Cheats methods for videogames using Computer Vision and Deep Learning techniques

Francesco Milizia, Stefano Saravalle and Emanuele Segatori

July 2023

Abstract

The idea of a project involving Artificial Intelligence in videogames cheats come from the lack of open source and public available code about complete tools that can simulate human behaviour in this field. Although there are some commercial product and programs out in the Internet, it seems like there are not ideas and softwares that chain together all the important feature needed for such a complicated task. Our ambitions was to explore this new field and find an original solution involving multiple aspect of human behaviour, like Computer Vision for Object Detection and Deep Learning for Mouse Movement.

Introduction

Video Games and Cheating

With the raising of video games to the public attention, an entire ecosystem started to proliferate around this new phenomenon. Since the late '80, this new kind of entertainment begun an every day activity for young people and *aficionados*, and then, decade after decade, this industry is now worth billions and move more money than the cinematographic industry. We could start a long dissertation around the history of video games, but it's not the purpose of this report.

This brief introduction was necessary to understand why there is an entire market orbiting around tools for cheating in a video game. People cheat just for fun, because they could be not so good and feel the peer pressure (especially for youngster) or because they are pro-player or streamer that live with money made from videogames.

That said, company try to detect and ban people who cheat, and with the growth of new technology, things started to adapt to new tools and softwares. Our purpose was to explore this new world of cheats based on artificial intelligence and try to be part of this new arm race between cheaters and companies.

We choose **Counter-Strike:Global Offensive** for it's popularity around the world and for the large datasets that can be found on the internet. The game is from the genre of FPS (*First Person Shooter*), that is a game where we control a player from a POV perspective where we can shoot with guns to eliminate other player. Those types of games contains a lot more, but for the sake of brevity we are not going more deeper, and we will focus just on the aspect of eliminate other player using AI.

Figure 1: *Frame for Counter-Strike:Global Offensive*

Traditional Approach VS Our Approach

We can classify the type of cheating software in two main branches, the ***Knowledge-based*** cheat and the ***Active*** cheat. The first, aim to gather info about the game that the player should not know, the second actively modify the game to improve the player capacity. To give an example for each one, a **wallhack** can make the player watch enemies behind solid structures and is of the first type, an example for the last one can be an **Aimbot**, that is a software that can aim with pinpointing precision, more than every human can physically do.

(a) *wallhack* **(b)** *aimbot*

Figure 2: *Example of said cheating methods*

There are some example out in the internet of cheating software that use AI, but there is not a single line

of code available for the public that doesn't use premade model as the entire project structure or use some very obvious technique that can be recognized even by humans. So we choose to create a program very similar to an aimbot, using object detection for enemies and deep learning for human-like mouse movements.

Project Structure

Our project can be divided in two main part, the OBJECT DETECTION and the MOUSE MOVEMENT.
The idea is to take a frame from the game, fed it to the model, extract the features (like position of enemy head and body) and then tell to the other model where we want to go in the screen. Done that, the second model will output a 100 point path in which our mouse will move to simulate an human.

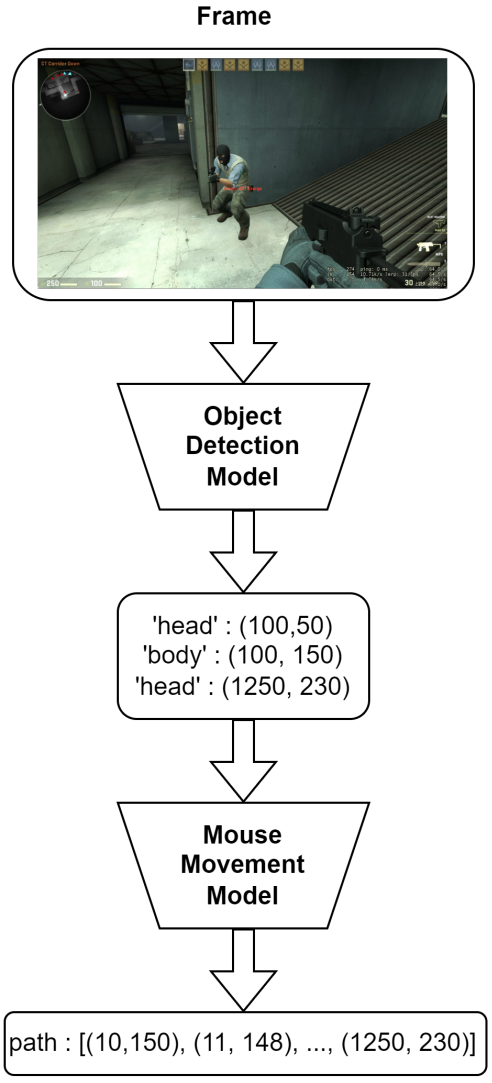

Figure 3: *Project Structure*

Attempt analysis for object detection

Dataset and preexisting models

As said in the first part, we took **Counter-Strike:Global Offensive** (from now on it will be referred as CS:GO) for the big datasets and models present on internet. The first part of our work was to found some already existing dataset and try to understand how to do Object Detection. At first, we wanted to use **YOLO**, a python solution for custom object detection that make this task very easy with the right dataset. Even if we found a file containing weights for object detection using said tool, we choose to implement it by our own, using transfer learning and PyTorch. A website called **Roboflow** contains ton of datasets for a lot of games, other than offering for some of them a premade model, so we started to try different datasets.

Dataset from internet

We tried first to look at some dataset online and try to do the training. For istance, the following are two of our dataset found online, the best in terms of images and the biggest:

- Best
- Biggest

Despite the goodness of the images, the model we choose and the training we had done, the precision was still not good enough. This, because the training was done on resized images that had very low resolution. For this motivation, everything a little bit far was not well recognized, even if the precision was getting higher and higher.

Custom dataset

Understood the limitation of found dataset, we decided to go on our own and try to create a custom dataset. To do so, we created a file to take screenshot of the screen every second. Done that, with tools of image labeling we labeled every image, and then with another script we split our dataset in the training part, the testing part and the validation part. Even if this approach would seem optimal, there are technical and human limitations. In order to do the training on higher resolution images, we had to pay for more computational power, because the one provided by Google Colab for free was not enough, and after a while the site kick you and delete all your advancements.
There are also problem related to the man power we

had. Labeling is a long and tedious process, and in order to create a good dataset at least some thousands of images must be labelled. However, as will be discussed later, we followed this path because of it's flexibility and extraordinary results obtained. All the material can be found in the Link section at the end.

(a) *Best dataset* **(b)** *Custom dataset*

Figure 4: *Comparison between images from the best dataset found and the custom one. There aren't bounding boxes on the second image because those are stored as coordinates in a separated file*

Weights found on internet

Finding a good dataset, training a model and try to create a custom dataset took us weeks. For this motivation, we choose to continue with a YOLO model using found weights on the internet [1] to speed up the process. Once we had a functioning prototype, we then tried to move to a custom solution using PyTorch and Transfer Learning.

Custom solution using PyTorch and Transfer Learning

After our research, we found a way to do custom object detection using tools provided by PyTorch. With the already trained network called *ResNet50* and the class *FastRCNNPredictor* the task of creating a custom object detector is very easy. In fact, ResNet is a model trained on thousands of images that can perform alone object detection of hundreds of things. Using the already provided weights, it can be done the so called **Transfer Learning**, that is the usage of pre-existing weights that will be fine tuned and adjusted using our images in order to train the model to do our task and not the general purpose task it was intended to do. It should be said that during the creation of our custom dataset, we just used the script for screen recording and the tools for image labeling, but we didn't split our dataset using the script described earlier because the images were not this much, the paid computational power on Google Colab was running out and fortunately, just during the first test we saw how well it performed without the using of any other metrics other than our eyes. Also, we decided to use a *decay learning rate* approach, since it helps the model to converge faster.

[1] `https://github.com/Lucid1ty/Yolov5ForCSGO`

Attempt analysis for mouse movement

Model research

As for the object detection, we tried first to search around the internet for model or dataset. There are plenty of solution that can be found on specialized papers, but those models seems to be way out of our pocket. We firtsly tried RNN (*Recurrent Neural Network*), but those can predict, given n steps, the $n+1$ step. So we explored other solutions, like GAN (*Generative Adversarial Network*), Bidirectional LTSM (*Long short-term memory*), cGAN (*Conditional Generative Adversarial Network*) and others, but none of those worked the way we wanted.

Dataset creation

While working on the networks, we also developed some script to create datasets to train the models. For the mouse dataset we wanted to train the model on precise, point to point movements. We therefore wrote a script to record the movements between each activation of the left button of the mouse, used to fire in a **aim trainer**, a common type of software used to improve humans mouse movement. It should be noticed that, as our dataset is composed of our movements, the model will inevitably pick up on our strengths and weaknesses (mostly).

PyTorch Solution

As a real life example of the Occam's razor, using a dataset found online we created a simple neural network in PyTorch using few lines of code. It takes as input the destination in which we want to land, and compute the path as a collection of point in which the mouse should land in order to create a human like mouse movement. It works perfectly for our intentions.

Unfortunately, we didn't use the script wrote by us for the mouse dataset creation because the one found on the internet was just already perfect, and trying to modify our program to match other people idea would have been a waste of time and useless overhead.

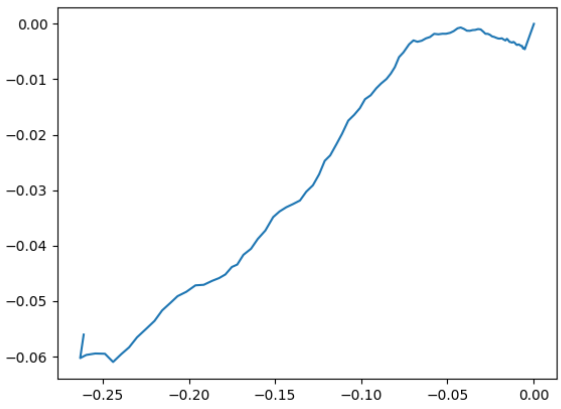

Figure 5: *Example of human-like path for mouse movement generated by the network*

Program structure Analysis

How the video is processed

The first thing to do in order to be able to process the frame, is take the video feed from the game application. Fortunately, there are plenty of solutions in Python for said task. We choose the *win32* library, for it's simplicity and vast documentation. After acquiring the window of the application, the frame are analyzed through the object detection model. The number of frame analyzed each second is obviously less than the actual frame rate that the game has, this because the computation take a while, more or less 50-100 ms between every calculation and the two models. After the frame analysis, the features are extracted and passed to the mouse mover script. The extracted point of interest are represented as the upper left corner coordinates on the screen and the bottom right corner coordinates on the screen of the box in which the head or the body of the enemy is. Other than the bounding box coordinates, also the confidence is outputted represented as a percentage of the precision. Then, before passing those data to the other part of the program, we check if this score is higher than a certain threshold (80%) to avoid miscalculation.

Also, we should notice that the program, once started, keep listening for the game application opening, so, the game can be already opened or we can open it after the initialization of our cheat software. Other than all the calculation printed on the terminal in which we start the scripts, a little window is opened in which there are the frame analyzed by the model in real time and the bounding boxes.

Figure 6: *Bounding box taken by the model on game screen*

How the mouse model is created and trained

The idea behind this cutting edge technology is very simple. The network take as input a pair of coordinates and return the path that the mouse should follow as a list of one hundred points. The network has two layer and in between there is an activation layer using the ReLU function, since is one of the most used nowadays.[2] The first layer as 2 as the dimension of the input, and then one thousand as the dimension of the output. The other layer is then connected with the first (so obviously *in_dimension = 1000*) and then it outputs 200 numbers. Those numbers are then parsed in an array of 100 couples, using an *Unflatten layer*. Each pair represent a coordinate.

Since the model is pretty easy, the training is fast and not so expensive in terms of computational resources. The model we use is trained on a public dataset found online for 1000 epoch. Before feed those data to the model for the training, we had to parse them using a little script. In fact, in this file each path was represented as two lists of size one hundred, the first containing the x values and the second containing the y values.

Data normalization

The model works with uniform value between one and zero, so if C is a set of coordinates, we have:

$$C = \{c_i \in \mathbb{R} | c_i \in [0,1], 1 \leq i \leq 100\}$$

To normalize the screen coordinates, each of them is divided by the screen resolution, and then, the inverse operation (multiplication by screen resolution), is performed on the output of the model.

Along with this type of normalization, the model assume that the starting point is (0,0), so we need to translate our coordinate system in the one of the

[2] `https://it.wikipedia.org/wiki/Rettificatore_(reti_neurali)`

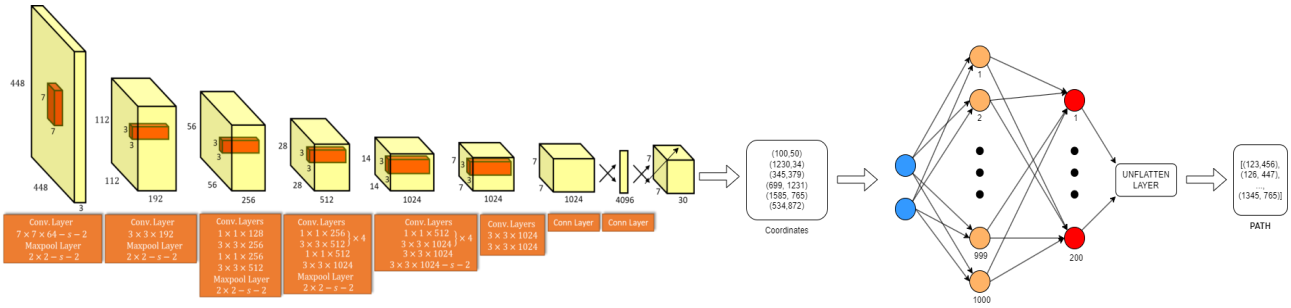

Figure 7: *Representation of the model*

model. For doing so, the program simply take the distance from the crosshair (that is, the point from which the gun shoot), and the actual target. So now we have our starting point in (0,0) and the destination point represented as the distance along both axis. We could add back the player crosshair position to the predictions to perform our movements but for other motivations the mouse movement should be relative to the position of the player perspective. This topic will be discussed in the next section.

Lastly, due to sensibility of the mouse, a fixed value has to be divided from the coordinates. In fact, the sensibility tell the game how many pixel move when a mouse movement is performed, so it can be very different for each value the player choose. The limitations and operating conditions will be discussed later on this relation.

How the mouse is moved

Even if the approach described in the previous section was very good in theory, there are some problems applying it to the things we want to do. First of all, we don't want all of those points. This, because even with the smallest delay between each movement, one hundred points are too much, and a key point in doing an aimbot is to achieve inhuman performance, so very fast and accurate moves. For said purpose, from the one hundred points we take one every 5. So be c_i a coordinate, we had $c_1, c_2, \cdots, c_{100}$, we obtain:

$$c_5, c_{10}, \cdots, c_{5i}, \cdots, c_{100} \quad \forall i : 1 \leq i \leq 20$$

That said, there is still a little problem. The model know the destination point but in the output, being a prediction, it can be a little out of phase . So, even if it's very precise, there are still some error of very few pixel, but if the target is far away, those pixels can be translated to a huge aiming error. So, in order to be one hundred percent sure in hitting the enemy, we add another last movement to move the mouse from the actual position to the destination point.

It should be noted that the movements on the screen are all relative to the previous position, therefore when we perform an absolute movement, the coordinate system will change every time we move the player perspective and our movements will be miscalculated

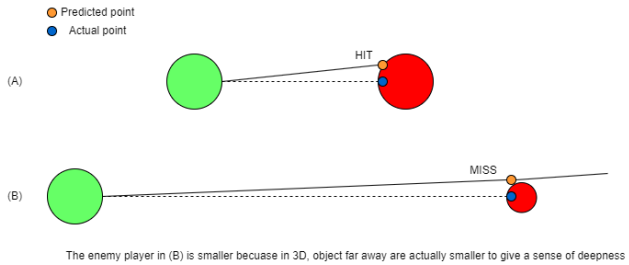

Figure 8: *Representation of little error translated in aiming errors*

Result and limitations

Experimental Result

The software perform very well under the right conditions, bettering by far human reactions time. It looks also pretty human in the movements by an expert eye. In fact, we should consider that pro players (that is, players that play games for living) have insane reactions time, and they train hours each day to perform like no casual player can do. So, even if some movements could be seen as robotic or done by a computer, there are still many legit players that were accused of cheating for moves more robotics than our program can perform.

Operating conditions

It should be said that our program contains some limitations related to the computer in which it is executed and setting of the games. First of all, the library used for acquiring video feed is bounded to windows, so the program can't be used on any other operating system. This choice was not so difficult, this because Windows can be considered the standard operating system for gamers for a variety of reasons that are far beyond the scope of this project.

Another important operating condition is the game sensibility and the mouse sensibility. The game let the player define a proper sensibility, and for the sake of simplicity, we choose to not alter the default value, that is 2.50 (absolute number). For the mouse settings, the DPI value (DPI stands for *dots per inches* and is a unit of measure for the mouse sensibility) was 4000.

The program should be run with all those precautions in mind, because as said in previous sections, there is a value for data normalization that is hardcoded in the program in respect of those value previously discussed.

Improvements

We admit that our program is not perfect. First of all, if the targets are moving fast, the cheat does not work because the mouse movements are done sequentially after the analysis of the frame, so if in this time between the feature extraction and mouse model computation the enemy moves, the human like mouse path will not be valid anymore. This can be improved easily with threads and parallelization. Also, the object detection model sometimes can do some errors. With more training, and with the right resources (as discussed previously),it can be improved to obtain better results.

Considering our limitations, if someone wants to create a proper cheating software that can be used even in pro gaming without being noticed, there are few and simple step to do in order to better our project.

Extensibility

Given the model limitations, further research can be done on other games. It should be noticed that all the code we have produced is game independent (with the assumption that sensibility between games is the same, or with the condition of changing the hardcoded value for sensibility). So, with enough time and resources, the problem of porting the program to other game reduces to create a proper dataset with the tool provided by us and then train the model for object detection. If the script are executed in the right order on the right data, this software is infinitely extensible.

Links

- Custom Object Detection Dataset
- Mouse Dataset
- Weights for the models
- Github link for all the code here described