UNIVERSITY
OF SUSSEX

# SIMULATION-BASED REINFORCEMENT LEARNING FOR SOCIAL DISTANCING

by

Candidate Number: 213150

A Dissertation

Submitted to the Department of Engineering & Informatics

University of Sussex

In Partial Fulfilment of the Requirements

For the Degree of Master of Science

September 2020

# CONTENTS

## ACKNOWLEDGEMENT / DEDICATORY

Throughout the writing of this dissertation, I have received a great deal of support and assistance. I would first like to thank my supervisor, Prof. T. Nowotny, whose expertise was invaluable in the formulating of the research topic and methodology in particular. Besides, I would like to thank my parents for their wise counsel and hospitality during these rough times. Specially thanks to my close-friend Utku Demir for helping me out with Cloud Computing when my laptop broke down. Finally, there are my other friends, who were of great support in deliberating over our problems and findings, as well as providing happy distraction to rest my mind outside of my research.

## ABSTRACT

Through agent-based modelling, and standard reinforcement learning algorithms at scale, we found AI agents can give insights about ongoing epidemics by simulating the disease. An epidemic simulation has been created with the Unity physics engine and we have analyzed its results with SIR graphs. We found clear evidence of the relation between social distancing and infection rates. We further provide evidence that multi-agent cooperation is effective on flattening the epidemic spread curve. In addition, in cooperative multi-agent settings agents found a way to self-quarantine themselves and decrease the infection rates significantly. Finally, we propose a flexible, easily extendable physics-based epidemic simulation framework that can be used as an RL environment benchmark both for single and multi-agent scenarios.

**KEYWORDS**

Epidemic simulation, reinforcement learning, cooperative multi-agent environment, agent-based-modelling, social distancing, COVID-19

## 1 INTRODUCTION

Ever since the outbreak of Severe Acute Respiratory Syndrome as known as Covid-19 came out, life has changed drastically. Many researchers dedicated themselves to fight against the spread of this fatal virus and minimize the loss of life. Artificial Intelligence researchers are focusing their expertise knowledge to develop mathematical models for analyzing this epidemic disease [1]. An epidemic disease requires quick decisions to be made about interventions that could reduce or contain the disease spread. Decision-makers need to be agile about their strategy since they race with time. Every second that is wasted increases the damage to humanity. In contrast, to decide confidently which strategy will work, decision-makers need to analyze many scenarios and variables since if they make a wrong decision, that can also cause harm. This is an optimization problem in which researchers cope by creating their own data and utilizing Reinforcement Learning to develop optimal strategies [2].

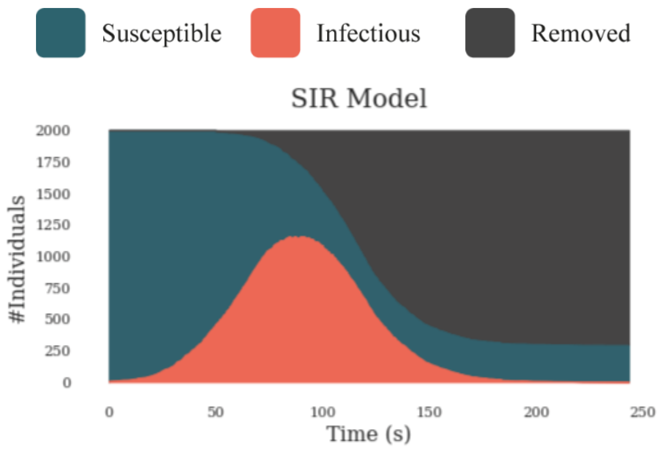


*Figure 1: An SIR curve as one of our epidemic simulation's output. The vertical axis represents the number of individuals. The horizontal axis represents the time pass in the simulation. The red area shows the number of infected people. The dark green area shows susceptible and the gray area shows recovered/removed individuals. The disease starts to spread from the first second of the simulation and peaks around 75 seconds in simulation time.*

Reinforcement Learning (RL) is an area of Machine Learning where an agent learns the best behavior by interacting with the environment. Creating these complex environments and artificially intelligent agents that solve complex human-relevant tasks has been a life-long challenge for RL researchers [3]. The environment is a crucial component of RL and determines broadly the task that the agent has to solve. Agents and environments cannot be considered separately, and it is only a design choice for the researcher to determine where the environment starts and the agent ends. Most of the time the environment is defined as anything that an agent cannot have direct control over. The agent perceives its environment through sensor observations and acts upon that environment through effectors. For each action selected by the agent the environment provides a reward and the aim of the agent is to maximize the total rewards that it gets. In every step, the agent performs an action by following a strategy which is called policy. The agent starts with a random policy and as training continues the policy is optimized by a learning algorithm.

For our RL task, we used deep learning which made our task called "deep reinforcement learning task". By using deep neural networks (DNN) researchers have been able to solve a wide range of complex decision-making tasks that were previously out of reach for a machine [4]. For our deep RL task we chose to use a policy gradient method, the Proximal Policy Optimization algorithm (PPO). Policy gradient methods are RL algorithms that rely on optimizing policies with respect to long-term cumulative reward. Whereas other family members of policy gradient methods perform one gradient update per data sample, PPO uses multiple epochs of minibatch updates. The algorithm was designed by OpenAI. [3] and it is used in numerous different tasks from robotics to Atari games. On a collection of benchmark tasks, PPO outperformed other online policy gradient methods and had a better balance between sample complexity and simplicity. OpenAI defines its algorithm with three features. Easy code, sample efficiency, and ease of tuning. We explained how this algorithm works in detail in following chapters.

Our deep RL task was decreasing the damage of an epidemic outbreak and flatten the curve by taking individual precautions such as social distancing. To create this task, we introduce a new cooperative multi-agent physics-based reinforcement learning environment for control of epidemic spread. Through only a health status-based sparse reward function, agents learn many human-relevant skills to protect themselves from epidemic outbreaks including social distancing and self-isolation when they get sick. For example, agents learned how to maintain a balance between collecting reward boxes and not risking getting infected. We find that for several tests, curriculum learning can visibly change the results of the training. Setting the task step by step with increasing difficulty levels helped agents to learn better strategies and to converge the loss function closer to the global minimum. In addition to that, we showed pre-trained agents learned the task faster than agents which are trained from scratch. Moreover, we observe signs of collaboration and simple communication between agents even though they don't get direct rewards from their actions. For instance, in one of the trainings, infected agents learned to gather up in a location where they avoid infecting others without knowing each other's health status. This behavior demonstrates that they found a way to tell that they are infected or not to other agents. In every training, agents learned to avoid each other and demonstrate social distancing.

Social distancing is one of the most effective precautions that individuals can apply to their daily life. It is a successful strategy to prevent infectious diseases from spreading. It has many forms but at its core, the aim is to keep people apart enough from each other by putting physical distance and/or confining them to their homes. In this article, we took the idea of social distancing and simplified it to physical interaction. In other words, we assume social distancing is just having a physical distance between individuals. An infection mechanism starts when individuals are closer to each other than the threshold value. The threshold value is defined as minimum distance between individuals. Being closer increases the chance of getting infected. In the following chapter, the infection mechanism is explained in detail.

To analyze possible outcomes, we used SIR graphs which are widely used mathematical models that provide insights about the infectious diseases. The model divides individuals into three categories. Susceptible, Infectious, and Recovered. Recovered ones can be also called Removed since they don't have any effect on the simulation. We also used the same categories in our simulation. Although it has been proven that the individuals have short-term immunity after recovering from the COVID-19, the character of the virus is still ambiguous. To avoid ambiguity, we assumed when individuals recovered from the disease, they will be immune to it forever. After recovery, they don't have any effect on the curve. We compared SIR graphs of test simulations with and without AI is added. In addition, we have considered wearing a mask is a risk reducing factor therewith we assumed it decreases the exposure distance of the disease. Then we evaluated the effect of mask wearing percentage among individuals on epidemic spread.

In this article, we introduce an environment that can simulate epidemic spread with the physics engine of Unity. The main contributions of this work are 1) clear evidence that social distancing is mathematically a correct way to flatten the SIR curve. 2) A demonstration of how agent-based strategies and advances in computing can be leveraged to determine the optimal policy in an epidemic outbreak for a particular environment without expert human guidance [5]. 3) Evidence that agents found a way to communicate with others gather up and self-isolate themselves.

# 2 RELATED WORK

In this section, we explain key concepts of reinforcement learning and learning algorithms. Then we discuss related work in the multi-agent reinforcement learning domain (MARL) and compare some of the known RL simulation frameworks. Lastly, we review some background on agent-based modelling, computational analysis of epidemic outbreaks and use of reinforcement learning in epidemic simulations.

## 2.1 REINFORCEMENT LEARNING

Sutton -who is considered one of the founding fathers of modern computational reinforcement learning (RL) - explains RL with the following example: An infant explores itself and its environment without any explicit teacher, but it does have a direct sensorimotor connection to its surroundings. By practicing the same actions, it produces a wealth of information about cause and effect, about the consequences of actions, and about what to do in order to achieve goals [8]. People learn by interacting with their environment and trying to make inferences from their experiences. Whether it is learning how to ride a bike or hold a conversation, the process works the same. We have an awareness about our environment via our observations and we constantly try optimizing ourselves by predicting our action's consequences. That's how any human learns from interaction and it is a foundational idea underlying nearly all theories of learning and intelligence [6]. The problems which are solved by interacting with them can be defined as Reinforcement Learning tasks. These tasks are essentially closed-loop problems in a way that taken actions shape future inputs. In this chapter, we describe how the RL tasks can be formalized as an agent that has to make decision in an environment to optimize a given notion of cumulative rewards [4].

Two main characters of RL are the **agent** and the **environment**. The agent lives and interact with the environment and the environment includes anything that is not directly controlled by the agent. At every step, the agent observes its surroundings and then decides on an action to take according to that observation $o$. The observations are partial description of the agent's state. In board games such as Chess and Go the environment is **fully observed**. This means agent has all the information about its current state. In contrast, In many RL applications, the state of the agent is **partially observed.** However, RL agent does not require complete knowledge or control of the environment and it uses trial-and-error experience to compensate the uncertainty and make logical inferences. The agent also perceives a **reward** signal from the environment, a number that tells it how good or bad the current world state is.

At each step, the agent comes to a state $S_t$ with a reward of $R_t$. With its current strategy, the agent gives a decision and takes an action $A_t$ which gets the agent one step further in a state called $S_{t+1}$. The agent doesn't get any external support about what to do to solve the problem, but instead, it learns from its own actions and consequences which may result in getting a reward. By taking the difference between actual and expected rewards it tries to optimize its strategy and takes another action. Thus, the process consists of a trial-and-error search.

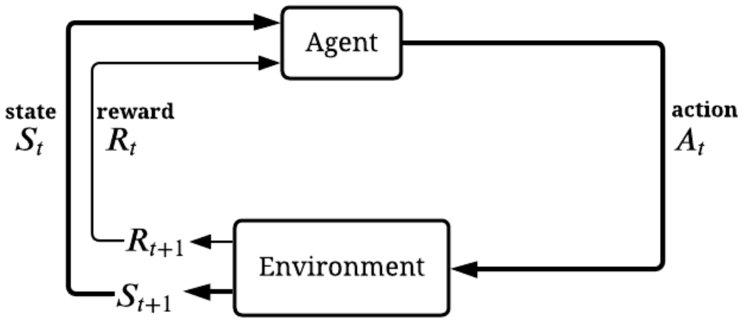


*Figure 2: Simplified diagram of RL closed-loop process representation (inspired by Sutton, Introduction to Reinforcement Learning)*

To understand how RL works better we need to introduce additional terminology such as action spaces, policies, reward and value functions.

### 2.1.1 Action Spaces

The set of all valid actions in a given environment is called **action space** and the environment may have either a **discrete** or **continuous** action space depending on the task. For instance, in a board or Atari game the action space is discrete since there is only finite number of moves are available to the agent. On the other hand, real-world RL tasks often require continuous action spaces where the agent may control a robot. Both action space types have their own advantages and some families of learning algorithms can only be applied in one of them. Since learning in a continuous action space is more difficult to converge and takes more time, we chose to use a discrete action space for our task.

### 2.1.2 Policies

The agent decides its actions by following a strategy called **policy**. During the training the agent tries to optimize it's policy in order to maximize the expected return. The policy can be **deterministic** or **stochastic**. Deterministic policies are used in deterministic environments where there is no uncertainty and the policy maps states to actions e.g. in Chess. For a deterministic policy **μ**, the action $\boldsymbol{a_t}$ taken at a specific state can be shown as:

$$\boldsymbol{a_t} = \mu(\boldsymbol{s_t})$$

On the other hand, stochastic policies output probability distributions over actions. For a stochastic policy $\boldsymbol{\pi}$, the probability of taking an action $\boldsymbol{a_t}$ can be shown as:

$$\boldsymbol{a_t} \sim \pi(.\,|\boldsymbol{s_t})$$

In our task, we optimized a stochastic policy by using a learning algorithm called Proximal Policy Approximation (PPO).

### 2.1.3 Reward Functions

The reward function $\boldsymbol{R}$ describes how the agent "ought" to behave and it is critically important for reinforcement learning. It depends on current state of the agent, next state and the action just taken. Although frequently this is simplified to just a dependence on the current state.

$$r_t = R(s_t, s_{t+1}, a_t) \;,\; r_t = R(s_t)$$

The agent's goal is to maximize the expected cumulative reward **G(t)** in the long run.

$$G(t) = R_{t+1} + R_{t+2} + R_{t+3} + \ldots + R_T$$

where T is a final step. This approach makes sense when there is a final step in the task such as finding the exit of a maze or checkmate in a chess. When there is a clear end of interactions between agent and environment and task breaks naturally into subsequences, we call every sequence as an **episode**. Each episode end with a special state called the terminal state, followed by a restart of the environment. Tasks with episodes of this kind are called **episodic tasks** [6]. In many cases the agent-environment interaction does not break naturally into sequences and goes on continually without limit. We call these **continuing tasks**. Calculating expected return and finding an optimal policy $\pi$ can be problematic in these situations since $T \to \infty$ and all policies that have obtain on average a positive reward would sum up to infinity. To overcome this problem researchers came up with an additional concept called **discounting**. The way expected return is calculated got slightly more complex conceptually but much simpler mathematically with the discount term.

$$G(t) = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

where $\boldsymbol{\gamma}$ is a parameter, $0 \le \gamma \le 1$, called discount factor. A reward received k time steps in the future worth only $\gamma^{k-1}$ times.

If $\gamma = 0$, the agent will be completely myopic and will only learn about actions that produce an immediate reward. The reason why discounting factor penalizes the rewards in the future is to avoid higher uncertainty of the future rewards. In addition, mathematically an infinite sum of rewards may not converge to a finite value without discount factor. Therefore, we also utilize discount factor in our reward function.

The way how designer choose to create reward functions in a RL task can significantly affect the convergence speed or not getting stuck to local minima. A common mistake is to design based on how the agent will solve the task. Instead, a correct design should be based on “What will the agent achieve” [14]. The suggested reward function for designing a simulation of epidemic outbreak is giving a penalty for getting infected and at the end of the episode giving a reward if the agent is not infected [2]. By this strategy, the agent has to discover a long sequence of “correct” actions to find an optimal policy $\pi^*$ [7]. These kinds of rewards are called **sparse-rewards** since the agent doesn’t get feedback very often. Moreover, these rewards are **extrinsic** since they are hand-designed by researchers and given externally into RL algorithms which can cause sub-optimal policy convergence.

One common way is to endow the agent with a sense of **curiosity** and to reward it based on how surprised it is by the world around it [8]. The idea comes from baby individuals who do not have any task except their intrinsic motivation to explore. This sense of curiosity provides needed entertainment for the them. Likewise, when an agent gets an unexpected reward, it surprises and tries to develop new strategies to explore unknown states and gets more surprised. Hopefully, along the way of getting confident on states agent will learn a better policy with higher cumulative rewards.

### 2.1.4 Value Functions

In order to determine the value of a state or an action in terms of expected total future rewards we use **value functions**. They create an efficient way of acquiring future rewards by calculating the current value of a state or action. We denote them by V(s), and they are used one way or another in almost every RL algorithms. The **state-value** function is used to calculate the state of s with the expected return as:

$$V_\pi(s) = E_\pi[G_t | S_t = s]$$

Similarly, **action-value** of a state-action pair is defined as:

$$Q_\pi(s, a) = E_\pi[G_t | S_t = s, A_t = a]$$

where Q is called “Quality-value” of that action. Moreover, the difference between action-value and state-value gives the **advantage function A** as:

$$A_\pi(s, a) = Q_\pi(s, a) - V_\pi(s)$$

Advantage function gives us a relative advantage of an action of how much better it is than others on average since for some RL algorithms we do not need to describe how good an action in an absolute sense. It will come handy when we use with a policy gradient method in following chapter.

## 2.2 LEARNING ALGORITHMS

One of the fundamental distinctions between RL algorithms is whether the agent has to access to a model of the environment or not. The advantage of having a model is that it creates sample efficiency by allowing the agent to plan possible action sequences and explicitly deciding between its options for finding the optimal action [9]. However, most of the time the ground-truth model of the environment is not available to the agent. Algorithms which use a model called **model-based** methods and those that don’t use a model called **model-free** methods. In our implementation we chose model-free methods due to easier implementation and tuning although they don’t have the advantage of sample efficiency as much as model-based methods.

The RL algorithms also can be divided into categories with the question of "what they learn". The learning can be based on policies, action-value functions and/or environment models. In model-free methods there are two main approaches as **Policy Optimization** and **Q-learning**. Policy optimization methods use stochastic policy $\pi_\theta$. In our case, the $\theta$ parameter corresponds to the weights of a deep neural network. Using gradient ascent, we can find the best θ that produces the highest return. This optimization process is almost always performed as an **on-policy** method which attempts to evaluate or improve the current policy that is used to make decisions. In contrast, Q-learning methods learn an approximator $Q_\theta(s, a)$ for the optimal action-value function and uses an **off-policy** method. Off-policy learning algorithms evaluate and improve a policy that is different from policy that is used for action selection. In addition, since off policy methods reuse past experience, they don't require new samples for each gradient step. This quickly becomes crucial as the number of gradient steps increases, the process becomes extravagantly expensive for on-policy algorithms. Furthermore, samples per step needed to learn an effective policy increases with task complexity.

On the other hand, on-policy algorithms such as Proximal Policy Optimization (PPO) do not use old data, which makes them weaker on sample efficiency. But in return, they can optimize the objective that we care about -policy performance- and this idea works out mathematically since in many scenarios the old data becomes useless as it gets too different from the current state. On-policy algorithms are better on not-yet understood states and actions since they always try to optimize what they have currently. Consequently, as the agent learns more about the environment, the agent's performance should approach optimality. As a result, we decided to continue with PPO for our deep RL task as it is discussed in detail in following chapters.

## 2.3 MULTI-AGENT REINFORCEMENT LEARNING

In the context of Multi-agent, there have been many kinds of researches in the literature. Many reinforcement learning tasks involve the participation of more than one single agent which fall into the area of multi-agent reinforcement learning (MARL) [10]. Recent years have witnessed astonishing advances in MARL such as OpenAI's Hide and Seek game and Dota 2 AI OpenAI Five [3][11] These developments became possible due to the development of deep neural networks (DNNs). In multi-agent RL tasks, agents operate in a common environment and each agent aims to optimize its own cumulative reward by interacting with the environment and other agents [12]. Due to the interaction between agents, the task complexity, and according to that the needed computation power can increased exponentially with problem size. On the other hand, under the favor of the complexity it provides, multi-agents systems reserve a place in many areas from social science to finance [13][14]. Multi-agent algorithms can be divided into 3 categories; cooperative, competitive, and the combination of these two depending on the task which agents solve. In cooperative settings, agents collaborate while trying to optimize the common long-term cumulative return. On the contrary, in competitive multiagent tasks, the cumulative reward of agents sums up to zero. Moreover, the combination of these two is called "Ecosystem" which is multiple interacting agents with independent reward signals. This kind of environment can be thought of as an environment full of animals where some of them will collaborate and some of them will compete.

Apart from interaction between agents in MARL, the basic framework of multi-agents differs from single-agent settings in terms of stationarity. Agents improve their policies concurrently which creates self autocurricula [3] and the environment faced by agents becomes non-stationary in MARL [15]. Eliminating the stationary environment settings is also a choice of design. Although RL methods optimized themselves in stationary environments better than non-stationary ones they are restrictive and usually overfit the task. As researchers, our goal was to get as close as possible to a real-world scenario since almost every one of the real-life applications are non-stationary [15]. Furthermore, in [16], authors discuss that MARL systems suffer from the curse of dimensionality also known as the combinatorial nature of MARL. One way to overcome this difficulty in multiagent scenarios is the use of search parallelization which is possible with neural networks. In upcoming paragraphs, we will discuss the advantage of using deep neural networks with approximation algorithms such as the PPO in deep reinforcement learning (DRL).

## 2.4 COOPERATIVE MULTI-AGENT

Cooperation between agents fits the definition of surviving in an epidemic outbreak therefore we focused on developing a cooperative environment. Agents create strategies together even though they don't share the same observations. This collaboration comes from having a common goal. There are two types of learning strategies in cooperative multi-agent RL. **Team learning** and **concurrent learning**. Team learning is a common and easy way to design agents where they learn the same set of behavior [17]. The advantage of the team learning approach is that it can use single-agent RL techniques which sidestep the complexity of the co-adaptation of several learners as it is the case in concurrent learning.

### 2.4.1 Team Learning

Team learning can be divided into two categories. *Homogeneous* and *heterogeneous* team learning.[17] In homogeneous team learning all agents have the same goals and actions. The only differences among them are their sensory observations and their current states which differentiate their decisions. They learn single-agent behavior which is known by every agent in the environment. On the other hand, heterogenous team agents can develop more complex behaviors with different roles in the team. Heterogeneous teams have larger actions space but generally converge to a better solution through agent specialization.[17] Choosing among these approaches depends on the reinforcement task and some problems do not require agent specialization. In our task, we believed individuals do not need specialization and therefore we implemented a homogeneous team learning multi-agents to represent the individuals in equal conditions. As an advantage of utilizing the same brain, the search space is remarkably reduced during the training. Surprisingly in our case, as the author mentions in [17], homogeneous agents learned to act heterogeneously due to the development of sub-behavior that differs based on the agent's health status. This behavior is discussed in the evaluation chapter.

### 2.4.2 Concurrent Learning

A common alternative learning strategy to team learning in cooperative MARL is concurrent learning. The fundamental difference is that each agent has its own brain and they attempt to improve different parts of the team easier since the tasks can be learned independently to a degree. However, the problem with concurrent learning is learners' co-adapting. In team learning where agents can use standard single-agent RL algorithms, they explore the environment while improving their policy. On the contrary, in concurrent learning agents can make obsolete assumptions about others' behaviors while others modify their current behavior [18]. One way to tackle this problem is assuming that other agents are also part of the dynamic environment although agents are more than dynamic units, they also improve their own behavior during training which makes convergence harder.

### 2.4.3 Communication in Cooperative MARL

Communication between agents in MARL is a subject of different opinions. In [19], Stone and Veloso argue that communicating agents are not really multi-agents. Instead having unrestricted communication between agents decreases the task to a single-agent RL problem. Real-world applications mostly have restrictions in terms of latency and throughput in communication. Therefore, we believe that a correct multi-agent problem should need some restrictions. In addition, knowing other agents' states via communicating can exceptionally increase the search space which can sabotage more than it helps to find the optimal policy. Edmund recommends in his paper a hard-coded communication system in order to simplify the learning process [20]. Direct and indirect are two types of communication styles in MARL. Direct communication can be defined in which agents inform each other by sharing their own sensor information. Indirect communication methods involve not explicit sharing but the modification of surroundings. For instance, leaving footsteps in snow or white smoke behind while flying with an airplane, standing in a special location where it means something to other agents. Most of the indirect communication literature comes from social insects' behaviors. We observed simplistic indirect communication in our simulation which is described in detail in further chapters.

## 2.5 RL SIMULATION FRAMEWORKS

Even though MARL is a recent area, there are plenty of simulators with different characteristics for RL agent training. Furthermore, these simulators are generally open-sourced and have built-in RL algorithms for ease of researchers. However, a small percentage of them have support for designing new environments. We described simulators that fit our research design below.

- A platform for designing artificial intelligence experiments in Minecraft called Malmo [21]. As the RL community expands, researchers looking for simulators to train their own agents and design their own tasks. Minecraft is an ideal base for such platform since it provides a sandbox with many varieties. Malmo provides an abstraction layer on top of the game and supports multiagent scenarios.

-Arcade Learning Environment (ALE) is an open-source platform that is built on top of the Atari 2600 Emulator [22]. The platform presents hundreds of Atari games and many research challenges including imitation-learning, transfer learning, and intrinsic motivation. Rather than creating new environments ALE is more suitable for testing domain independent agents.

-OpenAI Gym is a toolkit for reinforcement learning research published in 2016 [23]. The platform aims to combine the best features of previous platforms such as a variety of environments, continuous control, creating benchmark collection, and versioning the platform which guarantees that older results remain meaningful and reproducible. Although it is possible to create custom environments, the platform mostly focuses on developing and comparing RL algorithms. For physics-based training, it uses MujoCo physics simulation which is not free.

- Arena is a MARL platform based on a world-leading game engine in Unity [24]. It has 35 example environments of diverse logics and representations in it. The advantage of this platform is creating environments in Unity which is a highly used engine for creating games. With the help of user-friendly GUI and community support, Arena provides a highly flexible platform for researchers who want to design their own environments.

-The Unity Machine Learning Agents Toolkit (**ML-Agents**) is an open-source general platform for training intelligent agents [25]. Throughout an easy-to-use Python API, agents can be trained with reinforcement learning, imitation learning, neuroevolution, or other state-of-art machine learning methods. The platform has an expanding collection of benchmarks and +15 example environments from single-agent tasks to ecosystems. In addition, it contains some of the examples of Arena. The toolkit is one-step ahead of Arena by being a native product of Unity. In other words, researchers do not need to use any 3$^{rd}$ party product to use ML-Agents for training. The only thing that researcher needs to do is importing the ML-Agents package into the project and train the agents easily. Unity has its own physics engine called *PhysX* as it is a game engine therefore it doesn’t need any other rendering or physics simulation engine like MujoCo.

One drawback of Unity ML-Agents is that there are not as many RL algorithms that can be used by researchers as in OpenAI Gym. However, Unity solves this problem by creating environments that are convertible to Gym environments. Therefore, they can also be used with Gym instead of Unity if desired. Though gym convertibility is only supported for single-agents, Unity ML-Agents package is actively developing.

After comparing these many simulator options, we choose to continue with Unity and ML-Agents package as our simulation platform.

## 2.6 AGENT-BASED MODELLING

The history of agent-based modelling (ABM) can be traced back to the Simula programming language, which was developed in the mid-1960s and widely used as the first framework for automating step-by-step agent simulations [26]. ABM uses simple rules which can result in different types of complex behavior. These models consist of interacting rule-based agents to create real-world-like complexity. Back in the days, the rules were strictly defined by researchers as hard-coded and therefore it was hard to generalize and get interesting results [27]. Over time, an extensive literature has developed on creating simulations and a series of studies have indicated that there is two major approaches in developing them [28]. At the one end, there is what we call the "brute force" method which includes designing every piece of the simulation. This method works much faster and it doesn't need any AI training since every action is pre-decided. However, finding an optimal behavior of how the simulation works requires redoing large chunks of the model or even starting over, depending on the significance of the change. In addition to that, in many cases, the wanted behavior can be much more complex than what researcher codes. For example, in a real-world autonomous driving task, researchers created their own synthetic data by designing a camera that gets RGB images from the virtual environment [29]. They had to use augmented data in their training since coding images by hand is not possible.

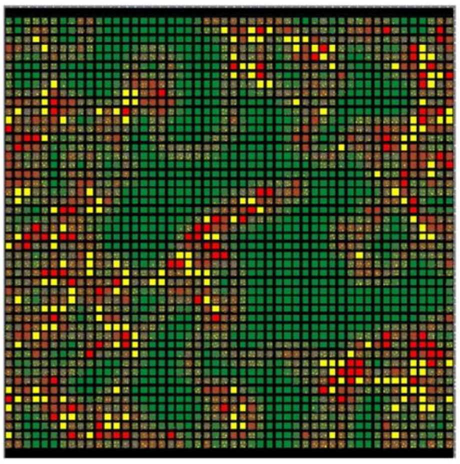


*Figure 3:Two-dimensional agent-based epidemic simulation visualization. Each square represents an individually programmable agent. Color-coding allows easy visual tracking of agents with different health status. (Microbial Threats to Health: Emergence, Detection, and Response 2003)*

The other end of this design process spectrum is an extremely flexible simulation in a way that researchers define rules as minimal as possible. This creates a suitable environment for AI [28]. In this kind of simulations, at first agents don't have any assumptions about their environment. By trial and error, they develop an internal model which represents how they understand the system by observing the surroundings and collecting data. Letting the agent create its own strategy rather than coding one, creates an opportunity to generate more comprehensive and complex behavior. Additionally, agent-based simulations generally have many parameters that need to be tuned. By using brute force, it would take much longer to explore every possible scenario that can happen. On the other hand, utilizing sub-sampling and creating scenarios most likely to occur, AI can deliver dramatic speed increases for large-scale ABM simulations.

## 2.7 EPIDEMIC SIMULATIONS

Epidemiology is a science interested in investigating all the factors that determine the presence or absence of diseases and disorders. A disease is called endemic if it persists in a population. Transmission mechanism of these infectious diseases have been a complex mathematical problem which cannot be solved without creating large scale dynamics of disease spread. Therefore, mathematics and statistics have been crucial for epidemiology since 1766 when Bernoulli published his evaluation of life expectancies and death rates [10]. Early work was formulated as deterministic differential equations models for the transmission called Susceptible-Infectious-Recovered (SIR) by Kermack and McKendrick [30].

The model consists of three differential equations, with three unknown function.

$$\frac{d}{dt}S = -\beta IS\,, \qquad \frac{d}{dt}I = \beta IS - \gamma I, \qquad \frac{d}{dt}R = \gamma I$$

where $\boldsymbol{S, I, R}$ are the numbers of susceptible, infected, and recovered individuals in the population and **t** is the independent variable time. $\boldsymbol{\beta}$ is transmission rate and $\beta IS$ the number of susceptible individuals that become infected per day. $\boldsymbol{\gamma}$ is the recovery rate and $\gamma I$ is the number of infected individuals that recover per day. $1/\gamma$ is the infectious period i.e. the average duration of time an individual remains infected. These formulas above are the simplest version of SIR since we didn't add vital dynamics as death rate and birth rate. The basic reproduction number $\boldsymbol{R_0}$ is an important epidemiologic metric that is used to describe how contagious an infectious disease is [31]. For the SIR model,

$$R_0 = \beta N/\gamma$$

where **N** is the total population size and $N = S + I + R$ is constant. $R_0$ is generally reported as single numeric number or low-high range and It is typically interpreted that if R is greater than 1, the outbreak will continue, and if R is less than one, it will end soon. For instance, as University of Oxford's COVID-19 Evidence Service Team says the disease has an estimated $R_0$ of around 2.63 [32]. That means a person who has the disease will transmit it to an average of 2.63 other people. Since $R_0$ and the infectious period $1/\gamma$ are more intuitive parameters, generally researchers use these as inputs in many epidemiological models. In contrast, in our simulation we directly worked with transmission and recovery rates $(\beta, \gamma)$ in order to reduce the assumptions**.**

Earlier, the biggest obstacle for epidemiology was not being an experimental science. Since the experiments were neither practical nor ethical population studies were having limitations even though the discipline concerns itself with large populations of ill humans. Embracing new powerful computational technologies to analyze, model, and simulate the dynamics of infectious disease has accelerated research in the field of epidemiology in the '90s [33]. Such simulations served as a dry lab where new interventions could be designed, evaluated, and optimized on outbreaks, with many advantages for real-world epidemic prevention and control efforts [27]. The development of this new science leads to a new interdisciplinary, collaborative area that consists of epidemiologists and other computationally-oriented academic disciplines. More recent work of modelling and simulating an epidemic spread classified in two categories: host and spread [34]. While host models investigate the effect of disease on individuals, spread models focus on predicting how the disease spreads among a group of people. In this paper, we created a spread model simulation where we investigate how an infectious disease passes from one individual to another instead of how it affects the host individual. The model will be discussed in the following chapter.

### 2.8 EPIDEMIC SIMULATION WITH RL

Although Reinforcement Learning has been widely studied in the literature there have not been many studies about epidemic spread control with RL. In epidemic outbreaks, particularly this method of machine learning is beneficial since researchers are not limited to real data. In addition, a brute force approach to this problem is computationally intractable and inefficient, since calculating every state -even not useful ones at all- require a large amount of computation power [35]. In lieu of identifying optimal policies, other computational AI-related methods have been used. For instance, Big data is used for estimating the severity of seasonal influenza [36]. Shrimp disease occurrence prediction has been made with neural networks and logistic regression [37]. In parallel, a network-based contact-tracing model has been developed to learn about outbreak propagation in STD's [38]. Even genetics algorithms are used for finding optimal vaccination strategies in influenza [39]. In [2], Yanez has created a baseline for how to design a reinforcement learning environment to represent the problem of epidemics and finding optimal interventions. In her study, agents represent the decision-makers such as governments, health institutions, and the task is finding the optimal intervention strategy in 3 categories: preventive interventions, treatment of disease, and reduce-transmission interventions. The state includes infection rates, reproducibility, etc. besides whether they are susceptible, infected, or recovered. The action set includes mask-wearing, social distancing, contact tracing, closing schools, lockdown, etc. The reward is given related to the death or infection spread rates depending on selection. We abstracted the idea of having an epidemic simulation to physical form and changed the representation of the agents to people. Instead of intervention strategies that governments can take, we have investigated precautions that can be applied by individuals. The action set was also inspired by Yanez [2], but rather than abstractly representing mask-wearing or social distancing, our task was physically showed that agents can create these actions on their own. A recent study in 2018, [40] approach the problem of selecting optimal strategies for influenza as a K-bandit problem, which is a common problem in reinforcement learning [6][41].

To the best of our knowledge, there has been no research recently done on physics-based epidemic simulation with reinforcement learning. Authors in [2] offer approach about how to design RL environments for epidemic spread however it is neither agent-based nor physics-based simulation. Authors in [27] , [33] use ABM simulations without utilizing artificial intelligence whereas, in our epidemic simulation, agents implicitly develop this instinct through RL training and multi-agent cooperation.

## 3 METHODOLOGY

Agents are tasked with cooperating as a team in a physics-based epidemic outbreak. The aim is to stay uninfected as much as possible and not to spread the disease once infected. The healthy agents are tasked to actively avoid each other, and infected ones are tasked to not infect other agents. Before designing a reinforcement learning environment, we started by creating an epidemic simulation that can be used as a base for our deep RL task.

### 3.1 EPIDEMIC SIMULATION IMPLEMENTATION

We have created the environment where the training will take place in Unity Engine with C#. The fundamental assumptions of our simulation are cubes represent agents as a group and indicates a community that lives together. Spreading infection and getting sick only depends on physical proximity between agents. In other words, we simplify the social distancing only to physical form. In addition, agents do not show any symptoms of the disease such as color changes to each other therefore they don’t know each other’s health status.

The environment is a square-shaped area with a wall around so agents cannot go through. Unity ML-Agents package comes with +15 environments and this area was used in one of the experiments.

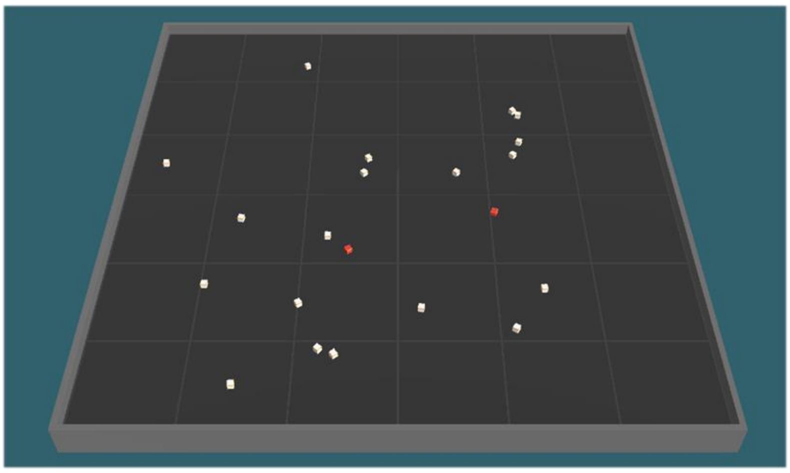


*Figure 4:The Environment and agents. The square shape court limits the area and the cubes are the agents. White ones represent healthy and red ones represent infectious agents. The simulation shows how one infected agent starts to spread the disease over time.*

We used the same 3D model for the area from *Food Collector* [25] which is a competitive multi-agent environment example where agents can shoot each other to get the *goodFood* and avoid *badFood*. The purpose of using very basic shapes like cubes was to avoid any error during the training due to the 3D objects and decrease the complexity of the computation which is also recommended by Unity.

First, we have created hard-coded dummy bots that do not have an artificial neural network instead they are coded to randomly move in the environment. The aim was to create stochastic and partially observable environment. As a part of the environment dummyBots represent individuals who do not take any precautions during an epidemic outbreak and cause the spread of the disease in real-life. We define two-movement style for the dummy bots: target-choosers and bouncers. The simulation based on the physics engine of Unity *PhysX* has some built-in components such as a rigidbody that allows the designer to control an object's position through the physics simulation. There were a couple of options for position manipulation such as adding force, adding velocity, and directly changing the position. For both the movement style of our dummy bots, we choose a directly changing the position to eliminate the complexity of the physics engine. For the bouncer dummy bots, when they collide with an object e.g. wall or another cube, they bounce in the opposite direction such as screen saver logos. The target-choosers choose a random position inside of the area and until they reach there, they try to go in that direction without interruption. When they reach their target, they instantly choose another one. From the designer perspective, this style was more unpredictable and convincing than bouncing cubes until our agents find out a weakness of target-choosers. Turns out the target-choosers don't go to the edges of the arena as much as bouncers since bouncers have to bounce from the walls. Therefore, after our first results came out, we found out our agents learned to hide in corners since it was a blind-spot for target-choosers and converge to suboptimal policy. Thus, we used the combination of two different movement styles in our dummy bots.

The infection mechanism works as follows: every cube has its own sphere-shaped colliders called *infectionColliders*. These colliders have a constant radius denoted as d. At each time step, Unity physics engine checks if any of these colliders touch to each other. If there is an intersection and one of the cubes is sick, the other one is exposed to infection every time step. The expose function gets the distance between agents and calculates the probability of getting infected inversely proportional to the distance. We define P(t) as exposure probability at time t:

$$\mathrm{P(t)} = \frac{\mathrm{d_{max}}}{\mathrm{d_t}}\beta$$

where $0 \leq \mathrm{P(t)} \leq 1$ and $0 \leq \mathrm{d_t} \leq \mathrm{d_{max}}$ .The $\mathrm{d_t}$ is the distance between two individuals at time t and $\mathrm{d_{max}}$ is the maximum distance that infection exposure can happen. $\beta$ is the transmission rate constant as we presented in Chapter 2.7 while explaining SIR curves. At each step, the exposure probability is calculated and there is a random chance that agents can get infected. Since every step P(t) is calculated the chance of getting infected increases if the collision exists longer.

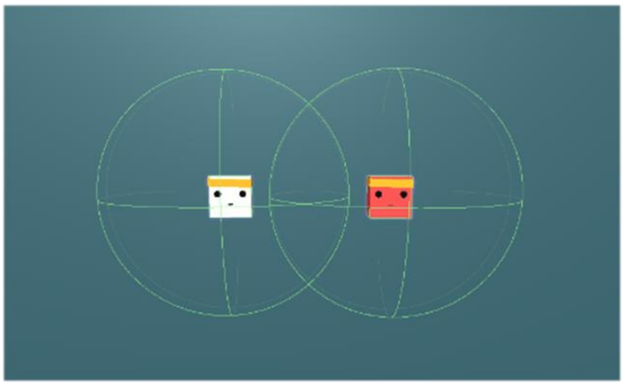


*Figure 5: Two cubes with their visible sphere colliders. One is infected and the other one is at risk to get infected every time step. The risk is getting higher if the distance between the two is closed.*

The agent is created as a cubic object with blue color. We choose a different color for the dummyBots for distinguishing whether it is controlled by a model or behaving randomly. The observation set of the agent was local velocity at x and y-axis and it's health status. Observation vector is stacked with 5 previous observations before being fed to the neural network. The number of stacked vectors parameter changes how many sets of observations into the past you'd like to stack. Increasing this allows the agent to "see" further into the past. In our case 5 steps back was enough. Stacking observations can be seen as a memory enhancement however it shouldn't be confused with Long Short Term Memory (LSTM), which is not implemented in our task yet, but it is mentioned in the future work.

The agent also has 360-degree LIDAR observations, as seen in the below figures. The agent has 16 rays with a sphere on their end. The agent's *infectionCollider* is masked out therefore other agents' rays cannot hit these colliders. Instead, they can only hit the body of the agent itself and walls. During observations, these rays are cast into the physics world, and the objects that hit determine the observation vector that is produced as a list of floats. The sphere radius increases the chance of the rays to collide with an object. When a ray hit an object, the ray's color turns to a red and fed the neural network with what type of object did it hit as an input. We gave different tags to objects and we split it into 3 different object types: walls, agents-bots, and reward cube. The reward cube tag only used in single-agent scenarios and removed in the multi-agent environment. In our case, there was 165 LIDAR sensor observations for single-agent scenarios and 132 in multi-agent scenarios.

As the output of the neural network, agents have 2 types of discrete actions set that can be chosen simultaneously at each time step. The first action set consist of 3 actions which is for moving in x and y-axis. The agent may choose to move forward, backward, or not move. The

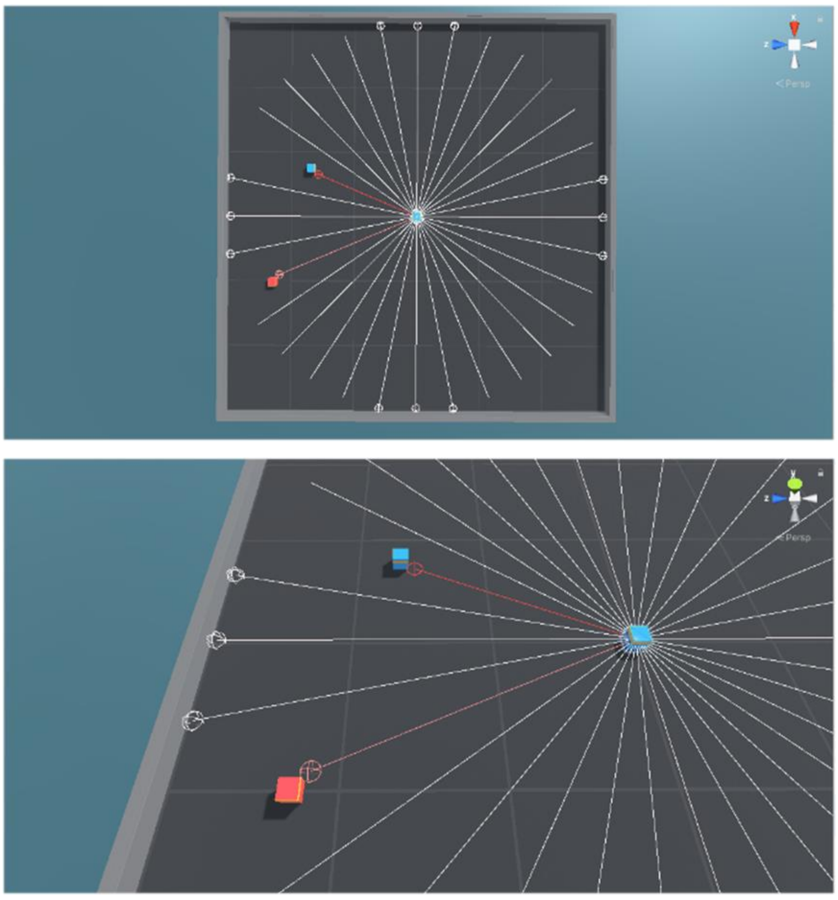


*Figure 6: LIDAR observations of an agent. 16 rays provide a 360-degree theoretical vision. However, this is not the optimal way of seeing since there could be a small object close to the agent which does not hit the spheres on the end of rays thus agents cannot see them. For optimizing the training 16 rays were enough for our task.*

second action set consist of 2 actions which is for rotating clockwise or counterclockwise. We did not implement "do not rotate" action due to decrease complexity of action space. Moreover, the agent was able to compensate for rotation with constant shifting. Since the action space was discrete the agents have constant speed while moving however shifting continuously the direction give them the opportunity of set their speed. For testing action space, a heuristic mode is also implemented.

### 3.2 POLICY OPTIMIZATION

The agent's policy is trained using Proximal Policy Optimization algorithm (PPO). The performance of the PPO has been compared with an off-policy algorithm called Soft-Actor Critic (SAC). Even though SAC's results were close to PPO, it was taking approximately 10 times slower to train. Having much slower training took away the sample efficiency advantage of the SAC since PPO was completing many more steps at the same time. Even in some of the runs, SAC was getting stuck due to the lack of performance of our local machines. Therefore, we decided to move on with PPO.

The core purpose behind the PPO is to strike a balance between easy implementation, sample efficiency, and ease of tuning. PPO provides these advantages by being an on-policy stochastic policy gradient method. A general policy optimization method usually starts by defining the policy gradient loss as:

$$L^{PG}(\theta) = \hat{E}_t[\log\pi_\theta(a_t|s_t)\hat{A}_t]$$

where $\pi_\theta(a_t|s_t)$ is a stochastic policy and $\hat{A}_t$ is an estimator of the advantage function at timestep t. It can be seen appealing to perform many updates for optimizing the loss function using the same observations however it may lead to destroy the current policy with large policy updates. To prevent this destructive behavior researchers developed an algorithm called Trust Region Policy Optimization (TRPO) before PPO. In fact, PPO was developed after TRPO to decrease the complexity of the algorithm. In TRPO the policy updates have a limited effect on the policy

By using a clipped surrogate objective while retaining similar performance of other policy gradient methods PPO smoothly converged to an optimal solution. More details of how the PPO works can be found in Appendix A.

At execution time, each agent act by using only their own observations, and at optimization time, we use all agents' observations to update our policy. So even though 20 different environments are used during training there was only one neural network as an output. In other words, agents share the same policy parameters but act and observe independently as each of them was in different states.

Furthermore, as training continues entropy value decreases. The entropy of a stochastic variable that is driven by an underlying probability distribution is the average amount of bits that

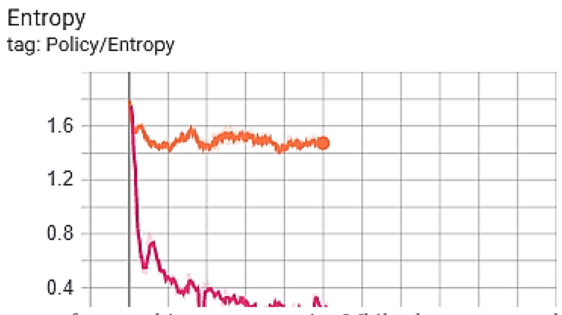


*Figure 7: Two example runs from multi-agent scenario. While the entropy value corresponds to red decreases as expected run corresponds to orange stays nearly constant. This means agents could not find a meaningful policy at all in 10 million steps therefore we stopped the run before it finishes.*

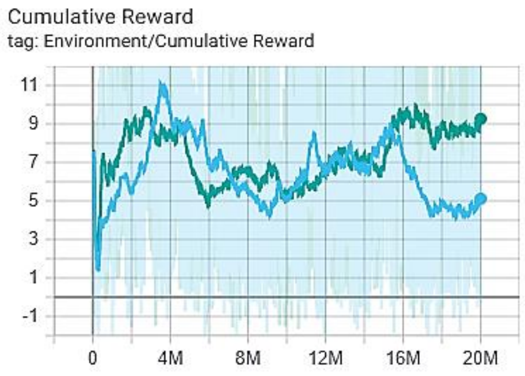
are needed to represent its outcome. In other words, it is a measure of how unpredictable the environment is. As the agent explores the possibilities in the simulation entropy value falls and the agent becomes more confident about its decisions. In a successful run, entropy value should slowly decrease as shown in the next figure.

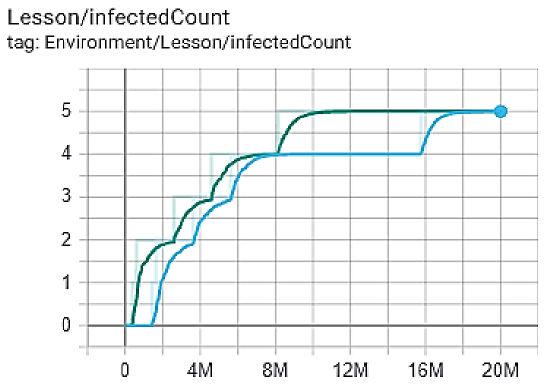
## 3.3 OPTIMIZATION SETUP

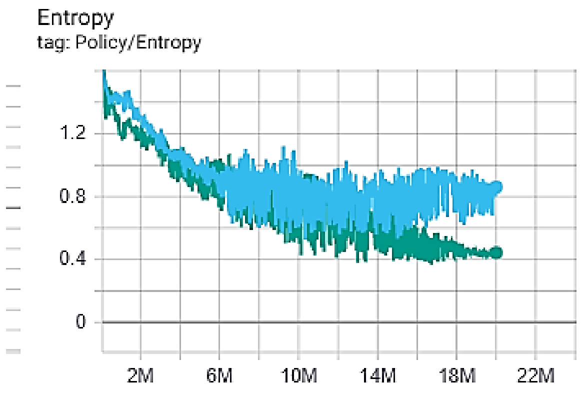
Training is performed using the Unity Engine. We took a build of our training scene which can be used as executable. We use Linux executables on Amazon Web Service (AWS) Cloud instances and windows executables on our local machines. Using executable decreased the run time since the code was compiled. We have created 3 different computed optimized CPU instances with different capacities, one free tier with the lowest system requirements and 1 CPU unit t2.micro, one with 8 CPU units c5.xlarge, and one with 16 CPU units c5.4xlarge. Note that the *c* in front of the instance types represent that they are instances which are computing optimized. We have implemented automation for running the simulation from the terminal. With this automation, it was possible to run 96 simulations in parallel to find the best hyperparameters. The total simulation time was more than 2000 hours. However, these were not only for one run, we tried many different parameters, neural network configuration, and environment settings in order to find the best fit. The maximum training lasted 30 hours. The highest score was obtained with 2048 batch size and our buffer size was 10 times more than our batch size as 20480. In every 500k steps, we saved our neural network model in order to avoid any possible error and create safe checkpoints.

### 3.3.1 Optimization Hyperparameters

We observed batch size as the most sensitive parameter to tune. Also having a higher batch size doesn't always equal to better convergence as we tried both 2048 and 4096 and find out 2048 was better for our RL task. Adding batch normalization to the neural network increased the score significantly and reduced the computation time visibly. To increase the stability of the neural network, batch normalization normalizes the output of the previous







activation layer by subtracting the batch mean and dividing by the batch standard deviation [42]. In the following figure 15 we presented two similar runs with only one difference, the batch normalization. As training continues the normalized run which corresponds to green, passes the lessons faster and in the end converges a much better score than the run without batch normalization which is shown as in blue color. The drops in the cumulative reward graph correspond to lesson changes in the Lesson/InfectedCount graph. Also, the bottom graph show entropy changes during training. For a just changing the normalization factor we had surprised with the effect in results.

Our optimization hyperparameter settings are as follows:

| | |
|---|---|
| Buffer size | 20480 |
| Batch size | 2048 |
| Learning rate | 0.0003 |
| Beta | 0.005 |
| Epsilon | 0.2 |
| Lambda | 0.95 |
| Number of epochs | 6 |
| Learning rate schedule | Linearly decreasing |
| Extrinsic Reward Gamma | 0.99 |
| Extrinsic Reward Strength | 1 |
| Intrinsic Reward Gamma | 0.99 |
| Intrinsic Reward Strength | 1 |

### 3.3.2 Policy Architecture Details

| | |
|---|---|
| Input Layer Shape (Multi-Agent) | (-1,1,1,152) |
| Action Shape (Multi-Agent) | [-1,1,1,5) |
| Size of Hidden Layer | 512 |
| Number of Hidden Layers | 2 |
| Batch Normalization | True |

*Figure 8: Using the same hyperparameters except for batch normalization. The green line corresponds to the training with normalization and the blue line corresponds to without normalization. Th left graph presents the visible difference in score convergence between two runs. Right graphs show at what step did they pass their lessons. As the graph illustrates the green run passes the lessons faster and having more sample efficiency. The bottom graph shows how normalization helps to decrease the entropy and the noise of it. While the green line converges to a smaller value and its noise decreases in the last part of the training the blue run's entropy converges much higher value and its noise stays constant. A lower entropy value indicates more confidently taking decisions.*

## 3.4 CURIOSITY-DRIVEN LEARNING

Another addition to our training was adding intrinsic motivation. In standard reinforcement learning algorithms, the reward function is defined by the researcher. Thus, rewards are extrinsic to the agent. However, having hand-designed, dense rewards is not scalable for most of the RL tasks. Curiosity is a type of intrinsic reward function which uses prediction error as reward signal. This intrinsic reward enables the agent to explore its environment and learn skills that might be useful later in its life. To create a hybrid reward function, we combined curiosity and extrinsic rewards. The implementation that we used is coming from [43] by Deepak Pathak and his colleagues at Berkeley. Let intrinsic curiosity reward at time t denoted as $R_t^i$ and extrinsic reward denoted as $R_t^e$. The policy is trained to maximize the sum of these rewards as

$$R = \sum_{t=0}^{H} \gamma^t \mathrm{R}_t^{extrinsic} + \gamma^t \mathrm{R}_t^{intrinsic}$$

where H is episode length and γ is a time discounting factor that biases agents toward choosing short term rewards. We determine the γ factor as 0.99 for both extrinsic and intrinsic rewards. See Appendix B for more details of how curiosity is calculated.

Formulations of intrinsic reward can be divided into two categories: 1)encouraging the agent to explore unknown states and 2) encouraging the agent to perform actions that reduce uncertainty in the agent's ability to predict the consequence of its own actions [43]. Intrinsic rewards are created based on how hard they are for the agent to predict the consequences of its own actions. However, in high-dimensional continuous state space environments like ours, it is hard to build prediction models. For this prediction, an addition challenge also lies in dealing with stochasticity of the environment. To deal with these challenges, researchers ignored changes in the environment that could possibly not due to the actions of the agent while predicting consequences of actions.

Instead of using raw observation data they created feature space vectors where only the information relevant to the action performed by the agent is represented. The algorithm which is proposed by Deepak Pathak and his colleagues was such a feature space can be learn by using two deep neural networks. Inverse DNN is for encoding the raw observation to feature space and the forward DNN is for taking these encodings and predicting the actions taken by the agent to move from one state to another. See Appendix B for more details.

The curiosity especially works when there are not relatively dense rewards in the environment. In our case the rewards are sparse except the survival bonus. Even with all extrinsic rewards the environment had the lack of feedback. Even though adding curiosity increased the needed computation power and the training time, in the end, the cumulative reward compared to just using extrinsic reward is increased and the policy that the agent were using converged on a much better level.

## 3.5 CURRICULUM LEARNING

Curriculum Learning is a type of learning where agents start with simple tasks and gradually the task complexity is increased [44]. It is closer to how humans learn complex tasks. We used curriculum learning and compared with standard training. Agents who trained with curriculum learning did not also have the advantage of sample efficiency but also, they converged a better policy. The technique is also more sophisticated in a way of dividing the RL task to sub-tasks and teach one by one. ML-Agents have curriculum learning functionality as built-in as shown in the below diagram. The only change was setting the lessons for agents. We define 5 lessons and

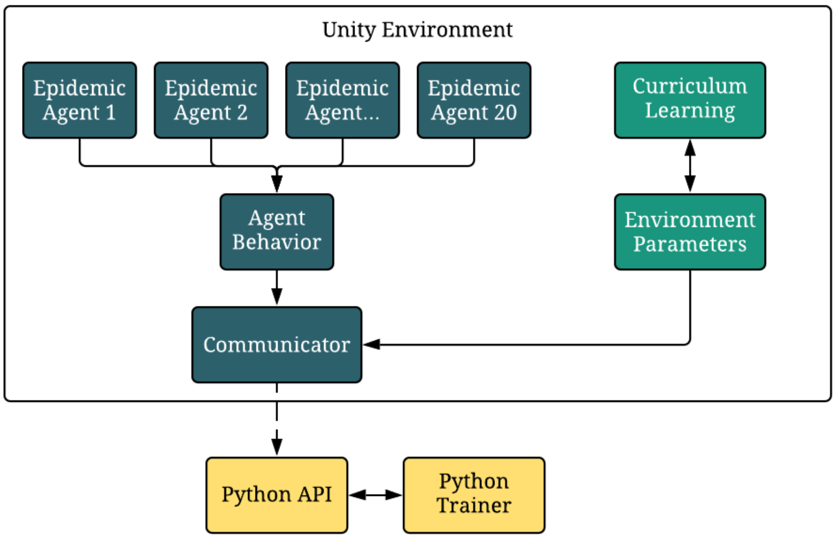


*Figure 9: Ml-agents Architecture with Curriculum learning and environment parameters randomization*

increased the infected bot counts with every lesson. In the first one, there were only 2 infected bots and we set a reward limit for them. If their cumulative reward passes the limit 100 times in a row, then they pass the lesson and the task complexity is increased. While training with curriculum learning, there were drastic decreases when lessons are changed but PPO recovered quickly from drops and return its former state with a more generalized policy which is a plus for us.

### 3.6 SINGLE-AGENT SCENARIO

We first started with a single-agent scenario where the agent is alone with other 1-20 infected dummy bots. The maximum time step of an episode is defined as 10000 steps. For each time step that the agent is alive, we gave 0.01f survival bonus. For getting infected we gave -1 and finished the episode. If the agent survives till the end, its cumulative reward would be 10. To avoid suboptimal behaviors such as hiding corners we took some precautions. A shiny yellow-colored cube has been spawned on the environment. This encouraged our agent to dive in group of infected bots and collect the reward instead of waiting for its death in the corner. As an extra help for the agent, we gave the distance and direction vector in space as an observation. Also, the number of detectable tags were increased by 1 with a reward cube tag. Also, we ended episodes when our single-agent got infected since there is no point to continue learning. Through simulation, the episode lengths increased.

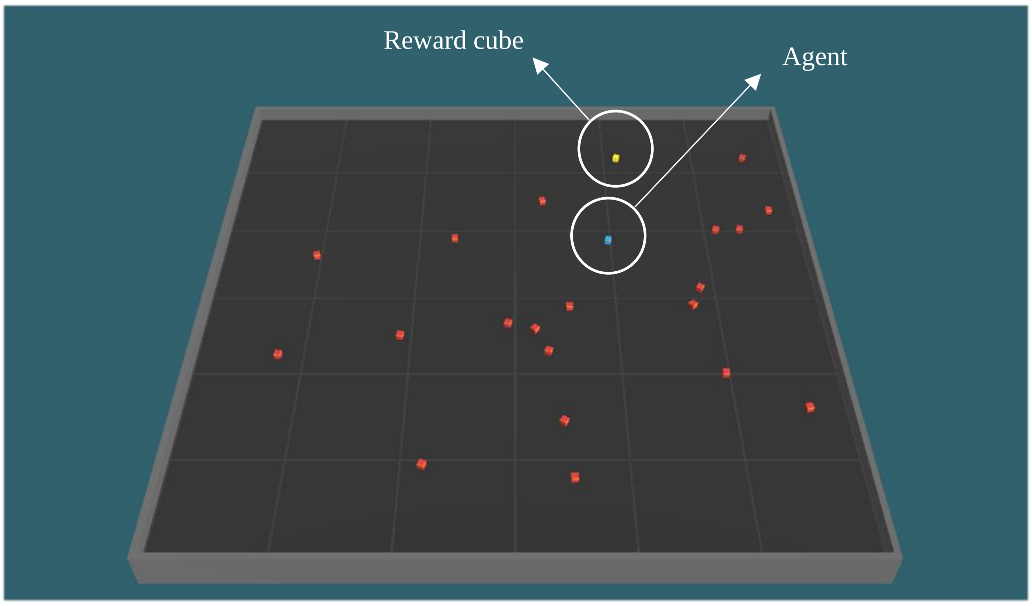


*Figure 10: A single-agent scenario where the agent tries to avoid infected bots and collect the reward. The reward spawns randomly when agents collide with it. Each collection of reward cube counts as +1 reward.*

The spawning mechanism of the infected bots was creating some problem since they were spawning randomly at the beginning of episodes and if they spawn too close to agents, they directly infect agents, without giving any time to them. Therefore, we updated the mechanism. Firstly, bots will freeze for 1 second just after the episode starts which saves some time for agents to run. Secondly, we separated the spawning areas of agents and infected bots. Bots spawning in the inner square with half of the arena's range and agents spawning between the inner square and the arena's walls. We aim to give a wake-up time for agents in the environment which visibly decreased our noise.

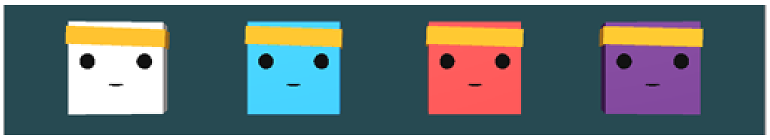


*Figure 11 Different states of the agents. Through the simulation, the agent's status of health changes. To represent the change, we used 4 different colors. a) White Bots indicates that the agent is not controlled by a brain. It only has simple hard-coded actions such as directly going targeted locations or bouncing from the walls. This represents individuals in a community who are not acting logically. b) Blue Bots are agents with a brain that controls them. c)Red Bots indicates whether with brain or not the bot is infected. d)Purple indicates that the agent is not infectious anymore. In SIR models' purple agents call as Recovered-Removed. Please see [https://github.com/Hsgngr/Pandemic_Simulation](https://github.com/Hsgngr/Pandemic_Simulation) for example videos.*

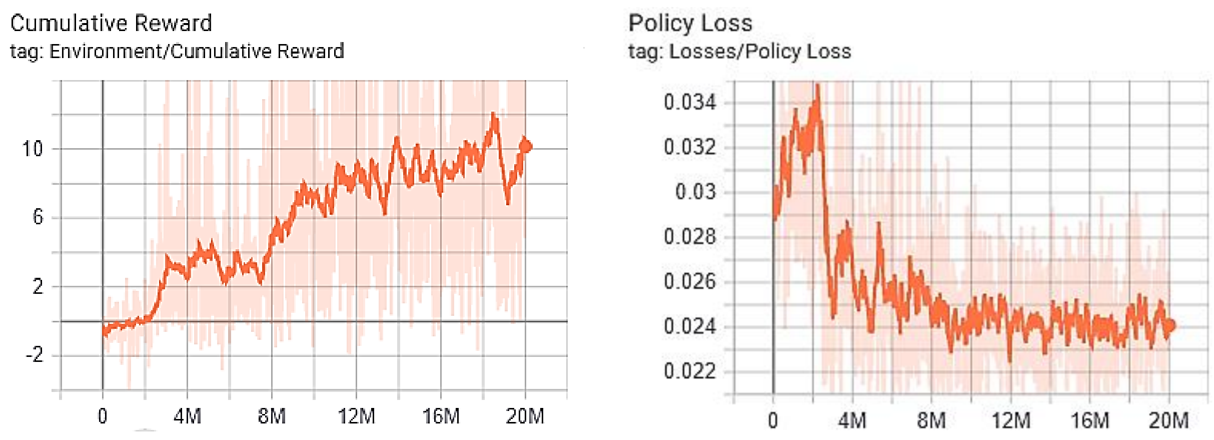


*Figure 12: Left: Cumulative Reward / Step Graph. In 20 million steps the agent learned how to avoid infected bots and collect rewards. Right: Policy Loss / Step. As episodes pass, the loss function decreases as it is supposed to be in successful trainings.*

The single-agent scenario was successful in a way that the agent learned how to maximize its cumulative rewards by collecting reward cubes and avoiding infected bots. As shown in the above graphs at the end of the 20 million steps, agent increased its reward from -1 to 10 in average. The policy loss correlates to how much the policy (process for deciding actions) is changing. The magnitude of this should decrease during a successful training session as it did. It was a design choice to start with a single-agent and eliminating multi-agent complexity at first for improving the experiment easier. We made a number of changes on the way. We optimized the training by changing the decision interval between steps. Agents decide a new action set at every 5 steps. This optimization worked since steps are minuscule and there was no difference in the cumulative reward, but training time is reduced after this update.

### 3.7 MULTI-AGENT SCENARIO

After getting successful results from single-agent training we migrate to the multi-agent scenarios. Unity Engine made migration extremely easy for researchers. There were only a few changes in the setting. We gather agents into one single area instead of separate areas next to each other. We changed the mechanism of ending episodes. The episodes were finishing once the agent got infected since now there are multiple agents on the environment the episode continues until there is no agent left uninfected. We also removed the reward cube, it's distance and direction observations, and the detectable tag on raycasts from the environment as it is not required in a multi-agent scenario. The total observations except raycasts decreased to 4 from 7. The raycast observation vector count decreased from 195 to 152. After a few optimizations and test runs, the multi-agent training was ready and working successfully. In fact, some stunning results came out of it. In addition to default graphs, we define our own parameters to observe the simulation better. One of them was collision counts of healthy agents. Since we aim for social distancing, we would like to see a decrease in the count of collisions between healthy agents while the training continues.

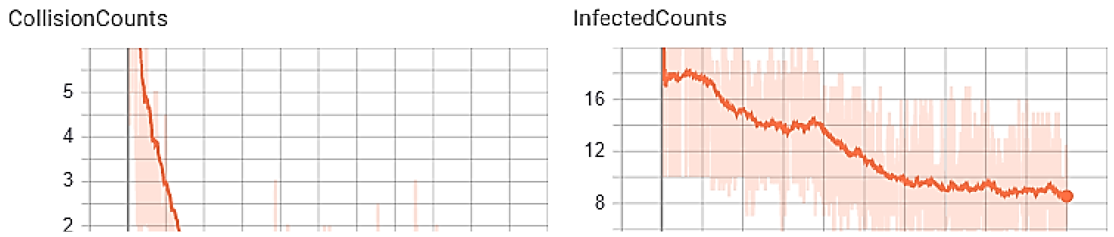


*Figure 13: Left: Number of Collisions/Step graph shows agents understand physical proximity cause them to lose rewards therefore they decreased that to almost 0. Right: Infected Counts/ Step graphs. In every run, there were 20 agents and at the start almost everyone got infected. As the training continues the number of infected agents also decreases. This means the standard deviation of the rewards decreased and the agent got more stable.*

In both single and multi-agent scenarios the training architecture was the same as shown in figure 12 below. We used multiple agents to feed one single-brain model whether it is used in single or many agents. The optimal environment number was 20 in parallel in our local machine and we used CPU for computation power. The ML-Agents package was built in python it serves with an API. The Unity game engine communicates with Python Trainer via a communicator in the Unity side and an API in the python side.

## 4 EVALUATION

In the previous sections, we presented how we design our agents, environment, and whole simulation. We show evidence that our agents learn and gradually become better in social distancing as infected counts and collision counts between agents decreasing and cumulative reward increasing throughout the training. However, tracking rewards is an insufficient evaluation metric in multi-agent settings as it can be ambiguous whether agents are improving or stagnated since these graphs do not illuminate at what circumstances agents lose reward. In epidemic diseases, SIR graphs can be used as evaluation metrics since these graphs can give better insights about epidemics such as when the disease peeks or how long it takes to eradicate it. In section 4.1 we first qualitatively compare the behaviors learned in epidemic simulation by displaying some screenshots of how agents line up in order to increase the distance between each other from the training. In section 4.2, we then test our agents in a much bigger area with many more infected bots and quantitatively measure and compare agent capabilities with SIR curves.

### 4.1 SELF-ISOLATION

We tried many environment settings, different initializations, and observations for the agents. In one of the settings, we provided the agent's health status as observation to itself. Therefore, they knew when they got infected, but they did not have any information about other agents. As we mentioned in section 3.1 there were 3 different health statuses as susceptible, infected, recovered. After a few million iterations agents found self-isolation to protect the others as shown in the figure below.

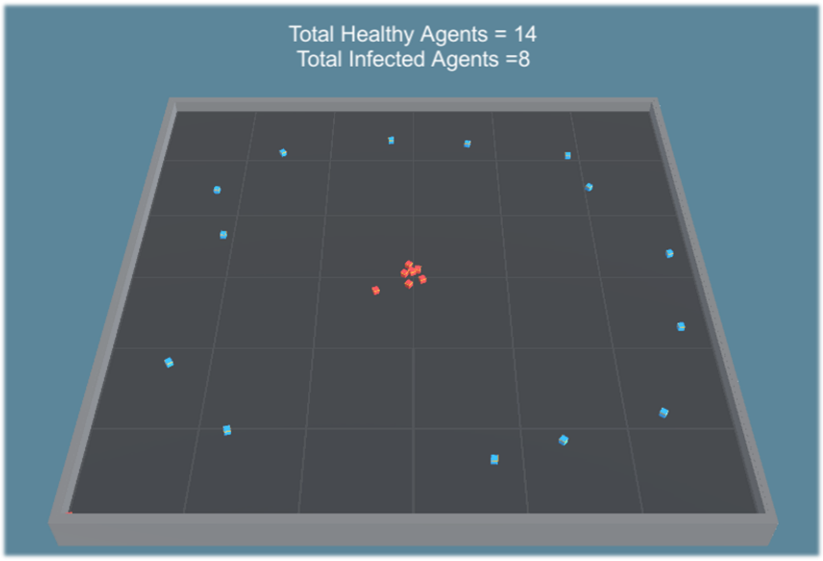


*Figure 14: Agents learned to gather in the middle to decrease the risk of infecting others. This was surprising for us since agents do not know which agent is sick or healthy.*

This was a surprising development for us since it indicates that a simple communication between agents has been evolved to protect each other. By giving the health status observation to each agent, they realized that they can use this observation to behave differently even though all of them follow the same policy. This behavior is also a proof that homogeneous agents can learn to act heterogeneously due to the development of sub-behavior that differs based on the agent's health status as its mentioned in section 2.4.1 Infected agents learn to self-quarantine themselves together with other infected agents and healthy ones learn to stay away from everybody in a circular shape. Looks like it is the mathematically the best way to be away from each other.

## 4.2 SIR GRAPH EVALUATION

We used SIR graphs to evaluate our trained agents' performance on an epidemic outbreak. Trained agents put in a testing area which is 10 times bigger than the training area with a total of 2000 individuals. First, we run the simulation and export the graph of the result to find out what happens if all agents were bot and act randomly. Then we decrease the bot count step by step 200,500,1000 respectively with trained agents by holding the total individual count constant. We compared the SIR graphs of the results.

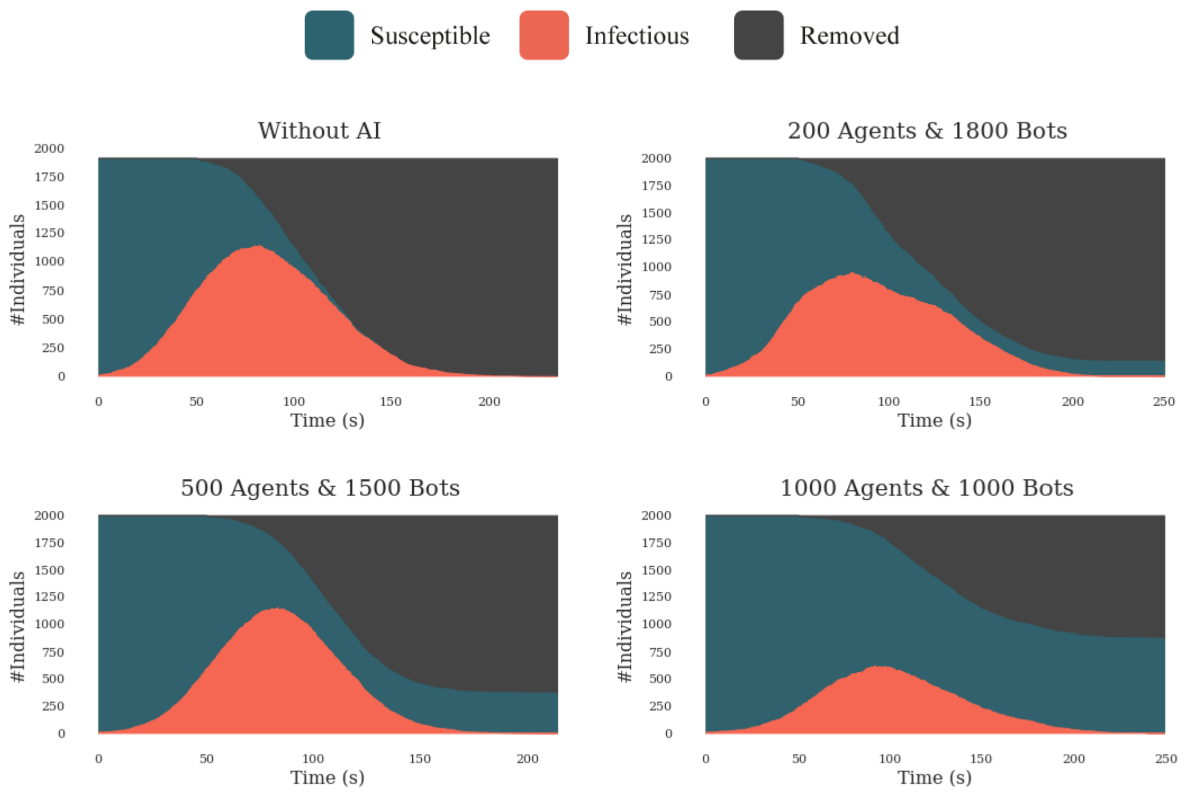


*Figure 15: Our SIR graph results with and without artificial intelligence agents. All simulations run with 2000 individuals. In the upper left corner, the graph shows what happens if there are no AI agents in the environment. We gradually increased the number of agents in our experiment to show the effect of our agents to the epidemic simulation. As the number of agents increase the curve gets flatter and the total infected people decrease. The area shows the whole population and their ratios explicitly.*

As shown in figure 17 above, without adding the AI element into the epidemic simulation everybody got infected. By adding even, a 10% trained agents curve changed but a very small number of individuals survived the outbreak. By increasing the ratio of agents in the environment, people who survived without getting infected increased significantly. These results were obtained with multiple trials and these are the average results. Sometimes as shown in the graph with 500 agents and 1500 bots, the disease could not be controlled and spread as fast as possible. But adding agents significantly decreased the chance of getting infected and just by learning physics-based social distancing and self-quarantine.

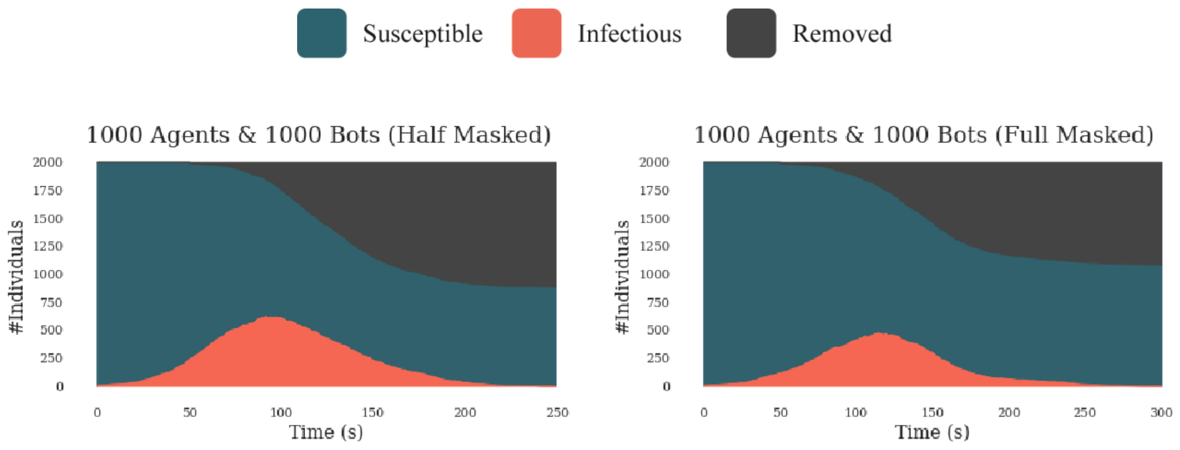


*Figure 16:Left graph shows the SIR graph results if half of the individuals wear a mask and, on the right, the graph illustrates if all of the individuals wear a mask.*

In addition, we formalized mask-wearing as a decrease in expose radius of individuals. Therefore, we run 2 more simulations to show the effect of mask-wearing. As figure 18 presents by wearing masks more individuals survive from the epidemic outbreak.

## 5 DISCUSSION AND FUTURE WORK

We have demonstrated that an epidemic simulation with a simple infection mechanism, multi-agent cooperative environment, and standard reinforcement learning algorithms at scale can induce agents to learn complex strategies and human-like behaviors. We observed many strategies from social distancing to self-quarantine that agents developed suggesting that it is possible to flatten the SIR curve by taking individual precautions in an epidemic outbreak.

Our results with epidemic simulation should be viewed as a proof of concept showing an agent-based simulation with reinforcement learning can be used to assist decision-makers during the epidemic. We acknowledge that the strategy space in this environment was limited and likely will not converge a better strategy for agents, however with this proof of concept the epidemic simulation that we build is grounded and very scalable.

In order to create more accurate and interesting environments, we have some future ideas. We are working on creating an environment that pushes agents to collect rewards to not starve to death and we provide a safe place for them where it's epidemic free. Agents should find an optimal behavior between staying safe at house and not starving to death. Therefore, they sometimes need to take the risk. We believe it will be a better the representation of current situation with COVID-19 since all individuals need to work and risk is inevitable.

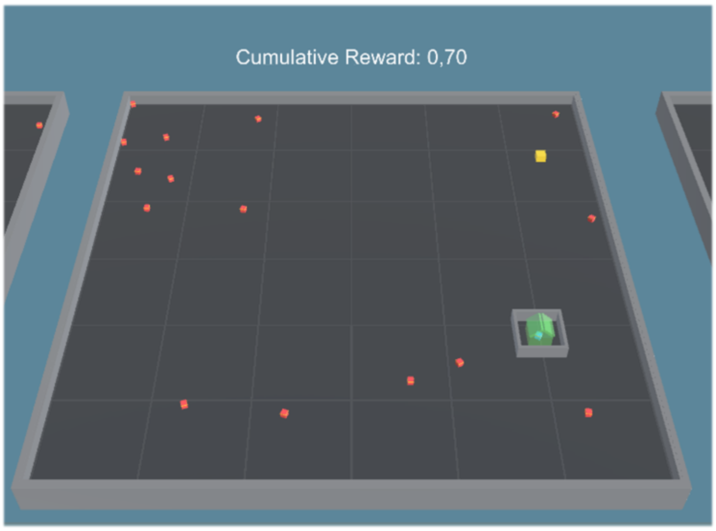


*Figure 17: Future work for epidemic simulation RL environment. The green house has walls when the agent locks it and infected individuals cannot go in. However, the agent who locked itself also needs to go out to get the yellow reward in order to survive.*

## 6 REFERENCES

[1] N. S. Punn, S. K. Sonbhadra, and S. Agarwal, “COVID-19 Epidemic Analysis using Machine Learning and Deep Learning Algorithms,” *medRxiv*, p. 2020.04.08.20057679, 2020.

[2] A. Yañez, C. Hayes, and F. Glavin, “Towards the control of epidemic spread: Designing reinforcement learning environments,” *CEUR Workshop Proc.*, vol. 2563, pp. 188–199, 2019.

[3] B. Baker *et al.*, “Emergent Tool Use From Multi-Agent Autocurricula,” 2019.

[4] V. Francois-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau, “An Introduction to Deep Reinforcement Learning,” *Found. trends Mach. Learn.*, vol. II, no. 3–4, pp. 1–140, Nov. 2018.

[5] O. Bent, S. L. Remy, S. Roberts, and A. Walcott-Bryant, “Novel Exploration Techniques (NETs) for Malaria Policy Interventions,” *32nd AAAI Conf. Artif. Intell. AAAI 2018*, pp. 7735–7740, Dec. 2017.

[6] R. S. Sutton and A. G. Barto, “An Introduction to Reinforcement Learning,” Cambridge University Press, 2018, pp. 1–352.

[7] M. Riedmiller *et al.*, “Learning by Playing - Solving Sparse Reward Tasks from Scratch,” *35th Int. Conf. Mach. Learn. ICML 2018*, vol. 10, pp. 6910–6919, Feb. 2018.

[8] Y. Burda, A. Storkey, T. Darrell, and A. A. Efros, “Large-scale study of curiosity-driven learning,” *7th Int. Conf. Learn. Represent. ICLR 2019*, 2019.

[9] J. Achiam, “Spinning Up in Deep Reinforcement Learning,” *SpinningUp2018*, 2018.

[10] K. Zhang, Z. Yang, and T. Başar, “Multi-Agent Reinforcement Learning: A Selective Overview of Theories and Algorithms,” pp. 1–72, Nov. 2019.

[11] OpenAI *et al.*, “Dota 2 with Large Scale Deep Reinforcement Learning,” Dec. 2019.

[12] L. Busoniu, R. Babuska, and B. De Schutter, “A Comprehensive Survey of Multiagent Reinforcement Learning,” *IEEE Trans. Syst. Man, Cybern. Part C (Applications Rev.*, vol. 38, no. 2, pp. 156–172, Mar. 2008.

[13] J. Z. Leibo, V. Zambaldi, M. Lanctot, J. Marecki, and T. Graepel, “Multi-agent reinforcement learning in sequential social dilemmas,” *Proc. Int. Jt. Conf. Auton. Agents Multiagent Syst. AAMAS*, vol. 1, pp. 464–473, 2017.

[14] J. W. Lee, J. Park, J. O, J. Lee, and E. Hong, “A multiagent approach to Q-learning for daily stock trading,” *IEEE Trans. Syst. Man, Cybern. Part ASystems Humans*, vol. 37, no. 6, pp. 864–877, 2007.

[15] S. Padakandla, P. K. J, and S. Bhatnagar, “Reinforcement Learning in Non-Stationary Environments,” *Appl. Intell.*, May 2019.

[16] P. Hernandez-Leal, B. Kartal, and M. E. Taylor, “A Survey and Critique of Multiagent Deep Reinforcement Learning,” *Auton. Agent. Multi. Agent. Syst.*, vol. 33, no. 6, pp. 750–797, Oct. 2018.

[17] L. Panait and S. Luke, “Cooperative Multi-Agent Learning: The State of the Art,” *Auton. Agent. Multi. Agent. Syst.*, vol. 11, no. 3, pp. 387–434, Nov. 2005.

[18] R. F. Denison and K. Muller, “The evolution of cooperation,” *Scientist*, vol. 30, no. 1, 2016.

[19] P. Stone and M. Veloso, “Multiagent Systems : A Survey from a Machine Learning Perspective 1 Introduction 2 Multiagent Systems,” *Auton. Robots*, vol. 8, no. 3, pp. 345–383, 1997.

[20] E. H. Durfee, V. R. Lesser, and D. D. Corkill, “Coherent Cooperation Among Communicating Problem Solvers,” *IEEE Trans. Comput.*, vol. C–36, no. 11, pp. 1275–

1291, Nov. 1987.

[21] M. Johnson, K. Hofmann, T. Hutton, and D. Bignell, “The malmo platform for artificial intelligence experimentation,” *IJCAI Int. Jt. Conf. Artif. Intell.*, vol. 2016-Janua, pp. 4246–4247, 2016.

[22] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, “The Arcade Learning Environment: An Evaluation Platform for General Agents,” *J. Artif. Intell. Res.*, vol. 47, pp. 253–279, Jul. 2012.

[23] G. Brockman *et al.*, “OpenAI Gym,” pp. 1–4, Jun. 2016.

[24] Y. Song *et al.*, “Arena: A General Evaluation Platform and Building Toolkit for Multi-Agent Intelligence,” no. 1, May 2019.

[25] A. Juliani *et al.*, “Unity: A General Platform for Intelligent Agents,” pp. 1–28, Sep. 2018.

[26] B. C. Perley, “Programming Language Maintenance,” in *Defying Maliseet Language Death*, UNP - Nebraska, 2017, pp. 63–84.

[27] E. C. Mark S. Smolinski, Margaret A. Hamburg, and Joshua Lederberg, on E. M. T. to H. in the 21st C. B. on Global, and H. I. of Medicine, *Microbial Threats to Health*. Washington, D.C.: National Academies Press, 2003.

[28] B. Zeigler, A. Muzy, and L. Yilmaz, “Artificial Intelligence in Modeling and Simulation,” in *Encyclopedia of Complexity and Systems Science*, New York, NY: Springer New York, 2009, pp. 344–368.

[29] B. Osiński *et al.*, “Simulation-based reinforcement learning for real-world autonomous driving,” 2019.

[30] Robert E. Serfling, “Summary for Policymakers,” in *Climate Change 2013 - The Physical Science Basis*, vol. 53, no. 9, Intergovernmental Panel on Climate Change, Ed. Cambridge: Cambridge University Press, 2019, pp. 1–30.

[31] P. L. Delamater, E. J. Street, T. F. Leslie, Y. T. Yang, and K. H. Jacobsen, “Complexity of the Basic Reproduction Number (R 0 ),” *Emerg. Infect. Dis.*, vol. 25, no. 1, pp. 1–4, Jan. 2019.

[32] J. K. Aronson, J. Brassey, and K. R. Mahtani, “‘When will it be over?’: An introduction to viral reproduction numbers, R0 and Re - CEBM,” *Cent. Evidence-Based Med.*, 2020.

[33] J. S. Koopman, “Emerging objectives and methods in epidemiology.,” *Am. J. Public Health*, vol. 86, no. 5, pp. 630–632, May 1996.

[34] M. Shatnawi, S. Lazarova-Molnar, and N. Zaki, “Modeling and simulation of epidemic spread: Recent advances,” *2013 9th Int. Conf. Innov. Inf. Technol. IIT 2013*, no. March, pp. 118–123, 2013.

[35] W. J. M. Probert *et al.*, “Context matters: using reinforcement learning to develop human-readable, state-dependent outbreak response policies,” *Philos. Trans. R. Soc. B Biol. Sci.*, vol. 374, no. 1776, p. 20180277, Jul. 2019.

[36] L. Simonsen, J. R. Gog, D. Olson, and C. Viboud, “Infectious disease surveillance in the big data era: Towards faster and locally relevant systems,” *J. Infect. Dis.*, vol. 214, no. Suppl 4, pp. S380–S385, 2016.

[37] P. Leung and L. T. Tran, “Predicting shrimp disease occurrence: artificial neural networks vs. logistic regression,” *Aquaculture*, vol. 187, no. 1–2, pp. 35–49, Jul. 2000.

[38] K. T. D. Eames and M. J. Keeling, “Contact tracing and disease control,” *Proc. R. Soc. B Biol. Sci.*, vol. 270, no. 1533, pp. 2565–2571, 2003.

[39] R. Patel, I. M. Longini, and M. E. Halloran, “Finding optimal vaccination strategies for pandemic influenza using genetic algorithms,” *J. Theor. Biol.*, vol. 234, no. 2, pp. 201–212, 2005.

[40] P. Libin *et al.*, *Machine Learning and Knowledge Discovery in Databases*, vol. 11053, no. June. Cham: Springer International Publishing, 2019.

[41] T. Alamo, D. G. Reina, and P. Millán, “Data-Driven Methods to Monitor, Model, Forecast and Control Covid-19 Pandemic: Leveraging Data Science, Epidemiology and Control Theory,” pp. 1–65, Jun. 2020.

[42] S. Ioffe and C. Szegedy, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift,” *32nd Int. Conf. Mach. Learn. ICML 2015*, vol. 1, pp. 448–456, Feb. 2015.

[43] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell, “Curiosity-Driven Exploration by Self-Supervised Prediction,” in *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2017, vol. 2017-July, pp. 488–489.

[44] S. Narvekar, B. Peng, M. Leonetti, J. Sinapov, M. E. Taylor, and P. Stone, “Curriculum Learning for Reinforcement Learning Domains: A Framework and Survey,” pp. 1–47, 2020.

# 7 APPENDIX

## 7.1 APPENDIX A: Proximal Policy Optimization Details

Policy Gradient Methods

### 7.1.1 Notation

We consider the standard multi-agent reinforcement learning formalism of $N$ agents interacting with each other in an environment and inspired form OpenAI' s *Emergent Tool Use From Multi-Agent Autocurricula* paper [3]. The interaction is defined by a set of state $S$ describing the agents' configuration, a set of observations $O^1 .... O^n$ of all agents, a set of actions $A^1 ... A^n$ of all agents In our formulation, there is only one on-policy $\pi_{\theta i} = \pi_\theta$ which keeps improving with the parameter $\theta$ and all agents use the same policy. Each agent aims to maximize its total expected discounted return $G(t)$. The action-value function is defined as

### 7.1.2 Proximal Policy Optimization (PPO)

PPO is on-policy, stochastic policy gradient method. A policy gradient method optimizes the policy of the agent directly with parameterized $\theta$. The value of the reward depends on this policy and various algorithms like PPO can be applied to optimize the value function with $\theta$.

To understand PPO, we need to understand how policy gradient methods works. The most used gradient estimator has the form

$$\hat{g} = \hat{E}_t[\nabla_\theta log\, \pi_\theta(a_t|s_t)\hat{A}_t]$$

where $\pi_\theta$ is a stochastic policy and $\hat{A}_t$ is an estimator of the advantage function at timestep t. The expectation $\hat{E}_t[...]$

Where $log\, \pi_\theta$ gives the probabilities of the policy $\hat{E}_t$ denotes the empirical expectation over timesteps and $\hat{A}_t$ is an estimate of the advantage function. In order to calculate the advantage function, we need two things, discounted rewards and baseline value function prediction $b(s_t)$.

$$\hat{A}_t = R - b(s_t)$$

During training, loss is calculated. If the advantage function $\hat{A}_t$ is positive PPO increased the probability of $a_t$ and if negative vice versa. The loss function is clipped in a way if $\hat{A}_t$ too high, do not update it too much since policy might get worse. Therefore, the objective function gets clipped to limit the effect of the gradient update. The aim is to not destroy our policy, based on a single estimate since Advantage function $\hat{A}_t$ is noisy and imperfect and there can be an error on that batch.

## 7.2 APPENDIX B: Intrinsic Motivation Method Details

The intrinsic reward $R_t^{intrinsic}$ is composed of two subsystems, a forward and an inverse neural network. The mechanism utilizes prediction error for creating a curiosity reward. The inverse model takes the input state $S_t$ and encodes into a feature vector $\varphi(S_t)$, forward model takes as inputs the feature encoding $\varphi(S_t)$, $\varphi(S_{t+1})$, and predicts the action $a_t$ which is taken by the agent to move to the next state [43]. Training this neural network amounts to learning function g defined as

$$\hat{a}_t = g(s_t, s_{t+1}; \theta_I)$$

where $\hat{a}_t$ is the predicted estimate of the action $a_t$ and the neural network parameters $\theta_I$ are trained to optimize,

$$\min_{\theta_I} L_I(\hat{a}_t, a_t)$$

where $L_I$ is the loss function that measures the discrepancy between predicted and actual actions. The learned function g is the inverse dynamics model

In the logits layer of our neural network, function g outputs probability which sums to 1 by utilizing a soft-max distribution across all possible actions. At the same time, the forward dynamic model is trained for predicting the $\varphi(S_{t+1})$ and then taking the difference between actual and prediction as an intrinsic reward.

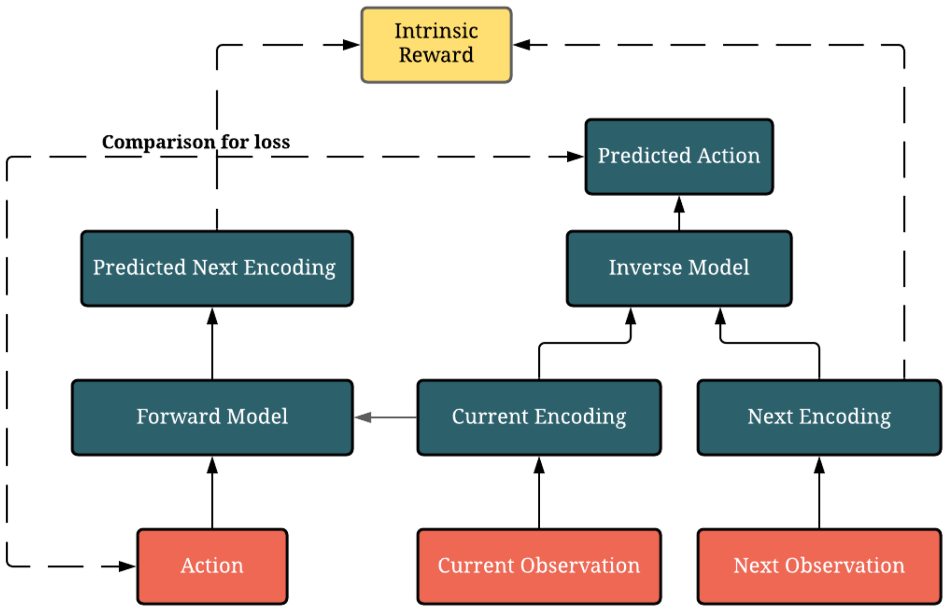


*Figure 18: Diagram of ML-Agents training architecture. The dark green boxes correspond to processes that occur in Unity Game Engine and the yellow boxes correspond to the Python-side of the project.*