Machine Learning ND

Capstone Project

Labeling Standardized Videogame Objects Using Computer Vision and Clustering

Isaac Chan

April 4th, 2018

A. Definition

Project Overview

The goal of this project is to develop an algorithm that can recognize video game objects without exorbitant amounts of training data. 20+ class classifiers trained on large datasets have high performance and accuracy in the real world, but generalize poorly to nuanced videogame environments with unique objects.

Part of training an agent is describing its environment. However, because many companies are protective of their API, many do not provide "raw" versions of the game where objects are already labeled, allowing for fast, and effective training using multiple instances.

To avoid labeling tens of thousands of images to train a separate classifier for each game, certain tricks can be used to exploit the uniformity and predictability of the objects within each environment, unlike cars, which have thousands of different models, colors, viewing angles and environments. The only data that will be used are the RGB and HSV values of the image along with detected SIFT features.

Problem Statement

Computers have no difficulty defeating humans in board games and "old-school" videogames. However, both scenarios often have very limited inputs (up, down, left, move, stay) and states (GO, e4, e5, brick), and can be solved using brute force alone.

Winning in modern videogames can be a matter of milliseconds and have as many inputs as there are pixels on the screen. For this reason, in only one simplified version of a modern videogame has AI bested humans. The first step is to recognize the objects within the environment at an interactable rate.

Metrics

The objects I am trying to detect and label are called 'minions'. They are non-player-characters that players kill to earn gold and experience to become stronger. There are 3 unique types of minions in 2 color variations.

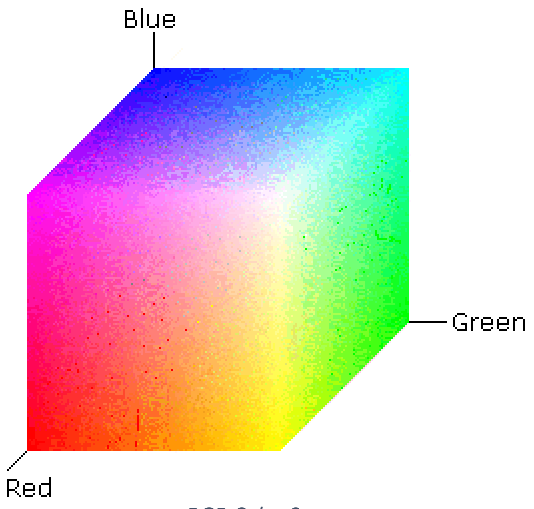
The two-main metrics I judged my algorithm on was accuracy and speed.

- Accuracy is simple. If there is a minion on the screen, detect it. For each minion, accurately label its color. For the scope of this project, I did not determine which type of minion it is, only it's color.
- Speed is measured in how many frames per second can be captured while simultaneously performing the necessary calculations per frame.

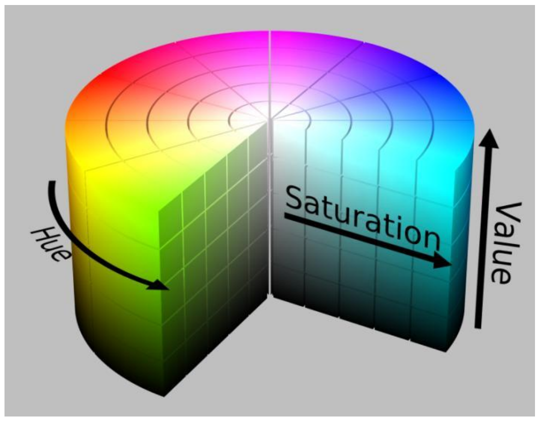
B. Analysis

Data Exploration

The input data used for detection and classification will be each screen capture's RGB pixel values, HSV pixel values, and its calculated SIFT features. Each screen capture I use will have the dimension of 1920 by 1080. Meaning there are 2073600 pixels on the screen that all have an HSV value, and red, green, blue intensity values from a scale of 0-255. RGB values are represented as an array, ranging from [0,0,0] (black pixel. no red, green or blue values), and [255,255,255] (white), along with everything in between.



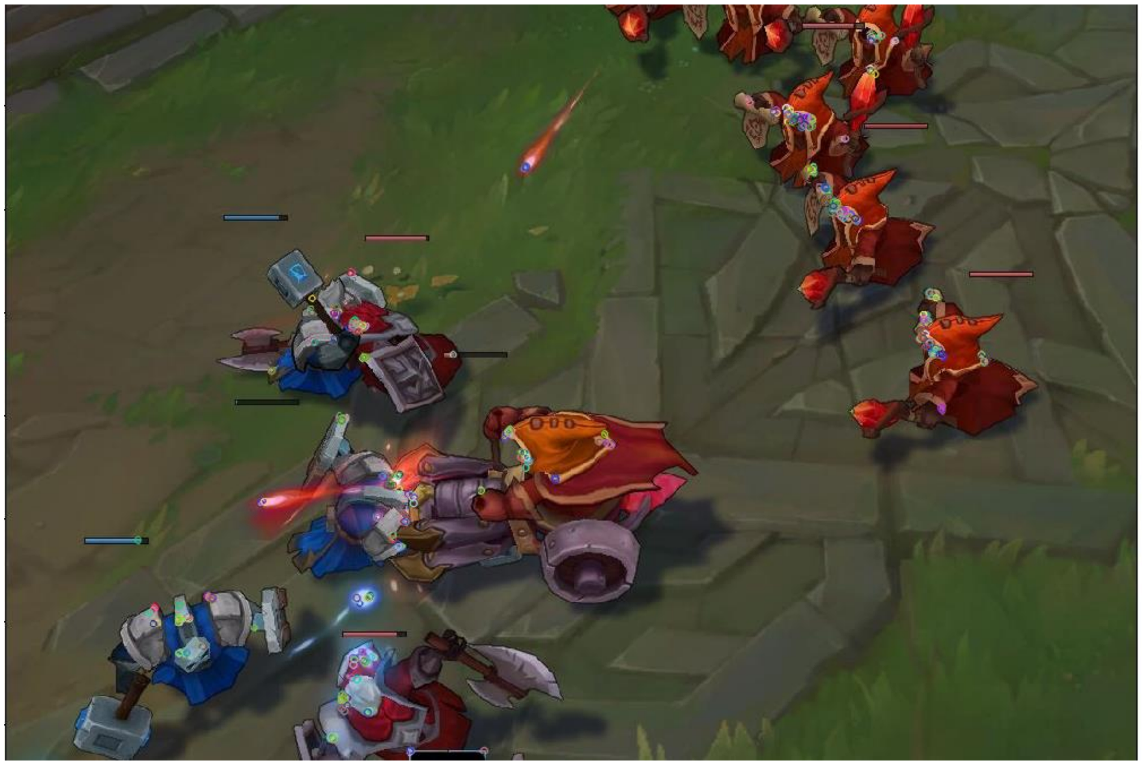
RGB Color Space



HSV Color Space

HSV values are also represented as arrays, that have different ranges but are also normalized to a range from 0-255 in opencv. H stands for hue, which basically captures the variance of an RGB array in a single number. S stands for saturation, which is the intensity of the color, and V stands for value, which is the frequency of that color.

SIFT features are locally distinct points found through smoothing an image multiple times with a larger and larger gaussian kernel. Then subtracting the intensity values between each image and the image with the next largest kernel to generate multiple difference of gaussians plot. Then repeating the entire process with differently scaled images (octaves) for robustness against scaling. The areas where there are large changes in gradient intensity between the DOG plots create locally distinct points (about 400-500 points). Note: KPs stands for Key Points.


SIFT Features Found on a Sample Screen Capture

For every frame, there are 2073600 pixels, each with 3 RGB values and 3 HSV values. Additionally, each frame must calculate ≈450 KPs, which is a length 10 array. If the program is running at 10 frames per second, there are ((2073600 * 6) + 4500) * 10 = 124461000 values returned per second.

Algorithms and Techniques

The first ML algorithm I used was a local outlier factor estimator. LOF calculates the local density deviation of a data point with respect to its neighbors. If the density is low surrounding a point, chances are it's not part of a cluster. I used LOF because undoubtedly, despite filtering out extraneous parts of the screen, some SIFT features detected will not belong to minions. For example, projectiles flying across the screen are also highly salient. Projectiles have few KP's within its pixel radius and will be removed.

The second ML algorithm I used is k-means clustering. K-means groups samples into different "clusters" based on similarity between samples. For example, given there are 500 KPs found on the screen belonging to 10 different minions, there should be 10 distinct clusters of KPs with ≈50 KPs each. The input to k-means are the pixel coordinates of each KP, and attempts to cluster them are based on proximity to each other on screen. The likelihood that a KP detected in coordinates (0,0) and a KP detected in location (1920,1080) belong to the same minion is zero, given that a minion's dimensions are roughly 100X100.

The computer visions techniques I'll be using are SIFT (described above) and HSV filtering. After a potential minion is identified by k-means, an ROI is rasterized around the cluster centroid and pixels not belonging to the minion are filtered out. By filtering based on HSV, the non-red or blue pixels and low saturation pixels are removed, leaving only the minions highly saturated red or blue pixels.

Benchmark Model

Object detection given a large enough training set is a proven accurate and generalizable method. With enough training images for each object we wish to detect, an accuracy of 99% would be easy. Even accurate classification in real time with over 15 frames per second is possible thanks to algorithms like YOLO9000. On the other side, pursuing a computer vision-based model that uses template matching to label clusters would have a close to random accuracy, but take a fraction of the time to setup and have higher frames per second thanks to its low processing demands. I will be comparing my model's accuracy against the template matching model (better than random), but comparing my models training and fps against estimated object detection model accuracy and training times.

C. Methodology

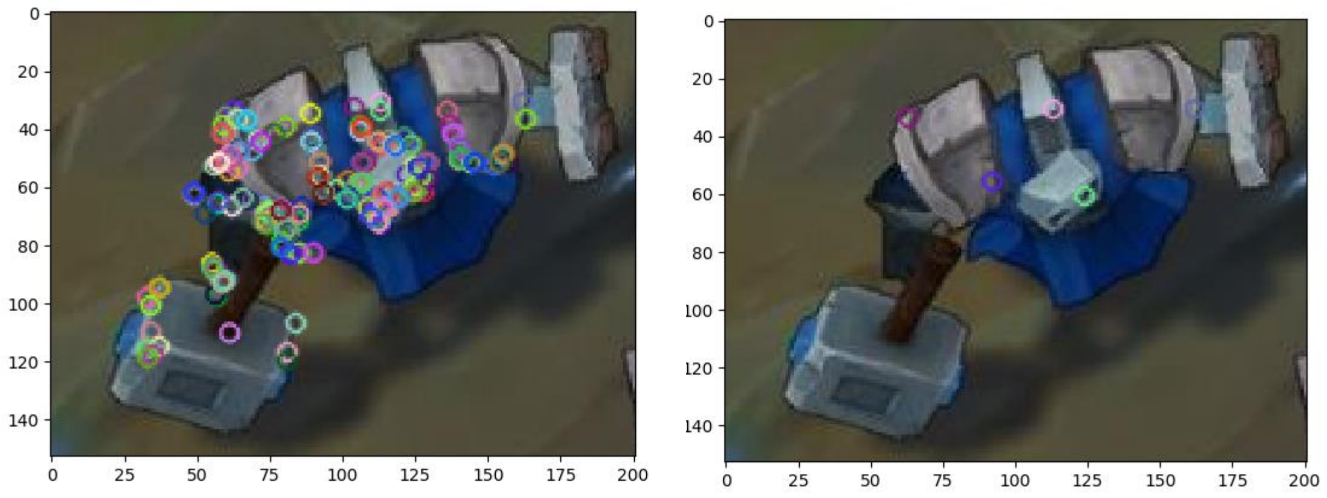
Data Preprocessing

Since I'm creating my own dataset, I can modify how I collect data to avoid preprocessing. All RGB and HSV values are stored in numpy arrays and are easily callable. The only data that's stored trickily are SIFT KPs, which are stored in opencv's unique 'KeyPoint' object. Within that object is stored the angle, class_id, octave, pt(pixel coordinates), response, and size. The only relevant attribute is pt, so I created a function that extracts the pt from each KP within the KP list and appends it to the KP_coord list.

Implementation

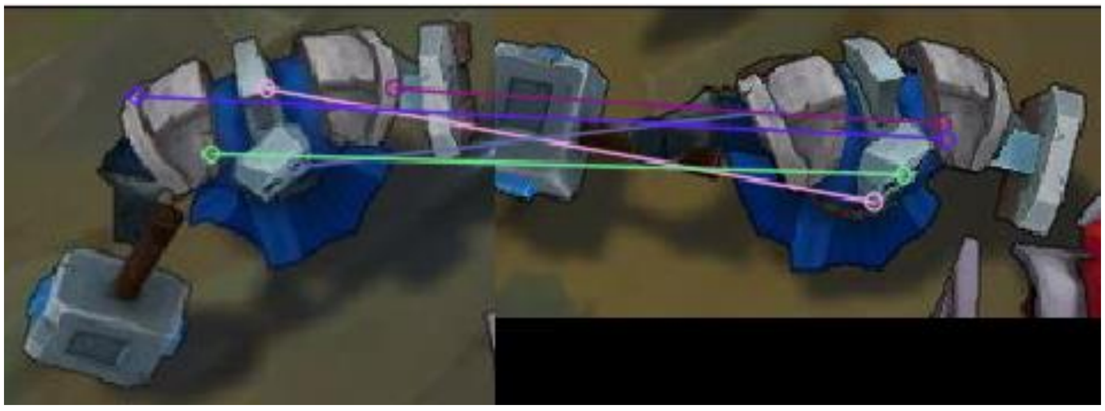
I. Young. Innocent. Naïve.

I originally tried to solve the problem of classification without large datasets using purely template matching/computer vision methods. I originally thought that the same KP descriptor values for each of the 6 minion types would exist throughout all variations in angle and stance. I then created a dictionary of template images and their extracted KP and descriptor values to compare test images to.



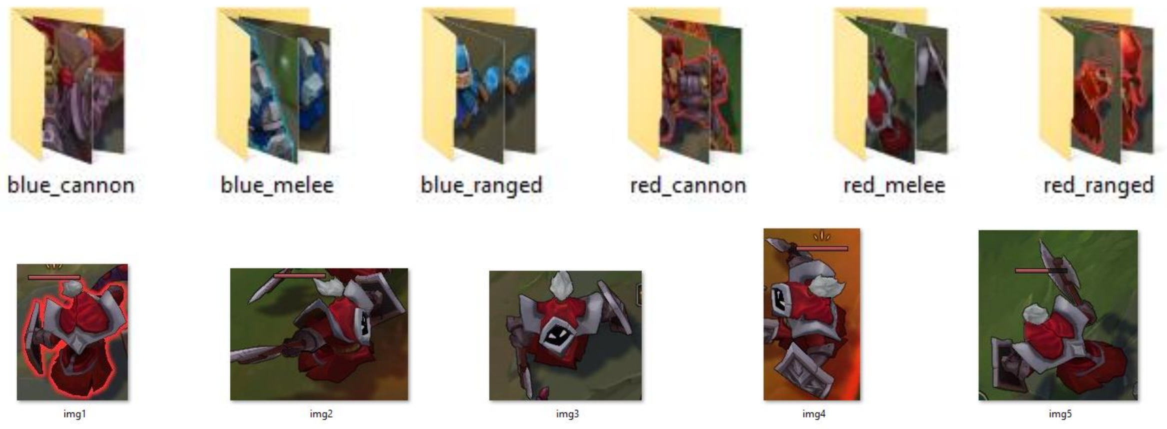
KPs on Template Image Before and After Filtering by Score and Distance

When matching the test image with the 6 template images (3 types * 2 colors), I brute-force matched only the 7 most similar KPs between them that were also a certain distance apart from each other. The template image would be labeled as the testing image it matched the best with. Unfortunately, whenever the minion angle deviated even slightly, the same or similar KPs were not found on the training image.



Matching the Template Image with a Slightly Angled Test Image

To remediate this, I expanded the template dictionary to include multiple images of each minion in varying stances and angles. Across the 30 total template images, it would find the single most similar image and then the best average set of images. But even when each of the 6 minion types had multiple images in different stances, accuracy was still worse than random.

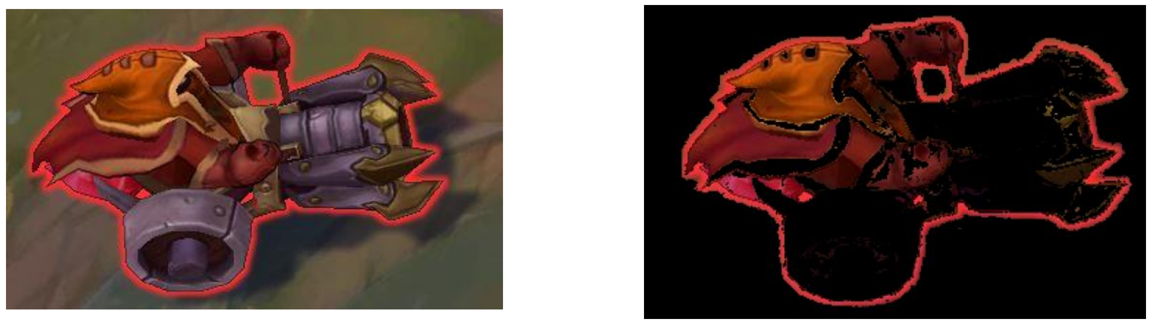


Top: Dictionary of 6 Sets of Template Images, Bottom: 5 Images of Varying Angle Within a Template Set

Despite efforts to curate a template set with images that captured as much variation in angle and action as possible, the highest scoring result returned by the brute force matcher was consistently wrong. There were far too any similarities between the KPs and descriptor values of the 6 minion types to differentiate. I tried using the KPs and descriptors in other ways, such as attempting to find KPs unique to specific types. For example: red ranged has KPs [**A**BCD] while blue melee has [BCD**E**], and blue cannon has [BCD**F**]. This proved problematic as the appearance of said specific KPs in the test images were inconsistent at best, and relying on their presence was risky.

II. Compromise

This led me to settle on merely identifying the color of the minion, rather than the type. There are several techniques that can be used to identify the dominant color of an image. The first method tried involved using k-means to divide the test image's RGB pixel values into clusters.



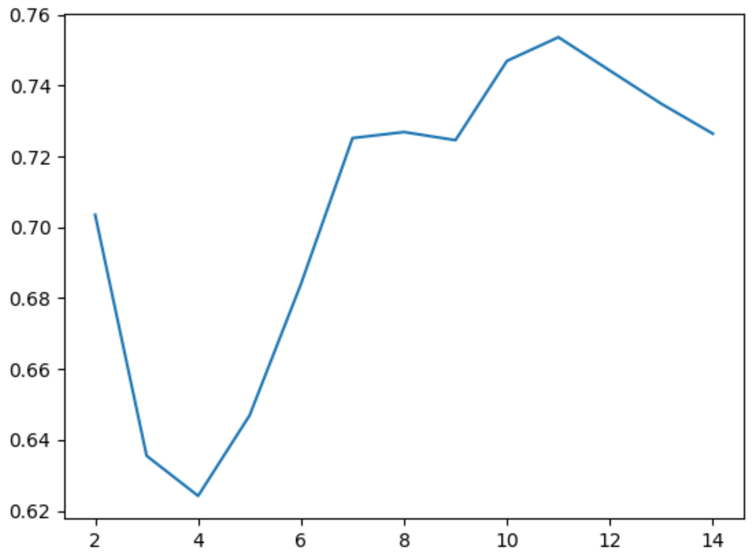
Before and After Filtering Extraneous Colors

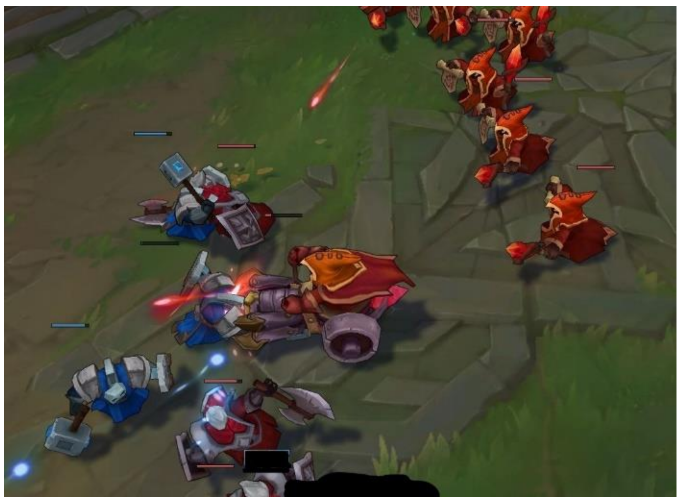
By HSV filtering low saturation values on the test image (upper left), the result is an image that only has the highly saturated blue or red of a minion's clothing. Before filtering there may have been 5 color clusters (red, grey, gold, brown, green), after filtering there are only 2 (black, red/blue). While black is the largest cluster and thus most dominant color, checking average RGB value for the second largest cluster tells you whether R is higher valued than B.

However, this meant k-means color had to be run the same number of minions on screen per frame, which proved too demanding, The more economical solution was taking the average RGB values of the non-zero (nonblack pixels) pixels after filtering, and labeling the image with whichever value R or B, was higher.

III. K-means Clustering

Up until this point, all attempts at labeling/classifying minions have been on pre-rasterized single minion images. For an agent to interact with its environment, it needs to recognize objects in real time and then classify them. In the same sample screen capture as above, there are about 500 KPs distributed among about 10 minions. K-means clustered the features into minions, and then an area around the centroid is rasterized producing captures like the test image singles above, and then fed into the classification system.





Silhouette Graph of the Sample Image with n_clusters on X and Score on Y

The problem is that n_clusters is a predetermined parameter. To solve this, we ran k-means 14 times with a n_cluster value between 2-15. The cluster that produced the highest silhouette score was chosen. In this example, the graph predicts that the best n_cluster is 11.

D. Results

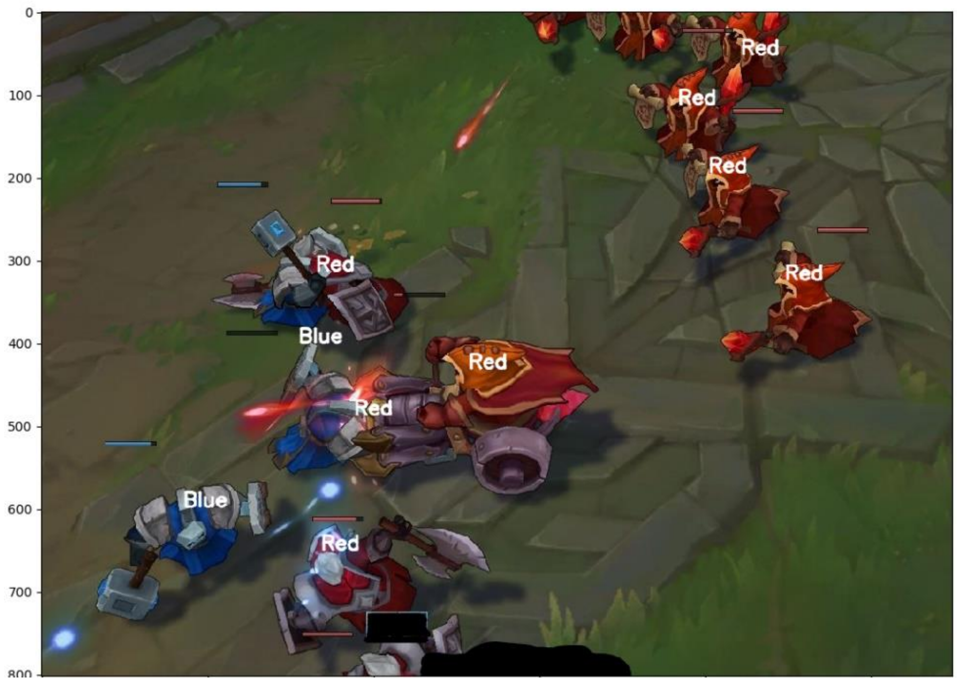
Model Evaluation and Validation

At the beginning of this project, I was going to compare my hybrid classifier to a proven classifier like TensorFlow as well as a computer vision only classifier. The two goals were 1) achieve a higher accuracy than the computer vision classifier 2) achieve a faster run time (fps) and training time than TensorFlow.

It was only halfway through working I realized that what I thought of 'object detection accuracy' was really two tasks disguised as one. The first is detecting potential objects, the second is the classification of potential objects.

Early brute force detectors created potential objects by basically treating all regions of an image as a potential object. Then, R-CNN's used selective search, which views the image through different sized windows, then grouped adjacent pixels together based on size to create potential objects. Today, the YOLO detector divides the image into 845 predetermined regions of interest of varying size, then scores each box based on how confident it is that an object is inside. After a object detector creates its regions of interests, it begins classifying with some form of CNN.

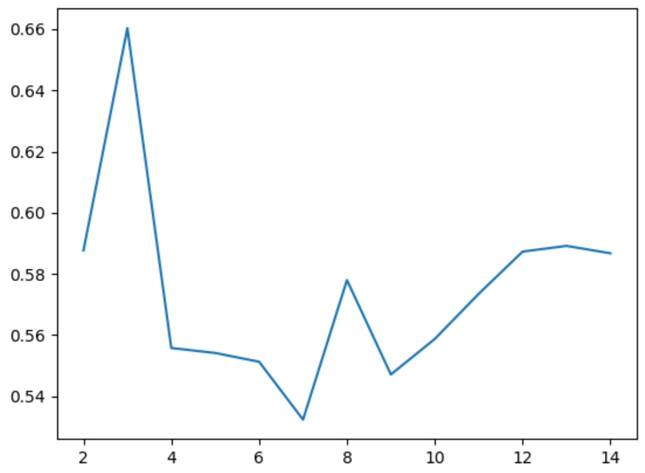
I wanted to see if both tasks could be completed using as few possible, if not zero ML techniques. I was able to create a low-accuracy classifier using only SIFT, but was unable to create an object detector using only SIFT without the help of k-means. There are other computer vision techniques that could aid or even replace SIFT in object detection such as foreground extraction and optical flow. However, I did not have a chance to implement such a model.

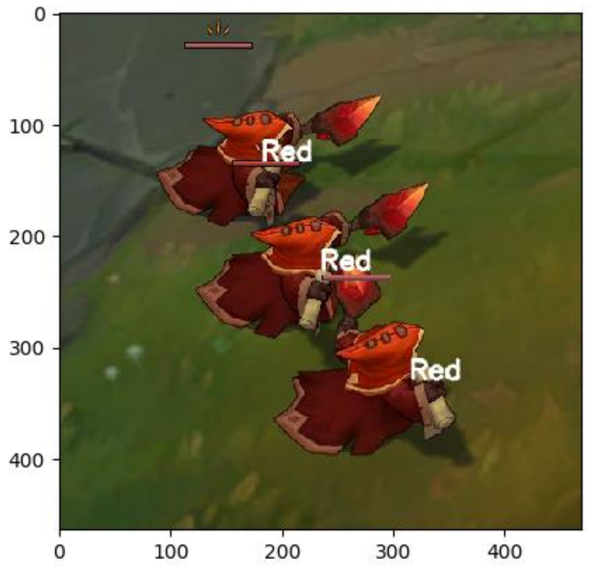


I was able to create classifier with a run time of about half of tensorflow's but required technically no training time (asisde from designing the algorithm). For the first task of detecting objects, when given an image and the n_clusters parameter, it was able to determine a minion's location on an image 90% of the time. However, there were few examples where every minion's location on an image was correctly identified.

In the result image above, n_clusters is hard coded to 10 (ignoring the clipped minions at (400,780) and (750, 0)). In the top right, four minions were successfully detected and classified as red, while the two in the bottom left were also identified and classified correctly. The center four overlapping minions proved difficult as the single cannon minion was detected as two minions because of it's size and resulting numer of KPs, and two blue minions were detected as one.

When n_clusters was determined by silhouette score for an image or screen, results were also inconsistent.





The above example shows the ideal scenario of selecting n_clusters based on silhouette score. The graph easily predicts n_clusters=3 as the best option thanks to visible and separated minions. However, it's unlikely that minions aren't overlapped like in the test image with 10 minions. For most test images, the silhouette predicted that n_cluster was only +/-1 of the actual number. For color detection, as long as the minion's location was accurately determined, 99.9% of the time the color was accurate as well.

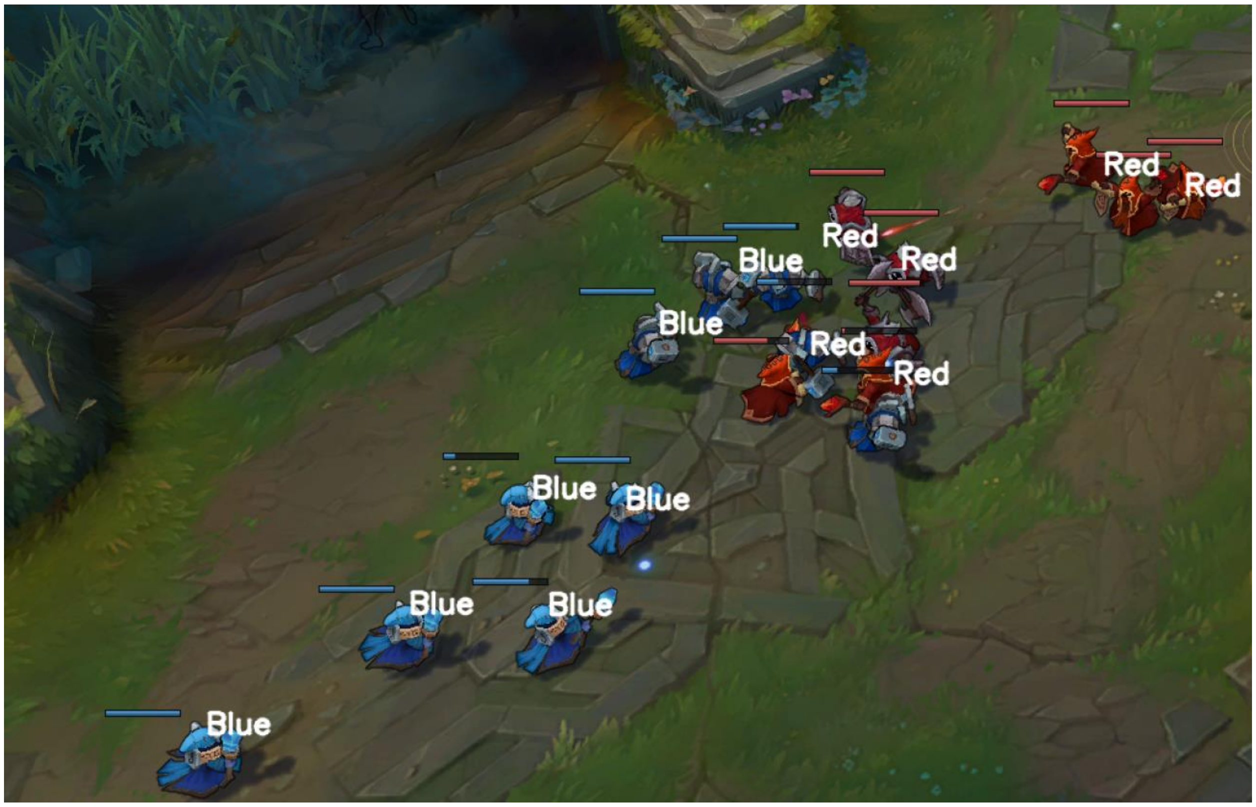
Justification

Halfway through the project, I'd kept the performance (fps) metric, but had divided the accuracy metric into two tasks. The computer vision benchmark model I created was unable to complete one of the tasks and performed miserably on the other. SIFT features without k-means was unable to locate minions and had worse than random accuracy determining their type or color. Other non-SIFT computer vision techniques could determine minion color and could likely determine its location and type as well.

The tensorflow benchmark is already known to have a 99%+ detection and classification rate; compared to my model's 90% detection rate, 16.67% classification rate (random) and 100%* color classification rate (*when minion is accurately located).

Thus, using my original metrics of speed and accuracy, my final model and solution is not significant enough to have adequately solve the program. The best present solution compared to my alternative is twice as fast, and 6.6 times more accurate in both detection and classification. A man can dream though, a man can dream.

D. Conclusion



a) Screenshot of the algorithm running on live captures
b) The max number of minions the algorithm is able to detect and label is 15, however there are 18 minions on the screen and the algorithm only displayed 13 labels
c) The five nicely separated blue minions in the bottom left are correctly detected and labeled
d) Of the 9 minions in the center, the two blue and two red from the 9 o clock to the 2 o clock position are labeled correctly, however the five overlapping in the center are only counted as two

Reflection

I didn't know what I was getting into when I had this goal of "train a League of Legends AI". That goal was broken down into 10 different problems, each of which were broken down into 10 different problems, etc. etc. In summary:

1) Attempt to capture the environment and objects the agent will interact with
    a. Categorize the environment into necessary and unnecessary parts
2) Create a detecter/classifier to locate objects and relay position to the agent
    a. Find benchmarks to compare performance to
3) Understand the different data found on screen (HSV, SIFT, RGB)

4) Experiment using SIFT to classify static pre-rasterized images. Settle on color classification using HSV and RGB values
5) Experiment using SIFT to detect potential objects in static images
6) Generalize to live screen captures

This entire project was difficult. I had some experience programming before this nanodegree but nothing close to this. Sometimes I'd wish that I'd chosen something simpler and more explored (especially on the 14th of each month) like self-driving cars, making this project easier and faster. Most difficult was when I had no direction or crumbs to follow as to what to do next, what solution I can try next. Similarly, pursuing a solution to the end only to realize it wasn't good enough. Even now, I wish I hadn't invested so much time on SIFT and pursued motion detection. Hindsight is 20/29.

From the beginning, the idea was to use as little machine learning as possible and try to exploit the faster run times of computer vision techniques and uniformity of video game environments. Using machine learning would have been easier and more accurate. I'd spend 3-4 hours a day for a week creating and labeling a dataset of 5,000 images for each of the minion types, and then train a CNN. Even if it didn't work out, I learned so much about programming and myself through this project and have picked up tools that will be useful for the rest of my life.

I started this project before using it as my capstone, and will continue to work on it after submitting it. I simply can't extend my enrollment any farther despite wanting to include the rest of my findings. I didn't even get to training an agent.

Improvement

Right now, color identification on a pre-rasterized image is 99%. The problem is correctly identifying all the minions and their location. The problem with using k-means and SIFT is that the coordinate locations of overlapping minions are too difficult to differentiate.

The less minions on a screen and the farther apart they are, better. One possible solution is to store the descriptor values of each minion's KPs and track those. Instead of performing k-means 5-6 times a second, perform it only once a second, and tracks each clusters KPs for the remainder of that second instead of creating 5 more clusters in that second.

Another solution is using foreground extraction to detect potential objects on a screen instead of using SIFT. By tracking groups of white foreground pixels against the black background the number and location of minions can be more accurately tracked.