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

June 15, 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

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. However, we will also discuss how can we train from external datasets, and which data they should have.

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 [1] [2], 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

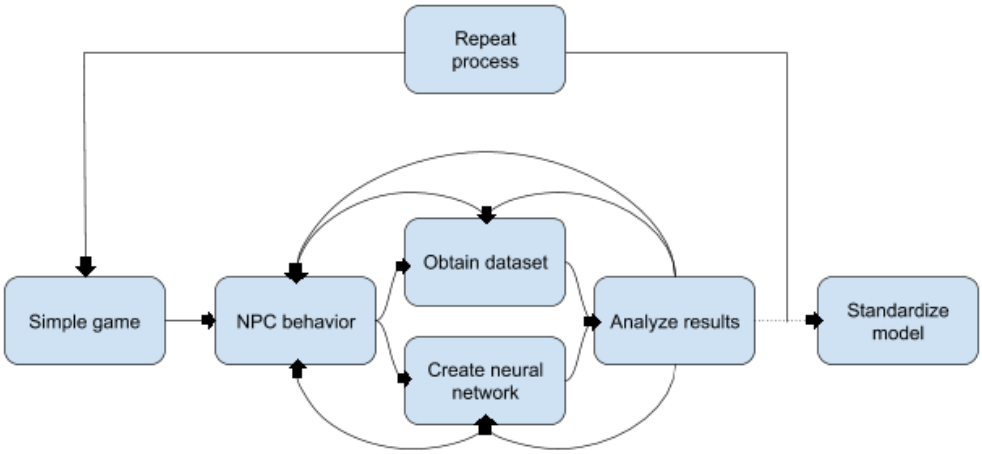


Figure 2.1: Planning graph

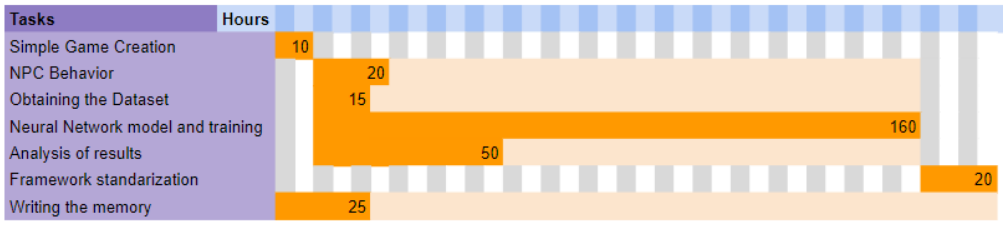


Figure 2.2: Initial planning

"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.

In addition, at this point it will be necessary to make decisions such as the size of the window (and therefore, of the frames), which will be a multiple of 16: 9, and the frame rate (FPS). The ideal will be the minimum possible without losing excessive information.

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. We will also take into account the implications of any specific case.

The proposed neural network, in simple terms, is a classification network: 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

[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.

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.

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 [4].

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. Both the neural network and the agent intended to be imitated will play games of very short duration (2-6 seconds) with the same conditions. It is important that they are short since small random differences due to inaccuracy of the original behavior could accumulate over time and create very different and incomparable situations.

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 behaviour 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.

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, the differences between the neural networks in each case (number of layers and neurons per layer) or specific cases in which the model, if they were given.

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

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 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 the model inference, the following requirements must be met:

- Unity 3D 2019.2.12.f1
- 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 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 made (continuous actions)
- The network will output the probability of performing an action (discrete actions)
- The network will adapt its actions to reaction times of who is imitating

- The network and the real player 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,

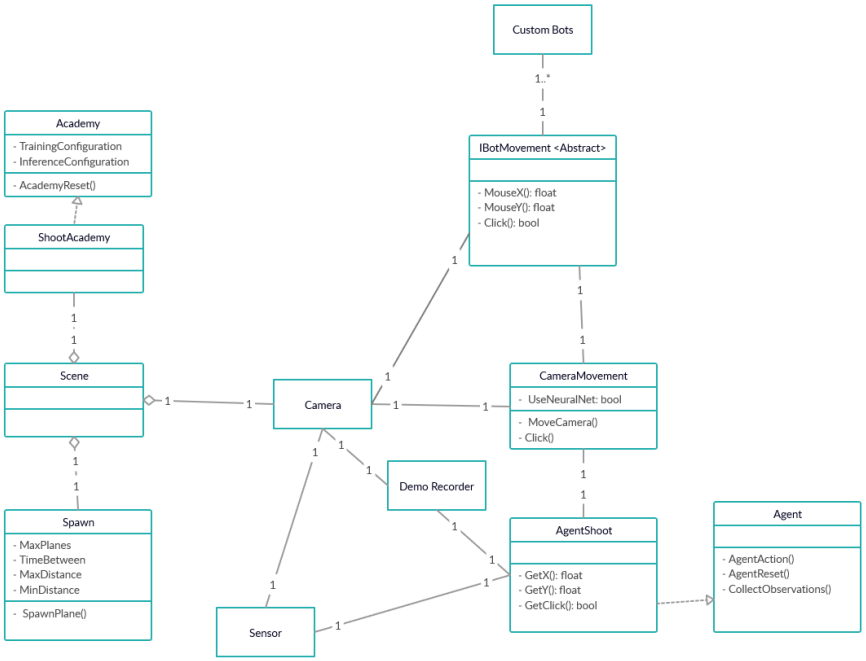


Figure 3.1: Class diagram of the training environment

a visual sensor and a Demo Recorder (when activated, it generates a dataset for Imitation learning).

The classes Academy, Agent, DemoRecorder and Sensor are provided by the ML Agents SDK [4].

3.3 System Architecture

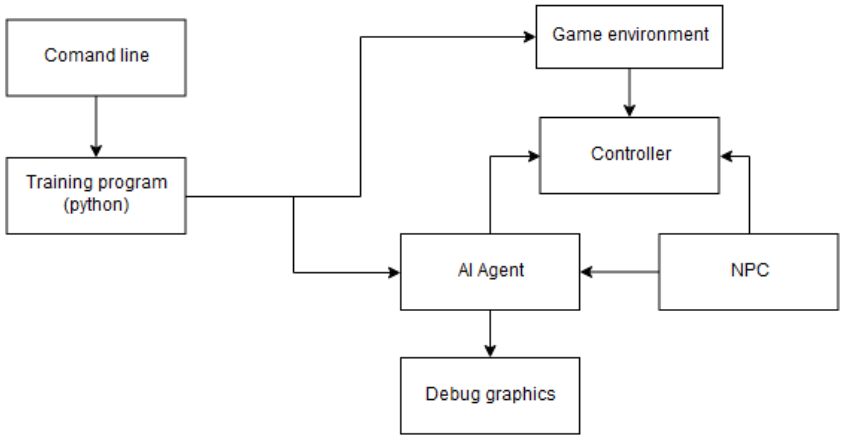


Figure 3.2: Diagram of the system architecture

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.

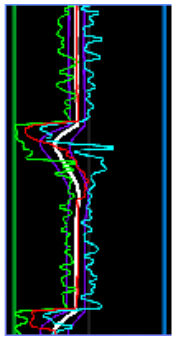
3.4 Interface Design

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.

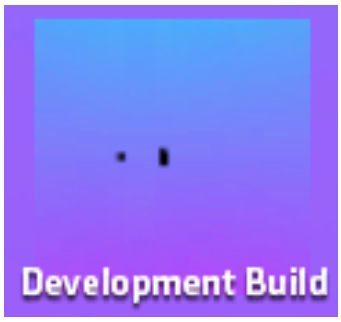
The graph displays 3 main lines (see Figure 3.4a): the red one is the move made by the bot, the blue and green ones are the highest and lowest guess of the bot (since the bot's movements are not perfect, these two lines can vary from being almost touching to being really wide). Other 3 lines in the back display 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 4.2.4 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 they 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 in the Y axis, and it will be considered that is always clicking (so the targets would be destroyed whenever the sight touches them).

In the following subsections, some training related issues will be taken into account. At first, we will train our models using Proximal Policy Optimization (PPO) [3].

4.2.1 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 Y 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.

4.2.2 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).

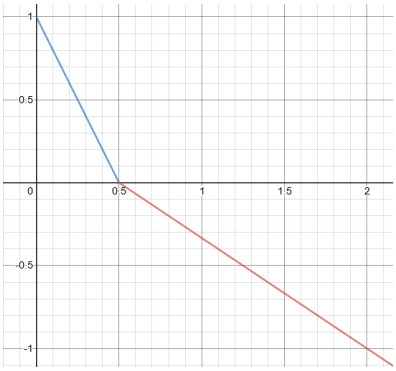


Figure 4.1: Rewards based in tolerable range.

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 caused the training to become unstable because it only got negative rewards, and higher tolerable range

Models trained using this 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 4.3 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 4.2.3.

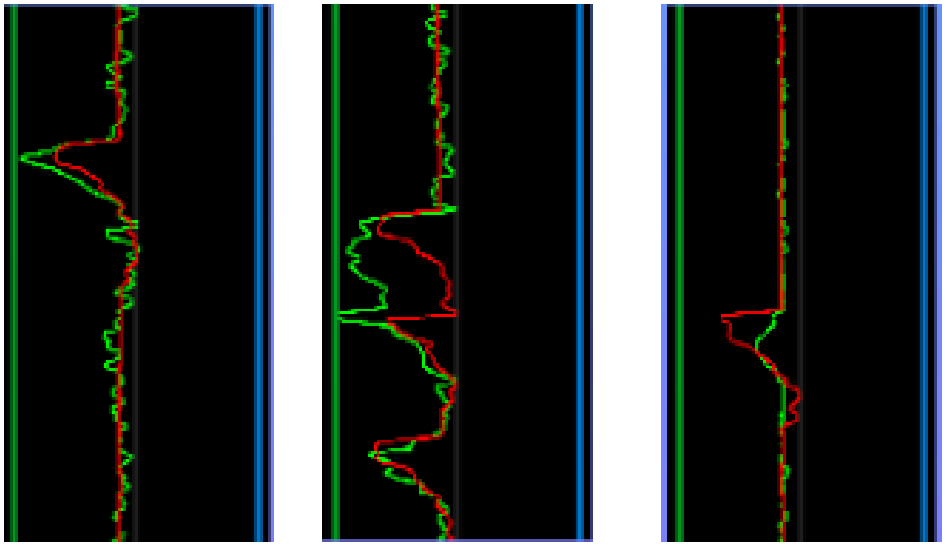


Figure 4.2: Bot movement (red) and neural network movement (green)

[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.

4.2.3 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.

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.

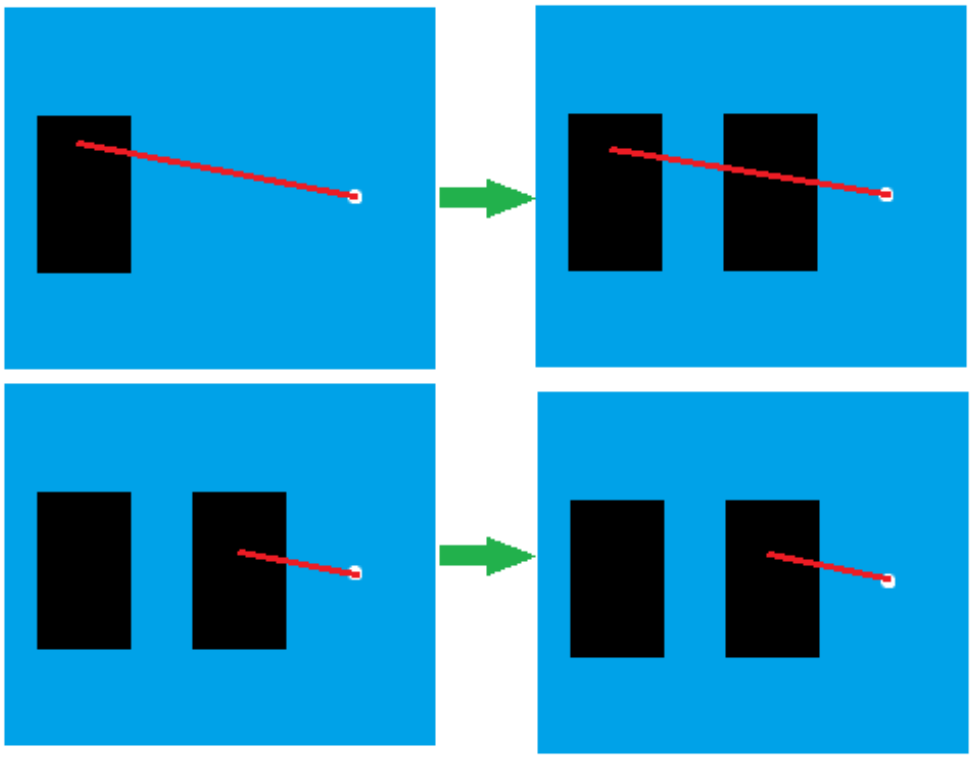


Figure 4.3: 2 situations that lead to different actions with the same frame

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

4.2.4 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.

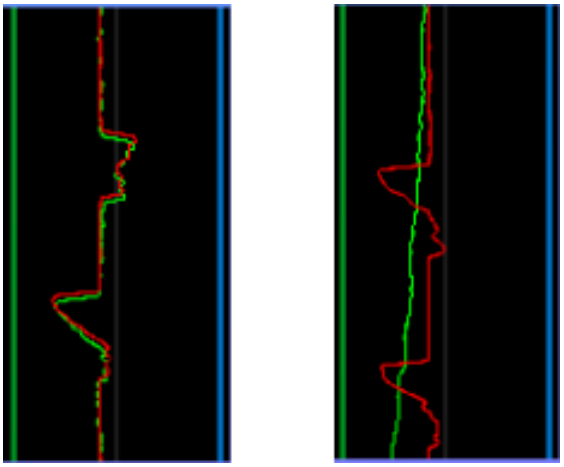
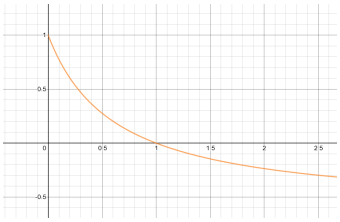


Figure 4.4: A badly trained model while training (up) and playing (down)

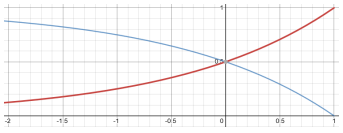
A better approximation would be interpolating the real move with the neural network's one: at the start, the movement added as observation in the next frames would be the NPC move. When the AI starts learning to adapt to the context (previous moves), the movement added would be an interpolation between 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.

4.2.5 Rewards based in standard deviation

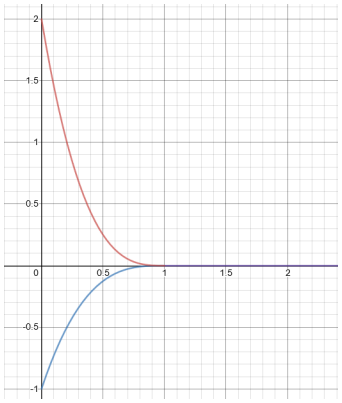
Since trained models using the methods explained in the previous sections tend to return the most common value, movements with more noise or imprecissions would not be



(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

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.

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.

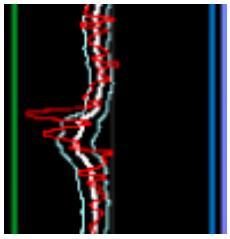


Figure 4.5: Weighted average and standard deviations of an irregular movement

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 precision 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.

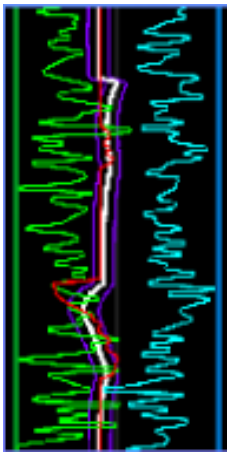


Figure 4.7: Trained model using SD rewards

4.2.6 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 4.2.5, 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 (0..N, coherence is negative) and the precision factor shown in section 4.2.5.

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). However, a learning rate higher than 1e-2 can easily lead to unstable models that don't learn at all.

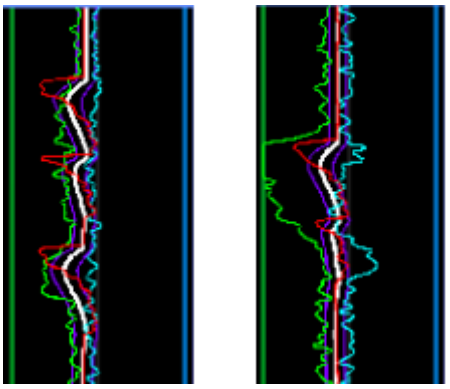


Figure 4.8: Coherence-based models with and different learning rate: left=2e-3, right=8e-3

4.2.7 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.)		

4.2.8 Training with Soft-Actor Critic (SAC)

Soft-Actor Critic is the second reinforcement learning policy provided in ML-Agents [4]. 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).

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 4.2.6): SAC converges to a higher reward than PPO with much less steps.

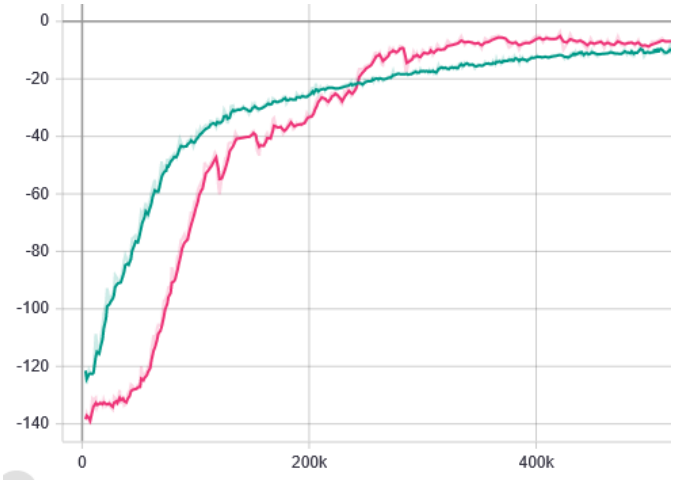


Figure 4.9: Total rewards of SAC (pink) and PPO (green).

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).

4.2.9 SAC hyperparameters

These parameters were the ones used when training with SAC:

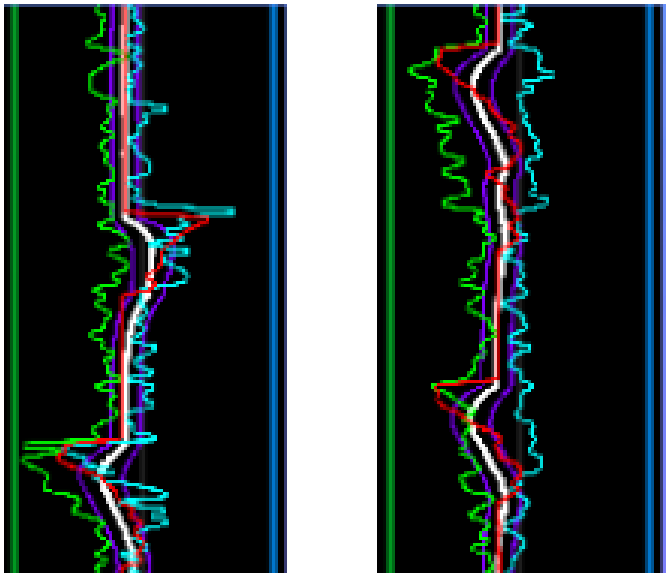


Figure 4.10: Comparison between SAC (left) and PPO (right).

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 4.2.5) 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 [2]
- 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 [3]

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

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

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

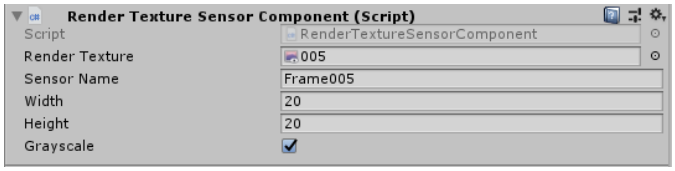


Figure 4.11: Example of a Render Sensor component.

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 [4] 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

Dynamic average and standard deviation

4.3.3 PPO vs. SAC

4.3.4 Behavioral cloning

4.3.5 Recurrent memory

4.3.6 Rare situations

4.3.7 Complexity of the task

[4] 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.

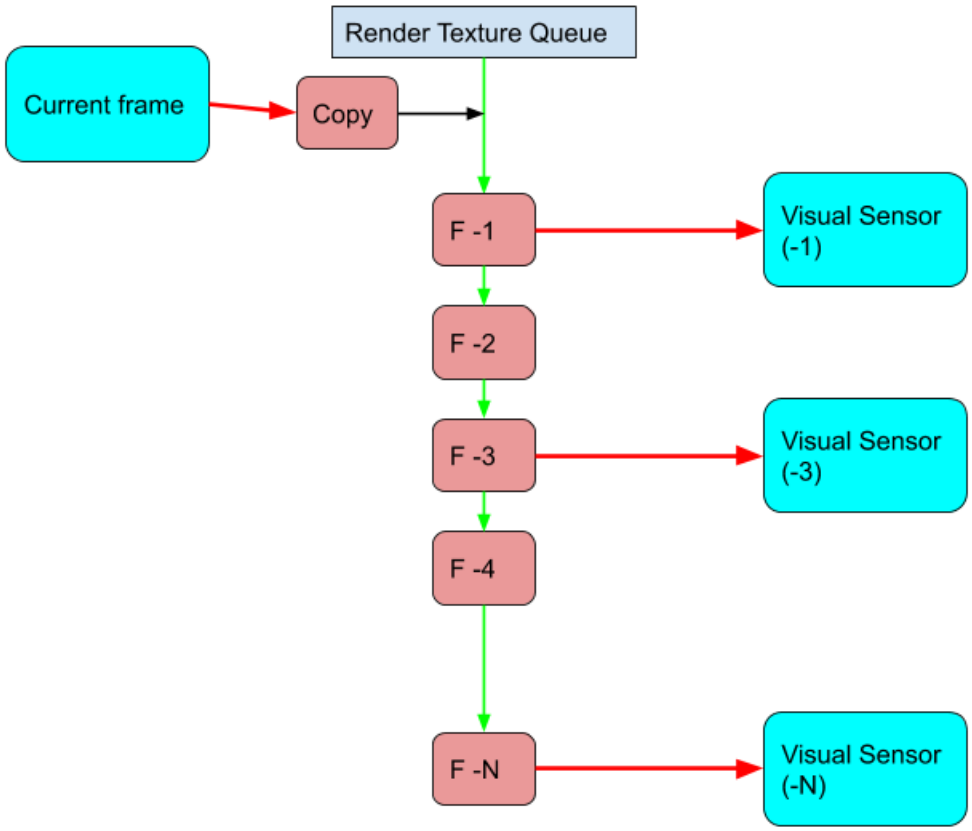


Figure 4.12: Render Queue: see that frame 2 isn't used as input for the neural network. Red arrows mean that the frame (or Render Texture) at the start is copied over the Render Texture at the end.

CHAPTER 5

Conclusions and Future Work

Contents

In this chapter, the conclusions of the work, as well as its future extensions are shown.

5.1 Conclusions

As preliminary conclusions, it has been proved that SAC policy is better than PPO to solve our task since it requires smaller datasets (or steps) and develops better behaviors than PPO (even though, PPO is better for testing since it is faster).

ML Agents makes easier the development of neural network and its inclusion in 3D environments, however it still has some errors that complicate this process. Moreover, its policies are not completely optimized for GPU usage, and some simple convolutions slow the training process heavily. Also, the latest versions of CUDA and tensorflow are not supported.

To end with, being able to design a custom neural network could improve the imitation results. It may be possible to do with ML Agents, but it could also be dangerous since it is necessary to modify source code in python. However, at this point we have already reached some of the milestones initially proposed by modeling reactive movements properly.

5.2 Future work

This framework could be continued in more complex games, using real player data to model its behaviors. However, I don't plan on doing it in the near future since the computing needed would be very high and it should be trained outside of Unity (and incorporated into real games).

Bibliography

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

[2] 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.

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

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

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}:

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

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

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

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

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

APPENDIX B

Dynamic standard deviation

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

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

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

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

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

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

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

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

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

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

[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

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

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

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

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

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

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

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

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

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