CS230

Counter-Strike Self-play AI Agent with Object Detection and Imitation Training

Chenyang Dai
Stanford University
qddaichy@stanford.edu

Abstract

In this project, I'm building an AI agent that can self play the popular first-person-shooter video game Counter-Strike:Global Offensive (CS:GO). The AI agent consists of two neural networks: One is a Yolov5 [1] network used for enemy detection; The other one is a custom deep network used for movement and orientation control. Unlike many other game agents that rely on game API support, this CS:GO agent is trained through supervised learning and imitation training (behavior cloning). The agent plays the game purely based on input game frames. The best performing agent achieved veteran players level of shooting accuracy and learnt notable human player like behaviors in real games.

1 Introduction

Deep learning has made remarkable progress on surpassing human performance in various video games. AlphaStar [2], the well-known real time strategy (RTS) game StarCraft II AI agent developed by DeepMind, reached GrandMaster level above 99.8% of officially ranked human players. OpenAI Five [3], the popular Multiplayer online battle arena (MOBA) game Dota2 AI agent developed by OpenAI, defeated world champion team OG in 2018.

Something these AI agents have in common is that learning the game is often modeled as a reinforcement learning problem. In-game API support for large scale simulation and massive computation power is required in order for the algorithm to learn effectively. AlphaStar experienced up to 200 years of real-time StarCraft play [2] and OpenAI Five experienced about 45,000 years of Dota2 self-play over 10 realtime months [3].

Unlike many other games, the famous First-person shooter (FPS) game Counter-Strike:Global Offensive (CS:GO, appendix contains more game details) does not have an in-game API to get real time game status or to simulate games at scale for training. To build the CS:GO agent, I am taking the imitation training and supervised learning approach with no pre-installed knowledge or pre-implemented playing strategy. Specifically, the agent will learn to move by watching how humans play the game and learn to detect enemies by labeled game images.

2 Motivation

I think this project is both challenging and interesting because CS:GO is a fast speed FPS running at 45 frames per second minimum. Every decision has to be made real-time for the agent to play the game successfully. Also, FPS games without API support are similar to many real world problems like auto driving cars, security systems and so on in many ways. Without access to large-scale simulations, the AI agent will only get the same information as humans when learning.

CS230: Deep Learning, Fall 2021, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

3 Game Setup

To reduce the complexity of the project, the agent is trained specifically for CS:GO aim mode and deathmatch mode. Collaboration with teammates and resource management are less important in these two modes.

Aim mode provides a controlled environment for human players to improve their shooting skill. The player stands fixed in the centre of a specifically designed training map. Enemies will respawn randomly around the player. It is possible to set the enemies to be static or to run towards the player. In this mode, the agent does not need to move, only aiming and shooting are tested.

Deathmatch mode rewards players for eliminating enemies on the opposing team. After dying the player respawns at a random location. In this mode, the agent should distinguish between teammates and enemies and navigate through the map to search for enemies. The chosen map is Dust 2 for this project.

4 Dataset and Training

Unfortunately, there was little ready to use dataset available. I only found one CS:GO dataset [4] which consists of about 1500 game images with players bonding boxes labeled. However, this dataset was captured using an older version of the game. Character models and map details have changed a lot since then. As a result, I collected the majority of the data in this project myself.

The first part of the dataset consists of about 8000 gameplay images and the corresponding XML label files with coordinates of counter terrorists (CTs), terrorists (Ts) and their heads in each image. Firstly, I captured 2000 images using a python script that periodically takes the game window screenshots while playing the game. All the bonding boxes are labeled by hand using an online Yolo labeling tool called Roboflow [5]. Secondly, I mixed in the 1500 images dataset I found online to form a 3500 images set. Next, I applied data augmentation. More precisely, I applied cutouts to create two training images per original image with two random 10% black boxes. Cutouts simulate enemies that are partially hidden. Finally, as the initial trained object detection model performed poorly on detecting far away players. I added another 500 images mostly containing far away enemies and applied the same augmentation to form the final 8000 images dataset.

The second part of the dataset consists of about 750k continuous game play frames (30 frames per seconds, around 7 hours in total) and the corresponding keyboard and joystick input on each frame. The first 5-hour data is captured during normal play and the remaining 2-hour data is captured specifically at places where the agent is likely to get stuck. I used a joystick rather than a mouse to control the player orientation because prior work showed that CS:GO resets mouse position at high-frequency and irregular intervals. Naively logging the mouse position at the required frame rate fails to reliably determine player orientation [6]. Using a joystick can reliably capture a human player's input. Each game frame is flattened as a one dimensional vector. The corresponding player input is stored as a (1x9) vector where the first 7 digits correspond to seven keyboard keys and the last 2 numbers are real X-axis and Y-axis values of the joystick (appendix contains more details). The recorded game frames are pre-processed before training the movement network as shown in Figure 1 .

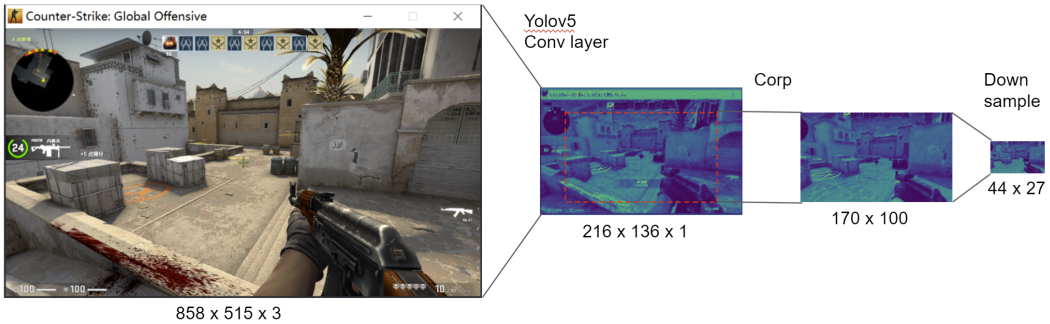

Figure 1: Game frame is compressed through Yolov5 conv layer, cropping and downsampling

The original game frames are captured at 858 x 515 x 3 resolution. Each frame will be passed through the first 2 stages of the trained Yolov5 network for feature extraction. The output becomes 216 x 136 x 1. Next, the output is cropped to a central region of 170 x 100 and downsampled to 44 x 27. Finally, 60 continuous frames are combined into one input sequence. This pre-processing significantly reduces input size and allows both fast training and fast inference.

5 Methodology and Network Architecture

The overall idea of the agent design is to have two networks collaborate with each other to play the game. One is the object detection network that detects and draws bonding boxes of enemies. The other one is the movement network that controls player movement and orientation.

Aiming accuracy and reaction speed are the most important factors in FPS game gunfight. After comparing several object detection solutions, I picked the Yolov5s network which has both high speed and accuracy. I modified Yolov5 intermediate implementation to output image feature vectors for certain Conv layers at inference time because these intermediate results are used as input to the movement network.

There are primarily three reasons to use Yolo's intermediate results as movement network inputs: Firstly, each captured game frame is in RGB color. Simply converting it to a grayscale image does not reduce the input dimension enough and there is a risk of losing information carried by color. Secondly, the Yolov5 network is trained on a dataset very relevant to CS:GO. We can probably trust its trained convolutional layers to extract the most important features from each given game frame. Thirdly, the custom designed movement network is not small. Adding more custom Conv layers can be both training and inferring expensive.

In order to better understand the intermediate results from Yolov5 and pick the most appropriate layer output to use, I plotted example outputs from different layers. After comparing different layers' visual outputs and dimensions, I picked the output from Stage 1 (Conv layer 2) as the input features to the movement network. The reason is that stage 1 output contains sufficient information for the movement network to make the correct decision. Using Figure 2 as an example, starting from stage 2 (Conv layer 3), the output image becomes vague. From a human perspective it is hard to tell whether the right half of the image is a wall or something else and it is hard to decide where to go assuming no pre-knowledge of the map. The assumption is, if a game frame is hard for human players to make a decision, it will be even harder for the network to behave correctly.

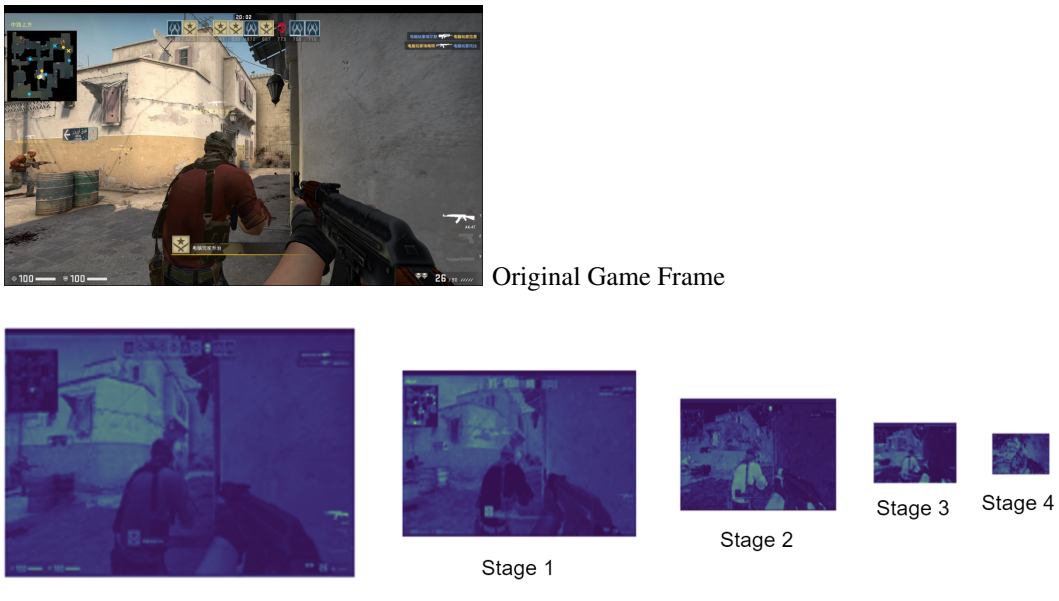

Figure 2: Different Yolov5 Conv layers output visualized (scaled up for demo purpose)

I built the movement network from scratch by applying somewhat standard but effective many-to-one architecture for time sequence data like continuous game frames. A sliding window containing past 60 game frames is one input sequence to the network. Each frame's feature vector, output by Yolov5 intermediate layer, is cropped and downsampled to dimension (27x44) so one sequence will be (60 x 1188) with the frame feature vector flattened. This sequence is passed through 5 layers of GRU with 256 nodes each layer (see section 6 for reasons behind these hyperparameters). Then the network is branched into two parallel fully connected layers to predict keyboard input and joystick values separately. The keyboard branch uses multi-label sigmoid with binary cross-entropy loss because more than one key can be pressed at the same time. If the predicted value for a certain key is greater than 0.5, the key is considered pressed. The joystick branch directly uses the fully connected layer as a regression layer with mean square error loss to predict the actual Joystick X-axis and Y-axis input.

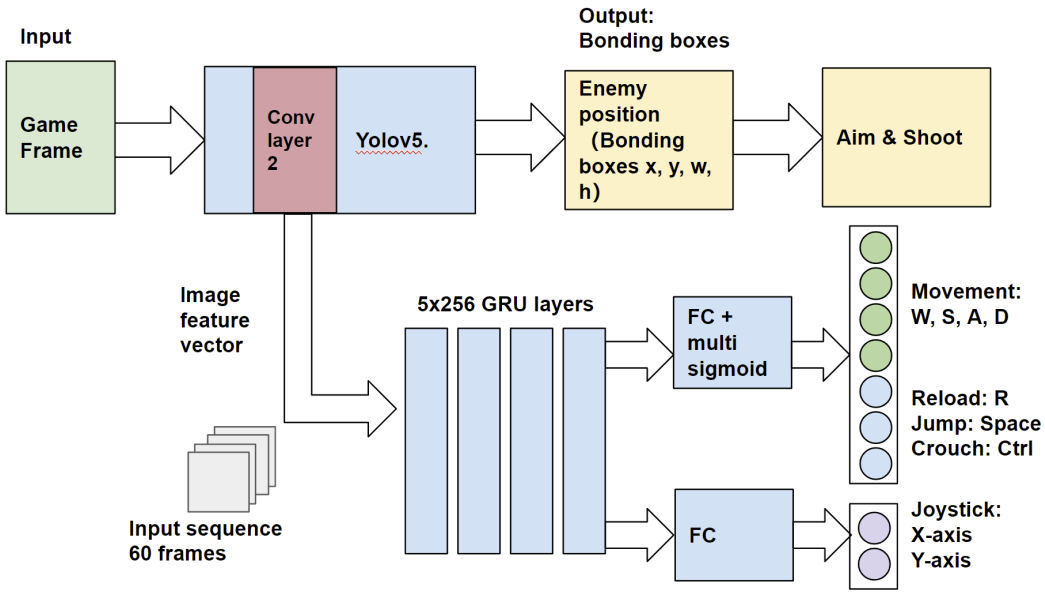

Figure 3: Movement network structure

The final total loss is defined as:

$$Loss = BCE\_Loss(keyboard\_prediction, Y\_keyboard) + \alpha \times MSE\_loss(joystick\_x\_prediction, Y\_joystick\_x) + \beta \times MSE\_loss(joystick\_y\_prediction, Y\_joystick\_y)$$

There are two weight parameters alpha and beta in this loss function and each of which controls the different weights for joystick X and Y axises' losses respectively. The reason to use separate weights is that during data exploration I realized that in real games Y axis adjustment happens much less frequently compared with X axis and both X and Y axises are adjusted less frequently than keyboard inputs. This scenario creates an "imbalanced" dataset. As a result, using different weights in the loss function can guide the agent pay more attention to the X axis and Y axis orientation during training.

6 Experiments with Hyperparameters

For the Yolov5 network, the default small setting is performing quite well. With only adjustment of the learning rate and other minor tweaks at training time, it achieves 92% precision and 73% recall on the testing set. I once tried with the medium setting but the trained model runs somewhat slow at inference time with very little performance gain so small setting is more suitable for this project.

For the movement network, there are many parameters to search. The most interesting ones are alpha and beta in the loss function and the model structures. Given the nature of game AI agents, loss value is not necessarily a good indicator of model performance. Therefore, I manually watched games played by different agents and recorded the number of times the agent got fully stuck during a 10-min interval. The stuck times is used as a rough indicator of model performance.

I performed a grid search for alpha and beta and the best combination is shown in table 1.

α	1.00	1.00	1.25	**1.25**	1.50
β	1.00	1.25	1.25	**1.50**	1.50
Number of stucks	12	6	9	**5**	7

Table 1: Grid search results for alpha and beta with fixed model structure

For model structures search, I defined average FPS as a satisfying metric with threshold 20 because agents running less than 20 frames per second will badly hurt aiming accuracy. The best GRU layers and sequence length combination is shown in table 2. Other hyperparameter choices can be found in the Appendix section.

GRU layers	3	3	4	4	5	**5**	6	6
Sequence length	30	60	30	60	30	**60**	30	60
Avg FPS	33.7	33.1	28.2	27.2	22.1	**20.1**	16.5	15.9
Number of stucks	10+	9	9	5	6	**3**	3	2

Table 2: Grid search results for model structures with average FPS as satisfying metric

7 Result and Future Works

After tuning hyperparameters, searching network structures as well as optimizing the entire system to interact with the CS:GO game client, the final agent was able to run at an acceptable speed (around 20 frames per second) on a modest graphic card (GTX 1080).

Having watched several games played by the best performing AI agent, it is clear that the agent has learnt to play CS:GO to some extent. In aim mode, the agent's shooting accuracy and headshot rate is comparable to veteran players. In deathmatch mode, the agent has learnt many notable behaviors: Jump over obstacles; Walk through complex areas like half closed doors and spiral staircase; Crouch when shooting to be more stable and navigate long distances across the map without getting trapped in a small area. Moreover, the agent learnt to turn around when facing a wall or in a dead end.

On some less frequently used map areas, however, the agent still struggles a bit to find a path and sometimes can still get completely stuck. Also, when enemies are partially hidden, the object detection network sometimes fails to detect the enemy. But I believe both cases can be improved with more specialized training data.

There are many places to improve in the future. First of all, sound is not taken into consideration by the agent today. Pro players often rely on sound to identify the enemy's position. Next, weapon switch, throw grenades and other utilities are important in real games, which the agent is not trained on today. Finally, training the agent to play 5 v 5 competitive game and learn to collaborate with teammates is both a challenging and interesting next step.

8 Conclusion and Future works

With only 4000 original images and about 7 hours of game play data, it is remarkable that the AI agent has learnt to play a video game as complex as CS:GO. With more training data and a more powerful GPU, the agent will certainly perform more properly. The model designing and training experience in this project let me take a first glimpse of how difficult to build one of those complex systems like self-driving car pedestrian detection or wall avoiding robot.

9 Appendix

9.1 Interacting with Game Client

Although not directly related to deep learning, two other major challenges of this project are: 1. Capturing game frames and player input accurately during human play. 2. Building reliably interaction between the agent and the game client. CS:GO has a strong anti-cheat mechanism making

it even challenging. Below are the open source packages I used. Thanks to the python open source community, I was able to get the agent work end to endly.

Game window capture: I used ctypes and win32gui to capture the game window.

Frame capture: The fastest screenshot library I could find is MSS [7], which is built purely on top of ctypes.

Input recording: As mentioned in section 6, the game irregularly resets mouse position at a very high frequency making it not possible to calculate mouse movement using mouse coordinates on two continuous frames. Instead, I used a joystick and pygame [8] package to capture joystick inputs at any given frame. For keyboard input, I used the python keyboard [9] package for recording.

Keyboard input: Keyboard inputs sent by many common packages are not recognised by CS:GO client. In the end, I had to use win32api with Windows hex key code [10] to simulate keyboard input. An example line of code to press "W" is:

win32api.keybd_event(0x57, win32api.MapVirtualKey(0x57, 0), 0, 0) # press w

Joystick input: To simulate joystick input, I used an open source project called VGgamepad [11].

9.2 Game Play AI Play

CS:GO gameplay by pro player: https://www.youtube.com/watch?v=KnpljMWwy3o

AI agent gameplay demo: https://youtu.be/PCtb60Q_KAk

Full 30-minute AI agent gameplay: https://youtu.be/ohWCUUvE6m4

9.3 Hyperparameter Choice

Hyperparameters	**Description (Selected value)**
Learning rate	0.0005
Learning rate decay rate	0.99 every epoch
GRU layers	5 layers
Hidden units per player	256
Input feature size	27 x 44 per frame
Sequence length	60 frames
Yolov5 Conv layer output	Use output from Yolov5 stage1_Conv layer
Mini-batch size	512 sequences (60 frames per sequence)
Training epochs	100
Loss_alpha (loss weight for joystick X axis)	1.25
Loss(loss weight for joystick Y axis)	1.5

Table 3: Hyperparameter choices for movement network

9.4 AI Agent Output Action Space

Action	Meaning	Action	Meaning
w,s,a,d	Moving	ctrl	Crouch
space	Jump	Joystick X	Horizontal Orientation
r	Reload	Joystick Y	Vertical Orientation

Table 4: CS:GO game agent output action space

References

[1] Yolov5 git repository: https://github.com/ultralytics/yolov5

[2] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander S.

Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom L. Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps David Silver. “Grandmaster level in StarCraft II using multi-agent reinforcement learning”, 2019 In: Nature 575, 350–354

[3] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław “Psyho" Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, Susan Zhang. “Dota 2 with Large Scale Deep Reinforcement Learning”, 2019 In: arXiv:1912.06680

[4] CS:GO image dataset:
https://github.com/pythonlessons/TensorFlow-2.3.1-YOLOv4-CSGO-aimbot

[5] Robotflow Yolo labeling tool: https://roboflow.com/

[6] Tim Pearce and Jun Zhu. “Counter-Strike Deathmatch with Large-Scale Behavioural Cloning”, 2021 In: arXiv:2104.04258

[7] MSS git repository: https://github.com/BoboTiG/python-mss

[8] Pygame git repository: https://github.com/pygame/pygame

[9] Python keyboard git repository: https://github.com/boppreh/keyboard

[10] Windows hex key code:
https://docs.microsoft.com/en-us/windows/win32/inputdev/virtual-key-codes

[11] VGgamepad git repository: https://github.com/PJSoftCo/VGamepad