Pothole Detection Using YOLOv5 Algorithm

Introduction

Pothole detection is a critical aspect of road maintenance and safety. Accurate identification of potholes allows for timely repairs, preventing potential hazards for motorists and pedestrians. In this report, we detail the steps taken to collect a dataset for pothole detection and the preprocessing methods applied to ensure optimal model training.

Data Collection and Annotation

We collected a diverse dataset containing 200 samples for group use and 100 samples for individual use, totaling 300 annotated images. The dataset is hosted on OneDrive, and the link is provided :

https://unhnewhaven-my.sharepoint.com/:f:/g/personal/dmota1_unh_newhaven_edu/EhLLZbSE9PNBtIWo4duBBxEBzF3yBb-DbZGcqwGgRMvOEg?e=EsrARW

. Annotations include precise locations of potholes within each image, enabling the development of an effective detection model.

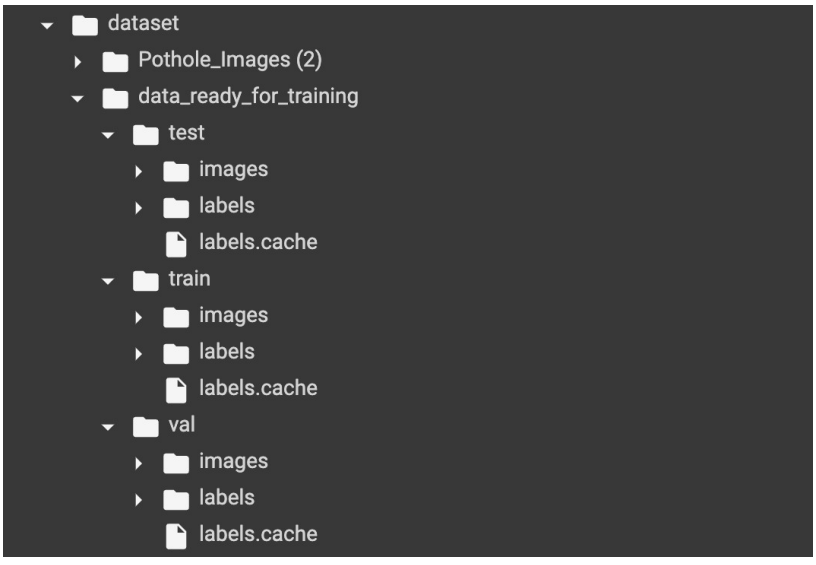
YOLO labeling format

Most annotation platforms support export at YOLO labeling format, providing one annotations text file per image. We have used LabelImg tool to annotate our dataset in YOLO format. 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>

Dataset Loading

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



We have written code in python to split the dataset into 80% train 10% test and 10% val. The code also futher creates the directory structure as shown in the image above and copies the images along with its annotation files to the respective folders.

Dataset Partitioning

To ensure a robust evaluation of the model, the dataset was partitioned into three sets: 80% for training, 10% for validation, and 10% for testing. This partitioning strategy helps maintain a balance in the distribution of pothole images across different sets.

Number of samples in the training set: 160

Number of samples in the validation set: 20

Number of samples in the test set: 20

Normalization and Augmentation

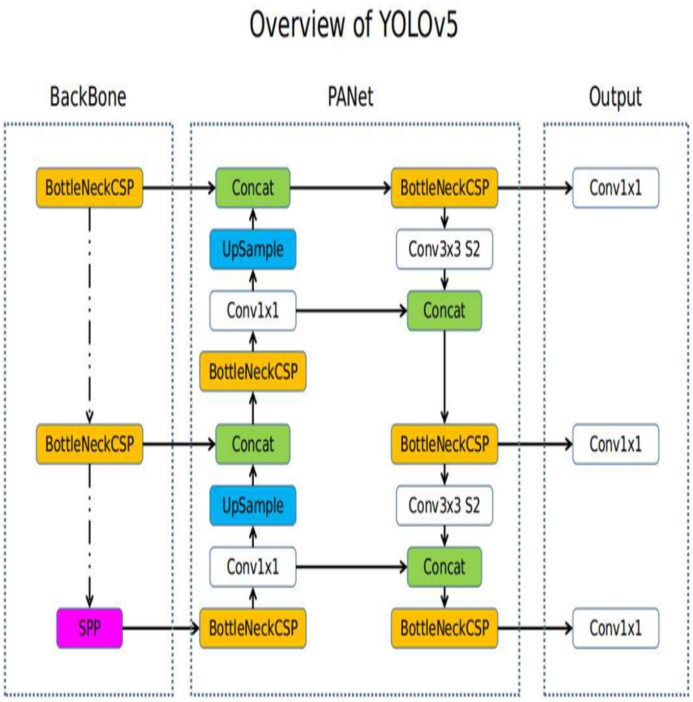
Normalization and augmentation techniques, in line with the recommendations from the selected paper, were incorporated into the data loading process. These techniques enhance the model's ability to generalize and detect potholes under various conditions.

```
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
```

Transfer learning

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. YOLOv5m backbone consists of 10 layers, who will be fixed by the 'freeze' argument.

```
# 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)
  ]
```



Feature Extraction

```
[9]: !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
```

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

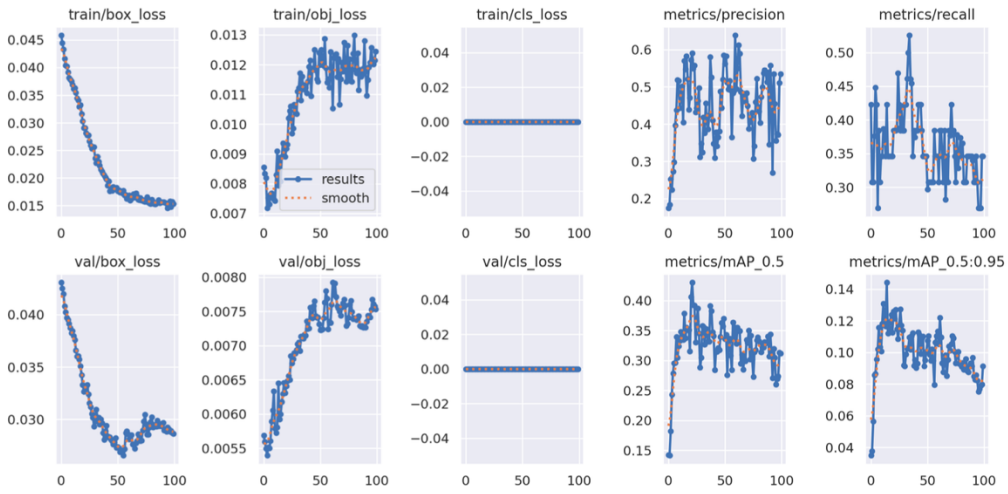
achieve meaningful improvements, by incrementally adapting the pretrained features to the new data. The learning rate parameter can be adjusted at the hyperparameters-configurations file. For the tutorial demonstration, we'll adopt the hyperparameters defined at built-in 'hyp.VOC.yaml' file, which has much smaller learning rate then the default. The weights will be initialized with the weights saved on the previous step.

Fine Tuning

```
[11]: !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
```

```
[12]: display.Image(f"runs_pothole/fine-tuning/results.png")
```

[12]:



box_loss — bounding box regression loss (Mean Squared Error).

obj_loss — the confidence of object presence is the objectness loss.

cls_loss — the classification loss (Cross Entropy).

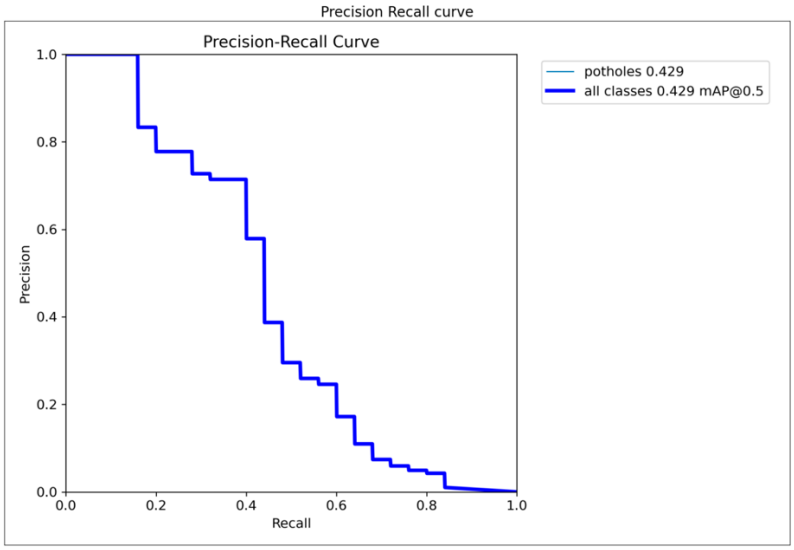
Validation

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:

Validation

[13]:
```
!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
```

We can also obtain the Precision-Recall curve:



Prediction

```
!python detect.py --source '/content/pothholedetection/dataset/
↪data_ready_for_training/test/images' --weights 'runs_pothole/fine-tuning/
↪weights/best.pt' --max-det 3  --conf-thres 0.25 --classes 0 --name␣
↪'detect_test'
```

Sample of Images :

Conclusion

This report outlines the essential steps taken to prepare our dataset, train , fine tune and finally perform pothole detection using the YOLOv5 algorithm. The combination of effective data collection, loading, partitioning, and preprocessing laid the foundation for the subsequent fine-tuning and evaluation phases which resulted in the almost accurate prediction of the potholes using the algorithm.