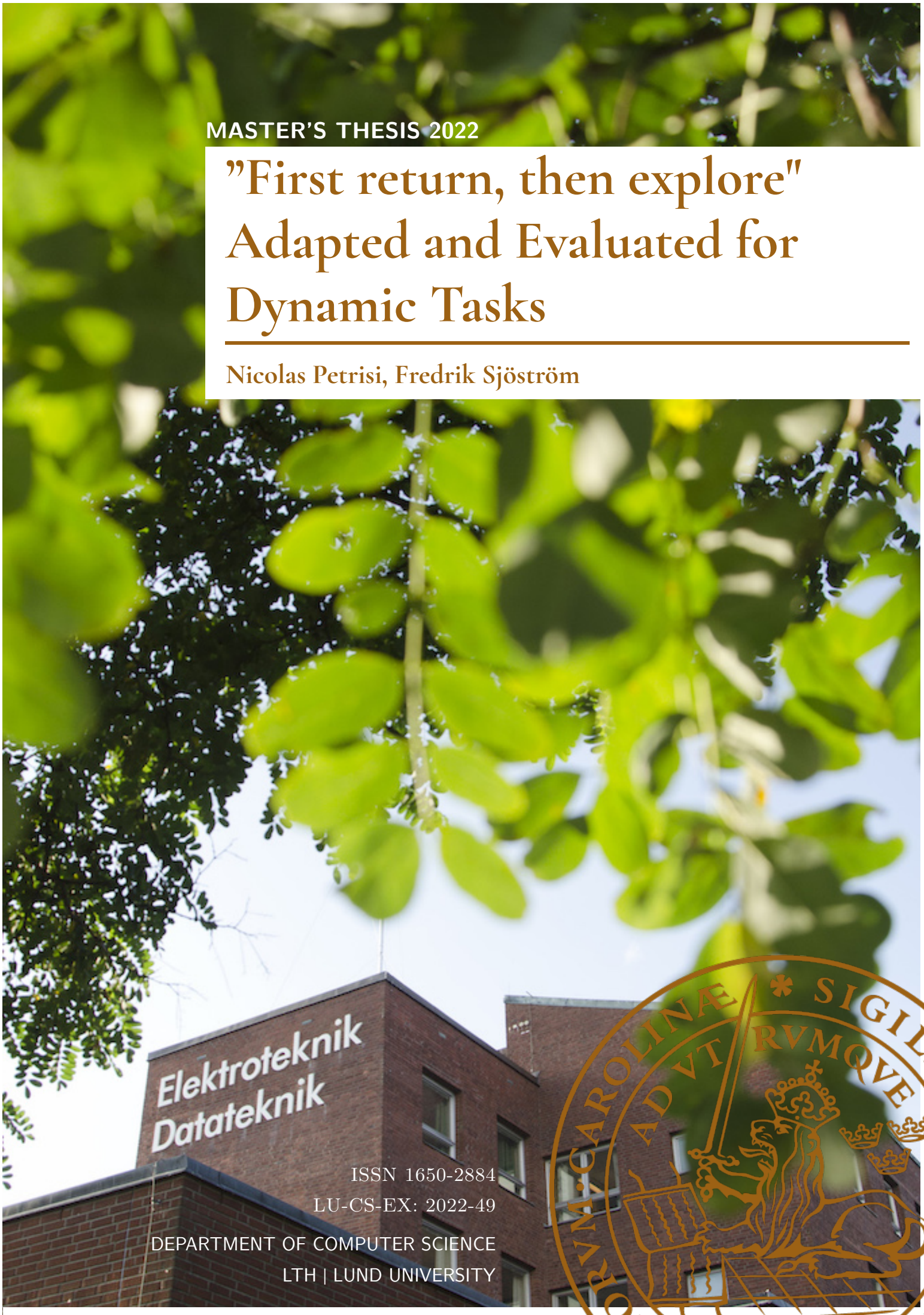


MASTER'S THESIS 2022

"First return, then explore" Adapted and Evaluated for Dynamic Tasks

Nicolas Petrisi, Fredrik Sjöström

ISSN 1650-2884
LU-CS-EX: 2022-49

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2022-49

”First return, then explore" Adapted and Evaluated for Dynamic Tasks

"First return, then explore" anpassad och utvärderad för dynamiska uppgifter

Nicolas Petrisi, Fredrik Sjöström

”First return, then explore" Adapted and Evaluated for Dynamic Tasks

(Adaptations for Dynamic Starting Positions in a Maze Environment)

Nicolas Petrisi
ni1753pe-s@student.lu.se

Fredrik Sjöström
fr8272sj-s@student.lu.se

July 8, 2022

Master’s thesis work carried out at
the Department of Computer Science, Lund University.

Supervisors: Volker Krueger, volker.krueger@cs.lth.se
Hampus Åström, hampus.astrom@cs.lth.se

Examiner: Elin A. Topp, elin_anna.topp@cs.lth.se

Abstract

In this thesis, we build upon the policy-based Go-Explore algorithm. It uses a special cell concept to link multiple states as a single cell and utilizes trajectories and self-imitation learning to achieve state-of-the-art performance. However, Go-Explore has limitations for dynamic starting positions. By changing how the algorithm defines the cell representation of its environment and how trajectories are represented, Go-Explore is adapted to work on dynamic tasks with the aim to make the algorithm even more applicable in other domains, such as robotics.

When trying to solve a maze with random starting positions, the newly adapted version performs more than seven times better than the original version, managing to reach the goal of the maze over 98% of the time. This shows that the adaptations enhance the performance of Go-Explore for dynamic starting positions which in turn opens a whole new class of problems for Go-Explore to tackle.

The code is available at: https://github.com/NicolasPetrisi/go-explore

Keywords: Go-Explore, exploration, maze, machine learning, reinforcement learning, Procgen

Acknowledgements

We, Nicolas Petrisi and Fredrik Sjöström, would like to thank Hampus Åström and Volker Krueger for support and discussion sessions as supervisors during the length of the entire project with additional thanks to Volker for supplying hardware well needed. We would also like to thank LTH for supplying this thesis with a location to work and additional hardware.

Contents

Chapter 1
Introduction

In this initial chapter, the context of the work is introduced in section 1.1, followed by the research questions and goals in section 1.2, methodology in section 1.3, contribution summary 1.4 distribution of work 1.5, and finally an outline of the report 1.6

1.1 Introduction

Reinforcement Learning (RL) is an interesting subfield of Machine Learning (ML) in which an agent interacts with an environment and gets rewards depending on what actions are performed in which states. The final goal for the agent is to maximize the sum of these rewards by training a policy that chooses the best action to take depending on the environment's current state. RL is used in a broad field with numerous application possibilities, from vehicle routing [1] to effective control systems of liquid level in tanks [2], games [3], and more.

In the domain of games, the collection of Atari games provided by OpenAI Gym [4] has a new state-of-the-art (SOTA) algorithm on most of its games that is called robustified Go-Explore [5]. A second, more RL-oriented version, called policy-based Go-Explore [6], also beat SOTA on the two notorious hard-exploration Atari games Pitfall and Montezuma's Revenge. The OpenAI Gym contains an environment for, amongst other things, a collection of Atari games, of which a list of the games can be found in [7]. An Atari game is a pixelated game generally in 2D typically made in the 1980s where the game can be everything from exploring in a maze-like environment while avoiding enemies [8] to racing games [9], shooter games [10], and more.

When the article that presented Go-Explore [6] was written, the gym contained a collection of 55 Atari games where it scored higher than previous SOTA in 85.5% of the games as well as above human performance in all of them. These results are interesting to more than just the gaming world, as learning to beat games can be seen as an evaluation of how well the algorithm can learn different skills, something that can be transferred to other areas. Go-Explore is also tested in a robot simulator and proves to yield good results when trained to

put a cup from a fixed location into a shelf, even if the shelf had a latched door that needs to be opened first [6], proving that the algorithm can be applied to more areas than just games.

However, the Atari games are often deterministic, which does not map well to the dynamic real world. To reduce this drawback, no-ops and sticky actions, which are explained in section 2.2, are recommended to be added to the environment [11] to make it more stochastic. Even with these additions present, the structure of each episode is still similar and the states for games using transition screens between rooms and levels, such as Montezuma's Revenge [8] and Pitfall [12], are still nearly identical. Further, the path to the goal in the Atari games is always in broad terms the same between every episode since neither the start nor goal changes, given that there is a goal location such as in an exploration game. Even with sticky actions present, the overall path is still the same, it is just impossible to predict an exact sequence of actions. The robotics simulation on which Go-Explore is tested also suffers from non-dynamic starting positions as it resets to the same starting position on every episode which is not true for many real-world cases.

In this thesis, we present a task that is more dynamic than the ones Go-Explore is previously tested on and introduce new adaptations for Go-Explore to be better suited for this task. We evaluate how well Go-Explore solves this dynamic task with the adaptations present, while also comparing it to how well it manages the task without them.

1.2 Research Questions and Goals

The main purpose of this thesis is to answer the following question: How can Go-Explore be adapted to perform better for a dynamic task? More specifically, performing a task using random starting positions. The task used to evaluate Go-explore in this thesis is the task of solving a maze where the start position is random for every attempt. The performance is measured by the success rate of solving the maze as well as how fast it does so. To fully answer this question, a set of sub-questions are defined below and are answered throughout this paper.

1. How do Go-Explore "off-the-shelf" and our adaptation of Go-Explore perform against each other in an environment with dynamic starting positions?
2. How do training time and performance scale against the size of the environment?
3. How do different exploration strategies affect training time and performance?
4. How can a good representation of the environment be defined when a predefined representation does not fit?

In addition to answering the questions above, the goals set for the project to complete are the following:

- Modify the Go-Explore algorithm to work with dynamic starting positions.
- Provide a working, well-documented code, for future use.
- Evaluate the adapted version of the code on Maze from the Procgen benchmark by OpenAI by modifying Procgen to use dynamic starting positions. Further, edit and optimize the code for the chosen game type.

1.3 Methodology

The methodology used in this thesis is to take an artifact, Go-Explore, modify it to fit a dynamic task, and evaluate the performance through experiments.

To answer the questions and achieve the goals presented in section 1.2, the policy-based version of Go-Explore, published in "First return, then explore" [6], is modified iteratively and adapted to the environment of Maze. Through training multiple agents on different setups in the chosen environment, the performance of the resulting program is evaluated. This evaluation is based on the average number of successful episodes where the maze is solved when starting on random positions as well as the average number of steps required. This is then compared against the same metric for "off-the-shelf" policy-based Go-Explore.

1.4 Contribution Summary

This thesis contributes to the scientific community with the following three items.

- Give data about how well Go-Explore performs in the maze environment with dynamic starting positions to help determine the usefulness of the algorithm for different cases.
- Expand Go-Explore to handle different cases to make it possible to use for more dynamic tasks, for instance in robotics.
- Providing a working and well-documented code for future work to expand upon. Link to our GitHub repository: https://github.com/NicolasPetrisi/go-explore

1.5 Distribution of work

Much of the code has been done with the practice of pair programming. Some areas that were focused on by a single person however are:

Fredrik took upon the responsibility of creating an initialization script to ease the installation process. He also had more focus on the modifications regarding Procgen as well as a lengthy attempt of getting the GPU to process the application. In the report, he had more responsibility for the result section.

Nicolas took upon the responsibility of creating the script to execute the program with, adapting the parameters of the program, most of the logic and functionality of on the fly trajectories as well as the Dynamic cells that are explained in section 3.3. In the report, he had more responsibility in the approach section.

1.6 Outline of Report

This thesis starts with chapter 1 which contains an introduction to the thesis, a brief problem description, the research questions, the methodology used, and a contribution statement. It is followed by chapter 2 where the terminology used in this report is defined and the two versions of Go-Explore are described. It also introduces the Procgen environment, which is

used as the base for the dynamic task in this thesis, and lastly, related work is mentioned in this chapter. After that follows chapter 3, which introduces the dynamic task used to evaluate Go-explore and more thoroughly goes through some problems with Go-Explore for the dynamic task and solutions to them. Chapter 4 describes how the experiments are set up and run as well as the results of them. The results are discussed in chapter 5 and finally, chapter 6 discusses some future research possibilities and concludes the report.

Chapter 2
Background

In this chapter, the terminology used in this report is described in section 2.1. Then, Go-Explore and its two different versions are introduced in section 2.2, where the policy-based version of Go-Explore is the one being used in this thesis. Then, the Procgen environment used in training and evaluation is introduced in section 2.3. Finally, related work is mentioned in section 2.4.

2.1 Terminology

Description of the terms used that are necessary to understand the report are presented in this section.

2.1.1 Episode

An episode is all the states, actions, and rewards in sequence from a start state to an end state. For example, if playing a game, all states, actions, and rewards in sequence from the start of the game until the game ends belong to the same episode.

2.1.2 Domain Knowledge

Domain knowledge is information about the state of the game that is not directly available to the player such as which room number the agent is currently in, the discretized x and y position of the agent, in which rooms held items were found, etc. Some, such as the agent's position, is information that is still presented to the player but in another way, like seeing the character on the screen instead of as discretized x and y coordinates.

2.1.3 Policy

A policy π is a function that takes the state of the environment and outputs the probabilities for taking different actions as seen in equation 2.1 where A is the action space and S is the state space. The goal is to find an optimal policy that maximizes the total reward i.e., for every state the optimal policy outputs the action that leads to the highest total reward.

$$\pi(a, s) = P[A = a|S = s] \tag{2.1}$$

A policy can also be used to reach specific goals in an environment and is then called a goal-conditioned policy. For this, a tuple of a goal you want to reach and the current state is used as input to the goal-conditioned policy function instead as seen in equation 2.2 where A is the action space, S is the state space and G is the goal space. Here, the goal is to find an optimal policy that maximizes the total reward while at the same time navigating to the provided goal state.

$$\pi(a, (s, g)) = P[A = a|S = s, G = g] \tag{2.2}$$

2.1.4 Exploration Strategy

Policy-based Go-Explore has two phases: first, a return phase and then an exploration phase, both of which are explained later in section 2.2.2. During the exploration phase of the algorithm, a choice is made between performing random exploration or policy exploration as described below. Note that during either type of exploration, an exploration target, as explained in more detail in section 2.2.2, is selected to explore towards.

Random Exploration

If random exploration is chosen during the exploration phase, the agent explores using uniformly distributed random actions by overwriting the action chosen by the policy whilst keeping the goal of reaching the chosen exploration target.

Policy Exploration

If policy exploration is chosen during the exploration phase, the agent follows the goal-conditioned policy to try to reach the chosen exploration target.

2.2 Go-Explore

Go-Explore (published as “First return, then explore” [6]) is a family of algorithms that saves interesting states in an archive, returns to them, and explores from there [6] using a combination of RL and tree search. This is meant to solve two problems occurring during exploration in RL: detachment and derailment. Detachment is when the algorithm loses track of previously visited interesting states to explore from, and derailment is when the exploring part of the algorithm prevents it from returning to previously visited states [6].

Go-Explore is tested on OpenAI's implementation of the Atari Learning Environments (ALE) [6], where ALE is a collection of old games for the Atari 2600, a gaming console made in 1977 by Atari, Inc., which has numerous games in different genres such as puzzle games, shooters, sports, and more [13]. The games are less complex than most games produced today in terms of graphics, controls, and general complexity but can still provide a challenge [13]. OpenAI's implementation contains, at the time of Go-Explore's creation, 55 games in which Go-Explore outperforms the state of the art in all the hard-exploration games [6]. Notably, it is the first to receive a higher score than the best human in the game Pitfall. Previous SOTA algorithms do not manage to score a single point in this game. It achieves these results through efficient exploration, by remembering interesting states found in the environment and exploring from them. This technique solves both derailment and detachment, which as described in the previous paragraph, other RL algorithms have problems with.

However, these games are deterministic [11], therefore an agent can memorize a sequence of actions that solve the game well instead of learning a broader skill. To combat this, two stochastic additions are added: no-ops and sticky actions. No-ops means that the agent does nothing for a random number of frames at the beginning of an episode, causing the environment to move and be different depending on the number of frames waited, resulting in slightly different starting states. Sticky actions try to mimic the limitation of human precision and give a 25% chance of every step to ignore the action presented by the agent and do the previous one instead. This makes the agent perform more like a human who can by accident go a little too far to the side than intended, making an exact series of actions required to solve a level impossible to predict and therefore forcing the agent to learn a more robust policy that can handle these missteps.

Two versions of Go-Explore have been made [6]. The first version is divided into two parts, an exploration part and a robustification part. This version has the requirement of states being restorable and is in this thesis called "robustified Go-Explore" and is described in section 2.2.1 for the sake of completeness. The second version, which is the version used in this thesis, is built upon the robustified version but uses a return policy that eliminates the requirement of states being restorable as well as the need for the robustification step. This version is called "policy-based Go-Explore" and is more flexible for changes in the environment and is described more in section 2.2.2.

2.2.1 Robustified Go-Explore

Robustified Go-Explore is the most efficient variant as it takes the least amount of time to train but has the requirement that visited states must be restorable, e.g., a simulator that resets to a selected state is needed. The algorithm has two major parts, the first part is centered around exploring and the second is a robustification step where the best trajectories found in the exploration step are used to find a robust policy for the agent [6].

The exploration phase can be seen in figure 2.1. A promising state is first probabilistically selected from the archive based on its weight. The weight of a state (or cell, as explained later in this section) is based on the number of actions performed in the state where fewer actions mean a more promising state. Then, the simulator is reset to that particular state and from this state, the method explores using random actions for a fixed number of steps or until the episode terminates such as when winning the game, dying, or running out of time. When done exploring, all new states found while exploring are added to the archive, and the

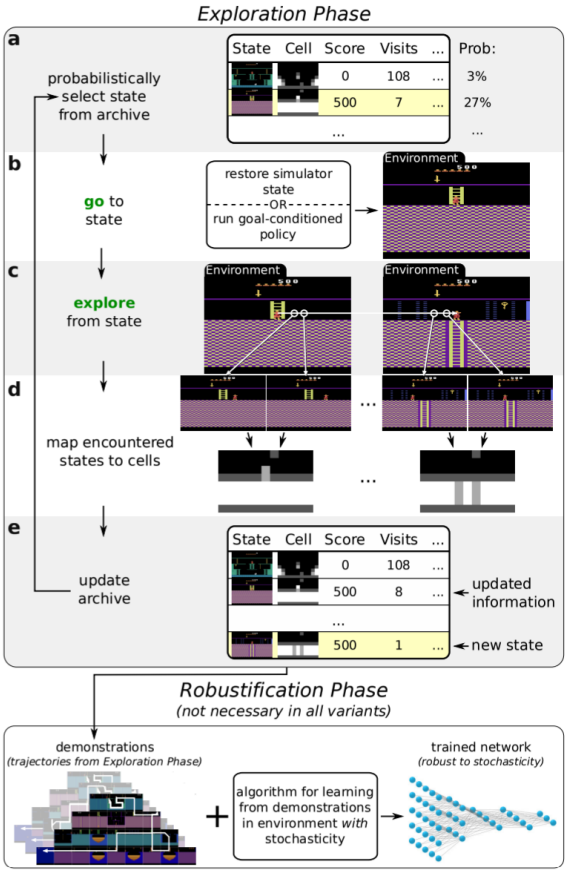


Figure 2.1: The Idea behind Go-Explore. It finds trajectories by performing steps **a)** to **e)** iteratively and uses the best ones in the robustification step to train a policy. See section 2.2.1 for details.

algorithm repeats. The exploration phase makes use of a cell concept and trajectories, which are described in detail in the paragraphs below.

In large environments, the number of states can become too high to keep track of efficiently. To solve this, a state-to-cell mapping is used where similar states are grouped into cells, and, instead of saving all states, only the best state for every cell is saved, i.e., a single cell represents a collection of states. Two techniques for this cell mapping are presented in [6]. The first technique downscales the game image and maps all states with the same downscaled game image to the same cell. The second technique makes use of domain knowledge and maps all states with the same domain knowledge to the same cell. Using domain knowledge for the cells is game-specific but shows to yield better scores [6].

As the algorithm explores the environment it keeps the path of cells it has moved through in memory, which is referred to as the trajectory. As it reaches different cells it adds the information of the trajectory to the cell if the current trajectory is better than the stored one in the cell. Values being compared are, among other things, the score obtained so far and the length of the trajectory, where shorter is better. In addition to this trajectory, a full trajectory is also saved which contains not only the path of cells, but also every action taken, reward received, and states observed during the entire trajectory.

When the exploration phase is done, the highest-scoring full trajectories are picked, and the algorithm moves on to the robustification phase. To train a policy for the agent from

these trajectories a technique called Learning from demonstration (LfD) is used. LfD uses trajectories and tries to find a policy that approximately follows these trajectories and thereby gives an agent with a good policy [14]. This entire process is possible by saving every action taken, reward received, observation, and state for each step in the full trajectory so they can be replayed as they were for the network in every detail.

This version is not being used in this thesis, but it is the foundation that policy-based Go-Explore expands upon, which is described in the section below.

2.2.2 Policy-based Go-Explore

Instead of assuming that states are restorable, as in robustified Go-Explore in section 2.2.1, a goal-conditioned policy can be trained to navigate back to a given cell [6]. The goal-conditioned policy is implemented as a neural network, similar, but not identical, to the one presented in figure B.1 in appendix B.1. This makes the returning part more expensive since a policy must now be trained and used instead of simply restoring the simulator to the state. However, the domain of problems applicable expands to tasks where restoring a certain state is impossible or misleading, such as when randomness makes it nearly impossible to return to a specific state by stepping in the environment.

Policy-based Go-Explore as presented in [6] is implemented using domain knowledge as the cell representation.

First Return

For policy-based Go-Explore, the algorithm is slightly different from the robustified version. First, the agent attempts to return to a previously visited cell by using the policy learned so far and navigating there while following a previously found trajectory instead of restoring to the specific state. The goal cell is probabilistically chosen based on the weights of the cells, where the weight of a cell approximates how well the cell and its surroundings are explored and is calculated according to equation 2.3 where C_{steps} is the number of actions taken in the cell.

$$weight = \frac{1}{1 + 0.5C_{steps}} \tag{2.3}$$

The algorithm then extracts the trajectory from the chosen cell and attempts to follow it. This is done by feeding the next cell in the trajectory to the goal-conditioned policy as the goal which in turn should output actions that lead to the given cell, leading the agent towards the return goal, cell by cell.

When following a trajectory, the agent gets a reward of 1 every time it successfully reaches the next cell in the trajectory and an additional 2 for the final cell. In comparison, the agent gets a reward equal to the score gained clipped to -2 to 2 from the environment. These are the driving factors that make the agent learn to follow trajectories and what it wants to seek out or avoid in the environment.

If it successfully reaches the cell chosen as the return target, the algorithm continues to the exploration phase.

Then Explore

In the exploration phase, the algorithm chooses one out of two exploration strategies: policy exploration or random exploration, as introduced in section 2.1, determined by a predefined probability. The agent then executes the selected exploration strategy for a set number of steps or until the environment terminates, whichever happens first. The goal-conditioned policy is trained during the exploration phase as well where the exploration target, which is described more below, is set as the goal. Even if the actions taken during random exploration are overwritten with a random one, the policy is then still trained *as if* it had chosen that random action as output.

During either type of exploration, a cell category of "unknown neighboring cell", "neighboring cell" or "random cell" is chosen based on a set of probabilities as seen below in table 2.1, as presented in [6]. "Unknown neighboring cell" includes any cells that are neighboring the current cell concerning its x and y coordinates and does not already exist in the archive, i.e., it has not been found yet. "Neighboring cell" includes any known or unknown cells that are neighboring the current cell concerning its x and y coordinates. "Random cell" includes any potential cell anywhere in the environment, including both known cells as well as states not yet discovered.

Table 2.1: The probability distributions when choosing a category for the exploration target during the exploration phase. See page 16 for details.

Choice	Probability
Unknown neighboring cell	0.1
Neighboring cell	0.15
Random cell	0.75

From the randomly chosen category, a random cell is chosen as the exploration target based on its weights the same way as a return target is chosen, by using equation 2.3. If there are no valid cells to choose from in the chosen category, such as the current cell has no unknown neighbors when "unknown neighboring cell" is chosen, the choice of category falls back to the next in order of appearance in table 3.1 from top to bottom to guarantee that the chosen category has at least one valid cell to choose as the exploration target.

The chosen exploration target is fed to the policy as it attempts to reach this cell. This differs from the return phase where the next cell in the trajectory leading up to the target is fed to the policy; during exploration, there are no sub-goals to follow to the final target. If the agent reaches this exploration target before the end of the episode, another target is chosen in the same way, repeating until the episode is terminated by the environment or after a set number of steps. After the episode is complete, the environment resets to the starting state and the algorithm selects a new return goal as before.

Policy exploration can improve exploration efficiency as it can learn how to overcome obstacles when returning and use this experience when exploring as well. Hence it can make it easier to get past an obstacle that is similar to one encountered before compared to only taking random actions when trying to get past it every time. It has been shown that policy exploration can explore better than random exploration in different environments, according to previous experiments [6].

The robustification step that comes after the exploration phase in robustified Go-Explore is not needed in this version as the agent already learns a policy during the return and exploration steps. A similar technique named Self Imitation Learning (SIL) is however still applied in parallel during training to enhance the learning phase. With SIL, a full trajectory, as described in section 2.2.1, is chosen probabilistically from the archive just like in step **(a)** in figure 2.1. The chosen trajectory is then replayed by the policy (without stepping in the environment) to train on following it using SIL and in principle robustifying the trajectory.

With the policy-based version of Go-Explore, the agent can explore directly in a dynamic context and potentially manage things not explored in the experiments of the article "First return, then explore" [6], such as Atari-like games in dynamic environments or tasks. This further expands the domain of applicable tasks for Go-Explore. There are however still limitations to the algorithm that prevent this, some of which can be solved using the approach presented in section 3.3. Expanding policy-based Go-Explore for dynamic contexts is suggested for future research by Ecoffet et al. [6] and is therefore the foundation upon which this thesis is built.

2.3 Procgen

In this thesis, a modified version of the Procgen environment is used. Procgen, which is developed by OpenAI[15], is an Atari-inspired game framework with procedurally generated levels for 16 different games. The game levels are random, making memorization infeasible as a solution. Instead, the agent is forced to learn the appropriate strategy associated with the game in question. The framework is designed for training reinforcement learning agents and is built upon OpenAI's gym [16]. The high diversity of levels within the different games maximizes the need for policy generalization. Fast evaluation of the game environments provides a possibility to train an agent with great speed, and with a tunable difficulty of the game levels from easy to hard, the computational power required can be adapted to the available resources.

Of the 16 games available in the Procgen suite, one is called "Maze" and is chosen as the game to test Go-Explore on. In Maze, the agent acts the role of a mouse in a maze trying to find its way to the cheese before the limit of 500 steps is reached. The game ends either when the cheese is found, or the 500 steps are exhausted. See appendix A for images of the game. The agent always starts at the bottom left corner of a new random maze when starting a new game, but this is modified to a random starting position in the same maze in this thesis as explained later in section 3.2.

2.4 Related Work

At the time of writing, we found no published work that expands directly upon policy-based Go-Explore. Policy-based Go-Explore is first published in 2020 in "First return, then explore" [6], this is not to be confused with "Go-Explore: a New Approach for Hard-Exploration Problems" [5] from 2019 which only introduces robustified Go-Explore.

There are, however, some papers that criticize "First return, then explore" by Ecoffet et al. J. Weng accuses "First return, then explore" of post-selection, training multiple networks

and only presenting the best one [17], [18]. J. Weng also accuses "First return, then explore" of lack of transparency [18].

H. Xu mentions the long training time and hardware requirements of Go-Explore and criticizes that it cannot handle the game Pitfall without game-specific domain knowledge [19]. The author also questions Go-Explore's usability outside of deterministic environments, something this thesis aims to explore if this can be remedied.

With these critiques in mind, Go-Explore is still deemed interesting enough to proceed with as the foundation of this thesis.

Additionally, T. Zhang et al. solve the same problem that Go-Explore aims to solve using a similar algorithm called BeBold [20]. BeBold aims to solve the same task by exploring beyond the boundary of explored regions. The agent only receives rewards from the algorithm itself, i.e., intrinsic rewards, when it pushes this boundary forward, hence motivating exploration. This results in the explored boundary being pushed uniformly as opposed to Go-Explore.

But in initial tests for BeBold on Montezuma's Revenge, the algorithm converges to a score between 10.000 and 13.000 at 2e9 steps compared to Go-Explore with a score of approximately 20.000 to 40.000 and still increasing after 2e7 steps [6]. Note that these are only initial results for BeBold but give an insight into the potential difference in performance between the two algorithms. However, it should be noted that the paper is a preprint and viewed with that in mind.

Chapter 3
Approach

In the following chapter, the environment with dynamic starting positions used for evaluation is presented in section 3.1 and core problems that prevent Go-Explore from being used with dynamic starting positions are addressed in section 3.2. Finally, our larger changes to the code and our solutions for the problem addressed are presented in section 3.3, where Dynamic cells and on the fly trajectories are explained.

3.1 Dynamic Task

The environment used to evaluate Go-Explore is a modified version of the Procgen game Maze, described in section 2.3. The game logic is the same as described in section 2.3 where the goal is for the agent to navigate through a maze and reach the cheese within 500 steps and the game ends when either the cheese is found, or the 500 steps are exhausted. However, after our alterations, instead of having a fixed starting position and different mazes, different starting positions and a fixed maze is instead used. This results in a fixed environment, but a dynamic task of finding the path to the cheese as it will be different in every attempt. This makes returning to previously visited states possible, as is required by Go-Explore, but along different paths. Three different mazes are used in this thesis and are illustrated in appendix A.

3.2 Problems for Go-Explore

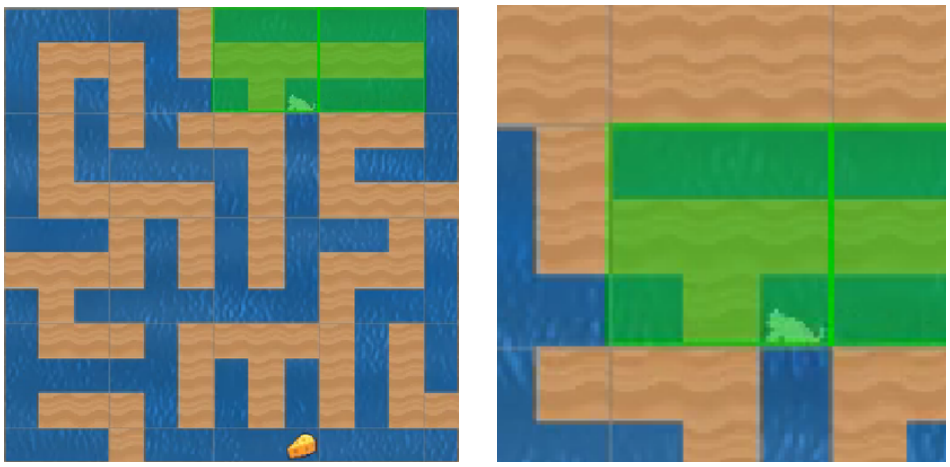
To train on the environment presented in section 3.1 there exists two major problems for Go-Explore. The cell representation described in 2.2.1 has problems with the narrow corridors of the environment, and the trajectories described in 2.2 only work when starting in the same starting position. The problems are further described in sections 3.2.1 and 3.2.2, respectively.

3.2.1 Cell Representation

Go-Explore uses a system of cells, as described in section 2.2.1 where several similar states map to a single cell by a cell mapping, reducing the amount of data needed to be stored. For this report, we say that states belonging to a cell are similar if the agent can reach all states within the cell without leaving said cell.

Two techniques to create a cell mapping are proposed in [6]: downsampling images or using domain knowledge, as mentioned in section 2.2. Policy-based Go-Explore is created by Ecoffet et al. only using the domain knowledge solution, where the relevant domain knowledge present for the maze environment is only the discretized x and y position of the agent. In the original code, the discretized position is found by putting a grid on the screen and letting all states that are in the same box in this grid belong to the same cell as can be seen in figure 3.1a.

Problems arise when dividing the screen into a grid, as selecting the size of the cells is problematic; choosing too small cells, going from one cell to the next in a trajectory is now trivial and the number of cells would be close to the number of states, but too large causes the same cell to represent vastly different states in the environment as can be seen in figure 3.1. If the mouse would be in the bottom left, bottom right, or top of the green cell centered in figure 3.1b, the states would be mapped to the same cell even though neither is similar as seen in figure 3.1a. This contradicts the assumption of states in the same cell being similar according to the definition presented earlier in this section. A solution to this problem is introduced in section 3.3.1.



(a) (b)

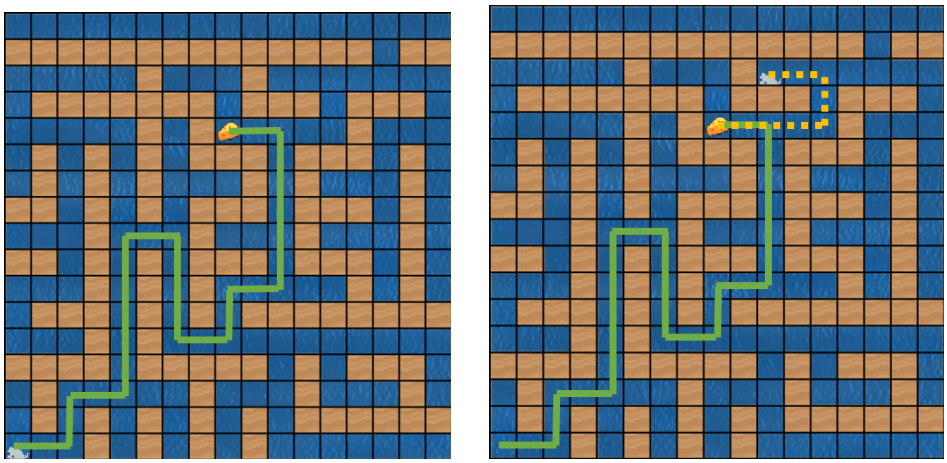
Figure 3.1: Example of a problematic cell representation when using a too wide grid for grid-based cells. See section 3.2.1 for details.

3.2.2 Trajectories

A further limitation of Go-Explore that must be solved is the requirement of a fixed starting position that comes with the deterministic nature of the Atari suite ALE always starting in the same position. As a result, Go-Explore is written in a way that assumes that the starting

position for the agent is always the same for every episode which causes every trajectory to begin from the same position.

As an example, given the initial starting position 'A', if the agent later starts at point 'B', the instructions will be faulty since they assume the agent is at position 'A' which makes the trajectory wrong, misleading, or in some cases even impossible to follow to get to the destination. An example of this can be seen in figure 3.2.



(a) (b)

Figure 3.2: The limitations of the original trajectories. If the agent starts as in figure 3.2a the green trajectory helps reach the goal, but if it starts as in figure 3.2b, the trajectory does not help the agent to reach the goal. See section 3.2.2 for details.

Simply training the agent on multiple different starting locations in a series does not show promising results as the agent seems to forget the previous starting locations as they change.

Training the agent on random starting positions is not feasible either, because when the agent is training it will find a path to the goal cell of the maze, i.e., the cell that contains the cheese, from one position and save this path as the trajectory leading there. Then, once it finds a path from *another* location, the previous path will be overwritten if the new location is initially closer than the previous and the resulting trajectory is, in turn, shorter, just like in figure 3.2b. As the only state with reward from the environment in the maze is the goal state at the cheese, the only metric for the trajectories that matters is their length when comparing against each other. This causes the trajectories leading to any cell to constantly switch between vastly different trajectories until they converge to trajectories that through random chance start at the position exactly next to the cell they go to, since no other path will ever be able to be shorter than this. This will effectively render the trajectories useless and in many cases even cause them to work against the agent rather than assisting when providing a path to follow in the wrong direction relative to the agent's position. One is thus forced to use a fixed starting position for Go-Explore trajectories.

To fix this limitation, the concept of On The Fly (OTF) trajectories is created as described in section 3.3.2.

3.3 Adaptations

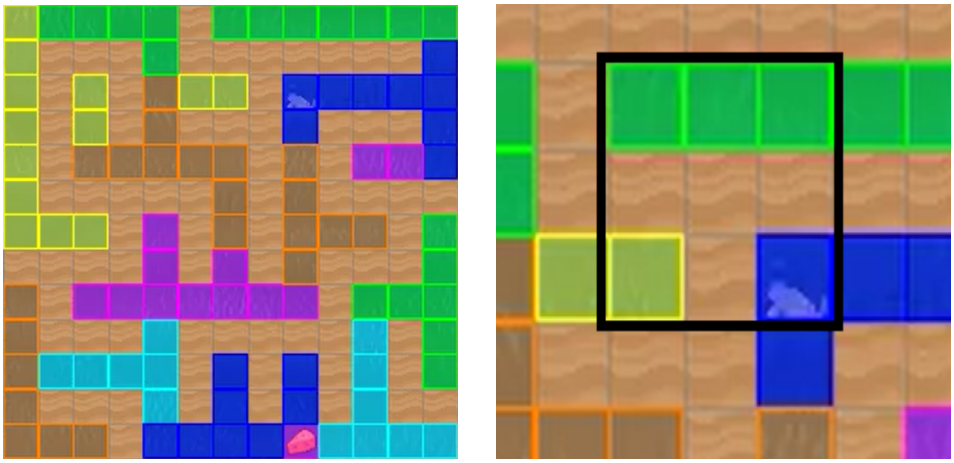
In response to the problems formulated in section 3.2, we introduce the concepts of Dynamic cells and on the fly trajectories (OTF-trajectories) as described in sections 3.3.1 and 3.3.2. Additionally, the code of policy-based Go-Explore is further adapted to fit the environment of Procgen and to be able to handle these two new additions to the code as described in section 3.3.3.

The version of the algorithm that uses Dynamic cells and OTF-trajectories is referred to as adapted Go-Explore in the rest of this report, while the version of the algorithm that does not make use of them is referred to as the original Go-Explore.

3.3.1 Dynamic Cells

An approach to solving the cell-size problem presented in section 3.2.1 is to first start with a small grid that only allows a single state in each square of the grid. Then, for every state, find which states are reachable from that state and vice versa with no intermediate states. If one state can directly reach another, they are said to be neighbors in that direction. After every neighbor has been found, neighboring state are then merged to form cells given that the neighboring status goes both ways, i.e. (A $\rightarrow$ B) as well as (B $\rightarrow$ A). Note that in the case of the game Maze, neighbor status in one direction is assumed to imply neighbor status in the other direction as well, but this is not necessarily true for other environments. Cells created this way are called "Dynamic cells" in this report and when states have been mapped to Dynamic cells the first time, the same strategy can be applied again where neighboring Dynamic cells now merge with other Dynamic cells. The process can be repeated until the cells are of a desired maximum size. This cell representation allows larger cells where all states within the same cell are similar and close to each other.

An example of Dynamic cells can be seen in figure 3.3 which is compared to the original problematic cell representation in figure 3.1. In figure 3.3a, the small squares represent the states the agent can be in, and all squares that share color and are connected represent a single Dynamic cell. In this thesis, the max size of the Dynamic cells is set to 10, meaning a single Dynamic cell may not be larger than a total of 10 states merged. The states belonging to the problematic cell highlighted in figure 3.1b belong, with Dynamic cells, to three different cells, as seen in figure 3.3b where the corresponding area is marked with a black square. This is a better cell division since the states belonging to the same cell in figure 3.1 are not similar according to the definition given in section 3.2.1, but the Dynamic cells fulfill this criterion.



(a) (b)

Figure 3.3: An example of cells in the same maze as in figure 3.1 when using Dynamic cells instead. See section 3.3.1 for details.

During the cell selection process in the exploration phase, the cell categories presented in table 2.1 and discussed in its corresponding section 2.2.2 are slightly changed. This is because it is no longer possible to find a neighbor to a dynamic cell by simply adding ± 1 to the coordinates since they have irregular shapes, as can be seen in figure 3.3. This prevents us from selecting unknown neighboring states in an efficient way.

Instead, during either type of exploration, a cell category of "neighboring cell", "random known cell", "random cell" or "goal cell" is chosen based on a set of probabilities as seen below in table 3.1. "Neighboring cell" includes any known neighbor to the current cell. "Random known cell" includes any cell that has been saved in the archive. "Random cell" includes any potential cell, including both known cells and states not yet discovered. Finally, the "Goal cell" category includes only the cell which contains the goal of the environment.

Table 3.1: The probability distributions when choosing a category for the exploration target during the exploration phase. See section 3.3.1 for details.

Choice	Probability
Neighboring cell	0.5
Random known cell	0.2
Random cell	0.2
Goal cell	0.1

The probabilities in table 3.1 above are changed to these values and categories to emphasize on learning how to navigate between the cells to compensate for SIL not working, as is explained later in section 3.3.2, whilst still exploring for new paths and states in the environment.

If there are no valid cells to choose from in the chosen category the choice of category falls back to "random cell" to guarantee that there is at least one valid cell to choose as the exploration target.

This can happen if the current cell does not have any known neighbors when the "neighboring cell" category is chosen, or the goal cell of the environment has not been found when the "goal cell" category is chosen. It is worth noting that for the Maze environment, "random cell" is chosen among all states in the maze that is *not* a wall segment. This means that the cell/state chosen will always be valid. This is done by looking at the color coding for wall segments in the frames from the observation of the environment.

3.3.2 On the Fly Trajectories

Policy-based Go-Explore creates trajectories to follow when returning to an interesting state and when performing SIL training, as explained in section 2.2.2. A trajectory is a sequence of cells to traverse through to get to the chosen target. For Go-Explore, the starting cell is always assumed to be the same cell across every trajectory, i.e., the cell containing the starting state for the environment. Trajectories are compared against each other regarding, amongst many other things, the length of the trajectory, to only save the best trajectories found. These trajectories are then used to navigate with, and as explained in section 2.2.1, the corresponding full trajectories are used to perform SIL with. But as this does not work when different starting locations are being used, two options are identified:

(a) No longer rely on saving the trajectories and instead create new ones on the fly.

(b) Take both the end cell *as well as* the starting cell into account when selecting a trajectory to be able to differentiate between different starting positions.

Of these two, option **(a)** is chosen since it is deemed easier to implement given the state of the code and as option **(b)** would increase the number of trajectories saved from N to N^2, which we want to avoid as it would be very memory inefficient.

Instead of saving trajectories, neighbors to cells are saved when moving from one cell to another. This enables a breadth-first search across the neighbors of the cells saved in the archive to find a trajectory of cells on the fly to a specific target when starting a return sequence. This results in On The Fly (OTF) trajectories that can replace the original trajectories of Go-Explore without the limitation of requiring a fixed starting position.

A problem can arise where there is no path of connected neighboring cells to the chosen target, such that the breadth-first is unable to find a path. This is a problem that only occurs with dynamic starting positions. To handle this case, a new target among the cells found in the search is probabilistically chosen instead as the new return target and an OTF-trajectory to this cell is created.

However, OTF-trajectories are not compatible with SIL, as explained in 2.2.2, at the time of writing since SIL requires well-performing full trajectories, as explained in section 2.2.1, to be saved for it to replay for the network to train on, and OTF-trajectories do not save any trajectories. If trajectories are saved, an evaluation metric is needed to evaluate the trajectory such that only the best performing trajectories are saved as is done in the original Go-Explore. This could allow SIL to function together with OTF-trajectories and implementing it would be interesting research for future work.

3.3.3 Other Structural Changes to Go-Explore

To have policy-based Go-Explore work with the environment of Procgen, some alterations are made. A cell representation using domain knowledge that matches the structure of Procgen is thus created. Some adaptations are required to the structure of the program that uses this cell representation to have the program match the new environment, such as frame size dimensions and other parameters. This results in, among other things, the network as described in more detail in appendix B.1.

The rewards given to the policy during training presented in section 2.2.2 are also modified since reaching the goal of the maze gives a high game reward but also ends the episode. To make the agent keener on returning rather than just greedily reaching the cheese, the reward for getting to the next goal in the trajectory is changed to 3 from the previous 1 and the reward for reaching the final goal in the trajectory is increased to 6 from 3. The reward from the game when reaching the cheese is still clipped to 2.

To create the Dynamic cells of the desired size quickly, if no new cell is found for a fixed amount of time during the first million frames that the algorithm discovers, the program starts to merge the current cells into larger Dynamic cells. In addition to this, the program performs a set number of merges equally distributed across the planned training time, to give cells found after the first million frames a chance to be merged into other cells. These merges only occur during training.

Finally, to have OTF-trajectories work with the program, extensive alterations are made to the program to isolate the old version of the trajectories, as well as the SIL, from the algorithm. Then, the old trajectories and the SIL can essentially be turned on and off as wished and enable the use of OTF-trajectories.

Chapter 4
Evaluation and Results

This chapter describes the results of this thesis. It starts with a description of how the evaluations are performed, together with the parameter setups and metrics used during testing, in section 4.1. The experiments and corresponding results of the four tests performed are then presented in section 4.2.

4.1 Evaluation

Testing is performed on the modified maze environment presented in section 3.1, where three different mazes, illustrated in appendix A, are used. When training the models, the starting position is random each episode unless otherwise stated and the agent selects cells to return to with the weight equation (equation 2.3) presented in section 2.2.2. The number of frames the models train on depends on the size of the maze, where models that train on the small, medium and large maze trains for 1e7, 2e7, and 2.5e7 frames, respectively.

When testing the trained models, the policy is frozen, meaning that it does not try to improve. The agent starts at a different starting position in every episode and selects the cell with the highest reward as the return target, i.e., the cheese. The starting positions are derived from a random function with the same seed in every run so that every test is done on the same sequence of random starting positions. The tests are played for 500 episodes for every setup.

Four measurements are used to evaluate the agents, two are measured during training and two are measured during testing. During training of the agents, the two measurements used are return success: the fraction of successful returns to a target cell, and exploration success: the fraction of successful exploration i.e., how often it reaches the exploration target in the exploration phase. During testing of the agents, the two measurements used are success rate: the fraction of episodes where the goal of the maze is reached, and mean length: the mean length of all episodes where an episode plays for a maximum of 500 steps, as explained in

section 3.1. The three mazes that are used in training and testing are visualized in appendix A, figure A.1 and are referred to as the small, medium, and large maze.

4.1.1 Parameter Setups

To answer the research questions as well as the main question proposed in section 1.2, different setups of hyperparameters are used during training as listed below. Note that SIL is not present in any setup except for the "Original Go-Explore" because of the reasons explained in section 3.3.2.

- **Random Exploration**: This setup is trained on random starting positions using only random exploration during the exploration phase. It uses Dynamic cells with max size of 10 and OTF-trajectories.
- **Policy Exploration**: This setup is trained on random starting positions using only policy exploration during the exploration phase. It uses Dynamic cells with max size of 10 and OTF-trajectories.
- **Mixed Exploration**: This setup is trained on random starting positions using a probability distribution of 2/3 for policy exploration and 1/3 for random exploration during the exploration phase. It uses Dynamic cells with max size of 10 and OTF-trajectories.
- **Original Go-Explore**: This setup is trained on fixed starting positions without major changes to the original code from "First return, then explore" [6], with no OTF-trajectories or Dynamic cells, and is using a probability distribution of 2/3 for policy exploration and 1/3 for random exploration during the exploration phase.
- **Control**: This is an untrained agent that takes uniformly distributed random actions.

4.1.2 Experiments

Five different experiments are performed to answer the research questions presented in section 1.2. The measurements used in the evaluation are return success, exploration success, success rate and mean length, which are all explained in section 4.1.

The first experiment examines the variance of the results acquired. This is performed to check the legitimacy of the results found in this thesis and it is done by training the same setup, policy exploration, three times on the small maze and examining the standard deviation of the return success, success rate and mean length for the three runs. The results are presented in section 4.2.1.

The second experiment examines what effect the choice of exploration strategy has on training time and performance. This is to find which type of exploration to use for the upcoming experiments as well as answer the third research question, "How do different exploration strategies affect training time and performance?", presented in section 1.2. To do this, three different setups with different exploration strategies: policy, random, and mixed exploration as listed in section 4.1.1, are trained on the small maze, and tested together with a control setup. The training is evaluated based on the return success and the testing is evaluated based on the success rate and mean length metrics. The results can be seen in section 4.2.2

The third experiment compares the training time and performance when training on fixed and dynamic starting positions. This experiment is performed to provide insight to the importance of the training type used, fixed or dynamic position, when the goal is to solve a dynamic task. To do this, two instances of the policy exploration setup are trained on the small maze, one where the starting position is fixed and one where it varies. The training time and test performance is evaluated with the same metrics as the second experiment with the addition of exploration success also being measured during training. The results are shown in section 4.2.3.

The fourth experiment compares the original code of policy-based Go-Explore with no major changes against the adapted version created by us with the new features Dynamic cells and OTF-trajectories. This is done to see how the adaptations to the code have changed the performance of the algorithm when dynamic starting positions are used during testing and provide answers to research questions one and four, "How do Go-Explore "off-the-shelf" and our adaption of Go-Explore perform against each other in an environment with dynamic starting positions?" and "How can a good representation of the environment be defined when a predefined representation does not fit?", presented in section 1.2. To do this, one instance of the original version of Go-Explore and one of the adapted version of Go-Explore are trained on the medium maze. Here, the original Go-Explore is trained on a fixed starting position as is required, as explained in section 3.3, and the adapted version is trained on dynamic starting positions. The same metrics are used as in the second experiment: return success is used to evaluate the training and success rate and mean length metrics are used to evaluate the testing. These results are presented in section 4.2.4.

The fifth and final experiment looks at how the training time of the adapted version of Go-Explore scales against the size of the environment. This is done as a check to see how the resulting program scales against larger tasks and answer the second research question, "How do training time and performance scale against the size of the environment?", presented in section 1.2. To do this, the same setup of Policy Exploration is trained on the small, medium, and large maze, respectively. The training time is evaluated based on the return success and exploration success during training and the performance is tested, together with a control, on the success rate during testing. Metrics used for evaluating the training are return success and exploration success, and the metric used for evaluating the testing is the success rate, but the mean length metric is not used as it is not easily comparable between the runs when the environments are of different sizes. The results from this final experiment can be seen in section 4.2.5.

4.2 Results

The results of the experiments described in section 4.1.2 are presented in this section. The return success and exploration success metrics are presented as graphs to show how they vary during training while the success rate and mean episode length metrics are presented in tables as it is only the final value of the test phase that is of interest.

4.2.1 Standard Deviation

To check the reliability of the results in this thesis, a stability experiment is performed by training three agents under the same conditions. The mean and standard deviation from these three is then calculated and presented in table 4.1 and plotted in figure 4.1.

Deviations of the data during training across all setups and across every experiment are assumed to follow a similar pattern as observed in figure 4.1. Each of the three agents performs very similarly up to 2e6 frames from where two of the three agents approach 1.0 in return success. The third stops improving at 0.8 in return success after 2e6 frames up until the point of 7e6 frames where it approaches 1.0 like the others. The data in said figure is presented to give an estimate of the spread of the data in the experiments throughout this thesis to indicate how accurate the results might be.

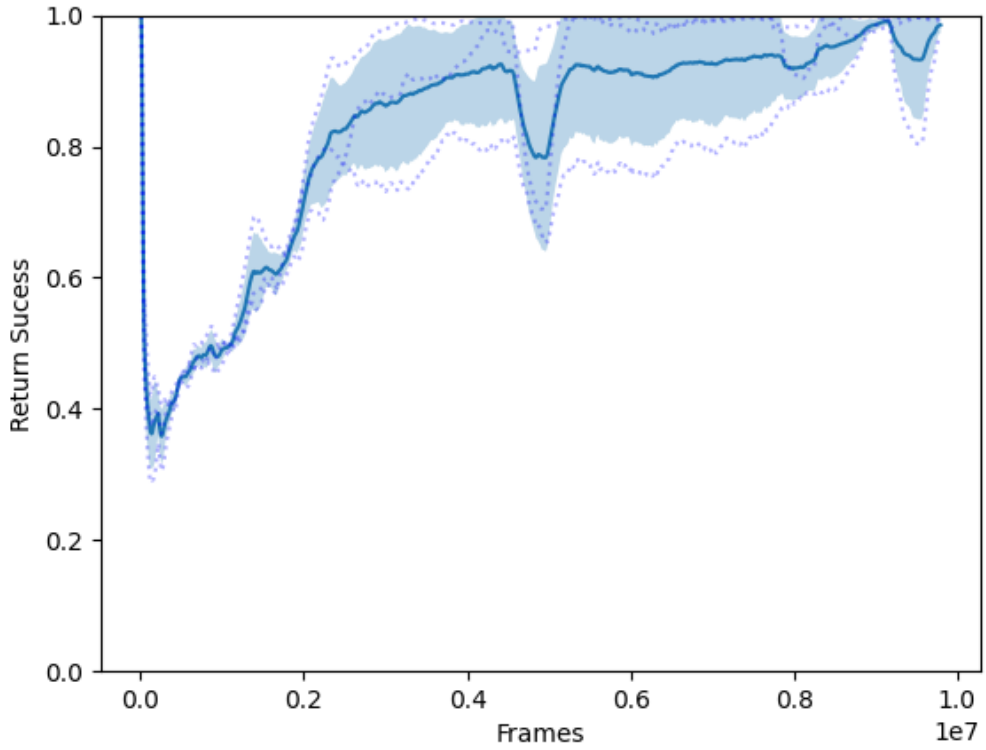


Figure 4.1: The standard deviation of the return success. Moving average (200 frames) of return success over training time measured in frames for three different agents trained using the Policy Exploration setup. The shaded area contains one standard deviation from the mean. The dotted lines are the three training sequences, and the solid line is their mean. See section 4.2.1 for details.

Table 4.1: Standard deviation of the test metrics. Mean and Standard Deviation of the success rate and mean length for three runs of Policy Exploration on the small maze. See section 4.2.1 for details.

Measurement	Mean	Standard Deviation
Success Rate	0.985	0.0462
Mean Length	28.0	5.46

4.2.2 Exploration Strategies

To examine how different exploration strategies affect performance, Random Exploration, Policy Exploration, and Mixed Exploration are tested on the small maze. In addition, a control setup is also tested, as this is the "base case" for the maze, i.e., how well an agent can solve the maze through only random actions. The return success for the training can be seen in figure 4.2 which shows similar training performance for all setups. The test scores for all trained setups are similar, with success rates ranging from 0.980 to 0.988 and mean length from 27.2 to 36.7, see table 4.2 for details. The control shows worse performance with a success rate of 0.250 and a mean length of 400.

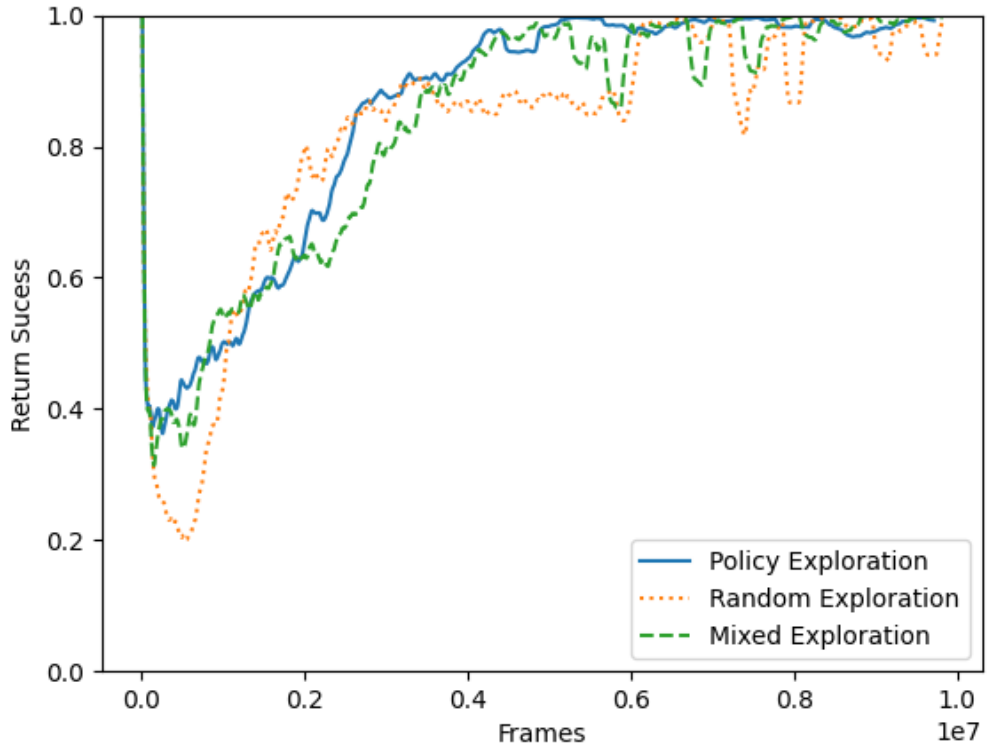


Figure 4.2: Return success for different exploration strategies. Moving average (200 frames) of return success over training time measured in frames for the small maze's Policy Exploration, Random Exploration, and Mixed Exploration. See section 4.2.2 for details.

Table 4.2: Test results for different exploration strategies. The success rate and mean length for different exploration distributions on the small maze. See section 4.2.2 for details.

Setup	Success Rate	Mean Length
Policy Exploration	0.988	30.0
Random Exploration	0.984	27.2
Mixed Exploration	0.980	36.7
Control	0.250	400

4.2.3 Starting Positions

To test how training on a dynamic starting position and training on a fixed starting position compares against each other, two setups of Policy Explorations are trained on the small maze: one with a fixed starting position and one with dynamic starting positions. During training, Fixed Position scores a higher return success faster than Dynamic Position but a lower exploration success as seen in figures 4.3 and 4.4, respectively. During testing, however, Fixed Position shows a substantially worse success rate and mean length than Dynamic Position, but better than the control run, as seen in table 4.3.

Fixed Position reaches 1.0 in return success much faster than Dynamic Position. However, this agent only needs to learn to navigate from one position to every other position in the maze (N paths), while training with a dynamic position the agent must learn how to navigate from every position in the maze to every other position (N^2 paths). Table 4.3 shows that the simpler problem Fixed Position faces does not teach the agent enough to handle the problem with random starting positions during testing.

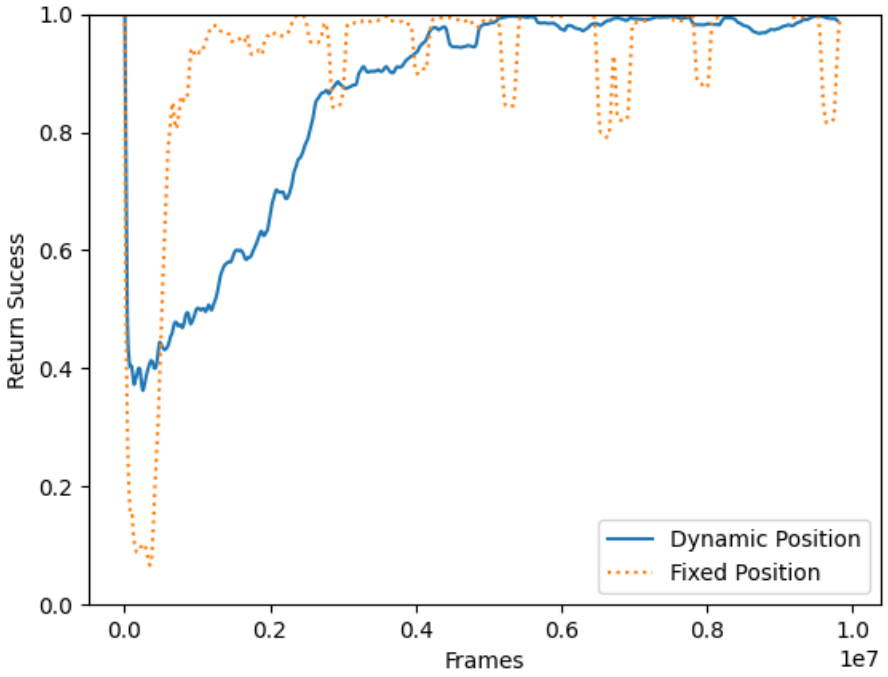


Figure 4.3: Return success for Fixed and Dynamic starting positions. Moving average (200 frames) of return success over training time measured in frames for Policy Exploration trained with either fixed or dynamic position. See section 4.2.3 for details.

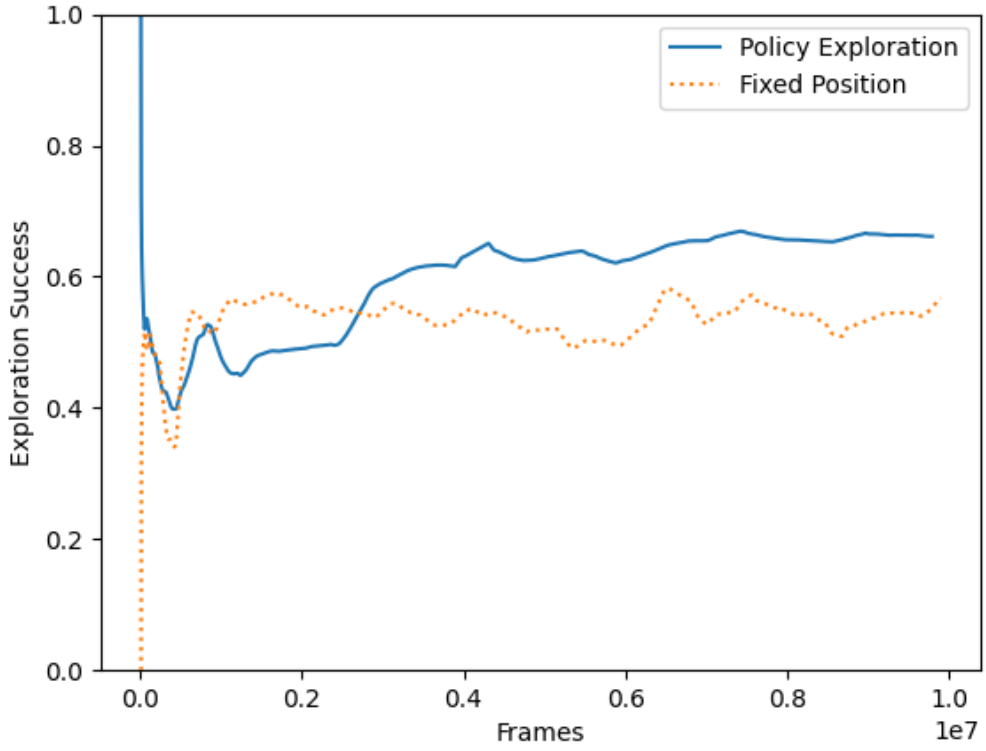


Figure 4.4: Exploration success for Fixed and Dynamic starting positions. Moving average (200 frames) of exploration success over training time measured in frames for Policy Exploration trained with either fixed or dynamic position. See section 4.2.3 for details.

Table 4.3: Test results for Fixed and Dynamic starting positions. The success rate and mean length for Policy Exploration trained with fixed and dynamic starting positions on the small maze. See section 4.2.3 for details.

Setup	Success Rate	Mean Length
Dynamic Position	0.988	30.0
Fixed Position	0.370	354
Control	0.250	400

4.2.4 Original vs Adapted Go-Explore

To see how the changes presented in this paper compare to the original version, two setups are trained: Original Go-Explore and Policy Exploration, where Policy Exploration implements the adaptations presented in section 3.3. The Go-Explore setup has small cells to guarantee that all cells are similar, and the original trajectories are used instead of OTF-trajectories. This forces the setup to be trained on a fixed starting position but allows the usage of SIL.

During training, the return success of Original Go-Explore is similar to Policy Exploration but has a lower return success rate at a higher number of frames, as seen in figure 4.5.

When testing, Policy Exploration outperforms Original Go-Explore with a success rate of 0.982 and a mean length of 36.9 compared to Original Go-Explore's success rate of 0.134 and mean length of 428. This falls in line with the results presented in section 4.2.4 as Original Go-Explore is trained on a fixed starting position as mentioned in section 4.1.

The dips in the return success for Original Go-Explore are hypothesized to be when the algorithm explores down a corridor for a while, but then switches to another corridor; switching to a not-as-explored part of the maze which causes the success rate to dip. Policy Exploration on the other hand is training on random starting positions and is constantly presented with new corridors, giving it a more even increase in success rate.

A control run is again tested, and it performs worse than both the other setups with success rate of 0.090 and mean length of 475, see table 4.4 for the full results.

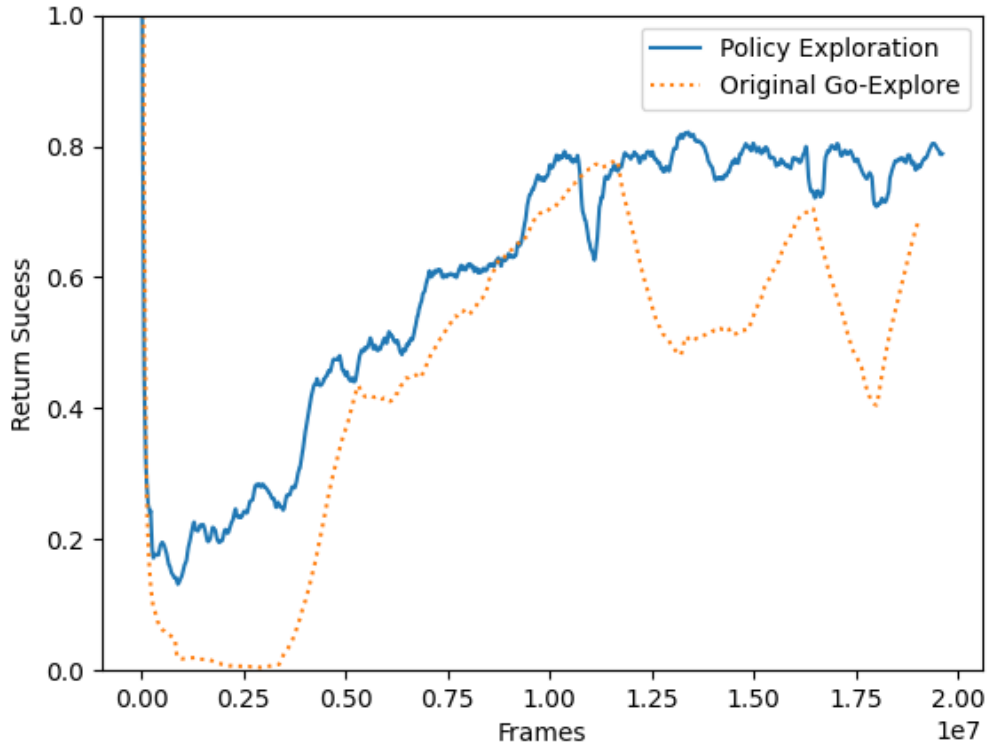


Figure 4.5: Return success for the original and adapted version of Go-Explore. Moving average (200 frames) of return success over training time measured in frames for Policy Exploration and Original Go-Explore on the medium maze. See section 4.2.4 for details.

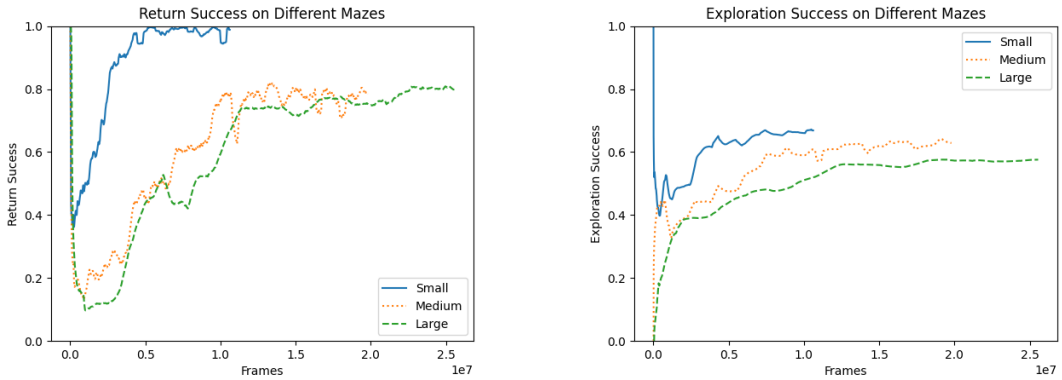
Table 4.4: Test results for the original and adapted version of Go-Explore. The results on the medium maze regarding the success and the length mean of the episodes for Original Go-Explore and Policy Exploration on the medium maze. See section 4.2.4 for details.

Setup	Success Rate	Mean Length
Policy Exploration	0.982	36.9
Original Go-Explore	0.134	428
Control	0.090	475

4.2.5 Scaling with the Environment

To examine how Go-Explore performs with different environment sizes, the Policy Exploration setup is trained on a small, medium, and large maze. The number of frames computed during training for different environments is decided depending on the size of the environments and how the training graphs look but is also held back by the time available for this thesis. The setup trained for 1e7 frames for the small maze, 2e7 for the medium, and 2.5e7 for the large. The return success difference between the small and medium maze is visible, but very small between the medium and large maze, see figure 4.6a. The exploration success is highest for the small maze and has a similar difference between the small and medium maze as between the medium and large mazes, see figure4.6b.

During the testing, the setup scores a high value in the success rate for all the mazes with 0.988, 0.982, and 0.940 for the small, medium, and large mazes, respectively, much higher than the corresponding control runs. See table 4.5 for the full results.



(a) **(b)**

Figure 4.6: Return and exploration success for different environment sizes. Moving average (200 frames) of return success in figure 4.6a and exploration success in figure 4.6b over training time measured in frames for Policy exploration on a small, medium, and large maze. See section 4.2.5 for details.

Table 4.5: Test results for different environment sizes. The success rate for reaching the goal of the maze for Policy Exploration setup and the control setup, on the small, medium, and large maze. See section 4.2.5 for details.

Setup	Success Rate
Small maze	0.988
Control small maze	0.250
Medium maze	0.982
Control medium maze	0.090
Large maze	0.940
Control Large maze	0.092

Chapter 5
Discussion

The following sections discuss and analyze the results from the experiments presented in chapter 4, where each section corresponds to the experiment with the same name.

5.1 Standard Deviation

From the results shown in figure 4.1 and table 4.1 in section 4.2.1, the standard deviation of the return success is moderate in the middle section of the training but otherwise small and the standard deviation of the success rate and mean length metrics are small. This gives an indication of how much the results can vary between individual runs. By knowing this approximated variation of the results, it is possible to give a fairer analysis of the results during the following sections.

All other evaluations are assumed to have a somewhat similar variance in the results if tested multiple times as the results in section 4.2.1.

5.2 Exploration Strategies

When comparing different exploration strategies; Policy Exploration, Mixed Exploration, and Random Exploration all perform similarly in the tests with only minor differences in the success rate and mean length metrics. Policy Exploration reaches the highest success rate of 0.988, while Random Exploration and Mixed Exploration are close to the same results with 0.984 and 0.980 in success rate. All these setups score substantially higher than the control run that has a much lower success rate of 0.250, as seen in table 4.2.

When training with Random Exploration, the agent manages to solve the maze with fewer steps, on average 27.23 steps per attempt, compared to Policy Exploration which need 30.0 steps on average during the evaluation. This means that even though training with Policy Exploration gives a slightly higher success rate, training with Random Exploration makes

the agent solve the maze slightly faster. These are only minor differences and fall within the approximated standard deviation according to table 4.1.

From the training graphs in figure 4.2, small differences can be seen, such as Policy and Mixed Exploration getting close to 1.0 in success rate with fewer frames compared to Random Exploration. This points to Policy and Mixed Exploration learning faster than Random Exploration, but these differences are minor, and no certain conclusions can be drawn from them as it might be by chance. Especially when taking the approximated standard deviation seen in figure 4.1 into account. This indicates that the choice of exploration strategy does not seem to significantly affect the performance in the small maze environment. But it has been shown in previous tests [6] that policy exploration can explore better than random exploration, as described in section 2.2.2.

One possible reason why Policy Exploration or Mixed Exploration does not outperform Random Exploration in this experiment might be that the maze environment does not have any recurring obstacles the agent must learn to overcome, such as enemies moving around in the environment or hazards one must navigate around. In such scenarios, Policy Exploration should, in speculation, outperform Random Exploration since it can learn how to overcome the obstacle and then apply the learned solution, instead of getting past it by taking random steps every time. But as the maze environment does not have these obstacles, random actions are sufficient to explore the entire environment while it would probably be less efficient in games like Montezuma's Revenge.

5.3 Starting Positions

Two different types of starting positions are examined in the "Starting Positions" experiment, where the Policy Exploration setup is trained in two ways. One agent is trained using a fixed starting position across every episode, while the other is trained with a random starting position every episode. The result shows quite different results during training as opposed to testing. Fixed Position approaches 1.0 in return success much faster than Dynamic Position, as can be seen in figure 4.3. It takes Fixed Position about 1e6 frames compared to 5e6 to come close to 1.0 return success. During testing, however, Fixed Position has a success rate of 0.370, Dynamic Position 0.988, and control 0.250, as seen in table 4.3. This indicates that for an agent to be able to handle random starting positions it must be trained from different starting positions, as expected.

A possible reason why Policy Exploration trained on a fixed position performs worse than when trained on dynamic positions is that it has learned to get to all cells/states in the maze from the starting position reliably, as seen in the return success in figure 4.3, but not from the other positions as evident from the test results in table 4.3. Then during the testing when the agent starts at different starting positions, the success rate seems to depend on where in the maze the agent starts. If the agent starts close enough to the goal of the maze so that random steps can take it there, or if it starts somewhere along the path between the starting position used in training and the goal, videos observed during testing suggest that it has a better chance of succeeding. Whenever it does not start close to the trained position, however, the agent seems to end up getting stuck or completely lost which causes the success rate during testing to become very low in comparison to the same setup trained on random starting positions.

As seen in figure 4.3, Fixed Position gets a high return rate very quickly. This is most likely because it is an easier problem to return to cells from one starting position compared to returning to cells from multiple starting positions; every new starting position puts the agent in a state where it must learn a path anew. In theory, an agent trained on a fixed position should be able to learn to navigate the entire maze from every direction since it returns everywhere in the maze and then explores everywhere. This part, however, is a much harder problem for the agent to solve. This is not reflected by the return success graph but instead in the exploration success graph as seen in figure 4.4, where the exploration success is higher for the setup using random starting positions. This can be explained through the return phase in which the network is being fed sub-goals; a cell-by-cell trajectory towards the return target. During exploration, however, it is only being fed the final target with no cell-by-cell guidance as explained in section Then Explore on page 16.

The reason for this difference is that you do not want to guide the agent towards the exploration target along a known path, the purpose of the exploration is to find paths and states we do not already know about. Then when starting at random positions and returning anywhere in the maze, the agent will get a guiding trajectory to the target during the return phase from every direction, making it easier to learn paths in both directions. When starting at a fixed location, however, the agent will only get the cell-by-cell guidance on the path from one direction to a location, but not the other. It will have to find the other direction without guidance in the exploration phase, making it a much harder task. This can explain why the evaluation of Fixed Position is much lower compared to Dynamic Position during testing in table 4.3 even though they seem equally good in their return success during training in figure 4.3.

5.4 Original vs Adapted Go-Explore

In the "Original vs Adapted Go-Explore" experiment, a 0.982 success rate is achieved using dynamic cells and OTF-trajectories for the Policy Exploration setup compared to Original Go-Explore which got a 0.134 success rate, as seen in table 4.4. As expected, it is in concordance with the results from the experiment "Different Starting Position" 4.2.3 as Original Go-Explore requires a fixed starting position during training. The results show that dynamic cells and OTF-trajectories work well in environments with dynamic starting positions and is a viable solution when a cell representation is difficult to define statically. The size of a Dynamic cell is in this thesis fixed to a max size of 10 states, as is visualized and explained in the example with figure 3.3 in section 3.3.1. This is to guarantee that a Dynamic cell would not become too large as this would reduce the homogeneity within the cells, even if they still would fulfill the definition of "similar" presented earlier in section 3.2.1.

Finding the right size of the cells can be crucial though; small cells imply more memory usage, longer trajectories, and more precise return instructions which can lead to a general policy not being learned. However, with too large cells the trajectories give very little help and traversing one cell to the next becomes a sparse problem in itself which is difficult to teach an agent to reliably solve. As such, investigating Dynamic cells without a fixed max size could be of interest in future research. The merging criteria could contain something on how well the agent can traverse the cells and merge two neighboring cells when both are easy for the agent to navigate within and between. This would reduce the difficulty spike from

the sparse reward problem that large cells would otherwise present while also reducing the memory usage and trajectory lengths by reducing the number of cells.

Further, *when* the merging should occur might be important to specify to maximize performance. Here, the results show that merging after a certain number of frames or iterations works fine in an environment like Maze where all states are discovered quickly. However, in an environment where new states are continuously discovered, even after a longer period, merging may need to be performed more adaptively when new states are found rather than at fixed intervals.

5.5 Scaling with the Environment

For the three different mazes, the results of the Policy Exploration setup can be seen in table 4.5, where the success rate for the small, medium, and large mazes are 0.988, 0.982, and 0.940, respectively. The results point towards the algorithm being able to reasonably scale to larger tasks given enough time to train. From graph 4.6a, it is seen that the small maze's return success seems to steadily increase until it reaches near a 1.0 return success, at about 5e6 computed frames. The medium and large maze, however, both also increase linearly for a certain number of frames and then stop improving after reaching roughly 0.8 in return success. For the medium maze this happens at around 1e7 frames and for the large a little later, at about 1.2e7 frames.

A possible explanation for them both not reaching 1.0 can be that they learn to return to most of the cells, but for some cells that are hard to return to, the agent figures it receives a higher mean reward if it goes to the goal of the maze which it almost always succeeds in, rather than trying and possibly failing to get to the return target even though it receives a higher reward from it. Although finding the goal of the maze is not bad, ignoring the return target is not optimal, as a robust goal-conditioned policy is wanted. An agent that can return to every cell and find the target reliably is better than an agent that can only return to the goal reliably. For example, if the goal of the maze switched location, the agent that can return to every cell can handle this by simply changing the cell to return to, while an agent that is only good at returning to the previous goal location needs to be trained anew.

There are some ways to improve the chances of learning a more robust policy and avoid this problem, such as lowering the reward of the goal of the maze compared to the rewards for reaching the next cell in the trajectory or moving the goal location dynamically, but these fall outside of the scope of this thesis. It is also possible that, given more time, the models will eventually learn to return to every cell and gain a near 1.0 return success, as can be observed occurring for Random Exploration in figure 4.2 after 6e6 frames. Letting the agent train until it might reach this point is however too resource intensive for this thesis.

Interestingly, as seen in figure 4.6a, the return success for the medium and large mazes are very similar throughout training even though the large maze has more than 2.0 times the medium maze's states with 337 states as seen in figure A.1c in appendix, compared to 161 states as seen in figure A.1b. The difference in return success between the medium and small maze through training is however clearly visible even though the medium maze is less than 1.7 times larger with 161 states compared to 97 states as seen in figure A.1a.

This pattern can also be observed in their exploration success in figure 4.6b where it is not as apparent, which can be explained by the exploration becoming harder the larger the

environment. Since the larger the environment, the longer the agent will potentially have to navigate without the cell-by-cell guidance during the exploration phase, as discussed in section 5.3.

The return and exploration success patterns discussed above might indicate that the training time, depending on environment size, could have a logarithmic or similar slow increase between the different maze sizes. This shows promising scalability for the algorithm; however, more tests are needed to confirm this.

As further speculation, another explanation for medium and large being vastly different from the small maze in figure 4.6a is that Policy Exploration on the small maze handles every cell as a special case and does not learn a broader skill. For the medium and large mazes with more cells, the agent instead learns a more generic skill that can be applied to most cells; this is possibly a more difficult feat, which would explain the uneven differences between the different maze sizes. This is, however, nothing that can be confirmed with only three data points.

Chapter 6
Outlooks and Conclusions

This final chapter presents the outlooks for this thesis in section 6.1 and then wraps it together with a conclusion in section 6.2.

6.1 Outlooks

The adapted algorithm in this thesis is tested on relatively small environments. Testing it on a larger, or even "endless", environment with dynamic starting positions is an interesting future work, as this would check how it performs on a more complex dynamic task where efficient exploration is important. For larger environments, however, SIL might be needed. It should be possible to implement it together with OTF-trajectories given a function to compare the performance of trajectories with different starting positions against one another. Additionally, the hypothesis of policy exploration being more useful than random exploration in larger and more complex environments can be examined.

Another interesting area to examine is how the algorithm performs when introducing more dynamic elements, e.g., moving goals or other entities moving around in the environment.

Finally, testing the adapted Go-Explore in a robot simulator would also be an interesting area for future work. The adapted version shows promising results with dynamic starting positions in the maze game, which indicates that it may work well for a robot for a similar dynamic task, but this needs to be examined further to be confirmed.

6.2 Conclusion

In a maze environment with dynamic starting positions, the adaptations "Dynamic cells" and "OTF-trajectories" to the Go-Explore algorithm that we introduce in this thesis makes it perform significantly better than the original version with a success rate of 0.982 for the

adapted version of Go-Explore compared to 0.134 for the original version of Go-Explore. These additions to Go-Explore enhance its ability to work with dynamic starting positions according to the results.

The training time and performance of the adapted Go-Explore seem to scale in one of two ways; either the training time scales logarithmically or similar to the size of the maze environment, or the time scaling is even lower, but more data is needed to confirm either hypothesis.

When training on the small maze environment using different distributions of exploration strategies, close to no difference between the different setups could be seen, especially when taking the standard deviation observed for one of the setups into account. In more complex environments containing reoccurring obstacles, however, policy exploration can be more effective according to previous studies [6].

Finally, for an environment where a cell representation of a predefined size can be either misleading or inefficient, dynamic cells have shown to work well together with OTF-trajectories, with a test score more than seven times better than the original Go-Explore on a task with dynamic starting positions.

In conclusion, the additions of Dynamic cells and OTF-trajectories make the algorithm Go-Explore from the article "First return, then explore" by Ecoffet et. al. [6] able to handle dynamic starting positions in a maze environment.

References

[1] Rafael Basso, Balázs Kulcsár, Ivan Sanchez-Diaz, and Xiaobo Qu. Dynamic stochastic electric vehicle routing with safe reinforcement learning. *Transportation Research Part E: Logistics and Transportation Review*, 157:102496, 2022.

[2] David Mathew Jones and S. Kanagalakshmi. Data driven control of interacting two tank hybrid system using deep reinforcement learning. *2021 IEEE 6th International Conference on Computing, Communication and Automation (ICCCA)*, pages 297–303, 2021.

[3] Ikumi Kodaka and Fumiaki Saitoh. A study on application of curriculum learning in deep reinforcement learning : Action acquisition in shooting game ai as example. *2021 IEEE 12th International Workshop on Computational Intelligence and Applications (IWCIA)*, pages 1–6, 2021.

[4] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016.

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

[6] Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O Stanley, and Jeff Clune. First return, then explore. *Nature*, 590(7847):580–586, 2021.

[7] OpenAI. Atari gym. https://www.gymlibrary.ml/environments/atari/. (accessed: 22.05.2022).

[8] Parker Brothers. Montezuma's revenge. https://www.retrogames.cz/play_124-Atari2600.php. (accessed: 13.05.2022).

[9] Activision. Enduro. https://www.retrogames.cz/play_028-Atari2600.php?language=EN. (accessed: 23.05.2022).

[10] Taito. Space invaders. https://www.retrogames.cz/play_016-Atari2600.php. (accessed: 23.05.2022).

[11] Marlos C. Machado, Marc G. Bellemare, Erik Talvitie, Joel Veness, Matthew J. Hausknecht, and Michael Bowling. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *CoRR*, abs/1709.06009, 2017.

[12] Activision. Pitfall! https://www.retrogames.cz/play_029-Atari2600.php. (accessed: 13.05.2022).

[13] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[14] Stefan Schaal. Learning from demonstration. In M.C. Mozer, M. Jordan, and T. Petsche, editors, *Advances in Neural Information Processing Systems*, volume 9. MIT Press, 1996.

[15] OpenAI. Procgen benchmark. https://openai.com/blog/procgen-benchmark/. (accessed: 28.01.2022).

[16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[17] Juyang Weng. Post-selections in ai papers in nature since 2015 and the appropriate protocol, 2021.

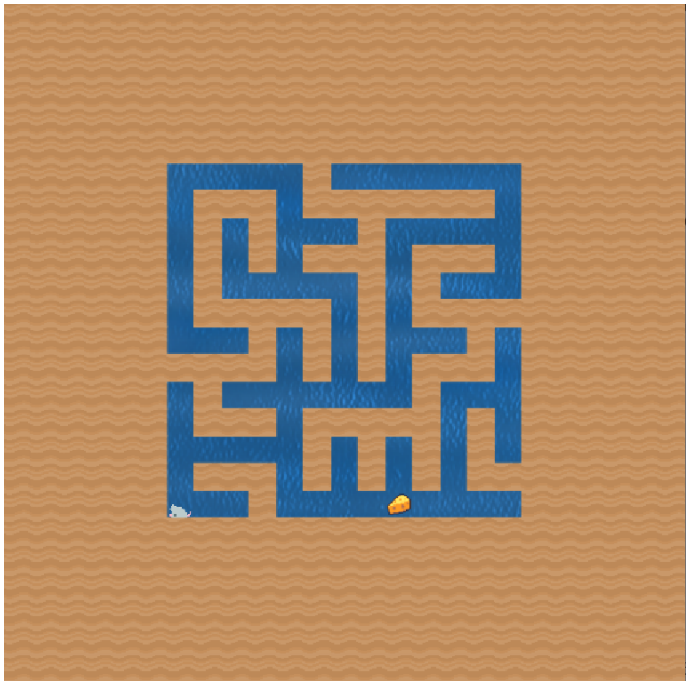
[18] Juyang Weng. Post-selections in ai and how to avoid them, 2021.

[19] Haitao Xu. *Intrinsic reward driven exploration for deep reinforcement learning*. PhD thesis, University of Otago, 2021.

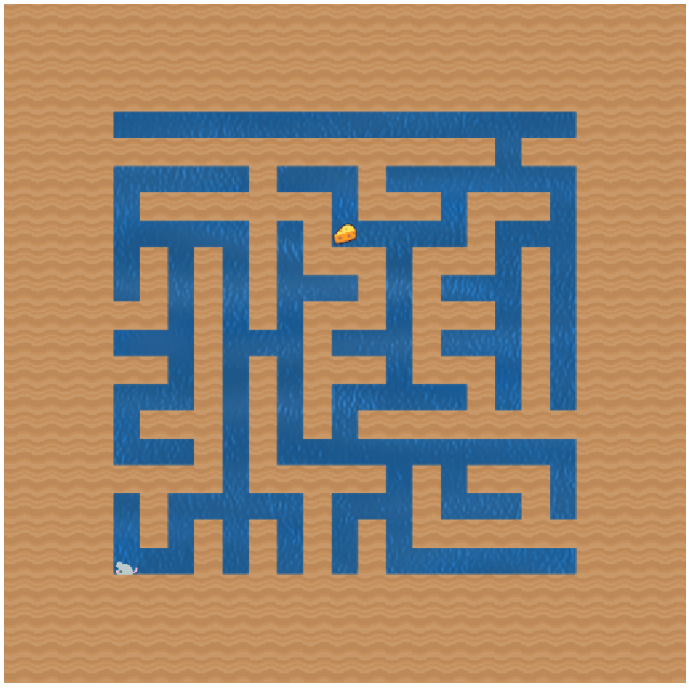
[20] Tianjun Zhang, Huazhe Xu, Xiaolong Wang, Yi Wu, Kurt Keutzer, Joseph E Gonzalez, and Yuandong Tian. Bebold: Exploration beyond the boundary of explored regions. *arXiv preprint arXiv:2012.08621*, 2020.

Appendices

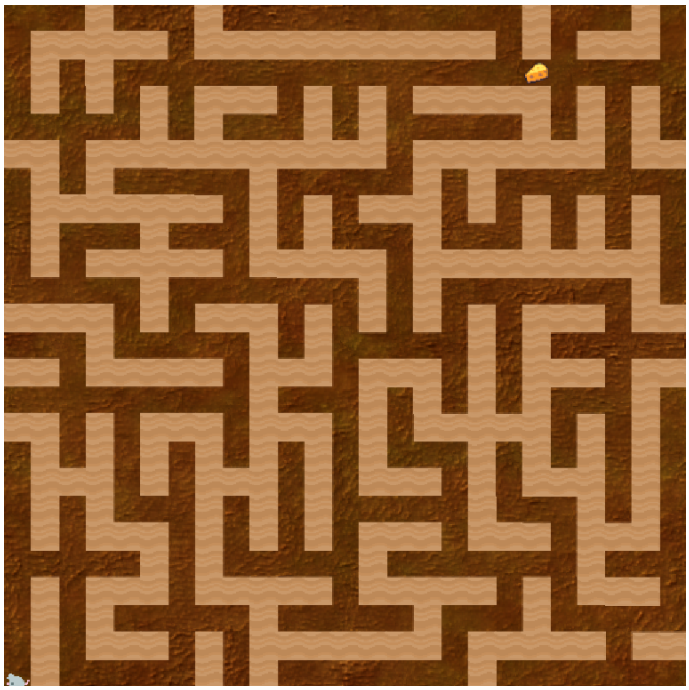
Appendix A
Mazes



(a) The small maze, level 137 in Procgen.



(b) The medium maze, level 118 in Procgen.



(c) The large maze, level 100 in Procgen.

Figure A.1: The mazes of Procgen game "maze" used in the experiments. To traverse the maze the agent (the mouse) can go up, down, left, or right. When the agent reaches the cheese, it gains a score of 10 and terminates the episode. If the agent does not find the cheese in 500 steps the episode also terminates but with a reward of 0.

Appendix B
Implementation Details

B.1 Return Policy

The return policy in this adapted version of Go-Explore is implemented using a neural network consisting of a Convolutional Neural Network (CNN) that processes the image from the screen and three fully connected layers that take the output of the CNN together with a target representation to produce the policy and value for the given state and goal. The network is the same as proposed in "First return, then explore" [6] except for a few minor changes.

In "First return, then explore", the last four frames are sent as input to the CNN. It is assumed that the original version of Go-Explore takes the last four frames as input because they only act every fourth frame. This does not apply to this thesis as it is desired to take a step at every frame, hence only the latest frame is used as input. The image resolution also differs where the article uses frames with a resolution of 105x80, but, since the Procgen screen is a square, a resolution of 128x128 is used instead. Finally, the target representation also uses fewer parameters than the target representation used in the original Go-Explore.

These changes create a lot more nodes in CNN layers due to the larger image but the input nodes from the target representation are a little fewer, overall increasing the number of trainable nodes from 9.7 e6 to 18.9 e6. See figure B.1 for full details of the network.

The network, as introduced in "First return, then explore", has a way to counteract cases where the agent can get stuck. For example, if the action from the policy is moving into a wall, the next state would be the same and therefore the next action could be to move into the same wall again and so forth. Go-Explore uses an entropy term just before the softmax activation for the policy in the final layer, see figure B.1, to work against this during training. This entropy term makes the policy more and more likely to take another action than it otherwise would the larger it grows, which hopefully breaks cycles of looping actions. The entropy is increasing constantly during the return phase, very slowly at first and then increasing in speed, and resets in both speed and acceleration once the next cell in the trajectory is

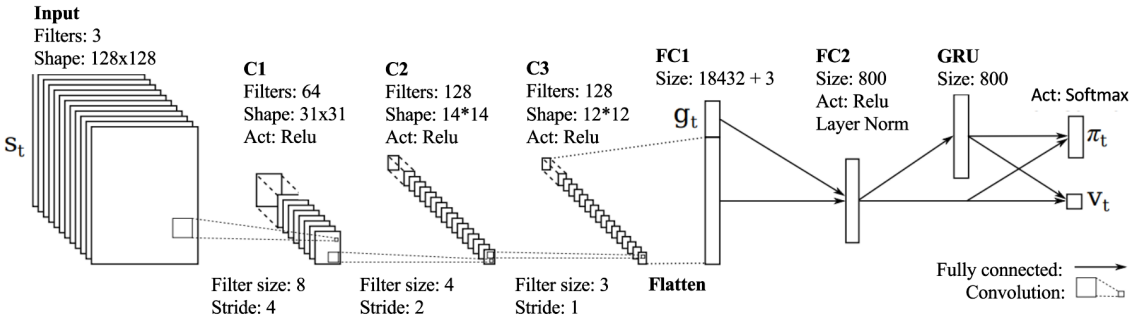


Figure B.1: The policy network, an adapted image from the article of Ecoffet et al [6] to match the network used in this adapted version of Go-Explore. The input S_t is the RGB channels from the game frame, rescaled to a size of 128x128. The input goes through a CNN of three layers with the intermediate states C1, C2, and C3, and is finally flattened to a 1-dimensional array. This array is concatenated with the cell representation of the target, g_t, which is fully connected to the state FC2. FC2 is again fully connected to a GRU module (Gated recurrent module, similar to a LTSM), the policy activation and value-function activation layer. The GRU also has a fully connected layer to both the policy and value-function activation which are softmax activation layers and produces a policy head $\pi(a|s)$ and value head $V(S)$ respectively

reached. During the exploration phase, however, the increase starts after 50 steps, resetting upon reaching the exploration target. Note that the entropy has no effect during random exploration as the actions taken are already random. The entropy term is on during training but off during testing to let the network have full control during tests.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2022-06-16

EXAMENSARBETE "First return, then explore" Adapted and Evaluated for Dynamic Environments
STUDENTER Nicolas Petrisi, Fredrik Sjöström
HANDLEDARE Hampus Åström (LTH), Volker Krueger (LTH)
EXAMINATOR Elin A. Topp (LTH)

Go-Explore anpassad för dynamiska miljöer

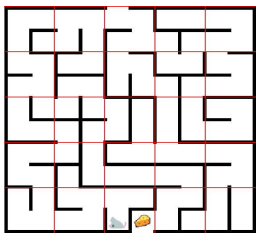
POPULÄRVETENSKAPLIG SAMMANFATTNING **Nicolas Petrisi, Fredrik Sjöström**

Genom förbättringar på den toppmoderna algoritmen Go-Explore fungerar den nu för dynamiska miljöer med mer än sju gånger bättre resultat. Dynamiska celler och "On The Fly"- (OTF) vägar ger Go-Explore förmågan att kunna navigera genom en labyrint med slumpade startpositioner, vilket den inte kunnat göra förr.

Go-Explore är en toppmodern algoritm inom förstärkningsinlärning gjord för att effektivt utforska miljöer. Den visar enastående resultat när den spelar Atari-spel såsom Montezuma's Revenge och Pitfall och i en robotsimulator klarar den att placera objekt i hyllor även när några av hyllorna har en hasp.

När Go-Explore-agenten navigerar genom miljön sparar den undan platserna den hittar i olika "celler". Alla tillstånd som är lika varandra, såsom att de är nära varandra på skärmen i en 2D-miljö, sägs tillhöra samma cell. Detta gör att en cell representerar flera olika tillstånd. När agenten utforskar och går genom dessa sparar den undan vägen den gått för att komma ihåg hur den har kommit fram till de olika cellerna.

Men problemet med Go-Explore är att den är byggd för att alltid börja och sluta på samma position, vilket inte alltid är verklighetstroget. När man flyttar på antingen start- eller slut-positionen fungerar inte längre algoritmen då vägarna som algoritmen sparar undan alltid antar att den börjar på samma position. Vidare så kan man inte heller anta att bara för att två platser på skärmen är nära varandra så behöver platserna i sig inte vara lika, vilket antas i orginalversionen. Att vara på ena eller andra sidan av en vägg i en labyrint kan



ha enorm betydelse för om man är nära att lösa labyrinten eller inte, som man kan se i figuren där det röda rutnätet delar upp miljön i sina celler.

Två stora ändringar i algoritmen är gjorda för att anpassa den till dynamiska miljöer. Närliggande celler slås ihop under körning vilket gör att de kan användas i områden där det är svårt att definiera bra celler av större storlek innan agenten undersökt området. Och istället för att komma ihåg exakt vilka celler agenten ska gå igenom för att komma till sitt mål så skapas OTF-vägar genom att kolla på vilka celler som är grannar för att bygga vägar som kan gå till målet oavsett var man startar.

Med ändringarna lyckas den anpassade Go-Explore lösa labyrinter av olika storlekar med slumpade startpositioner mer än sju gånger oftare än Go-Explore utan ändringarna.