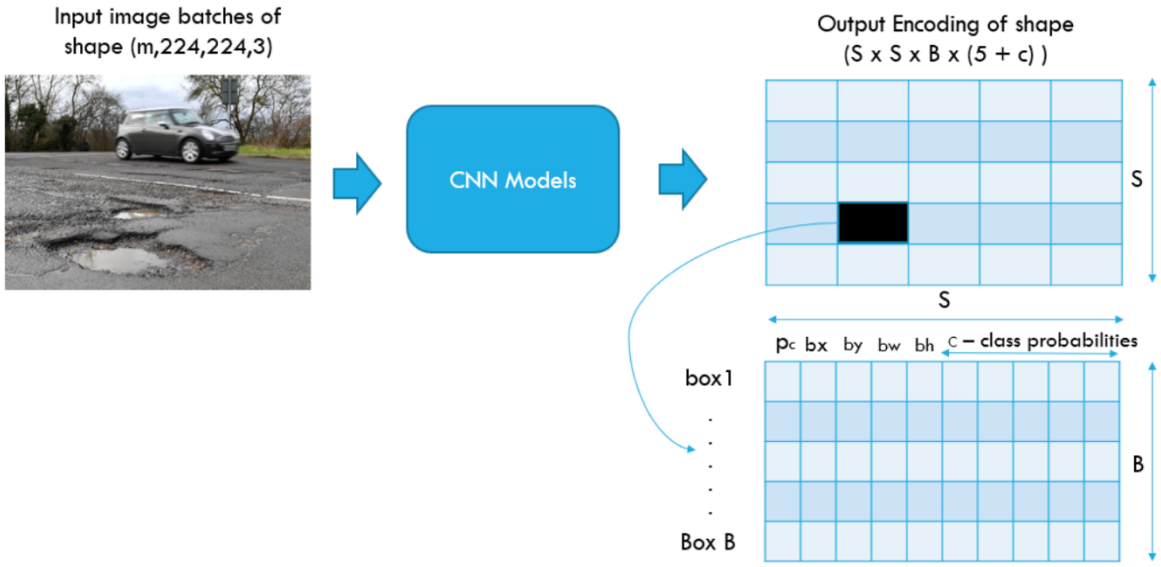
POTHOLE DETECTION USING YOLOV5

1. Introduction:

YOLO which stands for “You only look once” is one of the most popular object detector in use. The algorithm is designed in a way that by a single pass of forward propagation the network would be able to generate predictions. YOLO achieves very high accuracy and works really well in real time detection.

YOLO take a batch of images of shape (m, 224,224,3) and then outputs a list of bounding boxes along with its confidence scores and class labels, (pc,bx,by,bw,bh,c).



The output generated will be a grid of dimensions S x S (eg. 19 x 19) with each grid having a set of B anchor boxes. Each box will contain 5 basic dimensions which include a confidence score and 4 bounding box information. Along with these 5 basic information, each box will also have the probabilities of the classes. So if there are 10 classes, there will be in total 15 (5 + 10) cells in each box.

2. Dataset:

The dataset was collected manually by taking photos of potholes through the smartphone camera. Overall, 200 images were collected which in turn was a combination of images containing single potholes and images containing multiple potholes.

The images are in .jpg format and the dimensions are 3024 x 4032 and few images have the dimensions 1200 x 1600.

We have annotated the dataset using LabelImg tool and the annotations are in YOLO format.

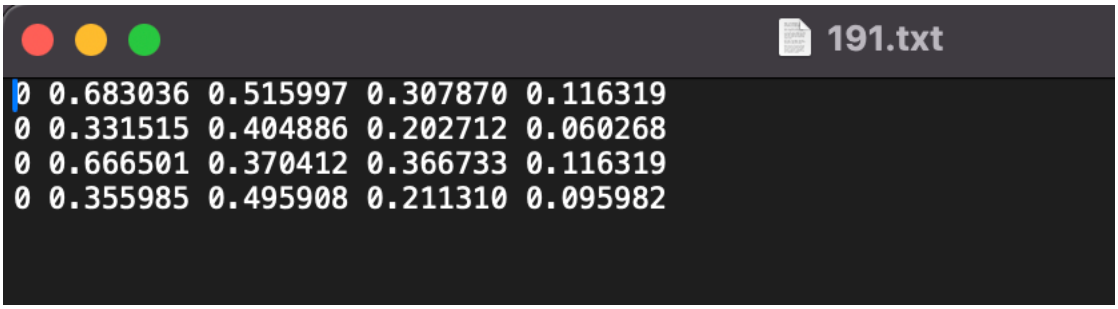
Examples of annotated photos:

YOLO labeling format

Most annotation platforms support export at YOLO labeling format, providing one annotations text file per image. Each text file contains one bounding-box (BBox) annotation for each of the objects in the image. The annotations are normalized to the image size, and lie within the range of 0 to 1. They are represented in the following format:

< object-class-ID> <X center> <Y center> <Box width> <Box height>

If there are mutiple objects in the image, the content of the YOLO annotations text file might look like this:



Data directories structure

To comply with Ultralytics directories structure, the data is provided at the following structure:

For convenience, on my notebook I supplied a function to automatically create these directories.

Configuration files

The configurations for the training are divided to three YAML files, which are provided with the repo itself. We will customize these files depending on the task, to fit our desired needs.

1)The data-configurations file describes the dataset parameters.

potholes_data.yaml

```yaml
names:
- potholes
nc: 1
test: /content/pothholedetection/dataset/data_ready_for_training/test
train: /content/pothholedetection/dataset/data_ready_for_training/train
val: /content/pothholedetection/dataset/data_ready_for_training/val

```

2) The model-configurations file dictates the model architecture.

3) The hyperparameters-configurations file defines the hyperparameters for the training, including the learning rate, momentum, losses, augmentations etc.

Augmentation in YoloV5 is done using Albumentations class.

Here's an example that applies Blur, MedianBlur and ToGray albumentations in addition to the YOLOv5 hyperparameter augmentations normally applied to our training mosaics.

```
class Albumentations:
    # YOLOv5 Albumentations class (optional, used if package is installed)
    def __init__(self):
        self.transform = None
        try:
            import albumentations as A
            check_version(A.__version__, '1.0.3')  # version requirement

            self.transform = A.Compose([
                A.Blur(blur_limit=50, p=0.1),
                A.MedianBlur(blur_limit=51, p=0.1),
                A.ToGray(p=0.3)],
                bbox_params=A.BboxParams(format='yolo',
label_fields=['class_labels']))

            logging.info(colorstr('albumentations: ') + ', '.join(f'{x}' for x in
self.transform.transforms))
        except ImportError:  # package not installed, skip
            pass
        except Exception as e:
            logging.info(colorstr('albumentations: ') + f'{e}')

    def __call__(self, im, labels, p=1.0):
        if self.transform and random.random() < p:
```

```
            new = self.transform(image=im, bboxes=labels[:, 1:],
class_labels=labels[:, 0])  # transformed
            im, labels = new['image'], np.array([[c, *b] for c, b in
zip(new['class_labels'], new['bboxes'])])
        return im, labels
```

3. Transfer learning:

i) model architecture and objective function

Model Architecture:

```yaml
# YOLOv5 🚀 by Ultralytics, AGPL-3.0 license

# Parameters
nc: 80  # number of classes
depth_multiple: 0.67  # model depth multiple
width_multiple: 0.75  # layer channel multiple
anchors:
  - [10,13, 16,30, 33,23]  # P3/8
  - [30,61, 62,45, 59,119]  # P4/16
  - [116,90, 156,198, 373,326]  # P5/32

# YOLOv5 v6.0 backbone
backbone:
  # [from, number, module, args]
  [[-1, 1, Conv, [64, 6, 2, 2]],  # 0-P1/2
   [-1, 1, Conv, [128, 3, 2]],  # 1-P2/4
   [-1, 3, C3, [128]],
   [-1, 1, Conv, [256, 3, 2]],  # 3-P3/8
   [-1, 6, C3, [256]],
   [-1, 1, Conv, [512, 3, 2]],  # 5-P4/16
   [-1, 9, C3, [512]],
   [-1, 1, Conv, [1024, 3, 2]],  # 7-P5/32
   [-1, 3, C3, [1024]],
   [-1, 1, SPPF, [1024, 5]],  # 9
  ]

# YOLOv5 v6.0 head
head:
  [[-1, 1, Conv, [512, 1, 1]],
   [-1, 1, nn.Upsample, [None, 2, 'nearest']],
   [[-1, 6], 1, Concat, [1]],  # cat backbone P4
   [-1, 3, C3, [512, False]],  # 13
```

```
  [-1, 1, Conv, [256, 1, 1]],
  [-1, 1, nn.Upsample, [None, 2, 'nearest']],
  [[-1, 4], 1, Concat, [1]],  # cat backbone P3
  [-1, 3, C3, [256, False]],  # 17 (P3/8-small)

  [-1, 1, Conv, [256, 3, 2]],
  [[-1, 14], 1, Concat, [1]],  # cat head P4
  [-1, 3, C3, [512, False]],  # 20 (P4/16-medium)

  [-1, 1, Conv, [512, 3, 2]],
  [[-1, 10], 1, Concat, [1]],  # cat head P5
  [-1, 3, C3, [1024, False]],  # 23 (P5/32-large)

  [[17, 20, 23], 1, Detect, [nc, anchors]],  # Detect(P3, P4, P5)
 ]
```

Backbone:

The backbone of the YOLOv5 model consists of a series of convolutional layers, C3 blocks, and a Spatial Pyramid Pooling with Factorization (SPPF) layer.

The backbone processes the input image and extracts features at different scales (P1/2, P2/4, P3/8, P4/16, P5/32).

Head:

The head of the YOLOv5 model refines the features obtained from the backbone and produces predictions for object detection.

It includes convolutional layers, upsampling layers, and C3 blocks. It concatenates features from different scales (P3, P4, P5) before making the final detection predictions.

Detect Layer:

The Detect layer at the end of the head is responsible for generating the final detection predictions.

It uses anchors specified in the configuration and produces bounding box predictions for the specified number of classes (nc).

Model Parameters:

nc: Number of classes (set to 80 in this configuration) which is later overridden since our dataset contains only 1 class that is pothole as mentioned in data-config file.

depth_multiple: Model depth multiple (set to 0.67).

width_multiple: Layer channel multiple (set to 0.75).

anchors: Anchor boxes used for detection at different scales.

Objective Function:

The objective function used for training YOLOv5 is typically a combination of classification and regression losses, including:

Classification Loss: Often implemented as Cross-Entropy Loss, measuring the difference between predicted class probabilities and ground truth class labels.

Regression Loss: Typically implemented as Mean Squared Error (MSE), measuring the difference between predicted bounding box coordinates and ground truth bounding box coordinates.

Objectness Loss: A binary classification loss that indicates whether an object is present in a grid cell.

IoU (Intersection over Union) Loss: Penalizes predictions based on the IoU between predicted and ground truth bounding boxes.

These losses are combined to form the final objective function, and the model is trained using techniques like stochastic gradient descent (SGD).

ii) Experiments and results

The pothole dataset is relatively small (200 images), transfer learning is expected to produce better results than training from scratch. Ultralytic's default model was pre-trained over the COCO dataset. COCO is an object detection dataset with images from everyday scenes. Our model will be initialize with weights from a pre-trained COCO model, by passing the name of the model to the 'weights' argument. The pre-trained model will be automatically download.

Feature extraction

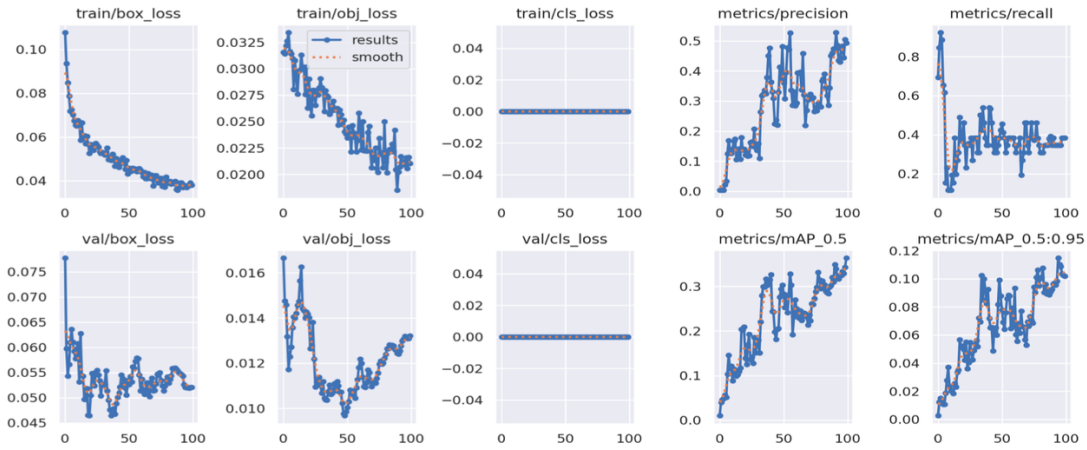
Models are composed of two main parts: the backbone layers which serves as a feature extractor, and the head layers which computes the output predictions. To further

compensate for a small dataset size, we'll use the same backbone as the pretrained COCO model, and only train the model's head. YOLOv5m6 backbone consists of 10 layers, who will be fixed by the 'freeze' argument.

```
!python train.py --img 640 --batch 8 --epochs 100 --data '/content/
↪potholes_data.yaml' --weights 'yolov5m.pt' --project 'runs_pothole' --name␣
↪'feature_extraction' --cache --freeze 10
```

- batch - batch size (-1 for auto batch size). Use the largest batch size that your hardware allows for.
- epochs - number of epochs.
- data - path to the data-configurations file.
- cache - cache images for faster training.
- img - image size in pixels.
- weights — path to initial weights. COCO model will be downloaded automatically.
- freeze — number of layers to freeze
- project— name of the project
- name — name of the run

```
from IPython import display
display.Image(f"runs_pothole/feature_extraction/results.png")
```



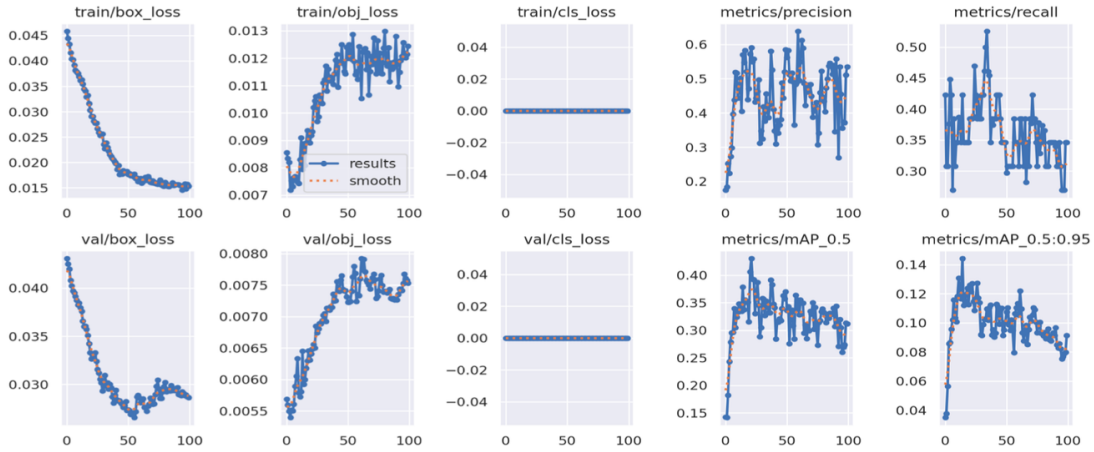
Fine Tuning

The final optional step of training is fine-tuning, which consists of un-freezing the entire model we obtained above, and re-training it on our data with a very low learning rate. This can potentially achieve meaningful improvements, by incrementally adapting the pretrained features to the new data. The learning rate parameter can be adjusted at the hyperparameters-configurations file.

```
!python train.py --img 640 --hyp '/content/yolov5/data/hyps/hyp.VOC.yaml'␣
↪--batch 8 --epochs 100 --data '/content/potholes_data.yaml' --weights '/
↪content/yolov5/runs_pothole/feature_extraction/weights/best.pt' --project␣
↪'runs_pothole' --name 'fine-tuning' --cache
```

- hyp — path to the hyperparameters-configurations file

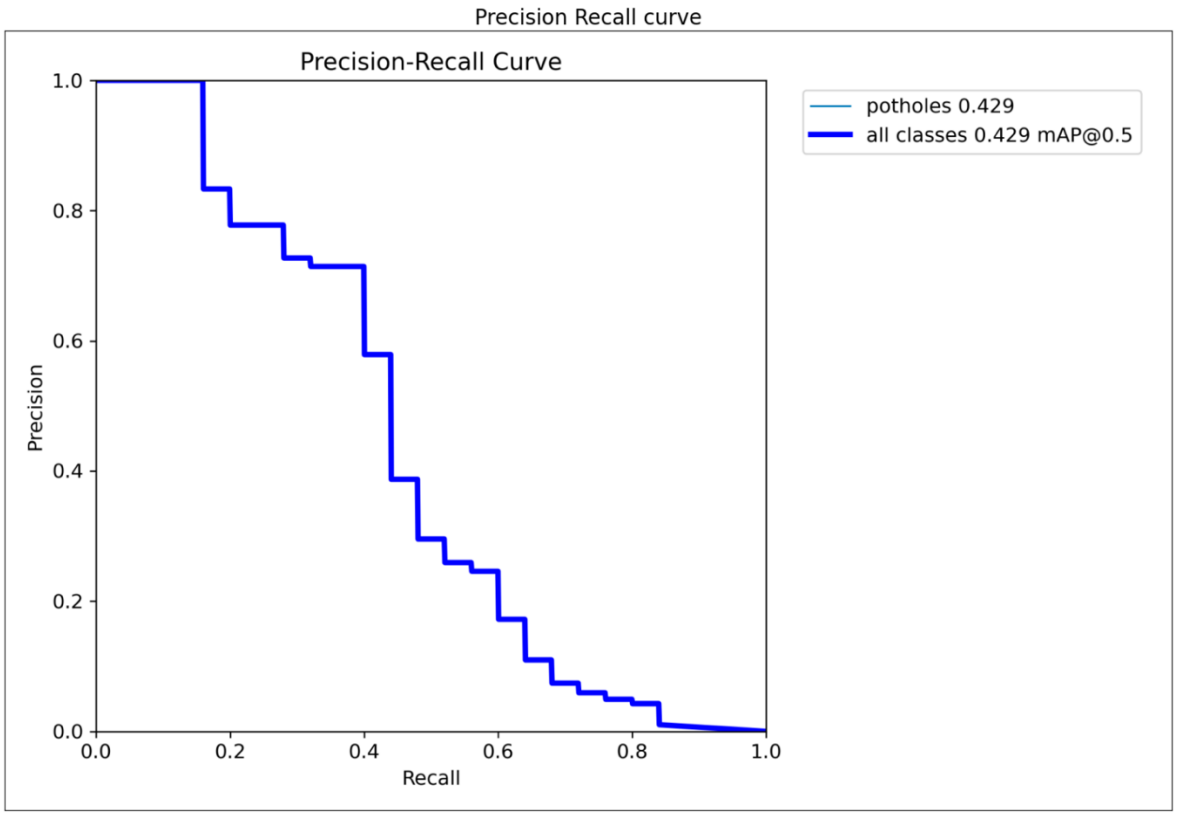
```
display.Image(f"runs_pothole/fine-tuning/results.png")
```



Validation

To evaluate our model we'll utilize the validation script. Performances can be evaluated over the training, validation or test dataset splits, controlled by the 'task' argument. Here, the test dataset split is being evaluated.

```
!python val.py --weights 'runs_pothole/fine-tuning/weights/best.pt' --batch 8␣
↪--data '/content/potholes_data.yaml' --task test --project 'runs_pothole'␣
↪--name 'validation_on_test_data' --augment
```



Precision — Recall Curve of the test data split

iii) Deployment

Gradio is a Python library that allows you to rapidly create UIs for machine learning models and deploy them for sharing or production use. It simplifies the process of building interactive applications around your models. Here is a general overview of the deployment process using Gradio:

1)Load Custom YOLOv5 Model: You load a custom YOLOv5 model using the provided weights file (best.pt).

2)Define Functions for Detection and Drawing:

i)detect_objects: Takes an image, runs it through the YOLO model, and returns the labels, coordinates, and the original image.

```
def detect_objects(image):
    results = model(image)
    labels, cords = results.xyxyn[0][:, -1].numpy(), results.xyxyn[0][:,
:-1].numpy()
    return labels, cords, image
```

ii)draw_boxes: Takes the labels, coordinates, and the original image, and draws bounding boxes around detected objects.

```
def draw_boxes(labels, cords, image):
    image = Image.fromarray(image)
    draw = ImageDraw.Draw(image)
    for cord in cords:
        x_min, y_min, x_max, y_max = cord[:-1]
        x_min, y_min, x_max, y_max = int(x_min * image.width), int(y_min * image.height), int(x_max * image.width), int(y_max * image.height)
        draw.rectangle([x_min, y_min, x_max, y_max], outline ="red", width=2)
    return image
```

3)Define the Prediction Function: predict calls detect_objects and draw_boxes functions, returning the output image.

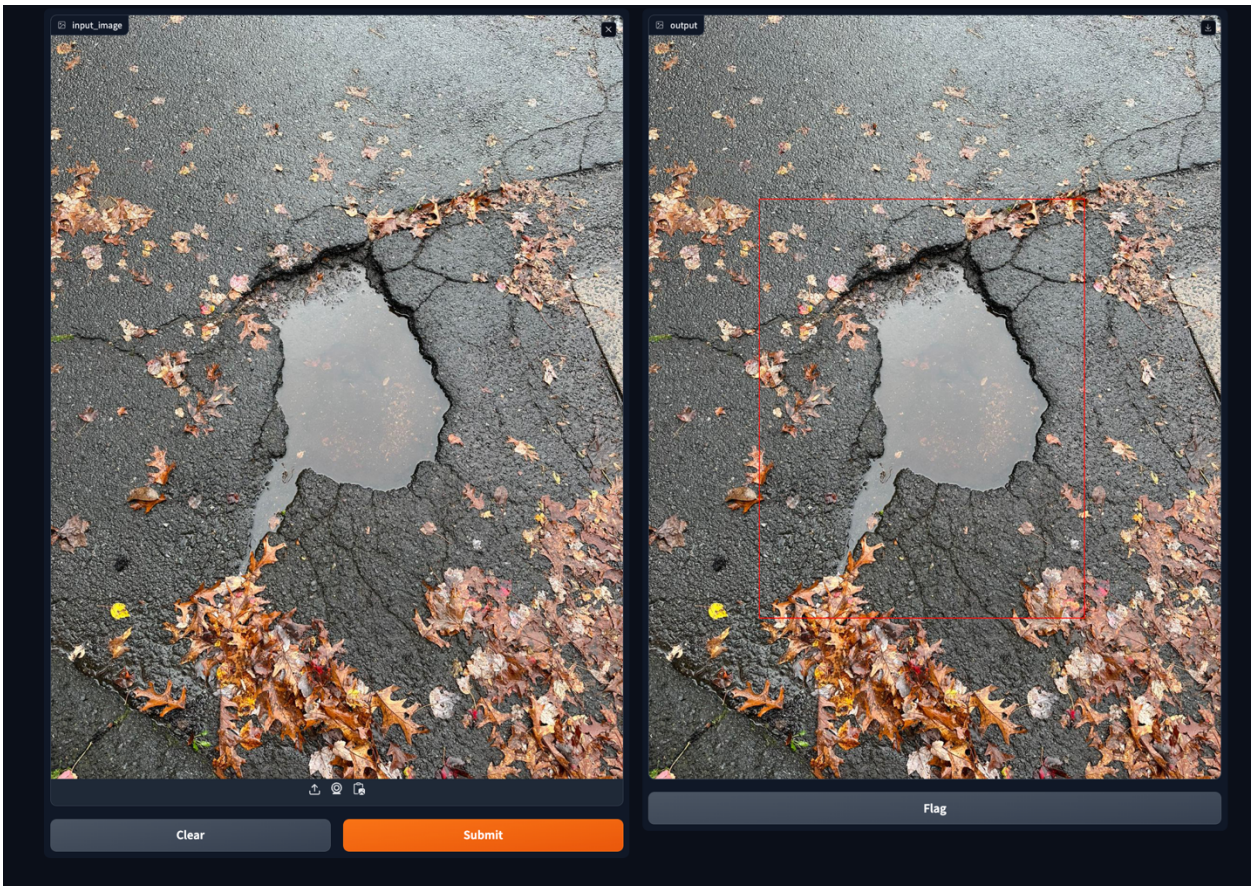
```
def predict(input_image):
    labels, cords, image = detect_objects(input_image)
    output_image = draw_boxes(labels, cords, image)
    return output_image
```

4)Create Gradio Interface: An interface is created with the predict function, taking image inputs and producing image outputs.

```
iface = gr.Interface(fn=predict, inputs="image", outputs="image")
```

5)Launch the Gradio Interface:The interface is launched with debugging enabled.

```
iface.launch(debug=True)
```



4. Mini-network

i) model architecture and objective function:

```python
# Model
class MiniYOLOTransfer(nn.Module):
    def __init__(self, num_classes):
        super(MiniYOLOTransfer, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

        )
        self.fc = nn.Linear(32 * (input_size // 4) * (input_size // 4), num_classes)

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x
```

Input Layer:

Type: nn.Conv2d

Parameters:

Input Channels: 3 (assuming RGB images)

Output Channels: 16

Kernel Size: 3x3

Stride: 1

Padding: 1

Input Size: (Batch Size, 3, input_size, input_size)

Output Size: (Batch Size, 16, input_size, input_size)

ReLU Activation Layer:

Type: nn.ReLU

No parameters.

Input Size: (Batch Size, 16, input_size, input_size)

Output Size: (Batch Size, 16, input_size, input_size)

Max Pooling Layer:

Type: nn.MaxPool2d

Parameters:

Kernel Size: 2x2

Stride: 2

Input Size: (Batch Size, 16, input_size, input_size)

Output Size: (Batch Size, 16, input_size/2, input_size/2)

Convolutional Layer:

Type: nn.Conv2d

Parameters:

Input Channels: 16

Output Channels: 32

Kernel Size: 3x3

Stride: 1

Padding: 1

Input Size: (Batch Size, 16, input_size/2, input_size/2)

Output Size: (Batch Size, 32, input_size/2, input_size/2)

ReLU Activation Layer:

Type: nn.ReLU

No parameters.

Input Size: (Batch Size, 32, input_size/2, input_size/2)

Output Size: (Batch Size, 32, input_size/2, input_size/2)

Max Pooling Layer:

Type: nn.MaxPool2d

Parameters:

Kernel Size: 2x2

Stride: 2

Input Size: (Batch Size, 32, input_size/2, input_size/2)

Output Size: (Batch Size, 32, input_size/4, input_size/4)

Flatten Layer:

Reshapes the tensor for the fully connected layer.

Input Size: (Batch Size, 32, input_size/4, input_size/4)

Output Size: (Batch Size, 32 * (input_size/4) * (input_size/4))

Fully Connected Layer (Linear Layer):

Type: nn.Linear

Parameters:

Input Size: 32 * (input_size/4) * (input_size/4)

Output Size: num_classes

Input Size: (Batch Size, 32 * (input_size/4) * (input_size/4))

Output Size: (Batch Size, num_classes)

Output Layer:

The output layer produces logits for each class.

No activation function is applied in the model's forward method.

ii) Experiments and results:

1. Fine-Tuning:

Fine-tuning with Ray Tune involves experimenting with various configurations of hyperparameters to find the optimal setup for your task. Ray Tune facilitates running multiple trials with different hyperparameter values, allowing you to efficiently explore the hyperparameter space and fine-tune your model.

2. Hyperparameter Selection:

Ray Tune provides a convenient framework for hyperparameter tuning. we define a search space for hyperparameters using tune.loguniform, tune.uniform, or other search space functions. Ray Tune then performs a search over this space, trying different hyperparameter combinations in parallel. The best configuration is automatically selected based on a specified evaluation metric. This allows us to efficiently explore a large hyperparameter space and identify the set of hyperparameters that yields the best performance.

3. Learning Rate Decay Policy:

In deep learning, adjusting the learning rate during training is crucial for convergence. Ray Tune supports integration with PyTorch schedulers for learning rate decay. For example, you can use torch.optim.lr_scheduler.StepLR and adjust the learning rate at predefined steps. Ray Tune's integration with schedulers allows us to explore different learning rate decay policies during hyperparameter tuning.

4. Model Selection:

Ray Tune allows you to experiment with different model architectures by varying hyperparameters related to the model structure. we can define your model in the training function and explore various architectural choices during the hyperparameter search. For instance, we may experiment with different layer sizes, depths, or even entirely different model types.

5. Evaluation:

Ray Tune automates the process of evaluating different hyperparameter configurations. It tracks metrics, such as accuracy or loss, during each trial and identifies the best-performing configuration. After hyperparameter tuning, we can further evaluate the chosen model on a separate validation or test set to assess its generalization performance.

Ray Tune streamlines the hyperparameter tuning process by automating the exploration of hyperparameter spaces, allowing for efficient experimentation with different configurations. It integrates well with popular deep learning frameworks like PyTorch, making it a powerful tool for optimizing model performance. By leveraging Ray Tune, we can conduct experiments in a more systematic and organized manner, leading to better fine-tuning, improved hyperparameter selection, and ultimately, enhanced model performance.

5. Conclusion

Detecting potholes using the YOLOv5 algorithm marks a significant advancement in road safety and infrastructure maintenance. By employing this cutting-edge technology, we can revolutionize the way we identify and address road defects, enhancing both driver safety and municipal efficiency.

The YOLOv5 algorithm's capability to swiftly and accurately detect potholes from various angles and distances offers a proactive approach to road maintenance. Its real-time processing allows for immediate alerts or notifications to road maintenance teams, enabling rapid responses to repair these hazards.

Implementing YOLOv5 for pothole detection ensures a more systematic and timely approach to road maintenance, potentially reducing accidents, vehicle damage, and traffic disruptions. Additionally, by leveraging this technology, municipalities can optimize their resources and prioritize repairs, leading to cost-efficiency and improved road conditions for communities.

However, while the YOLOv5 algorithm showcases tremendous potential, continued development and refinement are essential to enhance its accuracy and reliability further. Collaborative efforts between technology developers, road maintenance authorities, and communities are necessary to fine-tune this technology and ensure its widespread, impactful implementation.

In conclusion, the utilization of YOLOv5 for pothole detection signifies a significant step towards safer roads and streamlined maintenance processes. Its integration promises not only enhanced road safety but also cost-effective and efficient strategies for maintaining the integrity of our road infrastructure.