RBE 549 Computer Vision, Fall 2021

Chess Piece Detection and Orientation Estimation

Project Report

Team RoyalLumens

Member	Signature	Contribution (%)
Suketu Parekh	[illegible]	25
Purna Patel	Purna	25
Prasham Patel	Prasham	25
Neel Dharmadhikari	[illegible]	25

Grading:	Approach	_____/15
	Justification	_____/5
	Analysis	_____/15
	Testing & Examples	_____/15
	Documentation	_____/10
	Difficulty	_____/10
	Professionalism	_____/10
	Presentation	_____/20
	Total	_____/100

Chess Piece Detection and Orientation Estimation

Suketu Parekh, Purna Patel, Prasham Patel and Neel Dharmadhakari

Abstract* – *Object detection and recognition has found a large number of applications in the real world; more so since the advent of the self-driving vehicles have been stared to be thought as a achievable reality. There are a number of ways and algorithms that are available that perform this task of object detection with varying levels of success and accuracy in cluttered environments. The report outlines the work done towards the term project for the course of Computer Vision. ORB feature matching is employed to estimate the orientation of articles with respect to the same in a reference image. YOLO V5 is used for the detection of Chess Pieces. A brief overview of the process of Dataset generation and training has been included here

1. Introduction

Computer vision is the ability of the computer to understand the composition of various images and videos. While this is a simple task for humans, it is a big challenge for computers to detect the images. Many mathematical, hardware as well as software techniques have been developed to try to solve this problem. There are many applications of computer vision such as self-driving cars, facial recognition etc. We should be able to detect a particular object under different orientations, scales and lighting conditions. This approach is more useful than just detecting the object against a white background.

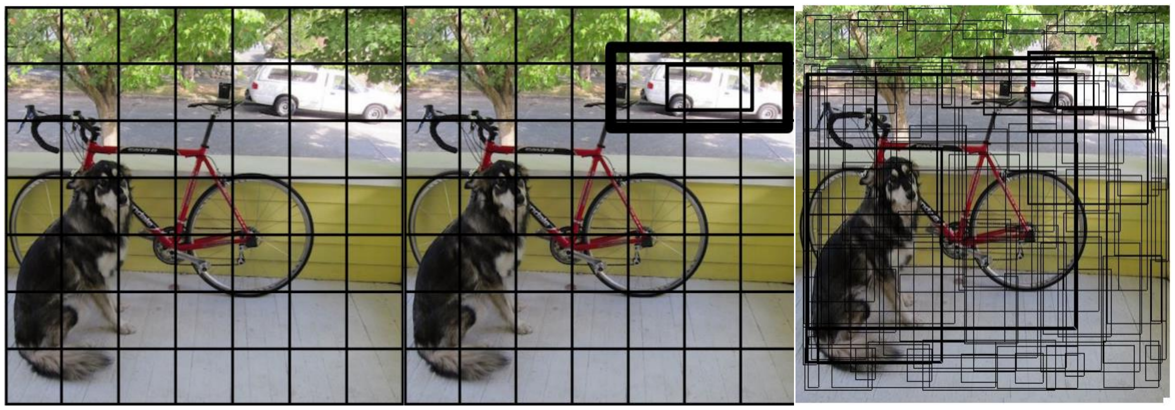
Our task is to detect the Chess pieces. The pieces may be in different orientation and under different lighting. The pieces may also be partially covered or may be at any place in the image. Our aim is to detect the pieces under any of the above defined conditions. For this task, we have used the YOLOv5 object detection algorithm.

2. Methodology

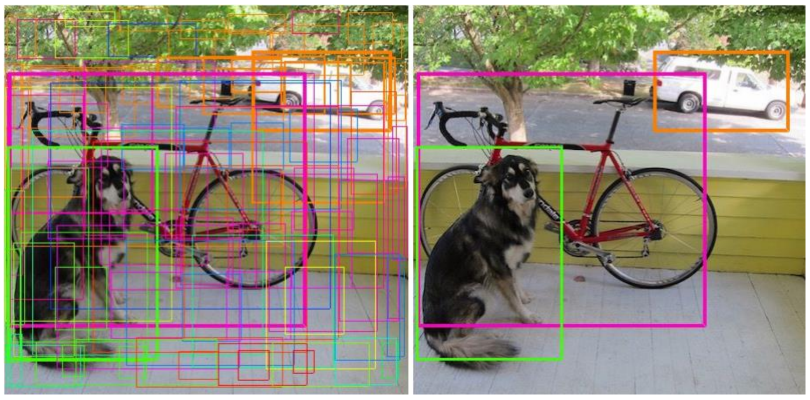
We have used the YOLOv5s model for the detection of chess pieces. YOLO is an acronym for 'you only look once'. It is one of the most famous and the most widely used object detection algorithms due to its speed and accuracy. YOLO provides real time object detection.

The algorithm divides images in grid systems where each cell in the grid is responsible for detecting objects within itself. Then, it predicts where to place bounding boxes. The bounding boxes are predicted with regression based algorithms, as opposed to a classification based one.

Generally, classification algorithms work in 2 steps: first, selecting the region of interest and then applying the convolutional neural network to the regions selected. The reason why YOLO is fast is because of YOLO's regression algorithm. It looks at the whole image at once to predict the bounding boxes.



a. b. c.



d. e.

Fig.2.1: Working of YOLO[1]

The working of YOLO can be seen in the figures above. First, as seen in fig a., the image is divided into a 7x7 grid. Then each cell calculates the probability of an object being in that cell as shown in figures b. and c. and the boxes are drawn. Each cell also predicts the class probabilities of that cell being in a particular class. After combining the box and class probabilities, figure d. is obtained. Finally, threshold detection is done as shown in the fig. e.

There are different types of YOLOv5 such as YOLOv5s, YOLOv5m, YOLOv5I, YOLOv5x. As the model gets bigger, it's speed decreases while it's accuracy increases. YOLOv5s is the lightest model with size 14MB and the fastest inference time. On the other hand, YOLOv5x is the largest model with a size of 168MB and is the most accurate one.

Performance of any neural network depends on the activation function it uses. YOLOv5 uses Sigmoid and Leaky ReLU activation functions. The Leaky ReLU is used in the middle layers and the Sigmoid activation function is used for the final detection layer. The default optimization function used in YOLOv5 is SGD. It can be changed to Adam.

The preparation of data also has an effect on the accuracy of the algorithm. Open source tools such as Roboflow Annotate, LabelImg can be used to label the images. More labelled images gives a better system performance. Pictures of the object should be taken from as many angles as possible to make the detection rotationally invariant.

The YOLO architecture consists of following blocks

- Backbone-

It uses CSPDarknet(cross stage partial networks) to extract key features from the input images. The backbone is run on a GPU or CPU platform.

- Neck-

It uses PANet to generate feature pyramids to perform aggregation on the features and pass it to the head for prediction. It helps in detection of the same object in various scales. Feature models are useful in assisting models to perform effectively on previous unseen data.

- Head-

The head is responsible for the final detection step. It generates the final output vectors with class probabilities and bounding boxes. A single stage head is used for dense prediction(YOLO, SSD, RetinaNet) and a two stage(Faster R-CNN) head can be used for the sparse prediction object detector.

The diagram for the architecture of the YOLOv5 is given below.

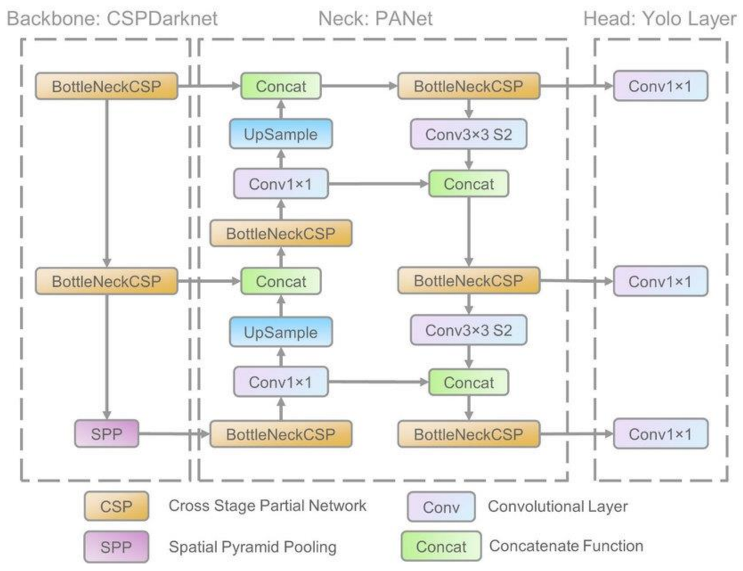


Fig.2.2: YOLO Architecture [2]

2.1 Dataset Preparation:

The dataset is the most crucial part of any deep-learning based object detector. The quality and quantity of data will decide the accuracy for our prediction. In our case we will have 4 classes white-knight, White-rook, black-bishop and black-pawn.

We have tried to capture images from different directions and orientations so as to make our detector rotation invariant. However, for pawn, it is symmetric about its vertical axis, so we do not need to vary its orientation about the vertical axis. Some of the images in the training data are shown below.

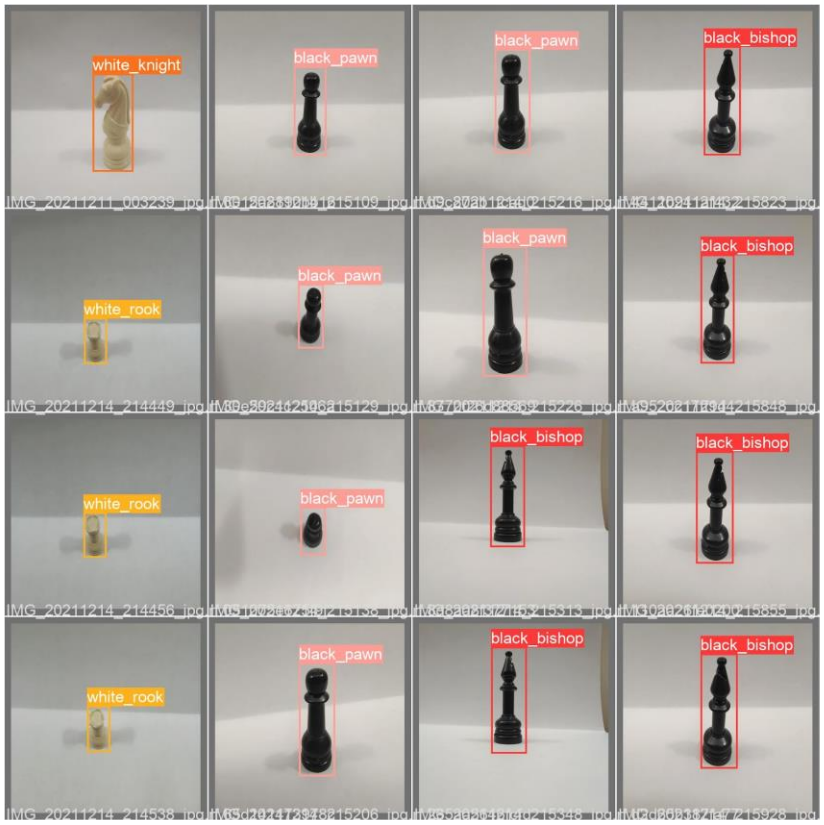


Fig.2.3: Training dataset

The .yaml file and annotation were prepared using the labelImg tool. The annotation for YOLO are in .xml format. A total of 165 images were used for training. The number is quite less. But given our detection with white background, they are sufficient.

2.2 Training:

Training was performed on Google collab as the training time decreased significantly. The images were resized to 416x416 and a total of 320 epochs were performed. Again any more serious dataset would require more than 3000 epochs, but for our purpose 320 are enough. The below given image shows the training and validation losses and other parameters which change with each epoch.

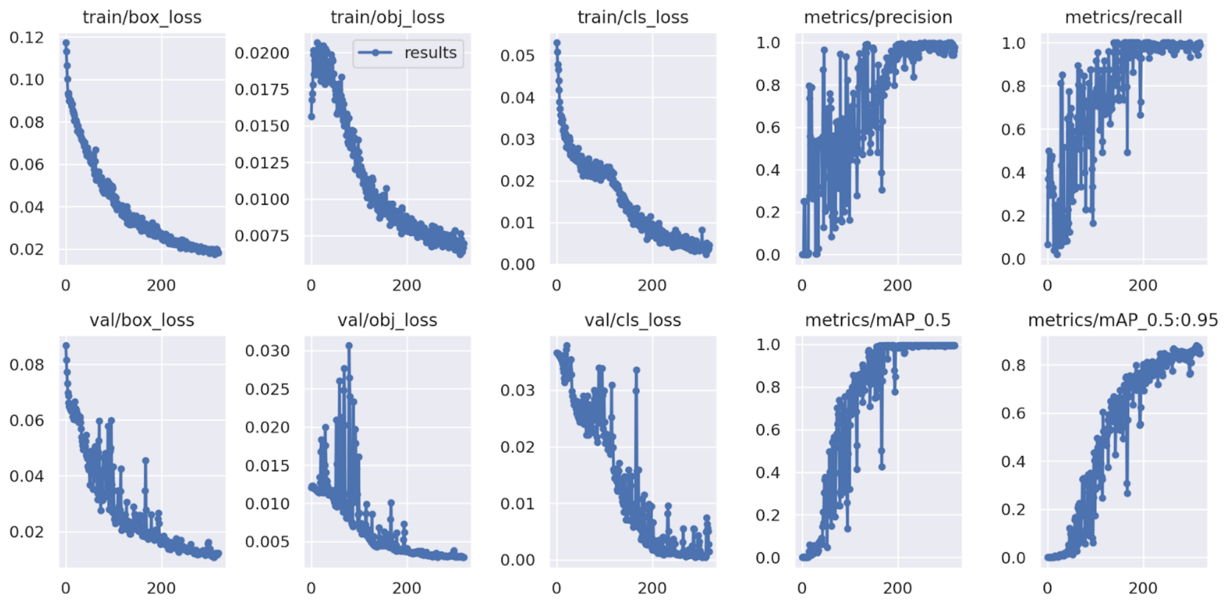


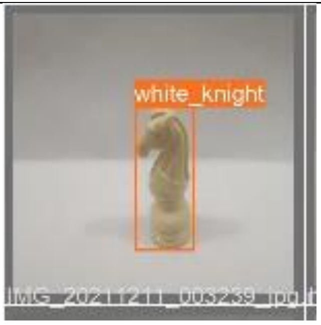
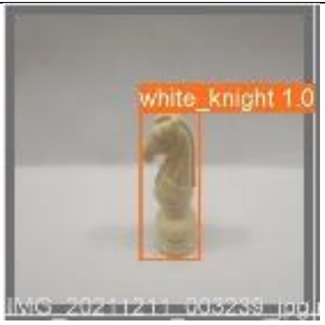
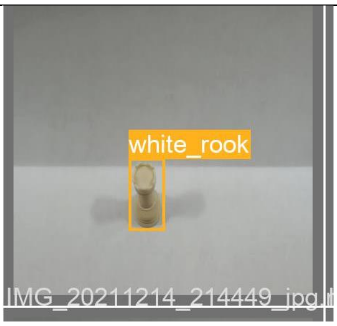
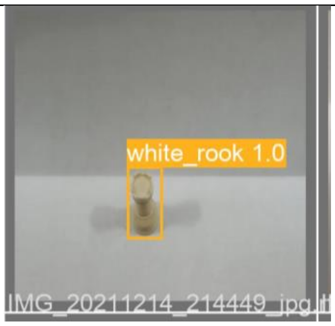
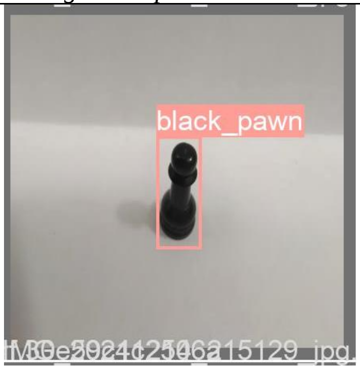
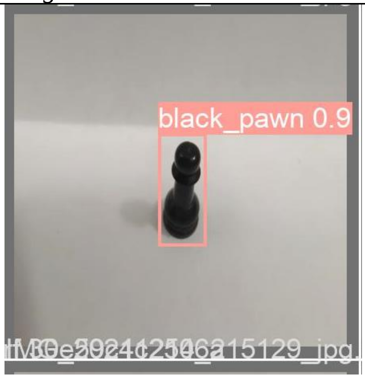
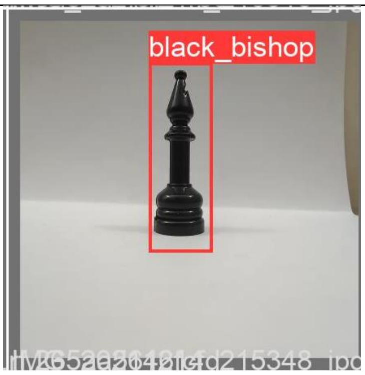
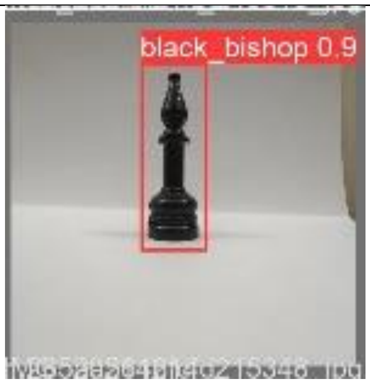
Fig.2.4: Training Progress Plots

The training and validation losses get pretty low. The model almost overfits the data, however as the background during application will also be white we will get accurate bounding boxes. These bounding boxes will be very useful for finding accurate orientation estimation. Thus this bit of overfitting is desirable.

3. Results

After training the generated Dataset for about 320 epochs, the desired results were obtained. The threshold confidence was experimented with and the finally settled for a value of 0.75 or 75%; meaning that the algorithm would not label a certain image if the confidence in the prediction is lower than 75%.

The following table depicts the actual class for an image as in a part of the dataset along with the designated label of the same on the left column. These images were used as a part of the test partition of the dataset. The predicted class of the same images as the given output by the model is shown in the right column. The confidence with which the model has predicted the class of the piece in the image is also mentioned besides the class.

Actual Class	Predicted Class
 *Fig.3.1: Input White Knight*	 *Fig.3.2: Predicted White Knight*
 *Fig.3.3: Input White Rook*	 *Fig.3.4: Predicted White Rook*
 *Fig.3.5: Input Black Pawn*	 *Fig.3.6: Predicted Black Pawn*
 *Fig.3.7: Input Black Bishop*	 *Fig.3.8: Predicted Black Bishop*

It is observed that the confidence for the predicted classes ‘black_pawn’ and ‘black_bishop’ are on the lower side as compared to the other classes. It is reflected that one of the factors playing a major role in this is the facts of rotational symmetry and the similarity of structure between the said classes.

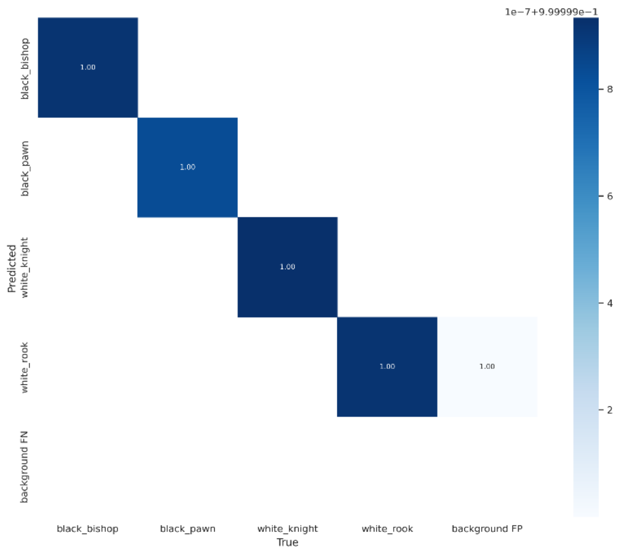


Fig.3.9: Confusion Matrix

Figure 3.9 shows the Confusion Matrix for the Sub-Dataset of 4 classes. As can be seen, the Confusion matrix is an Identity. This is because the model was overfitted to the input dataset. The figure 3.10 depicts a bar chart showing the number of instances of each class. One of the observations is that the number of instances for the class ‘black_pawn’ are significantly lower than the instances for other classes. This is again attributed to the fact that the structure of pawn is completely rotationally symmetric.

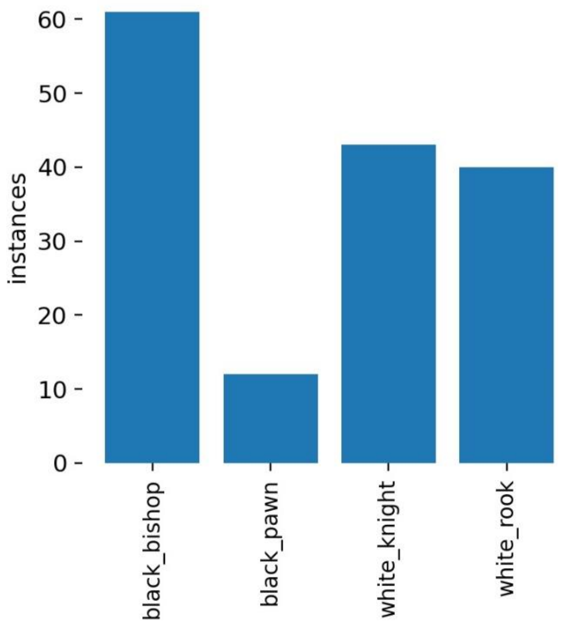


Fig.3.10: Instances of Classes

Figure 3.11 shows two of the scatter plots for the size and the location for the resulting bounding boxes. The plot on the left shows the location of bounding boxes within the input images. It is inferred that maximum frequency is observed in the center of the image, which is consistent with the expectations as most of the images were taken considering the pieces in the center. The plot on the right shows the sizes of the resultant bounding boxes in terms of height and width.

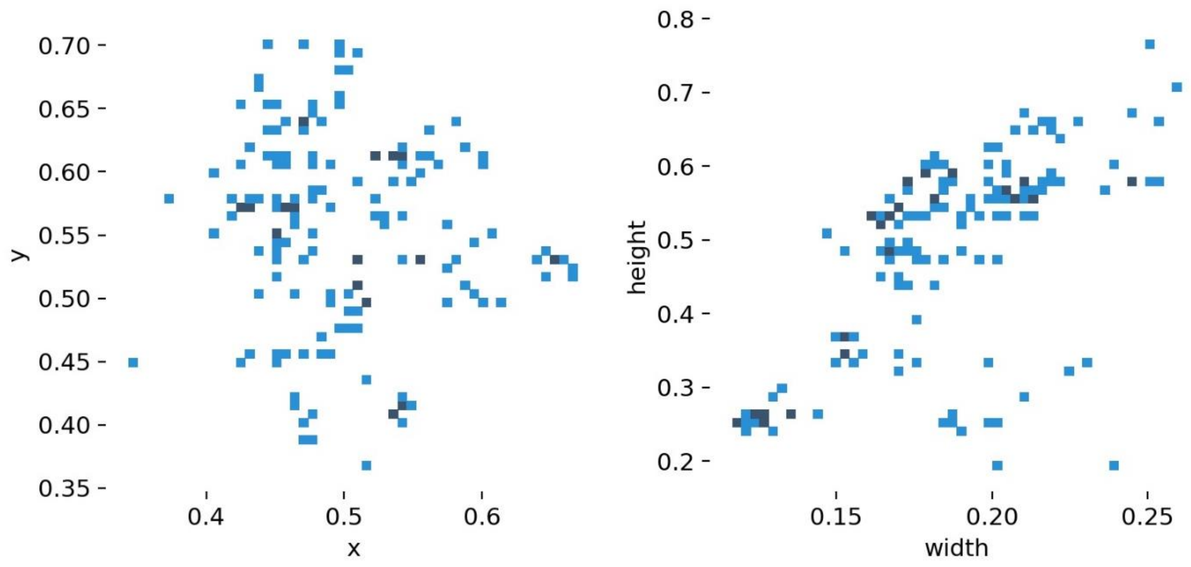


Fig.3.11: Scatter Plots for Bounding Boxes

4. Orientation Estimation

For orientation estimation we generate an essential matrix between each consecutive frames. The essential matrix basically encodes the epipolar geometry of the 2 camera scenes viewed from different angles. The below given is the equation for essential matrix.

$$P_r^T * E * P_l = 0$$

It should be noted that the Pr and Pl are the co-ordinates of the feature points in left and right images [5]. And a sufficient number of this points can help get Essential Matrix by using Singular Value Decomposition.

4.1 ORB Feature Detection

ORB stands for Oriented FAST and Rotated BRIEF, it is a fusion between the FAST features and the BRIEF features. For intuition, assume there is a pixel ‘p’ and 14 pixel surrounding pixel ‘p’, as shown in the Figure 4.1. Then the algorithm computes intensity for each of the pixel and if more than 8 pixel surrounding have intensity higher than ‘p’, pixel ‘p’ is registered as key point or a feature.

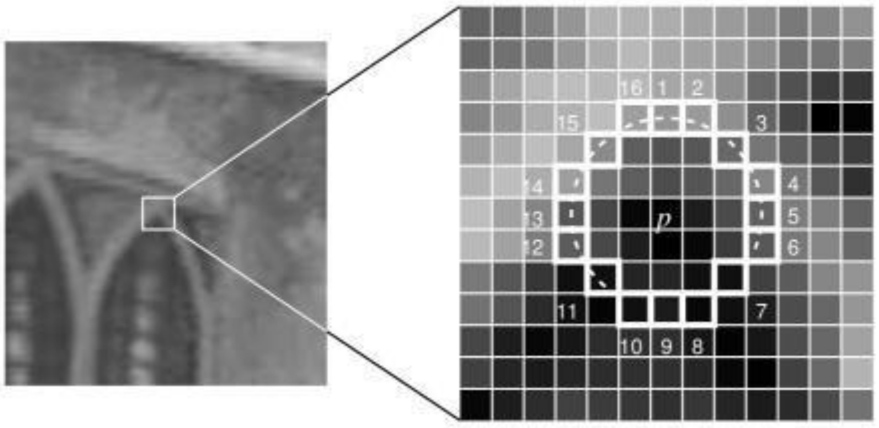


Fig.4.1: Key-point [3]

ORB algorithm also finds such features in multiple scaled copies of the image as shown in Figure 4.2. The reason behind this is to make it scale invariant and thus is particularly useful in our case. As the object moves far away or get near its scale on the projection plane changes. Thus a algorithm which is scale invariant will give us very robust features for real time tracking.

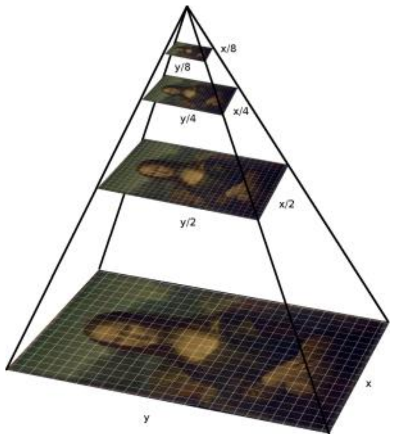


Fig.4.2: Scaled Images for finding features [3]

4.2 Matching features in consecutive frames

Matching of the pixels between two frames is done with help of a brute force matcher based on Norm-Hamming Distance. Norma-Hamming distance is just counting the number of ones after applying XOR operation between two arrays. The search for the feature in the corresponding frame is done with the help of K-nearest neighbor algorithm. And further a D.Lowe’s ratio test is applied to further increase the accuracy of matches.

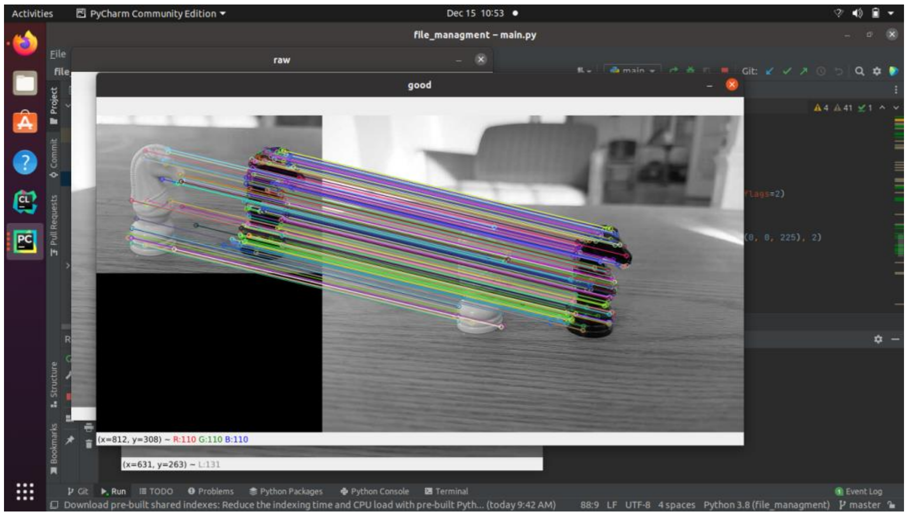

Activities
PyCharm Community Edition
Dec 15 10:53
file_managment – main.py
raw
good
(x=812, y=308) ~ R:110 G:110 B:110
(x=631, y=263) ~ L:131
Git
Run
TODO
Problems
Python Packages
Python Console
Terminal
Event Log
Download pre-built shared indexes: Reduce the indexing time and CPU load with pre-built Pyth... (today 9:42 AM)
88:9 LF UTF-8 4 spaces Python 3.8 (file_managment) master

Fig.4.3: KNN Matcher result

4.3 Generating Essential Matrix

After we get the feature points from two consecutive frame we generate the essential matrix by solving it as a least squares problem. The solution is provided by 8-point algorithm. The decomposition of essential matrix can be found in Hartley & Zisserman's book [4]. The Essential

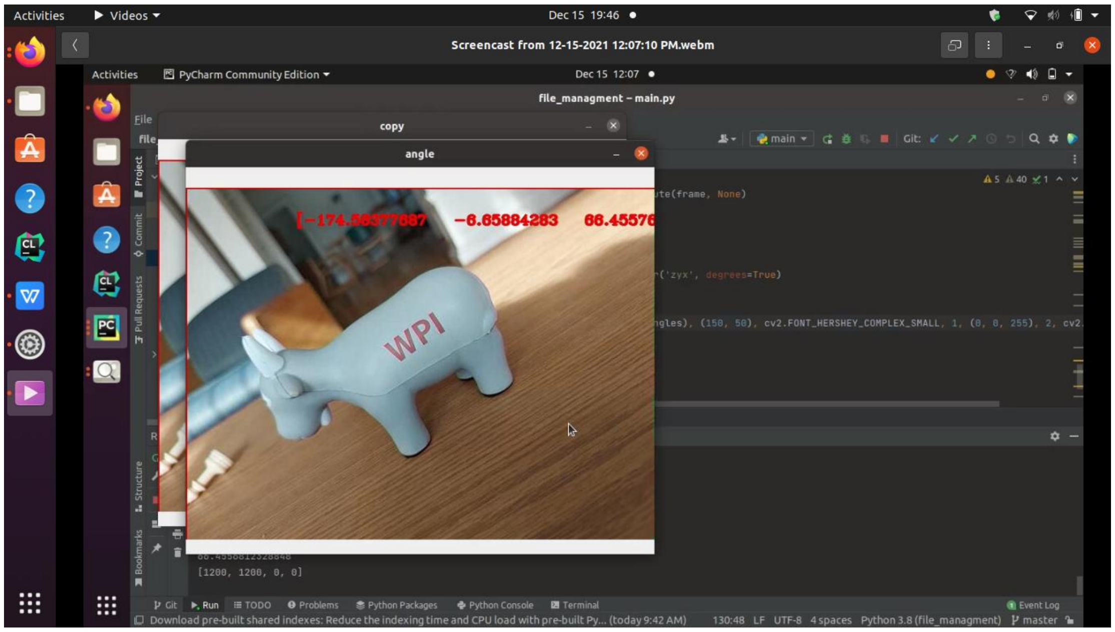
Matrix will give us a rotational matrix and a translational matrix, However, the translational matrix will not be upto scale.

Flow of the Code:

- First step is to get the object that we want to detect. generally it would be done by by using an object detector to get bounding box, but here we are manually selecting the object.
- Next, we detect the object features to match them with the object detected in the next frame.
- We again detect the object in the next frame and extract its features to match with the object in previous frame.
- The set of matching points are used to calculate the Essential Matrix. Which is decomposed to get the rotational matrix.
- We need to take inverse of these matrix to get the rotation of object with respect to camera.
- Rotational matrix obtained is multiplied with the older matrix to get the rotation with respect to first frame.
- Loop again to detect object in next frame and continue.

4.4 Setbacks in application

- We could not integrate the YOLO frame work with OpenCV to use YOLO's bounding box to get accurate object orientation.
- We had to manually detect the bounding box, and it was updated with max and min position of the object features.
- Current implementation will fail if there is some other movement in background.



```
ute(frame, None)
r('zyx', degrees=True)
ngles), (150, 50), cv2.FONT_HERSHEY_COMPLEX_SMALL, 1, (0, 0, 255), 2, cv2.
```

```
66.4556812328848
[1200, 1200, 0, 0]
```

Download pre-built shared indexes: Reduce the indexing time and CPU load with pre-built Py... (today 9:42 AM)   130:48 LF UTF-8 4 spaces Python 3.8 (file_managment) master

Fig.4.4: Final Output

5. Conclusion

In this project, we trained a YOLOv5 network to detect chess pieces from different angles. The detection confidence of the network for most test images is 0.85. The data might be overfitted but given our application, where the background is consistent, it is desirable.

These accurate bounding boxes then can be used to get the orientation of different objects detected even if they rotate differently. However, for the time being, the YOLOV5 framework is not integrated with the orientation estimation code.

The scope of application for such an orientation algorithm would be mostly in odometry. As only a single camera could be used to get the orientation of our system with respect to some known landmark. Also, to improve upon, segmentation should be preferred over bounding boxes to further increase the feature detection and matching accuracy which would in turn increase orientation estimation accuracy.

6. References

[1] Xu, Renjie, et al. "A Forest Fire Detection System Based on Ensemble Learning." Forests 12.2 (2021): 217.

[2] YOLO: You Only Look Once [Online] - https://web.cs.ucdavis.edu/~yjlee/teaching/ecs289g-winter2018/YOLO.pdf

[3] Introduction to ORB (Oriented FAST and Rotated BRIEF) [Online] -https://medium.com/data-breach/introduction-to-orb-oriented-fast-and-rotated-brief-4220e8ec40cf

[4] Hartley, Richard; Andrew Zisserman (2004). Multiple view geometry in computer vision (2nd ed.). Cambridge, UK. ISBN 978-0-511-18711-7. OCLC 171123855

[5] H. Christopher Longuet-Higgins (September 1981). "A computer algorithm for reconstructing a scene from two projections". Nature. **293** (5828): 133–135. Bibcode:1981Natur.293..133L. doi:10.1038/293133a0. S2CID 4327732

Appendices

YOLO.ipynb

```
# clone YOLOv5 and reset to a specific git checkpoint that has been verifi
ed working
!git clone https://github.com/ultralytics/yolov5  # clone repo
%cd yolov5
```

```
pip install torch==1.7.0+cu110 torchvision==0.8.1+cu110 torchaudio===0.7.0
 -f https://download.pytorch.org/whl/torch_stable.html
```

```
%cd /content
!curl -
L "https://app.roboflow.com/ds/0lEYr0jPPa?key=h9fBmOZLja" > roboflow.zip;
unzip roboflow.zip; rm roboflow.zip

# !curl -
L "https://app.roboflow.com/ds/APyxMvoJoF?key=TxZ6rZDbRK" > roboflow.zip;
unzip roboflow.zip; rm roboflow.zip
# %cd /content
# !curl -
L "https://app.roboflow.com/ds/foSYc9fSvM?key=FbqrlBNIOX" > roboflow.zip;
unzip roboflow.zip; rm roboflow.zip
```

```
# this is the YAML file Roboflow wrote for us that we're loading into this
 notebook with our data
%cat data.yaml
```

```
# define number of classes based on YAML
import yaml
with open("/content/data.yaml", 'r') as stream:
    num_classes = str(yaml.safe_load(stream)['nc'])
```

```
#this is the model configuration we will use for our tutorial
%cat /content/yolov5/models/yolov5s.yaml
```

```
#customize iPython writefile so we can write variables
from IPython.core.magic import register_line_cell_magic

@register_line_cell_magic
```

```
def writetemplate(line, cell):
    with open(line, 'w') as f:
        f.write(cell.format(**globals()))
```

```
%%writetemplate /content/yolov5/models/custom_yolov5s.yaml

# parameters
nc: {num_classes}  # number of classes
depth_multiple: 0.33  # model depth multiple
width_multiple: 0.50  # layer channel multiple

# anchors
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 backbone
backbone:
  # [from, number, module, args]
  [[-1, 1, Focus, [64, 3]],  # 0-P1/2
   [-1, 1, Conv, [128, 3, 2]],  # 1-P2/4
   [-1, 3, BottleneckCSP, [128]],
   [-1, 1, Conv, [256, 3, 2]],  # 3-P3/8
   [-1, 9, BottleneckCSP, [256]],
   [-1, 1, Conv, [512, 3, 2]],  # 5-P4/16
   [-1, 9, BottleneckCSP, [512]],
   [-1, 1, Conv, [1024, 3, 2]],  # 7-P5/32
   [-1, 1, SPP, [1024, [5, 9, 13]]],
   [-1, 3, BottleneckCSP, [1024, False]],  # 9
  ]

# YOLOv5 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, BottleneckCSP, [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, BottleneckCSP, [256, False]],  # 17 (P3/8-small)
   [-1, 1, Conv, [256, 3, 2]],
   [[-1, 14], 1, Concat, [1]],  # cat head P4
   [-1, 3, BottleneckCSP, [512, False]],  # 20 (P4/16-medium)

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

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

```
# train yolov5s on custom data for 100 epochs
# time its performance
%%time
%cd /content/yolov5/
!python train.py --img 416 --batch 16 --epochs 320 --data '../data.yaml' --cfg ./models/custom_yolov5s.yaml --weights '' --name yolov5s_results  --cache
```

```
# when we ran this, we saw .007 second inference time. That is 140 FPS on a TESLA P100!
# use the best weights!
%cd /content/yolov5/
!python detect.py --weights runs/train/yolov5s_results2/weights/best.pt --img 416 --conf 0.75 --source ../test/images
```

OrientationEstimation.py

```
import cv2
import numpy as np
import math
from scipy.spatial.transform import Rotation as R

def get_object_feat():
    ret, frame = vid.read()
    # frame = cv2.cvtColor(frame) # we don't need to take grey scale and
blur it as it degrades performance for our application
    roi = cv2.selectROI(windowName="ROI", img=frame, showCrosshair=True,
fromCenter=False)
    x, y, w, h = roi
    init_frame = frame[y:y + h, x:x + w]
    # init_frame = cv2.GaussianBlur(init_frame,(9,9),0)
    cv2.imshow("init_frame", init_frame)
    cv2.waitKey(3000)
    kp, des = orb.detectAndCompute(init_frame, None)
    cv2.destroyAllWindows()
    return kp, des, frame, init_frame, x, y

def matcher(kp1, des1, kp2, des2, x, y):
    matches = bf.knnMatch(des1, des2, k=2)

    pts1 = []
    pts2 = []
    good = []

    min_w = 1200
    min_h = 1200
    max_w = 0
    max_h = 0

    for m, n in matches:
        if m.distance < 0.95*n.distance:
            good.append([m])
            if (min_w < kp2[m.trainIdx].pt[0]):
                min_w = kp2[m.trainIdx].pt[0]
            elif (max_w > kp2[m.trainIdx].pt[0]):
                max_w = kp2[m.trainIdx].pt[0]
            if (min_h < kp2[m.trainIdx].pt[1]):
                min_h = kp2[m.trainIdx].pt[1]
            elif (max_h > kp2[m.trainIdx].pt[1]):
                max_h = kp2[m.trainIdx].pt[1]
            # print(m.distance)print(temp_co)
            pts2.append(kp2[m.trainIdx].pt)
            pts1.append([kp1[m.queryIdx].pt[0]+x,
kp1[m.queryIdx].pt[1]+y])

    frame = [int(min_w), int(min_h), int(max_w), int(max_h)]
    pts1 = np.asarray(pts1)
    pts2 = np.asarray(pts2)
    return pts1, pts2, frame, good
```

```
def get_pose(pts1, pts2, cam_mat):
    E, mask = cv2.findEssentialMat(pts1, pts2, cam_mat)
    R = cv2.recoverPose(E, pts1, pts2, cam_mat)
    return R

if __name__ == "__main__":

    # Object Definations
    url = 'https://192.168.93.187:8080/video'
    vid = cv2.VideoCapture(url)
    orb = cv2.ORB_create(nfeatures=10000000)
    bf = cv2.BFMatcher()

    # Camera params
    cam_mat = np.asarray([[1, 0, 0], [0, 1, 0], [0, 0, 1]])

    kp1, des1, last_frame,update_template, x, y = get_object_feat()

    # initial rotation matrix will be identity
    R_old = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]

    while (1):
        if cv2.waitKey(1) & 0xFF == ord('q'):
            vid.release()
            cv2.destroyAllWindows()
            break

        ret, frame = vid.read()

        # cv2.imshow("raw", frame)
        # cv2.waitKey(1)

        kp2, des2 = orb.detectAndCompute(frame, None)

        pts1, pts2, temp_co, good = matcher(kp1, des1, kp2, des2, x, y)
        # img3 = cv2.drawMatchesKnn(update_template, kp1, frame, kp2, 
good, None, flags=2)
        # cv2.waitKey(1)
        copy = frame
        cv2.rectangle(copy, (temp_co[0], temp_co[1]), (temp_co[2], 
temp_co[3]), (0, 0, 225), 2)
        # cv2.imshow("good", img3)
        #cv2.imshow("copy", copy)
        cv2.waitKey(1)
        print(temp_co)

        pose = get_pose(pts1, pts2, cam_mat)
        R_new = np.dot(R_old, pose[1])


        #print(R_new)

       # updating bounding box
```

```
        y_add = int((temp_co[3]+temp_co[1])/8)
        x_add = int((temp_co[2]+temp_co[0])/8)
        update_template = frame[temp_co[3]-y_add:temp_co[1]+y_add,
temp_co[2]-x_add:temp_co[0]+x_add]
        # cv2.imshow('up', update_template)
        # cv2.waitKey(1)
        # print("update_template")
        # cv2.waitKey(1)

        kp1, des1 = orb.detectAndCompute(frame, None)
        x = 0#temp_co[2]-x_add
        y = 0#temp_co[3]-y_add
        # print(x, y)

        # get angles from rotational matrix
        R_temp = R.from_matrix(R_new)
        Euler_angles = R_temp.as_euler('zyx', degrees=True)
        print("angles")
        print(Euler_angles[2])
        cv2.putText(copy, str(Euler_angles), (150, 50),
cv2.FONT_HERSHEY_COMPLEX_SMALL, 1, (0, 0, 255), 2, cv2.LINE_AA)
        cv2.imshow("angle", copy)

        # Reassignment
        last_frame = frame
        # kp1 = kp2
        # des1 = des2
        R_old = R_new
```