Title page removed for privacy reasons

Contents

List of Tables

List of Figures

Abstract

Fighting games are a complex genre that attracts dedicated players who enjoy challenging matches and often prefer training against real players. Unfortunately, some games have small communities, which makes finding online opponents troublesome. This paper proposes a solution and explains the bot implementation for a fighting game called Acceleration of SUGURI 2. The general system architecture accounts for a game with a closed source code. In this case, the Computer Vision component defines the system's computational and gameplay performance, thus proving that Reinforcement Learning agents can use Computer Vision to gather observations. Finally, this paper explains why the project was unsuccessful in the hands of a single inexperienced developer and suggests possible improvements and areas of additional research.

Chapter 1

Introduction

Video games get increased attention from people due to the growth of the Internet. Nowadays, people can play different games together online. While some genres dominate the market, others stay unpopular. For instance, Fighting games are a niche genre with unique gameplay differences. These quirks make fighting games more challenging, while other genres often offer a stress-free experience. On the other hand, the difficulty usually brings dedicated people ready to spend multiple hours learning the game.

I The nature and the problem of fighting games

Fighting games operate at three different levels simultaneously. With each level, the skill requirements and difficulty increase. These concepts are easier to observe in the fighting games genre, though they can exist in other games.

On the first level, the player has to press buttons to control their character. Usually, fighting games have multiple unique playable characters that might share basic moves, such as a punch or low kick. Nevertheless, they often have a wide variety of special attacks triggered by different combinations of inputs. Typically,

control schemes require the player to press not more than six buttons in total. Over time players move from executing single attacks to complex combinations of basic moves. One of the challenges is to learn precise timings. For example, some advanced mixups may require pressing the next button within a three-frame window. Another challenge is to build muscle memory and learn to execute desired combinations automatically without thinking. Players can acquire these skills alone in training mode.

The second level requires the player to understand the state of the match, decide which move to use next, and react appropriately to the opponent's actions. For that, the player has to learn the practice and the theory of the game that usually explores Matchups - fights between two specific characters. Matchups include the general strategy, recommended moves, some edge cases, and expected win rates for each side. A player who learns one character has to learn to fight against the rest of the roster. Depending on the matchup, a player should adjust the strategy depending on the strengths and weaknesses of each side. For example, one set of moves can effectively work against one character and not the other. The main challenge on this level is conceptualizing when and what moves each character should use.

Finally, on the third level, the player has to observe patterns in the opponent's behavior, use prediction skills, and be unpredictable. At this point, players play the metagame. A metagame is another game built on top of the current one, and its new goal is to outsmart the opponent. The metagame requires a fair amount of experience to recognize patterns in other players' actions, adapt, and trick the opponent. By doing so, players can win matchups even for a weaker side.

To go further than level one, the player should gain experience in real matches against other players instead of staying in training mode. However, fighting games

are a niche game genre due to their highly competitive nature, increased skill requirements, and some psychological aspects similar to real sports [1]. Significantly decreased popularity creates smaller communities where matches occur more rarely. The lack of online opponents makes learning the second and third levels more problematic.

Most fighting games offer a single-player mode where players can fight against built-in bots. They can familiarize new players with the rules and basic moves. However, after players learn more advanced moves, built-in game bots stop posing a challenge because built-in artificial intelligence (AI) is often simple and predictable. Therefore, hard-coded bots are not helpful for advanced players who want to compete with others at higher levels.

II The target game

Acceleration of SUGURI 2 (AoS2) [2] mixes shoot-them-up and fighting game genres. Both players shoot various projectiles and evade them. Opposed to a conventional block mechanic in other games, AoS2 revolves around dodging. Thus, Blocking is replaced with Dashing - a faster form of movement that simplifies and rewards dodging.

In addition to normal moves, the player can use more powerful attacks that cost one bar of the Hyper Meter gauge. Players can fill this gauge by dodging and shooting, even if projectiles miss. Instead of Hyper Attacks, players can also spend energy on emergency shields for short-term defense.

Even though the game is fast-paced, mindlessly dashing can raise a player's heat gauge quickly. The higher heat causes players to get more damage. Thus, players should balance slow and fast movements to stay cold. Otherwise, just a

Fig. 1. Acceleration of SUGURI 2 gameplay

couple of hits may result in a quick ending.

Overall, AoS2 offers a unique, engaging, and entertaining experience. Nevertheless, this game suffers from the same problems described in Sec. 1.1. It has a small community scattered between different countries and time zones, which results in rare online matches. Additionally, the difference in skill is notable even within the established community. Thus, new players find learning the game frustrating because of its unusual mechanics.

A possible solution for that problem is an offline bot based on machine learning to compensate for rare online matches. Since AoS2 is proprietary software, one cannot access or modify its source code. Nevertheless, Computer Vision can help to overcome this challenge.

Unfortunately, video game bots rarely combine Reinforcement Learning (RL) with Computer Vision (CV) because working with images is computationally expensive. Therefore, Computer Vision can become a bottleneck in the system.

However, a real-time bot can still combine both tools.

The goal for the project is a proof of concept – to show that combining CV and RL is viable in some scenarios. Since AoS2 is a complex game, creating a highly advanced bot would not be possible, considering the lack of experience in the field. However, the bot should be able to detect objects on the screen and train to dodge them. Thus, this project aims to make a bot that can only evade projectiles.

This paper describes the implementation that uses Computer Vision to gather observations from the Reinforcement Learning environment. In addition, this paper describes the drawbacks and possible improvements. Almost all system components are complex enough to be worth additional research.

Chapter 2

Literature Review

The literature search aims to investigate state-of-the-art technologies and review modern works in the game bots field. Sec. 2.1 explores different Machine Learning approaches. Sec. 2.2 briefly explains common Computer Vision methods. Sec. 2.3 reviews different implementations of machine learning bots.

I Machine Learning

Machine Learning (ML) [3] is a tool to analyze relations between the existing input data and outputs and predict outputs for new data. ML can use different algorithms, for instance, linear or polynomial regression. They try to find a function that better describes the dependency between the known data and outputs and yields accurate results for unseen data. However, due to poor performance on multi-dimensional data, pure regression tools are inferior to other advanced methods.

The Machine Learning area consists of three main paradigms [4]. The first one is Supervised Learning which requires labeled data to predict outputs. The second paradigm is Unsupervised Learning which aims to find features in unla-

beled data. The last one - Reinforcement Learning (RL) - introduces an agent that interacts with the environment, gets positive or negative rewards for its actions, and tries to maximize the final score. The RL paradigm is the most fitting approach to the problem because the fighting game arena is an environment, and players are agents.

Since the learning environment can increase in complexity, regression tools become impractical. They are replaced with Artificial Neural Networks (ANN) [5] - multilayered structures that imitate neurons and connections between them, similar to animal brains. During training, the ANN adjusts connections between neurons in different layers to recognize patterns in input data and produce appropriate outputs. ANNs become helpful when a simple mathematical function cannot represent the process.

A neural network with many layers forms a Deep Learning model [6] that can analyze the data on multiple abstraction layers. As a result, the Deep Neural Network can recognize patterns in extremely high-dimensional and complex data, often yielding superior results. Deep Learning applies to reinforcement learning agents as well.

In short, modern ML suggests several methods from which Reinforcement Learning and Deep fit the game bot area the most. Sec. 2.3 reviews popular game-related projects that apply these concepts.

II Computer Vision

Computer Vision can detect objects on the game screen using game sprites, such as characters and projectiles. This section describes the most commonly used Computer Vision techniques applicable to the project.

The first option is template matching [7]. It finds entries of a given template image in other frames. This method is easy to use, although it has two downsides. Firstly, template matching does not support rotations as it directly compares pixels. Secondly, it cannot detect more than one object type. Since the game has dozens of different projectiles that can turn, template matching may not be viable.

Another option is cascade classifier [8] that learns to detect features from simple to more complex. Although the initial design solves the face detection problem, one can train [9] the classifier using custom data. In contrast to template matching, cascade classifiers can detect rotated objects if trained with enough data. However, another challenge persists, as the cascade classifier excels at the detection of one object type only. Therefore, one may need to train separate classifiers for each object type, though this task is daunting.

Finally, You Only Look Once (YOLO) [10] is a promising system that extracts all objects from a single frame in one cycle. A single YOLO instance can replace many cascade classifiers and dramatically reduce the amount of work. Such a system can be more performant than multiple cascade classifiers, although that may come at an accuracy cost.

To conclude, the project can involve described techniques for different purposes. For example, trained classifiers can detect players and projectiles. On the other hand, template matching can detect User Interface (UI) elements.

III Game bots

Deep Reinforcement Learning (DRL) shows excellent results [11] in various games. Unfortunately, only a few successful examples exist. For instance, one of the projects creates a DRL agent for DOOM [12] where the bot demonstrates high

performance in a partially observable environment. One of the techniques applied to gameplay is divide and conquer. This concept can be applied to AoS2 to split the match into the early and end game. Another technique used by the DOOM bot is frame skip. While the most optimal number of Updates Per Second (UPS) should be closer to the game FPS, skipping frames and thus lowering to 20 UPS can improve training speed and possibly performance in real matches.

Regarding fighting games, the other study describes a DRL-based AI for a game called *Blade & Soul* [13]. The AI training uses a self-play method with agents focusing on different tactics, such as offense or defense. This technique can help to find the dominating strategy in AoS2 and simplify the training process.

Another well-known project is a Dota 2 bot by OpenAI [14]. This game is famous for its difficulty, which results in a complex environment with many available actions. A bot that competes with professional players is sensational. In addition to previously mentioned optimization techniques, the project implements Long short-term memory (LSTM) [15], which improves the analysis of connections between consecutive game states. LSTM should improve performance in dynamic environments where micromanagement skills define the outcome. In AoS2, projectiles have a specific behavior tied to in-game time, so the game probably has multiple game states logically connected. Thus, LSTM might improve the bot's performance and make it easier to learn the behavior of each projectile.

Finally, Deep Q-Networks (DQN) can create game AI trained with unprocessed screen data [11]. A bot designed for Atari games excels at different titles [16], although these games have significantly simpler visuals than AoS2. Nevertheless, DQN can render an external computer vision model unnecessary. With some optimizations, DQN can yield an acceptable computational performance despite the bigger size of input data. However, the main challenge is to tune the

model to complete the desired task.

In conclusion, deep reinforcement learning can produce excellent results in almost any game. However, one has to find the best approach for a specific problem since no universal solution exists.

IV Results

Reinforcement Learning seems the most fitting approach to the problem because RL can create superior bots in games of different genres. Computer Vision methods can become a part of the DRL even though they might become a bottleneck. Finally, one has to optimize the system for an acceptable computational performance and training speed.

Chapter 3

Methodology

This chapter shows which aspects of the gameplay and user interface to consider during implementation. Additionally, this chapter explains the general system design, plan, and methods.

I General architecture

Several factors affect the project design. The AoS2 provides no open Application Programming Interface (API) to interact with the game data at runtime. Therefore, one has to use either memory access tools or computer vision techniques. While direct memory access is the most computationally performant solution, one has to understand the memory layout to extract the data. Thus, computer vision seems more fitting and universal as it simulates a real player's sight and can work with different games.

Another essential point is keyboard input. Since the game is closed-source, the bot cannot directly send keystrokes to the game. However, programming languages usually have external libraries that can simulate human inputs [17].

Python 3 fits the project best due to its simple syntax, wide variety of side

packages, and quick prototyping speed.

Fig. 2 shows the architecture and workflow of the project. The Gym library provides the interface for the Environment class. Extending this class, one can describe any custom environment suitable for an RL agent.

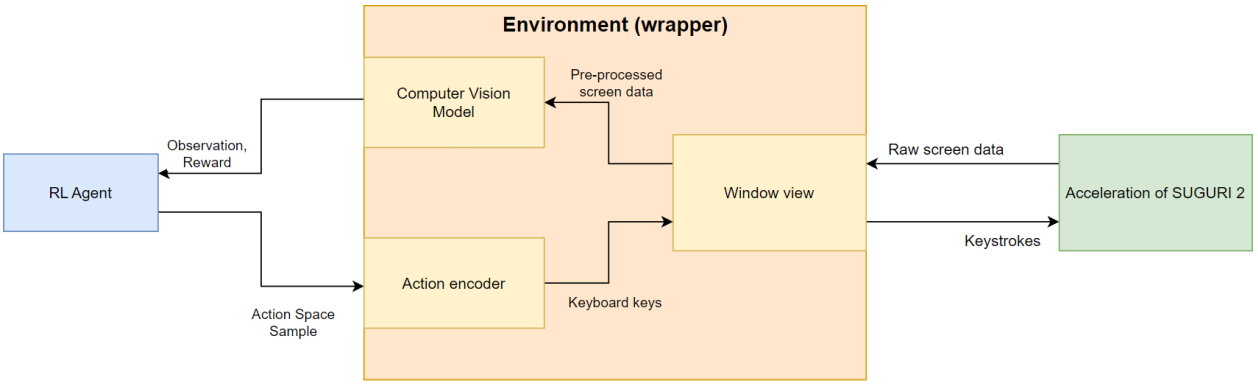


Fig. 2. The general project architecture

The environment is an adapter [18] class between the RL agent and the game window. This adapter has Computer Vision Model, Window view, and Action Encoder. The CV model extracts the player and projectile data from the screenshot. Window view interacts with the game window by capturing the screen in real-time and simulating keyboard inputs. The action encoder receives an action space sample from the RL agent and maps these actions to keystrokes.

II Formalizing the environment

AoS2 is a stochastic and partially-observable environment. Most projectiles have simple and predictable behavior, but their initial directions may depend on the Random Number Generator (RNG). AoS2 can be an RL environment because it meets the following conditions.

- **States**: video games are inherently discrete, and fighting games update

frame-by-frame. Thus, every frame represents a unique state of the environment.

- **Actions**: agents have a finite action space, and each move affects the environment.
- **Rewards**: AoS2 is a complex game. However, one can still build a reward system to cover basic scenarios, such as getting or dealing damage, winning, or losing.
- **Termination**: winning or losing a round clearly defines the end of an episode.

III Computer Vision model

AoS2 offers different fighting arenas - moving background images. However, some people may have trouble seeing projectiles due to bright and moving backgrounds. That is why AoS2 developers introduced various accessibility options, such as high-contrast mode, attack indicators, and numbers on health display.

Fig. 3 shows the difference between the two available view modes. Unlike the normal mode, the high-contrast mode uses a color modulation technique to recolor all character sprites. The player can choose between Red, Green, Blue, Yellow, Cyan, and Magenta. The high-contrast mode also forces the gray background, although arenas still render as usual.

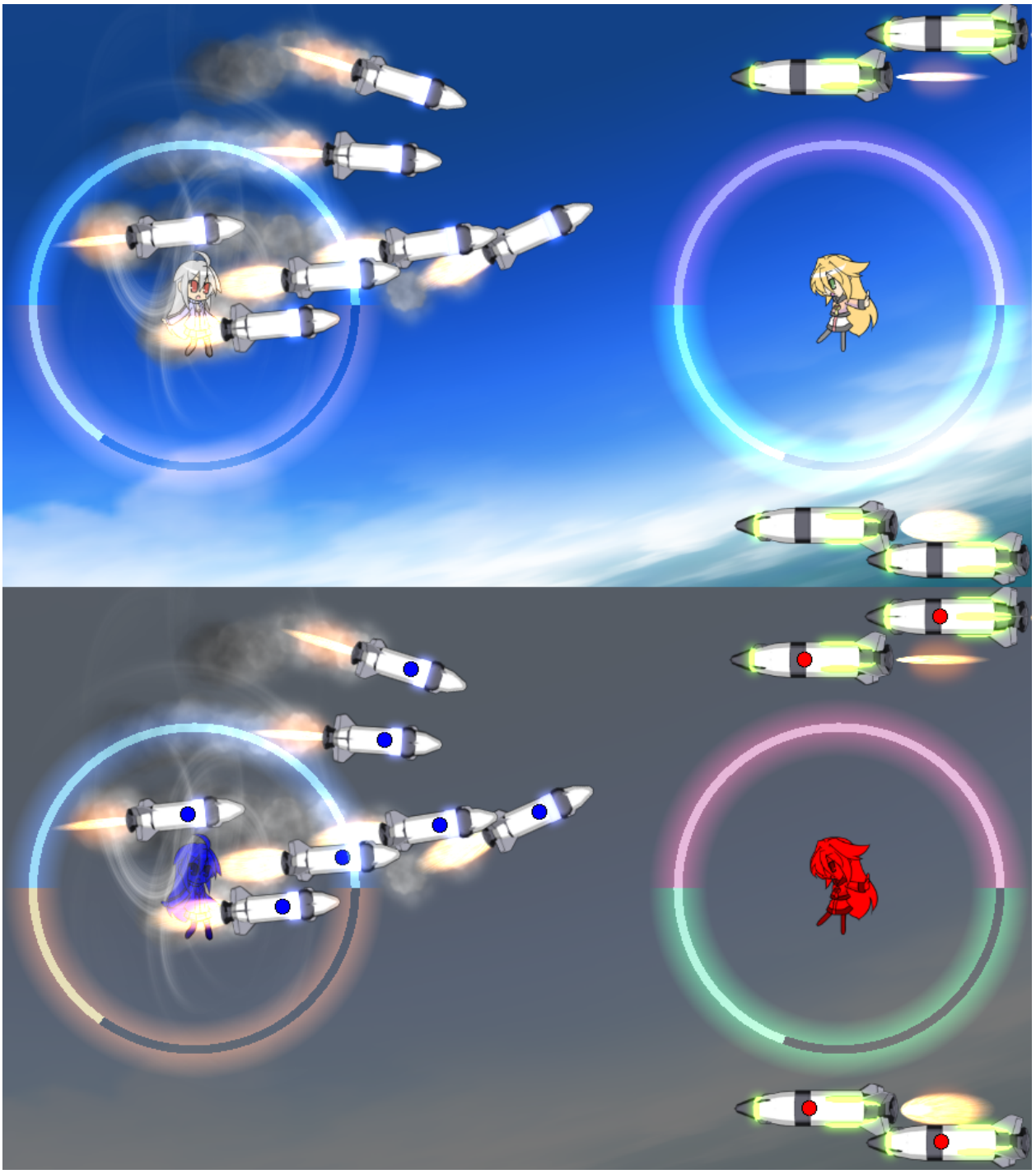


Fig. 3. Normal view and high-contrast mode

Attack indicators are a separate option intended for use with high-contrast mode. Fig. 4 shows that indicators appear above hitboxes, although the hitbox view is only available in training mode.

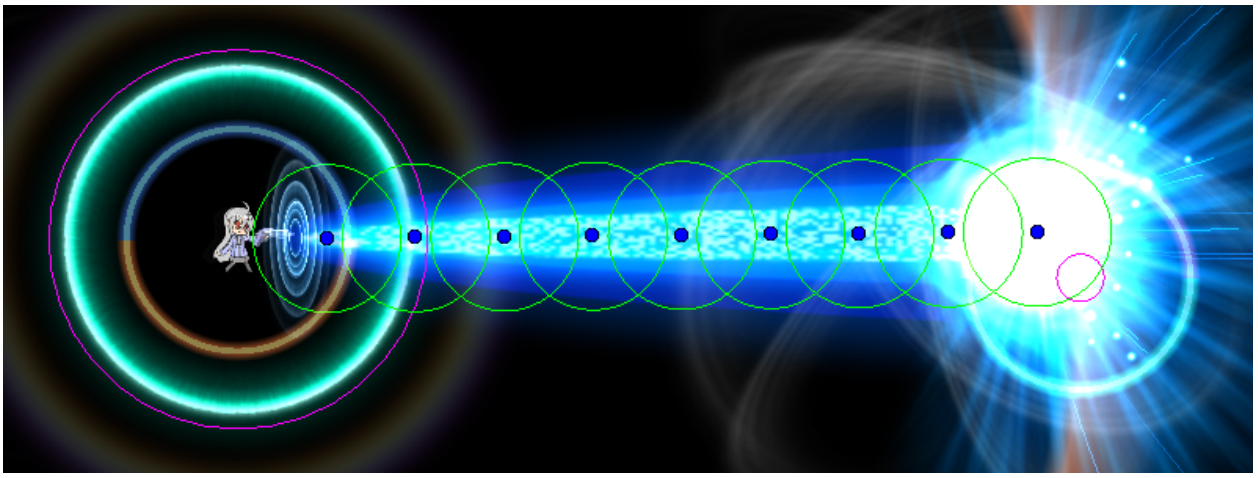


Fig. 4. Attack indicators and hitboxes of a Hyper Attack

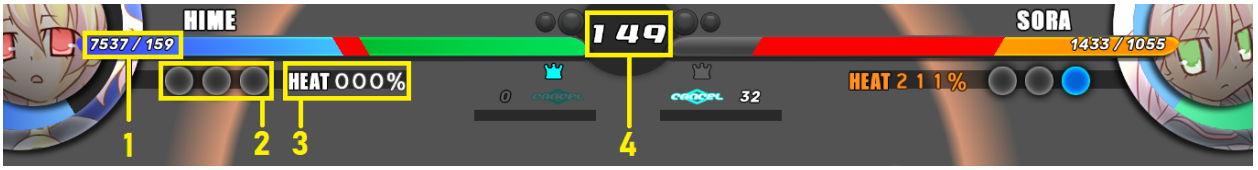


Fig. 5. Top UI

Finally, the top UI consists of multiple elements, although not all are essential for the bot. Fig. 5 shows the most important parts marked with yellow boxes. These elements are as follows:

1. **Current Health Points (HP)**, and the **Heat Damage**. Players start with 9000 HP split into three colored health bars displayed above each other. Characters can restore the Heat Damage by dodging. When the HP reaches zero, the player loses the round.

2. **Hyper meter gauge** that visually indicates how many hyper attacks or shields the player can use. These bars can show half values to account for short-duration shields that cost half a bar.

3. **Heat gauge** with values ranging from 0% to 300%. Higher values scale the damage player gets, so players usually try to keep the heat low.

4. **Time** left before the end of the round that typically lasts 180 seconds.

The plan is to separate the computer vision module into two submodules: detecting objects on the arena and parsing the top UI.

Chapter 4

Implementation

This chapter explains implementation details, such as used libraries, applied techniques, and other decisions.

To begin with, the project required using Anaconda [19] to create a reproducible Python environment. This approach has two benefits. First, sharing and re-creating environments becomes as simple as sending a file and running a single command in the terminal. Second, Anaconda provides different channels where users can easily find and download compiled binaries. Without using these channels, the developers would have to manually compile libraries that Pip – the default package manager – cannot install. Fortunately, only PyTorch [20] and `tesserocr` [21] required working with Anaconda, and Pip installed the remaining libraries.

I Game modifications

The game has uneven animated backgrounds that may decrease object detection accuracy. Moreover, player attacks can produce different particles. Finally, the UI has some simple animations as well. Therefore, the game was modified

using the community-made tool [22] to decrypt all textures.

The following changes helped to improve performance. First, arena backgrounds turned into solid black color. Second, some particles became transparent, which effectively disables them. These modifications should reduce the background noise and improve the accuracy of the computer vision model.

II Windows API for interactions with the game

The game has native support only for Windows OS. Thus, one has to use Windows API (WinAPI) to interact with the game. The `pywin32` [23] library provides access to WinAPI from the Python code.

Any interaction with the windows requires the Window Handle – a unique identifier for each window. The `FindWindow` function finds a window by name and returns its handle. Since one process can only own one unique handle, one does not need to call `FindWindow` more than once.

A. Input Simulation

Initially, the proposed solution was to send simulated keystrokes to the window using WinAPI. In simple cases, this method can work even with background windows.

Unfortunately, this solution did not work with AoS2 because the game ignores simulated keyboard events. This issue appears since AoS2 uses DirectInput – a different input method provided by DirectX.

The `pydirectinput_rgx` [24] library can simulate keystrokes recognized by DirectX. However, the game window recognizes events only if it stays in the foreground. Thus, sending keystrokes to the background windows is impossi-

ble with DirectInput.

Input simulation works as follows:

1. On each step, the bot sends an array of binary values to the `environment.step()`.
2. The environment class converts binary values to the list of triggered in-game controls. All in-game actions exist in the code as Enumeration constants.
3. The list of triggered in-game actions maps to a list of buttons to press. Users can configure keybindings using `.env` file loaded once the bot starts.
4. `pydirectinput` releases all previously pressed keys by sending `KEY_UP` events, and simulates new keystrokes with `KEY_DOWN` events.

Note that this way of input simulation is global for the entire OS, i.e., any foreground window can receive keystrokes and react to them. Switching to other windows may cause irreversible damage to the user's files or system. Therefore, the input simulation should work only when the AoS2 window is active. WinAPI provides the `SetWinEventHook` function that accepts an event and a callback function as a parameter.

Binding a specific callback to `EVENT_SYSTEM_FOREGROUND` enabled the bot to react to active window change. The callback function checks if the new foreground window has the same handle as the AoS2 window. The bot disables input simulation and releases all buttons if handles are different.

However, user safety comes with an overhead. Since the bot had to listen for updates in blocking mode, that job moved to a separate daemon thread.

B. Screen Capture

Since Computer Vision requires images, one should extract rendered game frames and feed them to the computer vision module in real time. The following steps explain how the screen capture works:

1. Create a Device Context [25] from the handle. It contains the necessary information to draw the window on the screen.
2. Create an empty Bitmap image.
3. Blit (copy) raw frame from device context to the Bitmap.
4. Clean up the Device Context.
5. Drop the alpha channel from the frame and return it.

That procedure can capture the game at native 60 FPS, even though the process is not optimal due to creating new objects on each call. Moreover, OS Windows updates only these parts of the window that are visible. Therefore, the game window must not clip with physical screen borders.

III Computer Vision module

The Computer Vision module consisted of two classes: `InterfaceData` that processes the top UI, and `ArenaData` that detects objects on the arena.

A. Reading the top UI

Various Computer Vision techniques can extract information from the top UI. However, most UI elements were static, so they did not require much extra

work. The most problematic items, however, were heat, health, and time values that required reading the text on the image.

Tesseract OCR [26] is an open source Optical Character Recognition (OCR) tool. The `Python-tesseract` [27] library allows users to call Tesseract OCR from Python code. However, this library has a drawback. Since `Python-tesseract` is a wrapper around a command-line tool, its performance slows dramatically due to inter-process communication and library initialization on each call.

In contrast, the `tesserocr` [21] library replaced `Python-tesseract` in the current project. In addition to the capabilities described above, `tesserocr` provides a raw Tesseract API that can be initialized once and reused during the entire program runtime. That feature made this library more suitable for real-time OCR.

Recognizing the health values required the least preprocessing: grayscaling, cropping, and thresholding were enough because the text never changed color and always had an opaque background. The final step was to expand the post-processed area in all directions by a fixed amount of pixels.

In addition, Fig. 6 shows the difference between fonts of the health text. The game supports multiple languages that change fonts. However, Tesseract failed to recognize some characters with English font due to unusual and disproportionate looks of numbers, especially the number five. Glyphs in the Japanese font are too small to detect characters reliably. Thus the remaining Russian font[1] is the most optimal solution for digits recognition.

Extracting the heat was more challenging because the text had a transparent background. Projectiles and other moving parts behind the text resulted

[1] I translated this game into Russian in 2021. Replacing the Nationalyze font was necessary for readability.

(a) Unprocessed fonts in the game (b) The same image after pre-processing

Fig. 6. English, Japanese, and Russian fonts

in lower precision. Moreover, the heat text gradually becomes red when values go above 150%. Thus, the process required extra preprocessing steps. The frame went through grayscaling, cropping, denoising, and thresholding. The `cv2.findContours` function then detected contours without hierarchy. These contours with small areas were likely to represent digits. Thus, the pipeline filled these small contours to create a mask. Finally, Tesseract processed the masked image that only had numbers.

Original Tesseract OCR has a character whitelist option. Unfortunately, the latest version of `tesserocr` ignores that option, which causes the library to detect characters incorrectly. Thus, each piece of recognized text goes through a simple error correction. The algorithm is to replace the letter `s` with the number five and the letter `o` with the number zero. Finally, the text converts to a positive 32-bit number or `-1` in case of failure.

Finally, the last step was reading icons on the top UI. Detecting them is unnecessary because they are static and opaque. Thus, the solution was to read individual pixels and compare their color to the list of allowed values. For example, the heat gauge consists of 6 half-circles. By examining the color of each half-circle, the CV module will understand how many hyper attacks or shields the bot can use.

Since the Computer Vision field has no strict guidelines, these pipelines were created by experimenting with different approaches. The current solution fitted

the project better because it detected more numbers on average. However, it may not be the most optimal solution.

B. Detecting objects on the arena

The most challenging part of the project was multiple object detection in the arena.

The initial idea was to detect attack indicators using `cv2.findContours`. After that, simple template matching would decide the object type. However, this approach would be inefficient due to slow template matching.

Another solution would be to train a cascade classifier because it can efficiently detect multiple objects. However, the drawback of cascade classifiers is that they struggle to identify objects of different types.

The first solution is inconvenient and unreliable. The second method requires too much extra work. Thus, the most fitting solution was YOLOv5 [28] classifier because it can efficiently detect multiple objects of different types.

By default, YOLO is pre-trained to work with real-life objects, so it is not ready for video games by default. Initial testing showed that it confused a rocket with a kite [2]. Therefore, it required extra training on the custom data.

A simple screenshot-gathering demo module acquired the training data. This module captured game screenshots every second and saved them to the disk. This module recorded one game round against a built-in computer opponent. The first human player did not attack and only dodged projectiles. The second player was a built-in bot that played a character without melee attacks and used as many projectiles as possible. The resulting dataset consisted of 200 images. Training consumed 75% of images. The validation used 15%. Finally, the remaining 10%

[2] `https://imgur.com/a/S3MJ3El`

went to the testing set.

The next step was to label the data using the *Make Sense* [29] website in the browser. Fig. 7 shows an example of labeled image, although *Make Sense* does not render the text. As seen on Fig. 7, many projectiles can overlap.

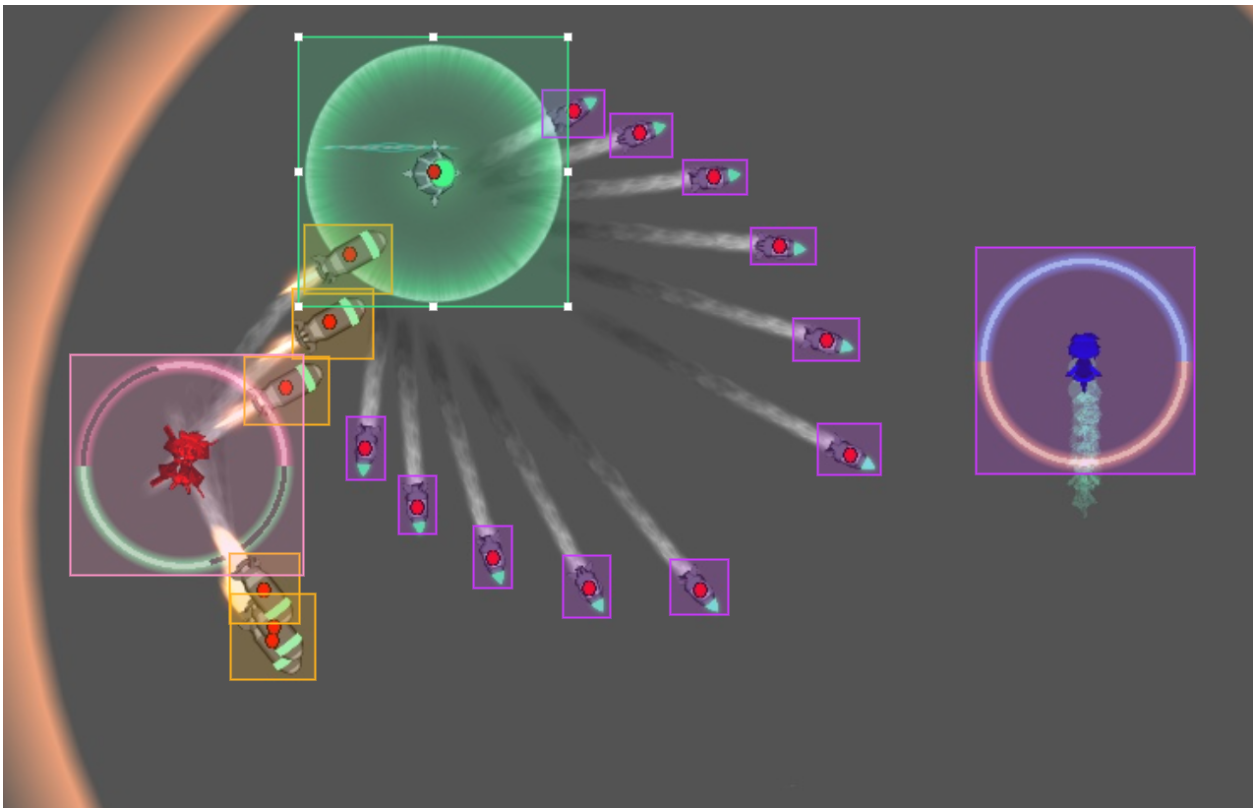
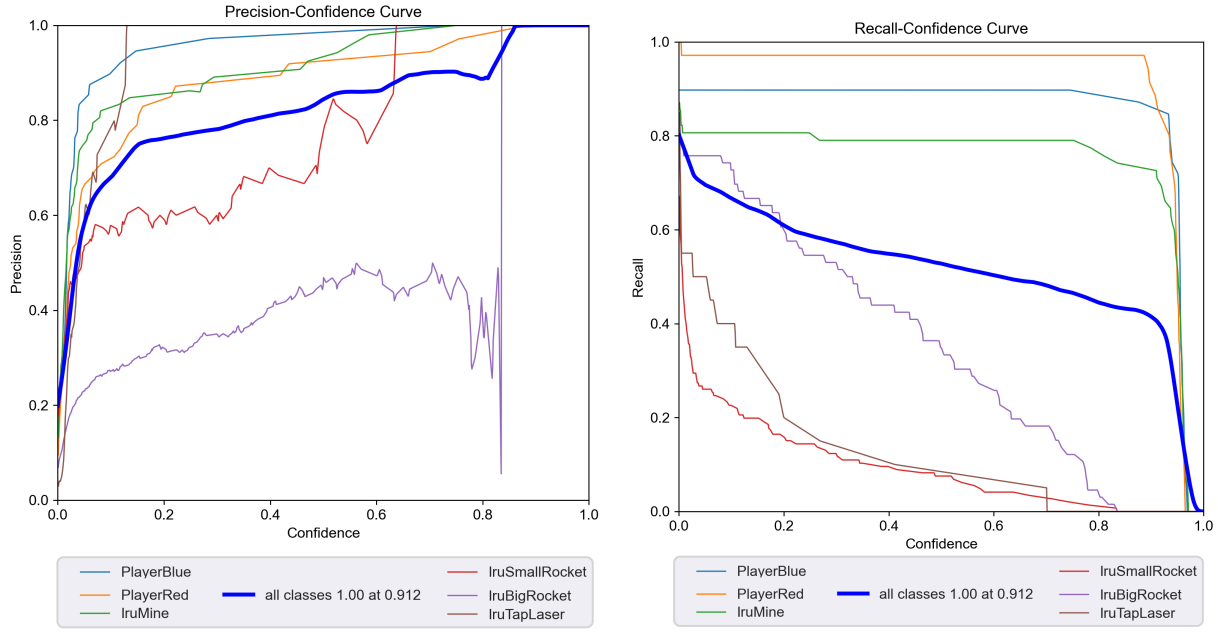


Fig. 7. Labeling images in *Make Sense* app

The dataset used only six object classes – two to identify players and four to define all the projectiles. Class identifiers ranged from 0 to 5. Thus, each label has a corresponding Class ID

YOLO trained on a separate virtual machine for 240 epochs for 12 hours. Fig. 8 shows the model validation results performed after training. Fig. 8(a) shows that at a confidence threshold above 0.85, the precision reached the maximum value of 1.0. On the other hand, Fig. 8(b) demonstrates that the recall dropped to zero for some classes starting at around 0.7 confidence threshold.



(a) Precision-confidence curve (b) Recall-confidence curve

Fig. 8. Results of the model validation

Fig. 9 shows the confusion matrix obtained after validation. The model successfully detected players in over 90% of the cases. However, it struggled to detect laser attacks and both types of rockets. This issue occurred due to various reasons. First, the dataset was small, although YOLOv5 documentation recommends [30] having more than a thousand images per class. Second, the dataset was imbalanced – small rockets were a prevalent class, while lasers were the least represented class. Finally, explosions and trails introduced extra noise in the training dataset. Removing these effects could improve the results.

The trained model had a `.pt` format compatible with the PyTorch [20] library. Compared to other alternatives, this library fitted the project best because it provided a simple and easy-to-use interface for loading models and running inference. Instead of the most recent version 2, the project used PyTorch version 1.13.1 because newer versions were incompatible with other packages.

List. 1 shows the code snippet that downloads the model from the PyTorch

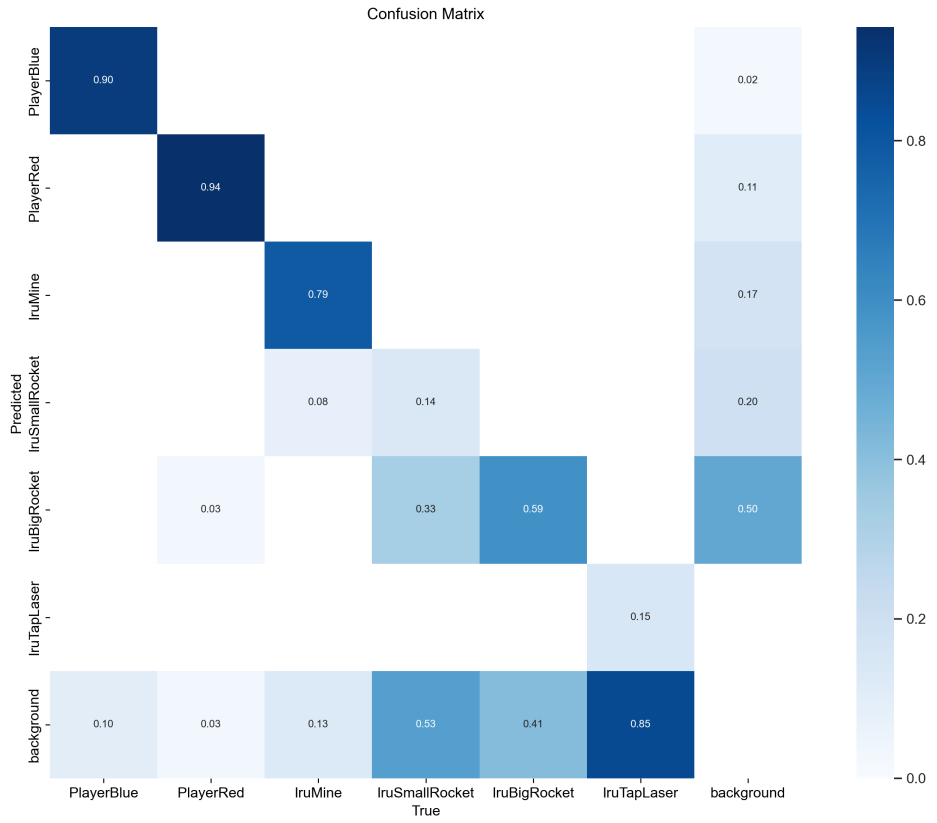


Fig. 9. Confusion matrix

hub and configures it to have custom weights. The `model.classes` field will have a dictionary with classes used during training.

```
yolo_model: torch.nn.Module = torch.hub.load(
    "ultralytics/yolov5",
    "custom",
    path="weights.pt"
)
```

List. 1. Loading the PyTorch model

Feeding the frame to the model by calling `yolo_model(rgb_image)` returns a tensor of tensors. This format is equivalent to a list of detected objects. Each detection has the following structure:

$$x1, y1, x2, y2, confidence, label$$

Numbers $x1, y1, x2, y2$ describe a bounding box of the detected object. The coordinates of the bounding box cannot exceed the screen resolution and cannot be negative. The $confidence$ value shows the probability that the detected object corresponds to the $label$ type. The $confidence$ values range from 0.0 to 1.0, which allows filtering only the most relevant detections. Finally, the $label$ is a class ID.

The demo Object Detection module proved that the model detected objects in real time. Along with inference, it drew bounding boxes and labels on the screen.

Nevertheless, the demo showed mixed results. The inference could process 7 FPS while running on CPU only. On the other hand, the detection accuracy was not astonishing. The confidence threshold was 0.6, i.e., the detection was successful for confidence values above the threshold. Fig. 10 shows the best-case scenario when the model successfully detected many small objects. However, in most cases, the model failed to detect projectiles as seen on Fig. 11. Additionally, object detection failed in more complex scenarios, for instance, when sprites overlapped.

Despite the poor performance and inaccurate results, the Computer Vision module became a part of the bot pipeline. The benefit is that interfaces of `ArenaData` and `InterfaceData` are designed so that the inner working of the system do not matter to the consumer - the `Environment` class. Tuning and replacing the model should not be troublesome, although the process is tedious.

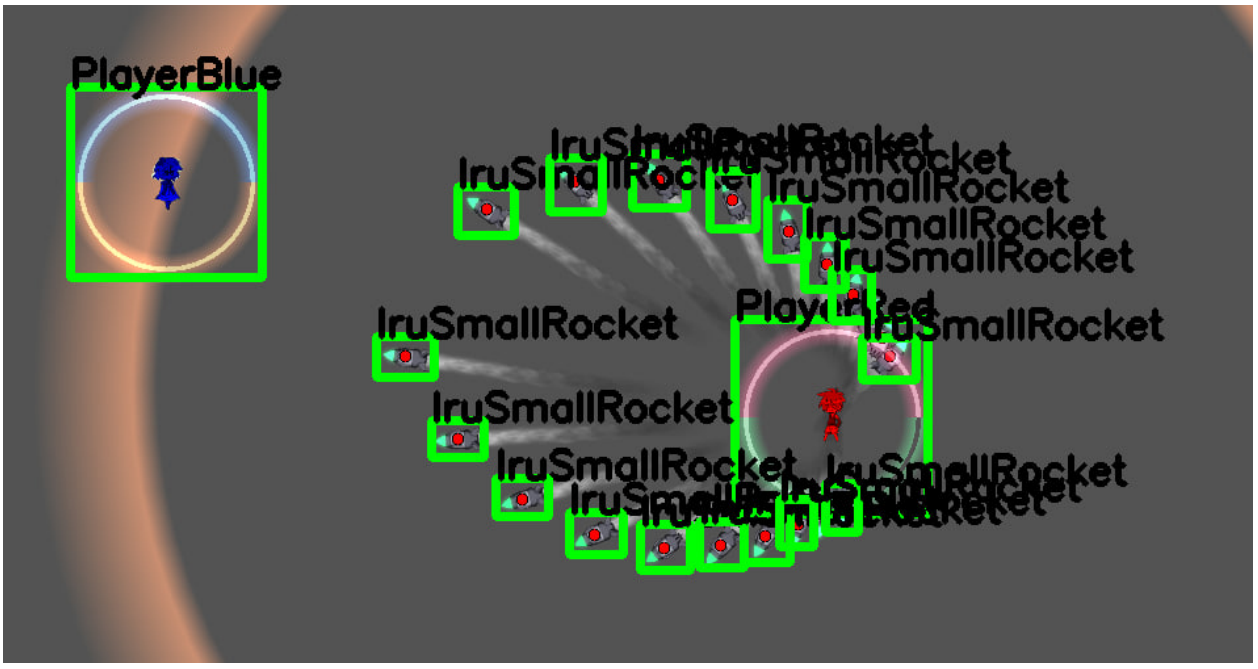


Fig. 10. Most projectiles successfully detected

IV Reinforcement Learning training routine

Poor results of the Computer Vision module caused the RL agent training to fail inherently. Since Reinforcement Learning is complex, every detail may affect the training results. The following factors would likely hinder the process. First, the object detection ran at a slow framerate, causing the training time to increase drastically. Second, the object detection had poor accuracy, which might result in learning the incorrect behavior. Finally, the bot could not detect complex scenarios during the game rounds, which resulted in a simplistic reward function.

Even though training the agent would not result in satisfactory results, one can still develop the training system to account for possible future improvements.

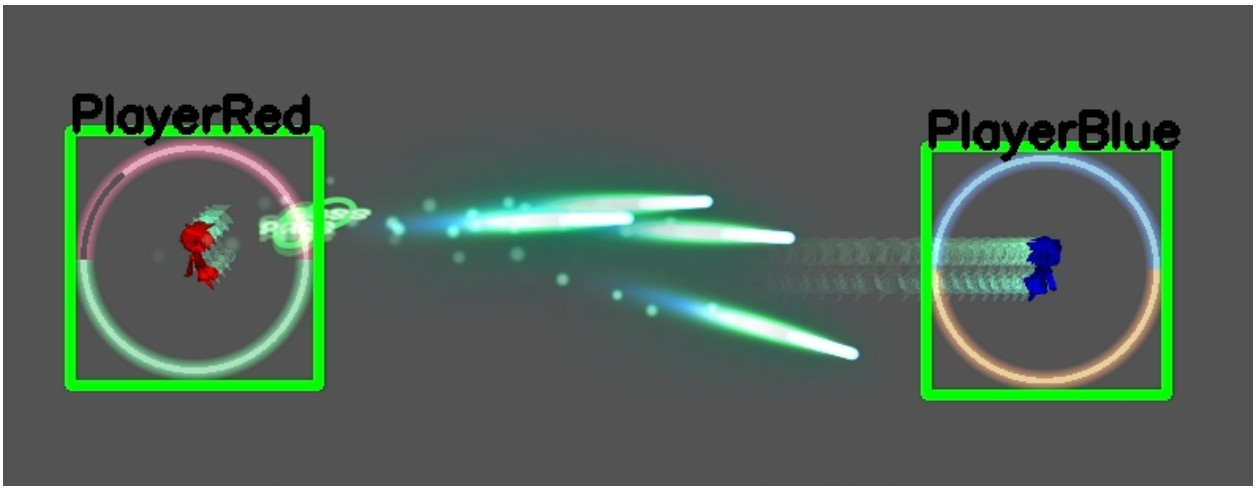


Fig. 11. Detections that fell below confidence threshold

A. Environment

The OpenAI Gym [31] library provides an interface for any RL environment implementation. The interface has the following methods.

- `reset()` - restarts the simulation and returns the first observation.
- `step(action)` - applies the agent action and returns the new observation and reward.
- `render()` - draws the current state of the environment on screen

In addition to these methods, the `Environment` class must define the following fields.

- `action_space` - the format of the action sample.
- `observation_space` - the format of what the agent can observe.

For the AoS2 bot, the action space was a Multibinary space, i.e., a list of binary values. Each value of this list represented the corresponding in-game action,

such as moving down or attacking. Since these actions can trigger simultaneously, the Multibinary space is the best solution.

However, building an observation space was a more challenging task. Regarding the top UI, the current system considered health, heat, time, and meter gauge values. Even though these attributes did not cover everything, the main problem still was object detection in the arena.

The current system detects the on-screen positions of players and projectiles. However, the system did not account for rotations and directions in which objects moved. In addition, the on-screen coordinates do not represent the actual position in the circular arena. Thus, mapping the on-screen coordinates to the arena space might improve the bot's gameplay performance.

For example, player positioning is important because staying too close to the wall limits movement options. This strategy is often fatal in evasion-based games. One could detect the arena border using circle arc detection and calculate its radius. After that, converting screen coordinates would yield the object's position inside the circle. Unfortunately, the arena border is not always visible, and characters have different movement speeds. Thus, coordinates mapping is a challenge worth additional research.

Nevertheless, all the gathered values formed a `Dictionary` observation space with the structure shown on Tab. I. Integer values can represent most player attributes. Positions have `Vector2` type - a tuple of two numbers that form a `Box` space. The `objects` key is the list of detected objects. The `Object` type is equivalent to `(ClassID, Vector2)` tuple. Each observation required flattening from key-value storage to the linear array, preserving the order of values.

TABLE I
Observation Space Structure

Key	Type	Value Range
`p1_health`	`int`	0-9000
`p2_health`	`int`	0-9000
`p1_red_health`	`int`	0-3000
`p2_red_health`	`int`	0-3000
`p1_heat`	`int`	0-300
`p2_heat`	`int`	0-300
`p1_half_meter`	`int`	0-6
`p2_half_meter`	`int`	0-6
`time_left`	`int`	180
`p1_position`	`Vector2`	(0, 0) - (1366, 768)
`p2_position`	`Vector2`	(0, 0) - (1366, 768)
`n_objects`	`int`	$\geqslant 0$
`objects`	`list[Object]`	...

B. Reward function

The bot could not detect complex scenarios, and the goal was to make a bot that only evades. Tab. II shows a placeholder reward function used for debugging. With the evolution of the Computer Vision module, the reward function should improve to encourage the agent to stay far from arena borders, use some attacks at the right time or distance, dodge specific projectiles, etc.

TABLE II
Simple reward function

Event	Reward
Lose N health points	$-N$
Stay close to projectile without getting hit	5

C. Problems with training

No universal approach to training exists. According to the Stable Baselines documentation [32], the choice depends on action space and the type of the task. The bot should actively learn from its actions during the gameplay. Thus, the on-policy algorithms such as Proximal Policy Optimization (PPO) seem fitting. Moreover, PPO supports the Multibinary action space used in the current implementation.

Nevertheless, the bot might need LSTM, and the PPO algorithm supports `MlpLstmPolicy` - Multi-Layer Perception with LSTM. Most projectiles have a specific behavior and time-to-live on the arena from 1 to 10 seconds. Learning their behavior teaches players to attack and evade more effectively. Therefore, the bot might need some short-term memory to aid with learning projectile behaviors.

Unfortunately, running the training routine comes with another set of challenges. First, the game can run only a few matches in a row. After the set finishes, the game returns to the menu. Thus, the bot has to understand match end conditions and automatically choose characters again to restart training. In addition, the bot has to account for a break between rounds and the round start delay.

Fig. 12 shows how the screen looks when the round starts. During this time, players can only move and cannot shoot projectiles. Thus, players should position themselves to stay safe from surprise attacks when the active phase starts. Besides,

the UI elements may cover the character sprites, which hinders object detection accuracy.

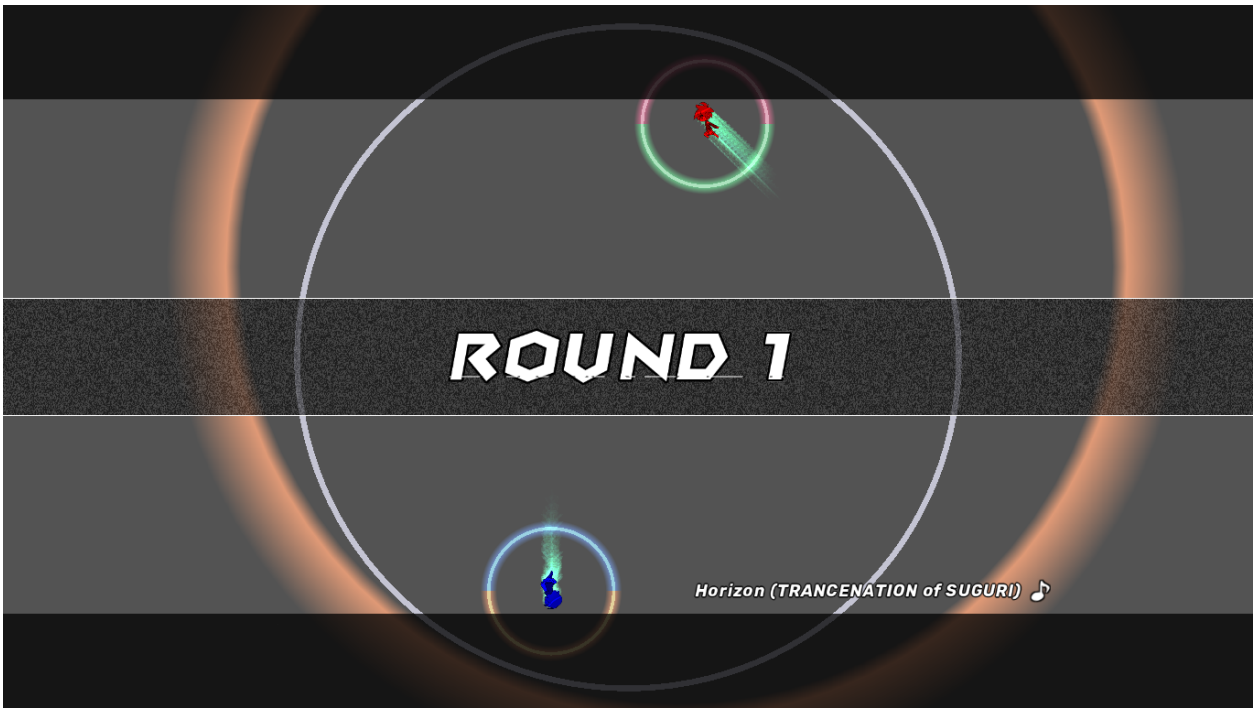


Fig. 12. Round start screen

Chapter 5

Results and Discussion

This chapter analyzes the project results, points out currently unsolved problems, and suggests possible improvements in future works.

Fortunately, the project was able to fulfill its minimal goal as a proof of concept. However, the result was still far from perfect due to numerous reasons. In short, the project was too ambitious for a single inexperienced developer. In addition, some parts of the work were complex enough to be worth an additional research topic. One could be tuning a YOLO classifier for optimal real-time performance with video game sprites. Another would be training the agent for a complex evasion-based 2D fighting game.

Fortunately, the project was successful from the developer's point of view. The system consisted of isolated modules that provided high-level access to their interfaces. Moreover, each component returned only high-level data models with clearly defined structures and types. Finally, the system at the highest level is universal enough to apply to most projects from the same scope.

I Computer Vision is the key

The computer vision model caused a significant performance drop even before introducing RL agent training. As a result, the bot was working at around 5 FPS out of the expected 20 FPS. In addition, the accuracy was not high enough to launch the training routine. Several factors resulted in such a poor performance.

The most obvious issue is the dataset size and quality. First, 200 images in total is a small dataset, which made small object detection inaccurate. Second, the confusion matrix shown in Fig. 9 demonstrates that the dataset was not uniform, i.e., some object classes appear more often than others. Thus, the model might filter out less frequent objects as noise. Finally, images in the dataset had extra noise, such as explosions and trails. Removing the noise, expanding the dataset, and balancing the classes would significantly improve the detection accuracy, although this task is daunting.

The bottleneck could occur because of inferior hardware. The bot was running on a slightly outdated processor with hardware bugs causing it to limit the tick rate to 540 MHz[1]. Newer hardware could potentially improve the computational speed.

Besides, the bot was running inference on the CPU only. Therefore, introducing the GPU calculations might increase the computational performance of object detection and some functions in the OpenCV library. However, this small addition required installing NVIDIA CUDA Toolkit [33] compatible with the installed GPU and libraries compatible with the installed CUDA Toolkit version. This process often breaks dependencies and creates invalid environments that Anaconda cannot reproduce.

[1] https://imgur.com/a/O9c6qn5

The alternative solution would be to re-write the project in C++. It would improve the overall performance of the system and libraries. On the other hand, it would still require careful version management.

Ultimately, the Computer Vision module is a questionable design decision for game bots based on Reinforcement Learning. Even though one can build a system where inference is fast and accurate enough, the bot's capabilities will still be limited by which scenarios the bot can detect. Thus, this project proved why Reinforcement Learning bots rarely involve Computer Vision for gathering observations. However, applying these methods seemed the only solution since developers did not design the game for such projects.

II Slow training

This project proved that gathering observations is possible using Computer Vision. However, several issues caused the bot training to fail.

First, Computer Vision performance defines the training speed. A single `environment.step()` call performed quite heavy computations, which caused it to become a bottleneck. Therefore, the entire system slowed down to `environment.step()` rate, which would cause the training efficiency to plummet.

Unfortunately, the inference speed was not the only issue. Unreliable object detection and a lack of complex scenario detection made such a bot practically useless. A possible solution is to replace a single CV Model with a more advanced pipeline that processes the inference results. This pipeline might include filters that detect different events, such as dodging specific attacks, getting hit by various projectiles, and other scenarios. These events would affect the reward function

and allow for fine-tuning the bot's behavior.

Another factor to note is the game framerate. AoS2 cannot run at more than 60 FPS. Thus, the environment would not perform more than 60 steps per second even if the bot had an ideal performance. This issue occurs because the game cannot run simulations in headless mode – i.e., without displaying the graphics. Thus, one might have to use transfer learning to overcome the framerate cap problem.

Transfer learning allows one to train a bot on a similar game and then fine-tune it for the original. For example, one could implement a game clone in Unity with headless mode support that runs more steps per second. While transfer learning is a sounding idea, one has to address several challenges. One of the possible challenges is to carefully re-implement the game mechanics. Another challenge is data consistency between games, such as player positions or attack speed. Finally, the bot would still have to detect complex scenarios in the original game.

In short, even though different approaches can optimize bot training, its efficiency is limited by the Computer Vision Model.

Chapter 6

Conclusion

The final project [34] fulfilled its minimal goal as a proof of concept. Even though the general system design, code quality, and some minor milestones were quite successful, the overall result was not because the bot did not serve its purpose. However, the project was too demanding for one inexperienced developer. The result would improve with a bigger team of more experienced developers. Finally, inference performance and complex behaviors detection are concerns that will improve the overall bot quality.

Bibliography Cited

[1] S. García-Lanzo, I. Bonilla, and A. Chamarro, "The psychological aspects of electronic sports: Tips for sports psychologists," eng, *Int. J. Sport Psychol.*, no. 51, pp. 613–625, 2020, ISSN: 0047-0767. DOI: 10.7352/IJSP.2020.51.613. [Online]. Available: https://doi.org/10.7352/IJSP.2020.51.613.

[2] Orange_Juice, *Acceleration of SUGURI 2*, [Video Game for PC], Mar. 2018. [Online]. Available: https://store.steampowered.com/app/390710/Acceleration_of_SUGURI_2/.

[3] T. M. Mitchell, *Machine Learning* (McGraw-Hill series in computer science). New York: McGraw-Hill, 1997, ISBN: 978-0-07-042807-2.

[4] A. S. Lampropoulos and G. A. Tsihrintzis, *Machine Learning Paradigms* (Intelligent Systems Reference Library). Cham: Springer International Publishing, 2015, vol. 92, ISBN: 978-3-319-19135-5. DOI: 10.1007/978-3-319-19135-5. [Online]. Available: http://link.springer.com/10.1007/978-3-319-19135-5.

[5] B. Yegnanarayana, *Artificial neural networks*. PHI Learning Pvt. Ltd., 2009.

[6] L. Deng, "Deep Learning: Methods and Applications," en, *Found. Trends® Signal Process.*, vol. 7, no. 3-4, pp. 197–387, 2014, ISSN: 1932-8346,

1932-8354. DOI: 10.1561/2000000039. [Online]. Available: http://nowpublishers.com/articles/foundations-and-trends-in-signal-processing/SIG-039.

[7] R. Brunelli, *Template matching techniques in computer vision: theory and practice*. Chichester, U.K: Wiley, 2009, OCLC: ocn289008992, ISBN: 978-0-470-51706-2.

[8] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *CVPR 2001*, vol. 1, Kauai, HI, USA: IEEE Comput. Soc, 2001, pp. I–511–I–518, ISBN: 978-0-7695-1272-3. DOI: 10.1109/CVPR.2001.990517. [Online]. Available: http://ieeexplore.ieee.org/document/990517/.

[9] OpenCV. "Cascade classifier training." Accessed: Feb. 14, 2023. (), [Online]. Available: https://docs.opencv.org/4.5.5/dc/d88/tutorial_traincascade.html.

[10] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," 2015, Publisher: arXiv Version Number: 5. DOI: 10.48550/ARXIV.1506.02640. [Online]. Available: https://arxiv.org/abs/1506.02640.

[11] K. Shao, Z. Tang, Y. Zhu, N. Li, and D. Zhao, "A Survey of Deep Reinforcement Learning in Video Games," 2019, Publisher: arXiv Version Number: 2. DOI: 10.48550/ARXIV.1912.10944. [Online]. Available: https://arxiv.org/abs/1912.10944.

[12] G. Lample and D. S. Chaplot, "Playing FPS Games with Deep Reinforcement Learning," 2016, Publisher: arXiv Version Number: 2. DOI:

10.48550/ARXIV.1609.05521. [Online]. Available: https://arxiv.org/abs/1609.05521. Accessed: Oct. 20, 2022.

[13] I. Oh, S. Rho, S. Moon, S. Son, H. Lee, and J. Chung, "Creating Pro-Level AI for a Real-Time Fighting Game Using Deep Reinforcement Learning," 2019, Publisher: arXiv Version Number: 3. DOI: 10.48550/ARXIV.1904.03821. [Online]. Available: https://arxiv.org/abs/1904.03821. Accessed: Sep. 12, 2022.

[14] OpenAI, : C. Berner, *et al.*, *Dota 2 with large scale deep reinforcement learning*, 2019. DOI: 10.48550/ARXIV.1912.06680. [Online]. Available: https://arxiv.org/abs/1912.06680.

[15] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, pp. 1735–80, Dec. 1997. DOI: 10.1162/neco.1997.9.8.1735.

[16] V. Mnih, K. Kavukcuoglu, D. Silver, *et al.*, *Playing Atari with Deep Reinforcement Learning*, 2013. DOI: 10.48550/ARXIV.1312.5602. [Online]. Available: https://arxiv.org/abs/1312.5602.

[17] *PyAutoGUI*, Accessed: Mar. 3, 2023. [Online]. Available: https://pypi.org/project/PyAutoGUI/.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley Professional Computing Series). Pearson Education, 1994, ISBN: 9780321700698. [Online]. Available: https://books.google.bg/books?id=6oHuKQe3TjQC.

[19] A. Inc., *Anaconda - Where packages, notebooks, projects and environments are shared*, Accessed: Mar. 3, 2023. [Online]. Available: `https://anaconda.org/`.

[20] T. L. Foundation, *Pytorch*, Accessed: Mar. 3, 2023. [Online]. Available: `https://pytorch.org/`.

[21] *A Python wrapper for the tesseract-ocr API*, Accessed: Mar. 3, 2023. [Online]. Available: `https://github.com/sirfz/tesserocr`.

[22] M. Jarjour, *Acceleration of Suguri 2 Data Ripper*, Accessed: Feb. 15, 2023. [Online]. Available: `https://github.com/MergeCommits/aos2ripper`.

[23] *Python for Windows (pywin32) Extensions*, Accessed: Mar. 3, 2023. [Online]. Available: `https://pypi.org/project/pywin32/`.

[24] *Python mouse and keyboard input automation for Windows using Direct Input*, Accessed: Mar. 3, 2023. [Online]. Available: `https://github.com/reggx/pydirectinput_rgx`.

[25] Microsoft, *Device Contexts*, Accessed: Mar. 3, 2023. [Online]. Available: `https://learn.microsoft.com/en-us/cpp/mfc/device-contexts?view=msvc-170`.

[26] *Tesseract Open Source OCR Engine (main repository)*, Accessed: Mar. 3, 2023. [Online]. Available: `https://github.com/tesseract-ocr/tesseract`.

[27] *A Python wrapper for Google Tesseract*, Accessed: Mar. 3, 2023. [Online]. Available: `https://pypi.org/project/pytesseract/`.

[28] Ultralytics, *YOLOv5*, Accessed: Feb. 14, 2023. [Online]. Available: https://github.com/ultralytics/yolov5.

[29] P. Skalski, *Make Sense*, Accessed: Mar. 3, 2023. [Online]. Available: https://skalskip.github.io/make-sense/.

[30] Ultralytics, *Tips for Best Training Results*, Accessed: Apr. 5, 2023. [Online]. Available: https://docs.ultralytics.com/yolov5/tutorials/tips_for_best_training_results/.

[31] G. Brockman, V. Cheung, L. Pettersson, *et al.*, "OpenAI Gym," 2016. DOI: 10.48550/ARXIV.1606.01540. [Online]. Available: https://arxiv.org/abs/1606.01540.

[32] Stable Baselines3, *RL Algorithms*, Accessed: Apr. 27, 2023. [Online]. Available: https://stable-baselines3.readthedocs.io/en/master/guide/algos.html.

[33] NVIDIA Corporation, *CUDA Toolkit*, 2022. [Online]. Available: https://developer.nvidia.com/cuda-toolkit.

[34] *AoS2 Bot*, Accessed: June 22, 2023. [Online]. Available: https://github.com/DOCtorActoAntohich/aos2-bot.