Variable-Player Learning for Simulation-Based Games

Madelyn Gatchel

May 2021

A thesis submitted to the faculty of Davidson College
in partial fulfillment of the requirements for
Departmental Honors in Computer Science.

Committee:
Bryce Wiedenbeck, *Advisor*
Raghu Ramanujan, *Second Reader*
Tim Chartier
Tabitha Peck
Carl Yerger

Acknowledgements

First I thank the Department of Mathematics & Computer Science for challenging and supporting me over the past four years as a mathematician, computer scientist, and person. You all have helped me find my passion for research and teaching, and I would not be going to graduate school, let alone graduate school in computer science, if not for your mentorship and support.

I also thank my major advisors, Dr. Smith and Dr. Mendes, for helping me manage the complicated course logistics between the math and computer science double major and honors, and for helping me navigate the challenges associated with a rigorous schedule. Dr. Smith, you have set an example for how to communicate complicated ideas in a way that is easy to understand, and I strive to explain concepts half as well as you do. Dr. Mendes, you have always conveyed your overwhelming confidence in me as a computer scientist, especially when I needed to hear it the most.

Dr. R, thank you for being there for all of the biggest moments from the start: convincing me to take CSC 121, encouraging me to do the DREU program and to apply for the AAAI-UC, and lastly for being the second reader for this thesis.

Dr. Kuchera, thank you for sharing your passion for and excitement about computer science and machine learning. I hope to embody that same level of enthusiasm with my future students.

Bryce: I think I could write a list longer than my longest list of questions with reasons I am thankful for you. In short, thank you for introducing me to algorithmic game theory, a field that combines elements from both my majors with a focus on solving real-world problems. I am excited to continue to explore this field in graduate school and beyond.

Contents

List of Figures

List of Algorithms

List of Tables

Abstract

Game theory is the branch of economics that aims to model how people or "agents" interact and make decisions. A normal-form game uses a payoff matrix to describe each agent's degree of happiness with each outcome. We study simulation-based games, a type of game where this payoff matrix is not known in advance but can be filled in through a series of multi-agent simulations. Traditionally, a normal-form payoff matrix is defined for a fixed number of players. In many real-world settings, the exact number of players is unknown, but we might know a range in which the number of players falls. In this thesis, we define variable-player games, present a machine learning technique to analyze variable-player simulation-based games, and discuss how this technique will enable more meaningful predictions about behavior in the real-world game.

1 Introduction

In this thesis we apply machine learning techniques to solve a problem in game theory which would otherwise be intractable. Many of the real-world interactions game theorists might want to model have a large, uncertain number of participants. However, current game-theoretic models assume a fixed number of players. We propose a new type of game-theoretic model which accommodates this uncertainty in the number of players. The approach uses machine learning to construct a variable-player game-theoretic model based on data from multi-agent simulations. We also extend current analysis techniques to analyze variable-player game models. In sections 1.1 and 1.2 we will introduce the subfields of game theory and machine learning, both of which are necessary to understand the relevance and novelty of this work. We will provide a more thorough background for each subfield in section 2. Then in section 1.3 we will expand on the real-world motivation for this thesis and highlight our main contributions.

1.1 Game Theory

Game theory is the branch of economics that aims to model self-interested participants or *agents* interacting and making decisions. In this definition, *self-interested* does not mean that the agents are selfish, but rather that each agent has certain outcomes that she prefers to others and will act strategically to try to bring about those outcomes. A *game* is a mathematical model of incentives for a particular interaction. A central objective in game theory is to construct and analyze a game-theoretic model in order to make predictions about future behavior in the interaction.

As interest in the field began to grow during the first half of the twentieth century with works from John von Neumann, Oskar Morgenstern, and John Nash [51, 35, 33, 34], researchers began to apply game theory to solve problems in political science [3, 10] and evolutionary biology [44, 48]. While game theory has always had strong ties to mathematics, it was not until more recently that the field of algorithmic game theory emerged [26, 36, 40]. *Algorithmic game theory* (sometimes called *computational game theory*) lies at the intersection of theoretical computer science, artificial intelligence, and economics. Within this subfield there are three main goals: to design algorithms or models to make game theory problems more tractable, to develop analyses that allow us to make claims about these algorithms or models with some certainty, usually via proof, and to understand incentives in computational systems.

Foundational to game theory is *preference theory*, which is based on the idea that every agent has preferences over the outcomes of a particular decision. An agent's preferences between arbitrary outcomes A and B can be described in one of three ways:

- The agent *prefers* A to B, meaning that the agent likes A at least as much as the agent likes B. This is denoted as $A \succeq B$.
- The agent *strictly prefers* A to B, meaning that the agent likes A more than the agent likes B. This is denoted as $A \succ B$.
- The agent is *indifferent* between A and B, meaning that the agent likes outcomes A and B equally. This is denoted as $A \sim B$.

Using an agent's preferences, we can construct a *utility function*, which tries to quantify an agent's degree of happiness with a particular outcome relative to alternative outcomes. This utility func-

tion becomes more complex when the interaction depends on others' preferences and actions. In the 1940s, John von Neumann and Oskar Morgenstern proved the following theorem [35]:

Theorem 1. *An agent will act to maximize their expected utility when faced with uncertainty about opponents' actions if and only if certain preference axioms are satisfied.*

For the purposes of this thesis, we assume that these preference axioms are always satisfied, and therefore all agents are expected utility maximizers. We will discuss expected utilities further in section 2.1.1.

Two of the most commonly studied types of games in game theory are non-cooperative games and cooperative games. For *non-cooperative games*, we care about how each individual agent acts and which decision they make, particularly when they face uncertainty over opponents' actions. For *cooperative games*, we care about how individual agents interact in groups or coalitions with respect to coalition formation and payoff division [29]. Note that the distinction is *not* based on whether the agents want to cooperate with one another. A *simultaneous-move game* is a type of non-cooperative game in which the agents play actions simultaneously as opposed to sequentially. Rock-Paper-Scissors is an example of a simultaneous-move game, whereas chess is an example of a sequential-move game.

A normal-form game is a type of simultaneous-move game. A *normal-form game* includes a set of players and, for each player, a set of available strategies as well as a utility function that describes their degree of happiness for each possible outcome. Figure 1 describes the relationships between these five types of games. In this thesis, we focus on normal-form games. In particular, we define a variable-player game by loosening the normal-form fixed-player requirement to account for uncertainties in the number of players. We use machine learning techniques to construct a variable-player game-theoretic model from simulation data.

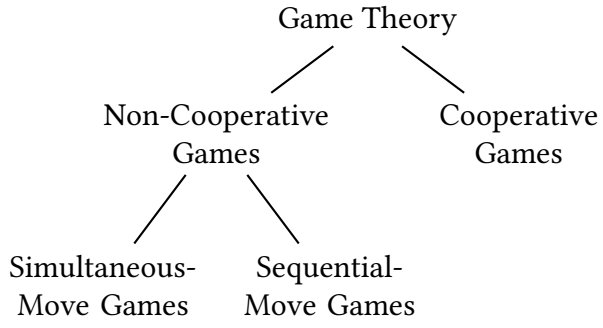


Figure 1: Both simultaneous-move games and sequential-move games are types of non-cooperative games, and normal-form games are a type of simultaneous-move game. In this thesis, we focus on simultaneous-move games, particularly normal-form games.

1.2 Machine Learning

Machine learning is the subfield of computer science that uses mathematical techniques to learn functions from data, frequently through an iterative refinement process. It is a subset of *artificial intelligence*, the branch of computer science that leverages computational techniques to

imitate actions or decision-making processes that usually involve human intelligence. Many recent machine learning applications have also involved elements of *data science*, a field devoted to extracting meaning from data. Finally, *deep learning* is a subset of machine learning that generally refers to neural networks. Figure 2 shows the relationships between artificial intelligence, machine learning, deep learning and data science. In practice, the boundaries between the four areas are not so rigid. For example, *theoretical machine learning*, an area devoted to proving run time and correctness guarantees for machine learning algorithms, would be classified as machine learning but not data science. Thus this figure primarily gives a high-level overview of the relationships between the four areas.

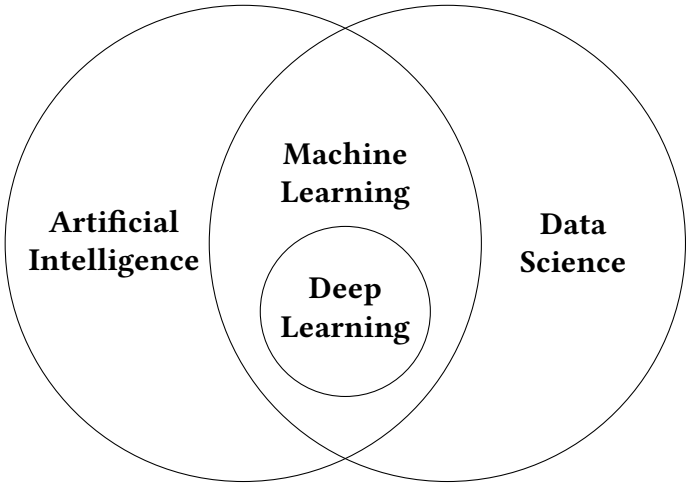


Figure 2: Machine learning falls at the intersection of artificial intelligence and data science.

The three main categories of machine learning include unsupervised learning, supervised learning, and semi-supervised learning. The type of learning used is frequently determined by the data available for a given problem. Data can either be *labeled*, meaning that each data point has both inputs and outputs, or it can be *unlabeled*, meaning that each data point contains only input values and no outputs are known. *Supervised learning* involves learning a mapping from inputs to outputs based on labeled data. This mapping is then used to predict outputs for new, unlabeled input data. Traditionally, most deep learning models are supervised learning models. *Unsupervised learning* involves learning a function from unlabeled data. This function seeks to identify patterns or relationships among the input data points. Unsupervised learning techniques include clustering and principal component analysis. *Semi-supervised learning* involves a combination of supervised and unsupervised learning. The most common example of semi-supervised learning is *reinforcement learning*. Note that inputs or input variables are sometimes referred to as independent variables or features; similarly, outputs or output variables are sometimes referred to as dependent variables, outcomes, or targets.

In this thesis, the function we are trying to learn has continuous outputs, which makes it a *regression* task. In particular, we use deep learning to infer the deviation payoff function from simulator data for a variable-player game.

1.3 Motivation

While the field has strong foundations in theoretical computer science, algorithmic game theory is driven by many real-world applications. Traditionally, game theorists have studied auctions, elections, and fair division of goods. In addition, algorithmic game theory has been applied to settings such as kidney exchange [1, 2, 8, 9], wildlife poaching prevention [64, 24, 11], physical security and cybersecurity [60, 32, 41, 47, 39], finance [54, 6, 53], and games like poker and Go [5, 43, 42].

Our techniques are particularly relevant for analyzing *simulation-based games*, where a normal-form payoff matrix is not known in advance but can be filled in through a series of multi-agent simulations. Simulation-based games are often used to model real-world interactions. In many of these settings, the exact number of players is both *large* and *unknown*. For example, suppose we want to model traders interacting in the stock market. We can reasonably assume that the number of traders in the interaction will be large and we might be able to predict a range in which the exact number of traders falls, but we likely will not ever know exactly how many traders are actively participating. As the number of players grows in a game, so does the size of the payoff matrix. This presents challenges when trying to compute equilibria, which represent predictions about behavior in the real-world interaction. More specifically, most equilibrium-finding algorithms require summing over the entire payoff matrix several times which can be computationally expensive when the payoff matrix is large. We address this problem using machine learning to generalize from partial data from the game.

Since the interaction we are studying might have an uncertain number of participants, a normal-form game with a fixed number of players might be an insufficient model. Thus, we focus on analyzing games with a variable number of players, where the number of players falls in a specified range. We hypothesize that the payoffs in a game with x players are similar or related to the payoffs in the same game with $x \pm 1$ players, given a large value of x. In the context of the stock market example: given the large number of traders present, adding one more trader likely will not change the overall incentives, and therefore payoffs, of the game. With this hypothesis, we generalize Sokota, Ho, and Wiedenbeck's results [45] to analyze games with a large, variable number of players. The new goal is to develop analyses that accommodate this uncertainty in the number of players, such as finding equilibria that are robust within the range of possible player counts. In this thesis, we define variable-player games, present a machine learning technique to analyze variable-player simulation-based games, and discuss how this technique will enable more meaningful predictions about behavior in the real-world game.

The remaining sections are organized as follows: section 2 will provide a more rigorous background in game theory and machine learning; section 3 will highlight several examples of learning game models from data in the literature and examine prior work that analyzes games with a variable number of players; section 4 will present our learned variable-player game-theoretic model as well as provide a technique to analyze variable-player games; section 5 will discuss our experimental results, comparing our model to existing work and showing the validation of our algorithm for analysis of variable-player games; finally, section 6 will summarize this thesis and detail avenues for future exploration.

2 Background

To fully understand and appreciate the novelty of the contributions of this thesis, it is essential to have a firm and thorough foundation in both game theory and machine learning. In this section, we will expand on the game theory and machine learning introductions from sections 1.1 and 1.2 by providing formal definitions of relevant terms and concepts, giving examples of more challenging concepts, and explaining how each concept relates to our work in modeling and analyzing variable-player games.

2.1 Game Theory

In this section we will discuss several classes of simultaneous-move games. In section 2.1.1 we will formally define a normal-form game as well as several other concepts which are essential when constructing and analyzing game-theoretic models. In section 2.1.2 we will describe symmetric games in which all players have the same strategy sets and utility functions, allowing us to use a more simplified notation to represent the game. In section 2.1.3 we will provide a more thorough motivation for and description of simulation-based games. Next, in section 2.1.4 we will define an action-graph game, a compact game representation of a normal-form game which is most useful when games exhibit player symmetries, and will preview how we use action-graph games in our experiments. Finally, in section 2.1.5 we will present several algorithms from the literature that are used to find approximate Nash equilibria.

2.1.1 Normal-Form Games

A *normal-form game* is a tuple $\Gamma_n = (P, S, u)$, where $P = \{1, \ldots, n\}$ is a set of indexed players, $S = S_1 \times \cdots \times S_n$ is the Cartesian product of each player i's strategy set S_i, and $u : S \mapsto \mathbb{R}^n$ is a utility function that maps *outcomes*, or combinations of strategies each player can play, to real-valued *utilities* or *payoffs*. Equivalently, $u = (u_1, \ldots, u_n)$, where $u_i : S \mapsto \mathbb{R}$.

The well-known game Rock-Paper-Scissors (RPS) can be represented as a normal-form game. There are two players, so

$$P = \{1, 2\}.$$

Each player has three possible actions or strategies: Rock (**R**), Paper (**P**), and Scissors (**S**). Thus

$$\begin{aligned} S_1 &= \{\mathbf{R}, \mathbf{P}, \mathbf{S}\} \\ S_2 &= \{\mathbf{R}, \mathbf{P}, \mathbf{S}\} \\ S = S_1 \times S_2 &= \{(\mathbf{R}, \mathbf{R}), (\mathbf{R}, \mathbf{P}), (\mathbf{R}, \mathbf{S}), \\ &\qquad (\mathbf{P}, \mathbf{R}), (\mathbf{P}, \mathbf{P}), (\mathbf{P}, \mathbf{S}), \\ &\qquad (\mathbf{S}, \mathbf{R}), (\mathbf{S}, \mathbf{P}), (\mathbf{S}, \mathbf{S})\}. \end{aligned}$$

We know that paper beats rock, scissors beats paper, rock beats scissors, and any outcome where both players play the same action results in a tie. For a given outcome, we can assign a utility of 1 to the winning player, -1 to the losing player and 0 to both players in the event of a tie. We can

awkwardly specify u as follows:

$$\begin{aligned} u_1 = \{&((\mathbf{R},\mathbf{R}),0),((\mathbf{R},\mathbf{P}),-1),((\mathbf{R},\mathbf{S}),1),\\ &((\mathbf{P},\mathbf{R}),1),((\mathbf{P},\mathbf{P}),0),((\mathbf{P},\mathbf{S}),-1),\\ &((\mathbf{S},\mathbf{R}),-1),((\mathbf{S},\mathbf{P}),1),((\mathbf{S},\mathbf{S}),0)\} \end{aligned}$$

$$\begin{aligned} u_2 = \{&((\mathbf{R},\mathbf{R}),0),((\mathbf{R},\mathbf{P}),1),((\mathbf{R},\mathbf{S}),-1),\\ &((\mathbf{P},\mathbf{R}),-1),((\mathbf{P},\mathbf{P}),0),((\mathbf{P},\mathbf{S}),1),\\ &((\mathbf{S},\mathbf{R}),1),((\mathbf{S},\mathbf{P}),-1),((\mathbf{S},\mathbf{S}),0)\} \end{aligned}$$

$$u = (u_1, u_2).$$

Note that while the winning, losing and tying payoffs are not dependent on player identity (i.e., the winning payoff is always 1 regardless of whether Player 1 or Player 2 wins, etc.), we still specify a utility function for each player. In section 2.1.2 we will discuss how this game can be rewritten as a symmetric game, where we care about the number of players playing each strategy as opposed to the identities or indices of the players playing each strategy.

While utility functions can be described formally as a set of (outcome, payoff) ordered pairs, this notation can be difficult to quickly interpret, as demonstrated by the Rock-Paper-Scissors example. Observe that in the Rock-Paper-Scissors utility function definitions, in the first row of outcomes Player 1 is always playing **R** and in the first column of outcomes Player 2 is always playing **R**. Overall, in a given row Player 1 is playing the same action across all columns and in a given column Player 2 is playing the same action across all rows. Using this organization we can translate the payoff function representation into a *payoff matrix*. In this payoff matrix, each of Player 1's actions is displayed on a new row and each of Player 2's actions is displayed on a new column. Further, each cell of the table corresponds to an outcome where Player 1 is playing the corresponding row strategy, Player 2 is playing the corresponding column strategy, and both players' payoffs for that outcome are stored within the cell. Note that Player 1's payoff is equal to the left value in the cell and Player 2's payoff is equal to the right value. Because payoff matrices are much easier to quickly understand, normal-form games are usually expressed using an n-dimensional payoff matrix, where n represents the number of players in the game.

For a game Γ_n, consider a player i with strategy set S_i. A *pure strategy* is any $s \in S_i$. In other words, S_i is the set of pure strategies available to player i. A *pure-strategy profile* $\vec{s} = (s_1, \ldots, s_n)$ is a vector that specifies which strategy each player has selected. The union of all pure-strategy profiles is equal to S. The outcomes in a game are all the possible combinations of actions the players can simultaneously play. While outcomes and pure-strategy profiles are effectively the same, we frequently use *pure-strategy profile* to describe what each player is playing whereas we use *outcome* to refer to a cell of a payoff matrix.

Figure 3a shows Rock-Paper-Scissors expressed using a normal-form payoff matrix. Suppose Player 1 plays rock (**R**) and Player 2 plays paper (**P**); equivalently, suppose $\vec{s} = (\mathbf{R}, \mathbf{P})$. The blue cell in Figure 3b highlights the payoffs associated with pure-strategy profile (**R**, **P**)—Player 1 receives a payoff of -1 and Player 2 receives a payoff of 1. This is consistent with the game rule that paper beats rock. The green cell in Figure 3b highlights the payoffs associated with pure-strategy profile (**P**, **R**). This might seem identical to the pure-strategy profile from before, but in fact this profile tells us that Player 1 is playing paper and Player 2 is playing scissors. The corresponding

		Player 2		
		R	**P**	**S**
Player 1	**R**	0, 0	-1, 1	1, -1
	P	1, -1	0, 0	-1, 1
	S	-1, 1	1, -1	0, 0

(a)

		Player 2		
		R	**P**	**S**
Player 1	**R**	0, 0	-1, 1	1, -1
	P	1, -1	0, 0	-1, 1
	S	-1, 1	1, -1	0, 0

(b)

Figure 3: (a) Rock-Paper-Scissors expressed as a normal-form game and (b) accompanying figure to three RPS examples describing how to interpret a payoff matrix.

outcome cell highlights that Player 1 receives a payoff of 1 and Player 2 receives a payoff of -1, which makes sense, again, because paper beats rock. Finally, consider the pure-strategy profile (**S**, **S**). The corresponding outcome cell is highlighted in purple, and tells us that both players receive 0 payoff, which makes sense because the outcome represents a tie.

Given uncertainty over opponents' actions, sometimes players might want to randomize over their possible actions. While this might not always make sense in a real-world context, it enables more avenues for analysis for the overall game from which meaning may still be extracted. A *mixed strategy* σ for player i is a probability distribution over the actions in S_i. Thus $\sum_{j=1}^{k}(\sigma_j) = 1$. Note that a pure strategy s is also a mixed strategy with probability 1 for strategy s and 0 for the remaining strategies in S_i. A mixed-strategy profile $\vec{\sigma} = (\sigma_1, \ldots, \sigma_n)$ specifies which (pure- or mixed-) strategy each player is playing.

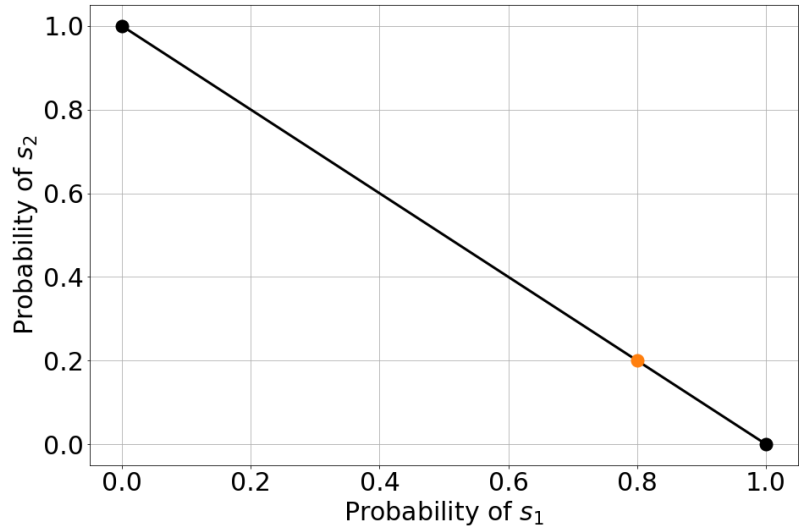
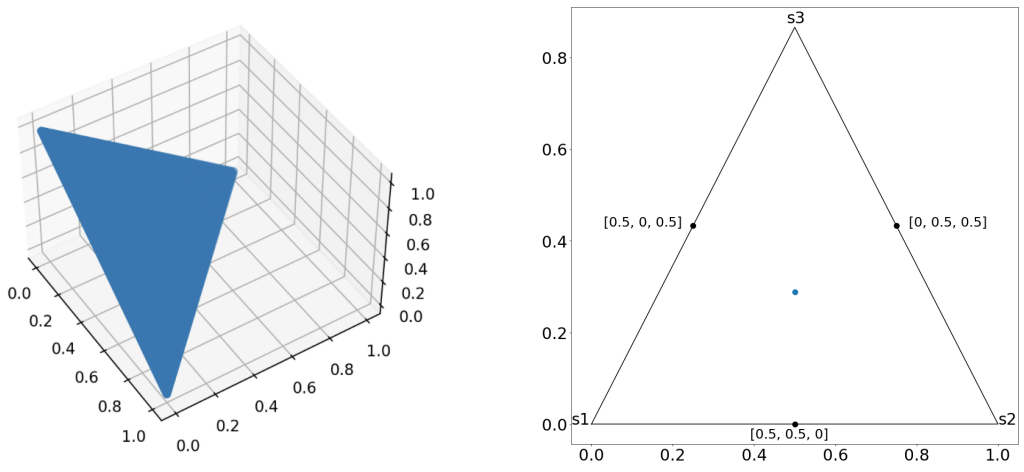


Figure 4: Line segment between (1,0) and (0, 1) describing the set of all mixed strategies containing two actions.

A mixed strategy σ with k actions is an element of $\mathbb{R}^k$. Further, the set of all mixed strategies containing k actions can be described by the $(k-1)$-simplex. When $k = 2$, the set of all mixed strategies is described by a line segment from (0, 1) to (1, 0), as shown in Figure 4. The two black



(a) (b)

Figure 5: Triangle describing the set of all mixed strategies containing three actions (a) in 3D and (b) projected into 2D.

points correspond to pure strategies, where the point at (1, 0) corresponds to pure strategy s_1 and the point at (0, 1) corresponds to pure-strategy s_2. The orange point corresponds to mixed strategy $\sigma = (0.8, 0.2)$, where the player plays s_1 with probability 0.8 and s_2 with probability 0.2.

When $k = 3$, as is the case in RPS, the set of mixed strategies is described by a triangle. Figure 5a shows the triangle in $\mathbb{R}^3$, and Figure 5b shows the triangle projected onto $\mathbb{R}^2$. Note that in Figure 5b, $s1$ corresponds to pure-strategy profile $\vec{s} = (1, 0, 0)$, $s2$ corresponds to $\vec{s} = (0, 1, 0)$, and $s3$ corresponds to $\vec{s} = (0, 0, 1)$. Additionally, the black points correspond to mixed strategies as labeled, and the blue point in the center corresponds to mixed strategy $\sigma = (1/3, 1/3, 1/3)$. A mixed-strategy profile $\vec{\sigma}$ is an element of $\mathbb{R}^m$, where $m = \sum_{i \in P} |S_i|$; equivalently, m is equal to the sum of the number of actions available to each player. Finally, the set of all mixed-strategy profiles can be described by a simplotope, which is the Cartesian product of the simplices for each player. These simplices and simplotopes will be used for visualizations in later sections.

Given that each player is playing a mixed strategy, how much payoff should player i expect to receive? This expected value of a player's utility function based on a mixed-strategy profile is called a player's *expected utility*. Recall that the components of the mixed-strategy profile describe the probability with which each player plays each strategy available to them. To calculate a player's expected utility, we can compute the probability of each outcome occurring according to $\vec{\sigma}$ and then compute the weighted average payoff based on the outcome probabilities. Formally, given a mixed-strategy profile $\vec{\sigma}$, player i's expected utility is defined as

$$u_i(\vec{\sigma}) = \sum_{s \in S} u_i(s) \prod_{j=1}^{n} \vec{\sigma}_j(s_j). \tag{1}$$

Consider the RPS game again and suppose $\vec{\sigma} = ((\frac{2}{3}, \frac{1}{3}, 0), (0, \frac{1}{2}, \frac{1}{2}))$. Then

$$\begin{aligned}
u_1(\vec{\sigma}) =& \frac{2}{3} \cdot 0 \cdot 0 + \frac{2}{3} \cdot \frac{1}{2} \cdot -1 + \frac{2}{3} \cdot \frac{1}{2} \cdot 1 + \\
& \frac{1}{3} \cdot 0 \cdot 1 + \frac{1}{3} \cdot \frac{1}{2} \cdot 0 + \frac{1}{3} \cdot \frac{1}{2} \cdot -1 + \\
& 0 \cdot 0 \cdot -1 + 0 \cdot \frac{1}{2} \cdot 1 + 0 \cdot \frac{1}{2} \cdot 0 \\
=& -\frac{1}{6} \\
u_2(\vec{\sigma}) =& \frac{2}{3} \cdot 0 \cdot 0 + \frac{2}{3} \cdot \frac{1}{2} \cdot 1 + \frac{2}{3} \cdot \frac{1}{2} \cdot -1 + \\
& \frac{1}{3} \cdot 0 \cdot -1 + \frac{1}{3} \cdot \frac{1}{2} \cdot 0 + \frac{1}{3} \cdot \frac{1}{2} \cdot 1 + \\
& 0 \cdot 0 \cdot 1 + 0 \cdot \frac{1}{2} \cdot -1 + 0 \cdot \frac{1}{2} \cdot 0 \\
=& \frac{1}{6}.
\end{aligned}$$

Therefore, when the players play according to mixed-strategy profile $\vec{\sigma} = ((\frac{2}{3}, \frac{1}{3}, 0), (0, \frac{1}{2}, \frac{1}{2}))$, Player 1 can expect to receive a payoff of $-\frac{1}{6}$ and Player 2 can expect to receive a payoff of $\frac{1}{6}$.

To motivate typical game-theoretic analysis, consider the following real-world game. Madelyn and Will are young siblings on a long car ride and they must decide which movie to watch on a portable television.[1] Will's favorite movie is *Thomas & Friends* (**T**), and Madelyn's favorite movie is *Barbie Swan Lake* (**B**). Each sibling must tell their father which movie they would like to watch. If the siblings pick different movies, their father makes them watch *Spirit*, a movie they strongly dislike. For these outcomes, both siblings receive a payoff of 0. If the siblings pick the same movie, the sibling for whom the movie is their favorite receives a payoff of 2 and the other sibling receives a payoff of 1 because they still prefer the movie to *Spirit*. Figure 6 shows this game expressed using a normal-form payoff matrix, where Madelyn is the row player and Will is the column player. Note that this is equivalent to a common game in introductory game theory courses with the name "Battle of the Sexes," but we prefer this "Battle of the Siblings" version for several obvious reasons.

		Will	
		B	**T**
Madelyn	**B**	2, 1	0, 0
	T	0, 0	1, 2

Figure 6: Will and Madelyn's "Battle of the Siblings" represented as a normal-form game.

In the "Battle of the Siblings" example, a primary objective is to make a prediction about Madelyn and Will's behavior based on the game-theoretic model. We can identify outcomes of interest in the game-theoretic model using various *solution concepts*, which are criteria which can

[1] A significant treat, as long car-ride movies did not require TV tickets.

be applied to distinguish relevant outcomes from irrelevant outcomes. We first present a naive initial solution concept to demonstrate the difficulty of finding relevant outcomes. Recall that players are expected utility maximizers. A naive initial approach might be to select the mixed-strategy profile $\vec{\sigma}$ that maximizes the expected utility for the greatest number of agents as the most likely outcome. Let Player p be one of the players for whom the mixed-strategy profile does *not* maximize their expected utility. Given that everyone else plays according to $\vec{\sigma}$, Player p can instead play the mixed strategy σ_i' that maximizes their expected utility. Thus this solution concept is not particularly helpful because not all players have an incentive to play according to the identified mixed-strategy profile $\vec{\sigma}$.

This example highlights the notion of a best-response strategy. Define $\vec{\sigma}_{-i}$ to be the $(n-1)$-tuple describing the mixed strategies of all players excluding i:

$$\vec{\sigma}_{-i} = (\sigma_1, \ldots, \sigma_{i-1}, \sigma_{i+1}, \ldots, \sigma_n)$$

Then $\vec{\sigma} = (\sigma_i, \vec{\sigma}_{-i})$. Player i's *best response* to $\vec{\sigma}_{-i}$ is a pure-strategy s_i' such that $u_i(s_i', \vec{\sigma}_{-i}) \geq u_i(s_i, \vec{\sigma}_{-i})$ for all $s_i \in S$. Note that by linearity of expectation, any mixed-strategy with non-zero probabilities only assigned to strategies that are best responses is also a best response to σ_{-i}. Equivalently, a *best response* for Player i is any strategy that maximizes their expected utility, holding opponents' actions constant. In the example from earlier, Player p's best response was to play a mixed-strategy other than $\sigma_i \in \vec{\sigma}$, which is why the solution concept was not an effective one.

One of the most important solution concepts in game theory is Nash equilibrium. A *Nash equilibrium* is a pure- or mixed-strategy profile such that every player is best responding. Other common solution concepts in game theory include $\epsilon-$Nash equilibrium, correlated equilibrium, coarse-correlated equilibrium, and pareto optimality. This thesis will exclusively focus on computing Nash equilibria and ϵ-Nash equilibria.

To find pure-strategy Nash equilibria in the "Battle of the Siblings" game, we want to determine the best-response strategy for each player and opponent strategy. If there exists a pair of strategies, one for Will and one for Madelyn, such that each strategy is a best response to the opponent playing the other strategy, then that pure-strategy profile is a Nash equilibrium. In Figure 7a, we fix Will's strategy as **B** as demonstrated by the gray background. The green highlighted values show the possible payoffs Madelyn could receive—if Madelyn plays **B**, she receives a payoff of 2 and if she plays **T**, she receives a payoff of 0. Since $2 > 0$, Madelyn's best response strategy to Will playing **B** is also to play **B**. As demonstrated by Figure 7b, when we fix Will's strategy as **T** (shown in gray), Madelyn's best response strategy is also **T** because $1 > 0$ (shown in green). In Figure 7c, we fix Madelyn's strategy as **B** as demonstrated by the gray cell background. Will's best response strategy to Madelyn playing **B** is also to play **B** because $1 > 0$. As shown by Figure 7d, when we fix Madelyn's strategy as **T**, Will's best response strategy is also **T** because $2 > 0$. Thus pure-strategy profile $\vec{s} = (\mathbf{B}, \mathbf{B})$ is a Nash equilibrium because both Madelyn and Will are best responding to their beliefs about the other's actions. Similarly, pure-strategy profile $\vec{s} = (\mathbf{T}, \mathbf{T})$ is a Nash equilibrium because both are best responding.

We can solve for a mixed-strategy Nash equilibrium algebraically in the "Battle of the Siblings" game through a system of equations where each player randomizes to make their sibling indifferent between their two strategies. Let p denote the probability with which Madelyn chooses **B** and q denote the probability with which Will chooses **B**. In order to make Will indifferent between

	B	T
B	2, 1	0, 0
T	0, 0	1, 2

(a)

	B	T
B	2, 1	0, 0
T	0, 0	1, 2

(b)

	B	T
B	2, 1	0, 0
T	0, 0	1, 2

(c)

	B	T
B	2, 1	0, 0
T	0, 0	1, 2

(d)

Figure 7: Using best responses to find pure-strategy Nash equilibria in the Battle of the Siblings game.

the two movies, Madelyn wants to randomize such that

$$\begin{aligned} u_W((\mathbf{B}, (p, 1-p))) &= u_W((\mathbf{T}, (p, 1-p))) \\ 1 \cdot p + 0 \cdot (1-p) &= 0 \cdot p + 2 \cdot (1-p) \\ p &= 2 - 2p \\ p &= \frac{2}{3}. \end{aligned}$$

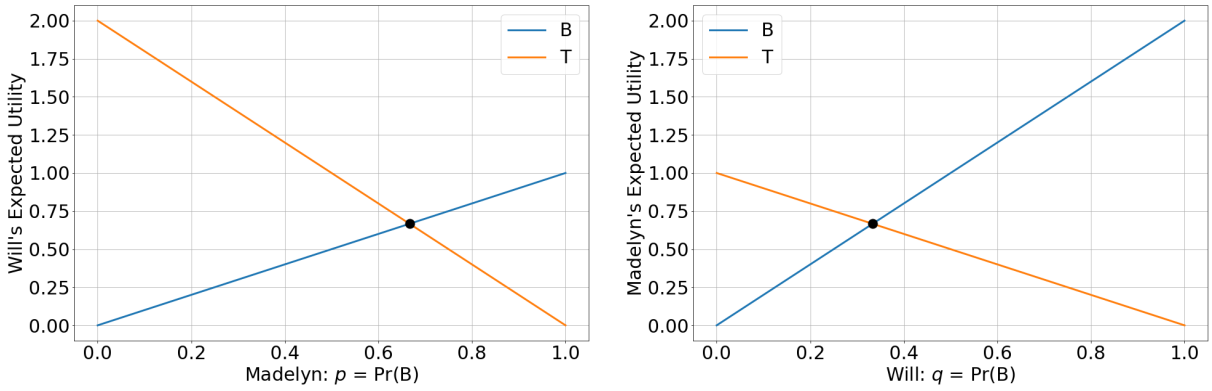
Therefore, Madelyn should play mixed-strategy $\sigma = (p, 1-p) = (\frac{2}{3}, \frac{1}{3})$ in order to make Will indifferent between the two movies. In order to make Madelyn indifferent between the two movies, Will wants to randomize such that

$$\begin{aligned} u_M((\mathbf{B}, (q, 1-q))) &= u_M((\mathbf{T}, (q, 1-q))) \\ 2 \cdot q + 0 \cdot (1-q) &= 0 \cdot q + 1 \cdot (1-q) \\ 2 \cdot q &= 1 - q \\ q &= \frac{1}{3}. \end{aligned}$$

Therefore, Will should play mixed-strategy $\sigma = (q, 1-q) = (\frac{1}{3}, \frac{2}{3})$ in order to make Madelyn indifferent between the two movies. Since for both players we solved for mixed-strategies that make their sibling indifferent, they are both best responding to each other. Thus $\vec{\sigma} = \left((\frac{2}{3}, \frac{1}{3}), (\frac{1}{3}, \frac{2}{3})\right)$ is a Nash equilibrium. Figure 8 shows solving for mixed-strategy equilibrium probabilities graphically. In general it is much more challenging to compute mixed-strategy Nash equilibria in a game. We will discuss Nash-finding algorithms in section 2.1.5.

Why is the Nash equilibrium such an influential solution concept in game theory? Nash's Theorem [34], one of the most important theorems in game theory, characterizes a guarantee for this solution concept for all non-cooperative games:

Theorem 2. *Every finite game with two or more players contains at least one Nash equilibrium.*



(a) (b)

Figure 8: (a) Solving for the probability p with which Madelyn plays action **B** such that Will is indifferent between the two movies and (b) Solving for the probability q with which Will plays action **B** such that Madelyn is indifferent between the two movies.

In this theorem *finite* means that the sets of players and strategies are finite. We have omitted the proof of this theorem, but the theorem can be proven using Brouwer's fixed-point theorem.

Suppose a mixed-strategy profile is not a Nash equilibrium because at least one player is not best responding. We can quantify how "close" the profile is to being an exact Nash equilibrium using *regret*. Given mixed-strategy profile $\vec{\sigma}$, the regret for player i is defined as

$$\epsilon_i(\vec{\sigma}) = \max_{s \in S_i} u_i(s, \vec{\sigma}_{-i}) - u_i(\vec{\sigma}).$$

The overall regret for a profile $\vec{\sigma}$ is defined as

$$\epsilon(\vec{\sigma}) = \max_{i \in P} \epsilon_i(\vec{\sigma}).$$

For a profile that is an exact Nash equilibrium, the regret is 0 for all players and therefore the overall regret is also 0.

There are many scenarios in which it is difficult (or even impossible[2]) to compute an exact Nash equilibrium. Often it is more realistic to compute approximate Nash equilibria instead. An *approximate Nash equilibrium* (also called an *ε-Nash equilibrium*) is a mixed-strategy profile $\vec{\sigma}$ such that $\text{regret}(\vec{\sigma}) \leq \varepsilon$ for some small value of ε. We will discuss motivations for finding approximate equilibria and will present several algorithms to compute approximate Nash equilibria in section 2.1.5.

An important intermediate quantity for this thesis is a deviation payoff. A *deviation payoff* is the expected payoff a player would receive by deviating or changing strategies, given the mixed strategies everyone else is playing. Given a mixed-strategy profile $\vec{\sigma}$, the deviation payoff for player i and strategy $s \in S_i$ is formally defined by

$$\text{devPay}_i(s, \vec{\sigma}) = u_i(s, \vec{\sigma}_{-i}),$$

where $u_i(s, \vec{\sigma}_{-i})$ is player i's expected utility when all opponents play according to $\vec{\sigma}_{-i}$ and player i plays strategy $s \in S_i$. Additionally, the *deviation payoff function* for a player i and mixed-strategy

[2]Due to representational limitations in floating point precision.

profile $\vec{\sigma}$ is defined as

$$\text{devPay}(\vec{\sigma}) = \Big(\text{devPay}_1(s_1, \vec{\sigma}), \text{devPay}_1(s_2, \vec{\sigma}), \ldots, \text{devPay}_n(s_{|S_n|}, \vec{\sigma})\Big).$$

One benefit of deviation payoffs is that we can characterize expected utilities, regret, Nash equilibria and approximate Nash equilibria in the context of deviation payoffs. Given a mixed-strategy profile $\vec{\sigma}$, player i's expected utility can also be defined as

$$u_i(\vec{\sigma}) = \sigma_i \cdot \text{devPay}_i(\vec{\sigma}).$$

The regret of a profile $\vec{\sigma}$ can also be defined as

$$\epsilon(\vec{\sigma}) = \max_{i \in P} \text{devPay}_i(\vec{\sigma}) - u_i(\vec{\sigma}).$$

Conceptually, the regret of a profile is the maximum payoff amount any player can gain by deviating to any other strategy. Thus a Nash equilibrium is a set of strategies such that no player can gain by deviating to any other strategy. Similarly, an ε-Nash equilibrium is a set of strategies such that no player can gain more than ε by deviating to any other strategy. In this thesis, we aim to compute ε-Nash equilibria and evaluate success with regret.

2.1.2 Symmetric Games

Let $\Gamma_n = (P, S, u)$ be a normal-form game with n players, where

$$\begin{aligned} P &= \{1, \ldots, n\} \\ S &= S_1 \times \cdots \times S_n \\ u &= (u_1, \ldots, u_n). \end{aligned}$$

Consider players $i, j \in P$ where $i < j$. Define $\Gamma'_n = (P', S', u')$ to be a normal-form game with n players, where

$$\begin{aligned} P' &= \{1, \ldots, i-1, j, i+1, \ldots, j-1, i, j+1, \ldots, n\} \\ S' &= S_1 \times \cdots \times S_{i-1} \times S_j \times S_{i+1} \times \cdots \times S_{j-1} \times S_i \times S_{j+1} \times \cdots \times S_n \\ u' &= (u_1, \ldots, u_{i-1}, u_j, u_{i+1}, \ldots, u_{j-1}, u_i, u_{j+1}, \ldots, u_n). \end{aligned}$$

Players i and j are *symmetric* in Γ_n if $\Gamma_n = \Gamma'$. Said another way, Players i and j are symmetric in Γ_n if they can be permuted and the same game results. A game Γ_n is *symmetric* if every player is symmetric to every other player in the game. A game Γ_n is *role-symmetric* if the set of players P can be partitioned into roles such that every player in a given role is symmetric to every other player with that role and at least one role has more than one symmetric player.

Because all players can be permuted in a symmetric game without the game changing, the identities or indices of the players ultimately do not matter. Thus, what determines each player's payoff is not *who* is playing which strategy, but rather *how many* players are playing each strategy. A *player configuration* $\vec{c}$ is a k-vector that specifies how many players are playing each strategy in a symmetric game, where k is the number of strategies in the game. Also, an *opponent profile* $\vec{s}$

is a k-vector that specifies how many opponents are playing each strategy in a symmetric game, where $\sum_{i=1}^{k} \vec{s}_i = n - 1$.

We can redefine a *symmetric game* as $\tilde{\Gamma}_n = (n, S, u)$, where n is the number of players, $S = \{s_1, \ldots, s_k\}$ is the set of strategies available to all players and u is the utility function which maps length-k player configurations to length-k payoff vectors. Each corresponding dimension in the player configurations and payoff vectors corresponds to a particular strategy. Thus $u_j(\vec{c})$ describes the payoff the players playing strategy s_j receive when all players play according to $\vec{c}$. Let C denote the set of all possible player configurations:

$$C = \left\{ \vec{c} \in \mathbb{Z}^k : \vec{c}_j \geq 0, \sum_{j=1}^{k} \vec{c}_j = n \right\}.$$

Then the symmetric utility function is defined by $u : C \mapsto \mathbb{R}^k$. Finally, a *symmetric profile* is a profile in which all players play the same strategy, either a mixed strategy or a pure strategy.

Based on experience playing the game, it makes sense that Rock-Paper-Scissors is symmetric because the winning, losing, and tying payoffs are not dependent on player identity. Using the simplified symmetric game definition, we can express Rock-Paper-Scissors as a symmetric game:

$$\begin{aligned}
&n = 2 \\
&S = \{\mathbf{R}, \mathbf{P}, \mathbf{S}\} \\
&C = \{(2,0,0), (0,2,0), (0,0,2), (1,1,0), (1,0,1), (0,1,1)\} \\
&u((2,0,0)) = (0,0,0) \qquad u((1,1,0)) = (-1,1,0) \\
&u((0,2,0)) = (0,0,0) \qquad u((1,0,1)) = (1,0,-1) \\
&u((0,0,2)) = (0,0,0) \qquad u((0,1,1)) = (0,-1,1).
\end{aligned}$$

One benefit of symmetric games is the simplified definition reduces the size of the payoff matrix and therefore the amount of memory required to store the matrix. In general for a symmetric game with n players and k actions, $|C| = \binom{n+k-1}{n} = \frac{(n+k-1)!}{n!\cdot(k-1)!}$ since we are counting the number of ways we can configure n players given $n + k - 1$ positions. If we fix k, it would require $O(k^n)$ outcomes to be stored in normal form compared to $O(n^{k-1})$ outcomes in symmetric form. With $n = 100$ and $k = 5$, that translates to $5^{100} \approx 10^{69}$ outcomes in normal form compared to $100^5 = 10^{10}$. Another benefit of symmetric games is the simplified definition reduces the amount of computation required to calculate expected utilities, deviation payoffs, regret, etc. Given a symmetric mixed-strategy profile, all players are mixing according to the same probabilities which means all computations involve k-dimensional space as opposed to $n \cdot k$.

2.1.3 Simulation-Based Games

For many of the interactions game theorists might be interested in modeling, historical data is insufficient to describe the interaction. Additionally, analytical or mathematical models require greater precision or a deeper understanding about the incentives in the interaction than is feasible. Thus a more realistic alternative is to combine agent-based modeling and multi-agent simulation to construct a game-theoretic model. Such games are called *simulation-based games* [58] (also known as empirical games [57] or black-box games [65, 31]). We can then apply any solution

concept to analyze the game. Often this analysis entails finding approximate Nash equilibria in the game and then investigating what happens in equilibrium to answer our initial question about the interaction. Because these game-theoretic models are constructed using simulation data as opposed to real-world data, the goal is less about precisely predicting future behavior, and instead about characterizing overarching trends about the interaction. Applications of simulation-based game theory include trading agent competitions (TACS) [16, 59, 21, 22], credit networks [6], high-frequency trading [54, 53, 55], and cybersecurity [60].

While simulation-based game theory can be used to construct other classes of game-theoretic models, including Bayesian games [31] or extensive-form games [14], in this thesis we focus exclusively on normal-form simulation-based games. Therefore we can think of a simulation-based game as a game where no payoff matrix is known in advance but can be filled in through a series of simulations to construct a game-theoretic model. A high-level description of the simulation-based game-theoretic model construction process is described in Figure 9. First, we define a set of players and a set of strategies for each player, and specify a set of rules by which the agents interact with each other. Next, we simulate various combinations of pure-strategy profiles, and finally, we record the associated noisy payoff estimates in the corresponding cell in the payoff matrix. We refer the reader to *Empirical Game-Theoretic Analysis* (EGTA) [57] and *Policy-Space Response Oracles* (PSRO) [27] for more details about popular simulation-based game theory frameworks. Note that in practice we do not actually fill out the entire normal-form payoff matrix but rather use it as a black box that we can query to get noisy payoff estimates for a given pure-strategy profile. Also, queries to the simulator are expensive, so we must develop analyses that minimize the number of simulator queries.

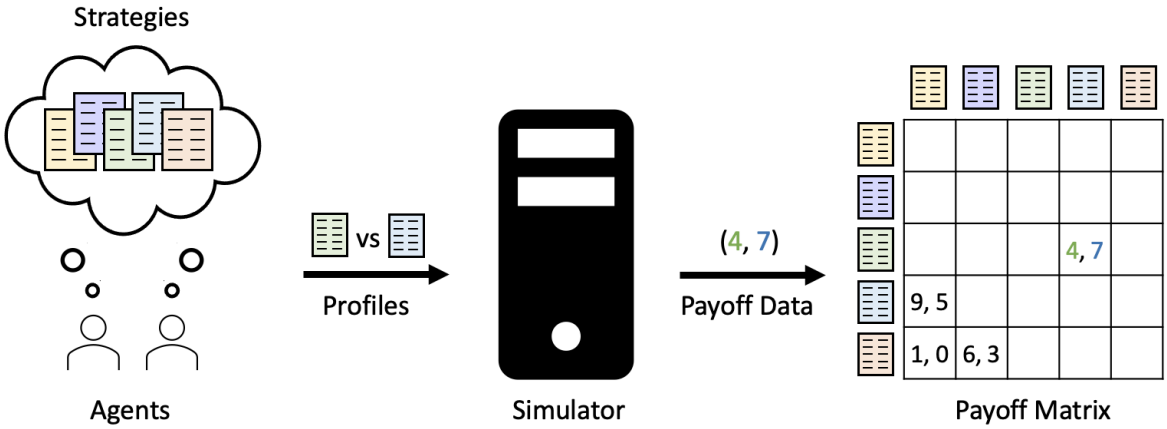


Figure 9: High-level description of simulation-based game-theoretic model construction: we define agents and their strategies, simulate each possible profile, and record payoff estimates in a payoff matrix.

Many of the real-world interactions we want to model with simulation-based game theory have a large, uncertain number of participants. The goal of this thesis is to develop analyses that accommodate this uncertainty in the number of players. While there are other classes of games in which the exact number of players is unknown, our initial focus is analyzing simulation-based games with a variable number of players as it is more straightforward to specify a variable

number of players in the simulation environment than in other settings. However, we hope our techniques will be applicable to other classes of games. For the purposes of this thesis, we assume we have access to some simulator and can query the simulator for noisy payoff estimates for a given pure-strategy profile. This simulator is used to help build our labeled training set for our neural network model.

2.1.4 Action-Graph Games

In this section we introduce action-graph games and action-graph games with additive function nodes, which we use as a proxy for simulator data in our experiments in this thesis. The entire section provides a short summary of work by [20].

An *action-graph game* (AGG) is a compact representation of a normal-form game where actions are represented as vertices and edges encode payoff dependence among actions [20]. Formally, an *action-graph game* is a tuple (P, S, G, u) where

- $P = \{1, \ldots, n\}$ is the finite set of players.
- $S = \prod_{i \in P} S_i$ is the Cartesian product of each player i's strategy set S_i.
- $G = (V, E)$ is a graph where each $v \in V$ corresponds to a distinct action in S and an edge (v_1, v_2) indicates that the number of players playing strategy 1 affects the payoff that the players playing strategy 2 receive. Note that the converse is not necessarily true unless $(v_2, v_1) \in E$.
- Let $v \in V$. Define $N(v) = \{u \in V : (u, v) \in E\}$ to be the set of actions in the neighborhood of action v which affect the payoff that players playing v receive.
- Further, let $c(v)$ denote a particular configuration of the neighborhood. That is, for each $u \in N(v)$, $c(v)$ denotes how many players are playing vertex u. Then define $C(v) = \bigcup c(v)$ to be the set of all possible player configurations of the neighborhood of action v.
- $u = (u_{v_1}, u_{v_2}, \ldots, u_{v_{|V|}})$, where $u_{v_j} : C(v_j) \mapsto \mathbb{R}$ for $v_j \in V$. Unlike in normal-form games where we define a utility function for each *player*, in action-graph games we are defining a utility function for each distinct *action*.

Consider the following real-world-inspired example. MumFest is an annual festival in historic downtown New Bern, NC that has a variety of vendors and attractions, including homemade treats and crafts, amusement park rides, street performers, music performances, and pretty much any type of fried food imaginable. For the festival the city closes several streets, including E Front St, Broad St, and Pollock St, in order to create space for as many vendors as possible. Our task is to construct a compact representation of a game which describes booth owners' incentives for choosing a particular booth location.

In this small example,[3] there are three different types of booths: booths which serve Pepsi-Cola (C), booths which have some sort of amusement park ride (R), and booths which both serve Pepsi-Cola and have an amusement park ride (B). Also, there are four possible booth locations along E Front St: L_1, located at the intersection of Broad St and E Front St, L_2, located at the

[3]Inspired by the Ice Cream Vendor game in [20].

intersection of Pollock St and E Front St, and finally, L_3 and L_4, which are located on E Front St bordering Union Point Park. Due to space constraints, vendors who both sell Pepsi-Cola and have rides are restricted to only selecting a Union Point Park booth (L_3 or L_4). A booth's payoff is negatively impacted by the number of booths at neighboring locations that sell the same product (C) or service (R). Additionally, Pepsi booths receive additional funding from Pepsi Co. the closer the booth is to the birthplace of Pepsi-Cola on Pollock St; thus Pepsi booths at L_2 receive the most additional funding, then those at L_1 and L_3 and the least amount of funding at L_4. Figure 10 shows a graphical representation of the four possible booth locations.

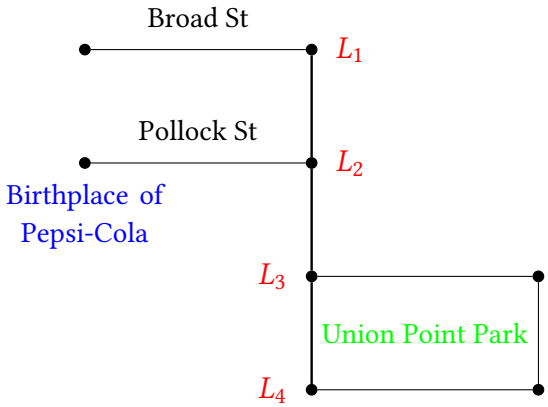


Figure 10: A graphical representation of the eastern section of downtown New Bern, NC showing the four possible MumFest booth locations.

We can represent this game as an action-graph game as shown in Figure 11a. Each of the four C_i nodes illustrate where Pepsi-Cola booths can be located, as shown by the S_C dotted box in 11b; similarly, the S_R dotted box in 11b shows the possible locations for amusement park ride booths. Finally, the S_B dotted box demonstrates that booths which both sell Pepsi-Cola and offer an amusement park ride can only be located at either location 3 or location 4. The edges between the nodes encode geographical neighboring relationships.

Like the MumFest AGG shows, action-graph games are constructed independent of the number of players in the game, which is one way they are able to represent games with a large number of symmetric or semi-symmetric[4] players compactly. Action-graph games are universal because they can represent any normal-form game, although their advantage comes from compactness in special cases. AGGs are not a particularly useful representation for simulation-based games because the underlying structure required to represent as an AGG is not known in advance and it would be inefficient to fill in the entire payoff matrix first and then represent the game as an AGG. However, AGGs, specifically BAGG-FNAs (described on the next page), are a powerful game representation because they can be structured and randomly generated in a way that is similar to the underlying structure of a simulation-based game. Therefore we use random BAGG-FNAs as proxies for simulator data.

[4]Semi-symmetric meaning that two players have some actions and corresponding utilities that overlap

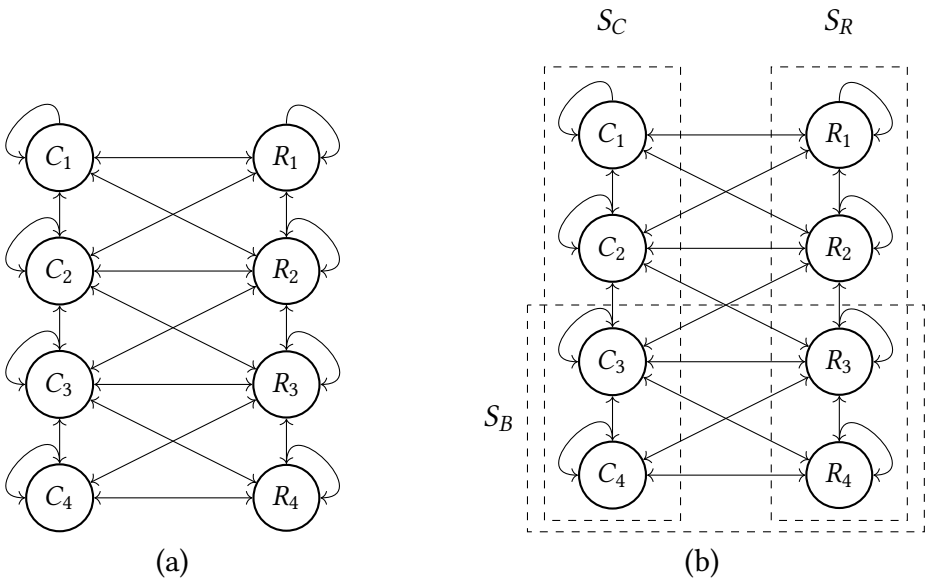


Figure 11: (a) Mumfest vendor game represented as an action-graph game and (b) Strategy sets for each role highlighted in Mumfest AGG.

A *bipartite action-graph game with additive function nodes* (BAGG-FNA) is a special type of action-graph game and is defined by tuple (P, S, F, G, W, u, f) where

- $P = \{1, \ldots, n\}$ is the finite set of players.
- $S = \prod_{i \in P} S_i$ is the Cartesian product of each player i's strategy set S_i. Define $\mathcal{S} = \bigcup_{i \in P} S_i$ to be the set of distinct strategies in S.
- F is the finite set of function nodes.
- $G = (V, E)$ is a graph where $V = \mathcal{S} \cup F$, $\mathcal{S}$ and F are independent sets and $E = \{(u, v) : u \in \mathcal{S}, v \in F \text{ or } u \in F, v \in \mathcal{S}$.
- $W = \{W_{(u,v)} : u \in F \text{ and } v \in \mathcal{S}\}$ is the set of function-strategy weights.
- Let $v \in V$. Define $N(v) = \{u \in V : (u, v) \in E\}$ to be the set of nodes in the neighborhood of node v. Note that if $v \in \mathcal{S}$ then $N(v) \subseteq F$ and if $v \in F$ then $N(v) \subseteq \mathcal{S}$.
- Further, let $c(v)$ denote a particular configuration of the neighborhood of v. That is, for each $u \in N(v)$, $c(v)$ denotes how many players are playing node u. Then define $C(v) = \bigcup c(v)$ to be the set of all possible player configurations of the neighborhood of node v.
- $f = (f_1, f_2, \ldots, f_{|F|})$, where $f_v : C(v) \mapsto \mathbb{R}$ for $v \in F$.
- $u = (u_{s_1}, u_{s_2}, \ldots, u_{s_{|\mathcal{S}|}})$, where $u_{s_j} : C(s_j) \mapsto \mathbb{R}$ for $s_j \in \mathcal{S}$. More specifically, $u_{s_j} = \sum_{v \in N(s_j)} W_{(v,s_j)} \cdot f_v(C(s_j))$. In other words, u_{s_j} is equal to the weighted sum of the function outputs for the function nodes in the neighborhood of strategy node $s_j \in \mathcal{S}$.

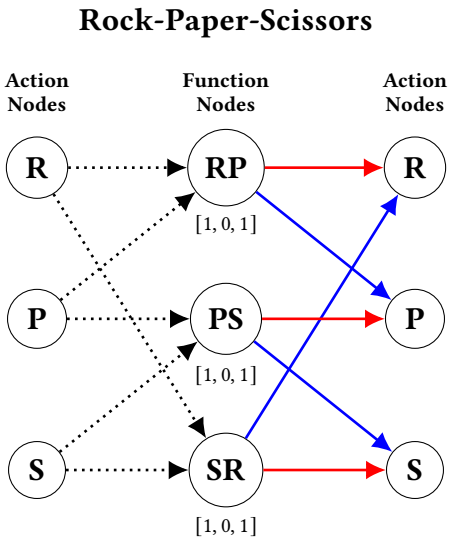


Figure 12: RPS represented as a bipartite AGG with additive function nodes.

Figure 12 shows Rock-Paper-Scissors encoded as a bipartite AGG-FNA. There are three action nodes corresponding to actions **R**, **P** and **S**. There are three function nodes corresponding to "counters" for the number of players playing (**R** or **P**), (**P** or **S**), and (**S** or **R**). The function tables are shown below each function node; all function tables happen to be the same, but this is not usually the case. Given function table [1, 0, 1] for a given function node v, the function node outputs 1 when 0 players are playing strategies in the neighborhood of v, 0 when exactly 1 player is playing a strategy in the neighborhood of v, and 1 when exactly 2 players are playing a strategy in the neighborhood of v. The blue edges mean that the corresponding edge weight is 1 and the red edges mean that the corresponding edge weight is -1. We have represented the bipartite graph in this way with action nodes showing up twice so it is easier to see the action nodes passing as input to the function nodes the number of players playing that action and then the function nodes sending the output to the corresponding action nodes.

Suppose one player plays **R** and the other player plays **P**. Then

$$\begin{aligned}\text{count}(\mathbf{R}\text{ or }\mathbf{P}) &= 1+1=2 \mapsto 1,\\ \text{count}(\mathbf{P}\text{ or }\mathbf{S}) &= 1+0=1 \mapsto 0,\\ \text{count}(\mathbf{S}\text{ or }\mathbf{R}) &= 0+1=1 \mapsto 0.\end{aligned}$$

So,

$$\begin{aligned}u(\mathbf{R}) &= -1\cdot 1 + 1\cdot 0 = -1,\\ u(\mathbf{P}) &= 1\cdot 1 + -1\cdot 0 = 1,\\ u(\mathbf{S}) &= 1\cdot 0 + -1\cdot 0 = 0.\end{aligned}$$

Now suppose both players play **R**. Then

$$\begin{aligned}\text{count}(\mathbf{R} \text{ or } \mathbf{P}) &= 2 + 0 = 2 \mapsto 1,\\ \text{count}(\mathbf{P} \text{ or } \mathbf{S}) &= 0 + 0 = 0 \mapsto 1,\\ \text{count}(\mathbf{S} \text{ or } \mathbf{R}) &= 0 + 2 = 2 \mapsto 1.\end{aligned}$$

So,

$$\begin{aligned}u(\mathbf{R}) &= -1 \cdot 1 + 1 \cdot 1 = 0,\\ u(\mathbf{P}) &= 1 \cdot 1 + -1 \cdot 1 = 0,\\ u(\mathbf{S}) &= 1 \cdot 1 + -1 \cdot 1 = 0.\end{aligned}$$

We can repeat this process for all possible player configurations to confirm that the game has been properly encoded as a BAGG-FNA.

In one sense generating a random game amounts to generating a random payoff matrix. However, a payoff matrix filled with random numbers is not useful because the interaction we are trying to model is not random. The additive function nodes in a BAGG-FNA allow us to generate random payoff functions that are complex but still have some underlying structure, which is much more realistic. As a result, in this thesis we use random BAGG-FNAs as proxies for simulator data in our experiments.

2.1.5 Approximate Nash Equilibrium Computation

The computational complexity of finding a Nash equilibrium in games with various settings (i.e., 2 vs $\geq$ 2 players, zero sum vs general sum, etc.) is a heavily studied problem within theoretical computer science. Computing a Nash equilibrium in a finite, n-player general-sum game is PPAD-complete [7]. This means that there is no polynomial-time algorithm we can use to find Nash equilibria in a game. In fact, the only setting in which there exists a polynomial-time algorithm to compute a Nash equilibrium is 2-player zero-sum. Even in a 2-player general-sum game it can take exponential time to compute a Nash equilibrium. As a result, it is frequently more realistic to compute an approximate Nash equilibrium as opposed to an exact Nash equilibrium.

Another factor which prevents us from computing an exact Nash equilibrium in an $n-$player general-sum game is computational limitations in floating-point representation. *Machine precision* describes the largest gap in floating-point representation, which is the gap between 1 and the next largest number. It provides a bound on the relative roundoff error due to the machine. For double-precision floating-point format, machine precision is equal to $2^{-52} \approx 10^{-16}$. This means that we can trust the computer to accurately represent a given number up to 16 decimal places. As a result, we are unable to compute an exact Nash equilibrium in a game if the mixed-strategy probabilities are irrational or require more than 16 decimal places of precision.

How can we compute an approximate Nash equilibrium in an n-player, general-sum game? We can classify approximate Nash-finding algorithms as either *complete*, *special case*, or *incomplete*. Lemke-Howson [28] and Govindan-Wilson [15] (also known as the global Newton method) are examples of complete Nash-finding algorithms—they are exponential but are guaranteed to converge to an approximate Nash equilibrium. Fictitious play [4] is an example of a special-case algorithm because it is guaranteed to converge in the 2-player zero-sum setting, but is incomplete

in the n-player general-sum setting. Incomplete Nash-finding algorithms are algorithms which are not guaranteed to converge to an approximate Nash equilibrium, but when they do they tend to converge quickly. Replicator dynamics [48] is an example of an incomplete algorithm for the n-player general-sum setting. When applied to games with a large number of players, these algorithms do not converge quickly (if they do converge) because both methods involve summing over the entire payoff matrix multiple times, which is computationally expensive in games with a large number of players. However, the benefit of the algorithms discussed—fictitious play, global Newton method and replicator dynamics—is that all algorithms can be rewritten to use the deviation payoff function (and its derivative in the global Newton algorithm). Sokota et al. [45] replaced this ground truth deviation payoff function with a learned deviation payoff function in order to compute approximate Nash equilibria relatively quicker.

In this thesis, we use replicator dynamics as our Nash-finding algorithm. Replicator dynamics [48] was initially used to study how competition affects population proportions for different species from an evolutionary perspective. To understand how this algorithm can be applied to game theory, we can think about the competing species as actions and the mixed-strategies as the population proportions.

Algorithm 1 provides pseudocode for the replicator dynamics algorithm using deviation payoffs. First, we compute the minimum payoff for each player i. For each dimension j of the mixed-strategy profile $\vec{\sigma}$ (and therefore the corresponding player i), we update $\vec{\sigma}_j$ by multiplying $\vec{\sigma}_j$ by the difference between the deviation payoff player i would receive by deviating to s_j and the minimum payoff for player i. In other words, we multiply $\vec{\sigma}_j$ by the maximum amount player i could *gain* by deviating to pure-strategy s_j. The larger the difference, the more player i can gain from playing s_j and therefore the more player i wants to mix over s_j. After we have repeated this process for all $s_j \in S$, we normalize the updated $\vec{\sigma}$ to ensure that the sum of each mixed strategy is equal to 1 and all dimensions have probability between 0 and 1. The updated $\vec{\sigma}$ now places a higher probability on strategies where the corresponding player can gain more payoff and a lower probability on strategies where the corresponding player can lose more payoff. We repeat this process for a pre-specified number of iterations. Then, if the regret of the final $\vec{\sigma}$ is less than or equal to some ε, we say $\vec{\sigma}$ is an ε-Nash equilibrium.

Algorithm 1 Computing approximate Nash equilibria using replicator dynamics.

```
replicatorDynamics(Γ_n, σ⃗, ϵ, numIters):
    offset ← [min(Γ_n, i) for each player i]
    repeat
        σ⃗ ← σ⃗ • (DevPays(Γ_n, σ⃗) - offset)
        σ⃗ ← σ⃗ / ∑(σ⃗)
    until numIters
    if Regret(Γ, σ⃗) ≤ ϵ
        return σ⃗
    return -1
```

2.2 Deep Learning

Deep learning is a subset of machine learning that generally refers to neural networks. In section 2.2.1 we will describe how artificial neurons are organized to create neural networks, and in sections 2.2.2 and 2.2.3 we will describe how to optimize a neural network model.

2.2.1 Introduction to Neural Networks

Artificial neurons are the basic units of neural networks. They are loosely inspired by biological neurons, which are the building blocks of the brain, and are used for sensing. A *neural network* is a group of artificial neurons organized into layers that collectively learn a mapping from inputs to outputs. Under certain conditions, an artificial neuron in a neural network may activate or "turn on." The intensity of this activation is roughly related to how much that neuron affects the outputs of the neural network. An artificial neuron's activation is influenced by the activations of neurons in the previous layer of the neural network. These neighboring neurons are sometimes called *input neurons* to the given artificial neuron. The artificial neuron also sends its activation to output neurons in the next layer of the neural network and influences the output neurons' activations to varying degrees.

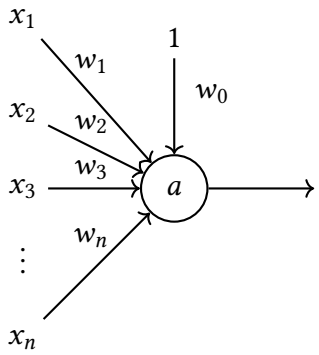


Figure 13: A graphical representation of a simple arbitrary artificial neuron.

Figure 13 shows a graphical representation of an artificial neuron. The artificial neuron is represented by a circular node in the center. The input neurons are labeled $x_0, x_1, x_2, \ldots, x_n$, where n represents the total number of input neurons. A connection between the artificial neuron and one of its input neurons is represented by a directed edge, where the source of the edge is the input neuron and the destination of the edge is the original artificial neuron. Each directed edge has an associated weight, which quantifies how much the input neuron affects the activation of the current neuron. Observe that input neurons $x_1, \ldots, x_n$ appear to the left of the artificial neuron, but a neuron x_0 labeled with a 1 appears above it. This special neuron's activation is always set to 1, and the weight w_0 corresponds to the bias term, which is similar to a y-intercept. In this figure the neuron sends output activation a to a singular output node, but in most cases this activation would be sent to many neighboring nodes in the next layer of the neural network.

Given a vector of inputs, an artificial neuron computes a weighted sum of these inputs added with the bias term; equivalently, an artificial neuron computes the dot product of the weight and input vectors. This sum is passed as input to some activation function, which determines the

intensity of the neuron's activation. This activation function determines mathematically how much that neuron contributes to the activation of neighboring neurons in the next layer and eventually the output(s). Table 1 defines several common activation functions. Note that the activation for a linear neuron is simply the dot product of the input and weight vectors. For this thesis, the Rectified Linear Unit (ReLU) activation function is the most frequently used.

Function Name	**Function Definition**
Linear	$f(x) = x$
Sigmoid (σ)	$f(x) = \frac{1}{1+e^{-x}}$
Rectified Linear Unit (ReLU)	$f(x) = \max(0, x)$
Hyperbolic Tangent (tanh)	$f(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$

Table 1: Definitions for common neural network activation functions.

A neural network *architecture* specifies an organization of artificial neurons in the neural network. More specifically, it establishes the number of layers, the number of neurons per layer, the activation functions for each neuron, and the connections or edges between neurons in adjacent layers. Every neural network has an *input layer* and an *output layer*. Any layer in a neural network that is not an input or output layer is called a *hidden layer*. A hidden or output layer is said to be *fully connected* or *dense* if every neuron in the layer is connected to every neuron in the previous layer. Neural networks are a part of *deep* learning because many neural networks are quite deep, meaning that they contain many (sometimes hundreds) of layers. Often the complexity of the architecture is related to the difficulty of the learning problem.

Figure 14 shows an example of a neural network architecture. The network contains three input neurons (x_1, x_2, x_3), a fully-connected hidden layer with six sigmoid neurons (σ), a hidden layer with four ReLU neurons (_/) and an output layer with two neurons (y_1, y_2). Note that because there are so many layers, the weight labels have been omitted to improve readability but each directed edge is still associated with a weight parameter. Additionally, note that all neural networks discussed in this thesis are *feed-forward neural networks* because they can be represented as directed acyclic graphs (DAGs).

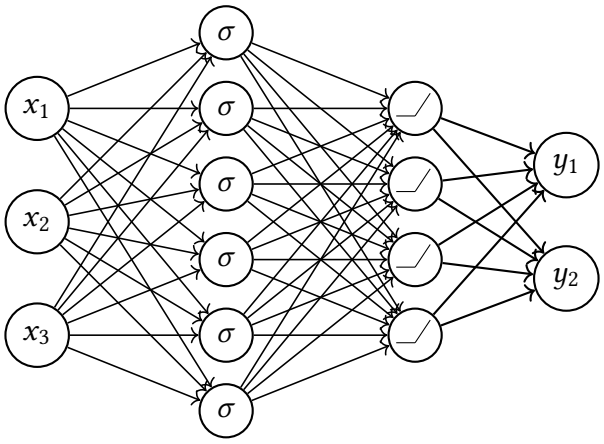


Figure 14: A simple neural network architecture with three input neurons, a hidden layer with six sigmoid neurons, a hidden layer with four ReLU neurons and an output layer with two neurons.

2.2.2 Neural Network Parameter Optimization

In the previous section, we discussed how each artificial neuron in a neural network has an associated activation function as well as a weight vector and bias term. How do we take this neural network architecture and turn it into something that can make meaningful predictions? The supervised learning process is divided into three phases: the training phase, the validation phase and the tesitng phase. The *training phase* is the phase in which the *parameters* of the model—the weights and biases—are optimized. The *validation phase* is the phase in which the hyperparameters are optimized. *Hyperparameters* are variables of the model that are not the weights or biases, but still have an effect on the outputs. This phase of the process will be discussed in greater detail in section 2.2.3. Finally, the *testing phase* is the phase where we use our optimized neural network to make predictions on new unlabeled data.

The training phase can be further subdivided into two stages: forward propagation and backpropagation. First, we initialize the weights and biases with random real values. Next, we propagate forward through the network. Equivalently, we compute the activations for each artificial neuron in the neural network (using the process described in section 2.2.1), starting with the neurons in the first hidden layer and ending with the neurons in the output layer.

These output activations (or output activation) serve as our prediction or estimate of the *ground truth*, which is the "correct" output associated with the given inputs, based on the labeled training dataset. We measure how good of a guess this estimate is using an error or *loss function*. The most common loss function for regression is *Mean Squared Error* (MSE) and is defined as

$$\text{MSE} = \frac{1}{N}\sum_{i=0}^{N}(y_i - \hat{y}_i)^2$$

where y is the ground truth and $\hat{y}$ is the predicted output. Another common regression loss function is Mean Absolute Error (MAE) which is defined as

$$MAE = \frac{1}{N}\sum_{i=0}^{N}|y_i - \hat{y}|.$$

Our objective in this training phase is to find the weights and biases that *minimize* the loss function. From calculus we know that we want to find the parameter values such that the derivative of the loss function is zero. Since we have more than one parameter in our model, we will be dealing with *partial derivatives* or *gradients*. Now, we can update each parameter according to a root-finding method. One of the most well known root-finding algorithms is Newton's method:

$$x_{i+1} = x_i - \frac{x_i}{f'(x_i)},$$

where x_i denotes the value of parameter x at iteration i and x_{i+1} denotes the new value of x after the update. This method is appealing because it has provable convergence and error guarantees, but it requires us to know both the function output and the (partial) derivative, which may not be defined or may not be straightforward to compute. The root-finding commonly used to update neural network parameters is called *gradient descent*. Recall that the negative gradient tells us the direction of steepest descent. Thus we can use the negative error gradient to update our weights

in the direction of steepest descent on the error surface to reach a local minima. Formally, we can update the weights of our neural network using gradient descent using the following formula:

$$w = w - \eta \cdot \frac{\partial \text{MSE}}{\partial w}, \tag{2}$$

where η represents the learning rate. The intuition behind this formula is that the error gradient $\frac{\partial \text{MSE}}{\partial w}$ tells us how much that weight contributed to the overall error, and so we want to update our weight in the opposite direction (to decrease the error). We multiply the negative error gradient by the *learning rate*, which is a hyperparameter that controls the speed of the learning—since we are making changes to all of the parameters in a given epoch, we want small tweaks to each weight to help ensure we are still decreasing the loss.

Because each artificial neuron has several parameters, we need partial derivatives for each of them. Instead of calculating partial derivatives fully for each parameter (independent from partial derivatives in previous layers), we can take advantage of the chain rule and store intermediate partial derivatives to save computation time. Given functions $y(x)$ and $z(y)$, the *chain rule* tells us that

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x}.$$

The process of using the chain rule to efficiently update the model parameters from the end of the network to the start is called *backpropagation*.

Consider the neural network in Figure 15. This neural network is represented using a *computational graph*, where the rectangular nodes represent input (left) or output (right) and the circular nodes represent artificial neurons with activation functions.

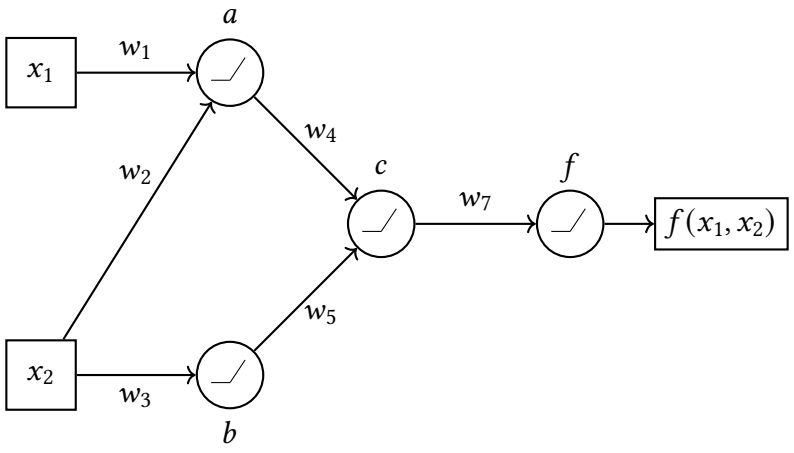


Figure 15: A simple example of a neural network represented by a computational graph.

Observe that the “usual" labels appear in the graph—the input, weight and output labels—but additionally there are labels a, b, c and f. We can rewrite the neural network as a composition of activation functions, where

$$\begin{aligned} a &= \max(0, w_1 x_1 + w_2 x_2) \\ b &= \max(0, w_3 x_2) \\ c &= \max(0, w_4 a + w_5 b) \\ f &= \max(0, w_6 c) \end{aligned}$$

and $\text{MSE} = (y - f)^2$. Note that f represents the neural network function $f(x_1, x_2)$.
Next, we can calculate the partial derivatives for each intermediate variable:

$$\frac{\partial f}{\partial c} = \begin{cases} w_6 & f > 0 \\ 0 & f \leq 0 \end{cases} \qquad \frac{\partial c}{\partial b} = \begin{cases} w_5 & c > 0 \\ 0 & c \leq 0 \end{cases} \qquad \frac{\partial c}{\partial a} = \begin{cases} w_4 & c > 0 \\ 0 & c \leq 0 \end{cases}$$

$$\frac{\partial b}{\partial x_2} = \begin{cases} w_3 & b > 0 \\ 0 & b \leq 0 \end{cases} \qquad \frac{\partial a}{\partial x_2} = \begin{cases} w_2 & a > 0 \\ 0 & a \leq 0 \end{cases} \qquad \frac{\partial a}{\partial x_1} = \begin{cases} w_1 & a > 0 \\ 0 & a \leq 0 \end{cases}$$

and the partial derivative of the loss function with respect to f:

$$\frac{\partial \text{MSE}}{\partial f} = -2 \cdot (y - f).$$

Additionally, we know that

$$\frac{\partial f}{\partial w_6} = c \qquad \frac{\partial c}{\partial w_5} = b \qquad \frac{\partial c}{\partial w_4} = a$$

$$\frac{\partial b}{\partial w_3} = x_2 \qquad \frac{\partial a}{\partial w_2} = x_2 \qquad \frac{\partial a}{\partial w_1} = x_1$$

Then, using the chain rule we have

$$\frac{\partial \text{MSE}}{\partial w_6} = \frac{\partial \text{MSE}}{\partial f} \cdot \frac{\partial f}{\partial w_6} = \delta_f \cdot \frac{\partial f}{\partial w_6} \qquad \frac{\partial \text{MSE}}{\partial w_3} = \frac{\partial \text{MSE}}{\partial f} \cdot \frac{\partial f}{\partial c} \cdot \frac{\partial c}{\partial b} \cdot \frac{\partial b}{\partial w_3} = \delta_b \cdot \frac{\partial b}{\partial w_3}$$

$$\frac{\partial \text{MSE}}{\partial w_5} = \frac{\partial \text{MSE}}{\partial f} \cdot \frac{\partial f}{\partial c} \cdot \frac{\partial c}{\partial w_5} = \delta_c \cdot \frac{\partial c}{\partial w_5} \qquad \frac{\partial \text{MSE}}{\partial w_2} = \frac{\partial \text{MSE}}{\partial f} \cdot \frac{\partial f}{\partial c} \cdot \frac{\partial c}{\partial a} \cdot \frac{\partial a}{\partial w_2} = \delta_a \cdot \frac{\partial a}{\partial w_2}$$

$$\frac{\partial \text{MSE}}{\partial w_4} = \frac{\partial \text{MSE}}{\partial f} \cdot \frac{\partial f}{\partial c} \cdot \frac{\partial c}{\partial w_4} = \delta_c \cdot \frac{\partial c}{\partial w_4} \qquad \frac{\partial \text{MSE}}{\partial w_1} = \frac{\partial \text{MSE}}{\partial f} \cdot \frac{\partial f}{\partial c} \cdot \frac{\partial c}{\partial a} \cdot \frac{\partial a}{\partial w_1} = \delta_a \cdot \frac{\partial a}{\partial w_1}$$

We can use this information to update each weight according to formula 2:

$$w_6 -= \eta \cdot c \cdot -2(y-f) \qquad w_5 -= \eta \cdot b \cdot \delta_f \cdot \begin{cases} w_6 & f > 0 \\ 0 & f \leq 0 \end{cases} \qquad w_4 -= \eta \cdot a \cdot \delta_f \cdot \begin{cases} w_6 & f > 0 \\ 0 & f \leq 0 \end{cases}$$

$$w_3 -= \eta \cdot x_2 \cdot \delta_c \cdot \begin{cases} w_5 & c > 0 \\ 0 & c \leq 0 \end{cases} \quad w_2 -= \eta \cdot x_1 \cdot \delta_c \cdot \begin{cases} w_4 & c > 0 \\ 0 & c \leq 0 \end{cases} \quad w_1 -= \eta \cdot x_1 \cdot \delta_c \cdot \begin{cases} w_4 & c > 0 \\ 0 & c \leq 0 \end{cases}$$

Generally speaking, we can update weights as follows:

$$w_{h \to o} -= \eta \cdot \text{act}'(h_i) \cdot -2(y - \hat{y})$$
$$w_{h_i \to h_j} -= \eta \cdot \text{act}'(h_i)\delta_{h_j} \cdot \sum_{k \in next} \delta_{h_k}$$

where $w_{h \to o}$ denotes a weight between a hidden layer node and an output node, $w_{h_i \to h_j}$ denotes a weight between two hidden layer nodes, and $\text{act}'(h_i)$ is the activation of from the input node associated with the given weight.

Thus far we have discussed using only one training example to update weights, but we want to find the best weights for the entire training set. Theoretically, we could do forward propagation and backpropagation on the entire training set. However, this is inefficient because many datasets are quite large and there is also likely some redundancy in that multiple data points might help the neural network learn the same thing. Often researchers use *stochastic gradient descent* which means that at each step, we select random samples from the training data to do forward propagation and backpropagation. This random sample from the data set used for backpropagation is called a *batch* and the *batch size* is an important hyperparameter of the model.

2.2.3 Neural Network Hyperparameter Optimization

Previously we observed that the objective of the training phase is to learn the "best" parameter values. We defined "best" to refer to the set of parameter values which minimize the loss function. Do we want our neural network to predict the *exact* outputs for each input in our training set? To answer this question, consider a simple regression problem shown in Figure 16. In Figure 16a, the line of best fit passes through every point exactly, whereas in Figure 16b, the line of best fit describes a linear trend through the points. Recall that we want to train a model so it can predict outputs for new inputs. Which model would likely give us better output predictions for new inputs? The model which crosses through every point exactly in the training set (Figure 16a) or the model which describes a general trend for the points (Figure 16b)? The answer is Figure 16b. In Figure 16a, the model is *overfitting* to the data whereas in Figure 16b the model will likely *generalize well to new data.*

What does this mean for our objective in the training phase and in the overall supervised learning process? While the mathematical objective in the training phase is to find the parameter values that minimize the loss function, the overall objective of the supervised learning process is to find the best parameter values such that the model generalizes well to new data. Mathematically, we want the value of our loss function on the trained dataset to be similar to that on the validation dataset, which contains unseen labeled data points. Figure 17 is a typical plot showing

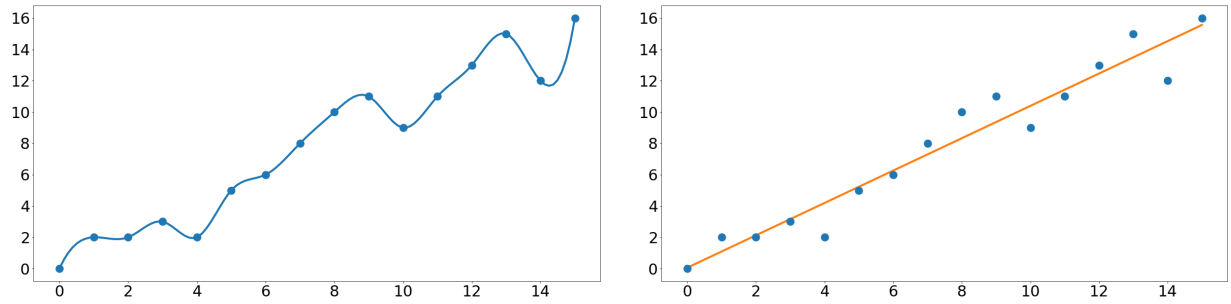


Figure 16: Comparison of regression models where model (a) overfits to the data and model (b) generalizes well to new data.

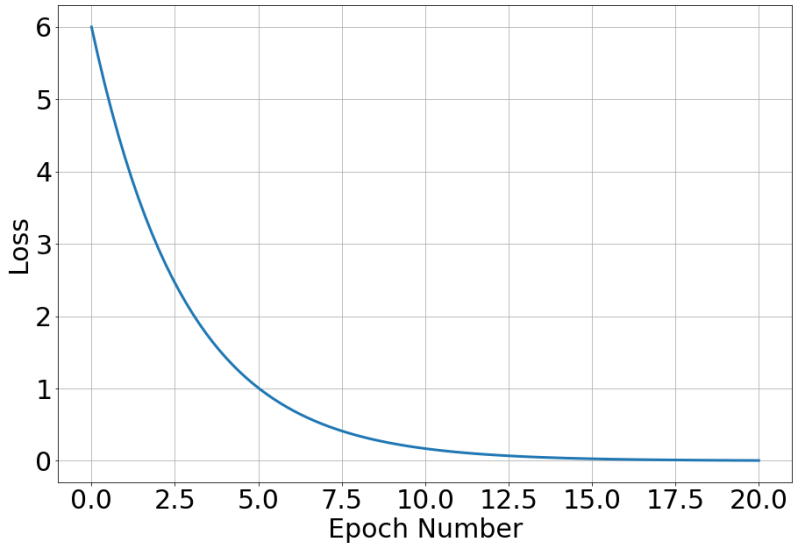


Figure 17: A typical plot showing loss vs epoch number.

the loss vs epoch number for a model during the training stage. In general, as the slope of the loss curve flattens, the model will tend to overfit to the training data more. Thus one of the objectives of the validation stage is to determine the maximum number of training epochs that can be used to decrease the loss before the model starts to overfit to the training data.

As previously stated, *hyperparameters* are variables of the model that are not the weights or biases, but still have an effect on the outputs. These variables can help decrease the loss and improve how well the model generalizes to new data. Since there is no principled way to select them, unlike the parameters of the model, one usually tunes these hyperparameters empirically. Common hyperparameters include:

- Number of hidden layers
- Number of nodes per layer
- Type(s) of layers
- Choice of activation function
- Optimizer
- Learning rate
- Batch size
- Number of epochs
- Weight initialization
- Learning rate decay
- Batch normalization
- Regularization

We determine the best hyperparameter values during the validation stage. Most often it is infeasible to enumerate all possible combinations of hyperparameter settings and evaluate each model variation on the training and validation sets. To make the hyperparameter optimization problem more tractable, for each hyperparameter we determine a range of values that we think is appropriate. Some intuition behind picking these appropriate hyperparameter ranges comes from prior experience with neural network hyperparameter optimization whereas some intuition comes from experience working with similar types of problems. For example, from experience we know not to set the learning rate to be large; instead we know that the learning rate will be smaller, potentially 0.001 to 0.1. We also know that convolutional layers are particularly useful for image classification problems, from previous experience working with similar problems.

Now we can attempt to optimize these hyperparameters through one of two approaches: grid search or random search. In a *grid search*, we specify representative intermediate values within the range for each hyperparameter, enumerate all possible combinations of hyperparameter settings and evaluate each model variation on the validation set. Because of the combinatorial nature of this search method, it is essential to restrict the hyperparameter ranges as much as possible. In a *random search* we randomly generate representative values within the range for each hyperparameter, enumerate all possible combinations of hyperparameter settings and evaluate each model variation on the validation set. Finally, we select the model variation which produces the lowest error on the validation set, where this error is close to the error of the training set.

In practice, it is often possible to "prune" hyperparameter settings—from one model variation to the next, vary one parameter value at a time to try to infer how that parameter might affect the model's validation error. If several model variations perform relatively poorer than other model variations for a fixed hyperparameter, that hyperparameter value is likely not the most appropriate for the particular problem. Thus we may "prune" (or at least deprioritize) future model variations where that particular hyperparameter has a similar value. Ultimately, these search methods are educated trial-and-error methods, and with experience these searches become more streamlined.

3 Related Work

3.1 Learning Game Models from Data

A number of examples in the literature use machine learning techniques to solve problems in game theory. In extensive-form settings, including Go and poker, significant progress has been achieved recently by using different types of learning, including supervised, unsupervised, and semi-supervised learning, to improve tractability for real-time game playing. For example, Silver et al. [42] combine supervised learning techniques using historical game data from human experts and reinforcement learning based on simulated games of self-play in their AI *AlphaGo*. More recently, Silver et al. [43] created *AlphaGo Zero* which defeated *AlphaGo* and no longer requires supervised learning with historical game data from experts due to a more sophisticated reinforcement learning technique. Brown and Sandholm [5] use reinforcement learning in their AI *Pluribus* which outperforms the best human professionals in six-player no-limit Texas hold'em poker. This success is particularly significant because many of the previous top poker AIs were limited to playing in two-player settings.

As previously discussed, many of the types of interactions we want to model using simulation-based game theory involve settings with a large number of players and in some cases a large number of strategies. As a result, several proposed techniques in the literature try to improve the tractability of simulation-based game analysis. To address challenges that arise when analyzing games with a large number of players, many works in the past have employed a player reduction technique [61, 13, 62] and more recent works have applied regression techniques [63, 45].

In section 2.1.4 we discussed action-graph games [20] as a compact game representation for normal-form games. Other compact game representations include graphical games [25] and resource-graph games [19]. While these compact game representations do simplify the space required to store payoff information as well as reduce the computation required for finding Nash equilibria, these representations generally require knowledge about the underlying structure of the payoff functions that is not available in simulation-based settings. Ficici et al. [13], Honorio and Ortiz [18], and Li and Wellman [30] have proposed methods for deducing this payoff structure information from empirical data, but these methods have limited applicability.

Several papers have employed machine learning techniques to learn game models from data. For example, Vorobeychik, Wellman, and Singh use low-degree polynomials, local regression, and support vector machines to learn the payoff function in infinite games with real-valued strategies. [52]. Wiedenbeck, Yang, and Wellman use Gaussian process regression to learn the utility function in symmetric games with a large number of players [63]. Areyan Viqueira, Cousins, and Greenwald propose two algorithms to uniformly approximate simulation-based games with a guaranteed finite number of queries to the simulator. However, these proven bounds likely do not scale well in settings with a large number of players. Li and Wellman introduce two algorithms: a clustering algorithm to partition a set of players into roles, and an algorithm to learn a graphical game model [30].

Most directly relevant is the work of Sokota et al. [45], who use a multi-headed neural network to learn a mapping from role-symmetric mixed strategy profiles to deviation payoffs. Their technique takes advantage of player symmetries common in simulation-based settings in order to learn the deviation payoff function in games with a large number of players. This learned deviation payoff function is used in Nash-finding algorithms to find approximate role-symmetric

equilibria in simulation-based games, without constructing an explicit payoff table. We generalize the work of Sokota et al. to analyze symmetric simulation-based games with a variable number of players.

3.2 Games with a Variable Number of Players

Many works in the literature evaluate how game-theoretic *algorithms* scale with the number of players. For example, Wang et al. [56] present a multi-agent Q-learning algorithm for service composition and evaluate the scalability of their method on a variable number of agents. Panagopoulou [37] proposes an algorithm to compute approximate Nash equilibria in symmetric bimatrix games, and evaluates the algorithm on two different numbers of players. Sureka and Wurman [46] use metaheuristic techniques including genetic algorithms and tabu search to find the best response strategy in order to find Nash equilibria in combinatorial auctions. They validate their results on a variable number of agents, ranging from 2 to 4.

However, few works investigate how game-theoretic *models* scale with the number of players. In section 3.2.1 we present several simulation-based game-theoretic models in which the number of players is a hyperparameter of the model. This is a limitation of how game theorists *conceive* of game-theoretic models. Finally, in section 3.2.2 we discuss examples of analysis of games with a variable number of players for other classes of games and economic models. For all classes of games discussed, we describe how each example could benefit from our technique.

3.2.1 Simulation-Based Games

Several examples in the literature analyze simulation-based games where the number of players is varied. For example, Wah and Wellman [54] construct a simulation-based game to analyze the effects of latency arbitrage in financial markets. *Latency arbitrage* is a high-frequency trading strategy in which "an advantage in access and response time enables the trader to book a certain profit." The authors analyze the same game with a variable number of background traders (24, 58, 238), but analyze each instance of the game *separately* because the number of players is a hyperparameter of the model. For each of the three instances, they find symmetric equilibria in the game and then evaluate background-trader surplus and latency arbitrageur profit (if applicable) to conclude that latency arbitrage reduces surplus overall. Using our techniques, Wah and Wellman could analyze the entire game at once, with the number of players ranging from 24 to 238, instead of analyzing each instance separately.

In addition to investigating the effects of latency arbitrage in financial markets [54], Wah and Wellman [53] compare the welfare of traders in frequent call markets versus continuous double auctions. They analyze four environments *separately*, varying the number of environmental agents (8, 14, 42) and the mean-reversion parameter ($\kappa = 0.05, 0.01$). Also, Wellman, Kim and Duong [60] construct a simulation-based game to evaluate complex network routing protocols. They *separately* analyze three different environments of the game, with total number of non-attacking nodes equal to 633, 2045, and 4956 non-attacking players (clients, ISPs, roots, servers) in the three environments. The hope is that having equilibria that are robust within the range of player counts will enable more meaningful analysis of and predictions about real-world games.

3.2.2 Other Examples

Game-theoretic analysis of settings with a variable number of players also exists outside of simulation-based game settings. For example, Tuffin and Maillé [50] model a TCP session using an Additive-Increase Multiplicative-Decrease (AIMD) process and find Nash equilibria in the game to determine a stable number of parallel TCP sessions to open. In particular, they analyze how the number of players (varied from 2 to 10) affects how many parallel TCP sessions should be opened in equilibrium, where the number of players is a hyperparameter of the model. Another hyperparameter of the model is α, which represents the price per TCP session. To determine which pricing scheme helps manage connection congestion the best, the authors vary the α hyperparameter for a small range of alpha values and look at the number of TCP sessions as well as corresponding revenue in equilibrium. Tuffin and Maillé were likely only able to perform such analysis because the number of players, the size of the player-count range and the size of the pricing range were small. Our technique could enable similar analysis more broadly, for a wider range of parameter values.

Thompson et al. [49] use action-graph games to model voting in plurality elections. They investigate the probability of truthful vs. pure-strategy Nash equilibrium existence when the number of voters is varied from 3 to 96 and the number of candidates is varied from 3 to 5. While the magnitude of this player-count range is large, the number of players is still a hyperparameter of the model. The authors also explore the proportion of equilibria in which each type of winner, truthful and Condorcet, is elected as the numbers of voters and candidates are varied. The authors observed that "when voter number was odd, the probability of having no equilibria at all increased dramatically." For the equilibria that do exist for each instance, it would be interesting to explore pure-strategy robustness metrics which identify equilibria which are robust to the number of voters as well as the voter-count parity. Further, it would be interesting to build a model that distinguishes between the parity of the player count, maybe even making the parity a feature of the model.

Additionally, game-theoretic analysis of settings with a variable number of players has been conducted on other classes of games as well as in pure economics. Petruzzi, Pitt and Busquets [38] evaluate the effect of varying the magnitude of population size on the performance of social capital, given equal proportions of dominant, random, and social players. In the simulations, the number of social players is varied from 15 to 300. Fatima [12] compares sequential and simultaneous auctions by varying the number of objects and bidders and evaluating the two mechanisms on three properties: the expected cumulative surplus, the bidder's ex-ante expected profit, and the auctioneer's expected cumulative revenue. Hanaki and Rouchier [17] use a differentiated-goods Cournot competition to study whether ignorant agents can behave strategically to become richer than informed agents instead of by luck. In their experiments, they specify a variable number of ignorant and informed agents and compare average level of outputs and profits for each type of agent. The economics literature includes numerous examples of models with a continuum of players, and a few deliberate attempts to generalize over the number of players. For example, Kalai [23] develops a repeated-game model where players respond to aggregations over an uncertain number of opponents.

In many of these examples, the number of players is a hyperparameter of the model. In the remainder, the magnitude or range of player counts is relatively small, likely due to limitations on available tools. Both demonstrate the need for analysis that spans a variable number of players.

4 Variable-Player Game Model and Analysis

This section presents the main contributions of this thesis which aim to answer the following questions:

1. How do we *construct* a variable-player game-theoretic model?
2. How do we *analyze* a variable-player game-theoretic model?

First, in section 4.1 we define a variable-player game as well as an instance of a variable-player game. In sections 4.2 and 4.3 we describe how we generalize Sokota et al.'s work [45] to learn a variable-player game-theoretic model. Finally, in section 4.4 we discuss how to analyze a variable-player game, define the concept of *robustness*, and present several equilibrium robustness metrics.

4.1 Variable-Player Games

We define a *variable-player symmetric game* as a tuple $\tilde{\Gamma}_{mn} = (P, S, u)$ where

- $P = \{m, \ldots, n\}$ defines a range of player counts, where m is the minimum number of players and n is the maximum number of players in the game.
- $S = \{s_1, \ldots, s_k\}$ is a set of strategies available to all players, where there are k total strategies.
- $C_{mn} = \{\vec{c} \in \mathbb{Z}^k : \vec{c}_j \geq 0, m \leq \sum_{j=1}^{k} \vec{c}_j \leq n\}$ is the set of all possible player configurations which varies with the number of players.
- $u : C_{mn} \mapsto \mathbb{R}^k$.

For $\vec{c} \in C_{mn}$, the utility function $u_j(\vec{c})$ describes the payoff players playing strategy $s_j \in S$ receive when players play according to $\vec{c}$. Note that in addition to C_{mn}, the utility function u also varies with the number of players. The number of players can be inferred from $\vec{c}$ and is equal to $\sum_{j=1}^{k} \vec{c}_j$. Further, we define an *instance* of a variable-player game by $\tilde{\Gamma}_p = (p, S, u)$ where

- p is the number of players, with $m \leq p \leq n$.
- $S = \{s_1, \ldots, s_k\}$ is a set of strategies available to all players, where there are k total strategies.
- $C_p = \{\vec{c} \in \mathbb{Z}^k : \vec{c}_j \geq 0, \sum_{j=1}^{k} \vec{c}_j = p\}$.
- $u : C_p \mapsto \mathbb{R}^k$

Observe that a instance $\tilde{\Gamma}_p$ of a variable-player symmetric game $\tilde{\Gamma}_{mn}$ is a fixed-player symmetric game.

4.2 Approximating Deviation Payoffs

Sokota et al. [45] use a multi-headed neural network to learn the deviation payoff function for games with a fixed number of players. We hypothesize that games with a variable number of players have a different, but related, payoff function for each instance of the game within the player count range. More specifically, we hypothesize that the payoffs in a game with x players are similar or related to the payoffs in the same game with $x \pm 1$ players, given a large value of x. As a result, the variable-player learning problem is much more complex than that of fixed-player learning. With this hypothesis we generalize [45] to learn the deviation payoff function in variable-player simulation-based games. This deviation payoff function maps symmetric mixed-strategy profiles and number of players to deviation payoffs for each strategy, and can be used in Nash-finding algorithms to compute robust approximate equilibria in games with a large, variable number of players.

One of the motivations for learning the deviation payoff function is that it is intractable to compute deviation payoffs in games where the number of players is large (and consequently in variable-player games where the player counts in the range are large). How can we construct our training dataset without actually calculating deviation payoffs? First, we can generate a symmetric mixed-strategy profile $\vec{\sigma}$. Next, we can draw a strategy for each opponent according to that profile and then we can query the simulator for the pure-strategy payoffs associated with the profile. If we repeat this process many times for the same profile and look at the distribution of payoffs, the deviation payoff is the expected value of the distribution. Thus we can use the noisy pure-strategy payoff vector as an estimate for the deviation payoff vector to prevent us from summing over the entire payoff matrix.

Formally, we train a multi-headed neural network on the mapping devPay : $(\vec{\sigma}, p) \mapsto u(\vec{\sigma}, p)$ where $\vec{\sigma}$ is a symmetric mixed-strategy profile, p represents the number of players where $m \leq p \leq n$, and $u(\vec{\sigma}, p)$ represents doubly noisy pure-strategy payoff estimates from an agent-based simulator. To generate an entry in our training set, we sample a symmetric mixed-strategy profile $\vec{\sigma} \sim \text{Dir}(\vec{\alpha})$ from a Dirichlet distribution, where $\vec{\alpha} = (\alpha_1, \ldots, \alpha_k)$, and where $\alpha_i < 1$ for all $\alpha_i \in \vec{\alpha}$. Sokota et al. [45] demonstrated the need to oversample the edges of the simplex, which is why we use a Dirichlet distribution as opposed to a uniform distribution. To generate the player count p associated with mixture $\vec{\sigma}$, we propose two methods. In the first method, we select several "representative instances" or player counts within the range and then randomly generate $p \sim U[m, i_2, i_3, \ldots, n]$ uniformly from the list of representative instances. Alternatively, we randomly select an associated player count $p \sim U[m, m+1, \ldots, n]$ uniformly across the entire range of player counts.

Once we have our inputs $(\vec{\sigma}, p)$, we can estimate the corresponding deviation payoffs according to the following two-step process. First, we sample an opponent configuration $\vec{s} \sim \vec{\sigma}$ according to $\vec{\sigma}$. Equivalently, for each of the $p-1$ opponents, we draw an independent random sample for the mixture, which effectively assigns that opponent a particular strategy to play; then we count how many opponents are playing each strategy in order to construct opponent configuration $\vec{s}$. Next, we query the simulator for the pure-strategy payoff vector $u(\vec{s})$. Note that the number of opponents is inferred from the fact that $\sum_{j=0}^{k} \vec{s}_j = p - 1$. The pure-strategy utility vector $u(\vec{s})$ serves as a doubly noisy payoff estimate for the ground truth deviation payoff vector devPay$(\vec{\sigma})$ because the pure-strategy utility vector is a noisy sample for a sampled opponent profile.

After we have constructed our training set, we must normalize the input and output variables. Let $(\vec{\sigma}, p, u(\vec{\sigma}, p))$ be an arbitrary entry in the training set. Because the first k dimensions of the neural network correspond to entries in the k-dimensional symmetric mixed-strategy profile $\vec{\sigma}$, we do not need to normalize these dimensions in the training set. We normalize p as follows:

$$p = \frac{p - m}{n - m},$$

where m is the minimum number of players in the game and n is the maximum number of players in the game. For each of the k dimensions of $u(\vec{\sigma}, p)$, we estimate the true game maximum and minimum payoff values with the maximum and minimum payoffs returned from the simulator for that dimension. Then for each dimension j corresponding to strategy $s_j \in S$, we normalize $u_j(\vec{\sigma}, p)$ as follows:

$$u_j(\vec{\sigma}, p) = \frac{u_j(\vec{\sigma}, p) - min\_pay_j}{max\_pay_j - min\_pay_j},$$

where min_pay_j and max_pay_j correspond to the minimum and maximum simulated payoffs in the training set for strategy s_j.

While the optimal neural network architecture likely depends on the particular class of game,we have found that a multi-headed neural network with ReLU activation functions works well. In this case, *multi-headed* refers to a "head" for each strategy, meaning that the network spends some time training the deviation payoff function for each strategy $s_j \in S$ individually. Figure 18 shows a visual representation for the multi-headed neural network architecture used in our experiments for the variable-player learning model. As the figure shows, the neural network has $k + 1$ inputs: the first k inputs correspond to probabilities for strategies s_1 to s_k for a given mixed strategy $\vec{\sigma}$, and the $(k + 1)$st dimension corresponds to a given player count p. In this architecture there are three 64-node hidden layers with ReLU activation and a head for each strategy with a 64-node ReLU layer followed by a linear layer. Finally, the neural network has a k-dimensional output, where output dimension j corresponds to the predicted deviation payoff for playing strategy s_j when there are $p - 1$ players playing according to $\vec{\sigma}$. We will discuss our variable-player learning experiments thoroughly in section 5.

4.3 Approximating Robust Nash Equilibria

Most algorithms for identifying ε-Nash equilibria, including replicator dynamics [48], fictitious play [4], and the global Newton method [15], can be rewritten using deviation payoffs (and the derivatives of deviation payoffs for the global Newton method). The neural network model can output estimates of these values in time proportional to the size of the network, as opposed to the size of the (exponentially larger) underlying normal-form game.

We present a variant on the algorithm proposed by Sokota et al. [45] to iteratively refine the learned deviation payoff model by focusing on areas of the simplex that are more likely to contain approximate Nash equilibria. After the model is trained, we run a Nash-finding algorithm using the learned deviation payoffs. For each returned candidate Nash equilibrium $\vec{\sigma}$ with associated player count p, we draw mixtures $\vec{\sigma}' \sim \text{Dir}(\vec{\alpha} = \omega_{mx} \cdot \vec{\sigma} + 1)$ in the neighborhood of $\vec{\sigma}$, where ω_{mx} represents the mixture resample factor and where $\omega_{mx} >> 1$. We draw normalized player

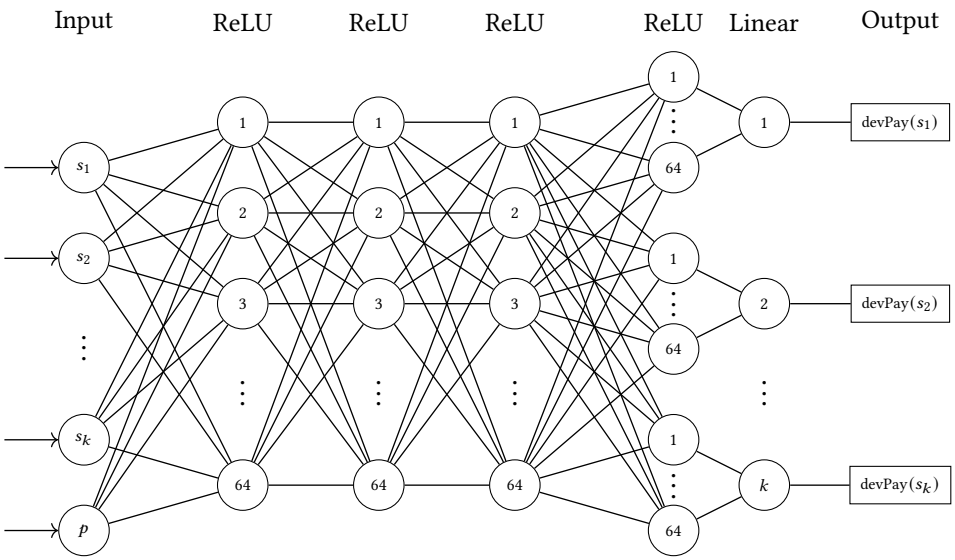


Figure 18: A visual representation for the multi-headed neural network architecture used for our variable-player learning model.

counts $p' \sim \text{Beta}(\vec{\alpha} = \omega_p \cdot [p, 1-p] + 1)$ in the neighborhood of p, where ω_p represents the player resample factor and where $\omega_p >> 1$. Figure 19 shows a simplex with color-coded points in the neighborhood of a given mixture (pink star) for three different values of ω_{mx} for a 3-strategy game.

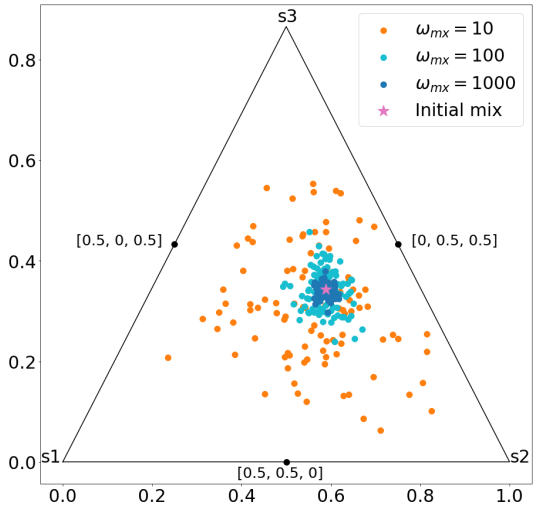


Figure 19: Simplex showing points in the neighborhood of a given mixture (pink star) for three different values of ω_{mx} for a 3-strategy game.

Using the same process as before, we sample an opponent profile $\vec{s} \sim \vec{\sigma}'$ and query the simulator to get a noisy estimate of the payoffs $u(\vec{s})$. All $(\vec{\sigma}', p', u(\vec{\sigma}', p'))$ entries are added to the initial training set, and the model is retrained. This process is repeated several times. On the last iteration, instead of sampling in the neighborhood of candidate equilibria and retraining the

network, we use the candidate equilibria returned from the Nash-finding algorithm and apply our robustness metrics (to be discussed in section 4.4) to make predictions.

Algorithm 2 provides pseudocode for computing approximate Nash equilibria in variable-player games using our learned deviation payoff model. We would like to point out four details regarding the findNash(regressor) call. First, for this thesis, we use replicator dynamics as our Nash-finding algorithm for reasons described in section 2.1.5. Also, we consider two variations of this AppxRobustNashEquilibria algorithm, one where we have an intermediate max regret check for each call to findNash(), eliminating candidate Nash with regret greater than the max regret, and one where we do not have any intermediate regret check. We evaluate these two variants in section 5. Further, the findNash() algorithm returns candidate Nash equilibria with associated player counts; the algorithm only applies robustness metrics at the end, and not after each call to findNash(). We will present several robustness metrics in section 4.4. Thus, without the findRobustNash() call at the end of the algorithm, this AppxRobustNashEquilibria algorithm essentially computes approximate Nash equilibria for each individual instance of the game, irrespective of the approximate Nash equilibria computed for other instances. If we wanted to compute robust approximate Nash equilibria in variable-player games using Sokota et al's technique [45], we would need to train a neural network for each instance of the range and iteratively refine each neural network separately according to the process above. In contrast, our variable-player learning model allows us to train and iteratively refine a single neural network. Further, as we will show in section 5, our variable-player learning model does so using less data.

Algorithm 2 Computing approximate Nash equilibria variable-player games using our learned deviation payoff model.

AppxRobustNashEquilibria(numInitQueries, numResampQueries, numIters, $\vec{\alpha}$, $\vec{\omega}$, m, n):

$[\vec{\sigma}] \leftarrow \text{Dir}(\vec{\alpha}, \text{numInitQueries})$
$[p] \leftarrow \text{Uniform}([\vec{\sigma}], m, n)$
$[\vec{s}] \leftarrow \text{sampleOppProfiles}([\vec{\sigma}], [p])$
$[\vec{u}] \leftarrow \text{samplePSPayoffs}([\vec{s}])$
data $\leftarrow ([\vec{\sigma}], [p], [\vec{u}])$
regressor.fit(data)
repeat
    $([\vec{\sigma}^*], [p^*]) \leftarrow \text{findNash}(\text{regressor})$
    $([\vec{\sigma}], [p]) \leftarrow \text{sampleNbhd}([\vec{\sigma}^*], [p^*], \vec{\omega}, \text{numResampQueries})$
    $[\vec{s}] \leftarrow \text{sampleOppProfiles}([\vec{\sigma}], [p])$
    $[\vec{u}] \leftarrow \text{samplePSPayoffs}([\vec{s}])$
    data $\leftarrow$ data + $([\vec{\sigma}], [\vec{s}], [\vec{u}])$
    regressor.fit(data)
until numIters
$([\vec{\sigma}^*], [p^*]) \leftarrow \text{findNash}(\text{regressor})$
return $\text{findRobustNash}([\vec{\sigma}^*], [p^*])$

4.4 Equilibrium Robustness Metrics

Typical game-theoretic analysis involves finding approximate Nash equilibria in a game with a fixed number of players. However, in variable-player games, finding approximate equilibria for the entire game is not necessarily feasible. For example, an approximate equilibrium in one instance of the game might not be an approximate equilibrium in any other instances and therefore it is not a good prediction of behavior in the interaction. Thus we seek to find a metric to evaluate the robustness of a candidate equilibrium across all instances of the game. Since there is no singular metric appropriate for all contexts, we present several metrics to evaluate a profile across the range of player counts: mean regret, median regret, max regret, and equilibrium frequency.

Mean Regret. The regret of a candidate Nash equilibrium $\vec{\sigma}$ is computed for each instance of the game and then averaged. If the mean regret falls under some specified regret threshold $\bar{\varepsilon}$, we say the candidate equilibrium is a robust equilibrium in the game. We hypothesized that average regret might not be as useful of a metric because in general, outliers can dominate in averages. For example, a candidate Nash equilibrium might be an approximate Nash equilibrium for many instances of the game but have a high enough regret for one or a few instances and thus not be classified as robust based on the mean regret metric. However, this did not occur often in our experiments.

Median Regret. Similar to the mean regret metric, the regret of a candidate Nash equilibrium $\vec{\sigma}$ is computed for each instance of the game; then the median regret is computed. If the median regret falls under some specified regret threshold ε, we say the candidate equilibrium is a robust equilibrium in the game. Our experimental results suggest that this metric is most useful when paired with the mean regret metric—if a candidate Nash equilibrium is considered robust based on both the mean regret metric and the median regret metric, then it is unlikely that any instances of the game have outlier regret for that profile.

Max Regret. The regret of a candidate equilibrium $\vec{\sigma}$ is computed for each instance of the game; then the max regret is computed. If the max regret falls under some specified regret threshold ε, we say the candidate equilibrium is a robust equilibrium in the game. With this metric we know that no player can gain more than ε by deviating to any other mixed strategy in any instance of the game. In a sense, for a given profile this metric summarizes the "worst-case scenario." As such it is often the case that the ε for max regret is slightly larger than the corresponding ε for other metrics that identify similar sets of equilibria.

Approximate Equilibrium Frequency. The approximate equilibrium frequency metric involves counting the number of instances in which the candidate Nash equilibrium $\vec{\sigma}$ is an ε-equilibrium, and if the count is higher than some threshold α, then the candidate equilibrium is a robust equilibrium in the game. This metric does not overpenalize a candidate equilibrium for having a high regret for a few instances but being a good approximate Nash equilibrium overall.

Figure 20 compares robust equilibria found by the three robustness metrics in a randomly generated 3-strategy game with 50 to 100 players using calculated regrets. Each point in the simplex corresponds to a symmetric mixed strategy. In Figure 20 plots (a), (b), and (c), the color shows the mean, median, and max regret respectively of the corresponding profile. Note that the white points correspond to profiles in which mean regret was greater than $\bar{\varepsilon}$ (plot (a)), median regret was greater than ε (plot (b)), or max regret was greater than ε (plot(c)). In Figure 20 plot (d), the color shows how many times each mixture was an approximate equilibrium (for a fixed ε). In this plot, the white points correspond to profiles that were never approximate equilibria.

In all four plots, the brighter points correspond to profiles that are considered more robust. The similarities between these four plots are typical for games with a variable number of players—in our experiments we have found that the four robustness metrics tend to identify similar sets of profiles as robust equilibria.

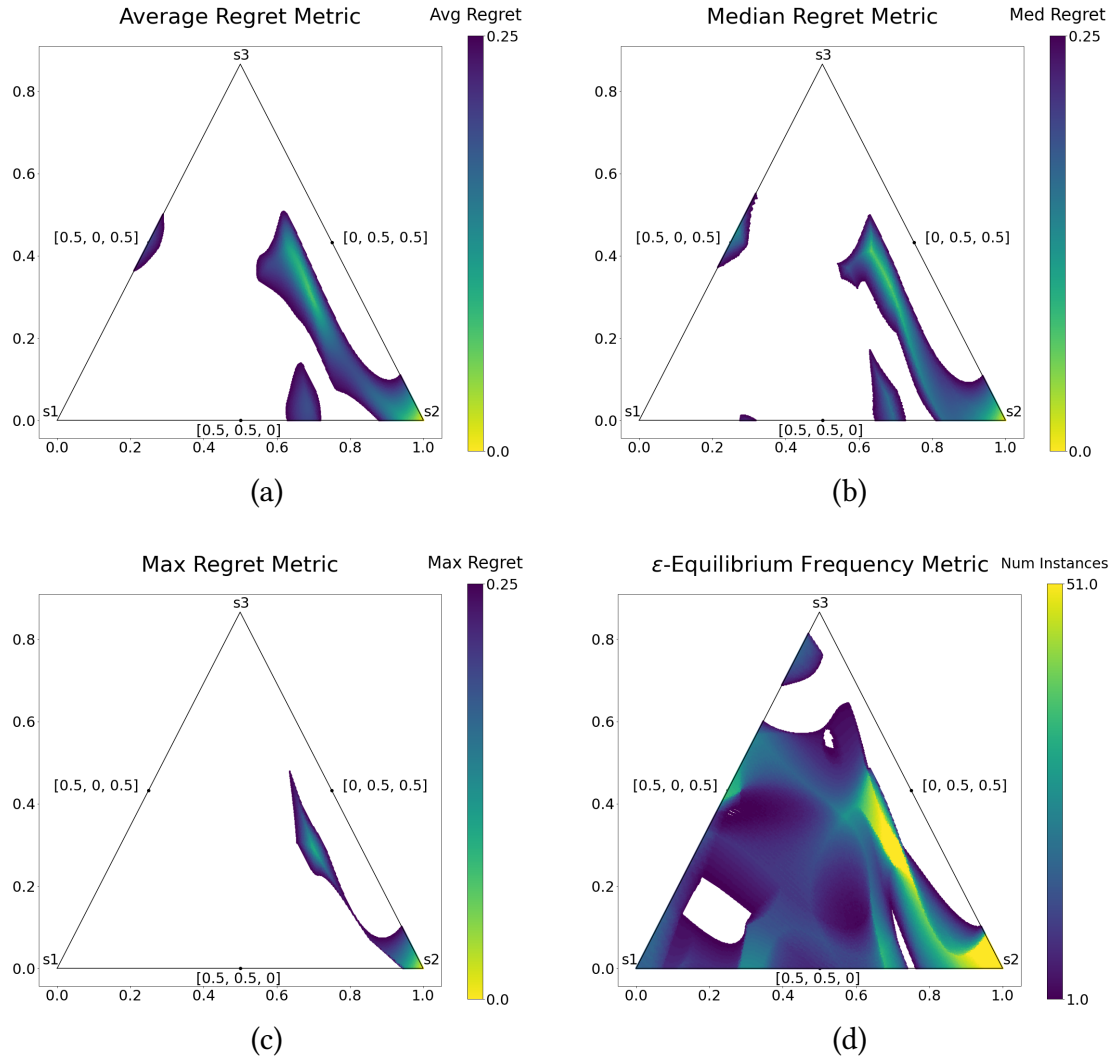


Figure 20: Comparison of four robustness metrics on a randomly generated game: (a) average regret metric, (b) median regret metric, (c) max regret metric, and (d) ε-equilibrium frequency metric.

5 Experiments

In our first set of experiments, we compare the deviation payoff learning performance of our model, which we call variable-player learning (VPL), with that of Sokota et al. [45], which we call fixed-player learning (FPL), on 250 randomly generated symmetric games. To compare these two models, we look at two different ways of generating the player counts for the VPL model training data: randomly choosing player counts from a set of representative instances (player counts) versus uniformly across the entire range. One hypothesis about the two methods is that the uniform random selection of player counts does not allow for sufficient training time for any single instance whereas the representative instances method would allow the learned model to learn more of the deviation payoff function for several instances and be able to interpolate in between those instances. The other hypothesis is that the representative instances method would cause the model to overfit to those representative instances and that the uniform random method would allow for the model to more evenly learn the deviation payoff function across the range of player counts. Our experimental results demonstrate dramatically better performance for the random method, supporting the second hypothesis.

In the second set of experiments, we evaluate our variable-player replicator dynamics algorithm. In particular, we compare the performance on a model which has been retrained to focus the learning on regions of the simplotope which are more likely to contain approximate Nash equilibria with the performance on a model which trains on the same total amount of data initially with no retraining. We hypothesize that the retrained model will outperform the model which receives all training data up front. While there remains room for further hyperparameter optimization, our preliminary experimental results support this hypothesis, serving as a proof-of-concept for the technique.

5.1 Random Game Generation

To serve as a proxy for simulator data, we generate random bipartite action-graph games with additive function nodes with 5 strategies and a player range of 50 to 100 (refer to section 2.1.4 for BAGG-FNA definition). The compact representation of these random symmetric games allows for efficient ground truth deviation payoff computation, which is useful to validate the learned models in our experiments. Also, the random games can be easily defined with a variable number of players. These random games have complex but learnable payoff functions, particularly compared to common game distributions in related literature: substantially more challenging than congestion games, but much more structured than uniform random games. We believe these random games provide the best-available proxies for simulator data.

For our experiments, we generate random directed additive polynomial sine BAGG-FNAs, as used by Sokota et al [45]. The subgraph containing edges from action nodes to function nodes (i.e., function inputs/neighborhoods) is an Erdős-Rényi random bipartite graph. The subgraph containing edges from function nodes to action nodes is a complete bipartite graph, which means that every function affects every action. The action weights are randomly generated according to a normal distribution with mean 0 and standard deviation 1 with some entries randomly masked. Each function node computes the sum of a random long-period sine function and a low degree polynomial with random coefficients.

Note that in a game with p players, the function table stores outputs associated with 0 to p players. In a game where the number of players ranges from m to n, the payoff table associated with the instance with p players, where $m \leq p \leq n$, is a subtable of the payoff table associated with the instance with n players. Thus we can define a BAGG-FNA with a variable number of players—generate the game with n players and then all payoff information for instances m to n is already stored in the function table.

5.2 Comparison to Existing Work

5.2.1 Experimental Specification

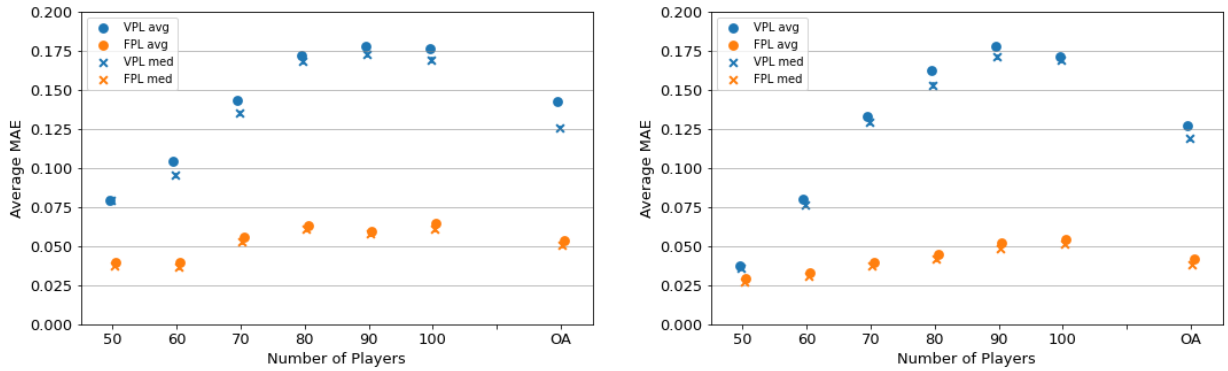
For our experiments, we evaluate both the VPL and FPL models on 250 random additive polynomial sine BAGGFNs with 50 to 100 players, 5 strategies and 10 function nodes. The FPL model consists of six multi-headed neural networks, where each neural network learns the deviation payoff function for a single instance of the game. The player counts associated with these six instances are $p = 50, 60, \ldots, 100$. The VPL model consists of a single multi-headed neural network with an additional input dimension that specifies the number of players; this model learns the deviation payoff function for the entire range of player counts. In all experiments, the total amount of training data is constant. This means that if the total amount of training data is 60,000 training examples, then the VPL model is trained with 60,000 examples and each neural network in the FPL model is trained with 10,000 examples. Based on Sokota et al.'s experiments [45], we use at minimum 10,000 training examples per FPL neural network.

The FPL architecture used is identical to that described in [45], and consists of a multi-headed neural network with 128-, 64-, and 32-node dense hidden layers and a head for each strategy with a 16-node dense layer followed by a linear layer. The VPL architecture consists of a multi-headed neural network with three 64-node dense hidden layers and a head for each strategy with a 64-node dense layer followed by a linear hidden layer. The hyperparameters for the two models are optimized separately, and for expediency, hyperparameters were tuned using other random BAGG-FNA instances; in a simulation-based game, such tuning would be performed on a hold-out set. Note that the FPL model hyperparameters are identical for all 6 neural networks.

For each instance, we evaluate network performance on 495 mixtures corresponding to points on a lattice that evenly covers the simplex. We evaluate our FPL model on 6 instances ($p = 50, 60, 70, 80, 90, 100$) and our VPL model on the same 6 instances in experiment 1 and on 12 instances ($p = 50, 55, 60, \ldots, 90, 95, 100$) in experiment 2. Note that since VPL has learned the deviation payoff function for the entire range, we could choose to evaluate network performance on any instance(s) within the range, but chose these 12 for simplicity.

We evaluate accuracy using Mean Absolute Error (MAE), averaged across mixtures. For each mixture we compute the absolute error between the predicted deviation payoff and the ground truth deviation payoff for each strategy, and compute the average error across the 5 strategies. These errors are computed on normalized deviation payoffs, so the average MAE tells us approximately what percentage we can expect our learned deviation payoffs to differ from the calculated deviation payoffs for each strategy. In both experiments we compute 95% confidence intervals for the mean and median of mixture-MAEs for each instance for a given model using the average MAEs from 495 mixtures for all 250 randomly generated games. Note that in our results plots (Figures 21 and 22), the confidence interval bars are not visible—our sample size was large enough

that the intervals are no larger than the width of the point or x that denotes the mean or median.



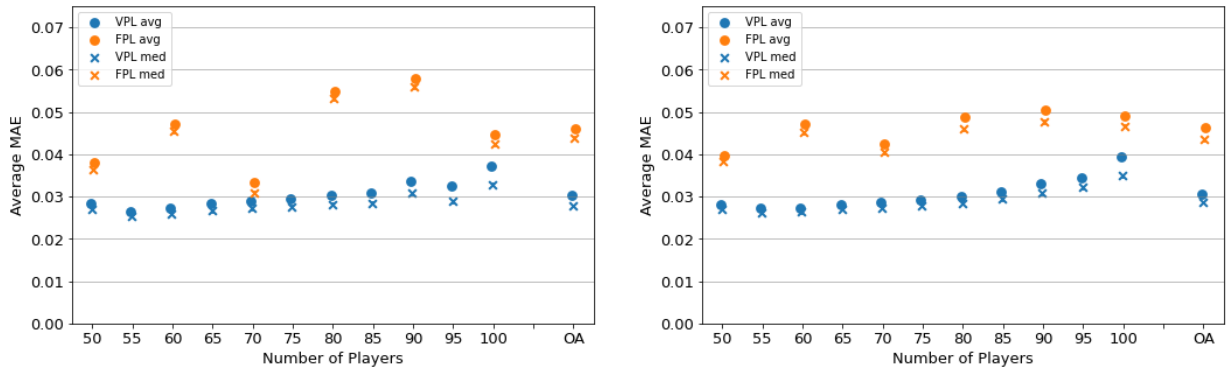
(a) (b)

Figure 21: (a) Experiment 1 with 60,000 training examples shows that FPL performs better when both methods receive identical training data. (b) Experiment 1 with 90,000 shows that FPL performs better when both methods receive identical training data, and shows diminishing returns to additional data.

5.2.2 Experimental Results

In the first experiment, the VPL model's training data player counts are generated using representative instances. We use the union of the training datasets from the FPL model to create the VPL model training dataset. This means that the representative player counts are $p = 50, 60, \ldots, 100$. Thus in this experiment, both models are trained using identical data. Figures 21a and 21b each compare the FPL model and VPL model's performances when trained on the identical data sets. The models in Figure 21a are trained with 60,000 training examples (10,000 per FPL neural network) and the models in Figure 21b are trained with 90,000 training examples. Note that *OA* denotes the overall performance across all measured instances. Clearly the FPL model performs significantly better than the VPL model trained on representative instances. We believe that the VPL model is overfitting to the deviation payoff functions for the representative player counts. VPL model performance for instances in between the representative player counts was omitted to improve figure readability, but the VPL model did not perform well on these instances either. Relative to FPL model performance, VPL model performance is also a lot less consistent across the range of player counts. Observe that FPL model performance improved slightly with 90,000 total training entries as opposed to 60,000. These experimental results demonstrate that this player count selection method is not effective in helping the VPL model learn the deviation payoff function in games with a variable number of players.

In the second experiment, the player counts in the variable-player learning model's training data are randomly selected uniformly across the entire range m to n. Figures 22a and 22b compare the FPL model and VPL model's performances when the player counts are selected randomly from the entire range. The models in Figure 22a are trained with 60,000 training examples (10,000 per FPL neural network) and the models in Figure 22b are trained with 90,000 training examples. Note that *OA* denotes the overall performance across all measured instances. Contrary to VPL model performance in experiment 1, the VPL model performs at least as well as the FPL model, if



(a) (b)

Figure 22: (a) Experiment 2 with 60,000 training examples shows the potential for VPL to significantly out-perform FPL, when VPL training data player counts are randomly selected from $m \leq p \leq n$. (b) Experiment 2 with 90,000 training examples shows greater consistency in FPL with additional data, but VPL with random player counts still performing better.

not better. In this experiment, VPL model performance is much more consistent across the range of player counts than FPL model performance. It is important to note that all FPL neural networks were trained using the same hyperparameters. While this might contribute to the less consistent performance across the range of player counts (as opposed to if each FPL neural network had been optimized separately), it is a drawback of the FPL method that one must optimize hyperparameters for many neural networks (especially when the single VPL neural network performs as well as it does). Consistent with experiment 1, observe that FPL model performance improved slightly with 90,000 total training examples as opposed to 60,000. VPL model performance does not show any noticeable improvements from 60,000 to 90,000 training examples, so we believe 60,000 training examples is sufficient for this model. These experimental results demonstrate that the random player count selection method is effective in helping the VPL model learn the deviation payoff function in games with a variable number of players.

5.3 Variable-Player Replicator Dynamics Evaluation

5.3.1 Experimental Specification

Through this set of experiments, we want to determine whether the iterative refinement is necessary. We evaluate performance on 500 random additive polynomial sine BAGG-FNAs with 50 to 100 players, 5 strategies and 10 function nodes. We compare model performance for the following four model variations:

Initial training points	Resample/retrain?	RD max regret	Total training points
10,000	Yes	$\varepsilon \leq 0.1$	~20,000
10,000	Yes	No check	~50,000
20,000	Yes	$\varepsilon \leq 0.1$	~50,000
60,000	No	$\varepsilon \leq 0.1$	60,000

Table 2: Summary of variable-player replicator dynamics model variations evaluated.

Note that “RD max regret" (sometimes referred to as “max regret check" or “intermediate regret check") means that we eliminate candidate Nash with regret greater than some specified max regret threshold for each call to replicator dynamics. All four model variations have the same VPL architecture—a single multi-headed neural network with three 64-node dense hidden layers and a head for each strategy with a 64-node dense layer followed by a linear hidden layer. For the learned variable-player replicator dynamics algorithm, there are a significant number of new hyperparameters, as summarized below:

- Number of initial queries
- Number of initial training epochs
- Number of replicator dynamics mixtures
- Max regret (if intermediate regret check)
- ω_{mx}, the mixture resample factor
- Number of resample queries
- Number of retrain epochs
- Number of replicator dynamics iterations
- Number of resample/retrain iterations
- ω_p, the player count resample factor

In the next section we will present our results from an initial pass at optimizing the hyperparameters. These results serve as a proof-of-concept for the learned variable-player model. That being said, we believe we can reduce the regret error with further hyperparameter optimization in the future.

Similar to the previous experiments, we evaluate accuracy using Mean Absolute Error (MAE). For each candidate Nash equilibrium and associated player count, we compute the absolute error between the predicted and ground truth regret for that instance, and then compute the MAE across all mixtures. These errors are computed based on normalized deviation payoffs which were used to compute predicted and ground truth regret values, so the MAE tells us approximately what percentage we can expect our predicted regret to differ from the ground truth regret for a given profile. We also compute 95% confidence intervals for the MAE for all candidate Nash equilibria and all 500 randomly generated games. Note that for some points in our results plots, the confidence interval bars are not visible—our sample size was large enough that the intervals are no larger than the width of the point that denotes the MAE.

5.3.2 Experimental Results

Results from the four model variants evaluated on variable-player replicator dynamics are shown in Figure 23. Observe that the variable-player learning model with training data spread out through initial training and retraining and with intermediate regret check (max regret = 0.1) outperforms the model with all training data used up front with no retraining and the model with data spread out through initial training but without intermediate regret checks.

Note that resample/retrain iteration 0 refers to model results after the initial training (before any resampling/retraining), and each subsequent iteration i specifies how many times the model has been retrained before it was evaluated. Also, we did evaluate performance on the model with 60,000 initial training points after all 5 iterations of resampling and retraining. In this experiment, adding more data through resampling and retraining did not improve model performance—the regret MAE hovered around 0.3 for all 6 iterations. As a result, that line has been omitted to improve figure readability and replaced with a star representing the regret MAE after the initial training. Additionally, note that every so often a mixture in replicator dynamics migrates to a region of the simplex which we have not focused training on, and the deviation payoff (and

therefore regret) MAEs are quite large. This problem only occurs every few experiments, and if it does occur in an experiment with 500 games, it only happens for a single game and one particular retraining iteration. That being said, the estimates are bad enough to mess up the MAE confidence bars for the entire experiment. Clearly, this is an issue that we need to address in the future. However, because it is so rare and likely caused by resampling hyperparameters that are not fully optimized, we temporarily ignored experiments where this corner-case occurred in order to give a better understanding of overall model performance.

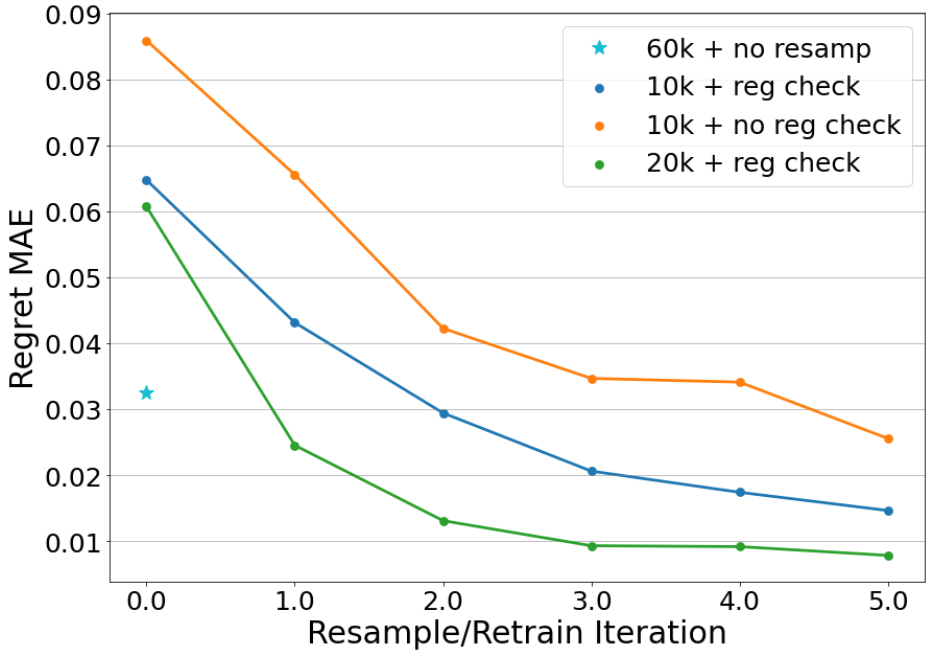


Figure 23: The variable-player learning model with training data spread out through initial training and retraining and with intermediate regret checks outperforms the model with all data up front and no retraining and the model with data spread out through initial training but without intermediate regret checks.

Figure 24 shows regret MAE performance for each instance from 50 to 100 players and for each of the 6 training/retraining iterations of the best-performing model (20k initial training points, resampling/retraining, intermediate regret check). Note that the final resample/retrain iteration (iteration 5) is depicted with a solid black line to emphasize that the model performance is best for this final iteration. Observe how model performance improves for each additional resample and retrain iteration, particularly on instances with higher player counts. This further supports the effectiveness of the iterative refinement in our variable-player learned model.

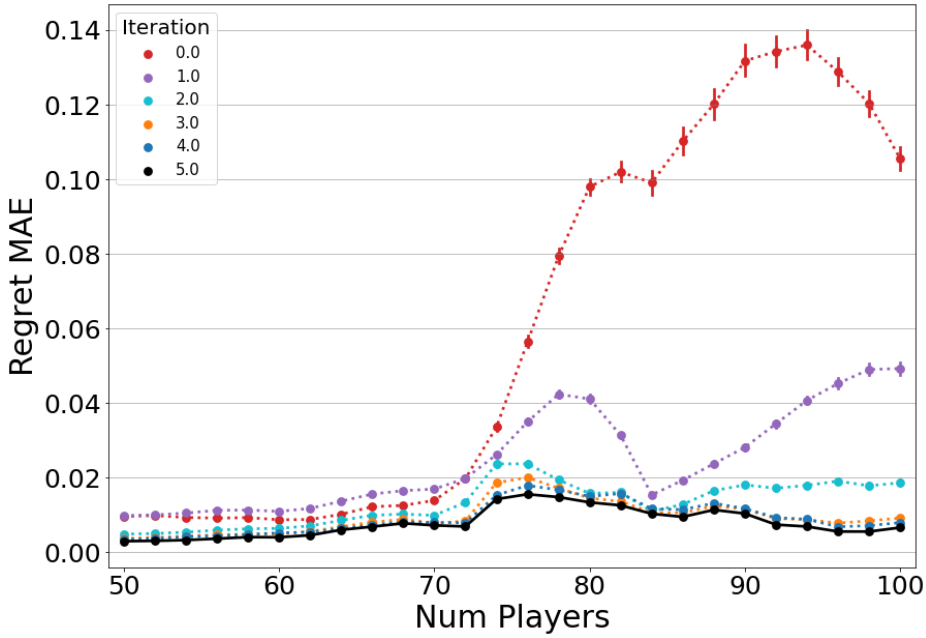


Figure 24: The model with 20,000 initial training data points, resampling/retraining, and intermediate regret check shows improvement in regret MAE after each resample/retrain iteration, particularly on instances with higher player counts.

6 Conclusion

Many of the real-world interactions we want to model with simulation-based game theory have a large, uncertain number of participants. However, current game-theoretic analysis is limited to analyzing games with a fixed number of players. Several works in the literature conduct game-theoretic analysis for settings with a variable number of players, but in all cases either the number of players is a hyperparameter of the model, therefore requiring that each instance is analyzed separately, or the magnitude or range of player counts is relatively small, possibly due to limitations on available tools. Both cases demonstrate the need for analysis that spans a variable number of players.

The main contribution of this thesis is that we propose a new type of analysis which accommodates this uncertainty in the number of players. First, we define a variable-player game, where the number of players is endogenous to the game-theoretic model as opposed to a hyperparameter of the model. Next, we present an algorithm for identifying robust approximate Nash equilibrium and propose several measures of equilibrium robustness to quantify how well a given candidate equilibrium generalizes across a range of player counts. Our experiments demonstrate that our variable-player learning model outperforms the state-of-the-art fixed-player learning model [45] and even uses less data. Additionally, we experimentally showed that our iterative refinement algorithm with intermediate regret check and training data spread out for each (re)training iteration outperforms the model with all training data used up front as well as the model with resampling but no intermediate regret check. Our hope is that this technique will enable more meaningful predictions about behavior.

Future work for this project includes extending the learning technique to analyze variable-

player role-symmetric games as well as evaluating model performance on other classes of random games and on actual simulator data. Further, we would like to generalize this technique on other continuous parameters of the simulation environment. Finally, we hope to continue to explore the types of analysis that are enabled by variable-player learning, from investigating alternative notions of robustness to exploring entirely new solution concepts for variable-player games.

References

[1] David J. Abraham, Avrim Blum, and Tuomas Sandholm. Clearing algorithms for barter exchange markets: Enabling nationwide kidney exchanges. In *Proceedings of the 8th ACM Conference on Electronic Commerce*, page 295–304, New York, NY, USA, 2007. Association for Computing Machinery.

[2] Pranjal Awasthi and Tuomas Sandholm. Online stochastic optimization in the large: Application to kidney exchange. In *IJCAI*, volume 9, pages 405–411, 2009.

[3] Steven J. Brams. *Game Theory and Politics*. Free Press, 1975.

[4] George W Brown. Iterative solution of games by fictitious play. *Activity analysis of production and allocation*, 13(1):374–376, 1951.

[5] Noam Brown and Tuomas Sandholm. Superhuman ai for multiplayer poker. *Science*, 365(6456):885–890, 2019.

[6] Pranav Dandekar, Ashish Goel, Michael P. Wellman, and Bryce Wiedenbeck. Strategic formation of credit networks. *ACM Trans. Internet Techn.*, 15(1):3:1–3:41, 2015.

[7] Constantinos Daskalakis, Paul W. Goldberg, and Christos H. Papadimitriou. The complexity of computing a nash equilibrium. In *Proceedings of the Thirty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '06, page 71–78, New York, NY, USA, 2006. Association for Computing Machinery.

[8] John P Dickerson, Ariel D Procaccia, and Tuomas Sandholm. Optimizing kidney exchange with transplant chains: Theory and reality. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 711–718, 2012.

[9] John P Dickerson, Ariel D Procaccia, and Tuomas Sandholm. Failure-aware kidney exchange. In *Proceedings of the fourteenth ACM conference on Electronic commerce*, pages 323–340, 2013.

[10] Anthony Downs. *An Economic Theory of Democracy*. Harper, 1957.

[11] Fei Fang, Peter Stone, and Milind Tambe. When security games go green: Designing defender strategies to prevent poaching and illegal fishing. In *IJCAI*, pages 2589–2595, 2015.

[12] Shaheen Fatima. Sequential versus simultaneous auctions: A case study. In *Proceedings of the 8th International Conference on Electronic Commerce*, ICEC '06, page 82–91. Association for Computing Machinery, 2006.

[13] Sevan G. Ficici, David C. Parkes, and Avi Pfeffer. Earning and solving many-player games through a cluster-based representation. In *24th Conference on Uncertainty in Artificial Intelligence*, pages 187–195, 2008.

[14] Nicola Gatti and Marcello Restelli. Equilibrium approximation in simulation-based extensive-form games. volume 1, pages 199–206, 01 2011.

[15] Srihari Govindan and Robert Wilson. A global newton method to compute nash equilibria. *Journal of Economic Theory*, 110(1):65–86, 2003.

[16] Amy Greenwald and Justin Boyan. Bidding algorithms for simultaneous auctions. In *Proceedings of the 3rd ACM Conference on Electronic Commerce*, page 115–124. Association for Computing Machinery, 2001.

[17] Nobuyuki Hanaki and Juliette Rouchier. If you are so rich, why aren't you smart? In *Proceedings of the 2013 Winter Simulation Conference*, WSC '13, page 1731–1741. IEEE Press, 2013.

[18] Jean Honorio and Luis Ortiz. Learning the structure and parameters of large-population graphical games from behavioral data. *Journal of Machine Learning Research*, 16(1):1157–1210, January 2015.

[19] Albert Xin Jiang, Hau Chan, and Kevin Leyton-Brown. Resource graph games: A compact representation for games with structured strategy spaces. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, page 572–578. AAAI Press, 2017.

[20] Albert Xin Jiang, Kevin Leyton-Brown, and Navin Bhat. Action-graph games. volume 71, pages 141–173, 01 2011.

[21] Patrick R. Jordan, Christopher Kiekintveld, and Michael P. Wellman. Empirical game-theoretic analysis of the tac supply chain game. In *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems*. Association for Computing Machinery, 2007.

[22] Patrick R. Jordan, Michael P. Wellman, and Guha Balakrishnan. Strategy and mechanism lessons from the first ad auctions trading agent competition. In *Proceedings of the 11th ACM Conference on Electronic Commerce*, page 287–296. Association for Computing Machinery, 2010.

[23] Ehud Kalai and Eran Shmaya. Large repeated games with uncertain fundamentals i: Compressed equilibrium. Working paper, 2013.

[24] Debarun Kar, Benjamin Ford, Shahrzad Gholami, Fei Fang, Andrew Plumptre, Milind Tambe, Margaret Driciru, Fred Wanyama, Aggrey Rwetsiba, Mustapha Nsubaga, et al. Cloudy with a chance of poaching: Adversary behavior modeling and forecasting with real-world poaching data. 2017.

[25] Michael Kearns. *Graphical games*, chapter 7, pages 159–180. Cambridge University Press, 2007.

[26] Elias Koutsoupias and Christos Papadimitriou. Worst-case equilibria. In *STACS 99*, pages 404–413. Springer Berlin Heidelberg, 1999.

[27] Marc Lanctot, Vinicius Zambaldi, Audrunas Gruslys, Angeliki Lazaridou, Karl Tuyls, Julien Perolat, David Silver, and Thore Graepel. A unified game-theoretic approach to multiagent reinforcement learning. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2017.

[28] Carlton E. Lemke and J. T. Howson. Equilibrium points of bimatrix games. *Journal of the Society for Industrial and Applied Mathematics*, 12:413–423, 1964.

[29] Kevin Leyton-Brown and Yoav Shoham. *Essentials of Game Theory*. Morgan & Claypool Publishers, 2008.

[30] Zun Li and Michael P. Wellman. Structure learning for approximate solution of many-player games. In *AAAI*, pages 2119–2127. AAAI Press, 2020.

[31] Zun Li and Michael P. Wellman. Evolution strategies for approximate solution of bayesian games. In *AAAI*. AAAI Press, 2021.

[32] Mohammad Hossein Manshaei, Quanyan Zhu, Tansu Alpcan, Tamer Bacşar, and Jean-Pierre Hubaux. Game theory meets network security and privacy. *ACM Computing Surveys (CSUR)*, 45(3):1–39, 2013.

[33] John Nash. Equilibrium points in *n*-person games. *Proceedings of the National Academy of Sciences*, 36(1):48–49, 1950.

[34] John Nash. Non-cooperative games. *The Annals of Mathematics*, 52:286–295, 1951.

[35] John Von Neumann and Oskar Morgenstern. *Theory of games and economic behavior*. Princeton University Press, 1st edition, 1944.

[36] Noam Nisan and Amir Ronen. Algorithmic mechanism design. *Games and Economic Behavior*, 35(1):166–196, 2001.

[37] Panagiota N. Panagopoulou. Simple algorithmic techniques to approximate nash equilibria. In *Pan-Hellenic Conference on Informatics*, page 4–9, 2018.

[38] Patricio E. Petruzzi, Jeremy Pitt, and Dídac Busquets. Electronic social capital for self-organising multi-agent systems. *ACM Trans. Auton. Adapt. Syst.*, 12(3), September 2017.

[39] James Pita, Manish Jain, Fernando Ordónez, Christopher Portway, Milind Tambe, Craig Western, Praveen Paruchuri, and Sarit Kraus. Using game theory for los angeles airport security. *AI magazine*, 30(1):43–43, 2009.

[40] Tim Roughgarden and Éva Tardos. How bad is selfish routing? *Journal of the ACM*, 49(2):236–259, 2002.

[41] Sankardas Roy, Charles Ellis, Sajjan Shiva, Dipankar Dasgupta, Vivek Shandilya, and Qishi Wu. A survey of game theory as applied to network security. In *2010 43rd Hawaii International Conference on System Sciences*, pages 1–10. IEEE, 2010.

[42] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[43] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.

[44] John Maynard Smith. *Evolution and the Theory of Games*. Cambridge University Press, 1982.

[45] Sam Sokota, Caleb Ho, and Bryce Wiedenbeck. Learning deviation payoffs in simulation-based games. *AAAI*, 33(1):2173–2180, 2019.

[46] Ashish Sureka and Peter R. Wurman. Applying metaheuristic techniques to search the space of bidding strategies in combinatorial auctions. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, GECCO '05, page 2097–2103, New York, NY, USA, 2005. Association for Computing Machinery.

[47] Milind Tambe. *Security and game theory: algorithms, deployed systems, lessons learned*. Cambridge university press, 2011.

[48] P. D. Taylor and L. B. Jonker. Evolutionary stable strategies and game dynamics. volume 40, pages 145–156, 1978.

[49] David R.M. Thompson, Omer Lev, Kevin Leyton-Brown, and Jeffrey Rosenschein. Empirical analysis of plurality election equilibria. In *Proceedings of the 2013 International Conference on Autonomous Agents and Multi-Agent Systems*, AAMAS '13, page 391–398, 2013.

[50] Bruno Tuffin and Patrick Maillé. How many parallel tcp sessions to open: A pricing perspective. In *Performability Has its Price*, pages 2–12. Springer Berlin Heidelberg, 2006.

[51] John von Neumann. Zur theorie der gesellschaftsspiele. *Mathematische Annalen*, 100(1):295–320, 1928.

[52] Yevgeniy Vorobeychik, Michael Wellman, and Satinder Singh. Learning payoff functions in infinite games. *Machine Learning*, 67:145–168, 05 2007.

[53] Elaine Wah, Dylan Hurd, and Michael P. Wellman. Strategic market choice: Frequent call markets vs. continuous double auctions for fast and slow traders. *EAI Endorsed Trans. Serious Games*, 3(10):e1, 2016.

[54] Elaine Wah and Michael P. Wellman. Latency arbitrage in fragmented markets: A strategic agent-based analysis. volume 5, pages 69–93, 2016.

[55] Elaine Wah, Mason Wright, and Michael P. Wellman. Welfare effects of market making in continuous double auctions. *Journal of Artificial Intelligence Research*, 59:613–650, 2017.

[56] Hongbign Wang, Xin Chen, Qin Wu, Qi Yu, Xingguo Hu, Zibin Zheng, and Athman Bouguettaya. Integrating reinforcement learning with multi-agent techniques for adaptive service composition. *ACM Trans. Auton. Adapt. Syst.*, 12(2), May 2017.

[57] Michael P. Wellman. Methods for empirical game-theoretic analysis (extended abstract). In *Proceedings of the National Conference of Artificial Intelligence*, pages 1152–1155, 2006.

[58] Michael P Wellman. Putting the agent in agent-based modeling. *Autonomous Agents and Multi-Agent Systems*, 30(6):1175–1189, 2016.

[59] Michael P. Wellman, Amy Greenwald, Peter Stone, and Peter R. Wurman. The 2001 trading agent competition. In *Eighteenth National Conference on Artificial Intelligence*, page 935–941. American Association for Artificial Intelligence, 2002.

[60] Michael P. Wellman, Tae Hyung Kim, and Quang Duong. Analyzing incentives for protocol compliance in complex domains: A case study of introduction-based routing. *CoRR*, 2013.

[61] Michael P Wellman, Daniel M Reeves, Kevin M Lochner, Shih-Fen Cheng, and Rahul Suri. Approximate strategic reasoning through hierarchical reduction of large symmetric games. In *AAAI*, volume 5, pages 502–508, 2005.

[62] Bryce Wiedenbeck and Michael P Wellman. Scaling simulation-based game analysis through deviation-preserving reduction. In *AAMAS*, pages 931–938, 2012.

[63] Bryce Wiedenbeck, Fengjun Yang, and Michael P. Wellman. A regression approach for modeling games with many symmetric players. In *AAAI*, pages 1266–1273, 2018.

[64] Rong Yang, Benjamin J Ford, Milind Tambe, and Andrew Lemieux. Adaptive resource allocation for wildlife protection against illegal poachers. In *AAMAS*, pages 453–460, 2014.

[65] Brian Zhang and Tuomas Sandholm. Finding and certifying (near-)optimal strategies in black-box extensive-form games. In *Proceedings of the Thirty-Fifth AAAI Conference on Artificial Intelligence*. AAAI Press, 2021.