DEEP REINFORCEMENT LEARNING FOR HUMANOID ROBOTS

A PROJECT REPORT

Submitted by

ABHISHEK WARRIER [Reg No: RA1511003010532]
ARPIT KAPOOR [Reg No: RA1511003010744]

Under the guidance of

Dr T. Sujithra, Ph.D
(Assistant Professor, Department of Computer Science and Engineering)

in partial fulfillment for the award of the degree

of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

of

FACULTY OF ENGINEERING AND TECHNOLOGY

S.R.M. Nagar, Kattankulathur, Kancheepuram District

MAY 2019

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

(Under Section 3 of UGC Act, 1956)

BONAFIDE CERTIFICATE

Certified that this project report titled "**DEEP REINFORCEMENT LEARNING FOR HUMANOID ROBOTS**" is the bonafide work of " **ABHISHEK WARRIER [Reg No: RA1511003010532], ARPIT KAPOOR [Reg No: RA1511003010744]** ", who carried out the project work under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form any other project report or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

SIGNATURE

Dr T. Sujithra, Ph.D
SUPERVISOR
Assistant Professor
Dept. of Computer Science and Engineering

SIGNATURE

Dr. B. Amutha
HEAD OF THE DEPARTMENT
Dept. of Computer Science and Engineering

Signature of the Internal Examiner

Signature of the External Examiner

ABSTRACT

The control of humanoid robots has always been difficult as humanoids are multi-body systems with many degrees of freedom. With the advent of deep reinforcement learning techniques, such complex continuous control tasks can now be learned directly without the need for explicit hand tuning of controllers. But most of these approaches only focus on achieving a stable walking gait as teaching a higher order task to a humanoid is extremely hard. But there have been recent advances in Hierarchical Reinforcement learning, in which a complex task is broken down into a hierarchy of sub-tasks and then learned. In this dissertation, we explore how a hierarchical learning inspired approach can be used to teach a higher order complex task, such as solving a maze, to a humanoid robot.

ACKNOWLEDGEMENT

We would like to express our deepest gratitude to our guide, Dr. T. Sujithra for her valuable guidance, consistent encouragement, personal caring, timely help and providing us with an excellent atmosphere for doing research. All through the work, in spite of her busy schedule, she has extended cheerful and cordial support to us for completing this research work.

Abhishek Warrier **Arpit Kapoor**

Contents

List of Tables

List of Figures

ABBREVIATIONS

RL	Reinforcement Learning
DDPG	Deep Deterministic Policy Gradient
HER	Hindsight Experience Replay
HRL	Hierarchical Reinforcement Learning
PPO	Proximal Policy Optimization
MuJoCo	Multi-Joint dynamics with Contact

Chapter 1

INTRODUCTION

1.1 The Reinforcement Learning Paradigm

Machine Learning refers to the set of techniques and algorithms that is used to teach a computer to perform a particular task, without any explicit programming. Reinforcement learning is one of the paradigms of machine learning, alongside supervised learning and unsupervised learning. It is a goal-based approach to machine learning that was was originally considered as an extension of optimal control theory. In RL, two main entities are defined - 1) The Agent and 2) The Environment. The agent is said to be in a given state at any point of time. From each state, it has the ability to choose a single action from a set of actions. Based on the action taken, the environment updates itself and returns the new state of the agent along with a reward signal. This reward signal is a scalar reward that denotes the quality of the action taken. The objective of the agent is to learn a plan/policy to maximize the overall reward. So it is different from the other machine learning paradigms as although there is some form of feedback (unlike unsupervised learning), it doesn't explicitly tell whether the action taken was right or wrong (unlike supervised learning).

But in recent years, there have been many breakthroughs in novel architectures and computational power which has led to the advent of Deep Learning. The combination of deep neural nets with traditional reinforcement learning techniques is known as Deep Reinforcement Learning.

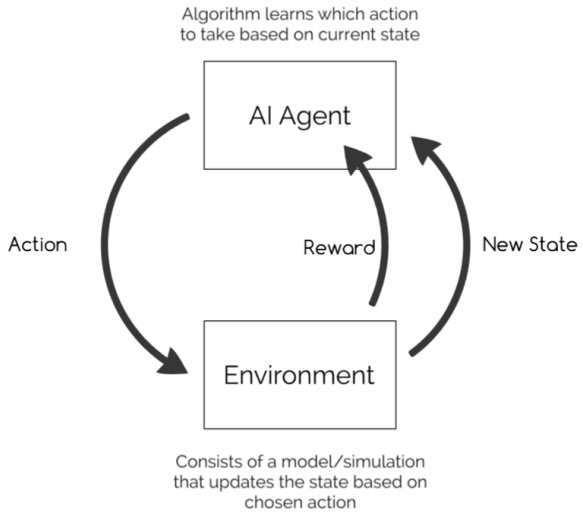

Figure 1.1: The Reinforcement Learning Paradigm

1.2 Types of State/Action Spaces

In the RL paradigm, at any given instance of time, the agent is in one of many states and from each state it can take one of many actions. The set of all possible state and action pairs is known as the State/Action Space. It can be **discrete** or **continuous**.

Discrete state/action spaces are finite in nature. An example would be a 2D Gridworld where the state is characterized by the pair $[row, column]$ and from each state it can take one of four actions $[up, down, left, right]$.

Continuous state/action spaces are not finite in nature and are usually characterized by the use of float variables. Very high dimensional discrete spaces can also be considered continuous. A general example would be a robotic arm with n joints, with state $[\theta_1, \theta_2, \ldots, \theta_n]$ where θ_i refers to the corresponding joint angle and action $[\phi_1, \phi_2, \ldots, \phi_n]$ where ϕ_i refers to the corresponding joint rotation to give.

1.3 Types of Rewards

The reward function describes the feedback value the agent gets based on the action taken. It is of the form $R(s, a)$, where s denotes the current state and a denotes the action taken from that state. This reward can be a **dense** or a **sparse** function. These terms are explained below in the context of a problem of training an agent to go from Location A to Location B on a 2D Map.

Dense rewards give constant feedback and guide the the policy optimization.
Eg - The Euclidean Distance between the two points A,B
Pros - Due to constant feedback, the RL agents learn much faster
Cons - The reward function needs to be engineered quite carefully which requires expertise and extensive domain knowledge

Sparse rewards give a binary signal indicating successful task completion.
Eg - +1 Reward if at Location B, +0 at all other points
Pros - Very simple. Do not require complex reward function engineering and domain expertise
Cons - Very slow and hard to train as the agent receives negligible feedback

1.4 Hierarchical Reinforcement Learning

Recent techniques in RL have made training agents in continuous spaces with sparse rewards a viable option. But RL agents still face difficulty in learning complex tasks, especially with long time horizons(duration). One possible approach to tackle this is Hierarchical Reinforcement Learning (HRL) in which rather than learning a single policy, an agent learns an entire hierarchy of policies. More specifically a complex task is divided into a hierarchy of sub tasks and a corresponding policy is learned for each sub task. Learning each of these sub-policies is much more simpler and feasible rather than learning to solve the problem as a whole.

Chapter 2

LITERATURE SURVEY

2.1 Reinforcement Learning in continuous spaces

The fact that robots have continuous state-action spaces makes it hard for traditional reinforcement learning techniques to learn a complex controller for any robot, in general. DDPG, an algorithm proposed by Lillicrap et al. (2015) is an Actor-Critic approach based on Deterministic Policy Gradients (DPG). This paper adapts ideas from Deep Q-Learning and extends them to DPG to create an algorithm that can operate over continuous spaces. This paper is quite pivotal as it allowed state of the art techniques to be extended to continuous action-spaces which makes it quite useful for application in Robotics. Since this paper, some improved techniques have been introduced such as TRPO by Schulman et al. (2015) and PPO by Schulman et al. (2017). Duan et al. (2016), presented a comparison of various DRL algorithms that can be implemented for continuous spaces.

Behaviours like locomotion can be quite sensitive to the choice of reward. It is common for most reinforcement learning algorithms to carefully design the reward functions in order to encourage a particular solution. Heess et al. (2017), proposed a method that enabled learning of complex locomotion behaviors with simple reward functions. In this paper, simple reward function was used to train the agents in rich, diverse environments which led to robust behaviours that generalize across diverse tasks. Also, a distributed implementation of PPO was introduced to achieve better performance. The Distributed PPO (DPPO) algorithm makes, a few augmentations which include normalization of inputs and rewards as well as an additional term in the loss that penalizes large violations of the trust region constraint.

Despite these great advances in Deep RL, one main drawback is that these techniques still require reward engineering and extensive domain knowledge. Recently, Andrychowicz et al. (2017) presented the Hindsight Experience Replay (HER), which

enables sample efficient learning from binary/sparse rewards. HER can be used along with existing state of the art off-policy RL algorithms, which not only speeds up the process of learning, but also enables training possible in challenging environments with no reward engineering. Many RL algorithms make use of a replay buffer to store experiences which are later played back to update the network. HER adds additional goals for replay to ensure that at any give point of time at least half of the replayed trajectories have a positive reward which speeds up learning.

2.2 Learning walking behaviours in humanoid robots using Deep RL

Stable walk is still one of the most challenging tasks for a humanoid robot to perform. In recent years, many researcher have explored the paradigm of deep reinforcement learning to demonstrate bipedal walk. Kumar et al. (2018), present an architecture to demonstrate walk on a 5-link bipedal robot using Gazebo simulator. They make use of Deep Deterministic Policy Gradient (DDPG) to train the bipedal walk. Their results show that the robot successfully demonstrated walking behaviour by learning through several of its trial and errors, without any prior knowledge of itself or the world dynamics. The paper also shows that with proper reward, the robot achieves a faster walking and it was even able to render a running gait with a speed of 0.83 m/s which was quite similar to that of a human.

Moreover, Gong et al. (2018) demonstrated deep reinforcement learning using and formulating a feedback controller for walking on a realistic model of Cassie, a 7 DOF bipedal robot developed by Agility Robotics. They make use of PPO to optimize the policy for Markov Decision Process. Cassie taking two steps is taken as the reference motion that defines how the robot is supposed to move with time. Time-scaling of the original reference motion is used to learn controllers for different waking speeds. It should also be mentioned that by interpolating between individual policies and that robustness of the controller can be improved.

Much attention has been given to Passive Dynamic Walking (PDW), due to it's en-

ergy efficient gaits. The challenge still remains in controlling a passivity-based bipedal robots which are inspired by Passive Dynamic walkers. However, deep reinforcement learning algorithms may help in learning walking in such scenarios as shown by (Wu et al.). The paper presents a Deep Q Network based reinforcement learning controller that learns the walking policy for a passivity-based biped robot by taking the PDW as the reference trajectory.

2.3 Transfer of policies from simulation to hardware

Training policies on the physical systems can be expensive and may even be dangerous in some cases. Which is why is most cases simulation environments that demonstrate real world like physics are preferred. But learning these policies in simulations does not ensure that these policies can directly be applied on actual robot hardware. A few papers in past have explored the techniques to transfer such policies between the simulator and the hardware.

Tobin et al. (2017) presented a technique called Domain Randomization. The main idea behind this technique is that: substantial variability in the simulation can facilitate simulation trained models to be generalized to realworld without any additional training. This is implemented in practice by randomizing various aspects of the simulation for each iteration. An extension of this approach is presented by Peng et al. (2017) which further bridge the reality gap. They introduce the methodology of randomly varying the dynamics of the simulator while the model is trained. Policies generated by such approach can adapt to dynamics that significantly differ from the dynamics used to train the policy. For each episode of training, the dynamics parameters of the environment are sampled at the beginning and are held fixed for that episode. Mass of each link, damping of each joint etc. are a few of a total of 9 such parameters. An extension of DDPG, called RDPG is then used to train along with HER.

Pinto et al Pinto et al. (2017) introduced Asymmetric Actor Critic that exploits the full state observability in the simulator to train better policies which only need partial observations as input. Their method builds upon actor-critic algorithms. For the critic, the exact position for an object is given whereas for the actor only an RGB+Depth

(RGBD) Image is given. This is further combined with Domain Randomization and HER.

One of the more recent papers to present transfer of policies is Li et al. (2018). The approach presented in this paper significantly speeds up the rate of learning in simulation. The technique learns a structured controller consisting of a neural network based policy. The approach, in this case, is to learn the neural network, while keeping the rest of the controller fixed, which can be tuned by the expert as required. The paper uses actor-critic framework to solve the problem of locomotion control with the main RL algorithm being PPO. The results are presented on an ATRIAS robot and effect of action spaces and cost functions on the rate of transfer between simulation and hardware is also explored.

Table 2.1: A few Deep RL Techniques for humanoid robots.

Author	Objective	Methodology	Results
Tedrake et al. (2004)	To implement policy gradient reinforcement learning on a simplified biped.	In this, paper instead of a full sized humanoid, a simplified humanoid is considered with 6DOF and only 4 actuators, modelled after a passive dynamic walker. This makes the robot mechanically stable, making it easier to learn. The main algorithm used is a stochastic policy gradient algorithm with TD error.	Although it performs well and converges to a stable gait in about 20 minutes, the technique doesn't scale well to robots with higher DOFs.
Morimoto et al. (2004); Morimoto and Atkeson (2007)	To implement model based RL on a Biped	A Poincare map based approach is proposed to learn the correct timing and foot placement for a walk cycle. The problem is considered a type of Semi-Markov Decision Process (SMDP) and Receptive Field Weighted Regression (RFWR) is used to approximate the model, policy and the value function. In the follow up work, the same technique is used to learn to imitate an observed walking cycle.	It is initially tested on a 5 link bipedal robot in simulation and later successfully applied to a real robot.
Endo et al. (2008)	To learn CPG based bipedal walking	A central pattern generator based feedback controller is proposed. Such a feedback controller usually requires a lot of hand tuning, but here the parameters are learned using policy gradients to obtain a stable gait.	A steady walking gait was obtained in about 1000 cycles in simulations and later implemented on an actual robot.
Kumar et al. (2018)	To device an architecture to simulate Planar Walking Bipedal Robot (PWB) in gazebo.	The paper demonstrated a planar walking bipedal robot achieve a successful walking behavior by learning through several of its trial and errors, without any prior knowledge of itself or the world dynamics. Since DDPG enables learning control in continuous action spaces, the paper uses DDPG to train the planar bipedal walk.	The effectiveness of reinforcement learning in learning complex behaviors is successfully demonstrated in by simulating a planar bipedal walker in real-world physics engine Gazebo.
Gong et al. (2018)	To demonstrate effectiveness of Deep Reinforcement Learning in Bipedal robots.	Deep reinforcement learning is used to formulate a feedback controller for walking on bipedal robots (a realistic model of Cassie robot). PPO algorithm is used to optimize the policies for the Markov Decision Process (MDP)s thus, formulating a feedback control problem.	This approach trains robust controllers for walking in bipedal robots, simulated in rich 3D environment. The controllers imitate a reference motion by employing PPO. It is also able to learn Controllers for different walking speeds.
Wu et al. (2019)	To learn walking policies for passivity-based walking bipedal robot using deep reinforcement learning.	Passive Dynamic Walking (PDW) is taken as reference trajectory to train deep Q network based controller. Experience replay and Q targets is used to ease the training of the neural network.	The deep Q network based controller is able to generate a stable planar bipedal robot walk on varying levels of slope and through disturbances.

2.4 Inference - Existing Systems and their Limitations

The control of humanoid robots using traditional control theory is extremely complex, which makes Reinforcement Learning (RL) a viable approach. Although RL techniques have been applied to humanoids before, as shown by Hester et al. (2010) and Stulp et al. (2010) and others, a lot of domain knowledge is still required which limit their range of applications.

Ideally, robotic control problems should be modelled as problems with **continuous** state/action spaces and **sparse** rewards, which are the hardest problems in RL. The **Deep Deterministic Policy Gradient (DDPG)** algorithm, proposed by Lillicrap et al. (2015), enabled Deep RL to be applied to problems with continuous spaces. Furthermore, Andrychowicz et al. (2017) introduced **Hindsight Experience Replay (HER)**, which is an elegant technique that made learning from sparse rewards viable.

The combination of both these techniques, referred to as **DDPG+HER** has shown much better performance on robotic control tasks, as shown by Peng et al. (2017). But this performance comes at a cost as the training time increases exponentially when training with multiple goals. For simple tasks a Dense reward with an algorithm like Proximal Policy Optimization (PPO) might perform better.

Albeit the improved performance on basic tasks, training complex tasks that require higher level decision making is still difficult. One possible solution is the application of **Hierarchical Reinforcement Learning (HRL)** along with hindsight, which was recently explored by Levy et al. (2018), in which a complex task was broken down into sub-tasks with different time horizons. Similar approches will be explored in this dissertation.

Chapter 3

SYSTEM OVERVIEW

3.1 Proposed System

The aim of this dissertation is to explore a variation of Hierarchical RL in which two distinct policies, $PolicyA$ and $PolicyB$ will be trained independently. The main objective is to explore ways to compose $PolicyA$ and $PolicyB$ to achieve a new $PolicyC$ that encapsulates the behaviour/functionality of both these policies.

To motivate this, we will tackle the specific problem of **teaching a humanoid to solve a maze**. More precisely,

$PolicyA$ will teach a humanoid to walk in a specific direction.
$PolicyB$ will be a general maze solving agent.
Objective : Obtain a new $PolicyC$ that enables a humanoid to solve a maze by composing $PolicyA$ and $PolicyB$.

3.2 Project Modules

The project consists of various modules which involve training individual policies, creating new environments when needed and finally obtaining the combined policy. The overall architecture or work flow is given in Figure 3.1

The first module involves training the Humanoid Environment to walk straight. The objective here is to simply verify and replicate results obtained by recent research papers. But this default environment is a single-goal environment i.e. it only learns to walk in a single goal direction. Thus the next stage involves defining and creating a multi-goal environment that can take in an input direction and make the humanoid walk

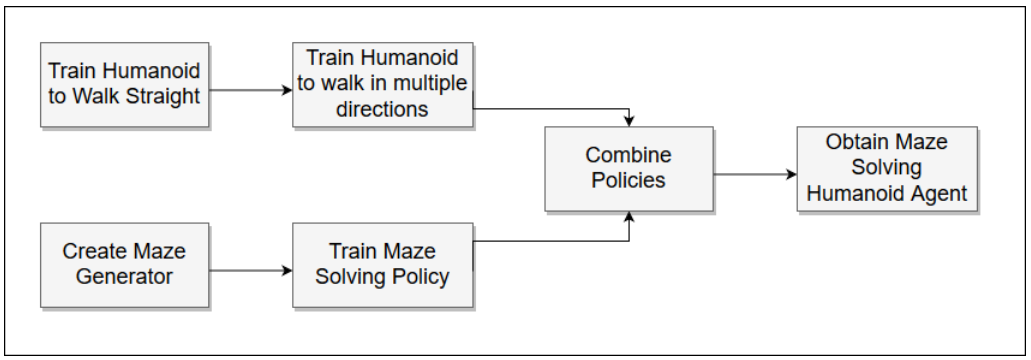

Figure 3.1: Project Modules

in that direction. This is important as the humanoid eventually needs the capability to walk in multiple directions to solve a maze.

Simultaneously, a maze-solving agent also needs to be trained. For this a random maze generator is created and an agent is trained on a simple 2D discrete maze environment. This environment is simple enough to be solved by algorithms like Q-Learning. Then a new 3D continuous maze environment is created. This is important so that it can be eventually combined with the humanoid walking policy which operates in 3D continuous spaces. After obtaining a policy trained on the 3D maze, various hierarchical techniques will be explored on how to combine the two polices to obtain the final maze solving humanoid agent.

3.3 Simulation Environment

A major bottleneck, when it comes to Reinforcement Learning is that these algorithms are quite sensitive. Even the slightest change in the implementation or simulation environment can produce drastically different results. Initially, **OpenAI Gym** along with the Multi-Joint dynamics with Contact (MuJoCo) physics engine was tested but the Humanoid environment in Gym is quite complex with a 378 dimension observation space and a 16 dimension action space which is very hard to train.

Thus we then tested with **Unity**. Unity is a 3D game engine that implements the Nvidia PhysX SDK for realtime accurate physics. Although it is traditionally used for designing games, Unity has recently released the **ML-Agents** Library which allows it to be used for various Reinforcement Learning Applications. It also comes with **Maratho-**

nEnvs which is a port of DeepMind Environments including the DeepMind Humanoid to Unity. The MarathonEnvs humanoid in comparision has only an 88 dimension observation space and 21 dimension action space which makes it easier to train. But although the humanoid manages to learn to walk straight, it is difficult to train a policy that can enable to walk in any direction.

The finally choice of simulator for the humanoid was Roboschool. Now Roboschool is a simulator, developed by OpenAI as an open source alternative to MuJoCo and it has slightly more realistic versions of the original MuJoCo environments. The roboschool humanoid environment has a 44 dimensional observation space and a 17 dimensional action space which is quite concise. But the main advantage is the inclusion of environments like **RoboschoolHumanoidFlagrun-v1** where, the objective is to not only teach a humanoid to walk, but walk to a specific target. The policies trained in this environment, although time consuming, are quite robust where the humanoid can turn and even get up after a fall.

But Roboschool Envs are quite rigid and it was difficult to setup maze in it. So the final decision was to use both Unity and Roboschool in conjunction. The maze agent is implemented in Unity whereas the humanoid is implemented in Roboschool. Both simulators communicate with each other over sockets.

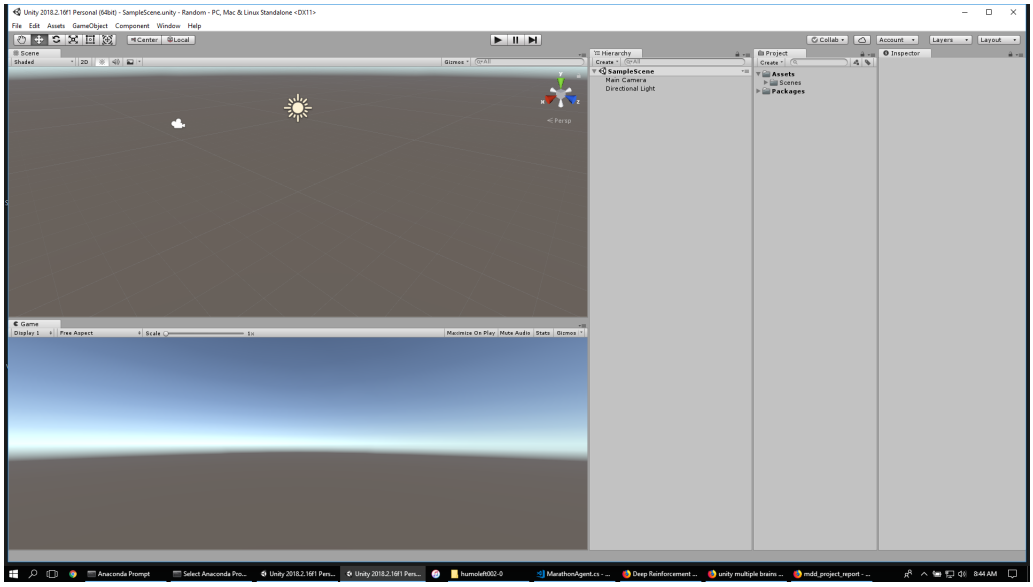

Figure 3.2: Unity Editor

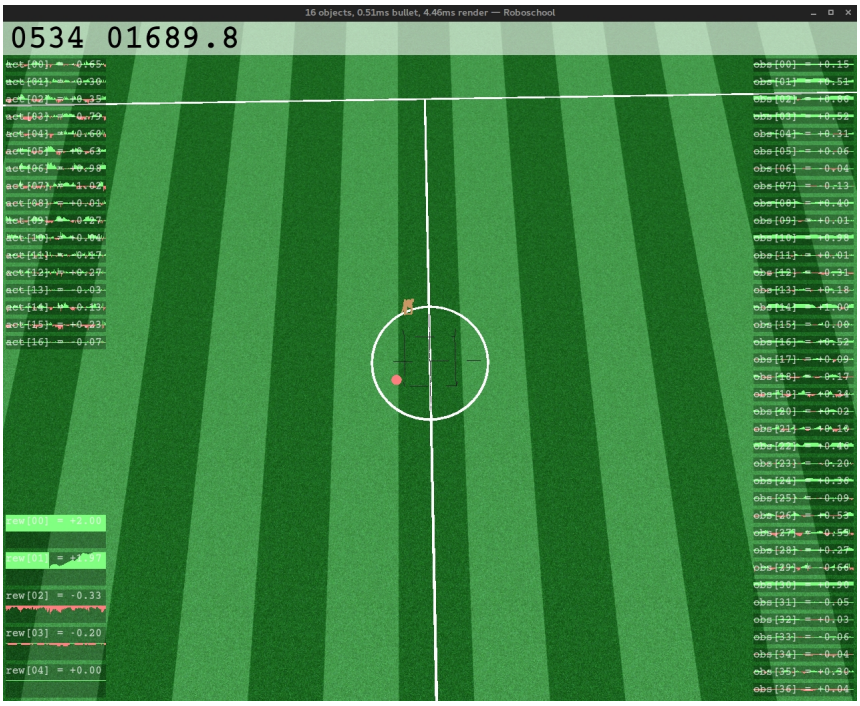

Figure 3.3: Roboschool

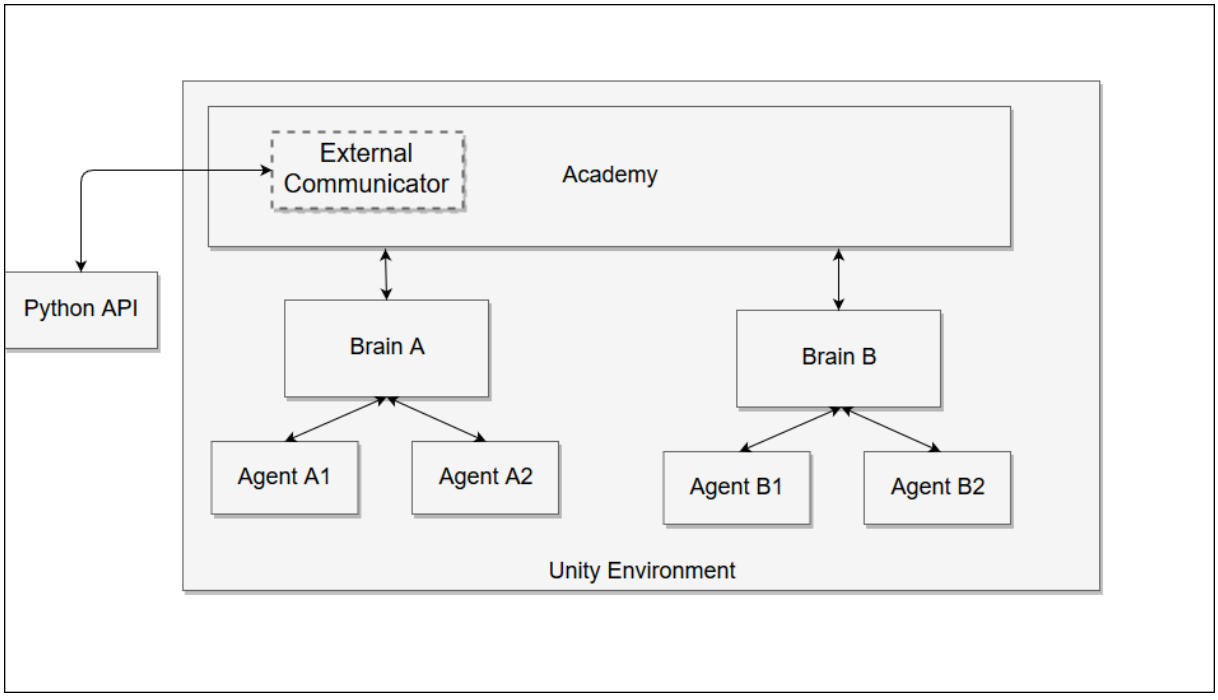

Figure 3.4: ML Agents Architecture Diagram

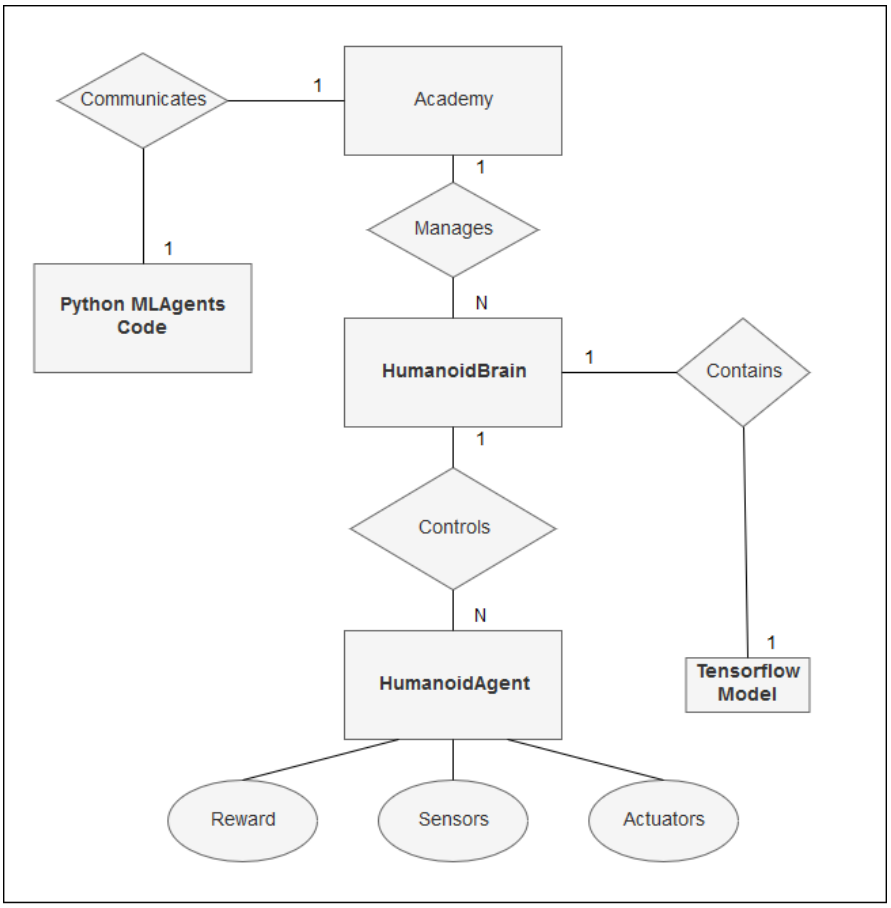

Figure 3.5: ML Agents Workflow - ER Diagram

Chapter 4

HUMANOID MOTION

4.1 Algorithm

The choice of algorithm depends on the type of environment and the reward function. For sparse rewards and multi-goal environments, the ideal algorithm would be **DDPG+HER**. But such environments are not sample efficient and take a lot of time to train. So currently for testing purposes, a dense reward, single goal environment along with the PPO algorithm would be used.

Proximal Policy Optimization (PPO)

Policy gradient methods work by computing an estimator of the policy gradient and plugging it into a stochastic gradient ascent algorithm. But such methods have convergence problems and can have destructively large policy updates. This led to the creation of Trust Region methods such as TRPO where the objective function is maximized subject to a constraint on the size of the policy update.

Instead of imposing a hard constraint, PPO formalizes the constraint as a penalty in the objective function. By not avoiding the constraint at all cost, a first-order optimizer like the Gradient Descent method can be used to optimize the objective. Even if the constraint is violated once a while, the damage is far less and the computation is much simple.

In PPO, the loss function is defined as follows -

$$L^{CLIP}(\theta) = E_t[min(r_t(\theta)A_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)] \tag{4.1}$$

4.2 Forward Motion

The Marathon Envs default reward function is defined as follows -

$$Reward = Velocity + UprightBonus + ForwardBonus - LimitPenalites \quad (4.2)$$

When trained with such a reward function for 1M steps, as expected the agent runs forward -

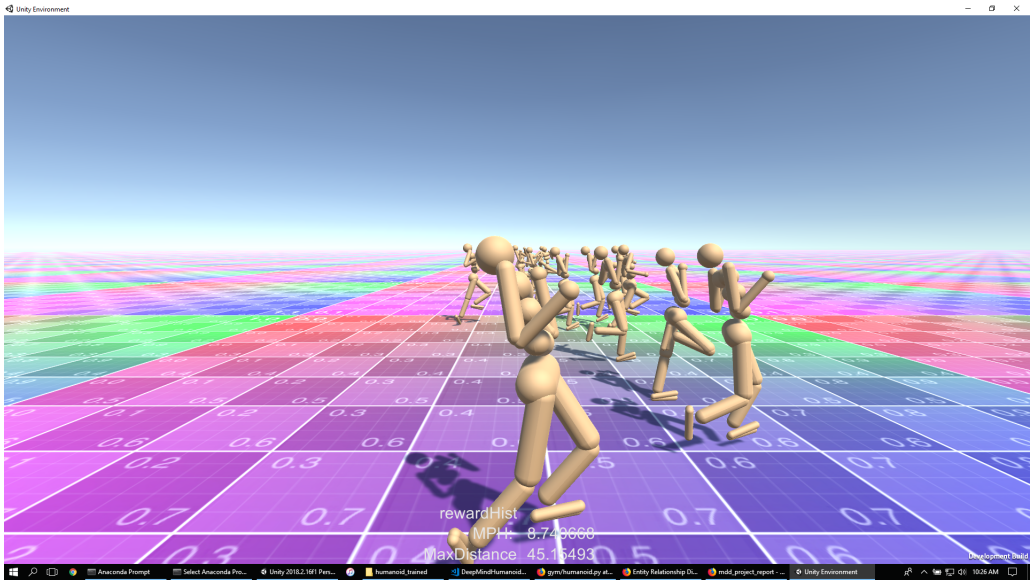

Figure 4.1: Forward Motion

4.3 Directional Motion

To make it move towards a particular direction, the reward function needs to be tweaked to elicit the required response. To move the agent to the left, the reward function was modified to -

$$Reward = Velocity + UprightBonus + \textbf{LeftBonus} - LimitPenalites \tag{4.3}$$

But instead of moving towards the left, it only makes the torso twist to the left, while still moving forward as seen in the figure below -

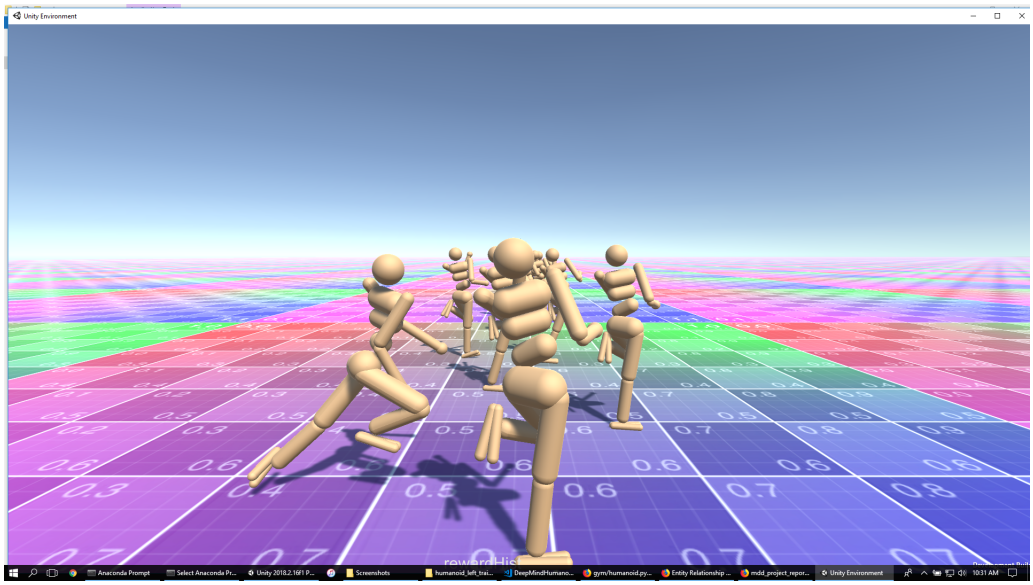

Figure 4.2: Forward Motion while Twisting Left

As shown by the results, the primary objective at this point is to enable the humanoid to be able to walk in any direction. This is important as the humanoid eventually needs the capability to walk in multiple directions to solve a maze. As in the Reinforcement Learning, the agent behaviour is driven solely by the reward function, the primary challenge is to correctly engineer a reward function that will elicit the desired behaviour.

The solution is to used Roboschool, which has more realistic physics. Roboschool has a standard environment known as RoboschoolHumanoidFlagrun that is designed to walk towards a particular flag. Although it takes a lot of time to train, the agent eventually does exhibit desired behaviour. The reward function here is

$$Reward = Alive + Electricity + Limits + FeetContact + Progress \tag{4.4}$$

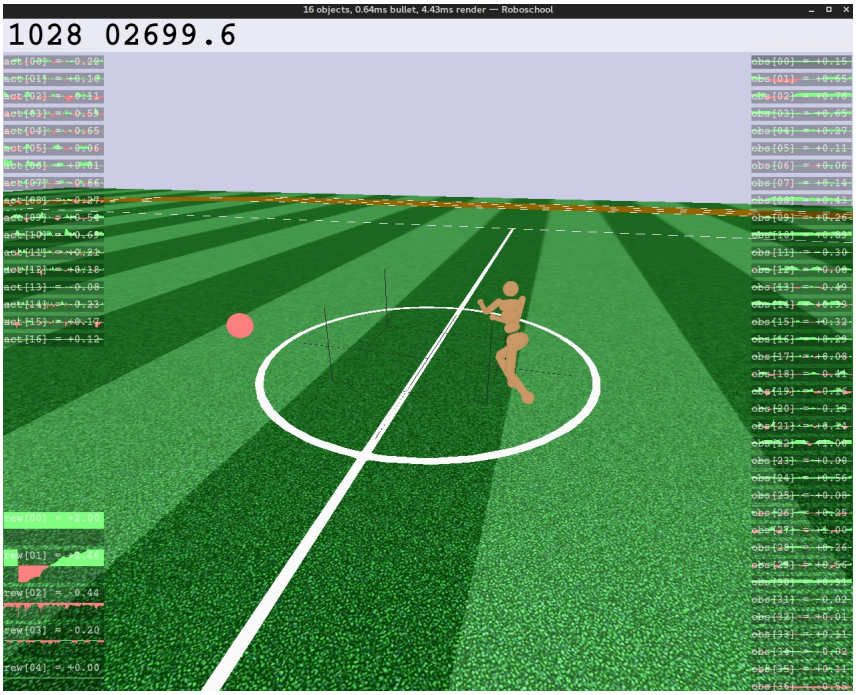

Figure 4.3: Roboschool Humanoid Flagrun

Chapter 5

MAZE SOLVING

5.1 Maze Generation

Humanoid needs the ability to navigate any randomly generated maze and thus a random maze generation module needs to be implemented in Unity. Many maze generation algorithms exist which provide varying degrees of performance and we decided to use the Growing Tree Algorithm, which is one of the more popular algorithms. But as solving random mazes is a much harder problem, for many later experiments, the mazes were fixed, as the main emphasis is on the hierarchical combination.

Algorithm 1: Growing Tree

```
Let there be a grid of dimension h, w
 Let C be an empty list
 Add one random cell to C
 while C is not empty do
 |  Choose a cell, x from C
 |   Carve a passage to any unvisited neighbor of x
 |   Add this neighbour to C
 |   if No unvisited Neighbours then
 |   |  Remove x from C
 |  end
end
```

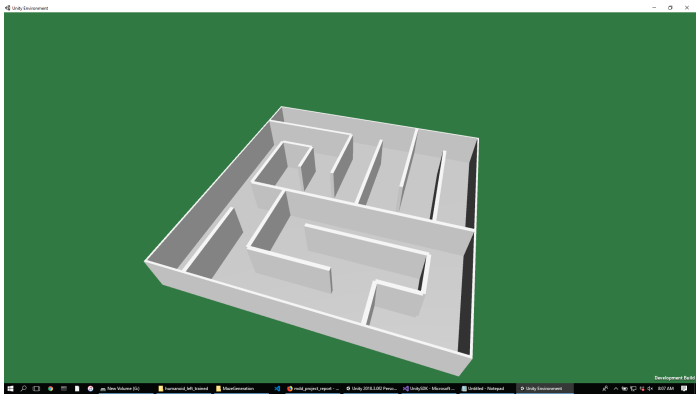

Figure 5.1: Randomly Generated Maze

5.2 Maze Navigation

5.2.1 Continuous State-Action Space

The main objective of this module is to train a continuous maze solving agent. To achieve this, we make use of Proximal Policy Optimization (PPO) Algorithm, discussed earlier, to train an agent with continuous action space. For the sake of simplicity, we take a red sphere as our agent which navigates the maze to reach target (green cube). We make use of a dense reward function which is defined as -

$$Reward = -Distance(Target, Agent) - TimePenalty$$

where the value $Distance(Target, Agent)$ is equal to the Euclidean distance between Target and the Agent. Fig 5.2 shows the agent solving a maze using the policy learnt after a couple of hours of training. But it is observed that the model training in the continuous state/action space can be a difficult task and require longer training period. Thus, we shift to Discrete State-Action space.

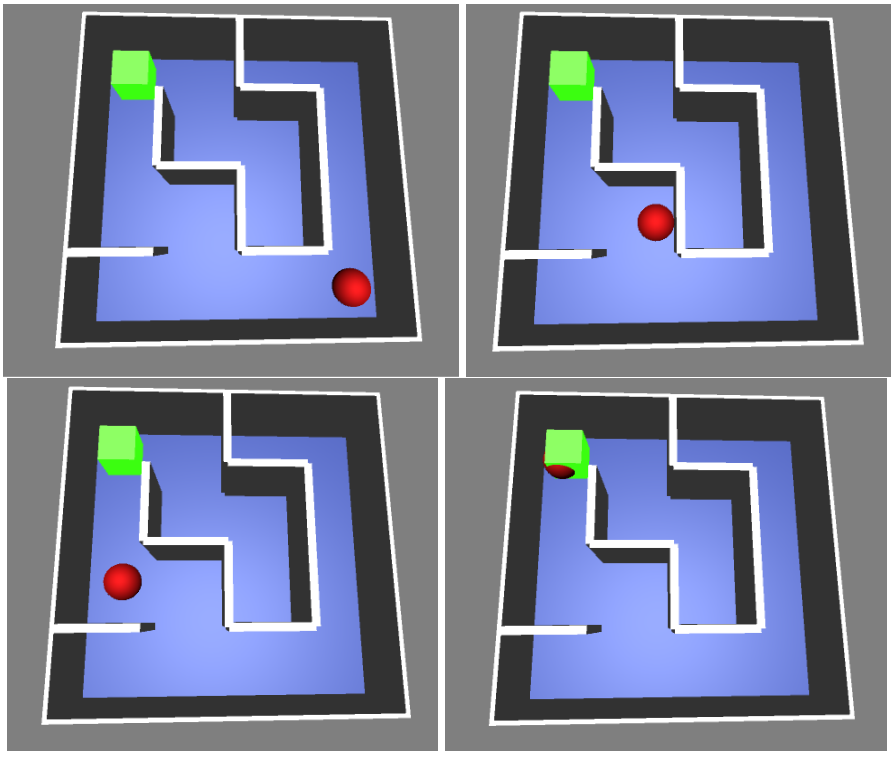

Figure 5.2: Agent solving the maze

5.2.2 Discrete State-Action Space

We create a discrete maze solver agent whose state space consists of the individual cells in the maze while it's action space consists of motion in 4 possible directions (North, South, East and West). We design the reward function for the agent as follows:
At each time step t,

if ($distanceToTarget < thresh$) then
$Reward_t = 1.0$
else
$Reward_t = -0.1/(mazeSize * mazeSize))$

where, the maze consists of $mazeSize$ x $mazeSize$ cells. This is trained using both the inbuilt Unity ML-Agents PPO and the much simpler Q-Learning Algorithm.

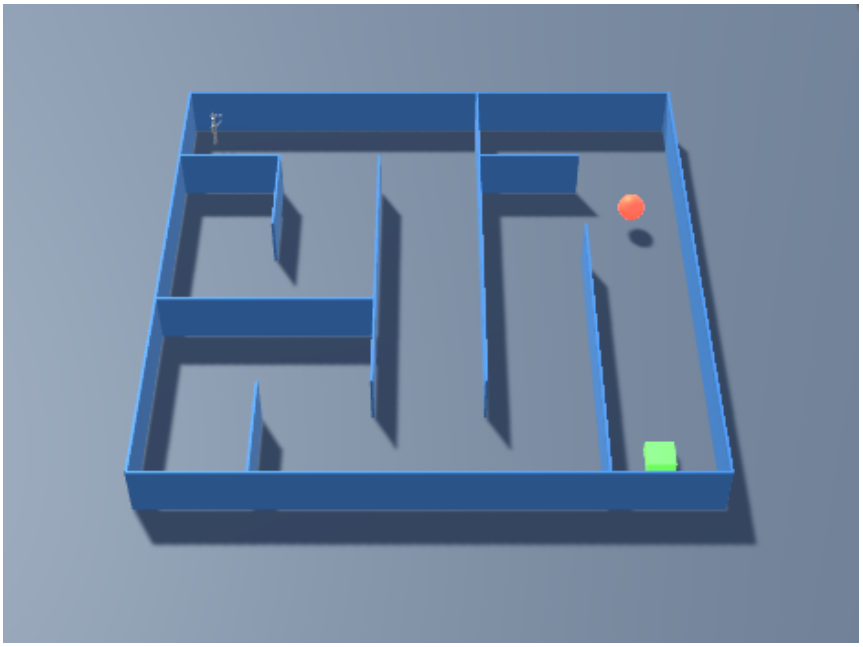

Figure 5.3: Agent solving the maze (discrete)

Chapter 6

COMBINING POLICIES

So far, the maze solver and the humanoid walker have been trained independently of each other. The final part is using the maze solver policy as a high level selector policy and using it to control the direction motion of the humanoid walker. For this, a hierarchical policy integration algorithm has been devised which is given as follows -

Algorithm 2: Hierarchical Policy Integration

Let π_h, π_l denote the higher level and lower level trained policies respectively

Let S_h be the current state of the higher level policy

Let g_h be the final goal

while S_h *is not* g_h **do**

New state $S_h^{'} = \pi_h(a_h|S_h)$

Set goal of low level policy $g_l = S_h^{'}$

Let S_l be the current state of the low level policy

while S_l *is not* g_l **do**

$S_l^{'} = \pi_l(a_l|S_l)$

$S_l = S_l^{'}$

end

end

The actual implementation of this algorithm to our problem is as follows

1. Initially, both the humanoid and the maze agent are at their starting positions.
2. The current position of the maze agent is given to the RoboschoolHumanoidFlagrun policy.
3. The humanoid starts approaching toward the maze agent and the distance between it and the maze agent is constantly checked.
4. When the distance goes below a threshold, the current state is passed to the maze agent policy that outputs a new direction to move in and updates the maze agent position.
5. Now the humanoid now starts moving to the updated position.

6. In this manner, the higher level maze policy slowly directs the lower level humanoid to solving the maze.

The final combined policy is shown in the figure below. Here, the green block is the goal for the higher level maze policy. The orange sphere is the current positon of the maze agent, which acts as the goal position for the lower level humanoid walker.

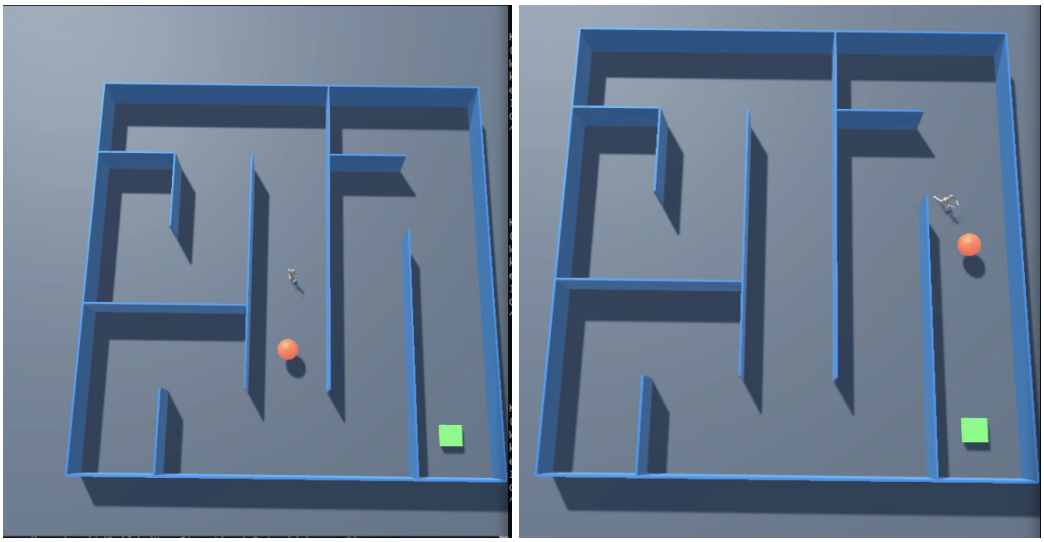

Figure 6.1: Maze and Humanoid Agent Combination

Chapter 7

RESULTS

Based on our observations, we conclude the following results - 1) We observed that the policies are effectively combined and the hierarchical behaviour is as intended.

2) Switching from continuous maze to a discrete maze environment does radically improve performance.

3) Both of these continuous/discrete versions of MLAgents-PPO is outperformed by our q-learning implementation.

4) But the overall performance and success rate varies widely as the individual policies have not been trained to perfection and it is only occasionally able to solve the maze so there is still scope for improvement.

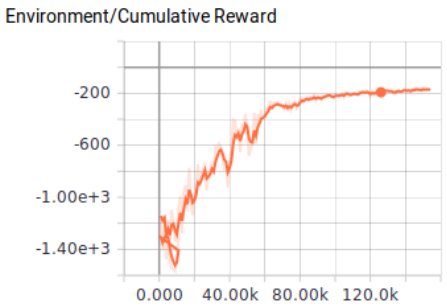

Figure 7.1: Cumulative Reward vs Number of Episodes using PPO on continuous state-action space

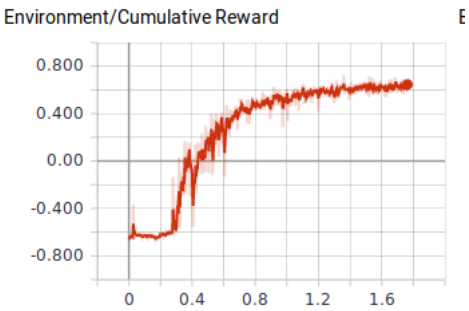

Figure 7.2: Cumulative Reward vs Number of Episodes for PPO on discrete state-action space

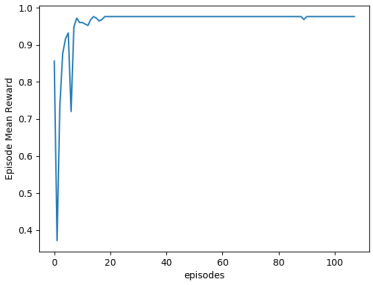

Figure 7.3: Cumulative Reward vs Number of Episodes for Q-Learning on Discrete state-action space

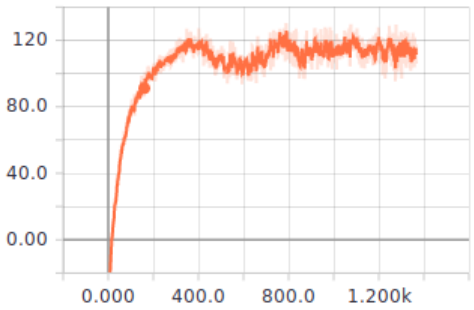

Figure 7.4: Cumulative Reward vs Number of Episodes for HumanoidFlagRun

Chapter 8

CONCLUSION

It has been demonstrated how a continuous high level task can be tackled using a hierarchical divide and conquer approach. The fact that the lower level policy is trained separately makes the overall task modular in nature. This means that the lower level policy could simply be replaced by an Ant/Hopper/Walker or any of the other continuous control agents to obtain respective maze solving agents. But yet in an ideal scenario, it would be better if efficient algorithms are developed where the entire hierarchy can be learned together at once.

Although in this paper, a humanoid was trained to solve a maze, it was not a humanoid robot. More precisely, the humanoids which are generally used in RL for benchmarking, are abstract dummy humanoid that do not correspond to any existing real humanoid robots. So the next logical task would be to create and experiment with simulations of popular humanoid robots such as Atlas, Darwin-OP, Nao etc. and study/-analyze their relative performance.

Appendix A

CODES

A.1 Proximal Policy Optimization (PPO) Algorithm Keras Implementation

```
import numpy as np
import gym
from keras.models import Model
from keras.layers import Input, Dense
from keras import backend as K
from keras.optimizers import Adam
import numba as nb
from tensorboardX import SummaryWriter
from gym_unity.envs import UnityEnv

CONTINUOUS = False

EPISODES = 100000

LOSS_CLIPPING = 0.2 # Only implemented clipping for the surrogate loss,
    paper said it was best
EPOCHS = 10
NOISE = 1.0 # Exploration noise

GAMMA = 0.99

BUFFER_SIZE = 256
BATCH_SIZE = 64
NUM_ACTIONS = 4
NUM_STATE = 8
HIDDEN_SIZE = 128
NUM_LAYERS = 2
ENTROPY_LOSS = 1e-3
LR = 1e-4 # Lower lr stabilises training greatly

DUMMY_ACTION, DUMMY_VALUE = np.zeros((1, NUM_ACTIONS)), np.zeros((1, 1))

@nb.jit
def exponential_average(old, new, b1):
   return old * b1 + (1-b1) * new

def ppo_loss(advantage, old_prediction):
```

```
    def loss(y_true, y_pred):
        prob = y_true * y_pred
        old_prob = y_true * old_prediction
        r = prob/(old_prob + 1e-10)
        return -K.mean(K.minimum(r * advantage, K.clip(r, min_value=1 -
            LOSS_CLIPPING, max_value=1 + LOSS_CLIPPING) * advantage) +
            ENTROPY_LOSS * (prob * K.log(prob + 1e-10)))
    return loss

def ppo_loss_continuous(advantage, old_prediction):
    def loss(y_true, y_pred):
        var = K.square(NOISE)
        pi = 3.1415926
        denom = K.sqrt(2 * pi * var)
        prob_num = K.exp(- K.square(y_true - y_pred) / (2 * var))
        old_prob_num = K.exp(- K.square(y_true - old_prediction) / (2 * var))
        prob = prob_num/denom
        old_prob = old_prob_num/denom
        r = prob/(old_prob + 1e-10)
        return -K.mean(K.minimum(r * advantage, K.clip(r, min_value=1 -
            LOSS_CLIPPING, max_value=1 + LOSS_CLIPPING) * advantage))
    return loss


class Agent(object):
    def __init__(self):
        self.critic = self.build_critic()
        if CONTINUOUS is False:
            self.actor = self.build_actor()
        else:
            self.actor = self.build_actor_continuous()

        self.env = UnityEnv(r'C:\Users\Bomotix\ml-agents\3DBall\3dball', 0)
        print(self.env.action_space, 'action_space',
            self.env.observation_space, 'observation_space')
        self.episode = 0
        self.observation = self.env.reset()
        self.val = False
        self.reward = []
        self.reward_over_time = []
        self.name = self.get_name()
        self.writer = SummaryWriter(self.name)
        self.gradient_steps = 0

    def get_name(self):
        name = 'AllRuns/'
        if CONTINUOUS is True:
            name += 'continous/'
        else:
            name += 'discrete/'
        name += ENV
        return name
```

```
def build_actor(self):
    state_input = Input(shape=(NUM_STATE,))
    advantage = Input(shape=(1,))
    old_prediction = Input(shape=(NUM_ACTIONS,))

    x = Dense(HIDDEN_SIZE, activation='tanh')(state_input)
    for _ in range(NUM_LAYERS - 1):
        x = Dense(HIDDEN_SIZE, activation='tanh')(x)

    out_actions = Dense(NUM_ACTIONS, activation='softmax',
        name='output')(x)

    model = Model(inputs=[state_input, advantage, old_prediction],
        outputs=[out_actions])
    model.compile(optimizer=Adam(lr=LR),
              loss=[ppo_loss(
                  advantage=advantage,
                  old_prediction=old_prediction)])
    model.summary()
    return model


def build_critic(self):
    state_input = Input(shape=(NUM_STATE,))
    x = Dense(HIDDEN_SIZE, activation='tanh')(state_input)
    for _ in range(NUM_LAYERS - 1):
        x = Dense(HIDDEN_SIZE, activation='tanh')(x)
    out_value = Dense(1)(x)
    model = Model(inputs=[state_input], outputs=[out_value])
    model.compile(optimizer=Adam(lr=LR), loss='mse')
    return model

def reset(self):
    self.episode += 1
    if self.episode % 100 == 0:
        self.val = True
    else:
        self.val = False
    self.observation = self.env.reset()
    self.reward = []

def generate_action(self):
    p = self.actor.predict([self.observation.reshape(1, NUM_STATE),
        DUMMY_VALUE, DUMMY_ACTION])
    if self.val is False:
        action = np.random.choice(NUM_ACTIONS, p=np.nan_to_num(p[0]))
    else:
        action = np.argmax(p[0])
    action_matrix = np.zeros(NUM_ACTIONS)
    action_matrix[action] = 1
    return action, action_matrix, p
```

```
def transform_reward(self):
    if self.val is True:
        self.writer.add_scalar('Val episode reward',
            np.array(self.reward).sum(), self.episode)
    else:
        self.writer.add_scalar('Episode reward',
            np.array(self.reward).sum(), self.episode)
    for j in range(len(self.reward) - 2, -1, -1):
        self.reward[j] += self.reward[j + 1] * GAMMA

def get_batch(self):
    batch = [[], [], [], []]
    tmp_batch = [[], [], []]
    while len(batch[0]) < BUFFER_SIZE:
        if CONTINUOUS is False:
            action, action_matrix, predicted_action = self.generate_action()
        else:
            action, action_matrix, predicted_action =
                self.generate_action_continuous()
        observation, reward, done, info = self.env.step(action)
        self.reward.append(reward)
        tmp_batch[0].append(self.observation)
        tmp_batch[1].append(action_matrix)
        tmp_batch[2].append(predicted_action)
        self.observation = observation
        if done:
            self.transform_reward()
            if self.val is False:
                for i in range(len(tmp_batch[0])):
                    obs, action, pred = tmp_batch[0][i], tmp_batch[1][i],
                        tmp_batch[2][i]
                    r = self.reward[i]
                    batch[0].append(obs)
                    batch[1].append(action)
                    batch[2].append(pred)
                    batch[3].append(r)
            tmp_batch = [[], [], []]
            self.reset()
    obs, action, pred, reward = np.array(batch[0]), np.array(batch[1]),
        np.array(batch[2]), np.reshape(np.array(batch[3]), (len(batch[3]),
        1))
    pred = np.reshape(pred, (pred.shape[0], pred.shape[2]))
    return obs, action, pred, reward

def run(self):
    while self.episode < EPISODES:
        print('running...')
        obs, action, pred, reward = self.get_batch()
        obs, action, pred, reward = obs[:BUFFER_SIZE],
            action[:BUFFER_SIZE], pred[:BUFFER_SIZE], reward[:BUFFER_SIZE]
```

```
            old_prediction = pred
            pred_values = self.critic.predict(obs)
            advantage = reward - pred_values
            # advantage = (advantage - advantage.mean()) / advantage.std()
            actor_loss = self.actor.fit([obs, advantage, old_prediction],
                [action], batch_size=BATCH_SIZE, shuffle=True, epochs=EPOCHS,
                verbose=True)
            critic_loss = self.critic.fit([obs], [reward],
                batch_size=BATCH_SIZE, shuffle=True, epochs=EPOCHS,
                verbose=True)
            self.writer.add_scalar('Actor loss',
                actor_loss.history['loss'][-1], self.gradient_steps)
            self.writer.add_scalar('Critic loss',
                critic_loss.history['loss'][-1], self.gradient_steps)
            self.gradient_steps += 1

if __name__ == '__main__':
    agent = Agent()
    agent.run()
```

A.2 Humanoid Agent Script (Python)

```
import os.path, gym
import numpy as np
import tensorflow as tf
import roboschool
import mjremote
import sys
import time
from squaternion import euler2quat, quat2euler, Quaternion
from zoopolicy import ZooPolicyTensorflow

config = tf.ConfigProto(
    inter_op_parallelism_threads=1,
    intra_op_parallelism_threads=1,
    device_count = { "GPU": 0 } )
sess = tf.InteractiveSession(config=config)

env = gym.make("RoboschoolHumanoidFlagrun-v1")

policy = ZooPolicyTensorflow("mymodel1", env.observation_space,
    env.action_space)

frame = 0
score = 0
restart_delay = 0
if_render = False
if_comm = True
```

```
frequency = 7
freq_count = 0

if if_comm:
   conn = mjremote.mjremote()
   conn.connect()

while True:

   if if_comm:
      conn.setinit()
   obs = env.reset()
   env_obj = env.unwrapped

   while True:
      action = policy.act(obs, env)
      obs, r, done, _ = env.step(action)

      if freq_count == frequency:
         x,y = conn.gettarget()
         env_obj.set_flag(x,y)
         freq_count = 0

      pos = env_obj.body_xyz
      quat = np.array(euler2quat(*env_obj.body_rpy))
      pad = np.append(pos,quat)
      joints = np.array([])
      for j in env_obj.jdict:
         joints = np.append(joints,env_obj.jdict[j].current_position())
      joints = joints[0::2]
      remote_data = np.concatenate([pad]+[joints])
      if if_comm:
         conn.setqpos(remote_data)

      score += r
      frame += 1
      if if_render:
         env.render("human")

      freq_count += 1
```

A.3 Mujoco XML Generation Script (Python)

```
import gym
import numpy as np
import sys

import xml.etree.cElementTree as ET
```

```
tree = ET.ElementTree(file='test.xml')

WIDTH = 0.05
HEIGHT = 0.5*2
SIZE = 5
FACTOR = 20
CLSZ = 0.1*FACTOR
LENGTH = SIZE*0.1*FACTOR

index = tree.find('worldbody')
geoms =[]
geoms.append(ET.Element('geom',type="box",rgba="0.2 0.6 1 1",pos="{} {}
    {}".format(LENGTH,0,HEIGHT) ,size="{} {}
    {}".format(LENGTH,WIDTH,HEIGHT)))
geoms.append(ET.Element('geom',type="box",rgba="0.2 0.6 1 1",pos="{} {}
    {}".format(LENGTH,-2*LENGTH,HEIGHT) ,size="{} {}
    {}".format(LENGTH,WIDTH,HEIGHT)))
geoms.append(ET.Element('geom',type="box",rgba="0.2 0.6 1 1",pos="{} {}
    {}".format(0,-LENGTH,HEIGHT) ,size="{} {}
    {}".format(WIDTH,LENGTH,HEIGHT)))
geoms.append(ET.Element('geom',type="box",rgba="0.2 0.6 1 1",pos="{} {}
    {}".format(2*LENGTH,-LENGTH,HEIGHT) ,size="{} {}
    {}".format(WIDTH,LENGTH,HEIGHT)))
geoms.append(ET.Element('geom',type="box",rgba="0.2 0.6 1 1",pos="{} {}
    {}".format(CLSZ,-2*CLSZ,HEIGHT) ,size="{} {}
    {}".format(CLSZ,WIDTH,HEIGHT)))
geoms.append(ET.Element('geom',type="box",rgba="0.2 0.6 1 1",pos="{} {}
    {}".format(2*CLSZ,-3*CLSZ,HEIGHT) ,size="{} {}
    {}".format(WIDTH,CLSZ,HEIGHT)))
geoms.append(ET.Element('geom',type="box",rgba="0.2 0.6 1 1",pos="{} {}
    {}".format(2*CLSZ,-6*CLSZ,HEIGHT) ,size="{} {}
    {}".format(2*CLSZ,WIDTH,HEIGHT)))
geoms.append(ET.Element('geom',type="box",rgba="0.2 0.6 1 1",pos="{} {}
    {}".format(2*CLSZ,-9*CLSZ,HEIGHT) ,size="{} {}
    {}".format(WIDTH,CLSZ,HEIGHT)))
geoms.append(ET.Element('geom',type="box",rgba="0.2 0.6 1 1",pos="{} {}
    {}".format(4*CLSZ,-5*CLSZ,HEIGHT) ,size="{} {}
    {}".format(WIDTH,3*CLSZ,HEIGHT)))
geoms.append(ET.Element('geom',type="box",rgba="0.2 0.6 1 1",pos="{} {}
    {}".format(6*CLSZ,-4*CLSZ,HEIGHT) ,size="{} {}
    {}".format(WIDTH,4*CLSZ,HEIGHT)))
geoms.append(ET.Element('geom',type="box",rgba="0.2 0.6 1 1",pos="{} {}
    {}".format(7*CLSZ,-2*CLSZ,HEIGHT) ,size="{} {}
    {}".format(CLSZ,WIDTH,HEIGHT)))
geoms.append(ET.Element('geom',type="box",rgba="0.2 0.6 1 1",pos="{} {}
    {}".format(8*CLSZ,-7*CLSZ,HEIGHT) ,size="{} {}
    {}".format(WIDTH,3*CLSZ,HEIGHT)))

counter = 1
for geom in geoms:
   index.append(geom)
```

```
    tree_temp = ET.ElementTree(file='empty.xml')
    index_temp = tree_temp.find('worldbody')
    index_temp.append(geom)
    tree_temp.write("temp/maze_piece"+str(counter)+".xml")
    counter+=1

tree.write('maze2.xml')
```

A.4 Mujoco Maze File (MJCF/XML)

```
<mujoco model="plane">
  <compiler angle="degree" inertiafromgeom="true" meshdir="meshes/" />
  <size nconmax="100" njmax="500" nstack="-1" />

  <option timestep="0.002">
    <flag warmstart="enable" />
  </option>

  <visual>
    <quality shadowsize="4096" />
  </visual>


  <default>
    <geom material="geom" type="box"/>
  </default>

  <asset>
    <texture builtin="gradient" height="100" rgb1="1 1 1" rgb2="0 0 0"
        type="skybox" width="100" />
    <texture builtin="checker" height="500" mark="edge" markrgb=".1 .1 .1"
        name="groundplane" rgb1=".7 .7 .75" rgb2=".9 .9 .95" type="2d"
        width="500" />

    <material name="MatPlastic" reflectance="0.5" rgba=".4 .41 .4 1"
        shininess=".6" specular=".8" />
    <material name="MatBlue" reflectance="0.1" rgba=".2 .6 1 1"
        shininess=".6" specular=".8" />
    <material name="MatFrame" reflectance="0.5" rgba=".21 .2 .2 1"
        shininess=".1" specular="1.2" />
    <material name="MatGnd" reflectance="0.1" shininess=".01" specular=".5"
        texrepeat="7.5 7.5" texture="groundplane" />
    <texture type="skybox" builtin="gradient" width="100" height="100"
        rgb1=".4 .6 .8"
         rgb2="0 0 0"/>
      <texture name="texgeom" type="cube" builtin="flat" mark="cross"
          width="127" height="1278"
```

```
        rgb1="0.8 0.6 0.4" rgb2="0.8 0.6 0.4" markrgb="1 1 1"
            random="0.01"/>
    <texture name="texplane" type="2d" builtin="checker" rgb1=".2 .3 .4"
        rgb2=".1 0.15 0.2"
        width="100" height="100"/>

    <material name='MatPlane' reflectance='0.5' texture="texplane"
        texrepeat="1 1" texuniform="true"/>
    <material name='geom' texture="texgeom" texuniform="true"/>
</asset>


 <worldbody>
   <include file="humo.xml"/>
   <geom material="MatGnd" name="ground" pos="0 0 0" size="30 30 30"
       type="plane" />
 <geom pos="10.0 0 1.0" rgba="0.2 0.6 1 1" size="10.0 0.05 1.0" type="box"
     material="MatBlue"/><geom pos="10.0 -20.0 1.0" rgba="0.2 0.6 1 1"
     size="10.0 0.05 1.0" type="box" material="MatBlue"/><geom euler="0 0
     90" pos="0 -10.0 1.0" rgba="0.2 0.6 1 1" size="10.0 0.05 1.0"
     type="box" material="MatBlue"/><geom euler="0 0 90" pos="20.0 -10.0
     1.0" rgba="0.2 0.6 1 1" size="10.0 0.05 1.0" type="box"
     material="MatBlue"/><geom pos="2.0 -4.0 1.0" rgba="0.2 0.6 1 1"
     size="2.0 0.05 1.0" type="box" material="MatBlue"/><geom euler="0 0
     90" pos="4.0 -6.0 1.0" rgba="0.2 0.6 1 1" size="2.0 0.05 1.0"
     type="box" material="MatBlue"/><geom pos="4.0 -12.0 1.0" rgba="0.2
     0.6 1 1" size="4.0 0.05 1.0" type="box" material="MatBlue"/><geom
     euler="0 0 90" pos="4.0 -18.0 1.0" rgba="0.2 0.6 1 1" size="2.0 0.05
     1.0" type="box" material="MatBlue"/><geom euler="0 0 90" pos="8.0
     -10.0 1.0" rgba="0.2 0.6 1 1" size="6.0 0.05 1.0" type="box"
     material="MatBlue"/><geom euler="0 0 90" pos="12.0 -8.0 1.0"
     rgba="0.2 0.6 1 1" size="8.0 0.05 1.0" type="box"
     material="MatBlue"/><geom pos="14.0 -4.0 1.0" rgba="0.2 0.6 1 1"
     size="2.0 0.05 1.0" type="box" material="MatBlue"/><geom euler="0 0
     90" pos="16.0 -14.0 1.0" rgba="0.2 0.6 1 1" size="6.0 0.05 1.0"
     type="box" material="MatBlue"/>
</worldbody>

   <tendon>
    <fixed name="left_hipknee">
       <joint coef="-1" joint="left_hip_y"/>
       <joint coef="1" joint="left_knee"/>
    </fixed>
    <fixed name="right_hipknee">
       <joint coef="-1" joint="right_hip_y"/>
       <joint coef="1" joint="right_knee"/>
    </fixed>
 </tendon>
 <actuator><!-- this section is not supported, same constants in code -->
    <motor gear="100" joint="abdomen_z" name="abdomen_z"/>
    <motor gear="100" joint="abdomen_y" name="abdomen_y"/>
    <motor gear="100" joint="abdomen_x" name="abdomen_x"/>
```

```
      <motor gear="100" joint="right_hip_x" name="right_hip_x"/>
      <motor gear="100" joint="right_hip_z" name="right_hip_z"/>
      <motor gear="300" joint="right_hip_y" name="right_hip_y"/>
      <motor gear="200" joint="right_knee" name="right_knee"/>
      <motor gear="100" joint="left_hip_x" name="left_hip_x"/>
      <motor gear="100" joint="left_hip_z" name="left_hip_z"/>
      <motor gear="300" joint="left_hip_y" name="left_hip_y"/>
      <motor gear="200" joint="left_knee" name="left_knee"/>
      <motor gear="25" joint="right_shoulder1" name="right_shoulder1"/>
      <motor gear="25" joint="right_shoulder2" name="right_shoulder2"/>
      <motor gear="25" joint="right_elbow" name="right_elbow"/>
      <motor gear="25" joint="left_shoulder1" name="left_shoulder1"/>
      <motor gear="25" joint="left_shoulder2" name="left_shoulder2"/>
      <motor gear="25" joint="left_elbow" name="left_elbow"/>
   </actuator>

</mujoco>
```

A.5 Maze Solving Agent (Unity C#)

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using MLAgents;
using System;

public class MazeAgentDiscrete : Agent
{
   public Transform Target;
   public GameObject mujoco;
   public IntVector2 coordinate;
   private int maze_size;
   string[,] Maze;
   public MJRemote mjr;

   public enum Direction
   {
      None,
      North,
      South,
      East,
      West
   }

   // Start is called before the first frame update
   void Start()
   {
      this.transform.position = new Vector3(-6.0f, 1.0f, 2.0f);
```

```
    coordinate = new IntVector2(1, 0);
    Target.position = new Vector3(-18.0f, 0.0f, 18.0f);
    maze_size = 5;
    //Height, Width => (z, x)
    //"NSEW"
    Maze = new string[5, 5]{{"1101","1000","1010","1101","1010"},
                            {"1011","0011","0011","1001","0010"},
                            {"0101","0110","0011","0011","0011"},
                            {"1001","1010","0011","0011","0011"},
                            {"0111","0101","0100","0110","0111"}};


    mjr = mujoco.GetComponent<MJRemote>();


}

int GetCellState(IntVector2 coordinate)
{
    int state = 0;
    int x = coordinate.x;
    int z = coordinate.z;
    string cell_state = Maze[z, x];
    state = System.Convert.ToInt32(cell_state, 2);
    return state;
}

bool IsSafe(IntVector2 coordinate, int direction)
{
    int x = coordinate.x;
    int z = coordinate.z;
    string cell_state = Maze[z, x];
    int wall_state = cell_state[direction-1];
    if ((wall_state - '0') == 1) {
        return false;
    }
    return true;
}

public override void AgentReset()
{
    this.transform.position = new Vector3(-6.0f, 1.0f, 2.0f);
    coordinate = new IntVector2(1, 0);
}

public override void CollectObservations()
{
    float[] position = {coordinate.x, coordinate.z};
    AddVectorObs((float)coordinate.x);
    AddVectorObs((float)coordinate.z);
}
```

```
void UpdateState(Direction direction)
{
   if (direction == Direction.North)
   {
      coordinate.z -= 1;
      this.transform.position += new Vector3(0.0f, 0.0f, -4.0f);
   }

   else if (direction == Direction.South)
   {
      coordinate.z += 1;
      this.transform.position += new Vector3(0.0f, 0.0f, 4.0f);
   }

   else if (direction == Direction.East)
   {
      coordinate.x += 1;
      this.transform.position += new Vector3(-4.0f, 0.0f, 0.0f);
   }

   else if (direction == Direction.West)
   {
      coordinate.x -= 1;
      this.transform.position += new Vector3(4.0f, 0.0f, 0.0f);
   }
}

public override void AgentAction(float[] vectorAction, string textAction)
{
   int action = (int)vectorAction[0];
   if (action == 0)
      return;
   Direction move = (Direction)action;
   if (IsSafe(coordinate, action))
   {
      UpdateState(move);
   }

   float distanceToTarget =
       Vector3.Distance(mjr.humanoid_pos.position,Target.position);
   // Reached target
   if (distanceToTarget < 1.42f)
   {
      SetReward(1.0f);
      Done();
   }

   else
   {
      SetReward(-0.1f / (maze_size * maze_size));
   }
```

```
    }
    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            Done();
        }

        if (mjr.move_maze)
        {
            RequestDecision();
            mjr.move_maze = false;
        }
        if (mjr.ag_reset)
        {
            AgentReset();
            mjr.ag_reset = false;
        }
    }
}
```

Bibliography

1. Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Abbeel, O. P., and Zaremba, W. (2017). "Hindsight experience replay." *Advances in Neural Information Processing Systems*, 5048–5058.

2. Duan, Y., Chen, X., Houthooft, R., Schulman, J., and Abbeel, P. (2016). "Benchmarking deep reinforcement learning for continuous control." *International Conference on Machine Learning*, 1329–1338.

3. Endo, G., Morimoto, J., Matsubara, T., Nakanishi, J., and Cheng, G. (2008). "Learning cpg-based biped locomotion with a policy gradient method: Application to a humanoid robot." *The International Journal of Robotics Research*, 27(2), 213–228.

4. Gong, Y., Hartley, R., Da, X., Hereid, A., Harib, O., Huang, J.-K., and Grizzle, J. (2018). "Feedback control of a cassie bipedal robot: Walking, standing, and riding a segway." *arXiv preprint arXiv:1809.07279*.

5. Heess, N., Sriram, S., Lemmon, J., Merel, J., Wayne, G., Tassa, Y., Erez, T., Wang, Z., Eslami, A., Riedmiller, M., et al. (2017). "Emergence of locomotion behaviours in rich environments." *arXiv preprint arXiv:1707.02286*.

6. Hester, T., Quinlan, M., and Stone, P. (2010). "Generalized model learning for reinforcement learning on a humanoid robot." *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, IEEE, 2369–2374.

7. Kumar, A., Paul, N., and Omkar, S. (2018). "Bipedal walking robot using deep deterministic policy gradient." *arXiv preprint arXiv:1807.05924*.

8. Levy, A., Platt, R., and Saenko, K. (2018). "Hierarchical reinforcement learning with hindsight." *arXiv preprint arXiv:1805.08180*.

9. Li, T., Rai, A., Geyer, H., and Atkeson, C. G. (2018). "Using deep reinforcement learning to learn high-level policies on the atrias biped." *arXiv preprint arXiv:1809.10811*.

10. Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2015). "Continuous control with deep reinforcement learning." *arXiv preprint arXiv:1509.02971*.

11. Morimoto, J. and Atkeson, C. G. (2007). "Learning biped locomotion." *IEEE Robotics & Automation Magazine*, 14(2), 41–51.

12. Morimoto, J., Cheng, G., Atkeson, C. G., and Zeglin, G. (2004). "A simple reinforcement learning algorithm for biped walking." *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, Vol. 3, IEEE, 3030–3035.

13. Peng, X. B., Andrychowicz, M., Zaremba, W., and Abbeel, P. (2017). "Sim-to-real transfer of robotic control with dynamics randomization." *arXiv preprint arXiv:1710.06537*.

14. Pinto, L., Andrychowicz, M., Welinder, P., Zaremba, W., and Abbeel, P. (2017). "Asymmetric actor critic for image-based robot learning." *arXiv preprint arXiv:1710.06542*.

15. Schulman, J., Levine, S., Abbeel, P., Jordan, M., and Moritz, P. (2015). "Trust region policy optimization." *International Conference on Machine Learning*, 1889–1897.

16. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). "Proximal policy optimization algorithms." *arXiv preprint arXiv:1707.06347*.

17. Stulp, F., Buchli, J., Theodorou, E., and Schaal, S. (2010). "Reinforcement learning of full-body humanoid motor skills." *Humanoid Robots (Humanoids), 2010 10th IEEE-RAS International Conference on*, IEEE, 405–410.

18. Tedrake, R., Zhang, T. W., and Seung, H. S. (2004). "Stochastic policy gradient reinforcement learning on a simple 3d biped." *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, Vol. 3, IEEE, 2849–2854.

19. Tobin, J., Fong, R., Ray, A., Schneider, J., Zaremba, W., and Abbeel, P. (2017). "Domain randomization for transferring deep neural networks from simulation to the real world." *Intelligent Robots and Systems (IROS), 2017 IEEE/RSJ International Conference on*, IEEE, 23–30.

20. Wu, Y., Yao, D., Xiao, X., and Guo, Z. "Intelligent controller for passivity-based biped robot using deep q network." *Journal of Intelligent & Fuzzy Systems*, (Preprint), 1–15.

21. Wu, Y., Yao, D., Xiao, X., and Guo, Z. (2019). "Intelligent controller for passivity-based biped robot using deep q network." *Journal of Intelligent & Fuzzy Systems*, (Preprint), 1–15.