Copyright

by

Animesh Goyal

2020

The Report Committee for Animesh Goyal
certifies that this is the approved version of the following report:

Multi-Agent Deep Reinforcement Learning for RoboCup Rescue Simulator

SUPERVISING COMMITTEE:

Peter Stone, Supervisor

Garrett Warnell, Co-Supervisor

Multi-Agent Deep Reinforcement Learning for RoboCup Rescue Simulator

by

Animesh Goyal

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

May 2020

Acknowledgments

First and foremost I would like to thank my advisor, Peter Stone, for his guidance and encouragement throughout this report. He always made himself available to provide help and I could always count on his lightning-fast email responses. He gave me autonomy in finding a research topic and provided the right amount of guidance to help me make progress when I felt stuck. It has been a privilege to work with him. Garrett Warnell and Tsz-Chiu Au deserves a special thanks as collaborators and mentors throughout my research. They have provided invaluable help, especially with regards to their extensive reviews of my writing. Discussions with them have helped me flesh out ideas fully, and my writing and research ability have improved through their mentorship.

I would like to thank several of my peers and colleagues for the help they've provided during my report. Aastha Goyal has provided invaluable input with her experience in Java and gRPC. She was always been very helpful and supportive. Ashutosh Shukla reviewed several of my work and provided extensive feedback. His constant motivation was very encouraging throughout my research. Thank you, Jaynish Vaghela for brainstorming with me during OpenAI gym implementation. I would also like to thank Ishan Durugkar and Siddharth for being always available to solve any doubts that I had regarding the simulator.

This work has taken place in the Learning Agents Research Group (LARG) at the Artificial Intelligence Laboratory, The University of Texas at Austin. LARG research is supported in part by grants from the National Science Foundation (CPS-1739964, IIS-

1724157, NRI-1925082), the Office of Naval Research (N00014-18-2243), Future of Life Institute (RFP2-000), Army Research Laboratory, DARPA, Lockheed Martin, General Motors, and Bosch. Peter Stone serves as the Executive Director of Sony AI America and receives financial compensation for this work. The terms of this arrangement have been reviewed and approved by the University of Texas at Austin in accordance with its policy on objectivity in research.

Animesh Goyal

The University of Texas at Austin
May 2020

Multi-Agent Deep Reinforcement Learning for RoboCup Rescue Simulator

Animesh Goyal, M.S.E.
The University of Texas at Austin, 2020

Supervisor: Peter Stone

Recent development in the field of Artificial Intelligence have dealt with building a winning strategy for video games where agents learn how to finish their task successfully using Deep Reinforcement Learning (DRL). The first major breakthrough came when Mnih et al. [22] showed how a DRL algorithm, termed Deep Q-Networks (DQN), can be applied to a collection of Atari 2600 games to surpass the performance of all previous algorithms and achieve a level that is comparable to a professional player. Their trained model received only raw pixels and game score as inputs to learn successful policies for single agents and was able to outperform professionals across a set of 49 Atari games. After a few years, focus shifted on training multiple agents using DRL, often known as multi-agent deep reinforcement learning (MADRL), for real time strategy games. Brockman et al. [4] achieved superhuman performance in the game of DOTA 2 which involves multi-agent collaboration, spatial and temporal reasoning, adversarial planning, and opponent modeling. Using Proximal Policy Optimization (PPO) algorithm and a LSTM layer as the primary component of the neural network, their trained model was able to defeat the human champion team, Team OG by 2:0. Most recently, Vinyals et al. [38] showed how a MADRL model can achieve grandmaster level in the game of StarCraft II.

In this work, we apply MADRL to RoboCup Rescue Simulator (RCRS), which is part of the annual RoboCup Competition. RCRS is an open-source virtual environment that evaluates how effective multiple agents like ambulance team, police officer and fire brigades are in rescuing civilians and extinguishing fire from a city where an earthquake just happened. RCRS is challenging, easy to use and customize multi-agent scenario. In order to create RCRS environment where deep reinforcement learning algorithms can be tested, RCRS-gym, an open-source OpenAI Gym environment was developed. In this report, we have focused on training multiple fire brigades to collaboratively accomplish their task of extinguishing fire in the city. Fire Brigades were trained using two DRL algorithms: DQN and PPO. The performance of the algorithms was then compared with a greedy approach on two different map setting, ”Small” map and ”Big” map, each having different number of fire brigades and buildings.

The agents were able to successfully finish their task of extinguishing fire on both map setting thus proving that RCRS is a suitable environment for developing deep reinforcement learning agent in a strategic multiagent game scenario. DQN outperformed PPO in the ”Small” map setting while PPO outperformed a variant to DQN, H-DQN in the ”Big” map setting. However, both the algorithms were not able to significantly outperform the greedy approach in either setting which opens up a promising avenue for future research.

Contents

List of Figures

List of Algorithms

List of Tables

Chapter 1

Introduction

Artificial Intelligence (AI) in video games is a long standing research area. It deals with studying the complex interaction between agents and environment and training them using AI techniques to achieve human-level performance. The reason behind selecting video games is because they are safe, faster than real-time environment and can provide a large amount of data for the machine learning algorithm to be trained on. A few applications of AI in video games involves Open AI's DOTA 2 and DeepMind's StarCraft II. In April 2019, OpenAI built a DRL model for the game of DOTA 2 that was able to beat 99.4 percent of players in a public match [4]. In the same year, DeepMind built a DRL model for StarCraft II that was able to perform better than 99.8 percent of all humans in a game of StarCraft II [38]. Both are real-time strategy games that involve multiple agents or players working towards achieving their respective objectives while interacting in a shared environment. These agents can communicate with each other and possibly coordinate their actions as well. Such systems commonly fall under the hood of Multi Agent Systems (MAS). As you may have noticed, both the models were trained using DRL techniques. The reason it proved so effective in achieving such performance is because it addressed several key challenges like handling high dimensional action and observation space, solving the problem of partially observed state space, and long time horizons [4]. Together MAS and DRL are referred to as Multiagent Deep Reinforcement learning (MADRL) [30].

Currently, most of the work done in MADRL has been done in the field of video games. There is still a lot of work to be done for more realistic applications with complex environments, which are not necessarily vision based. One such domain is RoboCup Rescue Simulator (RCRS) (Figure 1.1). RCRS is structured as a 2D discrete-time simulation system that depicts the situation after occurrence of earthquake in an urban area. It was built in response to the earthquake that hit the Japanese port city Kobe with a magnitude of 6.8 on richter scale [5].

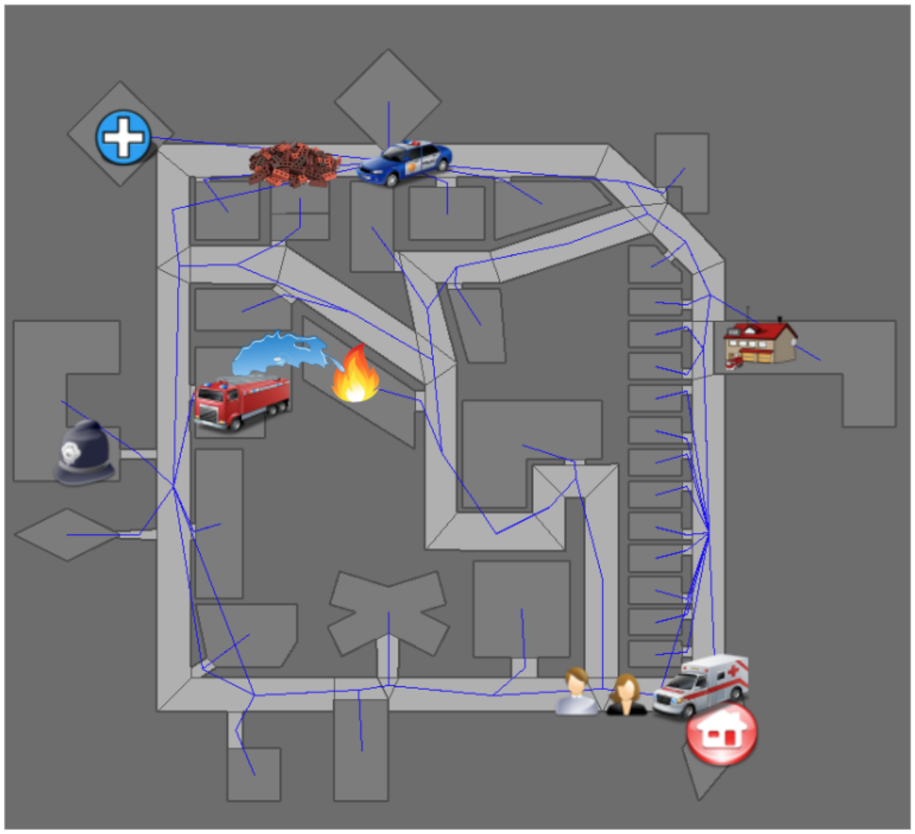

Figure 1.1: **RoboCup Rescue Simulation Environment:** RoboCup Rescue Simulator provides a 2D reinforcement learning environment where agents can be trained to save civilians and extinguish fire in a city where an earthquake has taken place

RCRS involves tasks like removing blockades, extinguishing fire and rescuing civilians that are to be performed by three different agents: fire brigades, police officers and ambulance teams. These tasks are allocated in a centralized manner. A central agent manages all the agents and has the global knowledge [23].

Contributions: In this report, using centralized task allocation approach we have applied Multi-Agent Deep Reinforcement Learning (MADRL) algorithms to RCRS in order to train multiple fire brigades to extinguish fire in the city on different sized maps having different number of agents and buildings. The major contributions are listed below:

- Built a RCRS-Gym interface, that can be utilized to apply various reinforcement learning algorithms to RCRS
- Evaluated state-of-the-art reinforcement learning algorithms on RCRS, providing an extensive set of results for comparison
- Successfully trained the agents to learn an optimal policy thus proving that RCRS is a suitable environment for applying MADRL
- Showcased promising future research directions in this environment i.e. partial observability and including police officer and ambulance as the agent

The rest of the report is structured as follows. In Chapter 2, we will introduce the background of RL, DRL and MAS. In Chapter 3, we focus on recent MADRL methods and how RL has been applied to RCRS. In Chapter 4, we describe the RCRS domain in greater detail, define what the state space, action space and reward function is, and the model architecture that was selected for various algorithms. Then in Chapter 5, we move to the implementation part, RCRS-gym environment setup. We also discuss the experiments that were performed and the results we got. Finally, in Chapter 6, we draw conclusion and discuss future research direction.

Chapter 2

Background

This chapter will briefly cover the basic but very important concepts like Reinforcement learning, Deep Reinforcement learning and Multi-agent system that will build the necessary background to understand the project properly.

2.1 Reinforcement learning

Reinforcement learning is a type of ML method where the agents learn the optimal policy by trial and error [33]. The agent interacts with the environment to learn the actions in a certain state which would produce the highest reward. The reinforcement learning model is depicted in Figure 2.1

Consider a discounted episodic Markov decision process (s, a, γ, P, r). At each time step t, the agent perceives a state s_t in state space S from which it selects an action a_t in the action space A by following a policy $\pi(a_t|s_t)$. The agent receives a reward r_t when it transitions to the state s_{t+1} according to the environment dynamics, the reward function R(s_t, a_t, s_{t+1}) and the transition probability P($s_{t+1}|s_t, a_t$). This transition probability is unknown in RL domain. The process continues until a terminal state is achieved. The objective is to maximize the expected discounted cumulative rewards

$$E_\pi[r_t] = E_\pi[\sum_{i=0}^{\infty} \gamma^i r_{t+i}] \tag{2.1}$$

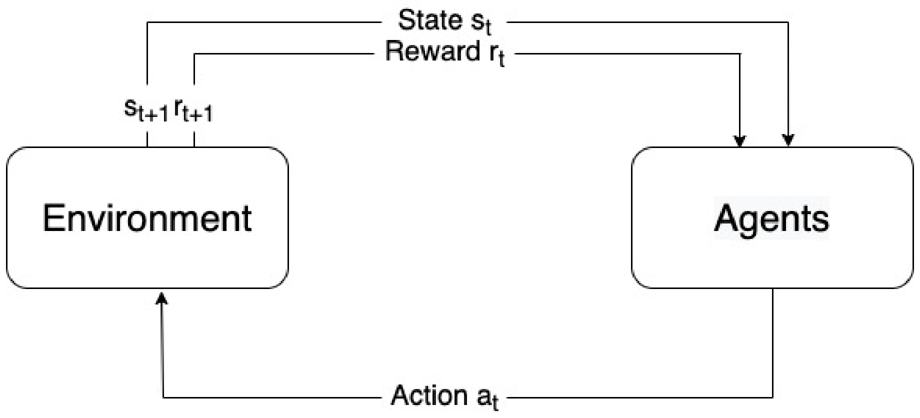

Figure 2.1: Abstract depiction of reinforcement Learning Model: At some point in time t, the RL agent experiences a state s_t and a reward r_t for his action in time $t-1$. In state s_t, the agent takes action a_t , for which the environment advances a time step, and the agent experiences a new state s_{t+1} and a reward r_{t+1} for action a_t

where discount factor $\gamma \in [0, 1]$ is applied to the future rewards.

There are two types of RL approach: model based and model free.

Model based approach uses a reduced number of interactions with the real environment during the learning phase. Its aim is to construct a model based on these interactions, and then use this model to simulate the further episodes, not in the real environment but by applying them to the constructed model and get the results returned by that model.

Model free approach act in real environment in order to learn. The most common model-free technique is Q-Learning and Policy Gradient methods. For our project, we will focus on Model free approach since agents will have to learn by acting in the real RCRS environment.

2.1.1 Q-learning

Q-learning stores a table for the Q-values Q(s,a) according to equation 2.2. Each state-action pair has a Q-value associated with it. Bellman equation is used to find out the optimal Q-value function whose unique solution is Q*(s,a):

$$Q^{\pi}(s,a) = E[\sum_{k=0}^{\infty}\gamma^k r_{t+k}|s_t = s, a_t = a, \pi] \tag{2.2}$$

$$Q^*(s,a) = (\beta Q^*)(s,a) \tag{2.3}$$

where β is the Bellman operator mapping any function K : S $\times$ A $\rightarrow$ R into another function S $\times$ A $\rightarrow$ R and is defined as follows:

$$(\beta K)(s,a) = \sum_{s' \in S} T(s,a,s')(R(s,a,s') + \gamma max_{a' \in A} K(s',a')) \tag{2.4}$$

But there are situations where the state space and action space become so large, that it's not feasible to learn all the Q-values for state-action pair. This is when Deep Reinforcement learning (DRL) is used where neural networks are utilized to model the components of RL. A representation of the architecture is given in Figure 2.2.

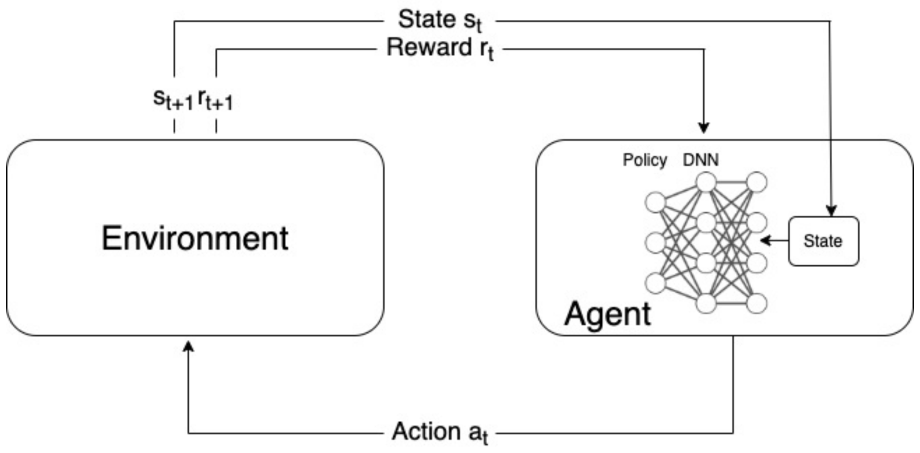

Figure 2.2: Deep Reinforcement learning Architecture

Deep Q-Networks

DQN is a combination of Q-Learning and deep neural networks. DQN addresses the instabilities caused by using non-linear approximator to represent the Q-value by using two insights: experience replay and target network. Using Convolutional Neural Network (CNN) , DQN parameterizes an approximate value function Q(s, a; θ_i) where θ_i are the weights of the network at iteration i. The experience replay stores the agent's experiences $e_t = (s_t$, a_t, r_t, $s_{t+1})$ at each time step t in a dataset $D_t = e_1$,e_t pooled over many episodes into a replay memory. Then, mini batches of experience drawn uniformly at random from the dataset (s, a, r, s) $\sim$ U(D) are applied as Q-updates during the training. The Q-learning function is updated using the following loss function:

$$L_i(\theta_i) = E_{(s,a,r,s)\sim U(D)}[(r + \gamma max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i))^2] \tag{2.5}$$

where θ_i are the Q-network parameters at iteration i and θ_i^- are the target network parameters. The target network parameters are updated with the Q network parameters every C steps and are held fixed between individual updates.

2.1.2 Policy Gradient Methods

Policy gradient methods are one of the most frequently used methods in RL. The logic behind Policy gradient is simple. The agent observes the state s of the environment and performs an action a based on a policy π. The agent then enters a new state s' and continue to take further action according to the policy. After a trajectory of motions τ, the agent adjusts its policy based on the total reward R(τ) that it received. Mathematically, let $\pi(a|s)$ be the probability of taking an action a when in state s. Our objective is to find a policy θ that create a trajectory τ [34]

$$\tau = (s_1, a_1, s_2, a_2, ..., s_h, a_h)$$

which maximizes the expected rewards $J(\theta)$.

$$\max_\theta J(\theta) = \max_\theta \sum_\tau P(\tau; \theta) R(\tau)$$

where $J(\theta)$ is:

$$J(\theta) = E[\sum_{t=0}^{H} R(s_t, u_t); \pi_\theta] = \sum_{\tau} P(\tau; \theta) R(\tau)$$

One of the drawbacks of policy gradient method is that it sometimes can get stuck in local maxima and will not be able to reach global maxima.

Proximal Policy Optimization (PPO)

PPO is a type of policy gradient method which performs comparably or better than state-of-the-art approaches while being much simpler to implement and tune. Careful tuning of the step size is required for achieving good results with policy gradient algorithms [28]. Moreover, most policy gradient methods perform one gradient update per sampled trajectory and have high sample complexity. Schulman et al. [29] introduced PPO algorithm that solves both these problems. It uses a surrogate objective which is maximized while penalizing large changes to the policy. They defined a likelihood ratio

$$l_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)}$$

PPO then optimizes the objective:

$$L^{CLIP} = \hat{E}_t[min(l_t(\theta)\hat{A}_t, clip(l_t(\theta), 1 - \epsilon, 1 + \epsilon)\hat{A}_t)]$$

where $\hat{A}_t$ is the generalized advantage estimate and $clip(l_t(\theta), 1 - \epsilon, 1 + \epsilon)$ clips $l_t(\theta)$ in the interval [1 - ϵ, 1 + ϵ]. The algorithm alternates between sampling multiple trajectories from the policy and performing several epochs of SGD on the sampled dataset to optimize this surrogate objective. Since the state value function is also simultaneously approximated, the error for the value function approximation is also added to the surrogate objective to compute the complete objective function.

2.2 Multi-Agent Systems (MAS)

A multi-agent system is a connected network of multiple agents that communicate with each other to solve problems which are beyond the reach of single agents. Several agents model each other's goals and actions [31]. There may be direct communication between the agents in a fully general multiagent scenario. In multiagent systems, environment dynamics of other agents can be known by each agent which is unlike single agent systems, other agents can affect the environment in unpredictable ways and can create uncertainty in the domain. Therefore, multiagent systems can be viewed as having dynamic environments. Application of MAS include aircraft maintenance, electronic book buying coalitions, military demining and supply chain management. Figure 2.3 shows a fully general multiagent scenario.

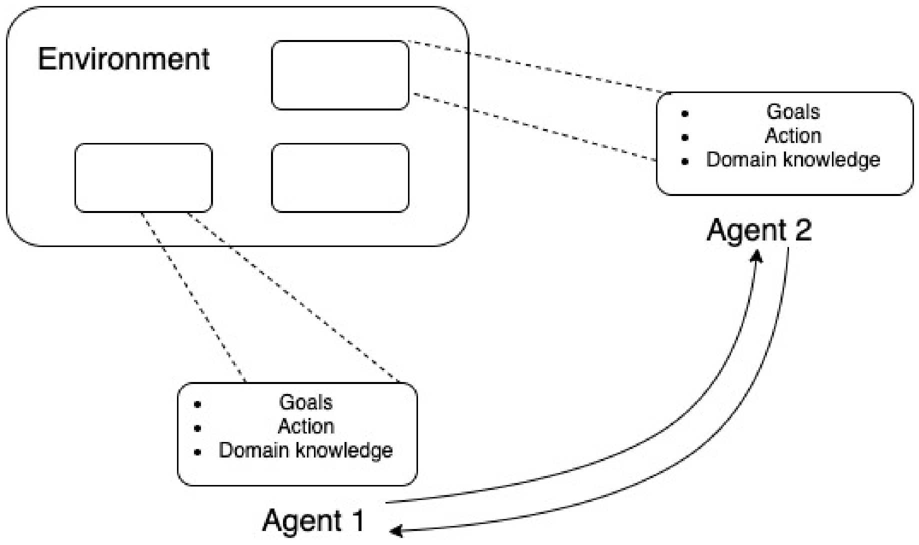

Figure 2.3: A fully general multiagent scenario. Agents can affect other agents actions, goals or domain knowledge or communicate directly with each other [31]

Multiagent Systems can be further divided into various categories mentioned below [16]:

- *Goals*: Environment setting can be cooperative, competitive or mixed. In cooperative setting, all agents usually share a common reward function. Competitive setting is usually modelled as a zero-sum Markov game, where reward of one agent is exactly the loss of the other. Mixed setting is also known as the general-sum game setting where no restriction is imposed on the goal and relationship of the agents. Each agent is self interested and can have conflicting rewards.

- *Actions*: Actions can be deterministic or stochastic. In case of deterministic actions, taking an action always results in the same action being taken whereas in case of stochastic actions, actions can change with a certain probability
- *Domain knowledge*: The environment can be partially or fully observable. In partial observability, each agent has access to only certain information about the environment.

For our case, environment is fully observable where all agents have total information about other agents and the environment as well. Its a cooperative setting where all agent have a common reward function and all the actions are deterministic in nature.

Chapter 3

Related Work

Majority of the practical applications of MADRL have taken place in cooperative setting. Tampuu et al. [35] trained the agents to play pong with competitive and collaborative reward scheme using a combination of DQN and independent Q-learning. Recently, MADRL was deployed by a team of unmanned aerial vehicles (UAVs) to accomplish a cooperative task, without the use of centralized controller. Each UAV was equipped with communication devices in order to exchange important information with other UAVs. Q-learning based method was adopted to allocate resource in UAVs while aerial defence and base defense for the fleet control was handled by policy optimization methods in a purely centralized fashion [41], [26], [36]. This is similar to our work where various fire brigades would try to accomplish a cooperative task without using any central agent. Foerster et al. [13] proposed the use of DQN for tacking the problem of communication among a team of agents without human intervention by proposing to use two Q networks that govern taking action and producing messages separately. Their algorithm was an extension of deep recurrent Q-learning (DRQN) [14] which combined RNN and DQN. Following Foerster et al., various other works involving neural network architectures to promote communication between aqents have been proposed [18], [32], [15], [11], [25].

A lot of work has been done for improving the micromanagement tasks for StarCraft II using MADRL as well using centralized controller having full access to state information. Peng et al. [25] uses an actor-crtic method that relies on RNNs to exchange information between the agents while Usunier et al. [37] implement DQN in a fully observable setting.

Omidshafiei et al. [24] assumes a decentralized training regime to address the instability of experience replay in multi-agent setting. Rashid et al. [27] proposed QMIX, which was a novel value based method that could train decentralized policies in a centralized end-to-end fashion. QMIX outperformed the then best existing value-based MADRL methods when applied to StarCraft II micromanagement tasks. Apart form this, a few more applications of MADRL include control of mobile sensor networks [9], smart grid operation [10] and robot navigation [8].

Previous applications of machine learning technique to RCRS uses either supervised learning methods or traditional RL methods. Martinez et. al [21] were the first one to implement RL to RCRS. They introduced evolutionary reinforcement learning to improve the ambulance decision making process such as deciding the number of ambulances required to rescue a buried civilian. Their trained agent was able to outperform the participants of 2004 RCRS competition and was also computationally efficient. Visser et. al [40] developed a new framework to incorporate state-of-the-art machine learning algorithms into RoboCup Rescue competition code using the MATLAB Engine API for Java. They showed how supervised machine learning methods like classification and regression can be applied to assess the strategies of the agents and found the most relevant predictors. For example, estimating the chance of survival for a trapped civilian by predicting the remaining health point at the end of the scenario. Abdolmaleki et. al [2] used SARSA to train the agent. They also proposed lesson-by-lesson learning which solved the problem of huge search space. Their proposed model increased the speed of learning and utilized very low memory. They discussed how temporal difference (TD) learning can be used to find the optimum policy for fire extinguishing tasks of a group of firefighter agents. Bitaghsir et. al [6] introduced a layered neuro-fuzzy paradigm which is inspired from incremental learning. The paradigm was used for developing intelligent firefighter robots which involved layering increasingly complex learned behaviors. Aghazadeh et. al [3] used parametric reinforcement learning to improve police force's decision making in RCRS. Using linear function approximator, they were able to perform the tasks using very less space. However with the success of current MADRL techniques to solve the problem of large state space and action space as well as task allocation for virtual environment, RCRS would be a good test bed to apply those techniques in order to achieve results better than the existing ones.

Chapter 4

Modeling DRL for RCRS

4.1 Domain Description

The competition is based on a complex simulation platform representing a city after an earthquake. In order to make the simulator as realistic as possible, its designed to have heterogeneous agents (different types of agent), limited reach (agents can only perform tasks occurring close to them), random fire spread, injured victims, agents having limited communication with messages often getting dropped, agents being only able to see a short distance. It is therefore considered as a multiagent simulator [19]. However, in this report, our focus will be on training multiple fire brigades to extinguish the fire as quickly as possible. Figure 4.1 gives a visual representation of how the simulator looks. Next, we will explain the various attributes of the simulator which will help the reader have an indepth understanding of the simulator.

RCRS has 2 major parts, Environment and Agents which are described in detail below:

4.1.1 Environment

The RCRS environment comprises of various kinds of entities. The most important are buildings, roads, blockages, refuges and humans [39].

Buildings are real constructions and have properties like area, fieryness and temperature, if its made of wood or cement. This information is useful for the simulator to calculate fire spread and building collapses. In case the building area is high and made of wood, fire will spread quickly. Simulator also changes the color of the building depending on the temperature and fieryness value. In case both the properties are high, color gets darker and eventually turning black if the building is burned out. Note that fieryness value means how strong the fire is.

Roads help the agents to move along the map. Some of the properties of roads are length, number of lanes. If a road is blocked due to agents standing in front or blockages, its not possible to move along it.

Refuge are buildings where Fire Brigades can refill their water tanks. These buildings do not catch fire or collapse.

4.1.2 Agents

The only we will be concentrating on is the fire brigade.

Fire Brigades These agents are responsible for extinguishing fires in the buildings. They have water tanks with certain water capacity that needs to refilled from the refuge. Fire Brigades also have health points that decreases in case they catch fire or enter a building on fire. In case there health points drops to zero, they are considered dead.

4.1.3 Search Strategy of Fire Brigades

Fire is initiated randomly following a Poisson distribution. Fire brigades start searching the map looking for fire spots in the environment. Once the fire trucks detect a fire, they start extinguishing it. Note that extinguishing only takes place if the fire trucks are in a close proximity to the building. Fire brigades extinguish the fire until one of the three conditions hold true: (i) It is able to extinguish the fire completely, (ii) building has been completely burned out, or (iii) water in the fire tank is empty. For the first two situations, fire brigades start searching for a new building. For the third case, fire brigades move to the refuge to refill their tanks. In case the fire brigade sees more than one building on fire, it selects the one with highest priority using the following equation:

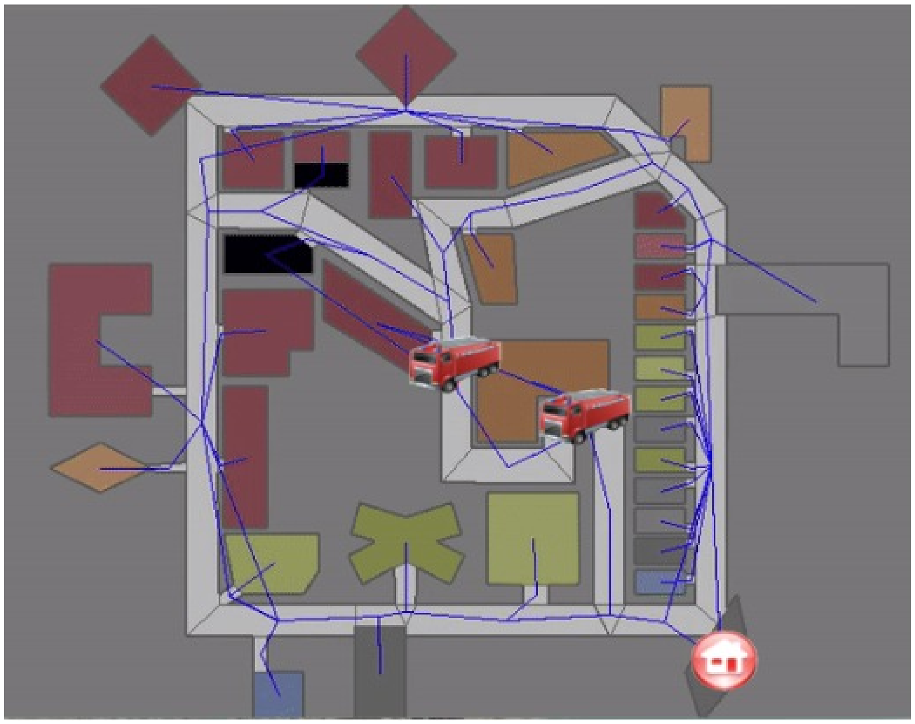

Figure 4.1: **Representation of the Simulator:** RoboCup Rescue Simulator with fire brigades. Different polygons are the buildings in the map. Grey color shows building is unburnt, yellow and red color shows building is burning, blue color depicts fire is extinguished and black shows the building is totally burnt. Note that as the shades get darker, building temperature and fieryness (degree of fire) increases. White house within red circle is the refuge

$$Priority = (\frac{Fieryness}{TotalArea}) + (\frac{1}{\sqrt{distance}}) + UnBurnedNeighbors$$

By this formula, priority is given to the building which has the highest fieryness, nearby and has the most number of neighbors.

Pseudo code for the algorithm that fire brigade follows is described in Pseudo-Algorithm 1.

Algorithm 1 Working Principle of *Fire Brigade*

```
if (agent is inside refuge) and (water capacity of the fire truck is not full) then
  Refill the tank
end if
if (agent is stuck in blockage) or (building is not reachable) then
  Search for buildings
end if
if Water tank gets empty then
  Start moving to the refuge
end if
if (agent has not target) then
  if (agent sees a fire) then
    Select target
  else if (communication is enabled) then
    Request target
  else
    Search for buildings
  end if
end if
if (Target is not visible) then
  Start moving towards the target
end if
if (Target is visible) then
  Extinguish fire
end if
```

4.1.4 Map Variants: "Small" map and "Big" map

In order to test the algorithms, two maps were chosen. A "Small" map having fewer number of buildings and fire brigades to fasten the training process and a "Big" map having greater number of buildings and fire brigades. The Map specification can be seen in Table 4.1. The representation of "Small" map can be seen in Figure 4.1 and of "big" map in Figure 4.2

Map Name	Number of buildings	Number of Agents (Fire Brigades)
Small	37	2
Big	100	4

Table 4.1: RCRS Map Specifications

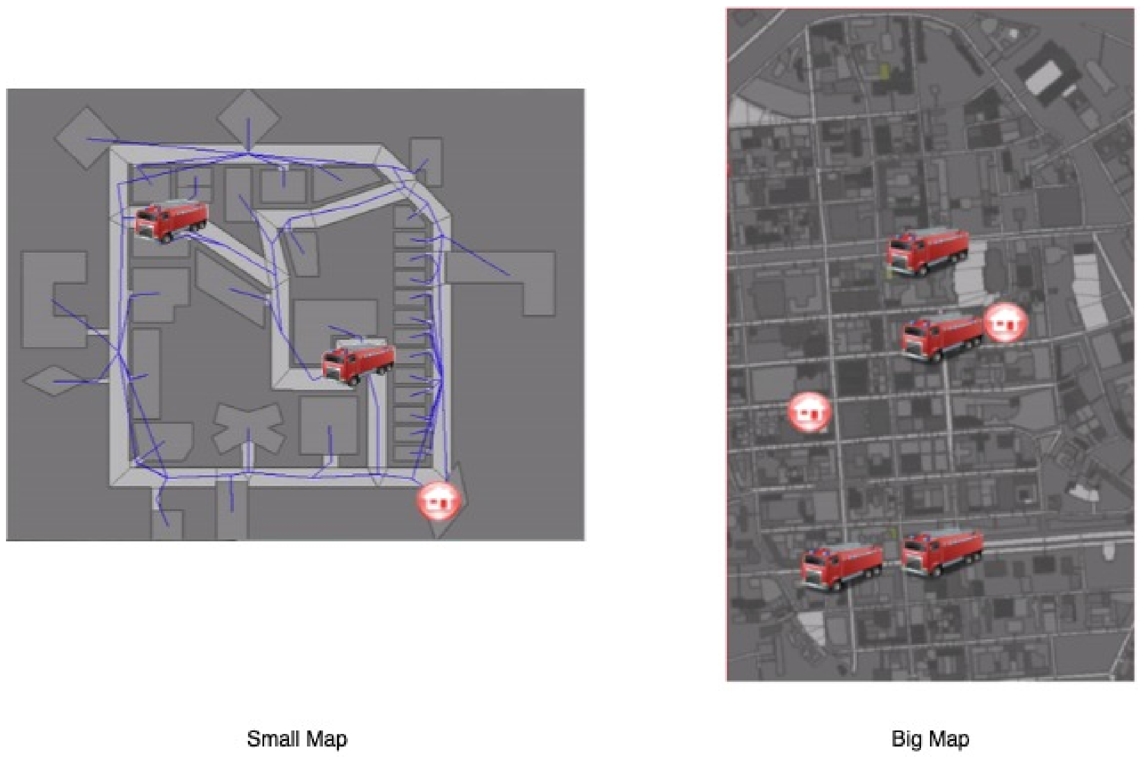

Figure 4.2: Representation of "Small" and "Big" Map. Small map contains 2 fire brigades and 1 refuge while Big map contains 4 fire brigades and 2 refuge

4.2 Defining State Space, Action Space and Reward function

In order to apply reinforcement learning techniques to RCRS, one needs to first define a specific state space, action space, and reward function. In the rest of this section, we detail the choices that were made under this project.

4.2.1 State Space

We define the state space here as follows: the first part is the building information which consists of the temperature and fieryness. Note that, fieryness is a parameter to measure the degree of fire in a building. The second part is the agent information which gives the location ((X,Y) coordinates), water in the fire tanks and the health points of the fire brigade at each timestep. The third part is the busy/idle information which is a binary variable. Fire brigades receive a building id at each timestep as their target location. But it sometimes takes more than one timestep for them to reach the building. In the meanwhile, actions are being sent continuously. Hence, fire brigades have to ignore the actions till the time they visit the building they have been told to visit in the previous timestep. This information is passed over as a state information which will be highly valuable for our algorithm to perform better. Whenever the actions that are sent by the algorithm are used in the simulator (busy), 1 is sent back as the state information otherwise 0 is sent (idle). The dimensionality of the state space can be generalized by the following formula:

State Space Dimensionality = 2*No. of buildings + 3*No. of Agents + Busy/idle

Therefore state space dimensionality for small map is 2*(37) + 3*(2) + 1 = 81 and for Big map is 2*(100) + 3*(4) + 1 = 213. Table 4.2 elaborates the range of value every state information can have.

4.2.2 Action Space

The only action available to our agent is to move to the building which is on fire and therefore the action space consists of the ID's of the buildings. Note that extinguishing fire

State	Parameter	Range
Building Information	Temperature of Building	0-100
	Fieryness of Building	0-10
Agent Information	(X, Y) Coordinates	0-10000
	Water Level	0-15000
	Health Points	0-10000
Busy/idle Information	Binary variable	0/1

Table 4.2: Ranges for state information parameters

and refilling water are default characteristics of our agent i.e. whenever our agent is near a building on fire, it will try to extinguish it and whenever it is out of water, it will move to the refuge to refill the tank. Therefore these actions are not included in the action space. The dimensionality of the action space can be generalized by the following formula:

Action space dimensionality = Number of buildings in the map

Therefore the action space dimensionality for small map is 37 and 100 for big map.

4.2.3 Reward Function

Since the ultimate goal of the fire brigades is to extinguish fire as quickly as possible, we created a reward function that awards the agents higher rewards for keeping the fire to a minimum and penalize them if the fire increases. Fieryness is one parameter that measures the degree of burn in the building and hence keeping the overall fieryness value to a minimum results in a higher cumulative reward. Table 4.3 shows the reward function used in our case study. Table 4.4 shows the severity of the fieryness value.

Fieryness Value	Reward Value
0-2	+10
3-5	-5
6-10	-10

Table 4.3: Reward Calculation

Fieryness Value	Severity
0-2	Slightly burned
3-5	Moderately burned
6-8	Critically burned
9-10	Totally burned

Table 4.4: Fieryness severity according to fieryness value

4.3 Model Architecture

We use DQN and PPO architecture with a few modifications. We do not have the set of convolutional layers since the input to the neural networks is not an image. The input to the network is state representation and there is a separate output unit for each possible action. Different architectures were tried by changing the number of hidden layers, using LSTM, using CNN or combination of CNN and LSTM. As an example of one of the architecture we tried, consider Figure 4.3. Input the neural network is not only the state information of the current agent but also other agents. The state information is fed to the first hidden layer having 64 units followed by another fully connected hidden layer with 64 units. The output to the second hidden layer is passed through a softmax layer. Dimension of the softmax layer is equal to the number of actions possible i.e. number of buildings present in the map. Figure 4.3 is the model architecture for Agent 1. For Agent 2, the architecture would be the same. Input to the neural network would be (X, Y) coordinates, water level and health points of agent 2 along with (X, Y) coordinates, water level and health points of the other agents. This will also include the building information and if agent 2 is in busy/idle condition.

4.4 Hyperparameter Search

For DQN and PPO, different parameter were tweaked like batch size, learning rate, discount factor. For each parameter setting, 5 repetitions were done and the average value of the reward was calculated. Values of the parameter that were tried can be found in Table 4.5. 400 episodes were run for "Small" map and 10000 episodes were run for "Big" map.

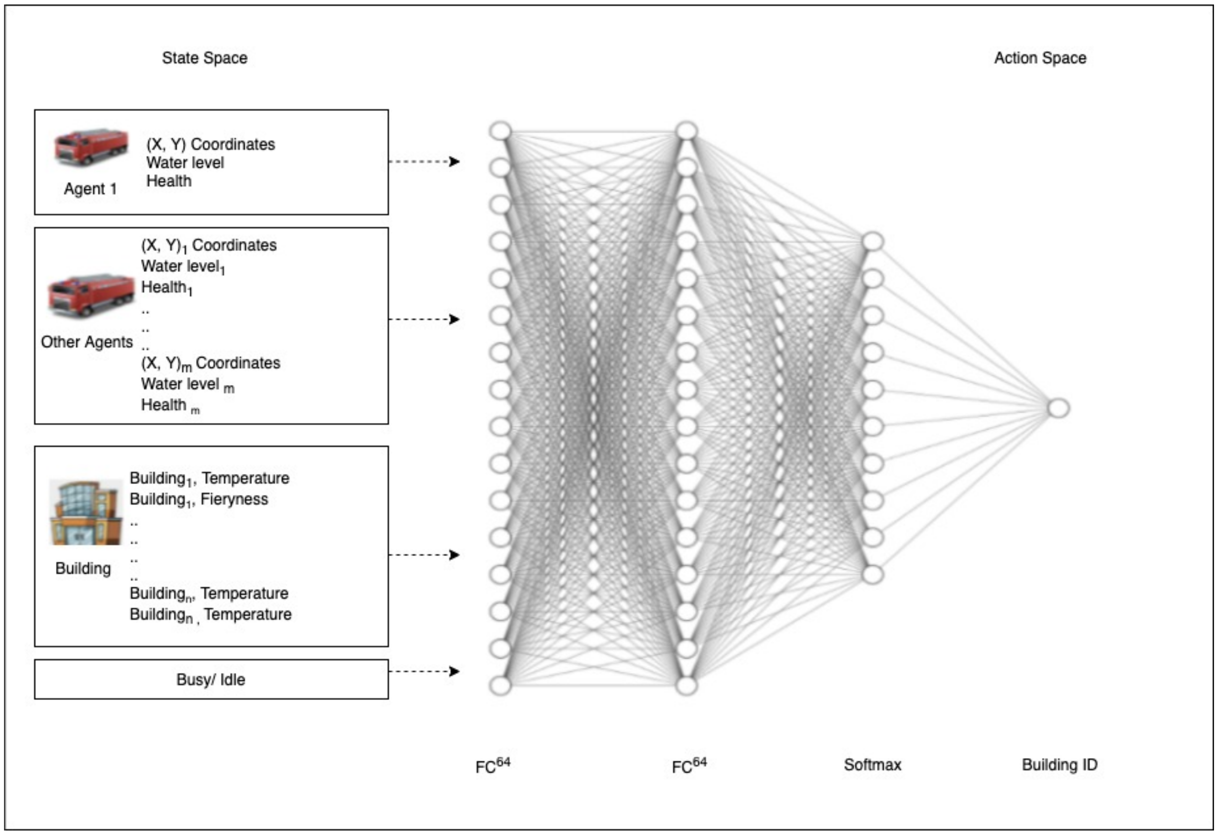

Figure 4.3: Schematic Illustration of the Model Architecture for Agent 1

Parameter	Values
Discount factor (γ)	.99, .993, .997, .999
n-step	32, 64, 128, 256
Entropy Coefficient	0.01
Learning rate	log-uniform (1e7 $\rightarrow$ 1e3)
Value Function Coefficient	0.5
Gradient Clipping value	0.5
Number of training minibatches	4
Clip Range	0.2
Optimizer	Adam

Table 4.5: PPO: Values used during hyper-parameter search and final values used for experiments with scoring

Parameter	Values
Discount factor (γ)	.99, .993, .997, .999
Learning rate	log-uniform (1e7 $\rightarrow$ 1e3)
Buffer size	25000, 50000, 100000
Exploration Fraction	0.1
Batch size	32, 64, 128, 256
Learning starts	1000, 2000
Target Network update frequency	500
Optimizer	Adam

Table 4.6: DQN: Values used during hyper-parameter search and final values used for experiments with scoring

Chapter 5

RCRS Gym Environment: Setup, Experiments and Results

In order to device the model presented in Chapter 3 and prove its concepts, an OpenAI gym environment [7] was created that allowed developing agents which are capable of extinguishing fire in the city. OpenAI is a framework which defines a standard structure that a DRL agent can be built with, thus making the structure of the agent easy to recognize by any developer and makes the agent compatible with any environment. Since OpenAI is built in Python, it can be easily connected with state-of-art DRL framework such as Tensorflow [1] with gym agents and use DRL techniques that use those frameworks provide. There are certain example algorithms available in OpenAI gym which simplifies the testing on a new environment.

The conceptual model of RCRS gym environment is presented in Figure 5.1. Section 4.1 elaborates about the setup that was created to adapt RCRS to allow the training of DRL agents. In Section 5.2, experiments that were carried out will be detailed and results will be analyzed.

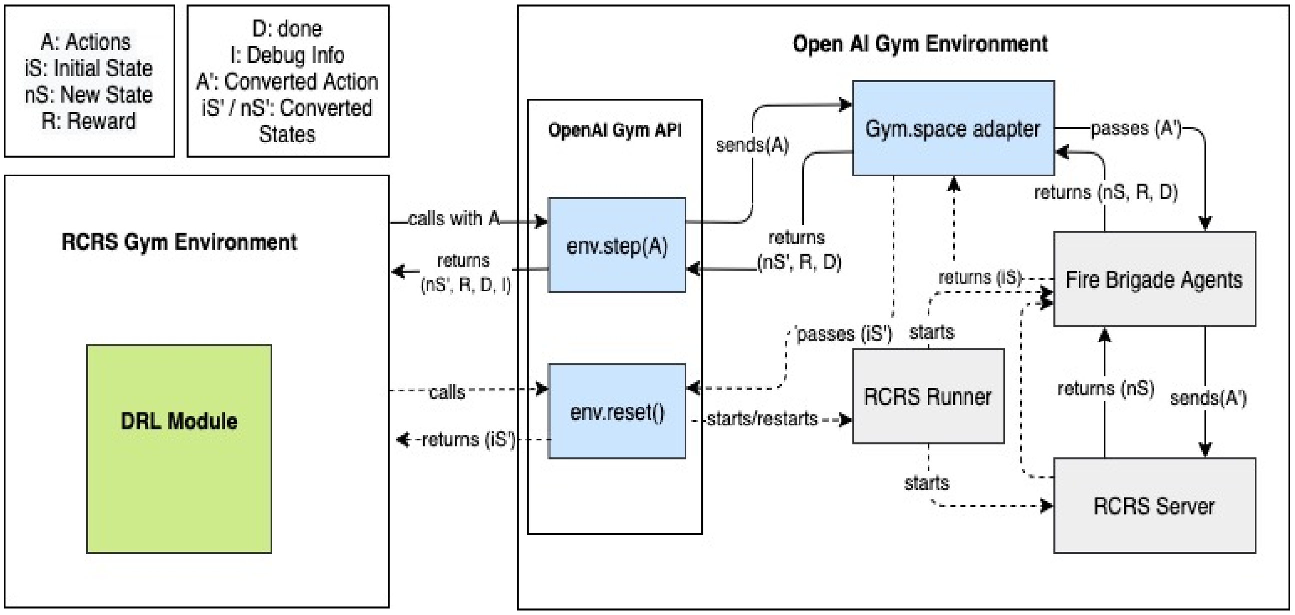

Figure 5.1: Conceptual model of the RCRS Gym environment and agent

5.1 Setup

This section will deal with the process of building the environment and the logic behind the decisions that are made. Section 4.1.4 will describe the goal of the agent, what tasks will it perform after training, introduction to various maps and the complexity of the environment. Section 5.1.1 will elaborate about gym framework and why it was selected. RCRS is written in Java and OpenAI gym in Python. There has to be a communication channel by which these two languages can communicate in order to send messages. Section 5.1.2 will discuss about how the communication between Java and Python was established. Finally, the overall implementation of RCRS-gym environment will be discussed in Section 5.1.3.

5.1.1 OpenAI Gym

Having a standard architecture where a reinforcement learning agent can be developed will be really helpful and this is what OpenAI gym provides. This standard architecture helps compare performance of different reinforcement learning algorithms in the same environment

[7].

OpenAI Gym maintains a repository known as *baselines* [12], which has examples of implementations of state-of-art DRL methods. But baselines have certain drawbacks that are rectified in a recently released fork – *stable baselines* that had unified code structure for all the algorithms, well documented functions and classes and PEP8 compliant. Hence we decided to go with stable baselines [1] [17]. These implementations can be used to validate the environment. The defined gym interface is made of two methods that the agent will use to interact with the environment:

- *reset*: This function is used to reset the environment to its initial state and returns the initial observation. It is called whenever a new episode is started.
- *step*: This function receives the action that the agent wishes to use in order to interact with the environment as the argument and returns the observation, reward, done and info.
  - *observation*: Current state information
  - *reward*: Reward collected after the end of every episode
  - *done*: returns true if episode is over
  - *info*: (optional) extra information that needs to be printed

In addition to the above attributes, "action space" and "observation space" should be defined in order to abstract the environment to generic code. Apart from the above mentioned functions, there are two more important attributes that need to be defined.

- *action space*: The space of possible actions that will be used to generate the actions. Possible values are
  - *Discrete:* A list of possible actions where each timestep only one of the action can be used
  - *MultiDiscrete:* A list of possible actions where each timestep only one action of each discrete set can be used

[1]`https://stable-baselines.readthedocs.io/en/master/`

  - *Box:* A N-dimensional box which contains every point in the action space
  - *MultiBinary:* A list of possible actions, where each timestep any of the actions can be used in any combination

- *observation space*: The space that defines the dimension of the environment's state. Possible values are the same as listed for action space.

In our case, since the actions are discrete, we selected Discrete action space. States are values that have integers as well as floating numbers between 0 to 15000, we decided to select Box state space that would allow all the values between 0 to 15000 to be possible.

5.1.2 Communication between Java and Python

RCRS is written is Java whereas OpenAI gym is in Python. Since at every timestep, state values have to be sent to the OpenAI gym agent and action values have to be received back, a connection has to be established. This connection between Java classes and Python classes was built using Google's protocol buffers [2] and gRPC [3].

Google's protocol buffers are extensible methods for serializing structures data like XML but is faster, smaller and simpler. They are independent of the programming language they are written in. In our case, protocol buffer's are used to generate methods in python and java so as to use the interpretation of both state space and action space.

Another important point to note down is that the original structuring of the RCRS code base, already had a server and a client connection which was done using socket programming. The server had all the information of the building state i.e. temperature and fieryness whereas client had the information of the agents i.e. (X,Y) coordinates, health points, water level and if the agent is idle or busy. Client was also the one that would receive the next action to be performed i.e. the next building id where the fire brigades should move to. Since RCRS would have to start first and act as a server to the Python agent, we had to build two servers, one that would send the building information and other that would send the agent information and take actions as well. The Python agent would act

[2] `https://developers.google.com/protocol-buffers/`

[3] `https://grpc.io/`

as client for these two servers. Listing 5.1 elaborates the protocol representation of the building information while Listing 5.2 elaborates the protocol representation of the agent information.

```
message BuildingInfo {
  int32 fieryness = 1;
  double temperature = 2;
  int32 building_id = 3;
}
```

Listing 5.1: Protocol buffer representation of the Building Information

```
message AgentInfo {
  int32 agent_id = 1;
  double x = 2;
  double y = 3;
  int32 water = 4;
  int32 hp = 5;
  int32 idle = 6;
}
```

Listing 5.2: Protocol buffer representation of the Agent Information

gRPC is an open-source remote procedural call (RPC) framework that can run in any environment. Its ease of implementation and pluggable support for load balancing made it our first choice for serialization. RPC server in our case is the gym environment whereas the fire brigade agents acts as RPC clients. Since everything is hidden in the RPC, RCRS (Java classes) will not need to know that it is interacting with the python agent.

The implementation part is detailed in Listing 5.3 and Listing 5.4.

```
service RCRSGymService {
  rpc getBuildingInfo (Empty) returns (BuildingInfo) {}
}
```

Listing 5.3: gRPC Implementation for Building Info

```
service RCRSGymService {
  rpc getAgentInfo (ActionInfo) returns (AgentInfo) {}
}
```

Listing 5.4: gRPC Implementation for Agent Info

5.1.3 RCRS-Gym Implementation

Lets elaborate on the two methods that were defined in Section 5.1.1 and show how the agents act in the RCRS environment using OpenAI gym. An environment is initialized by calling *gym.make* function. Then *reset* function is called that returns the initial state information about the agents. Next, a random action is sampled from the available action space and is sent to the RCRS environment using *step* function. *Step* function returns the subsequent state information, rewards achieved, if the episode is over and any additional information that the user may have defined. Pictorial representation of the working RCRS-gym working is shown in Figure 5.2.

In our case, an episode is considered *done* after the environment is run for a certain number of timesteps. For "Small" map the number of timesteps is 100 and for "Big" map it is 250. An example code that will run a random agent in RCRS environment is provided in Listing 5.5:

```python
import RCRS_gym
import gym

env = gym.make('RCRS-v2')
env.reset()
done = False
while not done:
    action = env.action_space.sample()
    obs, rew, done, info = env.step(action)
```

Listing 5.5: Example code that will run a random agent in RCRS environment

5.2 Experiments and Results

To start the testing and see if deep reinforcement learning works on RCRS, we started with experimenting on "Small" map. PPO and DQN algorithm were applied and the results were

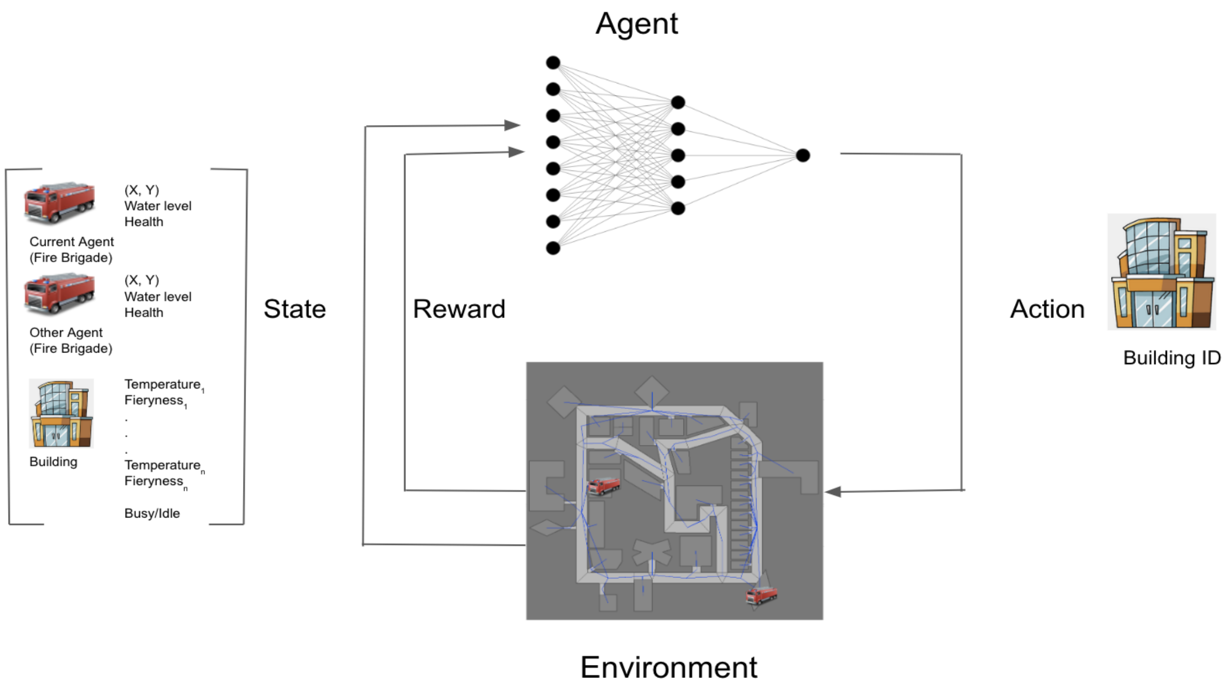

Figure 5.2: The agent takes in state information and the reward collected at every timestep from the environment. State information contains (X,Y) coordinates, water level, health of the all the fire brigades, temperature fieryness of the buildings and if the fire brigade is idle or busy. Agent then processes this information and outputs an action (Building ID) where the fire brigade needs to move next

compared with a greedy approach where the agents would always extinguish the building with highest fieryness value.

5.2.1 Small Map Experiments and Results

Simulations were run for 400 episodes (40,000 timesteps). Out of the several model architectures that were tried, a two hidden layer neural network, each layer having 64 units gave the best results for DQN. For PPO, a neural network having four fully connected layers, where first two layers had 128 units and last two layers had 64 units gave the best result along with the use of LSTM. Best hyperparameters are listed in Table 5.1 and Table 5.2.

Parameter	Best - Small Map
Discount factor (γ)	0.99
n-step	32
Entropy Coefficient	0.01
Learning rate	5e-4
Value Function Coefficient	0.5
Gradient Clipping value	0.5
Number of training minibatches	4
Clip Range	0.2
Optimizer	Adam

Table 5.1: PPO: Best hyperparameters

Parameter	Best - Small Map
Discount factor (γ)	0.99
Learning rate	5e-4
Buffer size	50000
Exploration Fraction	0.1
Batch size	64
Learning starts	1000
Target Network update frequency	500
Optimizer	Adam

Table 5.2: DQN: Best hyperparameters

Mean and standard deviation of the reward is calculated after every 5 episodes. Since the greedy algorithm doesn't learn, it had a constant reward value of 6.33. DQN achieved a highest cumulative reward of 5.89 and PPO achieved a highest cumulative reward of 5.88. Figure 5.3 shows that both the algorithms were able to learn in the environment improving their cumulative rewards over time. However, DQN attained highest cumulative reward in less number of episodes than PPO showing that it is a more effective algorithm for "Small" map setting.

Figure 5.4 and Figure 5.5 show how the agents gets better at extinguishing fire over the episodes. After 5 episodes, both the algorithms were performing badly with most of the buildings critically or totally burnt. After 150 episodes, agents started learning and higher number of buildings were extinguished. After 250 episodes, DQN was able to successfully figure out the buildings that caused the fire to spread throughout the map and was able to extinguish fire in those buildings. PPO was performing better than how it was performing after 150 episodes, but still not as good as DQN. After 300 episodes, PPO was also able to successfully extinguish the critical buildings. For better visualization on how the agents performed over episodes, refer to the GitHub page.

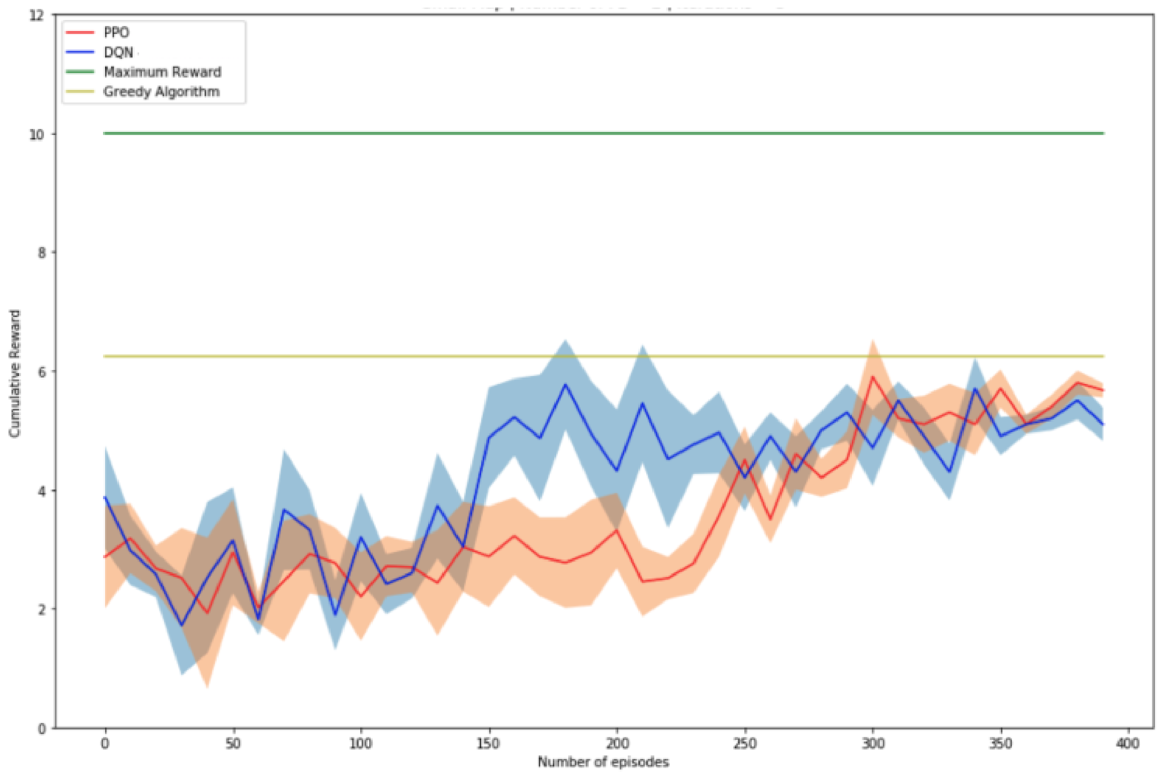

Figure 5.3: Rewards per episode for PPO, DQN and Greedy algorithm in "Small" Map setting. Dark line represents the mean value for rewards while shaded region is the standard deviation

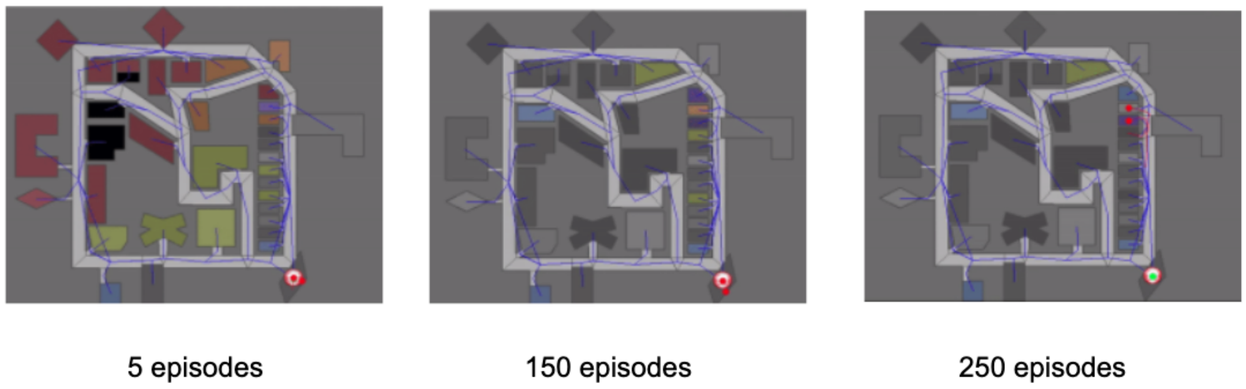

Figure 5.4: PPO: Fire Brigades learning to extinguish fire over 250 episodes. After 5 episodes, there was hardly any learning with most of the buildings either critically or totally burnt. After 250 episodes, agents learnt the critical buildings that need to be extinguished

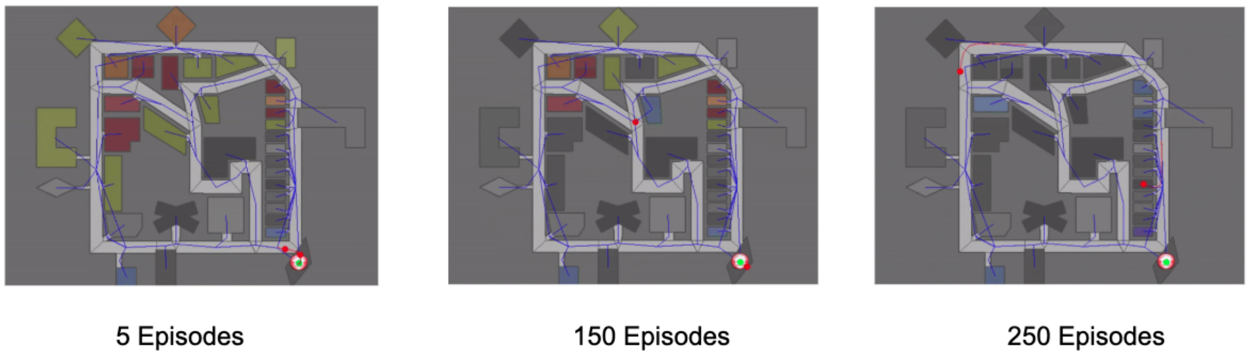

Figure 5.5: DQN: Fire Brigades learning to extinguish fire over 250 episodes. After 5 episodes, there was hardly any learning with most of the buildings either critically or totally burnt. After 250 episodes, agents learnt the critical buildings that need to be extinguished

5.2.2 Big Map Experiments and Results

Since the "Big Map" had greater number of buildings and fire brigade, both state space and action space increased. This increase in search space meant greater number of combination and increase in the number of episodes to run the simulation. We increased the number of episodes from 400 in ”Small Map” to 10000 episodes in ”Big Map”. This gave sufficient time for the agents to look for the optimal policy.

The simulations were started by testing DQN algorithms. Different network architectures and hyperparameters were tried but none of them showed any kind of learning. The primary reason could be that learning in a complex environment requires the agent to represent the knowledge at multiple level of spatio-temporal abstractions and to explore the environment efficiently. Kulkarni et al. [20] presented H-DQN, a framework to integrate DRL with hierarchical value functions that work at different temporal scales. A high level value function learns learns a policy over intrinsic goals and a lower-level function learns a policy over atomic actions to satisfy the given goals. H-DQN provides efficient exploration in complex scenarios. Hence we went ahead with H-DQN instead of DQN.

As mentioned above, several model architecture and hyperparameters, a two layer hidden neural network with first layer having 128 units and second layer having 64 units, showed the best performance for PPO. For H-DQN, a four layered neural network with first layer having 128 units and other three layers having 64 units gave the best result. Both the architecture were trained using LSTM. Best hyperparameters are listed in Table 5.3 and Table 5.4.

Figure 5.6 shows that H-DQN achieved a highest cumulative reward of 8.37 while PPO achieved a highest value of 8.84. If compared with the baseline greedy approach, PPO seems to perform better than H-DQN and shows learning as the episodes go on. After 5000 episodes, PPO performs consistently better than the greedy approach.

For both Small map and Big Map setting, models trained using LSTM performed the best. In order to confirm this, we compared the learning curves when model architecture includes LSTM and when it doesn't in the Big Map setting. Since PPO performed better in the Big Map setting, LSTM policy was trained using PPO. Figure 5.7 shows how agents starts to learn and perform better when trained using LSTM.

Parameter	Best - Big Map
Discount factor (γ)	0.99
n-step	32
Entropy Coefficient	0.01
Learning rate	5e-4
Value Function Coefficient	0.5
Gradient Clipping value	0.5
Number of training minibatches	4
Clip Range	0.2
Optimizer	Adam

Table 5.3: PPO: Best hyperparameters

Parameter	Best - Big Map
Discount factor (γ)	0.99
Learning rate	25e-4
Buffer size	50000
Exploration Fraction	0.1
Batch size	128
Learning starts	1000
Target Network update frequency	1000
Optimizer	Adam

Table 5.4: HDQN: Best hyperparameters

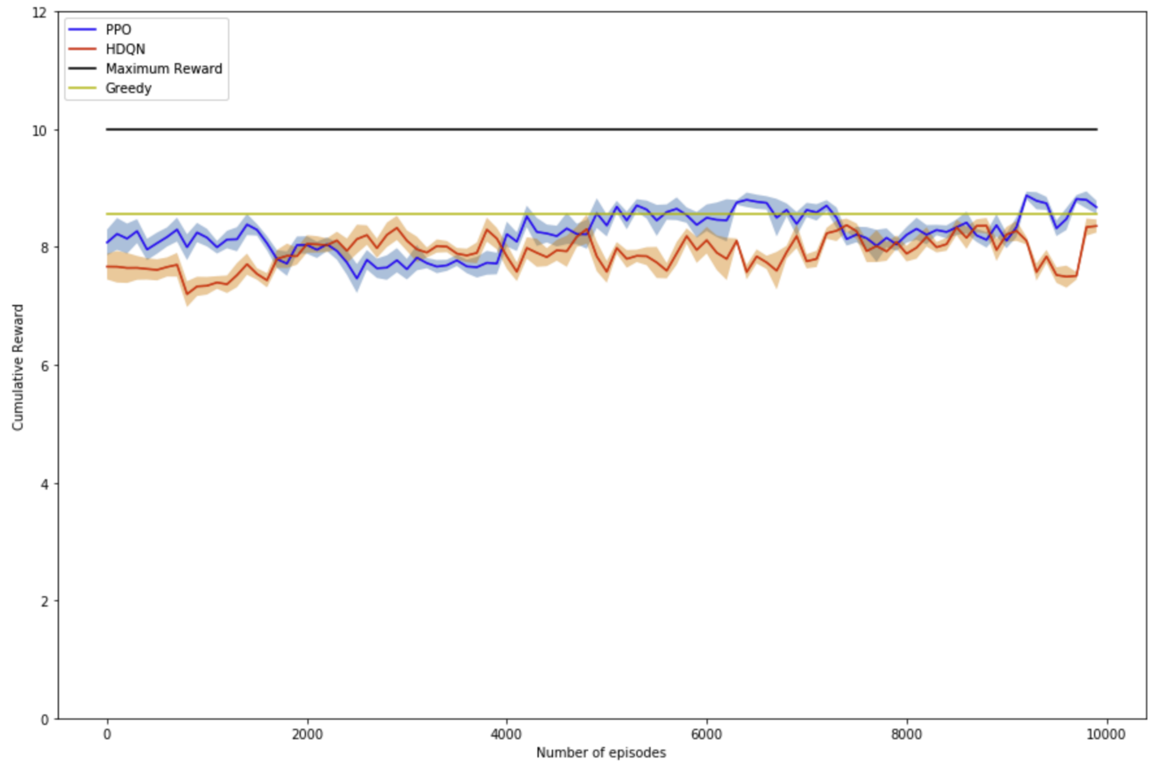

Figure 5.6: Rewards per episode for PPO, HDQN and Greedy algorithm in ”Big” Map setting. Dark line represents the mean value for rewards while shaded region is the standard deviation

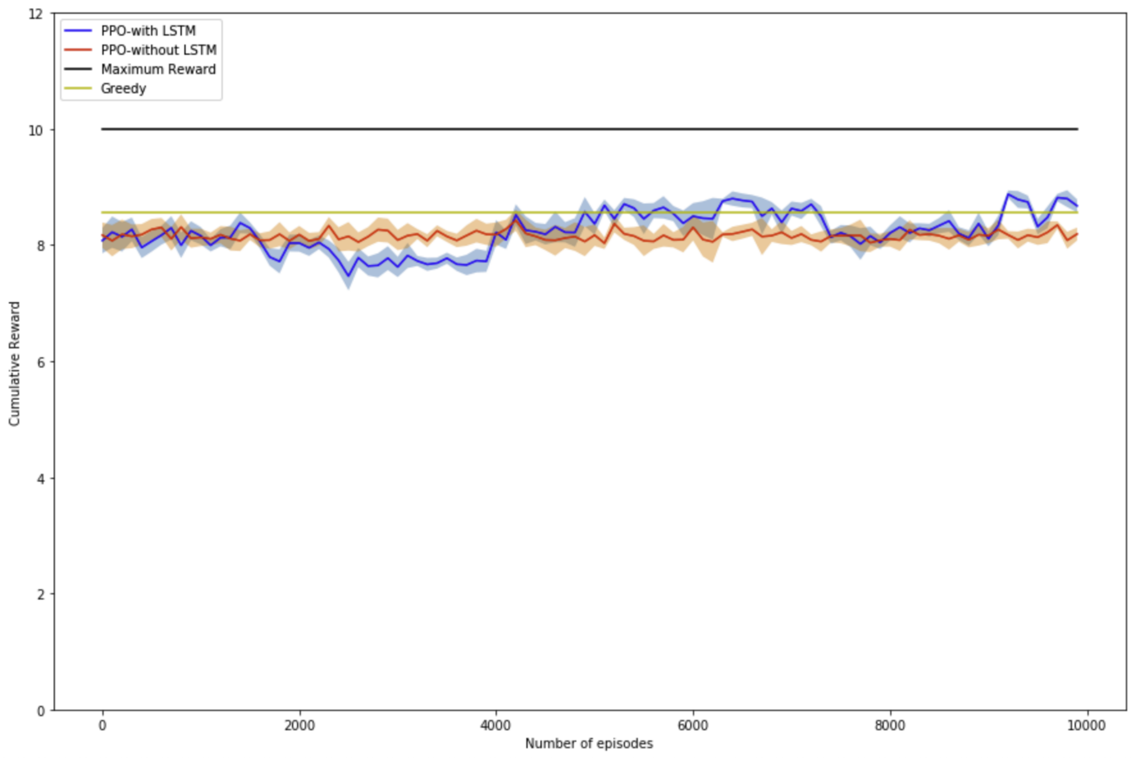

Figure 5.7: Comparison between the learning curves when PPO is used with LSTM and when it is not in the Big Map setting

Chapter 6

Conclusion and Future Work

In this report, we have presented how deep reinforcement learning can be applied to a multiagent scenario, specifically to RoboCup Rescue Simulator. We have built a RCRS-gym interface that will allow different RL algorithms to be applied to the simulator. The interface created provides an easy setup for developers to research RCRS using Python framework. The gRPC communication between Java and Python is also a feature that proved important to send actions and receive state information.

We evaluated two state-of-the-art DRL algorithms, Proximal Policy Optimization (PPO) and Deep Q-Networks (DQN). Both the algorithms were trained on two map setting: "Small" map and "Big" map. PPO and DQN were both able to demonstrate learning on the "Small" map with average cumulative reward increasing over episodes. DQN converged quickly than PPO showing that it performed better as compared to PPO. However, these algorithms were slightly subpar when compared to Greedy algorithm where the agents would always extinguish the building with highest fieryness value. In the Big Map setting, model was trained using PPO and and a variant of DQN i.e. H-DQN. PPO outperformed H-DQN and was able to show learning. However the learned agent was not able to significantly outperform the greedy algorithm and showed better performance for only a few episodes. One of the reason might be because the number of training episodes were not sufficient.

This study demonstrated that successfully trained agents can indeed learn a strategy to extinguish the buildings on fire proving that RCRS is a suitable environment to apply Deep Reinforcement learning techniques. However, there are various avenues that can be studied as future research. Is there a map setting where PPO and DQN performs better than greedy algorithm, are there situations where PPO learns quicker than DQN, can this be generalized by looking at the number of buildings in the map, are some of the questions that are still unanswered.

Another avenue to research for is how can we include other agents like police officers and ambulance teams to train them and accomplish their individual tasks successfully. Our current work assumes full observability to all the agents, future work can also focus on training the agents with partial observability.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, November 2016. USENIX Association.

[2] Abbas Abdolmaleki, Mostafa Movahedi, Sajjad Salehi, Nuno Lau, and Luís Reis. A reinforcement learning based method for optimizing the process of decision making in fire brigade agents. volume 7026, 10 2011.

[3] Omid Aghazadeh, Maziar Ahmad Sharbafi, and Abolfazl Toroghi Haghighat. Implementing parametric reinforcement learning in robocup rescue simulation. In Ubbo Visser, Fernando Ribeiro, Takeshi Ohashi, and Frank Dellaert, editors, *RoboCup 2007: Robot Soccer World Cup XI*, pages 409–416, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[4] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub W. Pachocki, Michael Petrov, Henrique Pond'e de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning. *ArXiv*, abs/1912.06680, 2019.

[5] Kakuchi S. Bilski, A. Kobe earthquake. *Maclean's*, 1995.

[6] Ali Bitaghsir, Fattaneh Taghiyareh, Amirhossein Simjour, Amin Mazloumian, and Babak Bostan. Uteternity's team description : Layered learning in robocup rescue simulation. 05 2020.

[7] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016. cite arxiv:1606.01540.

[8] Peter Corke, Ron Peterson, and Daniela Rus. *Networked Robots: Flying Robot Navigation Using a Sensor Net*, pages 234–243. 01 1970.

[9] J. Cortes, S. Martinez, T. Karatas, and F. Bullo. Coverage control for mobile sensing networks. *IEEE Transactions on Robotics and Automation*, 20(2):243–255, 2004.

[10] E. Dall'Anese, H. Zhu, and G. B. Giannakis. Distributed optimal power flow for smart microgrids. *IEEE Transactions on Smart Grid*, 4(3):1464–1475, 2013.

[11] Abhishek Das, Satwik Kottur, Jos'e M. F. Moura, Stefan Lee, and Dhruv Batra. Learning cooperative visual dialog agents with deep reinforcement learning. *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 2970–2979, 2017.

[12] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. Openai baselines. `https://github.com/openai/baselines`, 2017.

[13] Jakob N. Foerster, Yannis M. Assael, Nando de Freitas, and Shimon Whiteson. Learning to communicate with deep multi-agent reinforcement learning. In *NIPS*, 2016.

[14] Matthew J. Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. In *AAAI Fall Symposia*, 2015.

[15] Serhii Havrylov and Ivan Titov. Emergence of language with multi-agent games: Learning to communicate with sequences of symbols. *ArXiv*, abs/1705.11192, 2017.

[16] Pablo Hernandez-Leal, Bilal Kartal, and Matthew E. Taylor. Is multiagent deep reinforcement learning the answer or the question? A brief survey. *CoRR*, abs/1810.05587, 2018.

[17] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. `https://github.com/hill-a/stable-baselines`, 2018.

[18] Emilio Jorge, Mikael Kågebäck, and Emil Gustavsson. Learning to play guess who? and inventing a grounded language as a consequence. *ArXiv*, abs/1611.03218, 2016.

[19] H. Kitano, S. Tadokoro, I. Noda, H. Matsubara, T. Takahashi, A. Shinjou, and S. Shimada. Robocup rescue: search and rescue in large-scale disasters as a domain for autonomous agents research. In *IEEE SMC'99 Conference Proceedings. 1999 IEEE International Conference on Systems, Man, and Cybernetics (Cat. No.99CH37028)*, volume 6, pages 739–743 vol.6, 1999.

[20] Tejas D. Kulkarni, Karthik Narasimhan, Ardavan Saeedi, and Joshua B. Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. *CoRR*, abs/1604.06057, 2016.

[21] Ivette C. Martínez, David Ojeda, and Ezequiel A. Zamora. Ambulance decision support using evolutionary reinforcement learning in robocup rescue simulation league. In Gerhard Lakemeyer, Elizabeth Sklar, Domenico G. Sorrenti, and Tomoichi Takahashi, editors, *RoboCup 2006: Robot Soccer World Cup X*, pages 556–563, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[22] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[23] Ranjit Nair, Takayuki Ito, Milind Tambe, and Stacy Marsella. Task allocation in the robocup rescue simulation domain: A short note. In Andreas Birk, Silvia Coradeschi,

and Satoshi Tadokoro, editors, *RoboCup 2001: Robot Soccer World Cup V*, pages 751–754, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[24] Shayegan Omidshafiei, Jason Pazis, Christopher Amato, Jonathan P. How, and John Vian. Deep decentralized multi-task multi-agent reinforcement learning under partial observability. In *ICML*, 2017.

[25] Peng Peng, Ying ying Wen, Yaodong Yang, Quan Yuan, Zhenkun Tang, Haitao Long, and Jun Wang. Multiagent bidirectionally-coordinated nets: Emergence of human-level coordination in learning to play starcraft combat games. *arXiv: Artificial Intelligence*, 2017.

[26] Huy Xuan Pham, Hung Manh La, David Feil-Seifer, and Luan Van Nguyen. Cooperative and distributed reinforcement learning of drones for field coverage. *ArXiv*, abs/1803.07250, 2018.

[27] Tabish Rashid, Mikayel Samvelyan, Christian Schröder de Witt, Gregory Farquhar, Jakob N. Foerster, and Shimon Whiteson. Qmix: Monotonic value function factorisation for deep multi-agent reinforcement learning. In *ICML*, 2018.

[28] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015.

[29] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. 2017.

[30] Kun Shao, Yuanheng Zhu, and Dongbin Zhao. Starcraft micromanagement with reinforcement learning and curriculum transfer learning. *CoRR*, abs/1804.00810, 2018.

[31] Peter Stone and Manuela Veloso. Multiagent systems: A survey from a machine learning perspective. *Autonomous Robots*, 8, 05 2000.

[32] Sainbayar Sukhbaatar, Arthur Szlam, and Rob Fergus. Learning multiagent communication with backpropagation. *ArXiv*, abs/1605.07736, 2016.

[33] Richard Sutton and Andrew Barto. Reinforcement learning, second edition: An introduction. *MIT Press*, Second Edition, 2018.

[34] Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Neural Information Processing Systems*, NIPS'99, page 1057–1063, Cambridge, MA, USA, 1999. MIT Press.

[35] Ardi Tampuu, Tambet Matiisen, Dorian Kodelja, Ilya Kuzovkin, Kristjan Korjus, Juhan Aru, Jaan Aru, and Raul Vicente. Multiagent cooperation and competition with deep reinforcement learning. *CoRR*, abs/1511.08779, 2015.

[36] Jan Tozicka, Benedek Szulyovszky, Guillaume Chambrier, Varun Sarwal, Umar Wani, and Mantas Gribulis. *Application of Deep Reinforcement Learning to UAV Fleet Control: Proceedings of the 2018 Intelligent Systems Conference (IntelliSys) Volume 2*, pages 1169–1177. 01 2019.

[37] Nicolas Usunier, Gabriel Synnaeve, Zeming Lin, and Soumith Chintala. Episodic exploration for deep deterministic policies: An application to starcraft micromanagement tasks. *ArXiv*, abs/1609.02993, 2016.

[38] Babuschkin I. Czarnecki W.M. et al. Vinyals, O. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(350-354), 2019.

[39] Arnoud Visser, Nobuhiro Ito, and Alexander Kleiner. Robocup rescue simulation innovation strategy. volume 8992, 07 2014.

[40] Arnoud Visser, Luis G. Nardin, and Sebastian Castro. Robocup rescue simulation machine learning workshop. 2018.

[41] Bo Yang and Min Liu. Keeping in touch with collaborative uavs: A deep reinforcement learning approach. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 562–568. International Joint Conferences on Artificial Intelligence Organization, 7 2018.