RLAIT: a Reinforcement Learning Artificial Intelligence Testbed

Jack Duvall

Abstract

Machine Learning is a growing field of Computer Science, and a particular subset, Reinforcement Learning, is particularly promising because it allows computers to learn a task without prior human experience. This paper puts forth a framework defining the abstract components of Reinforcement Learning and the progress made on filling in that framework with various problems to be solved (Tasks) and algorithms to solve them (Approaches). Focusing on gameplay-oriented tasks and the recently developed general-purpose algorithm AlphaZero, RLAIT shows some promise as a unifying framework for describing Reinforcement Learning.

Contents

1 Introduction 1
2 Methods 1
2.1 Framework 1
2.2 Hardware 2
2.3 Software 2
2.4 Task Implementations 3
Simplest Task • Uno • Othello • Go • Stratego
2.5 Approach Implementations 3
Random • Q-Learning • AlphaZero
3 Results and Discussion 4
4 Conclusions and Open Questions 5

1. Introduction

Machine Learning is a growing field of Computer Science, and a particular subset, Reinforcement Learning (RL), is particularly promising. By allowing computers to generate and learn from experiences on their own, many difficult tasks can be solved without any sort of human expert help. However, it can be challenging to contrast the performance of different RL algorithms on the same problem, especially if the algorithms were developed in different domains.

My solution, the Reinforcement Learning Artificial Intelligence Testbed (RLAIT), aims to solve that problem by providing a consistent, flexible interface to describe general tasks and accommodate RL algorithms learning those tasks. The primary goal of RLAIT is to provide a high-level representation of RL and associated tasks; my project is novel in that it focuses on the representation of multi-agent tasks.

Two notable existing projects with similar goals are the OpenAI Gym [3] and Facebook's Extensible, Lightweight Framework (ELF) [10]. OpenAI Gym is more targeted towards real-time, single-agent tasks, and ELF I found too cumbersome in its beta stage to use, so I ended up writing my own framework.

While I was not able to produce an AI with superhuman game-playing capabilities using my framework, I did gain a lot of experience and familiarity with Reinforcement Learning my implementing the algorithms myself. More than just applying them, I am now able to grasp why they do what they do.

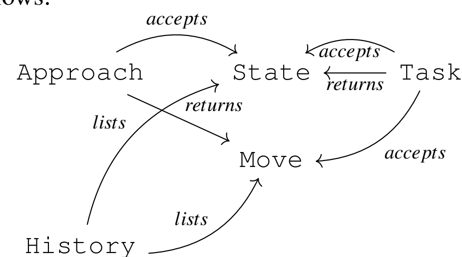
2. Methods

2.1 Framework

Before I got around to actually coding any RL Algorithms, I first designed the basic framework in which I would program. Here, I define some key vocabulary which I use in the rest of the paper:

Approach Any RL algorithm
Task Anything you want an RL algorithm to do
State All the information about the "world" of a Task
Move A description of how to move from one State to the next
History A list of States and Moves describing a complete run of a Task

States, Moves and Histories are just data containers which Approaches and Tasks act upon. A relationship diagram is as follows:



For my framework, I mostly focused on board games as tasks. This is because I wanted to try the AlphaZero algorithm [9], which claimed to be generalizable to any Task.

Approach API

`approach_name` A string containing the name of the current Approach.
`init_to_task(state)` Customizes an Approach to work on a specific Task.
`get_move(state)` Gets a move the Approach will play for the passed in State.
`save_weights(filename)` Saves the Approach's custom learned data (Neural Network weights, Q-table) to a file.
`load_weights(filename)` Restores the custom learned data from a file.
`save_history(filename)` Saves the current observed History to a file.
`load_history(filename)` Loads a previous observed History from a file.
`test_once()` Runs a single bout of self-play, appending to the current History.
`train_once()` Runs a single bout of training. For different Approaches, this can have different side-effects, ranging from calling `test_once` multiple times and automatically calling `save_weights` to only observing one game.

Figure 1. The methods and properties each Approach must implement

As I was developing, these APIs went through a few revisions to match the full functionality I needed. They ended up staying fairly close to my original plan, though.

2.2 Hardware

During my research, I had access to a couple of powerful machines to run my code on.

1. ASM
   - Operating System: Arch Linux
   - RAM: 12 GB
   - CPU: Intel i7 920 @ 2.67 GHz, 4 physical, 8 logical cores
   - GPU: 2x NVIDIA GTX TITAN
2. Duke
   - Operating System: Ubuntu Linux 18.04 LTS
   - RAM: 16 GB
   - CPU: AMD Ryzen 7 1700 @ 3 GHz, 8 physical, 16 logical cores
   - GPU: 2x NVIDIA 1080Ti

I was extremely fortunate to be able to use these machines for unlimited time at no cost, being able to configure them however I wanted. Still, I was not expecting to match the cutting-edge results, which use thousands of nodes just like these.

2.3 Software

For my programming language, I chose Python due to my familiarity with it and its large repository of Neural Network utilities. For the majority of my research, I used the following libraries:

- **CUDA** 9.0
- **cuDNN** 7.2
- **Tensorflow** 1.12
- **Keras** 2.2.4
- **Python** 3.6.8

I used the Anaconda virtual environment software to in-

Task API

`task_name` A string containing the name of the Task.
`num_phases` How many phases, ie different sets of rules defining States and Tasks returned, are contained in the Task. An example is the setup phase versus playing phase of Stratego, or the different betting phases of Poker.
`num_players` How many independent agents this Task will have.
`empty_move(phase)` Returns the default Move for a given phase.
`empty_state(phase)` Returns the default State for a given phase.
`iterate_all_moves(phase)` Yields all possible Moves for a given phase, even illegal ones.
`iterate_legal_moves(state)` Yields all the legal Moves for a given State.
`get_legal_mask(state)` Returns a binary mask which can be applied to the output of `iterate_all_moves` to filter out the legal ones. The reason for providing so many different interfaces to do the same thing is to accommodate all Approaches.
`get_canoncial_form(state)` Gets the canonical form of a state, normalized for how the current player would "see" the board as though they think they are the first player.
`apply_move(state, move)` Returns the updated State after the Move is applied. Will throw an error if the Move is illegal.
`is_terminal_state(state)` Checks if the State is terminal, i.e. the game is over.
`get_winners(state)` Returns the set of all winning players of a terminal State. If there are no winners or the State is non-terminal, returns an empty set.
`state_string_representation(state)` Returns a unique string representing a State. Suitable for acting as a key in tables/mappings.
`move_string_representation(state, move)` Returns a unique string representing a Move to be applied to a specific State. Again suitable for acting as a key. Whether or not this differs by phase or State depends on the Task type.
`string_to_move(move_str, state)` Reconstructs the Move for a given State based on its string representation.

Figure 2. The methods and properties each Task must implement

State and Move API

Note: `States` and `Moves` are both subclasses of `numpy.ndarrays` and are mostly used as such. Task implementations may also extend `States` and `Moves` to store their custom data as well, but the data in the array is all any `Approach` will see.
task_name String containing the name of the corresponding Task
type Integer describing how the data inside the `ndarray` is laid out. Has 4 options: {`0: 'flat', 1: 'rect', 2: 'deepflat', 3: 'deeprect'`}
phase Integer in the range $[0, \texttt{Task.num_phases})$ corresponding to the current phase.
next_player Integer in the range $[0, \texttt{Task.num_players})$ corresponding to the player who will make the next `Move`

Figure 3. Standard properties for each State and Move

stall these packages and all their dependencies, ensuring a consistent, stable, and reproducible environment across any system.

2.4 Task Implementations

In addition to defining my API, I also used it to implement various games for AIs to learn. The following sections are ordered by increasing complexity.

2.4.1 Simplest Task

Just about the simplest Task I could think of that wouldn't be *too* easy. In it, the first player plays a number in the range 1 to 4, which the second player then has to match. If they match, the second player wins; if they don't, the first player does.

Having something this simple allowed me to verify my Approaches worked at all, and debug them if they didn't.

2.4.2 Uno

Uno is a card game supporting a near-arbitrary number of players [5]. Each player starts with a hand of 7 cards. On their turn, they can play a card that matches the color or number of card in the center pile, or draw a card. The first person to play all the cards in their hand wins. I used a slight variation to the rules in that there are no special cards and no playing a card after you draw.

Uno was my only game that supported more than 2 players, helping me make sure my Approaches did generalize to multi-agent games. Uno's hidden information (you can't see what cards your opponent has) but simple rules also helped test hidden information compatibility.

2.4.3 Othello

Othello is a 2-player board game played with reversible black and white pieces [2]. It can be played on any NxN board where N is an even number greater than 2. Starting from a small 2x2 checker formation in the center, the players take turn bracketing each others pieces to flip them over until no more moves can be made. The winner is the player with the most pieces on the board at the end of the game.

Othello is a classic AI problem, one I already had a template set up for too. Implementing it as a Task was fairly straightforward too once I had the Task API completed.

2.4.4 Go

Go is a complex strategy game played on large board sizes, up to 19x19 [1]. Its rules and strategies are fairly complicated, stemming around the idea of capturing territory with your own stones while simultaneously preventing your opponent from doing the same. Notably, AlphaZero claims to achieve superhuman performance on this Task in only a few days [9].

I didn't expect to achieve superhuman performance on my machines. Instead, I included Go just to see if the Approaches could learn any of it at all.

2.4.5 Stratego

Stratego is a 2-player board game [4]. Both players set up their army, then wage war until the enemy flag is taken or all the enemy pieces are captured. Each player can see the value of their own pieces, but not the opponent. Pieces with higher power generally take pieces of lower power, except for the 1 piece being able to take the 10 piece. Pieces can move 1 space in any compass direction if it is unoccupied, or attack in a compass direction. There are also a few extra rules, like 3s being able to destroy bombs, stationary pieces that destroy any other piece, and 2s being able to move more than once in a given direction.

Originally, testing hard hidden-information problems was my main goal. That is why I started out with Stratego.

Implementing Stratego helped me expand and refine my API. The canonical form function and the move iteration functions were not in my original draft, but were necessitated by the complexity of the task. I also had to get creative with my handling of moves; encoding a starting and ending destination cleanly turned out to be a challenge.

2.5 Approach Implementations

Implementing Tasks meant nothing if I didn't also have Approaches to test them with. The sections are again ordered by increasing complexity.

2.5.1 Random

A random Approach is relatively straightforward: given a list of all the allowed Moves for a given State, return a random one. This just let me programmatically test my other AIs against a baseline.

2.5.2 Q-Learning

Q-Learning is an algorithm that takes in samples of State-Move pairs along with the "reward" for that pair and computes a policy, stored in the function Q, that is guaranteed to return the best Move given enough examples [6]. The Q update

procedure I used goes like this:

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

The rationale behind this is to iteratively update the function as it sees new things, while still "remembering" what it's seen in the past. It doesn't matter too much how it sees new examples, only that it sees infinitely many of them [6]. Intuitively, this makes sense because at the algorithm goes along, it keeps being able to optimize for both the future rewards it's seen as well as the specific action awards it knows.

2.5.3 AlphaZero

AlphaZero is a clever form of Deep Q-Learning, a category of Reinforcement Learning that uses Neural Networks to approximate a Q-table. What's special about AlphaZero is how it uses Monte Carlo Tree Search (MCTS) and Evolutionary Algoritm (EA) ideas to achieve its superhuman performance on tasks such as Go, Chess, and Shogi [9].

The bulk of my research was centered around AlphaZero, specifically getting it to work with the RLAIT framework in Python. I based my work off a previous general AlphaZero Python implementation [8], with heavy modifications.

As explained in [7] and [9], the training of AlphaZero is divided into repeatable parts called iterations. Each iteration contains 3 distinct steps:

- Self-play
- Neural Network Training
- Arena play

Self-play In this step, AlphaZero generates all of its training data. The finer details of how AlphaZero searches the State-Move tree to find the best move is described later in the Monte Carlo Tree Search section. The number of games to play against itself can be configured, but it must be divisible by the number of players in a task (so that it runs as each player). The arena play step can also be skipped entirely with minimal loss of strength [9].

Neural Network Training After each self-play, the batch of training data generated is added to the cumulative training history. The number of epochs, histories to remember, and optimizer can be configured when initializing the Approach.

Arena Play Arena play is one of the more important steps distinguishing AlphaZero from other algorithms. In order to make sure there are no regressions in model strength, AlphaZero plays the models from before and after training a configurable number of times, usually less than the self-play unmber, using the majority winner for the next iteration.

Monte Carlo Tree Search Another important part of AlphaZero (and its immediate predecessors) are it's modification to Monte Carlo Tree Search (MCTS) that effectively lets it learn "how" to search the state space. Each step of MCTS consists of traversing the State tree until it hits an unexplored State, then propagating the updated values back through the tree [9]. At each searched State, AlphaZero chooses the next move to explore by maximizing equation 2, as explained in [7]:

- s: current state
- a: action from state
- $Q(s,a)$: expected reward from action
- $N(s,a)$: number of times action was explored
- $P(s,\cdot) = \vec{p}_\theta(s)$: expected reward as returned by Neural Network
- c_{puct}: exploration factor, default set to 1.0

$$U(s,a) = Q(s,a) + c_{puct}\cdot P(s,a)\cdot \frac{\sqrt{\Sigma_b N(s,b)}}{1+N(s,a)} \tag{2}$$

Q and N are initially set to 0 for all a. Each time the network does take an action, producing state s', the value $v_\theta(s)$ returned by the neural network is propagated up the Q tree in a min-max fashion, unless there is a terminal state (+1 for win, -1 for loss).

At the end of exploration, we have a vector $\vec{\pi}(s)$ representing the normalized counts of exploration from any State in the History, $N(s,\cdot)^{1/\tau}/\Sigma_b(N(s,b)^{1/\tau})$. τ is a variable representing how likely the network is to explore random moves. One implementation of AlphaZero sets $\tau = 1$ for the first few moves (again, configurable exactly how many) to get normalized move counts, allowing it to potentially try new strategies, then to $\tau = 0$ to simply get the maximum counted move. $\vec{\pi}(s)$ and the eventual game outcome are added to the training data to train $\vec{p}_\theta(s)$ and $v(s)$ respectively [7].

A sample AlphaZero Neural Network architecture that I applied is shown in figure 4

3. Results and Discussion

While I spent a long time programming all these Tasks and Approaches, I did not have too much time to do much testing. In spite of this, I did get a few good graphs.

Figure 5 shows some preliminary results from testing the implementation of AlphaZero in [8] on 8x8 Othello. This was before it was completely integrated into RLAIT. The left chart shows the number of wins it attained per iteration versus a suite of $\alpha-\beta$ tree search AIs written by humans. The right charts shows the amount of the board AlphaZero was able to capture in the end. The trend is weak, but AlphaZero does appear to get better as it learns more.

Figure 6 shows AlphaZero vs Q-Learning on Othello 6x6 after both Approaches and the Othello Task were incorporated into RLAIT. As you can see, there does not appear to be any trend. Neither AlphaZero nor Q-Learning improve against the other as they train more.

These results are not indicative of AlphaZero or Q-Learning as algorithms, because I may have implemented them or Othello incorrectly. Without proper units tests (which I unfortunately did not write), it is hard to say. I had an unexpectedly tough time rooting out such implementation errors, which is why this report does not contain more results; too long was spent fixing small errors, leaving too little time to fully train more RL Approaches.

4. Conclusions and Open Questions

Reinforcement learning is a hot subfield of Machine Learning due to its massive potential to produce extremely intelligent algorithms without the upfront effort of collecting human expertise. This project was a learning experience, teaching me the basics of RL and how it can be represented in an abstract, object-oriented manner, all by building a framework around those ideas. This framework contributes a more technically definition of the object names like State, Task, and Move already in use. It also provides a well-, if not fully-developed Python framework to support those definitions. Some general improvements to AlphaZero, like a value vector and random play initialization, were also introduced.

One open question that I hoped to answer with this research, but didn't get around to, was the question of whether or not AlphaZero could adapt to hidden information games. Areas of future study include fully developing RLAIT to answer the previous question and implementing the AlphaStar Approach as put forth by DeepMind in [11].

Acknowledgments

I would like to thank my lab director Patrick White for being extremely supportive and helpful, both this year and all the time I've known him. I would also like to thank my old laptop for it's four years of generous service. I know I pushed you hard, buddy, but you can rest now. Finally, thank you to the Computer Systems Lab and TJHSST as a whole for providing me one of the greatest learning experiences of all time.

References

[1] American Go Association. *Official AGA Rules of Go*. 1991. URL: https://www.usgo.org/sites/default/files/pdf/completerules.pdf.

[2] United States Othello Association. *The Official Rules of Othello*. 2017. URL: http://usothello.org/misc/USOA_Tourn_Rules.pdf.

[3] Greg Brockman et al. *OpenAI Gym*. 2016. eprint: arXiv:1606.01540.

[4] Milton Bradley Company. *Stratego Instructions*. 1996. URL: https://www.hasbro.com/common/instruct/Stratego.pdf.

[5] Mattel Inc. *Uno*®. 2001. URL: https://service.mattel.com/instruction_sheets/42001pr.pdf.

[6] Fransisco S. Melo. "Convergence of Q-Learning: A Simple Proof". Posted on a user site for the Institute for Systems and Robotics, Instituto Superior Técnico, Lisboa, Portugal. URL: http://users.isr.ist.utl.pt/~mtjspaan/readingGroup/ProofQlearning.pdf.

[7] Surag Nair. *A Simple Alpha(Go) Zero Tutorial*. https://web.stanford.edu/~surag/posts/alphazero.html.

[8] Surag Nair. *AlphaZero General*. https://github.com/suragnair/alpha-zero-general. 2019.

[9] David Silver et al. "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". In: *Science* 362.6419 (2018), pp. 1140–1144. ISSN: 0036-8075. DOI: 10.1126/science.aar6404. eprint: https://science.sciencemag.org/content/362/6419/1140.full.pdf. URL: https://science.sciencemag.org/content/362/6419/1140.

[10] Yuandong Tian et al. "ELF: An extensive, lightweight and flexible research platform for real-time strategy games". In: *Advances in Neural Information Processing Systems*. 2017, pp. 2656–2666.

[11] Oriol Vinyals et al. *AlphaStar: Mastering the Real-Time Strategy Game StarCraft II*. https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/. 2019.

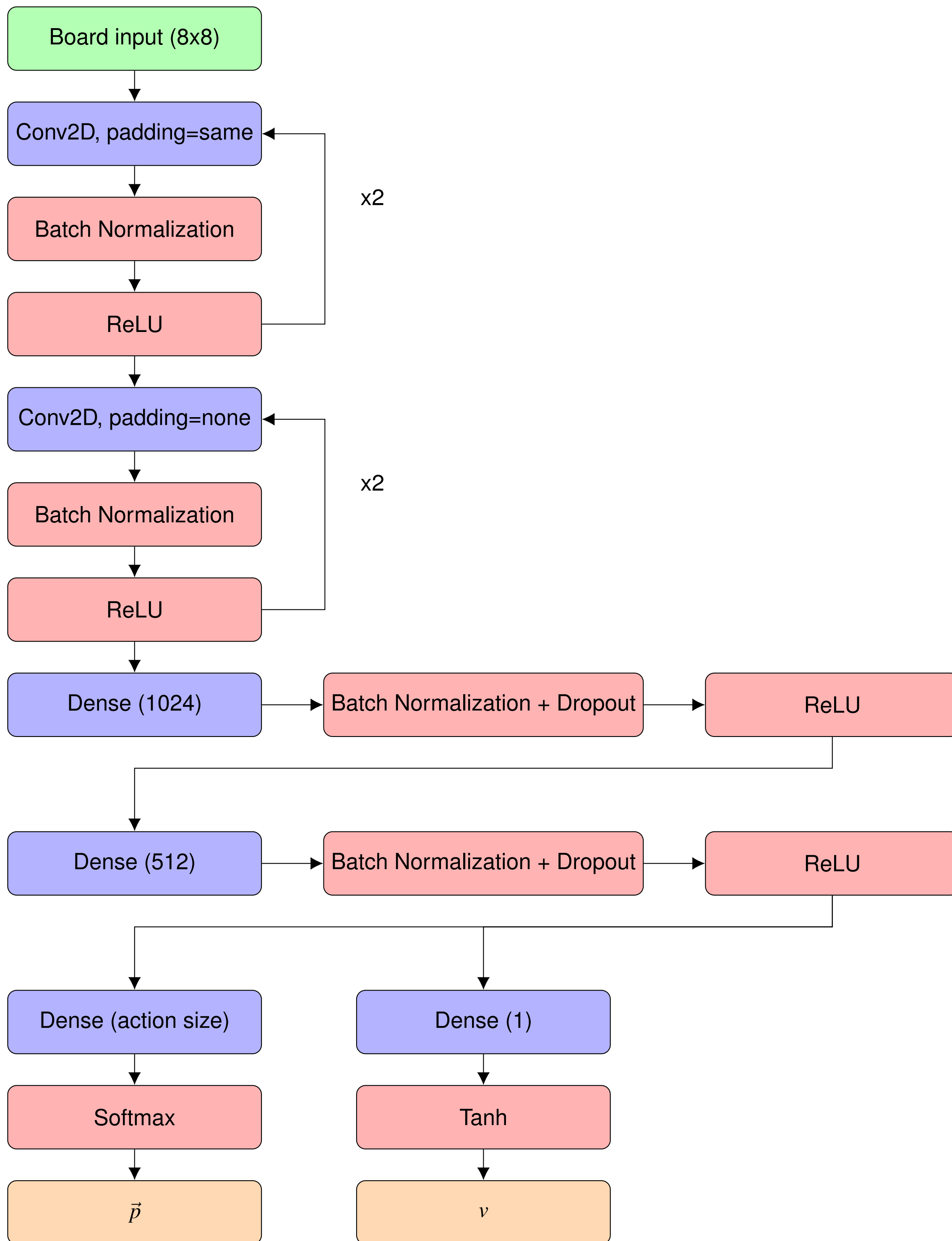


Figure 4. Default Open-source AlphaZero Othello Neural Network

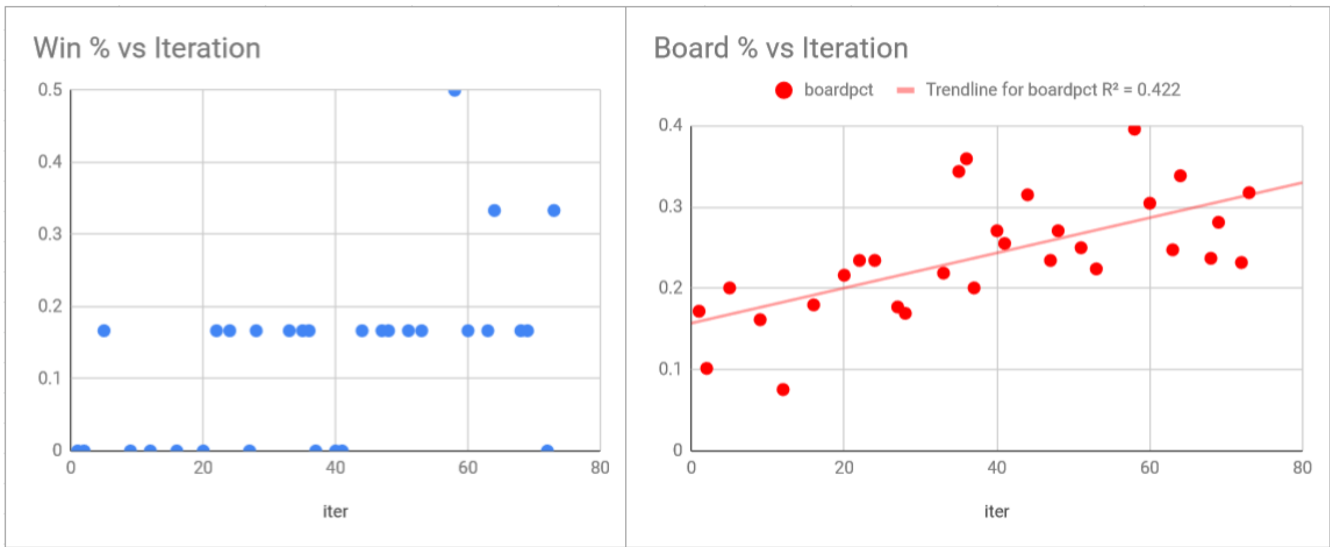


Figure 5. First Graph

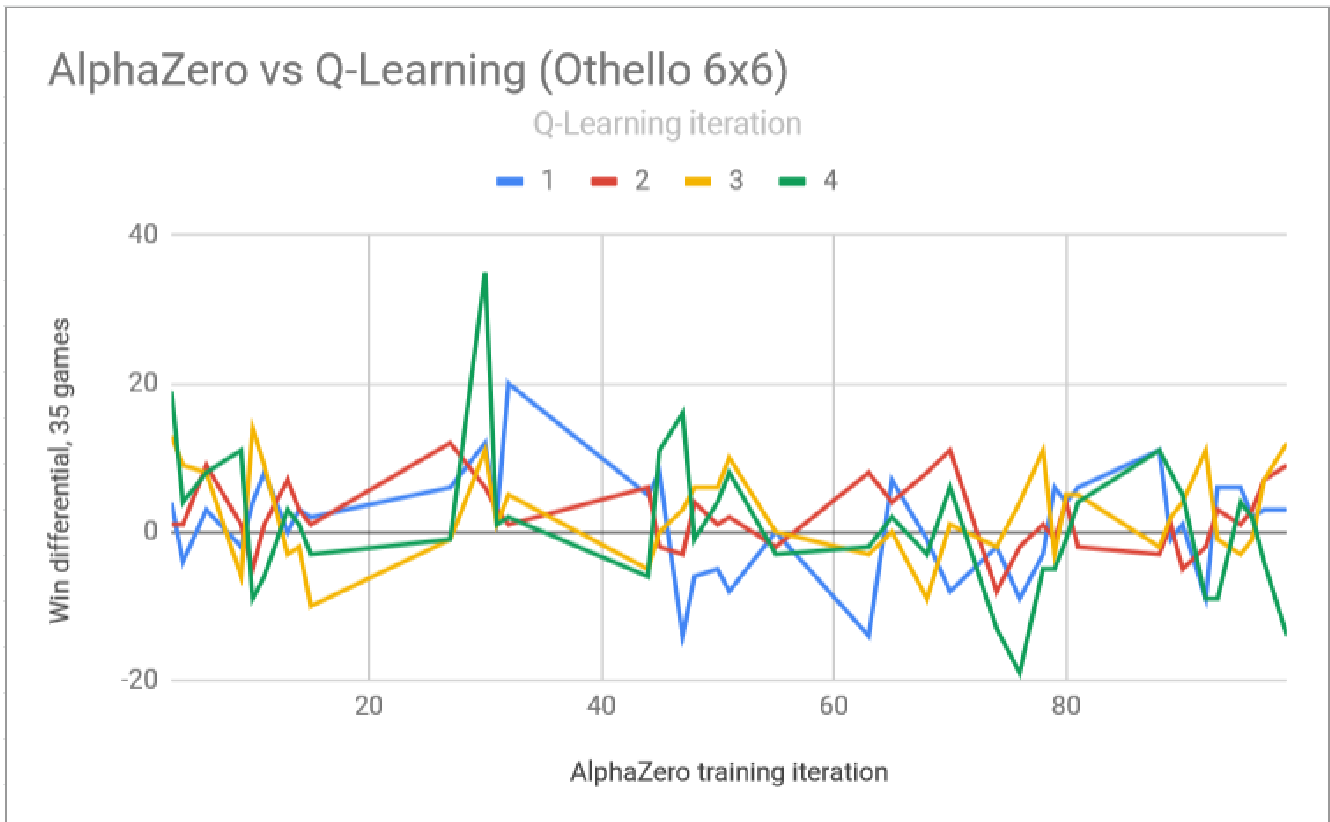


Figure 6. Second Graph