SOLVING CYBER ALERT ALLOCATION MARKOV GAMES WITH DEEP REINFORCEMENT LEARNING

by

Noah Dunstatter, B.S.

A thesis submitted to the Graduate Council of
Texas State University in partial fulfillment
of the requirements for the degree of
Master of Science
with a Major in Computer Science
May 2019

Committee Members:

Mina Guirguis, Chair

Jelena Tešić

Qijun Gu

FAIR USE AND AUTHOR'S PERMISSION STATEMENT

Fair Use

Duplication Permission

DEDICATION

In dedication to my wonderful Mother, whose unending love and support has allowed me to achieve more than I could ever imagine.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my amazing advisor Dr. Mina Guirguis whose limitless generosity has provided me with so many opportunities to learn and grow as a researcher. Without your encouragement I never would have embarked on this wonderful journey in the first place. Your diligent council and unending faith in my ability has left me with a profound sense of gratitude.

Many thanks to my loving friends Ders, Berry, Phu, Spanky, Randy, Frinkle, and Hibah who have been so patient throughout this arduous journey, with many late nights and missed occasions to make up for.

A special thanks to my Mother Juliane, who instilled in me the sense of wonder and curiosity for the world that fuels my ambitions to this day. To my Father Frank and Stepmother Donna, whose love and support kept me strong when spirits were low.

And lastly, an enormous thanks to my dear friend and colleague Alireza Tahsini. You pushed me to strive for the best in all things and I can't thank you enough for the stimulating discussions, the sleepless nights spent working together before deadlines, and for all the fun we have had in the last two years. I was so fortunate to have you at my side on this journey and I look forward to seeing your bright future begin to shine in the coming years.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

ABSTRACT

Most large scale networks employ some kind of intrusion detection software on their computer systems (e.g., servers, routers, etc.) that monitor and flag suspicious or abnormal activity. When possibly malicious activity is detected, one or more cyber-alerts may be generated with varying levels of significance (e.g., high, medium, or low). Some subset of these alerts may then be assigned to cyber-security analysts on staff for further investigation. Due to the wide range of potential attacks and the high degrees of attack sophistication, identifying what constitutes a *true* attack is a challenging problem – especially for organizations performing critical operations, like military bases or financial institutions, that are constantly subjected to high volumes of cyber-attacks every day. In this thesis, we develop a framework that allows us to study what game-theoretic behavior looks like from both the attacker and defender's perspective. Our approach considers a series of sub-games between the attacker and defender in which a state is maintained between each sub-game. We first derive optimal allocation strategies via the use of dynamic programming and Q-maximin value iteration based algorithms. We then move into approximation techniques, using deep neural networks and Q-learning to derive near-optimal strategies that allow us to explore much larger models. We assess the effectiveness of our allocation strategies by comparing them to other sensible heuristics (e.g., random and myopic) with our results showing that we consistently outperform these other strategies at minimizing risk.

1. INTRODUCTION

1.1 Motivation and Scope

The current state of cyber-defense is dire. On one hand, new trends of more sophisticated, economically-driven, state-sponsored cyber attacks are on the rise; evidenced by the recent security breaches at both public and private institutions (e.g., the Pentagon [1], Sony [2], Target [3]). On the other hand, the emergence (and deployment) of Cyber-Physical Systems have enabled cyber attacks to cross from the cyber realm into our physical infrastructure, making them appealing vehicles for terrorism and terrorism-related activities [4]. Various organizations, in both the public and private sectors, are constantly being subjected to attacks that seek to disable, disrupt, and/or breach their cyber infrastructure - especially during times of critical operations (e.g., before missions, political statements, software releases, etc.). The authors in [5] report an average of 17,000 alerts every week at surveyed organizations. Of these, roughly 19% (3,218) are estimated to be legitimate alerts, with only 4% (705) eventually being investigated. This relatively low proportion of assignment makes the defender's allocation strategy that much more critical when it comes to minimizing risk.

1.2 Cyber Emergency Response Teams

One critical component of any institution's cyber-defense infrastructure is maintaining the appropriate workforce of cyber-security analysts who handle the investigations of incoming alerts. When an attack is launched against an organization, sensors (e.g., Intrusion Detection Systems, Anti-malware tools, etc.) deployed on various systems or machines (e.g., computers, servers, routers, etc.) typically generate cyber-alerts with varying levels of severity. At the same time legitimate network activity can generate false positive alerts. This typically high volume of false positives can drown out relevant alerts, as seen in the Target

attack wherein malware alerts were repeatedly generated but not addressed [3]. To determine if an alert is a false positive it must first be assigned to a security analyst for investigation. Some of these alerts (e.g., those in a high risk category) have a higher probability of representing an ongoing attack and hence must be prioritized when being assigned for investigation. As such, an allocation of cyber-security analysts to cyber-alerts may depend on various factors, namely: (1) the expertise of the analysts, (2) the current availability of analysts, (3) the value of the machine from which the alerts originate, and (4) the associated threat level of the alerts.

1.3 The Need for Game Theory

It is common practice amongst cyber-security practitioners to operate under an assumption of the worst-case scenario. For example, the Pentagon is much more concerned with the threat of a high-profile zero-day attack from state sponsored professional hackers than they are with lone-wolf black-hat hobbyists using re-purposed code they found online. This general heuristic of "assuming the worst" motivates the use of game-theoretical defense policies where both agents (in our case, the defender and the attacker) are assumed to be rational and in possession of symmetric game information. Indeed, it could be possible for sophisticated adversaries to gain critical knowledge through probing attacks that observe the response time of the analysts in handling the attacks, and thus could choose the best attack method along with the correct time to strike (e.g., when all the analysts are busy investigating other alerts). Furthermore, the game-theoretic property of Nash equilibrium provides us with a useful notion of optimal play wherein the agents' strategies are robust against any non-Nash opponent (i.e., even if an attacker is not acting rational we achieve a reasonable utility). However, unlike previous works that considered a one-shot game with deterministic alert arrivals [6; 7], we consider a Markov game model in which the defender and the attacker play a series of games with a *state* maintained between

games. The stochastic nature of the Markov game is manifested in not knowing the exact numbers/levels of alerts that will arrive in the future, which can impact how allocations are made at the current time as well as the mixed strategy nature of the agents' policies. This novel application of a Markov game to the alert allocation problem allows us to achieve a finer grained view of play with more dynamic strategies that change with the state of the system.
We explore two solution methods in this domain, the first uses *brute force* dynamic programming and Q-maximin value iteration to find optimal solutions in small-scale models. The second uses approximate deep Q-learning methods which are capable of scaling to arbitrarily large model sizes while still maintaining near-optimal performance.

1.4 Related Works

Improving the scheduling and efficiency of cyber-security analysts is a highly studied area of research [8; 9; 10; 11]. The authors in [8] model the problem as a two-stage stochastic shift scheduling problem in which the first stage allocates cyber-security analysts and in the second stage additional analysts are allocated. The problem is discretized and solved using a column generation based heuristic. The authors in [9] study optimal alert allocation strategies with a static workforce size and a fixed alert generation mechanism. In [10] the authors develop a reinforcement learning-based dynamic programming model to schedule cyber-security analyst shifts with the model based on a Markov Decision Process framework with stochastic load demands. In [11], the author describes different strategies for managing security incidents in a cyber-security operation center. The authors in [12] propose a queuing model to determine the readiness of a Cyber-Security Operations Center (CSOC). This thesis departs from this previous research by explicitly considering the presence of a strategic and well informed adversary.
The use of game theory has been instrumental in advancing the state-of-the-art

in security games and their wide range of applications [13; 14; 15; 16; 6; 7]. Some of the most recent and relevant lines of work involving analyst alert allocation are studied in [6] and [7]. Here the authors introduce a game-theoretic model to determine the best allocation of incoming cyber alerts to analysts. Their model, however, assumes a one shot-game in which both the alert resolution time and the arriving alert distribution are deterministically known.

In contrast to the previously mentioned Stackelberg games, the authors in [17; 18; 19] explore solution methods to stateful Markov games. Convergence properties and Q-minimax value iteration are studied in [17; 18] providing encouraging guarantees for optimality and convergence of value functions. The authors in [19] use a least squares policy iteration approach to train a linear function approximator to predict Q-values.

The use of stateful Markov game models to study security games with real-world applications are limited. The authors in [20] used dynamic programming and value iteration to investigate attacks on power grids. Using a similar method, the authors in [21] investigated the use of full state space value iteration in the alert assignment domain. The authors in [22] used Markov games to model the level of worst-case threat faced by an institution given parameters surrounding its network infrastructure. Due to the use of full state space value iteration many previous works' solutions scale poorly to large real-world sized models, prompting the need for approximate solution methods in the security domain.

2. BACKGROUND

This chapter will cover background information regarding Markov decision processes (MDPs), game theory, reinforcement learning (RL), Markov games, and deep reinforcement learning.

2.1 Markov Decision Processes

An MDP is defined by the tuple $< S, A, T, R, \gamma >$ where:

- S is a finite set of states
- A is a finite set of actions
- T is a function mapping states and actions to a probability distribution over next states (e.g., $T : S \times A \rightarrow PD(S)$)
- R is a reward function that gives the utility returned when taking action a in state s (e.g., $R : S \times A \rightarrow \Re$)
- $\gamma \in [0, 1)$ is a discounting factor

Generally speaking, an MDP is a discrete time stochastic control process that provides a mathematical framework for modeling single agent decision making in a system where outcomes are partly random and partly under the control of some acting agent. MDPs have been used in many disciplines such as robotics, industrial control, and economics to find a working policy for the agent that attempts to maximize long-term expected rewards.

Given some currently inhabited state s an agent commits to some action a, receives a reward $r(s, a)$, and then the environment transitions to some next state s' wherein the process repeats. The goal of an MDP is to find some optimal policy π that maps states to actions in a way that maximizes the long-term expected reward, $\mathbb{E}\{\sum_{t=0}^{\infty} \gamma^t \; r(s_t, a_t)\}$, where $a_t = \pi(s_t)$.

There are many ways to solve the MDP optimization problem, with most methods being variants of either linear or dynamic programming (DP). Assuming we know both the reward function and the transition function, we can use dynamic programming via a value iteration approach to achieve an optimal solution.

In value iteration, a lookup-table V maintains a value for every $s \in S$. This lookup-table tells us the *value* of each state in our system. To learn the long-term expected reward we must iterate over this lookup-table while bootstrapping from previous iterations. This process begins with the initialization phase given in equation 2.1. Once the initialization phase is complete, we can begin iterating over equation 2.2 until our value function converges (i.e., when $V_t = V_{t+1}$). It is at this point that we can construct our optimal policy using the converged value function $\pi(s) = \text{argmax}_a \left[r(s,a) + \gamma \sum_{s'} T(s,a,s') V_{conv}(s') \right]$.

$$V_0(s) = 0, \ \forall s \in S \tag{2.1}$$

$$V_t(s) = \max_a \Big[r(s,a) + \gamma \sum_{s'} T(s,a,s') V_{t-1}(s') \Big], \ \forall s \in S, \ t > 0 \tag{2.2}$$

Due to the monotonic updating and the finite number of possible policies, such an algorithm is guaranteed to converge. A detailed proof of convergence as well as a thorough overview of MDP theory can be found in [23].

2.2 Game Theory

Game Theory is a broad discipline with implications in economics, social sciences, evolutionary biology, and computer science. It is primarily concerned with the study of rational actors behaving in an adversarial environment. Actors who are described as *rational* will always attempt to maximize their utility in a given game. A game is accompanied by a utility matrix whose rows and columns specify what actions each player can take. Each cell within the utility matrix

defines the utility awarded to either agent under the corresponding action-pair. The canonical example of a utility matrix is that of the game rock-paper-scissors, presented in Table 2.1.

Table 2.1: Utility matrix for a game of rock-paper-scissors

		Player 2		
		Rock	*Paper*	*Scissor*
Player 1	*Rock*	0,0	-1,1	1,-1
	Paper	1,-1	0,0	-1,1
	Scissor	-1,1	1,-1	0,0

In this example, Player 1 is the row player while Player 2 is the column player. It is also important to note that every cell's payoff to the players sums to zero. This is known as a zero-sum game, meaning each outcome is equally beneficial and detrimental to either player. Every player in a game maintains a strategy that informs how that agent takes actions. For simplicity, we can define this strategy as a probability distribution over their available actions (this is known as a mixed-strategy).

It was shown by Von Neumann in [24] that there exists at least one adversarial equilibrium point in any zero-sum game. This equilibrium point, commonly called the Nash equilibrium, defines the optimal strategy profiles for players in a zero-sum simultaneous-move game.

Informally, a set of player strategies is in a Nash equilibrium if neither player can benefit from unilaterally changing their strategy. If all players look at their opponent's strategy and cannot benefit from modifying their own, then the game has reached a Nash equilibrium point (i.e., either player's strategy is a best response to the strategy of their adversary).

Formally, a player's Nash equilibrium mixed strategy can be found by turning a utility matrix into a system of linear equations and maximizing for one players utility under certain constraints. A linear program solving for player 1's mixed strategy can be seen in Equations 2.3 - 2.6.

$$\max_{\pi(a_1)\in\Pi(\mathcal{A}_1)} u \tag{2.3}$$

$$u \leq \sum_{a_2\in\mathcal{A}_2} \pi(a_1)M(a_1,a_2) \tag{2.4}$$

$$\sum_{a_1\in\mathcal{A}_1} \pi(a_1) = 1 \tag{2.5}$$

$$0 \leq \pi(a_1) \leq 1, \qquad \forall a_1 \in \mathcal{A}_1 \tag{2.6}$$

Objective 2.3 is the value of the game, constraint 2.4 guarantees the worst-case over the actions available to the opposing player where M is an $\mathcal{A}_1 \times \mathcal{A}_2$ reward matrix. While constraints 2.5 and 2.6 ensure that we have a valid probability distribution. The solution to this linear program yields some mixed strategy $\pi(\cdot)$. If we solve this linear program for both players' π, then we can use the joint probabilities of all action pairs to calculate the expected value of playing that game under a Nash equilibrium (i.e., the value of the game).

2.3 Reinforcement Learning

Reinforcement learning (RL) is an area of unsupervised machine learning whose goal is to understand what actions an agent should take in an environment so as to maximize some concept of a cumulative reward. For example, one could imagine a 2-D maze where an agent can move in any cardinal direction (except when that direction possesses a wall) and seeks to reach the end of the maze. Each movement to another cell would reward the agent -1 utility with the only exception being the goal cell (the end of the maze), which would reward 0 utility. In this environment the agent's goal would be to reach this goal cell as quickly as possible and to remain their indefinitely.

Many problems (i.e., environments) can be formulated as an MDP and solved using the methods described in Section 2.1, however, what separates RL from classical DP solutions in these environments is that classical DP methods assume

that the exact mathematical model of the MDP is known. For instance, in Equations 2.1 & 2.2 both the transition and reward functions are known and we have a complete list of every state in the environment. In RL it is common to start with no knowledge of either the transition or reward function, and is typically applied to environments where the state space is prohibitively vast – meaning value iteration approaches like the one discussed in the previous section would become intractable. For these reasons RL uses approximations and sampling methods to learn an implicit model of the MDP, often balancing exploration (trying new things with the hope of learning something new) with exploitation (doing what we already know works). Due to the vast size of the state spaces typically being solved it is common to parameterize the states using some set of features that attempt to describe similar states in a similar way, making function approximations easier to learn (see section 2.5).

Q-learning is a popular RL algorithm that provably converges to an optimal policy for any finite MDP [25]. The algorithm uses the Q function that defines the quality of a state-action pair in the environment, given by $Q : S \times A \to \mathbb{R}$. The Q function's iterative update is defined by the following equation:

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha)\underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\substack{\text{learning}\\ \text{rate}}}\overbrace{\Big(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\substack{\text{discount}\\ \text{factor}}}\underbrace{\max_a Q(s_{t+1}, a_{t+1})}_{\substack{\text{estimate of optimal}\\ \text{future value}}}\Big)}^{\text{learned value}} \tag{2.7}$$

In each time step t, a state-action pair is realized within the environment and the agent updates its Q-value according to its old value and the new experienced value. These updates are weighted by some learning rate $0 < \alpha \leq 1$ that controls the rate at which new experiences replace old ones. The new experienced value consists of a sample from the reward function r_t and a $0 < \gamma \leq 1$ discounted greedy step that estimates the best action to take in the next state s_{t+1}. This greedy max step represents the exploitation step of the algorithm. To incorporate exploration many algorithms typically employ an ϵ-greedy approach

using a value $0 < \epsilon \leq 1$ where in each state if a random value is less than epsilon we choose a random action for a_{t+1}, otherwise we choose our exploitative max action. This exploration ensures that our agent will fully explore the state-action space and helps to avoid getting trapped in sub-optimal policies. This epsilon value typically starts close to 1 and decays over time as the agent becomes more intelligent.

Typically an agent is placed in a random state wherein many trajectories are rolled out so as to explore that *area* of the state space. Once a reasonable number of trajectories are performed we can once again randomize to some new starting state and repeat this process. Empirically, this style of learning works quite well, and with some restrictions can be shown to provably converge to optimal policies. A detailed explanation of Q-learning as well as proofs of convergence are provided in [25].

2.4 Markov Games

An n-player Markov game is defined by the tuple
$< S, A_1, ..., A_n, T, R_1, ..., R_n, \gamma >$ where:

- S is a finite set of states
- $A_1, ..., A_n$ are the finite sets of actions available to each agent
- $T : S \times A_1 \times ... \times A_n \to \Pi(S)$ is a function mapping every combination of actions the agents could take in the current state to a probability distribution over next states
- $R_i : S \times A_1 \times ... \times A_n \to \Re$ for $1 \leq i \leq n$ is the reward function for agent i that provides the utility awarded for every state-action pair in the system
- $\gamma \in [0, 1)$ is a discounting factor

Succinctly, a Markov game is a game-theoretic instantiation of an MDP. Rather than having a single agent interacting with the environment, we now have any

number of them. In every state s each agent commits to a single action a_i in their action space A_i, each agent then received a real-valued reward r_i from their reward function R_i, and the transition function is invoked evolving the system to some next state. Similar to MDPs, the goal of each agent is to find some optimal policy π_i that maximizes their long-term expected reward $\mathbb{E}\{\sum_{t=0}^{\infty} \gamma^t \; r_i(s_t, a_1, a_2, \; \ldots \; , a_n)\}$, where $a_i \sim \pi_i(s_t)$, from any state in the system. The main difference herein being that agents must also consider the actions and wishes of other (not necessarily adversarial) agents present in the system as well. These other agents' actions also contribute to the evolution of the system's state and therefore are a principal factor in understanding how to maximize one's own cumulative reward.

Solving Markov games has remained a difficult task as the run-time complexity of solving games with linear programs can be extremely cumbersome when combined with the exponential growth of the system's state-space. While Littman explored the use of Q-minimax reinforcement learning on Markov games in [18] and later provided convergence guarantees in [17], the methods used required an enumeration and storage of all state-action pairs in the Markov game. This limited his experimentation to smaller games where such tables could fit in memory. Approximation techniques were explored in [19] where the authors used a least squares policy iteration algorithm in conjunction with Monte Carlo simulations to learn approximate Nash policies with reasonable success. Overall, while the applicability of Markov games seems to be vast the number of fast and robust solution methods seems to be quite limited.

2.5 Deep Reinforcement Learning

Deep reinforcement learning refers to reinforcement learning that uses deep neural networks as the function approximators that attempt to map input features about states and actions to some real valued reward in a stationary environment. While the use of function approximation with Q-learning is

basically as old as the algorithm itself, these approximations were typically linear (and even worse, used hand-crafted features).

Take for example the chess playing algorithm Deep Blue, which utilized over 8,000 hand-crafted features for its binary-linear value function whose weights were almost entirely hand-tuned by expert human chess players [26]. While such an algorithm was capable of beating the world champion Gary Kasparov in 1997, its method of doing so was arduous and largely influenced by human knowledge and experience.

Contrast these linear function approximators with more contemporary deep neural network approaches capable of learning both their own internal features and weights with little to no human input. The obvious advantage of such algorithms is that one can have little to no domain knowledge about the task at hand and yet can still train agents to perform at an expert level.

The remainder of the section will focus on the methods presented in [27]. This work's authors describe an algorithm capable of learning to play a multitude of Atari games (in many cases at a super-human level) from pixel input alone. The algorithm was robust enough that it could be applied across many different games with very different goals – while still yielding great policies. This generality of deep RL, coupled with the need for little to no domain knowledge from human practitioners, makes it a very promising area of research. Please note that the algorithm presented in Algorithm 1 has been simplified for the purposes of this thesis. The original algorithm and methods presented in [27] involved an image pre-processing technique that we will not be discussing.

Most traditional reinforcement learning algorithms use value iteration to converge to the optimal value function (see section 2.3). Such value iteration approaches become unpractical in very large state spaces since there is no aliasing between states or actions. In other words, while two state-action pairs may be highly semantically similar, traditional methods will estimate their values completely separately. Instead, we can opt to use a function approximator (e.g., a

linear function parameterized by some weights θ and features ϕ) to estimate the action-value function. We can then train such an approximator by minimizing a sequence of loss functions $L_i(\theta_i)$ that change with each iteration of our algorithm.

$$y_i = \mathbb{E}_{s\sim\mathcal{E}}\big[r(s,a) + \gamma \max_{a'} Q(s', a'; \theta_{i-1})|s, a\big] \tag{2.8}$$

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\Big[\big(y_i - Q(s, a; \theta_i)\big)^2\Big] \tag{2.9}$$

Equation 2.8 is the target we are moving our approximator towards, with $\mathcal{E}$ being the environment we sample states from. Equation 2.9 is the loss function, with $\rho(s, a)$ representing the behavior distribution (in many cases simply the current ϵ-greedy policy) that defines a probability distribution over states and actions. Differentiating equation 2.9 with respect to the weights gives us the following,

$$\begin{aligned}\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\mathcal{E}}\Big[\nabla_{\theta_i} Q(s, a; \theta_i) \cdot \\ \Big(r(s,a) + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)\Big)\Big]\end{aligned} \tag{2.10}$$

To avoid computing the full expectations of equation 2.10 we can use single samples of actions from the behavior distribution ρ and transitions from our environment $\mathcal{E}$ while using stochastic gradient descent to minimize the loss. This allows us to reduce this process to the simple Q-learning algorithm discussed in Section 2.3.

Most supervised machine learning algorithms rely on learning a fixed distribution given some set of samples. The difficulty in using such methods (e.g., stochastic gradient descent) in an RL context is that the samples obtained from an on-policy RL algorithm come from the estimator itself. Every time we update our policy we change the agent's behavior and thereby the distribution of rewards we are going to see. This kind of moving target can lead to convergence issues and is addressed in part via the use of experience replay [28]. Experience

Algorithm 1 Deep Q-Network with Experience Replay

Initialize replay memory $\mathcal{Z}$ to capacity N
Initialize action-value function Q with random weights θ_0
for episode $i = 1, M$ **do**
  Initialize s_1 randomly and extract its features $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
      otherwise select $a_t = \text{argmax}_a\, Q(\phi(s_t), a; \theta)$
    Execute action a_t in $\mathcal{E}$ and observe reward r_t and next state s_{t+1}
    Set $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{Z}$
    Sample random mini-batch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{Z}$
    Set $y_j = r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta_{i-1})$
    Take gradient descent step on $\big(y_j - Q(\phi_j, a_j; \theta_i)\big)^2$ using equation 2.10
  end for
end for

replay is a method where we store a 4-tuple of the agent's experience $e_t = \langle s_t, a_t, r_t, s_{t+1} \rangle$ at each time step in some large data set $\mathcal{Z} = e_1, ..., e_N$ from which we can sample from in a way analogous to traditional supervised learning. In standard Q-learning samples are thrown away after each transition, whereas experience replay allows us to store and replay transitions many times. Additionally, since we sample randomly from our experience we are able to avoid the highly correlated nature of learning from consecutive transitions in a given area of the state space, thereby reducing the variance of our updates. Also notice that in Equation 2.8 we are using the weights from the previous episode. This helps keep our target stationary when performing updates. Again, it helps to think of the algorithm from a traditional supervised learning perspective – if the data distribution you are approximating changes every time you update your learning architecture, learning will prove to be quite difficult.

3. MODEL

This chapter will begin by formally describing the Markov game formulation of the cyber-alert assignment problem being addressed. We will then describe two optimizations used to reduce the run-time of our algorithms.

3.1 Cyber-Alert Assignment Markov Game

We consider a two-player zero-sum Markov game in which the Defender ($\mathcal{D}$) and the Attacker ($\mathcal{A}$) play a series of sub-games over an infinite time horizon. At each time step t, a new batch of alerts $\omega \in \Omega$ arrives in which $\mathcal{A}$ chooses some alert level(s) to attack in and $\mathcal{D}$ attempts to detect and thwart the incoming attack(s) by assigning available analysts to the incoming alerts. Without loss of generality a time step may represent any period of elapsed time, however for the remainder of this thesis we will assume a time step of one hour. We let $s \in \mathcal{S}$ denote the current state of the player resources (e.g., availability of analysts as well as the budget available to the attacker). We also let $\mathcal{D}_a$ and $\mathcal{A}_a$ denote the set of actions available to $\mathcal{D}$ and $\mathcal{A}$, respectively. We define a transition function $T : \mathcal{S} \times \mathcal{D}_a \times \mathcal{A}_a \to \Pi(\mathcal{S})$ which maps each state and player action pair to a probability distribution over possible next states. We let $T(s, a, d, s')$ denote the probability that, after taking actions $a \in \mathcal{A}_a$ and $d \in \mathcal{D}_a$ in state s, the system will make a transition into s'. In general, the system can be described as follows:

- **Alerts:** Our transition function T is manifested in the uncertainty of which batch of alerts will arrive in each state. At every time step t, some batch of alerts $\omega \in \Omega$ arrives according to a probability distribution $\Pi(\Omega)$, where $\Omega = \{\omega_1, \omega_2, \ldots, \omega_{|\Omega|}\}$. Each alert $\sigma \in \omega$ belongs to one of three categories: High (h), Medium (m), or Low (l). Resolving an alert requires a certain number of time steps based on its category. The set holding each category's work-time (i.e., the number of time steps needed by an analyst

to investigate and resolve an alert) is defined as $\mathcal{W} = \{w_h,\ w_m,\ w_l\}$ where $w_h > w_m > w_l$ and $w_h, w_m, w_l \in \mathbb{N}$. A similar reward structure $\mathcal{U} = \{u_h,\ u_m,\ u_l\}$, where $u_h > u_m > u_l$ and $u_h, u_m, u_l \in \mathbb{N}$, is defined for each category as well. If the alert σ^h is legitimate and is assigned to an analyst, $\mathcal{D}$ will receive a positive utility u_h. Whereas not assigning the legitimate alert results in a negative utility $-u_h$ for $\mathcal{D}$. If an alert is illegitimate (i.e., a false positive) then it awards no utility to either the attacker or defender, regardless of whether or not it is assigned. Since our model is zero-sum, the corresponding utilities for $\mathcal{A}$ are simply the additive inverse of those awarded to $\mathcal{D}$. We assume that both agents are aware of the set of possible alert batches Ω, as well as the respective arrival probabilities of each batch. For example, a possible arrival set may be as follows: $\Omega = \{\omega_1,\ \omega_2\}$ where $\Pr(\omega_1) = 0.4$ with $\omega_1 = \{\sigma_1^h,\ \sigma_2^m,\ \sigma_3^m\}$, and $\Pr(\omega_2) = 0.6$ with $\omega_2 = \{\sigma_1^m,\ \sigma_2^l\}$. At the beginning of a particular game, both agents are aware of the exact alert batch that has arrived (with future alert arrivals remaining probabilistic, thereby impacting the current resource allocations made by the agents). It is important to note that not every alert may be assigned to an analyst, and not every alert may represent a legitimate attack (i.e., alerts can be false positives).

- **The Defender:** $\mathcal{D}$ has n homogeneous cyber-security analysts available to handle incoming alerts. We define R_s as a vector of length n that describes the load of each analyst in state s. For example, $R_s = [0\ 2\ 1]$ means that $\mathcal{D}$ has 3 analysts on their team in which analyst 1 is free (has load 0), analyst 2 will be free after two time steps, and analyst 3 will be available after one time step. We also define the function $\mathcal{F}(R_s)$ as the number of analysts currently available for allocation in state s. In every time step t, $\mathcal{D}$ receives a batch of alerts ω and determines their allocation strategy based upon the current availability of analysts and the varying severity and volume of alerts in ω. Once the set of possible alert batches Ω and each alert category's

respective work-times $\mathcal{W}$ are known, we can construct the set of all possible analyst states. In general, $|R|$ is bounded above by the following:

$$|R| \leq (w_h + 1)^n \tag{3.1}$$

- **The Attacker**: $\mathcal{A}$ has an attack budget $B \in \mathbb{N}$ and decides when to attack and in what category. We assume $\mathcal{A}$ knows the alert level that would be generated due to their attack. The set $\mathcal{C} = \{c_h,\ c_m,\ c_l\}$ defines the respective cost to the attacker given the alert level their attack generates, where $c_h > c_m > c_l$ and $c_h, c_m, c_l \in \mathbb{N}$. $\mathcal{A}$ can attack with as many alerts as they wish as long as the sum of their costs is affordable given their current budget. The attacker's budget enables us to model the amount of risk they are willing to undertake. Attacking more frequently with attacks that generate high level alerts would likely expose $\mathcal{A}$. To capture this behavior, if $\mathcal{A}$ chooses to abstain from attacking in a state s with budget B_s they will be credited with 1 unit of budget in the subsequent state. However, their budget is capped to some value B_{max} representing the maximum amount of risk they are willing to undertake.

- **State Representation:** At the beginning of a time step, we assume that the system state is known to both agents. A state is thus defined as follows:

$$s = [R_s | B_s] \tag{3.2}$$

Given the state s and alert arrival ω, both the action space of the defender $\mathcal{D}_a(s, \omega)$ and the action space of the attacker $\mathcal{A}_a(s, \omega)$ can be generated. The size of the defender's action space, given in Equation 3.3, describes all possible combinations of assigning/not assigning the incoming alerts $\sigma \in \omega$ to the available analysts. Since we consider all analysts to be homogeneous we do not need to consider which particular analyst an alert is assigned to, but only

whether or not the alert is assigned. Equation 3.4 enumerates all the ways $\mathcal{A}$ could attack in the alerts in ω based on the current budget in s as captured by the indicator function $\mathbb{1}_{\{\cdot\}}$ (while also allowing them to abstain from attacking altogether)

$$|\mathcal{D}_a(s,\omega)| = \sum_{i=0}^{\mathcal{F}(R_s)} \binom{|\omega|}{i} \tag{3.3}$$

$$|\mathcal{A}_a(s,\omega)| = \sum_{i=1}^{2^{|\omega|}} \mathbb{1}_{\{B_s \,\geq\, Bin(i-1)\,\cdot\,\mathcal{C}_\omega\}} \tag{3.4}$$

where $Bin(i)$ is the binary representation of i and $\mathcal{C}_\omega$ is the cost vector for the alerts in ω according to $\mathcal{W}$. For example, an alert arrival $\omega = \{\sigma_1^m,\ \sigma_2^l,\ \sigma_3^l\}$ yields a cost vector $\mathcal{C}_\omega = [3\ 1\ 1]$. Thus, given an attacker budget $B_s = 2$ for this state and arrival $\mathcal{A}_a = \{[0\ 0\ 0],\ [0\ 1\ 0],\ [0\ 0\ 1],\ [0\ 1\ 1]\}$.

Based on the alert arrival and system state we can formulate a zero-sum game. Every one of these state-arrival pairs constitutes a sub-game where the defender attempts to detect an attack through an assignment of alerts to analysts and the attacker attempts to avoid detection when launching an attack that would result in some alert combination of the chosen categories. This game can be represented as a matrix $\mathcal{R}_s$ of instantaneous rewards with $\mathcal{R}_s^{\mathcal{D}}(a,d)$ denoting the utility awarded to the defender and $\mathcal{R}_s^{\mathcal{A}}(a,d)$ denoting the utility awarded to the attacker when $\mathcal{D}$ chooses action $d \in \mathcal{D}_a(s,\omega)$ and $\mathcal{A}$ chooses action $a \in \mathcal{A}_a(s,\omega)$. Recall that since we are playing a zero-sum game $\mathcal{R}_s^{\mathcal{D}}(a,d) = -\mathcal{R}_s^{\mathcal{A}}(a,d)$.

Both defender and attacker follow policies $\pi_{\mathcal{D}}$ and $\pi_{\mathcal{A}}$, respectively (e.g., $\pi_{\mathcal{D}}(s,d)$ is the probability $\mathcal{D}$ takes action $d \in \mathcal{D}_a(s,\omega)$). Given the current state s and alert arrival batch ω, we can formulate a zero-sum game over the payoff matrix $\mathcal{R}_s$ and solve it with the linear program presented in equations 2.3-2.6 of section 2.2 to derive these policies.

Once derived, both agents will then sample from their Nash equilibrium mixed strategy and commit to their respective actions. They are then awarded their

respective utility and the state evolves from s to s' according to the transition function T.

3.2 Action Space Compression

We can represent our agents' actions as a binary vector where a 1 represents that a given alert is either assigned to an analyst for the defender, or attacked in by the attacker. For example, if we are in a state $s = \big[R = [0,0,0] \mid B = 10\big]$ and a batch of alerts $\omega = \{\sigma_1^h, \sigma_2^h, \sigma_3^m\}$ arrives, where $c_h = 5$ and $c_m = 3$, our defender and attacker action spaces would be formulated as follows:

$$\mathcal{D}_a(s,\omega) = \{[0,0,0],\ [0,0,1],\ [0,1,0],\ [0,1,1],\ [1,0,0],\ [1,0,1],\ [1,1,0],\ [1,1,1]\}$$

$$\mathcal{A}_a(s,\omega) = \{[0,0,0],\ [0,0,1],\ [0,1,0],\ [0,1,1],\ [1,0,0],\ [1,0,1]\}$$

Where $d_6 = [1,0,1]$ means that the defender assigns alerts σ_1^h and σ_3^l and ignores alert σ_2^h. Similarly $a_4 = [0,1,1]$ means that the attacker attacks in alerts σ_2^h and σ_3^l and ignores alert σ_1^h.

However, consider the fact that alerts of the same severity level are homogeneous (i.e., their utility, work-time, and cost are equal). This means that our agents need not worry about which specific alerts are being assigned/attacked, only the number of alerts from each severity level being assigned/attacked. Thus we can represent actions as a 3-tuple $\langle h, m, l\rangle$ representing how many alerts from each severity level are being assigned/attacked:

$$\hat{\mathcal{D}}_a(s,\omega) = \{\langle 0,0,0\rangle,\ \langle 0,1,0\rangle,\ \langle 1,0,0\rangle,\ \langle 1,1,0\rangle,\ \langle 2,0,0\rangle,\ \langle 2,1,0\rangle\}$$

$$\hat{\mathcal{A}}_a(s,\omega) = \{\langle 0,0,0\rangle,\ \langle 0,1,0\rangle,\ \langle 1,0,0\rangle,\ \langle 1,1,0\rangle\}$$

where $d_3 = \langle 1,1,0\rangle$ means that the defender assigns one of the two high alerts and the one medium alert. Similarly $a_3 = \langle 1,0,0\rangle$ means that the attacker attacks in one of the high alerts and ignores the other high and medium alert. While this allows us to reduce the sizes of both player's action spaces, the defender receives the majority of the compression since their action space grows

combinatorially with respect to the number of alerts present in the arrival batch ω. This combinatorial growth severely increases the run-time of solving our sub-games which thereby increases the run-time of solving the Markov game as a whole.

To illustrate just how advantageous this compression is, consider an arrival $\omega_s = \{\sigma_1^h, \sigma_2^h, \sigma_3^m, \sigma_4^m, \sigma_5^l, \sigma_6^l\}$ in a state where the number of available analysts $\mathcal{F}(R_s) = 5$. Under the combinatorial action space $|\mathcal{D}_a(s, \omega)| = 63$ whereas the compressed action space $|\hat{\mathcal{D}}_a(s, \omega)| = 26$ (a 58% reduction). The compression is even more substantial when arrival batches posses more redundancy. For instance, if our $\omega_s = \{\sigma_1^h, \sigma_2^h, \sigma_3^h, \sigma_4^h, \sigma_5^h, \sigma_6^h\}$ then $|\hat{\mathcal{D}}_a(s, \omega)| = 6$ while our combinatorial action space remains unchanged at $|\mathcal{D}_a(s, \omega)| = 63$ (while this small example is given for illustration purposes, in our larger models the compression can routinely get as high as 96%).

Under the compressed action space formulation joint action rewards are calculated in expectation since we no longer specify exactly which alerts are being assigned/attacked. The simplified equation for deriving the compressed payoff matrix rewards $\hat{\mathcal{R}}(\cdot)$ is presented below:

$$z_{\max} = \min(a_k, \sigma_k)$$

$$z_{\min} = \begin{cases} 0, & \text{if } d_k - (\sigma_k - a_k) < 0; \\ d_k - (\sigma_k - a_k), & \text{otherwise.} \end{cases}$$

$$\hat{\mathcal{R}}(a, d) = \sum_{k \in \{h,m,l\}} \sum_{z=z_{\min}}^{z_{\max}} \frac{\binom{a_k}{z}\binom{\sigma_k - a_k}{d_k - z}}{\binom{\sigma_k}{d_k}} \, u_k(2z - a_k)$$

where σ_k is the number of alerts in severity level k, d_k and a_k are the number of alerts assigned/attacked from severity level k, and z represents the number of alerts caught by the defender in the current severity level.

Most advantageous of all is that this compression is completely loss-less with respect to finding the value of a Nash equilibrium in our sub-games (i.e., whether

using $\mathcal{R}$ or $\hat{\mathcal{R}}$ our linear program will find the same expected values). The mixed-strategies will necessarily be different (since the action spaces are different), however the mixed strategies derived from $\hat{\mathcal{R}}$ will be more meaningful as they they contain much less redundancy.

3.3 Fictitious Play

The biggest bottleneck for both of the algorithms developed in this thesis is the game solving mechanism. Solving the game via a linear programming approach, while accurate, incurs a a large cost in terms of run-time. In both of our solutions we are solving millions of games – often containing hundreds of potential action pairs. Thus while solving one of these games is more or less instantaneous, solving our Markov game can take many hours.

This motivated us to explore potential alternatives to using linear programming to solve our games. Ultimately we settled on the use of fictitious play, an iterative algorithm first introduced by G.W. Brown in 1951 [29]. In fictitious play each player tracks the empirical frequency of actions chosen by their opponent and best responds to this strategy. The other player, having also tracked the empirical frequency of their opponent, also plays their best response. By iterating this process many times we are able to derive close approximations of both the game's value and the Nash equilibrium policies of the agents. A proof of convergence for this iterative approach is given in [30].

The algorithm we use for our fictitious play was first introduced by J.D. Williams in [31] and is described in Algorithm 2. (please note that element-wise vector operations are implied here, with scalar values being broadcast to all elements of the vector in question).

After generating and solving 10,000 random games with $iterations = 500$ we found that on average the iterative fictitious play solver was around 20 times faster than the linear programming approach while the Nash game value it provided exhibited only a 6% error compared to the exact value obtained via

linear programming. Furthermore, in both of our solution methods we care more about the Nash value of a game than the Nash strategies. Since fictitious play is better at approximating Nash values rather than the strategies it makes it a perfect fit for our purposes.

Algorithm 2 Iterative Fictitious Play

$\mathcal{R}$ is an $m \times n$ matrix of rewards
$rowReward$ and $rowCnt$ are m-length arrays of zeros
$colReward$ and $colCnt$ are n-length arrays of zeros
Initialize $bestResponse$ to any random row action
for $iterations$ **do**
  $colReward = colReward + \mathcal{R}[bestResponse, \cdot\,]$
  $bestResponse = \text{argmin}(colReward)$
  $colCnt = colCnt + 1$
  $rowReward = rowReward + \mathcal{R}[\,\cdot\,, bestResponse]$
  $bestResponse = \text{argmax}(rowReward)$
  $rowCnt = rowCnt + 1$
end for
$gameValue = \Big(\big(\max(rowReward) + \min(colReward)\big) \,/\, 2\Big) \,/\, iterations$
$rowMixedStrat = rowCnt \,/\, iterations$
$colMixedStrat = colCnt \,/\, iterations$

4. SOLUTION METHODS

This chapter will formally describe the two approaches we used to solve the cyber-alert assignment Markov game. The first uses a Q-minimax value iteration approach that enumerates all states' Nash values explicitly. The second employs deep reinforcement learning to approximate Nash Q-values given features about a given state-action pair.

4.1 Q-minimax Value Iteration

The dynamic programming approach we use was first studied in [21] where Q-minimax values were learned via the use of value iteration. The authors explored both infinite and finite horizon domains whereas we restrict our experimentation to only the infinite domain.

To solve a Markov game we want to modify the value iteration approach described in 2.1 using the game-theoretic principles of optimal play discussed in 2.2, allowing our agents to maximize their worst-case expected utility. To do this we must first notice that every state's payoff matrix is incomplete in that it only reflects the immediate rewards each action pair may lead to. If we wish to find an optimal policy, we need every state's expected value to consider the future rewards available from that state. This is accomplished by replacing each action pair utility in the state's payoff matrix $\mathcal{R}_s$ with the value of the minimax quality function $Q(s, a, d)$, yielding the Q-matrix $\mathcal{Q}_s$. This new reward matrix includes the immediate reward $\mathcal{R}_s(a, d)$ as well as the expected discounted rewards provided by $Q(s, a, d)$. Furthermore, since we are operating in a zero-sum Markov game we need only store the value function of the defender as the attacker's value function is simply the additive inverse of the defender's.

It is important to note that while we do use the quality function Q to denote an action's long-term expected reward, the algorithm presented here is not a

Q-learning algorithm. Q-learning is model-free – meaning it has no explicit representation of environment dynamics, whereas our algorithm is given the transition function when calculating the quality function.

The value function is updated according to the following equations:

$$V_0(s) = 0, \qquad t = 0 \tag{4.1}$$

$$V_t(s) = \max_{\pi_{\mathcal{D}} \in \Pi_{\mathcal{D}}} \min_{a \in \hat{\mathcal{A}}_a} \sum_{d \in \hat{\mathcal{D}}_a} \pi_{\mathcal{D}}(s,d) Q_t(s,a,d), \qquad t > 0 \tag{4.2}$$

$$Q_t(s,a,d) = \mathcal{R}_s^{\mathcal{D}}(a,d) + \gamma \sum_{s' \in S} T(s,a,d,s') V_{t-1}(s'), \qquad t > 0 \tag{4.3}$$

Notice that in the first update ($t = 1$), equation 4.2 reduces to simply solving for the value of the game in each state under a Nash equilibrium because equation 4.3 has its second term reduced to zero. Each successive update then uses the previous iteration's values under a Nash equilibrium to find its own. This iterated nesting of the Nash equilibrium value allows us to build a sub-game Nash equilibrium that reflects the future rewards available from any state when all agents are playing game-theoretically. In short, We have simply replaced expected reward gained from a state with the expected reward of *playing a game* in that state under a Nash equilibrium. The pseudo-code is provided in Algorithm 3.

Algorithm 3 Maximin Value Iteration

Initialize:
$\gamma = 0.95$
for all s **do**
  $V_0(s) = \text{GameSolver}(\mathcal{R}_s)$
end for
Learn:
for $t = 1$ to iterations **do**
  for all s in S **do**
    for all a in $\hat{\mathcal{A}}_a(s,\omega)$ and d in $\hat{\mathcal{D}}_a(s,\omega)$ **do**
      $\mathcal{Q}_s(a,d) = \mathcal{R}_s(a,d) + \gamma \sum_{s'} T(s,a,d,s') V_{t-1}(s')$
    end for
    $V_t(s) = \text{GameSolver}(\mathcal{Q}_s)$
  end for
end for

where GameSolver can be either the linear program provided in 2.2 or the iterative fictitious play approach given in Algorithm 3.3.

4.2 Deep Nash Q-Network

Initial attempts at approximation for us were largely unsuccessful. Naturally we began the process with linear models using Q-learning or SARSA algorithms in conjunction with Monte Carlo rollouts, but these models were able to learn little to nothing – even in the presence of hand crafted features. This lead us to believe that estimating the sub-game Nash equilibrium values in our Markov game was just too complex for a linear function to successfully approximate. This is a reasonable assumption given the complex nature of the nested game playing. Motivated by the success in [27] we wanted to explore the possibility of applying Deep Q-Networks in the Markov game domain. After all, even complex systems like those of Atari games can be formulated as MDP's. And the transformation from the single agent MDP to the multi-agent Markov game is straightforward enough that the approximation power achieved in [27] would hopefully carry over.

This move from single agent to multi-agent would naturally necessitate some changes to Equations 2.8-2.10. Namely, while both algorithms employ a Q function to estimate future rewards attainable from an action pair, Equation 2.8 need only apply a greedy max over the next action for a single player whereas our game theoretic approach must choose actions for two players in a minimax fashion. Given the convergence proofs for value iteration in Markov games given in [17] and the empirical success of its use in our domain in [21], we can be quite sure the substitution of a game theoretic maximin in place of the greedy max will provide a stable and meaningful reward signal. Our modified equations are as follows:

$$y_i = \mathbb{E}_{s\sim\mathcal{E}}\big[\mathcal{R}_s(a,d) + \gamma \max_{d'\in\hat{\mathcal{D}}_a} \min_{a'\in\hat{\mathcal{A}}_a} Q(s',a',d';\hat{\theta})|s,a,d\big] \tag{4.4}$$

$$L_i(\theta_i) = \mathbb{E}_{s,a,d\sim\rho(\cdot)}\Big[\big(y_i - Q(s,a,d;\theta_i)\big)^2\Big] \tag{4.5}$$

$$\begin{aligned}\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a,d\sim\rho(\cdot);s'\sim\mathcal{E}}\Big[&\nabla_{\theta_i} Q(s,a,d;\theta_i)\cdot \\ &\Big(\mathcal{R}_s(a,d) + \gamma \max_{d'\in\hat{\mathcal{D}}_a} \min_{a'\in\hat{\mathcal{A}}_a} Q(s',a',d';\hat{\theta}) - Q(s,a,d;\theta_i)\Big)\Big]\end{aligned} \tag{4.6}$$

The authors in [27] mention that small changes to network weights can lead to large differences in derived policies. If this is true for single agent learning then this observation is at least doubly true in the case of our two agents. When using θ_{i-1} as our target network we experienced poor learning rates and could not achieve convergence of the loss function. To remedy this we employed a hyper-parameter τ that defines an update cycle for our target network. For instance, if $\tau = 100$ then every 100 updates we will set our target network's weights $\hat{\theta}$ to the current learning network's weights θ_i.

This essentially freezes the distribution of data we are approximating for τ learning steps. This addition has worked quite well in providing stability during learning. As always, this hyper-parameter required some tuning as smaller update cycles failed to remedy the issue and large update cycles lead to big spikes in loss when the target network was updated.

After the addition of hyper-parameter τ we experienced no issues with loss convergence and were able to make some promising observations regarding the approximation accuracy of the network. Upon initialization of the network and before any learning had occurred, Q-estimates of the network were very close to zero. This meant that the first τ steps of learning were calculating loss gradients on immediate reward alone. Once this first cycle was complete we then compared the predicted Q-values to the known immediate rewards and found a very high accuracy. Our algorithm was starting out its learning in a way quite similar to the value iteration approach in section 4.1.

This learning architecture, which we refer to as a Deep Nash Q-Network (DNQN), learns off-policy and is model-free. The off-policy learning stems from the fact that the learning gradients are calculated over samples of transitions taken at a different time and under a different policy. Off-policy learning, as with DQN's, allows us to leverage all the benefits of experience replay discussed at the end of section 2.5. The pseudo-code for DNQN learning is presented in Algorithm 4:

Algorithm 4 Deep Nash Q-Network with Experience Replay

Initialize $i = 0$
Initialize learning network with random weights θ_i
Initialize target network weights $\hat{\theta} = \theta_i$
Initialize τ to desired update cycle
Initialize replay memory $\mathcal{Z}$ to capacity N
for e $= 1, M$ **do**
  Initialize s_1 randomly
  for $t = 1, T$ **do**
    $\mathcal{Q}_s$ = GetQMatrix(s, θ_i)
    $\langle \pi_{\mathcal{A}},\ \pi_{\mathcal{D}} \rangle$ = GameSolver($\mathcal{Q}_s$)
    With probability ϵ select a random action a_t
      otherwise sample $a_t \sim \pi_{\mathcal{A}}$
    With probability ϵ select a random action d_t
      otherwise sample $d_t \sim \pi_{\mathcal{D}}$
    Execute actions $\langle a_t, d_t \rangle$ in $\mathcal{E}$ and observe reward r_t and next state s_{t+1}
    Store transition $(s_t, a_t, d_t, r_t, s_{t+1})$ in $\mathcal{Z}$
    Sample random mini-batch of transitions $(s_j, a_j, d_j, r_j, s_{j+1})$ from $\mathcal{Z}$
    $\mathcal{Q}_{s_{j+1}}$ = GetQMatrix(s_{j+1}, $\hat{\theta}$)
    $\langle v_{s_{j+1}} \rangle$ = GameSolver($\mathcal{Q}_{s_{j+1}}$)
    Set $y_j = r_j + \gamma v_{s_{j+1}}$
    Take gradient descent step on $\big(y_j - Q(s_j, a_j, d_j; \theta_i)\big)^2$ using equation 4.4
    Set $i = i + 1$
    if i mod $\tau == 0$, **then**
      $\hat{\theta} = \theta_i$
    end if
  end for
end for

where GetQMatrix(s, θ) is a function that re-populates the payoff matrix $\mathcal{R}_s$ with Q-values predicted using network weights θ and GameSolver($\mathcal{Q}_s$) is a function that solves the matrix game $\mathcal{Q}_s$ and returns the Nash equilibrium value v_s of the game as well as the mixed strategies for both players, $\pi_{\mathcal{A}}$ and $\pi_{\mathcal{D}}$, that

result in that value.

While we experimented with a number of hand-crafted feature representations our best results were obtained by simply normalizing the most basic information about the current state and agent actions. These features were as follows:

- For each analyst we calculated the current percentage of their remaining wait time resulting in n features
- The percent of currently available budget for the attacker
- Three features specifying the number of alerts from each severity level that had arrived (min-max normalized)
- Three features for the number of alerts assigned from each severity level (min-max normalized)
- Three features for the number of alerts attacked in each severity level (min-max normalized)

Additionally, all experiments presented herein were performed using the Adam optimizer described in [32] in favor of the simple stochastic gradient descent approach. While both optimizers performed well in practice, Adam consistently lead to smoother learning curves and better performing policies.

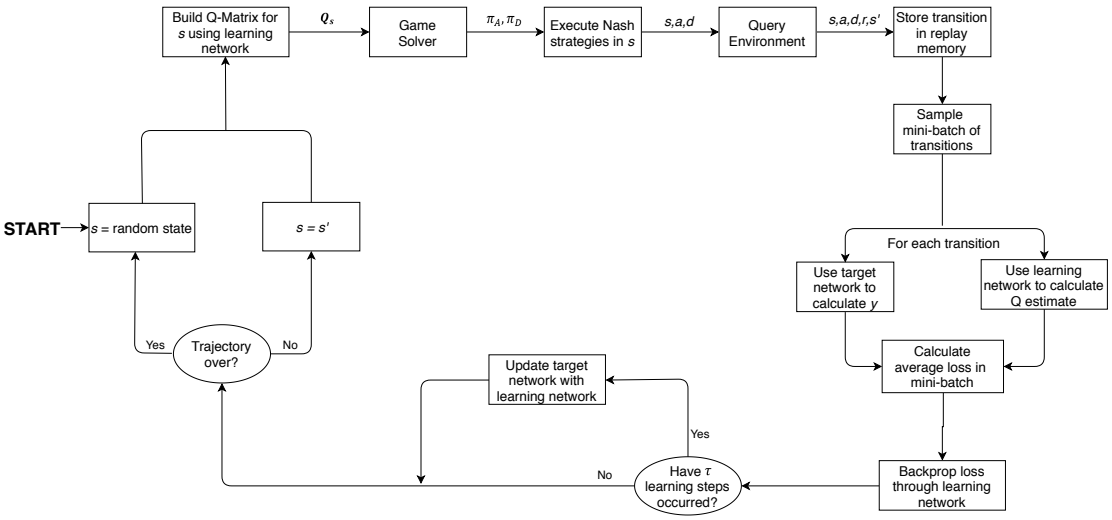

Figure 4.1: Diagram of the DNQN learning process

5. PERFORMANCE EVALUATION

This chapter will present our experimental results. All results presented herein were obtained on a machine with two-dozen 2.2GHz cores and 64GB of RAM. Furthermore, the dynamic programming approach was run in a fully parallel manner while the DNQN approach was run serially.

5.1 Dynamic Programming Tractable Model

The first set of results we will discuss were obtained within a state space small enough to be solved in a reasonable amount of time (30 hours) via the brute force dynamic programming approach discussed in Section 4.1. The parameters used when constructing the state space are presented in Table 5.1 and yield an environment with a total of 2,117,682 possible states our agents can inhabit.

Table 5.1: Parameters used when constructing the DP tractable model.

Parameter	Value
Number of experts n	5
Attack budget B	20
Utilities $\mathcal{U}$	$u_h = 100$, $u_m = 20$, $u_l = 5$
Attack cost $\mathcal{C}$	$c_h = 8$, $c_m = 4$, $c_l = 2$
Work times $\mathcal{W}$	$w_h = 6$, $w_m = 3$, $w_l = 1$
Alert batches in Ω where $\omega = \langle h, m, l \rangle$	$\omega_1 = \langle 0, 2, 2 \rangle$, $\omega_2 = \langle 1, 2, 2 \rangle$, $\omega_3 = \langle 0, 3, 3 \rangle$ $\omega_4 = \langle 1, 1, 4 \rangle$, $\omega_5 = \langle 2, 2, 3 \rangle$, $\omega_6 = \langle 3, 3, 3 \rangle$
Arrival prob. $\Pi(\Omega)$	$\omega_1 = 0.15$, $\omega_2 = 0.21$, $\omega_3 = 0.21$ $\omega_4 = 0.18$, $\omega_5 = 0.20$, $\omega_6 = 0.05$

The dynamic programming solution provides us with an example of what optimal behavior looks like in the Markov game domain, allowing us to understand how well our approximate DNQN approach compares in terms of both long-term cumulative utility and solution time.

Figure 5.1 plots the convergence of our value function under the dynamic programming approach. Due to the zero-sum nature of our games, we only

present the value function of the Defender as the Attacker's is simply its additive inverse. Once the average percent change between iterations approaches 1% we consider the value function converged, and end the program (this occurred around iteration 500).

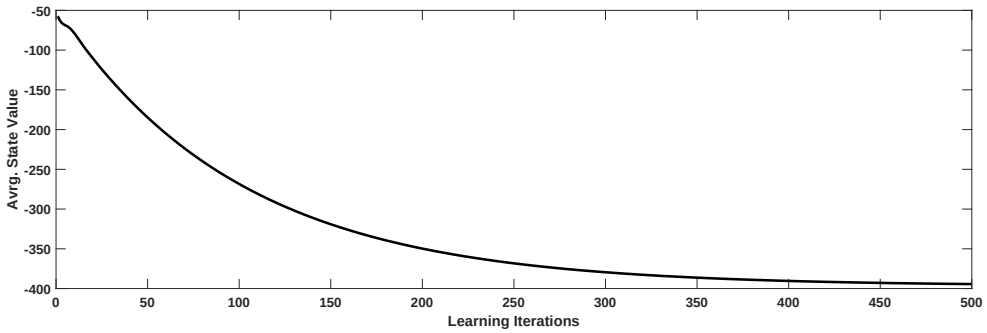

Figure 5.1: Convergence of the value function for the dynamic programming approach

When training our DNQN we used a fully connected $32 \times 64 \times 128 \times 128 \times 128$ architecture with ReLU activation functions and an update cycle $\tau = 1,000$. We experimented with various network sizes and update cycle lengths, finding the aforementioned values to be both the most stable and fruitful. While they did impact the overall accuracy of the model, our solution remained quite robust across all the network sizes we tried.

Figure 5.2 presents the mean loss of our network's predictions after each iteration. As discussed in Section 4.2, over the first update cycle (the first 1,000 iterations) our agent's are primarily learning immediate rewards and our network's loss drops quite rapidly. However, when the second update to the target network takes place our loss suddenly spikes by roughly 2,000%. This initial convergence and sudden spike demonstrate two things. First, our network can quickly learn the immediate rewards of a given action pair. Second, these immediate rewards are a poor reflection of long term reward (Q-values). After the first update cycle our agents understand immediate reward very well meaning solving the Q-matrices from Algorithm 1 yield near-perfect Nash play. However, once the normal form game becomes an extensive form game their strategies

must re-adjust greatly as their greedy actions now bear heavy consequences.

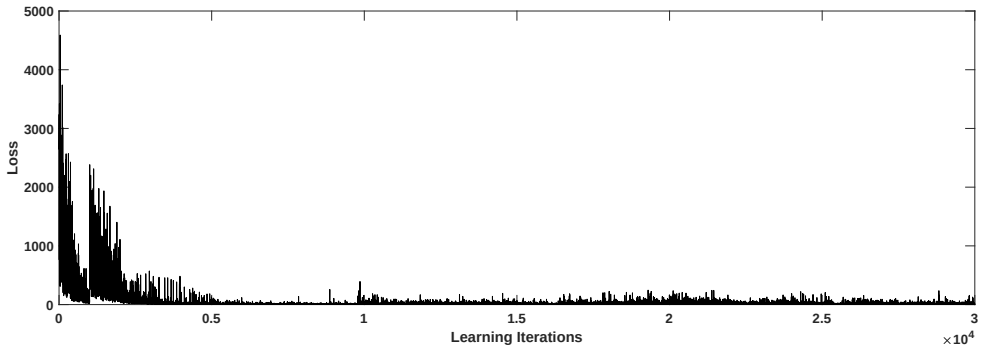

Figure 5.2: Loss values obtained while training the DNQN on the DP tractable model

While the early learning exhibits very noisy loss values this seems to stabilize drastically after about 5,000 iterations. Initially we thought this steadiness was implying that our network had essentially finished learning after the first 5,000 updates. To investigate this suspicion we ran simulations against an optimal dynamic programming opponent after each update cycle and plotted the average cumulative utility in Figure 5.3.

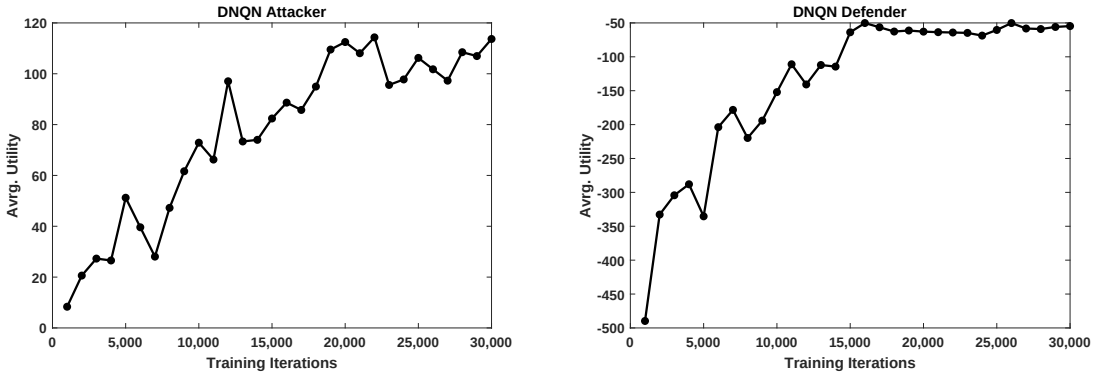

Figure 5.3: DNQN agents' utility against a DP opponent at various stages of learning

The results of these simulations show that both agents continue to learn well past the point implied by the loss curve. It is interesting to note that the DNQN Defender seems to reach convergence at around the 15,000 iteration mark while the DNQN attacker continues to progress up until the end of its training. Taken together, Figures 5.2 and 5.3 show that while later iterations' exhibit rather

accurate Q-value predictions (i.e., low loss) the strategies derived from these Q-values continue to evolve throughout the learning process.

After 30,000 training iterations we want to compare our solution methods' policies against one another to understand how well they perform with respect to cumulative utility. We also include two heuristic policies, random and myopic. A random policy for either agent simply randomizes over their action space in each state. A myopic policy plays as if the agent is in a normal form game, solving the payoff matrix of immediate rewards in each state and then sampling from the derived mixed strategy.

For each of the 16 possible policy pairs we run 1,000 independent simulations with a time horizon of 100 rounds (starting from random states). The average discounted cumulative utility of each round for the attacker and defender are presented in Figures 5.4 and 5.5, respectively. The title of each sub-graph indicates the opponent policy while the lines plotted therein show the utility obtained by the agent in question.

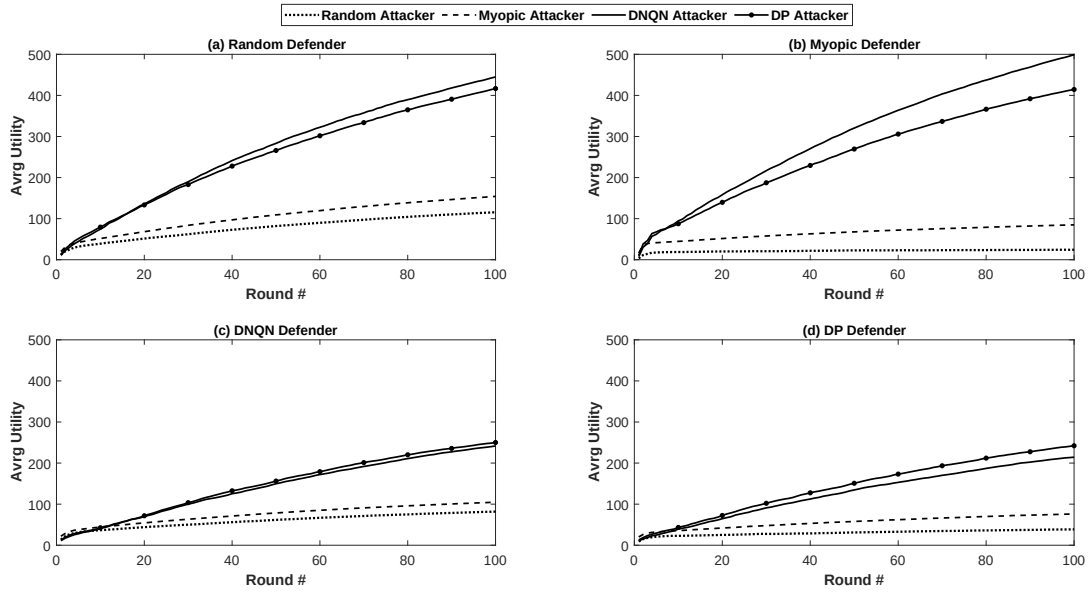

Figure 5.4: Average cumulative utility obtained by the various attacker policies in the DP tractable model

Figure 5.4 presents the utility obtained by the various attacker policies. The DNQN attacker came very close to the optimal DP attacker against all defender policies, and even surpassed its utility against the random and myopic defenders.

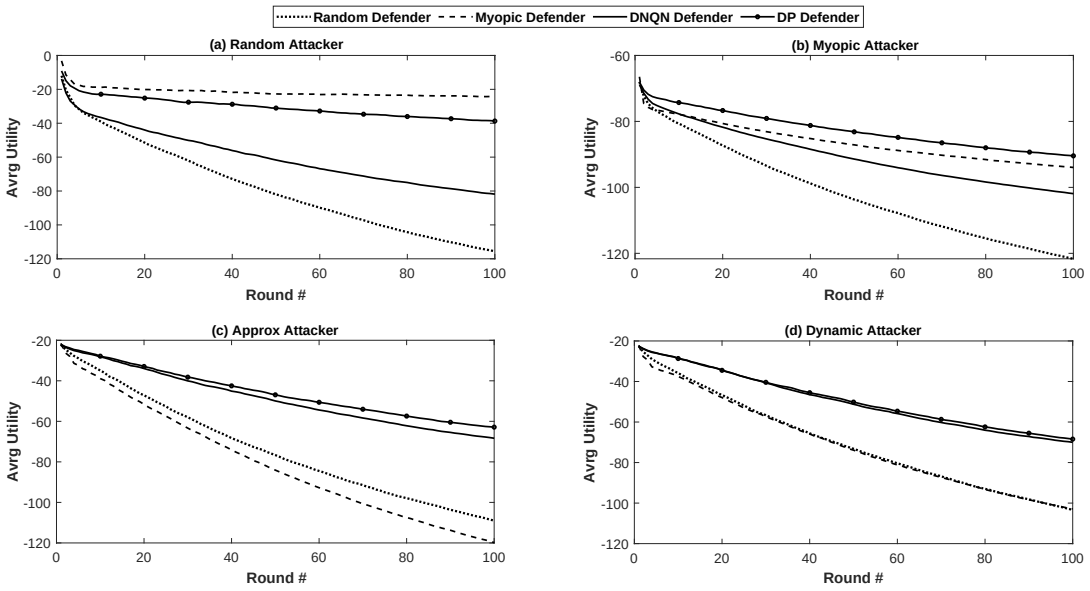

Figure 5.5: Average cumulative utility obtained by the various defender policies in the DP tractable model

Figure 5.5 presents the utility obtained by the various defender policies. In contrast to the DNQN attacker, our DNQN defender falls behind the DP policy when against random and myopic opponents. The reason for this has to do with the distribution of states our DNQN defender sees during learning. As the agents get more intelligent the DNQN attacker begins to ignore low severity alerts and stockpile budget until the defender inevitably has a high number of analysts assigned. Since the DNQN defender is learning in tandem with this attacker, it learns to be wary of over-assigning analysts and also begins to ignore low alerts. Since there is only one action to not attack and typically many actions representing an attack, the random attacker consistently has a low budget. This leads to many more low attacks (20x more than a DNQN attacker in our simulations) as they are all it can afford. This forces our DNQN defender into areas of the state-action space it has not typically encountered during learning – leading to poor performance. The DP defender however gives equal weight to every state in the system giving it a better understanding of optimal play in even the sub-optimal areas of the state space.

5.2 Dynamic Programming Intractable Model

The second set of results we will discuss were obtained within a state space too large to be solved by the dynamic programming approach. The parameters used when constructing the state space are presented in Table 5.2 and yield an environment with a total of 2.1 billion states. Performing the 30,000 training updates on our DNQN took little over 5.5 hours in this environment. For comparison, a liberal estimate obtained by extrapolating the run-times in Section 5.1 would put the DP solution time at roughly 114 years. Furthermore, our DNQN estimates state-action pair values (i.e., Q-values) which most likely number in the tens of billions as each state can have potentially dozens of action pairs.

Table 5.2: Parameters used when constructing the DP intractable environment.

Parameter	Value
Number of experts n	8
Attack budget B	30
Utilities $\mathcal{U}$	$u_h = 100$, $u_m = 20$, $u_l = 5$
Attack cost $\mathcal{C}$	$c_h = 8$, $c_m = 4$, $c_l = 2$
Work times $\mathcal{W}$	$w_h = 6$, $w_m = 3$, $w_l = 1$
Alert batches in Ω where $\omega = \langle h, m, l \rangle$	$\omega_1 = \langle 0,2,2 \rangle$, $\omega_2 = \langle 0,3,3 \rangle$, $\omega_3 = \langle 1,2,5 \rangle$ $\omega_4 = \langle 1,3,6 \rangle$, $\omega_5 = \langle 1,4,4 \rangle$, $\omega_6 = \langle 2,2,2 \rangle$ $\omega_7 = \langle 2,2,4 \rangle$, $\omega_8 = \langle 2,3,4 \rangle$, $\omega_9 = \langle 2,3,5 \rangle$ $\omega_{10} = \langle 3,3,3 \rangle$, $\omega_{11} = \langle 3,4,5 \rangle$, $\omega_{12} = \langle 4,5,6 \rangle$
Arrival prob. $\Pi(\Omega)$	$\omega_1 = 0.02$, $\omega_2 = 0.03$, $\omega_3 = 0.08$ $\omega_4 = 0.08$, $\omega_5 = 0.09$, $\omega_6 = 0.10$ $\omega_7 = 0.10$, $\omega_8 = 0.10$, $\omega_9 = 0.10$ $\omega_{10} = 0.13$, $\omega_{11} = 0.12$, $\omega_{12} = 0.05$

In state spaces as large as this it can be very difficult to understand what optimal behavior looks like. In the absence of our optimal DP solution we can make no concrete guarantees as to the efficacy of our results. Despite this fact, our algorithm still maintains a converging loss curve and a superior utility when compared to our previously mentioned random and myopic policies.

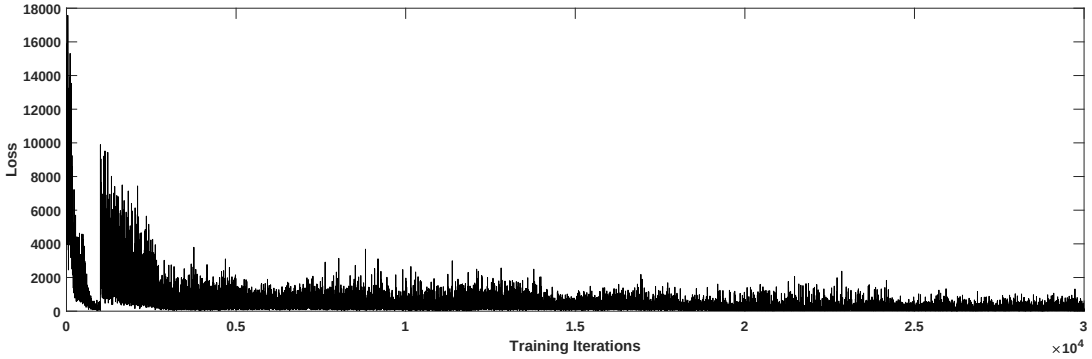

Figure 5.6: Loss values obtained while training the DNQN on the DP intractable model

Figure 5.6 shows the loss curve obtained while training the DNQN on the DP intractable model. Similar to Figure 5.2 we can see early convergence over the first update cycle followed by a large spike in loss as future rewards begin to be considered.

After the 30,000 training iterations we again want to compare our DNQN policy against the random and myopic policies. Without the DP solution we now have 9 possible policy pairs, again we run 1,000 independent simulations with a time horizon of 100 rounds to obtain the average cumulative utility in each policy pair. These utilities are presented for the attacker and defender in Figures 5.7 and 5.8, respectively.

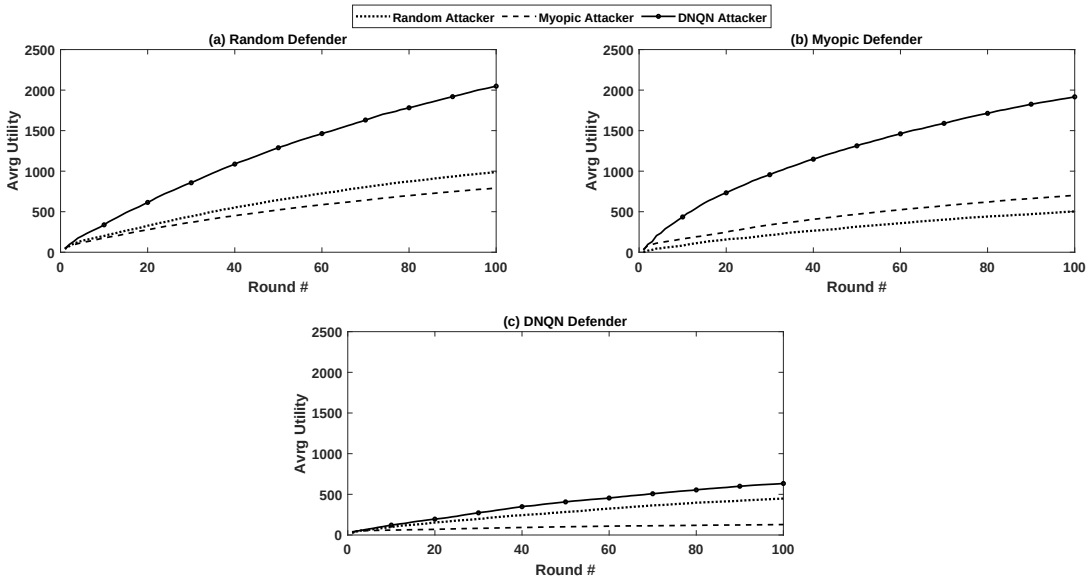

Figure 5.7: Average cumulative utility obtained by the various attacker polices in the DP intractable model

In Figure 5.7 we can see that the DNQN attacker maintains the highest utility across all prospective policies, performing exceptionally well against the random and myopic defenders. Further inspection of DNQN attacker strategies confirms that the agent consistently waits until the defender's analysts are mostly busy before attacking. Almost all attacks occurred when the analysts were above 75% utilization making it very difficult for the defender to stop the attack.

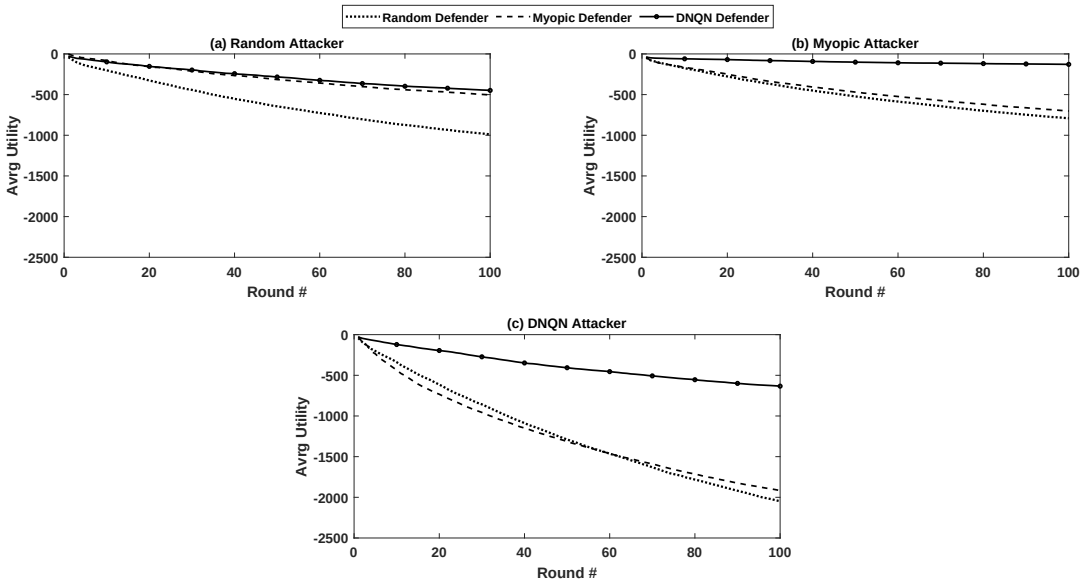

Figure 5.8: Average cumulative utility obtained by the various defender polices in the DP intractable model

Figure 5.8 presents the cumulative utility earned by the various defender policies. It appears that in these experiments the DNQN defender policy is only slightly better than the myopic policy at stopping a random attacker. As previously discussed, the problem of training these agents is that they naturally skew the distribution of state-action pairs observed towards more intelligent areas of the space. As with the smaller model this appears to again be the case. When investigating further into the DNQN defender strategy we notices that it would never fully allocate all its analysts, preferring to hover around a 75% utilization. Essentially the defender was playing a game of chicken with the attacker, trying to discourage attacks by always keeping some analysts ready for assignment. While this kind of behavior is quite irrational against the random and myopic attackers, the DNQN defender plays very well against the intelligent

DNQN attacker – almost more than quadrupling the utility of the other two policies. This represents a kind of trade-off between general coverage and acute prevention that is the core of a good defender policy.

It is important to note that, relatively speaking, an optimal attack policy is much easier to learn than an optimal defender policy. The attacker only needs to stockpile their budget (a single resource) until the defender allocates a high percentage of their analysts whereupon they flood the defender with attacks. Contrast this with the defender who must learn to balance the allocation times of their analysts (8 resources) with the expected volume of incoming alerts. We find it quite remarkable that we are able to derive two very different strategies from a single network while still maintain good performance for both.

As for implementing this kind of policy in the real world defenders would obviously always want to maintain 100% utilization of their analysts, keeping those who were unassigned by the model on call for reassignment if the model dictates. Furthermore, the strategies derived by our model could simply be viewed as a kind of threshold for game-theoretically sound behavior that could inform an organizations as to their level of security (or lack thereof). For instance, an organization seeing a terrible utility given the known volume of attacks they face may want to improve their coverage of lower level alerts or increase the number of analysts on hand.

6. CONCLUSION

In this thesis we provided a Markov game framework for modeling the adversarial interaction of computer network attackers and defenders in a game-theoretic manner. By framing this interaction as a series of zero-sum games wherein a state is maintained between each sub-game we were able to simulate long term periods of play and apply both traditional and novel reinforcement learning algorithms to derive intelligent policies.

An optimal minimax value iteration approach using dynamic programming was presented that was capable of learning Nash optimal policies given a model of its environment. Next, an approximate solution method using our deep Nash Q-network (DNQN) algorithm was presented that allowed for the aforementioned results to be extended to much larger state spaces where an explicit model of the environment was not known. This DNQN approach was capable of deriving intelligent policies on par with the optimal approach in a much shorter time and with less information about the environment.

These results motivate the use of DNQN-like architectures when solving very large Markov games as this approach proved to be both computationally expedient and empirically effective.

REFERENCES

[1] T. Brook and M. Winter, "Hackers Penetrated Pentagon Email," 2015.

[2] P. Elkind, "Inside the Hack of the Century."

[3] X. Shu, K. Tian, A. Ciambrone, and D. Yao, "Breaking the target: An analysis of target data breach and lessons learned," *CoRR*, 2017.

[4] J. Finkle, "U.S. official sees more cyber attacks on industrial control systems," 2016.

[5] P. Institute, "The Cost of Malware Containment, 2015."

[6] A. Schlenker, H. Xu, M. Guirguis, M. Tambe, A. Sinha, C. Kiekintveld, S. Sonya, N. Dunstatter, and D. Balderas, "Don't bury your head in warnings: A game-theoretic approach for intelligent allocation of cyber-security alerts," in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pp. 381–387, 2017.

[7] A. Schlenker, H. Xu, M. Guirguis, M. Tambe, A. Sinha, C. Kiekintveld, S. Sonya, N. Dunstatter, and D. Balderas, "Towards a game-theoretic framework for intelligent cyber-security alert allocation," in *Proceedings of the 3rd IJCAI workshop on Algorithmic Game Theory, Melbourne, Australia*, 2017.

[8] D. Altner and L. Servi, "A two-stage stochastic shift scheduling model for cybersecurity workforce optimization with on call options," 2016.

[9] R. Ganesan, S. Jajodia, S. A., and H. Cam, "Optimal Scheduling of Cybersecurity Analysts for Minimizing Risk," *ACM Trans. on Intelligent Systems and Technology*, 2015.

[10] R. Ganesan, S. Jajodia, S. A., and H. Cam, "Dynamic Scheduling of Cybersecurity Analysts for Minimizing Risk Using Reinforcement Learning," *ACM Trans. on Intelligent Systems and Technology*, 2016.

[11] C. Zimmerman, "Ten strategies of a world-class cybersecurity operations center," *MITRE corporate communications and public affairs*, 2014.

[12] A. Shah, R. Ganesan, S. Jajodia, and H. Cam, "A methodology to measure and monitor level of operational effectiveness of a csoc," *International Journal of Information Security*, pp. 1–14, 2017.

[13] M. Brown, A. Sinha, A. Schlenker, and M. Tambe, "One Size Does Not Fit All: A Game-Theoretic Approach for Dynamically and Effectively Screening for Threats," in *AAAI conference on Artificial Intelligence*, 2016.

[14] A. Sinha, T. Nguyen, D. Kar, M. Brown, M. Tambe, and A. Jiang, "From Physical Security to Cybersecurity," *Journal of Cybersecurity*, vol. 1, no. 1, pp. 19–35, 2015.

[15] Z. Yin, A. Jiang, M. Tambe, C. Kiekintveld, K. Leyton-Brown, T. Sandholm, and J. Sullivan, "Trusts: Scheduling randomized patrols for fare inspection in transit systems using game theory," in *Proceedings of the 24th IAAI*, (Palo Alto, CA), 2012.

[16] M. Jain, E. Kardes, C. Kiekintveld, F. Ordónez, and M. Tambe, "Security games with arbitrary schedules: A branch and price approach.," in *Proceedings of AAAI*, 2010.

[17] M. Littman, "Value-function reinforcement learning in markov games," *Princeton University Press*, 2000.

[18] M. L. Littman, "Markov games as a framework for multi-agent reinforcement learning," in *In Proceedings of the Eleventh International Conference on Machine Learning*, pp. 157–163, Morgan Kaufmann, 1994.

[19] M. G. Lagoudakis and R. Parr, "Value function approximation in zero-sum markov games," in *Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence*, UAI'02, (San Francisco, CA, USA), pp. 283–292, Morgan Kaufmann Publishers Inc., 2002.

[20] C. Y. T. Ma, D. K. Y. Yau, X. Lou, and N. S. V. Rao, "Markov game analysis for attack-defense of power networks under possible misinformation," *IEEE Transactions on Power Systems*, vol. 28, pp. 1676–1686, May 2013.

[21] N. Dunstatter, M. Guirguis, and A. Tahsini, "Allocating security analysts to cyber alerts using markov games," *2018 National Cyber Summit (NCS)*, 2018.

[22] C. Xiaolin, T. Xiaobin, Z. Yong, and X. Hongsheng, "A markov game theory-based risk assessment model for network information system," in *2008 International Conference on Computer Science and Software Engineering*, vol. 3, pp. 1057–1061, Dec 2008.

[23] R. A. Howard, *Dynamic Programming and Markov Processes*. Cambridge, MA: MIT Press, 1960.

[24] J. Von Neumann and O. Morgenstern, "Theory of games and economic behavior," 1947.

[25] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*. Cambridge, MA, USA: MIT Press, 1st ed., 1998.

[26] M. Campbell, A. Hoane, and F. hsiung Hsu, "Deep blue," *Artificial Intelligence*, vol. 134, no. 1, pp. 57 – 83, 2002.

[27] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013.

[28] L.-J. Lin, *Reinforcement Learning for Robots Using Neural Networks*. PhD thesis, Pittsburgh, PA, USA, 1992. UMI Order No. GAX93-22750.

[29] G. W. Brown, "Iterative solution of games by fictitious play," in *Activity Analysis of Production and Allocation* (T. C. Koopmans, ed.), New York: Wiley, 1951.

[30] J. Robinson, "An iterative method of solving a game," *Annals of Mathematics*, vol. 54, no. 2, pp. 296–301, 1951.

[31] J. D. Williams, *The Compleat Strategyst: Being a Primer on the Theory of Games of Strategy*. Dover, 1986.

[32] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014.