3D Spatial Navigation in Octrees with Reinforcement Learning

Using Sparse Rewards with Hindsight Experience Replay

Matias Hermanrud Fjeld

Thesis submitted for the degree of
Master in Informatics: Programming and Networks
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2018

3D Spatial Navigation in Octrees with Reinforcement Learning

Using Sparse Rewards with Hindsight Experience Replay

Matias Hermanrud Fjeld

3D Spatial Navigation in Octrees with Reinforcement Learning

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

The recent surge in research towards general artificial intelligence has produced a wealth of promising techniques, which when utilized for traditional robotics tasks can give rise to new and interesting algorithms.

This thesis proposes and implements a method of applying Reinforcement Learning (RL) to three-dimensional octree navigation. Octrees are spatial models used in robotics for navigation and collision avoidance. 3D navigation with octrees can be used in a variety of applications such as autonomous Search and Rescue quadcopter drones, or any other robotics task involving movement in three dimensions.

Octree navigation traditionally uses man-made path finding algorithms. In this work we present the first known application of Reinforcement Learning to 3D navigation with octrees. The proposed method uses sampling-based observations and continuous actions spaces, and applies Hindsight Experience Replay (HER), a data augmentation technique, to increase sample efficiency.

Along with an implementation of the methods, we design and implement a handful of simulated environments for evaluating performance on simple navigation tasks. From the experiments, we find that the combination of sparse rewards and continuous observations are beneficial over alternative setups.

Experiments show low success rates when trained and evaluated on the navigation tasks, and further study is necessary to determine if Reinforcement Learning is a viable approach to 3D octree navigation. Regardless, this thesis can serve as baseline for future research and shed light on a new potential application for machine learning.

Preface

They say any work of research is steeped in blood, toil, tears, and sweat. Mine is no exception.

I would like to thank the countless people who have helped me on my journey not just to write this thesis but who supported and cheered me on towards this extraordinary milestone in my life. Your support was crucial, and I will be forever grateful.

First, to my supervisors, Justas, Jim, (and Kai), thank you for supporting me in every way you could, and for channeling my frustation and dread into research and writing.

Thank you to my fellow students, for bringing me hope when I had none. Particularly Guilherme for help with proofreading, and Bjørn-Ivar and Eivind for providing advice and encouragement.

To all the members of the research group, who have provided a tremendous environment for learning, having interesting discussions and eating delicious birthday cakes, thank you.

I would also like to thank Bruno Castro da Silva at the Federal University of Rio Grande do Sul in Brazil for his suggestions, particularily on the possible future work of a scale-invariant variety of this method.

Thank you to my big brother Jørgen who has been my mentor, inspired me and spent countless hours answering my every question about programming when I was a teenager. To my family, and my grandparents in particular, thank you for supporting, worrying about and questioning the big decisions in my life. Education turned out to be a good idea, after all.

Most of all, to my girlfriend Tonje for her unwavering support. Without you, I would have given up a hundred times over. Thank you.

Contents

1 Introduction

The age of robots is ostensibly upon us with the promise of automation on a scale not seen since the industrial revolution. At the center is the challenge of robot mobility, as we all have seen when clumsy humanoid robots trip over seemingly benign obstacles. Their poor mobility falls short when compared to that of humans. Mammals are incredibly adept at moving around in unknown and complex environments. Research into rats suggests that specific navigational neurons are hard-wired into mammalian brains (Langston et al. 2010). In other words, we were likely born with a priori spatial awareness, evolved over countless generations.

Just as humans have specialized parts of our brain to model space, robots need internal models of the world to navigate. The octree, which we will describe in section 2.1, is a spatial model commonly used in robotics. Robots can typically employ graph search algorithms to solve the problem of planning 3D movement from octrees. These algorithms formulate path planning as a graph problem and use traditional graph traversal techniques to find optimal paths. Human-made algorithms, while useful, are limited by humans' ability to imagine and formulate them. Algorithms created with machine learning, on the other hand, can allow for more complex behavior at the expense of being elusive as well as difficult to reason about and comprehend.

Reinforcement Learning, which we will describe in section 2.2, is a fast-growing field of machine learning research devoted to learning by trial and error. This thesis proposes a method for how Reinforcement Learning can be combined with octrees to learn navigation in three-dimensional space. Building upon Hindsight Experience Replay, described in section 2.3, the method uses action spaces and sparse rewards. We implement the method in a simulated environment and run experiments on a handful of navigation tasks to see what a trained agent is capable of, and under which conditions. The results of these experiments are presented in chapter 4.

Finally, in chapter 5, we interpret the implications of our experiments, suggest further work and conclude by summarizing our findings.

1.1 Motivation

Before going into the technical detail, let's consider a use case to motivate this line of research. Imagine we are designing an autonomous flying quad-copter that will be used for Search and Rescue (SAR) operations. The SAR drone will search inside buildings for human beings, and has to map and navigate its environment without human intervention. The hypothetical drone will use the design illustrated in figure 1.1. It will be equipped with a lidar — a laser scanner that measures the distance to obstacles near the drone, similar to how a radar works. A lidar outputs point cloud information, a set of points measuring where the laser beams have collided with objects while scanning the environment. The point cloud is fed into an OctoMap, which builds a 3D representation of the drones' surroundings. To make the drone move we need two final components. Firstly we need a policy that looks at the OctoMap and decides where the drone should move. Secondly we need a motor controller to make the motors move, executing the plan output by the policy. In this thesis we focus on the policy, specifically *how* this policy can be trained by using Reinforcement Learning.

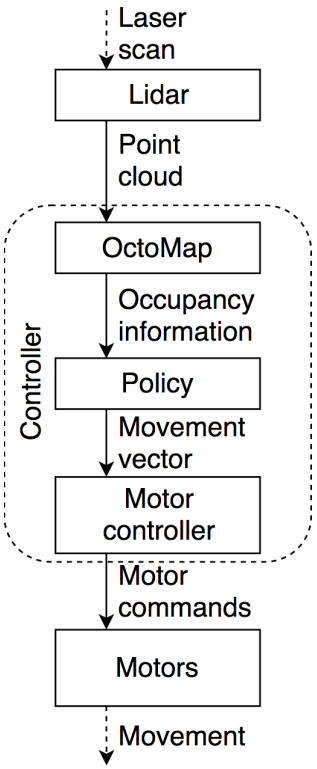

Figure 1.1: A conceptual overview of an RL-based robot architecture.

1.2 Goals

Reinforcement Learning has not yet been applied to 3D spatial navigation with octrees. This thesis sets out to do so with the following goals:

1. Design and implement a method and a set of simulated Reinforcement Learning environments for navigating 3D spaces modeled by octrees.
2. Assess the feasability and properties of this method experimentally.
3. Explore how different environment design choices affect learning outcomes.

1.3 Related work

Surprisingly, we find no existing research combining Reinforcement Learning with octrees for navigation. This shortage of related works might be because the problem is outside the boundaries of our current technological capabilities, or simply that it has not been attempted before.

2 | Background

This chapter provides the conceptual foundations for understanding how Reinforcement Learning can be applied to octrees.

2.1 Voxels, octrees and OctoMap

Voxels are the 3D equivalent of pixels, i.e. they are values in a three-dimensional array representing some property of space within a region. When robots map their surroundings, they are often interested in occupancy — whether a region of space is occupied or not. This can be represented with binary voxels where each voxel holds a boolean. If the voxel is true, it means that the corresponding space is occupied, and vice versa. This is illustrated in figure 2.1.

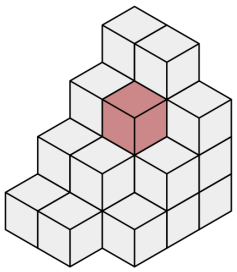

Figure 2.1: An array of voxels, with a single occupied voxel colored red (Wikipedia:Vossman 2017).

Describing large spaces with arrays of voxels is inefficient both with regards to memory usage and computational complexity. A voxel array describing a 5 m X 5 m X 2 m room at 1 cm resolution would require 50 million voxels. Using the array for motion planning would require looking at the occupancy of individual 1 cm voxels, where most adjacent voxels would have the same value.

Octrees compress voxel arrays by combining multiple, adjacent, same-valued voxels into fewer, larger voxels holding the same value. In other words, octrees implement a variable-resolution, lossless encoding of voxels which dramatically reduce memory requirements. A square metre of empty space could be described as a single node in an octree instead of as a million 1 cm voxels in an array. Figure 2.2 illustrates three renderings of an octree at different resolutions. The octree, somewhat humourously, depicts a physical tree.

Figure 2.2: Three octreees modeling the same physical tree at different resolutions (Hornung et al. 2013)

Octrees recursively combine voxels in pairs along each dimension, which in three dimensions combines eight ("octo" in latin) cubes into

one larger. This relationship is reflected in the tree structure: A node in an octree can have up to eight children. The relationship between nodes and subdivisions of space is illustrated in figure 2.3.

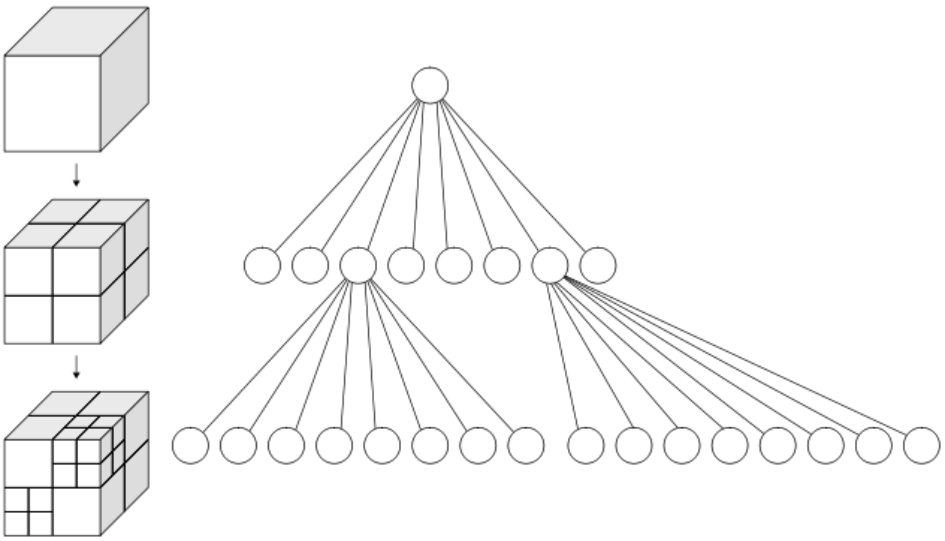

Figure 2.3: An illustration of how child nodes in an octree relate to subdivisions in space (Wikipedia:WhiteTimberwolf 2010)

OctoMap (Hornung et al. 2013) is an open source octree implementation written in C++. Octrees implemented with OctoMap are both easy to update with additional information and do not require pre-allocation — knowing the size of the environment beforehand.

The authors of OctoMap note that their framework combines three important aspects: (1) The model integrates probabilistic measurements in a robust way, i.e. it intregrates noisy sensor data. (2) It models unmapped areas, where no sensor data is available. Thus a volume in the OctoMap can be free, occupied or unknown. (3) It is also efficient, both when considering access-time and especially memory consumption.

One important design decision in OctoMap is worth noting: Inner nodes store aggregated occupancy values from their children. As a consequence coarse resolution occupancy can be obtained quickly without having to traverse deeply within the tree. This makes OctoMap not just memory efficient, but also gives algorithmic speed-ups. This property can be exploited to sample large areas of space quickly.

The following sections describe some key properties of the OctoMap.

2.1.1 Aggregation

The OctoMap tree structure allows for variable-resolution occupancy querying by overlaying coarser-resolution voxels onto finer ones. Each internal node in an OctoMap holds an aggregation of its child nodes, and stores the *maximum* occupancy probability of all its children [1]. This

[1] https://github.com/OctoMap/octomap/blob/v1.8.1/octomap/include/octomap/OcTreeNode.h#L83

aggregation continues recursively from the leaf nodes all the way up to the root node, which contains the maximum occupancy probability of all the leaf nodes in the tree. As a consequence, the tree can be queried at different depths to obtain occupancy readings at different resolutions with $O(\log n)$ time complexity. Queries at shallow depths (near the top of the tree) are faster because they iterate over fewer nodes, at the cost of diminished accuracy.

2.1.2 Data ingestion

Internally, each node in an OctoMap stores a floating point number measuring the collision probability for a ray passing through that voxel. These probabilities are used in a sensor model that integrates rays from a point cloud into voxels in the OctoMap. The probabilities are stored as log odds [2] to speed up computations. Outliers are clamped to allow nodes with high-confidence occupancy values to be combined, reducing memory requirements.

OctoMaps are created from point clouds. A point cloud is a set of vectors in octree space, that originate from the point of the sensor and end when they hit a wall. Essentially, the sensor emits rays from its position, measures the distances those rays travel, and then calculate the positions where the rays collided. These ray vectors, along with the sensor position, are fed into the OctoMap. Both are defined with Euclidian coordinates in octree space.

Because real-world measurements introduce random noise, the OctoMap sensor model was designed to handle probabilistic measurements. As a consequence, noise can be introduced when integrating simulated point clouds. A surface in an OctoMap can have seemingly random protrusions and dents even through the point cloud did not really measure these. These are a consequence of the OctoMap sensor model, in which the model trades away some accuracy for the benefit of being able to integrate noisy sensor readings over multiple time steps. We can see this effect on the octrees visualized in experiments.

2.1.3 Compression

One of the primary goals of the OctoMap data structure is compression. Voxels in octree space can be compressed by combining them together into larger voxels. In the octree, this is achieved by pruning child nodes that are all leaf nodes and have the same occupancy value as their parent. For example, a block of eight nodes indicating free space will propagate their maximum occupancy value to their parent node, so the parent node will also indicate that the space is free. Because the child nodes now hold redundant information at a finer resolution, they are all deleted and memory is saved.

[2] The logarithm of the odds $\frac{p}{1-p}$ for a probability p

2.1.4 Searching and iterating

There are two ways of obtaining octree occupancy values: *a*) searching for a specific node in the octree, or *b*) traversing all nodes and reading values one at a time.

In the first case, we want to find the occupancy probability of a specific point in space. To find the node corresponding to the voxel containing the point we are looking for, the OctoMap library first converts the position to a key. The key describes a distinct path through the tree with left/right directions along each dimension when descending downwards from the root through child nodes.

Using the second method of traversing leaf nodes can reduce time complexity slightly, but adds significant design complexity. Nodes have to be filtered and their octree keys translated to query positions position that might not be aligned with voxel boundries. For this reason, we have used searching instead of iteration in this thesis.

2.2 Reinforcment Learning

Reinforcement Learning, abbreviated RL, is a field of machine learning concerned with making optimal decisions over time. Like a dog learning to sit when repeatedly given a treat, an RL agent determines what actions to take based on expected rewards. RL divides time into discrete time steps $t = 0, 1, ...$, each in which the agent observes the state of its environment, decides how to act and receives a reward as a result. Figure 2.4 illustrates this agent-environment loop.

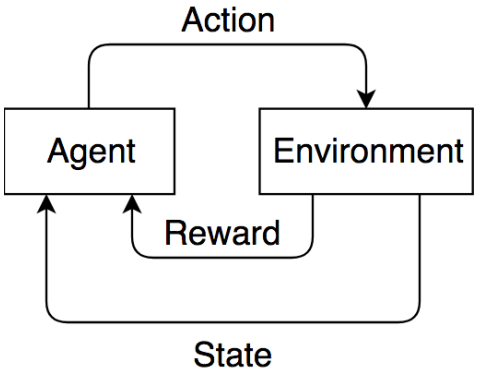

Figure 2.4: An illustration of how an agent interacts with its environment in Reinforcement Learning.

If our agent was a dog, heard us saying "sit", and chose to sit down, its treat could be the reward +1. Rewards in RL are scalar values, numbers, given by a reward function which is part of the environment. The goal of the agent is to maximize total reward over time, but that requires balancing short-term and long-term rewards. Rolling in the grass right now might be pleasurable, but a little patience and some sitting can produce the even bigger delight of a delicious treat. RL models this trade-off with a weighted sum of expected future rewards called a return.

Because the agent only observes rewards after it has acted, it has to balance exploration for new experiences with the exploitation of the experiences it already has. The first time our dog received a reward for sitting, it might have done so randomly, and just happened to stumble upon a rewarding action. If the dog then always sits, it might get a treat

for sitting but misses the second treat it could have obtained for giving paw. There are numerous techniques for how to explore efficiently, but a simple approach commonly used is epsilon-greedy exploration. In this case, the agent has a random chance of $\epsilon \in [0,1]$ of exploring by choosing a random action. Otherwise, the agent chooses the action with the maximum expected future return.

By learning from past experiences, the agent forms a policy, a function from states to actions. Usually, states in RL are vectors and actions are vectors or scalars. In our case, with an agent navigating a three-dimensional space, the state could contain the position of the agent as well as information about obstacles. The action could be a direction to travel in; either in the form of a discrete number indicating a choice between discrete directions; or a vector with continuous values indicating where and how far to go. The choice between continuous and discrete state and action spaces has consequences for sample-efficiency and choice of agent architecture.

When the agent observes the complete state of the environment at each time step, we say that the environment is fully observable. What happens if the agent receives only partial information? A partially observable learning task is similar to how we humans perceive the world, when we turn our head to see behind us while still remembering what is in front of us. Partial observability adds complexity because the agent has to extrapolate the state based on uncertain observations. Techniques in RL exist for dealing with this problem, but the added complexity can complicate experiments greatly.

Reinforcement Learning tasks can be episodic or continuous. Episodic tasks are those that end after a finite number of time steps. Once the agent reaches a terminal state, the task is completed, and the environment resets. Actions taken in previous episodes do not affect later ones except that the agent uses the gathered experience to improve its policy. Thus the agent can go through a task over and over again. The environment resets before each episode. Combined with a computer-simulated environment, episodic tasks can allow an agent to gather a great deal of experience in a relatively short amount of time.

An optimal policy is one that maximizes the future expected return from initial states. There is a multitude of ways to create policies from experiences, too many to detail here. Reinforcement Learning is a broad and active field of research, and more than ten thousand papers relating to RL have been published yearly since 2012 (Henderson et al. 2017). The choice of specific technique depends on the learning task; whether the environment is partially observable, has a continuous or discrete action space, and several other factors.

2.3 Hindsight Experience Replay

Formulating a reward function is one of the major challenges in using RL. Any reward function, which returns a scalar number after every

action the agent takes, exists on a spectrum anywhere from extremely simple to exceedingly complex. The set of possible reward functions is vast, and the choice of reward function can have surprising and unintended side-effects.

Consider, for example, a reward function for an agent navigating a maze. A complex reward function might return the distance to the goal, possibly combined with the number of walls between itself and the goal. The reward function attempts to combine *what* the agent should do, get out of the maze, with *how* the agent should do it, by moving closer to the exit. If the agent learns to always move in the direction of the exit, it might get stuck in a dead end corridor and never find the exit at all. Using complex reward functions can defeat the purpose of machine learning, which is to specify an objective and let the machine learn how to achieve it on its own. Reward functions that increase as the agent moves closer to the ultimate goal are called dense or shaped rewards, because they shape the agent's behavior a little bit at a time.

On the other end of the spectrum, the simplest reward function might return a binary number telling the agent if it reached the exit or not. This is called a sparse reward. Such a specification is very precise, and gives the agent a clear end-goal objective without restraining how the agent goes about reaching that goal. The problem with this approach is that the agent has to explore an exponentially large state space before receiving any feedback on its performance. Wandering the maze at random, the agent has to reach the exit by chance before learning that reaching the exit is good. The agent is then left with the incredibly difficult task of credit assignment — Figuring out which of the many actions it took that actually led to the reward it received.

Hindsight Experience Replay (HER) is a recently published RL technique for sample-efficient learning from sparse rewards (Andrychowicz et al. 2017). HER builds on Universal Value Function Approximators (Schaul et al. 2015), an extension to RL that formulates policies as functions of states *and goals* to actions. A goal can be any vector or scalar. In the context of the maze example above, a goal could be a position in Euclidian coordinates, a vector of two real numbers. Universal policies, functions taking goals as input, allow an agent to generalize over goals as well as states. By learning how to achieve small goals, the agent can gradually acquire the required knowledge to accomplish more complex ones. A simple goal might be to move down a corridor or around a wall. A more complex goal can be to reach the exit. Progressing from simple to complex goals requires a curriculum, and this is where HER comes in.

HER requires RL environments to output goals alongside state information. The agent is told at every time step what goal the environment wants it to achieve, as well as what goal it actually achieved with the action it last took. Before taking an action the agent does not know what goal will be achieved with that action. In hindsight, however, the agent will know what goal was *actually* achieved. This fact can be exploited to *pretend* that the achieved goal was what it was

meant to do all along, and receive a reward as if it had achieved it. This leads the agent to learn the relationship between actions, states and goals much faster.

2.3.1 Environment

We will now go into more detail on how HER works, starting with the environment. HER gives the agent access to the reward function so that it can compute rewards in hindsight. We can see this in figure 2.5. The figure describes an OpenAI Gym environment adapted to work with HER. (1) The state of the environment is reset at the beginning of an episode, and the environment returns an initial state. This state contains a desired goal that the environment wants the agent to achieve. (2) The agent uses the state received from the environment to choose an action, and sends this to the environment's step function. (3) The environment calculates the next state as well as the reward. The reward function takes the desired goal and the achieved goal as input. The state emitted by the step function includes the new desired goal as well as the last achieved goal. The agent can use the achieved goal with the reward function to re-calculate a reward in hindsight. Step 2 and 3 are repeated until the environment signals that the episode is done.

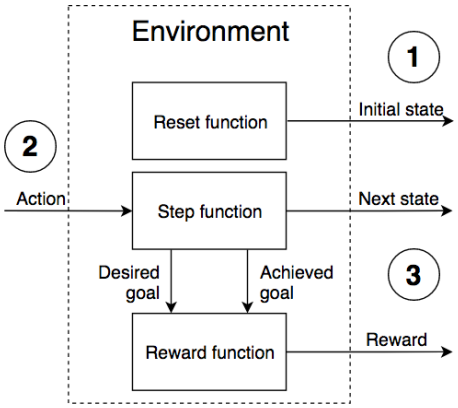

Figure 2.5: A flow chart of how the environment exposes its reward function.

2.3.2 Agent

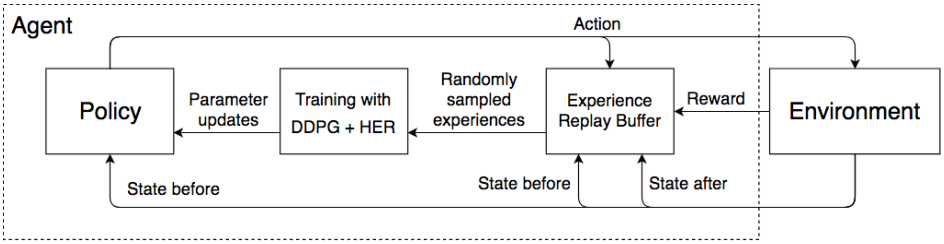

Figure 2.6: A flowchart of how the agent is trained.

Andrychowicz et al. provided an open source implementation of their agent as part of OpenAI Baselines, a repository of RL agents for others to build their research on. HER can be combined with any offline RL algorithm, but the authors of HER chose DDPG (Lillicrap et al. 2015). DDPG, Deep Deterministic Policy Gradients is an actor-critic, model-free RL algorithm for continuous action spaces. The details of DDPG are

not important to HER, except that DDPG improves a policy iteratively by sampling from an experience buffer and altering the parameters of the policy's neural network approximator via backpropagation and gradient descent.

Figure 2.6 shows the agent interacting with the environment and learning by collecting experiences using the current policy, storing them in the replay experience buffer, sampling random experiences and calculating weight updates using DDPG + HER. Note that the figure shows states from two subsequent time steps; The agent stores the state before the action as well as after. The state before, action, state after, as well as the reward are combined into a *transition tuple* and stored into the experience replay buffer. The buffer stores a fixed number of recent transitions, and these transitions are sampled to train the policy. Colloquially, we call these transition tuples "experience". It is between sampling and training that we find the core of the HER algorithm.

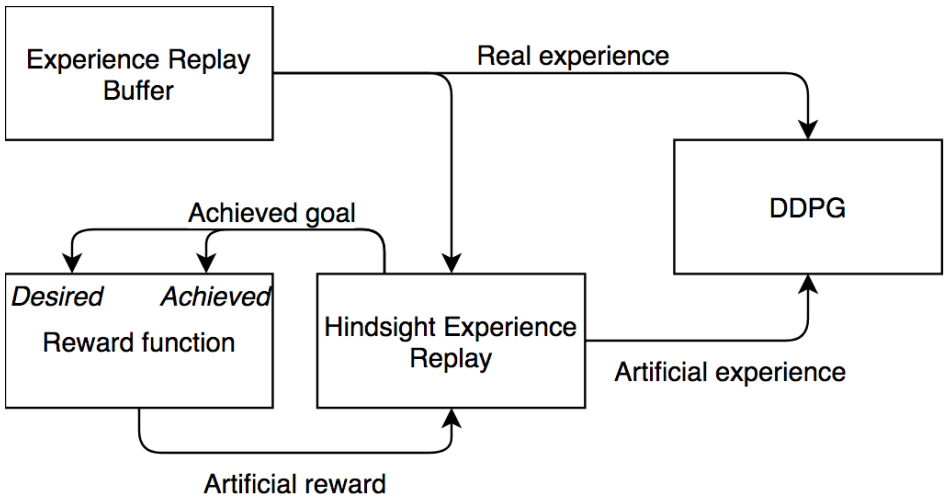

Figure 2.7: A flowchart of how Hindsight Experience Replay generates artificial experience by substituting the desired goal for the achieved goal in hindsight and re-calculating the reward.

When given sparse rewards, the agent initially fails to achieve the goals set for it by the environment. For example, the environment might want the agent to move right, but instead it moved left. It receives a negative reward for this, but since it mostly receives negative rewards anyway it doesn't really learn much from this. It would be better for the agent to learn *what to do* instead of *what not to do*. This is where HER comes in.

HER takes transition tuples as input and creates new ones by substituting the desired goal and re-calculating the reward. If the agent moved left, HER can pretend that the agent was supposed to move left all along. Since HER has access to the reward function it can reward the agent for achieving its goal. HER substitutes the desired goal for the achieved goal, something that is only possible in hindsight. Figure 2.7 illustrates how this works.

Andrychowicz et al. point out that HER provides a form of implicit curriculum. By rewarding the agent for what it achieved even an agent taking random actions will achieve something. As the agent learns more about how to achieve goals, it will naturally progress to more difficult ones. Another way at looking at HER is as a form of data-augmentation, which increases the diversity of samples by modifying them.

Interestingly, the original experiments with HER showed better performance with sparse rewards than with shaped ones. Andrychowicz et al. argue that this might be due to how shaped rewards often compromise on the metrics we care the most about.

3 | Methods

This chapter describes a novel method for applying Reinforcement Learning to 3D navigation in octrees. The proposed method uses continuous action spaces, sparse rewards and sampling based variable resolution voxel observations. Before going into details, however, a motivating comment on the complexities of RL research is necessary.

Reinforcement Learning-based research is often difficult to reproduce (Henderson et al. 2017), and involve many uncontrolled variables like hyperparameters, network architecture as well as randomness from weight initialization and in environments. Henderson et al. notes that RL techniques inherently yields results that have high variance. This has important implications for experiment design and reporting, particularily the need to reduce unnecessary complexity in experiments. For example, entire research papers (Mahmood et al. 2018) deal with the task of simply setting up a reproducible physical robot experiment.

To lessen the considerable variability inherent in RL, the experiments in this thesis will be conducted in a simplified simulation involving positions but not time and forces. Velocities and torques will not be modeled, avoiding the need for a physics simulator. Instead, the agent will simply control the position of a point in three-dimensional space.

As mentioned in the background (chapter 2), the agent will receive an observation of the state at each time step, choose an action and receive a subsequent scalar reward. Part of the design process involves deciding on observation and actions spaces. To aid the reader, we will first consider a simplified, two-dimensional illustration of the proposed setup before delving into and discussing each aspect of the full three-dimensional problem.

Figure 3.1 shows an agent standing in front of an open passage in an otherwise impenetrable wall. The green arrow symbolizes the agent's goal, but it cannot reach it with a single action because the wall blocks its way. Instead, it has to first step through the passage to be able to reach the goal with another action in the next time step.

The blue arrow shows the action taken, and the blue dot will be the agent's position at the next time step. The dashed blue and red square indicates the action space, the set of actions the agent can take. It centers on the agent's current position, and all actions will be position vectors relative to the current position. As a consequence, an action can be scaled and added to the agent's current position to obtain the position

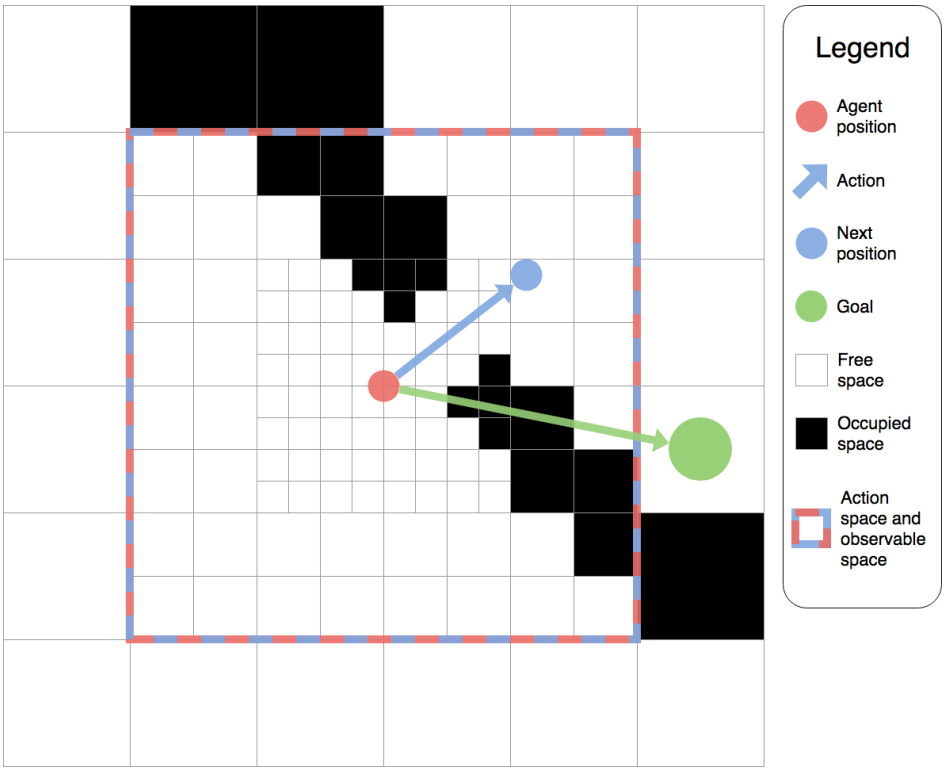

Figure 3.1: A 2D illustration of the simulated 3D environment. To reach the goal (green), the agent (red) has to step (blue arrow) through an opening (white) in the wall (black) to avoid collision. Notice that observation resolution decreases with distance from the agent.

at the next time step. In our case, the action space will be continuous instead of discrete. We describe actions further in section 3.1.

The large, dashed red and blue square symbolizes both the action space as well as the octree space observable by the agent. The pattern of variable-sized white or black backgrounds represent voxels corresponding to nodes in the octree. White indicates free space and black occupied or unknown space. In section 3.2 we describe how we use variable-resolution sampling to create observations from these voxels.

3.1 Actions

The action space used in this thesis is continuous, as the policy outputs real numbers as opposed to integers. The choice of this setup has several motivations.

The first is a desire for sample efficiency and larger step sizes. Consider the discrete alternative, which is to let the agent move from voxel to voxel by outputting one of six directions; Up, down, left, right, forwards, backwards. In such a setup a path from the starting position to the goal position consist of a series of transitions between voxels. For each time step, the agent chooses one of the six directions, so the number of possible paths the agent could take would be 6^t for t time

steps. Moving for ten time steps would require exploring more than 60 million possible paths. This is obviously infeasable.

By instead choosing a continuous action domain, the agent can learn to step over multiple voxels at a time. This should hopefully allow the agent to both explore less before finding a path, and also generalize by extrapolating new actions from a continuous space. In this thesis we use a continuous action domain for this reason. Using a continuous action domain would also be easier to apply to real world robots.

3.2 Observations

An observation holds the information upon which the agent can act. It is what the agent "sees" at any given time step. We will now define an *observation function* which takes an octree and the agents position and returns a one-dimensional observation vector suitable for feeding into a neural network. The observation space is the set of all possible observations. The observable space is the region of octree space the agent can observe through the observation function from its current position.

If we included the occupancy values of all voxels in the octree, the size of the observation vectors would be unmanageble, and defeat the purpose of octree compression. Instead we use variable-resolution sampling to strike a balance between the size of observable space and observation dimensionality. Figure 3.2 illustrates the three-dimensional sampling pattern we are proposing in two dimensions. The figure shows the agents position as a small red circle. The observation function is parametrized with a *visual range* parameter which controls the number of layers in the sampling pattern. Figure 3.2 shows the sampling pattern in the specific case when the visual range parameter is three. The size of the boxes, which double in length for each layer, illustrate the voxels are queried at lower and lower resolutions. Each layer in the pattern halves the resolution, so the resolution decreses with distance from the agent.

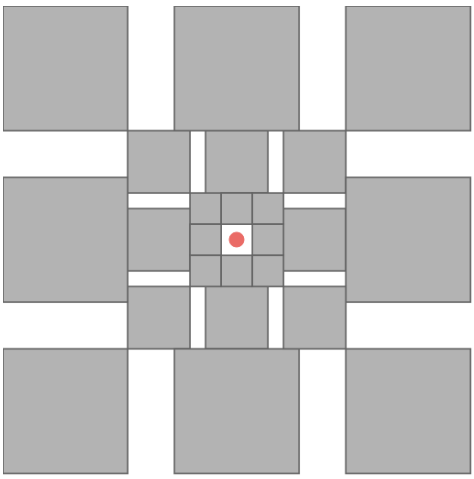

Figure 3.2: A 2D illustration of the proposed 3D sampling pattern with a visual range parameter of three.

Notice that the sampling pattern we define does not align with voxel boundries. This is a design choice which simplifies implementation of the observation function greatly. To implement the observation pattern, the observation function searches for the center positions of the voxel to

be queried with the depth given by the pattern. Positions are clamped by OctoMap to the voxel center position grid before being converted into query keys. This allows us to quickly sample voxels without having to consider the octree voxel alignment.

To define a general sampling pattern across all possible visual ranges, we use a inductive definition [1]. This is illustrated in figure 3.3. We define a visual range of zero as being "blind", in which case the agent observes no voxels at all and the observation function returns an empty vector. When the visual range is one the agent sees a dense pattern of the voxels surrounding it, leaving out the voxel the agent is currently in which will always be free. For each increment in visual range, an additional layer of voxel samples are added to the pattern.

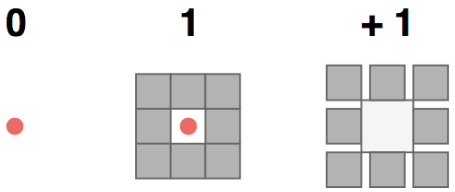

Figure 3.3: A 2D illustration of the inductive definition of the 3D sampling pattern.

Each node in an octree stores the clamped log odds probability of its voxel being occupied. When we extract features from voxels we can obtain two values. Either we can convert the log odds into regular probabilities in the range $[0,1]$, which gives us a continuous measure of occupancy. Otherwise we can use the octrees occupancy threshold, which is usually 0.5, to determine a discrete binary occupancy value. The voxel is occupied if the probability is above the threshold. These two variations, between continuous and discrete occupancy values, will be tested experimentally.

These “exteroceptive” [2] features resemble the random sampling used in some previous robotics research, notably in Emergence of Locomotion Behaviours in Rich Environments (Heess et al. 2017). One fundamental difference, however, is that we are sampling from aggregated data by decresing the tree depths the further away from the agent we sample. OctoMaps inherently integrate lower-resolution occupancy values over larger spaces. This aggregation is a key design feature that we employ to reduce the dimensionality of the observation space while still maintaining accuracy and observable range.

Figure 3.4 shows how visual range affects the size of the observable space as well as the number of dimensions in observation vectors. Because voxel sizes double for each additional layer, the observable space grows exponentially. The number of voxels, however, increase linearly with the number of layers. This is a useful property which possibly could allow the agent to observe very large spaces.

[1] A recursive definition of a set through initial elements and a step function
[2] Perceptions from external senses

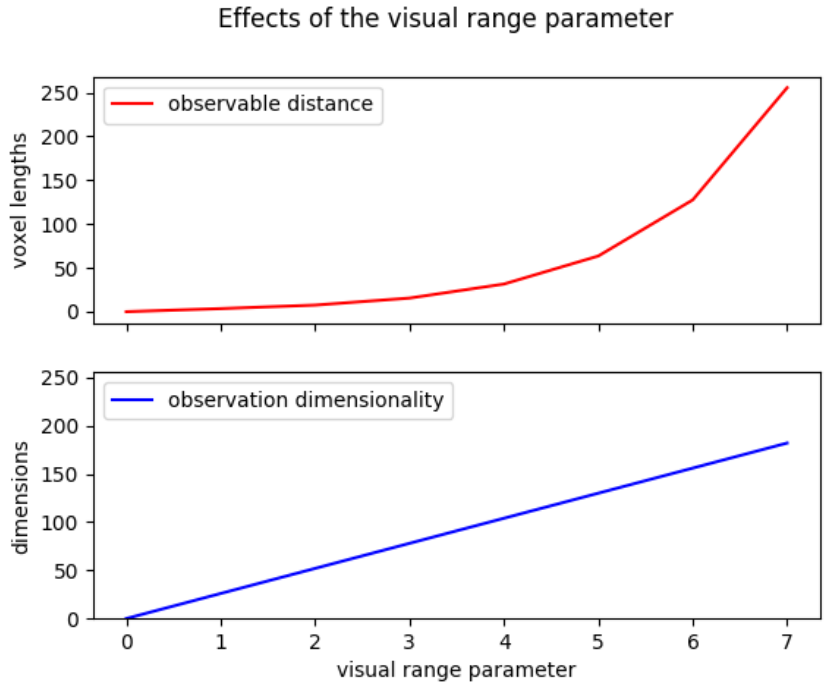

Figure 3.4: The observable range and observation dimensionality plotted as a function of the visual range parameter.

3.2.1 Goals

Hindsight Experience Replay requires observations to consist of three parts: a) an observation vector, b) a desired goal and c) an achieved goal.

We define the desired goal as element-wise difference between the goal position and the agents position. The achieved goal will similarly be the element-wise difference between the agent new position and its previous position.

3.3 Collisions

We want the agent to reach the goal but never collide, but this kind of multi-objective Reinforcement Learning is not trivial to implement. A naive approach of penalizing the agent for the rest of the episode introduces history and violates the markov assumption underlying RL. Instead we simply leave the agent in its current positions if it attempts to move in a way that would cause a collision. This causes the achieved goal to be zero, which we can handle as a special case when computing rewards in hindsight.

3.4 Rewards

The reward type-parameter controls the reward function so it can either return dense or sparse rewards. Because HER gives the agent access to

the reward function, it cannot directly calculate rewards from the state of the environment, but from achieved and desired goals as described above.

At the beginning of an episode, initial and goal positions are chosen within a given range. A goal is achieved when current position is equal to goal position, within a small tolerance.

Dense rewards, also known as shaped rewards, increase as the agent gets closer to the goal. They shape the agents behaviour towards the goal. Sparse rewards, on the other hand, occur only when the agent reaches the goal. They do not shape the agents behavior towards the goal, but the agent has to complete the goal in its entirety before a reward is given. We will experiment with both sparse and dense goals.

We define dense rewards as the negative distance between the agents position and the goal. This reward definition increases as the agent moves towards the goal, but does not consider obstacles and might lead the agent into a local minima.

We define sparse rewards as zero if the agent has reached the goal, -1 otherwise. When HER is used, the reward function can be called in hindsight to calculate goals. In this case the achieved and desired goals are identical, and the agent will always receive a rewards. There in one exception; If the achieved goal is zero the agent could have collided, and the reward will be -1 even though the desired and achieved goals are identical.

3.5 Implementation

We will now go on to describe the implementation of simulated navigation environments based on the method above.

3.5.1 Environments

OpenAI Gym Brockman et al. 2016, written in the Python programming language, has emerged as the de facto framework for implementing Reinforcement Learning tasks. We have implemented a handful of environment classes following the Gym programming interface. Each environment contains an octree with its three-dimensional Cartesian coordinate system specified in meters. We will refer to this as octree space. The state of the environment consists of two positions in octree space; The current position of the agent, which changes every time step, as well as a goal position that is constant throughout each episode. At the beginning of each episode, the initial and goal positions are sampled uniformly from experiment-specific regions of octree space.

Training consists of independent episodes each with a variable number of time steps. An episode ends after a maximum number of time steps, given as a parameter to each experiment, or when the agent reaches the goal, whichever comes first. Reaching the goal is defined as when the euclidian distance in octree space between the agent's current position and the goal position is lower than a configurable distance

threshold. Once the episode ends, the environment resets, and another episode begins.

Action space

Actions are three-dimensional vectors in actions space, a linear transformation of octree space centered on the agent's current position and scaled into units of maximum step size. If the agent, for example, chose the action [0.5, 0, 0], and the maximum step size was 10 centimeters, it would move in a straight line 5 centimeters along the X-axis in octree space. The observant reader will notice that the maximum step size defines a maximum along each independent axis in octree space, not a maximum cartesian distance across all three axes.

The choice of action space might seem arbitrary, but has several benefits; It can allow for scale-invariance and generalization so agents can transfer policies from one environment to another. Additionally, actions become more straightforward to interpret across different scales. Goal space will be a superset of the action space, described below.

Goal spaces

The environment presents the agent with a goal at each time step as part of the observation, and the agent then picks an action to try to achieve that desired goal. Once the environment has moved into the next state, the agent receives feedback on which goal the agent achieved. This achieved goal can then be used to compute a reward in hindsight which is stored as artificially generated experience in the experience replay buffer to increase sample-efficiency.

Both desired and achieved goals are defined in a superspace of the action space, meaning that they have the same scale (max step size) and origin (relative to the current agent position). Desired goals based on the distance between the agent position and the goal position in octree space, but scaled by the maximum step size. Consider an example with a maximum step size of 10 cm (0.1 meters), the goal position [1, 1, 1] in octree space and the agent at position [0, 0, 1]. The distance in octree space is [1, 1, 1] - [0, 0, 1] = [1, 1, 0], meaning that the agent has to move one meter along the X and Y axes to reach the goal. We scale by 0.1 meters to get the desired goal of [10, 10, 0] in goal space. We can immediately see that the agent requires at least ten steps to reach the goal because the goal unit is the maximum step size.

We must also define achieved goals to allow Hindsight Experience Replay to generate artificial experiences. Actual movement in octree space is the basis for achieved goals. We take the offset and map it to the same origin and scale as the desired goals, centering the space on the agents previous position and using units of maximum step size along each axis. An agent naturally cannot reach goals further away in a single time step, so the achieved goal space is identical to the action space. If an agent takes action [1, 0, 0] without colliding, it will have

achieved the goal [1, 0, 0].

Collision detection

When an agent attempts to move, we check for collisions by raytracing along a straight line from the original position to the target position. If all voxels in the path of the line are free, including the final voxel, the move is allowed. Otherwise the move would cause a collisions, and the agent stays in its current position. In the case of a collision, the achieved goal is set to zero, and the reward function will not reward such a move even when HER applies goal substitution.

3.5.2 Agent

The agent, also utilizing the Gym interface, comes from OpenAI Baselines Dhariwal et al. 2017. Baselines is an open source code repository containing agent implementations from several papers by OpenAI. The agent used in this thesis builds on Hindsight Experiment Replay.

HER was initially used to control a robot arm for picking or pushing objects in a simulator. The agent used a universal value function-based policy outputting continuous position offsets as actions. Those robot arm tasks were in some aspects similar to the ones we will experiment on in this thesis, making the Baselines HER implementation an obvious choice. A substantial benefit of using an existing implementation lies in avoiding a vast number of potential implementation bugs. With tried and tested code we can eliminate most, if not all, sources of error from the agent implementation and instead focus on other aspects likely to affect the experiments.

Parallelism

To be able to execute the experiments in a reasonable amount of time we require parallelism. The implementation uses multiprocess parallelism with OpenMPI, just as in the original HER experiments. Unlike the original experiments which used 19 cores, these experiments ran with 12 hyper-threads on six physical cores. The number of cores affects the training results because gradients average across all running cores, and fewer cores can increase the magnitude of gradients. This effect from the number of cores is significant for reproducing results, and thus all experiments are run on the same number of cores/hyper-threads.

Hyperparameter tuning and Neural Network Architecture

The policy uses three hidden layers of 256 nodes per layer, just as in the original HER experiments. We have used the same hyperparamers for DDPG and HER as in the original HER experiments. These are listed in appendix B. We have not performed hyperparameter tuning

nor experimented with different architectures due to limited time and computational resources.

4 Results

This chapter describes some learning tasks and the experiments performed on them. We use the implementation described in chapter **??** based on the methods described in chapter 3. The two overall goals of these experiments are to *a*) provide a baseline of which tasks the method can or cannot solve, and *b*) see how different parameters affect the success rate on the same learning task. The learning tasks, parameters, experiments, and results will be presented separately for each task in section 4.4.

4.1 Training and testing

Each experiment is executed five times in sequence with different random seeds to control for the effects of randomness and account for variation when measuring success rates. Each experiment runs for 100 epochs per seed, with one epoch consisting of 100 episodes trained in parallel on each of 16 cores, for a total of 1600 training episodes per epoch and 160000 training episodes per experiment seed. Each core executes the experiment independently, with separate random seeds, experience replay buffers, policy networks, seeds, and environments. Policy weight gradients, however, are averaged across cores to combine their shared experience. At the end of each epoch, each core independently tests its current policy on 20 episodes, and the average success rate across all cores and test episodes is recorded for that epoch. Policy weights are randomly initialized for each core at the beginning of the experiment.

4.2 Evaluation

Each episode succeeds if and only if the agent reaches the goal within a small distance tolerance. Policies are evaluated at the end of each epoch by running them without random exploration on 20 test episodes per core, for a total of 120 test episodes per epoch. The average success ratio is recorded as a measure of policy performance at that epoch, and we can plot the success ratio as a function of epochs. Because we are running five separate experiments with different random seeds, we plot the mean average success rate across all five experiments as well as the 95th percentile confidence interval. Re-running experiments

with multiple random seeds is in line with the recommendations by Henderson et al. 2017, which suggest best practices for Reinforcement Learning research.

4.2.1 Metrics

We will evaluate the policy using five metrics.

Success rate during training as a function of time. During training, the policy should improve. We can measure improvement with the ratio of successful test episodes at the end of each epoch. This metric also allows us to compare the rate of improvement for different experimental setups, e.g. when comparing sparse and dense rewards. We train the policy five time with different random seeds, and plot the median success rate as a function of epochs, as well as interquartile range as a measure of variation. Values are smoothed over five time steps to reduce noise. Some experiments have a zero success rate throughout training. The success rate plots are omitted for those experiments.

Distance to goal at each time step. After training a policy, we can evaluate it on one or more episodes.

Cosine similarity between action and desired goal. The policy takes a vector with the relative position of the goal, and outputs an action with a relative position to move to. If there are no obstacles, the most efficient direction to move in is directly towards the goal. The cosine similarity between the two vectors give us a measure of how close their directions are, regardless of their magnitudes. Calculating cosine similarity yields a number in the range $[-1, 1]$, with 1 meaning the directions of the vectors are the same, -1 meaning they are opposite, and 0 that they are orthogonal.

Action magnitude relative to longest possible step in direction of goal. While cosine similarity gives us a measure of direction, we're also interested in the *magnitude* of actions. To normalize across directions we calculate the action vector magnitude divided by the magnitude of the relative goal vector limited to the maximum step size. This gives a ratio in the range $(0, 1]$, with 0 indicating no movement, and 1 indicating an optimal step size if there are no obstacles.

Cumulative number of collisions Measuring the cumulative number of collisions gives us a picture of when and how often policies cause the agent to crash. As policies are deterministic when not training, a crash indicates the end of an episode. The metric in itself is not all that useful, but when displaying information from multiple episodes with different seeds, can give and indication of weather policies tend to cause collisions or not.

Some learning environments are deterministic, and other randomized. Deterministic environments are those with fixed start and goal positions. Randomized environments have either a randomized starting position, a randomized goal position, or both. When evaluating policies on deterministic environments, a single trial is ran. On ranomized environments, however, multiple episodes can be run with different random seeds. For randomized environments, we run 100 episodes and plot the median values as well as the interquartile ranges.

In addition to the metrics above, we also visualize the octree and movement of the agent in octree space. These plots show a 2D projection along each axis of the 3D octre space. Each voxel, in red, indicate average occupancy. The agents starting position is indicated with a blue circle, and the goal position with a green circle. Movement at each time step is indicated with a blue line ending in a cross. If the agent collides, a larger, red cross is shown at the position where the colliding movement was taken. The maximum step size is symbolized with a red square centered on the agents position.

Because OctoMap uses probabilistic sensor measurements, some voxels near walls are free or occupied even when point cloud data suggeest that they should not be. This is reflected in octree plots as white pixels outside walls, or colored pixels in the middle of an otherwise empty room. Partially colored pixels do not mean that all voxels are occupied, nor that occupied voxels float in empty space. They simply indicate that *some* voxels are occupied along the aggregated timension, usually outside or directly inside the walls.

4.3 Parameters

A handful of parameters control each experiment. Here we describe the parameters and what part of the experiment they control. Note that the parameters controlling programmatic octree scene creation are not applicable to experiments run on pre-created octrees. These parameters are the resolution as well as the room size and origin.

max episode steps The episode ends after this many steps, or when the goal is reached, whichever comes first.

max step size The maximum distance of octree meters which the agent can move *along each of the three octree axes*. The max step distance effectively controls the scale of the action and goal spaces with regards to the octree space.

visual range A parameter controlling the observable space and number of sampled voxels. Described in section 3.2.

resolution The length of a voxel measured in octree meters, only applicable when generating octrees. If the resolution is one octree meter, each voxel will occupy a 1 x 1 x 1 meter-sized region of octree space.

Room size and origin Applicable only when generating an octree with an empty room inside four walls, floor, and ceiling. The parameter gives the size in octree space meters between the walls, and the origin tells where the center of the room is in octree space. A room size of (10, 2, 2) centered on origin (5, 1, 1) will make a corridor starting at (0, 0, 0).

Start range Two points in octree space which define the region from which the initial position is uniformly sampled. If both points are the same, the initial position is constant.

Goal range Two points in octree space defining the uniform goal sampling region, similar to the start range.

Distance threshold A Euclidian distance in octree space. If the agent moves closer to the goal than this distance, the episode ends immediately and is considered a success.

Reward type Sparse or dense. If the reward is sparse, a binary 0 or -1 is given for being within the goal distance threshold or not, respectively. If dense, the reward is the negative Euclidian distance to the goal. Described in section **??**.

Observation type Continuous or discrete. Discrete observations use booleans 0 or 1 to indicate free or occupied voxels, respectively. Continuous observations contain collision probabilities as modeled by the octree. Described in section 3.2.

4.4 Tasks

The learning tasks were chosen to evaluate different aspects of the model at increasing difficulties. Experiments are run with different rewards types and observations, and success rates are plotted individually. We'll give an overview of each experiment here before describing them in detail in individual sections.

Fixed start, random goal, no obstacles (section 4.4.1)
This first experiment places the agent in the center of a room without obstacles. A goal is randomly sampled from within two maximum step sizes from the agent, and the agent has to reach it in two time steps. Different reward and observation functions are used and compared against each other. The experiment provides an initial indication of the feasability of the method used, as well a comparison of reward and observation types.

Fixed start and goal positions, no obstacles (section 4.4.2)
The second experiment occurs over 100 time steps in a larger, empty room. The start and goal positions are fixed at opposite ends of the room, and there are no obstacles in between. The

experiment gives an indication of how prone the method is to get stuck in local minima, as well as the model's ability to overfit.

Randomized start and goal positions, no obstacles (section 4.4.3)
The third experiment is similar to the second, but randomizes the start and goal positions so they are opposite ends of the room. There are no obstacles between them. The experiment is intended to measures how well the model generalizes.

Fixed start and goal positions, with obstacle (section 4.4.4)
The fourth and final experiment is similar to the second, in that start and goal positions are fixed at opposite ends of the room. This time there is an obstacle, a floating wall with blocking the center path. There are four openings through which the agent can pass. This final experiment compares two different observation and step sizes (near and far), unlike the earlier experiments. The experiment measures if the model able to avoid collisions through its observation.

The first experiment differs from the rest by having only two time steps, a smaller room size, and also compares combinations of different reward and observation functions.

The rest of the experiments are similar to each other and vary only in starting and goal positions, the presence of absence of a wall in the center of the room, as well as the step size (for the final experiment). All use sparse rewards and continuous observations because they performed favorably in the first experiment. The experiments also take place in a larger room and gives the agent up to 100 time steps to complete an episode.

4.4.1 Random directions

In this task, illustrated in figure 4.2, the agent starts at a fixed position in the middle of a small, empty room. Goals are sampled randomly from within the room so that goals can be in any direction from the agent. With the goal sampling space being twice the size of the action space, some goals will be reachable in one time-step and some in two time-steps. The agent has two time-steps to complete the goal.

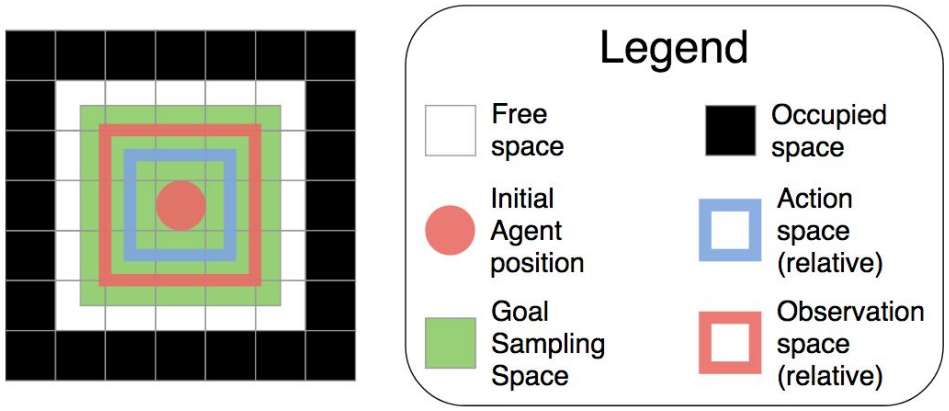

Figure 4.1: A 2D illustration of the 3D random directions task. The agent starts in the same position (red circle) every time. The goal position is sampled from within the green square, which can give a goal in any direction from the agent. The blue square indicates the action space, and the red the observation space. Both are relative to the agents position at any time step.

The task is intended to provide a baseline for how quickly (and if at all) the policy network learns to match actions with goals. Because there are no obstacles between the agent and the goal, the occupancy-parts of the observation are irrelevant, and the agent can always move towards the goal in a straight line. The experiment also compares different reward and observation types. We run the experiment with four different combinations of sparse and dense rewards, as well as discrete and continuous observations.

Parameters

Parameter	**Value**
max episode steps	2 timesteps
max step size	1 meter
resolution	1 meter
room size and origin	5 x 5 x 5 meters centered on (0, 0, 0)
start position	(0, 0, 0)
goal range	(-2, -2, -2) to (2, 2, 2)
distance threshold	0.5 meters
visual range	1
reward type	sparse or dense
observation type	continuous or discrete

Outcome

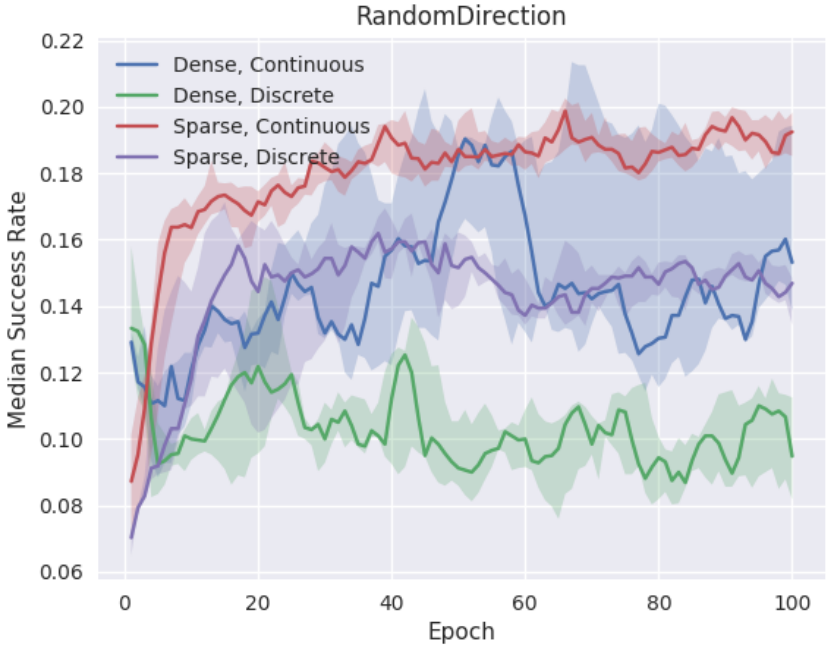

Figure 4.2: Median success rate as a function of epoch on the random directions task, with shaded area indicating interquartile range over five trials with different random seeds. Values are smoothed over 5 epochs to decrease noise.

We trained four variations of policies with five different random seeds each for 100 epochs on 32 cores, for a total of 6400000 episodes. Figure 4.2 compares success rates for the four different variations. We immediately notice that none of the trained policies achieve a higher success rate than 20%. Also, the experiment with sparse rewards and continuous occupancy observations achieve the highest success rate with a small interquartile range. The lowest score is obtained with dense rewards and discrete observations. In between lie the last two observations at similar values. However the variation with dense rewards and continuous observations have the highest variation, much higher than the others.

Figures 4.3 and 4.4 visualises the final policy for the sparse, continuous experiment. We can see from figure 4.4 that agents do not collide, but that they also move away from the goal. The first step moves towards the goal, but the second moves away. This is reflected in both the goal distances as well as the cosine similarities.

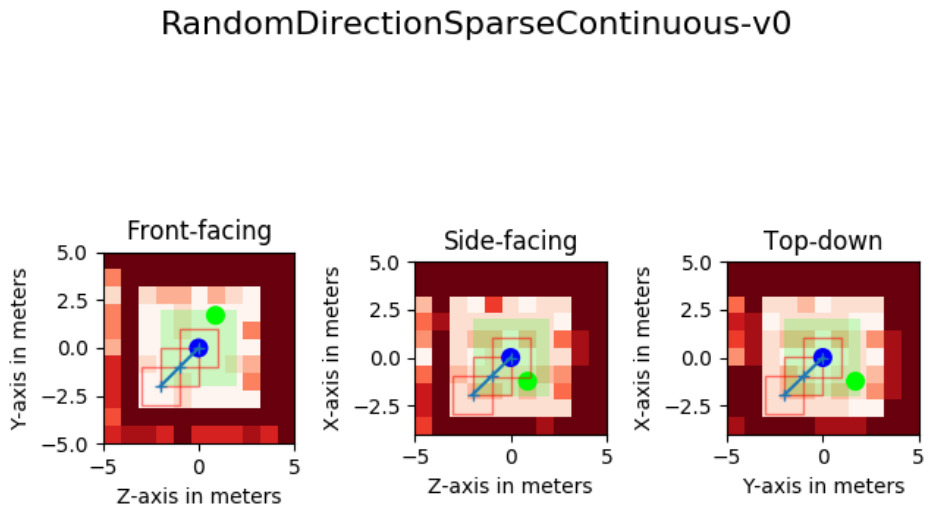

Figure 4.3: A visualization of an agent's movement in a single episode using a policy trained for 100 epochs.

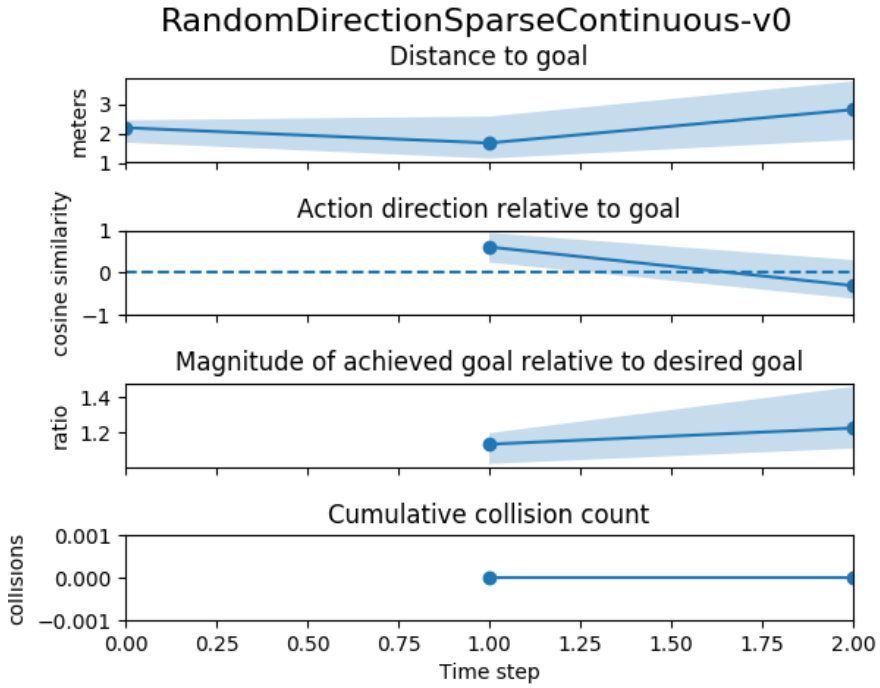

Figure 4.4: Visualization of the octree and a trained policy on the sparse, continuous random directions environment.

4.4.2 Empty corridor with deterministic positions

This experiment places the agent in a corridor without obstacles. Both the starting and goal positions are constant. This experiment tests if a policy can overfit to an environment. Testing for overfitting will give us an indication of the feasability of the proposed method, particularily for using HER with continuous domains.

Parameters

Parameter	**Value**
max episode steps	200 timesteps
max step size	3.5 meters
resolution	1 meter
room size and origin	20 x 10 x 10 meters centered on (0, 0, 0)
start position	(-7.5, 0, 0)
goal position	(7.5, 0, 0)
distance threshold	0.5 meters
visual range	1
reward type	sparse
observation type	continuous

Outcome

Training for 100 epochs yields a zero percent success rate, even with five different random seeds. In figure 4.5 we can see the agent moving straight into the wall. This is illustrated with a large red X that represents a collision. We will discuss why this happens in chapter 5.

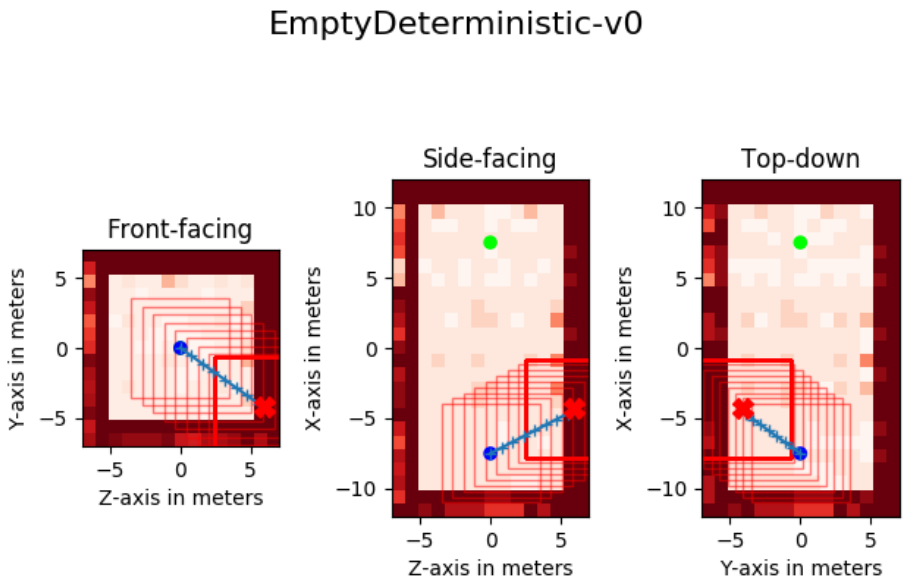

Figure 4.5: Visualization of an episode after training in an empty environment with fixed positions.

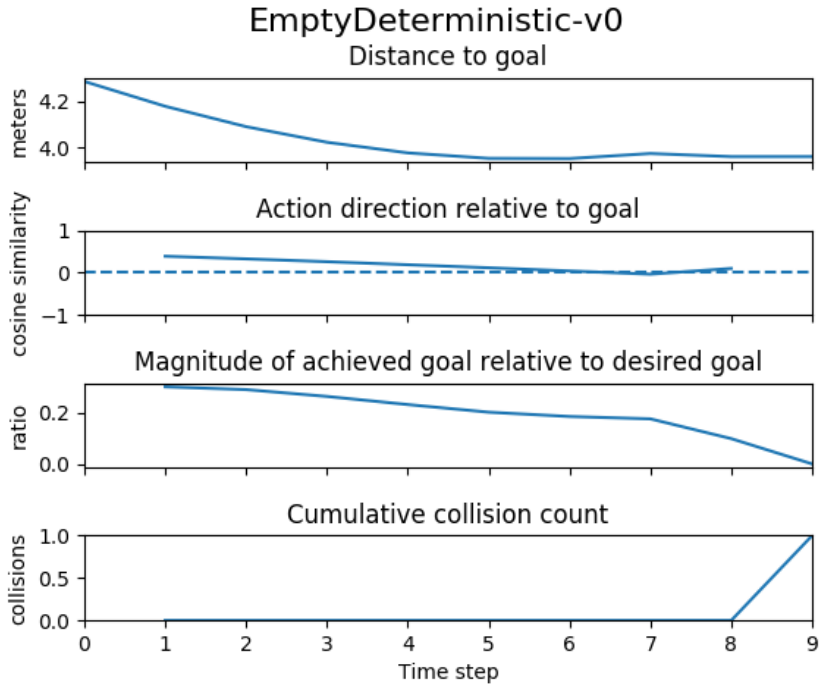

Figure 4.6: Evaluation metrics for one episode on a trained policy for the empty, deterministic environment.

4.4.3 Empty room with randomized positions

This experiment randomizes starting and goal positions to avoid overfitting. The start area is in one end of the room, and the goal area in the other. At the beginning of each episode a start and goal position is uniformly sampled from the start and goal areas.

Parameters

We use the same parameters as in section 4.4.2, but with the following changes.

Parameter	Value
start range	(-8, -4, -4) to (-7, 4, 4)
goal range	(7, -4, -4) to (8, 4, 4)

Outcome

Figure 4.7 shows an agent failing to move from the start to the goal position during a single episode. Instead, the agent collides with the wall. Figure 4.8 shows us several things. For one, the policy collides after a small number of time steps, never managing to complete the 200 time step episode. Secondly, distance to goal decreases for a few steps, and then stops decreasing. The step direction is slightly in the direction of the goal.

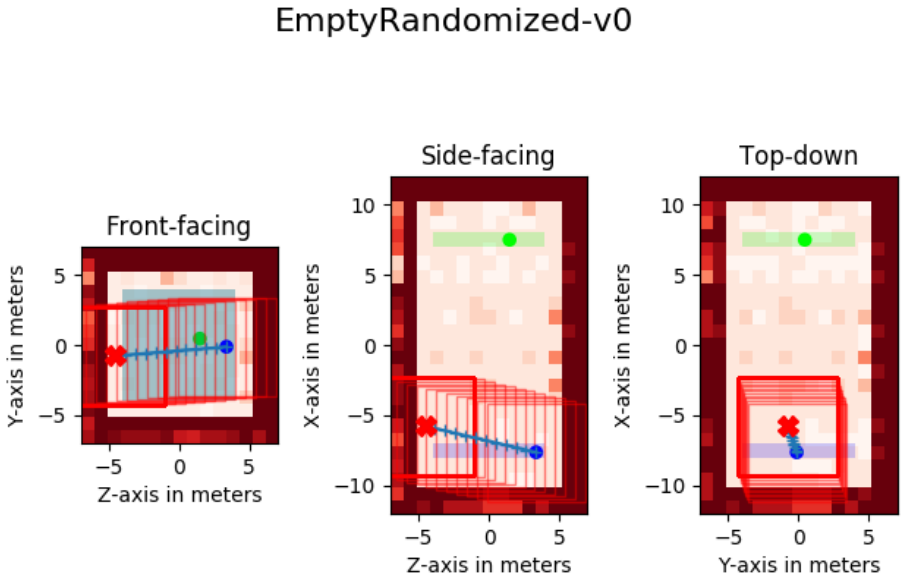

Figure 4.7: Example of policy trying and failing to move from a randomly chosen position at one end of the room to a randomly chosen goal position at the opposite end.

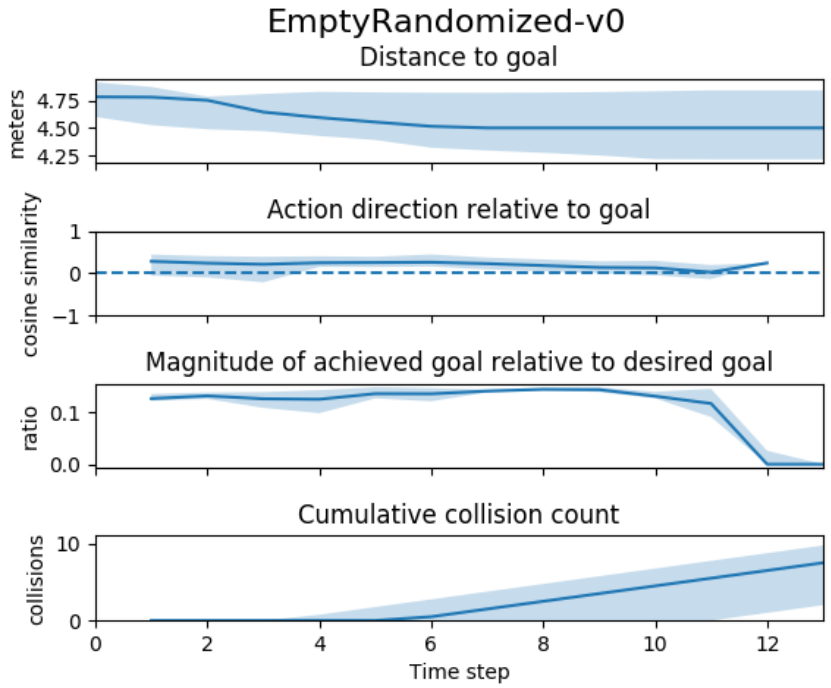

Figure 4.8: Evaluation metrics on 100 episodes of the EmptyRandomized task with a trained policy. Lines show median values, shaded areas show interquartile ranges.

4.4.4 Navigating around a wall from fixed positions

In this experiment we use the same setup as in the empty room with fixed positions, but insert a wall in the center. We test with two different visual ranges and corresponding step sizes. The "near" visual range is the same as in the previous experiments. The "far" visual range is slightly larger than the octree space.

Parameters

We use the same parameters as in section 4.4.2, but with the following modifications.

Parameter	Value
wall range	(-1, -4, -4) to (1, 4, 4)
visual range	1 or 3

Outcome

After training for 100 epochs, the success rate is zero.

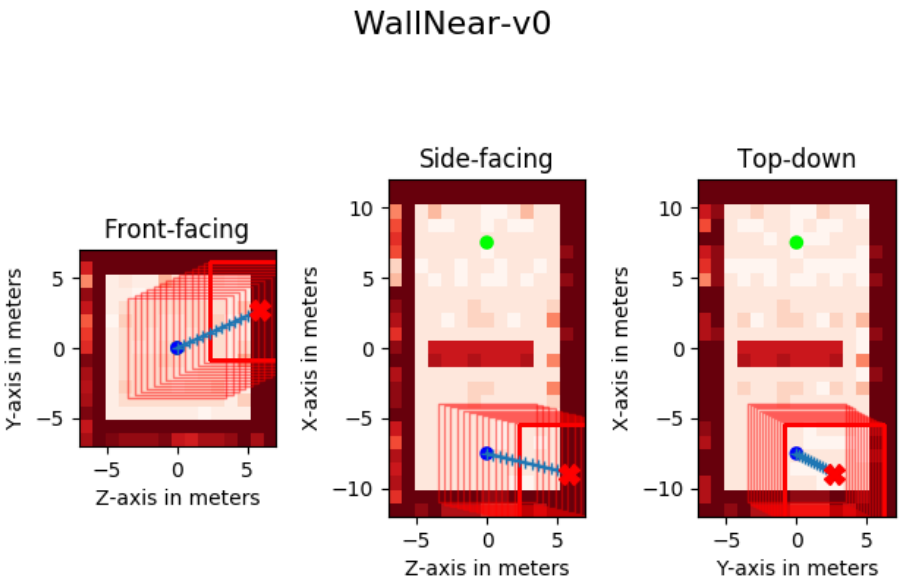

Figure 4.9: Visualization of a trained policy in a walled environment with "near" observations.

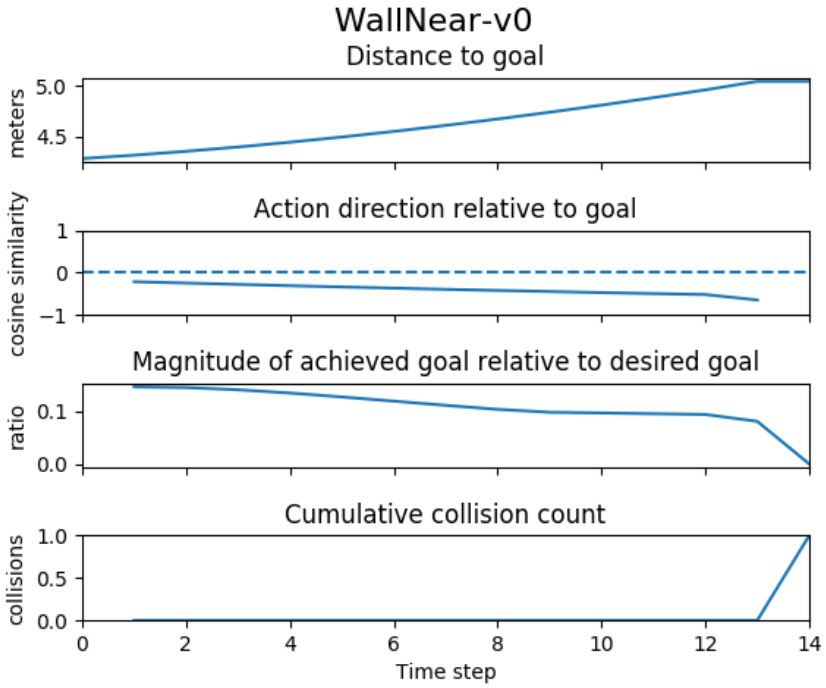

Figure 4.10: Evaluation metrics on an episode of the "near" walled environment with a trained policy

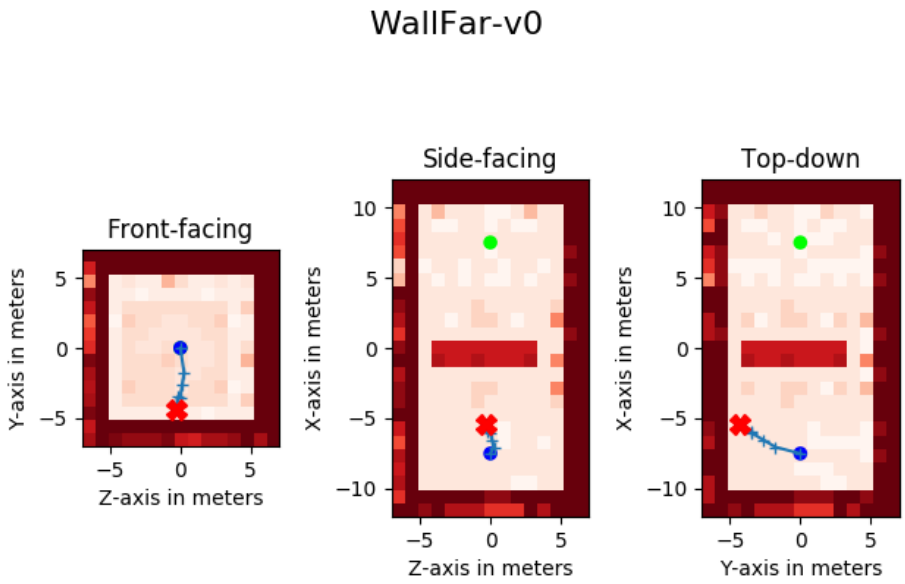

Figure 4.11: Movement of a trained policy on the "far" walled environment.

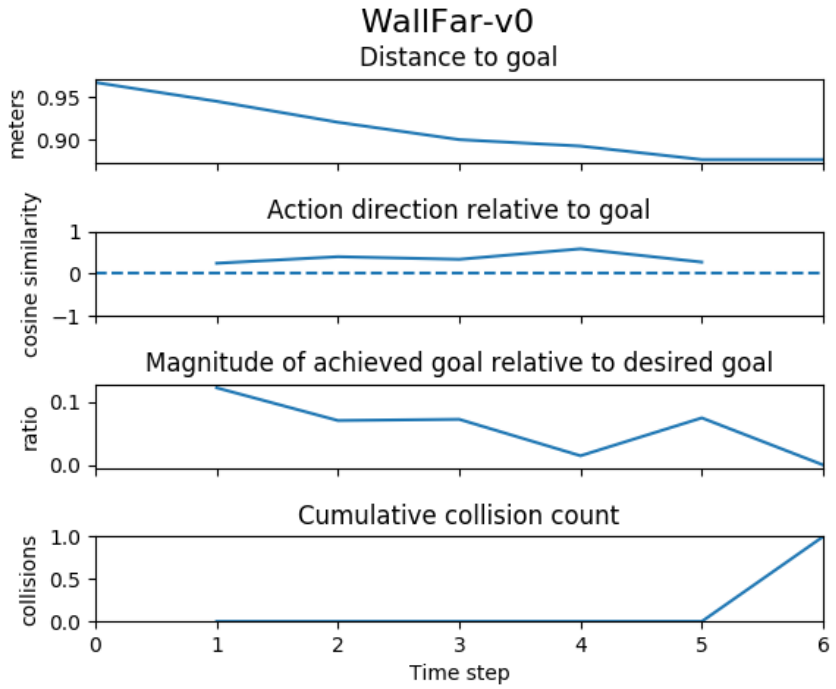

Figure 4.12: Evaluation metrics on an episode of the "far" walled environment with a trained policy.

5 | Discussion

The goals of this thesis were to design and implement a method of applying Reinforcement Learning to octree navigation, assess the feasability and properties of the new method experimentally, and finally explore how different environment properties affect learning outcomes.

Reinforcement Learning was applied in simulation with sampling-based octree observations and continuous action spaces, as well as with goals suitable for use with Hindsight Experience Replay. The resulting Python implementation conformed to the OpenAI Gym interface.

The metod was tested experimentally using an off the shelf baseline DDPG implementation, with and without HER, with sparse and shaped rewards, on continuous and discrete occupancy observations, as well as with different step and observations sizes.

Experiments followed two broad lines; The first line of experiments compared four combinations of reward and observation functions on very short episodes with randomized goal positions. The second line of experiments used the best performing observation and reward functions from the first, but we will get back to that in section 5.2 after discussing the results of the first line of experiments.

5.1 Reward and observation functions

Figure 4.2 shows that sparse rewards combined with continuous observations yielded the greatest success rates. This is in line with the findings by Andrychowicz et al. which showed that sparse rewards combined with HER outperformed shaped rewards without HER. The improved succes rates might arise from three properties of HER: a) The end goal is more clearly defined with sparse rewards than with dense, b) HER introduces data-augmentation which increases both the number and diversity of samples the neural networks trains on, and c) HER acts as an implicit curriculum shaping the agents behaviour towards better generalization over goals.

Dense rewards combined with discrete occupancy observations had the lowest success rates. Why discrete observations has this effect is unclear, but might arize from the fact that that neural networks are unable to generalize well over what is effectively sparse and dissimilar input.

Two combinations performed similarly: a) Dense rewards with con-

tinuous observations, and b) sparse rewards with discrete observations. The former has a much larger interquartile range, suggesting that results are highly affected by randomness. The latter has a slightly higher and less variable success rate.

Even the best-performing combination of reward and observation functions plateued at less than a 20% success rate. This low success rate coninus in the remaining experiments. We will discuss the possible causes in section 5.3.

5.2 Larger octree spaces

The other line of experiments used a larger octree space with and without obstacles, with deterministic and randomized start and goal positions, as well as with different step sizes. All these experiments yielded a zero percent success rate, meaning that the agent never reached the goal, suggesting issues with methodology or implementation. To obtain qualitative and quantitative measures of the training outcomes regardless, trained policies were visualized and evaluated on a handful of metrics described in section 4.2.1.

The looking at the metrics across all experiments seem to suggest that the agent is unable to learn effective actions for accomplishing even immediate goals. The step sizes are generally low, and the cosine distances do not correlate with the direction of the goals.

The experiment with fixed positions and no obstacles, described in section 4.4.2, is meant to test the models ability for rote learning. The policy initially moves the agent closer to the goal, but then begins moving orthogonally to the goal direction, decreases its step size and finally collides with the wall.

5.3 General discussion

Essentially, as many RL researchers before us, we are left with a fundamental problem of attribution. There are a vast number of factors that can have contributed to the poor results in our experiments, and the difficulty of figuring out which compounts when using RL and neural networks. We will go into some of them here.

Randomness Reinforcement Learning is very susceptible to randomness, as pointed out by Henderson et al. 2017 in their research into reproducability and methodology when working with RL.

Local minima Because model-free RL requires the agent to explore a lot, and the state space is continuous, the agent might not be able to explore enough to train a suitable policy. Instead, the policy might get stuck in a local minima, finding a sub-optimal solution and never improving to the optimal solution.

Collision handling When the agent attempts a move that would cause a collision, it does not change position at all. This design choice might even worsen the problem of local minima, because some actions are essentially no-ops.

Choice of neural network architecture The neural networks used in the agent have the same architectures as in the original HER experiments. Changing the number of layers, number of hidden nodes per layer, and the activation functions used, might result in different outcomes. This has not been explored at all in this thesis, and the poor experiment results might be caused by insuffucient or overly complex model capacity.

Lack of hyperparameter tuning Hyperparameters have not been tuned. Hyperparameters like learning rates, exploration rate, and weight penalty (for regularization) can strongly affect the outcome of RL experiments. There are several possible methods for hyperparameter tuning, but a random search can yield good results. This has not been done due to limited computation resources.

The experiments overall do not indicate that RL applied to octree navigation is a promising line of research. On the other hand, the results also do not indicate that the method could not eventually be feasable given further research. Because RL introduces many confounding variables that are difficult to account for when designing experiments, the results are inconclusive as to whether this approach is promising or not.

The thesis did, however, succeed in formulating a novel RL approach to 3D navigation with octrees, as well as provide a working implementation in the form of an OpenAI Gym environment. The experiments can serve as a baseline for further research, and might point in a direction of research that can eventually bear fruit.

This thesis provides a novel research contribution by exploring a new application of Reinforcement Learning that has not been published before.

If we go back the SAR drone scenario described in chapter **??**, the methods proposed in this thesis would not be suitable due to reasons mentioned above. A better solution with our current technology might be traditional path planning algorithms.

5.4 Future work

This section contain ideas that have emerged in the late phases of the thesis and that could potentially lead to further improvement.

5.4.1 Experiments on network architecture and hyperparameters

The method might be improved through different hypeparameter settings or different network architectures. For hyperparameter tuning, a random search might be applied. For network architecture different activation function like ReLu could yield better results, as well as deeper networks with more layers, weights per layer or both. Such a line of research would simply require more time and computing resources.

5.4.2 Scale-invariant navigation

The method proposed in this thesis uses normalization to formulate resolution-invariant action and goal spaces in units of maximum step size. Instead of using a constant normalization factor throughout the episode, the environment could variate the scale at every time step. This would effectively change the depth at which observations are sampled, scaling voxel sizes down or up along with step sizes. The resulting policy would be scale-invariant without any modification to the agent. When the agent traverses highly compressed areas of the octree space, with shallow leaf nodes, the step size would naturally increase.

5.4.3 Observing through raycasting

An alternative kind of observation function could be implemented using raycasting. The agent could observe its environment through a finite-sized point cloud. We can imagine a bunch of rays shooting out from the position of the agent, and each ray records the distance until it reaches an obstacle. The observation would be the vector of these distances. These observations would let the agent "sense" distance, but because they stop at the first obstacle the agent would not be able to receive information from beyond walls. Some nearby obstacles could also escape observation if they fall just outside of the sampling rays. These observations might be more time-costly to compute but provide features could be easier to learn from for the agent. Raycasting could possibly scale better and give the agent access to a larger observable space.

5.4.4 Application to a robot arm

Some recent RL experiments train robot arms in end-to-end policies from camera input to joint angles or torques. Octomaps could be tested as a component in this kind of setup. Such experiments could set a robot arm at an initial position, and train it to move towards a fixed goal with some static obstacles in between. Figure 5.1 illustrates the experiment setup.

The method described in this thesis could be (heavily) adapted to this type of application.

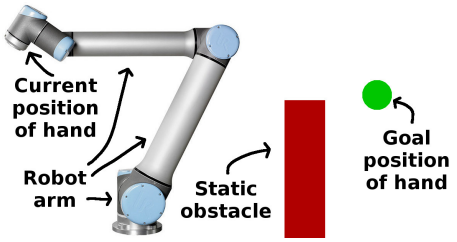

Figure 5.1: Illustration of a possible experiment setup. The goal of the agent would be to move the hand of the robot arm to a target position without colliding with the obstacle.

5.4.5 Observability

The method in the thesis introduces partial observability by only letting the agent observe the region of octree space near the agent. The thesis does not deal with this partial observability, but could have done so. To create general policies that could navigate any octree, the agent would have to either use memory or observe the entire octree space.

Memory can be implemented with recurrent neural networks. This would essentially require observations from multiple time steps to be fed into the network before taking an action from the final time step.

We could give the agent full observation of the environment through low-resolution observations of the entire octree space. The experiment 4.4.4 with a "far" visual range does this to some extent, but the effects of this are not explored at all in this thesis.

5.5 Conclusion

In this thesis we designed a method for applying Reinforcement Learning to 3D octree navigation using Hindsight Experience Replay, and implemented the method as OpenAi Gym environments. Experiments were carried out and showed that the combination of sparse rewards with continuous observations yielded higher success rates. Success rates were low on simple tasks and zero on more complex ones. The causes of the low success rates were not determined conclusively, and the results do not allow us to conclude if applying RL to octree navigation is feasable. Additional research is needed and might produce positive results.

Bibliography

Andrychowicz, Marcin et al. (2017). “Hindsight Experience Replay.” In: arXiv: 1707.01495.
http://arxiv.org/abs/1707.01495
(cit. on pp. 10, 11, 13, 39).

Brockman, Greg et al. (2016). “OpenAI Gym.” In: arXiv: 1606.01540.
http://arxiv.org/abs/1606.01540
(cit. on p. 20).

Dhariwal, Prafulla et al. (2017). *OpenAI Baselines*. \url{https://github.com/openai/baselines}
(cit. on p. 22).

Heess, Nicolas et al. (2017). “Emergence of Locomotion Behaviours in Rich Environments.” In: arXiv: 1707.02286.
http://arxiv.org/abs/1707.02286
(cit. on p. 18).

Henderson, Peter et al. (2017). “Deep Reinforcement Learning that Matters.” In: arXiv: 1709.06560.
http://arxiv.org/abs/1709.06560
(cit. on pp. 9, 15, 26, 40).

Hornung, Armin et al. (2013). “OctoMap: an efficient probabilistic 3D mapping framework based on octrees.” In: *Autonomous Robots* 34.3, pp. 189–206. ISSN: 0929-5593. DOI: 10.1007/s10514-012-9321-0.
http://link.springer.com/10.1007/s10514-012-9321-0
(cit. on pp. 5, 6).

Langston, Rosamund F et al. (2010). “Development of the spatial representation system in the rat.” In: *Science (New York, N.Y.)* 328.5985, pp. 1576–80. ISSN: 1095-9203. DOI: 10.1126/science.1188210.
http://www.ncbi.nlm.nih.gov/pubmed/20558721
(cit. on p. 1).

Lillicrap, Timothy P. et al. (2015). “Continuous control with deep reinforcement learning.” In: arXiv: 1509.02971.
http://arxiv.org/abs/1509.02971
(cit. on p. 11).

Mahmood, A. Rupam et al. (2018). “Setting up a Reinforcement Learning Task with a Real-World Robot.” In: arXiv: 1803.07067.
http://arxiv.org/abs/1803.07067
(cit. on p. 15).

Schaul, Tom et al. (2015). “Universal Value Function Approximators.” In: *Proceedings of The 32nd International Conference on Machine Learning*, pp. 1312–1320. ISSN: 1938-7228. http://machinelearning.wustl.edu/mlpapers/paper%7B%5C_%7Dfiles/icml2015%7B%5C_%7Dschaul15.pdf%20http://jmlr.org/proceedings/papers/v37/schaul15.html (cit. on p. 10).

Wikipedia:Vossman (2017). *File:Voxels.svg - Wikipedia*. https://en.wikipedia.org/wiki/File:Voxels.svg (visited on 05/15/2018) (cit. on p. 5).

Wikipedia:WhiteTimberwolf (2010). *File:Octree2.svg - Wikipedia*. https://en.wikipedia.org/wiki/File:Octree2.svg (visited on 05/15/2018) (cit. on p. 6).

A | Code

A.1 Environments

```python
# This implementation builds on ideas and a few minor code snippets
# from the robotics environments in OpenAI Gym, particularily from
# https://github.com/openai/gym/blob/v0.10.3/gym/envs/robotics/fetch_env.py
# Gym is distributed under the MIT license. See LICENSING for details.

from itertools import product, starmap, repeat
import os
from pprint import pprint
from collections import OrderedDict
from collections import namedtuple
from functools import partial

import numpy as np
import gym
import gym.utils.seeding
import octomap
from baselines import logger

registry = gym.envs.registration.EnvRegistry()

# Create a helper function to load octree files from disk
data_path = partial(os.path.join, os.path.dirname(__file__), '..', 'data')

def euclidian_distance(a, b):
    return np.linalg.norm(a - b, axis=-1).astype(np.float32)

class OctreeEnvironment(gym.GoalEnv):
    def __init__(self, *, octree, start_range, goal_range, distance_threshold,
                 visual_range, reward_type, observation_type, max_step_size=None,
                 depth=None, end_on_collision=False):
        self.end_on_collision = end_on_collision

        if callable(octree): # Factory function
            log_dir = logger.get_dir()
            path = log_dir and os.path.join(log_dir, 'environment.bt')
            if log_dir is None: # If we're running from py.test
                octree = octree()
            elif not os.path.exists(path): # First call during this experiment
                octree = octree() # Actually create the octree
                # Write the tree from file
                logger.info(f"Writing octree to {path!r}...")
                with open(path, 'wb') as f:
                    f.write(octree.writeBinary())
            else: # The original tree was already created
                octree = path # Read it below

        if isinstance(octree, str): # Path to octree file
            assert octree.endswith('.bt') or octree.endswith('.ot')
            open(octree, 'rb').close() # Test that the file can be opened
            octree = octomap.OcTree(octree.encode())
```

```
51
52          assert isinstance(octree, octomap.OcTree)
53          self.tree = octree
54
55          assert visual_range >= 0
56          self.visual_range = visual_range
57          self.depth = depth or self.tree.getTreeDepth()
58          assert 0 < self.depth <= self.tree.getTreeDepth()
59          self.resolution = (self.tree.getResolution()
60                             * (2 ** (self.tree.getTreeDepth() - self.depth)))
61          assert max_step_size is None or max_step_size > 0
62          self.max_step_size \
63              = np.float32(max_step_size \
64                           or (self.resolution * ((2 << visual_range) - 0.5)))
65
66          # Start and goal ranges can be either a single position (shape
67          # 3) or a range of positions (shape 2 x 3). Broadcasting to
68          # here will cover both cases correctly when sampling uniformly.
69          self.start_range = np.broadcast_to(start_range, (2, 3))
70          self.goal_range = np.broadcast_to(goal_range, (2, 3))
71          # TODO: Verify that goal and start ranges are free of obstacles
72          low, high = np.array((self.tree.getMetricMin(), self.tree.getMetricMax()))
73          assert ((low <= self.start_range) & (self.start_range <= high)).all()
74          assert ((low <= self.goal_range) & (self.goal_range <= high)).all()
75          self.distance_threshold = distance_threshold
76          assert reward_type in ('sparse', 'dense')
77          self.reward_type = reward_type
78          assert observation_type in ('discrete', 'continuous')
79          self.observation_type = observation_type
80
81          # Set up spaces and reward range
82          self.seed()
83          obs = self.reset()
84
85          # We'll scale goals by max_step_size
86          goal_space = np.array(self.tree.getMetricSize()) / self.max_step_size
87          self.observation_space = gym.spaces.Dict(dict(
88              desired_goal=gym.spaces.Box(low=-goal_space, high=+goal_space,
89                                          dtype=np.float32),
90              achieved_goal=gym.spaces.Box(low=-1, high=+1,
91                                           shape=goal_space.shape,
92                                           dtype=np.float32),
93              observation=gym.spaces.Box(low=0, high=1, # Booleans
94                                         shape=obs['observation'].shape,
95                                         dtype=obs['observation'].dtype)
96          ))
97          assert self.observation_space.contains(obs)
98
99          # The action space will be a continus, three-dimensional space
100         # of relative positions, but scaled so position + action *
101         # step_size gives a new position. We restrict the size of the
102         # action space so the agent can't move further than it can
103         # see.
104         self.action_space = gym.spaces.Box(low=-np.ones(3), high=np.ones(3),
105                                            dtype=np.float32)
106
107         self.reward_range = ((-1.0 if self.reward_type == 'sparse' else -np.inf), 0.0)
108
109
110     def seed(self, seed=None):
111         self.random, seed = gym.utils.seeding.np_random(seed)
112         return [seed]
113
114     def reset(self):
115         self.position = self.random.uniform(*self.start_range).astype(np.float32)
116         self.goal = self.random.uniform(*self.goal_range).astype(np.float32)
117         # TODO: Assert that goal and positions are empty
118         self.collisions = np.uint32(0) # Control the size to prevent overflows
```

```
119         self.last_achieved_goal = np.zeros(3, dtype=np.float32)
120         return self._get_obs()
121
122     def _is_occupied(self, position):
123         inside_bounds, key = self.tree.coordToKeyChecked(position.astype(float))
124         if not inside_bounds:
125             return True # Nodes outside bounds are considered occupied
126         node = self.tree.search(key)
127         try:
128             return self.tree.isNodeOccupied(node)
129         except octomap.NullPointerException:
130             # Sorry about this, but the octomap wrapper library has a strange API
131             return False # Missing key
132
133     def _get_obs(self):
134         """Return an observation for the current position"""
135
136         # Directions: (0, 0, -1), (0, 1, -1), etc.
137         directions = np.asarray(list(product(*repeat((0, -1, 1), 3)))[1:],
138                                 dtype=float)
139         observation = np.empty(len(directions) * self.visual_range, dtype=np.float32)
140         for i in range(self.visual_range):
141             resolution = self.resolution * (2 ** i)
142             depth = self.depth - i
143             magnitude = 1 if i == 0 else (resolution + 0.5)
144
145             for j, direction in enumerate(directions):
146                 position = self.position + direction * magnitude
147                 # Find the node, or a leaf node higher up in the tree
148                 # covering the same area.
149                 node = self.tree.search(position, depth)
150                 try:
151                     if self.observation_type == 'discrete':
152                         occupancy = self.tree.isNodeOccupied(node)
153                     else:
154                         assert self.observation_type == 'continuous'
155                         occupancy = node.getOccupancy()
156                 except octomap.NullPointerException:
157                     # Node does not exist, so the area is unmapped, i.e. unknown
158                     occupancy = 1 # Assume unmapped areas are occupied
159                 observation[i * len(directions) + j] = occupancy
160
161         # We use Hindsight Experience Replay (HER), so we need to
162         # output both the desired goal and the achieved goal
163         return {
164             'observation': observation,
165             'achieved_goal': self.last_achieved_goal, # Already scaled
166             'desired_goal': (self.goal - self.position) / self.max_step_size,
167         }
168
169     def _would_collide(self, offset):
170         # Now we need to check for collisions. Due to limitations in
171         # the octomap API, and to guard against implementation errors,
172         # this will be done in seven(!) steps:
173
174         target = self.position + offset
175
176         # 1. Is the current position within bounds? (it should be)
177         valid, key = self.tree.coordToKeyChecked(self.position.astype(float))
178         assert valid
179
180         # 2. Is the current position free? (it should be)
181         node = self.tree.search(key)
182         try:
183             occupied = self.tree.isNodeOccupied(node)
184         except octomap.NullPointerException: # Weird API, I know.
185             assert False # The node is unknown. This should never happen
186         else:
```

```
187             assert not occupied # The agent is standing inside a wall. Oh no!
188
189         # 3. Is the target position within bounds?
190         valid, new_key = self.tree.coordToKeyChecked(target.astype(float))
191         if not valid: # new position is out of bounds
192             return True # Would collide
193
194         # 4. Did the agent move at all?
195         if new_key == key:
196             return False # No collision because the agent is within the same voxel
197
198         # 5. Is the target position not free?
199         node = self.tree.search(new_key)
200         try:
201             occupied = self.tree.isNodeOccupied(node)
202         except octomap.NullPointerException: # Weird API, I know.
203             occupied = True # Treat unknown nodes as occupied
204         if occupied:
205             return True # Would collide
206
207         # 6. Is there occupied space between the two positions?
208         hit_position = np.full(3, np.inf, dtype=float)
209         hit = self.tree.castRay(origin=self.position.astype(float),
210                                 direction=target.astype(float),
211                                 end=hit_position, # Written to by castRay
212                                 maxRange=np.linalg.norm(offset))
213         # Description of castRay from octomap documentation: A ray is
214         # cast from 'origin' with a given direction, the first
215         # non-free cell is returned in 'end' (as center
216         # coordinate). This could also be the origin node if it is
217         # occupied or unknown. castRay() returns true if an occupied
218         # node was hit by the raycast. If the raycast returns false
219         # you can search() the node at 'end' and see whether it's
220         # unknown space.
221         if hit:
222             return True # Would collide
223
224         # 7. Is there unknown space between the two positions?
225         if not np.isinf(hit_position).all(): # Was hit_position written to?
226             node = self.tree.search(hit_position)
227             try:
228                 occupied = self.tree.isNodeOccupied(node)
229             except octomap.NullPointerException: # Weird API, I know.
230                 return True # Would collide with unknown node
231             else:
232                 assert False # This should never happen
233
234         assert np.isinf(hit_position).all()
235
236         # Some final checks, just to be sure we never ever step outside the octree
237         low, high = np.array((self.tree.getMetricMin(), self.tree.getMetricMax()))
238         assert all(low <= target)
239         assert all(target <= high)
240
241         return False # No collision
242
243     def step(self, action):
244         assert self.action_space.contains(action)
245         scale = self.max_step_size
246         assert scale > 0
247
248         # Determine if the agent would collide
249         offset = action * scale
250         collision = self._would_collide(offset)
251
252         # Remember the previously desired goal, to make sure we
253         # calculate the correct reward later
254         desired_goal = (self.goal - self.position) / scale
```

```
255             achieved_goal = (action.copy() if not collision else np.zeros_like(action))
256
257             # Modify the state of the environment
258             if not collision:
259                 self.position = self.position + offset
260             elif self.collisions < np.iinfo(type(self.collisions)).max:
261                 # Protected against overflowing
262                 self.collisions = self.collisions + 1
263             self.last_achieved_goal = achieved_goal
264
265             # Done?
266             distance = euclidian_distance(self.position, self.goal)
267             is_success = (distance <= self.distance_threshold)
268             done = is_success or (collision and self.end_on_collision)
269
270             # Reward
271             info = dict(is_success=is_success, collision=collision, scale=scale)
272             reward = self.compute_reward(achieved_goal, desired_goal, info)
273             assert self.reward_range[0] <= reward <= self.reward_range[1]
274
275             # Get new observation from octree
276             observation = self._get_obs()
277             assert self.observation_space.contains(observation)
278
279             return observation, reward, done, info
280
281         def compute_reward(self, achieved_goal, desired_goal, info):
282             # Compute the reward. In our case, the achieved goal is chosen
283             # in step() to be the action the agent actually took or zero
284             # if it would have collided. This function must work on both
285             # vectorized batches (when called from HER) and single samples
286             # (when called from step()). Note that desired goal might be a
287             # substituted goal if generating experiences in hindsight, in
288             # which case the achieved and desired goals will be the same.
289
290             # Both achieved and desired goals are in goal space, which has
291             # the same scale as the action space but can contain larger
292             # values. We need to scale the goals before comparing them
293             # with the distance threshold.
294             scale = self.max_step_size
295             assert scale > 0
296             distance = euclidian_distance(achieved_goal, desired_goal) * scale
297
298             if self.reward_type == 'sparse':
299                 is_success = ((distance <= self.distance_threshold)
300                               & (achieved_goal != 0).any(axis=-1))
301                 # -1 if no goal was achieved (by collision or zero step size)
302                 # 0 if near the goal (by goal_tolerance)
303                 # -1 otherwise
304                 return np.float32(is_success) - 1
305             else: # Shaped rewards
306                 assert self.reward_type == 'dense'
307                 return -distance
308
309 # Copied verbatim from https://stackoverflow.com/a/11146645
310 def cartesian_product(*arrays):
311     la = len(arrays)
312     dtype = np.result_type(*arrays)
313     arr = np.empty([len(a) for a in arrays] + [la], dtype=dtype)
314     for i, a in enumerate(np.ix_(*arrays)):
315         arr[..., i] = a
316     return arr.reshape(-1, la)
317
318 def create_empty_octree(*, resolution, size, origin, lazy_eval=False):
319     assert isinstance(resolution, float)
320     assert np.isfinite(resolution)
321     assert resolution > 0
322
```

```
323         assert size.dtype == float
324         assert size.shape == (3, )
325         assert all(np.isfinite(size))
326         assert all(size > 0)
327 
328         assert origin.dtype == float
329         assert origin.shape == (3, )
330         assert all(np.isfinite(origin))
331 
332         # Create an empty room by inserting a point cloud where all
333         # rays start at the origin and end in ends in each voxel of
334         # the wall. This gives a good enough approximation with
335         # reasonable time and memory complexity.
336         octree = octomap.OcTree(resolution)
337         r = resolution # Shorthand for the line below
338         x, y, z = [np.mgrid[-s // 2 - r:np.ceil(s / 2) + r * 2:r] for s in size]
339         bounds = np.concatenate(list(starmap(cartesian_product,
340                                              (((x.min(), x.max()), y, z),
341                                               (x, (y.min(), y.max()), z),
342                                               (x, y, (z.min(), z.max()))))))
343 
344         octree.insertPointCloud(origin + bounds, origin, lazy_eval=lazy_eval)
345         #for i in range(10):
346         #    noise = np.random.uniform(-1, 1, 3) * (size // 2 - r)
347         #    octree.insertPointCloud(origin + bounds, origin + noise, lazy_eval=lazy_eval)
348 
349         #threshold = octree.getOccupancyThresLog()
350         #pmin, pmax = origin + (-size / 2, +size / 2)
351         #for node in octree.begin_leafs():
352         #    p = node.getCoordinate()
353         #    occupied = (node.getValue() >= threshold)
354         #    inside = ((pmin <= p) & (p <= pmax)).all()
355         #    if inside == occupied: # xand
356         #        # Change value, but traverse node first
357         #        octree.search(node.getKey()).setValue(float(not inside))
358         return octree
359 
360 class EmptyEnvironment(OctreeEnvironment):
361     def __init__(self, *, resolution, size, origin, **kwargs):
362         resolution = float(resolution)
363         size = np.asarray(size, dtype=float)
364         origin = np.asarray(origin, dtype=float)
365         octree = partial(create_empty_octree, resolution=resolution,
366                          size=size, origin=origin)
367         # Goals and start positions will be uniformly sampled from
368         # anywhere within the empty room, but not too close to any
369         # walls.
370         inside_bounds = origin[:, np.newaxis] \
371                         + (size - resolution * 2)[:, np.newaxis] / (-2, 2)
372         kwargs.setdefault('goal_range', inside_bounds)
373         kwargs.setdefault('start_range', inside_bounds)
374 
375         super(EmptyEnvironment, self).__init__(octree=octree, **kwargs)
376 
377 class WallEnvironment(OctreeEnvironment):
378     @classmethod
379     def _create_octree(cls, resolution, size, origin, wall_range):
380         tree = create_empty_octree(resolution=resolution, size=size, origin=origin,
381                                    lazy_eval=True)
382 
383         ranges = starmap(partial(np.arange, step=resolution), wall_range.T)
384         for point in cartesian_product(*ranges):
385             # Update the node
386             occupied = True
387             node = tree.updateNode(point, occupied, lazy_eval=True)
388             node.getValue() # Will raise octomap.NullPointerException on error
389 
390         # Update occupancy of inner nodes to reflect their children's occupancy
```

```
391             tree.updateInnerOccupancy() # Because lazy_eval was True
392             return tree
393
394         def __init__(self, *, resolution, size, origin, wall_range, **kwargs):
395             resolution = float(resolution)
396             size = np.asarray(size, dtype=float)
397             origin = np.asarray(origin, dtype=float)
398             wall_range = np.asarray(wall_range, dtype=float)
399             assert wall_range.shape == (2, 3)
400             assert all(wall_range[1] - wall_range[0] >= resolution), \
401                 "wall_range must be at least as large as resolution in all dims"
402
403             # Prepare the octree factory function
404             octree = partial(self._create_octree, resolution=resolution,
405                              size=size, origin=origin, wall_range=wall_range)
406             super(WallEnvironment, self).__init__(octree=octree, **kwargs)
407
408
409     for reward_type, observation_type in product(('sparse', 'dense'),
410                                                  ('discrete', 'continuous')):
411         registry.register(
412             id=f'RandomDirection{reward_type.title()}{observation_type.title()}-v0',
413             entry_point=f'{__name__}:EmptyEnvironment',
414             max_episode_steps=2,
415             reward_threshold=0.0,
416             trials=10,
417             kwargs=dict(
418                 resolution=1,
419                 max_step_size=1,
420                 size=(5, 5, 5),
421                 origin=(0, 0, 0),
422                 start_range=(0, 0, 0),
423                 goal_range=((-2, -2, -2), (2, 2, 2)),
424                 distance_threshold=0.5,
425                 visual_range=1,
426                 reward_type=reward_type,
427                 observation_type=observation_type,
428             )
429         )
430
431
432     common_params = dict(
433         max_episode_steps=200,
434         reward_threshold=0.0,
435         trials=10,
436     )
437
438     common_kwargs = dict(
439         resolution=1,
440         size=(20, 10, 10), # room is empty in range ±(10, 5, 5)
441         origin=(0, 0, 0),
442         distance_threshold=0.5,
443         reward_type='sparse',
444         observation_type='continuous',
445     )
446
447     start_range = ((-8, -4, -4), (-7, 4, 4))
448     goal_range = ((7, -4, -4), (8, 4, 4))
449
450     for label in 'deterministic', 'randomized':
451         registry.register(
452             id=f'Empty{label.title()}-v0',
453             entry_point=f'{__name__}:EmptyEnvironment',
454             kwargs=dict(common_kwargs,
455                         visual_range=1,
456                         start_range=(np.mean(start_range, axis=0)
457                                      if label == 'deterministic'
458                                      else start_range),
```

```
                    goal_range=(np.mean(goal_range, axis=0)
                                if label == 'deterministic'
                                else goal_range)),
        **common_params
    )

for visual_range, name in ((0, 'Blind'), (1, 'Near'), (3, 'Far')):
    registry.register(
        id=f'Wall{name}-v0',
        entry_point=f'{__name__}:WallEnvironment',
        kwargs=dict(
            common_kwargs,
            # The wall will be in the center of the room along x-axis, but
            # will have openings on all four sides. It's a floating wall,
            # in other words.
            wall_range=((-1, -4, -4), (1, 4, 4)),
            visual_range=visual_range,
            start_range=np.mean(start_range, axis=0),
            goal_range=np.mean(goal_range, axis=0),
        ),
        **common_params
    )

registry.register(
    id=f'Office-v0',
    entry_point=f'{__name__}:OctreeEnvironment',
    max_episode_steps=1000,
    reward_threshold=0.0,
    kwargs=dict(
        octree = data_path('freiburg1_360.bt'),
        start_range = ( 0.22633858,  0.90999599,  2.50860099),
        goal_range =  (-1.45583376, -3.41167727,  2.29596730),
        distance_threshold=0.5,
        visual_range=5,
        depth = 9, # resulting in a resolution of 0.02 * (2 ** (16 - 9)) = 0.64
        reward_type='sparse',
        observation_type='continuous',
    )
)
```

B | Hyperparameters

Hyperparameter	Value
Actor learning rate	1e-3
Critic learning rate	1e-3
Experience replay buffer size	1e6
Polyak averaging coefficient	0.95
l2 weight penalty	1.0
observation clipping	200
relative goals	False
n cycles	50
rollout batch size	2
n batches	40
batch size	256
n test rollouts	10
test with Polyak	False
random eps	0.3
noise eps	0.2
replay strategy	future
replay k	4
norm eps	0.01
norm clip	5