FIFACoachBot

6.506 Final Report
May 14th, 2024
Ian Gatlin

Github Repo: https://github.com/ijga/FIFACoachBot
Parallel Demo Video: https://github.com/ijga/FIFACoachBot/blob/main/demo.mov
Slide Deck: https://docs.google.com/presentation/d/1hSuw2VB26ciQFEkj68bMX2M8g4vr7pfSs8QfzQeNUI0/edit?usp=sharing

Abstract

In this paper I discuss methods on how to consistently create high quality graph representations of a FIFA 23 match from streaming the screen into a program. After quickly deciding to use Yolov8 to create a bounding box object detection model, I needed to find a way to tune the training and prediction of the model to consistently identify all game objects on the screen. Consistently finding the ball was a challenge because of its size and importance to the game state. It needed to be found consistently, but it was the hardest object to find consistently. By primarily using tiling in both the training and prediction of the model, I created a pipeline that consistently found the ball and created a very representative game state. However, even after much testing and iteration, the performance of the pipeline is not what was desired. In the object prediction phase, I experimented with parallel tiling, serial tiling, and eliminating downscaling. Although the most trivial, eliminating downscaling achieved the best performance by taking 1.4s to run object detection on a frame. Reducing the magnitude of downscaling achieved better performance but caused spotty ball detection, which was unacceptable for our use. Serial tiling and parallel tiling took 1.8s and 1.9s in the best case scenarios. Introducing parallelism did not improve the runtime most likely due to lower level library implementations already taking advantage of parallelism. There are also likely effects from process management overhead, and interprocess communication.

1. Introduction

Esports is a growing industry and many aspiring pros spend hours practicing to make it big. Like any other sport, coaching is crucial to the development of an esport player's skills and intuition. However, esports coaching is not accessible and gamers are not willing to pay hefty coaching fees. Therefore, esports coaches are reserved for the already elite. The FIFA franchise is one of the most played video games in the world with a bustling esports scene. An AI coaching program has the chance to be adopted by many players.

The first step in this process is creating a representation of the game in a way the computer can better understand. Since we do not have access to the internal memory game state, we have to use computer vision to build this representation. With this, there is potential to create a FIFA coach using graph neural nets to identify game states that are crucial to the outcome of the game using classification. If a neural net can help a player learn what an non-obvious important game state looks like, they can become a better FIFA player.

An interesting application of this idea would be a real time coach. While you are playing FIFA, you could stream your game into your computer using a capture card. A program could quickly parse the video into a representation, run it through a coaching model, and recommend an action for you to take. For this to be useful, all of these steps need to happen in around a tenth to three tenths of a second, as the game very quickly changes.

2. Related Work

Although there has been no work based on the training of a FIFA coach neural net, there has been work regarding computer vision for soccer analysis and computer vision for small objects.

There is a YouTube video by Machine Learning with Hamza where he describes a pipeline to create a 2d representation of the real soccer matches [1]. His framework is much more general and does not go beyond the 2d representation. He does not make a graph. However, his process is much faster than mine and has acceptable ball detection. I believe he runs his code on better hardware and also he has a better trained model. He does not talk about if he manually tunes downscaling in his video, and this will be interesting to explore.

When I was exploring methods to find small objects, I found the SAHI framework that improves small image detection [2]. The main idea is that you run the model over tiles of the image, and then combine the results back on the main image. I found an implementation that I could easily customize to do this in parallel in Nicolai Nielson's GitHub [3]. He is a popular AI YouTuber. This approach serially goes over each tile and runs a computer vision model. I wanted to explore doing the processing of all tiles in parallel. I show how I did this in section 4.

3. Background

To begin my image processing pipeline, I used a Yolov8 model [4] that I trained with 100 images that I collected in 1080p from games I played as Manchester City against Manchester United. You can then pass images into the model object in python, and it will return an object containing bounding boxes with confidence levels and class ids for each of them.The final implementation was trivial, but it required tuning the training of the model. After experimenting with 9 different tuning parameters, I settled on a model that was trained on images from a 2 row, 3 column tiling scheme. It would be more useful to have around the order of 1500 instances of the ball to have enough examples, but the amount of epochs and w additional augmentations I used on the training set had the model perform well.

I then could use the model to find objects on new FIFA footage before they are processed more. This is the step I will discuss more in section 4.

Before we project the bounding boxes, we need to remove any bounding boxes that are around the same player. This happens more often when you tile the prediction (which is the approach taken in section 4). I removed extra bounding boxes in linear time with respect to the number of bounding boxes found in a frame. I round each of the corners of the box to the nearest 25th pixel, and then store that in a python set with the bounding box's area. If one of a box's rounded corner values is already in the set, I compare the bounding box's area with the one stored in the set. I take the bounding box that is larger as the most accurate one, and discard the other one.

Once the best bounding boxes are found, they need to be projected into an accurate plane that removes the warp of the low camera angle. We want an overhead view of the field. Applying homography matrices to these points will give us this result. I used the cv2 library following Francis Camarao's example in this article [5]. Using points from the raw image and specifying where I want them to move to in relation to each other gives us a homography matrix that I can save and use in my processing pipeline. I also did work to dynamically find homography matrices based on the camera angle. This is outlined more in my midterm report. To do this I used image segmentation and hough lines.

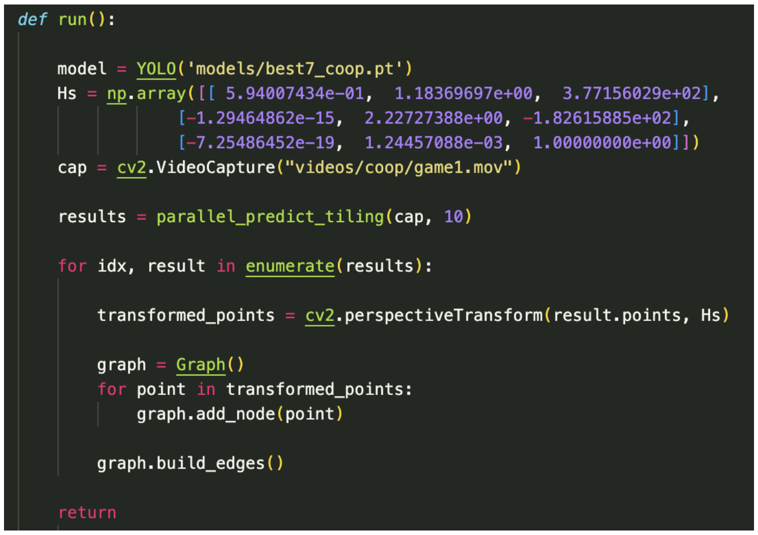
To transform this 2d representation into a graph that can potentially be used in graph neural networks to draw insight from the game state, I created a custom node, edge, and graph class. Nodes are categorized by what object they are, and they are given a certain set of edges based on its characteristics, other node's characteristics, and the distance between nodes. Methods for creating these graphs can be found in the "graph.py" in the project GitHub repository.

4. Contributions

When I ran the serial tiling implementation, it found the ball with great accuracy. It took 250ms to process each tile, and around 2 seconds to finish processing the entire image. This makes sense, there are 8 tiles. I thought that if we ran each tile in parallel, we could finish the entire processing step in around 400ms to account for the span of 250ms and additional overhead. I was expecting a 4-8x speedup. With this in mind, I implemented a parallel multiprocessing algorithm.

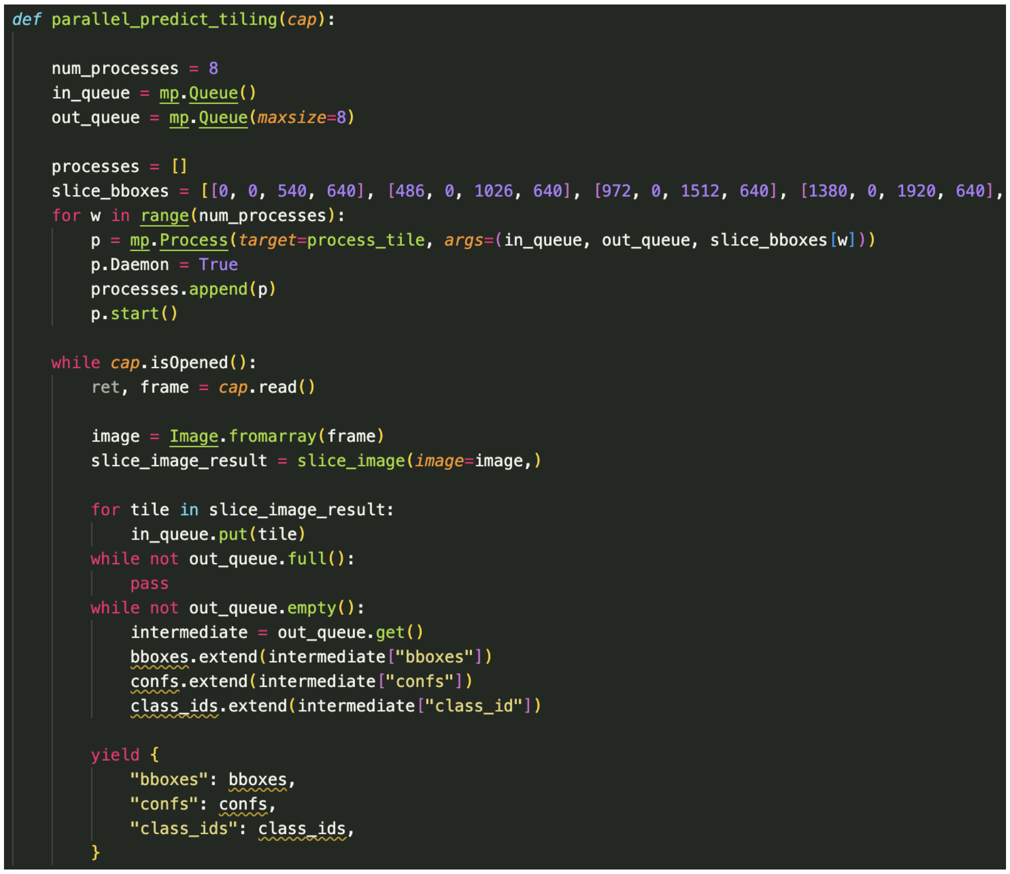
Starting on the next page, I abstract and walk through some code that contains only lines that are essential to understanding the meaning of the multiprocessing approach I took.

The parallel implementation example shown in the slides took around 2.3 seconds, and it took around 2.1 seconds to finish processing each tile. After exploring more into the CPU load, this result makes sense. This is because the serial implementation actually uses all of the CPU cores. The Yolov8 library most likely parallelizes the processing of each tile, which is why each tile takes around 250ms. The additional time that the parallel implementation incurs is from



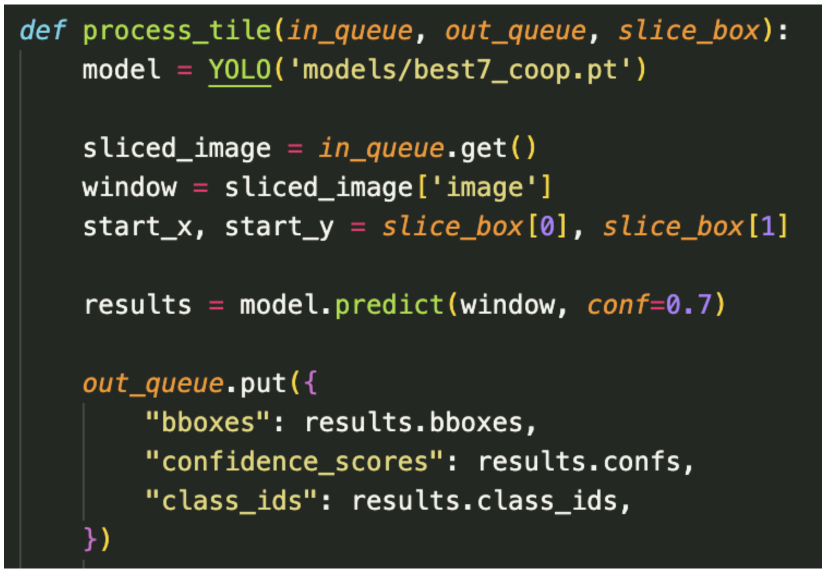
```python
def run():

    model = YOLO('models/best7_coop.pt')
    Hs = np.array([[ 5.94007434e-01,  1.18369697e+00,  3.77156029e+02],
                   [-1.29464862e-15,  2.22727388e+00, -1.82615885e+02],
                   [-7.25486452e-19,  1.24457088e-03,  1.00000000e+00]])
    cap = cv2.VideoCapture("videos/coop/game1.mov")

    results = parallel_predict_tiling(cap, 10)

    for idx, result in enumerate(results):

        transformed_points = cv2.perspectiveTransform(result.points, Hs)

        graph = Graph()
        for point in transformed_points:
            graph.add_node(point)

        graph.build_edges()

    return
```

Figure 1. Parent program that calls for the generator parallel_predict_tiling to provide bounding boxes to be transformed and made into a graph.



```python
def parallel_predict_tiling(cap):

    num_processes = 8
    in_queue = mp.Queue()
    out_queue = mp.Queue(maxsize=8)

    processes = []
    slice_bboxes = [[0, 0, 540, 640], [486, 0, 1026, 640], [972, 0, 1512, 640], [1380, 0, 1920, 640],
    for w in range(num_processes):
        p = mp.Process(target=process_tile, args=(in_queue, out_queue, slice_bboxes[w]))
        p.Daemon = True
        processes.append(p)
        p.start()

    while cap.isOpened():
        ret, frame = cap.read()

        image = Image.fromarray(frame)
        slice_image_result = slice_image(image=image,)

        for tile in slice_image_result:
            in_queue.put(tile)
        while not out_queue.full():
            pass
        while not out_queue.empty():
            intermediate = out_queue.get()
            bboxes.extend(intermediate["bboxes"])
            confs.extend(intermediate["confs"])
            class_ids.extend(intermediate["class_id"])

        yield {
            "bboxes": bboxes,
            "confs": confs,
            "class_ids": class_ids,
        }
```

Figure 2. Generator creates multiple processes and feeds them tiles. Once all of the tiles are completed they combine the data from each process and returns it.



```python
def process_tile(in_queue, out_queue, slice_box):
    model = YOLO('models/best7_coop.pt')

    sliced_image = in_queue.get()
    window = sliced_image['image']
    start_x, start_y = slice_box[0], slice_box[1]

    results = model.predict(window, conf=0.7)

    out_queue.put({
        "bboxes": results.bboxes,
        "confidence_scores": results.confs,
        "class_ids": results.class_ids,
    })
```

Figure 3. Each process graphs a tile, and runs the model on it. It then puts the bounding box data onto the output queue for the parent to retrieve.

process management overhead and interprocess communication. We can see the CPU is used more by the system in the parallel implementation compared to the serial implementation (Slide 5). In terms of interprocess communication, each tile of the image takes 1MB of space and is sent from the parent to the child processes through a queue. What could be more efficient is sharing the image in shared memory across all processes. Then each process would just index into their tile. This would reduce the amount of interprocess communication, since there would just be reads to this shared memory. However, there isn't much documentation about python's multiprocessing shared memory features, and people have found many bugs in that part of the library.

However, after doing this work, I discovered that manually setting the model to not downsize when running it over the entire image also gives us acceptable ball detection rates. This takes around 1.3 seconds per frame in the best case scenario. It makes sense that this would be faster than our tiling implementation since the tiling process takes up time, and the tiles overlap to find images that are across tiles. Since we are assuming that this is parallelized due to the underlying library code, running this approach on a machine with more cores can probably achieve better performance.

Additionally, it would be good to train a model on more examples of a ball. A better performing model could tolerate more aggressive downscaling when using it to find objects on new footage. Downscaling the image makes the performance of the detection operation much better.

5. Open Problems & Further Work

There is a lot of future work that can be built off of this project. From the work that I have done, I conclude that it is very difficult to create a graph representation of the game state in a quick enough time to get instant coaching feedback from a coaching model on personal laptops. However, creating a coach for end of game insights can easily be created with this performance. Improving performance is useful in this case as it can greatly reduce processing time needed to create insights. It is preferable to have this insight ready at the end of the game, so the game’s events are fresh in the player's head. This will maximize the coaching effect on the player’s skill and intuition development. The more frames the coaching pipeline can analyze, the better coaching the application can give. Most of the frames that will be put into the pipeline will not have any coaching insight to provide, therefore, we want to be able to process as many frames as possible.

As a starting point, it would be interesting to explore object detection outside of Yolov8. Yolov8 is very easy to work on, and performant, but may be optimized to work outside of what is needed in this project. There are a variety of different models such as Mask R-CNN and vision transformers that may work faster for the problem of finding all of the FIFA game objects. It is also important to create more labeled images to train the model on.

Now that I can create graph representations well, I want to work on the next part of the coaching pipeline. I need to either label each graph manually with whatever event is happening in the frame, or I need to train a separate model that can recognize events in FIFA games. I can also leverage the work I did to train the model to identify user inputs into the game controller to help aid in this work. Once I train this model, I can encode next actions that users should do in a certain state based on outcomes. The work I did this semester sets up a solid foundation for further work towards this FIFA coaching application.

References

[1] "Computer Vision for Football Analysis in Python with Yolov8 & OpenCV." *YouTube*, Machine Learning With Hamza, 11 Dec. 2023, www.youtube.com/watch?v=yJWAtr3kvPU.

[2] Akyon, Fatih Cagatay, et al. "Slicing Aided Hyper Inference and Fine-tuning for Small Object Detection." *2022 IEEE International Conference on Image Processing (ICIP)*, 16 Oct. 2022, pp. 966–970, https://doi.org/10.1109/icip46576.2022.9897990.

[3] Nielsen, Nico. "tiling.py" *GitHub*, 4 May 2024, https://github.com/niconielsen32/tiling-window-detection/commits/main/tiling.py. Accessed 14 May 2024.

[4] "Object Detection with Pre-Trained Ultralytics Yolov8 Model | Episode 1." *YouTube*, Ultralytics, 12 July 2023, www.youtube.com/watch?v=5ku7npMrW40&list=PL1FZnkj4ad1PFJTjW4mWpHZhzgJinkNV0&index=9.

[5] Camarao, Francis. "Image Processing Using Python-Homography Matrix." *Medium*, Medium, 17 June 2023, medium.com/@flcamarao/image-processing-using-python-homography-matrix-a5da44f3a57b.