eman ta zabal zazu

Universidad del País Vasco / Euskal Herriko Unibertsitatea

INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Bachelor Thesis

Bachelor Degree in Computer Engineering

Computer Science

Object detection with one-shot Convolutional Neural Networks for playing the game Cuphead

Julen Indias Garcia

Supervisor
Roberto Santana Hermida

June 21, 2023

Abstract

The objective of this work was to develop an artificial intelligence (AI) approach for playing the game Cuphead using deep reinforcement learning (DRL) and object detection algorithms. Both DRL and object detection have been widely studied in recent years, and different algorithms and models have been developed. The two main tasks that were covered in this project were the object recognition within the game using convolutional neural networks (CNN) and making decisions and taking actions in the game based on those detections. Three different types of CNNs were tested with images extracted from the game before starting to work on the DRL part: FasterRCNN+InceptionResNet and SSD+MobileNet built in TensorFlow and YOLOv5 built in PyTorch. After analyzing the performance obtained with those images and a gameplay video of the game, YOLO was chosen as the final model to be trained for the object detection job.

After choosing YOLO as the network to be used to carry out the detections in the game, a method was defined for getting labeled data from the game without having to manually annotate images using the pre-trained YOLO network. Images were collected simultaneously while playing the game and then, those images were processed for getting annotated data for finally using them for training the model in order to have more precise detections. Finally, a set of human-based rules have been defined for playing the game based on the detections.

The object detection module, the human-based rules module, and a module responsible to execute the commands as part of the game, were integrated and tested by playing 10 different games. The results showed that the AI-based agent was able to win in 2 out of the 10 games that were played.

Contents

List of Figures

List of Tables

List of Algorithms

CHAPTER 1

Introduction

1.1 The video game industry and applications of AI in video games

The video game and gaming industries have grown since the first video games were introduced in the 1970s, Pong (1972) and Space Invaders(1978) are a few examples of these games. From the first games to the release of home and portable consoles like the Nintendo Entertainment System (NES) released in 1983 or the Game Boy (1989) developed by Nintendo to the release of modern games such as Minecraft (2011) or The Last of Us (2013), videogames have grown considerably and have gained more and more popularity over the years. Classic games like the well-known The Legend of Zelda: Ocarina of Time (1998) and the first shooter game ever made, Doom (1993), have endured over the years along with video games produced and distributed by the most powerful video game companies, which typically have higher development and marketing budgets than other smaller games, these are known as triple-A games. The Call of Duty saga, first released in 2003 and whose latest release can be found in 2022 developed by the Activision company, is an example of a triple-A video game saga. Also, one of the most important fields in the video game industry are the electronic sports, the so-called e-sports. Many competitive-oriented games such as League of Legends (2009), Dota 2 (2013), Overwatch (2016) or Valorant (2020) have competitive leagues where professional teams from around the world compete with each other.

With the exponential growth of the applications of AI and virtual reality (VR), the release of more complex and revolutionary games has increased from the 2010s to this day and so has done the application of AI in this field. Some of the key applications of AI in terms of gameplay and game design are none playable characters (NPC), which create in-game characters that perform similar to how a human player would. A few examples of games that implement complex NPCs in game are Bioshock Infinite (2013) or Red Dead Redemption 2 (2018) [1]. Besides, AI can be also used for map generation and loading [2] where it can predict the location of a certain block in future frames. Finally, level testing [3] is also a practice that benefits from AI techniques and can be automated.

AI algorithms and machine learning (ML) models can benefit from being tested and developed using video games and even compared or overcome results with the ones achieved by human players. AI approaches for playing games can be found for titles like Dota 2 [4], the above mentioned Pong, Starcraft II, and more.

1.2 The Cuphead game

Cuphead is a 2D side-scrolling action game published in 2017 and developed by StudioMDHR. Inspired by the 1930s animation style, the game is divided into levels and the goal is to defeat all enemies called *bosses* in all levels until we reach a final enemy to finally complete the game. The game has a very unique and special aesthetic, featuring hand-drawn visuals and using animation techniques popular in the 1930s. In addition to this, the game has a jazzy soundtrack that completes the environment in where the player is involved while playing.

Cuphead was chosen among other games for being a 2D game, which could help developing all the project modules rather than if a 3D game was chosen. Other games considered for this project where The Binding of Isaac Rebirth (2014) and Minecraft but due to the complexity of The Binding of Isaac Rebirth where each time we play the environment changes and all possible scenarios are randomized and Minecraft being a 3D game, Cuphead was selected.

1.3 Goals and organization of the project

As stated before, the goal of the project has been to develop an AI approach for defeating one of the bosses in the game using object detection and DRL techniques. For this purpose, we will be exploring some of the state-of-the-art algorithms in computer vision to compare them and use them in order to make the in-game detections that will serve as input for the decision making.

The following chapters are organized as follows: the management for this project done will be presented in the next chapter, then some background about the game is presented in Chapter 3. After this, all the state-of-the-art work related to the topics covered in the project will be discussed in Chapter 4, e.g., the use of AI in games and the use of CNNs for object recognition tasks. The game analysis that has been done is described in Chapter 5 before going through the object recognition and the decision making parts which are discussed in chapters 6 and 7, respectively. Finally, the analysis of the experiments and the results obtained are presented in Chapter 8 before diving into Chapter 9 where the conclusions are explained.

CHAPTER 2

Project Management

This chapter covers the planning and management processes during the development of the project. The following planning is the initial planning that had some changes during the development process as discussed later in the document.

2.1 Planning

Before starting to develop the project, some tasks were defined to structure the project. These tasks were divided into two groups: the ones related to the project implementation and those regarding the documentation.

The documentation tasks are fully related to the thesis writing and project management and the implementation group includes the following tasks and subtasks:

1. Analysis of the state-of-the-art
    - Identify and read papers on the use of CNNs
    - Study previous work on the use of ML for games
2. Game analysis
    - Understanding full game mechanics
    - Define the game setup (e.g. game window resolution)
    - Finding a frame processing library
    - Develop the annotated frame processing method
3. Object detection task
    - Analysis and testing of CNNs with images and videos
    - Re-training the model with the labeled data that was collected

4. Decisions based on detections
    - Find the best representation for the extracted data from the prediction frames
    - Determine a way to link each taken action with its frame
    - Decide on the best metric for the action scoring (e.g., life bar)
5. Keyboard interaction
    - Finding a library for the automatic keyboard interaction
    - Testing the library on the game without the ML part

To plan and organize the expected work for the two main task groups, a work breakdown structure (WBS) diagram and a Gantt diagram are shown in figures 2.1 and 2.2. These two diagrams include all the subtasks to be done in order to complete the initial objectives with enough time to add corrections if needed.

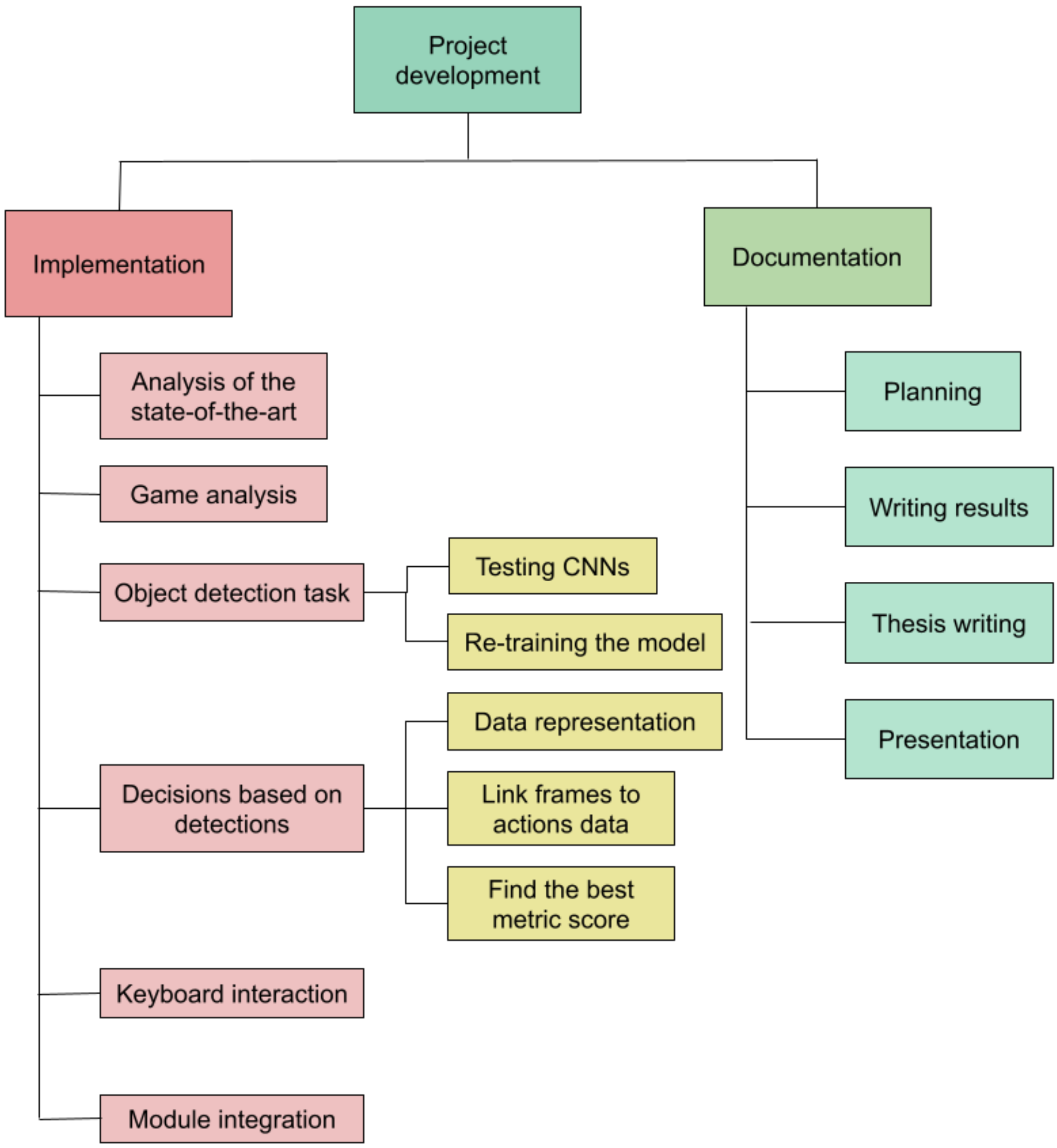
2.2 Risk prevention

To prevent future problems in the development of the project, a risk management task has been done in order to avoid facing these issues. The main risk in this project is losing data. Losing anything related to the repository that stores all the files of the project could cause a delay and finishing the project could be difficult in the time planned. Since all the work has been done in two PCs (a laptop and a desktop PC), three backup copies of the project repository have been done: the first one stored in the laptop, a second one stored in the desktop PC and the third one in an external hard drive. After every working session, all copies are updated.

Another risk to take into consideration is not having enough computational power to train and execute the models. Since the goal is to have simultaneously a game and other computationally expensive tasks running, two powerful GPU and CPU will be used to ensure that there are as less bottlenecks as possible. For the training phase, Google Colab will be employed as it provides powerful GPU and CPU support for running programs. After training any network using Google Colab, the file containing the final weights can be easily obtained for executing a network with custom weights for detecting the desired objects.

Finally, the module that is expected to take the longest to complete is the decision making related one. As explained before, the goal is to use DRL techniques for making decisions that will take actions for playing the game. In case of not having enough time for implementing this module using DRL techniques, simple human-based rules will be employed for playing the game. This would cause a less accurate system when it comes to defeating the enemy, but it would make possible finishing the project in time and focus the analysis and refinements in the object detection component.

Figure 2.1 Work Breakdown Structure diagram showing all tasks.



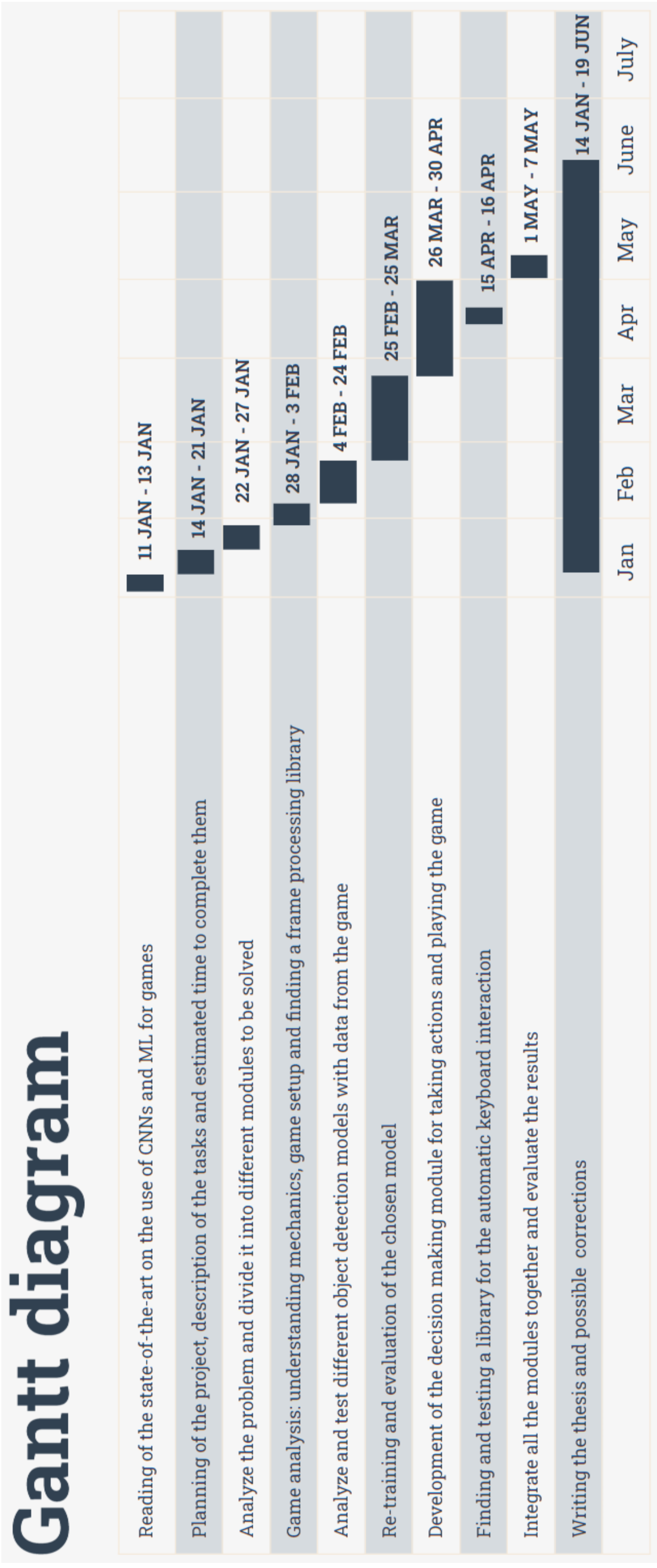


Figure 2.2: Gantt diagram showing the approximate completion dates for each task.

CHAPTER 3

Background

3.1 The YOLO algorithm

YOLO (You Only Look Once) is a popular real-time object detection algorithm that has changed the computer vision field and was firstly introduced in 2016 [5]. While other object detection methods require multiple passes in the image to give the final detections' bounding boxes coordinates, class labels and probabilities, YOLO differs from all the these models by only passing through the image once, which makes it considerably better for real-time detection tasks. Algorithms like YOLO that only go through the image once are called single-shot detectors or one-shot detectors, while the methods that require two passes are called two-shot detectors.

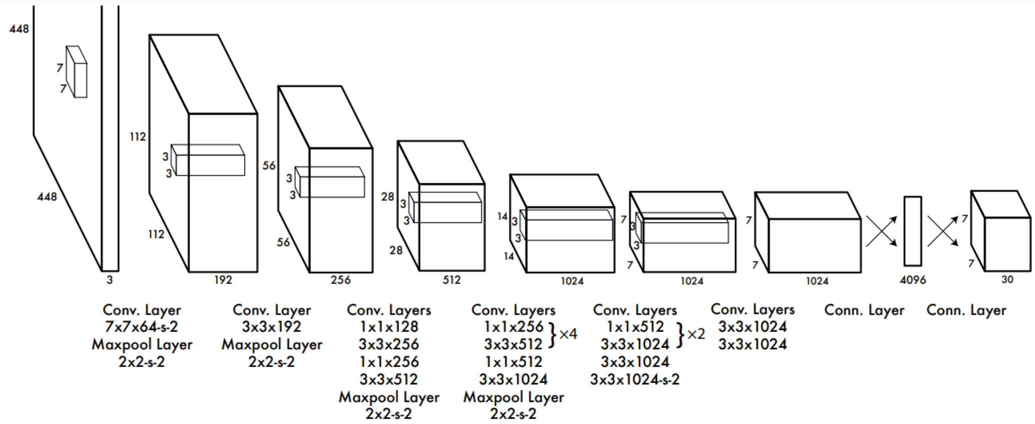
The fact that one-shot detectors such as YOLO or SSD (Single Shot multibox Detector) [6] traverse the image only once to make predictions about the presence of objects in the image, makes them computationally more efficient than other methods that require more than one pass, which also makes them the perfect choice for real-time detection tasks. However, these one-shot detectors are, in general, less accurate than the two-shot detectors and are less effective in detecting small objects.

Two-shot detectors make one pass to generate a set of proposals or potential object locations in the image, and with the second pass these proposals are refined to make the final detections. This approach is more accurate than the one proposed in the one-shot detectors but also more computationally expensive. Algorithms such as R-CNN [7] or Faster R-CNN [7][8] are a few examples among all the two-shot detectors known.

Methods like R-CNN or Deformable Parts Models (DPM) [9] treat object detection as a classification problem. They take a classifier for an object in the image and evaluate it at different locations and scales in a test image. The R-CNN algorithm uses region proposal methods to first propose potential bounding boxes in the image and then run the classifier in those proposed bounding boxes. After this, the bounding box proposals are refined by removing overlapping or duplicate detections.

Unlike these methods, YOLO treats object detection tasks as regression problems, it uses a single CNN in its backbone consisting of 24 convolutional layers for extracting features from the image and two fully-connected layers for predicting the object positions in the image by running it once. Figure 3.1 shows an overview of the architecture of YOLO.

Figure 3.1 Architecture of YOLO consisting of 24 convolutional layers and the final two fully-connected layers. Figure taken from [5]



There are different versions of YOLO as they will be covered later in this chapter. The first version resizes the image to 224x224 for training and then to 448x448 for making detections. It divides an image into a grid of cells and passes through all the cells only once for giving the final object location proposals along with a probability for each of them. It divides the image into a S x S grid and all the cells containing the center point of a certain object will be responsible for detecting that object. Each cell predicts B bounding boxes and the confidence scores for each one of them, this score being how confident the model is of an object being in that cell. Given the probability of that class object and the metric Intersection Over Union (IOU) between the ground-truth bounding box of the object and the bounding box predicted by the model, the confidence score is formally defined as described in Equation 3.1. If there is no object in the cell, the IOU will be zero, hence the confidence score will also be zero. In the equation, *truth* stands for the ground-truth bounding box area that contains the object and *pred* represents the bounding box area of the detection made by the model. An example of both is shown in Figure 3.2. An example of the YOLO detection process is shown in Figure 3.3.

$$P(Object) * IOU_{pred}^{truth} \tag{3.1}$$

Given an image as input for training, YOLO's input labeling format consists of the bounding box center coordinates and the bounding box width and height. On

Figure 3.2 Example of the ground-truth and predicted bounding boxes in an image containing a stop sign.

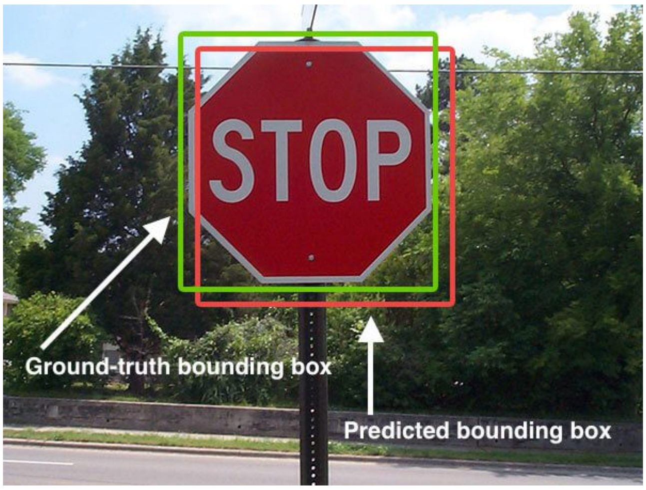
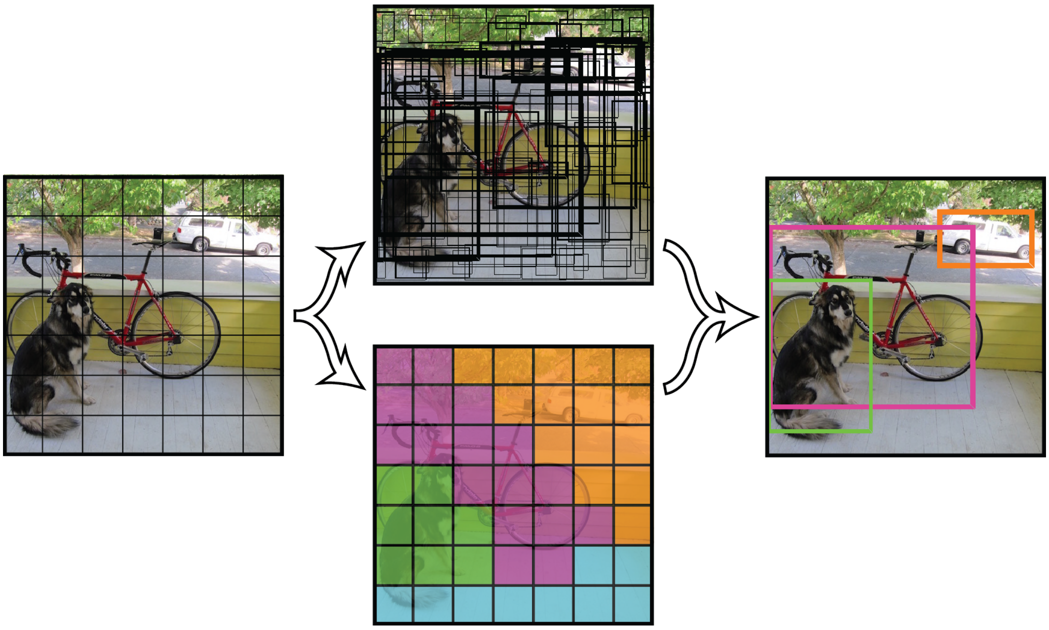


Figure 3.3 Detection process example of YOLO in an image. Figure taken from [5]



the other hand, YOLO returns the following as the prediction results for a given image. If any detections have been made in the image, the bounding box top-right and bottom-left coordinates will be returned along with the confidence score on how confident YOLO is of that object being of a certain class, and the class label and class number.

This first version of YOLO runs at a base 45 frames per second (fps) and a faster version achieves 150fps detections on a Titan X GPU. YOLO reaches a high mean Average Precision (mAP) and along with a good generalization capability makes it a great candidate for real-time labors. More advanced and complete versions of YOLO have been published since the release of the first version in 2016 achieving a higher speed along with higher fps. From YOLOv2 [10] (called YOLO9000) to the latest YOLOv8 [1], the YOLO algorithm has grown through the years and some of the in-between versions have been developed by the community such as YOLOv6 [11] and YOLOv7 [12]. After considering all the different YOLO versions, the YOLOv5 [2]. version has been chosen for this project for the real-time detection task as it is the latest, most stable, and fully finished version of YOLO by the original YOLO creators.

3.2 Cuphead

Cuphead is a 2D game where the player can choose between two main characters to play: Cuphead and Mugman (see Figure 3.4). In june, 2022 a new playable character called Ms. Chalice was added to the game, however, only the two base characters were considered. After taking the role of one of the characters (Cuphead in the case of this project), the player fights against different enemies in order to complete the game and take back Cuphead and Mugman's souls that have been taken from them by The Devil which is the final boss of the game. The game was originally released with four different levels called islands, each containing between five and eleven bosses to defeat. When fighting a boss, it goes through three different phases depending on the difficulty that has been selected (easy, medium and difficult). The game was originally released with 28 different bosses but in 2022 a new island and 7 new bosses where added making a total of 35 bosses to defeat. For this project, the boss Goopy le Grande has been chosen for its simplicity, and only the first phase in easy mode has been considered (see Figure 3.5).

The player starts the fight with three health points, also called hit points, and inflicts damage to the boss by taking actions such as moving left, right, jumping or shooting. If the enemy takes the three health points of the player by inflicting damage to them, the player loses the fight being forced to start the fight from the beginning.

[1]By the date this thesis is written, there is no paper that officially introduces YOLOv8. Instead, a link to the documentation will be provided: https://github.com/ultralytics/ultralytics

[2]Similarly to the v8 version, there is no paper that officially introduces YOLOv5 to this date. Instead, a link to the documentation will be provided:https://github.com/ultralytics/yolov5

All bosses have a certain amount of hit points that act as life points. The player has to take all these points off to win the fight and the player doesn't know these points when the fight is happening. Websites such as the fan-made Cuphead wiki [3] provide more details about the game rules, characteristics and contents about the game.

Figure 3.4 Image showing the two playable characters in the game, where Cuphead is shown on the left side of the image and Mugman on the right side.

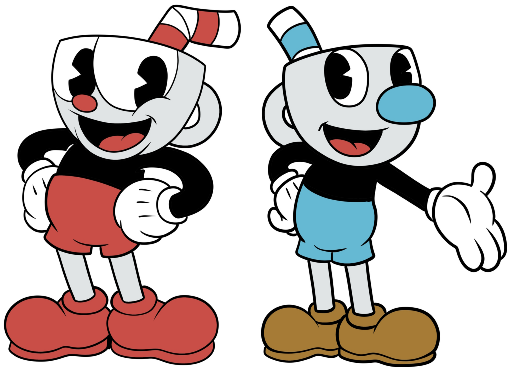


Figure 3.5 An image of Cuphead in the fight against Goopy le Grande at the beginning of the fight.

[3] https://cuphead.fandom.com/wiki/Cuphead_Wiki:Main_page

3.3 Relevant libraries used

Some of the most relevant libraries used are PyTorch for the training and inference of YOLO, `Pandas` for processing the results as a data frame and `OpenCV` and `Pillow` for manipulating images. But there were some libraries such as `win32gui`[4], `keyboard`[5] and `uuid`[6] that were also analyzed as part of the work.

The `win23gui` library was used for capturing the game window in both the frame saving task where data was collected in the training phase and the decision making phase. The `keyboard` library has been the one allowing the automated keyboard interaction, being useful in the training and the decision making phases. Finally, the `uuid` library allowed the generation of universal unique identifiers in the training phase where frames were collected to train the network. As frames were generated along with annotations, each frame had to be named with the same name as the labels file.

[4]https://github.com/wuxc/pywin32doc/blob/master/md/win32gui.md
[5]https://github.com/boppreh/keyboard
[6]https://docs.python.org/3/library/uuid.html

CHAPTER 4

State of the art

The use of many ML techniques and CNNs for games have been widely studied in the last couple of years. Among these techniques we can find DRL and object detection that focus on tasks such as automated playing, classification or object recognition. In [13], authors introduce a method using CNNs for detecting glitches in video game images. In their work, they use a supervised approach for training a ShuffleNetV2 network [14] in order to classify a given input image as one of five classes: normal image, stretched, low resolution, missing or placeholder textures. Each of the last four classes stand for four different kinds of image glitches. Their results, showed an accuracy of 88.6% where 88% of the glitches were detected with a false positive rate of 8.7% and with a good generalization ability to detect glitches in unseen data.

Other works such as [15] focus on explaining different AI based approaches for increasing video game resolution while maintaining a good performance. In some cases, players are forced to lower the in-game video resolution for increasing the game's performance and frame rate. Big companies such as NVIDIA with their named "Deep Learning Super Sampling (DLSS)" [1] or AMD with their open source technology called "FidelityFX" [2] have addressed this problem. Both DLSS and FidelityFX are technologies that use AI for boosting in-game performance by creating new frames through image reconstruction.

Focusing on papers that target DRL for playing video games, in [4] OpenAI developed "OpenAI Five", an AI system able to play and even defeat world champions at the Multiplayer Online Battle Arena (MOBA) game Dota 2. The system runs simultaneously with the game, making decisions every 4 frames based on the current game's state such as enemies' health points, positions and more. As the game and the code changes through the time, OpenAI researchers developed a way for training for over 10 months without having to re-start the training process from

[1] https://developer.nvidia.com/dlss/research

[2] https://gpuopen.com/fidelityfx-superresolution-2/

scratch while being able to make changes into the code and allowing the game to evolve through updates over these months. After over ten months of training and a human evaluation where the system played games against some amateur players, professional players and full professional teams to evaluate the performance in that state of the training phase, the system achieved extraordinary results [3] in a game that was live-streamed all over the world by beating the five world champions at the game. OpenAI Five was also opened to the Dota 2 players' community for competitive games and won 99.4% of the games over 7000 games.

In [16], DeepMind's "AlphaGo" was introduced, an AI system that uses a RL and tree search approach for playing the game Go by using different deep neural networks called "value networks" for evaluating board positions and "policy networks" for selecting moves. To achieve this, they introduced a new algorithm combining these deep neural networks and a Monte Carlo tree search [17] simulation. In the first training stage, a supervised learning (SL) policy network is trained to predict different movement options given a certain board state as input combining convolutional layers with the Rectified Linear Unit (ReLU) activation function. In the second training stage, policy gradient RL is used to improve the policy network and maximize the value of the action to take. This network is finally evaluated in the game against the SL policy network where the RL policy network won over 80% of the games against the SL policy network. Finally, the third stage of the training pipeline focuses on board position evaluation. Using this three-stage training and the mentioned search algorithm the system won 99.8% of the games and defeated the world Go champion.

Other works such as [18] describe how DRL based systems compare to human performance in playing video games and how fast could these systems surpass human performance. The authors implemented and integrated a Deep Q-learning [19] network with the General Video Game AI (GVGAI) software framework called *OpenAI Gym* [20] [21] environment along with the Arcade Learning Environment [22] and the *Petting Zoo* environment [23] to simulate an Atari 2600 console emulator. It runs the game Pong and takes a 160x210 pixel size image as an input for returning an integer as output representing the action to be taken. Although the environment supported all the 18 different actions that the original Atari 2600 supported, only four of them were taken into account: 0 (no action is taken), 1 (shooting), 2 (moving up) and 3 (moving down). The system was trained for over 3 million steps, which accounted for 797 games played and lasted over 49 hours in a desktop computer without GPU support, whilst the original game runs natively at 20 steps per second, which takes 41.67 hours to reach the 3 millions steps that were taken to train the network. Evaluation was made every 100.000 steps, and the results showed that after 300.000 steps, the system surpassed human performance, which took only 4 hours of playing.

[3] https://openai.com/research/openai-five-defeats-dota-2-world-champions

Lastly, in [24] authors created a Proximal Policy Optimization (PPO) based DRL agent and made use of the *OpenAI Gym* [20] [21] library for testing it's behaviour in three different randomly selected games. The agent was trained separately for each of the three games for 1 million steps for two of the games and up to 300.000 steps for the remaining game. Authors saw how the two games that trained for 1 million steps effectively kept increasing the mean rewards as time steps increased. However, in the game that trained for 300.000 steps, the agent only lasted about 10.000 steps before getting trapped in a certain point and not being able to go further with exploring the environment.

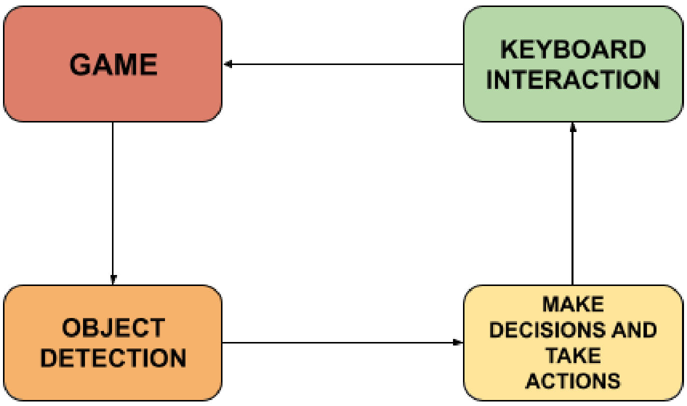
CHAPTER 5

Game analysis

5.1 Module decomposition

In this section, all the modules that compose the initial idea of the object detection+decision making are introduced. Each of these modules was implemented ass part of this thesis, although the AI application is constrained to the object detection module. The whole approach consists of four different but interconnected modules that use each other's outputs for their inputs. A general overview of the four modules is shown in figure 5.1.

Figure 5.1 Overview of the four modules that compose the solution.



5.1.1 Object detection module

The object detection module is responsible for detecting Cuphead and Goopy in the game, providing their x and y coordinates of the top right and left bottom bounding box corners, along with the confidence, class number and class name. In each state of the game, a frame is processed as explained in the next chapter for making the detections that will serve as input in the next module. After processing the frame, YOLO will make detections providing the information of the locations of Cuphead and Goopy which will be converted to the YOLO input format: x and y coordinates of the bounding box center point, bounding box width and height. Information has been converted from the YOLO output format into the YOLO input format for being more comfortable to work with.

5.1.2 Decision making module

After converting the results to the YOLO input format, they are given as input to the module responsible for taking actions. Based on the position of both characters in a certain state of the game, this module will decide what action to take giving a key or set of keys as the output. In each state of the game, Cuphead and Goopy's position may change significantly, therefore, a set of human-based rules has been defined taking into account all the possible actions that can be taken in the game. This will change the output of the module depending on the state of the game and the detections that YOLO made. Only a few actions were considered for defining the rules out of all that can be taken in the game, and both the actions considered and the defined rules will be covered later in Chapter 7.

5.1.3 Keyboard interaction module

Next, the output of the previous module serves as the input to the keyboard interaction module, which will be responsible for pressing the needed keys for executing actions that will change the state of the game. As mentioned in Chapter 3, the `keyboard` library has been used for this purpose. This library provides methods such as `keyboard.press(key)` which will press and hold the given key until it has been released or `keyboard.release(key)` which will release the given key. These two methods along with `keyboard.press_and_release(key)` which is a combination of the two previously mentioned methods and `keyboard.wait(key)` which blocks the execution of the program until the given key is pressed have been considered.

5.1.4 The game

Finally, these actions will be executed and observed in the game, which will change its state and a new frame will be processed by the object detection module for further making decisions as the game goes on. As decisions are made through time, each frame is displayed in a second with all the detections that YOLO made in it. Figure 5.2 shows how will the final setup be which will be covered in the last section of this chapter.

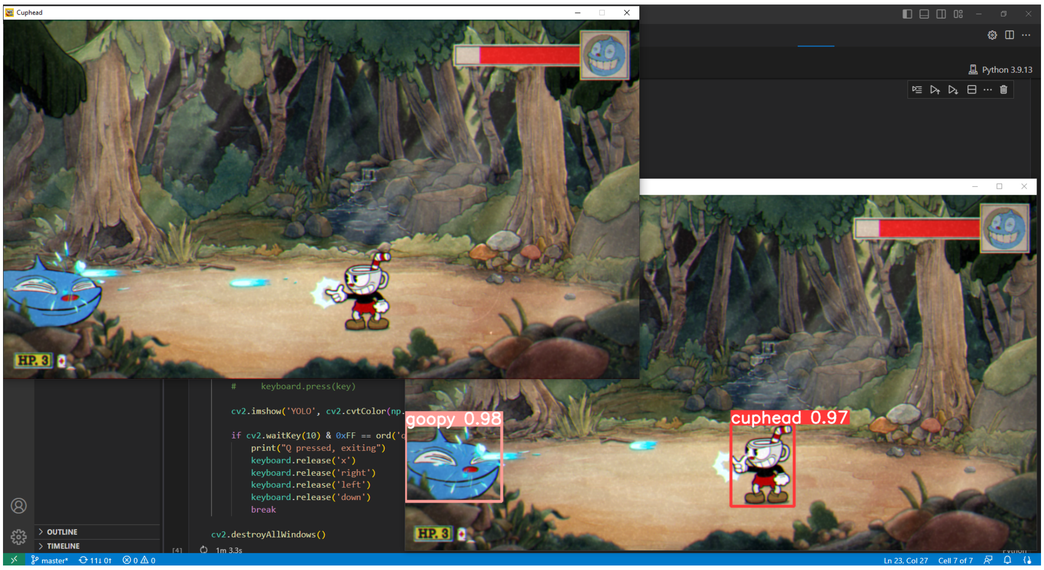
5.2 Game setup

At the moment that all modules have been integrated in the game, a certain setup will be needed for ensuring that all the results are displayed correctly and that the whole application runs as fast as possible without frame drops.

First, as two windows will be displayed in the screen, the one running the game and the one showing the frames with the detections, the game is set to windowed mode, the aspect ratio remains 16:9 and the resolution is lowered to 1176x664. This resolution turned out to be the best for fitting both windows and allowing them to be displayed properly. The final window distribution on the screen is shown in Figure 5.2.

Lastly, the model testing and the training processes have been be carried out using Google Colab, as it provides quality hardware and reduces significantly the inference and training times. For running the final trained model along with the game and all the modules, a desktop PC will be used with an Intel Core i5-9600KF CPU, 16GB of RAM memory and an NVIDIA GeForce GTX 1660ti GPU. As GPU support is provided by PyTorch, detections within the game will be carried out by the GPU, for a higher inference speed allowing the whole process to run as fast as possible.

Figure 5.2 Screen distribution with all the modules integrated. The image shows both screens, the one running the game and the one showing the detections for a frame in a certain state of the game.



CHAPTER 6

Object detection using CNNs

6.1 Proposed object detection approach

The goal when developing the object detection module was to get labeled data without having to manually annotate hundreds of images. To achieve this, a pre-trained network was used and some of the output detections were processed and used as labels for training the same network for finally making more accurate detections onto the game frames.

Firstly, a selection process was carried out were three different models were tested. These models include the FasterRCNN+InceptionResNet built in TensorFlow, SSD+MobileNet also built in TensorFlow, and finally YOLOv5 which was built in PyTorch. While FasterRCNN+InceptionResNet is a two-shot detector with a high accuracy but slower inference time, the SSD+MobileNet and YOLOv5 networks are one-shot detectors with a lower accuracy but faster inference time.

For evaluating all three models under similar conditions, 10 images from the game were extracted and detections were made with each one of them in order to compare their results. After analyzing the results and how each model behaved when detecting objects in the 10 testing images, two models were discarded and the most appropriate model was selected for training and carrying out the detection tasks in the game.

6.2 Model selection

The model selection process started with the three pre-trained models. The first model combines Faster R-CNN [8] (Faster Region based Convolutional Network) for region proposals and InceptionResNet [25] as a classifier for those proposed regions in the second image traversal. This network was originally trained on the

Open Images V4[1] data set, a data set with over 9 million images containing more than 78 million machine-generated labels, over 27 million human-verified labels and 14.6 million bounding boxes for 600 object classes. The experiments performed with this network showed a high accuracy as expected, however, inference time turned out to be the highest amongst all three models with an average time of 1.414s which is too large for real-time detections tasks such as the one presented in this work. Table 6.1 shows the time taken by the network for running inference in all 10 images.

Table 6.1: Inference time of the FasterRCNN+InceptionResNet network for detecting the 10 images from the game.

	Inference time(s)
Image 1	1.399s
Image 2	1.394s
Image 3	1.399s
Image 4	1.379s
Image 5	1.379s
Image 6	1.409s
Image 7	1.391s
Image 8	1.401s
Image 9	1.586s
Image 10	1.402s

Next, the SSD+MobileNet network was tested. This network combines the SSD [6] single-shot detector with the MobileNet [26] detector which acts as a feature extractor and runs much faster than the network previously evaluated. This network was also trained on the Open Images V4 data set and by combining a single-shot detector with a network built for running on mobile applications, makes the model a better candidate for running simultaneously with the game. Predictably, the inference time dropped compared to the previous network but also the accuracy decreased which was expected for being a single-shot detector. In this case, an average time of 0.256s was computed. Table 6.2 shows the time taken by this second network for detecting objects in all the 10 images from the game.

Finally, the YOLOv5 model was tested. Trained on the COCO data set [2] containing over 300.000 images and annotated with 80 classes, and as explained in previous chapters, YOLO is a one-shot detector similarly to the SSD+MobileNet

[1] https://storage.googleapis.com/openimages/web/index.html

[2] https://cocodataset.org

Table 6.2: Inference time of the SSD+MobileNet network for detecting the 10 images from the game.

	Inference time(s)
Image 1	0.274s
Image 2	0.266s
Image 3	0.256s
Image 4	0.241s
Image 5	0.269s
Image 6	0.242s
Image 7	0.258s
Image 8	0.269s
Image 9	0.244s
Image 10	0.245s

model. To the date, various versions of YOLO have been released, being YOLOv5 and YOLOv8 the two latest releases by the original YOLO creators. YOLOv5 was selected over the YOLOv8 version for being the most stable version. YOLOv5 has also five different sizes, each of which is suited for different tasks.

YOLOv5n which stands for nano, is the smallest in the YOLOv5 family and is meant for mobile and IOT solutions. It runs faster than the larger versions but has also the lowest accuracy. The next version is YOLOv5s which stands for small, this is the second smallest model and has a small increase in the GPU inference time compared to the nano version but also a higher accuracy as expected. This model is ideal for running inference on the CPU. YOLOv5m is the medium-sized model of the family, and is well-suited for many data sets and applications for providing a good balance between accuracy and speed. Next, YOLOv5l that stands for large, is the second biggest model of all and provides a good ability for detecting smaller objects along with a higher accuracy but also much more time is needed for inference. Lastly the YOLOv5x that stands for extra large, is the biggest among the five sizes and has the highest accuracy but is also the slowest. The number of parameters, accuracy, CPU time and GPU time for all five sizes of YOLOv5 are shown in Table 6.3 where the parameters are represented in millions, accuracy is computed using the mAP for an IOU threshold of 0.5 and both CPU and GPU times are measured in milliseconds (ms). This information has been taken from the YOLOv5 section [3] that can be found in the PyTorch webpage.

In this case, accuracy is computed using the mAP and for each class precision

[3] https://pytorch.org/hub/ultralytics_yolov5/

and recall are calculated for achieving the PR curve. With the PR curve, the Area Under the PR Curve (AUC) is computed which will represent the Average Precision (AP) for a given class. The Mean Average Precision is the mean AP of each class in the data set.

Table 6.3: Different characteristics from all YOLOv5 models for the different sizes.

Model Name	Parameters	Accuracy	CPU time	GPU time.
YOLOv5n	1.9 M	45.7	45 ms	6.3 ms
YOLOv5s	7.2 M	56.8	98 ms	6.4 ms
YOLOv5m	21.2 M	64.1	224 ms	8.2 ms
YOLOv5l	46.5 M	67.3	430 ms	10.1 ms
YOLOv5x	86.7 M	68.9	766 ms	12.1 ms

As we were looking for quality annotated images, the extra-large version was preferred over the other versions for being the version with the highest accuracy. It turned out to be slower than the SSD+MobileNet model but faster than the FasterRCNN+InceptionResNet network as expected with an average running time of 0.436s. Table 6.4 shows the speed of the YOLOv5x model when running with the 10 selected images from the game.

Table 6.4: Inference time of the YOLOv5x network for detecting the 10 images from the game.

	Inference time(s)
Image 1	0.542s
Image 2	0.466s
Image 3	0.466s
Image 4	0.465s
Image 5	0.407s
Image 6	0.407s
Image 7	0.402s
Image 8	0.402s
Image 9	0.402s
Image 10	0.403s

Looking only at the computational time needed by the networks, the best candidate for the task seems to be the SSD+MobileNet network but other aspects such as how well the models detect the relevant objects in the game or how do these models associate different classes from the dataset they were trained on with the objects from the game need to be analyzed. Detections were made in 10 different images by the three different networks which are shown in figures 6.1 to 6.10 and will be later on discused in this section.

If we analyze the detections that the FasterRCNN+InceptionResNet and the SSD+MobileNet models made, it can be seen how irrelevant objects in the background are detected such as trees or plants while YOLO's detections mostly focus on detecting objects in the front such as both characters or some particles on the screen. Although this may seem a good result, it is indeed a poor outcome as we don't want the background objects to be detected for not being relevant when playing the game. All figures show the overall difference between the detections made by all networks in this context.

Observing the relevant objects that need to be detected precisely, it is expected that the Faster R-CNN network is more accurate than the other two networks. However, for being much slower than the other two, it was discarded and only the SSD and YOLO networks were considered. Overall, YOLO does a better job than SSD.

If we look at images such as Figure 6.1 and Figure 6.2, we find that Goopy is being detected and that the bounding box placement is, in fact, precise. On the contrary, the SSD network detects most of the background objects that don't provide any meaningful information and also neither Cuphead nor Goopy are detected. Furthermore, we can appreciate that Cuphead is also being detected along with Goopy in other cases such as those in figures 6.3, 6.8 and 6.10. This reveals that YOLO can detect both Cuphead and Goopy although in most cases only Goopy is detected. Again, the SSD network detects all the irrelevant noise in the background as well as only Cuphead or only Goopy with many overlapping bounding boxes in all cases. Figure 6.8 shows a case where Goopy and Cuphead are close enough that their detections could not be as precise as needed, but YOLO does an exceptional work distinguishing between the two of them and even if one of the bounding boxes that the SSD detector places on Goopy is genuinely precise, Cuphead is not detected here.

Moreover, to address the issue where Cuphead need to go down in certain states of the game, figures 6.4 and 6.5 exhibit two of those cases. In Figure 6.4, Goopy is starting the animation of the special attack of the first phase of the fight and YOLO perfectly fits Goopy in its bounding box, while the SSD detector doesn't.

Finally, in figures 6.6, 6.7 and 6.9 YOLO does a poor work in comparison to the other cases. Here we address two issues which could be a problem at the time of processing the detections: the undesired particle detection and the overlapped bounding boxes. If we look at figures 6.6 and 6.7, we observe that some background noise is detected and even in the first one the *clock* label is given to Goopy and a particle of minor importance. In this case, the SSD network does a better work as bounding boxes are placed on both Cuphead and Goopy. On the other hand, while SSD is not doing a good job in Figure 6.9 as only background objects are detected, Goopy is not being detected by YOLO at all and we notice that two different bounding boxes are placed on Cuphead, which means that one of them would need to be removed.

While both FasterRCNN+InceptionResNet and SSD+MobileNet networks were trained on the same data set containing 600 classes, YOLO was trained using a data set with only 80 classes which would facilitate assigning classes from the original data set to the characters to be detected. Finally, taken into consideration these arguments and the ones previously discussed, YOLO was selected among the three models.

6.3 Dataset description

Once a model was selected, the next step has been to get a data set with the highest label quality possible without having to manually annotate hundreds of images.

Using Python and the libraries previously introduced such as `win32gui` or `uuid`, a method was developed for saving frames from the game while simultaneously playing. This would allow us to get hundreds or thousands of images in a few minutes only by playing. After collecting over 600 images, YOLO made detections in all of them and the output was processed and converted into the input format that YOLO uses for making predictions. Tables 6.5 and 6.6 describe the output and input format of YOLO, respectively.

Table 6.5: Output format of YOLO.

xmin	x coordinate of the bottom left corner of the bounding box
ymin	y coordinate of the bottom left corner of the bounding box
xmax	x coordinate of the top right corner of the bounding box
ymax	y coordinate of the top right corner of the bounding box
confidence	confidence of how certain YOLO is that the object belongs to the assigned class
class	class number, an integer between 0 and 79
name	class name, e.g. kite, motorcycle or teddy bear

The results given by YOLO went through some filters before being converted into the input format. Some detections showed that in a number of cases, more than one class was assigned to the same object, e.g. motorcycle and bike to Cuphead.

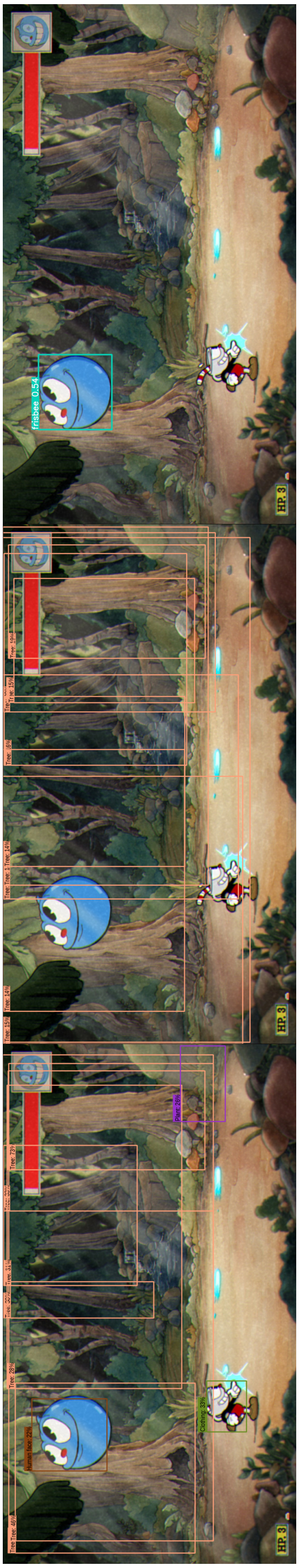


Figure 6.1: Detections made by each model in the 1st image from the game. From left to right, FasterRCNN+InceptionResNet, SSD+MobileNet and YOLOv5x detections are shown.

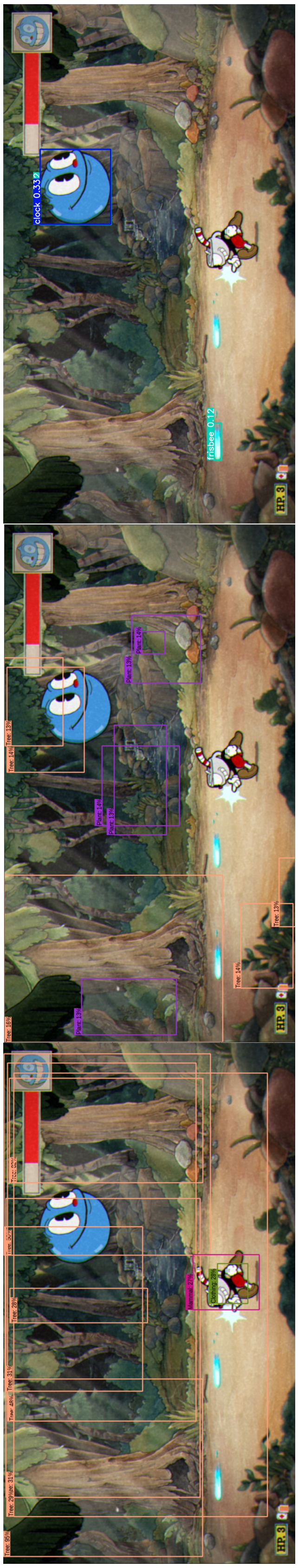


Figure 6.2: Detections made by each model in the 2nd image from the game. From left to right, FasterRCNN+InceptionResNet, SSD+MobileNet and YOLOv5x detections are shown.

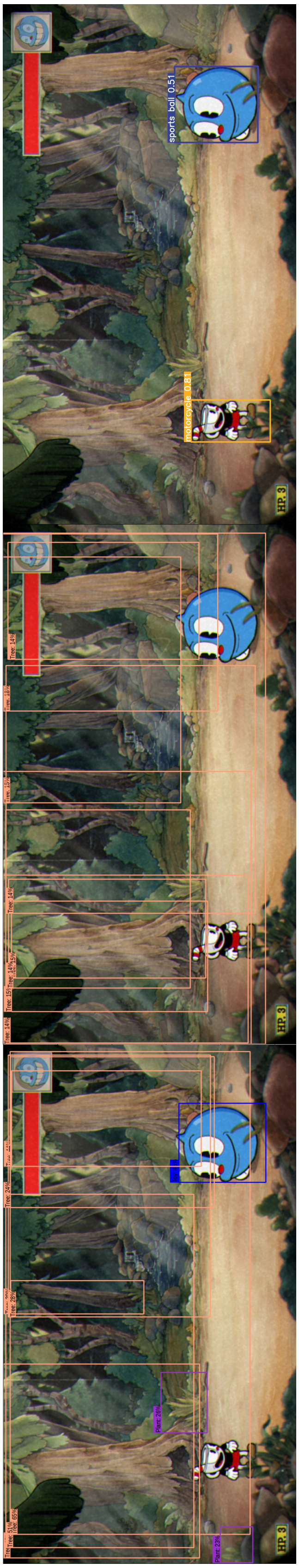


Figure 6.3: Detections made by each model in the 3rd image from the game. From left to right, FasterRCNN+InceptionResNet, SSD+MobileNet and YOLOv5x detections are shown.

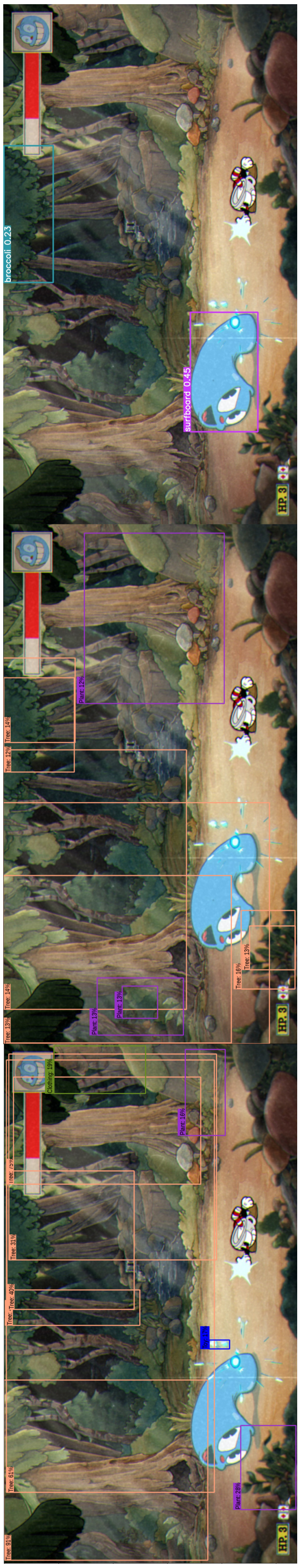


Figure 6.4: Detections made by each model in the 4th image from the game. From left to right, FasterRCNN+InceptionResNet, SSD+MobileNet and YOLOv5x detections are shown.

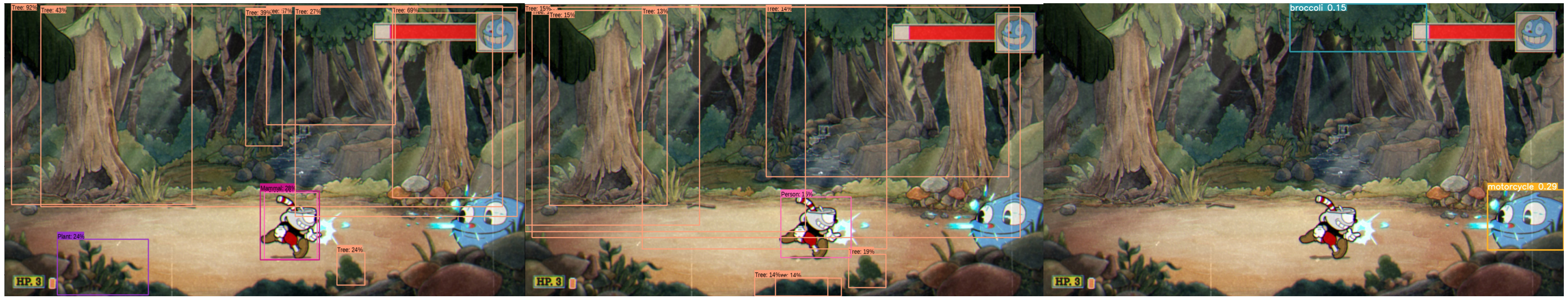


Figure 6.5: Detections made by each model in the 5th image from the game. From left to right, FasterRCNN+InceptionResNet, SSD+MobileNet and YOLOv5x detections are shown.

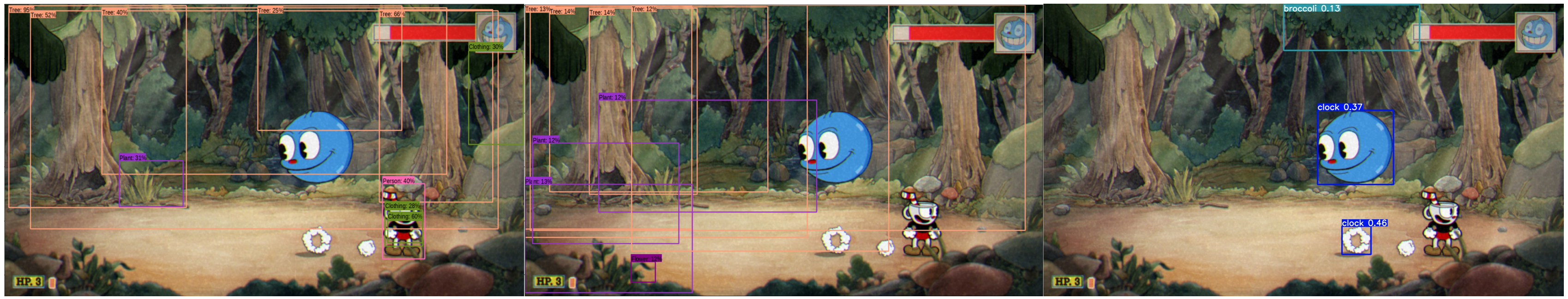


Figure 6.6: Detections made by each model in the 6th image from the game. From left to right, FasterRCNN+InceptionResNet, SSD+MobileNet and YOLOv5x detections are shown.

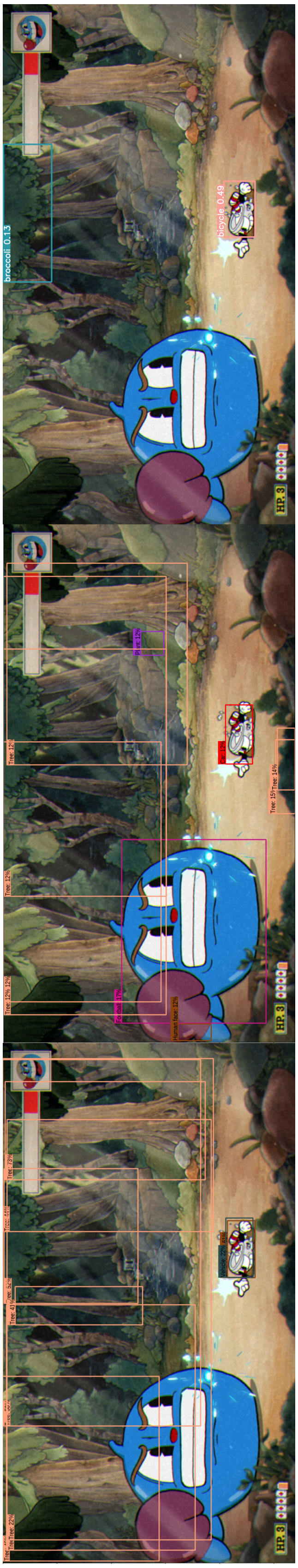


Figure 6.7: Detections made by each model in the 7th image from the game. From left to right, FasterRCNN+InceptionResNet, SSD+MobileNet and YOLOv5x detections are shown.

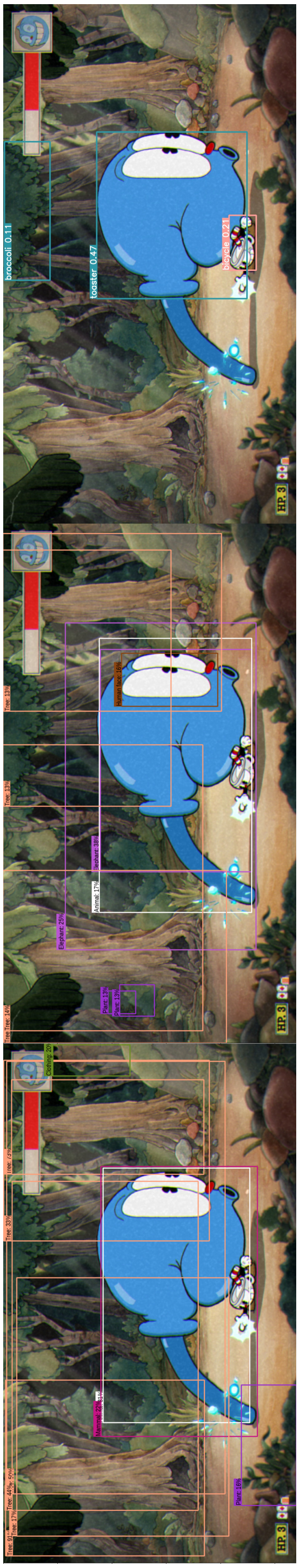


Figure 6.8: Detections made by each model in the 8th image from the game. From left to right, FasterRCNN+InceptionResNet, SSD+MobileNet and YOLOv5x detections are shown.

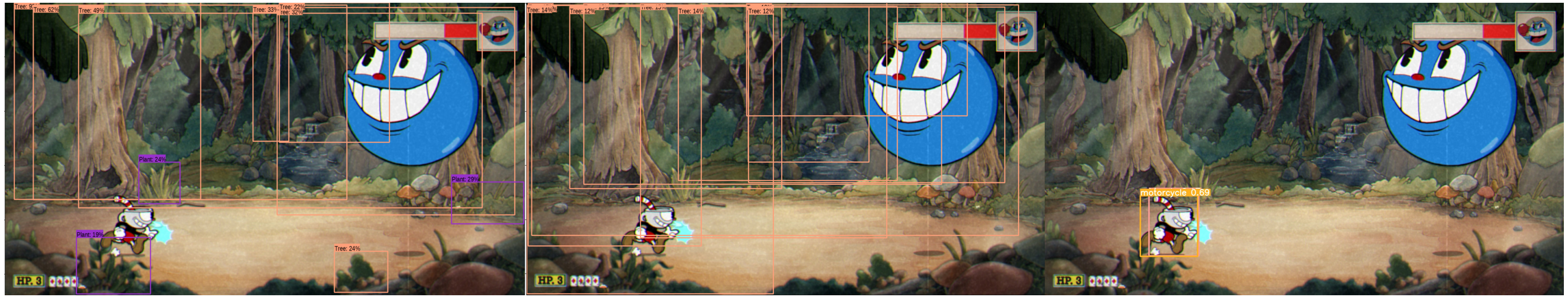


Figure 6.9: Detections made by each model in the 9th image from the game. From left to right, FasterRCNN+InceptionResNet, SSD+MobileNet and YOLOv5x detections are shown.

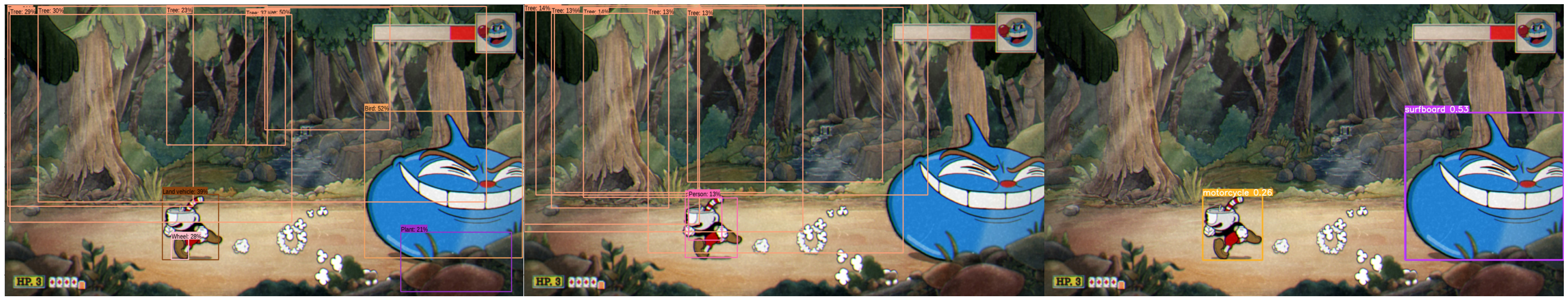


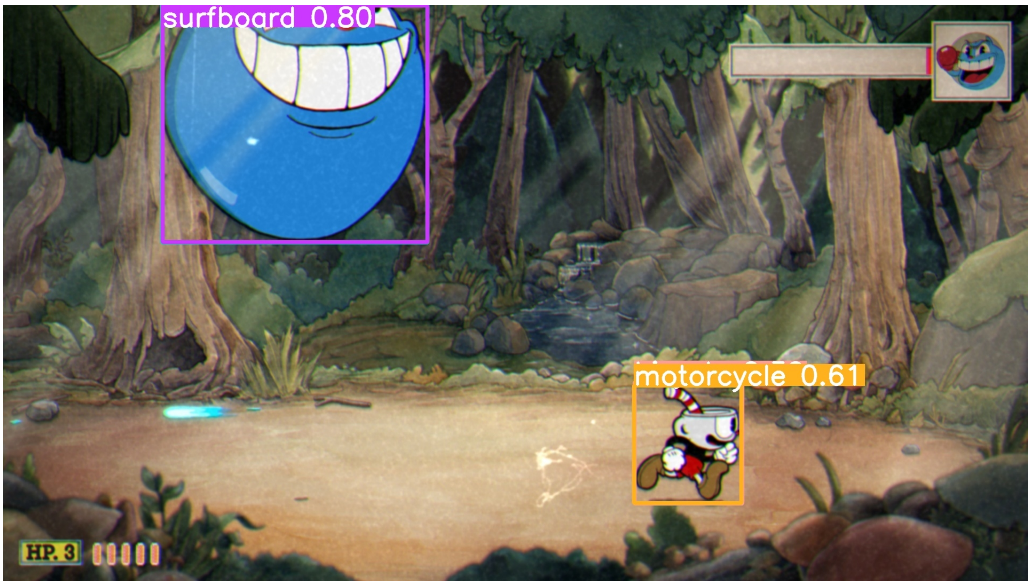
Figure 6.10: Detections made by each model in the 10th image from the game. From left to right, FasterRCNN+InceptionResNet, SSD+MobileNet and YOLOv5x detections are shown.

Table 6.6: Input format of YOLO.

x	x coordinate of the bounding box center
y	y coordinate of the bounding box center
width	width of the bounding box
height	height of the bounding box
class	class number of the object

An example of this problem is shown in Figure 6.11, where two bounding boxes are placed on Cuphead.

Figure 6.11 Example of an overlap in the detections made by YOLO. In this case the overlaping happens between the motorcycle class and another one onto cuphead.



To overcome this issue, detections with a confidence below 0.39 will be discarded and in the case that the overlapped bounding boxes have a confidence over 0.39, the one with the highest confidence out of all the overlapped boxes will be selected, thus eliminating the rest. In addition, some classes were assigned to each character by manually looking at the detections made by YOLO in a small set of images. It was observed that classes such as person, motorcycle or kite were more ofter assigned by YOLO to Cuphead and classes such as surfboard, frisbee or sports ball were associated to Goopy. Following this approach, the human effort is minimum as only few images needed to be checked to identify the most associated classes for the objects of interest. All the classes that were mostly associated to each character are shown in tables 6.7 and 6.8 for Cuphead and Goopy, respectively.

Table 6.7: Classes associated to Cuphead by YOLO. Class names are shown along with their class number of the original COCO dataset.

Class name	Person	Bicycle	Motorcycle	Bench
Class number	0	1	3	13

Table 6.8: Classes associated to Goopy by YOLO. Class names are shown along with their class number of the original COCO data set.

Class name	Bird	Frisbee	Sports ball	Surf-board	Bowl	Cake	Toaster	Clock	Vase
Class number	14	29	32	37	45	55	70	74	75

After removing all the irrelevant classes, they were renamed to the new classes depending on the original class. Classes associated to Cuphead were renamed to *"cuphead"* with the class number 0 and classes associated to Goopy were renamed to *"goopy"* with the class number 1.

Once the filtering was done, two subsets were extracted with properly labeled data. The training subset consists of 606 images and labels while the validation set contains 151 images and labels. In each of the images, one label is provided for each of the classes, while in some images two labels are provided for Goopy. These happens in a certain state where Goopy makes a special attack against Cuphead and two Goopys are detected. An example of this situation is shown in Figure 6.12. Finally, the class distribution is shown in Figure 6.13, and for the above mentioned, there are a few more Goopy annotations than Cuphead's.

6.4 Training process

The aim of the whole training process was to obtain a model capable of detecting Cuphead and Goopy within the game using data that has been labeled automatically, as manually annotating hundreds or even thousands of images would take considerable time.

Starting from the YOLOv5s weights, YOLO was fine-tuned twice before reaching the final results. The initial idea was to carry out a single training stage. However, as YOLO didn't made a satisfactory job at the time of detecting Cuphead in different poses and positions in the screen, later on it was decided to perform a second training stage with whose detections a higher quality data set was achieved. A first training phase of 60 epochs and a batch size of 8 was carried out with a first data set containing about 400 images for training and validation where classes were

Figure 6.12 Example of a frame where goopy is detected twice but still serves as an appropriate labeled image.

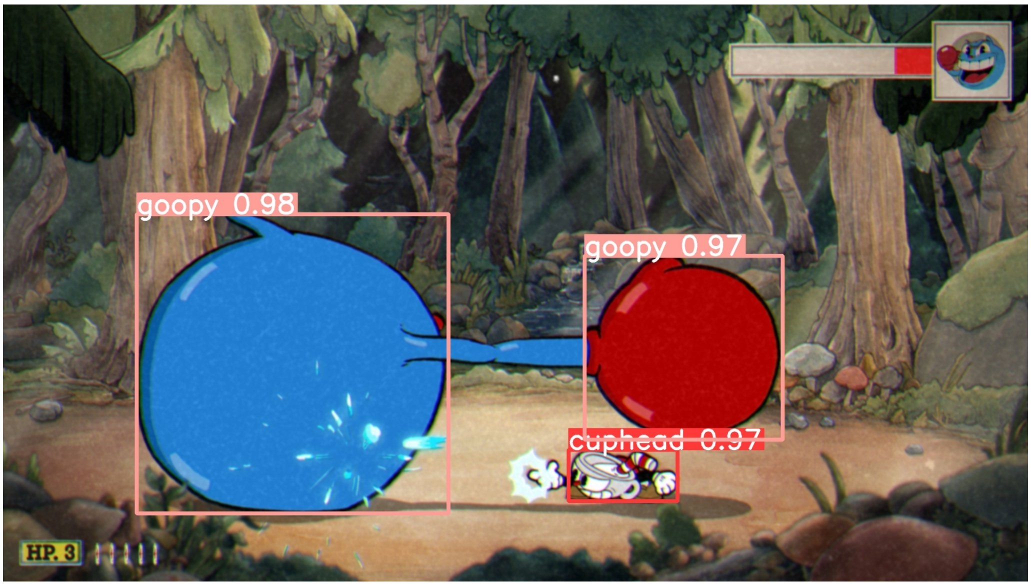
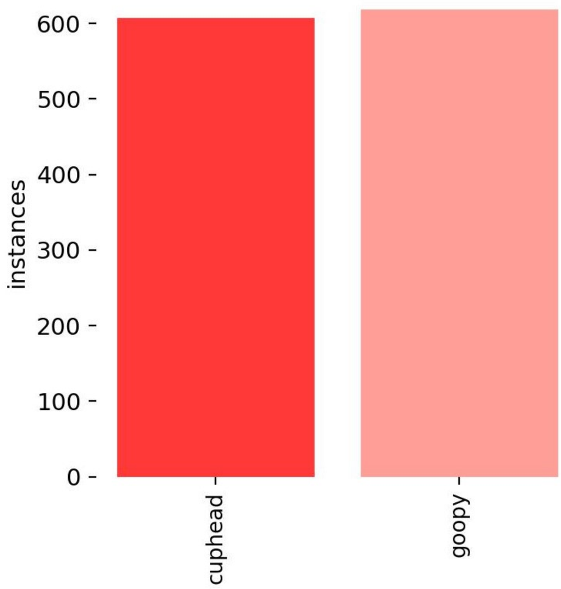


Figure 6.13 Class distribution of the data set used in the second training stage for the two classes defined.



unbalanced as shown in Figure 6.14. In this case, classes are unbalanced as the pre-trained YOLOv5x model struggled to detect Cuphead in different positions but did a decent job detecting Goopy in different positions, for this reason, this first data set contained more instances of Goopy than Cuphead. Despite the fact that this training phase showed better results than the expected as Goopy was detected in most cases, it wasn't enough for running the model in real time while running the game since Cuphead was being detected only in a few poses. As the objective was to develop a fully functional AI approach, high quality detections were needed and it was decided to carry out a second training stage with a new data set annotated using the weights after the first training stage instead of the pre-trained YOLOv5x model as it will be explained later on this chapter.

Let us introduce Algorithm 1 , which shows the pseudocode of the method for filtering images for molding the data set used in the first training phase. The filter is made for filtering the detections of the pre-trained YOLOv5x model, which labels objects according to the classes of the original COCO data set and the association explained in the previous section.

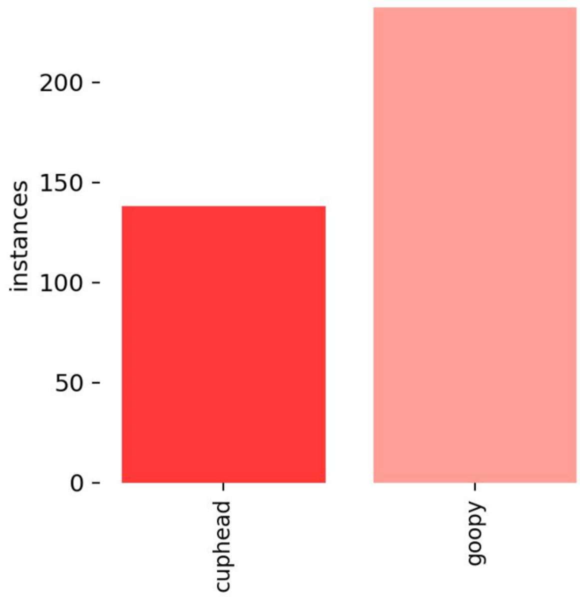
Algorithm 1 Method for filtering and labeling images in the first training stage. Both Cuphead and Goopy are associated with different class labels from the original COCO data set and two new classes are defined: *cuphead* and *goopy*.

```
1: for each detection do
2:     remove detections with a confidence below 0.39
3:     remove detections with an associated class different from the ones in tables
       6.7 and 6.8
4:     remove detections that have overlapped bounding boxes as shown in Figure
       6.11
5:     convert the remaining results into the YOLO input format
6:     change class number to 0 or 1 depending on the original class label
7: end for
```

The results obtained after the first training phase showed that detecting Goopy was relatively easy. On the other hand, the model struggled detecting Cuphead in all the possible positions that it could appear. Cuphead could be facing right, left, up or down while standing and facing left or right while being crouched down and the model generally only worked well when Cuphead was down.

After the first training stage, Cuphead is detected standing still and facing all sides only in a few frames and after observing some of the detections made by the model achieved after the first training phase, it was decided to do a second training stage using a new data set fully annotated using this model instead of the pre-trained YOLOv5x model used in the first filtering process. Consequently another $\sim$1500 images from the game were extracted again for labeling them with the model achieved after the first training phase. For ensuring that each image had at least one annotation for each class, only the images where both Cuphead and Goopy were simultaneosly detected were taken into account and the ones where

Figure 6.14 Class distribution of the data set used in the first training stage for the two classes defined.



only one of them was detected were discarded. After setting the new filter, the model acquired after the first training stage was executed on the new ~1500 images that were collected. From this process, 757 images were extracted where 606 were assigned to the test set and the remaining 151 to the validation set. This data set was going to be used to re-train for the second time as all the images contained quality Cuphead and Goopy labels. The second training stage performed for 150 epochs with a 16 batch size and after less than 15 minutes the trained model was able to successfully detect Goopy and Cuphead within the game.

We will discuss now Algorithm 2, which shows us the pseudocode of the method for filtering images in the second training phase. After labeling images according to the first filtering criteria and training YOLO with these images, this second filter takes the detections made by YOLO after the first training phase for re-training YOLO once again and thus getting more accurate results.

Algorithm 2 Method for filtering and labeling images in the second training stage. After training YOLO with the first data set obtained with the first filter, new data is extracted with this second method for the second training stage.

for each detection **do**
    remove detections with a confidence below 0.29
    if Cuphead and Goopy are detected **then**
        remove detections that have overlapped bounding boxes as shown in Figure 6.11
        convert the remaining results into the YOLO input format
        remove the detections with a bounding box area below 0.007
    end if
end for

CHAPTER 7

Human rules for the game based on object detection

7.1 Actions taken

As explained in the risk prevention section in Chapter 2, it has not been possible to implement this module using DRL techniques. Instead, a group of actions has been defined in order to achieve success on defeating Goopy based on simple human rules. The purpose of defining these rules was to complete the framework where we could evaluate the models for object detection introduced in previous chapters.

Out of all the possible moves that Cuphead can make in the game some of them were discarded and not taken into account for not being useful, e.g, jumping or walking. In the other hand, actions such as shooting, dashing or going down were considered for being necessary to avoid as much damage as possible, which is fundamental for beating the enemy. Table 7.1 shows all actions that were considered along with the key associated to each of them.

Table 7.1: Considered actions and associated key or key set to each of them.

Action	Associated key
Shoot	X
Right	Right-arrow
Left	Left-arrow
Down	Down-arrow
Dash (towards a direction)	Left-Shift + arrow key

Between the selected actions shooting is the most important among all. By keeping the shooting key pressed through the entire fight, the enemy receives as much damage as possible in each state of the fight. Without looking if Cuphead is standing still, moving or crouched down, the shooting key is always pressed.

Furthermore, the left and right arrow keys are needed for making Cuphead face Goopy. When the left arrow key is pressed, Cuphead will look left and by pressing the right arrow key Cuphead will face right. This action will allow Cuphead shooting directly Goopy when their bounding box center points are at almost the same level.

Going down is also a crucial action in the fight. Goopy makes two special attacks in both of his phases that require Cuphead to go down for avoiding the damage. Figure 7.1 shows two frames in the first and second phase, respectively where Goopy makes the mentioned attacks that force Cuphead to go down. In both cases, going down makes Cuphead avoid damage.

Figure 7.1 The left frame shows the attack where Cuphead needs to go down for avoiding damage in Goopy's first phase and the right frame shows another attack that requires Cuphead to go down but in Goopy's second phase.

Finally, dashing makes Cuphead to move from point a to point b in the x axis by disappearing for a very short period of time while avoiding all damage in the way. Dashing is a way of moving Cuphead without having to walk all the distance as it makes him teleport from one point to another. This action was selected over the simple movement action as walking from one point to another would potentially make Cuphead suffer damage. In the game, Cuphead can dash left by pressing the left-shift+left-arrow key sequence or right by pressing the left-shift+right-arrow key sequence. An example of the dashing action can be seen in Figure 7.2, where Cuphead dashes from right to left to avoid Goopy's damage.

As mentioned above, the jumping and walking actions were discarded between all the possible actions that can be taken in the game. Jumping serves no purpose when fighting against Goopy as all the possible attacks that Cuphead can face in the fight can be avoided by dashing or going down. And considering this action for the fight would require more effort at extracting labeled data for detecting Cuphead while jumping. In addition to this, it makes no sense considering the walking action over the dashing action since the dashing makes Cuphead avoid damage while walking exposes him through the entire time that the action is being taken.

Figure 7.2 Frame showing the final state of a dashing action of Cuphead. Cuphead can be seen on the left behind the clouds as some very little clouds appear at the initial and final point where Cuphead dashes. The clouds on the right indicate where does Cuphead come from.

7.2 Processing of the detections for taking actions

Each frame is processed and given to the trained network for detecting objects in the image. The results that YOLO returns can be stored in an object containing all the detections within the image. These results will be processed for converting them into the YOLO input format as it is easier working in this format. Every frame that had exactly two detections has been processed and the very few frames that may have more or less than two detections were discarded.

For each frame where two detections were made by YOLO, a class number, bounding box center coordinates, bounding box width and bounding box height are given after processing the results. In each frame, all the values mentioned above are normalized and set to a range between 0 and 1, while the class number takes either the 0 or 1 value.

The first goal was to make Cuphead face Goopy so the shot bullets would hit him. No matter the frame, the shooting key is always pressed, that way we ensure that Goopy takes as much damage as possible. Looking at both Cuphead and Goopy's detections, when Cuphead's x coordinate's value is smaller than Goopy's x coordinate value means that Cuphead is standing closer to the left side of the image than Goopy. On the other hand, if Cuphead's x coordinate value is greater than Goopy's value means that now Cuphead is standing closer to the right side instead of the left side of the frame. Figure 7.3 shows two frames were Cuphead faces both sides depending on Goopy's position. In the left frame Cuphead's bounding box center (x, y) coordinates are (0.18, 0.76) and Goopy's coordinates are (0.58, 0.75).

As Cuphead's x coordinate is smaller than Goopy's, the right key is pressed for Cuphead to face Goopy. Additionally, the second frame shows a case were Cuphead stands closer to the right side than Goopy where his coordinates are (0.81, 0.75) and Goopy's coordinates are (0.26, 0.75). While Cuphead's x coordinate is greater than Goopy's x coordinate, the left key is pressed in this case.

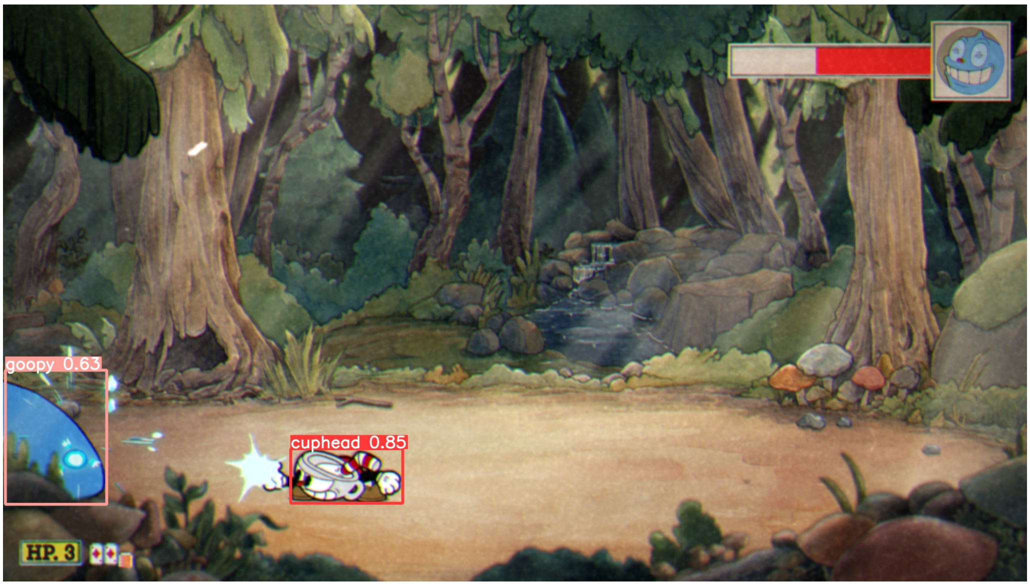
Figure 7.3 Two frames showing cases were Cuphead faces both sides for shooting Goopy. The left figure shows a case were Cuphead's x coordinate is smaller than Goopy's and in the right figure Cuphead is shown closer to the right side than Goopy as his x coordinate is greater than Goopy's.

When it comes to the dashing action, as explained before, it allows Cuphead to move from point a to point b in the x axis avoiding any damage that would take by walking. As Goopy jumps towards both sides during the entire fight, when Cuphead and Goopy are close enough that Cuphead would take damage, dashing makes Cuphead avoid the damage. When both are at a distance less than 0.15 and depending on the location of both, the left-shift+left-arrow or left-shift+right-arrow key sequences will be pressed. If Cuphead's x value is smaller than Goopy's and the absolute value of the subtraction of both x values is less than the defined threshold, Cuphead will dash rightwards and in the similar case were Cuphead stands closer to the right side and Goopy closer to the left side, Cuphead will dash leftwards. Figure 7.2 shows an example were Cuphead dashes leftwards as Goopy gets closer to him.

In addition to this, as showed in Figure 7.1, Goopy can make two special attacks in both of his phases that require Cuphead to go down. The first attack shown in the left frame makes Goopy prepare himself after charging left or right. In this case, Goopy's bounding box area grows approximately to 0.05 (see Figure 7.4), therefore in a frame where Goopy's detection bounding box area is greater or equal to the defined threshold of 0.05, the down arrow key will be pressed and only when Goopy's area goes back to a value less than 0.05, the down arrow key will be released. This makes Cuphead to stay down through the entire time that Goopy makes the attack, allowing him to avoid all damage. An issue was detected when Goopy stood beyond the left or right limits. When Goopy charges backwards for making the special attack while being positioned close to the left or right limits of the screen, the are of the bounding box containing the visible part of Goopy was less than the defined threshold value of 0.05. Therefore, when Goopy stands close to those limits Cuphead will automatically go down in order to be able to avoid the special attack as showed in Figure 7.5 shows an state where the issue mentioned occurs.

Figure 7.4 A frame where Goopy charges backwards before making the special attack that requires Cuphead to go down.

Figure 7.5 A frame where Goopy lies close to the left limit of the screen and its bounding box area is less than the defined threshold of 0.05.



In the second phase, Goopy gets bigger and so does the area of the bounding box of the detection in each frame. This means that the bounding box area will always be bigger than the 0.05 threshold defined for the attack of the first phase. However, being down allows Cuphead to shoot and dash if needed so even if Cuphead will stay down during the entire second phase, it won't stop him from executing other actions when needed, allowing him to avoid all the possible damage.

Finally, an issue was spotted that made Cuphead non detectable and therefore, stuck in a place of the screen were if couldn't be detected. As the environment doesn't change through the entire fight, we can see in both left and right sides of the frame two groups of rocks. These rocks appear in front of any character standing behind them and in some cases, making them non detectable. Figure 7.6 shows the case where Cuphead is behind this group of rocks on the right side and if he goes down YOLO won't detect him. If Cuphead is not detected, only Goopy will be detected in every frame and as frames with more or less than two detections are discarded, no action will be taken and a loop will be entered until Cuphead leaves this state. Cuphead will only be able to leave this state when Goopy hits him and YOLO detects him again. Leaving this situation forces Cuphead to take at least one hit point and makes that game less likely to be won. To avoid this issue, each time that Cuphead is detected beyond the left and right limits that are defined, he will dash to the opposite side. With this action we lower considerably the chance of Cuphead entering this state and therefore, increasing the chances of winning the fight. The defined left and right limits are 0.15 and 0.85, and if at any frame Cuphead's bounding box center's x coordinate is less than 0.15 or greater than 0.85, the dashing action will be executed.

Figure 7.6 A frame where Cuphead can't be detected for being behind a group of rocks.

CHAPTER 8

Experiments and analysis of the results

The goal of the experiments is to evaluate the capability of YOLO to detect both characters in the game after the two training stages and the ability of the whole AI-based approach at the time of playing the game.

For each training session a file called `results.txt` is created in the same folder where YOLO is stored, which contains information related to that training session such as the mAP, precision, recall or the change in the box, object and class losses on both training and validation sets. In order to evaluate the object detection module, the information provided in this file will be analyzed for both training stages, from the mAP to the precision and the recall achieved on the validation set. In the first training stage, these metrics are computed using the same data used for training. On the other hand, in the second training stage a set containing 151 annotated images have been used for validation, as described in previous chapters.

Then, for the purpose of testing the final trained model and to compare its performance on unseen data with the pre-trained YOLOv5x version and the model obtained after the first training stage, 10 images from the game have been extracted which contain the most relevant parts of the fight and the detections made by all the three models have been analyzed.

Finally, to test the performance of the AI-based solution at the time of playing the game and after putting all the modules together, 10 fights have been carried out and the statistics of those fights are discussed later on the chapter.

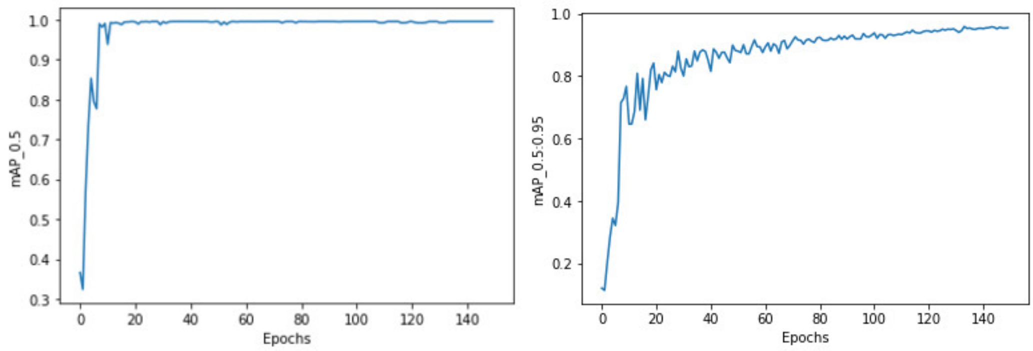
8.1 Experiments of the object detection module

8.1.1 Metrics analysis

First, YOLO was able to make some minor detections on the images but still not enough for running simultaneously with the game. After the first training stage, the model reached a mAP of 92.6% for an IOU threshold of 0.5 and 83.77% for an IOU threshold between 0.5 and 0.95 on the validation set. Although it may seem a good performance, when testing the model on unseen data from the game and running it in real time, Cuphead was only detected in few states of the fight and in the rest of the cases YOLO couldn't detect Cuphead at all. This was due to the lack of different Cuphead instances in the data set.

After extracting the second data set, the second training stage was carried out. It showed very good results compared to the previous results obtained after the first training phase and to the pre-trained model itself. A mAP of 99.5% for an IOU threshold of 0.5 and 95.82% for an IOU threshold between 0.5 and 0.95 was achieved on the validation set along with a 99.39% precision and a 99.5% recall. These metrics through the epochs are shown in figures 8.1 and 8.2.

Figure 8.1 Mean Average Precision reached by the model through the epochs on the validation set. From left to right: mAP for an IOU threshold of 0.5 and mAP for an IOU threshold between 0.5 and 0.95.

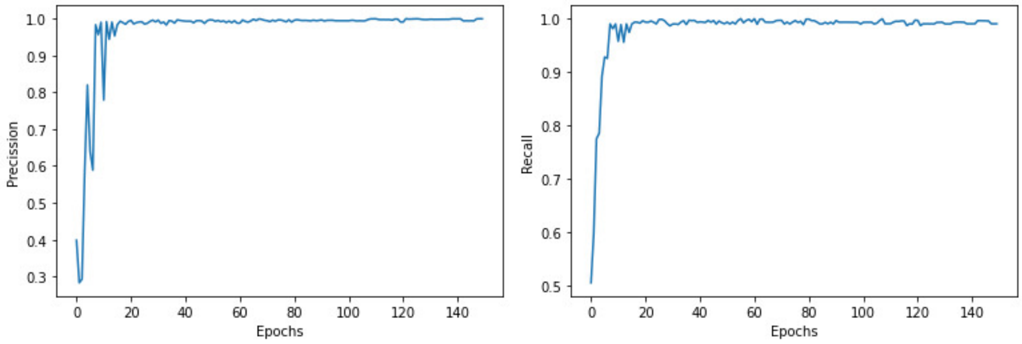


In Figure 8.3, it can be observed that the box, object and class losses kept decreasing through the epochs. This means that bounding box placements and class assignment qualities in detections were improving through time. In all cases, the validation curve stays under the training curve in most epochs, which means that the model was performing well in unseen data.

8.1.2 Model comparison

Finally, the pre-trained YOLOv5x model, the model achieved after the first training phase and the final model were evaluated on a new set of 10 images, different from the ones used in the model selection process to see the real impact of the training

Figure 8.2 From left to right: precision and recall reached by the model through the epochs on the validation set.



process and to evaluate the performance on detecting the desired objects in-game. Figures 8.4 to 8.13 show a comparison of each of the detections made by the three models on these testing images.

We can clearly see the improvement in the detections after each training stage in comparisson to the detections made by the pre-trained YOLOv5x model. Overall, the pre-trained version made a good job detecting Goopy but struggled detecting Cuphead. When it comes to the pre-trained YOLOv5x model, in almost all cases such as those in figures 8.4, 8.5, 8.8, 8.10 and 8.11, Goopy was detected but not Cuphead. In frames such as those in figures 8.6 and 8.9 Goopy was not being detected at all and after both training phases we can remarkably see the improvement. When it comes to the bounding box placement, we discussed in Chapter 6 that the pre-trained version of YOLOv5x did a good job by placing them very precisely in the cases where either Cuphead or Goopy was being detected and this performance was also observed after both training phases.

In relation to the accuracy in detecting Cuphead in the images, as stated in Chapter 6, YOLO faced difficulties trying to fulfill this task. Only in a few cases such as those in figures 8.6, 8.12 and 8.13, YOLO could detect Cuphead and in the rest of the cases, YOLO missed it. After the second training stage, Cuphead is detected in all frames. In cases where Cuphead is down and situated close to Goopy similarly to the case in Figure 8.7, YOLO could not detect Cuphead since both bounding boxes are overlapped. In addition, if we observe the 3rd frame in the figure, YOLO perfectly places two different bounding boxes on both characters.

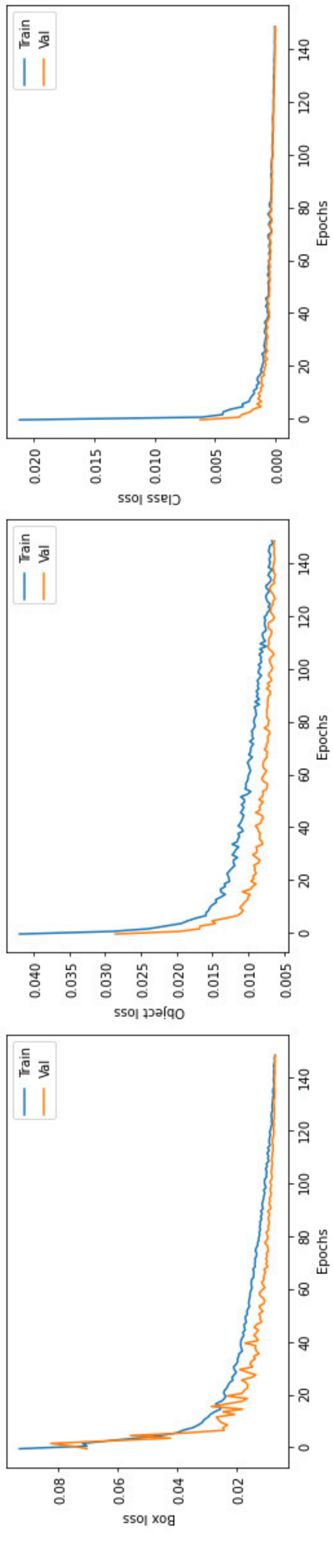


Figure 8.3: From left to right: comparison of the box, object and class losses through the epochs on the training and validation sets.

Figure 8.4: Detections made by YOLO in the 1st testing image from the game. From left to right, the pre-trained YOLOv5x, the model after the first training and the last model detections are shown.

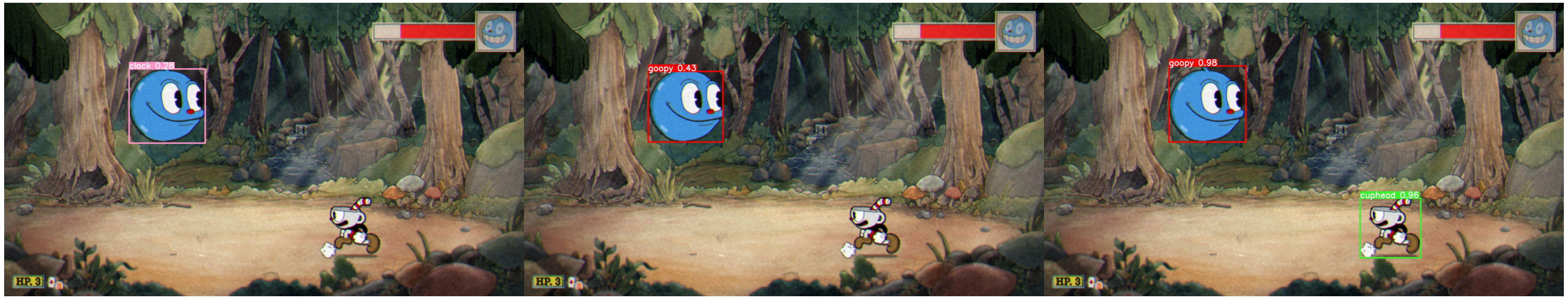


Figure 8.5: Detections made by YOLO in the 2nd testing image from the game. From left to right, the pre-trained YOLOv5x, the model after the first training and the last model detections are shown.

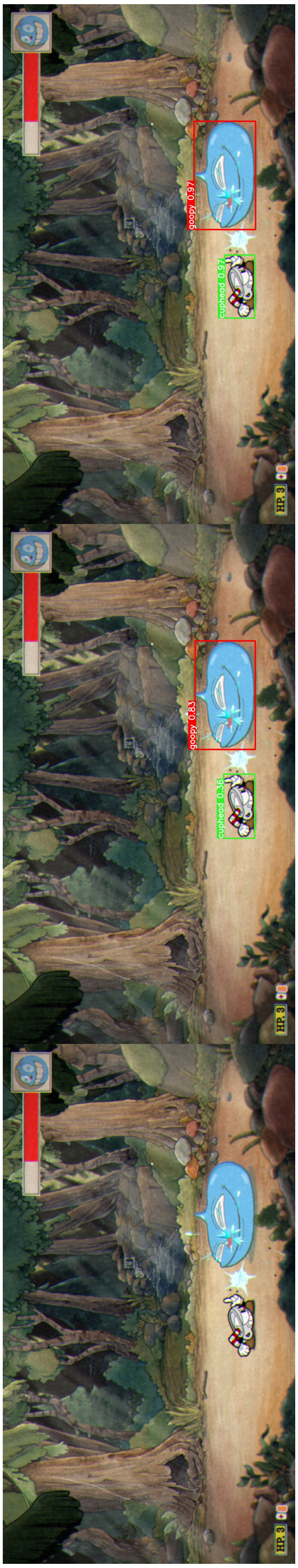


Figure 8.6: Detections made by YOLO in the 3rd testing image from the game. From left to right, the pre-trained YOLOv5x, the model after the first training and the last model detections are shown.

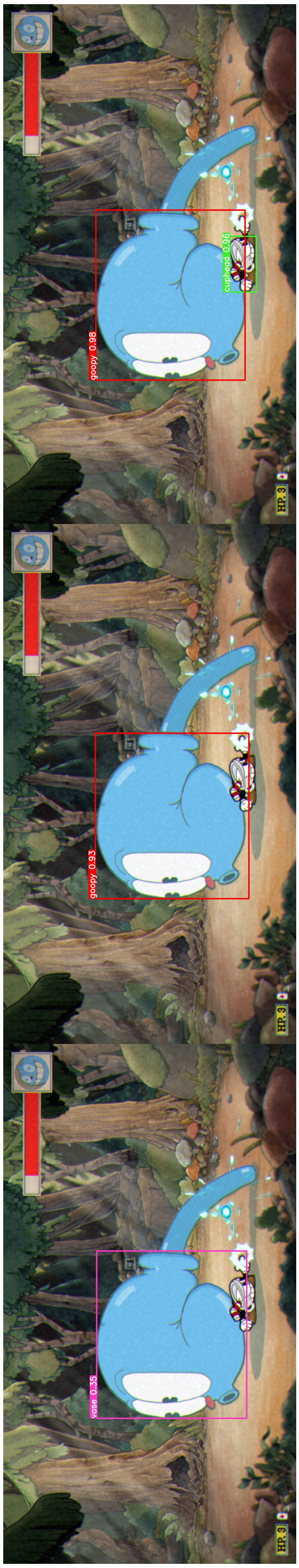


Figure 8.7: Detections made by YOLO in the 4th testing image from the game. From left to right, the pre-trained YOLOv5x, the model after the first training and the last model detections are shown.

Figure 8.8: Detections made by YOLO in the 5th testing image from the game. From left to right, the pre-trained YOLOv5x, the model after the first training and the last model detections are shown.

Figure 8.9: Detections made by YOLO in the 6th testing image from the game. From left to right, the pre-trained YOLOv5x, the model after the first training and the last model detections are shown.

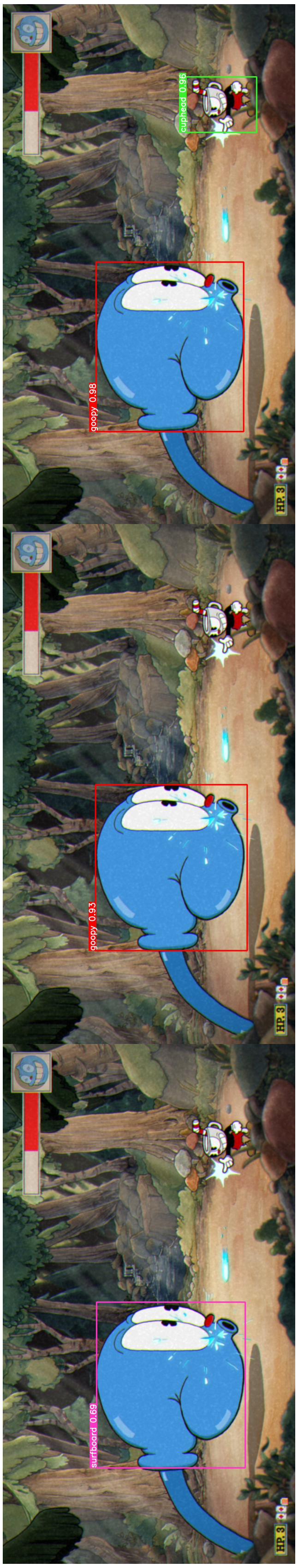


Figure 8.10: Detections made by YOLO in the 7th testing image from the game. From left to right, the pre-trained YOLOv5x, the model after the first training and the last model detections are shown.

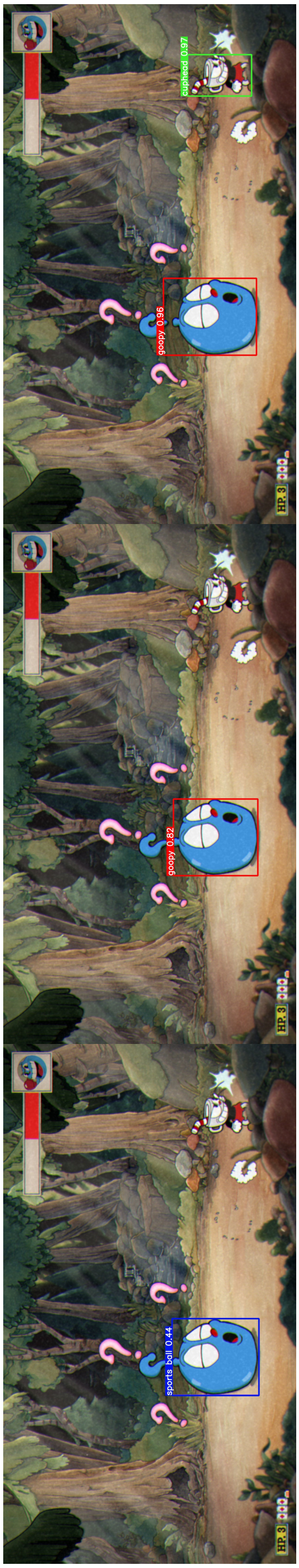


Figure 8.11: Detections made by YOLO in the 8th testing image from the game. From left to right, the pre-trained YOLOv5x, the model after the first training and the last model detections are shown.

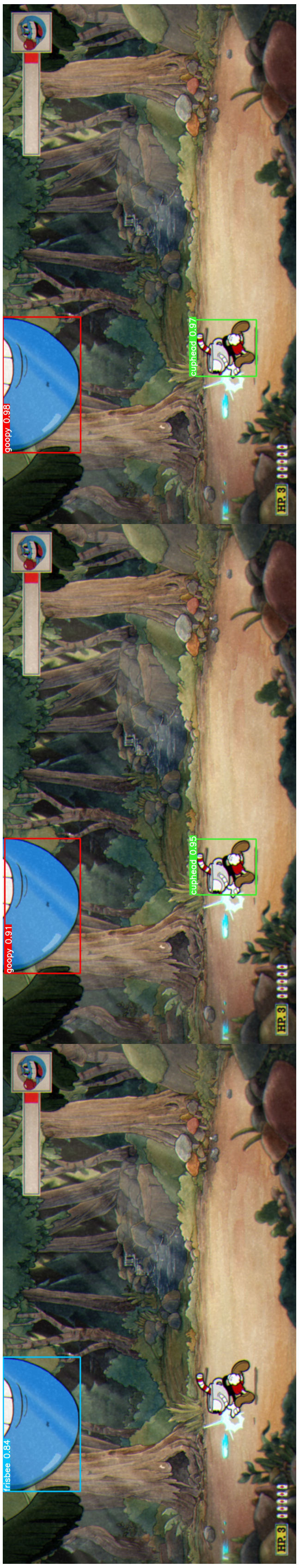


Figure 8.12: Detections made by YOLO in the 9th testing image from the game. From left to right, the pre-trained YOLOv5x, the model after the first training and the last model detections are shown.

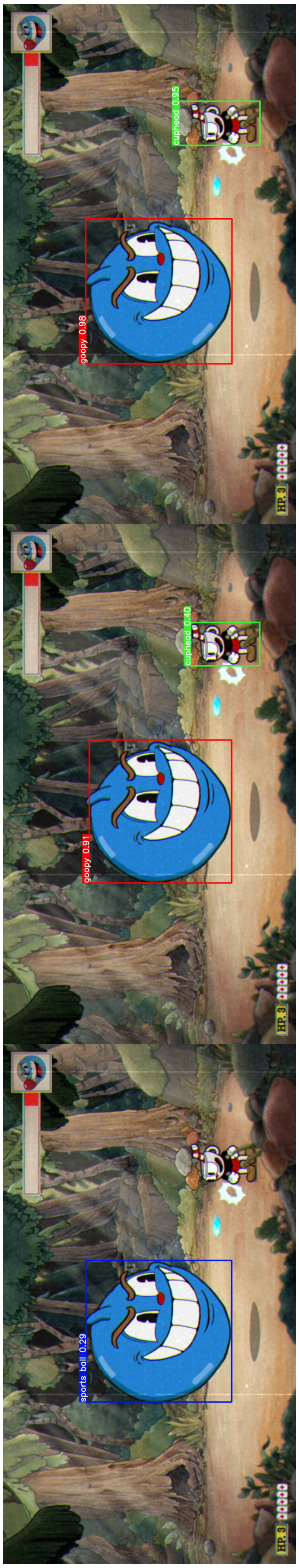


Figure 8.13: Detections made by YOLO in the 10th testing image from the game. From left to right, the pre-trained YOLOv5x, the model after the first training and the last model detections are shown.

8.2 Experiments with the integrated modules

Finally and with the object detection module finished and the defined and implemented rules, all the modules were put together in order to test the ability of the AI-based strategy to defeat Goopy.

As Figure 5.2 shows, in the final setup both the game window and the window where the frames with the detections are shown at the same time. For every frame in the game, the whole window is captured containing the frame of the current state of the game, along with the top bar and some other few pixels on the sides. After measuring the size of the top bar and the pixels on the sides, the image is processed to remove those pixels leaving only the frame with the content of the game. This frame is given as input to the trained YOLO network which is going to be in charge of making detections in the image. Next, the detections are converted into the YOLO input format and only the detections of frames where exactly two detections are made are taken into account for making a decision of which key or key sequence needs to be pressed. Regardless of the number of detections, every frame with its bounding boxes is shown in the window of the lower right corner, while the top left window shows the game window without any detections.

Algorithm 3 shows the steps of the whole process at each state of the game. If the detection that has been made in a certain state of the game has two detections a decision will be made for pressing the needed key or set of keys. On the other hand, if the number of detections is less or greater than two that frame wont be processed and we will move to the next state without taking any action.

Algorithm 3 Pseudocode of the algorithm for making decisions and taking actions

```
while True do
    press the shooting key
    capture the whole window of the current state of the game
    process the capture for leaving only the in-game content
    make detections with YOLO
    convert detection results into the YOLO input format
    if there are exactly two detections then
        make the decision of which action to take based on detections
        press the corresponding key or set of keys
    end if
    show the frame of the current state with the detections on
    if 'q' key is pressed then
        release all the used keys
        break from the loop
    end if
end while
```

After putting all the modules together, a test was made for evaluating the performance of the whole system. A total of 10 games were played, all using the same hardware mentioned in Chapter 5.2. In two of the games the agent managed to win the fight whereas the 8 remaining games turned out to be a defeat. In the first two games Cuphaead took the three hit points needed for being defeated and the first win came in the 3rd game where Cuphead took two hits and after 1 minute and 9 seconds Goopy was defeated. In the next game Cuphead also won the fight after exactly 1 minute and taking only one hit point. In the next two games Cuphead lost again and in the 7th game the situation in which Cuphead could not be detected happened. After a few seconds in the fight and dashing towards the right side of the screen while being down, Cuphead was not detected and the fight was aborted after taking only one hit point. In the remaining fights Cuphead also lost by taking all three health points. The statistics of the 10 games are shown in Table 8.1

Table 8.1: Statistics of the 10 games played by the agent. Each row represents a different game along with information related to that game such as wether it won that fight or not, the issues faced, the hit points taken by Cuphead and the time taken for winning the fight.

Game	Won	Issues	Hit points	Time
1	No	None	3	-
2	No	None	3	-
3	Yes	None	2	1min 9s
4	Yes	None	1	1min 0s
5	No	None	3	-
6	No	None	3	-
7	No	Cuphead stands beyond the right limit.	1	-
8	No	None	3	-
9	No	None	3	-
10	No	None	3	-

Lastly, some latency has been observed at the time of playing the game which caused some issues. The most significant was that the window showing each frame with the detections doesn't display the frames at the same time that the window containing the game does. After a frame is shown in the game window, it takes only a few more milliseconds to show that same frame with the detections in the other window. In addition to this, it was also observed that in some states Cuphead doesn't execute the expected action or actions for that state of the game given the detections. For instance, in some cases where Cuphead is not facing Goopy, the expected behaviour is that Cuphead turns and faces Goopy. However, in these cases Cuphead remains facing in the opposite direction where Goopy is and turns after a few frames have passed.

CHAPTER 9

Conclusions

As mentioned in the introduction, the goal of the project was to build an AI approach for playing the game Cuphead using Object detection and DRL. Although the initial proposal of using DRL couldn't be executed, given that only a few simple rules were defined based on the detections, the system managed to win 2 times out of 10 and the general idea of building a system capable of playing the game was achieved.

The method used for gathering data to build the data set showed that getting labeled data without manually annotating images is viable for purposes like this. By defining a method for linking the classes of the data set that has been used for training the network and then generating labeled images containing our desired classes, we create a way of creating labeled data without having to use other tools to annotate images one by one by hand. Analyzing the classes to associate a class or a group of classes of the original data set that the network has been trained on to each object to be recognized and implementing the method for generating the labeled images may take more time than annotating a few hundred images by hand. However, once we have the whole method implemented, we could get hundreds or even thousands of annotated images in a few minutes which makes it easier for building different and larger data sets.

After all the experiments and putting all the modules together we achieved a notorious improvement at the time of detecting both Cuphead and Goopy in the game. While the pre-trained YOLO model barely detected Cuphead and struggled detecting Goopy in different scenarios in the fight, after the final training process YOLO was able to accurately detect both Cuphead and Goopy in almost all frames. It could be seen how important quality data is, as the difference between the results after the first training stage and the second training stage was huge. The improvement could be seen in scenarios where Cuphead was not moving left or right, as after the first training process, YOLO could only detect Cuphead moving left, right or while being down and after the final training process Cuphead is detected in these scenarios. Similarly, in scenarios where Goopy stays close to the left or right sides of the screen, YOLO struggled to detect him even after the first training phase and this problem was solved after the second training phase.

In the end, there are some improvements that could be done in all modules that would significantly improve the system's performance and could increase the number of times in which Goopy is defeated. First, the method for labeling the data could be improved for making the object detection even more accurate. By finding more classes to associate to each object, selecting only the detections with the most precise bounding box placements or balancing the generated data to get an even more balanced data set that contains labeled data in all stages of the fight, we could make YOLO detect the desired object in a more precise way. More precise bounding box placements in the training data means more precise detections and this could help define more precise rules for playing the game.

Also, by improving the defined rules for playing the game the times where Cuphead wins the fight would increase. As mentioned before, by improving the quality of the bounding boxes in the detections, we could define a new set of rules, more precise than the ones currently defined that would help Cuphead avoid more damage and therefore, win more games. In scenarios where Goopy increases its size, the area of the bounding box that YOLO places in the image gets bigger. Goopy makes itself bigger in the first and second phase of the fight but each of them needs to be treated differently. In the first phase, if Goopy gets bigger means that the special attack is happening and Cuphead needs to be crouched down and standing still without moving, while in the second phase Goopy has always a bigger area in it's bounding box compared to the version of Goopy at the start of the fight. Different thresholds could be defined by measuring the bounding boxes more accurately, as the Goopy in the second phase of the fight has a different size than the one in the first phases' special attack, even if both are similar.

Finally, the most evident improvement to the system could be replacing the human based rules by a DRL approach for playing the game. A DRL approach means an extra computational cost to the system but also a more precise way of automatically playing the game.

Bibliography

[1] Leonardo Jose Gunawan, Brandon Nicolas Marlim, Neil Errando Sutrisno, Risma Yulistiani, and Fredy Purnomo. Analyzing AI and the impact in video games. In *2022 4th International Conference on Cybernetics and Intelligent System (ICORIS)*, pages 1–4. IEEE, 2022. See page 1.

[2] Tanvi Rath and N Preethi. Application of AI in video games to improve game building. In *2021 10th IEEE International Conference on Communication Systems and Network Technologies (CSNT)*, pages 821–824. IEEE, 2021. See page 2.

[3] Imants Zarembo. Analysis of artificial intelligence applications for automated testing of video games. In *ENVIRONMENT. TECHNOLOGIES. RESOURCES. Proceedings of the International Scientific and Practical Conference*, volume 2, pages 170–174, 2019. See page 2.

[4] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019. See pages 2, 13.

[5] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 779–788, 2016. See pages 7, 8, and 9.

[6] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part I 14*, pages 21–37. Springer, 2016. See pages 7, 22.

[7] Xiongwei Wu, Doyen Sahoo, and Steven CH Hoi. Recent advances in deep learning for object detection. *Neurocomputing*, 396:39–64, 2020. See page 7.

[8] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in Neural Information Processing Systems*, 28, 2015. See pages 7, 21.

[9] Pedro F Felzenszwalb, Ross B Girshick, David McAllester, and Deva Ramanan. Object detection with discriminatively trained part-based models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(9):1627–1645, 2009. See page 7.

[10] Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7263–7271, 2017. See page 10.

[11] Chuyi Li, Lulu Li, Hongliang Jiang, Kaiheng Weng, Yifei Geng, Liang Li, Zaidan Ke, Qingyuan Li, Meng Cheng, Weiqiang Nie, et al. Yolov6: A single-stage object detection framework for industrial applications. *arXiv preprint arXiv:2209.02976*, 2022. See page 10.

[12] Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. Yolov7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors. *arXiv preprint arXiv:2207.02696*, 2022. See page 10.

[13] Carlos Ling, Konrad Tollmar, and Linus Gisslén. Using deep convolutional neural networks to detect rendered glitches in video games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 16, pages 66–73, 2020. See page 13.

[14] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient CNN architecture design. In *Proceedings of the European Conference on Computer Vision (ECCV)*, September 2018. See page 13.

[15] Alexander Watson. Deep learning techniques for super-resolution in video games. *arXiv preprint arXiv:2012.09810*, 2020. See page 13.

[16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016. See page 14.

[17] Guillaume Maurice Jean-Bernard Chaslot Chaslot. *Monte-Carlo tree search*, volume 24. Maastricht University, 2010. See page 14.

[18] Jovana Markovska and Domen Šoberl. Deep reinforcement learning compared to human performance in playing video games. 2022. See page 14.

[19] Jesse Clifton and Eric Laber. Q-learning: Theory and applications. *Annual Review of Statistics and Its Application*, 7:279–301, 2020. See page 14.

[20] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016. See pages 14, 15.

[21] Ruben Rodriguez Torrado, Philip Bontrager, Julian Togelius, Jialin Liu, and Diego Perez-Liebana. Deep reinforcement learning for general video game AI. In *Computational Intelligence and Games (CIG), 2018 IEEE Conference on*. IEEE, 2018. See pages 14, 15.

[22] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The Arcade Learning Environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013. See page 14.

[23] J Terry, Benjamin Black, Nathaniel Grammel, Mario Jayakumar, Ananth Hari, Ryan Sullivan, Luis S Santos, Clemens Dieffendahl, Caroline Horsch, Rodrigo Perez-Vicente, et al. Pettingzoo: Gym for multi-agent Reinforcement Learning. *Advances in Neural Information Processing Systems*, 34:15032–15043, 2021. See page 14.

[24] Eddah K Sure and Xiaofeng Wang. A deep reinforcement learning agent for general video game ai framework games. In *2022 IEEE International Conference on Artificial*

Intelligence and Computer Applications (ICAICA), pages 182–186. IEEE, 2022. See page 15.

[25] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017. See page 21.

[26] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017. See page 22.