Master Thesis

A cognitive agent to control physical maze via hierarchical learning

Submitted in partial fulfillment of
the requirements for the award of the degree of

Master of Science
in
Intelligent Systems

Submitted by

Gaurav Kumar

Under the guidance of

Prof. Dr. Helge Ritter
Dr. Andrew Melnik

Universität Bielefeld

Bielefeld, Germany

Abstract

Dynamic navigation in a variety of environments is essential for many goal-oriented tasks. Learning to control an object in a physically-plausible environment with quick adaptations to changing parameters of the object and the environment is a challenging problem for vanilla deep reinforcement learning techniques. It becomes more difficult when the reward is temporally distant. To address this problem a two-level architecture for planning and adaptive control is proposed. The higher-level-planning module builds the model of the world as a graph, and the agent can plan a path on this graph. This work proposes an algorithm for how an AI agent can learn a representation of an environment, by making multiple human-like saccadic observations from an overhead view. Each observation provides information about a limited convex area of space and must be integrated with other observations in a graph. The convexity of sub-areas substantially simplifies learning and control tasks for the lower-level adaptive controller. The lower-level controller is trained with deep reinforcement learning techniques to navigate to a specified subgoal in a convex area. A naive low-level controller is also presented. This work highlights an effective decomposition of a task into a two-level problem motivated by classic cognitive science ideas. Waypoints connect adjacent convex areas only, facilitating learning of control at this level by setting targets in easily reachable limits. The designed two-level architecture is capable of fast learning of sensorimotor control in convex areas and building a structured representation of an environment for effective planning.

Acknowledgments

I would like to express my appreciation to my supervisors Prof. Dr. Helge Ritter and Dr. Andrew Melnik for the useful comments, remarks, and engagement through the learning process of this master thesis. They consistently steered me in the right direction whenever they thought I needed it. I would also like to thank Dr. Sascha Fleer and Dr. Guillaume Walck for their guidance and support. I am very grateful to all of those with whom I have had the pleasure to work during this project.

Finally, I thank my family and friends for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them.

Thank you.

Contents

List of Figures

Chapter 1

Introduction

Human cognition emerges not from a single, global optimization principle within a uniform neural network. Rather, the human brain is modular, with distinct but interacting subsystems working in parallel and contributing to an overall emergent behavior [1, 2, 3]. Whenever a complex behavior is seen to be controlled by one section of the brain, they are hysterical in nature. For instance, the action of consuming food starting from holding it in one's hand until putting it in one's mouth is represented by one region in the brain of primates [4]. The reason seems to be the frequency of the action causing the emergence of such behavior as a fundamental movement. In such cases, an obstacle is not avoided because there is no involvement of differential processing.

The emergence of deep reinforcement learning (DRL) came with solving a wide range of games with one general algorithm [5]. It was enough to promise the wide application of the technique to learn any BlackBox behavior via bootstrapping given a large dataset and processing power. Although frequent evaluation of available options in an environment is a basis for DRL like other learnings. But it did not guarantee the learning of goal achievement over a very far horizon [6, 7].

Long-term planning requires objectives for intrinsic motivation and global exploration directed towards the goal. It helps to quickly solve down-stream tasks that need reward maximization over a short period. Hence, appropriate learning levels are essential for learning within a human amount of experience and generalization to new environments. Using this stratified approach individual levels can be learned with different motivations. This also corresponds to the fact that processes in the human brain are also stratified into different levels. Also, the purpose of adaptation to changes is defeated by the design of a single network. Being modular in nature a hierarchical architecture also caters to the need for adaptation at one level without interfering with the

learning of other levels.

Applying this thought of stratification, we found that at least two separate processes are employed by humans for the solution of a maze (figure 1.1). One of them is sensorimotor contingency [8] that depends upon the implicit understanding of the physics of the environment. It allows easy calibration to reactive control and shows quick improvements in a small number of trials. However, perfect control shows a slow learning curve at the later stages. The other one is of planning which primarily searches for a connected path and intermediate waypoints [9]. The intrinsic motivation is to find waypoints which bring the agent closer to the goal. It can also be said that while theoretically the task can be learned with DRL, it has two problems. It would need a well-crafted and shaped reward system [10] or demonstrations [11, 12] and a change in the material or structure of the game might need to relearn the entire model. In this study, the focus is on an architecture capable of learning within limits of human experience as well as easy transfer to new maze instances and physical properties. The motivation behind the proposed architecture comes from the field of cognitive science. Having identified two sets of skills for the task in hand, the work would focus on appropriate representations of the skills as well as their interplay to bring the desired result.

1.1 Hierarchical Modelling

The planning is done at the beginning of the task and an eventual replanning based on certain triggers should be enough. Physical control needs to be done in every single time step without failure. Hence, they do not qualify for a cooperation based regime rather a master-slave architecture would be more efficient where a task is planned eventually and the low-level controller does a rollout to achieve the task. This can be appropriately called a Hierarchical architecture in this context.

1.1.1 High-Level Controller

The high-level controller is responsible for planning a trajectory from any start to the goal position. Considerations at this level can be the size of passage, obstacles, stable points, etc. with the main objective to make progress towards the goal. The main challenge for the planner is to derive a meaningful representation of the connectivity of the environment. One can think of this connectivity from different perspectives. It can be seen from the perspective of computational optimality, perfection or cognitive principles.

Figure 1.1: The game of hand-held labyrinth maze is about controlling the tilt of a board against two horizontal axes to induce a velocity in the ball inside and use this velocity to reach a goal position from the start position. Since the ball is made up of metal, it has a difficult physics and one needs a high degree of reactiveness to deal with it.

The cognitive principles depend upon the commonality of a set of similar tasks but also take care of the task-specific requirements. The work in the past is mainly done by random processes[13, 14] or processes principled in numerical structures [15].

Some cognitive principled architectures driven by eye movements will be evaluated for the planner. A search of convex spaces of variable shapes and sizes and decision points instead of tunnel spaces, to sample waypoints will be the focus of this work. The generated representation in this phase will be evaluated on the ground of quality of connectivity and location of waypoints. Also, the low-level planner coupled with these representations can shed more light on the usability of such graphs against the computationally optimal graphs.

1.1.2 Lower-Level Controller

Once we have a graphical representation of the environment we can have a planned trajectory for execution in terms of a sequence of intermediate goals. The low-level controller represents a class of methods that are reactive and need to take into account an immediate goal rather than a long term achievement. Specific to this task they are responsible to bring the ball to a local goal in the form of an intermediate waypoint given by the planner while avoiding any obstacle preferably including dynamic obstacles.

The main consideration at low-level controller is the environment dynamics. A naive controller can work with finding a direction vector from the current position to local goal and translating it to action with extremely low magnitude. The small magnitude is a direct consequence of the fact that it does not take care of the real limits of the movement, with due respect to the momentum and inertia. For example, a metal ball will move with very high velocity as a result of a very low magnitude tilt. However, to move a balloon filled with air, one needs to provide a high level of tilt for a movement to be induced at all.

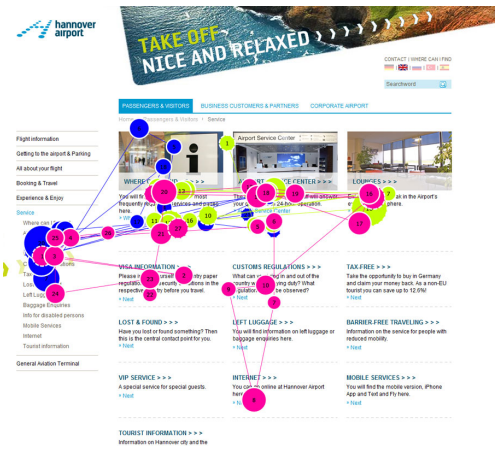
A hypothesis is that another naive approach can take care of the environment dynamics if we sample a few trajectories in the form of the previous velocity, applied tilt and resultant velocity. An interpolation of this data can be done by clustering it and using k-nearest neighbor to do a lookup. To impede the velocity of the ball while reaching the goal a control theory based PID controller can be used.

The above approaches will work as far as the waypoints or local goals are in Line-Of-Sight [14]. DRL can have an edge if the goal is around the corner and we need to learn movement behavior in a short-range. It would also cater to the need for collision avoidance and hence should be the desired method for a more capable controller. Also, the behavior such as slowing down while the local-goal is being changed can be learned with appropriately shaped rewards.

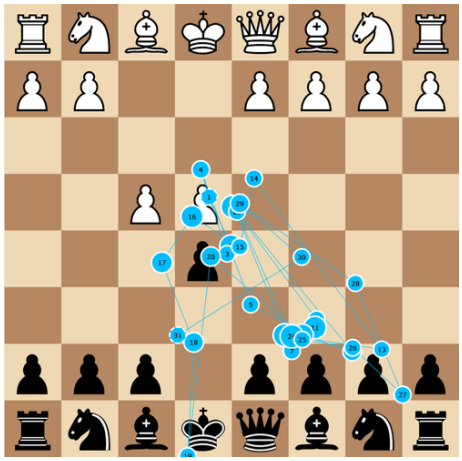
1.2 Sensory input for Planning

For humans, the input to solve this game is solely based on visual observation and to derive cognitive methods for building the graph one has to keep in mind the human visual system as well as the processing of visual input. The question is how do we generally inspect visually.

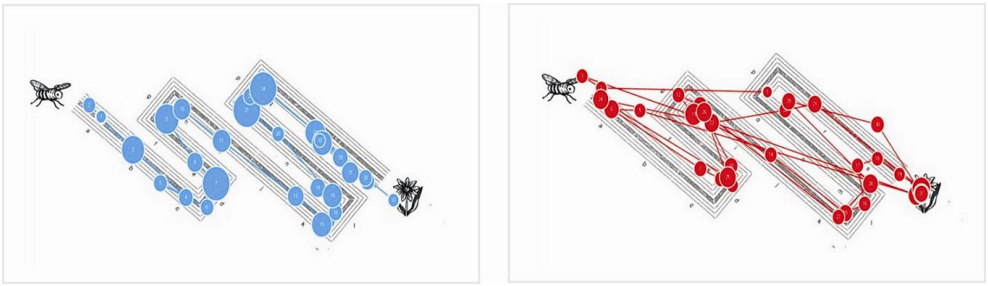
Eye-tracking studies suggest that humans have a particular saccade and fixation behavior when it comes to inspection [19, 20](figure 1.2 (a), (b)). The vision needs to find an approximate geometric center for symmetric spaces



(a) Fixations and saccades [16]



(b) Fixations and saccades [17]



(c) Comparing Smooth Pursuit(left) Vs. Fixation-saccade(Right) [18]

Figure 1.2: Eye-Movement

but it might drift to a special region based on some fine features. In our case, a region where there are no obstacles or options, do not attract the vision. Instead, a corner, an opening into a new region and an obstacle have interesting values. Important to mention that low-level control of the same game would rely heavily on smooth-pursuit instead of fixation and saccade cycles differentiated in figure 1.2 (c). For this work, visual inputs are not used as an input. The setup rather relies on external localization. But this work will try to account for the human eye movement in its approach to building the graph of the environment.

The following Chapter 2 "Theoretical Background", discusses the building blocks and concepts that are critical to our thought process in the design of experiments. Chapter 3 "Simulation Environment", explains the simulation environment built in Unity. Chapter 4 "Experiment", presents the high-level controller building and querying the graph as well as the low-level controller executing the generated trajectory. Finally Chapter 5 "Discussion

and Conclusion", discusses the results and concludes with possibilities of future work on the same line of thought.

Chapter 2

Theoritical Background

2.1 Neuron and Neural Network

An artificial neuron also called neuron is a fundamental unit of Artificial Neural Networks (ANN). It is is a mathematical function represneted by Figure 2.1(a) and equation (2.11), input to which is scaled by a weight and shifted by a bias. $\sum_i w_i \cdot x_i + b$ represents the accumulated activation which then passes through activation function $f()$. Here, x is a multi-dimentional input, w is the weight vector and b is the bias.

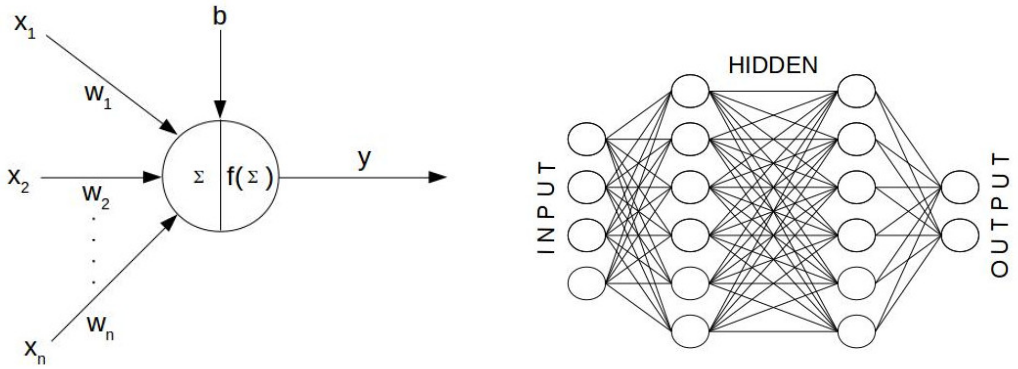
$$y = f(\sum_i w_i \cdot x_i + b) \tag{2.11}$$

An artificial neural network [21], as in Figure 2.1(b), is a complex arrangement of neurons. By convention, the leftmost layer is called an input layer from where the data is fed and rightmost is called the output layer from where the network gives a processed output. The intermediate layers are called hidden layers. The name derives its analogy from the biological neural structure in the brain. A network with more than one hidden layer is called a Deep Neural Network (DNN).

An arrangement of such neural units in a layer all of them getting some input was capable of solving any linearly solvable binary classification problem with guaranteed convergence. It does this by adjusting the weights in either direction, leading to lower error as below.

$$w_i(t+1) = w_i(t) + r \cdot (Target - f(\sum_i w_i \cdot x_i + b))x_{j,i}$$

The perceptron applies a threshold function as in equation (2.12).



(a) Artificial Neuron (b) Deep Neural Network

Figure 2.1: Neural Network

$$f(x) = \begin{cases} 1 & \sum_i w_i \cdot x_i + b \geq threshold \\ 0 & otherwise \end{cases} \tag{2.12}$$

Further research identified that a multiple-layer structural arrangement of perceptrons can solve more complex tasks and Multi-Layer Perceptron(MLP) [21] came into the picture. The use of more than one layer gives non-linearity essential to solve nonlinear problems. The task of weight adjustment was more complex in a multi-layer case. The method adopted is called back-propagation of error.

The back-propagation has to go through multiple layers. For this to work, the activation function was changed to sigmoid due to its close resemblance to thresholding. Such a shift gave a continuous function that was differentiable for error back-propagation via gradient descent optimization [21].

Gradient Descent

The below equation represents the way a gradient descent optimizer works. The loss to be minimized is represented by the objective function $J(\theta)$. The gradient descent takes advantage of two facts. First that we use only differentiable functions as an activation function for the neuron. Second, the chain rule can be used as the sequence from input to output preserves differentiability.

$$\theta' = \theta - \eta \cdot \nabla_\theta J(\theta)$$

Gradient descent came with its problems. Even though the differentiability works in principle via chain rule through the layers, the gradient becomes too small to be usable. This is called vanishing gradient, which restricts the depth of this layered arrangement. Solutions such as using different activation function e.g RELU, training in a hierarchical structure, or normalization at each layer are proposed to solve this problem.

Another is the complexity of updates when the learning data set is large. This vanilla version of gradient descent discussed so far is deterministic since we see the gradient given entire learning data. Such a method is not efficient for larger datasets since every parameter update step needs one full sweep of training data. Also, the minima found by this optimization depends upon the parameter initialization.

Stochastic Gradient Descent

The solution is proposed in the form of stochastic gradient descent (SGD) which is an iterative method for optimizing an objective function with suitable smoothness properties. It is called stochastic because the method uses one sample to evaluate the gradients, hence SGD can be regarded as a stochastic approximation of gradient descent optimization.

$$\theta' = \theta - \eta \cdot \nabla_\theta J(\theta; x^i; y^i)$$

Since, the parameter update takes one sample at a time the unstable updates make the parameter jump from one to another range and may reach a better or worse minimum.

Mini-Batch SGD

The potential instability of SGD can be countered by Mini-Batch gradient descent.

$$\theta' = \theta - \eta \cdot \nabla_\theta J(\theta; x^{i:i+n}; y^{i:i+n})$$

Here, instead of having a complete swap or else update on only one sample, a mini-batch of size n is used. This gives the best compromise between the two extremes. It counters the high instability while still multiple parameter updates are done dring one swap. Each swap is called an epoch in machine learning literature.

2.2 Reinforcement Learning

Supervised learning [21] is a class of machine learning algorithms where a true label y is known beforehand for every data point in the training set. The target is to learn a function of the form $y' = f(x)$ by minimization of loss $|y' - y|$ via adjustment of function parameters. Unsupervised learning [21] is the type of algorithm where there is no label and hence the underlying structure of the data has to be found directly based on the algorithm e.g. clustering or association. The expectation from both is to be able to generate labels for novel data points.

In Reinforcement Learning (RL) [22], the task is similar to supervised learning where we learn $y = f(x)$. The learned function represents an action distribution that is to be used to act over the environment. While this distribution is not directly evaluated against a known optimum action distribution, the resulting reward from the environment is used as an indicator of the quality of action. Being the only indicator for learning in the absence of any knowledge of the environment, this problem falls in the category of model-free learning [22]. The RL agent learns in an environment with the following salient features;

- Access to the current state representing the environment.
- Require decision of action in any state that changes the environment.
- The environment with fixed dynamics that remains unchanged.
- Reward as the only learning indicator representing the favourability of change.

With above in mind, let's see how it represents a Markov Decision Process (MDP) [22].

2.2.1 Markov Decision Process

The Markov property states that, although we could have been in a state only due to a long sequence of past actions, but having said that we are in this state, the knowledge of past is not required. Describing a system like this is called a Markov process. For example, to predict the future position of a projectile, the current state does not satisfy the Markov Property. The position together with velocity does satisfy the property and becomes a Markovian state which is sufficient knowledge indifferent to anything in the past. For the same example, the position and velocity can also be replaced

by the last two positions. Hence, the concatenation of the last two states in one state also forms a markovian state for this task.

A decision process where the input to the process is also called decision state follows the Markov property is called the Markov Decision Process (MDP) as in Figure 2.2. The decision is in the form of action from the set of possible actions and is sampled from an action distribution. We want an action distribution for sampling that maximizes an objective. Reward maximization is the objective of this Markov decision process. The notion of reward needs more discussion that follows in the next section.

Moreover, the process is stationary, which means that the dynamics of the environment is fixed and does not change over time. It is important to say that as the learning progresses, the dynamics of the environment should not change else the learning might never converge in such an environment. With these assumptions in place, a Markov decision process is a 5-tuple (S, A, P_a, R_a, γ) where:

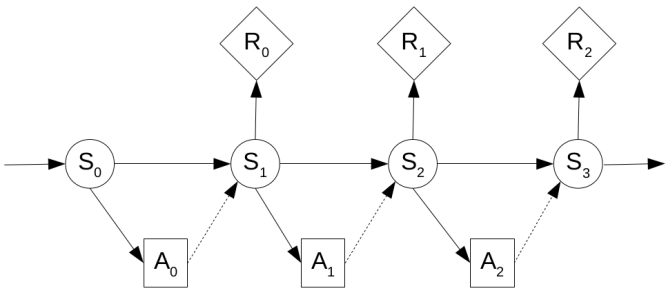


Figure 2.2: Markov Decision Process

- S is a finite set of states,
- A is a finite set of actions,
- $P_a(s, s') = P_r(s_{t+1} = s' | s_t = s, a_t = a)$ is the probability that action a in state s at time t will lead to state s' at time t+1,
- $R_a(s, s')$ is the reward received after transitioning from state s to state s', due to action a.
- γ is the discount factor for future rewards,

Above explains the problem structure. The task, however, is to find a policy $\pi(s) = a$ that emits an action to be taken in a given state. It can also be represented as a distribution $P(a|s)$.

2.2.2 Value function and optimal policy

The task in the environment might not be a one-step task. For example, a scenario of navigation to a goal position for a big reward. The victory or defeat can only be decided on the last step while all intermediate steps give a relatively low and uniform reward. The intermediate steps are critical to reaching the goal but do not have any importance in their own from the perspective of reward. We are talking about a sequence as in equation (2.1),

$$\{s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2 .....r_t, s_t, a_t, ..., r_T, s_T, a_T\} \tag{2.1}$$

where s_t represents the state of the environment, a_t the action taken by the agent, and r_t the reward received at a time step. A step reward can not be used to search for the sequence because this reward does not tell anything about the end reward. The problem is addressed by modifying the notion of reward into cumulative reward from the future. The task of RL is to maximize not the step reward but cumulative future reward termed as return R.

$$R_t = r_{t+1} + r_{t+2} + ... = \sum_{n=0}^{\infty} r_{t+1+n} \tag{2.2}$$

While many environments put a limit on the maximum number of time steps before the task is reset and are called episodic, there might be some tasks where the reset is not done by the virtue of environment. We cannot deal with an infinite sequence as above because the value can be maximized infinitely without reaching the goal. For example, if the step reward is 0.001 and the goal reward is 1, and we have an infinite sequence, the agent might learn to move to and fro between two intermediate states to accumulate much higher reward than to go to the goal.

We use a discount factor to make the return as cumulative discounted future rewards. We want the reward more distant in the future to be discounted more than less distant rewards. This should reduce the contribution of future rewards to an extent that after a horizon the reward should diminish completely. It prevents the return to go to infinity in the first place and it also prefers a shorter path to the goal than the longer even in episodic scenario. Hence the equation (2.2) becomes,

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{n=0}^{\infty} \gamma^n r_{t+1+n} \tag{2.3}$$

where $\gamma \in [0, 1]$ is the discount factor. This return R_t is the quantification of a value function which is a representation of how valuable it is to be in a state. A value function also called a state-value function [22] is written in the form of an expectation as the environment may be stochastic and the return might not be the same in several trails.

$$V^{\pi}(s) = \underset{\pi}{\mathbb{E}}[R_t|s_t = s] \tag{2.4}$$

It gives an estimate of the return from this state following the optimal policy. This includes the reward from the most rewarding action even in the current state. An action-value function [22]

$$Q^{\pi}(s, a) = \underset{\pi}{\mathbb{E}}[R_t|s_t = s, a_t = a] \tag{2.5}$$

however, estimates the value for all the possible actions in this state and following optimal policy then onwards. Hence, for an environment of 4 discrete actions, we will have one state-value but 4 action-values.

As iterated before, RL dealing with a Markov decision process takes an action in every state. For this, it takes the help of a distribution $p(a|s)$ over the set of action A, given a state $s \in S$, called a policy. Being a probability distribution $\underset{a \in A}{\Sigma} p(a|s) = 1$. It is also represented as $a = \pi(s)$. Selecting an action with maximum probability is the optimal action in that state. This greedy action selection process throughout the sequence defines the optimal policy $a^* = \pi^*(s) = \underset{a}{argmax}\ \pi(s)$. The distribution $p(a|s)$ is learned over the objective of reward maximization. This maximization is done by maximizing the state-value or action-value which are representatives of the accumulated environmental rewards.

A sequence-based problem would need an end-to-end search for the best sequence. such a search is costly both in terms of processing time and memory. This problem is addressed by looking at the problem from the perspective of MDP, Bellman optimality [22] and dynamic programming [22].

2.2.3 Bellman Equation and Dynamic Programming

The key idea of dynamic programming is to break the global sequence problem as in equation (2.1) into local subproblems and the emergence of the solution of these sub-problems into the solution of the global problem.

The structural unit of the above sequence is that being in any state $s_t \in S$ and taking an action $a \in A$ results in next state $s_{t+1} \in S$ and gives reward r. Independent of being an initial step, a final step or an intermediate step in a sequence or a good or bad choice of action, the transitional unit

remains the same. A Markov decision process deals with the representation of such systems and the above theory applies to all the problems that are Markovian in nature. The idea that was presented by Richard E. Bellman was to care about this transitional unit. For this, it is important that the sequence generated by the solution of the global problem and its accumulated reward should correspond to the sequence that is generated by solving the sub-problems. They should reach the same minima.

This recursive solution process is called dynamic programming and the Bellman Equation presents the recursive relation. It iteratively reaches the optimal solution that exists for the complete sequence by finding an optimal solution for every single transition.

We want to derive Bellman equation for Value Function. For convenience we wrote equation (2.4) for state s_t and used equation (2.3) in it to get,

$$V^{\pi}(s_t) = \underset{\pi}{\mathbb{E}}[R_t] = \underset{\pi}{\mathbb{E}}[\sum_{n=0}^{\infty} \gamma^n r_{t+1+n}]$$

We pull out the reward for the first step from the summation and leave the generalization for the next steps.

$$V^{\pi}(s_t) = \underset{\pi}{\mathbb{E}}[r_{t+1} + \gamma \sum_{n=0}^{\infty} \gamma^n r_{t+2+n}]$$

The leftover summation represents value function for the next states

$$V^{\pi}(s_t) = \underset{\pi}{\mathbb{E}}[r_{t+1} + \gamma V^{\pi}(s_{t+1})] \tag{2.6}$$

Now, we write the two terms of expectation in equation (2.6) separately. The expectation of reward r_{t+1} depends upon the probability of choosing an action from the action set in that state, the probability of getting a specific reward from possible rewards (as the environment is stochastic) and the value of reward itself.

$$\underset{\pi}{\mathbb{E}}[r_{t+1}] = \sum_{a} p(a_t|s_t) \sum_{r} p(r_{t+1}|s_t, a_t)\ r_{t+1}$$

Adding marginalization over the next state and rearranging terms.

$$\underset{\pi}{\mathbb{E}}[r_{t+1}] = \sum_{a} p(a_t|s_t) \sum_{r}\sum_{s} p(s_{t+1}, r_{t+1}|s_t, a_t)\ r_{t+1}$$

$$\underset{\pi}{\mathbb{E}}[r_{t+1}] = \sum_{a} p(a_t|s_t) \sum_{s}\sum_{r} p(s_{t+1}, r_{t+1}|s_t, a_t)\ r_{t+1} \tag{2.7}$$

Similarly, the second term from equation (2.6) becomes,

$$\underset{\pi}{\mathbb{E}}[\gamma V^{\pi}(s_{t+1})] = \gamma \sum_{a} p(a_t|s_t) \sum_{s} p(s_{t+1}|s_t, a_t)\ V^{\pi}(s_{t+1})$$

$$\underset{\pi}{\mathbb{E}}[\gamma V^{\pi}(s_{t+1})] = \gamma \sum_{a} p(a_t|s_t) \sum_{s}\sum_{r} p(s_{t+1}, r_{t+1}|s_t, a_t)\ V^{\pi}(s_{t+1}) \tag{2.8}$$

Using equation (2.7) and equation (2.8) in equation (2.6) and collecting terms, we get,

$$V^{\pi}(s_t) = \sum_{a} p(a_t|s_t) \sum_{s}\sum_{r} p(s_{t+1}, r_{t+1}|s_t, a_t)\ [r_{t+1} + \gamma\ V^{\pi}(s_{t+1})] \tag{2.9}$$

Similarly, for action-value function the above equation (2.9) becomes.

$$Q^{\pi}(s_t, a_t) = \sum_{s}\sum_{r} p(s_{t+1}, r_{t+1}|s_t, a_t)\ [r_{t+1} + \gamma\ Q^{\pi}(s_{t+1}, a_{t+1})] \tag{2.10}$$

$$\text{Since, } V^{\pi}(s_{t+1}) = \sum_{a} p(a_t|s_t)\ Q^{\pi}(s_t, a_t)$$

2.2.4 Partially Observable Markov Decision Process

A partially observable markov decision process (POMDP) [22] as in Figure 2.3, is a 7-tuple $(S, A, T, R, \gamma, \Omega, O)$ where

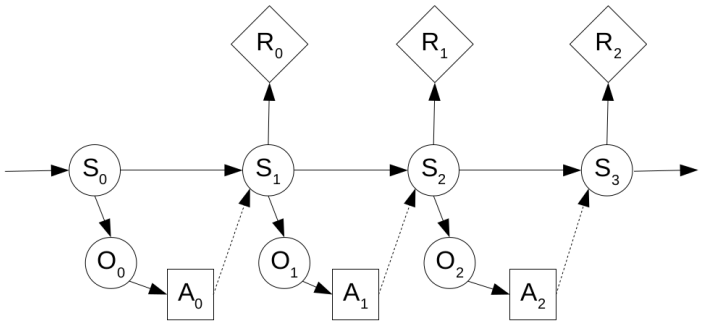


Figure 2.3: Partially Observable Markov Decision Process

- S is a finite set of states,

- A is a finite set of actions,
- $P_a(s, s') = P_r(s_{t+1} = s' | s_t = s, a_t = a)$ is the probability that action a in state s at time t will lead to state s' at time t+1,
- $R_a(s, s')$ is the immediate reward (or expected immediate reward) received after transitioning from state s to state s', due to action a.
- γ is the discount factor for future rewards,
- Ω is a set of observations, and
- O is a set of conditional observation probabilities

A POMDP is a more restrictive model where unlike MDP, the state can be partially observable and hence the complete knowledge of state is not available. To handle this a conditional observation distribution over S has to be maintained based on the observation and underlying MDP described by first four values in the tuple.

2.2.5 Q Learning

Q-Learning [23] is the algorithmic way to optimize action-value functions for each state as in equation (2.10). Q-Learning needs to maintain the value estimate of every state-action pair in a table called Q-table and does a lookup while selecting and optimizing an action. The optimization follows the reward maximizing action from the next state using the Q-table. The current action is chosen via the exploration-exploitation trade-off. An $\epsilon-$greed [22] policy is used where a random action is chosen with a probability ϵ and otherwise, a reward maximizing action is chosen in the current state as well. The new Q-Value is updated as follows, with $Y_t - Q(s_t, a_t)$ representing the loss.

$$Q'(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[Y_t - Q(s_t, a_t)]$$

$$\text{where, } Y_t = r_t + \gamma * \underset{a_{t+1}}{max} Q(s_{t+1}, a_{t+1})$$

Optimization process increases the Q-value of one action when an action in the future state gets a higher Q-value due to some new action taken in the following trajectory. This way a high rewarding future state increases the value in previous states iteratively.

The only challenge remaining is to explore the environment and try out every action sufficiently many times to reach the optimal action-value for

each state. A stopping criterion or indicator of convergence can be when the Q-Values or the actions that have the highest Q-Value do not change any further in a greedy action selection. A decay of ϵ can be seen as a heuristic to reach an optimal policy.

2.3 Deep Reinforcement Learning

2.3.1 DRL Algorithms

Deep Q-Network

With increasing dimentions of state and action, the Q-table becomes an infeasible solution. In such cases it is better to maintain a function that can estimate the Q value given the state input. Deep Q-Network [5] uses a neural network to estimate the action-value of every state. The Q Function is a neural network with parameters θ, which is updated using SGD as below,

$$Q(s_t, a_t; \theta) = r_t + \gamma * \underset{a_{t+1}}{max} Q(s_{t+1}, a_{t+1}; \theta)$$

$$\Delta_\theta = \alpha(Y_t - Q(s_t, a_t; \theta_t))\nabla_{\theta_t} Q(s_t, a_t; \theta_t)$$

$$\text{where, } Y_t = R_t + \gamma * \underset{a}{max} Q(s_{t+1}, a_{t+1}; \theta_{Target})$$

with samples which is a 5-tuple $(s_t, a_t, r_t, s_{t+1}, Terminal)$. There are challenges when training a DQN via bootstrapping. A neural network is a universal approximator and parameter update may affect all the approximations, unlike a tabular update where only one value is updated. Hence it can easily diverge. DQN uses a target network [5] which is a copy of the DQN but is fixed for a certain number of updates. This gives stability to the targets against which the main DQN is optimized. A simple copy of parameters, $Q_{Target} \leftarrow Q$, updates the target network every n steps.

Another problem is correlated updates due to the samples depending upon each other. The experience replay [5, 24, 25, 26] is about using a random sample of experiences saved in a replay buffer to update the network. This random sampling process de-correlates the update. Experience replay comes with some tunable parameters. A pre-fill of the buffer is done to avoid correlated updates from the beginning. The update of the network starts only after a certain amount of experiences are saved in the memory. The size of the buffer can also influence the results and hence can be tuned to the task.

Double DQN

DQN leads to overestimation of Q values because of the $\underset{a}{max}\ Q$, for both selection and evaluation of action. The idea behind Double Q-Learning [27] as the solution to this is problem is to decouple selection and evaluation.

$$Y_t = R_t + \gamma * Q(s_{t+1}, \underset{a_{t+1}}{argmax} Q(s_{t+1}, a_{t+1}; \theta); \theta_{Target})$$

To understand the difference, the main network can have an action $a1$ that has the highest Q-value in the next state. The target network due to being fixed for many updates may have another action $a2$ that gives the highest Q-value for the next state. As per DQN, the Q-Value from the target network for action $a2$ would be used to estimate the Q-value of action in the current state while we know that the main network would not take action $a2$ but $a1$ in the next state. We can still estimate the Q-value of action $a1$ by the target network. As per Double Q Network, it would still consider the action $a1$ for target estimation if the main network says that action $a1$ is the reward maximizing action in the next state.

Prioritised Experience Replay

There are many modifications possible to the experience replay itself. Prioritized Experience Replay [28] is the technique of prioritizing sampling based on some criteria. The TD-error that is already calculated above can be used as one of many options. The more the error, the higher the probability of being sampled. This is the way of giving more emphasis to the experiences that have more to teach to the network. The sample becomes a 6-tuple $(s_t, a_t, r_t, s_{t+1}, Terminal, \delta)$ where δ is the TD-error.

Hindsight Experience Replay

Another problem related to sample inefficiency of DQN is because a maximization of Q-value cares only about high rewarding samples. However, even if the negative samples do not tell anything about the reward maximization, they possess the knowledge about the environment dynamics. Hindsight Experience Replay [29] is an improvement that helps in learning from negative experiences as well. The negative experiences are replayed in a way that the goal is substituted with the one that is achieved in the episode and the reward is modified accordingly.

2.4 Continuous Action Space and DRL

2.4.1 Actor Critic

Due to a limited number of actions in discrete action space, policy evaluation and improvement is done hand-in-hand. This is a deterministic problem and the improvement of the policy is done in a recursive way of dynamic programming. Therefore, we have only one network that estimates the action-value and we take the action that maximizes the value as per our objective. The notion of action is implicit as a value related to the state-action pair is estimated.

Actor and Stochastic Policy Gradient

However, to find the maximizing action in continuous space one can go for action discretization as a solution, but only if the environment is easy enough. Otherwise, the requirement is to estimate the distribution of real-valued action that is a parameterized value, e.g. the degree of freedom of a robot. The action parameters are moved in a direction to maximize the objective of a higher return. This is the gradient of action parameters, hence, policy gradient. The return is a result of an action and hence the policy needs to integrate over the set of states and actions making it a stochastic policy gradient. The simplest form of this is the Reinforce Algorithm with the below rule.

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi\theta}[\nabla_\theta log \pi_\theta(s,a) R_t]$$

The problem is a heavy sample requirement to train a neural network for this and due to high variance, the learning might be unstable.

Critic as an estimator of return

The return in the above case has to be available and in the usual case, we need an estimation of the return as we do not have it deterministically. We reside at using the Q-Value of the state-action pair as the notion of return. Hence, there are two components in this architecture, an actor that takes the state and action as input and generates policy and a critic that takes the same input and generates a Q-Value.

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi\theta}[\nabla_\theta log \pi_\theta(s,a) Q^w(s,a)]$$

An advantage actor critic (A2C) with $A^w(s,a)$ as the advantage function.

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi\theta}[\nabla_\theta log\pi_\theta(s,a)A^w(s,a)]$$

The TD error δ of the critic can be used directly to optimize the policy.

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi\theta}[\nabla_\theta log\pi_\theta(s,a)\delta]$$

All of the above may give different results but are an indicator of the quality of action.

2.4.2 Deep Deterministic Policy Gradient

Deterministic Policy Gradient

In above the Q-value of state-action pair is used as an indicator of return for a state-action pair and the actor must adjust itself to predict the action for a higher return. Hence, there is a need of integrating over the entire state and action space thereby stochastic. A deterministic policy gradient [30] leverages the fact that instead of adjusting the policy to maximize return for state-action pairs, it can adjust the policy to maximize return in a state based on the gradient of state-action value. This way the policy does not need to integrate over the action space but state space. While the critic looks into what action gives higher state-action value, the actor looks into state and tries to learn action as a black-box in the direction of the gradient of state-action value.

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi\theta}[\nabla_\theta log\pi_\theta(s)\nabla_a Q^w(s,a)]$$

Deep Learning of Deterministic Policy Gradient

Deep Deterministic Policy Gradient (DDPG) [31] is an actor-critic algorithm that uses deep function approximators. It merges the idea of DPG with DRL. It uses target networks for both actor and critic but unlike DQN which copies the entire weights after some training steps, the target tracks the main network with a rate of τ. Formally the actor is $\mu : S \rightarrow A$ and the critic is $Q : SXA \rightarrow Q$.

The critic is trained against the target action-value corresponding to action generated by the target actor.

$$L = (Q(s_t, \mu(a_t)) - Q_{Target}(s_t, \mu(a_t)))^2$$

The actor is trained using a sampled policy gradient together with the gradient of action-value.

$$\bigtriangledown_{\theta^\mu} = \bigtriangledown_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} * \bigtriangledown_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

Algorithm : DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M do
    Initialize a random process $\mathcal{N}$ for action exploration
    Receive initial observation state s_1
    for t = 1,T do
        Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
        Execute action a_t and observe reward r_t and observe new state s_{t+1}
        Store transition (s_t, a_t, r_t, s_{t+1}) in R
        Sample a random mini-batch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
        Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
        Update critic by minimizing the loss: $L = \frac{1}{N}\Sigma_i (y_i - Q(s_i, a_i|\theta^Q))^2$
        Update the actor policy using the sampled policy gradient:
            $\nabla_{\theta^\mu} J = \frac{1}{N}\Sigma_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$
        Update the target networks:
            $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
            $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
    end for
end for

2.5 Multi-Goal Tasks

2.5.1 Universal Value Function Approximators

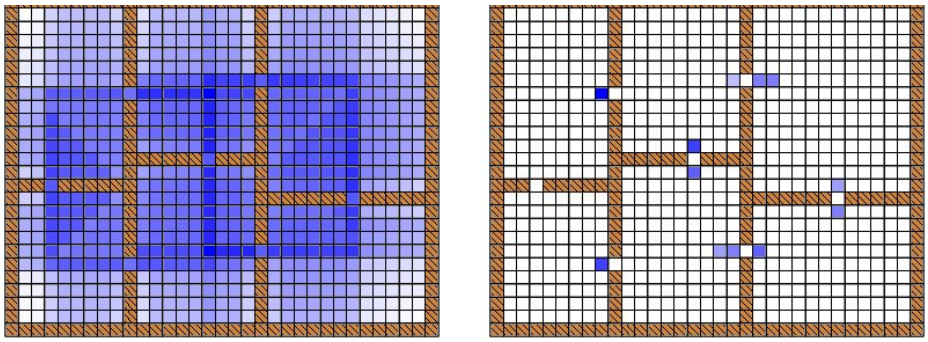
Universal Value Function Approximators [32] extends the value function estimation to a multi-goal scenario where we have more than one goal to achieve. It can be seen as extending $V(s, \theta)$ to $V(s, g, \theta)$ where $g \subset G$, is the set of goals.

Using this approach needs an episodic scenario. Every episode has a fixed goal which is also supplied as the state input to the approximator. The easiest mode is to concatenate the goal with the state input. This also opens the way to algorithmic improvements orthogonal to many DRL algorithms like

hindsight experience replay (HER) which has shown that even single goal tasks can be modified to take advantage of the multi-goal scenario. It also promises better results than training on one goal. This approach can be easily extended to DQN and DDPG.

2.5.2 Bottleneck Approach

The bottleneck approach [33] gives the advantage of working with a low dimensional representation of high dimensional data. Many algorithms in RL used autoencoders [34, 35] to control the information in a model. The trick is to see if the change of information from the state drives a change in the optimal policy. Such control of information is seen as an information regularization via variational autoencoders [36, 37].



(a) Relevant goal information for each state in the 6-room grid worldexample

(b) Subgoal discovery results in the 6-room gridworld example

Figure 2.4: Informational Treatment of MDP's [38]

Information Bottleneck for hierarchical Learning

This approach is about the treatment of different states of an MDP based on information. An information bottleneck can be used in the form of maximum likelihood of action conditioned over the state and goal [38] to see if there is information gain or change in entropy of information in that state. Using states where the entropy changes to a considerable threshold as terminal states in the options framework for a low-level policy and identification of such states as high-level policy helps to generate substructures in a task. Results shown in figure 2.4.

Information Regularizer and contention/cooperation

An autoencoder enables to learn a low dimensional representation of data in an unsupervised manner. A variational autoencoder scales the distance of the supplied data distribution from a uniform distribution. This property is useful as the encoding of data happens with reference to the same baseline uniform distribution. A regularizer based on a variational autoencoder can be used to train two agents over the same task [37] without access to either each other's model or interaction with the contending or cooperating agent. Since the divergence represents the amount of goal information used to take decision of action, training based on discounting or rewarding based on each other's regularization can be seen as information-theoretic regularizer. It brings contention or cooperation among the agents by revealing or hiding their intentions in terms of KL-divergence.

Information Regularizer and subgoal detection

A maximum-likelihood based entropy measurement or a variational autoencoder quantifies the same class of information. A subgoal in the environment is the state that gives relatively higher rewards than a normal transition. Instead of options framework, automatic detection of decision states [39] was done by adding the divergence to the environmental reward which makes some states more rewarding than the others if the divergence or the entropy of information is higher. It is also used as a regularizer in the update of the actor. Figure 2.5 shows variational autoencoder taking state and goal as input and encoded output used in Actor along with observation while Critic remains the same. Results are shown in figure 2.6 where one can see the junctions are identified as high rewarding stated as they have higher value of KL-Divergence which is added to the reward. Also, the gradient of the actor policy is adjusted so that the goal information is prominent in these states.

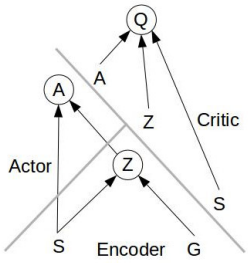


Figure 2.5: Variational Autoencoder feeding input to Actor

Set return,
$y_i = r_i + \beta D_{KL}[p_{enc}(z|s_t, g_t)|q(z|s_t)]$
$\quad +\gamma Q'(s_{i+1}, \mu'(s_{i+1}, Z|\theta^{(\mu+Z)'})|\theta^{Q'})$
and update Actor,
$\nabla_{\theta^{\mu+z}} J = \frac{1}{N}\Sigma_i(\nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i,Z)} \nabla_{\theta^{\mu+z}} \mu(s, Z|\theta^{\mu+Z})|_{s_i}$
$\quad -\beta\nabla_{\theta^{\mu+z}} D_{KL}[p_{enc}(z|s_t, g_t)|q(z|s_t)])$

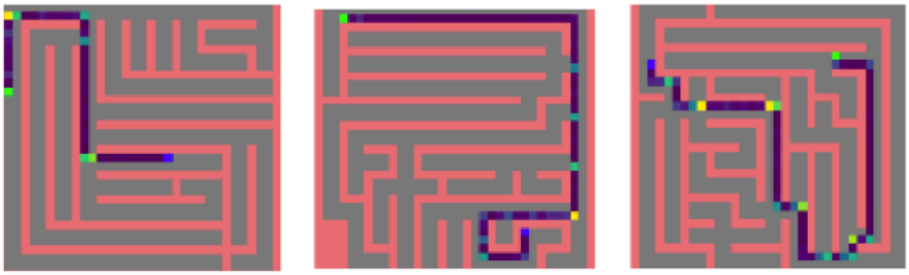


Figure 2.6: Waypoints with Information Regularization [39]

2.6 Grids and Spaces

The spaces need not be regular but we try to discretize them in terms of regular grids for the purpose of computational simplicity. This oversimplification of the spaces results in high computational costs during the planning. A wide range of hierarchical planning methods uses regular spaces because they can still provide an edge over DRL approaches. Another approach can be to build a graph with irregular spaces and find only those sections of the map which have a real appeal for navigation. This may reduce the graph considerably in size and provide a better trajectory for navigation.

2.7 Hierarchical Modelling

The problem of DRL approaches is that they try to learn all the aspects of the task in one network or process. However, the task might have subtasks that are widely different from each other. This is one of the reasons that they need such a high number of samples and learning resources. Also, they do not resemble the cognitive handling of the task. There are several approaches

that divide the task into a layer for the representation of the environment and another layer for control of the agent.

2.7.1 A*

We usually rely on external localization and mapping and the map of the environment is available in the form of a 2^d grid. As this map can be seen as a graph in itself an approach like A* [40] can be used directly to search a path. This algorithm provides better results than traditional search algorithms owing to its heuristics. The problem that remains with this approach is that every single bit of information is available for search while it can be reduced considerably.

2.7.2 Probabilistic Roadmaps (PRM)

A recent implementation by the name Probabilistic Roadmaps Reinforcement Learning (PRM-RL) [14] samples points randomly on a map. This approach finds which of the points are connected using a DRL-based low-level controller to slowly develop a representation of the environment. This representation can be used to query path for a longer distance. This results in a reduced representation of the map. The problem with this approach is that a uniform sampling of points on the map does not resemble cognitive mapping. For example with this approach, there can be many points sampled in a tunnel while there is only one decision point on either end of the tunnel. Also, a highly cluttered space is also sampled with a few points only. The reason for such a problem is the uniform sampling. This approach used an RL controller at the lower-level contrary to previous implementations of straight-line controllers. Having RL, the agent can go around the corners where there is no direct connectivity, yet with this approach there were disconnected regions on the map where there was a narrow path in between. The authors hinted upon post-processing by sampling more points in such a region and it clearly proves the problems of regular spaces.

2.7.3 Quadtrees

The approach of Quadtrees [15] divides the spaces into 4 equal spaces until it becomes obstacle-free. Hence, with this approach, the density of vertices of the graph represents the spatial situation of the real map and presents a better picture in terms of connectivity. This is a theoretic approach to solve the problem of stated above but still does not resemble a cognitive process.

2.7.4 Rapidly Exploring Random Tree (RRT)

RRT [13] searches any map with the goal to fill up all the space. It limits the sampling of a new point under the constraint of reachability with maximum distance from the current position. In this approach, a point is sampled anywhere within a distance threshold. In the beginning, the longer distances are covered while the shorter sections follow in the later stages. Also, any space is divided into spaces of unequal size and a higher density of vertices can be found in areas with higher obstacle density. A drawback of this approach is that it generates a graph that is loop-free while a loop might exist in the real map. Also, the maximum distance is a factor that needs to be chosen carefully and gives different results. The process represents computational optimality but no cognitive partitioning of spaces.

Chapter 3

Simulation Environment Unity

The physically-plausible maze environment is built using Unity[41] that is a cross-platform game engine widely used in the game industry. The availability of a physics engine makes it a comfortable option for our task. Additionally, ml-agents[42, 43] provides an easy medium to communicate with an external learning procedure. The model designed in Unity is shown in figure 3.1. In the figure, the components of the model can be seen in the left panel, scripts in the bottom panel and component configuration in the right panel.

3.1 Model Design

The model consists of following important components:

1. Base-Board
2. Walls and Goal
3. Ball

Base-Board

The base-board takes actuation from the external learner. The actuation is in the form of absolute angular position of the X and Z axis. The Y-axis is the vertical axis in Unity over which we want to retain 0 rotation. This actuation is applied at the center of the base-board. An instantaneous execution of high angular movement induces a linear velocity in the ball which is perpendicular to the surface. A similar effect is seen when the ball is far from the center of applied action wherein the same angular movement generates different perpendicular linear velocity. This causes the ball to bounce and get affected

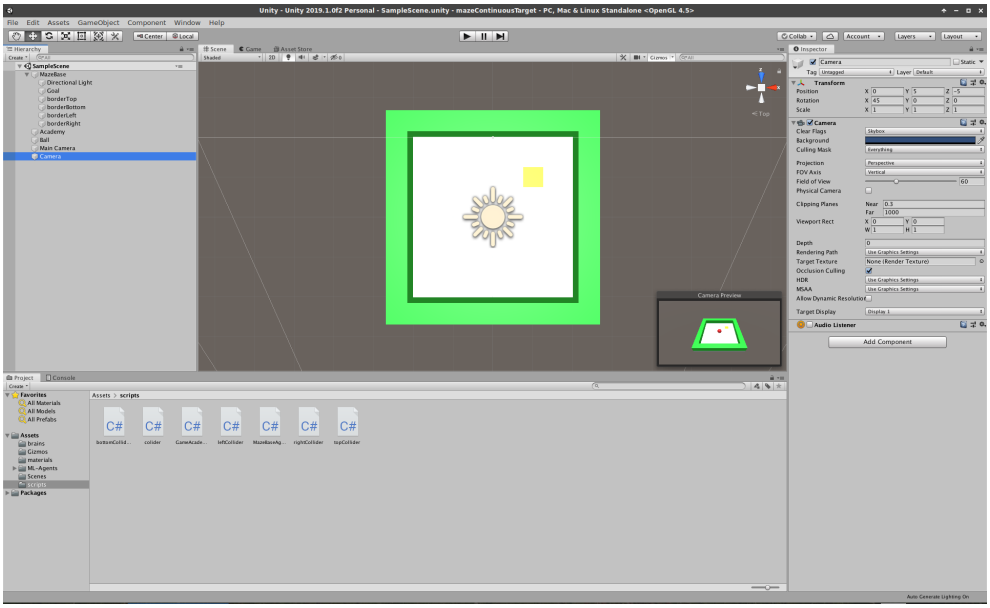


Figure 3.1: Model Design

by other physical parameters such as restitution and additional friction. Hence, the actuation takes place using an internal PID controller.

$$error = Vector3(command\_x - existing\_angular\_rotation\_x, \\ 0, \\ command\_z - existing\_angular\_rotation\_x)$$

$$P = 0.5$$
$$I = 0.001$$
$$D = 0.0001$$

$$integrator += error * Time.deltaTime$$
$$diff = error/Time.deltaTime$$
$$force = error * P + integrator * I + diff * D$$

The output of PID is clipped by a dynamically calculated value which is inversely proportional to the distance of the ball from the center.

$$maxForce\_x = absolute(Arctan(0.2/Ball\_position\_x))$$
$$maxForce\_z = absolute(Arctan(0.2/Ball\_position\_z))$$

$$finalForce\_x = Clip(force.x, -1 * maxForce\_x, maxForce\_x)$$
$$finalForce\_z = Clip(force.z, -1 * maxForce\_z, maxForce\_z)$$

While most of the configuration is done using GUI, the base-board component has a script attached which takes care of tasks that are done per frame update as well as per physics update. This includes action execution, detection of various triggers and preparation of observation for the external learner per frame update step. The PID controller works per physics update step based on the command set at the last frame update step. There are only two physical parameters to be tuned viz. friction and bounce set slightly above zero. The base-board is a kinematic object and hence a change in the position and orientation can only happen by explicit actuation. It does not act under gravity. There are also auxiliary components of light and camera attached to the base-board. They are required to generate visual observation with required brightness.

Walls and Goal

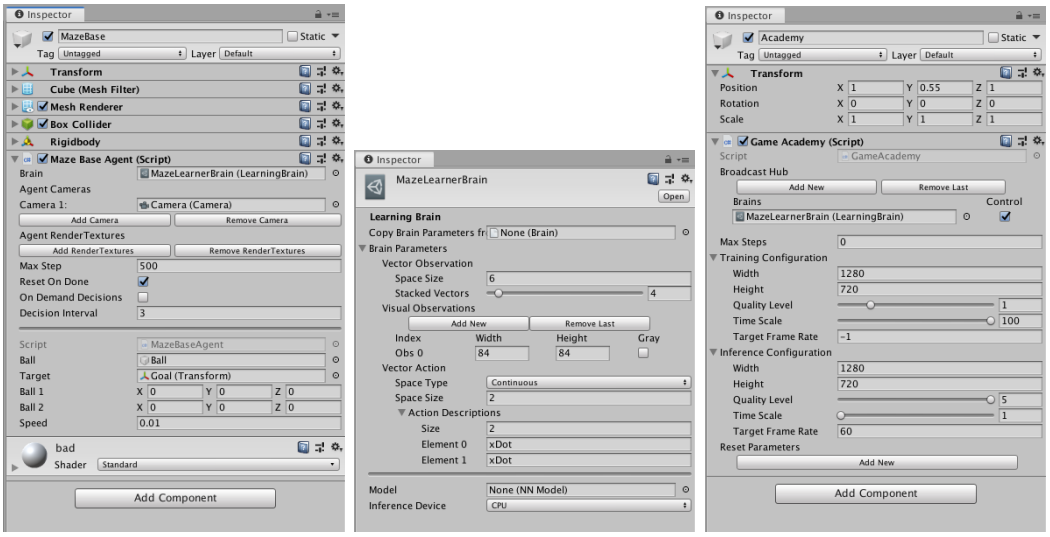
The structure of the maze is given as a one-hot encoded array. The walls are fixed on the base-board and constrain the movement of the ball in the maze. The goal is a differently colored (yellow) thin sheet that is also fixed to the base-board. The walls and goal have a collider component which sets a boolean of collision. This is used to generate a desired reward or penalty. The same values of base-board apply for friction and bounce and being attached to board they also do not act under gravity.

Ball

The ball is the only component that moves under the influence of gravity. The movement induced in the ball is a result of the actuation given to the base-board. It is not fixed to the base-board and can fall out if the condition may arise. The friction and bounce are similar to base-board.

3.2 Communication with Learner

This section focuses on the ml-agents which facilitate external learners to communicate with this environment. For the process of communication, an abstraction of all the above components can be called a Model. The academy controls various parameters of training like a parameter server. The important



(a) Model (b) Brain (c) Academy

Figure 3.2: Model Training and Communication Structure

parameters include the simulation speed and step limit for episodic scenarios. It can also have a script and can be used creatively for various purposes. The simulation can be sped up 100 times. The same model is built with two different simulation speeds. The real-time speed is used for test purposes while a 100 times increase is used for training. In our model, the step limit of the episodic scenario is controlled by the external learner than the model itself. The frame rate is defined as -1 while we set the time scale to be 0.01 signifying a time difference of 10ms between two consecutive frames.

Inside the environment, the external learner is represented by a wrapper component called external brain. The data transmitted to and from the model is defined in a brain. As in figure 3.2, one can see that outgoing data is a vector of values defining the state of the model. The values can be of any fundamental data type. A separate visual input of the model can also be added. An additional facility to stack some historical values in output is also available. However, a limitation is that this stacking can not be used together with the "gym" wrapper of Unity Environment. The brain also defines the values that are expected to come from an external learner to the model. This can be a continuous or discrete value. The brain and academy are attached to the model. Below figure 3.3 shows the model in action.

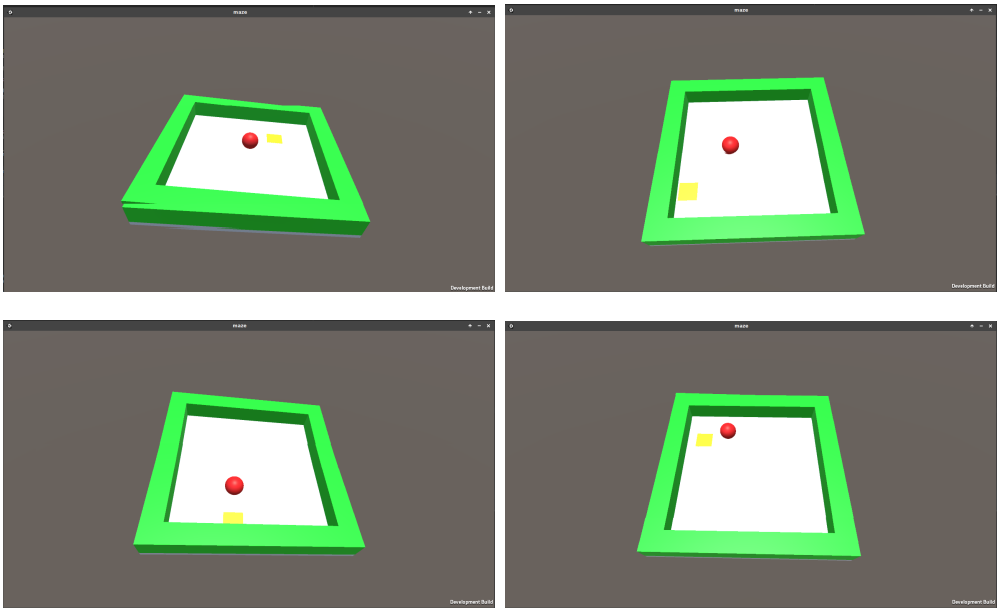


Figure 3.3: Model in Action

3.2.1 Miscellaneous Schedules

The model also has the capability of initializing the ball or goal at random places inside the maze. For this, it exempts the places that are already filled up by the walls using the array of maze structure. The reward for all the different collisions is defined separately by the collider component and hence, can be mapped to different values.

Chapter 4

Experiments

Definitions

Point : A point on the map represented by a 2^d cartesian coordinate system.

Fovea : This is an odd-sized 2^d square matrix centered at the current position of the gaze fixation representing the limit of visibility. It is called fovea analogous to the visible fovea.

Line of Sight (LoS) : If a line exists between any two points, without an obstacle on the way.

Flood fill : Starting at a point establishing the reachability of every other point in the given space.

Symmetric Flood fill : Similar to flood fill but progressing in all directions with equal steps.

Movable space : This is the space represented by the set of the points which are free from any collision object.

Directly accessible space : This is the space that can be accessed by the ball without violating the boundary of the fovea. The space in the fovea can be partitioned by collision objects into disconnected sections. The section that contains the ball in such a case is the Directly accessible space.

4.1 Higher-Level Controller

4.1.1 Representation of learning

The approaches discussed in the following sections establish the connectivity of the map. This is saved in the form of an undirected graph. The vertices of the graph are the points on the map. The edges of the graph represent the connectivity of the points in the environment. The possible movement is symmetric making it an undirected graph. Since the binary status of connectivity is the only concern for this planning, the weights of the edges are uniform. The vertices can save additional information at the time of graph construction but are irrelevant afterward. After this phase, the graph is available for queries from the low-level controller. The starting and goal positions of a trial may not necessarily correspond to the vertices of the graph. The nearest neighbor approach is used to find the vertices close to the start and the goal. The nearest neighbor should also qualify for direct reachability within the fovea wherever fovea is relevant. The graph search is done between these vertices to emit a trajectory. We used Dijkstra's algorithm for the search however, any other search algorithm does not interfere with the results of the below approaches.

4.1.2 Curiosity Driven Planning

This approach starts with a random point in the movable space. The directly accessible space in the current fovea is established by flood fill. On this directly accessible space, a boundary mask is applied and the direction of arrival is also masked to find the remaining directions. This is called the curiosity vector for that point. A point is then sampled from this curiosity vector. This sampled point becomes the current point and the same process is applied until a dead-end is reached. At a dead-end, the process backtracks the path and checks every point for residual curiosity until one is found. This process continues until all the points have all its curiosity satisfied. The detailed process is in the below algorithm 1. The obtained graph is shown in the figure 4.1.

Algorithm 1: Curiosity Driven Search

$current\_point$: (x, y) point for current position of the planner gaze
$previous\_point$: (x, y) point for previous position of the planner gaze

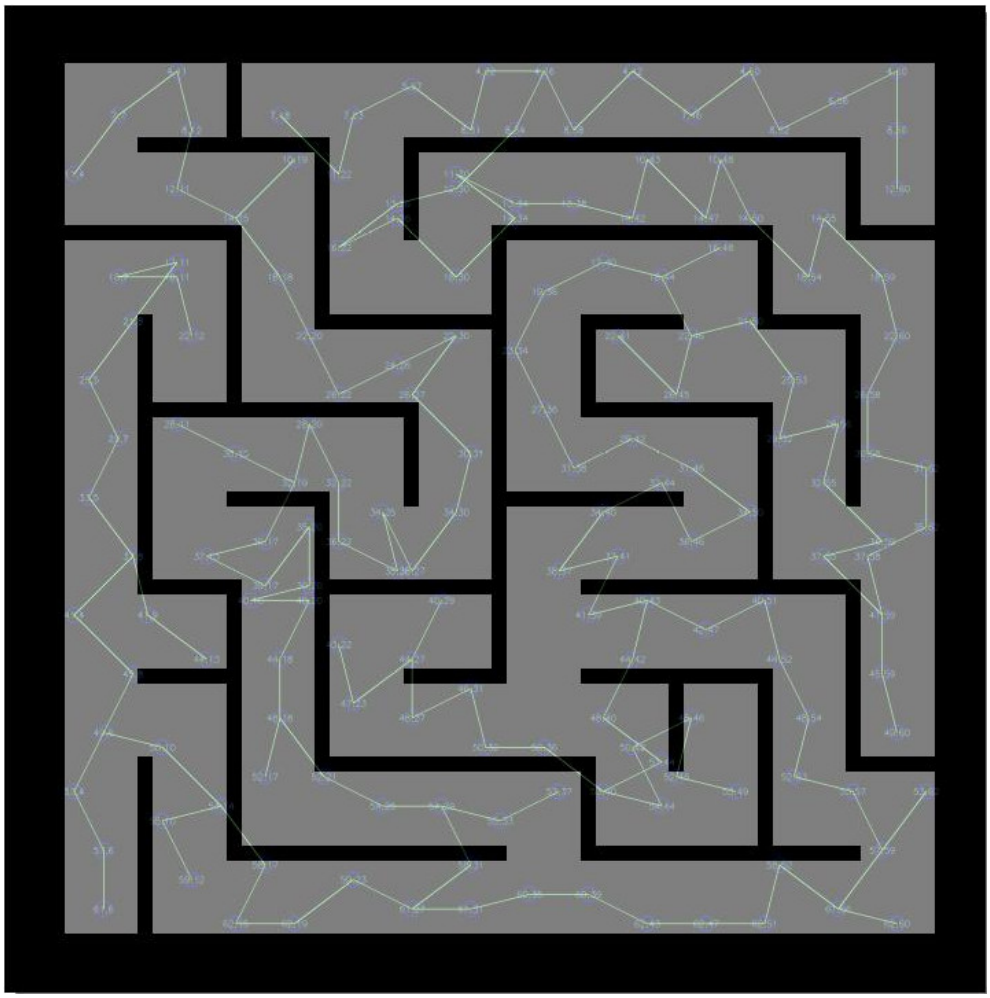


Figure 4.1: Fovea flood fill and Curiosity based next point sampling

$fovea$: A square matrix as per above definition.
b_mask : A matrix of the size of the fovea with elements in the first and last, row and
column as 1 and all other values as 0
d_mask : A matrix of size of fovea with elements between $previous\_point$ and $current\_point$ as 0 and all other values as 1
G : An undirected graph $G(V, E)$ with a set of
vertices $V(node\_id,\ curiosity\_vector)$
and edges $E(source\_node,\ dest\_node,\ distance)$

BEGIN:

Get a random *current_point*
Set *previous_point* = *current_point*
Build fovea around *current_point*
Build *curiosity_vector* by
    Flood fill fovea of *current_point*
    Apply boundary mask
    Apply direction mask to find unexplored directions
Add node to the graph with *current_point* as *node_id* and the obtained *curiosity_vector*

LOOP:
Find node with *current_point* as *node_id*
Obtain *curiosity_vector* of *current_point*

If *curiosity_vector* has unexplored directions:
    Sample an unexplored direction from *curiosity_vector* and set it as *current_point*
    Mask the *curiosity_vector* with *d_mask* between *current_point* and *previous_point*
    Update the *curiosity_vector* of *previous_point*
    Build fovea around *current_point*
    Build *curiosity_vector* by
        Flood fill fovea of *current_point*
        Apply boundary mask
        Apply direction mask to find unexplored directions
    Add node to the graph with *current_point* as *node_id* and the obtained *curiosity_vector*
    Add edge in the graph between *previous_point* and *current_point* with distance 1
    Set *previous_point* = *current_point*

Else:
    Backtrack the graph G to find a node with *curiosity_vector* having unexplored direction.
    Set its *node_id* as *current_point*, or declare complete.
END:

4.1.3 Convexity Driven Planning

This approach identifies convex regions and their connectivity in the map. The maze under consideration is built from rectangular blocks and hence, the

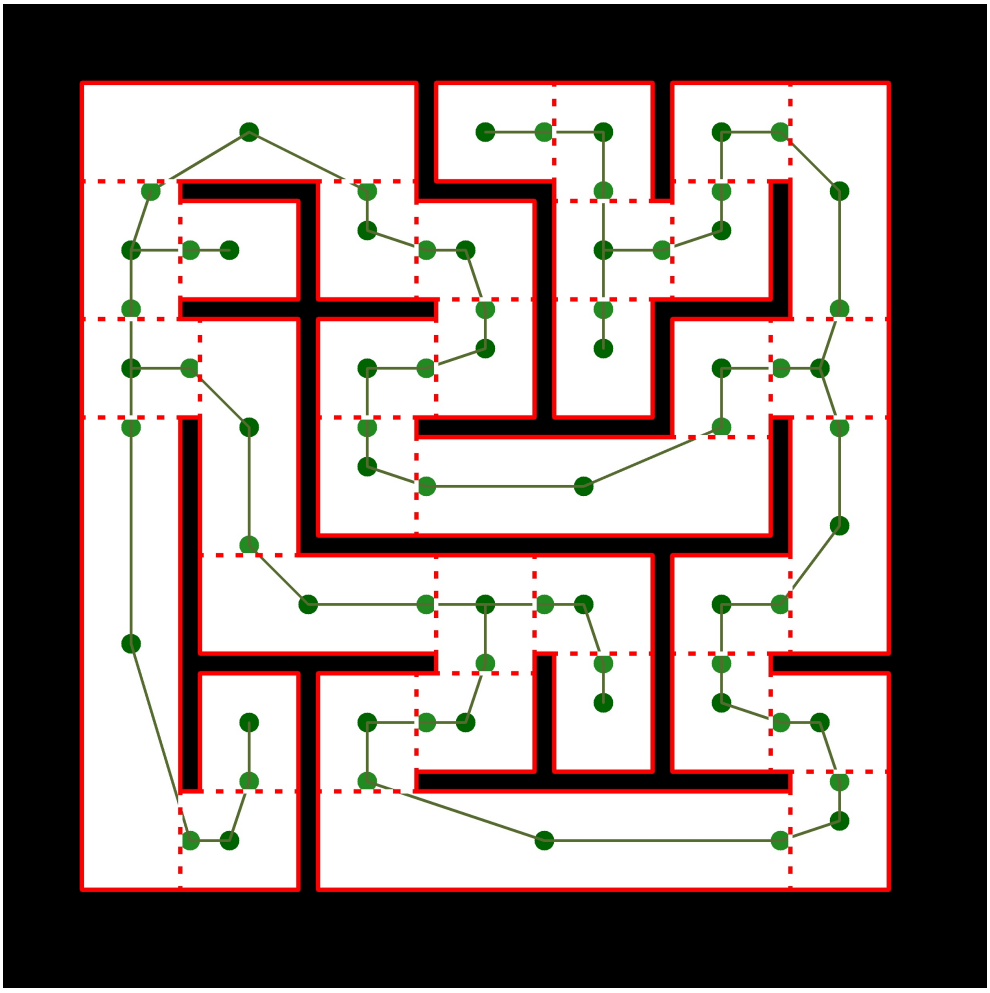


Figure 4.2: Rectangular Convexity and convex region gateway based next point sampling (additional images in figure 4.15)

algorithm tries to fit rectangular blocks to space and see the connectivity from that perspective. A rectangular shape will not only suffice for the map under consideration but it will also keep the algorithm simple for proof of concept. The generated graph is to be compared for the quality of connectivity and coverage instead of computational complexity.

Convex area by the best fit of geometrical (Rectangle) shapes

The algorithm starts by selecting a random point and expanding from that point using rectangular fitting until constrained by walls in all the directions. It then finds the gateways and does a similar expansion from these gateway

points to fill the entire map with rectangular spaces. The centers of the spaces are connected to its gateways which in turn are connected to the centers of next convex regions. The detailed process is in the below algorithm 2. The obtained graph is shown in figure 5.1. The built graph samples few points to represent the map. The graph built by geometric centers has one point sampled (slightly dark green) at the center of every convex region represented by red rectangles. Also, the points shown by slightly light green color are the gateway points. The gateways are marked with Red/white dotted line. The connectivity between two convex centers always goes through a gateway point to ensure a Line-Of-Sight for a naive low-level controller.

Algorithm 2: Convexity Driven Search (Rectangular Fit)

$current\_point$: current position of the planner gaze
$node\_center \rightarrow [node\_center\_x, node\_center\_y]$: the center of any convex area
$node\_limits \rightarrow [limit\_x\_min, limit\_x\_max, limit\_y\_min, limit\_y\_max]$: the limits of a convex region on (X, Y)-Axis
$node\_gateways \rightarrow [[gateway\_x_1, gateway\_y_1], ...[gateway\_x_n, gateway\_y_n]]$: the list of boundary points that open in another convex area.
$node\_property$: a list of, $[node\_center, node\_limits, node\_limits]$
G : An undirected graph $G(V, E)$ with a set of
vertices $V(node\_id,\ node\_property)$
and edges $E(source\_node,\ dest\_node,\ distance)$

BEGIN:
Get a random *current_point*
Build *node_property*
By expanding around *current_point* as a rectangle until bounded by walls on all sides.
Set limits of this rectangle as *node_limits*
Find *node_center*
Find *node_gateways*
Add node with *current_point* as *node_id* and the obtained *node_property*
Add node with each point in *node_gateways* as *node_id* and empty *node_property*
Add edge between *current_point* and each point in *node_gateways*

LOOP:
Find node with *current_point* as *node_id* and get *node_property*
If any point in *node_gateways* does not has two neighbors:

Build *node_property* as above
Add node with *current_point* as *node_id* and the obtained *node_property*
Add node with each point in *node_gateways* as *node_id* and empty *node_property*
Add edge between *current_point* and each point in *node_gateways*
set *current_point* = *node_center*
Else:
Backtrack the graph G to find a node with a point in *node_gateways* not having two neighbors
Set its *node_id* as *current_point*, or declare complete.
END:

Attraction Points instead of geometric center

A problem with the above approach is that usually, the geometric center is not a point that has intuitive appeal to a human planner. This approach will try to find better attraction points and modify the notion of the node center. For this, it narrows the tunnels and the junctions using blurring as in figure 4.3.

Blurring :

$$maze[i, j] = 1/9 * sum(maze[i - 1 : i + 1, j - 1 : j + 1])$$

The blurring highlights certain locations on the map. These locations (attraction points) (figure 4.4) seem to be a good candidate for the sampling of waypoints and hence the algorithm 3, can locate them programmatically to build the graph. The built graph is shown in figure 4.5. The attraction points are sampled based on the background brightness after blurring the maze image. In this case, with only a few outliers, the attraction points are always centered around turns and forks. This removes the need of including gateway points in the graph although they are required for the building process. Although the notion of attraction points seems sufficient to generate multiple waypoints at interesting positions on the maze, considering it globally might not be the best approach especially if the visual properties of the regions are not universally similar. Hence, it still uses the notion of rectangular convex regions for the segmentation of space before sampling the waypoints. Figure 4.6 compares the geometric centers Vs. attraction point-based centroids approach.

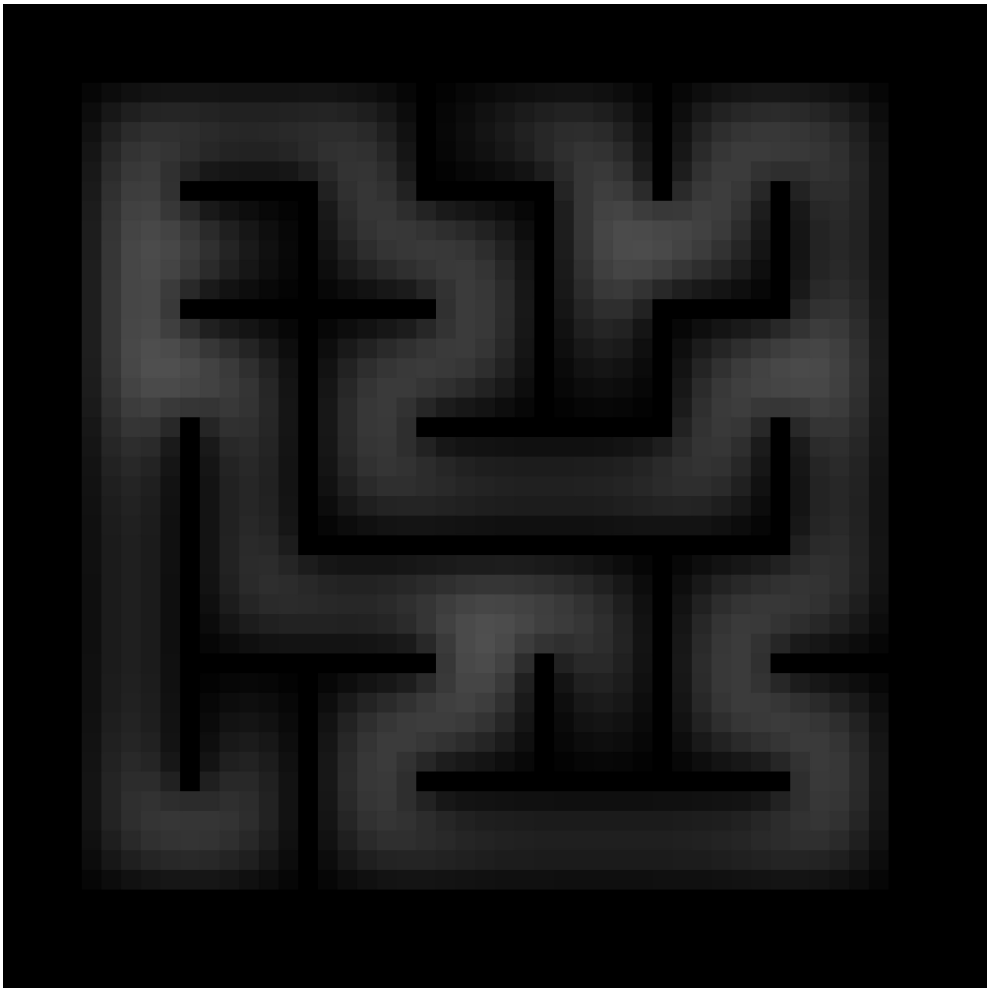


Figure 4.3: Blurring of the map (additional images in figure 4.16)

Algorithm 3: Search for Attraction Points

BEGIN:

$convex\_region\_limits \rightarrow [limit\_x\_min, limit\_x\_max,$
$limit\_y\_min, limit\_y\_max]$
the limits of a convex region on (X, Y)-Axis

$convex\_region \rightarrow maze[limit\_x\_min : limit\_x\_max,$
$limit\_y\_min : limit\_y\_max]$
using $convex\_region\_limits$

Flatten $convex\_region$

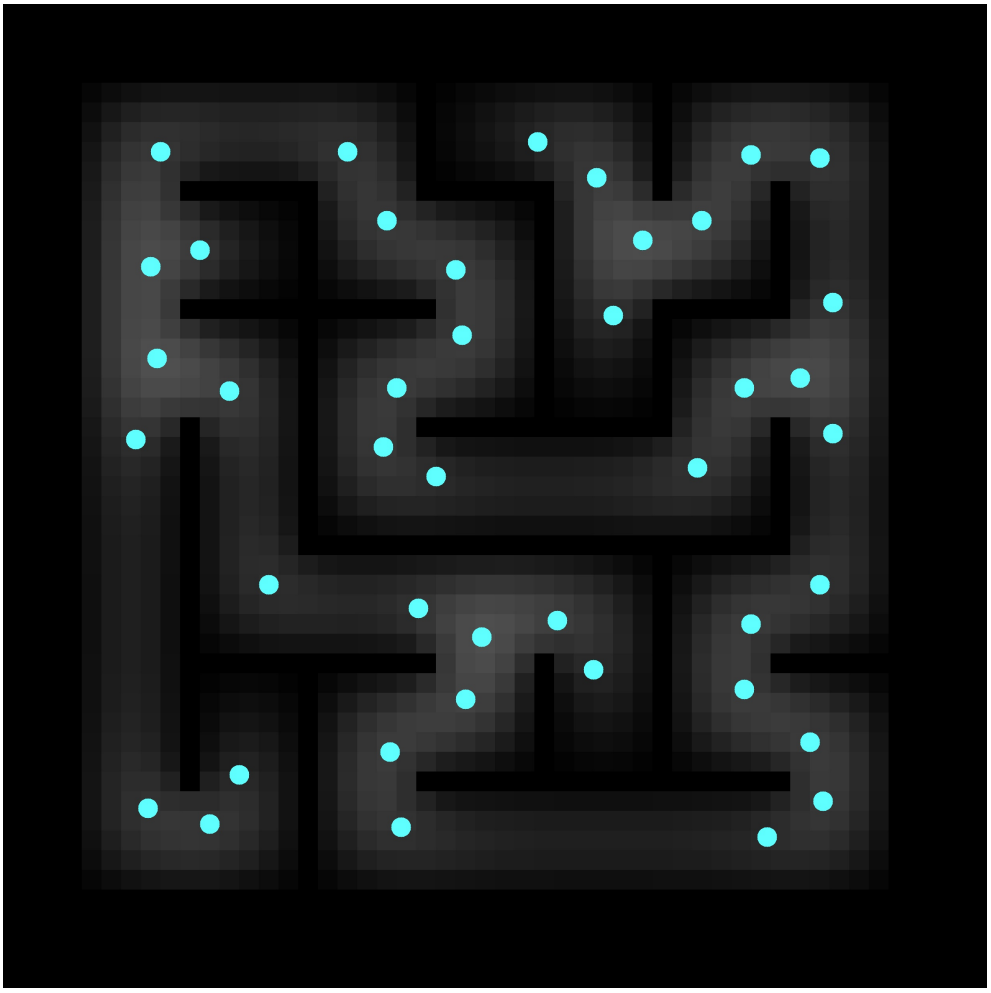


Figure 4.4: Attraction points by Blurring (additional images in figure 4.17)

Build *threshold* as top 20% pixels of *convex_region* based on intensity and save the lowest of this.
Change the *convex_region* values to 255 if greater than *threshold* and 0 if smaller than *threshold*.
Assign label to connected regions using *skimage.measure.label*
Search for unique labels and coordinates belonging to each label.
Find the centroid of these coordinates.

Return these centroids as Attraction Points.
END:

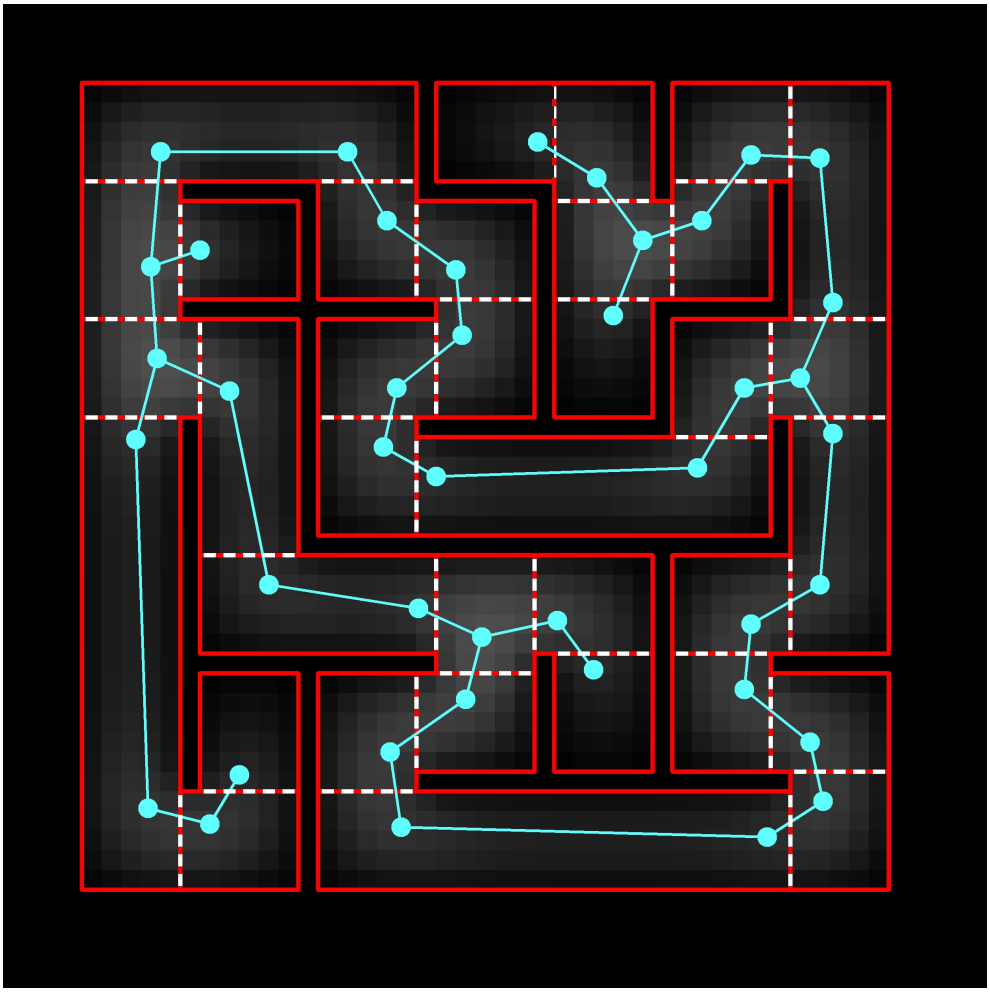


Figure 4.5: Graph Built by Attraction points (additional images in figure 4.18)

4.1.4 Other Options for convexity

Flood fill and convex area

To evaluate flood fill as a potential candidate for generating convex regions, the limit can no longer be a fixed fovea rather a violation of convexity. Also, the direction of progress in every step matters and hence the progress should be symmetric in all the directions. With these modifications, a notion of symmetric flood fill is developed. This approach starts at a random point and makes progress in all the directions until the condition of convexity is violated. Having a wall in any direction does not violate convexity until the

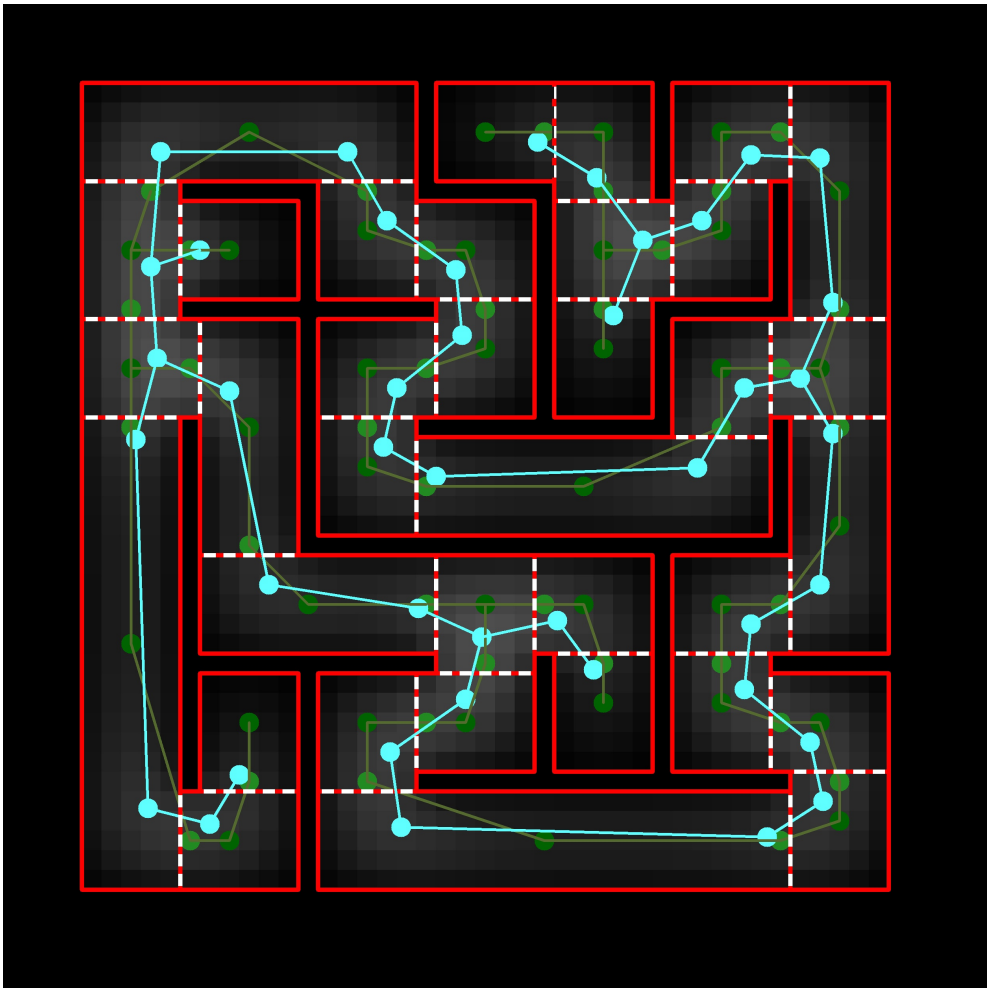


Figure 4.6: Comparison of graphs by Geometric centers (Green) Vs. Attraction Point (Cyan) based centroids (additional images in figure 4.19)

progress starts to encircle the wall.

Line-Of-Sight and convex area

A wide range of robots act inside the environment. They use light detection and ranging (LIDAR) sensors for the perception of the environment. LIDAR can be used to establish LoS and can be used in creative ways [14]. The task at hand observes the environment from an overhead perspective but a human expert can determine this fact even from a top view. Since it can be useful to establish the convexity of a region this method used a two-point form of the equation of a line and checking if any point on this line has an obstacle or

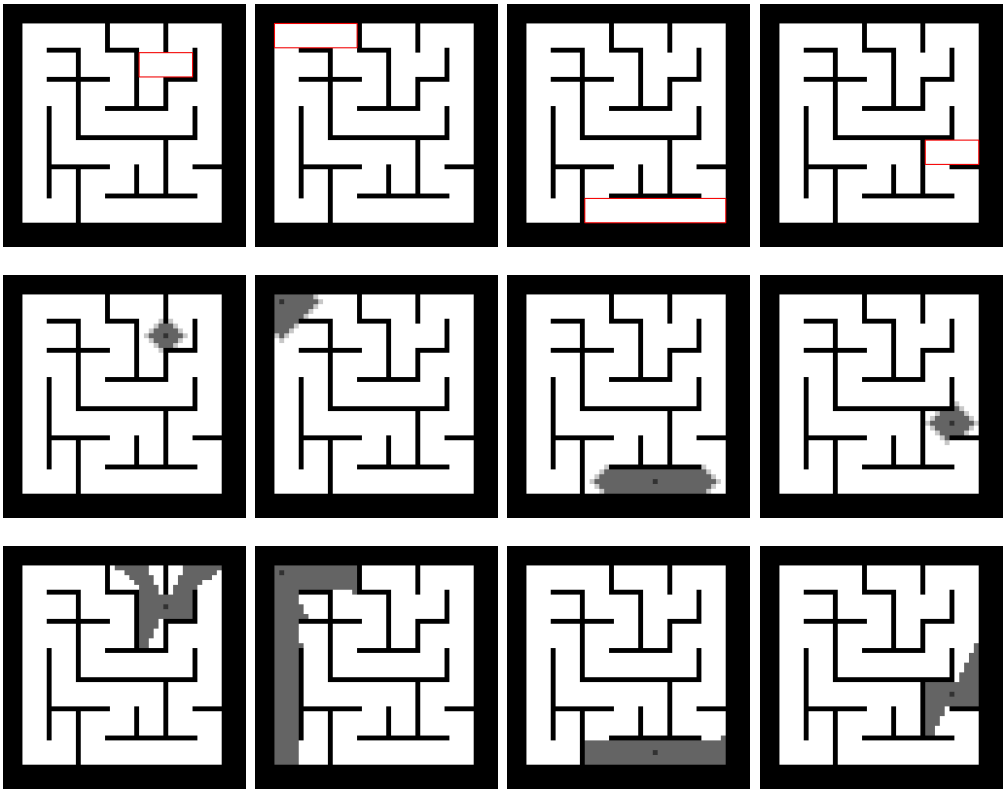


Figure 4.7: Row 1: Geometric Convexity, Row 2: Flood Fill, Row 3: Line Of Sight

not.

$$y = \frac{(y_2 - y_1)}{(x_2 - x_1)} * (x - x_1) + y_1$$

4.2 Low-Level Controller

4.2.1 Direction-vector Controller

This method is the most naive approach towards motor control for the maze. The method finds the direction vector from the current position of the ball to the intermediate waypoint. The direction vector is mapped to the command given to the maze on the X and Y-axis of rotation as below. The command is recalculated at every time step and hence as the size of the distance vector decreases, the command is also reduced. The problem seen is that such a

control responds poorly to the inertia and hence the ball shows little stopping or slowing down of the velocity while reaching a target/waypoint. Some smoothening can be achieved by changing the waypoint prematurely based on the location of the next waypoint. But the movement is still not well controlled and collision with the walls can be seen. Since this approach does not account for the dynamics of the environment, the approach works with actions of a very small magnitude and takes a long time to complete the trajectory.

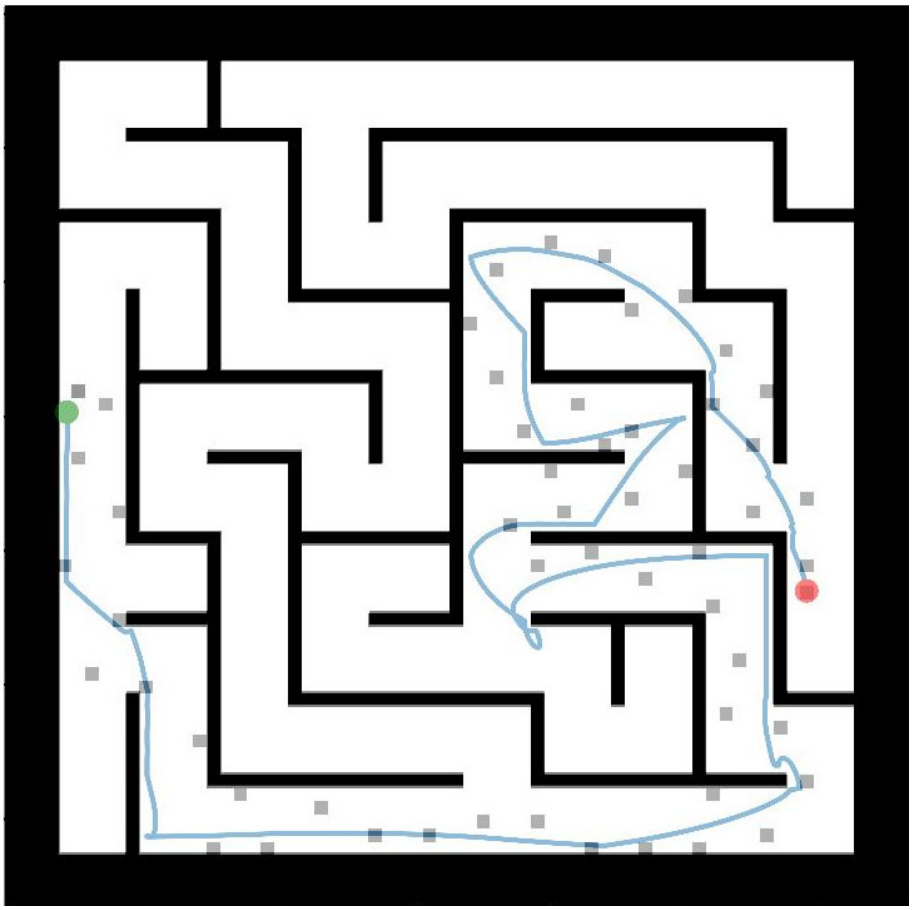


Figure 4.8: Direction-vector Controller; Agent path with planned trajectory.(Red - Start, Green - Goal)

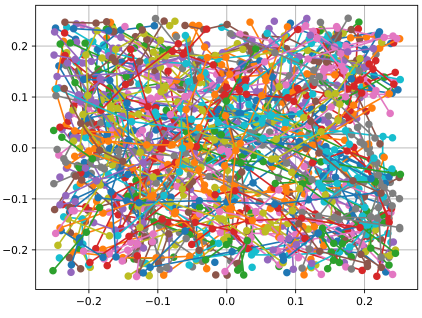
Direction Vector :

$$dir\_x = goal\_x - ball\_pos\_x$$

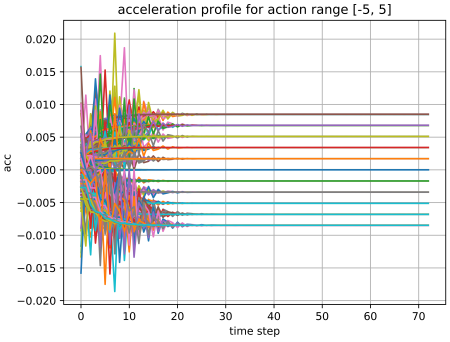
$$dir\_y = goal\_y - ball\_pos\_y$$

Command Vector :

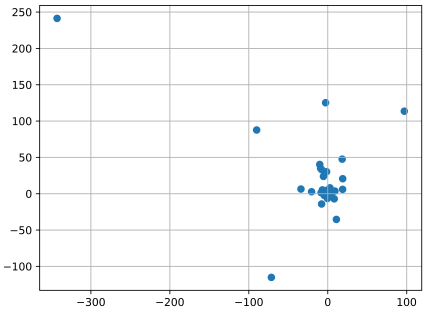
$$cmd\_x = cmd\_x\_prev + min(ball\_pos\_x, limit\_x)$$
$$cmd\_y = cmd\_y\_prev + min(ball\_pos\_y, limit\_y)$$
where, $[limit\_x, limit\_y] = [4, 4]$



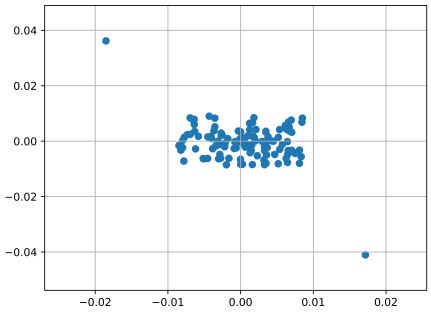
(a) Sampled Trajectories (Span of Board on X and Y axis from -0.25 to 0.25)



(b) Acceleration Profile (acceleration in $unit/sec^2$)



(c) Acceleration centroid (acceleration in $unit/sec^2$)



(d) Command centroid (Angular Position in Degrees)

Figure 4.9: Unsupervised Learning of Environment Dynamics

4.2.2 Controller using Unsupervised learning of environment dynamics

It was required to account for the dynamics of the environment. This approach samples a limited number of random commands (ranging between 1500 to 2000 repeated for 75 time steps of 10ms each) and induced effect on the ball as shown in figure 4.9(a). This data is then used to find quads of the following structure and the induced acceleration can be seen in figure 4.9(b).

$$[acceleration\_x,\ acceleration\_y,\ action\_x,\ action\_y]$$

These quads are used to learn clusters shown in figure 4.9 (c) and (d), using k-means. It is then used to do a reverse lookup of desired change in ball position to desired acceleration and use k-nearest neighbor to interpolate the desired command. The desired command is clipped beyond [-8,8] degree of movement as a safety against outliers.

4.2.3 PID

Using the same clusters as above a PID controller similar to the internal PID controller of the simulation environment is used. This is required to generate controlled command and remove the need for fixed clipping. A PID controller itself needs intensive tuning before they can start giving good results. An executed trajectory is shown in figure 4.11. It has an oscillation around a waypoint, some deviation from the planned trajectory and collision with walls due to its bad response to inertia and potentially lack of tuning.

$$\begin{aligned} error = Vector3(&desired\_command\_x - existing\_angular\_rotation\_x,\\ &0,\\ &desired\_command\_z - existing\_angular\_rotation\_z) \end{aligned}$$

$$P = 0.5$$
$$I = 0.01$$
$$D = 0.001$$

$$integrator += error * Time.deltaTime$$
$$diff = error / Time.deltaTime$$
$$force = error * P + integrator * I + diff * D$$

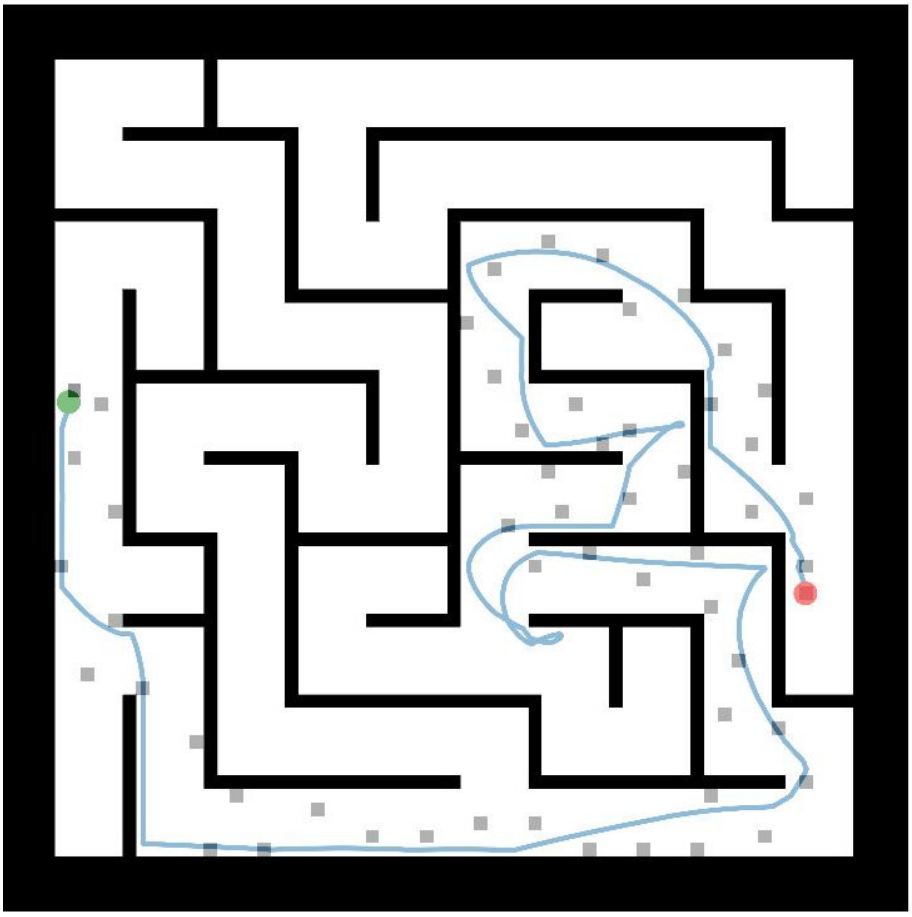


Figure 4.10: Controller based on unsupervised learning; Agent path with planned trajectory. (Red - Start, Green - Goal)

4.2.4 DRL based controller

None of the above approaches account for the environment dynamics completely and therefore a DRL-based method was used. Being in the domain of continuous control, the use of DDPG to learn the dynamics of the environment is the best option. The baseline implementation of OpenAI is used for this purpose. The learning model uses a concatenation of below as state input:

- The last four positions of Ball in (x,y) coordinates.
- The last four orientations of Maze on (x,y) axis.
- The location of Goal in (x,y) coordinates.

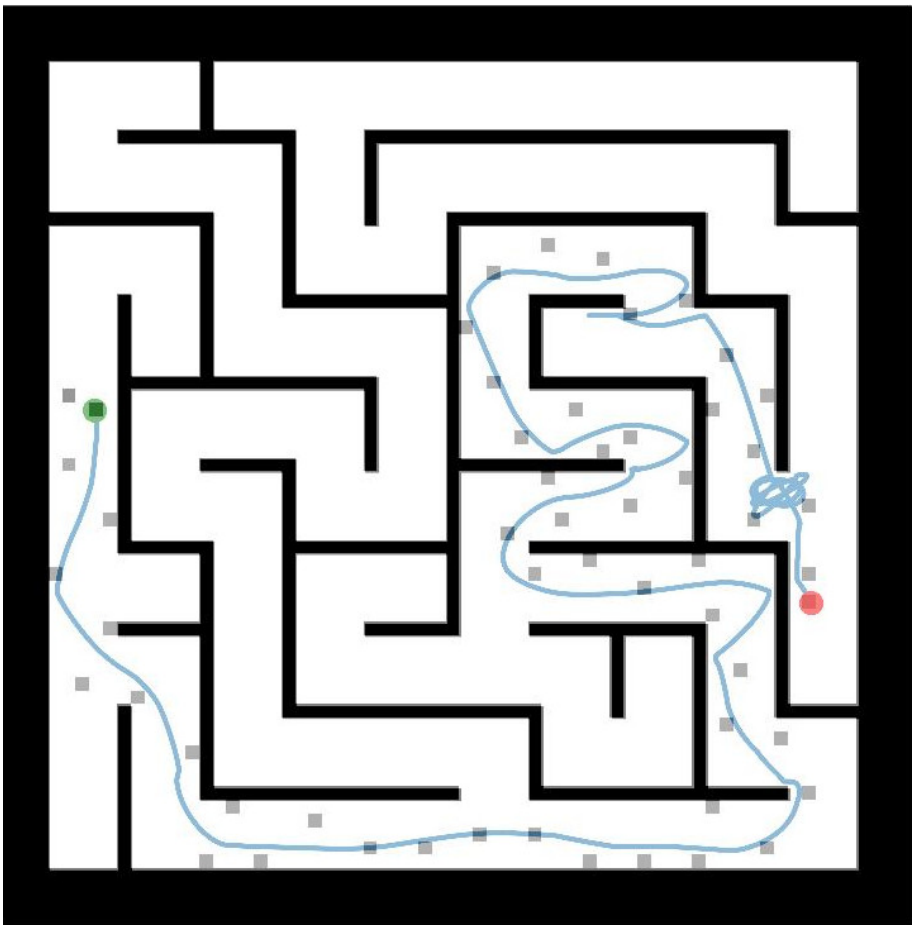


Figure 4.11: Controller based on PID; Agent path with planned trajectory. (Red - Start, Green - Goal)

- One-hot coded vector to represent the board with wall blocks as 0 and blank positions as 1.

The input action to the model was an absolute angular position on the X and Y-axis. There is no internal PID controller in this case and the range of action is [-5,5] degrees. The training was done in an episodic scenario with a step limit of 500.

The initial experiment shows learning of movement of the board which induces velocity in the ball to move towards the goal. The matrix used for the determination of learning is the success rate. The achieved success rate reached above 90.0% in around 20000 episodes however, the training was continued as the steps per episode were going down showing further

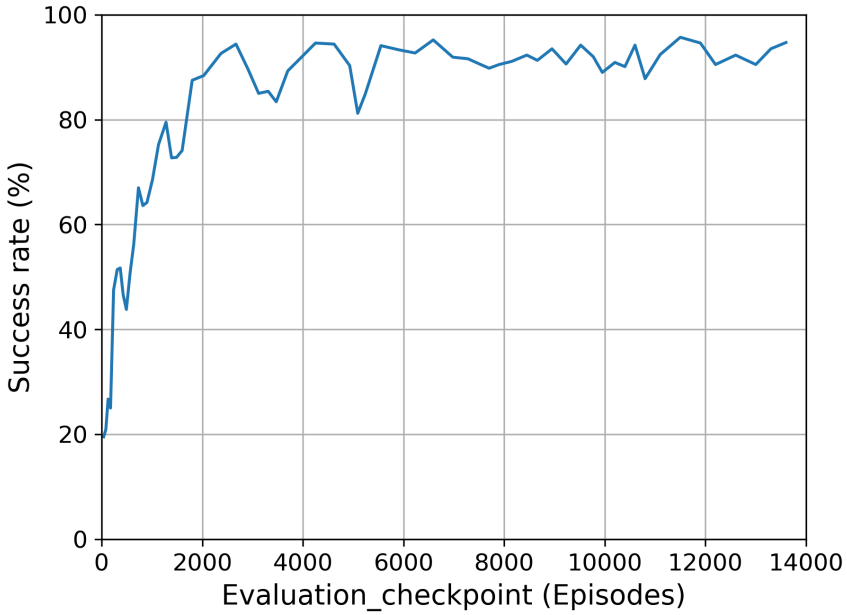
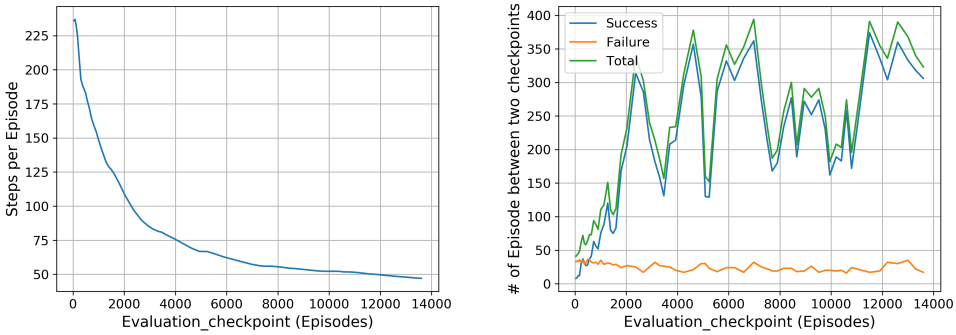


Figure 4.12: DRL based controller (Success Rate)

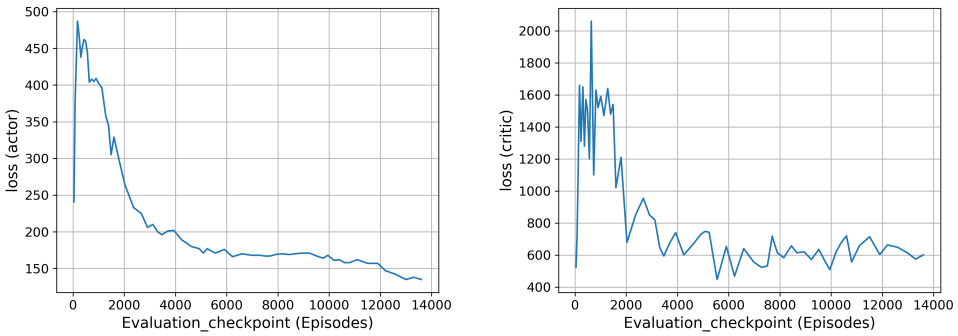


(a) (b)

Figure 4.13: DRL based controller (Brevity of episode)

improvement of the model. The number of episodes in the evaluation phase was also changing continuously.

The model uses goal as input and hence represents a Universal Value Function Approximator. It is also a pre-requisite of the Hindsight Experience Replay (HER). To train this model a very aggressive version of HER was used.



(a) (b)

Figure 4.14: DRL based controller (Losses)

Every experience is replayed by substituting the goal with the immediate next position of the ball and giving it a full reward. This way the number of successful experiences was highly increased and helped in learning the model.

Figure 4.12 shows the improvement of the success rate in percentage, between every evaluation cycle. Figure 4.13 (a) shows the decrease in the number of steps per episode between every evaluation cycle showing the refinement of the model. A similar matric is in figure 4.13 (b) showing an increase in the number of episodes per epoch as training progresses, but has more erratic behavior. The decrease in the loss of actor and critic is shown in figure 4.14 (a and b).

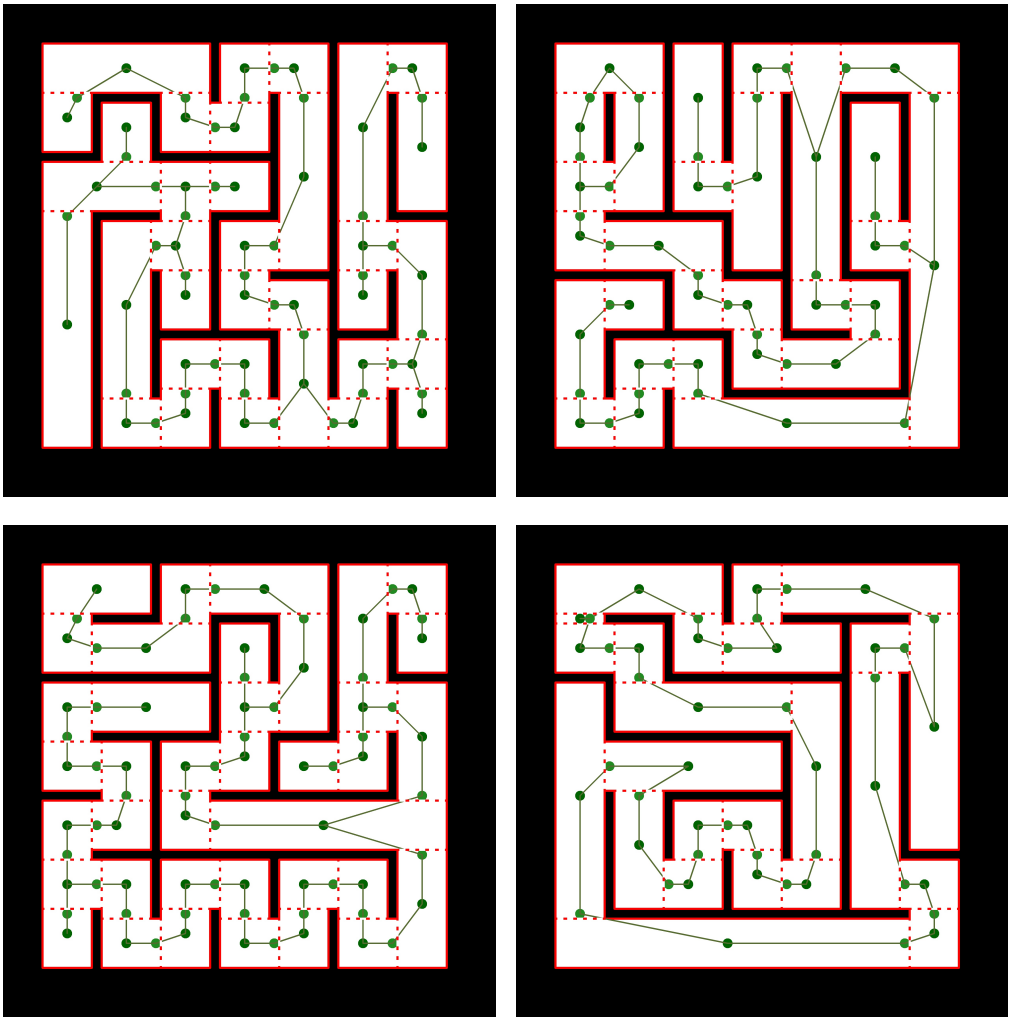


Figure 4.15: Rectangular Convexity and convex region gateway based next point sampling

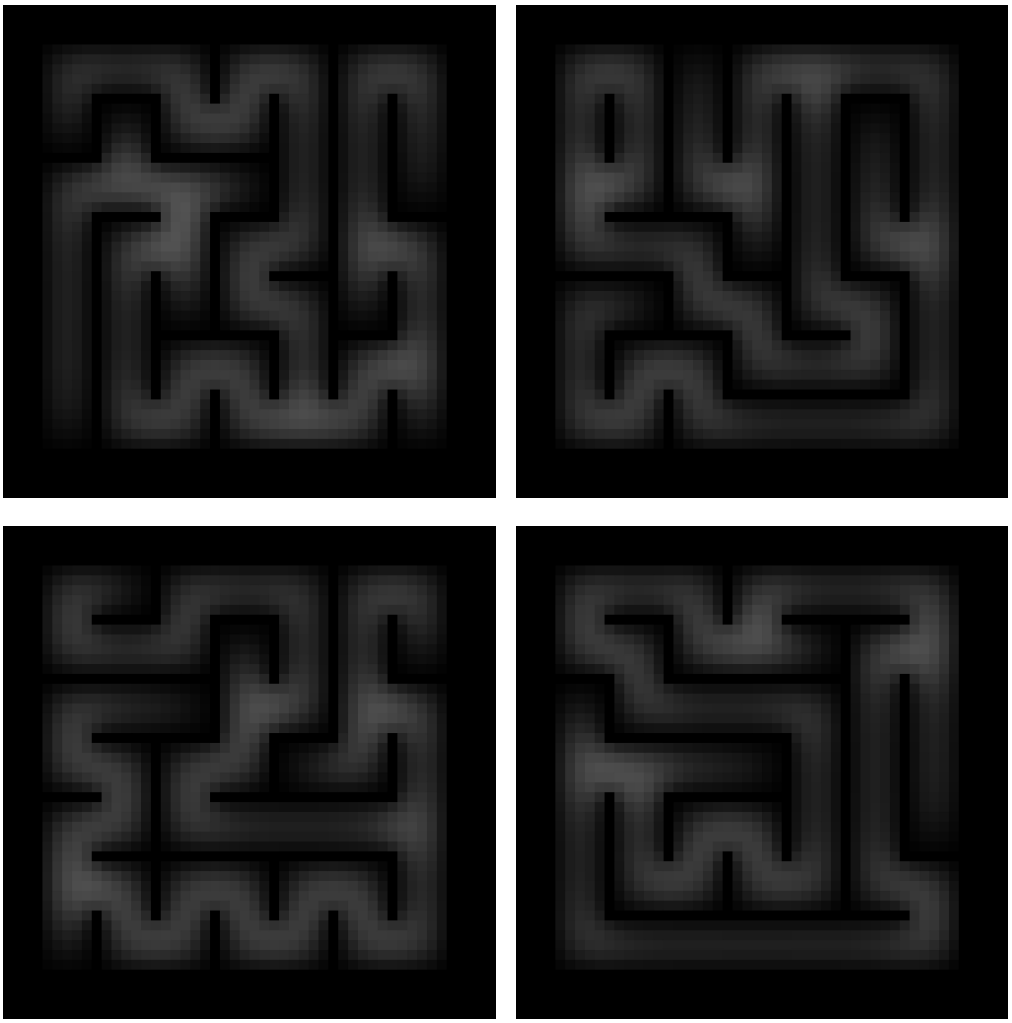


Figure 4.16: Blurring of the map

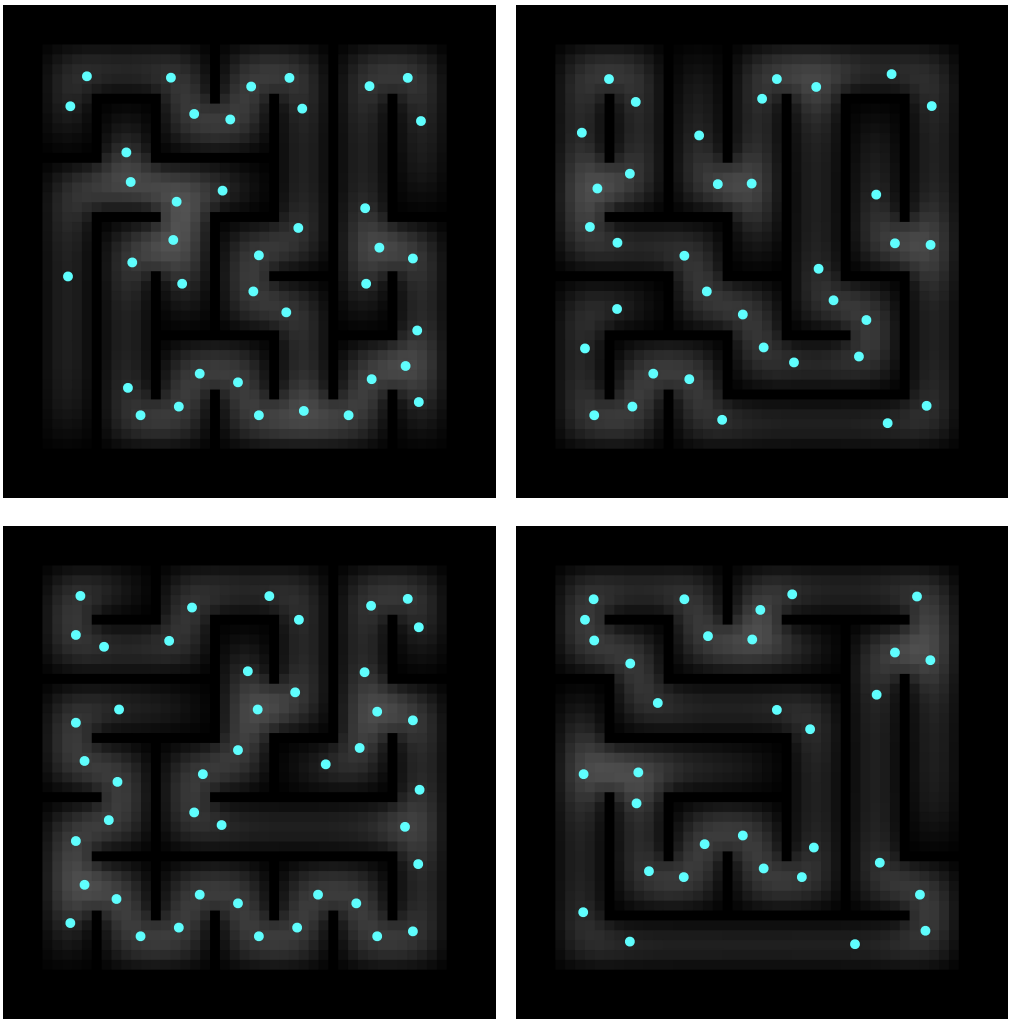


Figure 4.17: Attraction points by Blurring

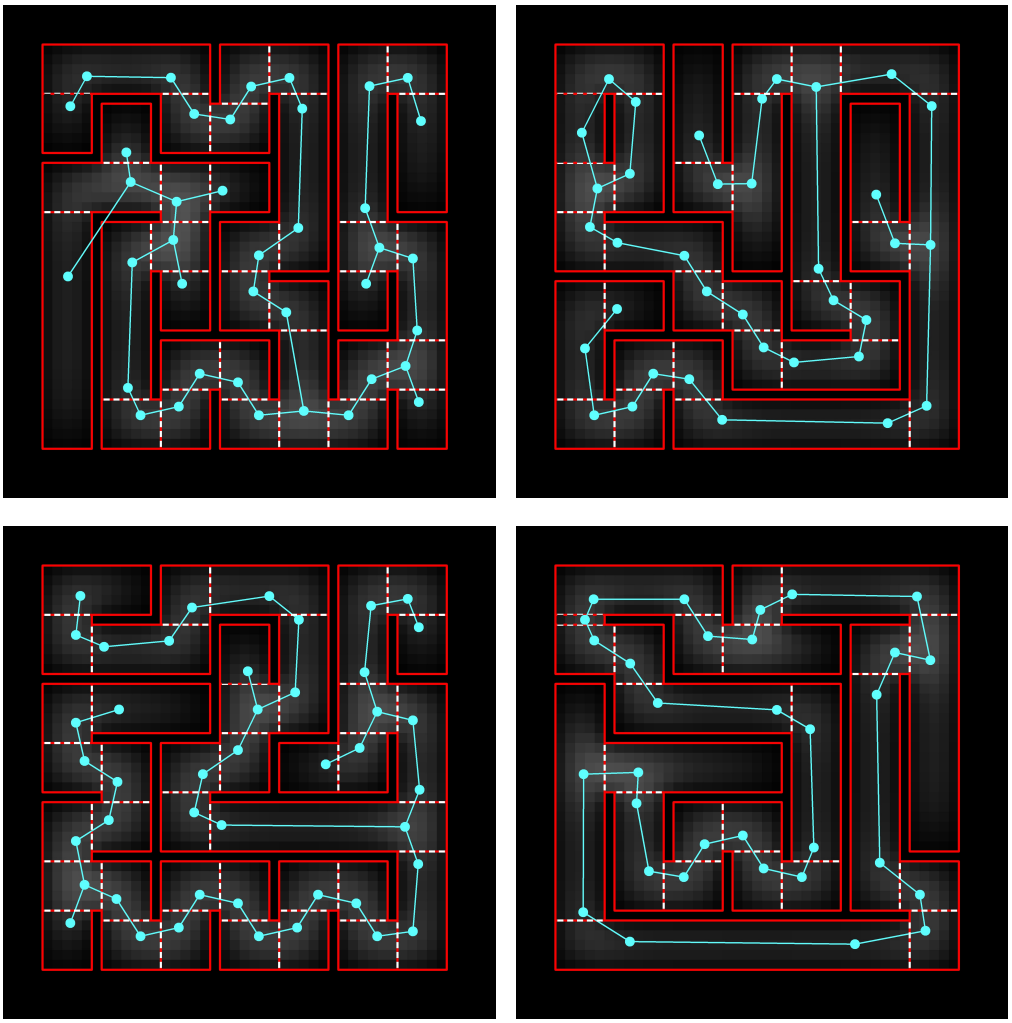


Figure 4.18: Graph Built by Attraction points

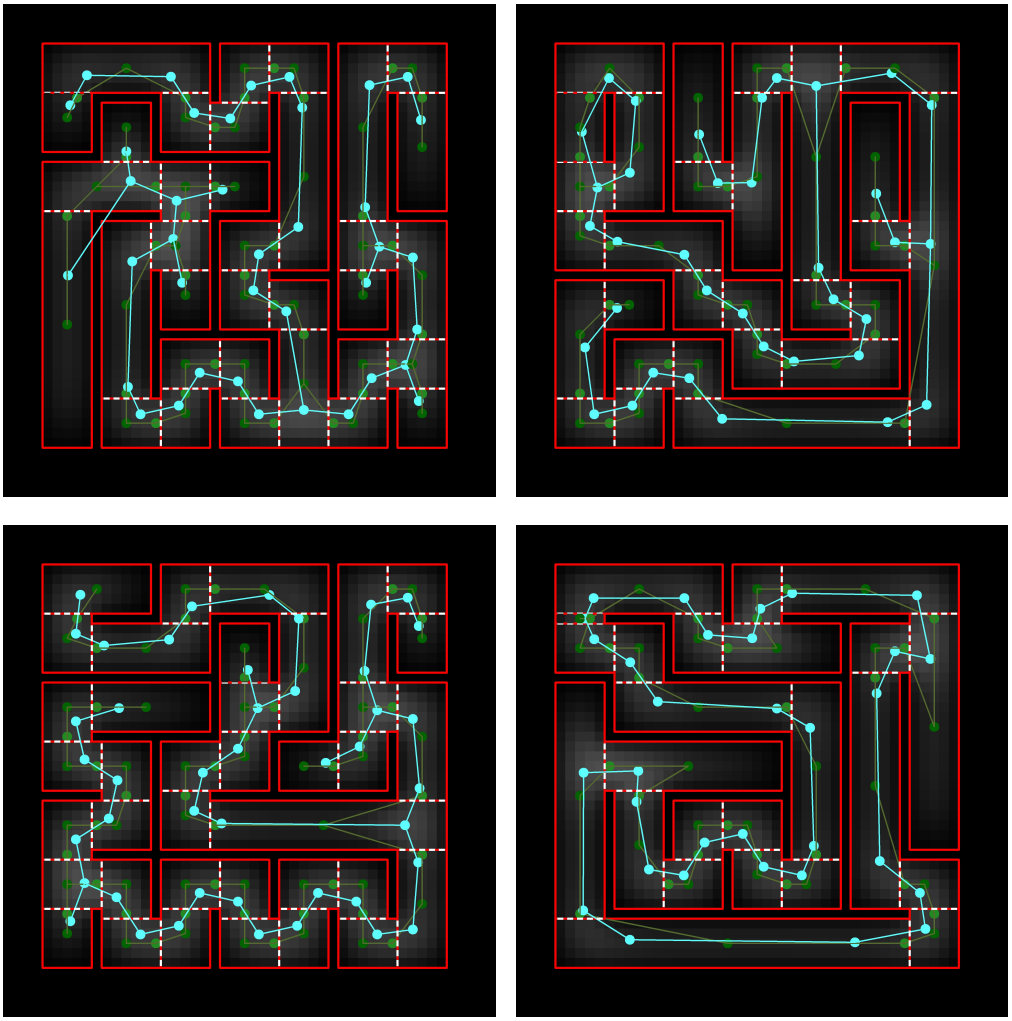


Figure 4.19: Comparision of graphs by Geometric centers Vs. Attraction Point based centroids

Chapter 5

Discussion and Conclusion

5.1 Discussion

In general, the approach of hierarchical modeling is better than DRL based approaches because the high-level planning is done in under 1000 foveal steps and simple control is learned in around 20000 episodes using DDPG. The same DDPG based learning did not show any progress with a full maze in a similar number of steps. Running the algorithm for a longer period also did not progress and showed the behavior of learned unsolvability.

5.1.1 Visual Planner

Curiosity-driven Vs. Random

If the curiosity-driven sampling is compared with random sampling [14], the number of sampled points is similar in the count. The reason for this is fixing the size of fovea as a limit and sampling a point on the boundary of the fovea gives at least three points on every axis inside the fovea. This kind of limitation of maximum space to sample a point is also seen in random sampling-based approaches. While none of them correspond to the structure of maze concerning cognitive mapping, the result is an unnecessarily dense graph which will have high computational cost at the time of execution of the trajectories built from this graph.

Convexity as segregating boundary

Random sampling is not a cognitive strategy for mapping. Also, being agnostic and sampling based on fixed criteria of distance is not very helpful. Rather it makes more sense to look for novelty, a segregation boundary or an

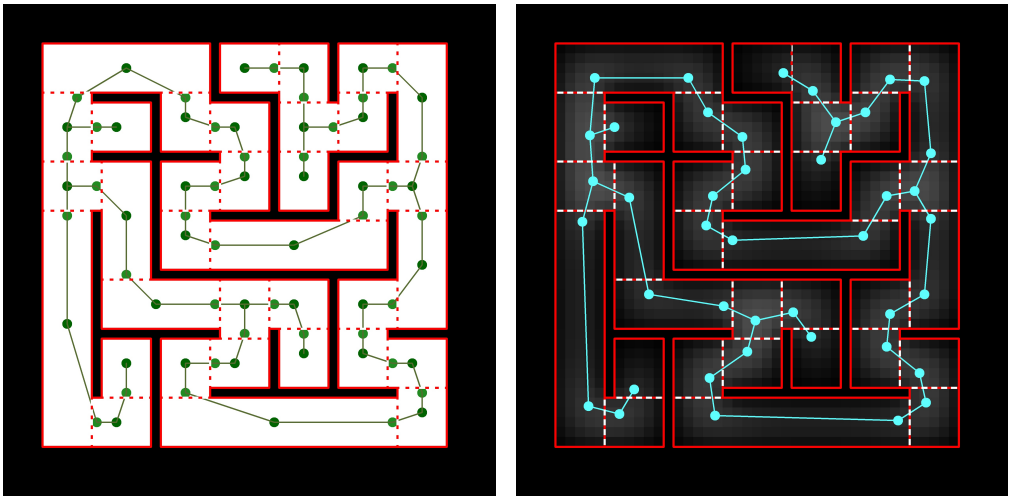
inherent structure. On this front, the hypothesis of convex regions represents a variable area which tells about a structural property. It also produces a sparsely populated graph (figure 5.1 (a)). Consequently, it promises less computational complexity at the execution level making itself a potential strategy for inspection.

Convexity and irregular shapes

The benefits of convex spaces with the constraint of fixed general geometric shapes degrade if the map has irregularly shaped regions. It will not generalize well on a wide range of tasks. Hence, the notion of convexity should be more fluid. On this front, an evaluation of two more strategies viz. symmetric flood fill and line-of-sight was done. Although they do not seem to solve the problem completely, they may be helpful in individual applications owing to their specific properties. Also, a suitable mix of them can give us a better view of convexity.

Center of gaze-fixation

Another problem is related to the fact that the geometric centers are not always useful. Stopping for decision in a tunnel where the gaze is coming from one direction and can go further in only the other direction adds very less to cognitive behavior. For example, in a room with two doors both of them in one corner, the stopping point would be centered around the corner instead of the geometric center of the room. In the case of the maze, the interesting points are at the edges of the convex regions from where new regions are connected as they represent the task-related options. It might need to sample more than one point in a convex region, and even if only one point is sufficient, that point may lie away from the center. An approach by blurring of the graph and then find the center of high pixel intensity as the location of sampled points (figure 5.1 (b)) caused the emergence of places which have specific features such as turns, forks, and cross-roads. The points are sampled at these locations. This way the generated graph corresponds better to the map of the maze. The technique of blurring is specific to the maze structures in use and one can explore approaches contemplating this phenomenon.



(a) (b)

Figure 5.1: Geometric Center Vs. Attraction Points

5.1.2 Sensomotoric Controller

Naive Control

On the front of sensorimotoric control, several naive approaches were evaluated including a controller that translates direction vector into command, using clusters to approximate the command and a PID controller to use this approximation for smoothness. They all perform poorly in the environment as none of them can account for inertia. The commands do not have a counterforce to slow down the movement when the ball is nearing the goal. And we do not provide a model for the desired movement profile with acceleration-decelaration secgments.

DRL-based Control

We can provide a model of the movement behavior that can stop the ball at the goal position. Alternatively, we can learn the clusters with the behavior of stopping at the goal under the influence of physics. Since these approaches come with intensive handcrafting, an approach based on Deep Reinforcement Learning is implemented. A DDPG [30] based model in the mode of Universal value Function Approximation [32] using Hindsight Experience Replay [29] was learned. The results show that the approach can learn to move the ball from the current location to a local goal. However, the desired behavior must care about the direction of approach and goal achievement with a desired

low-velocity profile and should be the future direction of work.

5.2 Conclusion

The representation of a map in terms of a graph is a well-established method. The search techniques are also widely tested and proven. However, to build such a graph for any arbitrary map, an algorithm must identify the features which are important. One stable feature is the locations of the task-related options on a map. Computational methods using KL-divergence [38, 39] to differentiate task-related option locations from non-option locations are being developed. However, this method is integrated into DRL approaches which is sample inefficient and the concept is shown on discrete action spaces relying on the action choices and goal information. We need to find methods that can determine the task-related option locations using visual inspection of the qualitative aspects of the map being investigated. The sample size of experience or observations should also resemble the human level of intelligence in visual inspection. We evaluated the convexity of spaces on the map as a feature for the overhead view and found it helpful to build a more informative map of the environment.

On the front of sensorimotoric control, DRL remains the best option because it does not need a model and can learn complex reactive dynamics in the short term. This is done with general algorithms and achieve performance comparable to or better than human performance.

5.3 Future Work

Methods used in this work viz. geometrical shapes, flood fill and line of sight rely upon expanding around a point. A direction to be explored further is the automatic identification of convex spaces either by mixing them or novel methods. Also finding the waypoints in these spaces need to find locations that are adjacent to next convex spaces. The low-level controller would need to learn a behavior that takes care of the direction of approach and goal achievement with a desired low-velocity profile.

References

[1] Demis Hassabis et al. "Neuroscience-inspired artificial intelligence". In: *Neuron* 95.2 (2017), pp. 245–258.

[2] Andrew Melnik et al. "Modularization of End-to-End Learning: Case Study in Arcade Games". In: *arXiv preprint arXiv:1901.09895* (2019).

[3] Malte Schilling and Andrew Melnik. "An approach to hierarchical deep reinforcement learning for a decentralized walking control architecture". In: *Biologically Inspired Cognitive Architectures Meeting*. Springer. 2018, pp. 272–282.

[4] Michael S.A. Graziano, Charlotte S.R. Taylor, and Tirin Moore. "Complex Movements Evoked by Microstimulation of Precentral Cortex". In: *Neuron, Vol. 34, 841–851* (2002).

[5] Volodymyr Minh et al. "Human-level control through deep reinforcement learning". In: *Nature* (2015). URL: `https://doi.org/10.1038/nature14236`.

[6] O. Nachum et al. *Data-efficient hierarchical reinforcement learning*. 2018.

[7] A. Levy, R. Platt, and K. Saenko. *Hierarchical reinforcement learning with hindsight*. 2019.

[8] Alexander Maye and Andreas K Engel. "Extending sensorimotor contingency theory: prediction, planning, and action generation". In: *Adaptive Behavior* 21.6 (2013), pp. 423–436.

[9] Anthony Francis et al. "Long-range indoor navigation with PRM-RL". In: *arXiv preprint arXiv:1902.09458* (2019).

[10] H. T. L. Chiang et al. *Learning navigation behaviors end-to-end with-autorl*. 2018.

[11] C. Lynch et al. *Learning latent plans from play*. arXiv preprint arXiv:1903.01973. 2019.

[12] A. Nair et al. *Overcoming explorationin reinforcement learning with demonstrations*. 2018.

[13] Munir Naveed, Diane E. Kitchin, and Andrew Crampton. "Monte-Carlo Planning for Pathfinding in Real-Time Strategy Games". In: 2010.

[14] Aleksandra Faust et al. *PRM-RL: Long-range Robotic Navigation Tasks by Combining Reinforcement Learning and Sampling-based Planning*. 2018.

[15] R. A. Finkel and J. L. Bentley. *Quad trees a data structure for retrieval on composite keys*. 1974.

[16] "https://digitaluncovered.com/wp-content/uploads/2016/06/m-et-gazeplot.jpg". 2020(accessed February 24, 2020).

[17] "https://www.researchgate.net/profile/Thomas_Guntz/publication/332494135/figure/fig1/AS:748877225402369@1555557652979/Example-of-eye-tracking-data-for-an-user-on-a-chess-configuration-Figure-3-Architecture.ppm". 2020(accessed February 24, 2020).

[18] "http://helpingdyslexia.com/wp-content/uploads/2011/10/EYE-TRACK-SCANS2.jpg". 2020(accessed February 24, 2020).

[19] Alfred L Yarbus. *Eye movements and vision*. Springer, 2013.

[20] G. Tirelli et al. "Saccades and smooth pursuit eye movements in central vertigo". In: *ACTA otorhinolaryngologica ITALICA 2011* (2011).

[21] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Ed. by Michael Jordan, Bernhard Schölkopf, and Jon Kleinberg. Information Science and Statistics. Springer, 2006.

[22] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press, 1998.

[23] Christopher J.C.H. Watkins and Peter Dayan. "Technical Note: Q-Learning". In: *Machine Learning* 8.3 (1992), pp. 279–292. ISSN: 1573-0565. DOI: 10.1023/A:1022676722315. URL: https://doi.org/10.1023/A:1022676722315.

[24] J. O'Neill et al. "Play it again: reactivation of waking experience and memory". In: *Trends Neurosci.* (2010).

[25] L. J. Lin. "Reinforcement learning for robots using neural networks". In: *DTIC Document* (1993).

[26] J. L. McClelland, B. L. McNaughton, and R. C. O'Reilly. "Why there are complementary learning systems in the hippocampus and neocortex: insights from the successes and failures of connectionist models of learning and memory". In: *Psychol. Rev. 102* (1995).

[27] Hado van Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-learning". In: *CoRR* abs/1509.06461 (2015). arXiv: 1509.06461. URL: http://arxiv.org/abs/1509.06461.

[28] Tom Schaul et al. "Prioritized Experience Replay". In: (2016). arXiv: 1511.05952.

[29] Marcin Andrychowicz et al. "Hindsight Experience Replay". In: *CoRR* abs/1707.01495 (2017). arXiv: 1707.01495. URL: http://arxiv.org/abs/1707.01495.

[30] David Silver et al. *Deterministic Policy Gradient Algorithms*. 2014. URL: http://proceedings.mlr.press/v32/silver14.pdf.

[31] Timothy P. Lillicrap et al. "Continuous control with deep reinforcement learning". In: (2016). arXiv: 1509.02971.

[32] Tom Schaul et al. "Universal Value Function Approximators". In: (2015).

[33] N. Tishby, F. C. Pereira, and W. Bialek. "The Information Bottleneck Method". In: (1999).

[34] Bharat Prakash et al. *On the use of Deep Autoencoders for Efficient Embedded Reinforcement Learning*. 2019. URL: https://arxiv.org/pdf/1903.10404.pdf.

[35] Pierre Baldi. *Autoencoders, Unsupervised Learning, and Deep Architectures*. 2012. URL: http://proceedings.mlr.press/v27/baldi12a/baldi12a.pdf.

[36] Diederik P Kingma and Max Welling. *Auto-Encoding Variational Bayes*. 2013. URL: http://arxiv.org/abs/1312.6114.

[37] D. J. Strouse et al. "Learning to Share and Hide Intentions using Information Regularization". In: *Advances in Neural Information Processing Systems 31*. Ed. by S. Bengio et al. Curran Associates, Inc., 2018, pp. 10249–10259. URL: http://papers.nips.cc/paper/8227-learning-to-share-and-hide-intentions-using-information-regularization.pdf.

[38] Sander G. van Dijk and Daniel Polani. *Grounding Subgoals in Information Transitions*. 2011.

[39] Anirudh Goyal et al. “InfoBot: Transfer and Exploration via the Information Bottleneck”. In: *arXiv e-prints*, arXiv:1901.10902 (2019), arXiv:1901.10902. arXiv: 1901.10902 [stat.ML].

[40] P.E.Hart, N.J.Nilsson, and B.Raphael. *A formal basis for the heuristic determination of minimum cost paths*. 1968.

[41] Anonymous. *A Guide To Moving From Internal Game Engine Technology*. Tech. rep. 2017. URL: http://unity3d.com.

[42] *ML-Agents*. URL: https://github.com/Unity-Technologies/ml-agents.

[43] Arthur Juliani et al. “Unity: A General Platform for Intelligent Agents”. In: *CoRR* abs/1809.02627 (2018). arXiv: 1809.02627. URL: http://arxiv.org/abs/1809.02627.

Statement

I certify that I have written the current scientific work, independently and have used no other than the specified aids. The parts of the work that are taken from the text or the meaning of the other works have been made clear by borrowing the source. The same applies to accompanying sketches and illustrations. This work has not yet been submitted to any examination authority in the same or similar form.

Gaurav Kumar

Bielefeld, March 2, 2020