Imitating Reactive Human Behaviours in Games Using Neural Networks

Manuel Alejandro Cercós Pérez

Final Degree Work

Bachelor's Degree in
Video Game Design and Development

Universitat Jaume I

July 6, 2020

Supervised by: Luis Amable García Fernández.

To Pau and Celia

Acknowledgments

First of all, I would like to thank my Final Degree Work supervisor, Luis Amable García for his help in planning this project

Miguel Chover, for helping me choosing the topic and teaching me how to investigate.

Miguel Blanco and Jon Hodei Martínez, for sharing ideas of their projects and discussing with me about ML Agents.

I also would like to thank Sergio Barrachina Mir and José Vte. Martí Avilés for their inspiring LaTeX template for writing the Final Degree Work report, which I have used as a starting point in writing this report.

Abstract

Deep learning has allowed to create neural networks that can play any game almost optimally. However, not so many have been trained to play like humans, or more concretely, like one specific person. Most people have recognizable ways of playing specific games, and imitating those behaviors would allow to create bots that don't appear to be artificially generated. Also, by imitating one person behaviors it would be easy to create bots that play at the same level of quality.

This document, which is a Final Degree Work report for the Bachelor's Degree in Video Game Design and Development, presents some techniques to create neural networks that can imitate human behaviors using Unity's ML Agents SDK, an analysis on what behaviors can be modeled more precisely, what are the training costs and how good are the results.

Contents

CHAPTER 1

Introduction

Contents

Neural networks have taken a big step in artificial intelligence, allowing to solve complex problems like playing games optimally, generating images or supressing noise. However, other fields are still to be explored, like the one that we treat in this document: imitating human behaviors.

In this document, we will detail the steps for the realization of a neural network model capable of imitating real player behaviors in simple games, from the programming of the game that we will use as a test case to the analysis of results.

To obtain the dataset, we will use Unity3D to program a shooter-type game with no player movement on the stage (Point-and-Click), and random targets. In order to obtain a reasonable dataset to train the neural network, an NPC behaviour will be programmed to simulate a large amount of games as the player. That NPC would have recognizable characteristics in his way of playing. We will use the ML Agents framework, which allows to simulate games and train directly from them and generate demos for imitation learning.

The dataset used as input for the neural network is formed by in-game simplified frames and the key/mouse inputs made in that moment (mouse movement and keys pressed). Using that dataset, a neural network will be trained to mimic that NPC by receiving simplified game frames as input, with the objective of obtaining a neural network that visually reproduces that NPC's way of playing.

1.1 Work Motivation

This topic was chosen because I found interesting the potential of neural networks in solving difficult problems and how well they solve them. Also, I wanted to learn to use neural networks and make them, challenging myself to carry out a complex project. Since I had some experience using the ML-Agents environment for Unity, it could serve as a foundation to develop neural networks using reinforcement learning to solve the problem presented in this document.

On the other hand, one of the main motivations of this work was to conduct a research article (in parallel, with the Study and Research at the UJI program). I found interesting that almost every scientific work related to neural networks was oriented to learn to play optimally specific games, but almost none had the objective of imitating real players in that games [2] [3], so I decided to investigate deeply in that area.

1.2 Objectives

The main objectives are the following:

- Program a simple shooting game using Unity3D.
- Obtain in-game information from Unity3D to train an agent
- Obtain a trained neural network that can reproduce the movements and reactions of one specific player.
- Develop and define a framework that allows to imitate real human players in more complex video games having their games' data (video games where you can walk or move in many other ways, games with more complex graphics or a larger amount of controls).

1.3 Environment and Initial State

This project was intended to be developed with one PC, and trained at the research laboratory of my supervisor in this TFG to speed up the training process. However, the fact of not being able to use the laboratory equipment due to the closure of the university because of COVID-19 delayed some steps of this project.

CHAPTER 2

Planning and resources evaluation

Contents

The following tasks would be performed iteratively (see Figure 2.1): First, a simple game will be developed to serve as an example case for the model. Then, different pre-programmed NPC behaviors will be used as training examples. Using ML Agents, a neural network will be trained to imitate that NPC. The results obtained in each training session will be analized to extract conclusions on why the trained network performs well or not. Then, we will start again with other NPC or neural network structures, until we gather enough data to extract conclusions and standardize a general model.

2.1 Planning

The simple game programming is expected to take 10 hours of work. Then, several neural networks will be trained iteratively and analyzed. At the end, we expect to standardize a framework for more complex behaviors. The memory will be written during all the process.

2.1.1 Simple game creation

Using Unity3D, the first step is to program a simple 3D video game that serves as the basis for this project.

The game devised is of the "shooter" type, although in this case it could be compared more with a "point-and-click". The player can only move the view with the mouse and

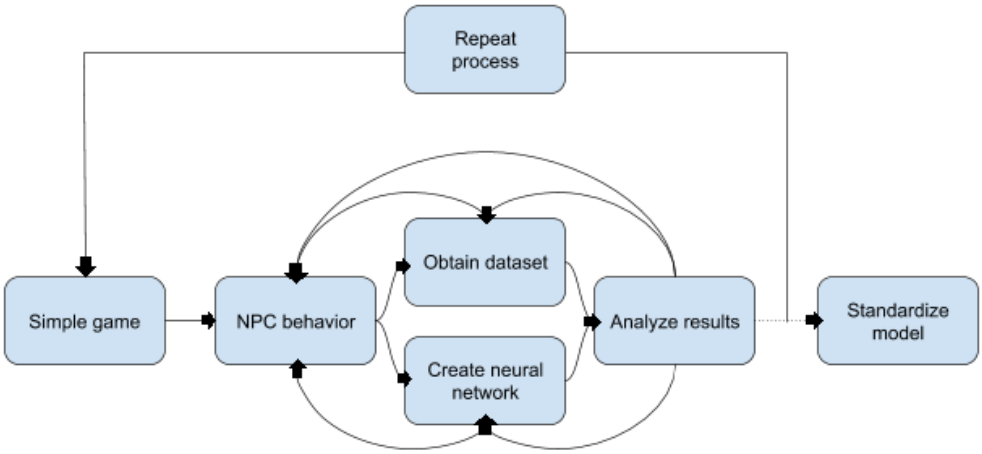


Figure 2.1: Planning graph

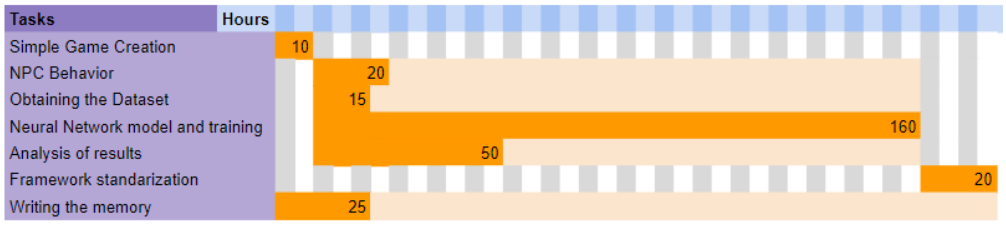


Figure 2.2: Initial planning

"shoot" by clicking. The game scenario would be very simple to facilitate training (see Figure 2.3)

The game screen would have a ratio of 16:9. In this image, the lighter colors correspond to the background and the black with the "enemies" to which you must click. To ease debugging, there would be a white point to represent the sight, which must be aligned with the target to be shot. The agent would receive a visual observation (with no GUI elements) as input.

Like in most shooters, the enemies would have a fixed shape, but the size they look would depend on their distance and inclination with respect to the player, but in this case they would not move. They disappear by clicking over them, and after a few seconds a new enemy appears in another position (the number of enemies will be limited). The player, when moving the mouse, would move the camera by way of rotation: the sight remains static in the center of the image but the rest of the elements move in the opposite direction of the mouse movement. In the case of vertical movements, the rotation has stops at the zenith and nadir angles, so that you cannot see “upside down”.

Figure 2.3: Actual screenshot of the final game

2.1.2 NPC Behavior

Within the same game, a very simple NPC will be programmed with sensors (rays or colliders) that can play games from the previously described game in random conditions. There are many parameters that could define the behavior of different bots, some examples are:

- Reaction time
- Speed to which the mouse moves
- How many clicks are made on an enemy
- Precision of its movements
- "Tics" (for example, sudden changes in direction)
- Movements made when not seeing enemies
- Order in which you select enemies from the same screen

In addition, to represent the randomness that real human behavior would entail, each behavior would have a range of imperfection, which could be represented with more parameters such as a random range or a standard deviation.

The bot, unlike the one we intend to create to imitate him, would receive information directly from the stage with sensors such as lightning or a frontal collider that detects collisions with the enemies he has in front of him.

2.1.3 Obtaining the Dataset

Since we intend to imitate a behavior based on the premise that the one who performs it could be a human, the dataset to train must be composed of the information that a human could have of a game: what is seen at each moment, what he has seen in the previous instant and the actions he has performed in that previous instant. With all these data, the action to be performed at this precise moment would be obtained.

The fact of receiving previous information allows to model behaviors with reaction times: it is impossible for a human to react in a frame. The actions allow the movements to have coherence: there could be an interval in which no enemies were seen on the screen, remembering the previous actions you could know if it was moving to the left, the right or it was still.

To obtain this data, it would be necessary to run the game (having the previous bot playing it) and save each of the images, in addition to the inputs that are being made (in Unity, this can be found in "Input.GetAxis" in the case of the movement, and in "Input GetMouseButton or GetMouseButtonDown" in the case of clicks). The Inputs could be saved in a text file, a table or a csv file.

To save the frames, RenderTextures are obtained from the main camera, "Image. EncodeToPNG()" is used to format the image and certain functions of the File class to save files (such as WriteAllBytes). Each frame and input would be assigned a numerical code to obtain them together. Saving files would be executed in LateUpdate(), which is recommended as it is executed after the update of each frame and before the next.

2.1.4 Neural network model and training

Training the neural networks is what takes most of the time. This process includes tuning parameters, designing and balancing rewards and the training itself. In this part of the process some methods to model the behaviors of the NPCs described in section 2.1.2 are designed.

The proposed neural network, in simple terms, is a classification network [10]: it receives images (and previous actions) as input and returns the action (or actions) to be performed in the current frame.

The input consists of the current frame, a certain number of previous frames and the actions performed on those previous frames. These 3 elements are subjected to convolutions [1] separately (or other kind of compressions) to simplify the information. This information is then processed in a simple network with at least one hidden layer, returning as output the expected action of the current frame, composed of: mouse click (true or false), horizontal and vertical (these last 2 are normalized values representing the speed of movement in the 2 axes, that is, the movement of the mouse). Simpler neural networks could have different architectures, inputs or outputs.

[1] Convolution is a mathematical operation on two functions that produces a third function expressing how the shape of one is modified by the other.

Since in ML Agents all the actions have to be coherent (either discrete or continuous) and the mouse movement needs to be continuous, discrete actions [2] (clicks or keyboard inputs) would be expressed as a probability of performing it (from 0 to 1).

To train the network, it will be fed with previous frames and actions, the returned action will be compared with the real one that has been performed in that frame, and the error corrected using gradient descent [8].

2.1.5 Analysis of results

Once the neural network is trained, it is necessary to incorporate it into Unity so that it can play games and receive inputs in real time.

A first way to check the quality of the behavior generated is visually: the neural network must not only show more or less "intelligent" actions, but must resemble the original. If the behavior doesn't look anything like the original in all cases, it would be discarded.

If they seem similar, we would make graphs with the actions performed in time (x = time, y = mouse speed on an axis) for each of the 3 actions in order to check if both the reactions and the speed of the movements fall within the range of imprecision that we have defined. From multiple simulations in the same conditions, we could know if the behavior is really similar or not.

After obtaining a result, we would either try to improve it by changing the dataset or the network structure; or repeat the process with another different NPC.

2.1.6 Framework standardization

In the case that we succeed in obtaining similar behaviors (for one or more agents with different behaviors), the next step would be to standardize this method to be able to apply it to more complex games, in which the image has many more elements or there are many more actions available. In that step we would discuss issues such as the feasibility of the model, the accuracy of the results, the cost of training in other cases or the differences between the neural networks in each case (number of layers and neurons per layer).

Also, whether the trained agents imitate the behaviours well or not, we would discuss why that techniques do or do not solve well the imitation problem and what could be done to improve the results.

2.2 Resource Evaluation

The development is intended to be done in an average home PC, but as stated in section 1.3 it would be better if a laboratory could be used in parallel. It could be done in

[2] In ML Agents, continuous actions are float numbers. Discrete actions are expressed as an integer, where each possible integer represents a different action.

reasonable amounts of time (1-4 hours of training per model) while also covering other tasks in parallel.

The only economic cost would be the energy spent in training the neural networks, which could be little high but viable.

In order to execute any of the trained models that were saved in this project, the only requirement is using **Unity 3D 2019.2.12.f1**. However, to train other neural networks the following requirements must be met:

- Python 3.6
- Tensorflow 1.15
- mlagents 0.11
- keras 2.3.1

These other requeriments are optional if the training is executed CPU-only, but needed to speed up the process by using GPU:

- Cuda 10.0
- CuDNN 7.6.5

To end with, the system used to train the models has these specifications. They are not a minimum requirement, but can be used as a reference point:

- OS: Windows 10
- CPU: Intel Core i7-4790
- GPU: NVIDIA GeForce GTX 1050
- RAM: 24 GB

CHAPTER 3

System Analysis and Design

Contents

In this sections, we will detail which requirements must be acomplished to consider that the neural network solves its task correctly. Also, we will specify the system used to develop this work and its minimum specifications.

3.1 Requirement Analysis

3.1.1 Functional Requirements

The following must be fulfilled to provide a realistic imitation:

- The neural network will be able to play indepently the game
- The network will receive as input what is seeing
- The network will receive as input immediate past actions and images
- The network will output the action/s made
- The network will adapt its actions to reaction times of the NPC being imitated
- The network and the NPC would not be differentiated when playing

3.1.2 Non-functional Requirements

- The network will be scalable to more complex problems
- The network will be decently trained in reasonable time
- The network will be sample efficient when training

3.2 System Design

To train an agent, an environment is needed. The training is executed in a game build, where the custom bot plays the game and the neural network tries to guess its moves. After trained, the neural network can be fed into the Agent class and play the game by itself in the editor. The Figure 3.1 shows the class diagram of the environment:

The scene has one custom Academy (ShootAcademy), a Camera and a Spawner that creates the enemies randomly in execution time. The camera contains one custom bot (the abstract class allows to create new bots and test them without changing references), a movement handler (CameraMovement) which handles the moves made by the bot or the neural network, a custom Agent that generates and executes trained neural networks, a visual sensor and a Demo Recorder (when activated, it generates a dataset for Imitation learning).

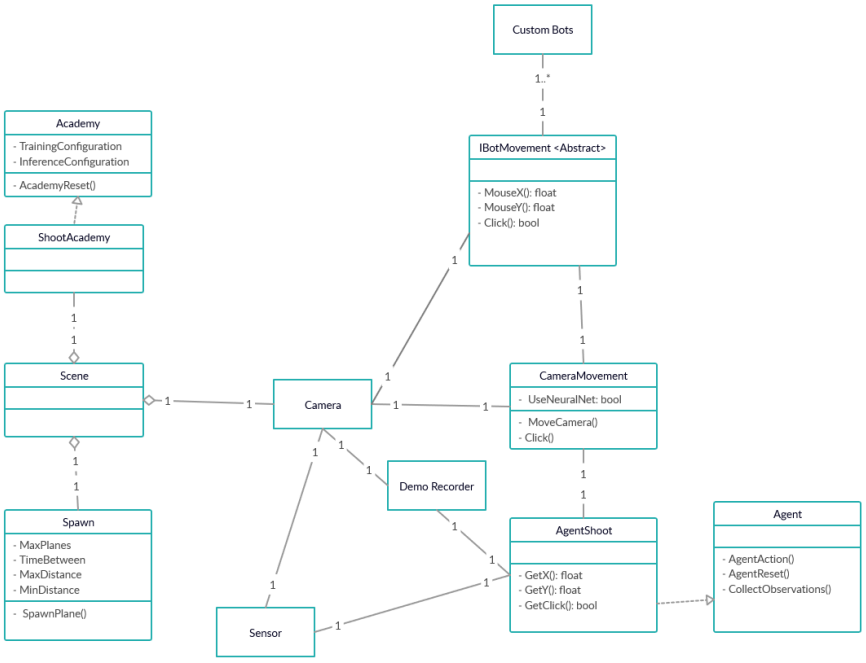


Figure 3.1: Class diagram of the training environment

The classes Academy, Agent, DemoRecorder and Sensor are provided by the ML Agents SDK [8].

3.3 System Architecture

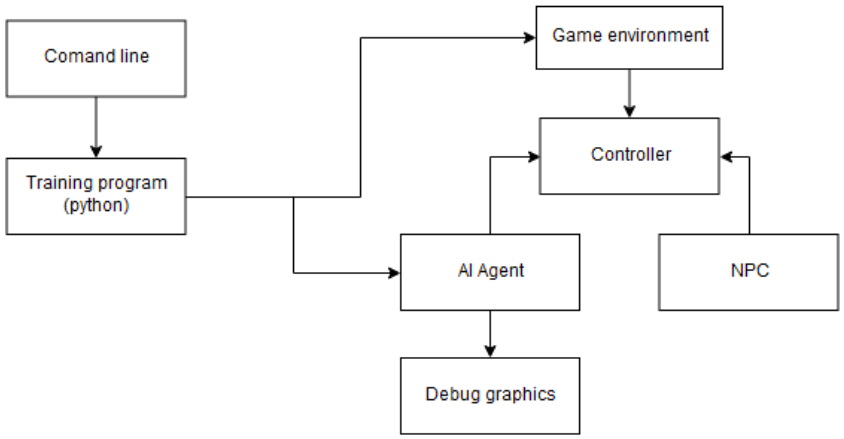


Figure 3.2: Diagram of the system architecture

ML Agents works using an environment in Unity to train neural networks. The networks are trained in a python program (executed using the command line) that communicates with that environment. In the environment, there will be a controller that allows either the NPC or the trained agent to play the game. A graph will be displayed in the UI to see live how well the agent is performing.

3.4 Interface Design

To acknowledge how well is the neural net adapting to the bot movement, the guesses of the bot are displayed in a real time graph alongside the bot's real move. There is also a centered sight to better see how the bot behaves. Figure 3.3 shows a complete game window while training.

The graph displays 3 main lines (see Figure 3.4a): the red one is the move made by the bot, the blue and green ones are the highest and lowest guess of the bot (since the bot's movements are not perfect, these two lines can vary from being almost touching to being really wide). Other 3 lines are used to show the weighted average move and its standard deviation.

The image at the bottom left is what the neural network receives as input. It helps verify that everything works correctly (Figure 3.4b): if the agent action and the NPC action look similar, then it's very likely that the agent would perform well [1].

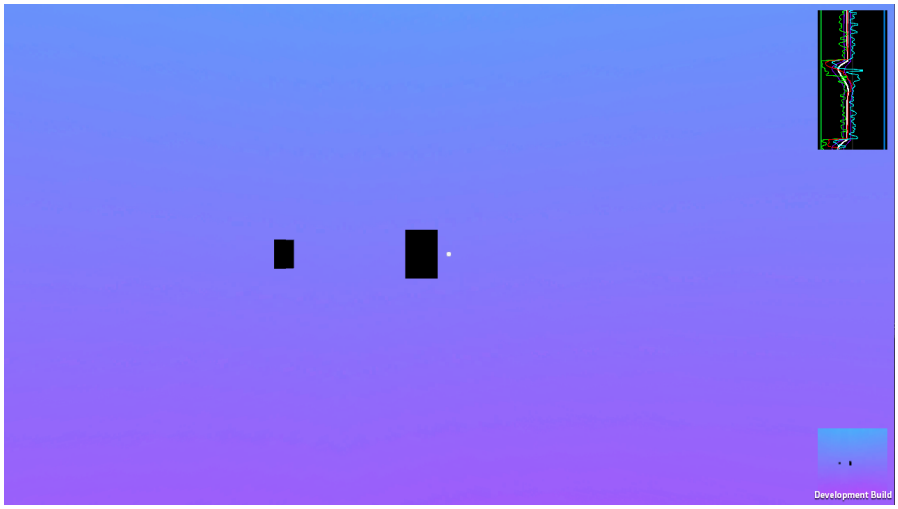
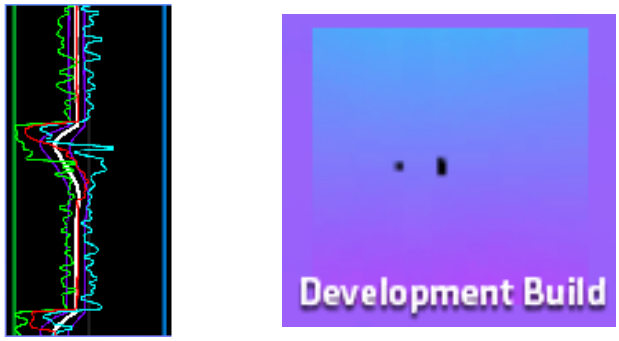


Figure 3.3: Complete screen of the game



(a) Move Graph (b) Input Vision

Figure 3.4: Debugging UI elements

[1] Even though sometimes the agent can look like the NPC that imitates during training, when playing it can behave differently depending on the observations it receives. In section 4.2.5 there is an example on why this phenomenon can happen.

Chapter 4

Work Development and Results

Contents

In this section, we will detail the development of the project starting from the game creation, and then describing each step of increased complexity of the behaviour to imitate, as well as the results obtained in each step.

To avoid confusion, we will refer to the programmed behaviors that we want to imitate as "bot" or "NPC", and the generated neural networks that have to learn to imitate that bot will be called "AI" or "agent".

4.1 Game Development

The game environment needs to include the ML Agents' classes (Agent and Academy) to train and play the game using the network. The game structure is the one shown in section 3.2.

The camera has an NPC and a trained AI attached: One of them controls its movement automatically, and who does that can be changed in play time. Both of these classes have getters to 3 variables that correspond to the possible movements: mouse X movement, mouse Y movement and mouse click, which are used to rotate and shoot.

The Spawner creates randomly and saves references of black planes in the scene, which correspond to the enemies.

To end with, a Debug Canvas has been added as interface, which draws lines with the movements made and the ones expected by the neural network. This allows to see how well the neural network is training.

4.2 Reactive Behaviors

The first human behavior we would analize is reactions, which can be defined as "sudden changes produced by a stimulus". To model this behavior, we created a Bot with the following requirements:

- While not seeing any target, it moves to the left uniformly
- When a target enters the screen, it reacts moving fast towards its center, then continues moving as normal

At this first step, the bot will only move horizontally, and it will be considered that is always clicking (so the targets would be destroyed whenever the sight touches them).

4.2.1 Training with Proximal Policy Optimization (PPO)

Proximal Policy Optimization [4] is the first and most simple reinforcement learning algorithm provided by ML Agents [8]. It uses a neural network to approximate the ideal function that maps an agent's observation to the best action it can take in a given state. Also, it is the fastest algorithm of all provided by ML Agents.

In the following subsections, some training related issues will be taken into account. At first, we will train our models using this policy (PPO).

4.2.2 Unnecessary actions

It is important not to add more actions than needed, since they would slow down the training process considerably. Even though it is possible to move in X and Y and perform clicks, since the bot only moves using the one axis, any additional action would add much noise to the AI.

That is caused because when training the AI is overfitted with demonstrations with Y movements of exactly 0, and when that AI is playing any slight up or down movement would go inside untrained cases, and then causing unexpected behaviors.

Therefore, in this case the neural network would only have 1 action output: the X axis movement.

4.2.3 Rewards based in tolerable range

Our first reward approach is based in tolerable ranges. This consists in giving positive rewards when the distance between the guess and the real move is less than the tolerable range:

-The maximum reward is given (1) if it the distance is exactly 0

-A reward of -1 is given when the distance is 2, which is the maximum distance possible (NPC moving at maximum speed in one direction and the AI in the opposite direction).

In Figure 4.1, you can see the reward function with tolerable range = 0.5. In our trainings, tolerable range was between 0.05 and 0.1: lower tolerable ranges than 0.05 caused the training to become unstable because it only got negative rewards, and higher tolerable range

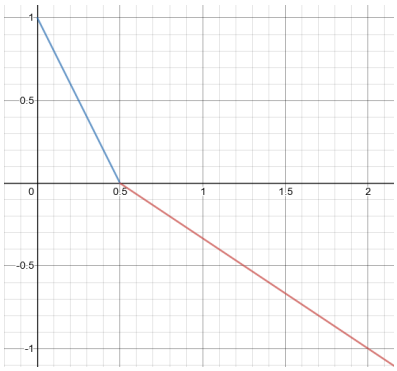


Figure 4.1: Rewards based in tolerable range.

Models trained using these rewards are not very time-efficient. If the tolerable range is too big (the agent receives positive rewards easily), the model doesn't fit the movement; if it is too small (receives negative rewards), the agent tends to stay only in the average movement, and doesn't react at all. That happens because the average is the point with biggest chance of reward (the agent is only punished when an impulse occurs).

Curriculum learning [1] does not improve the training performance since with the initial less exigent punishments, the neural network learns much slower than with higher ones.

Figure 4.2 shows some of the success cases. From left to right, the first image shows how the neural network model adapts to the idle movement and the impulses, after 180000 training steps (4100s). The middle image displays an imperfect behavior of the same model when successive impulses occur. The right image is the same model trained longer time (10000s, 435000 steps), and how it tends to excessively smooth its impulsive movements. The causes of these two problems (successive impulses and smoothing) are discussed in section 4.2.4.

4.2.4 Determinism of the behavior

Since at this point the neural network does not receive past events as input (neither moves or images), the movements performed by the bot have to be deterministic in order to train correctly: that is, given a frame, the bot would react with the exact same move every time (in the impulses, the default movement has a bit of noise in it). However, by how the bot was made it always took as objective the first image that it had seen, until destroyed.

[1] Curriculum learning is a technique provided by ML Agents to train complex behaviors with consecutive lessons that increase in difficulty. That way, when the agent learns one task it goes on to the next lesson.

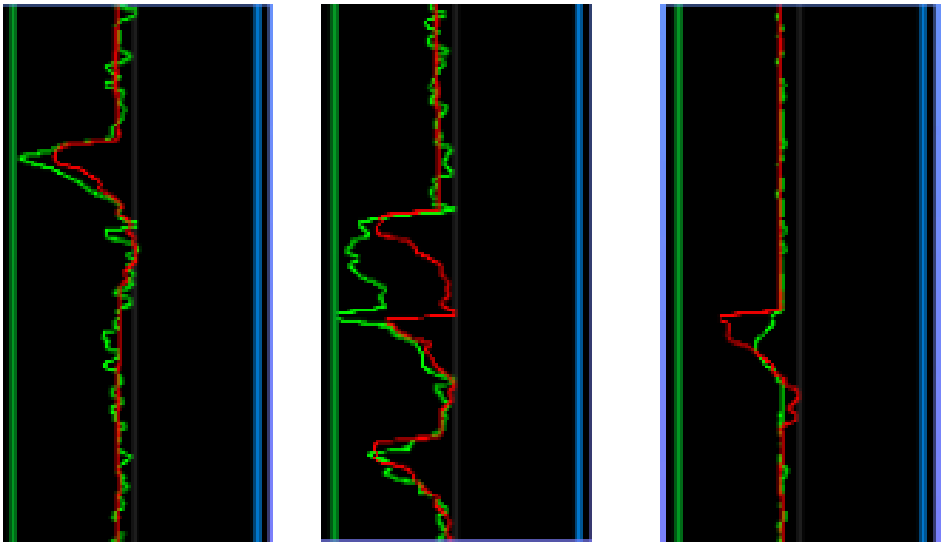


Figure 4.2: Bot movement (red) and neural network movement (green)

In some special cases, when a new target spawns nearer to the sight than the current objective, the bot would not change the target order, and so it would behave differently depending on the context, as you can see in figure 4.3. These repeated events cause the neural network to confuse when multiple targets are on screen, and if trained longer, it tends to do smaller impulses until only moving in the average move.

This issue is solved by making the bot behavior deterministic or adding a movement memory.

4.2.5 Movement memory

In order to prepare the bot to have reaction times, 25 previous moves distributed in the last 2 seconds are added as observations. What move is added as observation is critical.

If the real bot movement is added, the bot reaches high rewards very quickly but doesn't learn to imitate the bot: that's because the neural network learns to "mimic" the last move made by the bot, so it has high chance of reward with only one important observation. When playing the game with the trained neural network, it would not move (in the beginning, all the previous moves are 0) until it starts moving in one or another direction at maximum speed (See Figure 4.4). This happens when the movement starts increasing in value due to impressions in the returned action of the neural network that make it believe that it is accelerating in movement.

When using the neural network movement, it learns like before: correctly but a bit slower. However, the previous moves tend to have noise at first, and the network could learn to ignore them.

A better approximation would be interpolating the real move with the neural network's one: at the start, the movement added as observation in the next frames would be

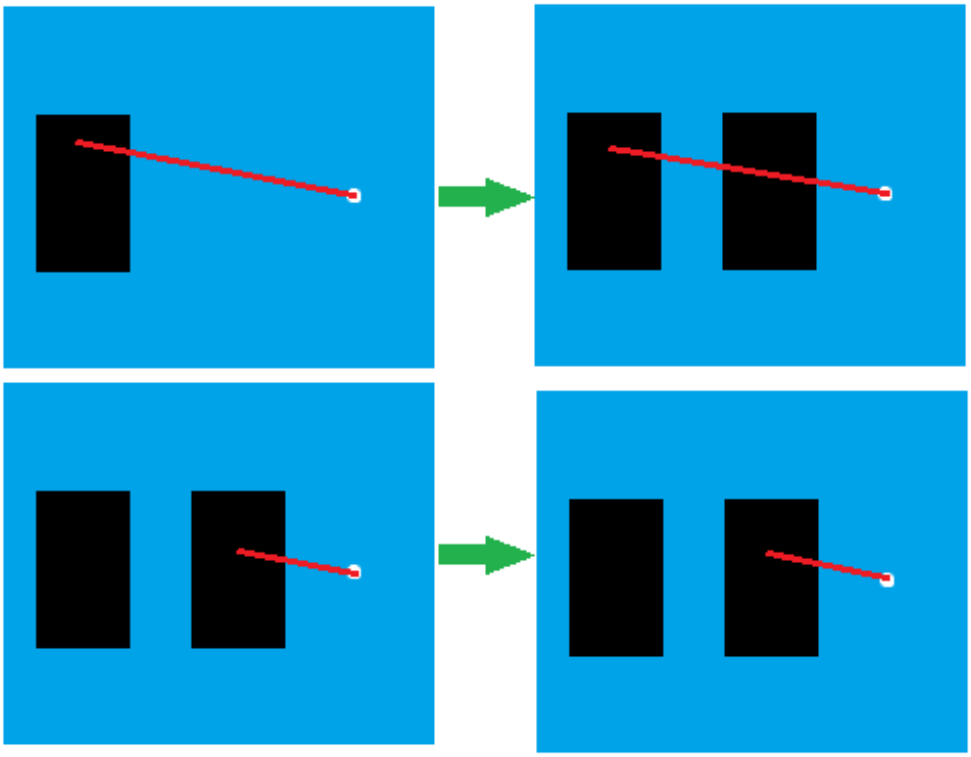


Figure 4.3: 2 situations that lead to different actions with the same frame

the NPC move. When the AI starts learning to adapt to the context (previous moves), the movement added would be an interpolation between the AI and the NPC movement (which would cause the AI to react in time), until the original AI moves are the ones added as observation. This can be made using curriculum learning: the lesson with least difficulty is the one where the AI receives past movements of the NPC as observations, and the hardest one where it receives its own movements as observation.

4.2.6 Rewards based in standard deviation

Since trained models using the methods explained in the previous sections tend to return the most common value, movements with more noise or imprecissions would not be produced correctly by the AI: when training, the AI could guess a move some units below the average of the previous moves but the NPC could have done a move the same units above the average, causing the network to be penalized, and causing the AI movement to converge to the average movement. To model these kind of noises more precisely, the actions and rewards should be changed.

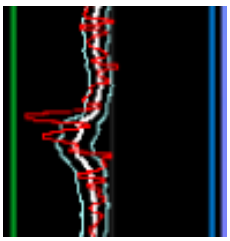


Figure 4.5: Weighted average and standard deviations of an irregular movement

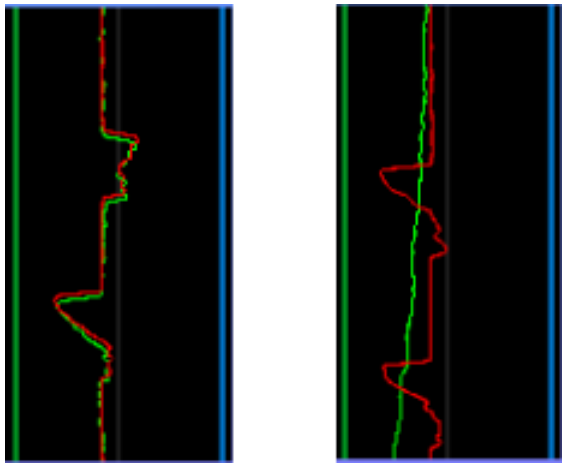
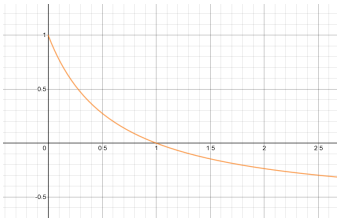
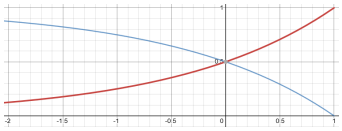


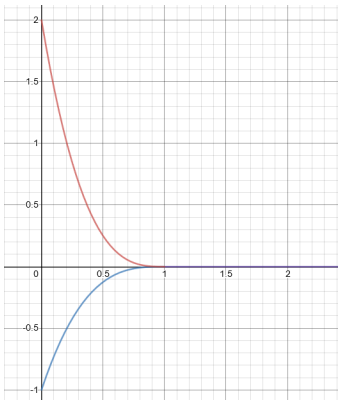
Figure 4.4: A badly trained model while training (up) and playing (down)



(a) Points by precision



(b) Reward and punishment factors by coherence



(c) Deviation factor divided by Pow(2,(1-coherence)), with coherence=1

Figure 4.6: Shape of the 3 parameters used for rewards

In this section we propose a reward system based on standard deviations (Figure 4.5): the relation between standard deviation, average and the actual move would determine how coherent is a move in a given context.

The *coherence* of a movement can be defined as how centered it is, in relation to the average. A movement with maximum coherence (1) would be the exact average, a movement at a standard deviation distance would have coherence 0, and movements outside of the standard deviations would be considered "incoherent". Then, default movement with noise would be coherent moves, and impulses would be incoherent.

To model the behavior, the agent would do 2 actions instead of one: a maximum and a minimum guess. The more precisely it encloses the real movement, the higher reward it gets; if it fails enclosing it, a punish is given.

Coherent moves give higher punishments if failing and smaller rewards, and incoherent moves (impulses) give high rewards. Given a maximum and minimum values (actions provided by the neural network), the real move, the average of the last 25 moves, its standard deviation and the coherence parameter explained in this section, the reward system follows these rules:

- *Coherence* is inversely proportional to the *reward factor*, and directly proportional to the *punish factor*: high coherence means lower rewards and higher punishes.
- A movement has higher *precision* if it's centered between the maximum and minimum, and less if it's outside. The *precision* is relative to the difference between the maximum and minimum values
- The *deviation factor* is calculated dividing the real standard deviation with the agent one (max - min)
- The *deviation factor* is inversely proportional to the coherence
- All the values are clamped to avoid excessively high rewards/punishments or zero division errors
- The final reward is calculated multiplying *precision * factor * deviation factor*

In figure 4.6 you can see the shapes of each parameter functions, used to calculate the final reward.

In this first approach using maximum and minimum estimations, the network doesn't fit well the movement: it encloses large areas continuously (See Figure 4.7). That could happen because it receives less punishment by enclosing the coherent movement than by fitting and sometimes failing, and also receives rewards from incoherent movement. Thus, the neural network finds an equilibrium enclosing wide ranges to catch high rewards from incoherent moves, at the cost of getting fewer rewards from coherent moves (which were low by definition) and not exposing to any punishment from failing to encase coherent moves.

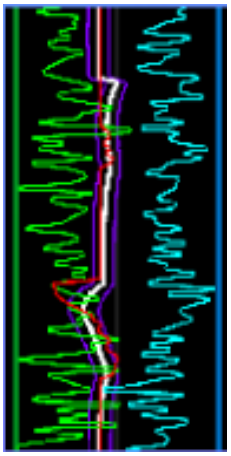


Figure 4.7: Trained model using SD rewards

4.2.7 Rewards based in movement coherence

Since last reward system didn't make the agent learn correctly, we need to change the rewards in a way that it worries about adjusting to the predictable coherent movement while also worrying about not to miss any impulsive incoherent move.

Rewards based on movement coherence are a simplification of the reward system exposed on section 4.2.6, where coherent moves can only punish and incoherent moves can only give rewards. These motivates the agent to receive the least punishments by enclosing coherent moves, but also to take profit of potential rewards of incoherent moves.

The punishments in coherent moves are calculated multiplying the punish factor, the coherence (0..1) and the relative distance between standard deviations, maximum and minimum.

The rewards in incoherent moves are calculated using the reward factor, the opposite to coherence and the precision factor shown in section 4.2.6.

Models trained with this system adapt better to both coherent and incoherent moves, however they need high learning rate and at least 300000 steps to see acceptable results (see Figure 4.8). Nevertheless, a learning rate higher [2] than 1e-2 can easily lead to unstable models that don't learn at all.

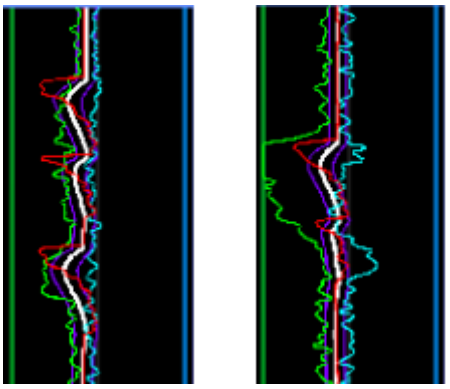


Figure 4.8: Coherence-based models with and different learning rate: left=2e-3, right=8e-3

4.2.8 PPO hyperparameters

To sum up, the trained models that got decent performance had the following hyperparameters (they also depend on the reward system):

batch size	32 or 1024	beta	5.0e-3..8.0e-3
buffer size	256 or 8196	epsilon	0.3
hidden units	mostly 256	learning rate	1.0e-4..2.0e-3
learning rate schedule	mostly linear	normalize	false
num layers	mostly 1	num epoch	3-5
summary freq	1000	time horizon	5-256
extrinsic strength	1.0	extrinsic gamma	0.8..0.9
curiosity strength (opt.)	0.01..0.1	curiosity gamma (opt.)	0.8..0.99
curiosity encoding size (opt.)	128-256	gail strength	0.01 (not recommended)
gail gamma	0.95 (not rec.)	gail learning rate	0.0005 (not rec.)
gail encoding size	64 (not rec.)	gail use vail	true (not rec.)
gail use actions	true (not rec.)		

4.2.9 Training with Soft-Actor Critic (SAC)

Soft-Actor Critic [6] is the second reinforcement learning policy provided in ML-Agents. It is characterized for being more sample-efficient and can learn from past experiences. However, it also executes slower, so the time needed to train a model is very similar both with PPO and SAC. Also, its training steps can be increased more easily since the learning rate is recommended to be constant (its Q function converges naturally).

[2] Learning rate is an hyperparameter that defines the strength of each gradient descent update step

To compare new methods with SAC and PPO, we've added simple linear rewards that affect the maximum and minimum individually, in addition to rewards based in movement coherence. These give reinforcement signals when one of the lines is well positioned, even when the cummulative reward is negative. In Figure 4.9 you can see a cumulative reward comparison between an agent trained with SAC and other agent trained using PPO, with rewards based in coherence (see section 4.2.7): SAC converges to a higher reward than PPO with much less steps.

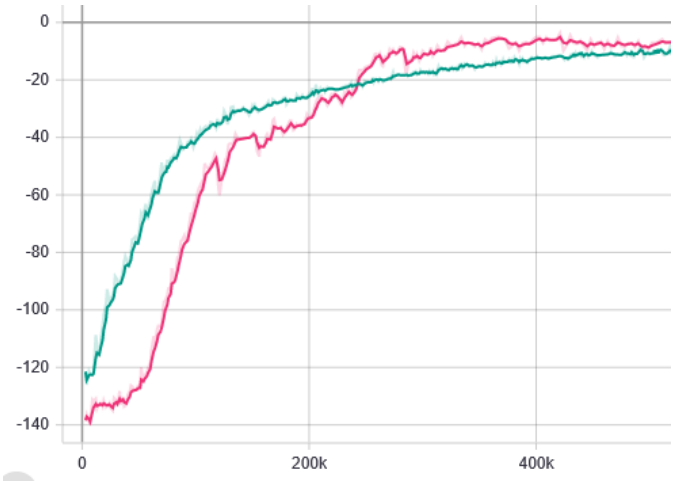


Figure 4.9: Total rewards of SAC (pink) and PPO (green).

As final result, Figure 4.10 shows a comparison between both trained neural networks: SAC adapts much better to impulses than PPO, even though PPO also manages to fit the real move between the two lines. However, when playing neither of them reacts correctly to targets that appear on the right side (mainly because it is an uncommon case). After the training both models still have much noise in their default movement, but it could be corrected by training longer or by rewarding the stability of both agent lines (maximum and minimum).

Another aspect to take into account is how both methods can be applied using GPUs to boost the training process. SAC makes better use of the GPU: by training with 3 environments in parallel the training speed doubles (being equally fast as PPO using CPU) and also improves its efficiency. PPO training using GPU and 3 environments is almost 2.5 times faster than with CPU or GPU-SAC, but is more likely to produce an application crash than any other training method (because of GPU overheating or running out of memory).

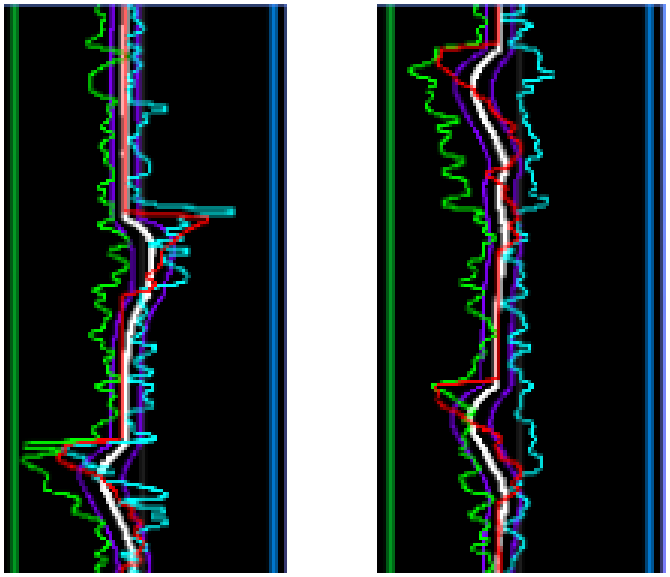


Figure 4.10: Comparison between SAC (left) and PPO (right).

4.2.10 SAC hyperparameters

These parameters were the ones used when training with SAC:

batch size	128	buffer size	200000
buffer init steps	5000	hidden units	256
init entcoef	1.0	learning rate	4.0e-4
learning rate schedule	constant	max steps	6.0e5
memory size	256	normalize	true
num update	1	train interval	5
num layers	1	time horizon	64
sequence length	128	summary freq	1000
tau	0.005	use recurrent	false
vis encode type	simple	pretraining strength	0.4
pretraining steps	20000	extrinsic strength	1.5
extrinsic gamma	0.99	curiosity strength	0.03
curiosity gamma	0.99	curiosity encoding size	128
gail strength	0.03	gail gamma	0.99
gail encoding size	128	use actions	true

4.3 Reaction time

In this section we will cover the development of bots with a behavior similar to the one presented in last section, but with delayed reactions: when the bot sees a target, it does not react instantly, but takes a few milliseconds to perform the action. This adds more complexity to the behavior and to the neural network, since it needs to receive information from previous frames (Moreover, the reaction time may not be exactly the same every time).

Even though the AIs with actions based in standard deviation (2 outputs to encapsulate a noised movement, see 4.2.6) did well modelling imprecise movements, we won't use this method in this section. That's because not only it would add more complexity to the task, but it could add much noise when moves are uncertain (if a bot reacts at 0.1-0.3 seconds, the AI would try to encapsulate a possible jump in all that range, and then if a random point in between was chosen as action each frame, the bot would not do a perfect impulsive movement. Instead, it would do a strange vibration). This feature will be solved with better reward systems (see section 4.3.2).

4.3.1 Render Textures

In order to provide the agent past observations, render textures must be used (Camera observations aren't useful in this context since they cannot provide past frames as input). ML Agents allows to provide multiple visual inputs (see 4.11), but they must follow these requirements:

- Each render texture must have the same width and height [3]
- All render textures must be the same size
- All render textures must be either grayscale or not, but there must not be render textures of each type
- The minimum size is 20x20 pixels
- Each visual input must have an unique name (We use "RenderTarget" for the current frame and "FrameXXX" for past frames)
- Each visual input's render texture should not change in execution time [4]

With these restrictions, there are two reasonable methods to manage render textures in Unity:

The first method consists in creating a Render Texture array with the desired frames in the N-1 position of the array (last frame in position 0, the 5th frame before in position 4...). The current frame isn't included in the array since it is rendered directly from a

[3]This requirement also applies when only using one render texture (at least in this ML Agents version)

[4]Since there are multiple components of the same type, they cannot be changed reliably in real time

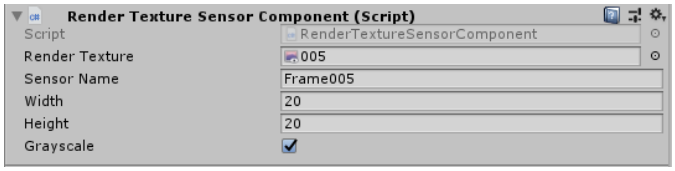


Figure 4.11: Example of a Render Sensor component.

virtual camera. The missing positions in the array must be filled with other Render Textures, even though the neural network would not receive them as input. Then, when a frame ends each render target copies the next frame (iterating the array backwards), until the current frame is rendered over the render texture at the position 0. This method is not very optimal and has some errors at the start (render textures appear empty until frame N, being N the size of the render textures array) but allows to personalize easily which frames we want to provide as input to the neural network. Also, sometimes the copying of frames can overlap (frame N is being drawn on frame N+1, but before ending the process the frame N-1 starts being drawn over frame N).

Other more optimal and safe [5] method is using a list to store previous frames like a queue (see Figure 4.12 to visualize how it is executed): after each frame, a copy of the current rendered frame is saved at the start of the list, and the last is deleted if it exceeds the last frame provided as input. Then, for each frame that the network receives as input, the corresponding frame in the list is copied over it. With this method, each frame in the list isn't modified after being copied from the original.

4.3.2 Reward systems

In this section, we've worked using 3 reward systems: tolerable range rewards, reward based in coherence and standard deviation and rewards after impulse.

Tolerable range

This reward system is the same that was described in section 4.2.3. With delayed reactions, this system is only effective if they are uniform: all reactions must occur at the same time. If not, the neural network would consider that is not worth the risk of doing an impulse if that had high punishments (See Figure 4.14 to see an example of a high punishment when doing a correct impulse if the bot has non uniform reaction times).

With this method, the AI learned to do impulses correctly (even some that came from the right side, which is an exception case) but it didn't adapt well to the average movement: it had much more noise than the bot in default moves (See Figure 4.13).

[5] Even though this method is safer, it is important to ensure that the render textures that won't be used again are deleted. Not doing so will cause the memory to overflow, and the environment to stop without apparent errors.

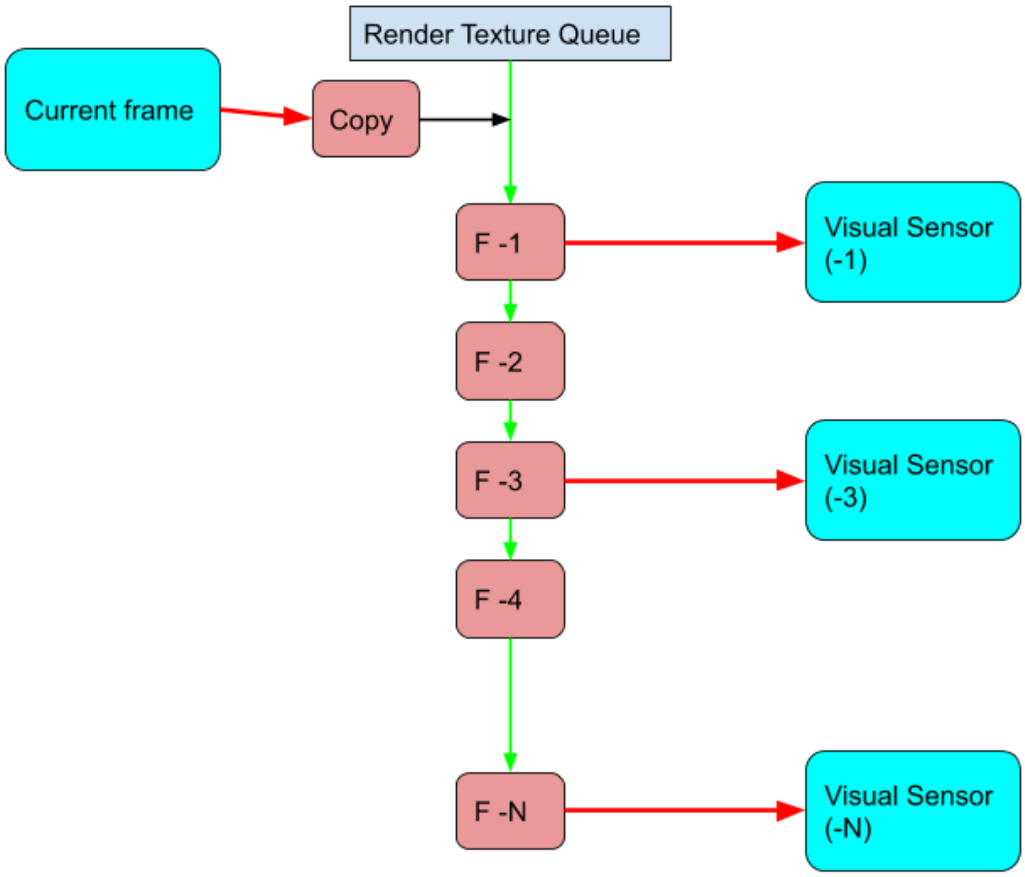


Figure 4.12: Render Queue: see that frame 2 isn't used as input for the neural network. Red arrows mean that the frame (or Render Texture) at the start is copied over the Render Texture at the end.

Rewards after impulse

This reward system uses 2 queues, one for the AI and other for the bot. Whenever one of them does an impulse, instead of giving a score to the moves, they are stored in a queue. Then, when the other one does another impulse it is compared with the first's movement (either the AI or the bot can do the impulse first). If one of them did an impulse but the other didn't, after some time the AI would be penalized (either by missing an impulse or by doing impulses when it shouldn't).

The results obtained using this reward system were not very good: the agents didn't learn to do impulses at all. Since the rewards are given after both impulses are completed, the AI can get confused about when to do an impulse and how [6]. Even though, this

[6]It is important to use large (0.99-0.995) gamma values for the rewards in this method: a big gamma parameter means that the agent looks for future rewards.

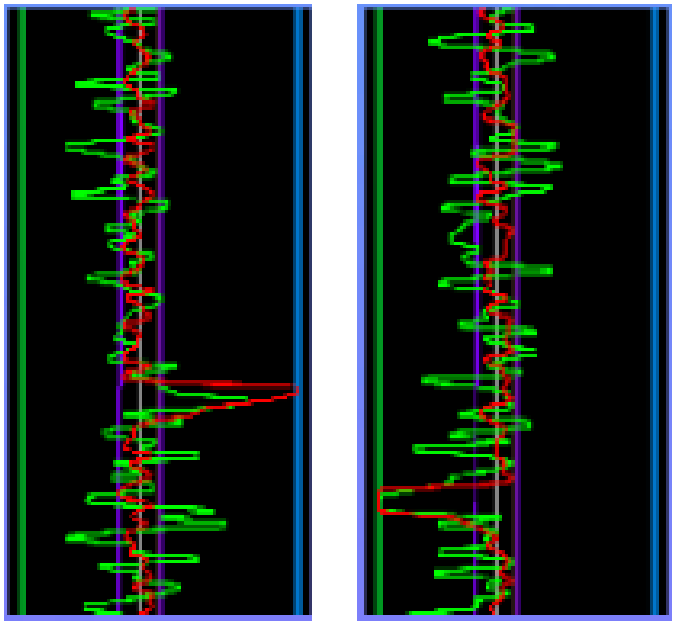


Figure 4.13: AI trained with tolerable range rewards. It adapts well to the impulses but not so much to the normal movement.

method is still the easiest way to model temporal noise, and with better balanced rewards it should perform well (See Figure 4.14).

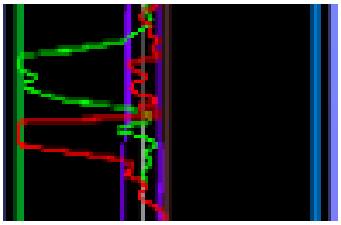


Figure 4.14: Case when the AI would receive a double punishment when doing an impulse: if not using 2 queues, the agent would have a big punish even though the impulse was correct (the bot can react at either moment)

Rewards based in coherence and standard deviation

Like Tolerable Range rewards, this method has been applied to uniform reaction times. In our test cases, we have used reaction times between 1 and 8 frames of difference, and used as input for the AI the last 10 consecutive frames. This method is similar to the one explained in section 4.2.7, but applied to one action instead of two (the expected move instead the maximum-minimum guesses).

This method differentiates between coherent and incoherent moves: when the bot's movement is coherent (it is inside the average±standard deviation range), the AI receives a consistent reward if its move is also inside that range, if not, it receives a punish that gets higher when the relative distance to that range increases. When the bot's movement is incoherent (impulse) there are 3 options:

- If the AI move is closer to the bot's move than to the average, it receives a high reward (higher when closer)
- If the AI move is closer to the average move but between the average and the movement made, it doesn't receive any reward or punish
- If the AI move is in the opposite direction, it receives a punish

Using this method we have obtained the better results until this point (see Figure 4.15), still, it has more problems imitating the exception cases (for example, when a target appears at the opposite side).

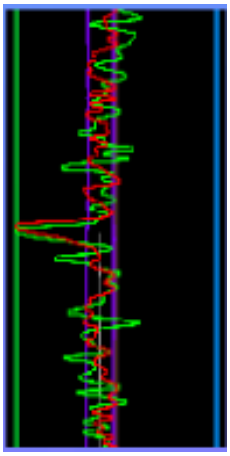


Figure 4.15: Model trained with 0.1 seconds of reaction time using this method

The quality of this results is very dependent on the precision of the average and the standard deviation. In section 4.3.2 we explain how both of them were improved to get better results.

Dynamic average and standard deviation

At this point, to calculate the average and the standard deviation we were considering the last 60 moves. Each frame, the last move was deleted, the new one added to the list; then both the average and the standard deviation were updated. Using this amount of values was correct in some cases, but when some consecutive impulses happened they lost precision (see Figure 4.16), thus spoiling the reward system.

To add precision more values are needed, but too much values would be highly inefficient. To solve both of these problems, we use dynamic averages and dynamic standard deviations. Dynamic parameters are calculated using available previous information to avoid recalculating both values each frame: instead, when adding a new value, we use the last average (and standard deviation) and incorporate the new value to obtain the new average (or standard deviation). Both of them have O(1) computational cost (each frame) instead of O(N).

The dynamic average formula is the following:

$$a_{n+1} = \frac{x_{n+1} + n \cdot a_n}{n+1}$$

Where we obtain the average for the next frame (a_{n+1}) from the last average (a_n), the new move value (x_{n+1}) and the amount of moves that have been used to calculate

a_n (n). Even though the formula can be obtained intuitively, the calculations used are explained at Appendix A.

The calculations needed to obtain the standard deviation σ_{n+1} of the next frame require the last standard deviation σ_n, the new and old averages (a_n, a_{n+1}), the new move value (x_{n+1}), the amount of moves (n), and the average of all the moves squared ($\frac{\sum_{i=1}^{n} x_i^2}{n}$). This last parameter can be easily tracked using the dynamic averages explained above.

The formula (see Appendix B) for the dynamic standard deviation [7] is the following:

$$v_{n+1} = v_n + a_n^2 - a_{n+1}^2 - \frac{\frac{\sum_{i=1}^{n} x_i^2}{n} - x_{n+1}^2}{n+1}$$

The standard deviation is obtained taking the square root of the variance (v): $\sigma_n = \sqrt{v_n}$

Even though these two formulas stabilize both values, the standard deviation fails enclosing the noise of the coherent movement (it should be smaller) when there is a relatively big amount of impulses. To adapt better to the coherent movement, we interpolate the values of the moves outside of the standard deviation range [8]. From several training sessions, the best interpolation parameter for the real move and its closer standard deviation appears to be 0.4 ($0.4 \cdot \text{move} + 0.6 \cdot \text{standard deviation}$). Smaller interpolation parameters still made the range too big, and bigger interpolation parameters caused the standard deviation to increase too slowly [9] (a parameter of 1 would cause the standard deviation to stay at value 0). See Figure 4.16 to view a comparation between smoothed dynamic standard deviation, pure dynamic standard deviation and the non dynamic one.

4.3.3 PPO vs. SAC

As we said in section 4.3.2, it is possible to model a correct behavior using a reward system based in standard deviations and movement coherence. However, it is important to clarify that all of those good results were obtained using PPO.

Even though SAC was more effective modeling non delayed reactive behaviors (see 4.2.9), PPO performed better with delayed reactions. This could be caused because how both algorithms work and because unbalanced rewards:

PPO tends to optimize the agent to have the highest rewards in each situation, but SAC optimizes it to have an overall higher reward. Agents trained with SAC tend to the average move, not doing any impulse. PPO follows the impulses since approaching them

[7] In the formula, the variance is used instead of the standard deviation to simplify the calculations, but we use only the standard deviation value

[8] Smoothing values may not be statistically correct for a standard deviation, but since the objective is to differentiate between Coherent and incoherent moves it is valid for our purpose

[9] The ideal interpolation parameter is approximately 0.4, however other parameters could work better with different amounts of noise. Still, this value works well in most of the cases.

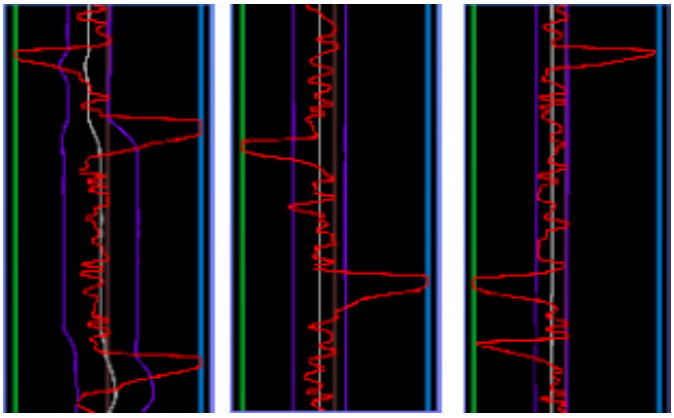


Figure 4.16: Standard variations and average of the same bot using last 60 values (left), dynamic values (mid) and dynamic values smoothed by a 0.6 interpolation (right)

gives a potentially higher reward in that situation. Still, SAC models had better scores just by not exposing themselves to the punishments of failing coherent moves (which were higher than the punishments of failing an impulse). In Figure 4.17 you can see a comparison between both methods using the same reward conditions.

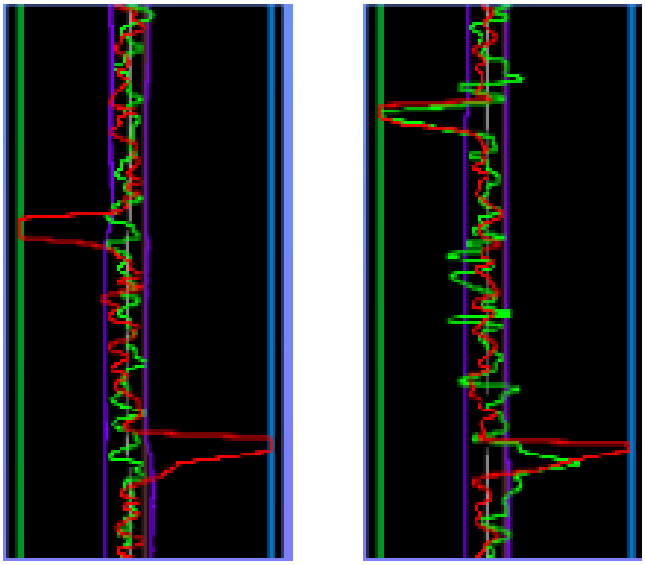


Figure 4.17: Comparison between an agent trained with SAC (left) and other trained using PPO (right). The bots in both cases have a reaction time of 0.1 seconds and 0.15 units of noise

At this point, even though PPO performs better, both methods still fail at performing some impulses, when targets appear at the right side, or when two consecutive targets appear at the same time, and they don't adapt to noised reaction times (at least with

the reward system exposed until this section).

4.3.4 Behavioral cloning

Behavioral Cloning [7] is the simplest algorithm provided by ML Agents in terms of difficulty to adjust: it directly copies the actions given in a demo (which can be recorded in the editor). However, it has its limitations: since it doesn't depend on environment rewards, the programmer cannot modify its behavior with reinforcement learning. Also, depending on the task, agents trained using these methods can have chaotic behaviors.

When trained with simple cases (in section 4.3.6, we treat how the rare cases were suppressed from the training), they perform really well. Also, the agents can model temporal noise effectively. The agent in Figure 4.18 is a simple case at the extreme: it receives the last 9 frames (and the current) as input and the bot can perform an impulse at each one of those frames (from 0.01s to 0.15s). The agent usually does the impulse at the average reaction time of the bot. It is worth noting that when the agent performs more or less correctly it's better to stop training, else it usually loses precision (see Figure 4.19).

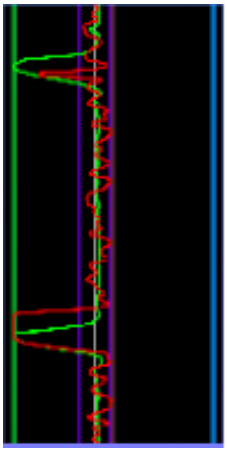


Figure 4.18: Model trained with BC in 2000 steps. The bot's impulses appear deformed because the agent is the one playing the game

In the next 3 subsections, we will explain some problems that appeared when using behavioral cloning, and how they were solved.

4.3.5 Recurrent memory

Recurrent neural network [9] are a feature that allow agents to have memory and remember past observations. They have the advantage of being optimized to "choose" what to remember, at the cost of giving less control to the user. Also, its training is much slower, and they have worse performance when infering.

A combination of past render targets, past moves and recurrent memory can obtain good models (even in some rare cases), but the resulting neural network is so heavy that the frame rate drops from roughly 70 fps to 25 fps. Even though it can perform most of the impulses, it usually has some strange artifacts (extra impulses) in its behavior (See Figure 4.20a).

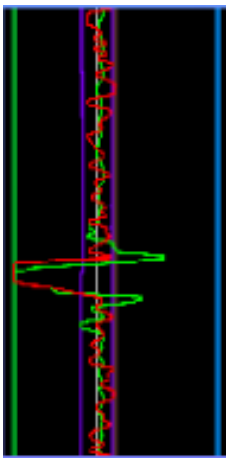
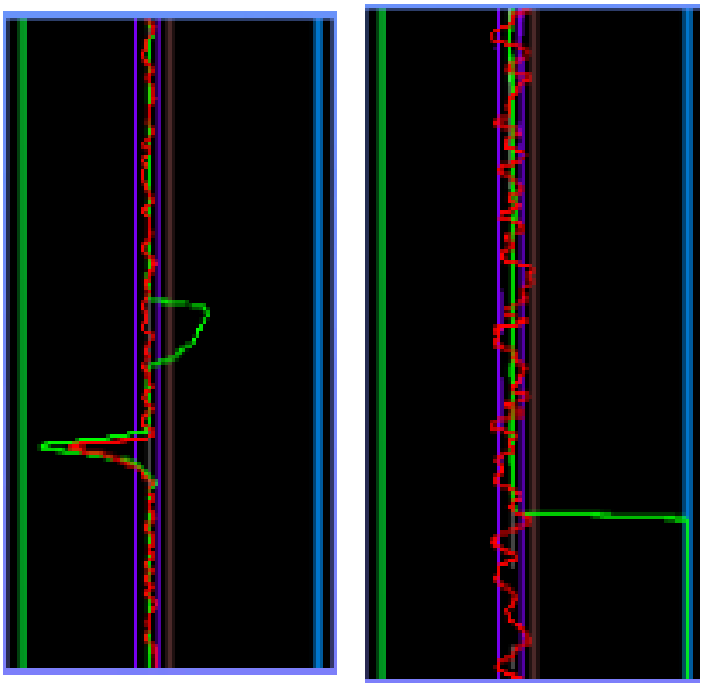


Figure 4.19: Model trained with BC in 7000 steps

One problem [10] that appears when using recurrent networks with simpler inputs is that they are not foolproof: whenever they receive an unexpected input (in most of its inputs), its behavior can become chaotic. In figure 4.20b you can see an example of this problem: the neural network appeared to have trained well, but when it received an empty input for the past moves (at the start, all previous moves are 0), when they were returned by the recurrent memory, the agent became chaotic.



(a) 20 visual inputs and 25 past moves

(b) 1 visual input and 25 past moves

Figure 4.20: Agents trained with recurrent memory and behavioral cloning

In conclusion, recurrent neural networks are not recommended for this problem: behavioral cloning can adapt to the movement without them, and the training sessions and performance become much slower.

4.3.6 Rare situations

As the game was programmed, targets spawn randomly at the stage. Since they can appear at any point, this provokes some situations that happen rarely: for instance, with the bots we have been using most targets appear by entering the screen in the left side (the bot moves continuously in that direction). However, some targets can spawn at its right side in a way that they can be seen, this happens approximately 1 in 12 times (the

[10] According to the ML Agents documentation, recurrent memory is not recommended for continuous action spaces, which we are using

bot has a field of view of almost 60º). In figure 4.21 you can see some examples of some types of situations that appear in game.

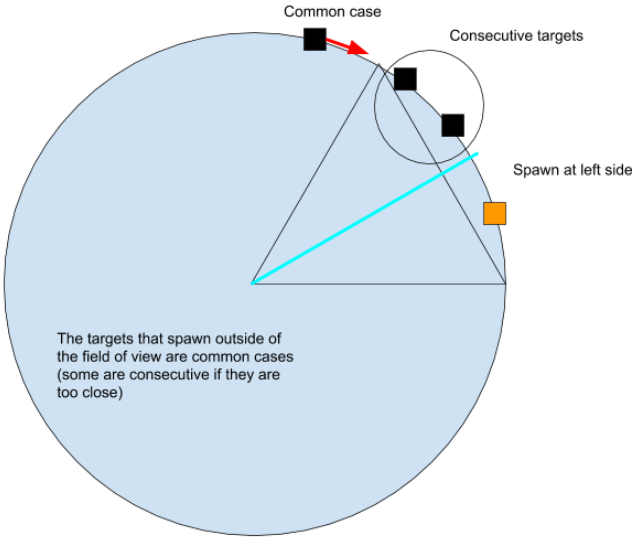


Figure 4.21: Some of the different situations the neural network can encounter in the game

Those rarer cases often suppose a harder task to solve by the neural network, but with temporal delays sometimes they cause the problem to be too complex to be approximated by the neural network (see Section 4.3.7). Also, when they appear less times, they are learned slowly or not at all.

To improve the performance of the network, we can generate some rare cases in purpose: instead of spawning targets randomly, they can be spawned in the right side of the field of view, or two targets can be spawned next to each other, etc. Then, how often each case occurs can be adjusted manually.

This method was intended to make the agents learn faster, however it served to prove that the structure of the neural network was not enough to solve this problem.

4.3.7 Complexity of the task

Sometimes a task is too complex to solve by one neural network structure, with behavioral cloning is easy to detect this problem [1]. These problems need more internal layers (and more training time) to be solved. For instance, the problem developed in this section (delayed reactions) needed at least 3 layers (with any policy) to be solved. When spawning rare cases more

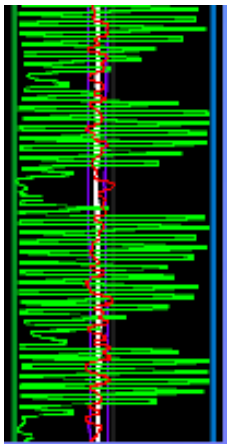


Figure 4.22: Neural network that has diverged

often, some agents that usually didn't learn the exceptions didn't also learn the common cases: the complexity to perform correctly each case was too high for 3 layers.

When using behavioral cloning this phenomenon occurs like this: the neural network starts adapting to the most common cases (but fails the least common), then the more time it is left training, the worse starts doing the common cases until the model diverges (Figure 4.22) and it starts showing strange behaviors. Sometimes it may cycle around all the process, but it would never learn.

4.4 Discrete actions

Discrete actions are those that can be represented by whole or boolean values (for example, a key can be pressed or not). In this case, we will develop agents with the objective of predicting if (and when) a bot is going to click, and the bot will always control the player's movement (the action of clicking would be controlled by either the AI or the NPC).

4.4.1 Bot with shaped movements

To create a more complex environment in which the neural network could learn to imitate the clicks of a bot, it was necessary to implement an NPC that could perform more complicated movements than in the previous sections. The initial goal was to create a bot that could follow a target with constant speed and in a given time.

In the project, it corresponds to the "BotOneMove" script. This bot is capable of choosing targets and directing them in a certain way, which can be edited using Unity animation curves.

To carry out the movement, an interpolation variable is used that goes from 0 to 1 in the duration of the movement. This variable is used to evaluate the animation curve at that point X. The animation curve has a domain from 0 to 1, and for it to work properly [11] f(0)=0 and f(1)=1 must be met. However, intermediate values can be extended beyond these values (this would result in the NPC moving in the opposite direction of its target if it is less than 0, or exceeding it by heading for a target if it is greater than 1).

The movement made in a frame corresponds to the angle to be reached in that frame minus the angle reached in the previous frame. Finally, a point is determined in the initial interpolation variable where the bot will perform the click action (only in the case where it is aiming at a target it can destroy). Some noise can be added to each variable to simulate the inaccuracy of the bot.

In our trainings, we have used a curve that exceeds the upper limits, and in which the bot clicks on the highest end (See figure 4.23). This means that when aiming at small

[11] If the bot has very sharp movement curves or a very short time to perform a movement, it can also have incorrect behaviors. These would be caused because the camera in the game is designed to have speed limitations

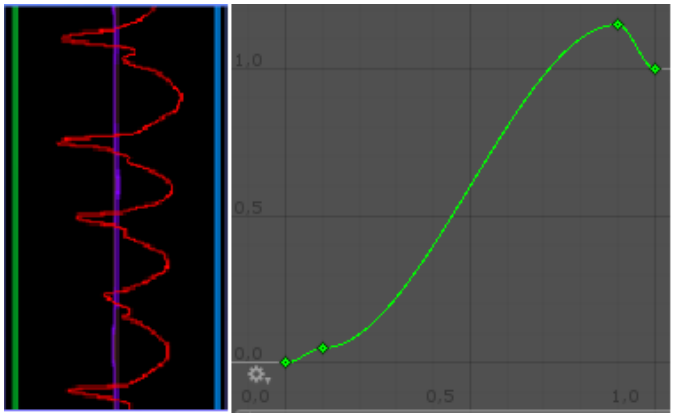


Figure 4.23: Horizontal bot movement (left) obtained when using the curve on the right

targets using long moves, the bot would overshoot the target and fire without hitting the opposite side of the one that started the move.

4.4.2 "Cheat sheet" rewards

As we have seen in the section 4.3.2 with movement queues, not giving instant rewards but a time later can make it difficult to learn the neural network, plus it is harder to balance rewards in these cases where there is randomness in the exact moment an event occurs.

To solve these problems and at the same time accelerate the learning process we have designed the "cheat sheet rewards". These rewards are based on that the NPC itself is the one that determines the accuracy with which the agent has performed the action of clicking, and it does that before (or after) the NPC has performed it. The reward is calculated using a simple tolerable range (See 4.2.3) by comparing the difference between the interpolation value in the frame the bot action is performed and the interpolation value at which the agent would perform the action.

Whenever the AI or the NPC performs a click action, an event is triggered. There are several factors to consider when using these rewards:

- If the AI performs an action outside of the tolerable range (or when the agent is in idle [12] move) is penalized
- An action performed inside the tolerable range is rewarded (the closest it is, the more the reward)
- Only the first action inside the tolerable range is rewarded, any other extra action is punished (not doing so would cause the agent to spam the click when it is close to the tolerable range)

[12] Idle move corresponds to when the agent isn't seeking any target

- If the agent misses a click, the bot sends it a signal that causes it to receive a punish

This method can be used either with SAC or PPO, and with both algorithms correct results have been obtained: both the agent and the bot fail and hit certain targets in a similar way (although not always in the same cases). Nevertheless, there are still a few occasions in which the agent can perform the click action in some situations where it should not, without apparent reason (See Figure 4.24).

4.4.3 Discrete and continuous action spaces

When creating agents, Unity allows to use two action spaces: discrete and continuous [13]. In discrete action spaces, there is a set of determined actions represented each one by an integer; in continuous action spaces, the action is represented with a float (this type of action is ideal for actions like mouse movement, which movement cannot be represented accurately using only integers).

However, ML Agents doesn't allow to combine both action spaces in one agent: all actions must be either discrete or continuous. If we wanted to create a neural network that could move and click by itself, its action space type would have to be continuous. Other option is to use two separate neural networks, one with all discrete actions and the other with the continuous ones. In this section this problem doesn't appear since we only want to imitate the clicking action, but still both methods have been tested.

To model discrete actions using continuous action space type, we consider the moment the action crosses the 0 line (when one action has negative action and the next positive value) as the moment when the agent triggers a click event. That can be consider as a mouse click, when the button is pressed a click event happens; but there cannot be another event until it is released (it returns to negative values). When using this method it is usually ideal to provide the agent the last action values as an observation, since it would tend to stay in positive values when being close to the target (if it fails to hit at first, the action has to return to negative values to perform another click).

To model discrete actions using discrete action space type, we have 2 actions: not clicking (0) and clicking (1). We can use the same method as with continuous action space (only the first consecutive "1" is considered as a click), or take every click action as an actual click. The second method is more reliable in practice, but produces more clicks that can cause the bot to receive very low rewards at first that can make the training session unstable.

In this test case, when using either SAC or PPO, continuous space type has better results (See figure 4.24 to compare how both methods appear in the debug). Agents with discrete action spaces tend to perform much many pointless click (during idle movement) than agents with continuous space type, which usually have a 50-60% success rate. However, since the environment was designed for continuous space actions, that

[13] Do not confuse action space with actions to be imitated (clicks in this case): as we will see later, discrete actions can be modeled using both continuous and discrete action space

can have caused that the agents with discrete action spaces perform worse than their continuous counterparts.

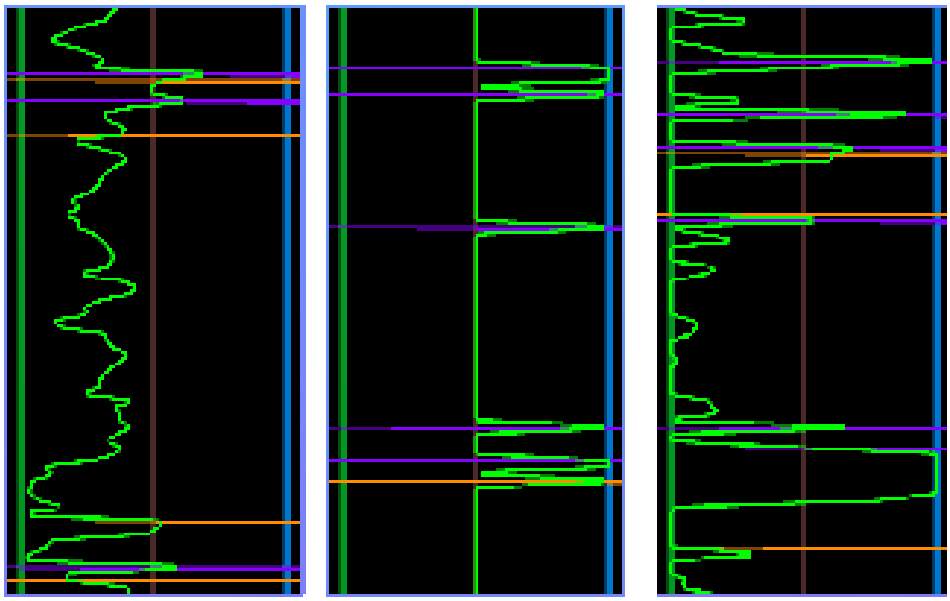


Figure 4.24: Discrete action agents: trained with SAC in continuous action space (left), with SAC but in discrete action space (mid) and with behavioral cloning in continuous action space (right). Orange lines correspond to the NPC clicks, and purple ones are click events triggered by the AI

4.4.4 Models with behavioral cloning

Even though behavioral cloning is very effective with time noise (as seen in section 4.3.4), it has a much harder time when imitating discrete actions. How well or badly it acts depends entirely on the demonstration [14] provided in the training session.

When using continuous action space, the action to be imitated is not just a boolean value, but a continuous number that meets the conditions to perform discrete actions described in section 4.4.3. The easiest way is to assign a random value between -1 and 0 when the NPC is not performing a click, and a random value higher than 0 in the frame when it is clicking (demonstrations with uniform integer values perform worse). After less than 10000 steps, the neural network starts performing some clicks, but its performance is much worse than any model trained using SAC or PPO (see figure 4.24).

Problems that appear when using behavioral cloning in discrete action spaces are described in the following section.

[14] In ML Agents, demonstrations are recorded using the actions provided in the heuristic function of the agent. These demonstrations are intended to be made by a human, but they can also be recorded from any NPC that can be accessed from the agent script

4.4.5 The frame problem

The one frame problem occurs in most demonstrations recorded for behavioral cloning (or GAIL), and it is caused because of how the information flows in each execution cycle in Unity.

As the game was built (See section 3.3), the AI and the NPC are connected only through the Camera controller [15]. Then, the information flows in the following way: the NPC makes an action and it is sent in the first frame, the camera controller receives it (and actually performs it) in the second frame, and then the AI reads that action in frame 3. This delay of 1 frame between the action in the game and the action in the demonstration was negligible in continuous movements (mouse), however, in this case it causes the agent to learn to click when the target has been destroyed.

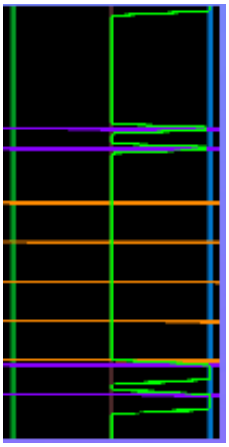


Figure 4.25: NPC getting stuck because of the one frame problem

This problem is not very noticeable when training, but when using the neural network to perform the clicks it makes the NPC get stuck at pointing a target (In this section, the NPC is always the one controlling the movement). Only when the NPC recovers control of the action and destroys the target, the AI does perform. In figure 4.25 you can see how the NPC gets stuck and clicks continuously when the AI is in control at the beginning (the first orange line corresponds to the first NPC click), and just after it recovers control in the click action (before the last orange line) and destroys the target, the AI performs a click action.

This problem can be solved by connecting the AI and the NPC so that it receives the information at the same time as the camera, but it is still outperformed by models trained with SAC and PPO in continuous action spaces.

4.5 Complex movements

In this section we try to create an agent that imitates a more complex movement. To do so, we have used the bot described in section 4.4.1, but with curves that have different shapes (see Figure 4.26). In the following training sessions we won't take into account the click actions to avoid interferences in the rewards. In this case, the bot always clicks at an interpolation value of 1, so it always destroys the target at the same time that a movement ends.

[15] In latter trainings they were connected to create Cheat Sheet rewards, but the the heuristic function remained unchanged

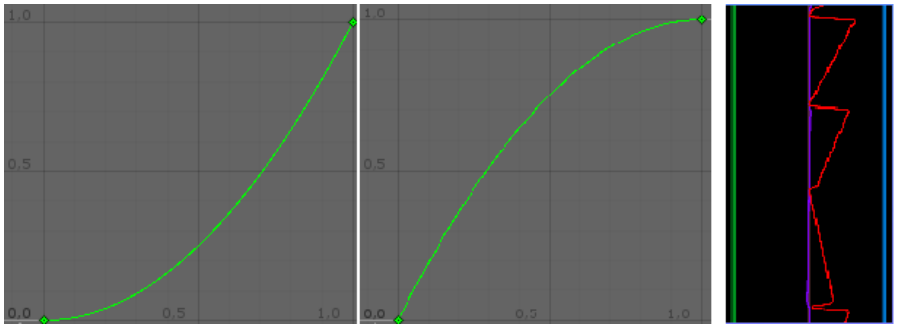


Figure 4.26: Movement curves for seeking target (left) and idle (mid). The right picture shows the bot's movement with one seek target in the middle (purple line is vertical move and red line is horizontal move, both have the same shape).

4.5.1 Debug for 2 axis

The graphics used so far are no longer completely effective in 2-axis movements: even if the bot can do one of the 2 axis well, if it did the other one wrongly, the movement obtained would be very different from that of the NPC to be imitated.

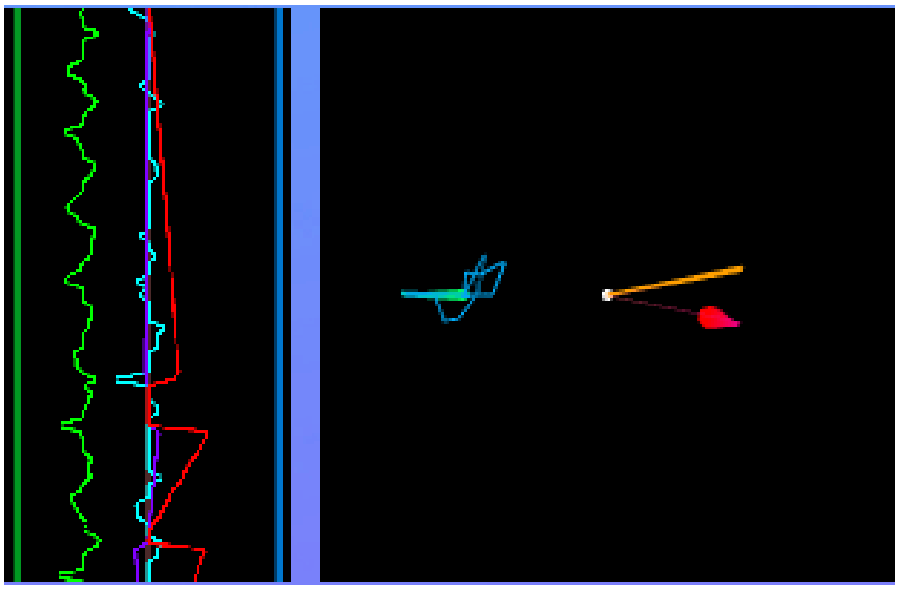


Figure 4.27: Shaped movement of the bot (red-purple lines) and a continuous movement to the left from one agent (green-blue lines) in both of the debug graphics

In this kind of movements, when following targets the direction the bot takes (and the speed in some sense) is more important than the X and Y components treated independently. For this reason, the following graph has been created to debug the bots, showing the direction and speed in each frame. The values shown on it could be

compared to the functioning of a joystick: the movement is in the direction of the point with respect to the origin, and the further from the center it is the faster it will move (See Figure 4.27).

This graph, in addition to the previous one, allows a better understanding of the progress of a bot during training and helps to find problems to correct.

4.5.2 Angle-Magnitude reward system

The first reward system devised for the task developed in this section was the tolerable range, but instead of applying it to the difference between each movement it is applied to the distance in the plane between the points (X, Y) of the NPC and the AI movements. As we can see in figure 4.28, in the line graph the movements seem to adapt to the shape (at least in the horizontal ones, which correspond to those of greater magnitude), but in the plane graph we can see that the movement is much more chaotic than it seemed (especially due to the noise in the vertical movements).

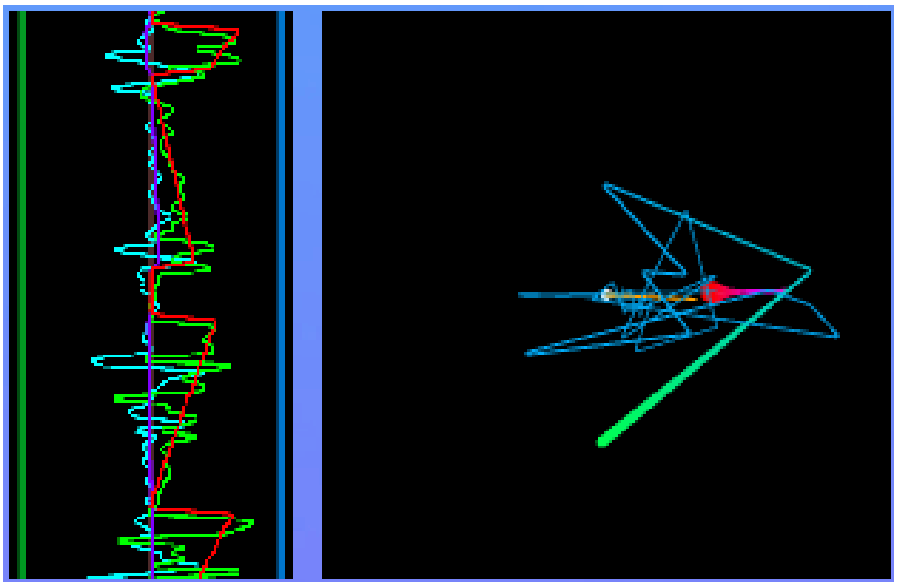


Figure 4.28: Agent trained with tolerable range, shown in both debug graphs

In order to achieve a reward system that is easier to balance and allows to get agents with less noise, a method has been designed to create rewards using angle and magnitude as parameters. In this way, more weight can be given to one of the 2 parameters depending on what is intended to be achieved in the agent (or to adjust the rewards to obtain better results). In Figure 4.28 you can see an example of an agent resulting from a training with angle-magnitude tolerable range: the AI adapts to the shape of the movement and follows the same angle although with still too much noise (in the first frame of a new idle movement it usually does not have much precision, since it is determined randomly).

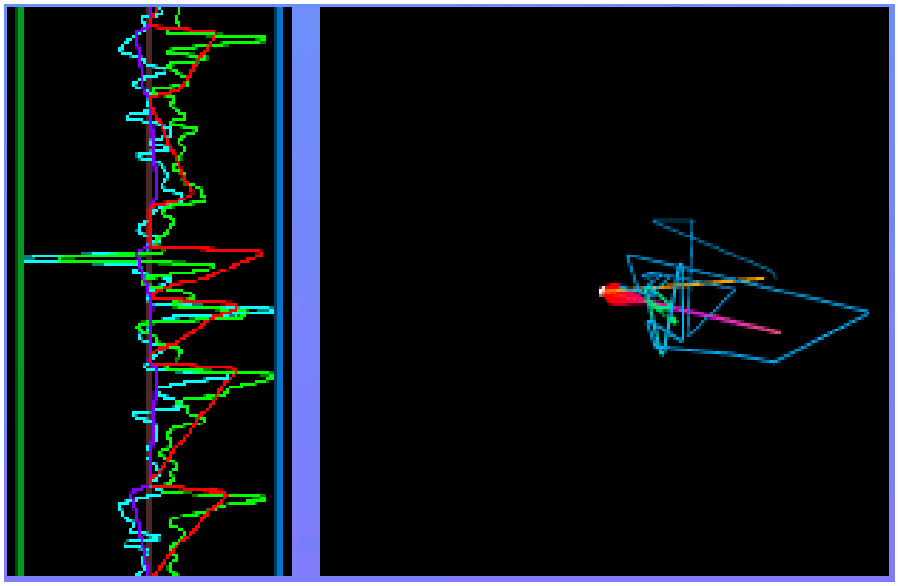


Figure 4.29: Agent trained with angle-magnitude tolerable range

4.5.3 Cheat sheet for movements

The main problem that appears when using tolerable range is that it scores or penalizes all movements equally, which causes it to: have very significant errors at the beginning (when the movement is not predictable), adapt more or less well in the center, and then in the final frames it makes errors again (in this case, because the neural network has more difficulty in matching movements that are very small in magnitude, and the range allows it to go further than it should despite being predictable). In addition, in vertical movements the agent usually has an excess of noise because its magnitude is much lower than that of the horizontal ones (a solution to this problem is proposed in section 4.5.5).

Weighing the penalties correctly using a traditional reward system may involve taking into account parameters such as the coherence of the movement as previously seen, but applied to a linear regression [16] that allows to predict the next movement, and modifying the weight of the rewards and penalties at each point in the movement to give more exigency in points where the movement is predictable and less where it is almost random.

To achieve a similar and easy to modify effect, we have created a "cheat sheet" shaped reward system (Section 4.4.2) in which you can change the value of the punishment and reward parameters at each point of the interpolation of a movement by means of curves. To speed up the process and simplify the task, the movement of chasing a target is not taken into account, so it has been possible to eliminate the camera (the agent only

[16] A linear regresion would make sense for the particular movement we are dealing with in this section, as it can be shaped with straight lines. However, these methods would be too dependent on the concrete form of the movement and very difficult to generalize

receives information about the movement it is doing). In the Figure 4.30 you can see the curves that have been used in the most successful trainings.

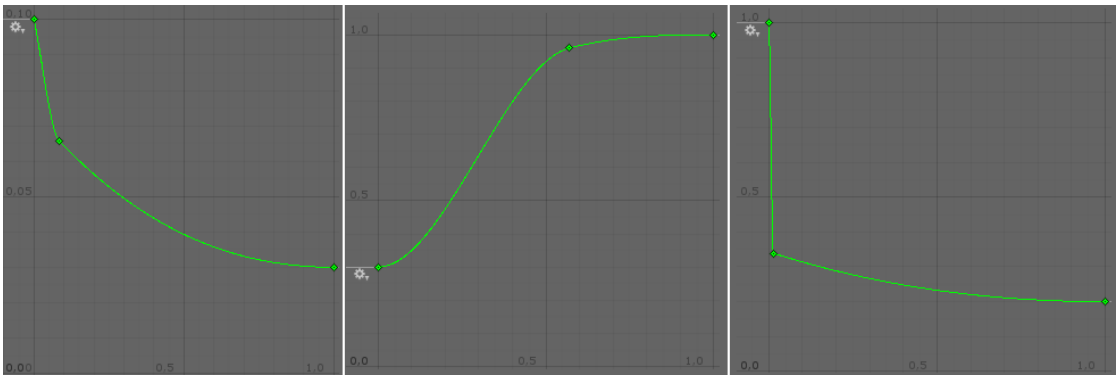


Figure 4.30: Curves for tolerable range (left), punish factor (mid) and reward factor (right)

In figure 4.31 you can see one of the models that best adapts to the NPC's movement: it is noticeable that the bot still has some noise and still doesn't perform very well in the beginning and at the end of the movements, but it tends to stay near the angle of the original move. However, in this case we didn't take into account the seeking cases and it only uses observations of the NPC movement, so this agent would not be able to play the game independently.

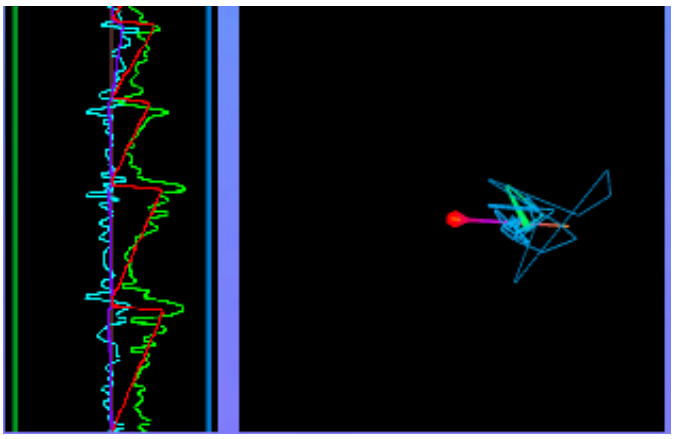


Figure 4.31: Agent trained with shaped rewards

4.5.4 Movement interdependency

In practically all the agents developed in this section, it has been used as observations the previous movements made by the NPC, instead of those made by the agent.

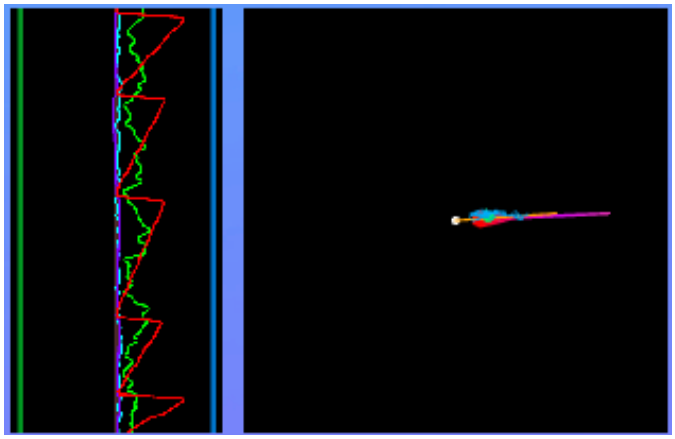


Figure 4.32: Agent trained using its own observations

The problem that presents the interdependence of movements is that in the training the agent learns from the movements generated by the bot, so it is easier to predict the next one. Once the agent plays by himself, it must not only perform movements but also use its previous movements as observations for the next ones. In cases where the agent has too much noise in its moves, this can cause it to end up getting stuck or making meaningless moves.

In some of the cases of this section, using observations of the agent (both in full training and at the end) causes them to move in the correct direction but with constant magnitude (See Figure 4.32). Other agents that have been trained using only observations of the bot can move very chaotically

Ideally, for a workout to be correct enough, it should end up showing correct behavior while receiving as observations its own movements rather than those of the bot. In a perfect training, the agent would end up controlling the movement of the game (and performing it in a similar way to the NPC). However, none of these 2 conditions could be achieved in this section.

4.5.5 Trying to improve performance

In this subsection, several experiments that were attempted to correct certain errors that appeared when training agents from this section are presented.

Mouse sensitivity

The sensitivity of the mouse was intended to give the bot more precision in vertical (Y axis) movements (which are usually much smaller in magnitude than horizontal ones). This was achieved by dividing the action of the neural network by a value. Although it improved the accuracy in most cases, the network was practically prevented from making minimally fast vertical movements (for example, when a target appears above). Figure 4.33 shows an example of agent that was trained using a sensitivity of 10: in the

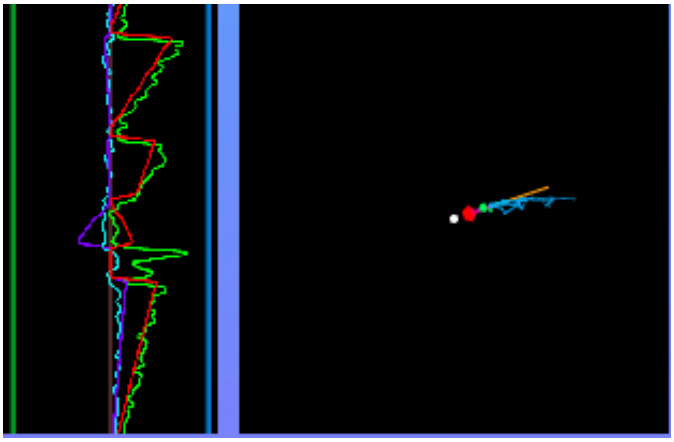


Figure 4.33: Agent with Y sensitivity of 10

line graph it can be seen that the agent adapts very well to almost all movements with precision, except when the vertical movement is slightly larger; in the plane graph it is also visible that the agent doesn't reach the higher vertical moves (the line is slightly curved).

Momentum

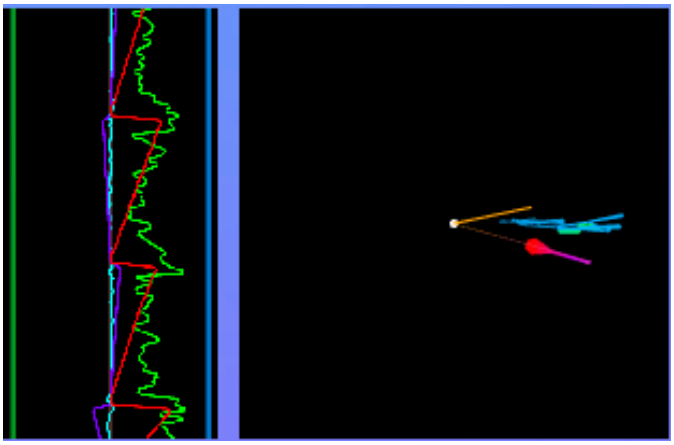


Figure 4.34: Agent that tends to anticipate next movements

When using the reward curves shown in Figure 4.30, the agents usually "anticipated" the next move instead of ending the current move well (see Figure 4.34). This was because the first frame of a move offered a much higher and easier reward than the last. *Momentum* is a variable that oscillates between -0.75 and 0.75: starting at 0, when the agent receives a positive reward, the momentum is increased by 0.125, if it receives a negative reward, it is decreased. For example, a momentum of 0.75 causes

positive rewards to be multiplied by 1.75, and negative rewards by 0.25 (and the opposite happens with negative momentum). Then, if the agent fails some consecutive actions in the last frames of a movement, the punishments are increased stepwise and the reward obtained in the first frame of the next movement is reduced considerably (depending on how many consecutive actions the agent has failed before).

The effectiveness of this parameter is difficult to be appreciated, however, the best results obtained in this section were trained using momentum.

Angle-Magnitude actions

It also was tried to create a bot that performed actions based on angle and magnitude (direction and velocity), instead of X movement and Y movement. It was tested using previous X and Y movements as observations, and previous angles and magnitudes.

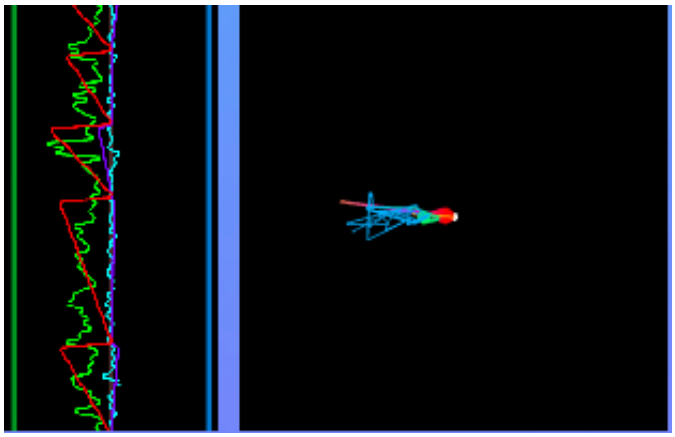


Figure 4.35: Agent with angle-magnitude actions (the bot was reversed to avoid crossing the 0-line)

In theory, imitating the bot with the information presented in this way should have been trivial for the agent (the angle remains constant throughout the movement, and the magnitude decreases linearly to 0), but the results obtained were not very different from those using the X and Y movements as action and observation, they were even worse (See Figure 4.35). Also, it had difficulties to follow the bot's movement when the angle changed crossed the 0 degree line (an angle of 0 degrees corresponded to an action of -1, and an angle of 360 to +1).

CHAPTER 5

Conclusions and Future Work

Contents

5.1 Conclusions

In this document, different methods have been presented that allow the creation of agents through reinforcement learning and behavioral cloning that imitate NPCs with simple behaviors. However, these methods could not be extended to more complex cases successfully (non-functional requirement).

Each of the behaviors that have been studied have had very different results depending on the algorithm provided:

- Instantaneous reactive movements can be replicated using any of the algorithms provided [1]. In the case of the reinforcement learning algorithms is also necessary a reward system that is usually more complex than the movement itself. Within the 2 algorithms of this type that are available in ML Agents, SAC usually obtains better results than PPO.
- In the movements with delayed reactions, the reinforcement learning algorithms have greater problems to understand the problem, except in the cases in which

[1] Behavioral cloning was not used for this task since it would be trivial for it to solve, however it was applied to delayed reactions successfully (instantaneous reactions can be considered as a sub-case of delayed reactions, with a reaction time of 0)

the variability of the reaction times is null or practically null. This is because receiving rewards later than expected can confuse the agent. When the variability is large, behavioral cloning is usually more successful if the dataset provided is good enough.

- For discrete actions, we took advantage of the possibility of obtaining information directly from the bot in order to model its behavior more effectively.In this type of actions, reinforcement learning algorithms clearly outperform the imitation learning ones (BC). Also, in our trainings related to discrete actions, continuous action spaces worked better than discrete action spaces.

- In our attempt to create more complex movements we trained agents to imitate the shape of a 2-axis movement. However, the agents had difficulties to imitate the bot with all the available algorithms, and it was not possible to get a good result in an agent that received its own previous movements as observations.

With these results, on the basis of the functional requirements provided in section 3.1.1:

- Some of the neural networks were able to play the game independently under specific conditions, and hardly ever under any situation.

- The network can receive as input and interpret what the camera is seeing

- The network is able to receive multiple past frames and actions. However, the more inputs it receives the more easily it can become unstable.

- The network can output one or multiple actions that the player can make

- The network can adapt its actions to reaction times of the NPC (but only if it receives enough past frames). Behavioral cloning allows to model reaction times that have more variability.

- Under some circumstances the network and the NPC can hardly be differentiated, however, when being exposed to rare or untrained situations the network does not act correctly.

The accomplishment of the non-functional requirements depends on the algorithms used and the complexity of the task: behavioral cloning is very sample efficient, and most of the networks that use reinforcement learning algorithms can usually be trained effectively in reasonable time (less than 1 hour). However, this framework is very difficult to scale to imitate a little more complex than those studied in this document.

5.2 Future work

ML Agents is designed to develop simpler behaviors than imitating NPCs (and, of course, simpler than imitating humans).

On the other hand, ML Agents trainings are executed in the editor in real time, which not only decreases the quality of the obtained results but also exposes to execution errors or memory overflows (especially in cases where agents receive multiple previous input frames).

Most of the time, reinforcement learning algorithms are ineffective for complex imitations, since the complexity of the reward systems increases much faster than the complexity of the behavior. The only exception is when there is access to both past and future information. This can be caused by imitating a bot whose actions can be determined sufficiently in advance or by having the information as a separate dataset. In the second case, there would be no reason to perform real-time training.

As future work, it may be possible to obtain better results with different network structures trained offline, such as classification neural networks [10], generative adversarial networks [5] or recurrent networks [9]. However, they should be reincorporated to a game engine to be executed, and they would need to perform in real time.

5.3 Final considerations

Because we have used ML Agents to develop this work, there are sections of the planning that did not need to be done, or that could not be done:

- Since the training sessions were executed in the editor, there was no need to create and save datasets [2].
- In ML Agents it's not possible to create custom neural networks without modifying its source code, which can be very risky. Configuration files allow to determine the number of layers and the number of hidden units in each layer. It can also apply convolutions to image inputs, but only individually.
- Since we were unable to obtain reliable agents after increasing the complexity of the behavior, the framework was not standardized.

To end with, the project can be accessed and downloaded at the following link: `https://github.com/alexcercos/ML-Agents`

The instructions to open and execute the project can be found at Appendix C.

[2] ML Agents allows to create datasets to be used in imitation learning, however its content cannot be seen and they cannot be extended

Bibliography

[1] Jayesh Bapu Ahire. The xor problem in neural networks. https://medium.com/jayeshbahire/the-xor-problem-in-neural-networks-50006411840b/. Accessed: 2020-06-14.

[2] D. Livingstone. *Turing's test and believable AI in games.* Computers in Entertainment (CIE), 4(1), 6., 2006.

[3] Hoshino J. Nakano A., Tanaka A. *Imitating the Behavior of Human Players in Action Games.* ICEC 2006. Lecture Notes in Computer Science, vol 4161. Springer, Berlin, Heidelberg, 2006.

[4] OpenAI. Proximal policy optimization. https://openai.com/blog/openai-baselines-ppo. Accessed: 2020-04-25.

[5] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks, 2015.

[6] Berkeley Artificial Intelligence Research. Soft actor critic—deep reinforcement learning with real-world robots. https://bair.berkeley.edu/blog/2018/12/14/sac/. Accessed: 2020-05-07.

[7] Caude Sammut. *Behavioral Cloning*, pages 93–97. Springer US, Boston, MA, 2010.

[8] Unity. Ml agents documentation. https://github.com/Unity-Technologies/ml-agents. Accessed: 2019-11-22.

[9] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization, 2014.

[10] G. P. Zhang. Neural networks for classification: a survey. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 30(4):451–462, 2000.

APPENDIX A

Dynamic average

Starting from the average formula: $\frac{1}{n}\sum_{i=1}^{n} x_i = a_n$

We solve the equation for a_{n+1}:

$$\sum_{i=1}^{n} x_i = n \cdot a_n$$

$$\sum_{i=1}^{n} x_i - n \cdot a_n = \sum_{i=1}^{n+1} x_i - (n+1) \cdot a_{n+1} = 0$$

$$\sum_{i=1}^{n} x_i - n \cdot a_n = \sum_{i=1}^{n} x_i + x_{n+1} - (n+1) \cdot a_{n+1}$$

$$(n+1) \cdot a_{n+1} = x_{n+1} + n \cdot a_n$$

$$a_{n+1} = \frac{x_{n+1} + n \cdot a_n}{n+1}$$

Then, it has been proven that the average of N+1 elements can be obtained with cost O(1) knowing the average of N elements and the new element.

APPENDIX B

Dynamic standard deviation

Starting from the variance[1] formula: $\frac{1}{n}\sum_{i=1}^{n}(x_i - a_n)^2 = v_n$

$$\frac{1}{n}\sum_{i=1}^{n}(x_i - a_n)^2 - v_n = \frac{1}{n+1}\sum_{i=1}^{n+1}(x_i - a_{n+1})^2 - v_{n+1}$$

We want to solve the equation for v_{n+1}:

$$(n+1)\sum_{i=1}^{n}(x_i - a_n)^2 - n(n+1)v_n = n\sum_{i=1}^{n+1}(x_i - a_{n+1})^2 - n(n+1)v_{n+1}$$

$$(n+1)\sum_{i=1}^{n}(x_i^2 - 2x_i \cdot a_n + a_n^2) - n(n+1)v_n = n\sum_{i=1}^{n+1}(x_i^2 - 2x_i \cdot a_{n+1} + a_{n+1}^2) - n(n+1)v_{n+1}$$

$$(n+1)(\sum_{i=1}^{n}x_i^2 - 2a_n\sum_{i=1}^{n}x_i + na_n^2) - n(n+1)v_n = n(\sum_{i=1}^{n+1}x_i^2 - 2a_{n+1}\sum_{i=1}^{n+1}x_i + (n+1)a_{n+1}^2) - n(n+1)v_{n+1}$$

$$(n+1)(\sum_{i=1}^{n}x_i^2 - 2a_n\sum_{i=1}^{n}x_i + na_n^2) = n(n+1)(v_n - v_{n+1}) + n(\sum_{i=1}^{n+1}x_i^2 - 2a_{n+1}\sum_{i=1}^{n+1}x_i + (n+1)a_{n+1}^2)$$

$$(n+1)\sum_{i=1}^{n}x_i^2 - 2(n+1)a_n\sum_{i=1}^{n}x_i + n(n+1)a_n^2 = n(n+1)(v_n - v_{n+1}) + n\sum_{i=1}^{n+1}x_i^2 - 2na_{n+1}\sum_{i=1}^{n+1}x_i + n(n+1)a_{n+1}^2$$

$$(n+1)\sum_{i=1}^{n}x_i^2 - 2(n+1)a_n\sum_{i=1}^{n}x_i + n(n+1)a_n^2 = n(n+1)(v_n - v_{n+1}) + n\sum_{i=1}^{n}x_i^2 + nx_{n+1}^2 - 2na_{n+1}\sum_{i=1}^{n+1}x_i + n(n+1)a_{n+1}^2$$

$$(n+1)\sum_{i=1}^{n}x_i^2 - 2(n+1)a_n\sum_{i=1}^{n}x_i = n(n+1)(v_n - v_{n+1} + a_{n+1}^2 - a_n^2) + n\sum_{i=1}^{n}x_i^2 + nx_{n+1}^2 - 2na_{n+1}\sum_{i=1}^{n+1}x_i$$

$$n\sum_{i=1}^{n}x_i^2 + \sum_{i=1}^{n}x_i^2 - 2(n+1)a_n\sum_{i=1}^{n}x_i = n(n+1)(v_n - v_{n+1} + a_{n+1}^2 - a_n^2) + n\sum_{i=1}^{n}x_i^2 + nx_{n+1}^2 - 2na_{n+1}\sum_{i=1}^{n+1}x_i$$

[1] We use variance instead of standard deviation to simplify the equations: the standard deviation can be obtained taking the square root of the variance

$$\sum_{i=1}^{n} x_i^2 - 2(n+1)a_n \sum_{i=1}^{n} x_i = n(n+1)(v_n - v_{n+1} + a_{n+1}^2 - a_n^2) + nx_{n+1}^2 - 2na_{n+1} \sum_{i=1}^{n+1} x_i$$

$$\sum_{i=1}^{n} x_i^2 - 2(n+1)a_n \cdot na_n = n(n+1)(v_n - v_{n+1} + a_{n+1}^2 - a_n^2) + nx_{n+1}^2 - 2na_{n+1} \cdot (n+1)a_{n+1}$$

$$\sum_{i=1}^{n} x_i^2 - 2n(n+1)a_n^2 = n(n+1)(v_n - v_{n+1} + a_{n+1}^2 - a_n^2) + nx_{n+1}^2 - 2n(n+1)a_{n+1}^2$$

$$\sum_{i=1}^{n} x_i^2 = nx_{n+1}^2 + n(n+1)(v_n - v_{n+1}) - n(n+1)(a_n^2 - a_{n+1}^2) + 2n(n+1)(a_n^2 - a_{n+1}^2)$$

$$\sum_{i=1}^{n} x_i^2 = nx_{n+1}^2 + n(n+1)(v_n - v_{n+1}) + n(n+1)(a_n^2 - a_{n+1}^2)$$

$$\frac{\sum_{i=1}^{n} x_i^2}{n} = x_{n+1}^2 + (n+1)(v_n - v_{n+1} + a_n^2 - a_{n+1}^2)$$

$$\frac{\sum_{i=1}^{n} x_i^2}{n} - x_{n+1}^2 = (n+1)(v_n - v_{n+1} + a_n^2 - a_{n+1}^2)$$

$$\frac{\frac{\sum_{i=1}^{n} x_i^2}{n} - x_{n+1}^2}{n+1} = v_n - v_{n+1} + a_n^2 - a_{n+1}^2$$

$$v_{n+1} = v_n + a_n^2 - a_{n+1}^2 - \frac{\frac{\sum_{i=1}^{n} x_i^2}{n} - x_{n+1}^2}{n+1}$$

Finally, the standard deviation can be obtained with $\sigma_n = \sqrt{v_n}$

It has been proven that the standard deviation of N+1 elements can be obtained knowing the standard deviation (or variance) of the N previous elements, the new element, the averages of the N and N+1 elements and the average of the N previous elements squared, with cost O(1).

Appendix C

Instructions for using the project

C.1 Opening the project

The only requirement to use the project is to have Unity 3D 2019.2.12.f1 installed (the project could work in higher versions, but it is not guaranteed).

There's a direct link to the main scene at the root of the project, the original can be found at ./UnitySDK/Assets/ShootingEnv.

C.2 Setting the scene

For the agents to work correctly, its parameters need to be adjusted. In the folder ./UnitySDK/Assets/TFG there are 3 files with information about the project:

- Agents_Parameters.pdf contains all the parameters of the agents that were saved
- Trainings_Complete.pdf contains all the notes and information about all trainings that were made for this project
- A copy of this document can also be found (Memory.pdf)

C.2.1 Spawners

The Spawn script by default can spawn planes until there are 6 in the scene. The "Spawn Cases" script can generate rare or common events at will.

Time between spawn determines the frequency at which targets are created (1.5s by default).

Height Diff **MUST** be set to 0 for IMPULSE or RANGE agents, and can be set to higher values for the other types of bots. It determines the maximum height at which planes can spawn.

If using Spawn Cases, make sure that Camera Y Axis is the active one (if there are multiple Camera Axis objects in the scene).

C.2.2 Debug

"GraphCamera" object contains 3 debug options: **"show graphic"** set to true displays the a line graph, **"show axis"** displays the plane graph and **"show grid"** changes the background to a grid that allows to see better the movement (it doesn't affect the bot's observations).

C.2.3 Bots

3 bots can be used to compare its behavior to the agents' ones: BotDelayUniform, BotOneMove and BotPrecision.

Each one has custom parameters that can be changed for each agent. The parameters that are not contained in the file Agents_Parameters.pdf were used for training and don't influence the behavior in the editor (but it is not recommended to change them).

C.2.4 Agent parameters

The agent parameters can be found at the child object of a Camera Axis (Main Camera).

AgentShoot and BehaviorParameters contain the most important parameters to execute the agents. These ones must be set for the agent to work properly:

Y Sensitivity: reduces the Y axis action range (default to 1)

Agent Type: generates different behaviors

Receive observations: type and quantity of observations

Observation Bot-Agent: allows to compare how an agent behaves when the observations received are from the bot or from the agent itself. 0 corresponds to observations from the Bot, and 1 from the agent. It can be changed in execution time, but be aware that some agents don't behave well using its own observations.

Demo Heuristic: used to record observations. For playing manually the game, it must be false (and the model in behavior parameters must be none). If using an agent, it doesn't have any effect.

Show Std: displays the average and standard deviation lines (for IMPULSE or RANGE agents).

Click Discrete: used only when executing CLICKONLY agents that use discrete action space.

Random Restart: puts the agent in a random situation after 1000 frames (Not recommended to use, especially with agents that only move in the X axis).

Vector Observation Space Size: size of the observation vector

Vector Action Space Size (Continuous): amount of actions

Vector Action Branches Size (Discrete): only used for discrete CLICKONLY agents: branches size is always 1, and branch 0 size must be 2.

Model: contains the agent.

C.2.5 Frames

There are 2 types of sensor, Camera (only used in some of the first bots) and Render Texture.

The sensor used for the current frame is named after its default value (RenderTextureSensor or CameraSensor).

If using previous frames, they must be named "FrameXXX", where XXX corresponds to the number (for example, the immediate last frame would be Frame001). There need to be all the Render Texture Sensors with their correct names for the agent to work.

The root object of camera axis contains a Render Texture Handler: Render queue must contain at least each frame used by the agent with the correct position (the position number is N-1, for instance the immediate last frame, which is Frame001, is at the position 0 of the render queue).

The ShootScene contains 3 different Camera Axis already set to be used. Make sure that only one of the is active, and that the Spawn Cases script contains its reference (and not one of the inactive ones).