Abstract

The Evolution of Trust:

Understanding Prosocial Behavior in Multi-Agent Reinforcement Learning Systems

David Nesterov-Rappoport

2022

This thesis looks into what factors contribute to intelligent agents making the decision to cooperate with one another in social dilemma-like interactions. Using concepts from game theory, artificial intelligence, and biology, the work explores what considerations push interacting agents towards prosocial or antisocial strategies. Cooperative behaviors form the backbone of social organization, furthermore understanding their governing mechanics is of the utmost importance. To achieve this, a custom piece of software is developed to enable experimentation in the domain, a number of advanced machine learning models are trained, and research from across different disciplines is synthesized into a single perspective. At the core of the quantitative research lies the stag hunt family of games, played by reinforcement learning agents which try to maximize their average number of points earned. By observing their learning behavior in relationship to configuration parameters, ideas from past research are validated, future avenues for exploration are identified, and concrete principles about these systems are unearthed. On the way there, the thesis summarizes the academic foundation for its methods and tools, explains how they work, and elaborates on how they are to be coupled into a single consistent system. Lastly, the implications of the research are related to the human context and framed in concrete terms.

The Evolution of Trust: Understanding Prosocial Behavior in Multi-Agent Reinforcement Learning Systems

An Honors Thesis
Submitted in Partial Fulfillment of the Requirements for the
Degree of Bachelor in Arts with Specialized Honors
in Computer Science at Drew University

by
David Nesterov-Rappoport

Thesis Director: Emily Hill

May 2022

Ø

Acknowledgments

To begin with, I would like to thank the person without which this project, and everything preceding it would not be possible: my grandmother, Natalia Nesterova. Without her support, I would have never had the educational opportunities I was lucky to have, and I wish to express my eternal gratitude. My grandmother taught me everything I know about creativity, hard work, and empathy - and this thesis carries her lessons.

Secondly, the completion of this project could not have been accomplished without the support of my professors. I want to express my gratitude especially to my thesis advisor, Dr. Emily Hill, my academic advisors, Dr. Barry Burd and Dr. Seung-Kee Lee, and members of my thesis committee, Dr. Minjoon Kouh and Dr. Yi Lu.

Lastly, I want to acknowledge the individuals whose encouragement and companionship has provided context to my work and continuously reminded me what truly matters. My dear friends, for always being there for me when I needed them. My family, for teaching me what it means to be strong when times are tough. And my girlfriend, Danie - who has been my pillar to lean on thorough this entire process, and created an environment which made this project possible. My heartfelt thanks to all.

”It is true that certain living creatures, as bees and ants, live sociably one with another... and therefore some man may perhaps desire to know why mankind cannot do the same.” - Thomas Hobbes, *Leviathan*

This paper was conceived of and written in uncertain times. And the times would only grow more uncertain with every page written. Through this thesis, I was hoping to understand what was happening around me, and how things came to be this way. I have always believed in humanity and the love that binds us; but it has been a strange time to be young. My research reassured me that a better tomorrow is possible, but also made it clear how long the path to get there is. I hope that in my work the willing reader sees what I saw also - the hopeful glow of of a far away peak, and the long road leading to it.

Contents

List of Figures

Glossary of Terms

action Something an agent can do within the rules of the environment. 7, 10, 11, 16, 59

agent The decision-making entity within a reinforcement learning system. Chooses future actions based on its experience and observations. viii, 7, 11–15, 26, 30, 31, 35, 37, 56, 59

alleles The possible values a gene can take on. 54

artificial intelligence The academic discipline concerned with achieving cognitive behavior in artificial systems. 7

chromosome A sequence of genes encoding the genotype of an individual solution in the run of a genetic algorithm. 54

crossover The part of a genetic algorithm which creates mating pairs when generating the next generation of the population. 54, 56, 57

emergence When an entity is observed to have properties that its composite parts do not posses on their own. In other words, emergent properties occur as a consequence of parts interacting with one another in a greater whole. 25, 58

environment All parts of the reinforcement learning system that are not the agent. This includes surroundings with which the agent can interact and the rules for how those interactions happen. 7, 13, 15, 16, 21, 30, 35–37, 59

fitness A measure of how well an individual is performing within the context of a genetic algorithm. Partially decides how much an individual will reproduce when a new population is being generated. 21, 54–56

game An interactive situation between rational players. 5, 31

game theory An academic discipline concerned with studying strategical interactions between rational agents from a quantitative standpoint. 5, 9

graph An abstract data type for representing complex, non-linear relationships between objects. 52

mapping A function; that is, a relation $f : A \to B$ such that for all $a \in A$, $f(a)$ corresponds to a unique $b \in B$. 8, 14, 16

model An abstract, information-based, representation of an object, person or system. 4, 9, 35

mutation Random variance in the genetic code of an offspring. 54

neural network A computer system inspired by biological neural networks, mainly used in the field of artificial intelligence. 18, 58

observation An agent's "perception" of an environments state. 16

policy A computing function which returns a valid action given a state of a problem. 10, 11, 14

population A group of candidate solutions generated in the run of a genetic algorithm. 22, 54–58

reinforcement learning An area of artificial intelligence which solves problems by continuously altering agent behaviors and beliefs in response to meaningful signals (reward) emitted by their surroundings. 7, 16, 30, 34

reward An instantaneous measure of how close the agent is to the environment's goal. Communicated to the agent at each time step via a *reward signal* emitted by the environment. 7, 9, 12–15, 35

social dilemma A class of game-theoretic interactions in which the non-cooperative payoff for a player exceeds the cooperative payoff. 20

stochastic Something which is well described by a random probability distribution. 11, 31

Chapter 1

Introduction

1.1 The Problem at Hand

We are living through a turbulent age. While economic productivity and technological potential are at an all-time high, people's capacity to organize and work together is an ongoing struggle. Political division, misinformation, and ideological conflict continues to challenge our social institutions and leaves our future as a civilization uncertain. With major threats looming over society, such as climate change and global war, it is now as important as ever to understand how people agree to cooperate and what it takes maintain a cooperative society once established.

To research the governing dynamics of social cooperation, one requires a model sufficiently complex to capture essential nuance, but simple enough to make large scale computation possible. In an effort to achieve this, we will break up the original modeling problem into two smaller ones. On one hand, we need a model of the cooperation problem itself, and on the other, a model for the learning behavior of individuals facing it. Through this arrangement, we plan to observe the phenomena in an abstract realm, and uncover principles governing this important subject matter.

1.2 A Game of Risk and Trust

Let us imagine two hunters tracking down a stag through the woods together. They have been on the hunt for most of the day and fatigue is starting to set in. However, leaving empty handed is not an option for either of them, so the two press on. They set a trap and hide in the bushes, hopefully awaiting the uncertain arrival of their prey. Hours pass, and suddenly, a hare emerges from the woods, starting to graze near their hiding spots. The hare is a much smaller catch, but has enough meat to feed one person, and, unlike the stag, is immediately within reach. Each hunter now faces a choice — leave their hiding spot to kill the hare, guaranteeing themselves a meal, or stay faithful to the plan and continue waiting for the stag. As both hunters are aware, if either of them goes after the lesser prey, the trap will not succeed. With the critter's arrival, doubt has been introduced into their partnership. They have now begun a delicate dance between trust and risk, set to the music of their thoughts. Do they trust their partner enough to bet on their continued cooperation? Or do they deem it too risky and opt for the hare, defecting away from their agreement? These contemplations occupying their minds, the hunters lie in wait, trying to guess what the other is thinking and whether to take off after the hare or wait for the stag.

The stag hunt, as first told by the french philosopher Jean-Jacques Rousseau, is a story that became a game [5]. Imagine that the aforementioned hunters may only choose between hunting hare or hunting stag, and that the chances of catching a hare are independent of what others do. Additionally, the stag is always worth more than the hare, and one may not possibly catch a stag alone. In this form, the stag hunt is a well-recognized area of study for scientific game theory, serving as a medium for researching cooperative decision making. Sometimes referred to as the assurance game, trust dilemma, or common interest game, its essential aspect is its expression of the natural conflict between trust and risk. Despite being simple, it can be used to accurately converse about a number of complicated real life analogues. The original author intended it as a metaphor for the establishment of society itself — the story describing how individuals give up their autonomy to participate in the collaborative project of civilization. Similarly, the stag hunt can be used to represent smaller consensus problems, such as recycling, wearing a mask in a pandemic, or holding a stock during a short squeeze.

By compressing the complexity of cooperative decision-making to a single mathematical formula, the stag hunt enables us to study incredibly complicated social interactions otherwise hidden from empirical analysis. The power of abstraction allows us to safely research an otherwise inaccessible problem space. Consequently, in the stag hunt we have found the first of our models - a representation of the problem of cooperation.

Figure 1.1: A 44,000 year old cave painting depicting a group of humans hunting a large mammal. The problem of the stag hunt is as ancient as human civilization itself.[1]

1.3 Learning Through Reward

Having established the game, we now decide on our players. To find them, we turn to the ripe field of artificial intelligence. Many candidate approaches emerge, but one stands out amongst the rest as the most promising and intuitively compatible with the game theoretic approach of the stag hunt. Called reinforcement learning, it is an area of artificial intelligence which solves problems by continuously altering A.I. behavior in response to meaningful signals emitted by their surroundings. We refer to the entity doing the thinking as an agent, the signals as reward, and the system making up the surroundings as the environment, which is illustrated in figure 1.2. Reinforcement learning algorithms attain desired behavior from agents by using the environment to communicate what actions constitute good and bad performance. The core approach is essentially similar to training an animal — good actions are encouraged with rewards, such as treats, and bad actions discouraged through punishment or lack of reward.

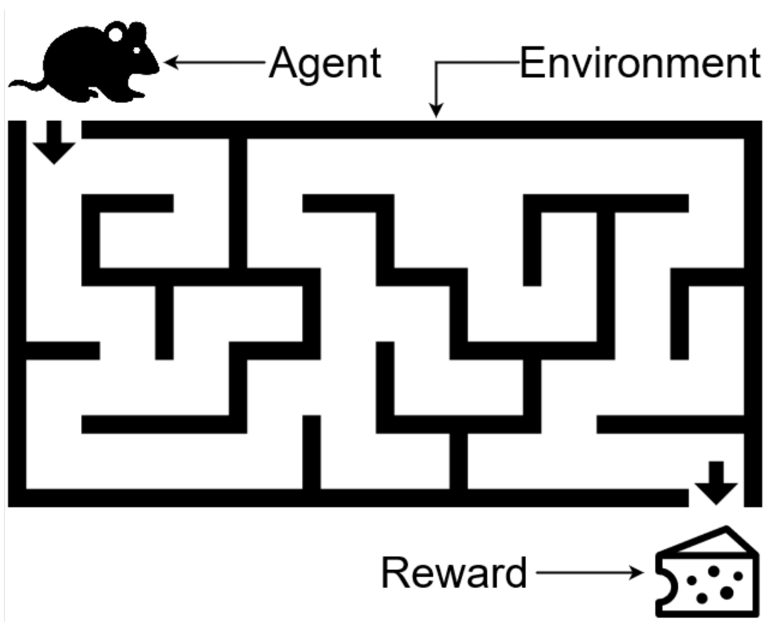


Figure 1.2: In this reinforcement learning scenario, an *agent* is learning strategies to solve their *environment*, in order to receive *reward* in the form of cheese.

There is a number of reasons for why reinforcement learning synergizes well with the stag hunt. First, the stag hunt's structure naturally lends itself to the reinforcement learning approach. Many other problems require considerable work to enable agents to accurately interface with their environment. In contrast, the stag hunt, being fundamentally a mapping from actions to rewards, is readily usable as an environment with minimal configuration. Second, reinforcement learning agents trained by playing against one another have been established to be capable of learning behavior much more complex than the environment itself [6][7]. This is both promising in terms of results, and intuitively seems to mirror human learning. Thirdly, reinforcement learning is exciting because of its established track record of achieving impressive results in game-based problems much like the stag hunt. Some researchers in the field even argue that reinforcement learning is capable of one day producing a true general intelligence [8].

With these things in mind, it can be seen how reinforcement learning postures itself as the appropriate choice for modeling individual behavior in our overall system. Naturally fitted to the stag hunt and multi-agent contexts, and ripe with exciting research, reinforcement learning is the clear choice for our second model.

1.4 Our Contributions

In summary, through analyzing a strong model of the cooperation problem, we hope to understand the social reality underlying agent cooperative dynamics. The thesis will engage with a number of relevant technologies, summarize relevant research in the area, implement an open-source piece of reinforcement learning software, and conduct experiments with said software. In these steps, we will be focusing on establishing concrete rules about the stag hunt, and how agents learn to interact with it.

Chapter 2

Background

2.1 The Stag Hunt

2.1.1 Formal Description

We begin with a formal description of the aforementioned stag hunt game. In the framework of game theory, the class of stag hunt games is a well-known model for studying the trade-off between trust and risk[9]. In stag hunt, two players must independently decide between two distinct plans of action, also called strategies. The first of these is the risky option, which yields a high reward, but only when both players have picked it. If only one player chooses it, they are punished with a bad payoff. Accordingly, this action is commonly referred to as *"cooperate"*, or, following the original story, *"hunt the stag"*. The alternative is the safe, low reward action, the success of which is not dependent on the action of the opponent, but has significantly smaller returns. Following literature, we will refer to it as *"defect"*[1]. Thus, the game is expressed by the following payoff matrix, where rows represent strategy choices for player 1, columns represent strategy choices for player 2, and cells show rewards to each player given a particular choice of strategies.

[1] Depending on the metaphor being employed, this will sometimes also be referred to as *"hunt the hare"* or *"forage"*. To avoid confusion, we will not use such synonyms.

		Player 2	
		$Hunt$	$Defect$
Player 1	$Hunt$	(h, h)	(g, c)
	$Defect$	(c, g)	(m, m)

Definition 2.1.1 (Stag Hunt)**.** A 2 x 2 game is a **generalized stag hunt** if $h > c \geq m > g$, where:

- h is the reward for a successful cooperative action
- c is the reward for being the sole defector
- m is the reward when both players defect
- g is the punishment for hunting alone

2.1.2 Generalized Stag Hunts

The above description of a stag hunt is an idealized version called the normal form. While normal form games have guided research on social dilemmas for decades, they do not accurately represent a number of important features of their real-world equivalents[10]. To begin with, real world cooperation problems are temporally extended. Secondly, cooperation and defection are labels that refer to an agents policy, not individual actions. Lastly, cooperate and defect are not atomic actions, and decisions must be made with only partial information about the state of the world[10]. With this in mind, it can be seen why the normal form stag hunt game has limited modeling capacity, and a more nuanced variant is needed for our ends.

Definition 2.1.2 (Normal form)**.** A normal-form game is a tuple (N, A, u), where[7]:

- N is a finite set of *n* players, indexed by *i*.

- $A = A_1 \times \cdots \times A_n$, where A_i is a set of actions available to player *i*. Each vector $a = (a_1, \cdots, A_n) \in A$ is called an action profile.

- $R = (r_1, \cdots, r_n)$, where $r_i : A \to R$ is a real-valued payoff function for player *i*.

Consequently, our tool set will include sequential stag hunt-like Markov games based on past research which approached this problem before us[10][11]. For a game to be rightfully considered stag hunt-like, it does not necessarily have to be played between two players or be composed of two possible actions. The defining feature of the model is the payoff difference between prosocial and antisocial strategies. Consequently, a given game, regardless of how complicated its policy space is, will be considered stag hunt-like if it preserves the high level properties of the normal form stag hunt[11].

Definition 2.1.3 (Markov Game)**.** A Markov game, also known as a stochastic game, is a tuple (Q, N, A, P, R), where:

- Q is a finite set of games.

- N is a finite set of n players.

- $A = A_1 \times \cdots \times A_n$, where A_i is a finite set of actions available to player i.

- $P : Q \times A \times Q \to [0, 1]$ is the transition probability function. $P(q, a, \hat{q})$ is the probability of transitioning from state q to state $\hat{q}$ after action profile a.

- $R = r_1, ..., r_n$, where $r_i : Q \times A \to R$ is a real-valued payoff function for player i.

In this paper, we will be relying on a number of N-strategy 2-player stag hunt-like games to explore the problem at hand. We model them using the framework of Markov, also called stochastic, games, which are generally accepted as a standard framework for modeling multiple adaptive agents with interacting or competing goals[12]. In each of our games, two agents move in any of the 4 cardinal directions on a N x N grid, with the goals depending on the specific game. Despite being composed of numerous sub-games, these games are considered stag hunt-like because any 2 x 2 sub-game within them is a stag hunt[11].

2.1.3 Nash Equilibria

A Nash equilibrium, in game theory lexicon, is a strategy match-up in which neither player has an incentive to deviate from their chosen strategy given what the other player is doing[7]. In other words, they are possible points of convergence for a learning process[11]. Now, let us assume A_1 and A_2 are the action spaces of the two players, and $R_i(a_1, a_2)$ is their reward given a particular choice of strategies.

Definition 2.1.4 (Nash equilibria)**.** Nash equilibria are strategy pairs (a_1^*, a_2^*) such that for any a_1^x, where ^{x} is the particular choice of strategy, we have

$$R_1(a_1^*, a_2^*) \geq R_1(a_1^x, a_2^*)$$

and for any a_2^x we have

$$R_2(a_1^*, a_2^*) \geq R_2(a_1^*, a_2^x)$$

Within the strategy space of stag hunt exist two Nash equilibria. The payoff-dominant equilibrium is $(Hunt, Hunt)$. It is referred to as payoff-dominant because it has the highest reward of all strategy pairs. Since it involves cooperation between the two agents, we will be referring to it as the prosocial equilibrium. The second equilibrium is $(Defect, Defect)$, and it is the risk-dominant equilibrium as it involves the least risk of all strategies. To contrast it with the alternative, we will refer to it as the antisocial equilibrium.

2.1.4 Risk Balancing

The existence of these two equilibria is what makes stag hunt such a great means of studying the problem at hand. An algorithm attempting to learn how to play this game is bound to discover, and settle on, one of the two strategy pairs. They may judge cooperation to be too risky and settle on acquiring steady, low returns. Or, alternatively, they develop a significant amount of trust towards their partner and choose the risky, high reward action. Neither option is, strictly-speaking, the ”better” one. The quality of the choices varies with environmental conditions. For example, past research has implied that the risk-dominant equilibrium is the optimal one in arrangements where the punishment for a failed hunt is sufficiently high, as the risk of the payoff-dominant strategy passes a certain threshold[13].

Consequently, despite always being the most efficient choice in theory, the prosocial strategy is not always the most reasonable to pick. Given how problems of cooperation are frequently complicated by limited information, it stands to reason that risk consideration is necessary for an intelligently approaching games of cooperation. For this reason, understanding the underlying competition between trust and risk holds the key to unearthing the governing dynamics of social dilemmas.

2.2 Reinforcement Learning

2.2.1 Formal Description

Reinforcement learning is an area of machine learning concerned with how intelligent agents ought to take actions in an environment in order to maximize the notion of cumulative reward. Reinforcement learning systems have to operate continuously within an uncertain environment based on delayed and frequently limited feedback[14]. An essential feature of the reinforcement learning protocol is how it decouples the problem into two sequentially interacting components[8]. The solution is formulated in the form of an agent - an entity which makes observations about its surroundings and decides on what actions to take next. The problem is the environment inhabited and acted on by the agent, providing feedback in the form of observations and reward.

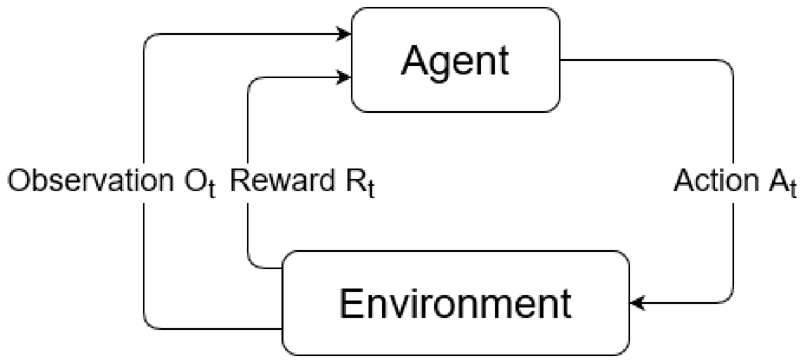


Figure 2.1: The general structure of a Reinforcement Learning system.

Furthermore, the central goal of a reinforcement learning application is to learn a what is called a policy – a mapping from the state of the environment to a choice of action which yields effective performance over time[14]. Relevant to our ends is the fact that reinforcement learning agents do not explicitly model the opponent's strategy – they are understood to be part of the environment[7].

2.2.2 Agents

An *agent* is defined as a system receiving at time t an observation O_t, providing in response an action A_t. More formally, the agent is a system $A_t = \alpha(H_t)$ that selects an action A_t at time t given its experience history $H_t = O_1, A_1, ..., O_t = 1, A_{t-1}, O_t$ [8]. In simple terms, the agent maintains a memory of its experience and when it receives an observation, selects the action which seems to be the best one given what the agent experienced in the past. The decision making process through which the agent decides on the next action is called the policy. The explicit goal of any reinforcement learning agent is to learn the optimal policy - a strategy with the highest reward. Given that the reward function is sufficiently accurate, the optimal policy constitutes a solution to the problem at hand.

2.2.3 Environments

An *environment* is defined as a system receiving at time t an action A_t and responding with an observation O_{t+1} at the next time step. In formal terms, the environment is a system $O_{t+1} = \in (H_t, A_t, \eta_t)$ which determines the next observation O_{t+1} given an experience history H_t, the latest agent action A_t, and potentially a source of randomness η_t [8]. An important aspect of the environment-agent distinction is that the agent is exclusively the entity in charge of decision making. Everything that is outside of it, even if intimately connected to the agent (such as a physical body in a robotics context), would be considered a part of the environment. To note, in a multi-agent setting, each individual agent considers the others to be a part of the overall environment.

2.2.4 Reward

The most essential part of the reinforcement learning approach is the reward. Given how reinforcement learning represents goals in the form of cumulative reward, an accurate reward function is essential for the agents to learn the important features of their environment. We define a *reward* as a special scalar observation R_t, emitted by the environment at every time-step t through what is called the reward signal. This reward is meant to provide an instant measure of the agent's progress towards the specified goal. Although simple, this formulation is sufficient to represent a great variety of goals and constitutes a fundamental strength of the reinforcement learning framework [8].

2.3 Q-Learning

2.3.1 Basic Description

Q-learning is a well-known reinforcement learning approach that has pioneered a lot of now-standard practices and ideas in the field. In essence, it is a simple way for agents to learn how to act optimally in controlled Markovian domains[15]. Conceptually founded on ideas of dynamic programming, the approach works by continuously improving its internal evaluations of how good a particular action is given some observation. Q-learning is considered to be a form of model-free reinforcement learning, as the agents do not build an internal model of their environment. Instead, the agents try some action in a particular state and observe the consequences. Through this, the agents learn a mapping from actions to reward which enables them to engage with the environment in an intelligent way. Consequently, Q-learning is a primitive form of learning, with results being achieved through simply trying all actions in all states[15]. None the less, the approach has been wildly successful for its high level of expandability and intuitive conceptual foundation.

Definition 2.3.1 (Q-Learning)**.** Q-learning is the following procedure [7]:
Initialize the Q-functions and *V* values (arbitrarily, for example)
repeat *until convergence*

1. Observe the current state s_t.
2. Select action a_t based on current Q-values and take it.
3. Observe the reward $r(s_t, a_t)$ returned from the environment.
4. Update the Q-value for the state-action pair (s_t, a_t) using a value iteration update function which uses the weighted average of the old Q-value and new information.

In words, the process revolves around the agent at each time step t, selecting an action a_t, taking it, seeing what reward r_t and new state $s_t + 1$ are returned by the environment, and updating the Q-value for the action using the information observed. Specifically, the Q-value is updated using a Bellman equation as a simple value iteration update.

Definition 2.3.2 (Q-Learning Bellman equation)**.**

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} + \overbrace{\underbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \times \underbrace{\max_a Q(s_t + 1, a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t, a_t)}_{\text{old value}}}^{\text{temporal difference}}$$

where,

- r_t is the reward the agent received when moving from the last state s_t to the new state s_{t+1}.
- α is the algorithm learning rate ($0 < \alpha < 1$).

The big-picture intuition behind this approach has to do with how it approximates the unknown transition probability by using the actual distribution of states reached in the duration of the game[7]. An important note is that while Q-learning guarantees good learning, it makes no promises on how quickly the desired convergence would occur[7].

The most common implementation of Q-learning is through the use of Q-tables - which are simply tables where one axis are the possible states, and the other axis are the possible actions (illustrated in figure 2.2). Each cell contains a Q-value corresponding to the quality of the action given that state, and the value is updated each time the combination is experienced using the rules described above.

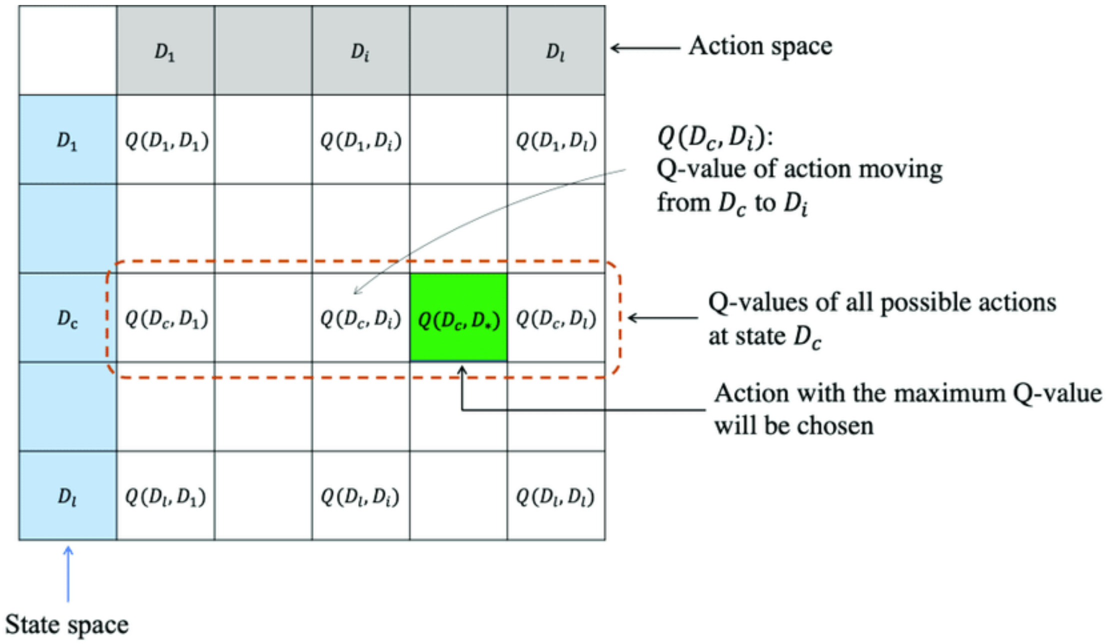


Figure 2.2: An illustration of a Q-Table[2]

2.3.2 Deep Q-Learning

Given how the foundational Q-learning approach is rather simple, it is an inappropriate tool for learning complicated environments. However, more sophisticated algorithms have been devised that build on top of the high-level ideas of Q-learning to create efficient approaches to solving complicated problem spaces. Of particular interest to us is the deep Q-learning approach, which leverages a convolutional neural network whose input is the environment state and whose output is a value function estimating future rewards[16]. The approach essentially translates the core mechanism of Q-learning, which is value iteration, onto the engine of neural networks. As the agent explores the environment, it updates the neural network weights according to the principles of Q-learning and gradient descent[16], as can be seen in figure 2.3. This allows the algorithm to surpass challenges which immobilize competing reinforcement learning methods, while doing so in a conceptually straight forward fashion. DQN was used to effectively solve stag-hunt like games in past research, which provides additional evidence for our choice[17][10]. For these reasons and more, this is the algorithm we used in our main line of experimentation.

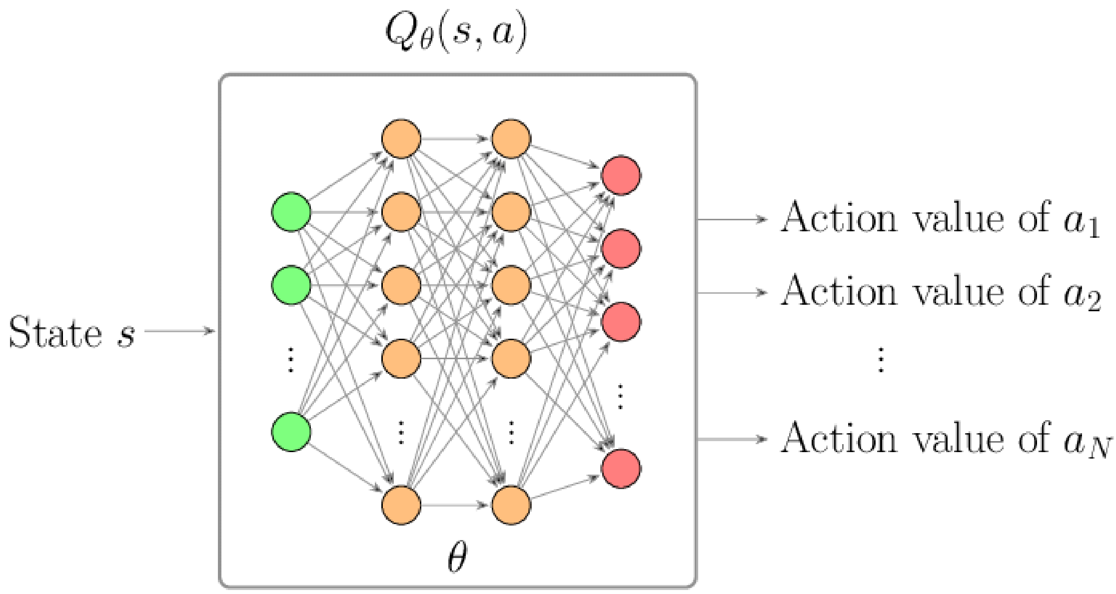


Figure 2.3: A visual demonstrating how a Neural Network is used to approximate Q-Table functionality in the DQN approach.[3]

Chapter 3

Related Work

3.1 Risk And Society

The beginning point of our research is the recognition of a strong connection between risk and the likelihood of cooperation in stag hunt type social dilemmas. This has been investigated and confirmed numerous times by different researchers in varying contexts. For example, Duguid et al. compared the abilities of chimpanzees and young children to coordinate with a partner in Stag Hunt interactions[18]. Their observation was that when the risks were low and information was cheap (the partner could be easily observed), both species successfully coordinated on the prosocial choice. However, when the risks were increased and information was less readily available, the chimpanzees failed to cooperate, while human children were still successful[18]. This is consistent with research on the relationship between risk and the likelihood of cooperation done by Bearden[19]. Increased risk correlates to a decrease in cooperative choices, but humans overcome this difficulty by leveraging social behaviors[18][5]. This general trend holds across particular instances, and establishing rules which govern is of the highest priority.

We know that people in a group will frequently be willing to give more and take less, which may appear irrational from the individual perspective, but makes sense when one considers the improved fitness of the group as a whole[20]. In an environment where learning rules are subjects to evolutionary pressure, selfish learning is sub optimal and will be out competed by prosocial learners. A great deal of evidence for this is presented in Kropotkin's *Mutual Aid*, which details the numerous ways in which inter-species cooperation is a necessary principle of evolution[21]. The work posits that ”the fittest are not the physically strongest, nor the cunningest, but those who learn to combine so as mutually to support each other, strong and weak alike, for the welfare of the community”[21]. In other words, there is considerable evidence in evolutionary theory which suggests that stag hunt like interactions are numerous, and absolutely essential to the dynamics of the biosphere. What is of utmost interest from an academic standpoint is how the behavior dynamic crystallizes on the basis of the underlying interaction pattern, and the social structure mediating it. Put simply, competition and cooperation are both valid considerations in social dilemmas, and what decides which one dominates has to do with the collective-level behaviors that have been adopted.

This reality is something usually left out of the archetypal stag hunt story. By abstracting away the fact that the hunters have a relationship before and after the hunt, the thought experiment fails to model a crucial feature of the social phenomena in question. Being able to consider the reasoning of other agents is necessary to enable individuals to cooperate and produce optimal group rewards[20]. Furthermore, it stands to reason that social modeling is essential for understanding how to create multi-agent systems capable of stable cooperative behaviors. Such behavior requires social structure of a particular kind, which is unachievable without modeling persistent relationships between individuals through some means. While attempting to model this may be out of the scope of this work, researching this is essential for long-term progress in the field.

3.2 The Evolution of Social Structure

Brian Skyrms is a famous researcher in the fields of cooperation and collective action. His work, *The Stag Hunt and the Evolution of Social Structure*, is an enormous source of inspiration for this thesis. In the book, Skyrms posits that successful coordination on the social choice in stag hunt-like interactions is dependent on the co-evolution of cooperation and social structure[5]. Specifically, he argues that three factors affect the emergence of social structure and collective action: location (interactions with neighbors), signals (transmission of information), and association (the formation of social networks)[5]. In the scope of our implementation, location and association are achieved through NGA population graphs, and signals through peer gifting.

A major premise of Skyrms work, which we also accept as true, is that ”rational choice is not necessary for solving the problem of the social contract”[5]. If the opposite was true, Skyrms argues, we would not observe phenomena such as eusocial insects or social bacteria like the *Myxococcus xanthus*[5]. Rather than human-like cognition, social cooperation seems to require emergent behavior instead. In the words of the author, ”transient phenomena [are] crucial to an understanding of real [social] behavior”[5]. Similar sentiments are expressed by quantitative research in the area[6][9].

3.2.1 Location

The essence of the location argument is that rational agents behave fundamentally different when bargaining with neighbors rather than with strangers. Skyrms shows how when who interacts with whom is decided through some representation of physical location, rather than through random match-ups, it has major implications on the evolutionary dynamics of the population as a whole[5]. Known models and past experiments demonstrate how ”interaction with neighbors on one or another spatial structure can allow cooperative strategies to persist in the population", whereas when "played in a well-mixed large population, the evolutionary dynamics drives cooperation to extinction"[5]. An important addition is that local interaction only produces these benefits in large populations. Selfish strategies are locally optimal, meaning that they will dominate small populations. The dynamics discussed above seem to be emergent, and therefore need a large set of individual structures to work with. Additionally, while local interaction making a difference is a "modest general truth", the actual dynamics underlying it are nuanced and not entirely clear[5]. The dimension, reproductive dynamics, and the kinds of neighborhoods modeled, all seem to play an important role in determining the outcome of a simulation[5]. In spite of this, in the context of the stag hunt, the author posits with confidence how "local interaction opens up possibilities of cooperation that do not exist in a more traditional setting"[5].

3.2.2 Signals

It is an observed fact that "signaling systems are ubiquitous at all levels of biological organization"[5]. Honeybees have a signaling system for communicating the location and quality of food sources, birds use signals to warn and woo one another, and perhaps more remarkably, some bacteria use signaling systems to make decisions at the colony level[5]. A salient illustration of this is the behavior of the aforementioned social bacteria, *Myxococcocis xanthus*. When food is aplenty, the bacteria are free-living individuals. When the collective senses that starvation is widespread, the bacteria will engage in coordinated attacks on larger microbial prey, overwhelming them with secreted enzymes[5]. In other words, these single-celled organisms, incapable of thought, have *successfully solved the problem of the stag hunt*. They have done so through the use of Quarum Signaling, which depends on a signaling molecule that the bacteria emit, diffusing it into the environment[5]. There is thus strong evidence that one does not require complicated communication strategies to solve complicated coordination problems.

Skyrms is particularly interested in how speech and language can emerge without presupposing themselves. In other words, how does one arrive at a convention? And how does this convention remain in force? Reframed in game theoretic language, these are problems of equilibrium selection and equilibrium maintenance[5]. Skyrms does a number of experiments looking into the relationship signaling has to evolutionary dynamics. What he is able to show through his analysis, is that one can ”have an account of the spontaneous emergence of signaling systems that does not require preexisting common knowledge, agreement, precedent, or salience”[5]. In other words, there is good reason to believe that signaling systems can arise naturally from the dynamics of learning itself[5].

3.2.3 Association

The author begins his discussion on association by revisiting the remarkable bacterium *Myxococcus xanthus*, which moves by gliding on slime trails. While the mechanism of it is not entirely known, it is observed that using and following an existing slime trail is significantly easier than making a new one[5]. Furthermore, bacteria will go out of their way to follow an existing slime trail. At that point, Thorndike's laws of learning come into force. Specifically, the *Law of Effect*, which states that an action that leads to positive rewards becomes more probable, and *Law of Practice* – that an action that is not successfully used tends to become less probable[5]. Consequently, trails that are successfully used to find food end up being reinforced, while trails leading nowhere useful dry up. Important to our ends is the fact that such biological adaptations, for all intents and purposes, are valid instances of reinforcement learning. Skyrms thus argues that reinforcement learning has something to teach us about the dynamics of association and that dynamics of interaction are a crucial factor in the evolution of collective action[5].

To uncover these lessons, the author conducts a series of experiments in a simple social reinforcement model, adding new mechanics one by one and observing their effects on the learning process. The original model is a metaphorical version of the Pólya urn process, where ten strangers find themselves in a new location and each morning, everyone chooses someone to visit that day. Each individual starts with a numerical weight for each other participant, visits them with probability proportional to that weight, and updates the weight positively or negatively depending on how pleasant the interaction is[5]. The model can be excessively tuned, such as by specifying whether the host, visitor, or both get the reward, and how large that reward is. Let us summarize what Skyrms learned from his experimentation.

To begin with, the ”boring” version of the simulation is one where the guests are uniformly reinforced for pleasant interactions. In other words, a pleasant interaction yields a reward of 1 to the guest, but nothing to the host. What is observed from this simulation is not new, but notably interesting in our context. This friend-making process is guaranteed to converge, meaning people will at some point only visit their “friends”, but it is completely random at what set of friendships we end up at. Stated otherwise, from a perfectly uniform starting point, a particular order will emerge with no specific reason for why that order emerged over a different one. The emergence of order is assured. The way in which these interactions crystallize into concrete forms is an essential fact that is ought to be recognized more by those studying multi-agent systems.

The simulation, however, changes if one considers an arrangement in which visits are uniformly unpleasant. Same set-up as above, but the interactions are always unpleasant, meaning individuals will be less likely to visit someone after interacting with them. Whereas positive reinforcement led to the spontaneous emergence of interaction structure, negative reinforcement wipes it out and leads to uniform random encounters[5]. The conclusion is that random encounters help with making enemies, but not with making friends.

What happens if both the guest and the host are reinforced for pleasant interactions? In this circumstance, reinforcement becomes interactive, as an individual’s weight changes not only based on who they visit, but also who chooses to visit them. In such a set up, structure still emerges, but a general characterization of that structure is far from evident[5]. Skyrms observes that convergence is slow, and long-lived transient behavior becomes an important part of the story.

Arguably the most interesting alteration one can make to this simulation is through adding fading memories. If one adds a moderate rate of memory fade by only preserving 90 percent of past experience, the variety of visiting probabilities outlined earlier disappears, and agent behavior crystallizes into deterministic visiting patterns[5]. The less our agents forget, the more the simulation behaves like the original case. Skyrms thus posits that any forgetting at all leads to “a deterministic interaction network crystallizing out of the flux of interaction probabilities”[5]. We wish to explicitly draw the readers attention to this observation, as an awareness of the crystallization process is necessary to understand the big picture which emerges from this research.

Skyrms then alters the simulation so that each visit is a Stag Hunt game, and individuals are assigned stag or hare hunter at the start. In this circumstance, the population eventually splits into two mutually exclusive groups based on prey preference, each group then engaging in a microcosmic version of the greater game. Consequently, stag hunters will group together and begin achieving far greater rewards than their hare-hunting rivals. In other words, once this sort of interaction structure has evolved, stag hunters prosper[5]. Choosing partners has an immense effect on the learning dynamic of the collective as a whole. A fluid interaction structure allows individuals to sort themselves into behavioral groups, which makes cooperative strategies much more evolutionary competitive.

3.2.4 Conclusions

Skyrms began his analysis by asking: ”How can you get from the non cooperative hare hunting equilibrium to the cooperative stag hunting equilibrium”[5]. Through his experimentation and inquiry, he is able to demonstrate the emergence of some general principles. The process begins with agents experimenting with stag hunting in small groups. Eventually, because of associative behavior, the stag hunters come to interact mostly or solely with one another. This takes time, but is sped up through means of fast interaction dynamics[5]. Once the stag hunting community is established, they come to dominate the population through reproductive and imitation dynamics. This process is further facilitated if the reproduction or imitation neighborhoods are larger than interaction neighborhoods. As the culture of cooperation spreads, it can maintain viability even in the unfavorable environment of a large, random-mixing population through the utilization of signaling.

3.3 Mutual Aid

Peter Kropotkin is mainly known as a political writer and avid advocate of anarcho-communism. His philosophy takes great inspiration from his naturalist background, which he has acquired during his extensive time in Siberia. Having observed the importance of cooperative structures in nature, Kropotkin would later use that knowledge to develop his political and societal views. These observations are described and discussed in his chief scientific contribution, the book *Mutual Aid: A Factor in Evolution*. In the text, Kropotkin argues that the understanding of Darwinian processes as fundamentally based on inter-species competition is limited, as it is but a part of the whole. To truly comprehend evolutionary dynamics, one has to acknowledge the essential role of inter-species cooperation. As he puts it, ”sociability is as much a law of nature as mutual struggle”[21].

Kropotkin's work helps ground the importance of the subject matter. By recognizing cooperation as a fundamental challenge of biological systems, and incorporating that reality into our understanding of human collectives, we are enabled to develop a clearer sense for how cooperative structures can be engineered among us. It is wrong to think that one has to invent new structures - rather, we are ought to observe what has already been achieved in nature, and translate that onto the human substrate.

3.3.1 Observations from Nature

Kropotkin begins by describing how during his time in Eastern Siberia, he “failed to find – although [he] was eagerly looking for it – that bitter struggle for the means of existence, among animals belonging to the same species, which was considered by most Darwininsts...as the dominant characteristic of struggle for life”[21]. On the contrary, he ”saw Mutual Aid and Mutual Support carried on to an extent which made [him] suspect in it a feature of the greatest importance for the maintenance of life, and the preservation of each species, and its further evolution”[21]. The argument is strong because of the powerful empirical evidence that the author puts forward. The importance of cooperation is overwhelmingly evidenced by colonies of rodents, the migrations of birds and deer, the pack behavior of wolves, and numerous more.

Furthermore, when one becomes acquaintanced with nature, the prevalence of stag hunt like interactions becomes obvious. Organic beings are said to have two essential needs: that of nutrition, and that of continuing the species[21]. According to Kropotkin, the former causes inter-species competition, and the latter brings them together and forces mutual support. Accordingly, competition over resources seems to be much less prevalent than mainstream evolutionary theory may suggest. For example, numerous birds of prey, who one would assume to be highly competitive among one another due to their predatory nature, have developed advanced cooperative practices. In fact, the species which rob each other are in decay, whereas those which practice mutual aid are thriving[21]. Living beings which best know how to combine, and to avoid competition, have the best chances of survival and further progressive development. This is echoed many times over on all levels of the biosphere - as was previously discussed by Skyrms as well[5]. Consequently, we are left with strong reasons to believe that the stag hunt is an absolutely essential aspect of evolutionary dynamics, and organic life is in great deal defined by how it approaches this fundamental challenge.

3.3.2 Rhymes of History

Humans are an undeniably social animal, and it is worthwhile to investigate how humans have developed social structures and how those structures evolved. Kropotkin spends the latter half of his book looking into this, and it is where his writing moves from biology to sociology and politics. At the beginning, Kropotkin diffuses the notion that humanity originated from a state of constant warfare, and that conflict has been the driving force behind human progress[21]. Mirroring his earlier line of argumentation, he posits that co-operative achievements instead take primacy, and the historical narrative focus on conflict is a dated cultural artifact. He then describes the history of humanity as a gradual increase in cooperative structures, from the clan organization of our origin species, to the industrial organization of recent times. The core idea, which remains conceptually sound even in consideration of potential ideological biases, is that human society-at-large is engaged in a continuous engineering endeavor of creating social structures which enable more reliable and efficient methods of collaboration. While conflict is an undeniable factor in how civilization evolves, once one moves out of the institutionally mandated historical perspective, it becomes clear that cooperation and structures which enable it are significantly more important. Understanding these structures is the key to solving problems of coordination.

Chapter 4

Experiments And Implementation

The core inquiry of our thesis is: how do individuals learn to cooperate and set aside their differences for the sake of the common good? The implementation problem we had to solve was the question of how this can be studied from the quantitative perspective. Having found our answer to that in stag hunt and reinforcement learning, we can reframe our core inquiry in domain language: what factors contribute to reinforcement learning agents learning to converge on the prosocial equilibrium? To answer this, we run experiments where the agent architecture and environment rules are kept constant, and risk configuration is varied, so we can analyze its impact on the speed and efficiency of behavioral convergences.

4.1 Environment Implementation

Experiments are ran on a custom environment, made compatible with OpenAI Gym and PettingZoo, developed in-house on the basis of Markov games with Stag Hunt properties described in the work of Peysakhovich and Lerer[22][23][11]. The environment is made to be a robust, efficient, and customizable tool for the study of prosocial behavior in multi-agent reinforcement learning. The code is published on GitHub at https://github.com/NullDefault/Gym-Stag-Hunt

under the MIT license, making it available to the open source artificial intelligence community. The repository has already received a fair amount of attention, with multiple enthusiasts downloading the code to run experiments or add functionality. This serves as evidence for the usefulness and applicability of the software developed, and we hope that as we advertise it to the reinforcement learning community, more researchers will make use of it in their work.

4.1.1 Peysakhovich and Lerer's Environments

Peysakhovich and Lerer's work looks into how changing the learning rule of a single agent can improve its outcomes in Stag Hunts that include other reactive learners[11]. Their experimentation shows that prosocial preferences applied to even one individual makes the prosocial equilibrium more desirable overall in stag hunt like interactions. This is first established in simple matrix form stag hunts, but is validated and confirmed in more complicated analogues. These analogues are stochastic games that preserve the high level properties of stag hunt interactions. Each game is modeled using the framework of Markov games[11][7]. These stag hunt-like games are a medium where analytical solutions are difficult, hence why they are an interesting means of testing if approaches developed on the simple stag hunt are applicable in more generalized domains[11].

The Three Games

In each game, a pair of agents move on a 5x5 grid in the 4 cardinal directions and, through their actions, have a choice between a prosocial and antisocial strategy. There is a total of three games: Markov Stag Hunt, Harvest and Escalation (Figure 4.1a).

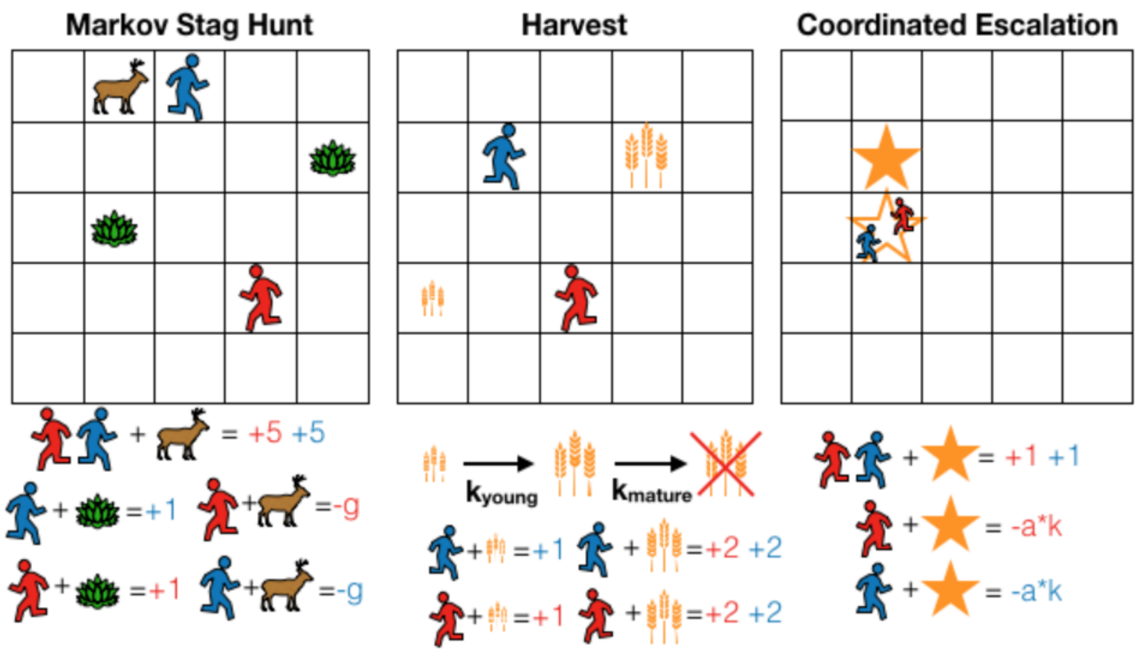
1. In the Markov Stag Hunt, the field is populated with one stag and two plants. Ending

a turn in the same cell as a plant rewards 1 point to the harvester and makes the plant re-spawn in a different location. Ending a turn in the same cell as the stag punishes the agent with negative *g* points, unless the other agent is also in the same spot. If both agents overlay the stag simultaneously, they each gain *g* points and the stag re-spawns elsewhere. The original *g* is 5. Each time step, the stag will move towards whichever agent is closest to it, however it can never catch an agent who continues to move away from it. Similar to the matrix-form stag hunt, there are two equilibria in this game - the agents either try together to hunt the stag, or opt for the guaranteed returns of harvesting plants. Lastly, the risk is asymmetric as agents are punished for hunting alone, while the defectors can always guarantee a small reward.

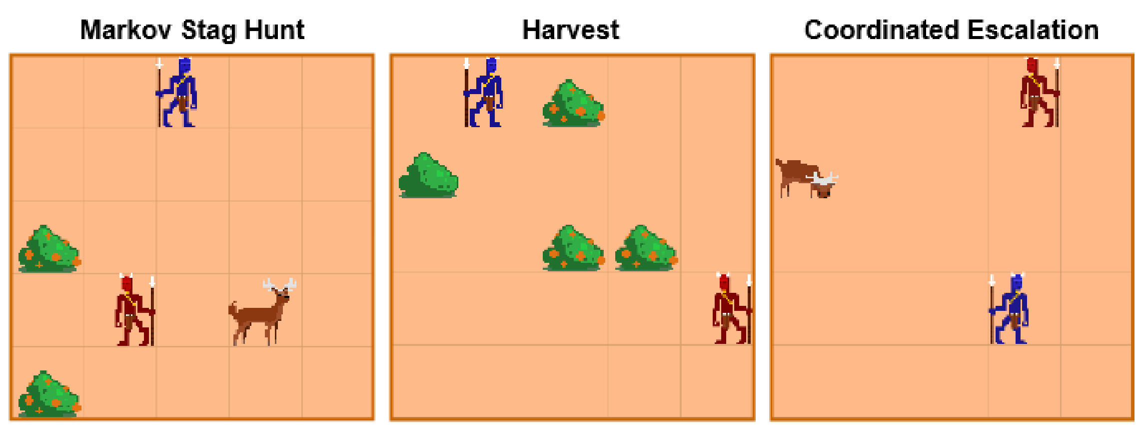
2. In the Harvest game, plants randomly spawn on the grid at each time step as long as there are less than *k* plants on the board. The original *k* is 4. The plants are spawned "young", and can turn into "mature" plants each time step with probability r_{mature}. Mature plants can die on each time step with probability r_{death}. Probabilities should be selected in a way such that each plant lives for *E* time steps on average. The original description has an *E* of 20. Agents can move over the plants to harvest them. A young plant yields 1 point to the harvester, but a mature plant rewards both players with 2 points. The prosocial strategy is thus to wait for the plants to mature before harvesting them. Much like in the original stag hunt, however, there is risk in waiting, as the other agent may defect and grab the plant before maturity.

3. In Coordinated Escalation, a marker appears in of the grid cells. If both agents step on the marker in tandem, they will each receive 1 point, at which point one of the adjacent cells will become the next marker. If the agents continue the streak by stepping onto the next marker together, they will again receive a point each. If at any moment the streak is broken by one of the agents stepping off the path, their

partner will receive a penalty calculated by multiplying the punishment multiplier p by the length of the streak T. The length of the streak is communicated to the agents as a part of the state observation.



(a) Stochastic Stag Hunt-like games as originally described in *"Prosocial learning agents solve generalized Stag Hunts better than selfish ones"*[11]



(b) Our implementation of the games for OpenAI Gym and PettingZoo.

Figure 4.1: Custom environment figures

4.1.2 Our Implementation

The environment is implemented using the OpenAI Gym interface. OpenAI Gym is a toolkit for reinforcement learning research that includes a growing collection of benchmark problems and exposes a common interface for agent interaction[22]. Being the de-facto standard for creating shareable reinforcement learning environments, OpenAI Gym is designed with creating new environments in mind. Furthermore, it was the clear choice for making our environment accessible to the maximum amount of people. However, Gym is not designed with multi-agent experiments in mind, and thus is not entirely appropriate for our context. Fortunately, the PettingZoo library was developed essentially as a multi-agent variant of Gym, sharing the same standard API, allowing for strong compatibility between the two[23]. Furthermore, our environment is provided in Gym and PettingZoo variants, allowing the user to select whichever is most appropriate for their context. In our research, we leverage both.

In the code, rendering is taken care of by the PyGame library, and load-bearing computation is executed using NumPy. The assets and textures were hand-made using Piskel.

Some additional engineering details:

1. The code was written with customization in mind, and each simulation variable is exposed as a parameter. Rewards and punishments can be freely configured. We leverage this to run experiments on high and low risk variants of the environments.

2. The matrix-form stag hunt is included as a fourth environment, although it is not provided in PettingZoo form.

3. It is enforced that each environment maintains high-level stag hunt properties. Simulations which do not obey the payoff matrix described in Figure 2.1.1 fail to instantiate.

4. The environments can be observed in two ways. Either as a 2D pixel array representing the PyGame render, or a 2D coordinate array representing the location of each entity. The pixel array can be rendered to the screen through PyGame, and the coordinate array can be printed to the terminal in a graphical format.

5. The PettingZoo environments are provided in parallel and raw variants to allow for diverse use cases.

4.2 Experiments

4.2.1 Experimental Methodology

To begin with, all of our experiments are arrangements in which a numerical model, the agent, learns a strategy by continuously interacting with the environment and altering their beliefs in response to maximize how much reward they earn on average. While the agents and environments vary between experiments, this general set-up is common to all of them. Our primary aim is to observe the learning process, see if the agents learn the cooperative or anti-social strategy, and consider what relationship seems to emerge between the simulation parameters and the behaviors learned by the players.

Specifically, we are using the risk configuration as the varying factor, to see what effect varying levels of risk have on what behaviors the agents settle on. Ultimately, we are paying attention to how the learning process is affected — since that is what is most relevant for eventually translating this into the human context.

Our experiment begins with a proof of concept stage where we test that our custom environment is behaving as expected and we are able to attain minimally meaningful results using simplified approaches. To achieve this, we train a pair of basic Q-table agents on the matrix and grid stag hunts. As the agent is simple, do not expect it to learn a strong policy in the grid stag hunt, and we are training the agent simply for testing purposes.

Once we confirm that the environment is functioning as expected, we conduct our main line of experimentation, which is training deep Q-learning models on the grid environments for a prolonged period of time. These nuanced models will offer a valuable opportunity to explore the dynamics of learning in the environment, and hopefully yield interesting results in the process.

4.2.2 Reading the Figures

In each figure, the x-axis corresponds to the number of iterations since the beginning of the experiment. The y-axis is the average reward for one or both of the players over a specified period of time. Steep curves mean a rapid change in strategy, and flat curves correspond to stagnated learning. Additionally, the lines jump up and down locally, as the agents are constantly experimenting and trying out new strategies which adds turbulence to the graphs.

4.2.3 Agent Structure

In our experiments, we will be using the basic Q-table set up for our proof of concept work, and DQN for our main experimentation. The DQN implementation is made possible with the Ray library — a general-purpose cluster-computing framework that enables simulation, training, and serving for RL applications[14][24][25]. Ray allows for high levels of interface flexibility, high throughput, and low latency in the experiments.

Successfully wiring up Ray and our environment together is a strong contribution of its own, as it enables an immense variety of possible experiments. Once the two are successfully wired up, there will be plentiful opportunities to uncover facts about the dynamics of the environment, as well as set up future research in the area.

Ray and RLlib

From a high level, Ray is a library that aims to provide a universal API for distributed computing. Modern AI applications, such as the one we are studying, continuously interact with their environment and learn from these interactions. This imposes new and demanding system requirements, both in terms of performance and flexibility[14]. To tackle this, Ray implements a unified interface that can express both task-parallel and actor-based computations, both being supported by a single dynamic execution engine[14]. On top of the foundational library, numerous other helpful tools have been built – such as Tune, a framework for model selection and training that streamlines hyper-parameter tuning during experimentation[26]. Most important to our experimentation is Ray RLlib, a library which provides scalable software primitives for reinforcement learning[24]. The library offers a collection of reference algorithms, which is where we get our DQN system[24]. When coupled with Tune and additional Ray helpers, RLlib becomes an incredibly powerful tool for running numerous complicated experiments in state-of-the-art speeds.

4.3 Proof of Concept

To begin with, we must confirm that our custom environment behaves as expected, and that our chosen agent architectures are capable of successfully interacting with it. To do this, we will have pairs of agents play each other on the matrix and grid stag hunts and observe the patterns of their interaction over a prolonged period of time. The goal here is to work bottom-up, beginning with the simple version of our model and then moving on to its complicated analogue once we have confidence that our set-up is generally functional. Accordingly, we do not expect consistent or even frequent prosocial convergence — just the evidence of learning taking place.

Parameter	Value
Learning Rate	.1
Learning Discount	.9
Epsilon Start Value	.99
Epsilon Decay Value	.9999
Epsilon Decay Start	1
Minimum Epsilon Value	.05

(a) Basic Q-Table agent learning parameters.

Parameter	Low Risk	High Risk
Cooperation	.45	.45
Defect Alone	.43	.40
Defect Together	.08	.20

(b) Matrix stag hunt environment configurations[13].

Parameter	Low Risk	High Risk
Forage Reward	1	1
Stag Reward	5	5
Mauling Punishment	-.5	-1

(c) Grid stag hunt environment risk configurations.

Figure 4.2: Experimental configuration tables

4.3.1 Matrix Stag Hunt

The results of the matrix stag hunt experiment, as can be seen in figure 4.3, are to be expected. In the low risk arrangement, the agents will converge to the cooperative equilibrium around 15 percent of the time. In the high risk arrangement, the agents fail to learn enough to converge to the prosocial equilibrium. This can be explained by referencing the simplicity of the agent architecture, and the low attraction basin of the prosocial choice in high-risk arrangements. The reason basic Q-table agents have difficulty learning the prosocial strategy here is because the signal is not salient enough for the learning algorithm to pick up on it. If the salience is artificially increased by manually making the prosocial reward significantly better, that can ensure prosocial convergence, at the cost of making the achievement rather uninteresting.

Given how our observations are consistent with past research, we can be rest assured that the environment is functioning as expected with even the simplest agent architecture being capable of learning it to a limited extent. With our testing phase complete, we are able to move on to the grid based environments with confidence that our interfacing is properly set up and we have access to the subject matter.

4.3.2 Grid Stag Hunt

The grid-based stag hunt is a much more complicated environment than the matrix form both conceptually and in terms of observations. In the matrix form, the observations are simply the last opponent action, whereas in the grid games they are the coordinates of all the game entities. Accordingly, we do not expect the agents to attain any meaningful results in terms of learning how to hunt the stag. If the set-up is functional, we can expect the agents to learn how to avoid the mauling punishment.

As can be seen in figures 4.6 and 4.7, our predictions seem to be correct. The basic Q-table agents did not make serious progress in the limited time allotted to them, but did begin slowly learning how to avoid being mauled by the stag, as can be seen in the positive slope of the reward graphs. Another hint towards this is that the slope is steeper in the high-risk graph, which is likely because the higher negative reinforcement for being mauled constitutes a stronger incentive to learn how to avoid it. The high amount of turbulence in the graphs is likely to be a consequence of the simple hyper parameter arrangement, where the agents explore a lot early on and after a certain point mostly only take actions already known to them. Furthermore, agents will always sometimes take random moves, which makes coordination on the prosocial choice significantly more risky since the agent has a small chance of deviating from the coordinated strategy each time they take a step on the grid. In our main line of experimentation, Ray takes care of hyper parameter tuning, meaning that the agents actively explore new avenues for the entirety of the simulation unlike here.

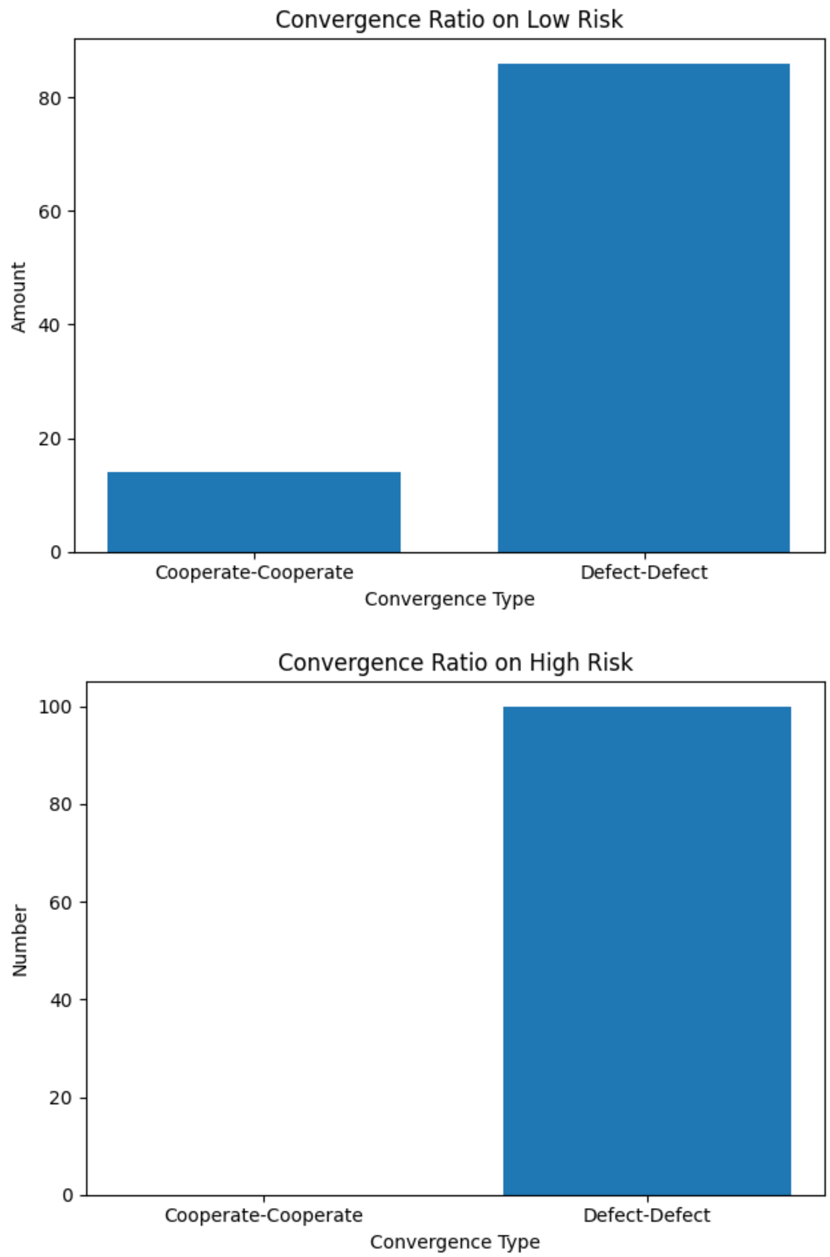


Figure 4.3: Convergence ratios in matrix stag hunt games played by basic Q-table agents.

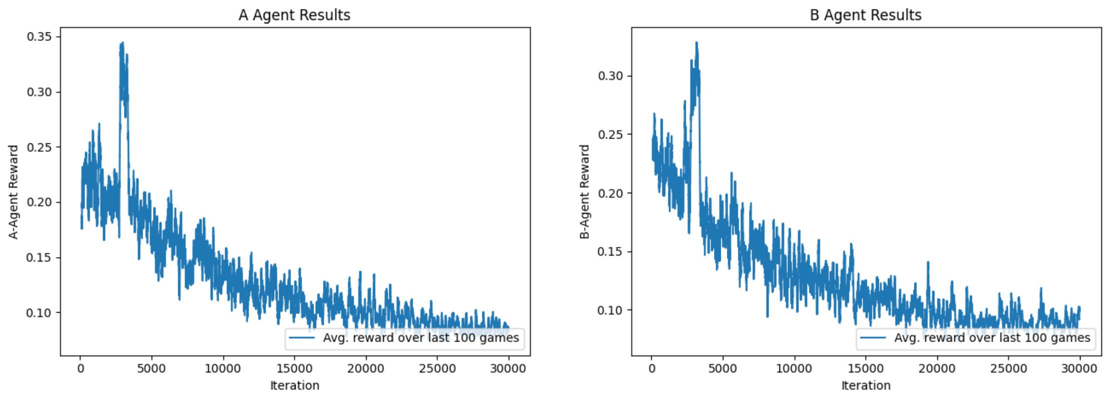


Figure 4.4: An illustration of a Defect-Defect convergence in the low risk matrix stag hunt environment between basic q-table agents.

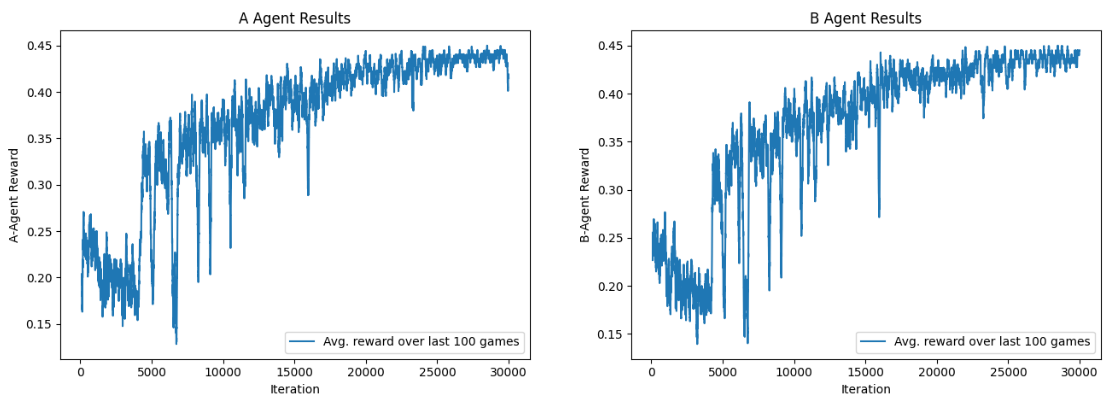


Figure 4.5: An illustration of a Cooperate-Cooperate convergence in the low risk matrix stag hunt environment between basic q-table agents.

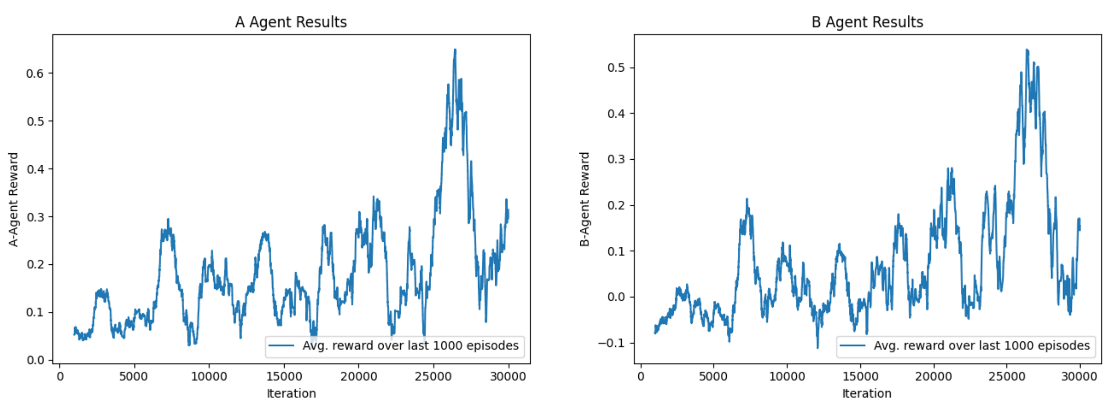


Figure 4.6: Low risk grid stag hunt proof of concept run.

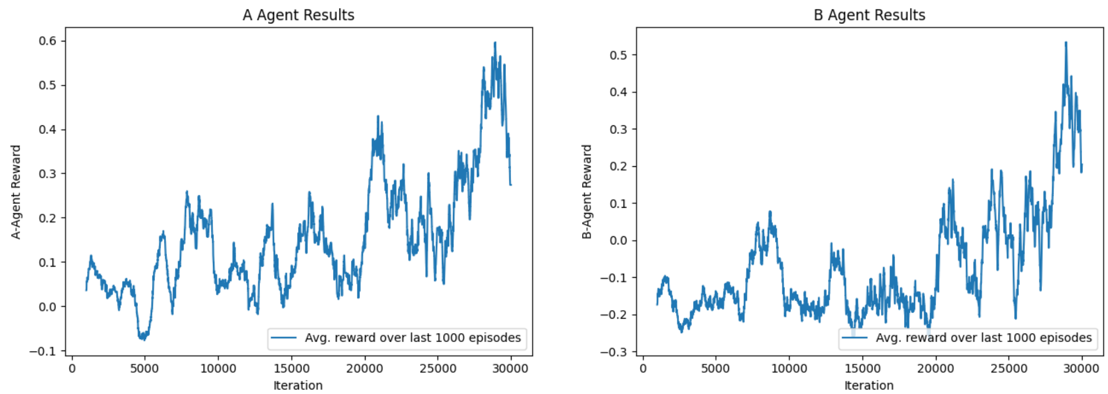


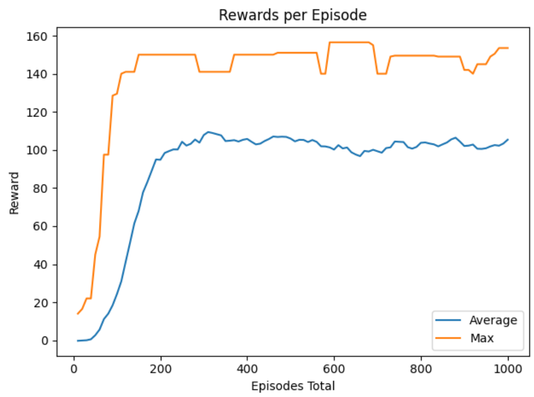
Figure 4.7: High risk grid stag hunt proof of concept run.

4.4 Main Learning Experiment

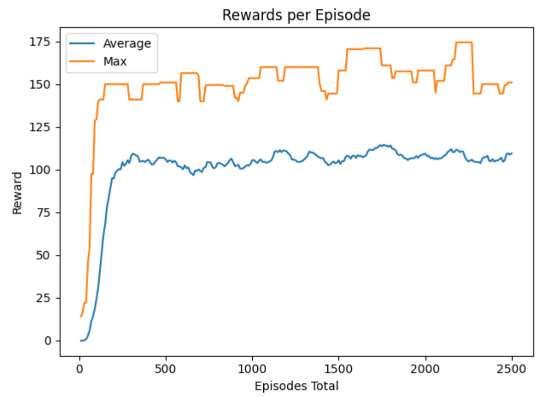
4.4.1 Low Risk Stag Hunt Experiment

As can be seen in figure 4.8, in the beginning of the low risk experiment, the agents learn what seems to be the optimal policy for their circumstance in the first 250 episodes. The majority of learning takes place in this initial period, and what follows after is a long period of stagnant exploration. This makes sense, as stumbling over plants is guaranteed to take place almost immediately, but it is a matter of chance when the agents accidentally coordinate on the stag and learn of its potential. The rewards stabilize at around 100 average per episode, and 150 max.

As we can observe in the first quarter of the experiment, once the agents settle on a policy, their innovation halts and the average reward does not change past some expected natural variance. In terms of actual behavior, this corresponds to the agents (1) learning how to keep moving to avoid the stag and (2) gathering the plants as they move around. Additionally, there is no way for the agents to average 1 reward per game step without interacting with the stag. Since plants yield 1 point, and it takes a few steps to get to a plant once it spawns, it is not possible to hit the 100 reward average simply through



(a) First 1000 episodes.



(b) First 2500 episodes.

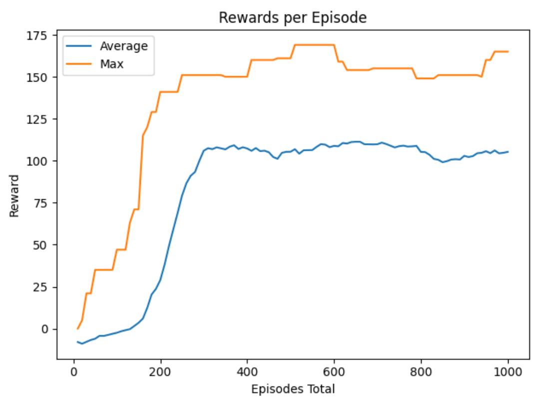
(c) Full run.

Figure 4.8: The low risk DQN stag hunt experiment.

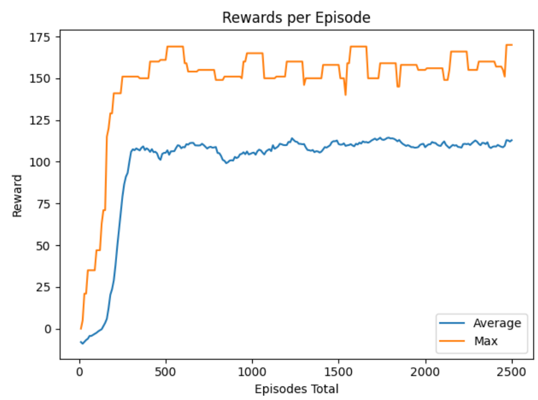
antisocial means. Therefore, we can conclude that the agents periodically hunt the stag. However, the agents seemingly do not seek the stag out as a part of the strategy, as the reward is too low for that to be true. The inference is thus that the agents have a defensive pact of sorts, pairing up if the stag approaches them while they are gathering, but doing their own thing otherwise. In terms of learning, this is reasonable as, given how agents must begin by harvesting plants, and the stag follows them around, they will continue being mauled until they learn how to avoid the stag. However, full avoidance would yield a low reward, incentivizing the agent to experiment. From there, it is easy to imagine that, as the two agents attempt to go for the same plant, and the stag attempts to attack one of them, they happen to be on the same cell. From that point, they will know that grouping up is a positive thing, but since the signal salience is still not that strong, doubtfully can infere mechanics behind why that action was successful.

In the full experiment graph, we see how the learning essentially does not progress after the initial climb. One can, however, notice a small positive incline as the agents do some minor optimizations. Perhaps, with enough time, the agents would eventually stumble onto the prosocial strategy, but given the reward arrangement that seems to be highly improbable.

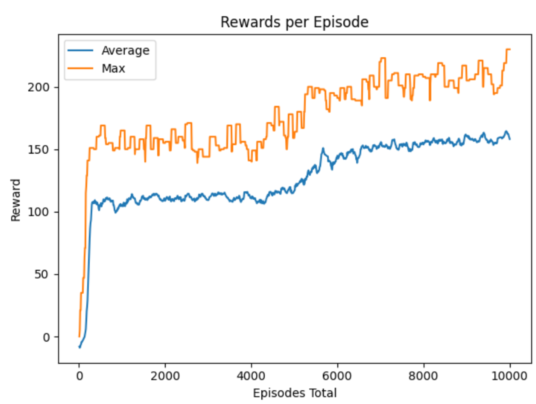
4.4.2 High Risk Stag Hunt Experiment



(a) First 1000 episodes.



(b) First 2500 episodes.



(c) Full run.

Figure 4.9: The high risk DQN stag hunt experiment.

As is observed in figure 4.9, the initial learning climb for the high risk experiment is evidently different from its low risk equivalent. The learning progresses at a much slower rate in the beginning, but speeds up quickly at around the 200th episode. This is likely due to the mauling punishment being higher, thus the agents being confused early on as they figure out how to avoid the stag.

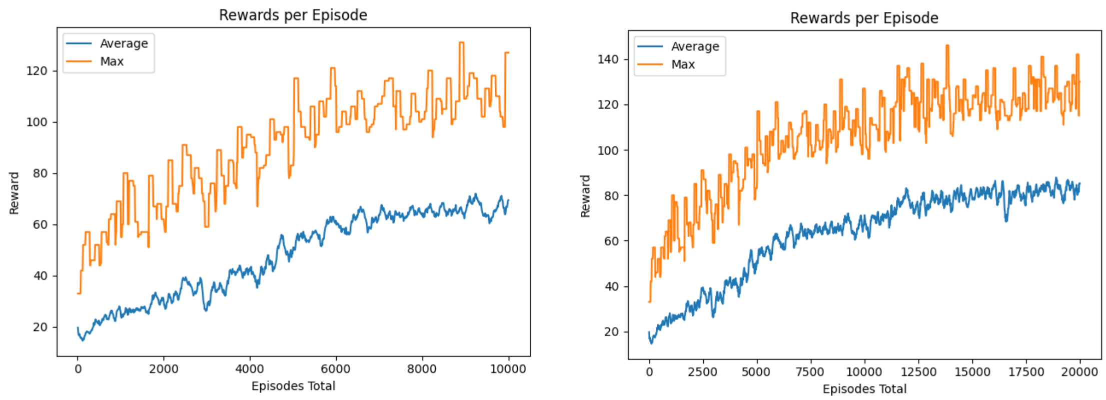
The graph of the first quarter of the high risk experiment is rather similar to its low risk equivalent. The agents, after settling on the policy they came up with in the first 250 episodes or so, stabilize and stop innovating.

Once we look at the full experiment, however, we see something fascinating. At the halfway point, the agents seem to have figured something out, as their rewards begin steadily climbing past what was achieved in the low risk experiment. The low risk agents, and the high risk agents in the first half, stabilize at around 120 average reward per episode, which corresponds to regular plant harvesting with occasional stag hunts. Furthermore, it seems that in the second half of the high risk run the agents seem to begin explicitly seeking out the stag! As the maximum reward achieved by the agents is higher than 200, we can decisively conclude that they are hunting down the stag as an explicit part of their strategy, since such a reward could not be achieved otherwise. This is interesting, as the risk seems to have made the prosocial equilibrium more desirable in this circumstance. We will be returning to this point in our discussion.

4.4.3 Harvest & Escalation

Additionally, to support our main line of experimentation, we trained DQN models on the harvest and escalation environments. The DQN models learned to play the games relatively well, their final performances being roughly equivalent to what one would expect from a human player.

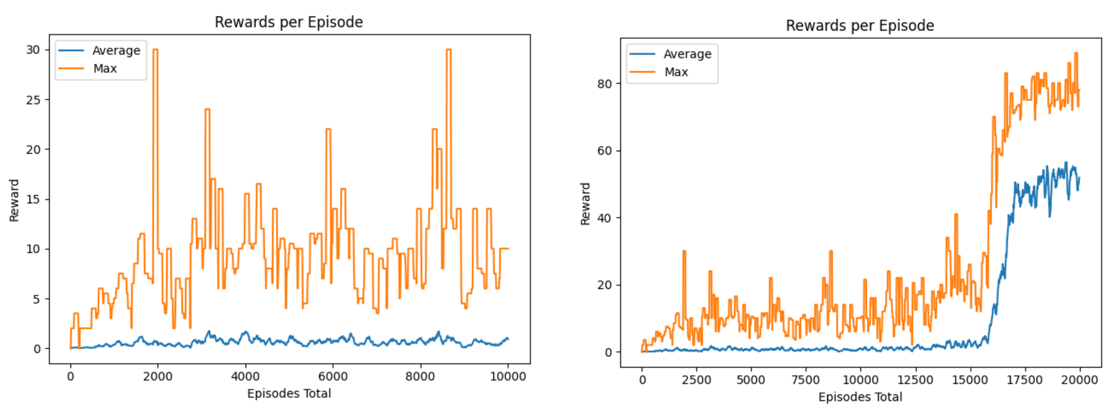
In harvest, we observe a slow and steady learning process that gradually gets better at harvesting plants and, once the benefits stagnate, realizes that their reward increases if they



(a) First 10000 episodes. Full 20000 episodes.

Figure 4.10: The default-settings DQN Harvest experiment.

wait before harvesting the plants. As we see in figure 4.10, the average rewards stabilize at around 80, while the maximum reward goes as high as 140. The steady learning rate makes sense in the context of opponent-awareness being less important in the harvest game. The agents can independently learn that waiting yields a higher reward, and therefore the higher reward does not require explicit coordination to the same extent as the other environments.



(a) First 10000 episodes. Full 20000 episodes.

Figure 4.11: The low risk DQN Escalation experiment.

In escalation, the learning barely makes any progress for the first three quarters of the experiment. As is seen in figure 4.11, the average rewards do not progress much past the original point, even as the maximum rewards hit upwards of 30. This, however, changes at around 15000 episode mark, at which point the learning rapidly takes off. Over the course of the next 5000 episodes, the model climbs to 50 average reward, and achieves the maximum reward of around 90. This translates to the agents having nearly fully solved the environment, as an individual episode is 100 game turns, which means the highest possible streak should be around the same number. Therefore it seems that the agents routinely maintain the streak for almost the entire episode, and play a game of chicken between each other at the end to see who gets the punishment.

Chapter 5

Discussion And Future Work

5.1 Discussion

5.1.1 Achievements

Environment Implementation

The biggest success of this project is the comprehensive implementation of stochastic stag hunt games using the OpenAI Gym and PettingZoo interfaces. While the thesis does not detail it, the development of the environment took a long time and involved numerous engineering problems that had to be solved. The code had to be (1) performant, to allow for efficient experimentation, (2) well documented and comprehensible, to make it truly accessible to the open-source reinforcement learning community, and (3) compatible with industry standard libraries to make it pragmatically usable by individuals without in-depth knowledge.

The first of these problems was solved through leveraging data structures and algorithm knowledge to optimize the load-bearing computations. Optimizations were achieved in part through algorithmic improvements, as well as using NumPy data structures under the hood to avoid the performance bottlenecks of native python. In terms of documentation, special attention was paid to make sure the project is well-commented, and all user-facing features and customizations are obvious and clearly communicated. Lastly, many hours were devoted to ensuring that the environment is fully compliant with industry specifications, and can be readily used with popular reinforcement learning libraries. The environments were checked using the built-in Gym and PettingZoo verification tools, and various tests were ran to ensure that edge cases do not break the game logic. The environment ended up interacting successfully with the Ray library in our experiments, and early explorations with Stable Baselines[27] yielded promising results as well. Furthermore, our environment can be an invaluable resource in conducting future experiments in the domain. While some of our other ambitions fell out of scope, the custom environment surpassed our expectations and we look forward to seeing how the repository evolves over time and what further attention it receives.

Cooperative Models

In terms of our models, we have successfully been able to train advanced reinforcement learning algorithms to perform well on our custom environments. While the success is varied, and there is an expected amount of randomness in terms of when and how often the optimal strategies are arrived at, the agents none the less achieve significant results on each of the three environments. As expected, the better strategies are those which exhibit prosocial behavior, much like past research has indicated[11]. The more interesting observation has to do with the relationship between risk and the likelihood of prosocial behavior being achieved. Risk was originally understood as the risk of the prosocial choice, the uncertainty being in whether or not the opponent would coordinate. Our analysis seems to indicate that it is also worthwhile to consider the risk of the anti-social choice, where the uncertainty lies in the missed opportunity if the prosocial choice is not sought out.

To relate this to our primary exploration of how to achieve stable prosocial behavior in multi-agent systems, there seems to be promise in explicitly engineering risk to incentivize the desired behavior. While changing the environment would, of course, be breaking the rules, playing around with the learning dynamics of the agents is fair game. Specifically, we would be interesting in exploring how reward signals can be pre-processed to heighten the risk of anti-social behavior while increasing the attraction basin of prosociality. For example, ants and bees, some of the most prosocial of animals, have intense punitive systems in place to eliminate individuals which defect from hive-level agreements[21]. A particularly promising path towards achieving something similar is that of peer gifting[9], which offers an intuitive way to achieve what we are looking for and confirms an additional time that this sort of arrangement would produce a positive effect.

5.1.2 Shortcomings

Social Modeling

Having excessively discussed the importance of modeling the social aspect of the cooperation problem, the lack of such in our experimentation is a major shortcoming of our work that is a consequence of time constraints and circumstances outside of our control. It is our hope that this work can be a starting point for future research which looks into the possibility of comprehensive social modeling and the use of said model in improving the learning process of member agents. Particularly, we are curious to see how social models can be used as tools for engineering risk structures which intentionally incentivize prosocial choices. We believe opportunity lies in the ideas discussed by Brian Skyrms[5], the concept of networked genetic algorithms[28], and principles conceptually related to peer gifting[9]. Similarly, having observed the success of neural networks in this context, we are highly optimistic about future work which researches how dynamic networks can be applied in novel ways to improve the efficiency of artificial intelligence approaches dealing with multi-agent domains.

Reproducibility

Reproducibility has historically been a significant obstacle to research in the behavioral sciences, and it is something we need to acknowledge in our work as well. Given how we study phenomena at least partially random at all levels, it is difficult to draw concrete causal lines between parameters and outcomes. With chaotic principles lying at the foundation of the systems we are studying, acknowledging this reality and incorporating it into our research methodology was a necessary step towards making meaningful progress. Statistical methods were incorporated to make our observations more aware of natural variance, and our writing reflects this. At the same time, the outcome for all of our experiments is highly dependent on the original arrangement of the system. While specific experiments can be reproduced using random number generator seeds, this does not get around the more essential issue of establishing which seeds produce which results. Our observations and conclusions are still meaningful, but should be approached from a naturalists perspective - we are observing the agents in their natural habitat and writing down what we see. It is not a guarantee that on the next visit we will be able to observe the same behaviors, or that particular environmental contexts hold direct causal influence over said observations. Our primary focus is on the patterns that emerge, and not on making concrete numerical conclusions.

5.2 Future Work

5.2.1 The Leviathan

The model, as currently defined, does not accurately represent a number of essential features of human interaction. The first of these is spatiality, which is the reality that social interactions take place on a network — frequent interactions with acquaintances, and rare run-ins with strangers. The second is that interactions are remembered, and these memories have an effect on future decision making — which we will refer to as temporality. Spatiality and temporality combined constitute ”socio-cultural arrangements” within the hypothetical model. The implementation of spatiality will manage the social network inhabited by the agents, therefore shaping who interacts with whom and when. The temporal implementation, on the other hand, will preserve knowledge over time — much like culture does in human society. These systemic arrangements are of particular interest to research as, unlike individual behavior and resource competition, these structures can be realistically changed in real life. Furthermore, let us outline the ways in which we believe these socio-cultural arrangements could be implemented.

Spatiality in the model can be implemented by using a graph to represent social relationships between the agents. Graphs are powerful data structures capable of efficiently representing convoluted social arrangements, which is essential long-term as the simulations are likely to get computationally complex. This structure would record which agents are related to one-another, as well as the strength of their connection. These connections will be used to establish who plays stag hunt with whom during a given iteration of the simulation. Such structures were used by numerous researchers in the work surveyed[11][28].A strong connection between agents will make it likely for them to be matched up, whereas a weak connection will make interactions improbable. One could think of these connections as similar to friendly and familial ties between humans.

In implementing temporality we are not concerned with making individual agents remember the past — rather the aim is for the collective to maintain some form of shared knowledge contributed to by the experience of each individual. In essence, the aim is to introduce a macro-scale learning process which improves learning at the individual level using information derived from the behavior of the system as a whole. A potential way to achieve this is by using a genetic algorithm coupled with peer gifting. These two processes would make high rewards more impactful through implicit reward-shaping, therefore increasing the advantages of cooperative behavior. The group will maintain shared knowledge through altering the structure of the social network and by preserving useful knowledge through reward shaping.

5.2.2 Genetic Algorithms

Based on the work surveyed in this thesis, we believe that next steps in the domain will have to do with establishing how evolutionary dynamics can be leveraged to improve agent-level learning. As was described extensively by Skyrms and Kropotkin, evolutionary principles constitute the foundation on which cooperation between individuals is built[5][21]. Consequently, the seeming lack of focus on population-level learning dynamics is a serious oversight of current research. In general, population-level systems are used to optimize the agent-level learning, but not to do the learning themselves. Consequently, what needs to be achieved is a system in which the population and agent level learning processes cooperate with one another to emerge a comprehensive meta-level learning dynamic. Numerous promising attempts at this, or something conceptually similar, have been made, but the work is still in its early stages and many avenues are left unexplored[29][7][30][31][32][33][34][35]. Furthermore, we will briefly overview genetic algorithms, discuss their applicability in the domain, and propose a promising future avenue for create a comprehensive social model of learning.

Formal Description

A genetic algorithm (GA) is a way of solving problems by simulating natural selection [28]. Unlike standard learning processes that iteratively improve a single solution, a GA creates a collection of potential solutions, called a population, and uses them to explore the problem space from different angles[4]. When we speak of learning in a population of agents, we refer to the change in the constitution and behavior of that population over a period of time[7]. Within the population, an individual is encoded as a gene sequence, called a chromosome. A chromosome is composed of a fixed number of genes, with each gene taking on one the possible values, called alleles[4].

The approach is built around the process of testing individuals to see how well they fare at the problem and generating a new population by using the genetic material of good solutions to create offspring [28]. This procedure, by continuously incentivizing good performance, gradually evolves the population to solve the task at hand.

The two main components of a genetic algorithm are the fitness function and the genetic operators. The fitness function calculates how good a particular specimen is at solving the task. We call this metric fitness. It is very similar in function to school tests - a small assessment, the score on which is meant to measure a learners progress. The fitness function is generally the most computationally expensive part of a GA, as a function which does not accurately represent the problem will prevent the algorithm from achieving meaningful results. Furthermore, designing a genetic algorithm is essentially synonymous with designing a fitness function which accurately estimates performance on the problem being solved.

The genetic operands are two processes - crossover and mutation. Crossover is what decides who reproduces with whom. The mechanics of this can range from random match ups weighed by fitness, to graph based approaches that emulate real-life population dynamics[28]. The only thing strictly required is that the process consistently rewards individuals with high fitness. Mutation introduces variability to the genetic material, preventing the evolutionary process from stagnating. Genetic alteration during the reproductive step is the main exploratory mechanism at the algorithms disposal. Without it, the method would have no means of making progress towards solving the problem.

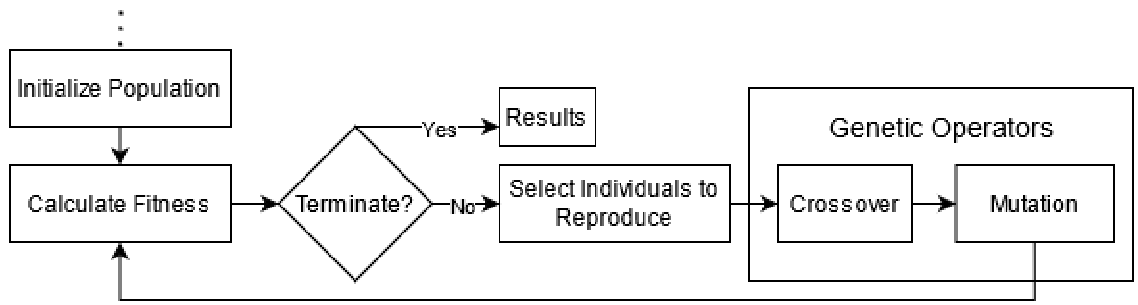


Figure 5.1: The standard structure of a Genetic algorithm.

Fitness Heuristics

Genetic algorithms belong to the larger class of hill-climbing algorithms. This class of algorithms represents problems as multi-dimensional search spaces. In these spaces, height corresponds to how "good" a solution is, and the other dimensions identify what that particular solution is within the full spectrum of possible solutions. When visualized in two dimensions, this appears as a series of hills. By beginning with some arbitrary solution, and by making incremental changes, the algorithm "climbs" to higher points.

In the context of GAs, the search space is the fitness landscape, with an individual's fitness being represented by their position on said landscape[4]. The goal of any hill-climbing algorithm is to find the highest peak, formally known as the global maximum, while avoiding getting stuck on local maxima. The main advantage of GAs over single-solution hill-climbing algorithms is how they sample the search space from multiple points, thus minimizing the risk of getting stuck on a local peak.

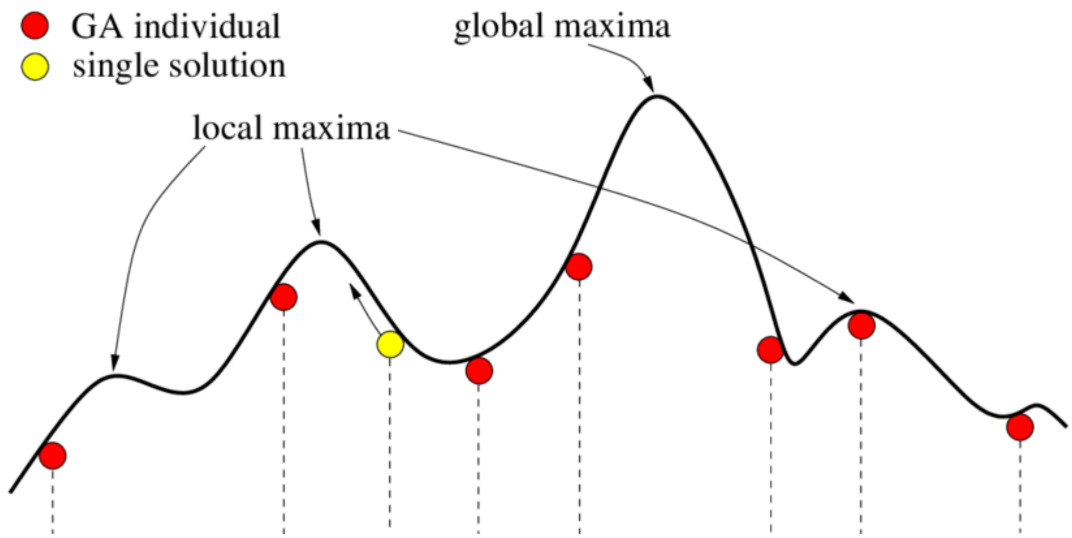


Figure 5.2: A hypothetical search space being explored by a single solution algorithm (yellow) and a genetic algorithm (red). The single-solution agent has a high chance of getting stuck on a local peak based on its starting point. Since the GA initially samples the space from different points, this risk is significantly less pronounced[4].

Genetic Operators

Within the function of a genetic algorithm, genetic operators are what makes it possible to search the problem space and eventually arrive at a solution. Amongst the two operators, crossover is conceptually responsible for ensuring that each consecutive generation moves up the gradient on the fitness landscape. This is achieved by rewarding high-fitness individuals with additional reproductive rights when picking mating pairs, and through the specific mechanics of how parent genetics are combined to produce a child. When set up correctly, this ensures that the genetic material of well-performing individuals is propagated through the offspring generation, without the loss of alternative, potentially-promising, genetics.

Mutation, on the other hand, is responsible for protecting the population from getting stuck on local peaks on the fitness landscape. By continuously introducing a random ele-

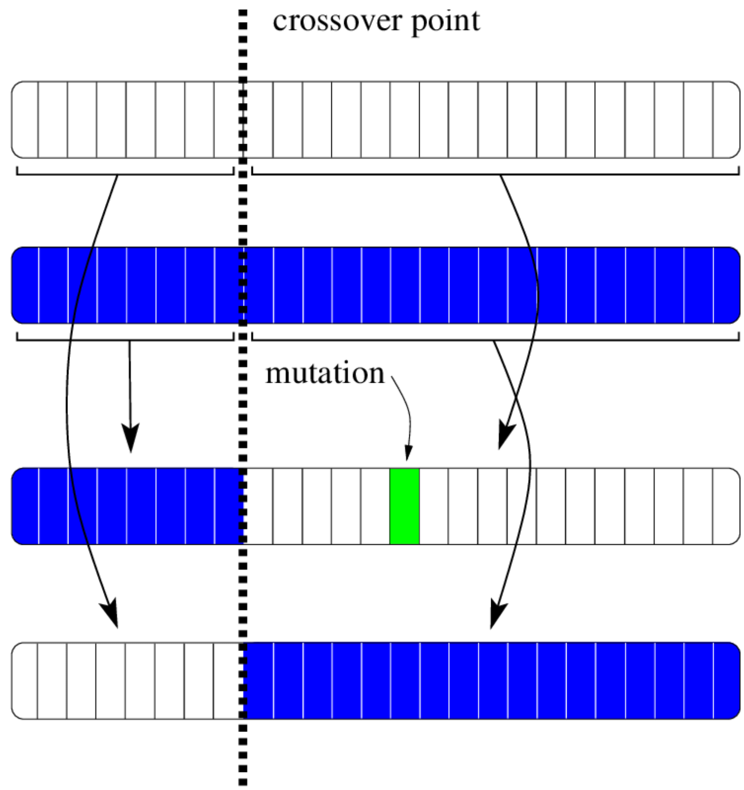


Figure 5.3: An example crossover process in which two children are generated from two parents by splicing their genes at a random point. Mutation then occurs through the random swap of one of the alleles in the children gene sequence.[4]

ment into the gene pool, mutation guarantees that the algorithm is always looking for new ways to approach the problem. Without it, the population will simply settle on whatever first strategy it discovers and get stuck as it has no means to further explore. Too much mutation is similarly a bad thing, as excessive genetic permutation has the potential to erase useful genetics from the gene pool and sabotage the work done by crossover. Finding a middle ground is key to an efficient genetic algorithm.

5.2.3 Networked Genetic Algorithms

Motivation

Past research has confirmed that population network structure can be tuned to improve GA performance, but has not yet explored how this can be done at scale[28]. It is our view that the answer to this question can be found by looking at the success achieved by algorithms which leverage emergent properties in similar contexts[6][9][8]. Specifically, neural networks serve as a shining example of how pragmatically applicable emergence can be in the context of computing[29]. Furthermore, we see an opportunity to leverage the emergent properties of population network structure as a means of achieving stable prosocial behavior in multi-agent environments. The goal is to create a setting in which prosocial behavior is achieved as a consequence of individual behavior and not the authority of some governing entity handing out orders. This is known to be possible based on empirical observation and established academic fact, therefore the question is that of specific implementation[21]. We believe progress in the area will follow a path similar to that of neural networks, where functionality is gradually achieved through individual innovations that take the tool from proof of concept to industry-grade applicability.

Theory-Crafting

The standard genetic algorithm does not necessarily contain a spatial aspect, as it assumes that any two solutions can mate[28]. This, however, does not reflect how ”in nature and social contexts, social networks can condition the likelihood that two individuals mate”[28]. Furthermore, we will be experimenting with a novel type of genetic algorithms called Networked Genetic Algorithms (NGA’s) which imbue the population with a network structure. The population network structure will be represented as a weighed undirected graph, where the vertexes are individual solutions, edges mean the two solutions can mate, and the weights (how thick the edges are) correspond to how likely the mating is.

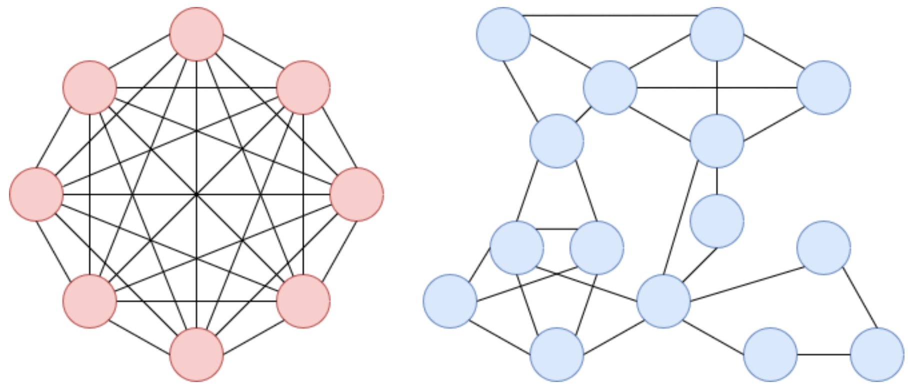


Figure 5.4: The population of a standard genetic algorithm (left) and what one would expect to see in a natural population graph (right).

Initial research has demonstrated that population networks characterised by intermediate density and low average shortest path length significantly outperform the standard complete-network GA [28]. The same research suggests that the population network structure, like the other GA parameters, could be tuned to improve performance[28]. Consequently, we believe the learning performance of the agents can be improved by letting them tune their population structure. Specifically, this could be done by giving the agents additional actions allowing them to (1) form, (2) break, and (3) tune their connections. Formally, these connections should be understood to be a part of the agents environment.

The motivation behind letting the agents tune their network connections is intuitively clear. In nature, population networks are highly dynamic structures that are constantly changing in response to environmental conditions. Humans, animals, and plants all have unique adaptations which shape their population network. Our hope is that empowering standard genetic algorithms with an additional learning process in the form of a dynamically restructuring population network, will create a population more likely to discover the global fitness maxima. If such structures are possible, and can be efficiently leveraged by large collective of agents, this would have immense implications for multi-agent systems theory and human organization as a whole.

Chapter 6

Conclusion

6.1 Summary

To summarize our work, we pondered the dynamics of cooperative behavior by exploring its abstract representation in the game of stag hunt. To do so, we began by creating custom game environments to serve as a playground for our experiments. With reinforcement learning as our main tool, we then created agents which learned how to coordinate in the environments by developing a sense of trust between each other. Not all agents were successful however, and by relating initial configuration parameters to simulation outcomes we were able to put forward some explanations for how agents converge to one equilibrium over the other. As was hinted to us by past research, and confirmed through our experiments, risk is the essential part of the arrangement. Evidently, when faced with stag hunt-like interactions, agents will settle on a strategy based on the inherent risk of coordinating. This risk can pull in both directions - towards the antisocial equilibrium if coordination is difficult for the reward it achieves, or the prosocial equilibrium if too much is lost by failing to be social. In light of these observations, we see a promising path towards achieving stable prosocial behavior through engineering population-level mechanisms which heighten anti-social risk while minimizing the risk of social coordination.

6.1.1 Next Steps

In terms of implementing these population-level risk-engineering mechanics, a promising avenue is seen in genetic algorithms, specifically those which leverage population graphs and reward propagation. The optimism is fueled by the success of these approaches in the original papers describing them, as well as the large body of philosophical analysis conducted in the area. Notably, the works of Skyrms and Kropotkin offered us a strong conceptual foundation on which to build such an implementation, and detailed numerous ways in which evolutionary dynamics can be used to guarantee prosocial outcomes. We conclude on an enthusiastic prediction that multi-agent research is still in its early stages, and academia at large has not yet fully explored the ways in which population algorithms can be used to break new grounds in cognitive science. If this new perspective is adopted, we are confident that major breakthroughs are to follow.

6.2 Closing Thoughts

Cognitive systems are deeply complicated. Our relationship to these systems is profoundly intimate - everywhere we turn our eyes to, we see thinking things that we must, frequently, communicate with. People, institutions, corporations, are all agents, striving to achieve what is best for them while not being taken advantage by another. To create a better world, society must orchestrate these infinite manifolds of the mind to a single symphony. For us to make these hopes tangible, we must understand what is required from individuals and the collective as a whole. And while there is enormous complexity in the specifics of this implementation, the foundation is rather simple. Do unto others as you would have them do to you. If we are all to hunt the stag, there are no horns too big to fell.

Bibliography

[1] Sulawesi art: Animal painting found in cave is 44,000 years old, Dec 2019. vii, 6

[2] Phi Le Nguyen, Van La, Anh Nguyen, Hùng Nguyen, and Kien Nguyen. An on-demand charging for connected target coverage in wrsns using fuzzy logic and q-learning. *Sensors*, 21:5520, 08 2021. vii, 18

[3] Heunchul Lee, Maksym Girnyk, and Jaeseong Jeong. Deep reinforcement learning approach to mimo precoding problem: Optimality and robustness. 06 2020. vii, 19

[4] Faustino Gomez and Risto Miikkulainen. Robust non-linear control through neuroevolution. 11 2002. viii, 54, 55, 56, 57

[5] Brian Skyrms. *The Stag Hunt and the Evolution of Social Structure*. Cambridge University Press, 2003. 5, 20, 22, 23, 24, 25, 26, 27, 28, 50, 53

[6] Trapit Bansal, Jakub Pachocki, Szymon Sidor, Ilya Sutskever, and Igor Mordatch. Emergent complexity via multi-agent competition. *CoRR*, abs/1710.03748, 2017. 8, 22, 58

[7] Yoav Shoham and Kevin Leyton-Brown. *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press, Cambridge, UK, 2009. 8, 10, 12, 14, 16, 17, 31, 53, 54

[8] David Silver, Satinder Singh, Doina Precup, and Richard S. Sutton. Reward is enough. *Artificial Intelligence*, 299:103535, 2021. 8, 13, 14, 15, 58

[9] Woodrow Z. Wang, Mark Beliaev, Erdem Biyik, Daniel A. Lazar, Ramtin Pedarsani, and Dorsa Sadigh. Emergent prosociality in multi-agent games through gifting. *CoRR*, abs/2105.06593, 2021. 9, 22, 50, 58

[10] Joel Z. Leibo, Vinícius Flores Zambaldi, Marc Lanctot, Janusz Marecki, and Thore Graepel. Multi-agent reinforcement learning in sequential social dilemmas. *CoRR*, abs/1702.03037, 2017. 10, 11, 19

[11] Alexander Peysakhovich and Adam Lerer. Prosocial learning agents solve generalized stag hunts better than selfish ones. *CoRR*, abs/1709.02865, 2017. 11, 12, 30, 31, 33, 49, 52

[12] Michael L Littman. Markov games as a framework for multi-agent reinforcement learning. In *Machine learning proceedings 1994*, pages 157–163. Elsevier, 1994. 11

[13] Neil Bearden. The evolution of inefficiency in a simulated stag hunt. *Behavior Research Methods, Instruments, Computers*, 33:124–129, 05 2001. 13, 38

[14] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging ai applications, 2017. 13, 14, 36, 37

[15] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. In *Machine Learning*, pages 279–292, 1992. 16

[16] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013. 18

[17] Andrei Cristian Nica, Tudor Berariu, Florin Gogianu, and Adina Magda Florea. Learning to maximize return in a stag hunt collaborative scenario through deep reinforcement learning. In *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 188–195, 2017. 19

[18] Shona Duguid, Emily Wyman, Anke Schirmer, Katharina Herfurth-Majstorovic, and Michael Tomasello. Coordination strategies of chimpanzees and human children in a stag hunt game. *Proceedings. Biological sciences / The Royal Society*, 281, 12 2014. 20

[19] Alex McAvoy, Julian Kates-Harbeck, Krishnendu Chatterjee, and Christian Hilbe. Evolutionary (in)stability of selfish learning in repeated games, 2021. 20

[20] Dung Nguyen, Svetha Venkatesh, Phuoc Nguyen, and Truyen Tran. Theory of mind with guilt aversion facilitates cooperative reinforcement learning. *CoRR*, abs/2009.07445, 2020. 21

[21] Peter Kropotkin. *Mutual Aid: A Factor in Evolution*. PENGUIN BOOKS, 2022. 21, 27, 28, 29, 50, 53, 58

[22] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016. 30, 34

[23] J. K. Terry, Benjamin Black, Nathaniel Grammel, Mario Jayakumar, Ananth Hari, Ryan Sullivan, Luis Santos, Rodrigo Perez, Caroline Horsch, Clemens Dieffendahl, Niall L. Williams, Yashas Lokesh, and Praveen Ravi. Pettingzoo: Gym for multi-agent reinforcement learning, 2020. 30, 34

[24] Eric Liang, Richard Liaw, Philipp Moritz, Robert Nishihara, Roy Fox, Ken Goldberg, Joseph E. Gonzalez, Michael I. Jordan, and Ion Stoica. Rllib: Abstractions for distributed reinforcement learning, 2017. 36, 37

[25] Eric Liang, Zhanghao Wu, Michael Luo, Sven Mika, Joseph E. Gonzalez, and Ion Stoica. Rllib flow: Distributed reinforcement learning is a dataflow problem, 2020. 36

[26] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E. Gonzalez, and Ion Stoica. Tune: A research platform for distributed model selection and training, 2018. 37

[27] 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. 49

[28] Aymeric Vié. Population network structure impacts genetic algorithm optimisation performance. *CoRR*, abs/2104.04254, 2021. 50, 52, 54, 55, 58, 59

[29] Rune Krauss, Marcel Merten, Mirco Bockholt, and Rolf Drechsler. Alf – a fitness-based artificial life form for evolving large-scale neural networks, 2021. 53, 58

[30] John D. Co-Reyes, Yingjie Miao, Daiyi Peng, Esteban Real, Sergey Levine, Quoc V. Le, Honglak Lee, and Aleksandra Faust. Evolving reinforcement learning algorithms, 2021. 53

[31] Jörg Stork, Martin Zaefferer, Nils Eisler, Patrick Tichelmann, Thomas Bartz-Beielstein, and A. E. Eiben. Behavior-based neuroevolutionary training in reinforcement learning. *CoRR*, abs/2105.07960, 2021. 53

[32] Daan Klijn and A. E. Eiben. A coevolutionary approach to deep multi-agent reinforcement learning. *CoRR*, abs/2104.05610, 2021. 53

[33] Nicholas Guttenberg and Marek Rosa. Bootstrapping of memetic from genetic evolution via inter-agent selection pressures, 2021. 53

[34] Ahmed Hallawa, Anil Yaman, Giovanni Iacca, and Gerd Ascheid. A framework for knowledge integrated evolutionary algorithms. *CoRR*, abs/2103.16897, 2021. 53

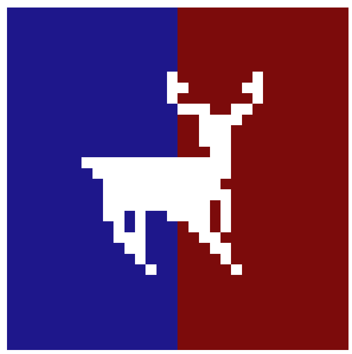
[35] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning, 2017. 53

[36] Ming Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *In Proceedings of the Tenth International Conference on Machine Learning*, pages 330–337. Morgan Kaufmann, 1993.

[37] Daan Bloembergen, Steven Jong, and Karl Tuyls. Lenient learning in a multiplayer stag hunt. pages 44–50, 11 2011.

Appendix A

Source Code



A.1 src/

Listing A.1: `entities.py`

```
import os

from pygame import image, Rect, transform
from pygame.sprite import DirtySprite

base_path = os.path.dirname(os.path.dirname(__file__))
entity_path = os.path.join(base_path, "assets/entities")

sprite_dict = {
    "a_agent": os.path.join(entity_path, "blue_agent.png"),
    "b_agent": os.path.join(entity_path, "red_agent.png"),
    "stag": os.path.join(entity_path, "stag.png"),
    "plant": os.path.join(entity_path, "plant_fruit.png"),
    "plant_young": os.path.join(entity_path, "plant_no_fruit.png"),
    "mark": os.path.join(entity_path, "mark.png"),
    "mark_active": os.path.join(entity_path, "mark_active.png"),
    "game_icon": os.path.join(base_path, "assets/icon.png"),
}

TILE_SIZE = 32


def load_img(path):
    """
```

```
    :param path: Location of the image to load.
    :return: A loaded sprite with the pixels formatted for performance.
    """
    return image.load(path).convert_alpha()


def get_gui_window_icon():
    """
    :return: The icon to display in the render window.
    """
    return image.load(sprite_dict["game_icon"])


class Entity(DirtySprite):
    def __init__(self, entity_type, location):
        """
        :param entity_type: String specifying which sprite to load from the sprite dictionary (sprite_dict)
        :param location: [X, Y] location of the sprite. We calculate the pixel position by multiplying it by cell_sizes
        """
        DirtySprite.__init__(self)
        self._image = transform.scale(  # Load, scale and record the entity sprite
            load_img(sprite_dict[entity_type]), (TILE_SIZE, TILE_SIZE)
        )
        self.update_rect(location)  # do the initial rect update

    def update_rect(self, new_loc):
        """
        :param new_loc: New [X, Y] location of the sprite.
        :return: Nothing, but the sprite updates it's state so it is rendered in the right place next iteration.
        """
        self.rect = Rect(
            new_loc[0] * TILE_SIZE, new_loc[1] * TILE_SIZE, TILE_SIZE, TILE_SIZE
        )

    @property
    def IMAGE(self):
        return self._image


class HarvestPlant(Entity):
    def __init__(self, location):
        Entity.__init__(self, location=location, entity_type="plant")
        self._image_young = transform.scale(
            load_img(sprite_dict["plant_young"]), (TILE_SIZE, TILE_SIZE)
        )

    @property
    def IMAGE_YOUNG(self):
        return self._image_young


class Mark(Entity):
    def __init__(self, location):
        Entity.__init__(self, location=location, entity_type="mark")
        self._image_active = transform.scale(
            load_img(sprite_dict["mark_active"]), (TILE_SIZE, TILE_SIZE)
        )

    @property
    def IMAGE_ACTIVE(self):
        return self._image_active
```

Listing A.2: `utils.py`

```
from itertools import product
from random import choice
from sys import stdout

from numpy import all, full, zeros, uint8

symbol_dict = {"hunt": ("S", "P"), "harvest": ("p", "P"), "escalation": "M"}

A_AGENT = 0  # base
B_AGENT = 1

STAG = 2  # hunt
PLANT = 3

Y_PLANT = 2  # harvest
M_PLANT = 3

MARK = 2  # escalation


def print_matrix(obs, game, grid_size):
    if game == "escalation":
        matrix = full((grid_size[0], grid_size[1], 3), False, dtype=bool)
    else:
        matrix = full((grid_size[0], grid_size[1], 4), False, dtype=bool)

    if game == "hunt":
        a, b, stag = (obs[0], obs[1]), (obs[2], obs[3]), (obs[4], obs[5])
        matrix[a[0]][a[1]][A_AGENT] = True
        matrix[b[0]][b[1]][B_AGENT] = True
        matrix[stag[0]][stag[1]][STAG] = True
        for i in range(6, len(obs), 2):
            plant = obs[i], obs[i + 1]
            matrix[plant[0]][plant[1]][PLANT] = True

    elif game == "harvest":
        a, b = (obs[0], obs[1]), (obs[2], obs[3])
        matrix[a[0]][a[1]][A_AGENT] = True
        matrix[b[0]][b[1]][B_AGENT] = True

        for i in range(4, len(obs), 3):
            plant_age = M_PLANT if obs[i + 2] else Y_PLANT
            matrix[obs[i]][obs[i + 1]][plant_age] = True

    elif game == "escalation":
        a, b, mark = (obs[0], obs[1]), (obs[2], obs[3]), (obs[4], obs[5])
        matrix[a[0]][a[1]][A_AGENT] = True
        matrix[b[0]][b[1]][B_AGENT] = True
        matrix[mark[0]][mark[1]][MARK] = True

    symbols = symbol_dict[game]

    stdout.write("

    \n")
    for row in matrix:
        stdout.write("      ")
        for col in row:
            cell = []
            cell.append("A") if col[0] == 1 else cell.append(" ")
            cell.append("B") if col[1] == 1 else cell.append(" ")
            cell.append(symbols[0]) if col[2] == 1 else cell.append(" ")
            if game != "escalation":
                cell.append(symbols[1]) if col[3] == 1 else cell.append(" ")
            else:
```

```
                cell.append(" ")
            stdout.write("".join(cell) + "  ")
        stdout.write("    ")
        stdout.write("\n")
    stdout.write("

    \n\r")
    stdout.flush()


def overlaps_entity(a, b):
    """
    :param a: (X, Y) tuple for entity 1
    :param b: (X, Y) tuple for entity 2
    :return: True if they are on the same cell, False otherwise
    """
    return (a == b).all()


def place_entity_in_unoccupied_cell(used_coordinates, grid_dims):
    """
    Returns a random unused coordinate.
    :param used_coordinates: a list of already used coordinates
    :param grid_dims: dimensions of the grid so we know what a valid coordinate is
    :return: the chosen x, y coordinate
    """
    all_coords = list(product(list(range(grid_dims[0])), list(range(grid_dims[1]))))

    for coord in used_coordinates:
        for test in all_coords:
            if all(test == coord):
                all_coords.remove(test)

    return choice(all_coords)


def spawn_plants(grid_dims, how_many, used_coordinates):
    new_plants = []
    for x in range(how_many):
        new_plant = zeros(2, dtype=uint8)
        new_pos = place_entity_in_unoccupied_cell(
            grid_dims=grid_dims, used_coordinates=new_plants + used_coordinates
        )
        new_plant[0], new_plant[1] = new_pos
        new_plants.append(new_plant)
    return new_plants


def respawn_plants(plants, tagged_plants, grid_dims, used_coordinates):
    for tagged_plant in tagged_plants:
        new_plant = zeros(2, dtype=uint8)
        new_pos = place_entity_in_unoccupied_cell(
            grid_dims=grid_dims, used_coordinates=plants + used_coordinates
        )
        new_plant[0], new_plant[1] = new_pos
        plants[tagged_plant] = new_plant
    return plants
```

A.1.1 games/

Listing A.3: `abstractgridgame.py`

```
from abc import ABC
```

```python

from numpy import zeros, uint8, array
from numpy.random import choice

# Possible Moves
LEFT = 0
DOWN = 1
RIGHT = 2
UP = 3
STAND = 4


class AbstractGridGame(ABC):
    def __init__(self, grid_size, screen_size, obs_type, enable_multiagent):
        """
        :param grid_size: A (W, H) tuple corresponding to the grid dimensions. Although W=H is expected, W!=H works also
        :param screen_size: A (W, H) tuple corresponding to the pixel dimensions of the game window
        :param obs_type: Can be 'image' for pixel-array based observations, or 'coords' for just the entity coordinates
        :param enable_multiagent: Boolean signifying if the env will be used to train multiple agents or one.
        """
        if screen_size[0] * screen_size[1] == 0:
            raise AttributeError(
                "Screen size is too small. Please provide larger screen size."
            )

        # Config
        self._renderer = None  # placeholder renderer
        self._obs_type = obs_type  # record type of observation as attribute
        self._grid_size = grid_size  # record grid dimensions as attribute
        self._enable_multiagent = enable_multiagent

        self._a_pos = zeros(
            2, dtype=uint8
        )  # create empty coordinate tuples for the agents
        self._b_pos = zeros(2, dtype=uint8)

    """
    Observations
    """

    def get_observation(self):
        """
        :return: observation of the current game state
        """
        return (
            self.RENDERER.update()
            if self._obs_type == "image"
            else self._coord_observation()
        )

    def _coord_observation(self):
        return array(self.AGENTS)

    def _flip_coord_observation_perspective(self, a_obs):
        """
        Transforms the default observation (which is "from the perspective of agent A" as it's coordinates are in the
        first index) into the "perspective of agent B" (by flipping the positions of the A and B coordinates in the
        observation array)
        :param a_obs: Original observation
        :return: Original observation, from the perspective of agent B
```

```
        """
        ax, ay = a_obs[0], a_obs[1]
        bx, by = a_obs[2], a_obs[3]

        b_obs = a_obs.copy()
        b_obs[0], b_obs[1] = bx, by
        b_obs[2], b_obs[3] = ax, ay
        return b_obs

    """
    Movement Methods
    """

    def _move_dispatcher(self):
        """
        Helper function for streamlining entity movement.
        """
        return {
            LEFT: self._move_left,
            DOWN: self._move_down,
            RIGHT: self._move_right,
            UP: self._move_up,
            STAND: self._stand,
        }

    def _move_entity(self, entity_pos, action):
        """
        Move the specified entity
        :param entity_pos: starting position
        :param action: which direction to move
        :return: new position tuple
        """
        return self._move_dispatcher()[action](entity_pos)

    def _move_agents(self, agent_moves):
        self.A_AGENT = self._move_entity(self.A_AGENT, agent_moves[0])
        self.B_AGENT = self._move_entity(self.B_AGENT, agent_moves[1])

    def _reset_agents(self):
        """
        Place agents in the top left and top right corners.
        :return:
        """
        self.A_AGENT, self.B_AGENT = [0, 0], [self.GRID_W - 1, 0]

    def _random_move(self, pos):
        """
        :return: a random direction
        """
        options = [LEFT, RIGHT, UP, DOWN]
        if pos[0] == 0:
            options.remove(LEFT)
        elif pos[0] == self.GRID_W - 1:
            options.remove(RIGHT)

        if pos[1] == 0:
            options.remove(UP)
        elif pos[1] == self.GRID_H - 1:
            options.remove(DOWN)

        return choice(options)

    def _seek_entity(self, seeker, target):
        """
        Returns a move which will move the seeker towards the target.
        :param seeker: entity doing the following
```

```
        :param target: entity getting followed
        :return: up, left, down or up move
        """
        seeker = seeker.astype(int)
        target = target.astype(int)
        options = []

        if seeker[0] < target[0]:
            options.append(RIGHT)
        if seeker[0] > target[0]:
            options.append(LEFT)
        if seeker[1] > target[1]:
            options.append(UP)
        if seeker[1] < target[1]:
            options.append(DOWN)

        if not options:
            options = [STAND]
        shipback = choice(options)

        return shipback

    def _move_left(self, pos):
        """
        :param pos: starting position
        :return: new position
        """
        new_x = pos[0] - 1
        if new_x == -1:
            new_x = 0
        return new_x, pos[1]

    def _move_right(self, pos):
        """
        :param pos: starting position
        :return: new position
        """
        new_x = pos[0] + 1
        if new_x == self.GRID_W:
            new_x = self.GRID_W - 1
        return new_x, pos[1]

    def _move_up(self, pos):
        """
        :param pos: starting position
        :return: new position
        """
        new_y = pos[1] - 1
        if new_y == -1:
            new_y = 0
        return pos[0], new_y

    def _move_down(self, pos):
        """
        :param pos: starting position
        :return: new position
        """
        new_y = pos[1] + 1
        if new_y == self.GRID_H:
            new_y = self.GRID_H - 1
        return pos[0], new_y

    def _stand(self, pos):
        return pos

    """
```

```
    Properties
    """

    @property
    def GRID_DIMENSIONS(self):
        return self.GRID_W, self.GRID_H

    @property
    def GRID_W(self):
        return int(self._grid_size[0])

    @property
    def GRID_H(self):
        return int(self._grid_size[1])

    @property
    def AGENTS(self):
        return [self._a_pos, self._b_pos]

    @property
    def A_AGENT(self):
        return self._a_pos

    @A_AGENT.setter
    def A_AGENT(self, new_pos):
        self._a_pos[0], self._a_pos[1] = new_pos[0], new_pos[1]

    @property
    def B_AGENT(self):
        return self._b_pos

    @B_AGENT.setter
    def B_AGENT(self, new_pos):
        self._b_pos[0], self._b_pos[1] = new_pos[0], new_pos[1]

    @property
    def RENDERER(self):
        return self._renderer

    @property
    def COORD_OBS(self):
        return self._coord_observation()
```

Listing A.4: `escalationgame.py`

```
from numpy import zeros, uint8, array
from numpy.random import randint

from gym_stag_hunt.src.games.abstract_grid_game import AbstractGridGame
from gym_stag_hunt.src.utils import overlaps_entity

"""
Entity Keys
"""
A_AGENT = 0
B_AGENT = 1
MARK = 2


class Escalation(AbstractGridGame):
    def __init__(
        self,
        streak_break_punishment_factor,
        opponent_policy,
        # Super Class Params
        window_title,
```

```
        grid_size,
        screen_size,
        obs_type,
        load_renderer,
        enable_multiagent,
    ):
        """
        :param streak_break_punishment_factor: Negative reinforcement for breaking the streak
        """

        super(Escalation, self).__init__(
            grid_size=grid_size,
            screen_size=screen_size,
            obs_type=obs_type,
            enable_multiagent=enable_multiagent,
        )

        self._streak_break_punishment_factor = streak_break_punishment_factor
        self._opponent_policy = opponent_policy
        self._mark = zeros(2, dtype=uint8)
        self._streak_active = False
        self._streak = 0
        self.reset_entities()

        # If rendering is enabled, we will instantiate the rendering pipeline
        if obs_type == "image" or load_renderer:
            # we don't want to import pygame if we aren't going to use it, so that's why this import is here
            from gym_stag_hunt.src.renderers.escalation_renderer import (
                EscalationRenderer,
            )

            self._renderer = EscalationRenderer(
                game=self, window_title=window_title, screen_size=screen_size
            )

    def _calc_reward(self):
        """
        Calculates the reinforcement rewards for the two agents.
        :return: A tuple R where R[0] is the reinforcement for A_Agent, and R[1] is the reinforcement for B_Agent
        """
        a_on_mark = overlaps_entity(self.A_AGENT, self.MARK)
        b_on_mark = overlaps_entity(self.B_AGENT, self.MARK)

        punishment = 0 - (self._streak_break_punishment_factor * self._streak)
        if a_on_mark and b_on_mark:
            rewards = 1, 1
        elif a_on_mark:
            rewards = punishment, 0
        elif b_on_mark:
            rewards = 0, punishment
        else:
            rewards = 0, 0

        if 1 in rewards:
            if not self._streak_active:
                self._streak_active = True
            self._streak = self._streak + 1
            self.MARK = self._move_entity(self.MARK, self._random_move(self.MARK))
        else:
            self._streak = 0
            self._streak_active = False

        return float(rewards[0]), float(rewards[1])
```

```

    def update(self, agent_moves):
        """
        Takes in agent actions and calculates next game state.
        :param agent_moves: List of actions for the two agents. If nothing is passed for the second agent, it does a
                            a random action.
        :return: observation, rewards, is the game done
        """
        if self._enable_multiagent:
            self._move_agents(agent_moves=agent_moves)
        else:
            if self._opponent_policy == "random":
                self._move_agents(
                    agent_moves=[agent_moves, self._random_move(self.B_AGENT)]
                )
            elif self._opponent_policy == "pursuit":
                self._move_agents(
                    agent_moves=[
                        agent_moves,
                        self._seek_entity(self.B_AGENT, self.MARK),
                    ]
                )

        iteration_rewards = self._calc_reward()
        obs = self.get_observation()
        info = {}
        done = False

        if self._enable_multiagent:
            if self._obs_type == "coords":
                return (
                    (obs, self._flip_coord_observation_perspective(obs)),
                    iteration_rewards,
                    done,
                    info,
                )
            else:
                return (obs, obs), iteration_rewards, done, info
        else:
            return obs, iteration_rewards[0], done, info

    def _coord_observation(self):
        """
        :return: list of all the entity coordinates
        """
        return array([self.A_AGENT, self.B_AGENT, self.MARK]).flatten()

    def reset_entities(self):
        """
        Reset all entity positions.
        :return:
        """
        self._reset_agents()
        self.MARK = [randint(0, self.GRID_W - 1), randint(0, self.GRID_H - 1)]

    """
    Properties
    """

    @property
    def MARK(self):
        return self._mark

    @MARK.setter
    def MARK(self, new_pos):
```

```
        self._mark[0], self._mark[1] = new_pos[0], new_pos[1]

    @property
    def STREAK_ACTIVE(self):
        return self._streak_active

    @property
    def STREAK(self):
        return self._streak

    @property
    def ENTITY_POSITIONS(self):
        return {
            "a_agent": self.A_AGENT,
            "b_agent": self.B_AGENT,
            "mark": self.MARK,
            "streak_active": self.STREAK_ACTIVE,
        }
```

Listing A.5: `harvestgame.py`

```
from numpy import array
from numpy.random import uniform

from gym_stag_hunt.src.games.abstract_grid_game import AbstractGridGame
from gym_stag_hunt.src.utils import overlaps_entity, spawn_plants, respawn_plants

# Entity Keys
A_AGENT = 0
B_AGENT = 1
Y_PLANT = 2
M_PLANT = 3


class Harvest(AbstractGridGame):
    def __init__(
        self,
        max_plants,
        chance_to_mature,
        chance_to_die,
        young_reward,
        mature_reward,
        # Super Class Params
        window_title,
        grid_size,
        screen_size,
        obs_type,
        load_renderer,
        enable_multiagent,
    ):
        """
        :param max_plants: What is the maximum number of plants that can be on the board.
        :param chance_to_mature: What chance does a young plant have to mature each time
    step.
        :param chance_to_die: What chance does a mature plant have to die each time step.
        :param young_reward: Reward for harvesting a young plant (awarded to the
    harvester)
        :param mature_reward: Reward for harvesting a mature plant (awarded to both
    agents)
        """

        super(Harvest, self).__init__(
            grid_size=grid_size,
            screen_size=screen_size,
            obs_type=obs_type,
            enable_multiagent=enable_multiagent,
```

```python
        )

        # Game Config
        self._max_plants = max_plants
        self._chance_to_mature = chance_to_mature
        self._chance_to_die = chance_to_die
        self._tagged_plants = []  # harvested plants that need to be re-spawned

        # Reinforcement variables
        self._young_reward = young_reward
        self._mature_reward = mature_reward

        # Entity Positions
        self._plants = []
        self._maturity_flags = [False] * max_plants
        self.reset_entities()  # place the entities on the grid

        # If rendering is enabled, we will instantiate the rendering pipeline
        if obs_type == "image" or load_renderer:
            # we don't want to import pygame if we aren't going to use it, so that's why
    this import is here
            from gym_stag_hunt.src.renderers.harvest_renderer import HarvestRenderer

            self._renderer = HarvestRenderer(
                game=self, window_title=window_title, screen_size=screen_size
            )

    """
    Collision Logic
    """

    def _overlaps_plants(self, a, plants):
        """
        :param a: (X, Y) tuple for entity 1
        :param plants: Array of (X, Y) tuples corresponding to plant positions
        :return: True if a overlaps any of the plants, False otherwise
        """
        for x in range(0, len(plants)):
            pos = plants[x]
            if overlaps_entity(a, pos):
                is_mature = self._maturity_flags[x]
                if x not in self._tagged_plants:
                    self._tagged_plants.append(x)
                return True, is_mature
        return False, False

    """
    State Updating Methods
    """

    def _calc_reward(self):
        """
        Calculates the reinforcement rewards for the two agents.
        :return: A tuple R where R[0] is the reinforcement for A_Agent, and R[1] is the
    reinforcement for B_Agent
        """
        a_collision, a_with_mature = self._overlaps_plants(self.A_AGENT, self.PLANTS)
        b_collision, b_with_mature = self._overlaps_plants(self.B_AGENT, self.PLANTS)

        a_reward, b_reward = 0, 0

        if a_collision:
            if a_with_mature:
                a_reward += self._mature_reward
                b_reward += self._mature_reward
            else:
```

```
                a_reward += self._young_reward

        if b_collision:
            if b_with_mature:
                a_reward += self._mature_reward
                b_reward += self._mature_reward
            else:
                b_reward += self._young_reward

        return float(a_reward), float(b_reward)

    def update(self, agent_moves):
        """
        Takes in agent actions and calculates next game state.
        :param agent_moves: List of actions for the two agents. If nothing is passed for the second agent, it does a
                            a random action.
        :return: observation, rewards, is the game done
        """
        if self._enable_multiagent:
            self._move_agents(agent_moves=agent_moves)
        else:
            self._move_agents(
                agent_moves=[agent_moves, self._random_move(self.B_AGENT)]
            )

        for idx, plant in enumerate(self._plants):
            is_mature = self._maturity_flags[idx]
            if is_mature:
                if uniform(0, 1) <= self._chance_to_die:
                    self._maturity_flags[idx] = False
                    self._tagged_plants.append(idx)
            else:
                if uniform(0, 1) <= self._chance_to_mature:
                    self._maturity_flags[idx] = True

        # Get Rewards
        iteration_rewards = self._calc_reward()

        if len(self._tagged_plants) > 0:
            self._plants = respawn_plants(
                plants=self.PLANTS,
                tagged_plants=self._tagged_plants,
                grid_dims=self.GRID_DIMENSIONS,
                used_coordinates=self.AGENTS,
            )
            self._tagged_plants = []

        obs = self.get_observation()
        done = False
        info = {}

        if self._enable_multiagent:
            if self._obs_type == "coords":
                return (
                    (obs, self._flip_coord_observation_perspective(obs)),
                    iteration_rewards,
                    done,
                    info,
                )
            else:
                return (obs, obs), iteration_rewards, done, info
        else:
            return obs, iteration_rewards[0], done, info

    def _coord_observation(self):
```

```python
        """
        :return: tuple of all the entity coordinates
        """
        a, b = self.AGENTS
        shipback = [a[0], a[1], b[0], b[1]]
        maturity_flags = self.MATURITY_FLAGS
        for idx, element in enumerate(self.PLANTS):
            new_entry = [0, 0, 0]
            new_entry[0], new_entry[1], new_entry[2] = (
                element[0],
                element[1],
                int(maturity_flags[idx]),
            )
            shipback = shipback + new_entry

        return array(shipback).flatten()

    def reset_entities(self):
        """
        Reset all entity positions.
        :return:
        """
        self._reset_agents()
        self._plants = spawn_plants(
            grid_dims=self.GRID_DIMENSIONS,
            how_many=self._max_plants,
            used_coordinates=self.AGENTS,
        )
        self._maturity_flags = [False] * self._max_plants

    """
    Properties
    """

    @property
    def PLANTS(self):
        return self._plants

    @property
    def MATURITY_FLAGS(self):
        return self._maturity_flags

    @property
    def ENTITY_POSITIONS(self):
        return {
            "a_agent": self.A_AGENT,
            "b_agent": self.B_AGENT,
            "plant_coords": self.PLANTS,
            "maturity_flags": self.MATURITY_FLAGS,
        }
```

Listing A.6: `staghuntgame.py`

```python
from numpy import zeros, uint8, array, hypot

from gym_stag_hunt.src.games.abstract_grid_game import AbstractGridGame

from gym_stag_hunt.src.utils import (
    overlaps_entity,
    place_entity_in_unoccupied_cell,
    spawn_plants,
    respawn_plants,
)

# Entity Keys
A_AGENT = 0
```

```
B_AGENT = 1
STAG = 2
PLANT = 3


class StagHunt(AbstractGridGame):
    def __init__(
        self,
        stag_reward,
        stag_follows,
        run_away_after_maul,
        opponent_policy,
        forage_quantity,
        forage_reward,
        mauling_punishment,
        # Super Class Params
        window_title,
        grid_size,
        screen_size,
        obs_type,
        load_renderer,
        enable_multiagent,
    ):
        """
        :param stag_reward: How much reinforcement the agents get for catching the stag
        :param stag_follows: Should the stag seek out the nearest agent (true) or take a random move (false)
        :param run_away_after_maul: Does the stag stay on the same cell after mauling an agent (true) or respawn (false)
        :param forage_quantity: How many plants will be placed on the board.
        :param forage_reward: How much reinforcement the agents get for harvesting a plant
        :param mauling_punishment: How much reinforcement the agents get for trying to catch a stag alone (MUST be neg.)
        """

        super(StagHunt, self).__init__(
            grid_size=grid_size,
            screen_size=screen_size,
            obs_type=obs_type,
            enable_multiagent=enable_multiagent,
        )

        # Config
        self._stag_follows = stag_follows
        self._run_away_after_maul = run_away_after_maul
        self._opponent_policy = opponent_policy

        # Reinforcement Variables
        self._stag_reward = stag_reward  # record RL values as attributes
        self._forage_quantity = forage_quantity
        self._forage_reward = forage_reward
        self._mauling_punishment = mauling_punishment

        # State Variables
        self._tagged_plants = []  # harvested plants that need to be re-spawned

        # Entity Positions
        self._stag_pos = zeros(2, dtype=uint8)
        self._plants_pos = []
        self.reset_entities()  # place the entities on the grid

        # If rendering is enabled, we will instantiate the rendering pipeline
        if obs_type == "image" or load_renderer:
            # we don't want to import pygame if we aren't going to use it, so that's why this import is here
```

```
            from gym_stag_hunt.src.renderers.hunt_renderer import HuntRenderer

            self._renderer = HuntRenderer(
                game=self, window_title=window_title, screen_size=screen_size
            )

    """
    Collision Logic
    """

    def _overlaps_plants(self, a, plants):
        """
        :param a: (X, Y) tuple for entity 1
        :param plants: Array of (X, Y) tuples corresponding to plant positions
        :return: True if a overlaps any of the plants, False otherwise
        """
        for x in range(0, len(plants)):
            pos = plants[x]
            if a[0] == pos[0] and a[1] == pos[1]:
                self._tagged_plants.append(x)
                return True
        return False

    """
    State Updating Methods
    """

    def _calc_reward(self):
        """
        Calculates the reinforcement rewards for the two agents.
        :return: A tuple R where R[0] is the reinforcement for A_Agent, and R[1] is the reinforcement for B_Agent
        """

        if overlaps_entity(self.A_AGENT, self.STAG):
            if overlaps_entity(self.B_AGENT, self.STAG):
                rewards = self._stag_reward, self._stag_reward  # Successful stag hunt
            else:
                if self._overlaps_plants(self.B_AGENT, self.PLANTS):
                    rewards = (
                        self._mauling_punishment,
                        self._forage_reward,
                    )  # A is mauled, B foraged
                else:
                    rewards = (
                        self._mauling_punishment,
                        0,
                    )  # A is mauled, B did not forage

        elif overlaps_entity(self.B_AGENT, self.STAG):
            """
            we already covered the case where a and b are both on the stag,
            so we can skip that check here
            """
            if self._overlaps_plants(self.A_AGENT, self.PLANTS):
                rewards = (
                    self._forage_reward,
                    self._mauling_punishment,
                )  # A foraged, B is mauled
            else:
                rewards = 0, self._mauling_punishment  # A did not forage, B is mauled

        elif self._overlaps_plants(self.A_AGENT, self.PLANTS):
            if self._overlaps_plants(self.B_AGENT, self.PLANTS):
                rewards = (
                    self._forage_reward,
```

```
                    self._forage_reward,
                )  # Both agents foraged
            else:
                rewards = self._forage_reward, 0  # Only A foraged

        else:
            if self._overlaps_plants(self.B_AGENT, self.PLANTS):
                rewards = 0, self._forage_reward  # Only B foraged
            else:
                rewards = 0, 0  # No one got anything

        return float(rewards[0]), float(rewards[1])

    def update(self, agent_moves):
        """
        Takes in agent actions and calculates next game state.
        :param agent_moves: If multi-agent, a tuple of actions. Otherwise a single action
     and the opponent takes an
                            action according to its established policy.
        :return: observation, rewards, is the game done
        """
        # Move Entities
        self._move_stag()
        if self._enable_multiagent:
            self._move_agents(agent_moves=agent_moves)
        else:
            if self._opponent_policy == "random":
                self._move_agents(
                    agent_moves=[agent_moves, self._random_move(self.B_AGENT)]
                )
            elif self._opponent_policy == "pursuit":
                self._move_agents(
                    agent_moves=[
                        agent_moves,
                        self._seek_entity(self.B_AGENT, self.STAG),
                    ]
                )

        # Get Rewards
        iteration_rewards = self._calc_reward()

        # Reset prey if it was caught
        if iteration_rewards == (self._stag_reward, self._stag_reward):
            self.STAG = place_entity_in_unoccupied_cell(
                grid_dims=self.GRID_DIMENSIONS,
                used_coordinates=self.PLANTS + self.AGENTS + [self.STAG],
            )
        elif (
            self._run_away_after_maul and self._mauling_punishment in iteration_rewards
        ):
            self.STAG = place_entity_in_unoccupied_cell(
                grid_dims=self.GRID_DIMENSIONS,
                used_coordinates=self.PLANTS + self.AGENTS + [self.STAG],
            )
        elif self._forage_reward in iteration_rewards:
            new_plants = respawn_plants(
                plants=self.PLANTS,
                tagged_plants=self._tagged_plants,
                grid_dims=self.GRID_DIMENSIONS,
                used_coordinates=self.AGENTS + [self.STAG],
            )
            self._tagged_plants = []
            self.PLANTS = new_plants

        obs = self.get_observation()
        info = {}
```

```python

        if self._enable_multiagent:
            if self._obs_type == "coords":
                return (
                    (obs, self._flip_coord_observation_perspective(obs)),
                    iteration_rewards,
                    False,
                    info,
                )
            else:
                return (obs, obs), iteration_rewards, False, info
        else:
            return obs, iteration_rewards[0], False, info

    def _coord_observation(self):
        """
        :return: list of all the entity coordinates
        """
        shipback = [self.A_AGENT, self.B_AGENT, self.STAG]
        shipback = shipback + self.PLANTS
        return array(shipback).flatten()

    """
    Movement Methods
    """

    def _seek_agent(self, agent_to_seek):
        """
        Moves the stag towards the specified agent
        :param agent_to_seek: agent to pursue
        :return: new position tuple for the stag
        """
        agent = self.A_AGENT
        if agent_to_seek == "b":
            agent = self.B_AGENT

        move = self._seek_entity(self.STAG, agent)

        return self._move_entity(self.STAG, move)

    def _move_stag(self):
        """
        Moves the stag towards the nearest agent.
        :return:
        """
        if self._stag_follows:
            stag, agents = self.STAG, self.AGENTS
            a_dist = hypot(
                int(agents[0][0]) - int(stag[0]), int(agents[0][1]) - int(stag[1])
            )
            b_dist = hypot(
                int(agents[1][0]) - int(stag[0]), int(agents[1][1]) - int(stag[1])
            )

            if a_dist < b_dist:
                agent_to_seek = "a"
            else:
                agent_to_seek = "b"

            self.STAG = self._seek_agent(agent_to_seek)
        else:
            self.STAG = self._move_entity(self.STAG, self._random_move(self.STAG))

    def reset_entities(self):
        """
        Reset all entity positions.
```

```
        :return:
        """
        self._reset_agents()
        self.STAG = [self.GRID_W // 2, self.GRID_H // 2]
        self.PLANTS = spawn_plants(
            grid_dims=self.GRID_DIMENSIONS,
            how_many=self._forage_quantity,
            used_coordinates=self.AGENTS + [self.STAG],
        )

    """
    Properties
    """

    @property
    def STAG(self):
        return self._stag_pos

    @STAG.setter
    def STAG(self, new_pos):
        self._stag_pos[0], self._stag_pos[1] = new_pos[0], new_pos[1]

    @property
    def PLANTS(self):
        return self._plants_pos

    @PLANTS.setter
    def PLANTS(self, new_pos):
        self._plants_pos = new_pos

    @property
    def ENTITY_POSITIONS(self):
        return {
            "a_agent": self.A_AGENT,
            "b_agent": self.B_AGENT,
            "stag": self.STAG,
            "plants": self.PLANTS,
        }
```

A.1.2 renderers/

Listing A.7: `abstractrenderer.py`

```
import pygame as pg
from numpy import rot90, flipud

from gym_stag_hunt.src.entities import Entity, get_gui_window_icon

"""
Constants
"""
BACKGROUND_COLOR = (255, 185, 137)
GRID_LINE_COLOR = (200, 150, 100, 200)
CLEAR = (0, 0, 0, 0)
TILE_SIZE = 32


class AbstractRenderer:
    def __init__(self, game, window_title, screen_size):
        """
        :param game: Class-based representation of the game state. Feeds all the
    information necessary to the renderer
        :param window_title: What we set as the window caption
```

```
        :param screen_size: The size of the virtual display on which we will be rendering stuff on
        """
        pg.init()  # initialize pygame
        pg.display.set_caption(window_title)  # set the window caption
        pg.display.set_icon(get_gui_window_icon())  # set the window icon
        pg.display.set_mode(
            (1, 1), pg.NOFRAME
        )  # set video mode without creating display
        self._clock = pg.time.Clock()  # create clock object
        self._screen = None  # temp screen attribute
        self._screen_size = screen_size  # record screen size as an attribute
        self._game = game  # record game as an attribute

        grid_size = game.GRID_DIMENSIONS
        game_surface_size = TILE_SIZE * grid_size[0], TILE_SIZE * grid_size[1]

        # Create a background
        self._background = pg.Surface(
            game_surface_size
        ).convert()  # here we create and fill all the surfaces
        self._background.fill(BACKGROUND_COLOR)
        # Create a layer for the grid
        self._grid_layer = pg.Surface(game_surface_size).convert_alpha()
        self._grid_layer.fill(CLEAR)
        # Create a layer for entities
        self._entity_layer = pg.Surface(game_surface_size).convert_alpha()
        self._entity_layer.fill(CLEAR)

        # Load sprites for the game objects
        entity_positions = self._game.ENTITY_POSITIONS

        self._a_sprite = Entity(
            entity_type="a_agent", location=entity_positions["a_agent"]
        )
        self._b_sprite = Entity(
            entity_type="b_agent", location=entity_positions["b_agent"]
        )

    """
    Controller Methods
    """

    def _init_display(self):
        self._screen = pg.display.set_mode(
            self._screen_size
        )  # instantiate virtual display

    def update(self):
        """
        :return: A pixel array corresponding to the new game state.
        """
        try:
            img_output = self._update_render()
            for event in pg.event.get():
                if event.type == pg.QUIT:
                    self.quit()
        except Exception as e:
            self.quit()
            raise e
        else:
            return img_output

    def quit(self):
        """
        Clears rendering resources.
```

```
        :return:
        """
        try:
            pg.display.quit()
            pg.quit()
            quit()
        except Exception as e:
            raise e

    """
    Drawing Methods
    """

    def _update_render(self, return_observation=True):
        """
        Executes the logic side of rendering without actually drawing it to the screen. In other words, new pixel
        values are calculated for each layer/surface without them actually being redrawn.
        :param return_observation: boolean saying if we are to (create and) return a numpy pixel array. The operation
                                   is expensive so we don't want to do it needlessly.
        :return: A numpy array corresponding to the pixel state of the display after the render update.
                 Note: The returned array is smaller than screen_size - the dimensions are 32 * grid_size
        """
        self._update_rects(self._game.ENTITY_POSITIONS)
        self._background.fill(BACKGROUND_COLOR)
        self._entity_layer.fill(CLEAR)
        self._draw_entities()
        # blit the surfaces to the main surface
        self._background.blit(self._grid_layer, (0, 0))
        self._background.blit(self._entity_layer, (0, 0))

        if return_observation:
            return flipud(rot90(pg.surfarray.array3d(self._background)))

    def render_on_display(self):
        """
        Renders the current frame on the virtual display.
        :return:
        """
        surf = pg.transform.scale(self._background, self._screen_size)
        if self._screen is None:
            self._init_display()
        self._screen.blit(surf, (0, 0))
        pg.display.flip()

    def _draw_grid(self):
        """
        Draws the grid lines to the grid layer surface.
        :return:
        """

        # drawing the horizontal lines
        for y in range(self.GRID_H + 1):
            pg.draw.line(
                self._grid_layer,
                GRID_LINE_COLOR,
                (0, y * TILE_SIZE),
                (self.SCREEN_W, y * TILE_SIZE),
            )

        # drawing the vertical lines
        for x in range(self.GRID_W + 1):
            pg.draw.line(
```

```python
                    self._grid_layer,
                    GRID_LINE_COLOR,
                    (x * TILE_SIZE, 0),
                    (x * TILE_SIZE, self.SCREEN_H),
                )

    def _draw_entities(self):
        # Agents
        self._entity_layer.blit(
            self._a_sprite.IMAGE, (self._a_sprite.rect.left, self._a_sprite.rect.top)
        )
        self._entity_layer.blit(
            self._b_sprite.IMAGE, (self._b_sprite.rect.left, self._b_sprite.rect.top)
        )

    def _update_rects(self, entity_positions):
        """
        Update all the entity rectangles with their new positions.
        :param entity_positions: A dictionary containing positions for all the entities.
        :return:
        """
        self._a_sprite.update_rect(entity_positions["a_agent"])
        self._b_sprite.update_rect(entity_positions["b_agent"])

    """
    Properties
    """

    @property
    def SCREEN_SIZE(self):
        return tuple(self._screen_size)

    @property
    def SCREEN_W(self):
        return int(self._screen_size[0])

    @property
    def SCREEN_H(self):
        return int(self._screen_size[1])

    @property
    def GRID_W(self):
        return self._game.GRID_W

    @property
    def GRID_H(self):
        return self._game.GRID_H

    @property
    def CELL_W(self):
        return float(self.SCREEN_W) / float(self.GRID_W)

    @property
    def CELL_H(self):
        return float(self.SCREEN_H) / float(self.GRID_H)

    @property
    def CELL_SIZE(self):
        return self.CELL_W, self.CELL_H
```

Listing A.8: `escalationrenderer.py`

```python
from gym_stag_hunt.src.entities import Mark
from gym_stag_hunt.src.renderers.abstract_renderer import AbstractRenderer


```

```python
class EscalationRenderer(AbstractRenderer):
    def __init__(self, game, window_title, screen_size):
        super(EscalationRenderer, self).__init__(
            game=game, window_title=window_title, screen_size=screen_size
        )

        self._mark_sprite = Mark(location=self._game.ENTITY_POSITIONS["mark"])

        self.cell_sizes = self.CELL_SIZE
        self._draw_grid()

    """
    Misc
    """

    def _draw_entities(self):
        """
        Draws the entity sprites to the entity layer surface.
        :return:
        """
        if self._game.ENTITY_POSITIONS["streak_active"]:
            self._entity_layer.blit(
                self._mark_sprite.IMAGE_ACTIVE,
                (self._mark_sprite.rect.left, self._mark_sprite.rect.top),
            )
        else:
            self._entity_layer.blit(
                self._mark_sprite.IMAGE,
                (self._mark_sprite.rect.left, self._mark_sprite.rect.top),
            )

        # Agents
        self._entity_layer.blit(
            self._a_sprite.IMAGE, (self._a_sprite.rect.left, self._a_sprite.rect.top)
        )
        self._entity_layer.blit(
            self._b_sprite.IMAGE, (self._b_sprite.rect.left, self._b_sprite.rect.top)
        )

    def _update_rects(self, entity_positions):
        """
        Update all the entity rectangles with their new positions.
        :param entity_positions: A dictionary containing positions for all the entities.
        :return:
        """
        self._a_sprite.update_rect(entity_positions["a_agent"])
        self._b_sprite.update_rect(entity_positions["b_agent"])
        self._mark_sprite.update_rect(entity_positions["mark"])
```

Listing A.9: `harvestrenderer.py`

```python
from gym_stag_hunt.src.entities import HarvestPlant
from gym_stag_hunt.src.renderers.abstract_renderer import AbstractRenderer


class HarvestRenderer(AbstractRenderer):
    def __init__(self, game, window_title, screen_size):
        super(HarvestRenderer, self).__init__(
            game=game, window_title=window_title, screen_size=screen_size
        )

        self.cell_sizes = self.CELL_SIZE
        entity_positions = self._game.ENTITY_POSITIONS

        self.plant_sprites = self._make_plant_entities(entity_positions["plant_coords"])

```

```python
        self._draw_grid()

    """
    Misc
    """

    def _make_plant_entities(self, locations):
        """
        :param locations: locations for the new plants
        :return: an array of plant entities ready to be rendered.
        """
        plants = []
        for loc in locations:
            plants.append(HarvestPlant(location=loc))
        return plants

    def _draw_entities(self):
        """
        Draws the entity sprites to the entity layer surface.
        :return:
        """

        maturity_flags = self._game.ENTITY_POSITIONS["maturity_flags"]

        for idx, plant in enumerate(self.plant_sprites):
            if maturity_flags[idx]:
                self._entity_layer.blit(plant.IMAGE, (plant.rect.left, plant.rect.top))
            else:
                self._entity_layer.blit(
                    plant.IMAGE_YOUNG, (plant.rect.left, plant.rect.top)
                )

        # Agents
        self._entity_layer.blit(
            self._a_sprite.IMAGE, (self._a_sprite.rect.left, self._a_sprite.rect.top)
        )
        self._entity_layer.blit(
            self._b_sprite.IMAGE, (self._b_sprite.rect.left, self._b_sprite.rect.top)
        )

    def _update_rects(self, entity_positions):
        """
        Update all the entity rectangles with their new positions.
        :param entity_positions: A dictionary containing positions for all the entities.
        :return:
        """
        self._a_sprite.update_rect(entity_positions["a_agent"])
        self._b_sprite.update_rect(entity_positions["b_agent"])

        for idx, plant in enumerate(self.plant_sprites):
            plant.update_rect(entity_positions["plant_coords"][idx])
```

Listing A.10: huntrenderer.py

```python
from gym_stag_hunt.src.entities import Entity
from gym_stag_hunt.src.renderers.abstract_renderer import AbstractRenderer


class HuntRenderer(AbstractRenderer):
    def __init__(self, game, window_title, screen_size):
        super(HuntRenderer, self).__init__(
            game=game, window_title=window_title, screen_size=screen_size
        )

        entity_positions = self._game.ENTITY_POSITIONS

```

```
        self._stag_sprite = Entity(
            entity_type="stag", location=entity_positions["stag"]
        )
        self._plant_sprites = self._make_plant_entities(entity_positions["plants"])

        self._draw_grid()

    """
    Misc
    """

    def _make_plant_entities(self, locations):
        """
        :param locations: locations for the new plants
        :return: an array of plant entities ready to be rendered.
        """
        plants = []
        for loc in locations:
            plants.append(Entity(entity_type="plant", location=loc))
        return plants

    def _draw_entities(self):
        """
        Draws the entity sprites to the entity layer surface.
        :return:
        """
        self._entity_layer.blit(
            self._stag_sprite.IMAGE,
            (self._stag_sprite.rect.left, self._stag_sprite.rect.top),
        )
        for plant in self._plant_sprites:
            self._entity_layer.blit(plant.IMAGE, (plant.rect.left, plant.rect.top))
        # Agents
        self._entity_layer.blit(
            self._a_sprite.IMAGE, (self._a_sprite.rect.left, self._a_sprite.rect.top)
        )
        self._entity_layer.blit(
            self._b_sprite.IMAGE, (self._b_sprite.rect.left, self._b_sprite.rect.top)
        )

    def _update_rects(self, entity_positions):
        """
        Update all the entity rectangles with their new positions.
        :param entity_positions: A dictionary containing positions for all the entities.
        :return:
        """
        self._a_sprite.update_rect(entity_positions["a_agent"])
        self._b_sprite.update_rect(entity_positions["b_agent"])
        self._stag_sprite.update_rect(entity_positions["stag"])
        plants_pos = entity_positions["plants"]
        idx = 0
        for plant in self._plant_sprites:
            plant.update_rect(plants_pos[idx])
            idx = idx + 1
```

A.2 envs/

A.2.1 gym/

Listing A.11: abstractmarkovstaghunt.py

```python
from abc import ABC

from gym import Env

from gym_stag_hunt.src.utils import print_matrix


class AbstractMarkovStagHuntEnv(Env, ABC):
    metadata = {"render.modes": ["human", "array"], "obs.types": ["image", "coords"]}

    def __init__(self, grid_size=(5, 5), obs_type="image", enable_multiagent=False):
        """
        :param grid_size: A (W, H) tuple corresponding to the grid dimensions. Although W=H is expected, W!=H works also
        :param obs_type: Can be 'image' for pixel-array based observations, or 'coords' for just the entity coordinates
        """

        total_cells = grid_size[0] * grid_size[1]
        if total_cells < 3:
            raise AttributeError(
                "Grid is too small. Please specify a larger grid size."
            )
        if obs_type not in self.metadata["obs.types"]:
            raise AttributeError(
                'Invalid observation type provided. Please specify "image" or "coords"'
            )
        if grid_size[0] >= 255 or grid_size[1] >= 255:
            raise AttributeError(
                "Grid is too large. Please specify a smaller grid size."
            )

        super(AbstractMarkovStagHuntEnv, self).__init__()

        self.obs_type = obs_type
        self.done = False
        self.enable_multiagent = enable_multiagent

    def step(self, actions):
        """
        Run one timestep of the environment's dynamics.
        :param actions: ints signifying actions for the agents. You can pass one, in which case the second agent does a
                        random move, or two, in which case each agent takes the specified action.
        :return: observation, rewards, is the game done, additional info
        """
        return self.game.update(actions)

    def reset(self):
        """
        Reset the game state
        :return: initial observation
        """
        self.game.reset_entities()
        self.done = False
        return self.game.get_observation()

    def render(self, mode="human", obs=None):
        """
        :param obs: observation data (passed for coord observations so we dont have to run the function twice)
        :param mode: rendering mode
        :return:
        """
        if mode == "human":
```

```
            if self.obs_type == "image":
                self.game.RENDERER.render_on_display()
            else:
                if self.game.RENDERER:
                    self.game.RENDERER.update()
                    self.game.RENDERER.render_on_display()
                else:
                    if obs is not None:
                        print_matrix(obs, self.game_title, self.game.GRID_DIMENSIONS)
                    else:
                        print_matrix(
                            self.game.get_observation(),
                            self.game_title,
                            self.game.GRID_DIMENSIONS,
                        )
        elif mode == "array":
            print_matrix(
                self.game._coord_observation(),
                self.game_title,
                self.game.GRID_DIMENSIONS,
            )

    def close(self):
        """
        Closes all needed resources
        :return:
        """
        if self.game.RENDERER:
            self.game.RENDERER.quit()
```

Listing A.12: `escalation.py`

```
from gym.spaces import Discrete, Box
from numpy import Inf, uint8

from gym_stag_hunt.envs.gym.abstract_markov_staghunt import AbstractMarkovStagHuntEnv
from gym_stag_hunt.src.entities import TILE_SIZE
from gym_stag_hunt.src.games.escalation_game import Escalation


class EscalationEnv(AbstractMarkovStagHuntEnv):
    def __init__(
        self,
        grid_size=(5, 5),
        screen_size=(600, 600),
        obs_type="image",
        enable_multiagent=False,
        opponent_policy="pursuit",
        load_renderer=False,
        streak_break_punishment_factor=0.5,
    ):
        """
        :param grid_size: A (W, H) tuple corresponding to the grid dimensions. Although W=H is expected, W!=H works also
        :param screen_size: A (W, H) tuple corresponding to the pixel dimensions of the game window
        :param obs_type: Can be 'image' for pixel-array based observations, or 'coords' for just the entity coordinates
        """
        total_cells = grid_size[0] * grid_size[1]
        if total_cells < 3:
            raise AttributeError(
                "Grid is too small. Please specify a larger grid size."
            )

        super(EscalationEnv, self).__init__(
```

```python
            grid_size=grid_size, obs_type=obs_type, enable_multiagent=enable_multiagent
        )

        # Rendering and State Variables
        self.game_title = "escalation"
        self.streak_break_punishment_factor = streak_break_punishment_factor
        window_title = (
            "OpenAI Gym - Escalation (%d x %d)" % grid_size
        )  # create game representation
        self.game = Escalation(
            window_title=window_title,
            grid_size=grid_size,
            screen_size=screen_size,
            obs_type=obs_type,
            enable_multiagent=enable_multiagent,
            load_renderer=load_renderer,
            streak_break_punishment_factor=streak_break_punishment_factor,
            opponent_policy=opponent_policy,
        )

        # Environment Config
        self.action_space = Discrete(5)  # up, down, left, right or stand
        if obs_type == "image":  # Observation is the rgb pixel array
            self.observation_space = Box(
                0,
                255,
                shape=(grid_size[0] * TILE_SIZE, grid_size[1] * TILE_SIZE, 3),
                dtype=uint8,
            )
        elif obs_type == "coords":
            self.observation_space = Box(0, max(grid_size), shape=(6,), dtype=uint8)

        self.reward_range = (
            -Inf,
            Inf,
        )  # There is technically no limit on how high or low the reinforcement can be
```

Listing A.13: `harvest.py`

```python
from gym.spaces import Discrete, Box
from numpy import uint8

from gym_stag_hunt.envs.gym.abstract_markov_staghunt import AbstractMarkovStagHuntEnv
from gym_stag_hunt.src.entities import TILE_SIZE
from gym_stag_hunt.src.games.harvest_game import Harvest


class HarvestEnv(AbstractMarkovStagHuntEnv):
    def __init__(
        self,
        grid_size=(5, 5),
        screen_size=(600, 600),
        obs_type="image",
        enable_multiagent=False,
        load_renderer=False,
        max_plants=4,
        chance_to_mature=0.1,
        chance_to_die=0.1,
        young_reward=1,
        mature_reward=2,
    ):
        """
        :param grid_size: A (W, H) tuple corresponding to the grid dimensions. Although W
    =H is expected, W!=H works also
        :param screen_size: A (W, H) tuple corresponding to the pixel dimensions of the
    game window
```

```python
        :param obs_type: Can be 'image' for pixel-array based observations, or 'coords'
    for just the entity coordinates
        """
        if young_reward > mature_reward:
            raise AttributeError(
                "The game does not qualify as a Stag Hunt, please change parameters so
    that "
                "young_reward > mature_reward"
            )
        total_cells = grid_size[0] * grid_size[1]
        if max_plants >= total_cells - 2:  # -2 is for the cells occupied by the agents
            raise AttributeError(
                "Plant quantity is too high. The plants will not fit on the grid."
            )
        if total_cells < 3:
            raise AttributeError(
                "Grid is too small. Please specify a larger grid size."
            )

        super(HarvestEnv, self).__init__(
            grid_size=grid_size, obs_type=obs_type, enable_multiagent=enable_multiagent
        )

        self.game_title = "harvest"
        self.max_plants = max_plants
        self.chance_to_mature = chance_to_mature
        self.chance_to_die = chance_to_die
        self.young_reward = young_reward
        self.mature_reward = mature_reward
        self.reward_range = (0, mature_reward)

        window_title = (
            "OpenAI Gym - Harvest (%d x %d)" % grid_size
        )  # create game representation
        self.game = Harvest(
            window_title=window_title,
            grid_size=grid_size,
            screen_size=screen_size,
            obs_type=obs_type,
            enable_multiagent=enable_multiagent,
            load_renderer=load_renderer,
            max_plants=max_plants,
            chance_to_mature=chance_to_mature,
            chance_to_die=chance_to_die,
            young_reward=young_reward,
            mature_reward=mature_reward,
        )

        self.action_space = Discrete(5)  # up, down, left, right or stand

        if obs_type == "image":
            self.observation_space = Box(
                0,
                255,
                shape=(grid_size[0] * TILE_SIZE, grid_size[1] * TILE_SIZE, 3),
                dtype=uint8,
            )
        elif obs_type == "coords":
            self.observation_space = Box(
                0, max(grid_size), shape=(4 + max_plants * 3,), dtype=uint8
            )
```

Listing A.14: hunt.py

```python
from gym.spaces import Discrete, Box
from numpy import uint8
```

```python

from gym_stag_hunt.envs.gym.abstract_markov_staghunt import AbstractMarkovStagHuntEnv
from gym_stag_hunt.src.entities import TILE_SIZE
from gym_stag_hunt.src.games.staghunt_game import StagHunt


class HuntEnv(AbstractMarkovStagHuntEnv):
    def __init__(
        self,
        grid_size=(5, 5),
        screen_size=(600, 600),
        obs_type="image",
        enable_multiagent=False,
        opponent_policy="random",
        load_renderer=False,
        stag_follows=True,
        run_away_after_maul=False,
        forage_quantity=2,
        stag_reward=5,
        forage_reward=1,
        mauling_punishment=-5,
    ):
        """
        :param grid_size: A (W, H) tuple corresponding to the grid dimensions. Although W=H is expected, W!=H works also
        :param screen_size: A (W, H) tuple corresponding to the pixel dimensions of the game window
        :param obs_type: Can be 'image' for pixel-array based observations, or 'coords' for just the entity coordinates
        :param stag_follows: Should the stag seek out the nearest agent (true) or take a random move (false)
        :param run_away_after_maul: Does the stag stay on the same cell after mauling an agent (true) or respawn (false)
        :param forage_quantity: How many plants will be placed on the board.
        :param stag_reward: How much reinforcement the agents get for catching the stag
        :param forage_reward: How much reinforcement the agents get for harvesting a plant
        :param mauling_punishment: How much reinforcement the agents get for trying to catch a stag alone (MUST be neg.)
        """
        if not (stag_reward > forage_reward >= 0 > mauling_punishment):
            raise AttributeError(
                "The game does not qualify as a Stag Hunt, please change parameters so that "
                "stag_reward > forage_reward >= 0 > mauling_punishment"
            )
        if mauling_punishment == forage_reward:
            raise AttributeError(
                "Mauling punishment and forage reward are equal."
                " Game logic will not function properly."
            )
        total_cells = grid_size[0] * grid_size[1]
        if (
            forage_quantity >= total_cells - 3
        ):  # -3 is for the cells occupied by the agents and stag
            raise AttributeError(
                "Forage quantity is too high. The plants will not fit on the grid."
            )
        if total_cells < 3:
            raise AttributeError(
                "Grid is too small. Please specify a larger grid size."
            )

        super(HuntEnv, self).__init__(
            grid_size=grid_size, obs_type=obs_type, enable_multiagent=enable_multiagent
        )
```

```

        self.game_title = "hunt"
        self.stag_reward = stag_reward
        self.forage_reward = forage_reward
        self.mauling_punishment = mauling_punishment
        self.reward_range = (mauling_punishment, stag_reward)

        window_title = (
            "OpenAI Gym - Stag Hunt (%d x %d)" % grid_size
        )  # create game representation
        self.game = StagHunt(
            window_title=window_title,
            grid_size=grid_size,
            screen_size=screen_size,
            obs_type=obs_type,
            enable_multiagent=enable_multiagent,
            load_renderer=load_renderer,
            stag_reward=stag_reward,
            stag_follows=stag_follows,
            run_away_after_maul=run_away_after_maul,
            forage_quantity=forage_quantity,
            forage_reward=forage_reward,
            mauling_punishment=mauling_punishment,
            opponent_policy=opponent_policy,
        )

        self.action_space = Discrete(5)  # up, down, left, right or stand

        if obs_type == "image":
            self.observation_space = Box(
                0,
                255,
                shape=(grid_size[0] * TILE_SIZE, grid_size[1] * TILE_SIZE, 3),
                dtype=uint8,
            )
        elif obs_type == "coords":
            self.observation_space = Box(
                0, max(grid_size), shape=(6 + forage_quantity * 2,), dtype=uint8
            )
```

Listing A.15: `simple.py`

```
from sys import stdout

from gym import Env
from gym.spaces import Discrete, Box
from numpy.random import randint

COOPERATE = 0
DEFECT = 1


class SimpleEnv(Env):
    def __init__(
        self,
        cooperation_reward=5,
        defect_alone_reward=1,
        defect_together_reward=1,
        failed_cooperation_punishment=-5,
        eps_per_game=1,
    ):
        """
        :param cooperation_reward: How much reinforcement the agents get for catching the stag
        :param defect_alone_reward: How much reinforcement an agent gets for defecting if the other one doesn't
```

```
        :param defect_together_reward: How much reinforcement an agent gets for defecting if the other one does also
        :param failed_cooperation_punishment: How much reinforcement the agents get for trying to catch a stag alone
        :param eps_per_game: How many games happen before the internal done flag is set to True. Only included for
                             the sake of convenience.
        """

        if not (
            cooperation_reward
            > defect_alone_reward
            >= defect_together_reward
            > failed_cooperation_punishment
        ):
            raise AttributeError(
                "The game does not qualify as a Stag Hunt, please change parameters so that "
                "stag_reward > forage_reward_single >= forage_reward_both > mauling_punishment"
            )

        super(SimpleEnv, self).__init__()

        # Reinforcement Variables
        self.cooperation_reward = cooperation_reward
        self.defect_alone_reward = defect_alone_reward
        self.defect_together_reward = defect_together_reward
        self.failed_cooperation_punishment = failed_cooperation_punishment

        # State Variables
        self.done = False
        self.ep = 0
        self.final_ep = eps_per_game
        self.seed()

        # Environment Config
        self.action_space = Discrete(2)  # cooperate or defect
        self.observation_space = Box(
            low=0, high=1, shape=(2,), dtype=int
        )  # last agent actions
        self.reward_range = (failed_cooperation_punishment, cooperation_reward)

    def step(self, actions):
        """
        Play one stag hunt game.
        :param actions: ints signifying actions for the agents. You can pass one, in which case the second agent does a
                        random move, or two, in which case each agent takes the specified action.
        :return: observation, rewards, is the game done, additional info
        """
        self.ep = self.ep + 1
        if self.ep >= self.final_ep:
            done = True
            self.ep = 0
        else:
            done = False

        if isinstance(actions, list):
            a_action = actions[0]
            if len(actions) > 1:
                b_action = actions[1]
            else:
                b_action = randint(0, 1)
        else:
```

```
            a_action = actions
            b_action = randint(0, 1)

        b_cooperated = b_action == COOPERATE

        if a_action == COOPERATE:
            reward = (
                (self.cooperation_reward, self.cooperation_reward)
                if b_cooperated
                else (self.failed_cooperation_punishment, self.defect_alone_reward)
            )
        else:
            reward = (
                (self.defect_alone_reward, self.failed_cooperation_punishment)
                if b_cooperated
                else (self.defect_together_reward, self.defect_together_reward)
            )

        obs = (a_action, b_action)

        return obs, reward, done, {}

    def reset(self):
        """
        Reset the game state
        """
        self.done = False
        self.ep = 0

    def render(self, mode="human", rewards=None):
        """
        :return:
        """
        if rewards is None:
            print("Please supply rewards to render.")
            pass
        else:
            top_right = "  "
            top_left = "  "
            bot_left = "  "
            bot_right = "  "

            if rewards == (self.cooperation_reward, self.cooperation_reward):
                top_left = "AB"
            elif rewards == (
                self.defect_alone_reward,
                self.failed_cooperation_punishment,
            ):
                bot_left = "A "
                top_right = " B"
            elif rewards == (
                self.failed_cooperation_punishment,
                self.defect_alone_reward,
            ):
                top_left = "A "
                top_right = " B"
            elif rewards == (self.defect_together_reward, self.defect_together_reward):
                bot_right = "AB"

            stdout.write("\n\n\n")
            stdout.write("      B   \n")
            stdout.write("    C   D \n")
            stdout.write("                        \n")
            stdout.write(" C    " + top_left + "   " + top_right + "   \n")
            stdout.write("                \n")
            stdout.write("A                       \n")
```

```
            stdout.write("                \n")
            stdout.write(" D    " + bot_left + "   " + bot_right + "   \n")
            stdout.write("                        \n\r")
            stdout.flush()

    def close(self):
        quit()
```

A.2.2 pettingzoo/

Listing A.16: escalation.py

```
from gym_stag_hunt.envs.gym.escalation import EscalationEnv
from gym_stag_hunt.envs.pettingzoo.shared import PettingZooEnv
from pettingzoo.utils import parallel_to_aec


def env(**kwargs):
    return ZooEscalationEnvironment(**kwargs)


def raw_env(**kwargs):
    return parallel_to_aec(env(**kwargs))


class ZooEscalationEnvironment(PettingZooEnv):
    metadata = {"render_modes": ["human", "array"], "name": "escalation_pz"}

    def __init__(
        self,
        grid_size=(5, 5),
        screen_size=(600, 600),
        obs_type="image",
        enable_multiagent=False,
        opponent_policy="pursuit",
        load_renderer=False,
        streak_break_punishment_factor=0.5,
    ):
        escalation_env = EscalationEnv(
            grid_size,
            screen_size,
            obs_type,
            enable_multiagent,
            opponent_policy,
            load_renderer,
            streak_break_punishment_factor,
        )
        super().__init__(og_env=escalation_env)
```

Listing A.17: harvest.py

```
from gym_stag_hunt.envs.gym.harvest import HarvestEnv
from gym_stag_hunt.envs.pettingzoo.shared import PettingZooEnv
from pettingzoo.utils import parallel_to_aec


def env(**kwargs):
    return ZooHarvestEnvironment(**kwargs)


def raw_env(**kwargs):
    return parallel_to_aec(env(**kwargs))
```

```python


class ZooHarvestEnvironment(PettingZooEnv):
    metadata = {"render_modes": ["human", "array"], "name": "harvest_pz"}

    def __init__(
        self,
        grid_size=(5, 5),
        screen_size=(600, 600),
        obs_type="image",
        enable_multiagent=False,
        load_renderer=False,
        max_plants=4,
        chance_to_mature=0.1,
        chance_to_die=0.1,
        young_reward=1,
        mature_reward=2,
    ):
        harvest_env = HarvestEnv(
            grid_size,
            screen_size,
            obs_type,
            enable_multiagent,
            load_renderer,
            max_plants,
            chance_to_mature,
            chance_to_die,
            young_reward,
            mature_reward,
        )
        super().__init__(og_env=harvest_env)
```

Listing A.18: `hunt.py`

```python
from gym_stag_hunt.envs.gym.hunt import HuntEnv
from gym_stag_hunt.envs.pettingzoo.shared import PettingZooEnv
from pettingzoo.utils import parallel_to_aec


def env(**kwargs):
    return ZooHuntEnvironment(**kwargs)


def raw_env(**kwargs):
    return parallel_to_aec(env(**kwargs))


class ZooHuntEnvironment(PettingZooEnv):
    metadata = {"render_modes": ["human", "array"], "name": "hunt_pz"}

    def __init__(
        self,
        grid_size=(5, 5),
        screen_size=(600, 600),
        obs_type="image",
        enable_multiagent=False,
        opponent_policy="random",
        load_renderer=False,
        stag_follows=True,
        run_away_after_maul=False,
        forage_quantity=2,
        stag_reward=5,
        forage_reward=1,
        mauling_punishment=-5,
    ):
        hunt_env = HuntEnv(
```

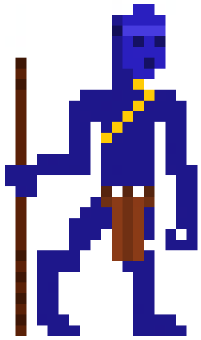
```
            grid_size,
            screen_size,
            obs_type,
            enable_multiagent,
            opponent_policy,
            load_renderer,
            stag_follows,
            run_away_after_maul,
            forage_quantity,
            stag_reward,
            forage_reward,
            mauling_punishment,
        )
        super().__init__(og_env=hunt_env)
```

Listing A.19: `shared.py`

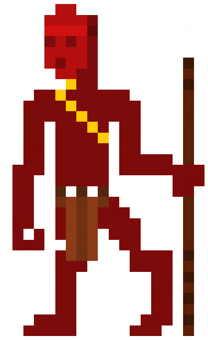
```
from pettingzoo.utils import wrappers
from pettingzoo import ParallelEnv
from pettingzoo.utils import agent_selector
import functools


def default_wrappers(env_init):
    """
    The env function wraps the environment in 3 wrappers by default. These
    wrappers contain logic that is common to many pettingzoo environments.
    We recommend you use at least the OrderEnforcingWrapper on your own environment
    to provide sane error messages. You can find full documentation for these methods
    elsewhere in the developer documentation.
    """
    env_init = wrappers.CaptureStdoutWrapper(env_init)
    env_init = wrappers.AssertOutOfBoundsWrapper(env_init)
    env_init = wrappers.OrderEnforcingWrapper(env_init)
    return env_init


class PettingZooEnv(ParallelEnv):
    def __init__(self, og_env):
        super().__init__()

        self.env = og_env

        self.possible_agents = ["player_" + str(n) for n in range(2)]
        self.agents = self.possible_agents[:]

        self.agent_name_mapping = dict(
            zip(self.possible_agents, list(range(len(self.possible_agents))))
        )
        self.agent_selection = None
        self._agent_selector = agent_selector(self.agents)

        self._action_spaces = {
            agent: self.env.action_space for agent in self.possible_agents
        }
        self._observation_spaces = {
            agent: self.env.observation_space for agent in self.possible_agents
        }

        self.dones = dict(zip(self.agents, [False for _ in self.agents]))
        self.rewards = dict(zip(self.agents, [0.0 for _ in self.agents]))
        self._cumulative_rewards = dict(zip(self.agents, [0.0 for _ in self.agents]))
        self.infos = dict(zip(self.agents, [{} for _ in self.agents]))
        self.accumulated_actions = []
        self.current_observations = {
            agent: self.env.observation_space.sample() for agent in self.agents
```

```
        }
        self.t = 0
        self.last_rewards = [0.0, 0.0]

    # this cache ensures that same space object is returned for the same agent
    # allows action space seeding to work as expected
    @functools.lru_cache(maxsize=None)
    def observation_space(self, agent):
        return self.env.observation_space

    @functools.lru_cache(maxsize=None)
    def action_space(self, agent):
        return self.env.action_space

    def render(self, mode="human"):
        self.env.render(mode)

    def close(self):
        self.env.close()

    def reset(self):
        self.agents = self.possible_agents[:]
        self._agent_selector.reinit(self.agents)
        self.agent_selection = self._agent_selector.next()
        self.rewards = dict(zip(self.agents, [0.0 for _ in self.agents]))
        self._cumulative_rewards = dict(zip(self.agents, [0.0 for _ in self.agents]))
        self.infos = dict(zip(self.agents, [{} for _ in self.agents]))
        self.dones = dict(zip(self.agents, [False for _ in self.agents]))
        obs = self.env.reset()
        self.accumulated_actions = []
        self.current_observations = {agent: obs for agent in self.agents}
        self.t = 0

        return self.current_observations

    def step(self, actions):
        observations, rewards, env_done, info = self.env.step(list(actions.values()))

        obs = {self.agents[0]: observations[0], self.agents[1]: observations[1]}
        rewards = {self.agents[0]: rewards[0], self.agents[1]: rewards[1]}
        dones = {agent: env_done for agent in self.agents}
        infos = {agent: {} for agent in self.agents}

        return obs, rewards, dones, infos

    def observe(self, agent):
        return self.current_observations[agent]

    def state(self):
        pass
```

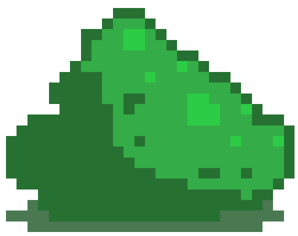
A.3 assets/



(a) Blue Agent

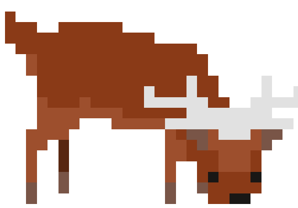


(b) Red Agent

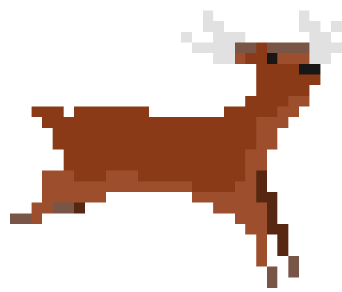


(c) Plant With No Fruit

(d) Plant Fruit



(e) Inactive Mark



(f) Active Mark

(g) Stag

Figure A.1: Game Assets