CAP5415: Computer Vision

Course Project: League CS Helper

By: Shane Davis

12/5/2021

Abstract

For my project, I have created a program that will aid League of Legends players in the task of CS-ing. This is done by using the YOLOv5 model trained on my own artificially created dataset to localize and identify minions, which I then use to find their health bars and calculate the health to determine if they are below a threshold [1]. If the minion's health is below the threshold, a bounding box is displayed on screen highlighting the minion to the player, indicating they should attack the minion. In practice, the model performed well however the program as a whole wasn't responsive enough to be useful as there are delays between the model processing the image to the final step of calculating the thresholds and displaying to the screen.

Introduction

League of Legends is a 5 versus 5 MOBA (Multiplayer Online Battle Arena) video game where the main objective is to take down the enemy team's base [2]. There are various characteristics of the game, and my project is focused on improving the task of CS-ing, which is one of the ways to generate gold for your player. CS-ing is when you are killing the minions that are at low enough health that they will die in one shot. When a player kills a minion, they receive gold which they can buy items with to improve their character's abilities. There are three different types of minions:

Melee	**Caster**	**Cannon**

Table 1: Types of Minions

It is important to identify the type of minion as they have varying levels of health, however they all display the same size health bar. To determine when the minion will die in one shot, the program needs to know what type the minion is while calculating their health since it counts the number of red pixels in the health bar and compares it to a stored threshold for that minion type.

Method

Developing Model

To train the model, I didn't have a pre-existing dataset to work from however there has been a similar project that I was able to take inspiration from on how to generate the dataset. This project is LeagueAI, the idea from the project was to develop and train an AI to play League of Legends autonomously [3]. Luckily the author of the project included a report on how he generated the dataset to train his model to recognize not only minions but other objects in the game. Therefore, I followed his steps along with using his code (although I had to modify for my uses), and these are the following steps:

Generating Source Images for Minions

By using a model viewer such as https://teemo.gg/model-viewer I was able to record multiple videos of the minions in their variation animations and from various perspectives [4]. From those source videos, I used the pyFrameexporter.py program to export every 3 frames into an image, creating hundreds of source images of the minions.

Example (cropped):

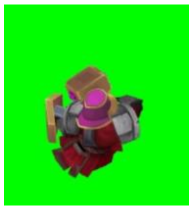

Figure 1: Melee Minion with Green Screen

I then used pyExportTransparentPNG.py to take all the individual frames and remove the green screen to make the images transparent.

Output from the previous example:

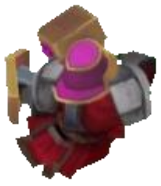
Figure 2: Melee Minion after Post-processing

I do this for the three minion types and generated over 300 source images for each of the types.

Creating the Background

Now that I have various images of minions, I need to have a background to put them on. To make these backgrounds, I took various screenshots around the map where minions are generally located.

Example:

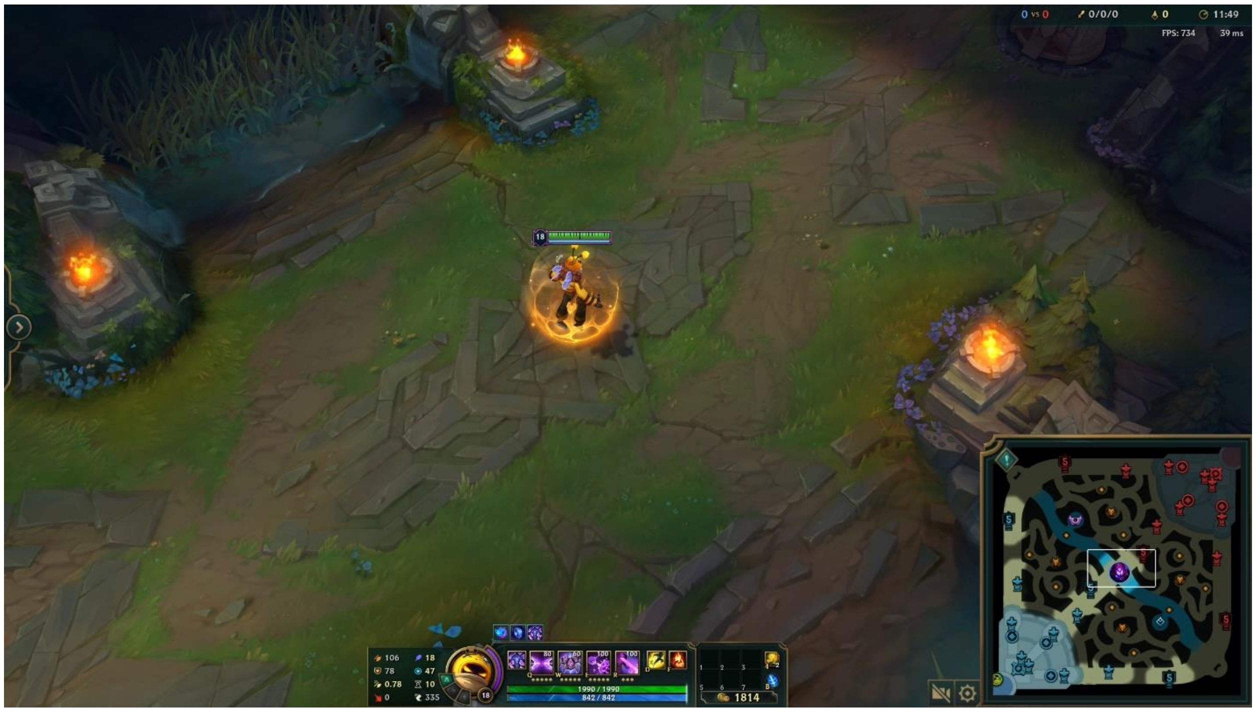
Figure 3: Map screenshot

I chose to include various UI elements such as the mini-map and player information since that is more realistic to what the model is going to be processing while it is running.

Generating the Dataset

Now that I have both the source images from the minions and various map screenshots, I used the bootstrap.py to generate the dataset. Essentially, the program will pick some random map screenshot along with a random number of minions to place on the screen and will place them in various locations along with various orientations, which I used to generate 10,000 images.

Example:

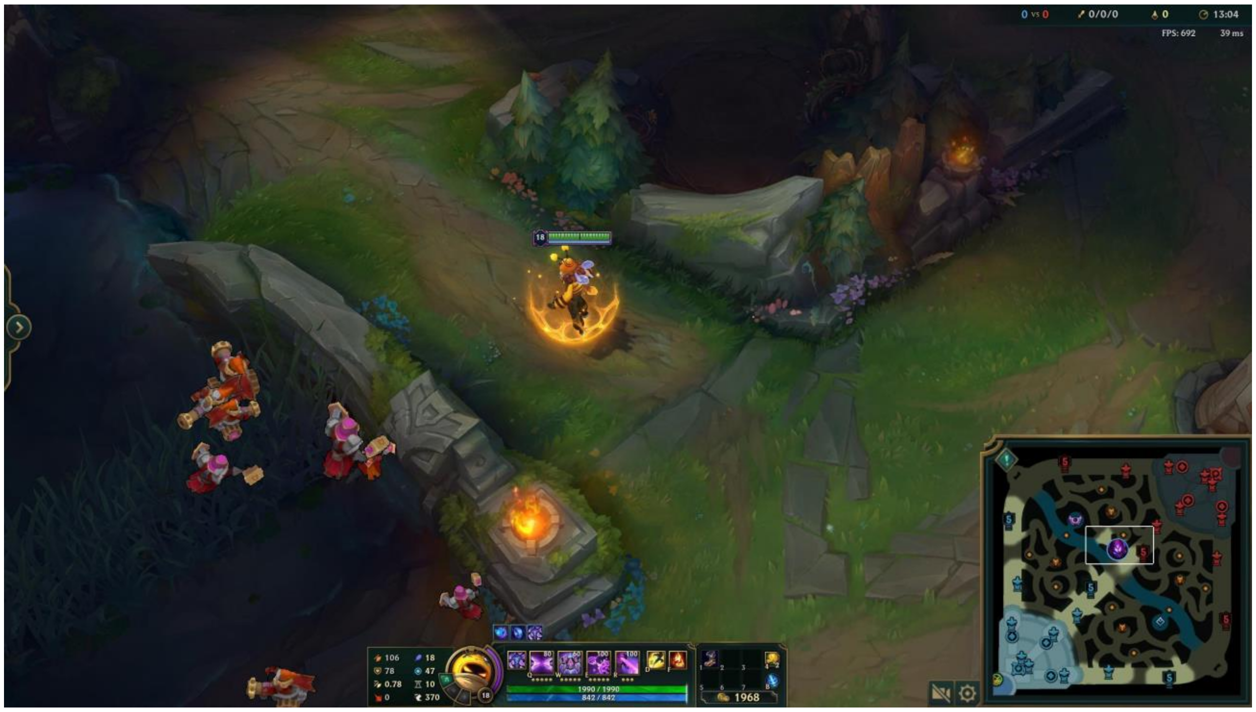

Figure 4: Dataset example

The bootstrap.py will also save the labels in the form that YOLOv5 expects. After all the images have been generated, I upload them to https://roboflow.com/ to process the images and separate them out into training, testing, and validation sets [5]. During processing, it allows me to remove errors such as minions being placed off screen or invalid sizes and I can also resize the images to 1280x720.

Training YOLOv5

Now that I have the dataset on Roboflow, I followed the tutorial from YOLOv5's GitHub on how to train the model using a custom dataset [6]. In the tutorial, they give a Google Colab notebook which has all the steps you need already coded for you and you just need to follow it, however since my computer has a faster GPU than Google Colab, I decided to run the notebook locally which made training the model much faster. I initially used YOLOv5s as my starting model with an image size of 640x360 (1/4 of my native resolution), however in the end I used YOLOv5x6 with an image size 1280x720 (1/2 of my native resolution) since it had higher performance with a cost of slightly more processing time and larger *.pt file. After following the steps in the notebook, I had a *.pt file that I could use in my program to load my custom model and process actual game images through it.

Developing the Program

To develop this program, I broke the program up into four main components: initialization, scanning the screen, processing the screen with the model, and calculating the resulting data to determine what information to display to the player.

Initialization

To begin, I load all the values from the config.ini into various data structures to be used throughout the rest of the program. I then load the overlay, using code from PoopStuckToYourMouse (yes that is the name of the GitHub) which had the base code to keep a PyGame window always on top of the screen along with making it click through [7] [8]. Finally, I load the custom model into memory, setting it to the evaluation mode so that nothing changes.

Scanning the Screen

For scanning the screen, I use PIL's ImageGrab() function to take a screenshot and store it as a PIL Image object [9]. This is important since I found other screenshot libraries had issues with the overlay always being on top of the screen.

Processing the Screen

During this phase, I take the screenshot and process it through my trained model. This will return a list of found objects and their associated labels, which I then use the following checks to remove any false positives:

- If the confidence level is below 60%, ignore the given object
- If the object's size is below 50x50, ignore the given object
- If the object is within any UI elements, ignore the given object

At this point, I can display all the bounding boxes of the detected minions as a debugging step using the --debug_display parameter.

Calculating Data and Displaying Results

Now that I know the minion's locations and type, I need to search for the minion's health bar and check if it is below the threshold. The health bar will always be above the minion, however since my model doesn't perfectly localize the minion every time, I need to be generous with my searching algorithm for finding the health bar. The search algorithm works by creating a padded region (which can be changed in the config) to search for potential health bars, and I choose the one that is closest to the minion. This is because in game the minions tend to clump together, therefore making their health bars clump together. Once I have the associated health bar position for the minion, I count the number of pixels that are the same color as the health bar to get an estimate of the minion's health. I then check if it is below the defined threshold for the given minion type, and if it is then I display it to the player, indicating that the minion should be attacked.

Results, Discussion, and Analysis

For the actual artificial dataset, the model performed well. Here are the graphs as I trained the model, which I trained for 10 epochs, from Weights & Biases [10]:

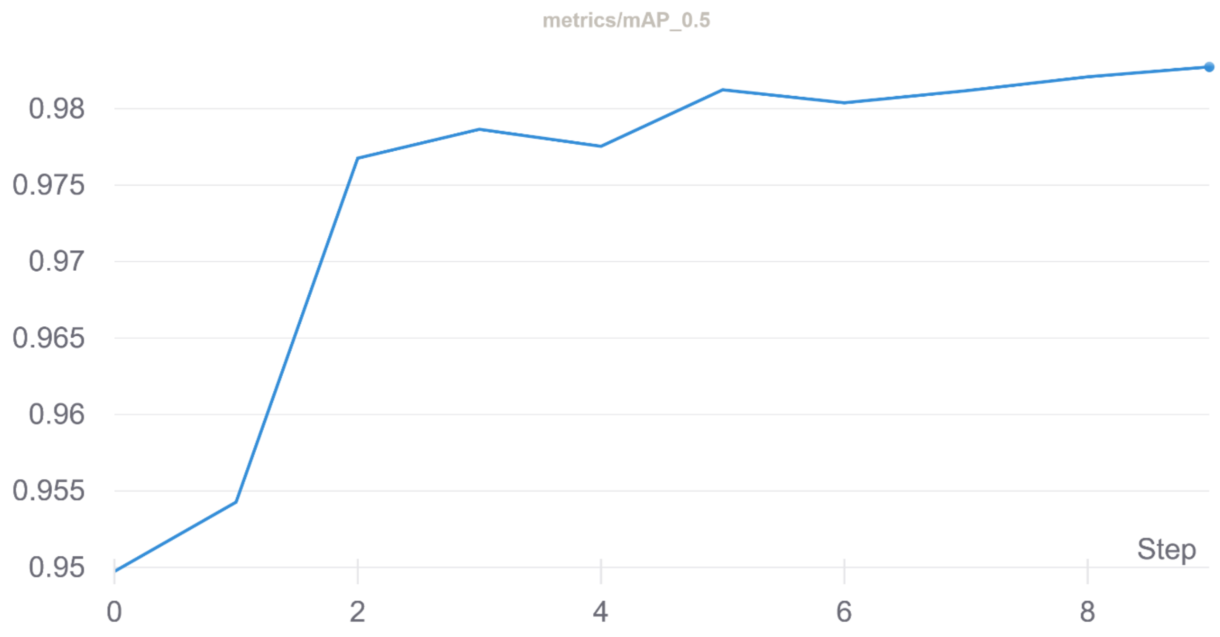

Figure 5: mAP 0.5

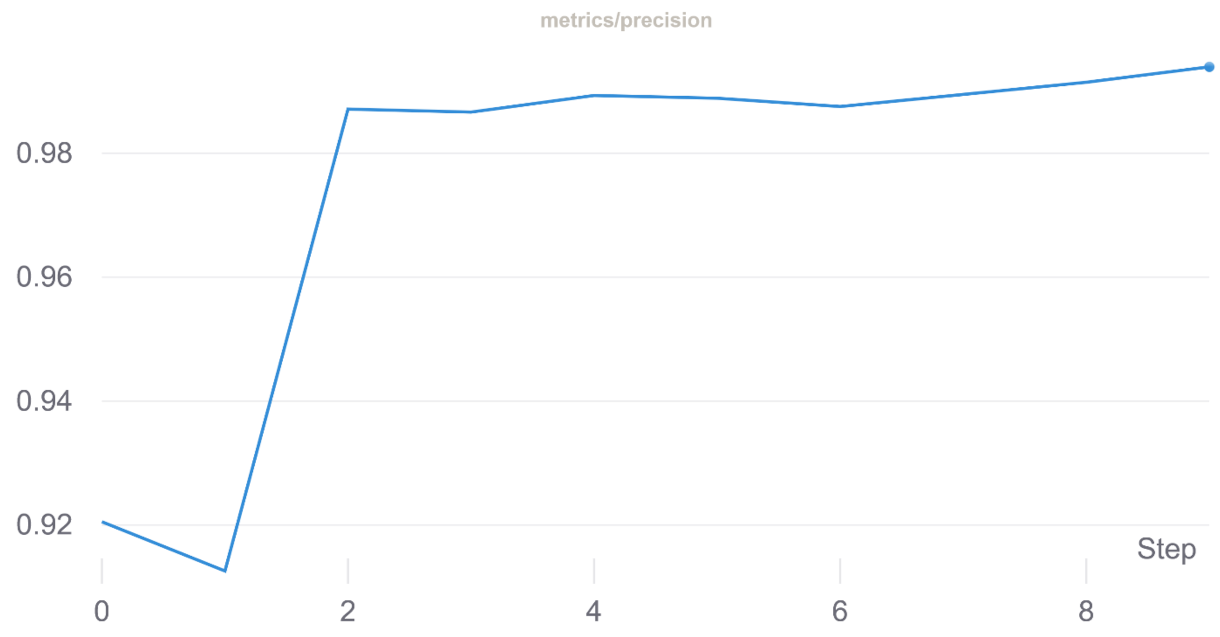

Figure 6: Precision

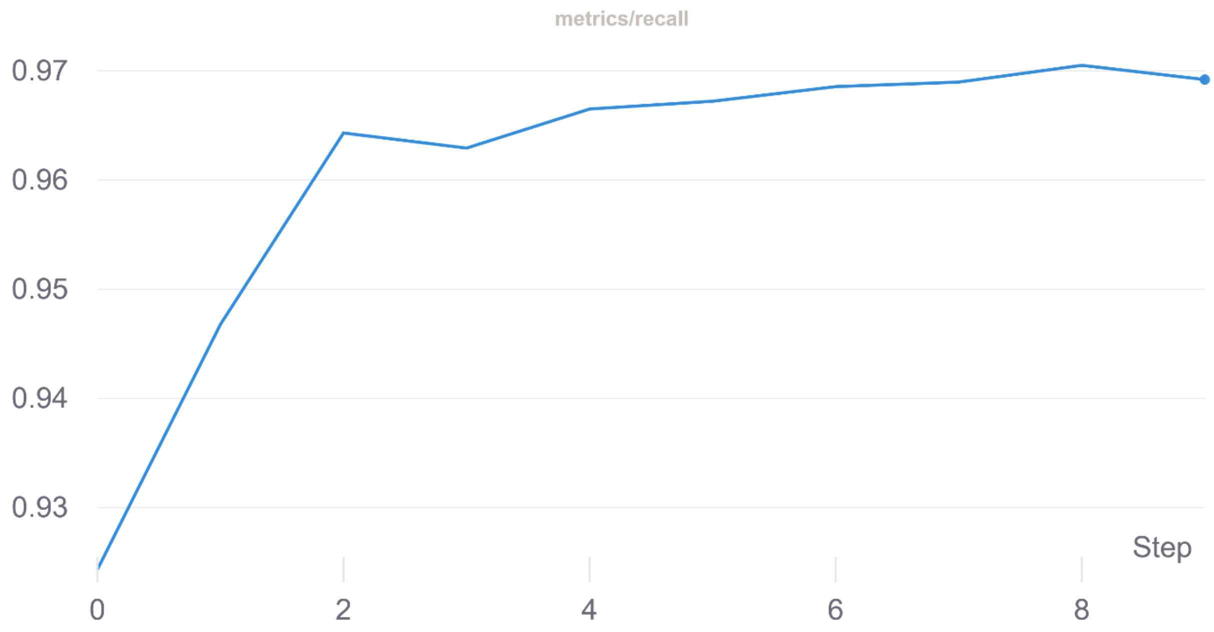

Figure 7: Recall

While these graphs show great results with the mAP .5, precision, and recall all above 95%, this is only on the artificial dataset. In the actual game, the program doesn't perform as well as you would expect from these results, however it isn't as easily quantifiable without lots of manual post processing on gameplay footage.

Therefore, here is a sample video of the program running to see the actual results of the model, with --debug_display to show all minions regardless of health (also removes threshold checks): https://youtu.be/1C-G5RKF2Ls [11]

In the video, the model performed decent, however there were some failure points:

- Detecting the blue (friendly) cannon minions
- Detecting dead minions
- Not detecting/struggling to detect minions when clumped together

This could have been improved with more training, I could've trained the model to recognize friendly minions/dead minions and have it not do anything. However, for the clumped minions I could have changed the settings in my dataset generation to try to place minions closer together to give the artificial dataset a more realistic environment.

Other than those issues, the model recognizes and finds the minions quite well, here are the average times it took to process a frame:

Screenshot took: 0.07 seconds
Model took: 0.03 seconds
Drawing took: 0.002 seconds
Total time: 0.102 seconds

From this I can estimate that the program was running at 10 FPS **without the threshold checks**.

Now, here is another sample, but with threshold checks turned on (this is how the program is intended to be used): https://youtu.be/7tiFq17zWGw [12]

From this we can see that most of the minions were shown an indicator when at low health. For the minions that weren't shown an indicator, that was because they lost their health too fast, and my program did not process the images fast enough to update for that. Here are the average times it took to process a frame:

Screenshot took: 0.07 seconds
Model took: 0.03 seconds
Determining thresholds took: 0.06 seconds
Drawing took: 0.002 seconds
Total time: 0.162 seconds

From this I can estimate that the program was running at 6 FPS, indicating that the thresholds had a performance decrease to the program's run time. This would explain why some minions weren't shown an indicator, I believe that if my program was running on a faster computer or if the performance was improved then I would expect to see more minions being shown an indicator.

Conclusion

Overall, I believe my program is a good showcase of what is possible with the object detection I have trained and the algorithms I have written, however while still useable, due to lower performance of the program it is not perfect for running in real time. I have found in practice that the program did improve my gameplay, however not as much as I wanted it to due to the program not updating fast enough to show the indicators quick enough. To solve these performance problems, I would attempt to reduce the threshold calculation time, since it almost doubles the frame time. Currently, I search for the health bars manually, however I believe if I trained the model to also recognize health bars that search time can be reduced drastically, lowering the threshold calculation time. Also, I would look into other approaches for taking and storing screenshots since that added a lot of frame time, however the PIL ImageGrab was the only method I could find that wouldn't have issues with the overlay being on the screen. I would also improve on my dataset, adding detection for friendly minions and dead minions so that in my program I can choose to do nothing with those since I was running into issues with the model detecting those as enemy minions. Other than that, I believe my program is a good tool to improve League of Legends players at CS-ing.

References

[1] Ultralytics, "YOLOv5," GitHub, 12 October 2021. [Online]. Available: https://github.com/ultralytics/yolov5. [Accessed 4 December 2021].

[2] League of Legends, [Online]. Available: https://www.leagueoflegends.com/en-us/. [Accessed 5 December 2021].

[3] Oliver, "LeagueAI," GitHub, 6 November 2019. [Online]. Available: https://github.com/Oleffa/LeagueAI. [Accessed 4 December 2021].

[4] A. &. Kervin, "3D Model Viewer," Teemo.GG, [Online]. Available: https://teemo.gg/model-viewer. [Accessed 5 December 2021].

[5] roboflow, [Online]. Available: https://roboflow.com/. [Accessed 5 December 2021].

[6] G. Jocher, "YOLOv5 Custom Training," GitHub, 30 September 2021. [Online]. Available: https://github.com/ultralytics/yolov5/wiki/Train-Custom-Data. [Accessed 4 December 2021].

[7] LtqxWYEG, "Poop Stuck To Your Mouse," GitHub, 1 October 2021. [Online]. Available: https://github.com/LtqxWYEG/PoopStuckToYourMouse. [Accessed 5 December 2021].

[8] P. Shinners, "PyGame," 2011. [Online]. Available: https://www.pygame.org. [Accessed 5 December 2021].

[9] A. Clark, "Pillow (PIL Fork) Documentation," readthedocs, 2015. [Online]. Available: https://buildmedia.readthedocs.org/media/pdf/pillow/latest/pillow.pdf. [Accessed 5 December 2021].

[10] L. Biewald, "Experiment Tracking with Weights and Biases," Weights & Biases, [Online]. Available: http://wandb.com/. [Accessed 5 December 2021].

[11] S. Davis, "LeaugeCSHelper with Debug Display shown," YouTube, 5 December 2021. [Online]. Available: https://youtu.be/1C-G5RKF2Ls. [Accessed 5 December 2021].

[12] S. Davis, "LeagueCSHelper with Thresholds shown," YouTube, 5 December 2021. [Online]. Available: https://youtu.be/7tiFq17zWGw. [Accessed 5 December 2021].

[13] A. Sayed, "Checking whether two rectangles overlap in python using two bottom left corners and top right corners," stackoverflow, 5 October 2021. [Online]. Available: https://stackoverflow.com/questions/40795709/checking-whether-two-rectangles-overlap-in-python-using-two-bottom-left-corners. [Accessed 5 December 2021].