Project: Go-Explore, a New Approach for Hard-Exploration Problems

ADEIKALAM Pierre
CHEN Guangyue
XU Kevin

Lecturer : LE PENNEC Erwan

March 2020

1 Introduction

Hard-Exploration Reinforcement Learning Problems are problems whose domain has sparse, delayed or deceptive rewards and require exploration to solve [2] [3]. An example for this kind of problems is the Montezuma's Revenge video game [6]. In this game, the player has to score points by gathering jewels and killing enemies. However, most jewels are near the end of the levels, and the player needs to open doors with specific keys found throughout the level to reach these jewels. Therefore, to score the most amount of points, the player has to first explore the level and find keys, then go to the corresponding doors, open them and pick up the jewels. In this game, rewards are very sparse and basic RL algorithms fail to understand what the objective of the game is and how to solve it.

The traditional way to solve these problems is to use intrinsic motivation algorithms that reward the agent when it finds itself in a state it has never seen before [2]. There are many variants of this method but at the time of publication of Go-Explore, none of them could consistently solve the first level of Montezuma's Revenge.[4]

The reason why these algorithms failed to solve Hard-Exploration problems such as Montezuma's Revenge are the two problems of intrinsic motivation reinforcement learning known as "Detachment" and "Derailment".[4]

Detachment: After exploring a promising path, the agent stops getting reward signals for further exploring and it has forgotten the other paths it could have taken. Intuitively, if an agent finds itself at an intersection and decides to explore the path on the left, after exploring this path it will have completely

forgotten that the intersection and the other paths it could have explored existed.

Derailment: Suppose the agent has found a promising path. By adding random perturbations to the policy it might not be able to reach that promising path again, meaning that the exploration cannot continue. This problem becomes worse as the length, complexity and precision needed to reach that spot increases.

2 Approaches to solving the detachment and derailment problems

In the same year as the Go-Explore results were published, OpenAI and Deepmind claimed that they could solve the first level of Montezuma's revenge [1] [5]. Their approaches rely on expert human demonstrations which fundamentally changes the nature of the learning problem. Indeed, now the learning problem is an "imitation" learning problem in that the agent now is guided by an external demonstration and its objective is to obtain at least the same reward. Hence, the detachment and derailment problems are solved by forcing the agent to follow a path that already explores promising locations while also solving the game.

While the use of demonstrations is a compelling way to provide agents with meaningful knowledge about a task, the problem with Montezuma's revenge is that the game is deterministic so the agent could just memorize a sequence of actions to solve the level. Therefore, what is learned is not how to solve complex exploration problems but rather how to memorize a sequence of movements.

The Go-Explore approach is similar but humans are taken out of the equation, which justifies the use of imitation learning to solve exploration problems. Indeed, now the process of finding sequences to solve the game is automated so that we can generate infinitely many different demonstrations which might help robustify the learned policy of the agent. In our implementation of this approach, we will see that this approach actually generalizes to other exploration problems without relying on human demonstrations unlike the approaches proposed by OpenAI and Deepmind.

3 Solving the detachment and derailment problems without human help

3.1 The Go-Explore Approach

The Go-Explore approach is done in 2 phases.

The first phase consists in explicitly remembering promising locations and getting back to them to explore further. Each time the path of the agent improves upon the game's score or the length of the previous best path while having at least the same score, the path is stored. This process is repeated until the game is solved.

The second phase consists in training the agent's policy via imitation learning on the optimal paths found during phase 1. This learning process is exactly the same as in the OpenAI paper [5].

The first step to the phase 1 algorithm is to choose a cell to explore from. A cell is a representation of the agent and the game's state. A cell can be represented by a dictionary whose keys are pieces of metadata that allow us to perfectly reconstruct the state of the game and the agent, kind of like a saved game file. The information contained within the cell *with domain knowledge* are:

- The location of the agent.
- The actions of the agent that led to this location.
- The score obtained by the agent at that point.
- Attributes that we will define later that allow us to choose a cell to explore from.

If the agent takes two trajectories that end up in the same location, the cell associated to that location will only store the trajectory with the highest score, and if both have the same score the cell will contain the trajectory with the shortest length.

3.2 Scoring a cell

In order to choose a cell to explore from, the authors engineered different cell attributes that quantify how "promising" a cell is. These attributes are:

- Number of times a cell was chosen to explore from.
- Number of times the location of the cell was seen at any point during training.
- Number of times since the choosing of this cell has led to the discovery of the new cell.

The first two attributes quantify how new a cell is while the third one quantifies how productive it is. The Cell Count score is the weighted sum of inverse of these attributes. The weights associated to each attribute allow the user to control which quality of a cell they wish to target.

The next scoring function is the Neighbor Score. If a cell has neighbors that have not yet been discovered, the Neighbor Score of this cell will increase. The amount of which this score increases depends on the fact if the undiscovered cell contains a reward and if the undiscovered cell is a vertical neighbor or horizontal neighbor.

The final score of a cell is the sum of these two scores. For each cell that has been discovered, we compute their score after each exploration round, such that the probability of choosing a cell in the next exploration round is given by:

$$CellProb(c) = \frac{Cellscore(c)}{\sum_{c' \in archive} Cellscore(c')}$$

It is worth noting that even when all possibles cells have been discovered, the probability of a choosing a cell is never set to 0.

In the next Section, we will implement the Phase 1 Algorithm and give plenty of visualisation aid to help understand how the paths are generated.

4 Implementation and Experiments

4.1 Implementation

One problem with the phase 1 algorithm is that it requires to store a cell for every single location that was visited, which can be memory intensive. This is why we have developed our own exploration environment in order to simplify the implementation, better visualize how the phase 1 algorithm works and reduce memory costs. Our full implementation can be found in the Jupyter notebook associated with this report.

The environment we created is a labyrinth with 4 rooms to explore and each room has rewards to collect. Each room is created procedurally so that we can customize the size of each room in order to tune the exploration difficulty of our game. In Figure 1, we generate different environments with different room sizes.

In order to maximise the game's score, the agent has to collect all the rewards before leaving the map through one of the exits. To go from one room to the other, the agent needs to go through a narrow corridor and make a turn towards the the room it wishes to visit.

We can now run phase 1 in the environment. In Figure 2 we show the state of the phase 1 algorithm after a few iterations but we encourage the reader to rather look at the animation *run1.gif* found in the same folder as this report as it illustrates the full phase 1 run.

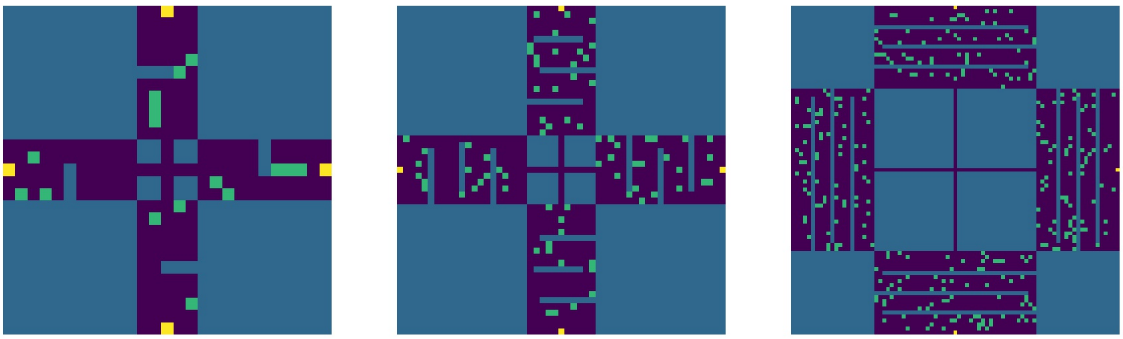

Figure 1: Different generated environments. On the left map, each room was generated with a room size of 11x5 pixels. In the middle, the room size was set to 21x11 pixels and on the right the room size was set to 41x21 pixels. The green pixels are the rewards, the yellow pixels are exits and the blue pixels are walls the agent cannot go through. The agent starts the game in the middle of the map and its objective is to collect all the rewards.

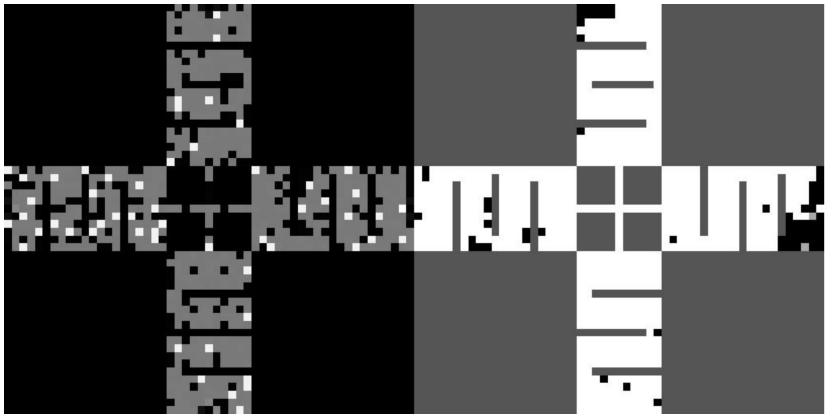

Figure 2: State of the Phase 1 Algorithm after a few hundred iterations on a map where each room has a size of 21x11 pixels. This is a screen shot from *run1.gif* that animates the full phase 1 run on this map. On the left is shown the cell scores of all the discovered locations in the map. On the right is shown the best trajectory found by the algorithm.

4.2 Results

In Figure 3, we have found that by setting the weights of the Neighbor Score to a very high value, we can tune the algorithm to be extremely greedy, unlike

in Figure 2 where the weights of the CountScore and NeighborScore have been set to where the authors recommend, which balances greed and curiosity. This is what gives this algorithm the flexibility to adapt to settings such as ours, Montezuma's Revenge, Pitfall and maybe even Mario, Zelda, etc.

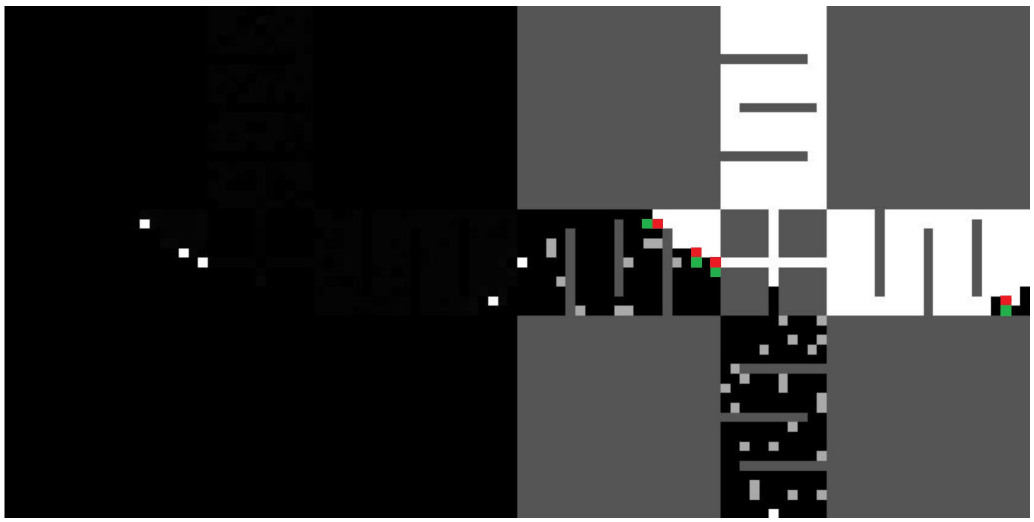

Figure 3: State of the Phase 1 Algorithm when the Neighbor Score is tuned to be very high. In this case, we see that the exploration algorithm is very greedy and gives a very high score to cells whose neighbors contain rewards while giving an almost null score the other ones. In red we have highlighted the cells that correspond to the bright spots on the left image and in green the rewards next to these cells that have not yet been collected by the agent. This is a screen shot from the run in *high_neighbor_weights.gif*.

The Phase 1 algorithm is able to solve our game, which means we do have obtained the "expert" demonstration needed to train the policy of the agent via imitation learning. However, the illustrations we have shown until now in Figure 2, Figure 3, *run1.gif* and *high_neighbor_weights.gif*. can be misleading as they might lead the viewer into thinking that the trajectories taken by the agent are efficient. What we have not shown is that it actually takes thousands of steps for the agent to solve this simple game where a human might solve it in a couple hundreds. In *agent_run.gif*, we animate the shortest trajectory found by the phase 1 algorithm that solves the game and in Figure 4, we show how inefficient this trajectory can be.

Moreover, we have experimented on different map sizes and we see that as the size of the rooms get larger, a lot more exploration iterations are needed to solve the game. The largest map we have tried is a map of size 83x83 pixels, which is a lot smaller that the map of Montezuma's Revenge. In this map, completing 5000 exploration iterations took about 3 hours with an AMD Ryzen 5 3500x **desktop** CPU. Since CPU load was only about 15% to run phase 1,

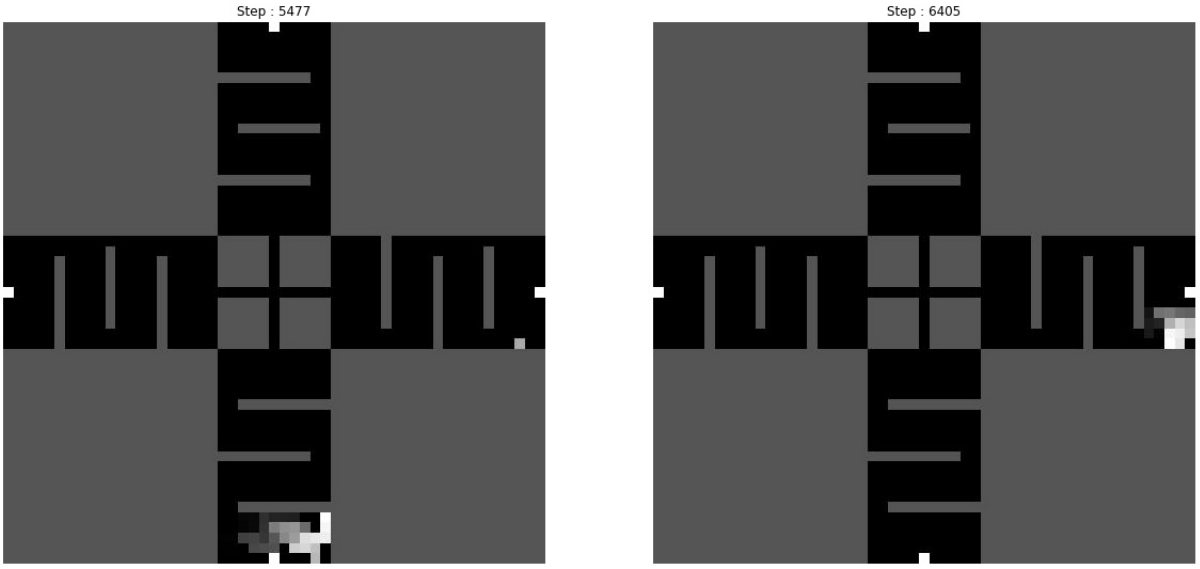

Figure 4: On the left we show the trajectory at step 5477. The agent is in the bottom room and it has found all rewards except one that is located in the right room. The agent will collect that final reward at step 6405. Therefore, the agent will need 928 steps to go from the bottom room to the room on the right, while a human could have done that in about 100, thus showing the inefficiency of the agent's trajectory.

we were able to run 6 instances of phase 1 in parallel, for total of 12 runs of 30000 exploration iterations each. Obtaining the 12 green runs shown in Figure 5 took a little under 24 hours. We also note that saving each run with its cell data took about 55GB of drive space with the **pickle** package.

This shows the clear weaknesses of the phase 1 algorithm. Finding a trajectory that solves the game is time-consuming for large maps and is quick only for small maps where tabular methods might have been just as fast. This explains why the authors first needed to heavily shrink the state space of Montezuma's Revenge to implement phase 1, and the shrinkage factor had to be found through cross validation, which removes the generalization power of this approach. Moreover, the trajectories found are highly inefficient compared to human demonstrations which is why robustification with **many runs** is needed. Thankfully, demonstrations can be generated in parallel. Another approach

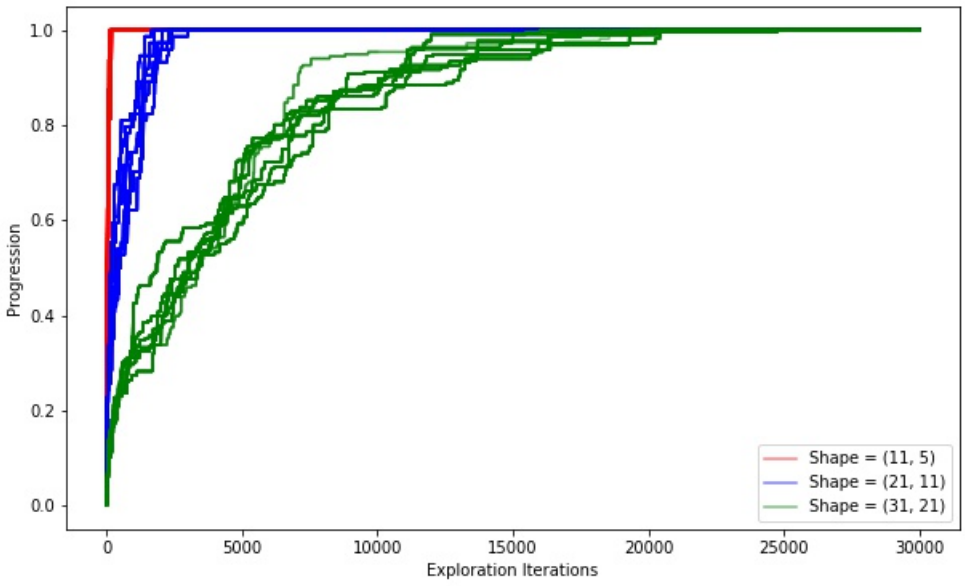

Figure 5: Experiments on how many exploration iterations are needed to solve the game for different map sizes. **Each exploration iteration corresponds to 200 random agent steps**, meaning that 10000 exploration iterations correspond to 2 million agent steps. The red curves correspond to runs on maps where the rooms have a size of 11x5 pixels, the blue curves correspond to maps with a room size of 21x11 pixels (the same as in the figures we have shown until now) and the green curves correspond to maps with a room size of 31x21 pixels.

could have been to fully explore the game to find the locations of the rewards and then run a path-finding algorithm that finds an optimal trajectory by interpolating through each reward. However, this still can only apply to our game where there are no enemies and traps to avoid.

5 Conclusion

While the Go-Explore approach can only find trajectories that are not globally optimal, it completely puts humans out of the loop. This justifies the use of imitation learning as a general approach to solving exploration problems. Moreover, by generating a large amount of trajectories to learn from, we can produce a learning algorithm that could generalize to new settings, which is shown by the performance of Go-Explore on Montezuma's Revenge where it can complete new levels that were not in the trajectories generated during phase 1. For our

environment, we feel that if we had enough computation power, we could have taught our agent to explore maps of different sizes and maybe maps that consist in concatenations of many map instances.

Sadly, we were not able to implement phase 2 as imitation learning through deep policy learning was out of the scope of the course and our attempts to implement the Backward Algorithm of Salimans and Chen [5] on our environment were unsuccessful. Traces of our attempts can be found at the end of the Jupyter Notebook of this project.

References

[1] Yusuf Aytar, Tobias Pfaff, David Budden, Tom Le Paine, Ziyu Wang, and Nando de Freitas. Playing hard exploration games by watching youtube. 2018.

[2] Yuri Burda, Harri Edwards, Deepak Pathak, Amos Storkey, Trevor Darrell, and Alexei A. Efros. Large-scale study of curiosity-driven learning. 2018.

[3] Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. Exploration by random network distillation. 2018.

[4] Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Go-explore: a new approach for hard-exploration problems. 2019.

[5] Tim Salimans and Richard Chen. Learning montezuma's revenge from a single demonstration. 2018.

[6] Wikipedia contributors. Montezuma's revenge (video game) — Wikipedia, the free encyclopedia, 2020.