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 27, 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. 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 [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

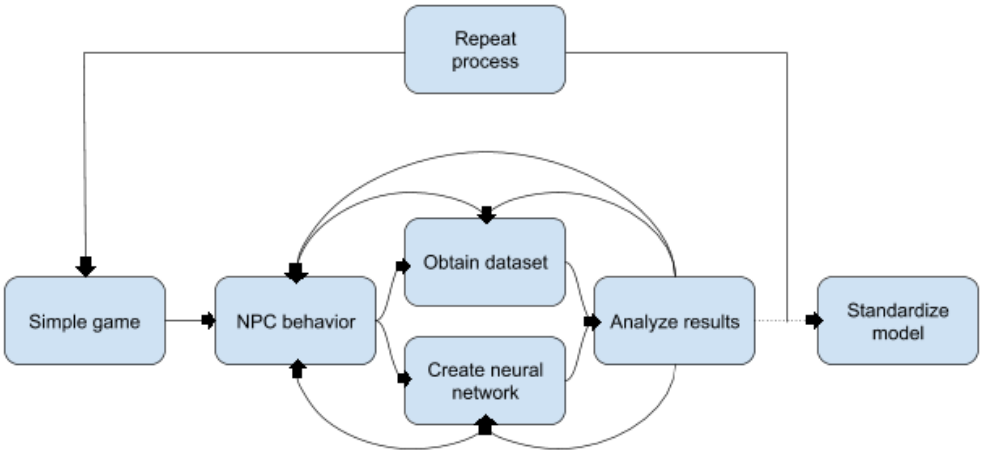


Figure 2.1: Planning graph

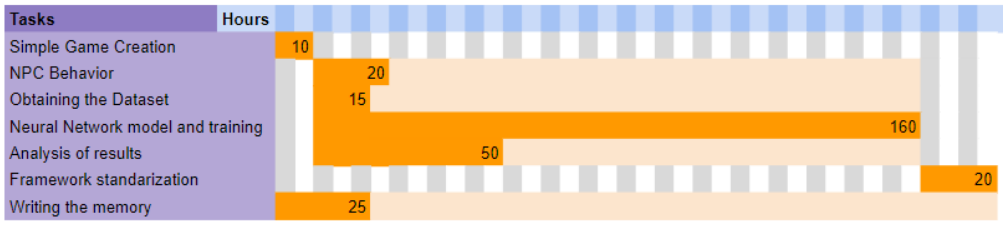


Figure 2.2: Initial planning

"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.

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 [6].

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,

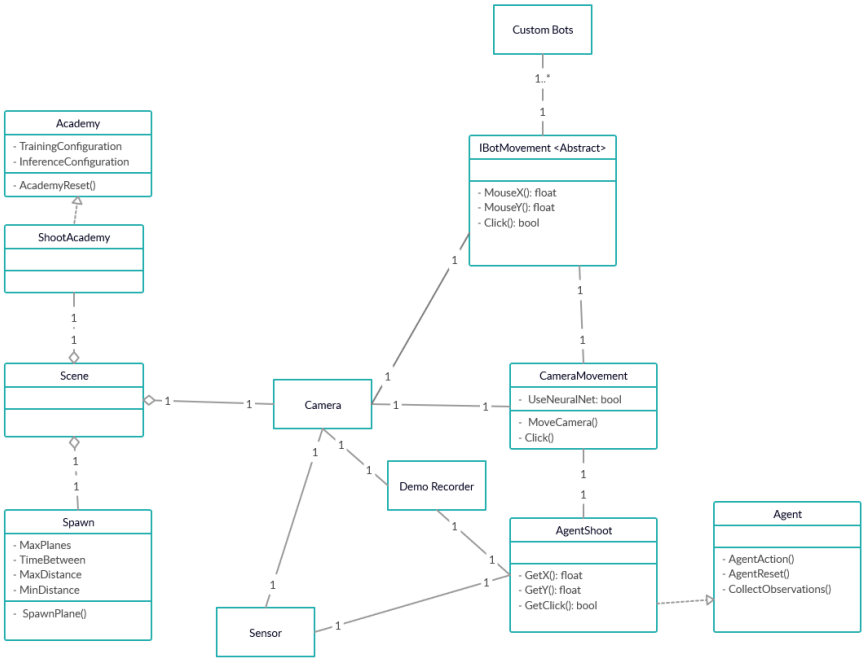


Figure 3.1: Class diagram of the training environment

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 [6].

3.3 System Architecture

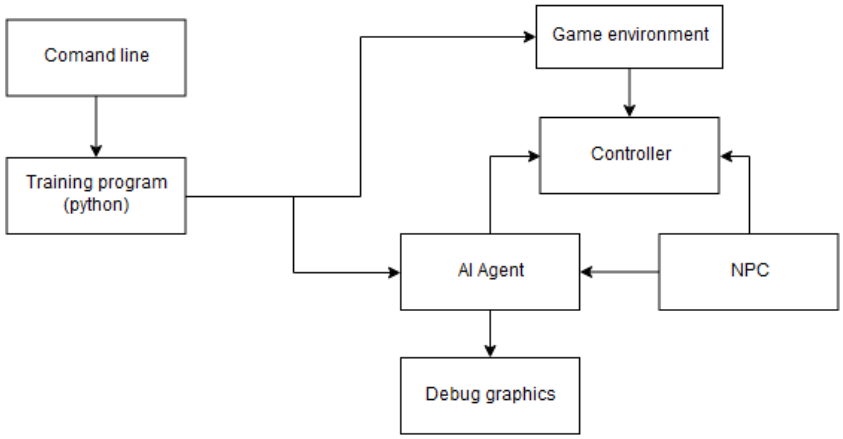


Figure 3.2: Diagram of the system architecture

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.

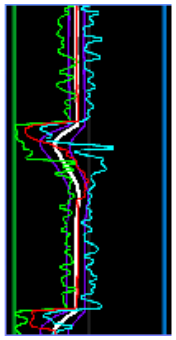
3.4 Interface Design

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.

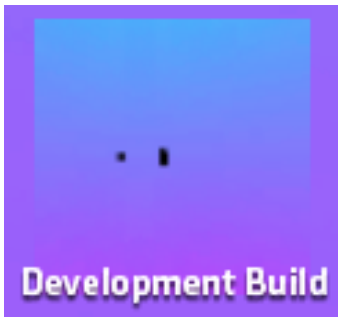
The graph displays 3 main lines (see Figure 3.4a): the red one is the move made by the bot, the blue and green ones are the highest and lowest guess of the bot (since the bot's movements are not perfect, these two lines can vary from being almost touching to being really wide). Other 3 lines in the back 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 4.2.5 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 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).

For more information about the trainings that were made for this section, see Appendix C.

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 [6]. 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 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.

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

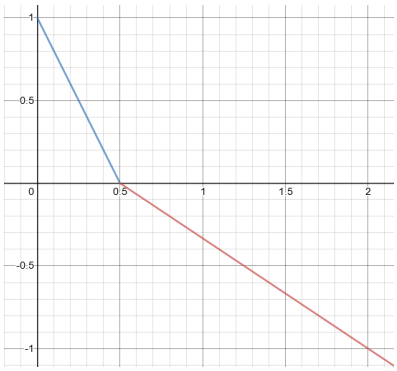


Figure 4.1: Rewards based in tolerable range.

Models trained using 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 4.2.4.

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.

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

[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.

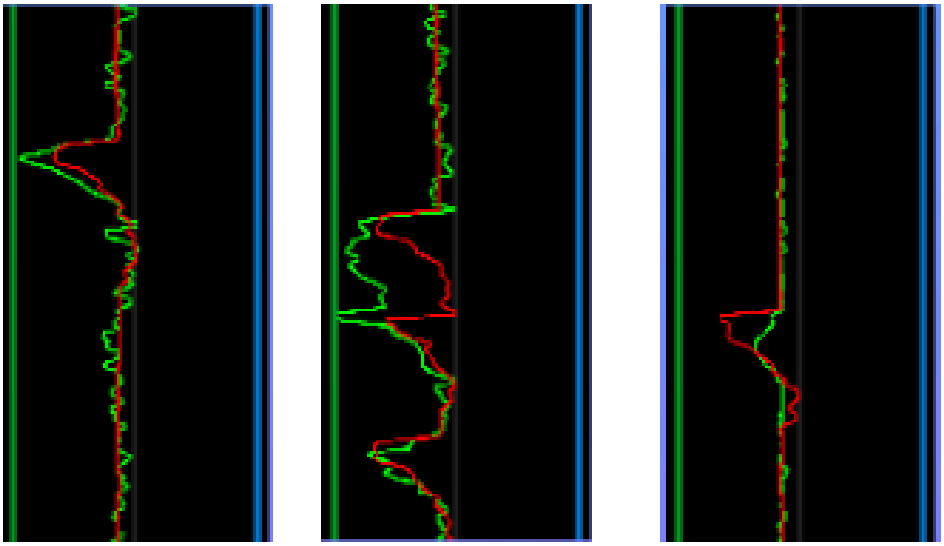


Figure 4.2: Bot movement (red) and neural network movement (green)

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 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

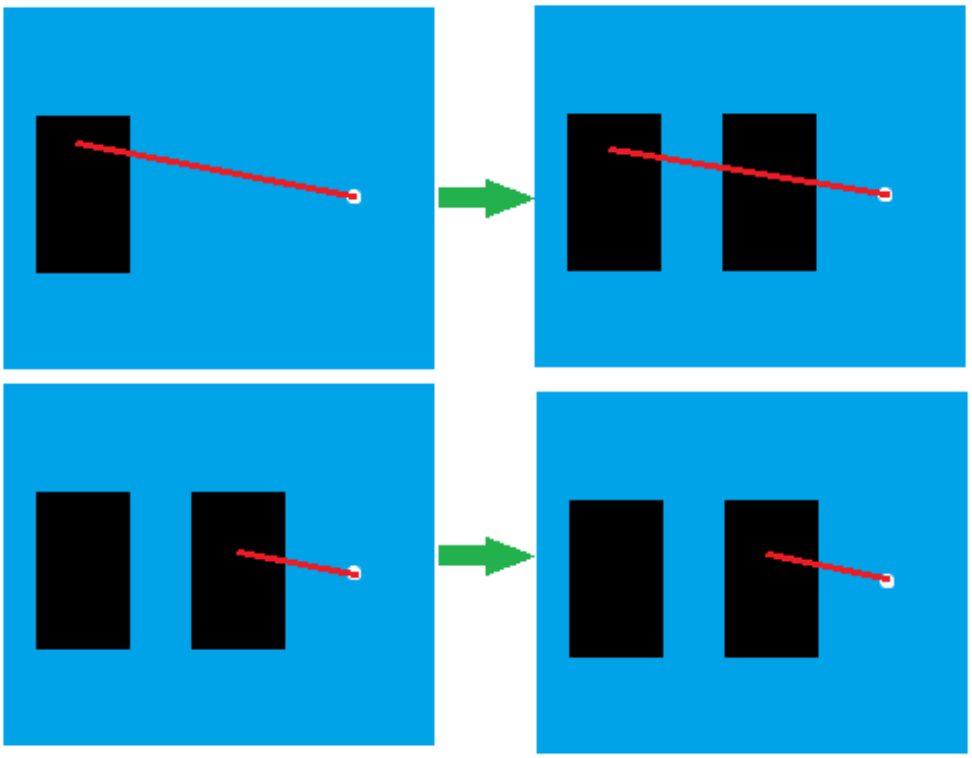


Figure 4.3: 2 situations that lead to different actions with the same frame

(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.

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.

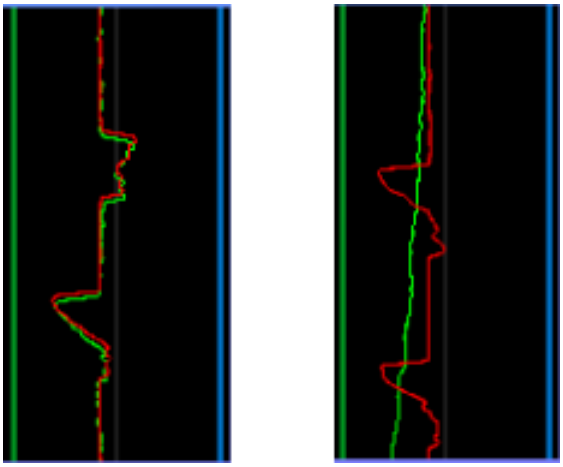


Figure 4.4: A badly trained model while training (up) and playing (down)

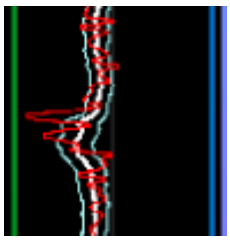


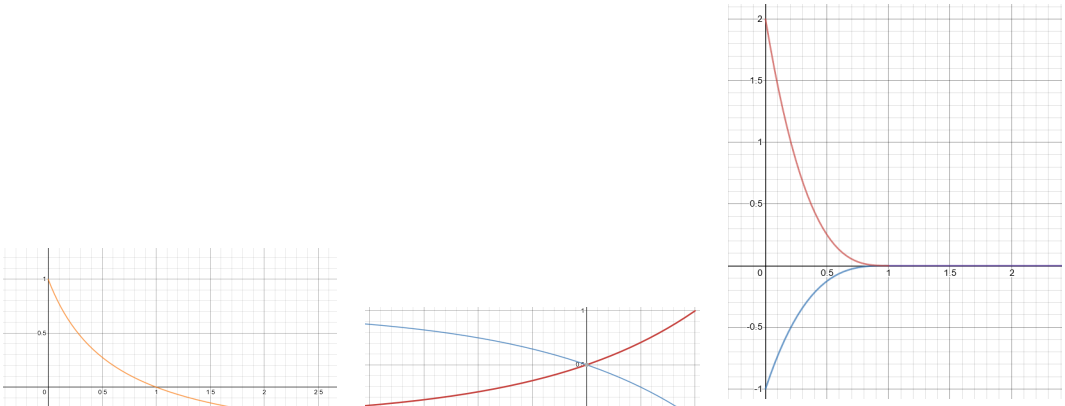
Figure 4.5: Weighted average and standard deviations of an irregular movement

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 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)



(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

- 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.

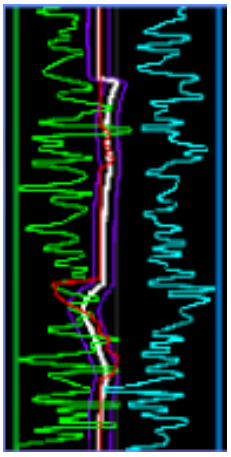


Figure 4.7: Trained model using SD rewards

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

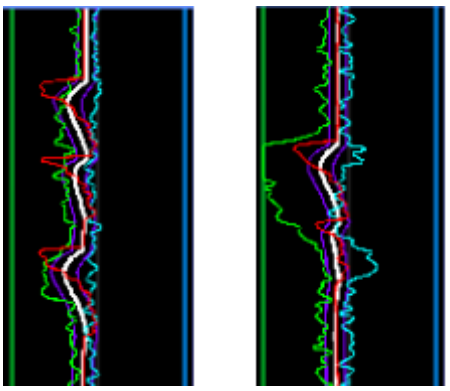


Figure 4.8: Coherence-based models with and different learning rate: left=2e-3, right=8e-3

4.2.8 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 [5] 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).

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.

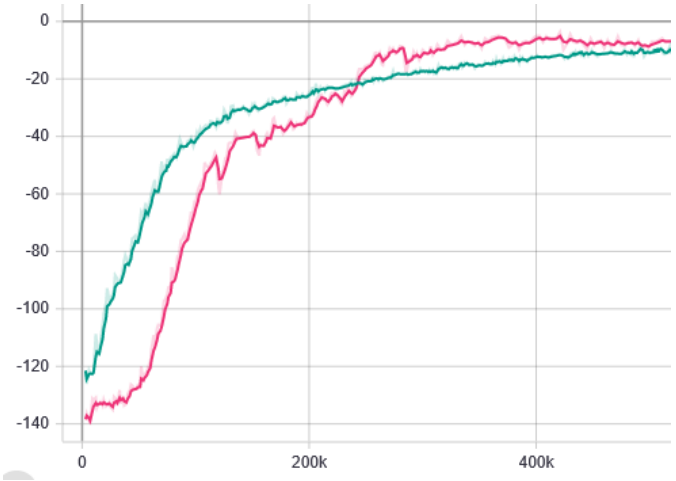


Figure 4.9: Total rewards of SAC (pink) and PPO (green).

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).

4.2.10 SAC hyperparameters

These parameters were the ones used when training with SAC:

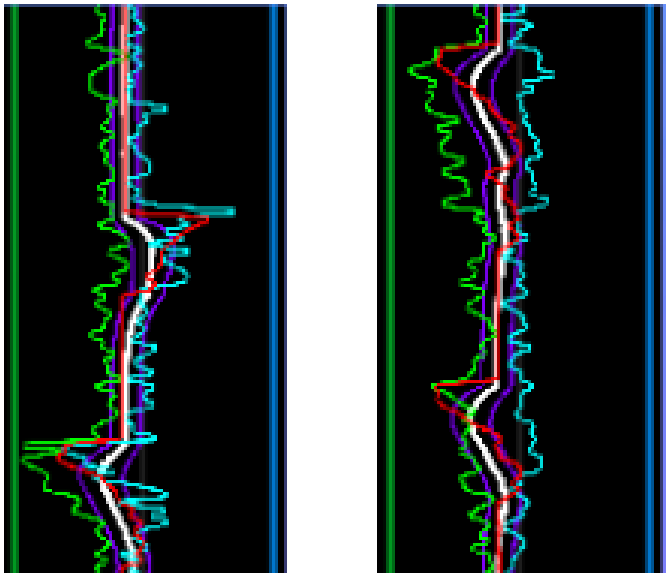


Figure 4.10: Comparison between SAC (left) and PPO (right).

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 4.2.6) 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).

For more information about the training sessions performed with the objective of creating agents with reaction time, see Appendix D.

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:

[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

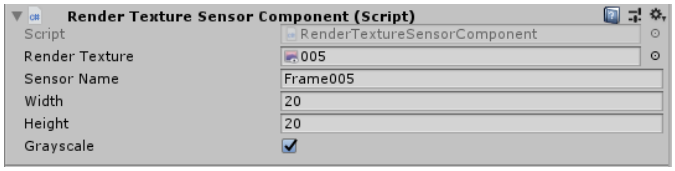


Figure 4.11: Example of a Render Sensor component.

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 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

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).

[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.

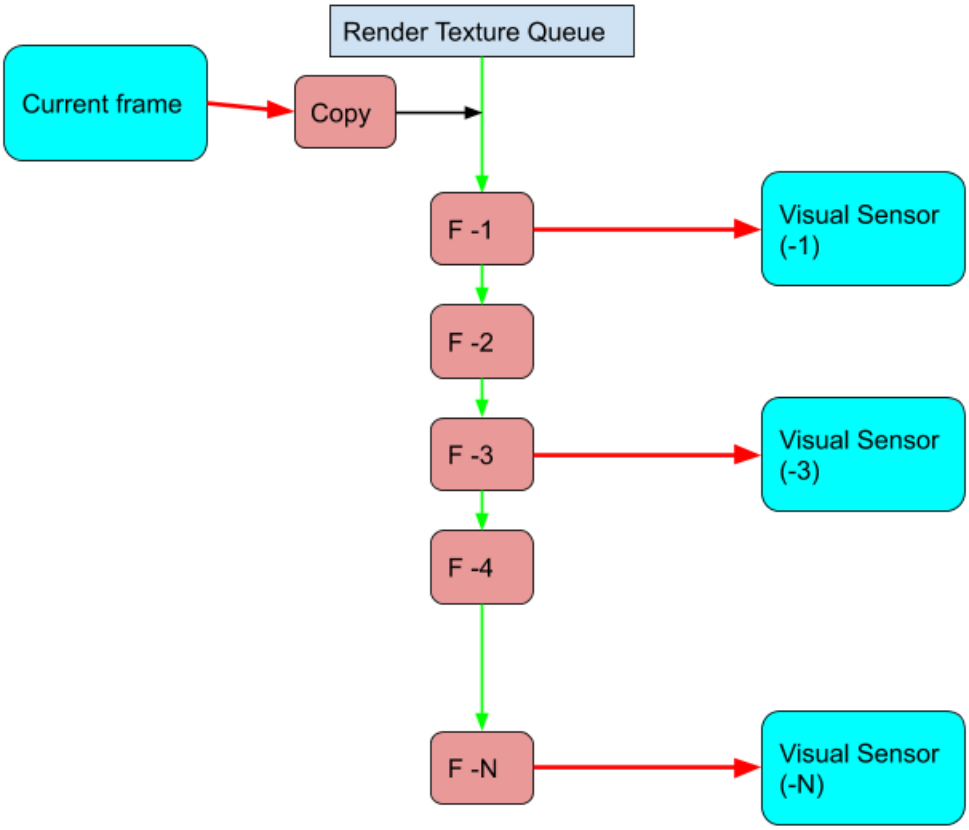


Figure 4.12: Render Queue: see that frame 2 isn't used as input for the neural network. Red arrows mean that the frame (or Render Texture) at the start is copied over the Render Texture at the end.

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).

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,

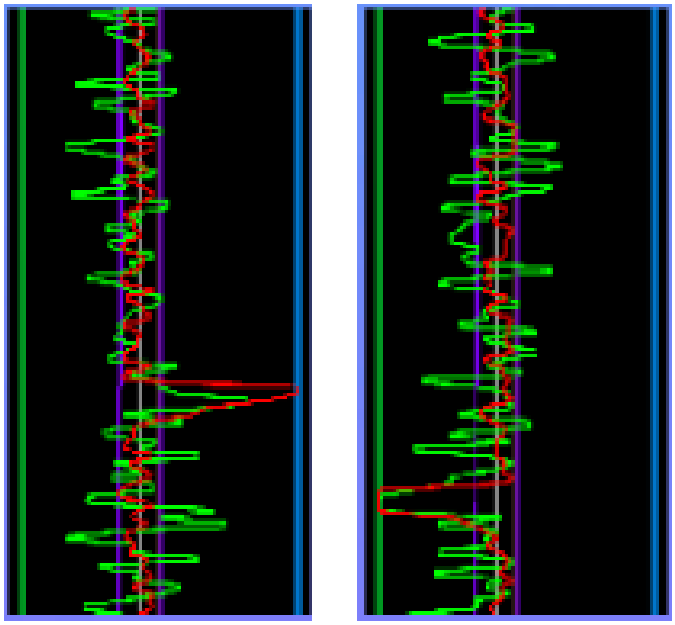


Figure 4.13: AI trained with tolerable range rewards. It adapts well to the impulses but not so much to the normal movement.

the AI can get confused about when to do an impulse and how [5]. Even though, this method is still the easiest way to model temporal noise, and with better balanced rewards it should perform well (See Figure 4.14).

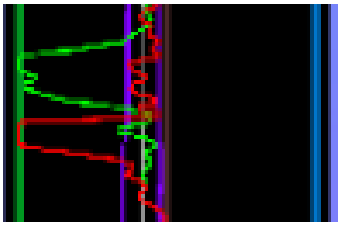


Figure 4.14: Case when the AI would receive a double punishment when doing an impulse: if not using 2 queues, the agent would have a big punish even though the impulse was correct (the bot can react at either moment)

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

[5] 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.

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 (either a constant or 0) if its move is also outside 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).

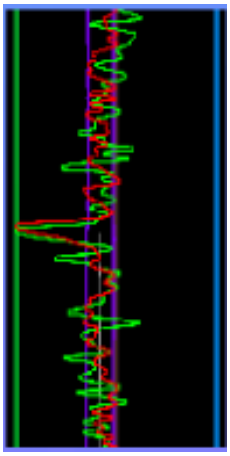


Figure 4.15: Model trained with 0.1 seconds of reaction time using this method

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 [6] 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 [7]. The best interpolation parameter for the real move and its closer standard deviation was 0.4 ($0.4 \cdot$ move $+ 0.6 \cdot$ standard deviation). Smaller interpolation parameters still made the range too big, and bigger interpolation parameters caused the standard deviation to increase too slowly [8] (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

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

[7] 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

[8] 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.

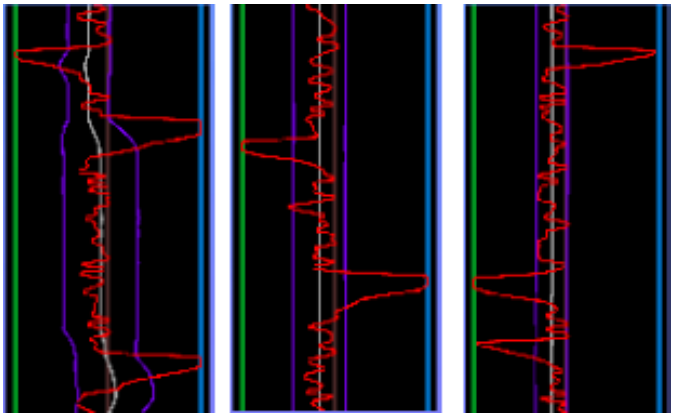


Figure 4.16: Standard variations and average of the same bot using last 60 values (left), dynamic values (mid) and dynamic values smoothed by a 0.6 interpolation (right)

average move, not doing any impulse. PPO follows the impulses since approaching them 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.

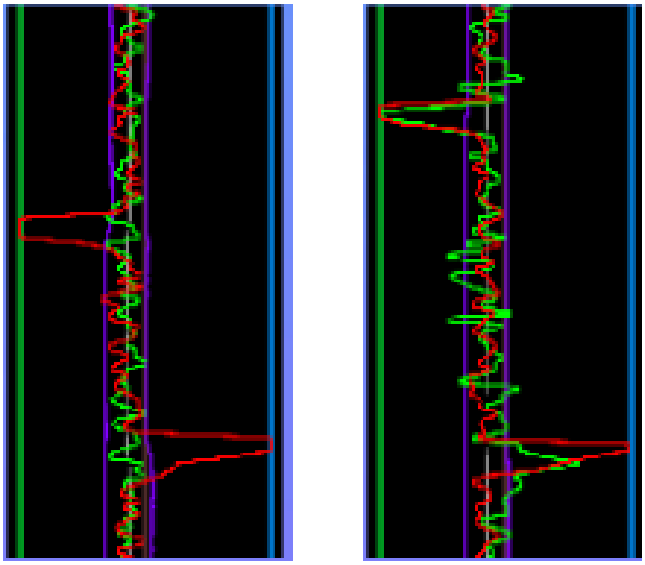


Figure 4.17: Comparison between an agent trained with SAC (left) and other trained using PPO (right). The bots in both cases have a reaction time of 0.1 seconds and 0.15 units of noise

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 is the simplest algorithm provided by ML Agents: 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).

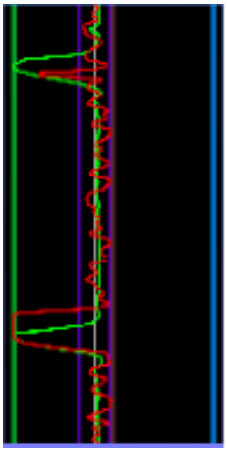


Figure 4.18: Model trained with BC in 2000 steps. The bot's impulses appear deformed because the agent is the one playing the game

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 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).

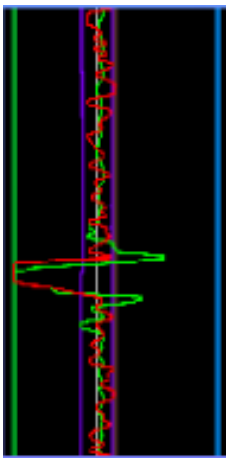
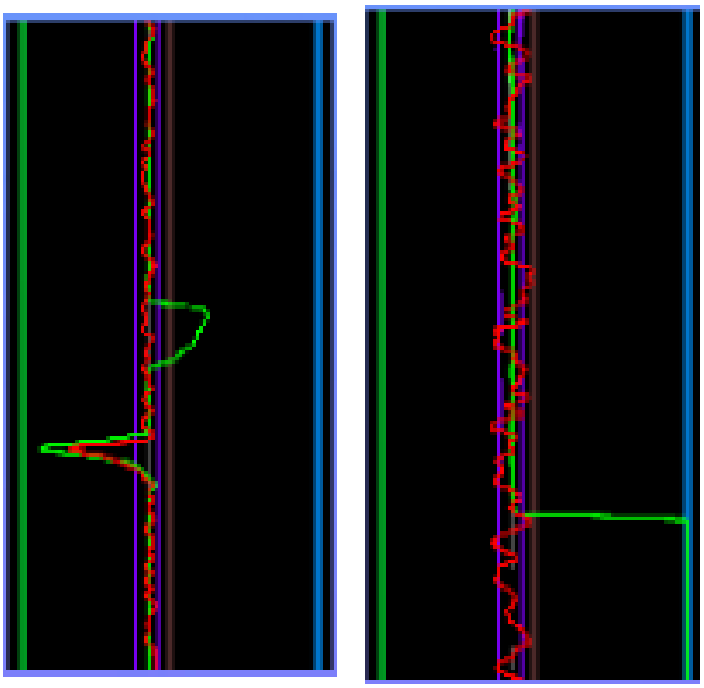


Figure 4.19: Model trained with BC in 7000 steps

One problem [9] 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

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

(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 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.

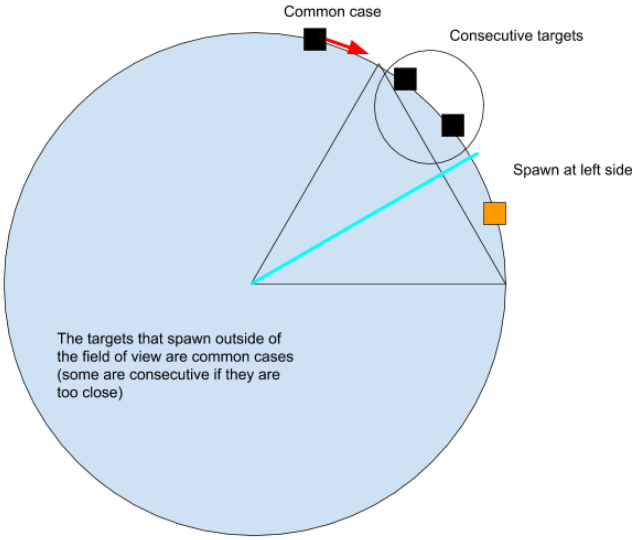


Figure 4.21: Some of the different situations the neural network can encounter in the game

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 often,

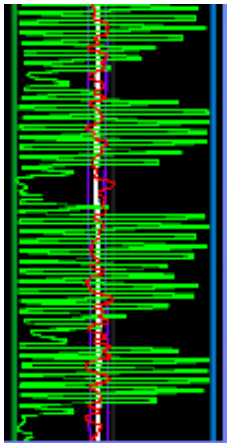


Figure 4.22: Neural network that has diverged

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.

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] 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] 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.

[6] 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}:

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

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

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

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

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

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

Reactive movements

This appendix contains some notes that were taken when (and after) training sessions related to modeling reactive behaviors.

Any parameter that does not appear in one training is set to default (see ./config/-trainer.config file).

Trained models with green titles are considered good or any improvement in the investigation. Models with red titles are considered failures.

Some of the models have not been saved, either because they don't perform well or they perform in much the same way as another model (agent).

Most of the notes should not be taken literally or as certain, as they usually are theories or preliminary conclusions drawn during the training itself.

20/4 10:42 (4_20_1042)

Trainer: PPO

batch size	32	beta	5.0e-3
buffer size	256	hidden units	256
learning rate	5.0e-4	max steps	5.0e5
normalize	false	num layers	1
summary freq	100	time horizon	5
extrinsic strength	1.0	extrinsic gamma	0.9

Camera Sensor: 40x40
Grayscale: false
Observation space size = 25 (eje X)

Action space size = 3

Episode steps = 3000
Total steps = 51400
Time = 1000s

It may be necessary to decrease the action vector to only 1 (whichever it is; the clicks are treated as automatic). Improve coherence of movement.

20/4 13:20 (4_20_1320)

Trainer: PPO

batch size	32	beta	5.0e-3
buffer size	256	hidden units	256
learning rate	5.0e-4	max steps	5.0e5
normalize	false	num layers	1
summary freq	**1000**	time horizon	5
extrinsic strength	1.0	extrinsic gamma	0.9

Camera Sensor: 40x40
Grayscale: false
Observation space size = 25 (eje X)
Action space size = 1

Episode steps = 3000
Total steps = 60000
Time = 1200s

It needs more training, or different parameters. It looks like it's moving in the direction but it's still "vibrating" a lot. It may be necessary to think about raytracing instead of a camera.

20/4 13:57 (4_20_1357)

Trainer: PPO

batch size	32	beta	5.0e-3
buffer size	256	hidden units	256
learning rate	5.0e-4	**max steps**	**5.0e6**
normalize	false	num layers	1
summary freq	1000	time horizon	5
extrinsic strength	1.0	extrinsic gamma	0.9

Camera Sensor: 40x40
Grayscale: false
Observation space size = 25 (eje X)
Action space size = 1

Episode steps = 3000
Total steps = 170000
Time = 2200s (+)

Same agent as 4_20_1320 (checkpoint)

20/4 14:40 (4_20_1440)

Trainer: PPO

batch size	32	beta	5.0e-3
buffer size	256	hidden units	256
learning rate	5.0e-4	max steps	5.0e6
normalize	false	num layers	1
summary freq	1000	time horizon	5
extrinsic strength	1.0	extrinsic gamma	0.9

Camera Sensor: 40x40
Grayscale: false
Observation space size = 25 (eje X)
Action space size = 1

Episode steps = 3000
Total steps = 500000
Time = 6770s (+)

It seems to have stabilized at a reward of 12,500 - 12,600.
In reality it stays practically paralyzed, moving very little (almost nothing).

21/4 12:32 (4_21_1232)

Trainer: PPO

batch size	32	beta	5.0e-3
buffer size	256	hidden units	256
learning rate	5.0e-4	max steps	5.0e6
normalize	false	num layers	1
summary freq	1000	time horizon	5
extrinsic strength	1.0	extrinsic gamma	0.9

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 25 (eje X)
Action space size = 1

Episode steps = 3000
Total steps = 350000
Time = 7530s

The Bot has been improved with reaction time and more speed. More demanding reward conditions.

The graph seems to show good behavior. Keeps training.

21/4 14:44 (4_21_1444)

Trainer: PPO

batch size	32	beta	5.0e-3
buffer size	256	hidden units	256
learning rate	5.0e-4	max steps	5.0e6
normalize	false	num layers	1
summary freq	1000	time horizon	5
extrinsic strength	1.0	extrinsic gamma	0.9

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 25 (eje X)
Action space size = 1

Episode steps = 3000
Total steps = -
Time = -

Same agent as 4_21_1232.

It doesn't work properly, but if you start the bot the graph works as it should. On the other hand, although the graph follows the same patterns, it stagnates at an amount below (maybe the movement limit). It also causes untrained cases to occur.

21/4 17:31 (4_21_1731)

Trainer: PPO

batch size	32	beta	5.0e-3
buffer size	256	hidden units	256
learning rate	5.0e-4	max steps	5.0e6
normalize	false	num layers	1
summary freq	1000	time horizon	5
extrinsic strength	1.0	extrinsic gamma	0.9

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 25 (eje X)
Action space size = 1

Episode steps = 3000
Total steps = 310000
Time = 6700s

Fixed that the agent reads half of the movement value (bug), instead of what the bot actually returned.

I think it uses the previous movements as "cheat sheet" to know what to do, maybe you have to detect the previous movements of the agent to learn or make the agent to move, and the bot to correct.

-I had to interrupt the training because the reaction was coming out of the graph (now it's -2 to 2, instead of 1).

-Another interruption, the vector action only returns normalized, so you have to change the value that happens to the camera (not the other way around).

The graph fits perfectly in the training, but I think it's because you can see what the real bot has done in the previous movement. It may be necessary to add curiosity rewards in following trainings.

The model seemed to make a good default move, even with a change of direction, but then it tends to turn at full speed without stopping. It should certainly stop reading what the bot does in the last move.

22/4 10:00 (4_22_1000)

Trainer: PPO

batch size	32	beta	5.0e-3
buffer size	256	hidden units	256
learning rate	5.0e-4	max steps	5.0e6
normalize	false	num layers	1
summary freq	1000	time horizon	5
extrinsic strength	1.0	**extrinsic gamma**	**0.95**

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 25 (eje X)
Action space size = 1

Episode steps = 2000
Total steps = 41000
Time = 900s

The previous movements have been changed to those of the agent.
It tends towards chaos.

22/4 10:31 (4_22_1031)

Trainer: PPO

batch size	32	beta	5.0e-3
buffer size	256	hidden units	256
learning rate	5.0e-4	max steps	5.0e6
normalize	false	num layers	1
summary freq	1000	time horizon	5
extrinsic strength	1.0	**extrinsic gamma**	**0.9**

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 0
Action space size = 1

Episode steps = 3000
Total steps = 185000
Time = 4000s

All observation other than the camera has been removed. The bot reacts in time 0 now. An idle address has also been removed.

It was on the right track but tends to go into chaos after 1h.

22/4 11:43 (4_22_1143)

Trainer: PPO

batch size	32	beta	5.0e-3
buffer size	256	hidden units	256
learning rate	**2.5e-4**	max steps	5.0e6
normalize	false	num layers	1
summary freq	1000	time horizon	5
extrinsic strength	1.0	extrinsic gamma	0.9
curiosity strength	**0.01**	**curiosity gamma**	**0.99**
curiosity encoding size	**256**		

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 0
Action space size = 1

Episode steps = 3000
Total steps = 180000
Time = 4100s

Decreased idle randomness and increased acceptable range.

The graph fits well, although with some noise (perhaps out of curiosity), but it also tends to hit the planes that appear on the opposite side (this didn't happen before, it only reacted well to those that appeared in front). In viewport it also seems very similar, and the noise is not so noticeable.

Saved as a checkpoint.

–Note: the behaviour is a bit different because when training the screen is displayed in a 16:9 ratio (instead of 1:1 from the editor), that makes the camera react a bit differently.

22/4 13:19 (4_22_1319)

Trainer: PPO

batch size	32	beta	5.0e-3
buffer size	256	hidden units	256
learning rate	2.5e-4	max steps	5.0e6
normalize	false	num layers	1
summary freq	1000	time horizon	5
extrinsic strength	1.0	extrinsic gamma	0.9
curiosity strength	0.01	curiosity gamma	0.99
curiosity encoding size	256		

Render Target Sensor: 64x64

Grayscale: false
Observation space size = 0
Action space size = 1

Episode steps = 3000
Total steps = 256000
Time = 1500s (+)

Correct the proportion in the build. Actually, as the render target's camera goes out from the agent's one, it doesn't cause problems. However the isVisible should be changed in relation to that camera.

It tends again to chaos.

22/4 13:52 (4__22__1352)

Trainer: PPO

batch size	32	beta	5.0e-3
buffer size	256	hidden units	256
learning rate	**1.0e-4**	max steps	5.0e6
normalize	false	num layers	1
summary freq	1000	time horizon	5
extrinsic strength	1.0	extrinsic gamma	0.9

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 0
Action space size = 1

Episode steps = 3000
Total steps = 373000
Time = 8050s

Before training the reward is increased by being in range.

VERY slow training, consider setting a "learning rate schedule" so that the learning rate goes from more to less (but it may already be by default, you would have to decrease the number of training steps).

It seems more or less stable (like the previous one achieved).

22/4 16:15 (4__22__1615)

Trainer: PPO

batch size	32	beta	5.0e-3
buffer size	256	**hidden units**	**128**
learning rate	**1.0e-3**	**learning rate schedule**	**linear**
max steps	**5.0e4**	normalize	false
num layers	**2**	summary freq	1000
time horizon	5	extrinsic strength	1.0
extrinsic gamma	0.9		

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 0
Action space size = 1

Episode steps = 3000
Total steps = 50000
Time = 1075s

Learning rate added (so that it can be changed explicitly) The steps are now determined. The structure of the network has also been changed.

It has reached a reward of -0.313, maybe for increasing the number of layers.

22/4 16:40 (4_22_1640)

Trainer: PPO

batch size	32	beta	5.0e-3
buffer size	256	**hidden units**	**256**
learning rate	1.0e-3	learning rate schedule	linear
max steps	**1.0e5**	normalize	false
num layers	**1**	summary freq	1000
time horizon	5	extrinsic strength	1.0
extrinsic gamma	0.9		

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 0
Action space size = 1

Episode steps = 3000
Total steps = 100000
Time = 2160s

It may be necessary to lower the initial learning rate to increase the effectiveness of the training. In the first 50,000 steps it makes very little progress (from -.524 to -.415). In the end it reaches -.353.

22/4 17:19 (4_22_1719)

Trainer: PPO

batch size	**1024**	beta	5.0e-3
buffer size	**4096**	hidden units	256
learning rate	**2.0e-4**	**learning rate schedule**	**constant**
max steps	**2.0e5**	normalize	false
num epoch	**5**	num layers	1
summary freq	1000	**time horizon**	**8**
extrinsic strength	1.0	extrinsic gamma	0.9

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 0
Action space size = 1

Episode steps = 3000
Total steps = 25000
Time = 540s

Several changes to adapt the network to continuous space. Constant Learning Rate to check when it converges (or diverges) Does not converge after 25000 steps.

22/4 17:54 (4_22_1754)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	4096	hidden units	256
learning rate	**1.0e-4**	learning rate schedule	constant
max steps	**2.0e6**	normalize	false
num epoch	**3**	num layers	1
summary freq	1000	**time horizon**	**32**
extrinsic strength	1.0	extrinsic gamma	0.9

Render Target Sensor: 64x64
Grayscale: false

Observation space size = 0
Action space size = 1

Episode steps = 3000
Total steps = 115000
Time = 2420s

Long training to try to get it stabilized.
–It doesn't stabilize after 100,000 steps, I think it has a bigger problem with how the rewards are given. I'll turn them into linear.

22/4 18:42 (4_22_1842)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	4096	hidden units	256
learning rate	**5.0e-4**	**learning rate schedule**	**linear**
max steps	2.0e6	normalize	false
num epoch	3	num layers	1
summary freq	1000	time horizon	32
extrinsic strength	1.0	extrinsic gamma	0.9

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 0
Action space size = 1

Episode steps = 3000
Total steps = 61000
Time = 1200s

It shows no sign of stabilizing, so the range of success may still be very demanding.
To save time in converging to the original bot function, I have thought to use learning curriculum that determines the thresholds of the rewards, making them smaller and smaller.

23/4 10:49 (4_23_1049)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	4096	hidden units	256
learning rate	5.0e-4	learning rate schedule	linear
max steps	2.0e6	normalize	false
num epoch	3	num layers	1
summary freq	1000	time horizon	32
extrinsic strength	1.0	extrinsic gamma	0.9

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 0
Action space size = 1

Episode steps = 3000
Total steps = 116000
Time = 2400s

Training with curriculum. Still having trouble converging.

23/4 12:46 (4_23_1246)

Trainer: PPO

batch size	**32**	beta	5.0e-3
buffer size	**256**	hidden units	256
learning rate	**3.0e-4**	learning rate schedule	linear
max steps	**1.0e6**	normalize	false
num epoch	3	num layers	1
summary freq	1000	**time horizon**	**5**
extrinsic strength	1.0	extrinsic gamma	0.9

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 0
Action space size = 1

Episode steps = 2000
Total steps = 394000
Time = 8500s

Increased the minimum reward to pass the level. Changes in variables until a stable one is achieved.

It manages to stabilize the idle movement (after quite a while) but doesn't do impulses well.

23/4 15:51 (4_23_1551)

Trainer: PPO

batch size	**1024**	beta	5.0e-3
buffer size	**4096**	hidden units	256
learning rate	**5.0e-4**	learning rate schedule	linear
max steps	1.0e6	normalize	false
num epoch	3	num layers	1
summary freq	1000	**time horizon**	**8**
extrinsic strength	1.0	**extrinsic gamma**	**0.8**
curiosity strength	**0.05**	**curiosity gamma**	**0.9**
curiosity encoding size	**128**		

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 0
Action space size = 1

Episode steps = 2000
Total steps = 453000
Time = 7300s

Added penalty for out of range (beyond maximumRange) and curiosity. Also the extrinsic gamma is lowered to make the present (at this time) count more.

Reaches level 2 with 0.4 reward but shows no improvement. The reward range may need to be increased.

23/4 19:15 (4_23_1915)

Trainer: PPO

batch size	1024	**beta**	**5.0e-4**
buffer size	4096	hidden units	256
learning rate	**1.0e-3**	learning rate schedule	linear
max steps	1.0e6	normalize	false
num epoch	3	**num layers**	**2**
summary freq	1000	**time horizon**	**2048**
extrinsic strength	1.0	**extrinsic gamma**	**0.9**
curiosity strength	0.05	curiosity gamma	0.9
curiosity encoding size	128		

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 0
Action space size = 1

Episode steps = 2000
Total steps = 760000
Time = 16700s

Training by watching tensorboard. Test with 2 layers, it's probably slower.
The training is too slow, although the graphs seem to have correct progression. The cause may be the form of the reward function or the fact that it had a higher range before (up to -120).

24/4 10:26 (4__24__1026)

Trainer: PPO

batch size	**32**	**beta**	**5.0e-3**
buffer size	**256**	hidden units	256
learning rate	**2.5e-4**	learning rate schedule	linear
max steps	**5.0e6**	normalize	false
num epoch	3	**num layers**	**1**
summary freq	1000	**time horizon**	**5**
extrinsic strength	1.0	extrinsic gamma	0.9
curiosity strength	**0.01**	**curiosity gamma**	**0.99**
curiosity encoding size	**256**		

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 0
Action space size = 1

Episode steps = 3000
Total steps = 112000
Time = 2400s

Training with old reward functions (TR=0.005), and with the old parameters In 1500-2000s it starts to get reasonably close. The graphs don't show any strange behaviour, but the rewards are always negative (the network should break in 180k-200k steps, now it is at 100k). The model is correct.

24/4 16:00 (4_24_1600)

Trainer: PPO

batch size	32	beta	5.0e-3
buffer size	256	hidden units	256
learning rate	2.5e-4	learning rate schedule	linear
max steps	5.0e6	normalize	false
num epoch	3	num layers	1
summary freq	1000	time horizon	5
extrinsic strength	1.0	extrinsic gamma	0.9
curiosity strength	0.01	curiosity gamma	0.99
curiosity encoding size	256		

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 0
Action space size = 1

Episode steps = 3000
Total steps = 435000
Time = 10000s

Training with the same parameters and without curriculum (with TR=0.01, MR=1). Using linear rewards but multiplied by 40.

From 80000 steps or a little earlier you can see that it adapts to the form (except for some transitions between jumps).

At 150000 steps it almost acts like one of the previous models: it goes from -16 to -2 reward and the graphs don't show anything strange.

At 190000 steps first positive reward (0.340).

At 275,000 steps the idle stages are virtually out of noise. However, it can be seen that the network does not fit well when there are more than 2 targets at once, this may be because of how the bot algorithm prioritizes them and the fact that the network still has no memory. The reward is 3.

Afterwards, the reward increases steadily because it adapts to the idle with more precision, but the more time goes by the less I am convinced about how it adapts to some impulses (it may be due to underfitting, since it doesn't have the memory installed yet either in image or in movements).

Putting it in Unity is less convincing because it doesn't follow the same movement if it's executed autonomously (it's smoothed out even more, the impulses are slower).

In the following tests: increase the learning-rate and put the values normalized; in another one increase even more the multiplication factor. Changing the shape of the function may improve (or worsen) the learning curve. After the 2 tests, it would be convenient to put the movement memory back (25 previous values).

24/4 19:35 (4_24_1935)

Trainer: PPO

batch size	32	beta	5.0e-3
buffer size	256	hidden units	256
learning rate	**1.0e-3**	learning rate schedule	linear
max steps	5.0e6	normalize	false
num epoch	3	num layers	1
summary freq	1000	time horizon	5
extrinsic strength	1.0	extrinsic gamma	0.9
curiosity strength	0.01	curiosity gamma	0.99
curiosity encoding size	256		

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 0
Action space size = 1

Episode steps = 3000
Total steps = 190000
Time = 4300s

Standardized rewards, but learning rate multiplied by 40.

1.0e-2 is too much learning rate, it starts at the extremes. 5.0e-3 too. 2.5e-3 seems stable at the start but becomes unsettled on the first pass.

1.0e-3 seems to be the stable maximum, which is 4 times more than the previous one.

It doesn't seem to be as effective, in 100000 steps it doesn't catch the jumps well, even if the reward increases from -0.43 to -0.2

24/4 21:30 (4_24_2130)

Trainer: PPO

batch size	32	beta	5.0e-3
buffer size	256	hidden units	256
learning rate	**2.5e-4**	learning rate schedule	linear
max steps	5.0e6	normalize	false
num epoch	3	num layers	1
summary freq	1000	time horizon	5
extrinsic strength	1.0	extrinsic gamma	0.9
curiosity strength	0.01	curiosity gamma	0.99
curiosity encoding size	256		

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 0
Action space size = 1

Episode steps = 3000
Total steps = 366000
Time = 8400s

Reward multiplied by 80, parameters of 4_24_1600.
Similar results are obtained as when multiplied by 40.
In this case you get to 0 at 196000, and to -4 at about 150000 (practically the same proportion).
It would be advisable to adjust the learning curriculum again with these rewards (or those of *40) and check whether or not it is faster. Then add the movement memory again.

25/4 19:15 (4_25_1915)

Trainer: PPO

batch size	32	beta	5.0e-3
buffer size	256	hidden units	256
learning rate	2.5e-4	learning rate schedule	linear
max steps	5.0e6	normalize	false
num epoch	3	num layers	1
summary freq	1000	time horizon	5
extrinsic strength	1.0	extrinsic gamma	0.9
curiosity strength	0.01	curiosity gamma	0.99
curiosity encoding size	256		

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 0
Action space size = 1

Episode steps = 3000
Total steps = 305000
Time = 7000s

Training as 4_24_1600 but with curriculum (adapted, TR from 0.5 to 0.01) It should approach the form in 80000 steps and have positive reward (in the last level) at 190000 to equal the training without curriculum.

In the end it seems to work similarly with less restrictive parameters, but as the minimum conditions are unbalanced it has not passed the first level (behaving almost the same). A next training will be done by setting the 0 of limits to overcome levels.

Probably the curriculum can be used to increase the restrictions, but this requires more observations.

The parameter of curiosity is of no use to me.

26/4 14:21 (4_26_1421)

Trainer: PPO

batch size	32	beta	5.0e-3
buffer size	256	hidden units	256
learning rate	2.5e-4	learning rate schedule	linear
max steps	5.0e6	normalize	false
num epoch	3	num layers	1
summary freq	1000	time horizon	5
extrinsic strength	1.0	extrinsic gamma	0.9
curiosity strength	0.01	curiosity gamma	0.99
curiosity encoding size	256		

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 25
Action space size = 1

Episode steps = 3000
Total steps = 573000
Time = 13200s

Training without curriculum, with the 25 added observations (from the agent). The change of direction has also been activated, to see if it learns the context.

The reward is stagnant over -10 from the 160000 steps. The graph shows how it oscillates over the center. For future models it may be necessary to increase the complexity of the training and the network. Another way is to make the network in control and the bot indicate how to act.

26/4 18:15 (4_26_1815)

Trainer: PPO

batch size	**1024**	beta	5.0e-3
buffer size	**8192**	hidden units	256
learning rate	2.5e-4	learning rate schedule	linear
max steps	5.0e6	normalize	false
num epoch	**5**	**num layers**	**2**
summary freq	1000	**time horizon**	**256**
extrinsic strength	1.0	extrinsic gamma	0.9

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 25
Action space size = 1

Episode steps = 3000
Total steps = 763000
Time = 16750s

It should improve accuracy, but be much slower.
Idea: put as previous values of movement the average between the bot and the agent.
The graph recognizes impulses very well, but the idle oscillates between 0 with quite a lot of noise. This must be because the network doesn't remember the previous images, and doesn't distinguish when to change direction. The growth is much slower and it stagnates again on the same level of reward. This could be solved by pre-recording the image or simplifying the algorithm in cases where more than one target appears.

27/4 10:35 (4_27_1035)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	8192	hidden units	256
learning rate	2.5e-4	learning rate schedule	linear
max steps	5.0e6	normalize	false
num epoch	5	num layers	2
summary freq	1000	time horizon	256
extrinsic strength	1.0	extrinsic gamma	0.9

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 25
Action space size = 1

Episode steps = 3000

Total steps = 132000
Time = 2900s

Improved bot so that it always chooses the closest target, at the moment the change of direction is removed. In this one we are going to use curriculum to see if it improves the learning speed.

The curriculum doesn't seem effective, after almost 3000s it has only reached -14 of reward (from the easy level).

27/4 11:23 (4_27_1123)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	8192	hidden units	256
learning rate	2.5e-4	learning rate schedule	linear
max steps	5.0e6	normalize	false
num epoch	5	num layers	2
summary freq	1000	time horizon	256
extrinsic strength	1.0	extrinsic gamma	0.9

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 25
Action space size = 1

Episode steps = 3000
Total steps = 366000
Time = 8000s

Same model as before but without curriculum. Similar result, it stagnates at -10.

27/4 13:39 (4_27_1339)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	8192	hidden units	256
learning rate	2.5e-4	learning rate schedule	linear
max steps	5.0e6	normalize	false
num epoch	5	num layers	2
summary freq	1000	**time horizon**	**32**
extrinsic strength	1.0	extrinsic gamma	0.9

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 25
Action space size = 1

Episode steps = 3000
Total steps = 303000
Time = 6600s

Tests with different parameters: time horizon (from 256 to 32) does not seem to change anything.

27/4 15:33 (4_27_1533)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	8192	**hidden units**	**64**
learning rate	**5.0e-4**	**learning rate schedule**	**constant**
max steps	5.0e6	normalize	false
num epoch	**3**	**num layers**	**3**
summary freq	1000	**time horizon**	**256**
extrinsic strength	1.0	extrinsic gamma	0.9

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 25
Action space size = 1

Episode steps = 3000
Total steps = 942000
Time = 19900s

Different network architecture. It has more layers but fewer units per layer.
It's close to the shape but still has a lot of noise, plus it's very slow. The curiosity parameter may need to be retrieved and add more observations. After many hours it gets to -6 reward.
It may also be possible to reduce the length of each episode.

28/4 18:00 (4_28_1800)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	8192	**hidden units**	**128**
learning rate	**4.0e-4**	learning rate schedule	constant
max steps	5.0e6	normalize	false
num epoch	3	**num layers**	**2**
summary freq	1000	time horizon	256
extrinsic strength	1.0	extrinsic gamma	0.9
curiosity strength	**0.05**	**curiosity gamma**	**0.99**
curiosity encoding size	**256**		

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 25
Action space size = 2

Episode steps = 1000
Total steps = 74000
Time = 1550s

Drastic changes in rewards (factors and standard deviations). Now returns 2 values in vector action (the expected maximum and minimum).

With this test I know that the model can set the maximum and minimum. Incomplete.

28/4 18:33 (4_28_1833)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	8192	hidden units	128
learning rate	4.0e-4	learning rate schedule	constant
max steps	5.0e6	normalize	false
num epoch	3	num layers	2
summary freq	1000	time horizon	256
extrinsic strength	1.0	extrinsic gamma	0.9
curiosity strength	0.05	curiosity gamma	0.99
curiosity encoding size	256		

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 25
Action space size = 2

Episode steps = 1000
Total steps = 360000
Time = 7600s

Same test as before, but longer. It manages to wrap the movement but with too wide a range.

29/4 10:34 (4_29_1034)

Trainer: PPO

batch size	1024	**beta**	**1.0e-2**
buffer size	8192	**hidden units**	**256**
learning rate	**6.0e-4**	**learning rate schedule**	**linear**
max steps	5.0e6	normalize	false
num epoch	**10**	num layers	2
summary freq	1000	**time horizon**	**512**
extrinsic strength	1.0	extrinsic gamma	0.9
curiosity strength	0.05	curiosity gamma	0.99
curiosity encoding size	256		

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 25
Action space size = 2

Episode steps = 1000
Total steps = 226000
Time = 5400s

The reward now takes into account the relationship between standard deviations.

Like the previous one, it stagnates at a certain point (this time at -1, but with similar behavior).

29/4 12:09 (4_29_1209)

Trainer: PPO

batch size	1024	beta	1.0e-2
buffer size	8192	hidden units	256
learning rate	**2.0e-5**	learning rate schedule	linear
max steps	5.0e6	normalize	false
num epoch	10	num layers	2
summary freq	1000	time horizon	512
extrinsic strength	1.0	extrinsic gamma	0.9
curiosity strength	0.05	curiosity gamma	0.99
curiosity encoding size	256		

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 25
Action space size = 2

Episode steps = 1000
Total steps = 200000
Time = 4700s

Same training but with a lower learning rate (and scores multiplied by 20). Doesn't seem to advance too far so it's not saved.

29/4 13:34 (4_29_1334)

Trainer: PPO

batch size	1024	**beta**	**5.0e-3**
buffer size	8192	hidden units	256
learning rate	**5.0e-5**	**learning rate schedule**	**constant**
max steps	5.0e6	normalize	false
num epoch	10	num layers	2
summary freq	1000	time horizon	512
extrinsic strength	1.0	**extrinsic gamma**	**0.8**
curiosity strength	**0.1**	**curiosity gamma**	**0.9**
curiosity encoding size	**128**		

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 25
Action space size = 2

Episode steps = 1000
Total steps = 297000
Time = 7920s

For some reason the 2 curves end up below the bot value, and it doesn't get much of a penalty. There must be some mistake in the rewards that allows him not to learn properly and still increase the reward.

29/4 18:27 (4__29__1827)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	8192	hidden units	256
learning rate	**5.0e-4**	learning rate schedule	constant
max steps	5.0e6	normalize	false
num epoch	**5**	num layers	2
summary freq	1000	time horizon	512
extrinsic strength	1.0	extrinsic gamma	0.8
curiosity strength	**0.01**	curiosity gamma	0.9
curiosity encoding size	128		

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 25
Action space size = 2

Episode steps = 1000
Total steps = 265000
Time = 6200s

The reward rises to positive but stabilizes at +12 after 130,000 steps. It reaches a positive reward at 90000.

The network learns quickly to separate the maximum and minimum but with very wide ranges. It would be necessary to increase the requirement by penalizing more neutral scores (and perhaps vary how each reward factor affects it).

30/4 10:43 (4__30__1043)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	8192	hidden units	256
learning rate	5.0e-4	learning rate schedule	constant
max steps	5.0e6	normalize	false
num epoch	5	num layers	2
summary freq	1000	time horizon	512
extrinsic strength	1.0	extrinsic gamma	0.8
curiosity strength	0.01	curiosity gamma	0.9
curiosity encoding size	128		

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 25
Action space size = 2

Episode steps = 1000
Total steps = 317000
Time = 7440s

Training with curriculum. It has the parameter "requirement", which makes that when the reward reaches 0.5, 1 is subtracted from future rewards so that it must adjust the graph more. It has several levels until subtracting 25 points (it goes from 1 to 1).

There are several levels of the curriculum that go up too fast, they should be compressed into less. On the other hand, I have found that it is necessary to use a weighted average to make it react correctly to impulses.

30/4 13:16 (4_30_1316)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	8192	hidden units	256
learning rate	5.0e-4	learning rate schedule	constant
max steps	5.0e6	normalize	false
num epoch	5	num layers	2
summary freq	1000	time horizon	512
extrinsic strength	1.0	extrinsic gamma	0.8
curiosity strength	0.01	curiosity gamma	0.9
curiosity encoding size	128		

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 25

Action space size = 2

Episode steps = 1000
Total steps = 265000
Time = 6215s

Same training but with several errors corrected (such as the relationship between stds or some score limits), in addition to having added a weighted mean and a minimum standard deviation. The learning curriculum levels should be adjusted later.

The network stagnates over 0, it does not get past the first level. It should learn to adjust the lines.

30/4 15:12 (4_30_1512)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	8192	**epsilon**	**0.3**
hidden units	256	**learning rate**	**1.0e-3**
learning rate schedule	constant	max steps	5.0e6
normalize	false	num epoch	5
num layers	**1**	summary freq	1000
time horizon	**5**		
extrinsic strength	1.0	extrinsic gamma	0.8
curiosity strength	0.01	curiosity gamma	0.9
curiosity encoding size	128		

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 25
Action space size = 2

Episode steps = 1000
Total steps = 128000
Time = 3050s

Tests with different parameters. It learns faster but it gets stuck again. This seems to have its cause in the fact that if the agent estimates a very large range in relation to the deviation of the bot even though the reward is decreased, the punishment is also decreased. To adjust this phenomenon I thought to change the deviation ratio parameter to another factor that will also depend on coherence (using clipped cubic functions).

It may be necessary to combine learning with behavioral cloning at first.

30/4 17:14 (4_30_1714)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	8192	epsilon	0.3
hidden units	256	learning rate	1.0e-3
learning rate schedule	constant	max steps	5.0e6
normalize	false	num epoch	5
num layers	1	summary freq	1000
time horizon	5		
extrinsic strength	1.0	extrinsic gamma	0.8
curiosity strength	0.01	curiosity gamma	0.9
curiosity encoding size	128		

Render Target Sensor: 64x64

Grayscale: false

Observation space size = 25

Action space size = 2

Episode steps = 1000

Total steps = 225000

Time = 5500s

Corrected the deviation ratio factor, now penalizes more or less depending on coherence (if by covering a large range fails, the punishment is increased, if it succeeds, the reward is decreased).

Apart from what was mentioned before of BC with GAIL, it would also be good to try a multiplicative (instead of subtractive) requirement once the rewards are balanced. In spite of lowering the condition to move up a level, it does not converge but stays where it was.

30/4 18:49 (4_30_1849)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	8192	epsilon	0.3
hidden units	256	**learning rate**	**2.0e-3**
learning rate schedule	constant	max steps	5.0e6
normalize	false	num epoch	5
num layers	1	summary freq	1000
time horizon	5		
extrinsic strength	1.0	extrinsic gamma	0.8
curiosity strength	0.01	curiosity gamma	0.9
curiosity encoding size	128		

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 25
Action space size = 2

Episode steps = 1000
Total steps = 225000
Time = 5500s

Same training as before but with a higher learning rate and greater initial demand.
Still not stabilized. It will be necessary to penalize the fact that the range is greater at the moment of stability (instead of decreasing reward).

01/5 10:05 (5_01_1005)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	8192	epsilon	0.3
hidden units	256	learning rate	2.0e-3
learning rate schedule	constant	max steps	5.0e6
normalize	false	num epoch	5
num layers	1	summary freq	1000
time horizon	5		
extrinsic strength	1.0	extrinsic gamma	0.8
curiosity strength	0.01	curiosity gamma	0.9
curiosity encoding size	128		

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 25
Action space size = 2

Episode steps = 1000
Total steps = 90000
Time = 2100s

Training with an initial demand of 15. Compare with the previous ones to see where it stagnates.

Curiously, it is still stagnating at 0, but this one seems to be behaving a little better. Now I will test a training with much more demand, then the multiplicative demand and finally rewards that only penalize for moves being consistent and only reward for moves being inconsistent (more or less).

01/5 11:10 (5_01_1110)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	8192	epsilon	0.3
hidden units	256	learning rate	2.0e-3
learning rate schedule	constant	max steps	5.0e6
normalize	false	num epoch	5
num layers	1	summary freq	1000
time horizon	5		
extrinsic strength	1.0	extrinsic gamma	0.8
curiosity strength	0.01	curiosity gamma	0.9
curiosity encoding size	128		

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 25
Action space size = 2

Episode steps = 1000
Total steps = 75000
Time = 1750s

Same training but with 50 demands. It may come closer to the target or divert it by being excessively negative. The graph converges to -50 so that the subtractive requirement has proven to be ineffective.

01/5 12:45 (5_01_1245)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	8192	epsilon	0.3
hidden units	256	learning rate	2.0e-3
learning rate schedule	constant	max steps	5.0e6
normalize	false	num epoch	5
num layers	1	summary freq	1000
time horizon	5		
extrinsic strength	1.0	extrinsic gamma	0.8
curiosity strength	0.01	curiosity gamma	0.9
curiosity encoding size	128		

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 25
Action space size = 2

Episode steps = 1000
Total steps = 420000
Time = 10700s

Simpler training without a curriculum: if the movement is coherent it can only be penalized, if it is incoherent it can only be rewarded. At the moment I use linear formulas.

As the rewards are now, it slowly goes up to -7 and at 360000 steps it diverges. Halfway through it seemed to fit the coherent movement, but it doesn't capture the impulses well (it doesn't get enough reward).

01/5 15:52 (5_01_1552)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	8192	epsilon	0.3
hidden units	256	learning rate	2.0e-3
learning rate schedule	constant	max steps	5.0e6
normalize	false	num epoch	5
num layers	1	summary freq	1000
time horizon	5		
extrinsic strength	1.0	extrinsic gamma	0.8
curiosity strength	**0.1**	**curiosity gamma**	**0.8**
curiosity encoding size	128		

Render Target Sensor: 64x64

Grayscale: false
Observation space size = 25
Action space size = 2

Episode steps = 1000
Total steps = 390000
Time = 10000s

Inconsistent movement reward improved by 10, and increased curiosity.

This time it has not collapsed, and although it adapts quite well to coherent movement it does not carry impulses very well (maybe because it is not penalized at all, although sometimes it "feigns" to follow them).

01/5 18:51 (5_01_1851)

Trainer: PPO

batch size	1024	**beta**	**8.0e-3**
buffer size	8192	epsilon	0.3
hidden units	256	learning rate	2.0e-3
learning rate schedule	**linear**	**max steps**	**6.0e5**
normalize	false	num epoch	5
num layers	1	summary freq	1000
time horizon	5		
extrinsic strength	1.0	extrinsic gamma	0.8
curiosity strength	0.1	curiosity gamma	0.8
curiosity encoding size	128		

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 25
Action space size = 2

Episode steps = 1000
Total steps = 600000
Time = 15800s

Added small negative rewards when an impulse fails (less than for coherent movements). In addition, training has been limited to 600,000 steps with a linear learning rate.

The movement seems correct on the graph, although it would be nice to decrease the amount of noise. However, it is enough to use GAIL with correct rewards.

*Thinking about it, it would also be good to treat cases where the standard deviation is 0, since a model like this where the deviation has a certain minimum would not be

good for behaviours where the bot can stand completely still. FURTHER, it would be convenient if the deviation could be decreased once it has reacted, to make the transitions between movements more fluid (having a wide random range, the movement can be somewhat chaotic).

03/5 17:58 (5_03_1758)

Trainer: PPO

batch size	1024	beta	8.0e-3
buffer size	8192	epsilon	0.3
hidden units	256	learning rate	2.0e-3
learning rate schedule	linear	max steps	6.0e5
normalize	false	num epoch	5
num layers	1	summary freq	1000
time horizon	5		
extrinsic strength	1.0	extrinsic gamma	0.8
curiosity strength	0.1	curiosity gamma	0.8
curiosity encoding size	128		
gail strength	**0.01**	**gail gamma**	**0.99**
gail encoding size	**128**		

Demo path: demos/MyStdDevDemo.demo

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 25
Action space size = 2

Episode steps = 1000
Total steps = 516000
Time = 15200s

Added agent demonstration and GAIL reward. Pretraining parameter could be added later to check if it improves training. The result is very similar to the previous one, and that is why it is not considered as good.

03/5 22:23 (5_03_2223)

Trainer: PPO

batch size	1024	beta	8.0e-3
buffer size	8192	epsilon	0.3
hidden units	256	learning rate	2.0e-3
learning rate schedule	linear	max steps	6.0e5
normalize	false	num epoch	5
num layers	1	summary freq	1000
time horizon	5		
extrinsic strength	1.0	extrinsic gamma	0.8
curiosity strength	0.1	curiosity gamma	0.8
curiosity encoding size	128		
gail strength	**0.6**	gail gamma	0.99
gail encoding size	128		

Demo path: demos/MyStdDevDemo.demo

Render Target Sensor: 64x64

Grayscale: false

Observation space size = 25

Action space size = 2

Episode steps = 1000

Total steps = 280000

Time = 8000s

More weight to GAIL rewards. With 0.7 gamma, it diverts at 82k steps.

With the above parameters it has strange behaviors, like placing the 2 lines over 0. It may need pre-training or a larger dataset.

04/5 10:08 (5_04_1008)

Trainer: PPO

batch size	1024	beta	8.0e-3
buffer size	8192	epsilon	0.3
hidden units	256	learning rate	2.0e-3
learning rate schedule	linear	max steps	6.0e5
normalize	false	num epoch	5
num layers	1	summary freq	1000
time horizon	5		
pretraining strength	**0.5**	**pretraining steps**	**10000**
extrinsic strength	**0.7**	extrinsic gamma	0.8
curiosity strength	0.1	curiosity gamma	0.8
curiosity encoding size	128		
gail strength	0.6	gail gamma	0.99
gail encoding size	128		

Demo path: demos/MyStdDevDemo.demo

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 25
Action space size = 2

Episode steps = 1000
Total steps = 153000
Time = 4300s

At 40,000 steps it has received a great reward in curiosity that has caused the model to collapse, from there it has not recovered.

04/5 11:36 (5_04_1136)

Trainer: PPO

batch size	1024	beta	8.0e-3
buffer size	8192	epsilon	0.3
hidden units	256	learning rate	2.0e-3
learning rate schedule	linear	max steps	6.0e5
normalize	false	num epoch	5
num layers	1	summary freq	1000
time horizon	5		
pretraining strength	0.5	**pretraining steps**	**20000**
extrinsic strength	0.7	**extrinsic gamma**	**0.9**
gail strength	**0.5**	gail gamma	0.99
gail encoding size	**256**		

Demo path: demos/MyStdDevDemo.demo

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 25
Action space size = 2

Episode steps = 1000
Total steps = 600000
Time = 16200s

Curiosity eliminated, changes in other parameters and more pretraining. Doesn't seem to get as close to the target as it used to, it stagnates earlier. For the next training, base rewards on gail only.

04/5 16:12 (5_04_1612)

Trainer: PPO

batch size	1024	beta	8.0e-3
buffer size	8192	epsilon	0.3
hidden units	256	**learning rate**	**5.0e-4**
learning rate schedule	linear	max steps	6.0e5
normalize	false	num epoch	5
num layers	1	summary freq	1000
time horizon	**64**		
pretraining strength	0.5	pretraining steps	20000
gail strength	**1.0**	gail gamma	0.99
gail encoding size	**128**		

Demo path: demos/MyStdDevDemo.demo

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 25
Action space size = 2

Episode steps = 1000
Total steps = 393000
Time = 9400s

Rewards based on GAIL only. The learning rate has been lowered to stabilize learning.

No improvement at all. Maybe the problem is that the dataset is small.

05/5 10:58 (5_05_1058)

Trainer: PPO

batch size	1024	beta	8.0e-3
buffer size	8192	epsilon	0.3
hidden units	256	learning rate	5.0e-4
learning rate schedule	linear	max steps	6.0e5
normalize	false	num epoch	5
num layers	1	summary freq	1000
time horizon	64		
pretraining strength	0.5	**pretraining steps**	**40000**
gail strength	1.0	gail gamma	0.99
gail encoding size	128		

Demo path: demos/LongStdDDemo.demo

Render Target Sensor: 64x64

Grayscale: false

Observation space size = 25

Action space size = 2

Episode steps = 1000

Total steps = 60000

Time = 1500s

Found a mistake that made the demos all wrong, the previous models do not serve as an example. Retraining with parameters similar to 5_04_1612 and a longer and more correct demo. In this one only GAIL is used to see the result.

At the beginning it has had a very high rise, but it has stagnated afterwards. As the adversary network was not going to train anymore, I don't expect it to go up much. Set pretraining steps to 0 to keep the opponent net training. Maybe it can be turned off at the end. Also increase the strength.

05/5 11:58 (5_05_1158)

Trainer: PPO

batch size	1024	beta	8.0e-3
buffer size	8192	epsilon	0.3
hidden units	256	learning rate	5.0e-4
learning rate schedule	linear	max steps	6.0e5
normalize	false	num epoch	5
num layers	1	summary freq	1000
time horizon	64		
pretraining strength	**0.7**	**pretraining steps**	**600000**
gail strength	1.0	gail gamma	0.99
gail encoding size	128		

Demo path: demos/LongStdDDemo.demo

Render Target Sensor: 64x64

Grayscale: false

Observation space size = 25

Action space size = 2

Episode steps = 1000

Total steps = 110000

Time = 2700s

The reward goes up but ends up swinging in negative values. It may be necessary to increase the learning rate, increase entropy regulation or add environmental rewards. It seems that the network manages to fool the opponent's network faster than it learns.

05/5 12:24 (5_05_1224)

Trainer: PPO

batch size	1024	beta	8.0e-3
buffer size	8192	epsilon	0.3
hidden units	256	**learning rate**	**2.0e-3**
learning rate schedule	linear	max steps	6.0e5
normalize	false	num epoch	5
num layers	1	summary freq	1000
time horizon	64		
pretraining strength	**0.8**	pretraining steps	600000
extrinsic strength	**1.0**	**extrinsic gamma**	**0.99**
curiosity strength	**0.1**	**curiosity gamma**	**0.99**
curiosity encoding size	**128**		
gail strength	1.0	gail gamma	0.99
gail encoding size	128		

Demo path: demos/LongStdDDemo.demo

Render Target Sensor: 64x64

Grayscale: false

Observation space size = 25

Action space size = 2

Episode steps = 1000

Total steps = 50000

Time = 1300s

Standard rewards restored. Imitation has been given more strength but has less influence on the model's reward.

I have interrupted the model to train a reference one, with the values of the one that was successful but the new rewards to see if it is still correct. The model has managed to raise the rewards several steps, although I should analyze issues such as entropy or losses to adjust more parameters. The goal is to get a faster workout than the benchmark, so that it can be scaled.

05/5 13:00 (5_05_1300)

Trainer: PPO

batch size	1024	beta	8.0e-3
buffer size	8192	epsilon	0.3
hidden units	256	learning rate	2.0e-3
learning rate schedule	linear	max steps	6.0e5
normalize	false	num epoch	5
num layers	1	summary freq	1000
time horizon	64		
extrinsic strength	1.0	**extrinsic gamma**	**0.8**
curiosity strength	0.1	**curiosity gamma**	**0.8**
curiosity encoding size	128		

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 25
Action space size = 2

Episode steps = 1000
Total steps = 600000
Time = 14100s

Training with the previous values (except time horizon) of the one that came closest to success. Quite correct model, serves as a reference for rewards (training time should be improved). However, it doesn't handle impulse linking or the ones on the right side very well. From the demo a reward of 0.25 can be expected as a maximum optimistic (this would be quite correct behaviour).

05/5 17:15 (5_05_1715)

Trainer: PPO

batch size	1024	beta	8.0e-3
buffer size	8192	epsilon	0.3
hidden units	256	learning rate	2.0e-3
learning rate schedule	linear	max steps	6.0e5
normalize	false	num epoch	5
num layers	1	summary freq	1000
time horizon	64		
pretraining strength	**0.5**	**pretraining steps**	**600000**
extrinsic strength	1.0	extrinsic gamma	0.8
curiosity strength	0.1	curiosity gamma	0.8
curiosity encoding size	128		
gail strength	**0.02**	**gail gamma**	**0.99**
gail encoding size	**256**		

Demo path: demos/LongStdDDemo.demo

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 25
Action space size = 2

Episode steps = 1000
Total steps = 77000
Time = 2000s

Test with the 3 parameters (inspired by Pyramids), following the indications of the documentation. The aim is to achieve a faster or more rewarding workout than the reference one (5_05_1300).

It has started well but ends up being between -60 and -80. In the following I will add other optional parameters to see if the learning is stabilized.

05/5 17:52 (5_05_1752)

Trainer: PPO

batch size	1024	beta	8.0e-3
buffer size	8192	epsilon	0.3
hidden units	256	learning rate	2.0e-3
learning rate schedule	linear	max steps	6.0e5
normalize	false	num epoch	5
num layers	1	summary freq	1000
time horizon	64		
pretraining strength	0.5	pretraining steps	600000
extrinsic strength	1.0	extrinsic gamma	0.8
curiosity strength	0.1	curiosity gamma	0.8
curiosity encoding size	128		
gail strength	0.02	gail gamma	0.99
gail encoding size	256	**gail use vail**	**true**

Demo path: demos/LongStdDDemo.demo

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 25
Action space size = 2

Episode steps = 1000
Total steps = 88000
Time = 2300s

Vail has been added to the training (bottleneck). It should be more stable, but, decrease curiosity, add use actions (to imitate, in theory) and/or add learning rate of GAIL.

Very similar result.

05/5 18:34 (5_05_1834)

Trainer: PPO

batch size	1024	beta	8.0e-3
buffer size	8192	epsilon	0.3
hidden units	256	learning rate	2.0e-3
learning rate schedule	linear	max steps	6.0e5
normalize	false	num epoch	5
num layers	1	summary freq	1000
time horizon	64		
pretraining strength	0.5	pretraining steps	600000
extrinsic strength	1.0	extrinsic gamma	0.8
curiosity strength	0.1	curiosity gamma	0.8
curiosity encoding size	128		
gail strength	0.02	gail gamma	0.99
gail encoding size	256	**gail use vail**	**false**
gail use actions	**true**		

Demo path: demos/LongStdDDemo.demo

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 25
Action space size = 2

Episode steps = 1000
Total steps = 110000
Time = 2900s

Added use actions to true, and removed the use vail.

It oscillates on the same values, although a little higher, but still not improving on the original.

05/5 19:25 (5_05_1925)

Trainer: PPO

batch size	1024	beta	8.0e-3
buffer size	8192	epsilon	0.3
hidden units	256	learning rate	2.0e-3
learning rate schedule	linear	max steps	6.0e5
normalize	false	num epoch	5
num layers	1	summary freq	1000
time horizon	64		
pretraining strength	0.5	pretraining steps	600000
extrinsic strength	1.0	extrinsic gamma	0.8
curiosity strength	0.1	curiosity gamma	0.8
curiosity encoding size	128		
gail strength	**0.01**	**gail gamma**	**0.95**
gail learning rate	**0.00005**	**gail encoding size**	**64**
gail use vail	**true**	gail use actions	true

Demo path: demos/LongStdDDemo.demo

Render Target Sensor: 64x64

Grayscale: false

Observation space size = 25

Action space size = 2

Episode steps = 1000

Total steps = 98000

Time = 2500s

Changes in a few variables, and added learning rate. It shows the same trend.

05/5 20:17 (5_05_2017)

Trainer: PPO

batch size	1024	beta	8.0e-3
buffer size	8192	epsilon	0.3
hidden units	256	learning rate	2.0e-3
learning rate schedule	linear	max steps	6.0e5
normalize	false	num epoch	5
num layers	1	summary freq	1000
time horizon	64		
extrinsic strength	1.0	extrinsic gamma	0.8
curiosity strength	0.1	curiosity gamma	0.8
curiosity encoding size	128		
gail strength	0.01	gail gamma	0.95
gail learning rate	**0.0005**	gail encoding size	64
gail use vail	true	gail use actions	true

Demo path: demos/LongStdDDemo.demo

Render Target Sensor: 64x64

Grayscale: false

Observation space size = 25

Action space size = 2

Episode steps = 1000

Total steps = 600000

Time = 15200s

Pretraining removed. The result was practically the same as in the reference, and has reached the same reward. However, it does not seem that the GAIL rewards have influenced anything, maybe the curiosity is too high, besides, it has taken 20 minutes more to do the same steps. It might be worth trying out new rewards.

06/5 10:36 (5_06_1036)

Trainer: PPO

batch size	1024	beta	8.0e-3
buffer size	8192	epsilon	0.3
hidden units	256	learning rate	2.0e-3
learning rate schedule	linear	max steps	6.0e5
normalize	false	num epoch	5
num layers	1	summary freq	1000
time horizon	64		
extrinsic strength	1.0	**extrinsic gamma**	**0.9**
curiosity strength	**0.05**	**curiosity gamma**	**0.9**
curiosity encoding size	128		

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 25
Action space size = 2

Episode steps = 1000
Total steps = 527000
Time = 12500s

Benchmark training with the new linear rewards. A jump may have to be added to the penalties depending on the outcome.

The reward goes up in parallel to the previous reference, but it doesn't take much to detect impulses. It will be necessary to add a jump to the penalties.

06/5 14:15 (5_06_1415)

Trainer: PPO

batch size	1024	beta	8.0e-3
buffer size	8192	epsilon	0.3
hidden units	256	learning rate	2.0e-3
learning rate schedule	linear	max steps	6.0e5
normalize	false	num epoch	5
num layers	1	summary freq	1000
time horizon	64		
extrinsic strength	1.0	extrinsic gamma	0.9
curiosity strength	0.05	curiosity gamma	0.9
curiosity encoding size	128		

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 25

Action space size = 2

Episode steps = 1000
Total steps = 242000
Time = 5200s

Added a jump in the rewards (locking everything up penalizes the same as failing by 0.00001 units). It has a very similar trend, so it doesn't improve the previous one.

06/5 15:45-18:48 (5_06_1848)

Trainer: PPO

batch size	1024	beta	8.0e-3
buffer size	8192	epsilon	0.3
hidden units	256	learning rate	2.0e-3
learning rate schedule	linear	max steps	6.0e5
normalize	false	num epoch	5
num layers	1	summary freq	1000
time horizon	64		
extrinsic strength	1.0	extrinsic gamma	0.9
curiosity strength	0.05	curiosity gamma	0.9
curiosity encoding size	128		

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 25
Action space size = 2

Episode steps = 1000
Total steps = -
Time = -

Parameter tests. None of them work as well as they should. The next test with SAC.

07/5 09:56 (5_07_0956)

Trainer: SAC

batch size	128	buffer size	50000
buffer init steps	0	hidden units	128
init entcoef	1.0	learning rate	1.0e-3
learning rate schedule	constant	max steps	6.0e5
memory size	256	normalize	true
num update	1	train interval	5
num layers	2	time horizon	1024
sequence length	64	summary freq	1000
tau	0.01	use recurrent	false
vis encode type	simple		
pretraining strength	0.5	pretraining steps	10000
extrinsic strength	1.0	extrinsic gamma	0.99
curiosity strength	0.01	curiosity gamma	0.99
curiosity encoding size	128		
gail strength	0.01	gail gamma	0.99
gail encoding size	128		

Demo path: demos/LongStdDDemo.demo

Render Target Sensor: 64x64

Grayscale: false

Observation space size = 25

Action space size = 2

Episode steps = 1000

Total steps = 72000

Time = 4600s

Training with SAC. It is much slower than PPO but requires less samples, has risen fast but has diverged by 70k.

07/5 11:42 (5_07_1142)

Trainer: SAC

batch size	128	buffer size	50000
buffer init steps	0	hidden units	128
init entcoef	1.0	**learning rate**	**8.0e-4**
learning rate schedule	**linear**	max steps	6.0e5
memory size	256	normalize	true
num update	1	train interval	5
num layers	2	**time horizon**	**64**
sequence length	64	summary freq	1000
tau	**0.007**	use recurrent	false
vis encode type	simple		
pretraining strength	**0.7**	**pretraining steps**	**20000**
extrinsic strength	1.0	extrinsic gamma	0.99
curiosity strength	0.01	curiosity gamma	0.99
curiosity encoding size	128		
gail strength	0.01	gail gamma	0.99
gail encoding size	128	**use actions**	**true**

Demo path: demos/LongStdDDemo.demo

Render Target Sensor: 64x64

Grayscale: false

Observation space size = 25

Action space size = 2

Episode steps = 1000

Total steps = 194000

Time = 11900s

Back to previous rewards and changes in some parameters. The network adapts quite well to impulses although it has intermittent drops in rewards (smaller each time, it may be part of how the model works). Better results could be obtained with this model, but the training is much slower in proportion.

07/5 15:12 (5_07_1512)

Trainer: SAC

batch size	128	buffer size	50000
buffer init steps	**2000**	**hidden units**	**256**
init entcoef	**1.5**	**learning rate**	**1.0e-3**
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	64	summary freq	1000
tau	**0.005**	use recurrent	false
vis encode type	simple		
pretraining strength	0.7	pretraining steps	20000
extrinsic strength	**2.0**	extrinsic gamma	0.99
curiosity strength	**0.02**	curiosity gamma	0.99
curiosity encoding size	128		
gail strength	**0.02**	gail gamma	0.99
gail encoding size	128	use actions	true

Demo path: demos/LongStdDDemo.demo

Render Target Sensor: 64x64

Grayscale: false

Observation space size = 25

Action space size = 2

Episode steps = 1000

Total steps = -

Time = -

Changes in layer structure. Cancelled.

07/5 17:30 (5_07_1730)

Trainer: SAC

batch size	128	buffer size	50000
buffer init steps	2000	hidden units	256
init entcoef	1.5	learning rate	1.0e-3
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	64	summary freq	1000
tau	0.005	use recurrent	false
vis encode type	simple		
pretraining strength	0.7	pretraining steps	20000
extrinsic strength	2.0	extrinsic gamma	0.99
curiosity strength	0.02	curiosity gamma	0.99
curiosity encoding size	128		
gail strength	0.02	gail gamma	0.99
gail encoding size	128	use actions	true

Demo path: demos/LongStdDDemo.demo

Render Target Sensor: 64x64

Grayscale: false

Observation space size = 25

Action space size = 2

Episode steps = 1000

Total steps = 215000

Time = 4600s

GPU training tests, I don't know if it really works or not

-The time scale of the training cannot be changed.

-Adding twice as many environments causes twice as many steps to be taken, but the curve increases by half as fast (more or less).

In this case, the agent ends up farther away from the reward. The next, train with less learning rate.

07/5 20:39 (5__07__2039)

Trainer: SAC

batch size	128	**buffer size**	**200000**
buffer init steps	**5000**	hidden units	256
init entcoef	**0.7**	**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**	**true**
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

Demo path: demos/NoObsDemo.demo

Render Target Sensor: 64x64

Grayscale: false

Observation space size = 0

Action space size = 2

Episode steps = 1000

Total steps = 57000

Time = 3500s

The 25 previous observations have been changed to “use recurrent”. The CNN encoder has also been changed, so it may take a little longer to train (it shouldn’t be as relevant as before, when inferring by GPU). It seems that increasing the buffer causes it to go much slower: contrast with the final learning result (if it doesn’t improve, check the same without buffer, if it doesn’t change it will be that the 25 observations are essential). It has not learned anything.

07/5 22:09 (5_07_2209)

Trainer: 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

Demo path: demos/NoObsDemo.demo

Render Target Sensor: 64x64

Grayscale: false

Observation space size = 0

Action space size = 2

Episode steps = 1000

Total steps = 600000

Time = 14000s

Use recurrent is what causes training to slow down considerably.

Without memory parameters it still seems to adapt correctly, and has a speed similar to ppo. However, it performs a kind of “counterpulses”, which perhaps would be corrected with more training time.

The rewards of each standard deviation could be separated (by giving 1 of each + the composite), and if noise were added, the actual previous steps could also be used (in SAC they may work better).

09/5 12:22 (5_09_1222)

Trainer: 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

Demo path: demos/IndDemo.demo

Render Target Sensor: 64x64

Grayscale: false

Observation space size = 0

Action space size = 2

Episode steps = 1000

Total steps = 600000

Time = 14000s

Individual rewards have been added for each line (they are currently only linear, and with less weight). The demo is changed so that the rewards are consistent (maximum is 3). It is saved as a checkpoint for further training. It adapts surprisingly well to impulses, although I think there is still room for further noise reduction.

09/5 16:43 (5_09_1643)

Trainer: 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

Demo path: demos/IndDemo.demo

Render Target Sensor: 64x64
Grayscale: false
Observation space size = 0
Action space size = 2

Episode steps = 1000
Total steps = 600000
Time = 14000s + 7700s

Same model as the last one but woth more training steps. Max steps has been changed to leave it indefinitely. It has not had any major performance improvements.

Appendix D

Reaction time

This appendix contains some notes that were taken when (and after) training sessions related to modeling delayed reactive behaviors, or in other words, agents with reaction time.

Any parameter that does not appear in one training is set to default (see ./config/-trainer.config file).

Trained models with green titles are considered good or any improvement in the investigation. Models with red titles are considered failures.

Some of the models have not been saved, either because they don't perform well or they perform in much the same way as another model (agent).

Most of the notes should not be taken literally or as certain, as they usually are theories or preliminary conclusions drawn during the training itself.

09/5 18:55 (5_09_1855)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	8192	epsilon	0.2
hidden units	256	learning rate	2.0e-3
learning rate schedule	linear	max steps	6.0e5
normalize	true	num epoch	6
num layers	1	summary freq	1000
time horizon	64		
pretraining strength	0.8	pretraining steps	20000
extrinsic strength	1.0	extrinsic gamma	0.99
curiosity strength	0.01	curiosity gamma	0.99
curiosity encoding size	128		
gail strength	0.01	gail gamma	0.99
gail encoding size	128		

Demo path: demos/IndDemo.demo

Render Target Sensor: 64x64

Grayscale: false

Observation space size = 0

Action space size = 2

Episode steps = 1000

Total steps = 524000

Time = 5580s

Speed test using ppo with 3 envs and GPU, same rewards as in 5_09_1643

It works twice as fast, so it can be good for testing (and using SAC for final versions). However, it often gives errors like Out of memory (and other stranger ones that may be caused by overheating).

12/5 20:00 (5_12_2000)

Trainer: 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		
extrinsic strength	1.5	extrinsic gamma	0.99
curiosity strength	0.03	curiosity gamma	0.99
curiosity encoding size	128		

Frames [1]: 0, 5, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 37, 40, 45, 50, 55, 60, 70, 80

Render Target Sensor: 21 * (20x20)
Grayscale: true
Observation space size = 0
Action space size = 2

Episode steps = 1000
Total steps = 278000
Time = 17650s

Bot with 20 frames of memory (up to 1.4 seconds), and motion memory. No reactions are used, it serves to check efficiency with SAC and 3x envs. At the moment it gives negative dimension error when making convolution.

Using 2 images of 16x16 fails, but with 2 of 64x64 it works. This last one also goes with grayscale.

With 3 16x16 images grayscale doesn't work. 16 of 64x64 fails.

With 3 images 64x64 grayscale does work. With 4 images it also works.

20x20 seems to be the minimum (or close) to be able to convolute without errors. However, it is necessary that all images are the same (I could check this later, but it seems to be the case). It gives error later. THIS ERROR was due to the fact that the demo was made with 16x16 render targets, so I remove GAIL at this moment.

Personally, I don't like very much the "21x20x20" thing.

The training becomes about 3 times slower despite using grayscale, but the results are not bad. It might be better to use PPO in these test cases.

GAIL may be quite relevant, as it has improved the reward but has not adapted to the idle movement too closely.

[1] Frame 0 is the current frame, which is used in every training, the other frames are the n-th previous frame.

13/5 11:18 (5_13_1118)

Trainer: 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**	**gail use actions**	**true**

Demo path: demos/Frames20Demo.demo
Frames: 0, 5, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 40, 45, 50, 55, 60, 70, 80

Render Target Sensor: 20 * (20x20)
Grayscale: true
Observation space size = 0
Action space size = 2

Episode steps = 1000
Total steps = 254000
Time = 20750s

Removed the frame 37. Training with a bot with noise and delay (BotDelayNoised, before it was BotNoised). Now the impulses have a different shape.

The curve is similar to the previous training. GAIL makes it even slower (from 3h 20m to 4h 20m at 200k steps). The mean and deviations are better being more invariant, I should restore the previous settings (and improve the demo bot for this model).

13/5 18:19 (5_13_1819)

Trainer: 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	gail use actions	true

Demo path: demos/Frames20Demo.demo

Frames: 0, 5, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 40, 45, 50, 55, 60, 70, 80

Render Target Sensor: 20 * (20x20)

Grayscale: true

Observation space size = 0

Action space size = 2

Episode steps = 1000

Total steps = 308000

Time = 25000s

Improved the trajectory of the averages, also in the demo.

The training is slow but with better proportion than the previous one (probably because the demo occupies less space). It manages to learn certain things but it is necessary to refine more how it adapts to the movements (both impulses and coherent).

14/5 09:52 (5_14_0952)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	8192	epsilon	0.2
hidden units	256	learning rate	2.0e-3
learning rate schedule	linear	max steps	6.0e5
normalize	true	num epoch	6
num layers	1	summary freq	1000
time horizon	64		
pretraining strength	**0.5**	pretraining steps	20000
extrinsic strength	1.0	extrinsic gamma	0.99
curiosity strength	0.01	curiosity gamma	0.99
curiosity encoding size	128		
gail strength	0.01	gail gamma	0.99
gail encoding size	128		

Demo path: demos/Frames20Demo.demo

Frames: 0, 5, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 40, 45, 50, 55, 60, 70, 80

Render Target Sensor: 20 * (20x20)

Grayscale: true

Observation space size = 0

Action space size = 2

Episode steps = 1000

Total steps = 600000

Time = 18450s

It greatly improves the reward curve, perhaps because of the simplification of the frames (individually). Adapts to various reactions at the right time but has too much noise. It may be better to change the reward system by adapting it to the time dimension and have the agent take only one action again (and not discount it by how far it is, but by the average). It also greatly improves the SAC times, although the latter could have better results in the very long term. It is considered as good but the method will be changed. On the other hand, it should be noted that this model does not handle well the targets that appear on the right side.

After changing back to the previous 1 action method, train the model with pure behavioral cloning.

14/5 20:10 (5_14_2010)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	8192	epsilon	0.2
hidden units	256	learning rate	2.0e-3
learning rate schedule	linear	max steps	6.0e5
normalize	true	num epoch	6
num layers	1	summary freq	1000
time horizon	64		
pretraining strength	0.5	pretraining steps	20000
extrinsic strength	1.0	extrinsic gamma	0.99
curiosity strength	0.01	curiosity gamma	0.99
curiosity encoding size	128		
gail strength	0.01	gail gamma	0.99
gail encoding size	128		

Demo path: demos/OneLineMem.demo
Frames: 0, 5, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 40, 45, 50, 55, 60, 70, 80

Render Target Sensor: 20 * (20x20)
Grayscale: true
Observation space size = 25
Action space size = 1

Episode steps = 1000
Total steps = 378000
Time = 12000s

Simple training based on tolerable range of 0.05.

As expected, it works the same as before: it adapts to the average but does not take impulses. The reward should be modified to accommodate the time difference (and the parameter of future rewards).

15/5 10:19 (5_15_1019)

Trainer: BC

batches per epoch	10	batch size	64
hidden units	128	learning rate	3.0e-4
max steps	5.0e4	memory size	256
num layers	2	sequence length	32
summary freq	1000	use recurrent	false

Demo path: demos/BCDemo.demo
Frames: 0, 5, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 40, 45, 50, 55, 60, 70, 80

Render Target Sensor: 20 * (20x20)
Grayscale: true
Observation space size = 25
Action space size = 1

Episode steps = 1000
Total steps = 9000
Time = 3200s

Behavioral cloning training. It begins with very good rewards but the model becomes unbalanced after a few iterations. Well trained one could say that it needs very few steps, but it is very slow in number of steps per minute.

15/5 11:08 (5_15_1108)

Trainer: BC

batches per epoch	10	batch size	64
hidden units	**256**	**learning rate**	**5.0e-5**
max steps	5.0e4	memory size	256
num layers	**1**	sequence length	32
summary freq	1000	use recurrent	false

Demo path: demos/BCDemo.demo
Frames: 0, 5, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 40, 45, 50, 55, 60, 70, 80

Render Target Sensor: 20 * (20x20)
Grayscale: true
Observation space size = 25
Action space size = 1

Episode steps = 1000
Total steps = 16000
Time = 5000s

Changes in architecture and learning rate. The reward decreases but more slowly. Next, train at a lower learning rate and with the previous layers.

15/5 12:37 (5_15_1237)

Trainer: BC

batches per epoch	10	batch size	64
hidden units	**128**	**learning rate**	**1.0e-5**
max steps	5.0e4	memory size	256
num layers	**2**	sequence length	32
summary freq	1000	use recurrent	false

Demo path: demos/BCDemo.demo
Frames: 0, 5, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 40, 45, 50, 55, 60, 70, 80

Render Target Sensor: 20 * (20x20)
Grayscale: true
Observation space size = 25
Action space size = 1

Episode steps = 1000
Total steps = 30000
Time = 10500s

Training with even less learning rate. I have noticed that in the beginning he does not do impulses anymore, it would be good to record some of the previous models at the beginning of the training to illustrate the memory (when he takes impulses).

It is considered good but still fails on lateral impulses.

15/5 17:03 (5_15_1703)

Trainer: BC

batches per epoch	10	batch size	64
hidden units	128	learning rate	1.0e-5
max steps	5.0e4	memory size	256
num layers	2	sequence length	32
summary freq	1000	use recurrent	false

Demo path: demos/BCDemo.demo
Frames: 0, 5, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 40, 45, 50, 55, 60, 70, 80

Render Target Sensor: 20 * (20x20)
Grayscale: true
Observation space size = 25
Action space size = 1

Episode steps = 1000
Total steps = 25000
Time = 8700s

Training with Online BC. The results are slightly better, since they do not depend on an external dataset but on live action. Training times are equivalent. It still takes a long time to learn the impulses on the right side.

15/5 19:32 (5_15_1932)

Trainer: BC

batches per epoch	10	batch size	64
hidden units	128	learning rate	1.0e-5
max steps	5.0e4	memory size	256
num layers	2	sequence length	32
summary freq	1000	use recurrent	false

Demo path: demos/BCDemo.demo

Frames: 0, 5, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 40, 45, 50, 55, 60, 70, 80

Render Target Sensor: 20 * (20x20)

Grayscale: true

Observation space size = 25

Action space size = 1

Episode steps = 1000

Total steps = 36000

Time = 12000s

Online training with the learning rate of 5_15_1237. It follows the same curve and ends up being chaotic.

21/5 10:08 (5_21_1008)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	8192	epsilon	0.2
hidden units	256	learning rate	2.0e-3
learning rate schedule	linear	max steps	6.0e5
normalize	true	num epoch	6
num layers	1	summary freq	1000
time horizon	64		
pretraining strength	0.5	pretraining steps	20000
extrinsic strength	1.0	extrinsic gamma	0.99
curiosity strength	0.01	curiosity gamma	0.99
curiosity encoding size	128		
gail strength	0.01	gail gamma	0.99
gail encoding size	128		

Demo path: demos/QueueDemo.demo

Frames: 0, 5, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 40, 45, 50, 55, 60, 70, 80

Render Target Sensor: 20 * (20x20)

Grayscale: true

Observation space size = 25

Action space size = 1

Episode steps = 1000

Total steps = 180000

Time = 5900s

Training with a movement queue. It is important that the parameter of future rewards (gamma) is high. Training may be flawed by how the average and deviation is calculated right now.

Goes up slowly. It has failed at 180000 steps.

21/5 12:42 (5_21_1242)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	8192	epsilon	0.2
hidden units	256	learning rate	2.0e-3
learning rate schedule	linear	max steps	6.0e5
normalize	true	num epoch	6
num layers	1	summary freq	1000
time horizon	64		
pretraining strength	0.5	pretraining steps	20000
extrinsic strength	1.0	extrinsic gamma	0.99
curiosity strength	0.01	curiosity gamma	0.99
curiosity encoding size	128		
gail strength	0.01	gail gamma	0.99
gail encoding size	128		

Demo path: demos/QueueDemo.demo

Frames: 0, 5, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 40, 45, 50, 55, 60, 70, 80

Render Target Sensor: 20 * (20x20)

Grayscale: true

Observation space size = 25

Action space size = 1

Episode steps = 1000

Total steps = 493000

Time = 15700s

Changes: normal movement does not score, any failed or overdone impulses discount, and successful impulses add up (more than failures).

Similar to previous models trained with PPO, it tends to average out but does not learn the impulses.

The score could be improved if impulses could be treated separately (or movements in general).

21/5 17:14 (5_21_1714)

Trainer: 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	6.0e6
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	gail use actions	true

Demo path: demos/QueueDemo.demo

Frames: 0, 5, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 40, 45, 50, 55, 60, 70, 80

Render Target Sensor: 20 * (20x20)

Grayscale: true

Observation space size = 25

Action space size = 1

Episode steps = 1000

Total steps = 254000

Time = 21000s

Training with SAC, in the same build as the previous one. Factors may need to be increased as it rarely receives rewards.

The graph converges quite well to the average, retaining some noise, but not reacting to any impulse. The reward is -0.34. Curiously, almost all of the score increase has been between 190-200k.

Penalties need to be given much more weight, and perhaps even more weight to the extra impulses from the net.

22/5 10:45 (5_22_1045)

Trainer: 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	6.0e6
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	gail use actions	true

Demo path: demos/QueueDemo.demo

Frames: 0, 5, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 40, 45, 50, 55, 60, 70, 80

Render Target Sensor: 20 * (20x20)

Grayscale: true

Observation space size = 25

Action space size = 1

Episode steps = 1000

Total steps = 255000

Time = 20750s

Rewards multiplied.

The rewards follow strange patterns, and the movement forms a kind of frequency. I think what it does is it accumulates little impulses to add up to a real one. I should better redefine how to score the impulses and what is considered impulse or not (and the standard deviation should be increased or made fixed).

In the end it stabilizes at the average, like most previous models.

23/5 10:36 (5_23_1036)

Trainer: 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	6.0e6
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	gail use actions	true

Demo path: demos/QueueDemo.demo

Frames: 0, 5, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 40, 45, 50, 55, 60, 70, 80

Render Target Sensor: 20 * (20x20)

Grayscale: true

Observation space size = 25

Action space size = 1

Episode steps = 1000

Total steps = 320000

Time = 27500s

Changes in the reward, now eliminates whole impulses if it fails at first. It doesn't react to impulses after many hours. He may not have enough incentive to try.

The factors are: 1500 reward and 750 penalty (in general).

23/5 18:25 (5_23_1825)

Trainer: 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	6.0e6
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	gail use actions	true

Demo path: demos/QueueDemo.demo

Frames: 0, 5, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 40, 45, 50, 55, 60, 70, 80

Render Target Sensor: 20 * (20x20)

Grayscale: true

Observation space size = 25

Action space size = 1

Episode steps = 1000

Total steps = 203000

Time = 17000s

Reward=2000, fail=-500, miss=-1500. It has been corrected that it penalizes the final part of an impulse (if it succeeds, it does not continue to penalize).

It still doesn't have enough incentive to make impulses. Next time try with PPO and new parameters, and maybe also improve the reward system.

24/5 16:25 (5_24_1625)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	8192	epsilon	0.2
hidden units	256	learning rate	2.0e-3
learning rate schedule	linear	max steps	6.0e5
normalize	true	num epoch	6
num layers	1	summary freq	1000
time horizon	64		
pretraining strength	0.5	pretraining steps	20000
extrinsic strength	1.0	extrinsic gamma	0.99
curiosity strength	0.01	curiosity gamma	0.99
curiosity encoding size	128		
gail strength	0.01	gail gamma	0.99
gail encoding size	128		

Demo path: demos/QueueDemo.demo

Frames: 0, 5, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 40, 45, 50, 55, 60, 70, 80

Render Target Sensor: 20 * (20x20)

Grayscale: true

Observation space size = 25

Action space size = 1

Episode steps = 1000

Total steps = 260000

Time = 8300s

PPO training. Reward=500, fail=-150, miss=-1500. Higher learning rate.

24/5 18:50 (5_24_1850)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	8192	epsilon	0.2
hidden units	256	learning rate	2.0e-3
learning rate schedule	linear	max steps	6.0e5
normalize	true	num epoch	6
num layers	1	summary freq	1000
time horizon	64		
pretraining strength	0.5	pretraining steps	20000
extrinsic strength	1.0	extrinsic gamma	0.99
curiosity strength	0.01	curiosity gamma	0.99
curiosity encoding size	128		
gail strength	0.01	gail gamma	0.99
gail encoding size	128		

Demo path: demos/QueueDemo.demo
Frames: 0, 5, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 40, 45, 50, 55, 60, 70, 80

Render Target Sensor: 20 * (20x20)
Grayscale: true
Observation space size = 25
Action space size = 1

Episode steps = 1000
Total steps = 323000
Time = 10750s

PPO training. Reward=750, fail=-250, miss=-1500. It does not perform impulses.

24/5 21:54 (5_24_2154)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	8192	epsilon	0.2
hidden units	256	learning rate	2.0e-3
learning rate schedule	linear	max steps	6.0e5
normalize	true	num epoch	6
num layers	1	summary freq	1000
time horizon	64		
extrinsic strength	1.0	extrinsic gamma	0.99
curiosity strength	0.01	curiosity gamma	0.99
curiosity encoding size	128		

Frames: 0, 5, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 40, 45, 50, 55, 60, 70, 80

Render Target Sensor: 20 * (20x20)
Grayscale: true
Observation space size = 25
Action space size = 1

Episode steps = 1000
Total steps = 213000
Time = 5800s

PPO training. Reward=250, fail=-250, miss=-5000. Penalty for not performing impulses taken to the extreme. The program failed but the behavior didn't improve either.

25/5 16:07 (5_25_1607)

Trainer: BC

batches per epoch	10	batch size	64
hidden units	128	learning rate	5.0e-5
max steps	5.0e4	memory size	256
num layers	2	sequence length	32
summary freq	1000	use recurrent	false

Demo path: demos/BCDemo.demo
Frames: 0, 5, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 40, 45, 50, 55, 60, 70, 80

Render Target Sensor: 20 * (20x20)
Grayscale: true
Observation space size = 25
Action space size = 1

Episode steps = 1000
Total steps = 13000
Time = 4500s

Online training. A deterministic bot is used without any noise (constant movement at 0.3 and impulses at 0.2 seconds). It gets worse with time, although it doesn't start badly (if it catches just one impulse).

25/5 17:30 (5_25_1730)

Trainer: BC

batches per epoch	10	batch size	64
hidden units	128	**learning rate**	**1.0e-5**
max steps	5.0e4	memory size	256
num layers	2	sequence length	32
summary freq	1000	use recurrent	false

Demo path: demos/UniformDemo.demo
Frames: 0, 5, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 40, 45, 50, 55, 60, 70, 80

Render Target Sensor: 20 * (20x20)
Grayscale: true
Observation space size = 25
Action space size = 1

Episode steps = 1000
Total steps = 27000
Time = 8000s

Changed the demo and the learning rate.
The reward decreases constantly, and the behavior gets worse, although in some specific moments it gave good results.

25/5 19:58 (5_25_1958)

Trainer: BC

batches per epoch	10	batch size	64
hidden units	128	learning rate	1.0e-5
max steps	5.0e4	memory size	256
num layers	2	sequence length	32
summary freq	1000	use recurrent	false

Demo path: demos/UniformDemo.demo
Frames: 0, 5, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 40, 45, 50, 55, 60, 70, 80

Render Target Sensor: 20 * (20x20)
Grayscale: true
Observation space size = 25
Action space size = 1

Episode steps = 1000
Total steps = 15000
Time = 4500s

Now checks if there's a non-null target. There's been a bot failure, it's repeated.

25/5 21:57 (5_25_2157)

Trainer: BC

batches per epoch	10	batch size	64
hidden units	128	learning rate	1.0e-5
max steps	5.0e4	memory size	256
num layers	2	sequence length	32
summary freq	1000	use recurrent	false

Demo path: demos/UniformDemo.demo
Frames: 0, 5, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 40, 45, 50, 55, 60, 70, 80

Render Target Sensor: 20 * (20x20)
Grayscale: true
Observation space size = 25
Action space size = 1

Episode steps = 1000
Total steps = 46000
Time = 13300s

Several corrections to the bot's behavior (at some point it stopped reacting to targets, because the closest variable didn't restart).

It gets worse over time, making very strange effects after pulling several impulses. It is saved to test it in execution.

It does a good continuous movement, but when there's a target between the previous frames it oscillates in strange ways, although it keeps moving in the direction it should (as a whole). PPO, or more complex layers with imitation learning, should be tried.

26/5 10:59 (5_26_1059)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	8192	epsilon	0.2
hidden units	256	learning rate	2.0e-3
learning rate schedule	linear	max steps	6.0e5
normalize	true	num epoch	6
num layers	1	summary freq	1000
time horizon	64		
extrinsic strength	1.0	extrinsic gamma	0.99
curiosity strength	0.01	curiosity gamma	0.99
curiosity encoding size	128		

Frames: 0, 5, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 40, 45, 50, 55, 60, 70, 80

Render Target Sensor: 20 * (20x20)
Grayscale: true
Observation space size = 25
Action space size = 1

Episode steps = 1000
Total steps = 122000
Time = 4600s

Training with a single PPO thread (not multi) It should get better results since the movement is deterministic. The real bot does strange behaviors so it is stopped.

26/5 12:25 (5_26_1225)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	8192	epsilon	0.2
hidden units	256	learning rate	2.0e-3
learning rate schedule	linear	max steps	6.0e5
normalize	true	num epoch	6
num layers	1	summary freq	1000
time horizon	64		
extrinsic strength	1.0	extrinsic gamma	0.99
curiosity strength	0.01	curiosity gamma	0.99
curiosity encoding size	128		

Frames: 0, 5, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 40, 45, 50, 55, 60, 70, 80

Render Target Sensor: 20 * (20x20)
Grayscale: true
Observation space size = 25
Action space size = 1

Episode steps = 1000
Total steps = 200000
Time = 7750s

By setting the distance of closest to 2.0f, it reacted earlier (corrected). It does not seem to converge with PPO.

26/5 15:07 (5_26_1507)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	8192	epsilon	0.2
hidden units	256	learning rate	2.0e-3
learning rate schedule	linear	max steps	6.0e5
normalize	true	num epoch	6
num layers	1	summary freq	1000
time horizon	64		
extrinsic strength	1.0	extrinsic gamma	0.99
curiosity strength	0.01	curiosity gamma	0.99
curiosity encoding size	128		

Frames: 0, 5, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 40, 45, 50, 55, 60, 70, 80

Render Target Sensor: 20 * (20x20)
Grayscale: true
Observation space size = 25
Action space size = 1

Episode steps = 1000
Total steps = 460000
Time = 17200s

Use of simpler linear rewards, assuming the movement is deterministic and has no noise.

The reward grows slowly, and catches the impulses more or less well but may lack the complexity of the network structure to learn well.

26/5 19:58 (5_26_1958)

Trainer: BC

batches per epoch	10	batch size	64
hidden units	128	learning rate	1.0e-5
max steps	5.0e4	**memory size**	**32**
num layers	**3**	**sequence length**	**128**
summary freq	1000	**use recurrent**	**true**

Demo path: demos/UniformDemo.demo
Frames: 0, 5, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 40, 45, 50, 55, 60, 70, 80

Render Target Sensor: 20 * (20x20)
Grayscale: true
Observation space size = 25
Action space size = 1

Episode steps = 1000
Total steps = 12000
Time = 13000s

More complex network with imitation learning. The training time is very slow, but the results are good. However, the model loads the CPU a lot when inferring, decreasing the fps by half (25-30). When using recurrent memory, it may not be necessary to save previous movements and images.

As a bad part, from time to time random impulses appear without any target in sight, they may disappear with more training or they may occur due to lack of complexity (react time is 0.25).

In other experiments, different architectures could be tested using recurrent (1, 2 or 4 layers, with more or less hidden units), and some without more inputs than the current image (and maybe the previous motion, depending on how the recurrent memory works).

27/5 09:53 (5_27_0953)

Trainer: BC

batches per epoch	10	batch size	64
hidden units	128	learning rate	1.0e-5
max steps	5.0e4	memory size	32
num layers	3	sequence length	128
summary freq	1000	use recurrent	true

Demo path: demos/UniformNoRTDemo.demo

Render Target Sensor: 1 * (20x20)
Grayscale: true
Observation space size = 25
Action space size = 1

Episode steps = 1000
Total steps = 18000
Time = 16500s

From tests carried out, it seems that BC cannot be used with a vector observation of size 0 (I have left it at 25 as it was). It's also important to point out that it is necessary to turn off the demonstration recorder to train, and that it is possible to train online with PPO and SAC (I suppose that in a single thread).

Compare the training speed (and results) with the previous model, and the FPS when running it.

The result looks good, at the expense of testing it. It has the typical random impulses, we will have to see if it is more efficient too.

In the editor it goes crazy after a few seconds, maybe because it receives the empty previous movement vector. Try to do it with less movements.

Reaction 0.25.

27/5 16:56 (5_27_1656)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	8192	epsilon	0.2
hidden units	256	learning rate	2.0e-3
learning rate schedule	linear	max steps	6.0e5
memory size	**32**	normalize	true
num epoch	6	num layers	1
sequence length	**128**	summary freq	1000
time horizon	64	**use recurrent**	**true**
extrinsic strength	1.0	extrinsic gamma	0.99
curiosity strength	0.01	curiosity gamma	0.99
curiosity encoding size	128		

Render Target Sensor: 1 * (20x20)
Grayscale: true
Observation space size = 1
Action space size = 1

Episode steps = 1000
Total steps = 70000
Time = 1550s

An enumeration has been created to support more types of observations, in this case it only receives the last movement. Attempting to train with imitation learning does not work using only one observation (with 25).

In this training we use PPO with recurrent memory, and the last movement as visual input. It is done online. However, it has less layers than the previous ones with imitation learning.

I cancel it to try with SAC, it doesn't seem to improve the previous PPO results.

27/5 17:27 (5_27_1727)

Trainer: SAC

batch size	128	buffer size	200000
buffer init steps	5000	**hidden units**	**128**
init entcoef	1.0	learning rate	4.0e-4
learning rate schedule	constant	max steps	6.0e6
memory size	**32**	normalize	true
num update	1	train interval	5
num layers	**2**	time horizon	64
sequence length	128	summary freq	1000
tau	0.005	**use recurrent**	**true**
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	gail use actions	true

Demo path: demos/UniformNoRT1Demo.demo

Render Target Sensor: 1 * (20x20)

Grayscale: true

Observation space size = 1

Action space size = 1

Episode steps = 1000

Total steps = 235000

Time = 24000s

Online training with SAC and 2 layers in the network.

It looked like it was going to converge to the average, but in the end it has started to make very irregular impulses. It may work with more network complexity.

28/5 12:40 (5_28_1240)

Trainer: SAC

batch size	128	buffer size	200000
buffer init steps	5000	hidden units	128
init entcoef	1.0	learning rate	4.0e-4
learning rate schedule	constant	max steps	6.0e6
memory size	32	normalize	true
num update	1	train interval	5
num layers	2	time horizon	64
sequence length	128	summary freq	1000
tau	0.005	use recurrent	true
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	gail use actions	true

Demo path: demos/UniformNoRT1Demo.demo

Render Target Sensor: 1 * (20x20)
Grayscale: true
Observation space size = 1
Action space size = 1

Episode steps = 1000
Total steps = -
Time = -

–Tests
Training with higher requirements in rewards and more layers.

Does not work with 1 observation and 2 or 3 layers. The error seems to have to do with sequence length, I don't know if it should not be greater than the hidden layer units or if the relationship between memory size and sequence length should be kept to a minimum. I'm going to do some tests (with sequence length=128, like the hidden units, it works with an observation). The curious thing is that the failures were in step 5000.

sl=256, hu=128, ms=32: FAILS
sl=256, hu=256, ms=32: FAILS, must not be for equality with hidden units
sl=256, hu=128, ms=64: FAILS
sl=128, hu=128, ms=16: DOES NOT FAIL

For some reason it always fails when sequence length is greater than or equal to 256, the buffer size is also not the key. Therefore, the most that can be used is 128.

Could the encoding size of curiosity and gail be the cause, since both were 128? Test it later.

28/5 13:16 (5_28_1316)

Trainer: SAC

batch size	128	buffer size	200000
buffer init steps	5000	hidden units	128
init entcoef	1.0	learning rate	4.0e-4
learning rate schedule	constant	max steps	6.0e6
memory size	32	normalize	true
num update	1	train interval	5
num layers	**3**	time horizon	64
sequence length	128	summary freq	1000
tau	0.005	use recurrent	true
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	gail use actions	true

Demo path: demos/UniformNoRT1Demo.demo

Render Target Sensor: 1 * (20x20)
Grayscale: true
Observation space size = 1
Action space size = 1

Episode steps = 1000
Total steps = 272000
Time = 28000s

It didn't look bad after about 150000+ steps, it went from converging to average to start making impulses (with some noise), but then it started to go crazy. However, the reward has increased, so I think there may be a problem that makes the rewards not very well balanced. Either that, or it may require some additional layer.

I've checked that sequence length also fails with equal encoding sizes, so it can't be 256 in general.

28/5 21:08 (5_28_2108)

Trainer: SAC

batch size	128	buffer size	200000
buffer init steps	5000	hidden units	128
init entcoef	1.0	learning rate	4.0e-4
learning rate schedule	constant	max steps	6.0e6
memory size	32	normalize	true
num update	1	train interval	5
num layers	**4**	time horizon	64
sequence length	128	summary freq	1000
tau	0.005	use recurrent	true
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	gail use actions	true

Demo path: demos/UniformNoRT1Demo.demo

Render Target Sensor: 1 * (20x20)
Grayscale: true
Observation space size = 1
Action space size = 1

Episode steps = 1000
Total steps = 100000
Time = 10300s

One more layer in the network. I stop the training but it seems to be following the same path as the previous one. I could try again by combining the use of recurrent and the previous images.

Despite having one more layer, it seems to have no added cost. It would be nice to continue with this same training (–load)

29/5 09:26 (5_29_0926)

Trainer: BC

batches per epoch	10	batch size	64
hidden units	128	learning rate	1.0e-5
max steps	5.0e4	memory size	32
num layers	3	sequence length	128
summary freq	1000	use recurrent	true

Demo path: demos/UniformNoRT1Demo.demo

Render Target Sensor: 1 * (20x20)
Grayscale: true
Observation space size = 25
Action space size = 1

Episode steps = 1000
Total steps = 18000
Time = 17200s

BC training using an input. The rest of the parameters are the same as in the last training that worked.

Testing if it works well in the editor, the result seemed correct (at least with the ones on the left).

In editor it doesn't work well after encountering a left-hand pulse: when it stands still it enters untrained cases.

29/5 15:04 (5_29_1504)

Trainer: BC

batches per epoch	10	batch size	64
hidden units	128	learning rate	1.0e-5
max steps	5.0e4	memory size	32
num layers	**2**	sequence length	128
summary freq	1000	use recurrent	true

Demo path: demos/UniformNoRT1Demo.demo

Render Target Sensor: 1 * (20x20)
Grayscale: true
Observation space size = 25
Action space size = 1

Episode steps = 1000
Total steps = 23000
Time = 21500s

BC training and one less layer. It's very similar to the previous one.

In the game it stops working in about 2 seconds, when it starts spinning uncontrollably.

29/5 21:26 (5_29_2126)

Trainer: SAC

batch size	128	buffer size	200000
buffer init steps	5000	hidden units	128
init entcoef	1.0	learning rate	4.0e-4
learning rate schedule	constant	max steps	6.0e6
memory size	32	normalize	true
num update	1	train interval	5
num layers	**2**	time horizon	64
sequence length	128	summary freq	1000
tau	0.005	use recurrent	true
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	gail use actions	true

Demo path: demos/UniformNoRT1Demo.demo

Render Target Sensor: 1 * (20x20)

Grayscale: true

Observation space size = 1

Action space size = 1

Episode steps = 1000

Total steps = 90000

Time = 9000s

Training with adjusted rewards (squared). Although the demo has not been changed, so it will have old rewards there. In this first test we only use 2 layers.

It's close to average, but it doesn't make any impulses.

31/5 19:29 (5_31_1929)

Trainer: SAC

batch size	128	buffer size	200000
buffer init steps	5000	hidden units	128
init entcoef	1.0	**learning rate**	**7.5e-4**
learning rate schedule	constant	max steps	6.0e6
memory size	32	normalize	true
num update	1	train interval	5
num layers	**3**	time horizon	64
sequence length	128	summary freq	1000
tau	0.005	use recurrent	true
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	gail use actions	true

Demo path: demos/UniformNoRT1Demo.demo

Render Target Sensor: 1 * (20x20)

Grayscale: true

Observation space size = 1

Action space size = 1

Episode steps = 1000

Total steps = 153000

Time = 15700s

Higher learning rate. In following trainings it would be good to base the rewards on coherence (adapted to a single input). It has grown faster but falls back after 90k steps. It doesn't get to make impulses as far as I've seen, it would be convenient to search among previous models to readapt them.

01/6 10:50 (6_01_1050)

Trainer: SAC

batch size	128	buffer size	200000
buffer init steps	5000	hidden units	128
init entcoef	1.0	**learning rate**	**5.0e-4**
learning rate schedule	constant	max steps	6.0e6
memory size	32	normalize	true
num update	1	train interval	5
num layers	3	time horizon	64
sequence length	128	summary freq	1000
tau	0.005	use recurrent	true
vis encode type	simple		
pretraining strength	0.4	pretraining steps	20000
extrinsic strength	**1.0**	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	gail use actions	true

Demo path: demos/UniformNoRT1Demo.demo

Render Target Sensor: 1 * (20x20)

Grayscale: true

Observation space size = 1

Action space size = 1

Episode steps = 1000

Total steps = 190000

Time = 19700s

Training with higher gail and curiosity weight (in ratio) and reward factor of 300 (against 10). Seems to be stagnating on the same behaviors.

01/6 16:23 (6_01_1623)

Trainer: SAC

batch size	128	buffer size	200000
buffer init steps	5000	hidden units	128
init entcoef	1.0	learning rate	5.0e-4
learning rate schedule	constant	max steps	6.0e6
memory size	32	normalize	true
num update	1	train interval	5
num layers	3	time horizon	64
sequence length	128	summary freq	1000
tau	0.005	use recurrent	true
vis encode type	simple		
pretraining strength	0.4	pretraining steps	20000
extrinsic strength	1.0	extrinsic gamma	0.99
curiosity strength	**0.1**	curiosity gamma	0.99
curiosity encoding size	128		
gail strength	0.03	gail gamma	0.99
gail encoding size	128	gail use actions	true

Demo path: demos/UniformNoRT1Demo.demo

Render Target Sensor: 1 * (20x20)

Grayscale: true

Observation space size = 1

Action space size = 1

Episode steps = 1000

Total steps = 305000

Time = 32000s

It goes up until it adapts to the average (without impulses) but then it goes down again, it is not useful to use recurrent memory in these cases without other information (it might work but with a much more perfected system of rewards). Next time try the combination of render targets and recurrent but with SAC (and a single motion input).

02/6 10:23 (6_02_1023)

Trainer: SAC

batch size	128	buffer size	200000
buffer init steps	5000	hidden units	128
init entcoef	1.0	learning rate	5.0e-4
learning rate schedule	constant	max steps	6.0e6
memory size	32	normalize	true
num update	1	train interval	5
num layers	3	time horizon	64
sequence length	128	summary freq	1000
tau	0.005	use recurrent	true
vis encode type	simple		
pretraining strength	0.4	pretraining steps	20000
extrinsic strength	1.0	extrinsic gamma	0.99
curiosity strength	0.1	curiosity gamma	0.99
curiosity encoding size	128		
gail strength	0.03	gail gamma	0.99
gail encoding size	128	gail use actions	true

Frames: 0, 5, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 40, 45, 50, 55, 60, 70, 80

Demo path: demos/Frames20Demo.demo

Render Target Sensor: 20 * (20x20)

Grayscale: true

Observation space size = 1

Action space size = 1

Episode steps = 1000

Total steps = 90000

Time = 18000s

Training with 20 render targets, the demo has also been updated. The objective is to see if it improves the performance with respect to the one trained with SAC. It is still only adjusted to the average, it doesn't seem to have any incentive to make impulses. In the next training I will use as input the movements of the bot to see how long it takes to learn (if it does), if it does not learn, change reward system (even to TR) and if it learns, check the reward in which it behaves decently to incorporate learning curriculum.

02/6 15:45 (6_02_1545)

Trainer: SAC

batch size	128	buffer size	200000
buffer init steps	5000	hidden units	128
init entcoef	1.0	learning rate	5.0e-4
learning rate schedule	constant	max steps	6.0e6
memory size	32	normalize	true
num update	1	train interval	5
num layers	3	time horizon	64
sequence length	128	summary freq	1000
tau	0.005	use recurrent	true
vis encode type	simple		
pretraining strength	0.4	pretraining steps	20000
extrinsic strength	1.0	extrinsic gamma	0.99
curiosity strength	0.1	curiosity gamma	0.99
curiosity encoding size	128		
gail strength	0.03	gail gamma	0.99
gail encoding size	128	gail use actions	true

Frames: 0, 5, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 40, 45, 50, 55, 60, 70, 80

Demo path: demos/Frames20Demo.demo

Render Target Sensor: 20 * (20x20)

Grayscale: true

Observation space size = 1

Action space size = 1

Episode steps = 1000

Total steps = 86000

Time = 17600s

Receives actual values from the bot. Doesn't react anyway.

Add noise in next tests.

02/6 20:54 (6_02_2054)

Trainer: SAC

batch size	128	buffer size	200000
buffer init steps	5000	hidden units	128
init entcoef	1.0	learning rate	5.0e-4
learning rate schedule	constant	max steps	6.0e6
memory size	32	normalize	true
num update	1	train interval	5
num layers	3	time horizon	64
sequence length	128	summary freq	1000
tau	0.005	use recurrent	true
vis encode type	simple		
pretraining strength	0.4	pretraining steps	20000
extrinsic strength	1.0	extrinsic gamma	0.99
curiosity strength	0.1	curiosity gamma	0.99
curiosity encoding size	128		
gail strength	0.03	gail gamma	0.99
gail encoding size	128	gail use actions	true

Frames: 0, 5, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 40, 45, 50, 55, 60, 70, 80

Demo path: demos/R0N02.demo

Render Target Sensor: 20 * (20x20)

Grayscale: true

Observation space size = 1

Action space size = 1

Episode steps = 1000

Total steps = 55000

Time = 9500s

Parameters in the bot (reaction=0, noise=0.2) and rewards with tolerable range (TR=0.05, MR=1), as a reference point to see if it works. The demo has an identifying name. This model doesn't work because there was an error in the tolerable range rewards that made it always give 0 as a punishment.

02/6 23:41 (6_02_2341)

Trainer: SAC

batch size	128	buffer size	200000
buffer init steps	5000	hidden units	128
init entcoef	1.0	learning rate	5.0e-4
learning rate schedule	constant	max steps	6.0e6
memory size	32	normalize	true
num update	1	train interval	5
num layers	3	time horizon	64
sequence length	128	summary freq	1000
tau	0.005	use recurrent	true
vis encode type	simple		
pretraining strength	0.4	pretraining steps	20000
extrinsic strength	1.0	extrinsic gamma	0.99
curiosity strength	0.1	curiosity gamma	0.99
curiosity encoding size	128		
gail strength	0.03	gail gamma	0.99
gail encoding size	128	gail use actions	true

Frames: 0, 5, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 40, 45, 50, 55, 60, 70, 80

Demo path: demos/R0N02.demo

Render Target Sensor: 20 * (20x20)

Grayscale: true

Observation space size = 1

Action space size = 1

Episode steps = 1000

Total steps = 100000

Time = 7300s + 11000s

Same parameters but corrected. The demo is redone because the previous one has incorrect rewards. Also the noise has been reduced to 0.05 to enter the tolerable range.

Training interrupted, continue.

Next models could be trained with PPO, as it should be faster in reaching a certain level. It doesn't seem to go too high, and it may be caused by the resolution of the render targets (by discard). In the next test PPO.

03/6 12:41 (6_03_1241)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	8192	epsilon	0.2
hidden units	256	learning rate	2.0e-3
learning rate schedule	linear	max steps	6.0e5
memory size	32	normalize	true
num epoch	6	num layers	1
sequence length	128	summary freq	1000
time horizon	64	use recurrent	true
extrinsic strength	1.0	extrinsic gamma	0.99
curiosity strength	0.01	curiosity gamma	0.99
curiosity encoding size	128		

Frames: 0, 5, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 40, 45, 50, 55, 60, 70, 80

Render Target Sensor: 20 * (20x20)
Grayscale: true
Observation space size = 1
Action space size = 1

Episode steps = 1000
Total steps = 154000
Time = 7000s

PPO training with the parameters that were. No demo included.
Converge to the average, no impulses. Next I try with 64x64 render targets.

03/6 14:54 (6_03_1454)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	8192	epsilon	0.2
hidden units	256	**learning rate**	**2.5e-4**
learning rate schedule	linear	max steps	6.0e5
memory size	32	**normalize**	**false**
num epoch	6	num layers	1
sequence length	128	summary freq	1000
time horizon	64	use recurrent	true
extrinsic strength	1.0	extrinsic gamma	0.99
curiosity strength	0.01	curiosity gamma	0.99
curiosity encoding size	128		

Render Target Sensor: 64*64

Grayscale: true
Observation space size = 1
Action space size = 1

Episode steps = 1000
Total steps = 600000
Time = 14000s

Change in render target, normalize and learning rate. If it works well it serves as a reference, if not there is a problem with another parameter or the reward system.

It didn't work. Next time I'll try with rewards with coherence (that I should create).

05/6 12:05 (6_05_1205)

Trainer: SAC

batch size	128	buffer size	200000
buffer init steps	5000	hidden units	128
init entcoef	1.0	learning rate	5.0e-4
learning rate schedule	constant	max steps	6.0e6
memory size	32	normalize	true
num update	1	train interval	5
num layers	3	time horizon	64
sequence length	128	summary freq	1000
tau	0.005	use recurrent	true
vis encode type	simple		
pretraining strength	0.4	pretraining steps	20000
extrinsic strength	1.0	extrinsic gamma	0.99
curiosity strength	0.1	curiosity gamma	0.99
curiosity encoding size	128		
gail strength	0.03	gail gamma	0.99
gail encoding size	128	gail use actions	true

Frames: 0, 5, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 40, 45, 50, 55, 60, 70, 80
Demo path: demos/Frames20Demo.demo

Render Target Sensor: 20 * (20x20)
Grayscale: true
Observation space size = 1
Action space size = 1

Episode steps = 1000
Total steps = 82000
Time = 15900s

First training after migrating repository. It has given some error at the beginning for not finding references (to something of python or ml agents).

Now I use rewards inside range (they are similar to TR, but they give maximum reward as long as it is within range).

It gets close to the average as expected.

05/6 16:59 (6_05_1659)

Trainer: SAC

batch size	128	buffer size	200000
buffer init steps	5000	hidden units	128
init entcoef	1.0	learning rate	5.0e-4
learning rate schedule	constant	max steps	6.0e6
memory size	32	normalize	true
num update	1	train interval	5
num layers	3	time horizon	64
sequence length	128	summary freq	1000
tau	0.005	use recurrent	true
vis encode type	simple		
pretraining strength	0.4	pretraining steps	20000
extrinsic strength	1.0	extrinsic gamma	0.99
curiosity strength	0.1	curiosity gamma	0.99
curiosity encoding size	128		
gail strength	0.03	gail gamma	0.99
gail encoding size	128	gail use actions	true

Frames: 0, 5, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 40, 45, 50, 55, 60, 70, 80
Demo path: demos/Frames20Demo.demo

Render Target Sensor: 20 * (20x20)
Grayscale: true
Observation space size = 1
Action space size = 1

Episode steps = 1000
Total steps = 117000
Time = 25200s

Change in Rewards (StdCoherenceIndividual): Failing coherent moves penalizes, and succeeding inconsistent moves gives reward (penalizes only if done in the opposite direction). The average is a bit unstable but should be fine. In future models it would be good to use dynamic averaging and deviation.

It doesn't converge.

06/6 14:33 (6_06_1433)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	8192	epsilon	0.2
hidden units	256	learning rate	2.5e-4
learning rate schedule	linear	max steps	6.0e5
memory size	32	normalize	false
num epoch	6	num layers	1
sequence length	128	summary freq	1000
time horizon	64	use recurrent	true
extrinsic strength	1.0	extrinsic gamma	0.99
curiosity strength	0.01	curiosity gamma	0.99
curiosity encoding size	128		

Render Target Sensor: 20 * (20x20)

Grayscale: true

Observation space size = 1

Action space size = 1

Episode steps = 1000

Total steps = 137000

Time = 6200s

Dynamic average and deviation implemented. The previous reward is used but now takes into account half of the deviation to start penalizing (that's how far it usually expands).

It tends to exploit the rewards by staying in the margin where the impulses usually appear. Needs more penalty.

06/6 16:30 (6_06_1630)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	8192	epsilon	0.2
hidden units	256	learning rate	2.5e-4
learning rate schedule	linear	max steps	6.0e5
memory size	32	normalize	false
num epoch	6	num layers	1
sequence length	128	summary freq	1000
time horizon	64	use recurrent	true
extrinsic strength	1.0	extrinsic gamma	0.99
curiosity strength	0.01	curiosity gamma	0.99
curiosity encoding size	128		

Render Target Sensor: 1 * (20x20)
Grayscale: false
Observation space size = 1
Action space size = 1

Episode steps = 1000
Total steps = 487000
Time = 10000s

Slight changes in the linearity of rewards and factors Now I go back to NOT using grayscale in the rendering. In next tests I may put the previous 25 moves back.

Failed training by leaving the reaction time at 0.2 by mistake.

06/6 19:33 (6_06_1933)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	8192	epsilon	0.2
hidden units	256	learning rate	2.5e-4
learning rate schedule	linear	max steps	6.0e5
memory size	32	normalize	false
num epoch	6	num layers	1
sequence length	128	summary freq	1000
time horizon	64	use recurrent	true
extrinsic strength	1.0	extrinsic gamma	0.99
curiosity strength	0.01	curiosity gamma	0.99
curiosity encoding size	128		

Render Target Sensor: 1 * (20x20)

Grayscale: false
Observation space size = 1
Action space size = 1

Episode steps = 1000
Total steps = 411000
Time = 9000s

Training with react time = 0, in theory it should converge, unless the rewards are not yet balanced.

It doesn't converge at all.

06/6 23:19 (6__06__2319)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	8192	epsilon	0.2
hidden units	256	learning rate	2.5e-4
learning rate schedule	linear	max steps	6.0e5
memory size	32	normalize	false
num epoch	6	num layers	1
sequence length	128	summary freq	1000
time horizon	64	**use recurrent**	**false**
extrinsic strength	1.0	extrinsic gamma	0.99
curiosity strength	0.01	curiosity gamma	0.99
curiosity encoding size	128		

Render Target Sensor: 1 * (64x64)
Grayscale: false
Observation space size = 1
Action space size = 1

Episode steps = 1000
Total steps = 432000
Time = 9400s

This time I take away recurrent and double the reward factor (50 against 100).

It has failed a training (it has stopped in 8000 steps, but without giving error).

It has managed to follow impulses with great precision but it lacks a jump in the penalties so that it tends to the average in the coherent movements.

In addition, the buffer size may have to be decreased to avoid errors like the previous one (check how it was in previous successful models). Keep in mind that using the model as it is will probably cause it to diverge (since it has only trained with real values). In

the next training the result of using the bot values instead of the real ones could be compared.

07/6 10:00 (6_07_1000)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	8192	epsilon	0.2
hidden units	256	learning rate	2.5e-4
learning rate schedule	linear	max steps	6.0e5
memory size	32	normalize	false
num epoch	6	num layers	1
sequence length	128	summary freq	1000
time horizon	64	use recurrent	false
extrinsic strength	1.0	extrinsic gamma	0.99
curiosity strength	0.01	curiosity gamma	0.99
curiosity encoding size	128		

Render Target Sensor: 1 * (64x64)

Grayscale: false

Observation space size = 1

Action space size = 1

Episode steps = 1000

Total steps = 235000

Time = 5000s

In this training it receives the observations of the bot itself. This allows you to compare the learning curve of both.

Remember to change the linearity of the rewards after this training, and the buffer size.

It follows exactly the same path as the previous model.

07/6 11:39 (6_07_1139)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	**4096**	epsilon	0.2
hidden units	256	learning rate	2.5e-4
learning rate schedule	linear	max steps	6.0e5
memory size	32	normalize	false
num epoch	6	num layers	1
sequence length	128	summary freq	1000
time horizon	64	use recurrent	false
extrinsic strength	1.0	extrinsic gamma	0.99
curiosity strength	0.01	curiosity gamma	0.99
curiosity encoding size	128		

Render Target Sensor: 1 * (64x64)
Grayscale: false
Observation space size = 1
Action space size = 1

Episode steps = 1000
Total steps = 588000
Time = 12700s

More pressure has been added to the rewards, and 2 jumps in value. The parameters are still 50 reward and 100 punish, and the agent's observations. The buffer has also been changed. The result seems similar to the previous 2.

07/6 15:15 (6_07_1515)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	4096	epsilon	0.2
hidden units	256	learning rate	2.5e-4
learning rate schedule	linear	max steps	6.0e5
memory size	32	normalize	false
num epoch	6	num layers	1
sequence length	128	summary freq	1000
time horizon	64	use recurrent	false
extrinsic strength	1.0	extrinsic gamma	0.99
curiosity strength	0.01	curiosity gamma	0.99
curiosity encoding size	128		

Render Target Sensor: 1 * (64x64)

Grayscale: false
Observation space size = 1
Action space size = 1

Episode steps = 1000
Total steps = 418000
Time = 9000s

Even more penalty pressure has been added (it now always penalizes inconsistent movement, unless it is exactly the same, and linearly increasing with distance). If it doesn't work, the reward/penalty factors will have to be adjusted. The other parameters are the same.

It works the same as the previous ones, that's why it is not saved.

07/6 17:47 (6_07_1747)

Trainer: PPO

batch size	1024	beta	5.0e-3
buffer size	4096	epsilon	0.2
hidden units	256	learning rate	2.5e-4
learning rate schedule	linear	max steps	6.0e5
memory size	32	normalize	false
num epoch	6	num layers	1
sequence length	128	summary freq	1000
time horizon	64	use recurrent	false
extrinsic strength	1.0	extrinsic gamma	0.99
curiosity strength	0.01	curiosity gamma	0.99
curiosity encoding size	128		

Render Target Sensor: 1 * (64x64)
Grayscale: false
Observation space size = 1
Action space size = 1

Episode steps = 1000
Total steps = -
Time = -

Penalty factor at 150 and reward factor at 50.

Up to 230000 steps (-20 reward) follows all similar (makes perfect impulses but has a lot of noise in normal movement), so I'm going to increase the penalty to 250 at this point. The reward goes down to -42, if it evolves well from now on it may work as a learning curriculum, if it follows the same trend it will be better to change the form of

the penalties and/or use SAC (Also keep in mind that the learning rate has already gone down).

It does not change.