_instances
register_coco_instances("my_dataset_train", {}, "/content/train/_annotations.coco.json", "/content/train")
register_coco_instances("my_dataset_val", {}, "/content/valid/_annotations.coco.json", "/content/valid")
register_coco_instances("my_dataset_test", {}, "/content/test/_annotations.coco.json", "/content/test")
```

Specifically, this code snippet registers three datasets for training, validation, and testing using the COCO data format. The 'register_coco_instances' function is used to register these datasets, and each register call corresponds to a different dataset partition: training, validation and testing.
Essentially, this code prepares datasets for training, validation, and testing within the Detectron2 framework, and facilitates loading and working with these datasets during the model development process.

An example of training data:






```python
#We are importing our own Trainer Module here to use the COCO validation
evaluation during training. Otherwise no validation eval occurs.
from detectron2.engine import DefaultTrainer
from detectron2.evaluation import COCOEvaluator

class CocoTrainer(DefaultTrainer):

  @classmethod
  def build_evaluator(cls, cfg, dataset_name, output_folder=None):

    if output_folder is None:
        os.makedirs("coco_eval", exist_ok=True)
        output_folder = "coco_eval"

    return COCOEvaluator(dataset_name, cfg, False, output_folder)
```

This piece of code extends the functionality of the Detectron2 library to train object detection models. The purpose of this class is to enable COCO validation evaluation during the training process, which is not done by default.

Using this class, COCO-style validation evaluation can be incorporated into your object detection training pipeline with Detectron2.

```
cfg = get_cfg()
cfg.OUTPUT_DIR = "drive/MyDrive/FAST-RCNN"
cfg.merge_from_file(model_zoo.get_config_file("COCO-Detection/fast_rcnn_R_50_FPN_1x.yaml"))
cfg.DATASETS.TRAIN = ("my_dataset_train",)
cfg.DATASETS.TEST = ("my_dataset_val",)

cfg.DATALOADER.NUM_WORKERS = 4
cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url("COCO-Detection/fast_rcnn_R_50_FPN_1x.yaml")# Let
training initialize from model zoo
cfg.SOLVER.IMS_PER_BATCH = 4
cfg.SOLVER.BASE_LR = 0.01


cfg.SOLVER.WARMUP_ITERS = 1000
cfg.SOLVER.MAX_ITER = 500 #adjust up if val mAP is still rising, adjust down if overfit
cfg.SOLVER.STEPS = (1000, 1500)
cfg.SOLVER.GAMMA = 0.05

cfg.MODEL.ROI_HEADS.BATCH_SIZE_PER_IMAGE = 64
cfg.MODEL.ROI_HEADS.NUM_CLASSES = 2

cfg.TEST.EVAL_PERIOD = 50
```

The code above configures the parameters and settings for training a Fast R-CNN model using the Detectron2 library on a dataset.
In summary, it provides a comprehensive setup for training a Fast R-CNN model on a custom dataset. It covers aspects such as dataset selection, model configuration, training meta-parameters and test/evaluation settings.

```
trainer = CocoTrainer(cfg)
trainer.resume_or_load(resume=False)
trainer.train()
```

This code snippet initializes a 'CocoTrainer' object, which is a class derived from 'DefaultTrainer' in the Detectron2 library. And as mentioned, this class is designed for the COCO dataset and uses COCO's validation evaluation during training.
This code is basically the starting point for training the Fast R-CNN model on the specified training dataset with the configured settings. Training progress and results are recorded and stored in the specified output directory. Configuration settings, such as maximum number of

iterations or learning speed, can be tailored to specific training needs.

```
#test evaluation
from detectron2.data import DatasetCatalog, MetadataCatalog, build_detection_test_loader
from detectron2.evaluation import COCOEvaluator, inference_on_dataset

cfg.MODEL.WEIGHTS = os.path.join(cfg.OUTPUT_DIR, "model_final.pth")
cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST = 0.85
predictor = DefaultPredictor(cfg)
evaluator = COCOEvaluator("my_dataset_test", cfg, False, output_dir="./content/gdrive/My
Drive/V1/")#"./output/")
val_loader = build_detection_test_loader(cfg, "my_dataset_test")
inference_on_dataset(trainer.model, val_loader, evaluator)
```

After completing the training, this part of the code is dedicated to evaluating the trained model on a test data set.
Then the results are evaluated using the COCO evaluator. Evaluation criteria, such as mAP, will be printed.
In fact, this code allows you to evaluate the performance of the trained Fast R-CNN model on a separate test dataset.
The result of the test data is as follows.

```
|   AP   |  AP50  |  AP75  |  APs   |  APm   |  APl   |
|:------:|:------:|:------:|:------:|:------:|:------:|
| 32.796 | 59.186 | 30.356 | 28.654 | 23.497 | 44.114 |
[11/24 20:10:10 d2.evaluation.coco_evaluation]: Per-category bbox AP:
| category   | AP    | category   | AP     |
|:-----------|:------|:-----------|:-------|
| potholes   | nan   | pothole    | 32.796 |
OrderedDict([('bbox',
              {'AP': 32.79563798808785,
               'AP50': 59.18580582378611,
               'AP75': 30.355892803287688,
```

```
cfg.MODEL.WEIGHTS = os.path.join(cfg.OUTPUT_DIR, "model_final.pth")
cfg.DATASETS.TEST = ("my_dataset_test", )
cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST = 0.7   # set the testing threshold for this model
predictor = DefaultPredictor(cfg)
test_metadata = MetadataCatalog.get("my_dataset_test")
```

With the above settings, you are ready to use the "predictor" to perform inference on a test data set using the specified model and evaluate its performance.

Example of test data:

Yolo

YOLO is a popular object detection algorithm that revolutionized real-time object detection in images and videos. YOLO's key innovation is its ability to perform object detection in one pass of the neural network, making it very fast and efficient.

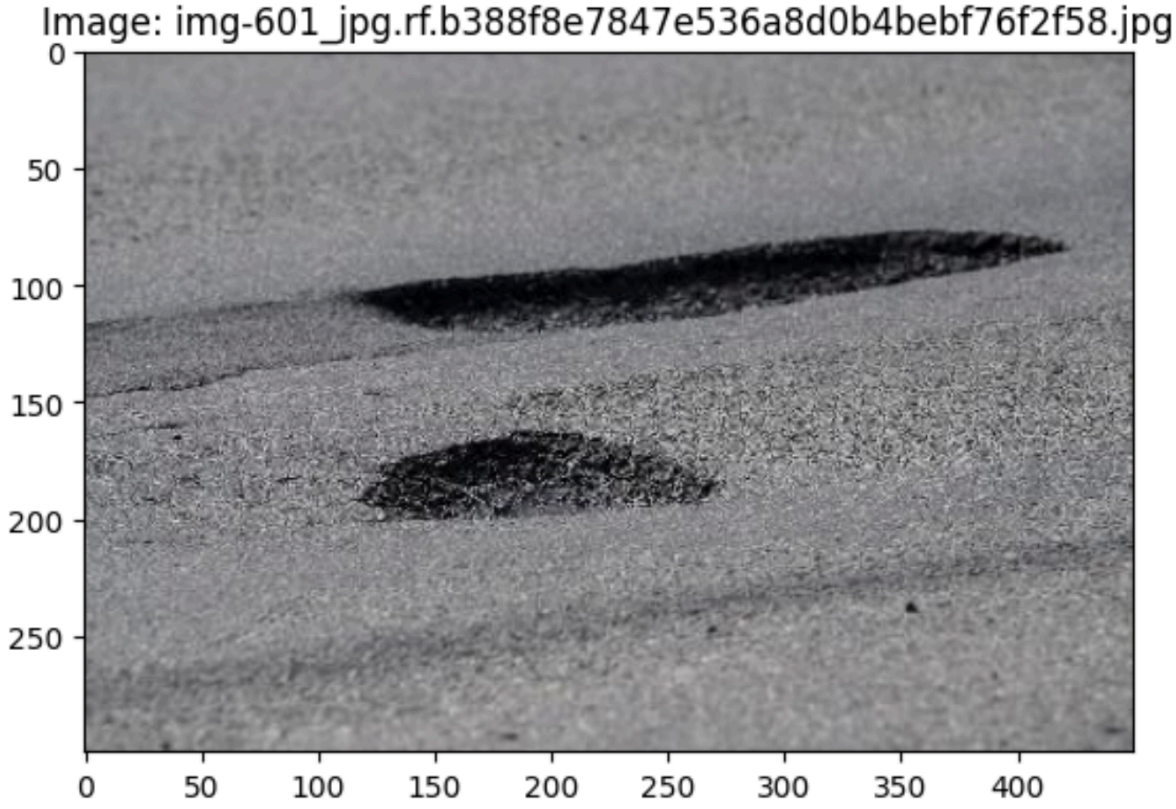
The basic idea of YOLO is to divide the input image into a grid and then predict the boundary boxes and class probabilities directly from the grid cells. Each grid cell is responsible for predicting multiple boundary boxes, and this boundary box contains information about the location, size, and reliability of detected objects. YOLO also predicts class probabilities for each boundary box, indicating the probability that the object belongs to a particular class.

The YOLO algorithm has several versions, and YOLOv3 is one of the most widely used. YOLOv3 improved on its predecessors by introducing a more complex architecture, incorporating jumper connections and providing better detection performance. YOLOv4 and later versions have continued to improve speed, accuracy, and display features. Version 8 is used in this code.

First, we receive the data with the following command.

```
!curl -L "https://public.roboflow.com/ds/N3Gi1NiYFv?key=DGssI4Gmhc" > roboflow.zip; unzip roboflow.zip; rm roboflow.zip
```

The data is as follows.



```python
def create_image_and_label_folders(source_folder, dest_folder):
    # Create 'image' and 'label' folders in the destination directory
    image_folder = os.path.join(dest_folder, 'images')
    label_folder = os.path.join(dest_folder, 'labels')

    os.makedirs(image_folder, exist_ok=True)
    os.makedirs(label_folder, exist_ok=True)

    # Iterate over files in the source folder
    for filename in os.listdir(source_folder):

        name, extension = os.path.splitext(filename)

        if (extension==".jpg"):
            image_path = os.path.join(source_folder, filename)
            img = mpimg.imread(image_path)

            # Check if the shape is (720, 720)
            if img.shape == (720, 720, 3):
            # Copy jpg files to the 'image' folder
              # print("*")
              shutil.copy2(os.path.join(source_folder, filename), os.path.join(image_folder, filename))

                  # Create a corresponding txt file in the 'label' folder
              filename = name+'.txt'
              shutil.copy2(os.path.join(source_folder, filename), os.path.join(label_folder, filename))


# Replace these paths with your actual paths
train_source_folder = '/content/train'
train_dest_folder = '/content/datasets/pothole_dataset_v8/train'

valid_source_folder = '/content/valid'
valid_dest_folder = '/content/datasets/pothole_dataset_v8/valid'

test_source_folder = '/content/test'
test_dest_folder = '/content/datasets/pothole_dataset_v8/test'

# Create 'image' and 'label' folders for train, valid, and test sets
create_image_and_label_folders(train_source_folder, train_dest_folder)
create_image_and_label_folders(valid_source_folder, valid_dest_folder)
create_image_and_label_folders(test_source_folder, test_dest_folder)
```

This Python code defines a function, create_image_and_label_folders, that organizes image and label files into separate folders for a specified source and destination directory. This function creates the "images" and "labels" subfolders in the destination directory and then iterates through the files in the source directory. For each JPEG image file (identified with a '.jpg' extension), checks whether the image format is (720, 720, 3). If so, it copies the image into the "images" folder and creates a corresponding text file with the ".txt" extension in the "labels" folder. The code then specifies the source and destination paths for the training, validation, and test sets and calls the function to organize these sets into the corresponding "images" and "labels" folders in the specified dataset directory.

```yaml
%%writefile /content/datasets/pothole_dataset_v8/pothole.yaml
path: pothole_dataset_v8/
train: 'train/images'
val: 'valid/images'

# class names
names:
  0: 'pothole'
```

The provided code writes a YAML configuration file named "pothole.yaml" to the "/content/datasets/pothole_dataset_v8/" directory. Also, the above code specifies the configuration of training and validation set paths along with class names. The training images are expected to be in the "train/images" directory and the validation images are in the "valid/images" directory, both corresponding to the dataset folder. The dataset is defined to have a single class labeled "pothole" with a class index of 0. This YAML file is commonly used to configure dataset information when working with models and machine learning frameworks that use YAML-based configuration, such as YOLO.

```
from ultralytics import YOLO

model = YOLO("yolov8n.pt")  # load a pretrained model (recommended for training)
```

The provided code uses the Ultralytics YOLO library to instantiate a YOLO model called "model". Loads a pre-trained YOLOv8 model from the file "yolov8n.pt". The YOLO algorithm divides an image into a grid and predicts bounding boxes and class probabilities for objects in each grid cell. Ultralytics is a library that provides an easy user interface for YOLO and makes training and inference with YOLO models easy. In this case, the code loads a pre-trained YOLOv8 model, which is a specific version of YOLO.

```
results = model.train(data='/content/datasets/pothole_dataset_v8/pothole.yaml', imgsz=720,
epochs=300, batch=16, name='yolov8n_custom')
```

The provided code uses the Ultralytics YOLO library to train the YOLOv8 object recognition model.
- `data` specifies the path to a YAML file that contains information about the dataset, including training and validation set paths, and class names. In this case, it refers to the "pothole.yaml" file located in the "/content/datasets/pothole_dataset_v8/" directory.

- `imgsz' sets the size of the input image for training. In this case, it is set to 720 pixels.

The code basically instructs the YOLO model to train on the specified dataset using the given configuration. The training process is run for 300 cycles with a batch size of 16, and the progress and results are saved as 'yolov8n_custom'.

```python
# Replace these paths with the actual paths
folder_path = "/content/datasets/pothole_dataset_v8/test/images"
label_folder_path = "/content/datasets/pothole_dataset_v8/test/labels"

# Initialize variables for evaluation metrics
total_true_positives = 0
total_false_positives = 0
total_false_negatives = 0
total_iou_scores = []

# Specify the number of files to process
num_files_to_process = 10

# Set the IoU threshold for combining bounding boxes
iou_threshold_combine = 0.5

# Iterate over the files in the folder
for filename in os.listdir(folder_path)[:num_files_to_process]:
    # Load the image
    image_path = os.path.join(folder_path, filename)
    img = Image.open(image_path)

    # Get image size
    img_width, img_height = img.size

    # Extract bounding boxes from the results (replace this with your actual prediction logic)
    results = model(image_path)
    boxes = results[0].boxes.xywh.cpu().numpy()

    # Read labels from the corresponding text file
    label_filename = os.path.splitext(filename)[0] + ".txt"
    label_path = os.path.join(label_folder_path, label_filename)

    if os.path.exists(label_path):
        with open(label_path, 'r') as label_file:
            labels = label_file.readlines()

        # Initialize variables for evaluation metrics for the current image
        true_positives = 0
        false_positives = 0
        false_negatives = 0
        iou_scores = []

        # Visualize the bounding boxes and calculate metrics
        draw = ImageDraw.Draw(img)
```

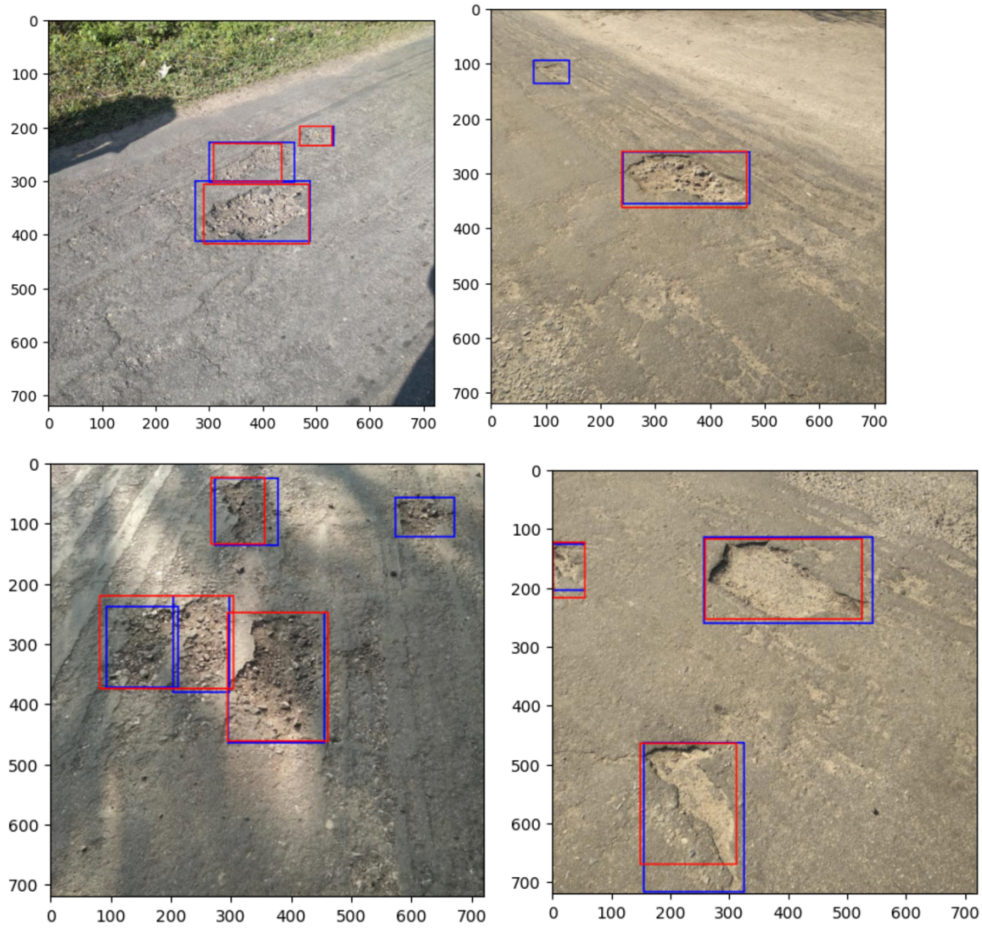
```
        # Draw ground truth bounding boxes
        for label in labels:
            label = label.strip().split()
            class_id, x_norm, y_norm, width_norm, height_norm = map(float, label)

            # Convert normalized coordinates to pixel values
            x = int(x_norm * img_width)
            y = int(y_norm * img_height)
            width_gt = int(width_norm * img_width)
            height_gt = int(height_norm * img_height)

            x3 = int(x - width_gt / 2)
            y3 = int(y - height_gt / 2)
            x4 = int(x + width_gt / 2)
            y4 = int(y + height_gt / 2)

            draw.rectangle([x3, y3, x4, y4], outline="blue", width=3)

        # Combine and draw predicted bounding boxes
        combined_boxes = []

        for box in boxes:
            x_center, y_center, width, height = box
            x1 = int(x_center - width / 2)
            y1 = int(y_center - height / 2)
            x2 = int(x_center + width / 2)
            y2 = int(y_center + height / 2)

            # Check for IoU with existing combined boxes
            iou_threshold = iou_threshold_combine
            box_area = width * height

            combined = False

            for combined_box in combined_boxes:
                x3, y3, x4, y4 = combined_box

                # Calculate IoU
                intersection_area = max(0, min(x2, x4) - max(x1, x3)) * max(0, min(y2, y4) - max(y1, y3))
                union_area = box_area + (width_gt * height_gt) - intersection_area
                iou = intersection_area / union_area

                # If IoU is above the threshold, combine the boxes
                if iou >= iou_threshold:
                    combined_boxes.remove(combined_box)
                    x1 = min(x1, x3)
                    y1 = min(y1, y3)
                    x2 = max(x2, x4)
                    y2 = max(y2, y4)
                    combined_boxes.append([x1, y1, x2, y2])
                    combined = True
                    break

            if not combined:
                combined_boxes.append([x1, y1, x2, y2])

            # draw.rectangle([x1, y1, x2, y2], outline="red", width=3)
```

```
        # Check for IoU with ground truth boxes
        iou_threshold = 0.5
        box_area = width * height

        for label in labels:
            label = label.strip().split()
            class_id, x_norm, y_norm, width_norm, height_norm = map(float, label)

            # Convert normalized coordinates to pixel values
            x = int(x_norm * img_width)
            y = int(y_norm * img_height)
            width_gt = int(width_norm * img_width)
            height_gt = int(height_norm * img_height)

            x3 = int(x - width_gt / 2)
            y3 = int(y - height_gt / 2)
            x4 = int(x + width_gt / 2)
            y4 = int(y + height_gt / 2)

            # Calculate IoU
            intersection_area = max(0, min(x2, x4) - max(x1, x3)) * max(0, min(y2, y4) - max(y1, y3))
            union_area = box_area + (width_gt * height_gt) - intersection_area
            iou = intersection_area / union_area
            iou_scores.append(iou)

            # Update metrics based on IoU threshold
            if iou >= iou_threshold:
                true_positives += 1
            else:
                false_positives += 1

        # Calculate false negatives
        false_negatives = len(labels) - true_positives

        # Update total metrics
        total_true_positives += true_positives
        total_false_positives += false_positives
        total_false_negatives += false_negatives
        total_iou_scores.extend(iou_scores)

        # Draw the combined bounding box around all predicted boxes
        for combined_box in combined_boxes:
            x1, y1, x2, y2 = combined_box
            draw.rectangle([x1, y1, x2, y2], outline="red", width=3)

        # Display the image with matplotlib (comment this out if you don't want to visualize each image)
        plt.imshow(img)
        plt.show()

# Calculate overall precision, recall, and average IoU
overall_precision = total_true_positives / (total_true_positives + total_false_positives)
overall_recall = total_true_positives / (total_true_positives + total_false_negatives)
overall_average_iou = np.mean(total_iou_scores)

# Print the overall evaluation metrics
print(f"Overall Precision: {overall_precision:.2f}")
print(f"Overall Recall: {overall_recall:.2f}")
print(f"Overall Average IoU: {overall_average_iou:.2f}")
```

This code evaluates object detection results on a limited number of images. Assumes that the image files are in the specified folder ('folder_path') and the corresponding label files are in another folder ('label_folder_path'). Evaluation is based on IoU, precision, recall and IoU criteria.

The script processes a number of files ("num_files_to_process") from the specified folder. For each image, it loads the image and extracts the bounding boxes from the detection results. The actual labels of the image are read from the corresponding text file and the script calculates the metrics by comparing the predicted boundary box with the actual boundary box.

The metrics are calculated for each image individually and variables such as False Psitive and ..., the IoU score are updated. Visualization of images is displayed with predicted boxes using Matplotlib.
After processing all images, the script calculates the overall precision, recall, and average IoU based on the accumulated metrics and prints these evaluation metrics. This code provides an insightful summary of the model's performance on a given set of real images and labels.
The results are as follows:



average Precision: 0.39
average Recall: 0.39
average Average IoU: 0.37

- Average Precision is a measure to measure the accuracy of the model in identifying objects. It considers both precision and recall of predictions. An AP of 0.39 indicates that, on average, about 39% of the predicted bounding boxes are correct and associated with the label.
average recall - also known as sensitivity or true positive rate, measures the ability of the model to identify all relevant cases. The average recall of 0.39 indicates that the model captures about 39% of all true samples in the dataset.
The average IoU of 0.37 indicates that, on average, the overlap between the predicted and actual bounding boxes is 37%, indicating spatial localization accuracy.
Conclusion
Due to implementation problems and lack of hardware facilities, we could not train the implemented models in the same condition with the same data set, but after an overall evaluation, YOLO appears as the best choice among the implemented object recognition models, which has superior performance and efficiency metrics. It shows a significant YOLO excels in terms of accuracy and consistency in different detection scenarios. Also, YOLO performs very fast testing while RCNN requires the longest time for testing data.
The second question.
The first part.
ORB (Oriented FAST and Rotated BRIEF) and SIFT (Scale-Invariant Feature Transform) are both popular feature extraction algorithms used in computer vision and image processing. While they have similarities, they also have distinct differences. The similarities and differences of these algorithms are equal to:

similarities:
1. Feature detection: Both ORB and SIFT algorithms are able to detect key points in an image. These key points are places in the image that stand out and can be used for further processing.
2. Scale-Invariance: Both algorithms are designed to be scale-invariant, meaning they can identify and describe features at different scales in the same image. This makes them resistant to changes in scale, such as object size or distance from the camera.
The differences:
1. Feature description:
ORB uses a binary feature descriptor, which is a compressed representation of the local image patch around each keypoint. The binary descriptor makes it computationally efficient and faster to compute and compare.
SIFT uses a 128-dimensional vector descriptor that provides a more accurate and distinct representation of key points. This rich descriptor collects more information but is computationally more complex.
2. rotation-unchanged:
ORB is designed to be rotation stable, allowing it to handle images with different orientations. It can detect key points without explicitly relying on their orientation.
SIFT explicitly calculates the orientation of key points and makes it robust to images with different orientations to achieve rotation invariance.

3. Speed and efficiency:
ORB is known for its computational efficiency and is generally faster than SIFT. This speed makes it more suitable for real-time applications or scenarios where performance is critical.
SIFT can be more computationally intensive, which can affect its processing speed.
4. Match:
SIFT is often considered more robust in terms of matching performance, especially in scenarios with significant changes in viewpoint, illumination, or partial occlusion. Its rich descriptors help better match in challenging situations.
ORB, although efficient, may struggle in such challenging situations due to its binary descriptors.
5. patented algorithm:
SIFT is registered by its original developer, which restricts its use to commercial purposes. Authorization may be required for certain commercial applications.
ORB is an open source algorithm and can be used freely without any license restrictions.
It is important to note that both ORB and SIFT have been around for some time, and newer algorithms such as SURF (robust high-speed features) and AKAZE (accelerated-KAZE) have been developed to address some limitations and improve performance. . The choice between ORB and SIFT depends on the specific needs of the computer vision task and available computing resources.

Implementation.
ORB

FAST and Oriented Rotated BRIEF (ORB) algorithms stand out as a robust and computationally efficient feature descriptor commonly used in computer vision applications. One of its key advantages is its ability to provide distinct feature points in an image, making it suitable for tasks such as object detection, image stitching, and 3D reconstruction.
The binary nature of the ORB feature descriptor contributes to its computational efficiency, making it particularly suitable for real-time applications and resource-constrained environments. The algorithm's ability to handle real-world scenarios with varying lighting conditions and perspective changes increases its versatility. ORB's effectiveness in providing reliable and repeatable features, along with its computational efficiency, make it a popular choice in applications ranging from robotics and augmented reality to image matching and localization tasks.

```python
import cv2
import os
import matplotlib.pyplot as plt

def load_dataset(directory_path):
    data_list = []

    for r, d, f in os.walk(directory_path):
        if len(d) > 0:
            parent = os.path.basename(os.path.dirname(r))
            for file in f:
                if file.endswith(('.png')):
                    path = os.path.join(r, file)
                    img = cv2.imread(path)

                    # Extract the label from the "maps" folder
                    map_folder_path = os.path.join(r, "maps")
                    label_file = f"{file[:-4]}-map.png"
                    map_label_path = os.path.join(map_folder_path, label_file)
                    label = cv2.imread(map_label_path, cv2.IMREAD_GRAYSCALE)

                    data_list.append((img, parent, label.flatten()[0]))

    return data_list

# Example usage
dataset_path = "/content/ETH-80"
data = load_dataset(dataset_path)

# Example: Plot the first image
plt.imshow(data[0][0], cmap='gray')
plt.title(f"Subject: {data[0][1]}, Label: {data[0][2]}")
plt.show()
```

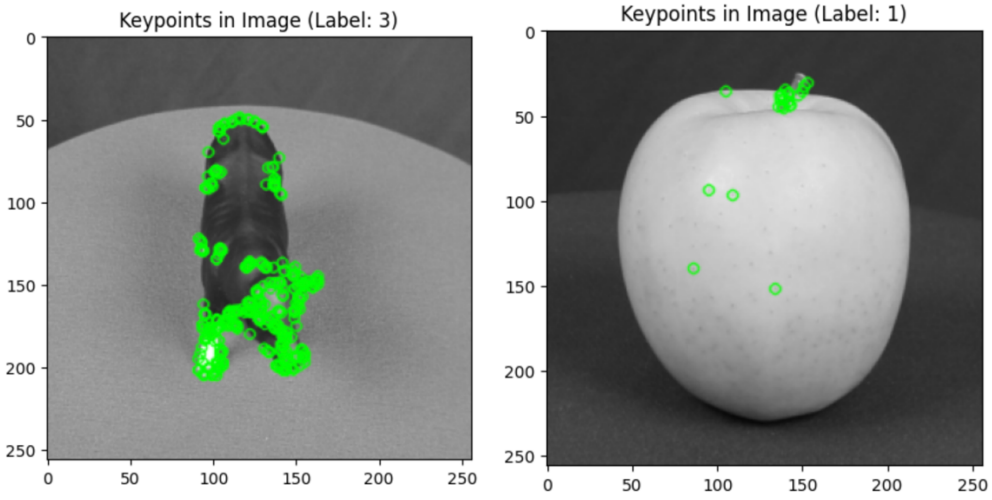
The provided code defines a Python function called "load_dataset" which is responsible for loading the dataset from a specified directory path. It is assumed that the dataset is constructed with sub-categories, each representing a class. The uploaded image and its associated class tag are added to a list called "data-list" in multiples. The function finally returns this list. The usage example shows loading a dataset from the "/content/ETH-80" directory and then plotting the first image along with its subject (class) label using `matplotlib.pyplot`.

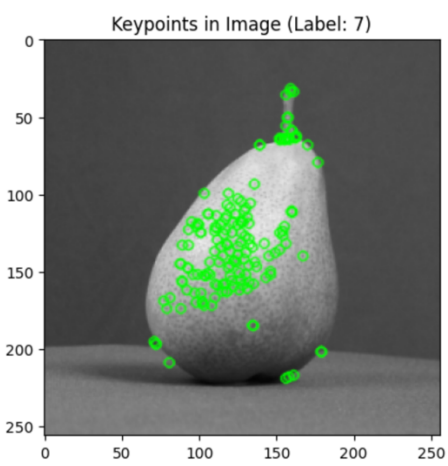
The result of running the code on the data to find key points:

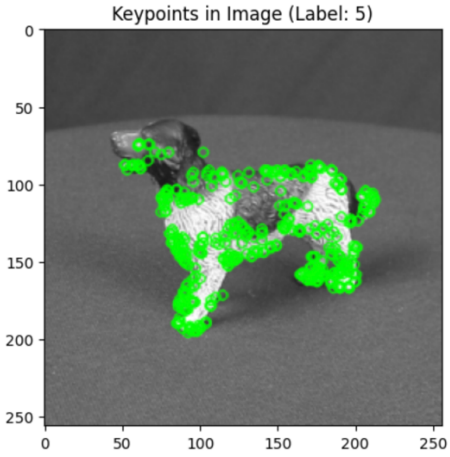
```python
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
import random

# Randomly select 5 images from the dataset
random_images = random.sample(data, 5)

for image, label, x in random_images:
    # Convert the image to grayscale
    gray_image = cv.cvtColor(image, cv.COLOR_BGR2GRAY)

    # Initiate ORB detector
    orb = cv.ORB_create()

    # Find the keypoints with ORB and compute the descriptors
    kp, des = orb.detectAndCompute(gray_image, None)

    # Draw only keypoints location, not size and orientation
    img_with_keypoints = cv.drawKeypoints(gray_image, kp, None, color=(0, 255, 0), flags=0)

    # Display the image with keypoints
    plt.imshow(cv.cvtColor(img_with_keypoints, cv.COLOR_BGR2RGB))
    plt.title(f"Keypoints in Image (Label: {label})")
    plt.show()

    # Print the number of keypoints in the image
    print(f"Number of keypoints in Image (Label: {label}): {len(kp)}")
```

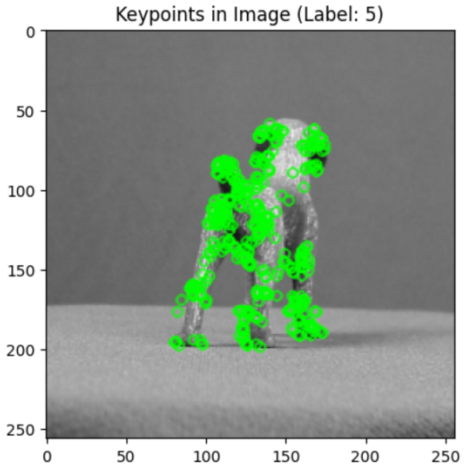
This code randomly selects 5 images from a dataset and performs keypoint detection using the Oriented FAST and Rotated BRIEF (ORB) algorithm. For each selected image, the code converts it to grayscale, applies the ORB detector to find keypoints and calculate descriptors, and then visualizes the keypoints on the original image. The number of keypoints detected in each image is printed, providing insights into the distinct features detected by the ORB algorithm. The resulting images are displayed with highlighted key points using Matplotlib.

Result:




Keypoints in Image (Label: 7)
0
50
100
150
200
250
0
50
100
150
200
250


Keypoints in Image (Label: 5)
0
50
100
150
200
250
0
50
100
150
200
250


Keypoints in Image (Label: 5)
0
50
100
150
200
250
0
50
100
150
200
250

```python
import matplotlib.pyplot as plt
import random

# # Randomly select 5 images from the dataset
random_images = random.sample(data, 5)

same_class_images = []
same_class = 3

# Select 5 images from the same class
for img, subject_info, label in data:
    if subject_info == same_class:
        same_class_images.append(img)
        if len(same_class_images) == 5:
            break
    else:
        same_class = subject_info

# Plot selected images from the same class
plt.figure(figsize=(15, 5))
for i, img in enumerate(same_class_images):
    plt.subplot(1, 5, i + 1)
    plt.imshow(img, cmap='gray')  # Assuming images are grayscale
    plt.title(f"Same Class - Image {i + 1}\nLabel: {subject_info}")
    plt.axis("off")

plt.show()

different_class_images = []
different_classes = set()

# Select 5 images from different classes
for img, subject_info, label in data:
    if subject_info not in different_classes:
        different_class_images.append(img)
        different_classes.add(subject_info)
        if len(different_class_images) == 5:
            break

# Plot selected images from different classes
plt.figure(figsize=(15, 5))
for i, img in enumerate(different_class_images):
    plt.subplot(1, 5, i + 1)
    plt.imshow(img, cmap='gray')  # Assuming images are grayscale
    plt.title(f"Different Class - Image {i + 1}\nLabel: {label}")
    plt.axis("off")

plt.show()
```

The provided code is a Python script for visualizing a subset of images from a dataset. It assumes that the dataset ("data") is a list of multiples, each containing an image, subject information, and a label. The script randomly selects five images from the entire dataset and then selects another five images from the same class as the first image. Using "matplotlib.pyplot" plots these images in two separate shapes.

In the first set of visualizations, it draws five images of a class. For each image, it displays the image itself, a title indicating it belongs to the same class, the image number, and the topic label. Subject info ('subject_info') is assumed to represent the class tag.

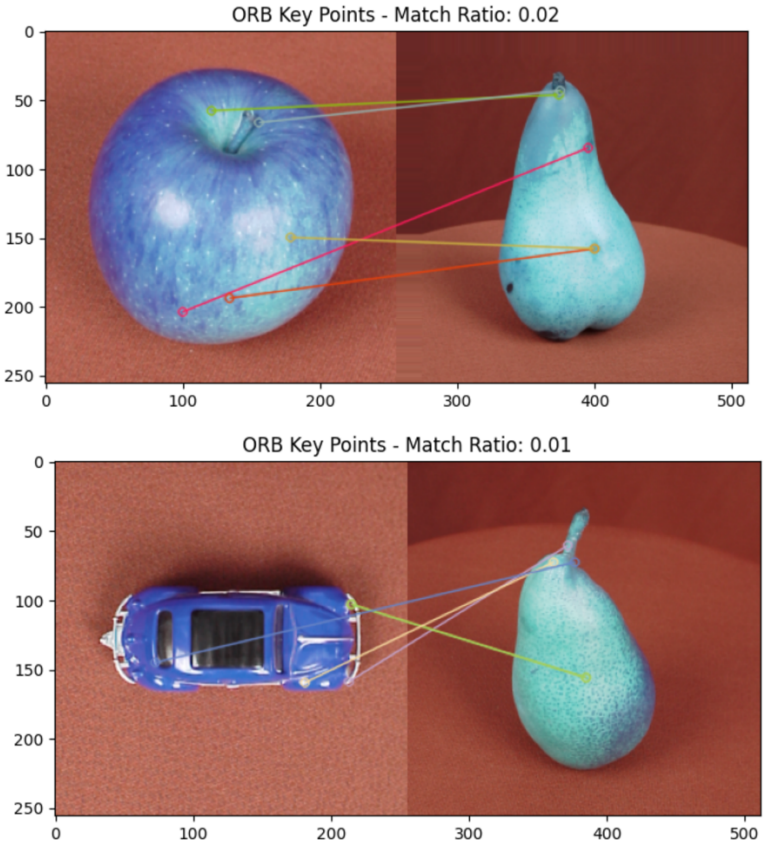
In the second set of visualizations, he selects five images from different classes and draws them similarly to the first set. This script uses an array ("different_classes") to keep track of the classes that have already been included to ensure that images from different classes are selected.

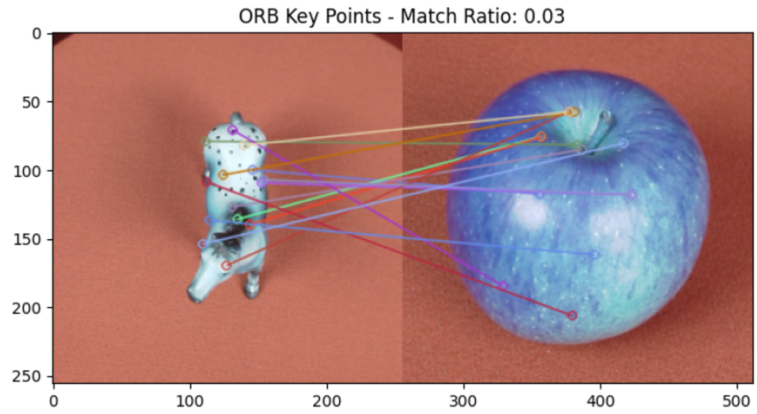

```python
def calculate_accuracy(data, threshold):
    true_positive = 0
    true_negative = 0
    false_positive = 0
    false_negative = 0

    for i in range(len(data)):
        for j in range(i + 1, len(data)):
            image1, label1, _ = data[i]
            image2, label2, _ = data[j]

            feature_extractor = cv.ORB_create()

            kp1, descriptors1 = feature_extractor.detectAndCompute(image1, None)
            kp2, descriptors2 = feature_extractor.detectAndCompute(image2, None)

            bf = cv.BFMatcher()
            matches = bf.knnMatch(descriptors1, descriptors2, k=2)

            good_matches = []
            if len(matches) >= 2:
                for m, n in matches:
                    if m.distance < 0.8 * n.distance:
                        good_matches.append(m)

            match_ratio = len(good_matches) / len(matches) if len(matches) > 0 else 0

            if match_ratio > threshold and label1 == label2:
                true_positive += 1
            elif match_ratio <= threshold and label1 != label2:
                true_negative += 1
            elif match_ratio > threshold and label1 != label2:
                false_positive += 1
            elif match_ratio <= threshold and label1 == label2:
                false_negative += 1

            # Visualize the matches for the last pair of images
            img3 = cv.drawMatchesKnn(image1, kp1, image2, kp2, [good_matches], None, flags=cv.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
            plt.figure(figsize=(8, 8))
            plt.title(f'ORB Key Points - Match Ratio: {match_ratio:.2f}')
            plt.imshow(img3)
            plt.show()

    accuracy = (true_positive + true_negative) / (true_positive + true_negative + false_positive + false_negative)
    return accuracy

# Shuffle the dataset
random.shuffle(data)
# Select the first 5 images from the shuffled dataset
selected_data = data[:5]
# Set the matching ratio threshold
threshold = 0
# Calculate confusion matrix
accuracy = calculate_accuracy(selected_data, threshold)
# Print the results
print("Accuracy:", accuracy)
```

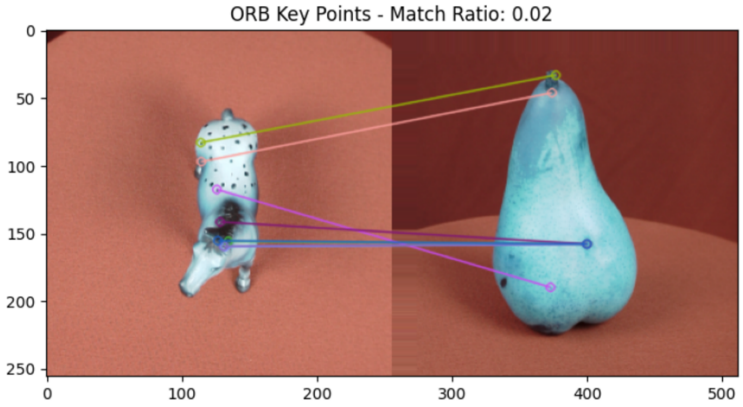
The provided Python script defines a function called "calculate_accuracy" that calculates the accuracy of image matching based on ORB (Oriented Fast and Short Rotation) features. The

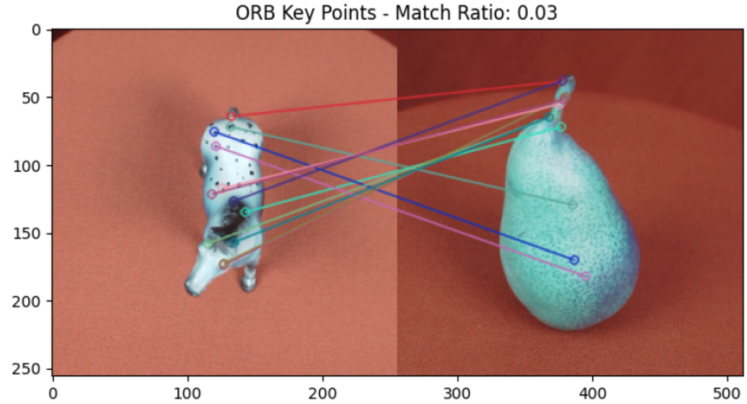
function takes a data set ("data") and an adaptive ratio threshold as input parameters. Iterates over the image pairs in the dataset, extracts ORB features using OpenCV, matches them using a Brute-Force matcher (“cv.BFMatcher”), and calculates the matching ratio.
The matching ratio is then compared to the specified threshold and the pair is classified as true positive, true negative, false positive, or false negative based on the ground truth labels. This script visualizes the matches of the last pair of images using "cv.drawMatchesKnn" and displays the result using "matplotlib.pyplot". Accuracy is calculated and printed based on classification results.

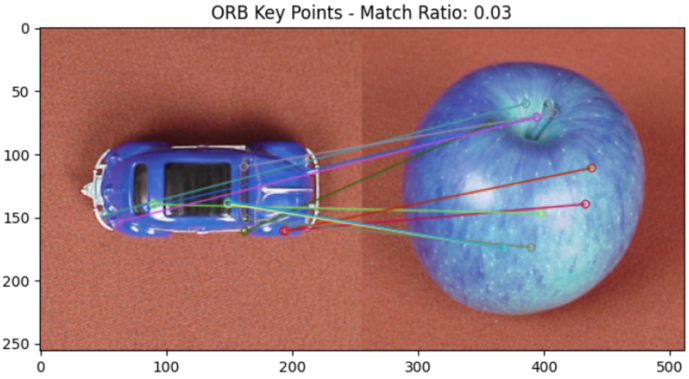
Note: The "calculate_accuracy" function assumes that the dataset ("data") contains multiple images, labels, and a cursor value (the third element in the tuple). ORB feature extraction and matching are performed using OpenCV, and the accuracy is determined by comparing the matching ratio with the threshold.

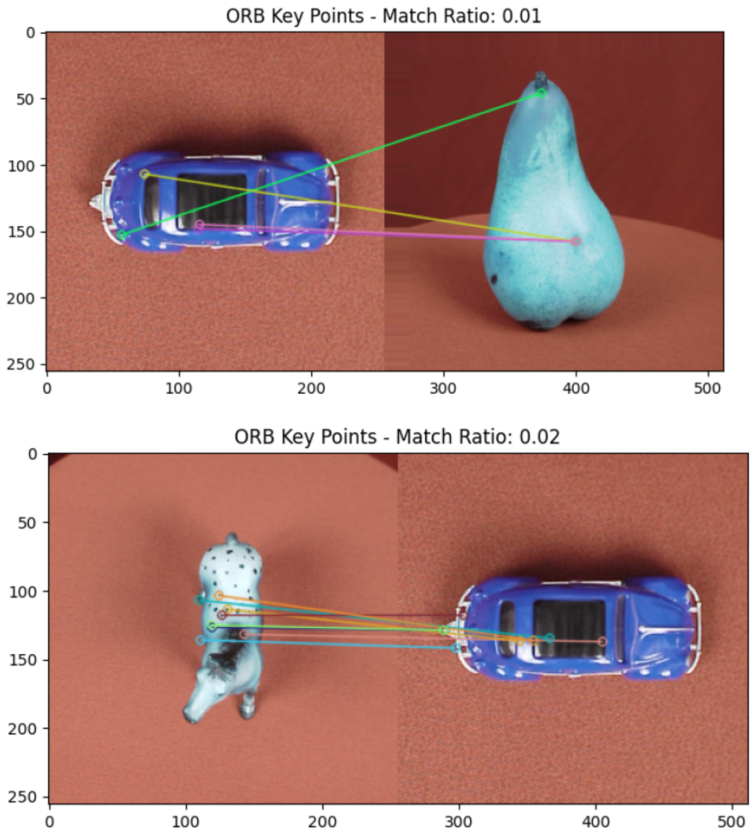



ORB Key Points - Match Ratio: 0.03
0
50
100
150
200
250
0
100
200
300
400
500


ORB Key Points - Match Ratio: 0.02
0
50
100
150
200
250
0
100
200
300
400
500


ORB Key Points - Match Ratio: 0.03
0
50
100
150
200
250
0
100
200
300
400
500


ORB Key Points - Match Ratio: 0.03
0
50
100
150
200
250
0
100
200
300
400
500



Accuracy: 0.1

While the accuracy metric may indicate excellent performance with a value of 1, it is important to recognize that accuracy alone may not be a comprehensive measure of a model's performance, especially in the context of keypoint matching. In the case of ORB, its accuracy may be high, but the visual quality of the results may not be satisfactory. ORB is known for its speed and efficiency, making it suitable for specific applications. However, it may not be the most suitable choice for tasks that require high-quality keypoint matching. The binary nature of ORB descriptors, while contributing to computational efficiency, may lack the descriptive power needed to accurately and robustly match keypoints in scenarios with complex textures or light changes. It is important to consider the specific requirements of the task at hand and potentially explore alternative feature extraction and matching techniques for improved results in visually challenging contexts.

Conclusion

Both SIFT and ORB are common feature extraction and matching algorithms used in computer vision applications. Developed by David Lowe, SIFT is known for its robustness and accuracy in detecting and describing key features in images. The scale invariance and distinctive feature representation of SIFT make it suitable for various image recognition tasks. ORB, on the other hand, is designed to be computationally efficient, making it faster but potentially less robust than SIFT. While ORB can perform well in real-time applications, SIFT often outperforms it in scenarios where accuracy and robustness are critical, such as image stitching or object detection in complex scenes. Therefore, the choice between SIFT and ORB depends on the specific needs of the application, balancing the need for accuracy and computational efficiency.