Modeling User-Behavior to Construct Counter Strategies

By
Gregory Hyde

A Thesis Submitted in
Partial Fulfillment of the
Requirements for the Degree of

Master of Science
Computer Science

At
The University of Wisconsin-Whitewater

August, 2019

Graduate Studies

The members of the Committee approve the thesis of

Gregory M. Hyde presented on 5/20/2019

_____

Dr. Hien Nguyen, Chair

_____

Dr. Jiazhen Zhou

_____

Dr. Lopamudra Mukherjee

Modeling User-Behavior to Construct Counter Strategies

By

Gregory Michael Hyde

The University of Wisconsin-Whitewater, 2019
Under the Supervision of Dr. Nguyen

We are working on the development of an adaptive learning framework addressing covariate shift, experienced in Behavioral Cloning (BC). BC user-modeling is a technique in which user-data, taken from observing a user's navigations, is used to train a neural network classifier. The classifier learns to map the user's actions to the state of the environment to reproduce their behavior in the future. The main challenge of this problem is the insufficiency of data which leads to under-performed models.
The motivation of this research is to provide an adaptive framework that encourages users to demonstrate their unobserved strategies when they otherwise would not. This work takes advantage of the inefficiency of BC user-modeling. By training a Reinforcement Learning (RL) agent to compete against each user-model, it will learn to

exploit the model's inefficiencies. This RL agent provides a means to drive the user in the next learning step. While this technique is designed for BC user-modeling, this targeted form of data collection provides a solution to amassing comprehensive user-datasets. To facilitate these goals, we develop a testbed called Turn-Based Adversarial Game (TAG) which addresses key problems in alternative testbeds for user-modeling. With our adaptive framework applied to TAG, we show how we can drive human subjects to demonstrate new strategies, organically. We have tested our approach with one real human user and plan for another two real users. The findings show a significant change in their strategy at each step.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Inferring a counter-strategy to a human strategy is a highly complex problem and the implications of achieving this feat are extraordinary. Though, if a counter-strategy is to be inferred, then a comprehensive understanding of what is being countered, is necessary. Specifically, to produce an artificial intelligence (AI) capable of perfectly countering a human subject, requires that the human's strategy be fully analyzed and understood. The policy that encapsulates their strategy can be thought of as their decision-making process [1]. Producing agents capable of capturing human-level decision-making is a highly sought after goal, as these agents must discover strategies in the form of cues, patterns, and mental heuristics [2]. It should be understood that a human's strategy does not suggest that the strategy is a globally optimal solution for an environment. Humans often perform behaviors that appear irrational but are still capable of producing high levels of expertise [3]. One approach to human-like behavior is through self-supervision [6, 8, 9, 14]. Self-supervised frameworks often overproduce in ability and need to be limited in their reaction times to achieve human-like behavior. For instance, OpenAI's Dota 2 agent, OpenAI Five, was hardcoded to limit its response time to 180ms [8], and DeepMind's Starcraft II agents were scaled back to 180 actions per minute [9]. This was done specifically for the purpose of

achieving a more human-like performance. Additionally, self-supervised frameworks learn through trial and error until an ideal strategy is formed that maximizes its reward function. However, as stated previously, optimal solutions aren't necessarily indicative of human-like behavior. To achieve the apparent irrational characteristic of human decision-making, many turn to learning frameworks that benefit from observing human behavior [1,7, 20, 22, 23, 24]. These frameworks train their models on a human data, collected as they navigate an environment. From this data, the human's strategy is inferred and replicated. However, for the model to achieve behavior identical to the human, the human's full strategy must be embedded within the training data. This framework seeks to achieve a fully comprehensive user dataset by reducing redundant data and leading the human into scenarios that were previously unobserved. Data sets that lack a complete view of the user strategy suffer from scarcity, and this translates into the user-models. Only by filling these gaps can we observe the human's full strategy and model their behavior.

To facilitate these goals, an adaptive learning environment, Turn-Based Adversarial Game (TAG), is proposed. TAG is a dyadic turn-based game pitting two opposing forces against each other. While the game rules are intuitive to pick up, it provides an environment rich with strategy and intricacy. Within TAG, a player can attack, defend and occupy towers with the overall objective of removing the enemy from the game board. The novelty of this testbed is that it was modularly designed such that various AI, controlling either the Player 1 or Player 2 positions, can be readily implemented, making it useful for adversarial learning and imitation learning alike. Additionally, the testbed comes with a default greedy AI for a human subject to play against, as well as human vs. human functionality. TAG

allows for autonomous data collection and data processing, reducing error and expediting these, otherwise, time consuming processes. TAG also has many mutable game features for scenario construction, providing variety to game types and scenario difficulties. Lastly, TAG offers a solution to the above scarcity problem, albeit indirectly. Through controlling the adversary role in TAG, this framework provides a means to drive the human to new game scenarios, provoking them to demonstrate a strategy they would otherwise not have demonstrated. Control of the adversary role is critical to the success of framework, as both the human and the adversary maintain half of the control in the direction of each game.

The qualitative and quantitative results from this work provide a great deal of insight to the adaptive prowess of humans in dynamic environments. While a complete counter-strategy was not accomplished, the effectiveness of driving the user experiences has proven to be paramount. This framework accrues data from the user gradually, and upon each iteration, an adaptive AI is successfully prescribed to take advantage of what additional user-strategy has been learned. Ideally, this process would continue until the human is entirely countered, but this was not achieved. However, the successes of this work are many. First, this framework is capable of training an adversary with enough driving power to demand the user adjust their strategy in order to be victorious. This functionality can provide a prescriptive technique in diagnosing potential flaws in a human's strategy. While this may not be crucial within a simple adversarial game, this is an important question pertaining to some of my previous work in identifying insufficiencies within the decision-making style of commander's in naval warfare [1]. This framework also poses as a powerful tool in data collection. The novelty of this approach is that the user-models

constructed on user data are primitive in nature, so that they more obviously expose the limitations of what can be inferred by the current dataset. These limitations then provide an objective destination for what scenarios we would like the user to experience next. Driving the user's experience to achieve a more comprehensive data set, means that more sophisticated user-models can be constructed off of it. Lastly, this framework provides a solution to the covariate shift problem encountered in training behavioral cloning classifiers on limited data sets without incorporating any synthetic user data. This is an important problem to solve as much of the literature on user-modeling has moved to alternative techniques due to the inadequacies of behavioral cloning. However, the implications of successfully addressing this problem for behavioral cloning can lead to more concrete datasets for all user-models. Even though these modern approaches to user-modeling suffer less from covariate shift, lacking the necessary descriptive data produces less robust models in general.

The framework being presented is an ambitious one. It requires the construction of an entirely new testbed, TAG, as well as the incorporation of two flavors of AI, the BC user-model and a RL agent. As such, a comprehensive explanation of what TAG is and why it is necessary will be provided, followed by the experimental setup for the BC user-model and the RL agent. The adaptive learning process being presented in this framework will then be described, and how the three components mentioned above will interact seamlessly. Lastly, the significant findings of this work will be presented. Results are collected by observing the human subject as they navigate TAG, along each step in the learning process. These results show significance in the diversity of data collected on the users, and how

they demonstrate their strategies. As the objective of this work is produce counter-strategies that drive the human subjects to display their unobserved strategies it is important that each new series of data tells us something new about the user. Only with a complete view of the human's strategy will covariate shift in BC user-modeling be addressed.

Chapter 2

Background

If a counter strategy is to be derived from a user's model, then it is imperative that the user-model be as comprehensive as possible. Not only must it carry the intuition of the user, but it should perform naturalistically rather than robotically. Therefore, this work is heavily inspired by the natural processes in which a human makes decisions. Decision-making has been defined as "the learned habitual response pattern exhibited by an individual when confronted with a decision situation" [4]. In an attempt to understand such a complex process, various techniques have been employed by researchers to produce (or reproduce) naturalistic decision-making (NDM). NDM incorporates human heuristics, domain knowledge, and experience, in real-world environments characterized by ill-structured goals, uncertainty, time-constraints, and high stakes [5]. One technique, Inverse-Reinforcement Learning (IRL), was proposed by Russell [33] as a means to infer an agent's reward function from previous behavior. IRLs counterpart, Reinforcement Learning (RL) also uses a reward function but optimizes its reward function via experience [6], rather than from observation. Another technique, that carries a nuance difference from IRL, is Behavioral Cloning (BC). Rather than reproducing an agents behavior from a reward

function, an agent receives observed states and actions as training data, which is then used to train a classifier or some other function approximator [7]. Regardless of approach, the scientific world has been enthralled with producing an agent that encapsulates NDM.

A substantial portion of the framework described in this thesis hinders on the ability to effectively model a user's behavior; which includes being faithful to the user's NDM properties. Without an accurate representation of the user, any techniques employed to derive a counter-strategy would therefore be made useless. To construct the user's model, we use Markov Chain neural networks (MCNN) and reinforcement learning (RL). A MCNN is a variation of a Feedforward neural network (FNN), extended to allow a probabilistic characteristic to an otherwise deterministic model [12]. The MCNN model shares an intimate relationship with the RL agent as the RL agent is responsible for determining the flaws of the MCNN through a process called Adversarial Learning (AL). More specifically, the RL agent discussed in this thesis is a network trained with OpenAI's approach as a Deep Q-Network (DQN) [8] and used to compete against the MCNN.

2.1. Feedforward Neural Networks (FNN)

A FNN is the first type of artificial neural network designed and is composed up neurons – its basic building block [10]. Neurons are connected to other neurons via edges and are used to form complex networks capable of approximating complicated functions. Each neuron contains a weight, often times referred to as a threshold, that is altered throughout the training process. A FNNs most notable feature is that it is completely devoid of cycles, whereas a recurrent neural network allows for this. All FNNs contain an input

and an output layer, but a multi-layer FNN is one that contains any number of hidden layers (Figure 1.1 depicts a single hidden layer FNN). In each layer exists a set of nodes, or neurons. In Figure 1.1 the input layer consists of three nodes, but the number changes for every problem. Every input used in classification would represent a single node in the input layer. To create a successful classifier there isn't a perfect rule for the number of nodes needed in each hidden layer, however more general rules (such as having the hidden node count be anywhere from the input count to twice that) do exist. Like the input count for a FNN, the output count should be an appropriate size to present its creator with the information necessary to classify (in many classification problems, only one output is needed – the classification). The FNNs used in this framework for approximating user-behavior are designed for simplicity. For many domains smaller, more simple, networks can prove to be much more beneficial than larger, more complicated, ones [12]. Smaller networks require less memory to store and are more easily implemented. In addition, training smaller networks requires less computations, and therefore training is faster. Bebis and Georgiooulos also argued that bigger networks tend to require larger numbers of training examples to achieve good performance. Further, Hornik and Stinchombe [11] provided a general theoretical result to show that a single hidden layer FNN can approximate virtually any function.

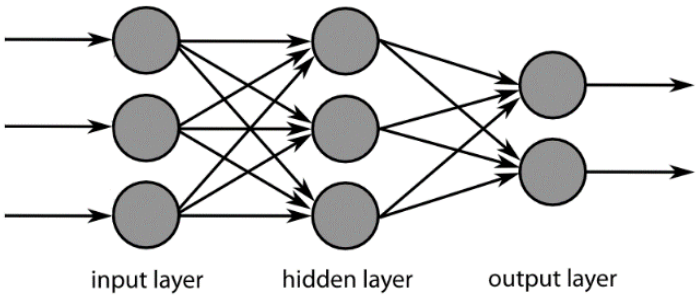

Figure 1.1: Simple FNN with one hidden layer

2.2. Markov Chain Neural Networks (MCNN)

A MCNN model is one that incorporates the characteristics of a Markov Chain (MC) [13]. A MC is a stochastic model that describes a sequence of possible events under the assumption that any future states depend only on that current state. The memoryless features of a MC is known as the Markov Property. An MC can be described as a graph $G = (V, E, T)$. V is a set of vertices or states that can be transitioned to, and E is a set of edges that connect the vertices in V. $T_{i,j}$ contains the probabilty of using $E_{i,j}$ given V_i. To produce an MCNN, Awiszus and Rosenhahn artificially alter training data for a FNN to simulate randomness. Imagine a simple MC model with three nodes, s_1, s_2 and s_3, and two directed edges, e_1 and e_2 connecting s_1 to s_2 and s_1 to s_3. The probabilty distribution of all outgoing edges from any node must be equivalent to one, so if e_1 and e_2 share an equal variance of the distribution then e_1 = 0.5 and e_2 = 0.5. To adjust the training data to share in this property, the state s_1 (along with all other states containing transitions) would be duplicated by a factor of 10 (depending on desired specificity), and a value r [0-1] would be prepended to each record of that input. The r value increases on each iteration of a duplicate state from 0 all the way to 1. Depending on the probability distribution of s_1 (in

this case 50/50) the ideal output for each record should reflect according to the r value. This can be visualized in Figure 2.1, where s_1 is duplicated 100 times (with corresponding r value granularity). Once an MCNN has completed its training, to simulate randomness a random number must be generated and prepended to the state information. Training the MCNN on the altered training data provides a neural network that simulates the probabilistic property of a Markov chain.

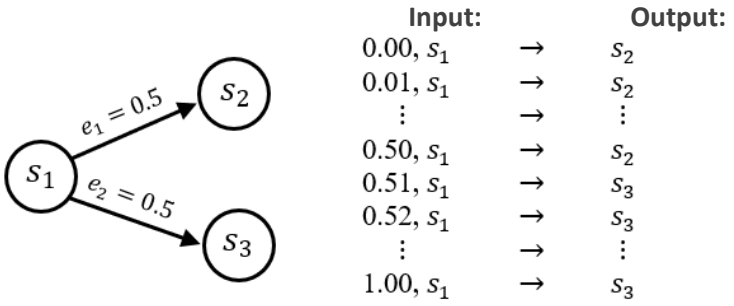

Figure 2.1: Altered training data for MCNN

2.3. Reinforcement Learning (RL) and Deep Q-Networks (DQN)

RL is a type of learning based on experiences and the positive and negative associations made from those experiences. In the Machine Learning community (ML), RL is a means of mapping states with actions via a reward function [6]. This type of learning has found its way into its own category of ML, as it is neither supervised nor unsupervised learning. In RL, an agent explores an uncharted environment with nothing but a reward and value function, and a set of possible actions. The reward function infers what actions

should be taken, given a state, to provide the agent the highest, immediate reward ($R(S, A)$). A value function is less short sighted in that it learns trajectories that maximize the agents rewards long term. Initially, the agent tries actions at random, but as it infers more optimal strategies, the effectiveness of the agent increases. The transitioning of states and actions is referred to as a trajectory and can be denoted as: $S_0, A_0, R_1, S_1, A_1, R_2, S_2, \dots, S_{n-1}, A_{n-1}, R_n, S_n$, where S_i is a given state, A_i is the action taken at S_i, and R_i is the reward for taking action A_i. The learning process of an agent can be seen in Figure 2.2.

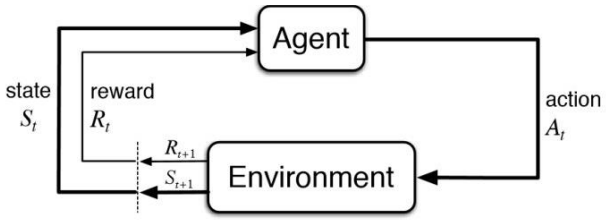

Figure 2.2: The training loop for a RL agent

Ultimately the RL agent will construct a policy, or a mapping of actions to states for some reward function [6].

DQN is a type of RL which is used to estimate an optimal policy in advance by determining Q-values for each state [15], where Q-values are the quality assessment for executing an action, on a given state. The ultimate goal of Q-learning is to maximize these Q-values. This portion of our framework extends upon OpenAI's DQN framework and their Gym environment library. OpenAI used Q-learning in addition to stochastic gradient descent in their DQN framework to obtain optimal policies in various Atari games, with many of them outperforming expert-level human players [14].

2.4. Adversarial Learning (AL)

AL is an approach in ML in which two opposing models (the target model, and the discriminator) continuously learn as they critique and expose vulnerabilities in the other [16]. A considerable amount of work has been put into AL on highlighting potential flaws in classifiers for cyber security applications [17, 18]. A general approach to AL is to add small mutations to training data that lead a network to misclassify [19]. Both the objective and discriminator model learn dynamically to counter the other. While this certainly has its uses in intrusion detection classifiers, this technique has been used to expand on models with limited data pools. Ho and Ermon [20] use AL to allow their model to overcome covariate shift (albeit in a self-supervised way). Specifically, they use sampled trajectories from the user to form a policy. This policy would initially be suffering from covariate shift, however using RL the model self supervises its way to fill out its covariance matrix. The discriminator's role in this case is to determine whether synthetic trajectories produced by the target model are akin to the original data collected on the user. Depending on its score the model is adjusted. In this work we too, will be dealing with incomplete models of the user. These give way to vulnerabilities, and therefore creates a perfect environment for adversarial learning. In the framework being proposed we use our Turn-based Adversarial Game (TAG), as a way for the user-model and RL agent to compete against each other. Once a RL agent learns strategies to maximize its reward function against the user-model, the agent can then be introduced to the user (whose data the user-model was trained off of) to push the user into unexplored areas, effectively providing a complete view of their behavior.

Chapter 3

Related Work

Within the IL community, many have long been inspired by the cognitive processes in which a human behaves, and so many computational models have been constructed to capture this process [1, 20, 22]. The two sub-communities of IL (IRL and BC) have been carefully explored as a means to construct agents capable of producing NDM [20, 22], or evaluate and prescribe expert behavior. Santos [1] captures a Commander's decision-making style by leveraging their computational model, the Double Transition Model (DTM), to construct a dynamic Markov Decision Process (dMDP). The dMDP's structure is initially void of reward values, but through IRL rewards are inferred. These models are then analyzed to infer decision making styles based on the models reward values, number of unique states compared to actions, node incoming and outgoing cardinality, and node density. The decision making-styles (rational, intuitive, spontaneous, learner and avoidant) were validated via a comprehensive phycological examination [21]. While this study is less interested in replicating behavior, it offers a unique computational model that provides insight to preferred decision-making styles and is used as a quantifiable measurement of decision making to help Commander's refine their craft.

Another computational model, Generative Adversarial Imitation Learning (GAIL) was proposed by Ho and Ermon [20] as a means to tackle two key issues in the IL; covariate shift and the computational strain of IRL. In Ho and Ermon's work, their model learns a cost function and prioritizes entire trajectories over others (rather than fitting single-timestep decision) to avoid the compounding error experienced in BC [20]. Trajectories are composed of a sequence of individual transitions between states. The transitions inside of each trajectory form a 3-tuple (s_i, a_j, s_i') where s_i and $s_i' \in S$ are states and $a_j \in A$ are actions. Combing these elements in a 3-tuple represents a transitioning between states s_i and s_i' brought about by action a_j. IRL focuses on learning the complete trajectory rather than learning each 3-tuple individual, and as a result has had many successes in various testbeds or real life applications [23, 24, 25]. However, the cost of learning entire trajectories is that IRL algorithms are computationally expensive procedures to run and therefore are difficult to scale to larger environments. GAIL was a means to solve these problems in that their framework bypasses the intermediate IRL step by inferring a cost function learned by RL[20]. Their work was highly successful and has been used to produce near expert level performances in the MuJoCo environments [36]. It should be noted that GAIL often outperformed the human experts behavior. This is due to the RL step, which gives way to self-supervision. Self-supervision provides a human-like model, but deviations from the human's true behavior are unavoidable.

In Response to the critiques of BC, Torabi [22] provided a BC learning framework that incorporates two phases of learning; a pre-demonstration phase in where an agent learns a policy in a self-supervised fashion, and a post-demonstration phase where the agent

observes an experts behavior and refines the learned pre-demonstration policy. Further, they extend the MDP structure by inversing the probability function$P(s_{i+1}|\ s_i, a)$, where the probability, P, of transitioning to state s_{i+1} given an action, a, and state s_{i+1}, is changed to $P(a|\ s_i, s_{i+1})$. They argue that this is more akin to typical human observation in that the human observer typically isn't provided the perfect information of which action was performed in a state. While their model compares quite effectively (even when provided a limited set of expert trajectories) to GAIL and other state-of-the-art models, the issues of covariate shift is not addressed, rather the information missing in the covariance matrix is first filled by the pre-processing step, and then manipulated by the expert trajectories. Again, like GAIL this is a form of self-supervision, which suffers similar deviations from the human's true behavior.

In order to best represent the expert, the data used to fill the covariance matrix should not only be as complete as possible but be entirely made up of user data. Even though the models proposed by Torabi et al. and Ho and Ermon have a fully fleshed out covariance matrix, there is the risk of misrepresenting the expert, as parts are synthetically filled by self-supervised agents. In this thesis, all data used to construct a user-model is real data captured by observing the expert. To address the covariate shift in this framework, while also avoiding the synthetic production of user data, the user is instead driven into game scenarios they had not previously experienced, in order to provided organic responses. This process allows the expert to more evenly disperse the information in the covariance matrix promoting data quality, rather than requiring unrealistic amounts of data to fill it, or self-supervision. In addition, we also provide a system of neural networks for our BC classifier

that captures the expert actions with regard to the context of each game state. Capturing the distinction between detail and context helps the classifier to learn the more intricate relationships of their behavior in a manner typical to human decision making.

Chapter 4

Approach

In order to construct a counter-strategy there must be a well observed user-strategy. This requires that we address the issue of covariate shift. Failing to do so leads to poorly viewed user-strategy and therefore an improper counter-strategy. To bring these goals to fruition two requirements must be met for this framework to be applicable. (1) The testbed in which the counter-strategy is derived from must be an adversarial one where the adversarial position is accessible by an AI, and that AI must be mutable. Having an adversary is an intuitive requirement when considering a counter-strategy as learning a counter-strategy suggests there are two competing entities. Being able to mutate the adversary, while less intuitive, is also needed as adjusting the adversarial strategy is the means to drive the user to unique trajectories. (2) The testbed must also be able to produce user data, in the form of states and actions, or another form of observation must be present (whether it be documented by hand, or a testbed be extend to autonomously collect the data). Previous work within various IRL testbeds where autonomous data collection wasn't available has been one of the primary motivators to support this feature [1]. Therefore, to facilitate the needs of this framework an entirely new testbed will be provided as part of

the contribution this work. The testbed is a turn-based adversarial game, appropriately named TAG.

Modeling a user's behavior with BC trained on limited data exaggerates the covariate shift issue. This is an important premise as the implications of such creates a user-model that underperforms with game states too dissimilar from what it has seen before. Thus, an RL-agent that is rewarded for beating the user-model as quickly as possible will learn to take advantage of the user-model's inefficiencies. For these two AIs to be able to compete, they must be put into their respective roles within the TAG environment. The learning process of the RL agent and the user-model is an iterative one. TAG must be able to swap out multiple iterations of both of these models throughout the process. First, the user forms a strategy against the TAG default AI (greedy in nature and deterministic). The data is collected and used to construct the first BC user-model. From there the BC user-model is placed into the TAG environment in the user's player position, and an RL agent in the adversary position. The RL agent learns a policy that exposes the weaknesses in the BC user-model. Once the RL agent has effectively trained against the user's BC user-model, the human will then compete against the RL agent. Being that the RL agents has trained to take advantage of their BC user-model (i.e. the areas where information is lacking), when the human competes against the RL agent it will drive them into these scenarios, allowing us to capture their genuine responses. These new trajectories will be added to the previous pool of user-data, to construct a more complete BC user-model. This process continues until the covariance matrix is well enough defined to produce an accurate representation of the human.

The rest of this section will be presented as follows. First, an in-depth explanation of what TAG is, and why it is necessary, will be provided. Furthermore, the various features that TAG offers will outlined, as well as the surplus utility TAG offers that may be useful to other learning models. Second, the learning process of the framework will be presented in detail, and in of itself can be broken into four parts: (1) the data collection process, (2) the alteration of data into training data, (3) training the BC user-model, and (4) training the RL agent. To follow the flow of how the framework learns, these are presented in the order in which learning takes place. Lastly, the theoretical significance of bringing together a BC user-model and an RL agent as adversaries will be rationalized.

4.1. Turn-Based Adversarial Game (TAG) Mechanics

TAG is a turn-based strategy game, written in the C# programming language, that pits a human player against an AI opponent, with the goal of removing the enemy from the board. The game is inspired by battlefield scenarios where commanders send troops to either attack or defend different locations. The board contains 14 towers $\{A, B, \dots N\}$ with different unit capacities and unit regeneration rates, as shown in Figure 4.1. The varying tower capacities and regeneration rates were designed such that there was variability in playstyle between different users, and stalemate scenarios could be avoided. At the beginning of the game, both the human player and the AI opponent are provided an initial tower at opposing corners. Player 1 (Human - blue), by default, starts with 25 units, and Player 2 (AI - red) starts with 31 units. This unit advantage is to compensate Player 2 for having the disadvantage of going second. Also giving Player 2 an advantage of 6 additional units allows them to gain two towers immediately within two turns, whereas Player 1 can

only gain at most two new towers by their third turn. All other towers start neutral (white) and have an initial cost of 5 units to acquire. Giving an initial cost to take a tower, even if the cost is as small as 5 units, incorporates another working piece to the game mechanics that the user must process in their strategy. On each player's turn, units may be moved from any tower they own, to any other game tower (given a few game restrictions to be outlined). Player 1 and Player 2 take turns acquiring towers and engaging in combat until a player's units fall to 0.

Three types of unit movement exist in TAG; movement of units to an enemy tower (attacking), movement to a neutral tower (exploring), and movement to a friendly tower (reinforcing). The units that leave from the sending tower to be received by the receiving tower is equal to half of the sending tower's units (rounded up).

- **Attacking**: When units arrive at an enemy tower, combat ensues. The victor is determined by taking the difference between the number of units for both parties. The player with more units either maintains or gains the tower. The remaining units are equal to the absolute value of the difference of units.
- **Exploring**: When units arrive at a neutral tower, the initial cost of 5 units is taken into account before the tower is acquired. If less than 5 units are sent, the cost is reduced by the units, but the tower will remain neutral. If more than 5 units are sent, the player will acquire the tower and the remaining units will populate it.
- **Reinforcing**: When units arrive at a friendly tower, the arriving units are simply added to the present units. This allows for the bolstering of a tower, whether it be for anticipation of an enemy attack, or to prepare for a future attack.

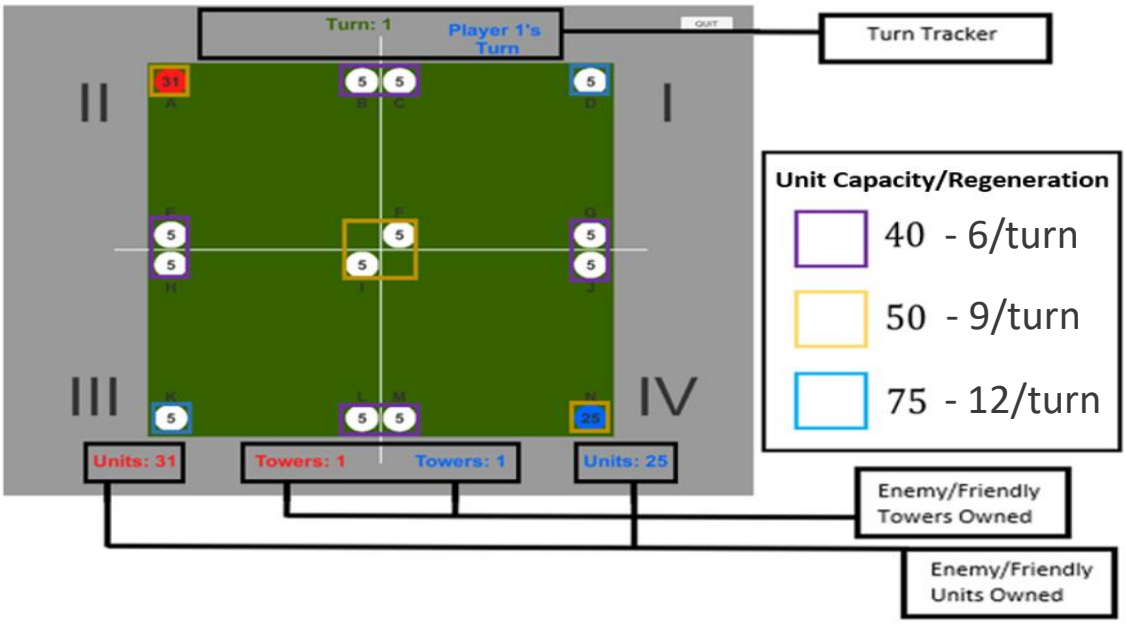

Figure 4.1: TAG gameboard and configuration

Regardless of which type of unit movement is happening, all unit movement follows restrictions on how quickly units can arrive at another tower. Similar to the types of unit movements, the restrictions to these movements fall into three categories; close movement, far movement, and adjacent movement.

- **Close movement:** This movement type describes movement between towers of the same quadrant (e.g. N → M). This movement takes 1 turn for the units to arrive. If Player 1 were to choose this type of an action, then Player 2 would be allowed a turn before Player 1's units arrive at their destination.
- **Far movement:** This movement type describes movement between towers of differing quadrants (e.g. N→ K). This movement takes 2 turns for the units to arrive. If Player 1 were to choose this type of an action, then Player 2 would be

allowed two turns before Player 1's units arrive at their destination. Player 1 still takes their intermediate turn.

- **Adjacent movement:** This movement type describes movement between adjacent towers (F and I, and all purple towers in Figure 4.1). This movement happens immediately regardless of the quadrant properties of the towers. If Player 1 were to choose this type of an action, Player 2 wouldn't have any turns before Player 1's units arrive at their destination.

To allow growth in the game, at the beginning of each player's turn, every tower that is under their control will generate units according the tower's regeneration rate seen in Figure 4.1. It is important to note that the sequence of events for a game turn are as follows: (1) moving units arrive at a tower if they were set to reach their destinations on this turn, (2) towers controlled by the player whose turn it is, will generate units according to their regeneration rate, and (3) the player will select an action. This then is immediately followed by the turn being passed to the opposite player. This means that if units arriving on the player's turn happen to take control of a tower, that tower will also benefit from the unit generation. The action space for a player to choose from is at most 183 different actions. The number of actions that can be performed by a player at any turn is determined by the number of towers under their possession multiplied by all other towers, excluding itself (units in a tower can't be sent to the same tower). This leaves the player with, at most, 14 * 13 actions, or 182. The additional action that makes 183 is the ability for the player to skip their turn, providing each owned tower with an immediate generation of 5 additional units before their turn ends (generation on their following turn is applied as normal). The

number of actions a player can perform at any moment, then is $13n + 1$, where n is the number of towers they have. If a player doesn't own any towers, then they must skip there turn.

Finally, the last mechanic to be described in TAG is the limitation of unit movement given by the tower capacities. Typically, half of the units from the sending tower are sent to the receiving tower, but on every action a calculation is performed to ensure that a tower won't receive more units than its capacity allows. In these situations, units that leave from the sending tower will be restricted, or if no units can be sent, the user will be prompted to pick another action. As such, this can also limit the number of actions a player can perform on their turn. For actions moving units to an enemy tower, the maximum units that can be sent would be:

$$maxUnits = tCapacity + eUnits + eMoving1 + eMoving2 - fMoving1 \pm (tRegen * turns)$$

This formula considers all possible factors altering a tower's unit count. *tCapacity* is the receiving tower's specific capacity and *eUnits* indicates the number of enemy units already residing. Units on the move (that would arrive before the desired action) also need to be measured. *eMoving1* and *eMoving2* indicate enemy trips, and *fMoving1* indicates a friendly trip. In this case all enemy unit information is added to the maximum number of units that can be sent, and friendly information decrease this number. Lastly, *tRegen* accounts for tower growth for the number of turns the desired action would take. Additional, this must account for the fact that the tower could change ownership before units arrive. Similarly, for reinforcing a tower, this calculation is performed:

$$maxUnits = tCapacity - fUnits + eMoving1 + eMoving2 - fMoving1 \pm (tRegen * turns)$$

Finally, a calculation for exploring:

$$maxUnits = tCapacity + nUnits + eMoving1 + eMoving2 - fMoving1 \pm (tRegen * turns)$$

In this calculation, *nUnits* is equal to the cost of acquiring a neutral tower. *tRegen* is only considered in circumstances where the tower will be owned before these units arrive.

4.2. Why Tag?

TAG was designed to overcome a number of problems faced in the IL community, and offer various features to streamline the processes in IL. First, I will demonstrate the key features. These features are necessary to the success of this thesis and can be used to explain why TAG is necessary. Secondly, I will demonstrate the additional features that were constructed to enhance the IL experience for future research. While these aren't necessary to this project, the goal was that TAG might find use in other learning models, and that it be an asset to the IL community. The former features can be broken into the categories: automated data collection, replaceable enemy AIs, and the ability to plug in user-models. The latter features include: feature generalization, scenario reconstruction, and adjustable in game values.

4.2.1. TAG Features Needed for this Framework

Autonomous data collection is an extremely useful tool in to the IL community in that, all techniques of IL include some sort of learning via observed expert behavior. Data is written to a simple text file where every record contains a game state followed by the action taken by the user at that game state. All entries are comma separated, and a header for each column is provided. While not every feature is used in this framework, all game data is

recorded. Having this data automatically recorded expedites the process of collecting expert behavior, drastically reducing the time that would be spent recording by hand. Removing the human recording factor also reduces the probability of error. Replacing of the enemy AI, and the ability to plug in user-models go hand in hand. Throughout the learning process iterations of the enemy AI and the user-model will be created. As one is constructed, it will be used to drive the other. Data will be collected to construct the user-model and this user-model needs to be plugged into the TAG environment to allow training of the RL agent against the user-model. To make this possible the RL agent also needs to be plugged into the adversarial position.

4.2.2. TAG Features Relevant to Machine Learning Community

BC classifiers, and especially IRL algorithms, are very computationally taxing processes. Given the nature of how these frameworks learn, the dimensionality of the data is a primary concern. Feature selection is the process of reducing the dimensionality of a dataset while still maintaining the features that best represent the data. Often times this process includes some sort of generalization of the data or a pruning of features that are deemed to carry the least amount of variance for a problem. Regardless, of which is used, feature selection is a primary practice in the IL community. Not only can feature selection lead to faster run times, but it can also produce more capable classifiers and user-models. As an example, consider training a classifier on human gene data. The number of genes (or features) can often be larger than the sample size of many datasets, as there are a tremendous number of genes [34]. Therefore, being able to select the genes that carry the most variance for the classifier reduces the noise in the dataset and leads to a more effective

classifier. Written discretely, the TAG environment can produce a tremendous number of unique states. However, the gameboard was organized in such a way that it can easily be broken down into quadrants. Rather than comparing each tower individually, the group of towers in the quadrant can be represented as a single force, and each players presence can be measured. Similarly, rather than discretely documenting a tower's units, they can be put into bins to represent a weak, medium or strong ownership for a player. Various generalization approaches and their respective state spaces are shown in Table 4.1. The desire to create a testbed that allows for the generalization of the state space was motivated by previous work in IRL on commander's decision making [1].

Table 4.1: TAG state space descriptions

Number	Description	Number of States
1	Consider all possible tower unit combinations, together with ownership descriptions (0-none, 1- $player_1$, 2 – $player_2$)	$= 3^{14} * 50^4 * 75^2 * 40^8 \approx 1.1 * 10^{30}$
2	Divide tower units into 3 categories: • 75 towers: 1 – (0, 25); 2 – (26, 50); 3 – (51, 75) • 50 towers: 1 – (0, 16); 2 – (17, 33); 3 – (34, 50) • 40 towers: 1 – (0, 13); 2 – (14, 27); 3 – (28 – 40)	$= 3^{14^2} \approx 2.2 * 10^{13}$
3	Only consider tower ownership	$= 3^{14} \approx 4.78 * 10^6$
4	Consider player presence in each quadrant, where the presence is taking total units allowed in that quadrant split into three categories: • Q1 & Q3 – 205 total units: 1 – (0, 68); 2 – (69, 137); 3 – (138, 205) • Q2 & Q4 – 130 total units: 1 – (0, 43); 2 – (44, 87); 3 – (88, 130)	$= 3^8 = 6561$

All of the game values within the TAG environment are mutable, allowing for the construction of varying game scenarios. Changing game settings has many benefits to it. Currently, in most environments a human is limited to a preset number of scenarios, designed by the game developers. Changing the game settings by altering tower regeneration rates and capacities can lead to different game flows and requires the user to be dynamic in developing varying strategies. For example, bringing all tower generation rates to 5 and all tower capacities to 50 leads to consistent stalemates, which is potentially

valuable to areas of research in which a user is making decisions while stressed or agitated. Not only can the game settings be altered to change the mechanics of the game, like tower capacities and regeneration rates, but the preset number of units and towers owned by both players can be altered. This could potentially be a valuable tool as it allows us to present a game state, that the individual has already experienced before. There are a number of reasons scenario reconstruction can be valuable to the community. Game states that heavily change the flow of the battle can be further explored by an individual to determine alternate outcomes, or these game states can be presented to the user periodically through a testing scenario to see if their decisions change over time. Even though these features have been added to TAG, the value of these contributions are left open for future implementation of the TAG environment.

4.3. Learning Process

The learning process of this framework is iterative. All data used to construct the user-model is entirely made up of user data and includes no self-supervised functionality in the creation of it, unlike some of the related work referenced previously. The process (as shown in Figure 4.2) can be described in two parts, an *observing phase* (Parts 1, 2 and 6) and a *learning phase* (Parts 3, 4 and 5). In the observing phase, the human plays a number of games against the adversary. (1) For the very first observing phase the human plays against TAG's default AI. TAG collects the user's data as they play the game (~10 games worth of data, each taking ~3 minutes to complete) where it is then altered into training data (2) for the BC classifier, thus ending our observing phase. The learning phase takes the training data provided by the observing phase and constructs the user-model (3). The user-model is

a system of two MCNNs using the BC user-modeling approach (mapping of states to actions). The model is inherently flawed due to its lack of sufficient data. The result is a model trained on a partial view of the user's strategy. If pushed too far from the narrow scope it knows, it underperforms. With the user-model created, knowing its shortcomings, we replace the human with the AI approximation of its behavior (4). The adversary for the user-model in the learning phase is the RL agent. The RL agent's objective is to infer a counter strategy to the user-model's behavior (5) by competing against it in a self-supervised manor. This self-supervision is only a means to learn the model's shortcomings, not influence the model in anyway. Once the RL agent constructs a policy imbedding the user-models counter strategy, it replaces the greedy AI the human plays against, and the process is repeated. Upon each iteration more is learning about the human, thus compiling a comprehensive user-dataset.

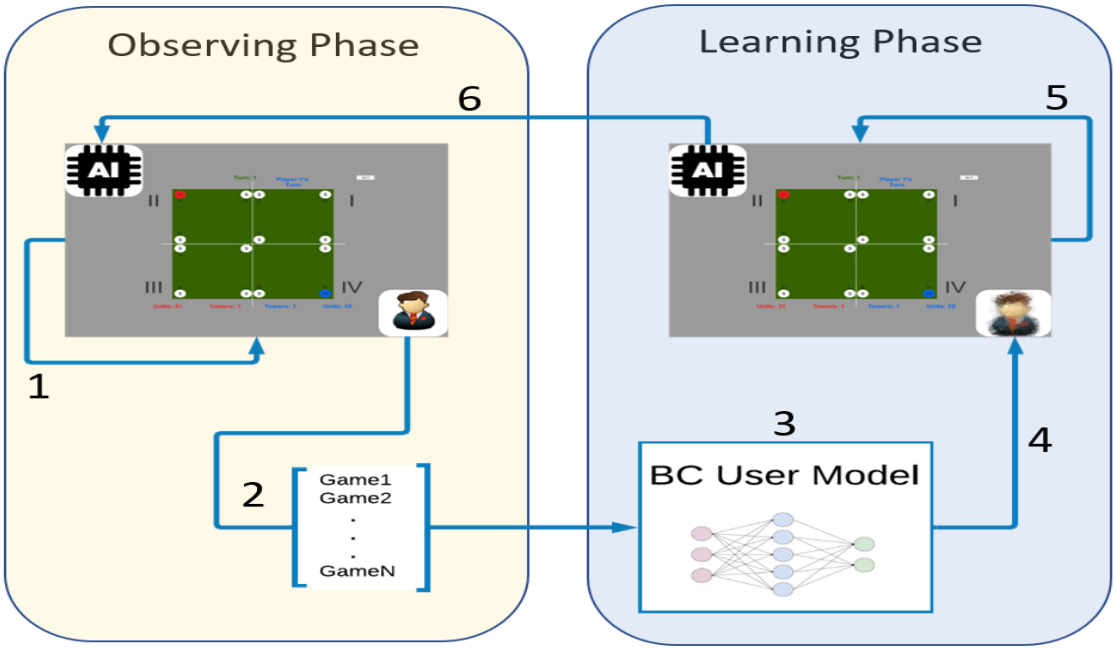

Figure 4.2: Learning process being demonstrated in this framework

Ironically, it is the very flaw of using BC to approximate user-behavior that leads to the success of this project. In this framework the RL agent is rewarded for not just besting the user-model but besting it as quickly as possible. Intuitively, an agent whose job it is to learn to beat its competitor would take advantages of the inefficiencies of its competitor. While the RL agent can infer flaws in the user-model, it should be understood that these may not necessarily be flaws in the human's behavior, but simply a by-product of the relevant information not being observed. However, the goal is to then have the RL agent drive the human into the scenarios where this information can be observed. Ideally, this process would continue until convergence. For there to be convergence there needs to be a clear stopping point, and as such multiple will be explored in this framework. Generally speaking however, convergence means that the data collected accurately represents the user and the model's covariate shift issue has been addressed. Figure 4.3 shows the theoretical result of implementing this framework. Although this example is simplified to a two dimensional space, the goal is to spread the data collected over the human's entire function, rather than collect duplicate data. Without an adaptive AI driving the user to demonstrate new strategies, there is no incentive for the user to adjust their already winning strategy. Therefore, the continued collection of redundant data tells us nothing new about the user. By adjusting the AI, the human is driven into unique game scenarios, providing a more comprehensive view of what the human's function is. In Figure 4.3 one can intuitively understand why this is crucial, as the overall function tends down. This is only visible when the data collected is dispersed and more comprehensive. With only the redundant data being collected, one might mistakenly expect the function to continue to tend upwards.

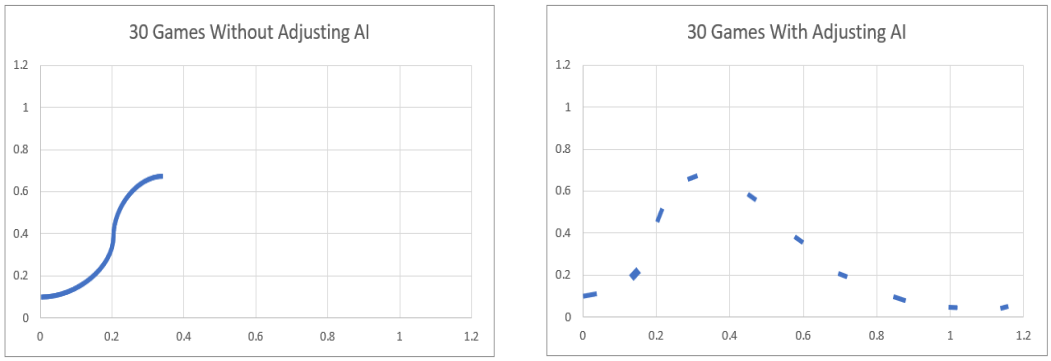

Figure 4.3: Theoretical data representing a function with and without adjusting the adversary AI

4.4. Data Collection

TAG data is an essential component of the TAG environment and the form in which it is written provides a fluid transition into most IL learning frameworks. Every file written into the database directory of TAG represents an entire game experience, start to finish. The file names follow the schema, mm-dd-hh-mm-ss, and is of a .csv format. The mentality in deciding what was to be documented in each game file was record everything, and prune later. This ensures that all possible information is recorded, and it can be decided later what is needed. Each data file has 56 features, 28 of which represent the ownership status, and the units of each of the 14 towers. 16 of the features represent units on the move (previous actions that are still in progress) including their origin tower, destination tower, units sent, and turns left until arrival. Given the game mechanics, there can only ever be 4 unit trips happening, and therefore only 16 cells are needed to represent these. 6 of the cells hold general board game information including, game turn number, a flag for whose turn it is (1 or 2), Player 1's tower count, Player 2's tower count, Player 1's unit count, and Player 2's unit count. 5 of the cells represent the action taken at that turn, including action type (unit movement, or a pass), and if it is a unit

movement action, the origin tower, the destination tower, the units being moved, and the number of turns before arrival (otherwise null if the action type is pass). The very last cell included is a time code representing the number of seconds passed since the start of the game. Finding how long it took for a player to make a decision is determined by taking the timecode at state $\boldsymbol{s_i}$, and subtracting the timecode from state $\boldsymbol{s_{i-1}}$. For each record the features representing the state information are what the user was presented with on their turn, and that lead them to take the action described by the subsequent action information. Both Player 1 and 2's information is recorded at every time step until a victor is determined.

Alteration of game data into training data for the BC user-model is an important step, as the preparation of the data is slightly different from most classifiers. Humans behave in a stochastic manner, meaning that if you were to present a person with an image and have them tell you what the image is of, you could receive a number of answers. Likewise, in decision making, if a human was presented with an identical game state to one they have seen before, it is entirely possible that they would choose an action different from the ones previously observed. This is especially true for a human that is learning to navigate an environment for the first time. This concept is considered in RL literature and is described as exploitation vs. exploration [27]. Exploitation is the desire of the human to exploit the information it knows that leads it to an outcome that is has deemed positive. Exploration is the desire of the human to explore new actions in the hopes to find an even more suitable outcome. If the model is to accurately portray a human is needs to capture this stochastic property, meaning that for any state observed, it

must be able to reflect the distribution of observed actions taken by the user. In some cases, only one action will have been taken for a particular state, but in others there can be a multitude of actions taken. For instance, the initial game state will appear in every data file and presents the user with 13 unique actions. The likelihood of the user taking the same action in this initial game state every time is very slim.

There are a number of strategies to train a stochastic neural network model. The strategy used in this framework is the MCNN structure proposed by Awiszus and Rosenhahn [13]. The process involves duplicating each observed game state by some factor of 10 and appending an r value to the front of each duplicate state counting from 0 to 1. The r value increments accordingly to the magnitude of the duplication factor (i.e. {0.01, 0.02, …, 0.99, 1.00} for a duplication factor of $\mathbf{10^2}$). The output for the duplicate set of a game state should represent the observed action distribution. The MC depicted in Figure 4.4 shows a set of states and their distribution of transitions. Table 4.2 and 4.3 shows how the state information for $\boldsymbol{s_1}$ and $\boldsymbol{s_3}$ could be altered into training data that would allow a neural network to capture their respective transition distributions. The duplication factor in this example is $\mathbf{10^1}$, however for all MCNNs used in this framework, $\mathbf{10^2}$ is used as it offers a more granular specificity.

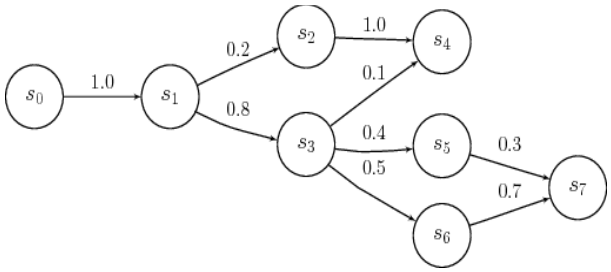

Figure 4.4: A simple Markov Chain with transition distributions

Table 4.2 and 4.3: Example training data preparation for states s_1 and s_3 according to their transition distribution

Training data for s_1		
Input		**Output State**
***r* value**	**state**	
0.1	s_1	s_2
0.2	s_1	s_2
0.3	s_1	s_3
0.4	s_1	s_3
0.5	s_1	s_3
0.6	s_1	s_3
0.7	s_1	s_3
0.8	s_1	s_3
0.9	s_1	s_3
1.0	s_1	s_3

Training data for s_3		
Input		**Output State**
***r* value**	**state**	
0.1	s_3	s_4
0.2	s_3	s_5
0.3	s_3	s_5
0.4	s_3	s_5
0.5	s_3	s_5
0.6	s_3	s_6
0.7	s_3	s_6
0.8	s_3	s_6
0.9	s_3	s_6
1.0	s_3	s_6

After training is completed for an MCNN, using one is just a matter of generating a random number between 0 and 1, and appending it to the desired state. With respect to the random number, a properly trained MCNN will provide an output according the distribution it was trained on. However, the distinction for our application is that in order for an MCNN to be a BC user-model, it must learn the distribution of observed actions by the user, rather than the distribution of possible states the user could go to. This is so that when the user-model is fed with state information, it will produce the action to be executed. Once these relationships are learned, the model will be able to traverse the TAG environment and behave in a non-deterministic fashion. The schema used for the training data is as follows:

$$\{r, t1_o, t1_u, t2_o, t2_u, \ldots, t14_o, t14_u, p1t1_d, p1t1_u, p1t1_t, p1t2_d, p1t2_u, p1t2_t, p2t1_d, p2t1_u, p2t1_t, p2t2_d, p2t2_u, p2t2_t\}$$

Each record of training data consists of 41 items. *r* contains each records *r*-value determined in the preprocessing phase according to the MCNN structure. The following

28 records contain tower information for all 14 towers (t1-t14), where those denoted with the subscript *o* represent the towers owner. The towers denoted with the subscript *u* contain unit information. To normalize, all units within a tower are divided by the tower's capacity. *p1t1, p1t2, p2t1,* and *p2t2* contain trip information. *p1t1* and *p1t2* represent Player 1's trip information, with the subscripts *d, u* and *t* representing the trips destination, units, and arrival time, respectively. *p2t1* and *p2t2* represent Player 2's trip information. All information is normalized between 0 and 1. Mapped to each input is the output representing an action. These actions mapped according to the action distribution mentioned previously. To describe the action, the output is made up of 14 cells representing each of the towers (A-N). The cell representing the origin tower of the action would be filled with a 0, the cell representing the destination tower would be filled with a 1, and all other towers filled with 0.5. For a passed turn, all cells are simply set to 0.5.

One final process is considered in training data preparation. A common technique in preprocessing data for image classification models is to scale and rotate an image. While the classification of the image stays the same, changing the image's orientation can provide unique insight for the model as well as producing additional training points. A similar approach is adapted in this framework. Given the symmetry of the TAG gameboard along either diagonal, any game state can be symmetrically translated. Figure 4.5 shows the translation of an original game state into three symmetrically equivalent game states. Even though the game states look drastically different, no change has been made to the strategy captured by the user. These additional training points also give the model a wider perspective in capturing user-strategy. This is of course only true for games following

TAG's default game settings. If settings are changed, breaking the symmetric properties of the board, this technique should not be used.

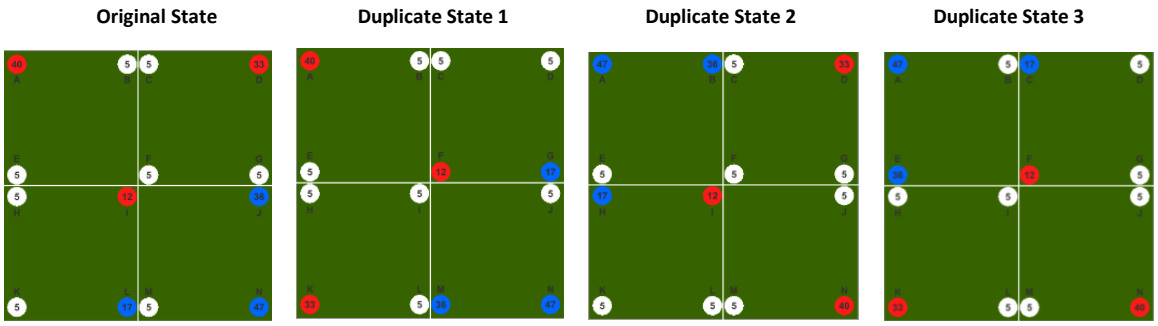

Figure 4.5: A game state translated across the diagonal lines of the TAG game board

To ensure the strategic integrity of symmetrically duplicating states is met, all unit trips have to be adjusted as well. For instance, in Figure 4.5 if the Original State has a trip for Player 2 to be arriving at tower L in two turns, the Duplicate State 1 would have a trip for Player 2 arriving at G in two turns. The action taken at this game state will also be translated similarly. In situations where taking the symmetrical duplicate of a state results in a direct copy of the original, but provides different actions, these duplicates are thrown away. Otherwise the result would be two directly equivalent states, with opposing actions, and this type of translation would affect the action distribution being captured in the MCNNs.

4.5. Constructing the User-Model

Being that this model follows the typical BC structure of mapping states to actions, the ultimate goal of the user-model is to be able to take a game state as input and provide us with an action. The final iteration of the user-model should not only provide us with an action, but the action that best represents the user, given their behavior. For every iteration,

the model will need to be plugged into the TAG environment, where it must act autonomously, so that the RL agent can infer its counter-strategy. The overall architecture of the user-model was designed with these requirements in mind and uses the user's training data described previously. The user-model is made up of two MCNNs wherein the system that combines them will be referred to as the neural network system (NNS). The first MCNN will be named the Detail Network (DN) and the second, the Context Network (CN). Each will be described as well as the training process for NNS.

4.5.1. Training the User-Model

The training process for constructing the NNS has two parts as both the DN and the CN are trained separately. The DN is trained first due to its output being required as input for the CN. The DN takes training data in as input, and maps this to the observed action taken by the user. However, some of the state space information relevant to the state of each tower is scrambled. All tower information, not directly relevant to the action that it is mapped to, is replaced with a random number between 0 and 1. For instance, if the action observed for a particular state is N→K, all of the 28 cells representing the tower information will be assigned a random value, except for tower details pertaining to N and K. The DN's job is to make sense of the input in how it determines an action. When noise is incorporated, no sense can be made of these features, and therefore the DN will learn to ignore them. However, the unfiltered tower information will start to develop relationships between towers. The objective of this network is to learn these tower relationships that the user found ideal when taking an action. If the towers don't contain the characteristics that make up these learned relationships, the DN will learn to treat them as noise. It should be noted

that after the both the DN and CN are trained, no filtering is done to the state information to infer an action. Ideally, when presented with the raw game state data, the DN should be able to parse what is relevant and produce an action according to the 14 cell action vector schema presented.

While the DN is trained to learn the finer details of the game state, the CN provides a much more contextual view of the game state. After the DN has been trained on all the training data, it is the CN's turn. The CN's input combines the initial training record and the action vector produced by the DN for that same record. No modification happens to the training record for either the DN or the CN in this case. Doing this takes the action determined by the DN and provides context to it. The CN produces the determined action to be executed. The behavior of the NNS is designed to capture one of the natural processes in how humans make decision. Fantino and Stolarz-Fantino [35] make the argument that successful problem solving in humans is dramatically affected by how the information relevant to the problem is presented. That behavior analysis, with its emphasis on the role of context in behavior, plays a significant role in human decision-making. Further, when making decisions human's will weigh actions against the bigger picture (which is captured in context) before deciding. The NNS approximates this behavior by first producing an action given the immediate factors contributing to it (DN), and then determining its practicality against the context of the situation (CN). The DN and CN don't necessarily have to agree in their action output, and often times don't. The action set provided by the DN uses a limited scope of the game board. It could be that when these actions are compared to the context in the CN, that the relevancy of these actions is decreased. The

CN, having the final say, would then produce the action to be executed. The training process for the NNS can be seen in Figure 4.6. Note that Phase 1 includes the modification of the training data, where Phase 2 does not.

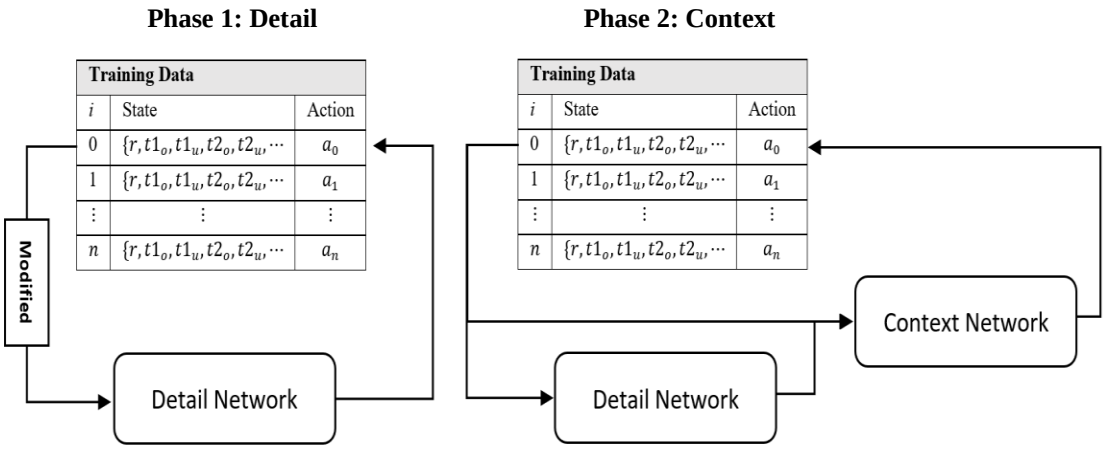

Figure 4.6: The learning process for the NNS consisting of two learning phases

4.5.2. User-Model Architecture

The DN's structure consists of one input layer, one hidden layer, and one output layer. The input count is adjusted according to the number of items within each training record (the input vector of size 41 described previously). The hidden count of the DN is twice the size of the input count (82). The output count consists of 14 cells for the desired action schema described above. The CN is quite similar in structure; however, its input includes the output of the DN in addition to the training record, thus leaving us with an input count of 55, a hidden count of 110, and the same output count of 14. To train both of these networks, gradient descent was used, with a learning rate and momentum of 0.003. The hidden counts as well as the learning rate and momentum were determined via trial and

error with the goal of minimizing their error, as well as attempting to avoid overfitting the model.

4.5.3. Plugging User-Model into TAG

After a model is trained, having it work in practice is as simple as taking the state information at hand and processing it through the NNS. Pinned to the front of the input must be a randomly generated number between 0 and 1, in order to fulfill the MCNN requirement. As the model navigates the TAG environment, there is no need to provide noise to the game states (as this only happens in the training process) for the DN or CN. The DN should handle what state information is relevant, given how it was trained. First, the data is processed by the DN to retrieve the initial action vector. Once passed through the DN, the CN takes the proposed action vector in addition to the original input of the DN. Ultimately, the CN produces the action to be taken. This process can be seen in Figure 4.7.

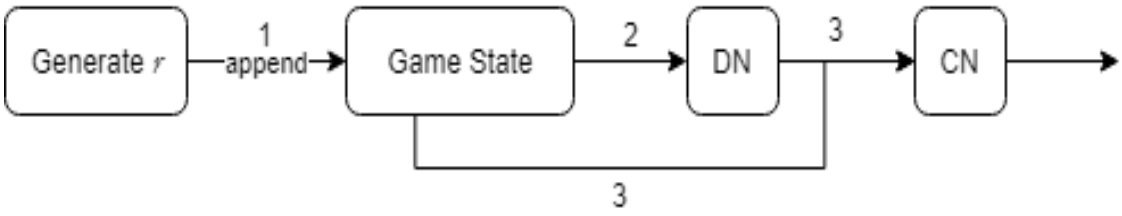

Figure 4.7: The process in which the NNS provides an action

The code that trains the user-model produces a file containing the memory matrix of the neural network. Effectively, this memory matrix is the stored memory of the user-model. To integrate the user-model into TAG one need only place the user-model file within the assets folder of the project directory. From there, functionality exists to read in the memory matrix of the file and form the user-model. TAG was written to be modular,

so that anyone can provide their own logic for Player 1's turn or Player 2's turn. Once the user-model has been loaded into memory, logic exists to read in the state space information from the game board and pass this through the NNS following the process seen in Figure 4.7. From the NNS an action will be provided, which in turn will be executed. All of the logic to operate the user-model exists within TAG but remains flexible enough to be adjusted for other AIs.

4.5.4. Describing User-Model Actions

One distinction to be made about the behavior of the CN and DN is that the action returned by either is not necessarily a clear-cut answer. Once the user-model is incorporated into TAG, it is going to experience a large amount of states that it has not seen in training. For reference, when considering every state discretely, the state space of TAG reaches an astounding $1.1 * 10^{30}$ (Table 4.1), and for all user data collected we will cover approximately 1000 unique games states. If every state the user-model was going to experience while navigating TAG was identical to the states it was trained on then the actions might fit closer to the 14 cell action schema described above of origin tower = 0.0, destination tower = 1.0, and every other tower =0.5. Given the state space complexity of TAG it is highly unlikely to encounter identical states past the first few moves. When the user-model is competing against the RL-agent it is going to experience a tremendous number of new game states, and as there is no action directly mapped to these new states. Actions provided by the NSS will more likely resemble a distributions of possible actions. For example, consider a potential action described by the DN to be (0.713, 0.461, 0.512, 0.496, 0.123, 0.011, 0.997, 0.419, 0.551, 0.508, .932, 0.398, 0.457, 0.228). A possible

explanation for this action is that the new state carried a number of tower relationship that the DN had learned to identify. Therefore, all of these towers carry some variance in determining the action vector. Reading this particular example would suggest that potential candidates for the destination tower could be A, G, or K and the potential origin towers could be E, F or N. This distribution allows all potential action candidates to be compared against the context when passed to the CN to check for their strategical integrity. This gives a certain degree of flexibility in deciding the ultimate answer. This also requires a fair amount of interpretation on the resulting action vector of the CN. Within TAG, the action vector provided by the CN is interpreted to be a valid action if it contains at least one tower of 0.7 or greater and one tower of 0.3 or less. If multiple of each are chosen, then a pair (origin and destination) is constructed by randomly selecting one from each. This was done so that the RL agent contesting the user-model would experience a number of varying strategies that the user might select. If no destination or origin tower is presented, this is to be interpreted as a pass.

4.5.5. Explaining the User-Model Structure and Other Applications

An entirely valid question to ask at this point is, if the purpose of the BC user-model was to suffer from covariate shift, why would this amount of detail be put into the user-model to enhance its performance? It is perhaps true that if a single FNN was used to learn the relationship of game states to actions, this would result in a model more susceptible to error. Then, if the model is more susceptible to error the RL agent would have a more obvious flaw to exploit. However, it is important to keep in mind, what it is exactly that we are trying to exploit. Observed strategy of the user should be aptly demonstrated by the

user-model as having it underperform within the realm of what the it was trained on would be less than desirable. Rather, the model should only underperform on game states where the user's strategy has not been observed. Additionally, for what has been observed, the NNS should be able to apply the user's strategy for seemingly similar states. This is due to the generalizing capabilities observed in neural networks. If the model is not able to generalize effectively, this would be an example of overfitting. Overfitting has been shown to lead to reduced generalizability of a model, restricting its applicability to new data [28]. If an overfit model were to be produced, it would be more likely to treat each state discretely, therefore applying a strategy different to what the user would have. The user on the other hand might have found the difference between two states to be trivial and apply the same strategy. This is an essential distinction to make, as overfitting can still be an issue regardless of a sparsity in data or not.

The difference between a properly fit user-model and an overfit user-model is monumental in this study. For an overfit user-model, the information that it has learned expands to far less of the state space than a properly fit one. As a result, the user-model's ability to perform in similar scenarios to what is has been trained on is diminished, thus decreasing the information the RL agent can infer from its strategy. If the user-model can only behave on such a limited number of states, then the RL agent need only push the game into a state with the smallest degree of difference to take advantage of it. Assuming the user-model can effectively behave in similar states, then the RL agent must drive the game states further away to take advantage of it. It has been shown that ensemble classifiers (classifiers made up of two or more base classifiers) have been used to capture complex

behaviors with more ease than a single classifier. However, ensemble classifiers are more prone to overfitting [29]. Each NN should be considered a point of failure. In training, if one NN becomes overfit, this can propagate through the whole system. Therefore, a great deal of care has been put into preventing overfitting. Even though this NSS is made up of two networks, each contain only a single hidden layer for simplicity, and the role that each one plays is modularized (one for prescribing actions on the immediate details, and the other for comparing to context). With their disjoint functionality, even if DN because overfit, it still provides a distribution of possible actions, and these actions will all be compared to the context in the CN. Additionally, given the nature of the MCNN, in that it incorporates probability, it allows for a fair amount of flexibility, and helps avoids local optimums [13].

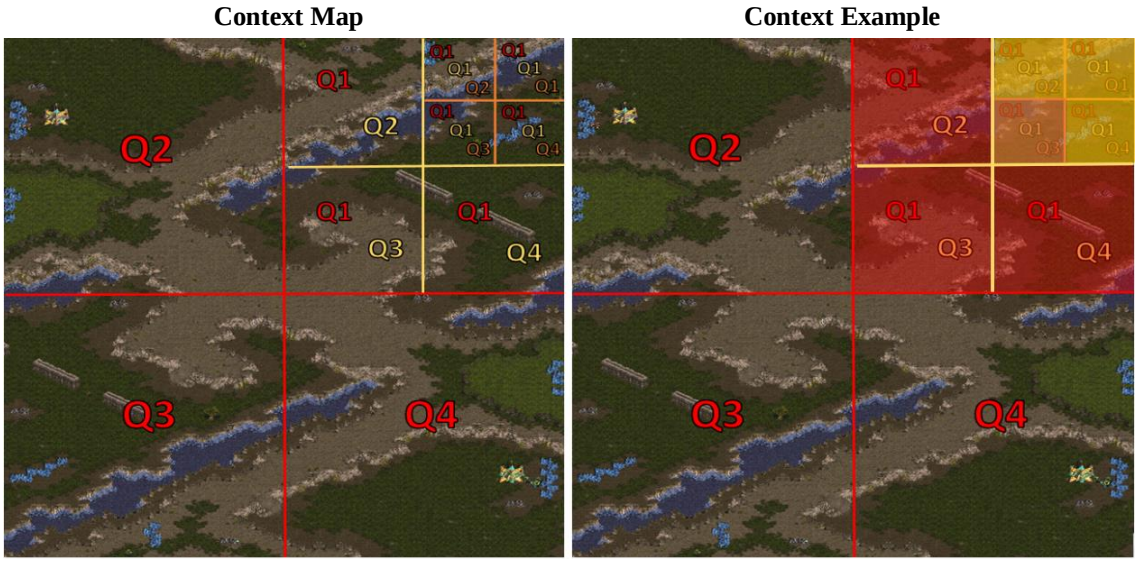

Figure 4.8: Official StarCraft II map, applying context breakdown

One of the primary focuses in this thesis is its ability to be extended to other domains, in addition to TAG. The NNS architecture has been constructed according to the TAG

environment (determining what is detail vs context), but how might other domains account for the modular characteristics of the NNS. For a StarCraft II map, the testbed DeepMind used to construct their RL agents [9], imagine continuously splitting the gameboard up into fourths as seen in Figure 4.8. At its smallest level of granularity, the detail for an action would be all information contained within this section. Context in this framework would consist of many layers. Let the naming scheme for the most atomic cell be $Q_aQ_bQ_c$, where Q_c represents tile 1-4 within Q_b, Q_b represents tile 1-4 within Q_a, and Q_a represents tile 1-4 within the complete picture. According to the context example in Figure 4.8, if an action occurred in $Q_1Q_1Q_3$, then all information pertaining to this section would be pass through the DN. As multiple CNs would be needed, the first level up (CN_1) would pass all information pertaining to the remaining three cells ($Q_1Q_1Q_1, Q_1Q_1Q_2$ *and* $Q_1Q_1Q_4$). The next Layer up (CN_2) would pass the remaining three cells from that section, and so on until the whole picture is formed. This particular example would require four MCNNs, the DN, CN_1, CN_2, and CN_3 which would feed into each other as seen in Figure 4.9.

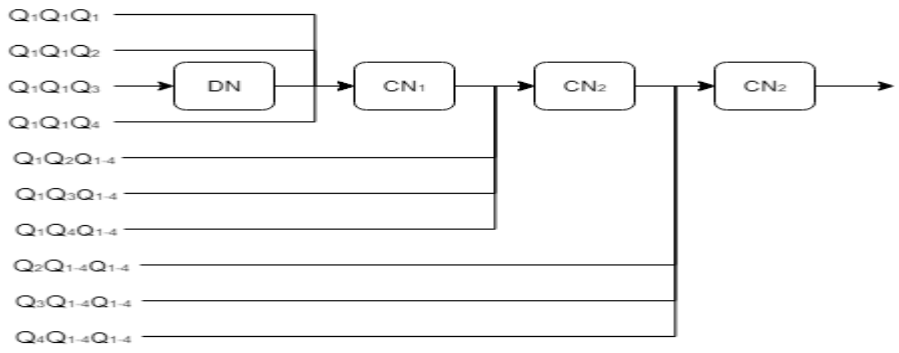

Figure 4.9: Context formulation for the given StarCraft II scenario

4.6. Reinforcement Learning with TAG

TAG was written in C# with the Unity Engine. Unity provides a number of assets that help developers create their software. One such asset, the ML-Agents toolkit [30], provides a communication tool to work between the Unity environment and the OpenAI Gym environment, which uses TensorFlow. This infrastructure streamlines the development of RL algorithms, which were used to develop the RL agents described in this work. With ML-Agents comes the ability to construct models that are easily adaptable into the Unity Engine. As TAG was designed in Unity, this allows for easy integration within the TAG environment. The goal of the RL agent is to learn a counter-strategy to contest the strategy captured within the user-model. The RL agent does this by learning to maximize its reward function, $r(s, a)$, through a series of self-supervised trials until an optimal solution is determined. Specifically, in this framework, it is to find flaws relating to the covariate shift problem of BC models on scarce datasets. In this section, the overall neural network structure of the RL agent will be explained, as well as the specifics on forming the reward function. The RL agent's implementation into the TAG environment will need to be described, as there are slight nuances to that of the user-model. Lastly, some theory on how the user-model and the RL agent drive each other will be presented, as well as some thoughts on expanding this to non-adversarial games.

4.6.1. Reinforcement Learning Architecture and Training

Contrary to the architecture of the user-model, the architecture forming the RL-agent is relatively complex. The simplicity of the BC user-modeling allows for the exacerbation of the covariate shift problem. However, for the RL agent, the information learned from

contesting the user-model should be as precise as possible in order to more accurately pinpoint what in the user-strategy we are missing. The more information the RL agent can infer, the more beneficial it will be to drive the user to descriptive states. The ML-Agent package uses TensorFlow and allows for nearly every aspect of the infrastructure to be modified. All parameters will be included in the agent's architecture will be displayed.

- Gamma: 0.995 – Gamma is the discount factor, meaning that the higher it is the more the agent acts to prepare for future states. Lower Gamma represents an agent that acts more in the present. The game chosen is high to help the agent prepare strategies early on that will develop through the entire episode.
- Epsilon: 0.1 – Epsilon is the threshold for divergence between old and new policies. Gradient descent updates policies during training but will only diverge from the old policy as much as Epsilon permits. The epsilon chosen is low so as to provide smaller steps and stabilize the training process.
- Beta: 0.01 – Beta corresponds to the randomness of policies early on in the training. The beta chosen is relatively high, however with 183 possible actions, this allows the agent to thoroughly investigate the action space.
- Sequence length: 64 – Sequence length corresponds to the length of the agent's experiences passed through the network at a time. This should be large enough that it captures an entire episode of the agent exploring the TAG environment.

- Number of Epochs: 4 – The number of epochs is the number of passes through the experience buffer during gradient descent. This is set relatively low, but it corresponds to the smaller batch size.
- Batch size: 512 – Batch size is the number of iterations used to update the agent's policy in gradient descent.
- Learning rate: 0.0001 – Learning rate indicates the strength of each step in gradient descent. This was set fairly low to prove a more stable training session.

Some of these parameters were domain specific, for instance time horizon and sequence length were set to their values given the estimates of how long a given episode might last. The main focus of the architecture was to provide a stable learning environment for any user-model, so that changes wouldn't need to be made when handling different user's models. Figure 4.10 shows the summary of a successful training. The general trend for a successful training session is that the reward steadily increases over time, before it stables out, but there are other important indicators.

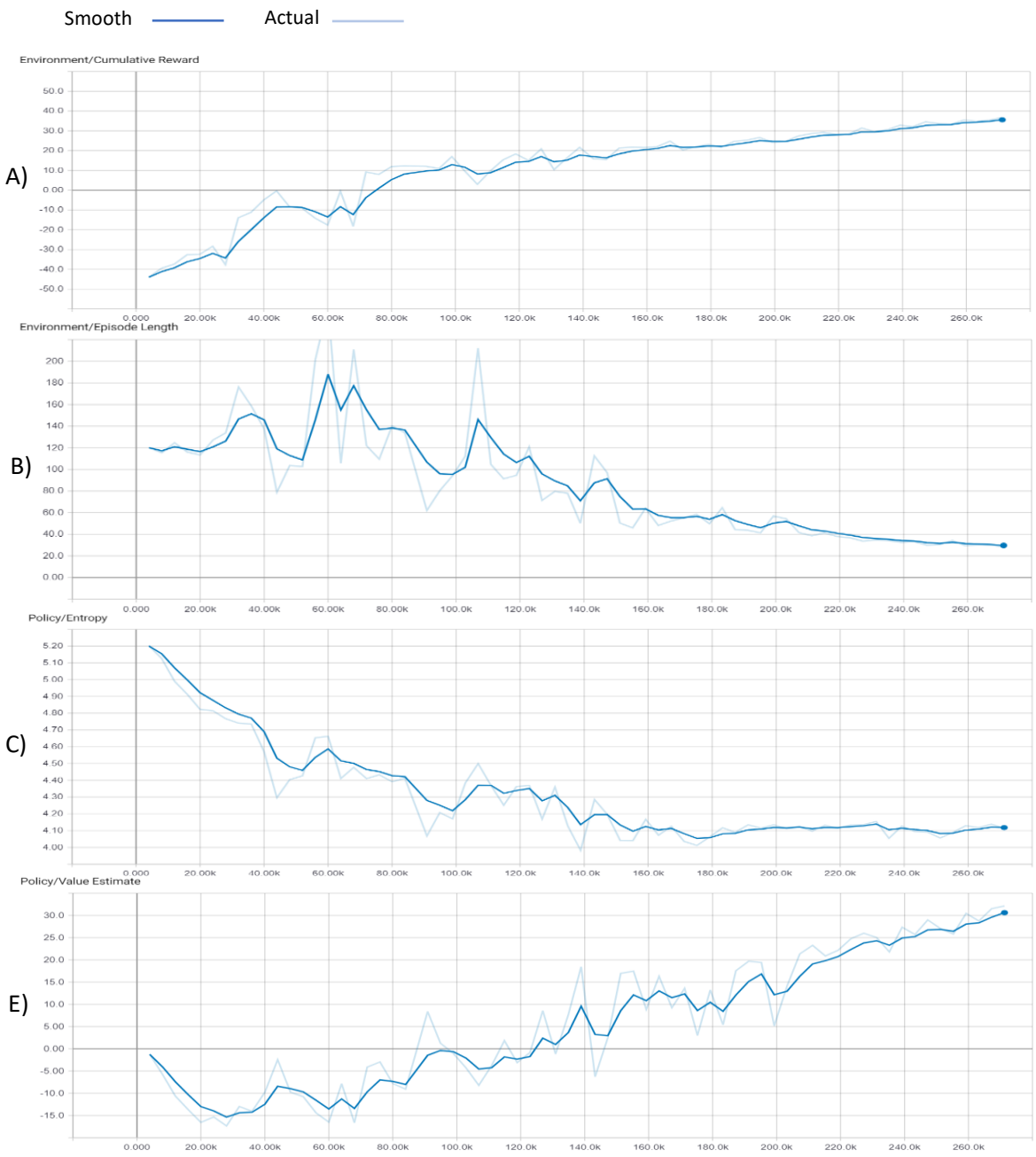

Figure 4.10: Successful RL agent training session

Figure 4.10 A shows the reward value the RL agent is earning over time. During a successful training, this should tend upwards, the x-axis shows the number of steps taken, which in this training example is ~270,000. Figure 4.10 B shows the average episode length, which is a complete game start to finish. The goal of each RL agent is to win as

quickly as possible, so for this framework, it is ideal for this graph to tend down. Figure 4.10 C shows the randomness of the RL agent. The higher the entropy, the more random its policy. As the RL agent constructs concrete strategies, it becomes less random and more calculated, so during a successful training this should get smaller. Lastly, Figure 4.10 D shows what possible reward the agent thinks it can achieve for an entire episode. This should increase with Figure 4.10 A.

The agents only source of information during training is the same tower, and trip information that is available when training the user-model. No pixel information is used, as the learning processes wouldn't benefit much from TAG's relatively simple visuals. That being said, TensorFlow is a very powerful tool and the game state features used to train strongly define the game state. Most work in RL focuses on learning to map actions with pixel information, and thus convolutional neural networks are best suited. Being that this RL agent doesn't use pixel data, the architecture style for this model would be considered a recurrent neural network. What makes the RL agent recurrent is that the observation vector, that describes the state, stacks a total of fifteen previous states worth of information. The chosen parameters may not necessarily be ideal for other domains, but for the purposes needed in this framework, the designed RL agent architecture has led to the production of concrete RL agents capable of learning long-term strategies that counter the strategies imbedded within the user-models.

4.6.2. Deciding the Reward Function

Deciding a reward function for an RL agent can be a tricky task. Generally speaking, the agent should be rewarded for successfully navigating the environment and reprimanded

for the opposite. Sometimes to expedite the process, if the person training the RL agent is knowledgeable enough of the environment they can artificially lead the agent to that destination. However, artificially leading the RL agent to a solution may in fact result in a local optimum, as the optimal strategy may not have been known previously. In many cases, it is suggested to offer the least amount of influence, and provide rewards, with the least amount of bias, that lead the agent to the optimal goal. In the case of this framework, the agent is only reward in the event that it beats the user-model and is punished for losing to it. The only added restriction is that the reward, in the event of a win, decreases for the number of actions the agent had to take to get there. There are a tremendous number of trajectories that could lead the agent to a win over the user-model, however, the trajectories that most effectively take advantage of the user-model's inadequacies would be those that lead the RL agent to a win over the user-model as quickly as possible. Therefore, giving the agent an incentive to win faster helps accomplish this goal. To incorporate this strategy, for a win the agent receives a reward of 1,000/turns. In the event of a loss, the agent receives a flat reward of -50. The value of 1,000 was chosen, as the typical reward distribution of an optimal RL agent returns approximately 20-60 points which translates to about 50-17 actions respectively. Counter strategies of the RL agent that must perform more actions is indicative of it competing against a defensive player.

4.6.3. Plugging Reinforcement Learning Agent into TAG

Like the user-model, the RL agent must be able to be plugged in to the TAG environment. The ML-Agents asset provides framework to take the NN file produced from TensorFlow and load it into memory. For Unity, the object that loads the NN file is called

the Player Brain and attached to the Player Brain object are scripts that help interact with the memory of the trained model. The same scripts used to train the model will also be used to have the model execute actions, but in this case no alteration to the memory will take place, as it would when learning. At each turn, the model will simply read from the TAG game board, process the data, and produce an action. As the RL agents don't use pixel data, the observations used to guide the agent to a decision are all tower owners and units, as well as the trips currently in progress. For a continuous action space, RL agents typically perform actions with some form of a delay on them, however being that this state space is discrete, rather than have an action perform every number of game ticks, the logic demands an action of the model on its turn to ensure the game flows seamlessly. Trained RL agents will only ever compete against the human whose data was used to construct the user-model. Therefore, the RL agent will take the position of Player 2, and the human will take the position of Player 1. In Figure 4.6, this step of the process is denoted as the 6th step, which is the last step of the cycle before it repeats.

4.6.4. Driving the User with Reinforcement Learning

The interaction within TAG is dyadic as there are two competing players. The state that each player is presented with on their turn not only relies on their previous action, but the action of the competing player as well. Before, trajectories were defined as S_i, A_j, S_{i+1} where S_i is the state the user is presented with, A_j is the action determined by the user, and S_{i+1} is the resulting state. However, in an adversarial game this should be formed as $S_i, A_j, S_i', A_j', S_{i+1}$ where S_i' is the state the opposing player is presented with, and A_j' is the action the opposing player took. This is relevant to the problem as the opposing player

has half of the control when it comes to determining new states. When the RL-agent is learning the optimal counter-strategy to combat the user-model's strategy it will learn the actions the user-model prefers, as a byproduct of learning its own. This is the exploiting property of the RL agent to take advantage of the covariate shift in BC. Effectively, the RL agent learns what actions it needs to take, in order to present the user-model with state information that it underperforms in. When the user competes against the RL agent, the trajectories they will experience are subject to partial control of the RL agent. The RL agent will try to push the user, expecting the same inefficiencies as the user-model, but if more of the user's strategy is left to be explored, then this approach will force the user to demonstrate that strategy. If there is no remaining strategy the user has to demonstrate, this would result in convergence, and the complete counter-strategy will have been inferred. Imagine the RL agent's strategy and the User-model's strategy as opposing forces. In Figure 4.11, the Anticipated Trajectory is a result of the RL agent finding equilibrium in the form of an optimal strategy with respect to not only the actions it takes, but the actions the user-model takes. However, this framework concedes that covariate shift is a problem with BC and that there will be a difference between the user-model's strategy and the user's actual strategy (shown by the yellow highlighted section). When the user competes against the RL agent, the actual trajectories that they will be experience will be a result of their actual strategy and the RL agent's strategy. In an ideal scenario, the yellow highlight will shrink over time, as covariate shift is accounted for.

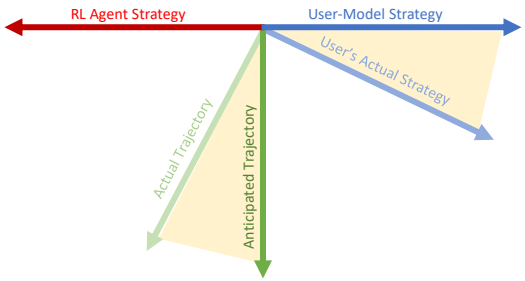

Figure 4.11: Theoretical result of the RL agent driving the user. The yellow highlight indicates the anticipated covariate shift

Chapter 5

Results

In order to prove the success of this framework, two important aspects must be considered. First, the RL agent in this framework should be capable of driving the user to unique game scenarios. Second, these new trajectories should contain relevant information, meaning that the driving force of the RL agent is strong enough that it demands the user to adjust their strategy in order to beat it. Remember, the counter-strategy inferred by the RL agent may not be the global counter-strategy, but rather each iteration should be considered a, "what if" for the user. When the user adapts to a strategy, they will have successfully demonstrated their response to the, "what if" and a new one will be imposed on them in the next iteration. So then, if every iteration of the RL isn't demanding the user adjust their strategy, it can be safely assumed the, "what if" wasn't a strong enough force to drive the user. Each new iteration should be a calculated move in what it wants the user to demonstrate. This technique of provoking new responses from the user aims to minimize the redundancy in data collected. If a framework were to only have the adversarial agent pick moves at random to compete against the user, this may very well present the user with new game states, but this force may not be strong enough to encourage the user to change

their strategy. This sort of random approach also leads to diminishing returns in the uniqueness of game data as well. A random agent might cover a significant amount of state space early on, but ultimately will produce redundant data after some time.

To reiterate, the data collection and the processing of that data occurs after every fifth game the user plays. The only exception to this rule is when the user is collecting a baseline of ten games against the default AI. In order to demonstrate the driving force of each new iteration of the RL agent, Figure 5.1 shows a heat map of Player 1's (average) tower possession after the 5^{th}, 10^{th}, 15^{th}, 20^{th}, and 25^{th} action. This demonstrates how these tower ownerships differ according to which AI the user played against. These averages were determined by computing the percentage of times the player owned that tower, after the indicated turn, in that set of games. Every column shows their tower expansion over time for the given AI, and every row shows the user's ownership position after their nth move. The user always starts with tower N, but every other tower achieved is Player 1's genuine response to the AI. Column 1 shows an even expansion from N, moving towards the bottom left and top right corners. Likewise, column 2 also spreads towards the bottom left and top right, however expansion appears to be slower, and the user prioritizes the tower D and K, rather than F and I. In column 3, the RL agent contests tower K early on, so the user abandons tower K, to achieve tower D and a strong presence in the middle of the board. Surprisingly, by the last iteration of the RL agent, the user becomes more aggressive and strikes the RL agent's default tower very early on. An explanation for this could be that the RL agent learned to contest tower D and K effectively but left itself open for an aggressive attack from the user.

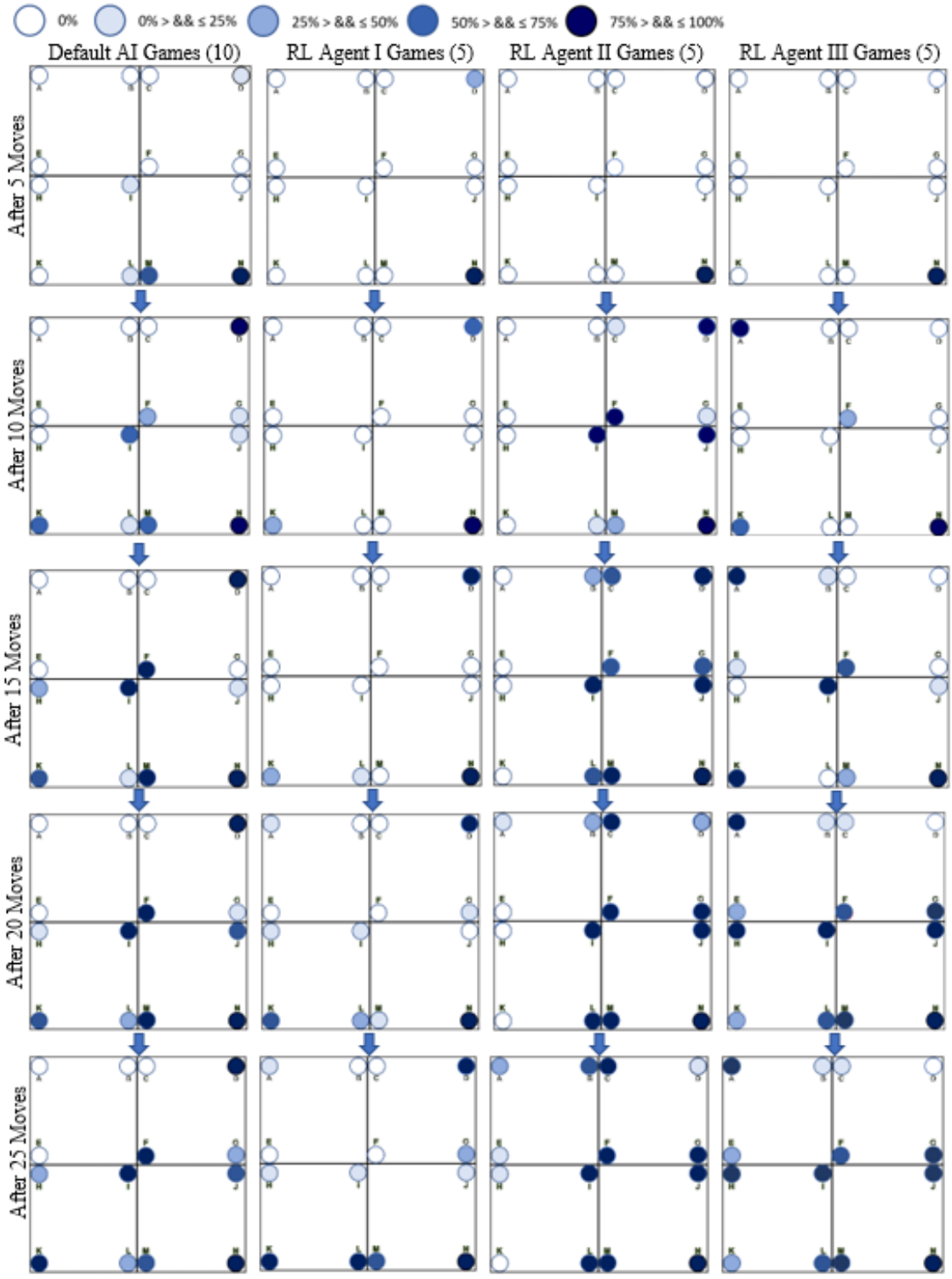

Figure 5.1: Player 1 heatmap of tower ownership after 5th, 10th, 15th, 20th, and 25th move compared against varying adaptive AIs

Figure 5.1 demonstrates the adaptation of the user's strategy to each AI qualitatively. The quantitative data for Figure 5.1 can be seen in Table 5.1. Specifically, the squared and absolute difference between these ownerships from the heatmap is shown. Tower ownership differences are clearly seen between these AIs, but in order to determine how different these are, Table 5.2 was constructed, by randomly sampling five of the original ten default AI games, and comparing them to the remaining five games. Only by comparing these two tables is the relevancy of this framework realized, as this shows the frameworks ability to drive the user to demonstrate new strategies.

Table 5.1: The squared and absolute difference in tower ownership percentages over time between iterations of the AI

	Default – RL AgentI		**RL AgentI – RL AgentII**		**RL AgentII – RL AgentIII**	
	x^2	$\|x\|$	x^2	$\|x\|$	x^2	$\|x\|$
5 Moves	0.55	1.10	0.20	0.60	0.52	1.00
10 Moves	1.29	2.70	2.88	4.40	4.28	5.80
15 Moves	2.49	3.50	3.48	5.40	4.72	6.40
20 Moves	2.68	3.60	5.24	7.00	4.04	6.20
25 Moves	2.16	3.60	4.48	6.60	2.08	4.40

Table 5.2: The squared and absolute difference in tower ownership between samples of the default AI

	Default Sample1 – Default Sample2		**Default Sample2 – Default Sample3**		**Default Sample3 – DefaultSample4**	
	x^2	$\|x\|$	x^2	$\|x\|$	x^2	$\|x\|$
5 Moves	0.08	0.40	0.08	0.40	0.12	0.60
10 Moves	0.36	1.40	0.52	1.80	0.40	1.60
15 Moves	0.28	1.00	0.36	1.40	0.48	1.60
20 Moves	0.48	1.60	0.28	1.00	0.32	1.20
25 Moves	0.2	1.00	0.16	0.80	0.12	0.60

Like Table 5.1 and 5.2, Figures 5.2 – 5.5 display how wide the gap is between the varying strategies observed by applying this framework, and when no such techniques are being employed. Figure 5.2 and 5.4 show the cumulative squared and absolute difference (respectively) in tower ownerships between each transition of the AI employed by this

framework, whereas Figures 5.3 and 5.5 show the differences in tower ownership when sampling the user's games against the default AI. At a simple glance, with each iteration of the AI, a substantial amount of variation occurs in the user's strategy.

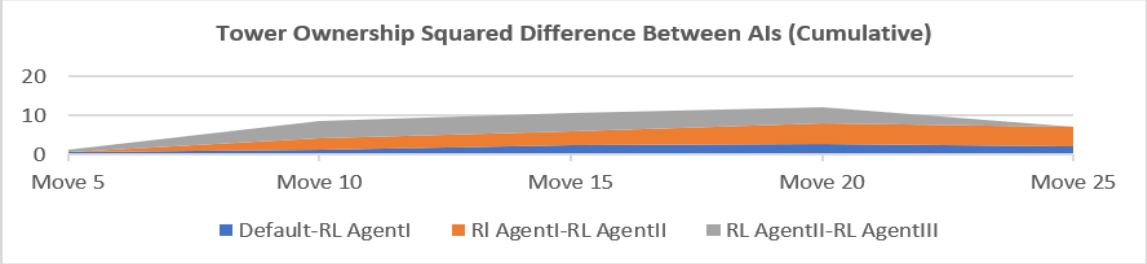

Figure 5.2: Shows the compounded squared differences in tower ownership for Table 5.1 using adaptive AI

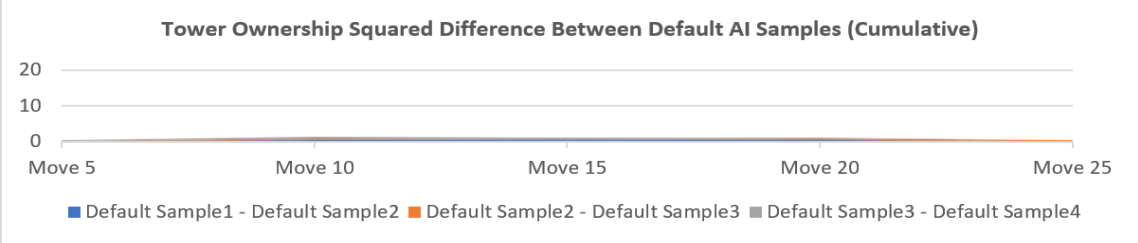

Figure 5.3: Shows the compounded squared differences in tower ownership for Table 5.2 using no adaptive AI

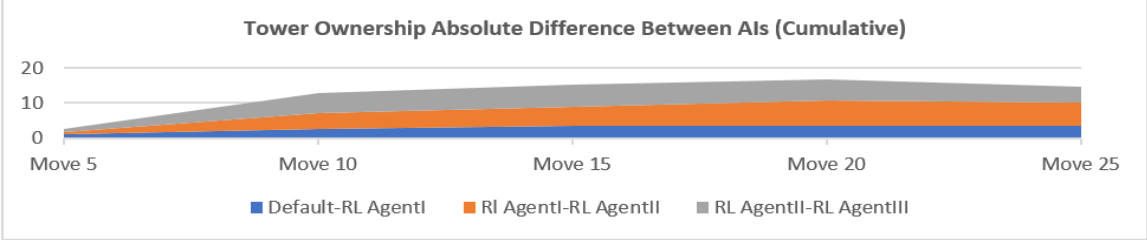

Figure 5.4: Shows the compounded absolute differences in tower ownership for Table 5.1 using adaptive AI

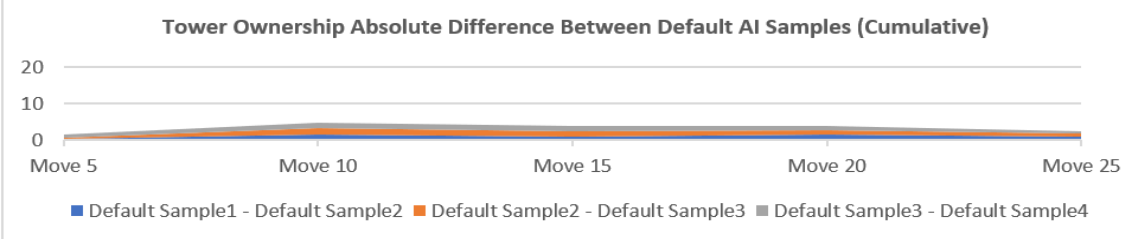

Figure 5.5: Shows the compounded absolute differences in tower ownership for Table 5.2 using no adaptive AI

Chapter 6

Conclusion and Future Work

The results found in this thesis present an exciting new means to address the covariate shift problem encountered in BC user-modeling. Not only is this applicable to BC user-modeling community, but the ability to actively drive the user to demonstrate new strategies is relevant to the user-modeling community as a whole. This thesis also promotes an authentic user-modeling experience as all models are created entirely from user-data, rather than filling the gaps with some form of self-supervision. Further, outside of the results, a valuable tool to the user-modeling community has been proposed in TAG. TAG's functionality was deliberate in its design to be as valuable to the community as possible. This primarily includes streamlining data collection and processing, as well as providing a modular code base to allow various types of AI to navigate the TAG environment. The TAG environment has already been used to evaluate the IRL processes in my work collaborating with Thayer School of Engineering at Dartmouth. This work appeared at the Naturalistic Decision-Making conference in September 2019 [31].

A number of interesting research questions stem from the work presented here. First and foremost, the goal of driving the user experience to new trajectories, was to construct

an RL agent capable of directly countering the user. This was not necessarily achieved, but perhaps with the targeted data collection performed in this framework, after a number of iterations enough relevant data could be collected to train an IRL model that is capable of capturing the user's behavior in its entirety. This could in turn be used to compete against an RL agent to achieve a fine-tuned counter strategy. Although, exactly how much data is needed to make this a reality, and is this number achievable in a realistic amount of time, even with the support of this framework? Lastly, can this framework be applied to non-adversarial games? For instance, if an environment were to be capable of dynamically changing, such that it can act as the RL agent with the goal of minimizing the user-model's reward, then would this framework still be applicable? As it is now, demanding a testbed to be adversarial in order to apply this framework, narrows the applicability of this work. All of these are to be left unanswered in this work due to time constraints but pose to be research worthy questions.

LIST OF REFERENCES

[1] Santos, E., Jr. & Nguyen, H. (2017). Capturing a Commander's Decision Making Style, in Proceedings of the SPIE 10184, Sensors, and Command, Control, Communications, and Intelligence Technologies for Homeland Security, Defense, and Law Enforcement Applications XVI.

[2] Bingham, C. B., & Eisenhardt, K. M. (2011). Rational heuristics: the 'simple rules' that strategists learn from process experience. Strategic management journal, 32(13), 1437-1464.

[3] Ariely, D. (2008). Predictably irrational : the hidden forces that shape our decisions (1st ed.). New York, NY: Harper.

[4] Scott, Susanne G and Bruce, Reginald A. (1995). Decision-making style: The development and assessment of a new measure. Educational and Psychological Measurement 55(5): 818–883.

[5] Klein, G. A., Orasanu, J. E., Calderwood, R. E., & Zsambok, C. E. (1993). *Decision making in action: Models and methods.* In This book is an outcome of a workshop held in Dayton, OH, Sep 25–27, 1989.

[6] Sutton, R., & Barto, A. (1998). *Reinforcement learning : an introduction* . Cambridge, Massachusetts: MIT Press.

[7] Ross, S., & Bagnell, J. (2010). Efficient reductions for imitation learning. In Proceedings of the thirteenth international conference on artificial intelligence and statistics, pages 661–668, 2010.

[8] OpenAI. (2018, August 31). OpenAI Five Benchmark. Retrieved from https://blog.openai.com/openai-five-benchmark/

[9] Vinyals, Oriol and Ewalds Timo (2017). StarCraft II: A New Challenge for Reinforcement Learning. Cornell University.

[10] M. Kearns and U. V. Vazirani (1994). An introduction to computational learning theory. MIT Press.

[11] K Hornik and M. Stinchombe (1992). Multilayer feed-forward networks are universal approximators in Artificial Neural Networks: Approximation and Learning Theory, H. White et. al., Eds., Blachvell press, Oxford.

[12] Bebis, G., & Georgiopoulos, M. (1994). *Feed-forward neural networks*. IEEE Potentials, 13(4), 27-31.

[13] Awiszus, Maren, & RosenHahn Bodo. (2018) *Markov Chain Neural Networks*. Cornell University.

[14] Mnih Volodymyr and Kavukcuoglu Koray. (2013) Playing Atari with Deep Reinforcement Learning. Cornell University.

[15] Watkins, C., & Dayan, P. (1992). Technical Note: Q-Learning. *Machine Learning*, *8*(3), 279–292. https://doi.org/10.1023/A:1022676722315

[16] Goodfellow, I., Mcdaniel, P., & Papernot, N. (2018). Making machine learning robust against adversarial inputs. *Communications of the ACM*, *61*(7), 56–66. https://doi.org/10.1145/3134599

[17] Yang, K., Liu, J., Zhang, C., & Fang, Y. (2018). Adversarial Examples Against the Deep Learning Based Network Intrusion Detection Systems. *MILCOM 2018 - 2018 IEEE Military Communications Conference (MILCOM)*, 559–564. https://doi.org/10.1109/MILCOM.2018.8599759

[18] Cody, T., Adams, S., & Beling, P. (2018). A utilitarian approach to adversarial learning in credit card fraud detection. 2018 Systems and Information Engineering Design Symposium (SIEDS), 237-242.

[19] Miller, D., Hu, X., Qiu, Z., & Kesidis, G. (2017). Adversarial learning: A critical review and active learning study. *2017 IEEE 27th International Workshop on Machine Learning for Signal Processing (MLSP)*, *2017-*, 1–6. https://doi.org/10.1109/MLSP.2017.8168163

[20] Ho, Jonathan and Ermon Stefano. Generative Adversarial Imitation Learning. Cornell university.

[21] Scott, S. G., & Bruce, R. A. (1995). Decision-making style: The development and assessment of a new measure. *Educational and Psychological Measurement, 55*(5), 818-831.

[22] Torabi, F., Warnell, G., & Stone, P. (2018). Behavioral Cloning from Observation. Cornell University.

[23] Ziebart, B. D., et al. "Maximum Entropy Inverse Reinforcement Learning." AAAI. Vol. 8. (2008).

[24] Tucker, A., Gleave, A., & Russell, S. (2018). Inverse reinforcement learning for video games. Cornell University

[25] N. D. Ratliff, D. Silver, and J. A. Bagnell (2009). Learning to search: Functional gradient techniques for imitation learning. Autonomous Robots, 27(1):25–53.

[26] Wieland, Kerkow, Früh, Kampen, & Walther. (2017). Automated feature selection for a machine learning approach toward modeling a mosquito distribution. Ecological Modelling, 352, 108-112.

[27] Sutton, R., & Barto, A. (1998). *Reinforcement learning : an introduction* . Cambridge, Massachusetts: MIT Press.

[28] Vaughan, I.P. & Ormerod, S.J. (2005) The continuing challenges of testing species distribution models. Journal of Applied Ecology, 42, 720–730

[29] Pourtaheri, Z., & Zahiri, S. (2016). Ensemble classifiers with improved overfitting. *2016 1st Conference on Swarm Intelligence and Evolutionary Computation (CSIEC)*, 93–97. https://doi.org/10.1109/CSIEC.2016.7482130

[30] Matín abadi, et al (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

[31] ML-Agents (Version 0.8.1). (n.d.). Retrieved from https://github.com/Unity-Technologies/ml-agents

[32] Nyanhongo, Clement & Hyde, Gregory (2019). A decision-making framework for agents in complex environments. *International Conference on Naturalistic Decision Making.*

[33] Y. Ng, Andrew & Russell, Stuart. (2000). Algorithms for Inverse Reinforcement Learning. ICML '00 Proceedings of the Seventeenth International Conference on Machine Learning.

[34] Abu-Jamous, Basel, Fa, Rui, and Nandi, Asoke (2015). K. Feature Selection. *Integrative Cluster Analysis in Bioinformatics*. Chichester, UK: John Wiley & Sons. 109-17. Web.

[35] Fantino, and Stolarz-Fantino (2005). "Decision-making: Context Matters." *Behavioural Processes* 69.2 165-71. Web.

[36] Todorov, E., Erez, T., & Tassa, Y. (2012). MuJoCo: A physics engine for model-based control. *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 5026–5033. https://doi.org/10.1109/IROS.2012.6386109