Linear Function Approximation with Q-Learning for Rich Dynamic Action Spaces

Marius Kilian
Game AI Research Group
Queen Mary, University of London
London, UK
m.kilian@se22.qmul.ac.uk

***Abstract*—A reinforcement learning (RL) using linear function approximation and Q-learning is explored in the Tabletop Games Framework (TAG), a Java framework for artificial intelligence in tabletop games. Linear function approximation RL allows training on rich dynamic action spaces that board games can offer, while providing transparent and interpretable results, giving insight into the agent's "thought process". Implementing the agent in TAG takes advantage of the extensive library of board games that are implemented in TAG, as well as the numerous types of search agents, allowing the RL agent to be compared against many methods in many environments. Further, a direct comparison between search-based agent and the planning-based RL agent is possible. Here, the RL agent is tested in three game, *Tic-Tac-Toe*, *Dots and Boxes* and *Sushi Go*, against different Monte-Carlo Tree Search agents. While it reaches high performance in *Dots and Boxes*, it does not perform nearly as well in the other two games. The overall performance in all games based on amount of training is inconsistent and unpredictable. Generally, the agent still shows potential to be a well-playing artificial intelligence agent for the framework, but needs adjusting in the methods used for its training, as well as the extent of its training.**

***Index Terms*—Reinforcement learning, linear function approximation, Q-learning, the Tabletop Games Framework, artificial intelligence**

I. Introduction

Artificial intelligence (AI) has been used in games for a long time, from creating suitable opponents for casual players, to challenging the most highly skilled players of certain games. AI game agents have helped players become better in their respective games, sometimes even to the point of discovering or identifying new strategies to be adopted by human players, such as AlphaGo in Go (Holcomb et al., 2018), or OpenAI Five in Dota 2 (OpenAI et al., 2019). Both of these agents were trained using reinforcement learning (RL), a popular method in which the agents learn to interact with complex environments by doing self-play over countless iterations and using its experiences to make more and more informed decisions.

The area of application for these two agents, as well as most other RL agents, usually has a fixed action space, which, while it can offer high complexity, usually stays within the bounds of being iterable for the computers these agents are trained on. This allows these agents to use deep neural networks, with the state as the input layer, and all possible actions as the output layer. The output node with the highest activation can then be chosen as the action to take. However, the bigger the action space gets, the more complex the entire algorithm becomes, greatly increasing compute times. Modern board games often have very rich and highly complex action spaces, that change depending on the game state. Not only can listing all the actions within a game prove to be a very difficult task, but the sheer amount of actions that these algorithms would be tasked with processing would be highly inefficient. Since RL agents often need to play millions or even billions of games to be good at their respective games, reducing compute time to play such a game is a very important issue to tackle. (Ye et al., 2023)

To use an RL agent for such a rich, dynamic action space, linear function approximation (LFA) is used, rather than the common approach of a neural network. Neural networks are very efficient for a fixed action space, as one set of inputs instantly provides the best action to take for the given input state. Using LFA, the inputs for the function have to include the action, so it has to be called once for each possible action of this space. While this is generally more compute inefficient, it can be more efficient in the context of flexibility with the rich action spaces board games can provide, as only the actions that are possible from a given state can be checked, while the others can be ignored. Thus, the LFA RL agent can effectively be used for the rich, dynamic action spaces.

To use LFA, a list of features needs to be constructed for a given game, abstracting the game state down to core features that define the state. The LFA RL agent is trained using Q-learning, and, as the name suggests, uses a linear function to approximate the best action to take in each state. This specific RL agent is implemented in the Tabletop Games Framework (TAG) in Java, allowing a rich and expanding variety of board games for testing, as well as a variety of other types of AI game agents to compare against.

Using this approach also allows for easily interpretable results, as each feature of the LFA is clearly defined, and by seeing the weight assigned to it, conclusions can be drawn about how vital the agent assesses the feature to be to winning. This allows an insight into what decisions the agent is taken that is much harder, often close to impossible, for RL agents using neural networks. Further, if the agent is adept at the game, this allows conclusions to be drawn about the balance of the game, and possible adjustments to be made, which could be useful for the game design process.

Adding such an agent to TAG allows a direct comparison against a variety of planning algorithms. Further, TAG currently only has planning algorithms with no pre-planning, therefore adding a different type of agent provides value to the framework as it allows the games to be tested for a very different type of agent.

Chapter II will provide more details into TAG as well as other RL agents, before discussing implementation details of the agent in chapter III. The agent is then compared with varying parameters against other agents in a small set of different games in chapter IV, to test its performance, as well as how big the trade-off in training has to be to compete with the on-the-fly agents that TAG offers, such as the Monte-Carlo Tree Search (MCTS) agent; before finally drawing conclusions and discussing future directions of this work in chapters VI and V, respectively.

II. Background and Related Work

A. Background

1) The Tabletop Games Framework: TAG is a Java framework by Gaina et al. (2020a) that provides methods and interfaces to implement modern tabletop board games in a way that allows various AI agents to play them. For example, game states in TAG have to be implemented in a way that a deep copy can be made from them, and players (or agents) in the framework are supplied with a forward model, which allows them to use such a copied state to apply an action, and receive a simulation of the next state. Any information that is randomized in a game is randomized differently in the simulated state than it is in the actual game, to not provide the agents with an unfair advantage. This setup allows many agents that use look-ahead to work on the framework, and be used in a plug-and-play manner in the variety of games the framework supports. Further, TAG provides a variety of ways to let agents play against each other, outputting useful statistics about win rates, scores, decisions and other metrics, along with providing interfaces to program customized metrics. (Gaina et al., 2020b)

2) Reinforcement Learning: RL is a subset of AI, where the given model is trained based on a reward system and autonomous training. At first, the agent makes random decisions, based on the initialization values of its weights, be it the weights between the nodes of a neural network, or the Q-weights of a LFA an RL agent, as described in chapter II-A3. Through these random decisions, a state giving a reward is eventually reached. This could be a positive reward, such as winning the game, or a negative reward, such as losing the game. This reward is then back-propagated through the list of states and actions the agent traversed, adjusting the weights in the process. There are several learning algorithms to adjust these weights, each with its own upsides and downsides. Here, Q-learning is used as the learning algorithm. In the context of board games, the autonomous training is done with multiple RL agents with the exact same knowledge about the environment playing against each other repeatedly. After each game, the learning algorithm is applied, and all agents receive the new, updated weights to interact with the game with.

Algorithm 1: A single step of non-tabular Q-Learning

Input: State s, possible actions A_s, learning rate α, exploration factor ϵ, discount factor γ, Q-weights vector θ
Output: Updated Q-weights vector θ

$Q_s \leftarrow \theta \cdot \phi(s,a) \,\forall\, a \in A$

$$a_s \leftarrow \begin{cases} \underset{a \in A_s}{\operatorname{argmax}}\, Q_s(a) & \text{with probability } 1-\epsilon \\ \text{random action in } A & \text{with probability } \epsilon \end{cases}$$

$r \leftarrow$ observed reward for action a_s in state s

$s' \leftarrow$ observed next state for action a_s in state s

$\delta \leftarrow r + \gamma \max_{a \in A} Q_{s'}(a) - Q_s(a_s)$

$\theta \leftarrow \theta + \alpha\, \delta\, \phi(s,a)$

3) Linear Function Approximation: This is a method of approximating unknown state-action value functions. There is a number of variables as inputs, and the same number of weights, called Q-weights. The dot product of this variable vector, called the feature vector (FV), and the Q-weights vector, is then taken, outputting a value that determines the *quality* of this FV, called the Q-value. The weights, called Q-weights, are modified by the RL agent during training, as described in chapter II-A4.

4) Learning Algorithm: The learning algorithm is responsible for updating the Q-weights after each game. In this implementation, the only used learning algorithm is Q-learning, as shown in algorithm 1 (Watkins and Dayan, 1992). First, ϵ decides whether a random exploration action is taken or the best action based on Q-weights vector θ and feature vector ϕ. The next state s' and reward r are observed based on chosen action a_s. The temporal difference δ is then calculated, before being used to update θ. Due to implementation details, as described in chapter III-D, algorithm 1 shows only a single step of the algorithm.

5) Monte-Carlo Tree Search: MCTS is one of the more popular types of AI agents for games (Browne et al., 2012). Within a certain game state, it explores a tree where each child of a node is the state resulting from taking a certain action, and does this until a certain budget is reached, which, in this case, is a certain amount of time per turn. It has no prior information about the game and requires no pre-planning.

B. Games

The games used for testing were, in increasing order of complexity, *Tic-Tac-Toe*, *Dots and Boxes* and *Sushi Go*. While *Tic-Tac-Toe* is a well known game that does not need an introduction, *Dots and Boxes* and *Sushi Go* are briefly described.

1) Dots and Boxes: Dots and Boxes is a 2-player game played on a dotted grid. Players take alternating turns to draw a line between two dots, trying to draw the fourth line of a square (or box). When they do so, they score a point and gain another turn. It is a simple game requiring some strategic forward-thinking.

2) Sushi Go: In *Sushi Go*, 2-5 players take simultaneous turns, playing one card from their hand into their play area. The players then pass the remaining cards in their hand to the

next player. This means players first have imperfect information, that later develops into perfect information, testing the agent's ability to plan ahead, and not only consider its own board and what would benefit the agent itself, but also maybe taking away cards from the next player(s) instead. The game is played over three full rounds, so after each player played their last card for the first time, two more such rounds are played. There are a variety of cards that score points in different ways, requiring a strategic approach. (Gamewright, 2014)

C. Related Work

1) AlphaGo: A well-known example of a very successful AI in board games, *AlphaGo* uses MCTS to play *Go*. Developed by *DeepMind*, the MCTS algorithm makes decisions based on prior learned knowledge, which it attained through a deep RL approach using neural networks. Early versions of *AlphaGo* learned to play the game by studying human-played games in their training process. However, in 2017, *DeepMind* introduced *AlphaGo Zero*, which was solely trained on games against itself. (Holcomb et al., 2018)

2) Biasing MCTS with Linear Function Approximation Reinforcement Learning: The paper proposes a method to bias an MCTS algorithm's upper confidence bounds, which is used as part of the algorithm's decision which part of the tree to explore next. A LFA RL approach is used to train the biasing, using generalized features that can be applied to a wide variety of games, rather than domain-specific features. Only a small amount of training on these features lead to significantly improved results. (Soemers et al., 2019)

3) Q-Learning in Various Tabletop Games: Souidi et al. (2021) describe an approach using Q-learning to tackle multi-agent path planning in the classic board game *Ludo*. The paper found success in increasing cooperation between agents as well as having faster execution times per task.

He et al. (2019) explore applying Q-learning RL to train a self-learning agent in the game of *Xiangqi*, more commonly known as *Chinese Chess*, using a radial basis function neural network model. It results in an agent that can successfully play the game at a moderately high level.

Angelopoulos and Metafas (2021) construct an agent in the popular deck-building card game *Dominion* using Q-learning. State abstraction is performed to focus the agent on key aspects of the game, while limiting the number of unique states from becoming excessive. The agent achieves a 40-50% win rate in a 4-player game against various other AI agents, where an average win rate of 25% is expected for opponents of equal skill, thus training a successful agent.

4) Research in TAG: Much research in AI for modern board games has been done using TAG. Gaina and Balla (2022) introduce research in cooperation between AI and Humans in the cooperative board game *Pandemic*, providing a complex state and action space to explore cooperative behavior in.

Gaina et al. (2021) experiment with AI agents in the highly complex board game *Terraforming Mars*, with an incredibly rich dynamic action space, with every card in the game having different behavior and consequences, and a high variety of ways to play and win the game.

5) PyTAG: PyTAG is a Python API for interacting with TAG to apply popular RL methods in TAG, introduced by Balla et al. (2023). Python is a very adept programming language for AI and machine learning, with an extensive list of libraries to make use of for simplicity and efficiency, allowing the possibility to build powerful AI tools and, in this case, game agents. These libraries generally are specialized for approaches using neural networks rather than LFA. As such, implementing an LFA RL agent in Java directly in TAG provides added value.

III. Methodology

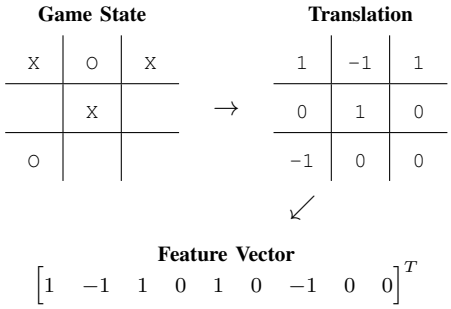
While TAG already has a multitude of different AI agents implemented into the framework, none of these agents learn about the game ahead of time, but are instead search-based agents that search the state-action space for the best action to take in each given state. The RL agent is different, as it needs to be trained on a specific game ahead of time, in order to play against other agents. For this, it requires an FV, representing the core features of a state.

A. Feature Vectors

1) State Abstraction: An FV is usually a manually crafted vector, that includes features about the game. These features can be anything that can be derived from a given state (and sometimes a given action) for a given player. The features could even be the exact components and elements that make up a game state, perfectly recreating every unique game state. However, mostly, an FV abstracts the game state space, by only focusing on core features about the game. What defines the core features depends on the person creating the FV, the level of play the agent should aspire to be at, and the complexity of the game and its state space. While this may result in several game states mapping to the exact same FV, this is not necessarily a bad thing, as those states might share inherent similarities that result in the same actions being good and bad.

2) Choosing Features: Each feature is assigned a value, based on the game state, a specific action and the player whose turn it is. The features can either be represented by a 1-hot vector, where different states of the feature each get their own entry in the FV, and at most one of those sub-features has value 1, while the others have value 0; or by arbitrary numeric values. Small subsections of the FV can be a 1-hot vector, without the entire FV needing to be. For example, a feature such as *"How many players are part of this game?"* might be better off as a 1-hot vector, as the feature would otherwise always have double the impact in a 4-player game as opposed to a 2-player game. Having one feature for *"Is this a 2-player game?"*, *"Is this a 3-player game?"*, etc. might be more effective, where each of them acts as a true-false feature. On the other hand, a feature such as *"How many cards do I have in my hand?"* might be better off as a single numeric feature, as then the agent can decide whether it prefers a lot

Fig. 1: An example of an FV for *Tic-Tac-Toe*



of cards or fewer cards in its hand, all with a single feature. Figure 1 shows a simple example FV for the game *Tic-Tac-Toe*, which has one feature for each of the nine cells on the board, and sets it to 0 if no player occupies it, -1 if the opponent occupies it and 1 if the agent itself occupies it.

B. Tabular Approach

Before implementing the LFA RL agent, a tabular agent was implemented, both as a stepping stone for testing functionality during implementation, and as another agent to compare against. While a tabular agent normally requires a full list of all possible states this implementation instead uses a map with the key being a string made of the concatenated values of the FV. For example, the *Tic-Tac-Toe* state seen in figure 1 would result in the string "$\{1; -1; 1; 0; 1; 0; -1; 0; 0\}$". Then, the Q-value is inserted as the value for that key into the map. The tabular approach then, when looking for the action to take, finds the FV from the current state and each of the possible actions, and looks for that specific FV in the map. If contained in the map, the Q-value of that entry is taken as the Q-value of that state and action, otherwise a default value is taken and added to the map for this previously unexplored state. Eventually, every unique state is mapped to a Q-value, and the agent learns to play the game perfectly, relative to the given FV. For complex states, this approach is very slow and not viable, for example mapping more than 90,000 states, and growing, after 10,000 games of *Dots and Boxes*.

C. Linear Function Approximation

As opposed to the tabular approach described in chapter III-B, the LFA approach has a different `get_action` function. As previously explained in chapter II-A3, this approach uses a linear function to approximate complex behavior. Each feature in the FV is assigned a weight, and the dot product of the FV and the Q-weights vector results in the Q-value. For the *Tic-Tac-Toe* FV from figure 1, the LFA RL agent has a Q-weights vector of size nine.

However, since the function is linear, it does not understand or draw any conclusions from the relationship between any of these nine features. One approach to tackle this problem is to include non-linear features inside the FV. For example, including one feature for each combination of two different cells will allow the RL agent to start learning about a cell in relation to any other cell. Further, including a feature for each combination of three distinct cells could be a good idea, since this will allow the agent to consider rows, columns and the two diagonals.

In *Tic-Tac-Toe*, this makes sense, since looking at a combination of three cells allows understanding the win condition. Chapter IV briefly discusses the effectiveness of each of these FVs.

```
1  {
2    "Metadata" : {
3      "Game" : "DotsAndBoxes",
4      "Seed" : 1692425893506,
5      "Type" : "LinearApprox",
6      "Alpha" : 0.0010000000474974513,
7      "Gamma" : 0.75,
8      "Epsilon" : 0.10000000149011612,
9      "Solver" : "Q_LEARNING",
10     "Heuristic" : "players.heuristics.WinOnlyHeuristic",
11     "FeatureVector" : "games.db.DBStateFeaturesReduced",
12     "NGamesTotal" : 819200,
13     "NGamesWithTheseSettings" : 409600,
14     "TimeTaken" : "23h 42m 02.9s",
15     "ContinuedFrom" : {
16       "Seed" : 1692377634196,
17       "NGamesTotal" : 409600,
18       "NGamesWithTheseSettings" : 409600,
19       "ContinuedFrom" : null
20     }
21   },
22   "Weights" : {
23     "POINT_ADVANTAGE" : 0.2855091730858064,
24     "POINTS" : 0.034057713698649184,
25     "THREE_BOXES" : -0.002159902471093063,
26     "TWO_BOXES" : 8.698242549985469E-4,
27     "ORDINAL" : -0.06361673051224066
28   }
29 }
```

Code Listing 1: Example JSON file written by the RL agent for *Dots and Boxes*, including metadata and Q-weights

D. Learning Algorithm

The learning algorithm described in chapter II-A4 was used. In the actual implementation within TAG, the process of choosing the action a_s happens while playing the game. The state s_t, the chosen action a_{s_t}, the possible actions A_{s_t} and the observed reward r_t are saved for every turn t, and the rest of the algorithm, from line 3 onwards, is then applied in reverse turn order after the game has been completed. The reverse turn order allows for quicker updated Q-weights when the only observable reward is winning or losing at a terminal game state.

E. Saving Q-Weights

The Q-weights resulting from training are written to a JSON file, including several bits of metadata about the training, as well as the Q-weights themselves. This metadata includes all parameters used, including the seed, specific classes for the FV, heuristic to determine the reward and more. A `ContinuedFrom` field is included that is filled when an existing one of these JSON files is used as a starting point for another training, and includes only the metadata that is different to the one being used at the top level. This can stack as a non-null `ContinuedFrom` field includes another `ContinuedFrom` field. An example of this is shown in code listing 1.

F. Experimentation

The environment of the experimentation was decided mainly by the time and hardware resources available. Tests were run on a 2021 M1 14" base model Apple MacBook Pro, with 8 CPU cores (6 "Performance" and 2 "Efficiency") and 16 GB of RAM. Values for this chapter were chosen with this in mind, exhausting the processing power of the computer without throttling the testing.

All games were played as 2-player games. The 2 players were both RL agents, sharing a Q-weights vector. After the game, they would, one after the other, update this vector. The Q-weights vector was saved in steps of $100 * \log_2$, in order to compare the performance based on number of games trained on. Due to the inherent randomness of RL, six different Q-weights vectors were trained on the same game independently, based on six unique random seeds. Training was further subdivided into phases. After each phase, the six agents were compared against each other, and the best one was used as the baseline to continue training from for the six new agents of the next phase. Phase 1 and 2 consisted of 409,600, or $100 * 2^{12}$, games, while the next three phases consisted of 819,200, or $100 * 2^{13}$, games each. *Tic-Tac-Toe* was played for three phases, for a total of 1,638,400, or $100 * 2^{14}$, games; *Dots and Boxes* for two, for a total of 819,200, or $100 * 2^{13}$, games; and *Sushi Go* for five, resulting in 3,276,800, or $100 * 2^{15}$, games. This training took around 2 hours, 24 hours and 16 hours, respectively. All comparisons, including between phases as well as against MCTS agents, were run over 1,000 games per matchup, evenly split between first and second player. Three MCTS opponents were compared against, differentiating in their time budget per turn (8ms, 40ms and 200ms).

G. Sushi Go

As *Sushi Go* is the most complex of the tested games, it was the one with the highest interest. As opposed to the other tested games, it required the creation of a custom FV, and was expected to provide the most clear results. Gamewright (2014) provides the rules of play.

The FV went through several iterations, starting with a simple list of cards in hand and on the board. After some testing, the agent did not perform very well, and so, over time, more and more features were added, in hopes to improve the performance. This included features combining relevant cards together, such as *"Is a Wasabi card in my play area and a Nigiri card in my hand?"*, or *"Is a Tempura card in my play area and another one in my hand?"*. Further, since cards not played in a turn are passed to the next player, most features are replicated for the opponent, looking at their play area and hand as well. This would hopefully allow the agent to sometimes play cards based denying the opponent a card, rather than only regarding its own play area, if it is deemed beneficial.

1) Pudding and Maki Cards: Sushi Go features two types of cards that score a fixed amount of points to the player(s) that have the most of the type of card. *Pudding* cards persist between rounds, and score the player with the most cards points at the end of the game, while *Maki* cards score the player with the most and second-most cards points at the end of each round, only considering players that have played at least one. These cards reward competing heavily but not hoarding.

2) Splitting Card Lead and Trail: To allow the RL agent to differentiate, the features *"Pudding/Maki difference to the opponent"* was split into a *lead* and a *trail* feature instead. For example, having 2 cards less than the opponent would make the *lead* feature 0 and the *trail* feature 2. The agent was expected to place a high negative value on the *"trail"* features and a small positive value on the *"lead"* features, understanding that having a trail should be heavily avoided, but a lead should be kept small. In reality, both values were kept almost identical in absolute value. This resulted in even agents with millions of games of training to hoard all *Pudding* cards against the MCTS agents, who simply let the agent waste its turns and have them.

3) Stepwise Lead and Trail: To address this issue, a final change was implemented for these cards, which was to have a step-wise 1-hot vector to represent the differences for these cards. The steps for *Pudding* cards were: a lead of 5 or more, 4, 3, 2 and 1; even; and a trail of 5 or more, 4, 3, 2 and 1. Similarly, for *Maki* cards, the steps were: a lead of 10 or more, 8-9, 6-7, 4-5 and 1-3; even; and a trail of the same steps as the lead steps. In a given state with a given action, only the single feature for each card type that was true for the inputs would get value 1, while all the others got value 0. This proved a very effective way to help the agent understand that a large card lead is not beneficial, and improved win rates by up to 10 times.

IV. Evaluation

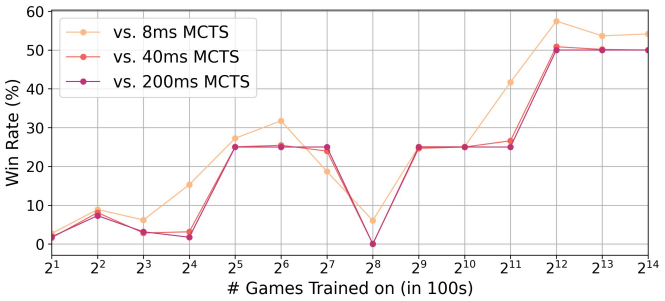
In all results, the win rate is the number of games won plus half the number of games drawn, divided by the total number of games. Thus, a 50% win rate, could signify all games drawn, as well as 50% of games won and 50% of games lost. The details of this are unimportant to these results, as the goal is to see how the RL agent competes with the MCTS agent, trying to get it up to the same level of play. If it successfully manages to draw every game, this is considered a success. From repeated testing, the error rate in the resulting values is expected to be 2% or less.

A. Tic-Tac-Toe

Tic-Tac-Toe is a very simple game with a small action space. A perfect way to play it can easily be determined, and a perfect player never loses, while two perfect players playing against each other always draw.

1) Feature Vector: The base FV used was one with nine features, one for each cell on the 3x3 grid, with a value of -1 for the cell being occupied by an opponent's piece, 0 for no placed piece, and 1 for the agent's own piece, as per the example in figure 1. Since the LFA RL agent cannot draw conclusions about relationships between different features, two additional FVs were used, referenced as the 2D and 3D FVs, respectively, with the base FV being the 1D FV. The 2D FV

Fig. 2: Tabular RL vs. MCTS for *Tic-Tac-Toe*



adds a feature for each unique combination of 2 different cells of the grid, while the 3D FV additionally includes a feature for each unique combination of 3 different cells.

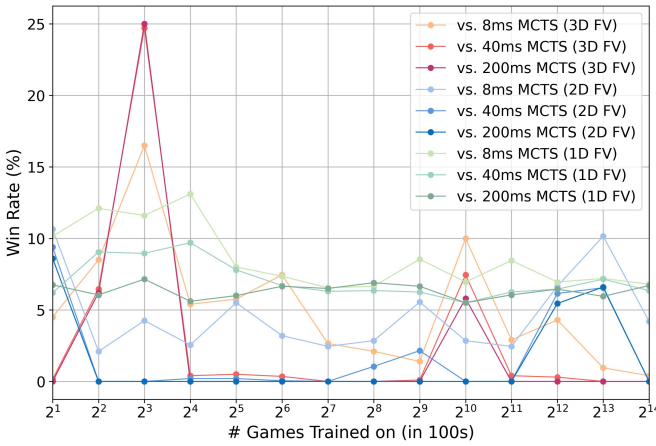
2) Tabular Agent: Because of the aforementioned criteria, the tabular approach described in chapter III-B is expected to learn to play the game perfectly using the 1D FV described in chapter IV-A1, since it can map each state to a value, as the state space is not particularly big. The number of unique states in *Tic-Tac-Toe* is 5,478, including the empty board. This number does not account for symmetrical or point-symmetrical board states. After only around 2,000 games, the tabular RL agent discovered every state, listing 5,399 states in its hash map. This number is slightly lower than the 5,478 existing states, as it excludes the empty board state, as well as the 78 full board states. The latter are excluded since TAG does not let the agent make a decision when there is only one action to take, so the last token being placed is never added to the hash map. Figure IV-A2 shows this agent's performance improving mostly steadily, besides a dip at 2^8, until every game against MCTS is a draw. Against the 8ms MCTS agent, the RL agent even achieves a win rate of above 50% from 2^{12} onwards.

3) Linear Function Approximation Agent: For the RL agent using LFA, *Tic-Tac-Toe* is already a significantly harder game than it is for the tabular agent. While it can prefer certain cells to other cells with the 1D FV, the 2D and, particularly, the 3D FVs were expected to provide more meaningful results. One agent was trained for each of the FVs. Figure 3 shows how all three of these agents performed versus the MCTS agents on one graph, where the red-yellow lines are the 3D FV, the blue lines the 2D FV and the green lines the 1D FV.

The performance of all the FVs is unimpressive and sporadic. The 1D and 2D FVs have overall more straight-lined graphs. The 1D FV starts at a bit of a high, but settles into a 6-7% win rate. This is expected, as the agent cannot learn a lot about the relationships between different cells with such a FV, and therefore plays more or less at random, where, in *Tic-Tac-Toe*, drawing occasionally is inevitable. Still, the higher win rates between 2^2 and 2^4 suggests that there is a better solution to be found than the one at 2^{14}.

The 2D FV is less stable, and spikes up and down, suggesting it can make more calculated decisions, but the calculations are based on too little data to really be good at the game, resulting in some sets of Q-weights to provide notably higher results than others, such as at 2^{13} and 2^{14}, respectively, while

Fig. 3: LFA RL vs. MCTS for *Tic-Tac-Toe*



generally resulting in a worse performance than the 1D FV.

The 3D FV agent achieves the overall highest highs of 25% win rate, but does so after only 2^3, which signifies a lucky constellation of features rather than a calculated approach. Interestingly, its toughest opponent at this point is the 8ms MCTS agent, which again suggests that it is more likely drawing games at random than it is doing so calculated. This is further confirmed as the agent quickly starts performing at a 0-10% win rate for the rest of the tests. A significant spike happens at 2^{10}, but is quickly lost again. This could be due to a number of reasons, such as overfitting (V-A1), an insufficient FV or simply not enough games being played to get the perfect FV. The high win rate at 2^3 suggests that a better set of Q-weights for the FV is certainly possible, even if the agent has not found it yet. Further, the 3D FV RL agent also often performs worse than the 2D, and particularly the 1D FV agent. This likely comes down to the aforementioned reasoning, that it is simply judging the game in the wrong way, ultimately leading to bad decisions that are worse than a random move would be.

B. Dots and Boxes

While *Dots and Boxes* is a more complex and strategic game than *Tic-Tac-Toe*, it is still simple in nature, allowing for a simple set of features to represent the state in a meaningful way. The FV used contains only five features: *"Point difference"*, *"points"*, *"number of boxes with 3 drawn lines"*, *"number of boxes with 2 drawn lines"* and *"current ranking"*. In any given state in *Dots and Boxes*, it is not important who drew which line on the board, which is why only looking at how many of each important type of line combination there were was expected to be enough to be a decent player. It should be noted that the MCTS agent's rollout length was set to 1 for these tests, as it lets it perform better in *Dots and Boxes*, specifically.

As figure 4 shows, the RL agent has the potential to play the game extremely well against the MCTS agent, with only such a small amount of features. However, it also has huge dips in the middle of the graph, showing that it is not quite as robust and reliable as expected and hoped. It does find its

Fig. 4: RL vs. MCTS for *Dots and Boxes*

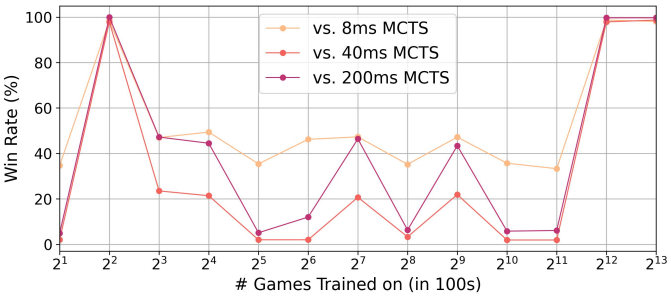
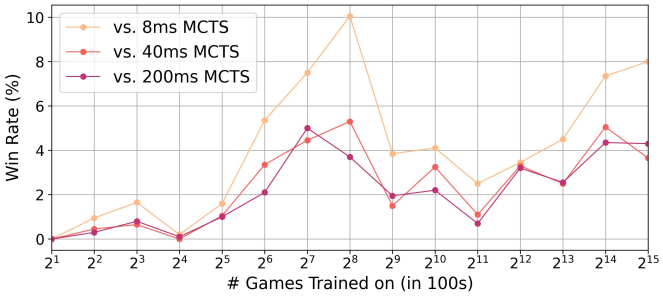


Fig. 5: RL vs. MCTS for *Sushi Go*



footing again at 2^{12}, but it cannot be assumed that it will keep such a high win rate if more games were used for training.

Nevertheless, achieving an at times 100% win rate, and an above 98% win rate against even the 200ms MCTS agent is an impressive feat, especially considering that it is making its decisions solely based on five features. The exact resulting Q-weights for these features can be seen in code listing 1.

C. Sushi Go

Since *Sushi Go* is a more complex game than both *Tic-Tac-Toe* and *Dots and Boxes*, agents with more training were expected to perform clearly and visibly better than agents with less training. Figure 5 shows the performance of the *Sushi Go* RL agent versus the three different MCTS agents.

In general, a clear upwards trend is visible from start to finish. This is interrupted by a spike in the middle section of the graph, between 2^6 and 2^9. Clearly, the agent found a way to play that is, compared to its other iterations, decent against MCTS agents, but during continued training moved away from this strategy again. In general, even though an upwards trend is visible, it is hard to say whether it would continue with more games or whether it is simply another spike that would eventually drop again. Further, the trend is very slight, never exceeding a 5% win rate against the 40ms and 200ms MCTS agents, and, besides the spike at 2^8, staying within an 8% win rate versus the 8ms MCTS agent. The earlier versions of the *Sushi Go* RL agent, discussed in chapter III-G never had a win rate above 1%, mostly achieving a 0.5% or lower win rates. Thus, the change discussed in chapter III-G3 was very effective in improving the style of play against the MCTS agents.

Looking at the specific FV, there are a few interesting features to look at, as shown in table group I. The values,

TABLE I: Q-weights of some features of the *Sushi Go* FV

Points	Point lead	Point trail
-0.010	0.789	-0.779

	Pudding					
	0	**1**	**2**	**3**	**4**	**5+**
Lead	0.043	0.089	0.105	0.090	0.036	-0.027
Trail		-0.012	-0.019	-0.012	0.011	0.045

***Feature:** 1 if row & column condition met, otherwise 0*	Missing from complete set		
	1 Tempura	**1 Sashimi**	**2 Sashimi**
No other considerations	-0.019	-0.090	-0.029
One in hand	0.018	0.074	0.006
One in next hand	0.020	0.060	0.004
One in both hands	-0.026	-0.030	0.034

which are the Q-weights for the features, should be interpreted as follows: A high absolute value means that the agent places a lot of emphasis on the specific feature. Values will generally range from -1 to 1, but can exceed either of these. The sign signifies whether the agent aims to maximize or minimize the feature, with a positive and negative weight, respectively.

The first table shows the point features, with the actual points the agent has on the left, and the point lead and trail to its opponent divided up into two features, much like what was described in chapter III-G2. This was done with the intention of letting the agent place different values on catching up and on holding a lead. The absolute values of these two features stayed around the same, implying a similar level of importance to the agent. The *"points"* feature received a negative value, which means that, regarding only the three point features in the table, the agent would prefer an action that denies the opponent a certain amount of points over an action that would have it gain those same number of points.

The *Pudding* features show the values given to each of the steps of chapter III-G3, with the 0-column representing no lead or trail, with the number of *Pudding* cards being even. The agent puts a lot more emphasis on the lead-features than the trail-features, implying it prefers to play for these cards than give them up. Ideally, the agent wants to have a 2 card lead, which seems reasonable, considering that it means the opponent generally cannot catch up in one turn. It then gives higher leads lower values, understanding that these are not rewarded effectively, as described in chapter III-G3.

The last table shows the set features. There are two cards that only score points in sets in *Sushi Go*. *Tempura* cards only score points for every 2 cards a player has, while *Sashimi* cards need a set of 3 to score points. As such, each column shows how many cards are missing from the next complete set. The table shows the Q-weights for features that combine a variety of conditions. While there are some surprising values, most values make sense and are in line with how a human would place importance on these values. One surprising value is that the agent, without other considerations, prefers missing two *Sashimi* cards over only missing one. The only sensible explanation is, that this value, in combination with the other

values of the same columns provided the best results for the agent. The other values are very in line with expectations.

Lastly, individual games were also opened in order to observe the agent's style of play. The agent generally made comprehensible moves that could be understood and explained given the game state. While an experienced player may see a better move, a novice player wouldn't be perplexed by a nonsensical move.

V. Future Work

The work done showed promise, but lacked consistency and robustness. Since the work was limited by time and resources, expanding the existing work might be beneficial, rather than applying it for different research questions.

A. Limitations

Many of the experiments ran into quite a few limitations, which leads to a sizeable amount of future work to be done for full testing.

1) Overfitting: A potential issue is overfitting, where the agent gets stuck in a solution too specific to its training environment. Since it always played against an identical version of itself, this could easily happen. This could be seen in earlier versions of the agent where the agent put a high value on hoarding *Pudding* cards, which makes sense if the opponent plays the same way, but got exploited by the MCTS agent, as described in chapter III-G2.

2) Parameter Testing: Due to time and resource constraints, parameters were tested at the start to find a suitable value, and then this value was taken throughout the experiments. Better and game-specific values for α, γ and ϵ, might enhance results. A decaying α value might have helped the agent stick to a good strategy, instead of diverting, as seen in chapter IV.

B. Alternative Training Methods

The training method chosen was one of training six individual agents over a series of games, taking the best one, and continuing training with that agent as a base. While this may provide good results, alternative methods for training the agents could be explored, maybe taking the best few agents from each phase, or simply letting agents train from start to finish on their separate Q-weights.

C. Training Against Better Opponents

One method that could be explored to further test the potential of these algorithms is training against other agents than only self-training. This could either come in the form of using differently trained agents with individual Q-weights, rather than having all agents of the same training session share Q-weights, or testing against other agents entirely, such as MCTS agents. In this case, the moves of the MCTS agent could also be recorded, and later used in the Q-learning algorithm the same way. This would then train the RL agent to approximate the MCTS agent's behavior, which might lead to better and more consistent results.

D. Different Feature Vectors

Different specific FVs could be tested for the games, to compare the performance of these FVs against each other. In particular, an FV for *Dots and Boxes* could be used that records the full game state, but only whether a line is drawn in each location or not, since it is irrelevant for the state who drew which line. This would be a decently large feature vector, but with which it should still be possible to work with. A tabular approach might still be impractical, since each line can have two states, drawn or not drawn, so for an amount of lines L, there are still L^2 unique states.

E. Generalized Feature Vectors

Generalized FVs are something to be researched after exploring more of the potential of this RL agent. Currently, an FV has to be created manually for each game. Either working on a generalized, or an automatically generated, feature vector, so the agent can quickly be applied to new games, could prove very beneficial. It would allow doing initial testing of the RL agent for a game, after which a custom FV could still be constructed.

VI. Conclusion

The RL agent proved to show a couple of different results. While it showed some promise, the graphs don't have the expected growth over time and instead have great inconsistencies and remain unpredictable. Constraints in time and resources meant that certain decisions had to be made in how the experimentation process and training process would be conducted. It is possible that conducting the experiments in different environments or with different parameters might result in improvements of the agent. While the results in *Dots and Boxes* had very high performant agents, the results in *Tic-Tac-Toe* and *Sushi Go* were more sporadic and generally not very impressive. However, the core ideas still implied that the potential of this method is not fully exhausted. Further, while it did not show great flexibility in strategy in *Sushi Go*, the agent still made generally meaningful moves that can be understood by humans, and are not simply random, incomprehensible actions.

References

Angelopoulos, G. and Metafas, D. (2021). Q learning applied on the board game dominion. In *Proceedings of the 24th Pan-Hellenic Conference on Informatics*, PCI '20, page 34–37, New York, NY, USA. Association for Computing Machinery.

Balla, M., Long, G. E. M., Jeurissen, D., Goodman, J., Gaina, R. D., and Perez-Liebana, D. (2023). Pytag: Challenges and opportunities for reinforcement learning in tabletop games.

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. (2012). A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43.

Gaina, R., Balla, M., Dockhorn, A., and Colas, R. M. (2020a). Tag: A tabletop games framework. AIIDE Workshops.

Gaina, R. D. and Balla, M. (2022). Tag: Pandemic competition. In *2022 IEEE Conference on Games (CoG)*, pages 552–559.

Gaina, R. D., Balla, M., Dockhorn, A., Montoliu, R., and Perez-Liebana, D. (2020b). Design and implementation of tag: A tabletop games framework.

Gaina, R. D., Goodman, J., and Perez-Liebana, D. (2021). Tag: Terraforming mars. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 17(1):148–155.

Gamewright (2014). *Sushi Go! Rules of Play*. Newton, MA.

He, W., Zhao, W., and Jiang, Y. (2019). Application of q-learning and rbf network in chinese chess game system. *IOP Conference Series: Materials Science and Engineering*, 677(2):022101.

Holcomb, S. D., Porter, W. K., Ault, S. V., Mao, G., and Wang, J. (2018). Overview on deepmind and its alphago zero ai. In *Proceedings of the 2018 International Conference on Big Data and Education*, ICBDE '18, page 67–71, New York, NY, USA. Association for Computing Machinery.

OpenAI, :, Berner, C., Brockman, G., Chan, B., Cheung, V., Debiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., Jozefowicz, R., Gray, S., Olsson, C., Pachocki, J., Petrov, M., d. O. Pinto, H. P., Raiman, J., Salimans, T., Schlatter, J., Schneider, J., Sidor, S., Sutskever, I., Tang, J., Wolski, F., and Zhang, S. (2019). Dota 2 with large scale deep reinforcement learning.

Soemers, D. J. N. J., Piette, E., and Browne, C. (2019). Biasing mcts with features for general games. In *2019 IEEE Congress on Evolutionary Computation*, pages 450–457.

Souidi, M. E. H., Maarouk, T. M., and Ledmi, A. (2021). Multi-agent ludo game collaborative path planning based on markov decision process. In Suma, V., Chen, J. I.-Z., Baig, Z., and Wang, H., editors, *Inventive Systems and Control*, pages 37–51, Singapore. Springer Singapore.

Watkins, C. J. C. H. and Dayan, P. (1992). Q-learning. *Machine Learning*, 8(3):279–292.

Ye, J., Li, X., Wu, P., and Wang, F. (2023). Action pick-up in dynamic action space reinforcement learning.