Development of a Q Learning Agent Using Sushi Go

Manveer Kaur, Dan Nygard

Final Project Report
CSCI 724

1. INTRODUCTION

Sushi Go is a card game for 2-5 players in which the goal is to score the most points by drafting a set of cards. Each player is dealt an initial hand, from which they select one card. All players reveal their chosen card simultaneously, and each player passes their hand to their left. Play proceeds until all cards have been chosen.

Although a relatively new game (first published in 2013), some have begun to note Sushi Go's potential as a tool for developing and testing artificial intelligence agents. As noted by (Soen, 2019), "Sushi Go is a zero-sum, full information, deterministic game, with the additional complication of initial hands being randomized and sampled from a fixed distribution.". Soen's article also notes the challenges involved in the game: a random initial setup (in other words, a different starting hand each game) which makes it challenging to evaluate strategies, and no "spatial locality", which means that there is no location to root the state to as in Chess or Go.

Our research focused on the Sushi Go implementation created by (Liu, 2020) and outlined in his article "Reinforcement Learning and Sushi Go!". We elected to adapt and test variations of Liu's Q Learning Implementation against the AI agents he created. Our goal was to create a stronger Q Learning agent, one that can outperform Liu's Q Learning, Deep Learning, and Rules-Based agents.

1.1 What is Q Learning?

As explained by (Deeplizard, n.d.), Q Learning is "a reinforcement learning technique used for learning the optimal policy in a Markov Decision Process." In a Markov Decision Process, an agent attempts to maximize the total amount of rewards it receives for its actions over time. It does this by following an iterative process: it begins in a state, chooses an action, receives its reward, and then repeats the process (State, Action, Reward) from the new state.

In Q Learning, the objective is to find the optimal policy such that the expected value of total rewards over all successive steps is the maximum achievable. When taking an action, it receives a reward and then relies on the below formula to determine the new Q Value that will be assigned to that state:

$$q^{new}(s,a) = (1-\alpha)\underbrace{q(s,a)}_{\text{old value}} + \alpha\overbrace{\left(R_{t+1} + \gamma \max_{a'} q\left(s',a'\right)\right)}^{\text{learned value}} \tag{1}$$

In the above, to calculate the value a learning rate (alpha) is used along with a decay rate (gamma). The learning rate determines how quickly an agent abandons the previous Q value for a given state-action pair in favor of the new Q value. The higher the learning rate, the more quickly the agent will adopt the new value (and vice versa). In the above equation, the previously-stored Q value (zero if not previously stored) is multiplied by (1 - alpha) and added to the product of the newly-learned value and alpha. The gamma value is applied to each step, and the result is that later steps in a series of decisions receive lower values (and change more deliberately) than Q values in more immediate steps.

2. METHODS

We started with Liu's Sushi Go implementation, which includes a standard Q Learning agent, a Deep Learning agent, a Rule-based agent (which selects cards based on a fixed ranking system), and a Random-selection agent. Liu also includes a test suite in which agents can be set against each other to train and evaluate over a series of a large number of games.

Figure 1: Sushi Go - Quick Game Summary (from Sushi Go Instruction Manual).

The Liu implementation is missing the "Chopsticks" card from the original game, as it is too computationally expensive to implement and still run the large number of games we require. There was an attempt at including the "Pudding" card, but we removed this as we did not trust that it was providing accurate scores. In addition, while Sushi Go is traditionally played over three rounds, our tests focused on single-round games. Our belief is that this initial, simplified environment allows us to focus on developing Q Learning agents without worrying about processing speed or game implementation details.

One other note: In standard Q Learning, an agent takes an action from a state and receives a reward. In Liu's implementation, all actions are made during one game and stored in a stack. At the end of the game a reward is applied (+1 for a win, -1 for a loss) and then each state is popped from the stack and given the reward (as modified by the learning rate and decay gamma). One interesting aspect of this is that the decay gamma is highest at the earliest actions (in other words, the Q values are lowest here), whereas in standard Q Learning the opposite applies. Liu's rationale is that this will "allow the agent to learn early states more slowly and carefully." We agreed with this rationale but future investigation into this (for example, applying the highest decay gamma to the later game states) may yield positive results.

Our tests focused on adjusting the learning rate, adjusting the reward function, implementing a database-backed Q-Learning agent, and improving the game state (which also requires a database to handle the large amount of possible state combinations). We used modified versions of the initial test suite provided in Liu's code base.

2.1 Learning Rate

Liu's implementation uses a .01 learning rate, which appears at first to be very low. We thought it was worth testing higher learning rates to see if this affected performance. Many variations were tried and are outlined in the results section.

```
for i in tqdm(range(100)):
    # Train
    p1.exp_rate = 0.3
    p2.exp_rate = 0.3
    p3.exp_rate = 0.3
    p1.lr = 0.01
    p2.lr = 0.02
    p3.lr = 0.03
    state.play_games(200)
```

2.2 Reward Function

Liu's Q Learning agent gives a reward of +1 to winning games, and, for multiplayer games, this number increases (for example, +2 for winning a three-player game, +3 for a four player game). A -1 reward is given to players at the end of a losing game. Our aim was to modify this reward with the goal of strengthening our agent. Instead of winning and losing being the main factor, we focused here on setting a modified version of the final score as the reward. We ran one test which passed (score / 10) as the reward, and another that factored in winning and losing by giving +1 (or more for multiplayer) to (score / 10) in winning games, and -1 to losing games.

```
for i, p in enumerate(self.players):
    # Feeds score-based reward
    p.feed_reward_score(p.get_score() / 10)
    if self.scoreboard[i] == max_score:
        self.stats[i] += 1
        p.feed_reward(max(1, len(self.players) - 1))
        # Feeds score-plus-win/loss reward
        p.feed_reward_score_plus_minus(p.get_score() / 10 + (max(1, len(self.players) - 1)))

    else:
        p.feed_reward(-1)
        p.feed_reward_score_plus_minus(p.get_score() - 1)
```

2.3 Database

The traditional approach for writing a Q Learning algorithm is four-fold. The first step is to determine the size of Action and State sets, followed by defining the learning rate, decay rate, rewards and other variables. Once we have these, the next step is to create a q-table where q-values are stored for the corresponding state/action pairs. The agent is then trained with the help of the q-table to play games. The challenge with this game is that the number of states is incomprehensible. Liu considers the combination of cards on the board as a state. In order to make a strong Q Player, we decided to use a database to store states and their q-values. As the game proceeds, new state/value pairs are added to the table. In case a state exists in the database, the TD Rule is applied to generate the new q-value for that state, which is then updated in the database. The Q Player retrieves the state/value pairs from the database at the beginning of the game and uses them to make decisions.

```
def addDataToDb(self, curr_state, q_value, table):
    db_cursor=self.db_cursor
    selectQuery = 'Select count(*) from '+table +' where state = %s;'
    db_cursor.execute(selectQuery,[curr_state])
    isStateInDb = str(db_cursor.fetchone())
    updateQuery = 'update '+table+' set q_value = %s where state = %s; '
    insertQuery = 'Insert into '+table+'(state,q_value) values(%s,%s); '
    if isStateInDb != '(0,)':
        db_cursor.execute(updateQuery,[q_value,curr_state])

    else:
        db_cursor.execute(insertQuery,[curr_state,q_value])
```

2.4 State Improvements

In the Liu implementation, each game state is a python list representing the player's tableau of the cards they have already chosen. In the below, the states of a winning game are unwound (the end game state is at the top), and in the dictionary at the bottom each state is stored and given the reward (note how the decay gamma decreases the reward given to early game states.

```
(sushi_go) nygard@nygard-Inspiron-3501:~/sushi_go$ python State.py
Dan
State: [1, 0, 2, 1, 1, 0, 0, 0, 2, 1, 2]
State: [1, 0, 2, 1, 1, 0, 0, 0, 1, 1, 2]
State: [1, 0, 2, 0, 1, 0, 0, 0, 1, 1, 2]
State: [1, 0, 2, 0, 1, 0, 0, 0, 0, 1, 2]
State: [1, 0, 2, 0, 1, 0, 0, 0, 0, 1, 0]
State: [1, 0, 2, 0, 0, 0, 0, 1, 0, 1, 0]
State: [1, 0, 2, 0, 0, 0, 0, 0, 0, 1, 0]
State: [0, 0, 2, 0, 0, 0, 0, 0, 0, 1, 0]
State: [0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0]
State: [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
State: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
{'[1, 0, 2, 1, 1, 0, 0, 0, 2, 1, 2]': 0.9099999999999999, '[1, 0, 2, 1, 1, 0, 0, 0, 1, 1, 2]': 0.819000
0000000001, '[1, 0, 2, 0, 1, 0, 0, 0, 1, 1, 2]': 0.7371, '[1, 0, 2, 0, 1, 0, 0, 0, 0, 1, 2]': 0.6633900
000000001, '[1, 0, 2, 0, 1, 0, 0, 0, 0, 1, 0]': 0.5970510000000001, '[1, 0, 2, 0, 0, 0, 0, 1, 0, 1, 0]'
: 0.5373459000000002, '[1, 0, 2, 0, 0, 0, 0, 0, 0, 1, 0]': 0.4836113100000001, '[0, 0, 2, 0, 0, 0, 0, 0
, 0, 1, 0]': 0.4352501790000001, '[0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0]': 0.3917251611000001, '[0, 0, 1, 0,
 0, 0, 0, 0, 0, 0, 0]': 0.35255264499000005, '[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]': 0.31729738049100015}
```

Figure 2: Standard game state.

The problem with this use of state is that it depicts only one part of the game environment: the cards a player has chosen. In our modified version, we add to our state the cards that are in the player's hand. In the below, each state contains the player's current tableau of chosen cards on the left, plus their current hand on the right. Note how the hand of cards gets smaller with each turn.

```
(sushi_go) nygard@nygard-Inspiron-3501:~/sushi_go$ python State.py
State: [[1, 0, 2, 1, 0, 0, 0, 1, 1, 0, 6], []]
State: [[1, 0, 2, 1, 0, 0, 0, 0, 1, 0, 6], [5]]
State: [[1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 6], [0, 4]]
State: [[1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 5], [2, 5, 5]]
State: [[1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 5], [0, 0, 4, 7]]
State: [[1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 5], [2, 2, 3, 5, 5]]
State: [[0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 5], [0, 0, 2, 4, 4, 7]]
State: [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5], [0, 1, 2, 2, 3, 5, 5]]
State: [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3], [0, 0, 2, 4, 4, 5, 7, 7]]
State: [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1], [0, 1, 2, 2, 3, 5, 5, 5, 8]]
State: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
{'[[2, 0, 1, 1, 0, 0, 0, 0, 1, 2, 7], []]': 0.1, '[[2, 0, 1, 1, 0, 0, 0, 0, 0, 2, 7], [6]]': 0.09000000
000000001, '[[2, 0, 1, 1, 0, 0, 0, 0, 0, 1, 7], [5, 5]]': 0.08100000000000002, '[[1, 0, 1, 1, 0, 0, 0,
0, 0, 1, 7], [4, 6, 6]]': 0.0729, '[[1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 7], [0, 5, 5, 7]]': 0.06561000000000
002, '[[0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 7], [3, 4, 6, 6, 8]]': 0.059049000000000002, '[[0, 0, 1, 0, 0, 0,
0, 0, 0, 1, 4], [0, 0, 0, 5, 5, 7]]': 0.05314410000000002, '[[0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 2], [3, 4,
6, 6, 8, 8, 9]]': 0.047829690000000015, '[[0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0], [0, 0, 0, 2, 5, 5, 7, 8]]'
: 0.04304672100000002, '[[0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0], [1, 3, 4, 6, 6, 8, 8, 8, 9]]': 0.0387420489
0000002, '[[1, 0, 1, 0, 0, 0, 0, 0, 3, 1, 8], []]': -0.1, '[[1, 0, 1, 0, 0, 0, 0, 0, 3, 1, 6], [4]]': -
0.09000000000000001, '[[1, 0, 1, 0, 0, 0, 0, 0, 2, 1, 6], [5, 8]]': -0.08100000000000002, '[[1, 0, 1, 0
, 0, 0, 0, 0, 1, 1, 6], [0, 4, 5]]': -0.0729, '[[1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 6], [5, 5, 5, 8]]': -0.0
```

Figure 3: Updated game state.

Of course, the challenge with this agent is the sheer number of individual states that can be generated (as we are combining the number of permutations possible in the chosen cards list with the number of permutations possible in the current hand list.

3. RESULTS AND DISCUSSION

3.1 Adjusted Learning Rate

To test our adjusted learning rate, we ran a series of games (100 rounds with 200 training games plus 100 evaluation games in each round).

In our first test, our agents were set to have a learning rate of .1, .3, and .5 (much larger than the .01 of the original), and we also included a Deep Learning player. After 100 rounds of simulated games, we can observe that Q1 (our .1 player) appears to be strongest overall, having the most total victories in 61 rounds (note that ties are added to both players so totals may not equal 100).

Player:	Q1 (.1 lr)	Q2 (.2 lr)	Q3 (.3 lr)	Deep Learning
Rounds Won:	48	10	15	33

Figure 4: Results of Experiment One.

As Q1 was the clear winner in the first test, I elected to set it against two agents, one with the original .01 value, and another with a .02 learning rate. I was hoping that our .1 agent would be victorious but it was not to be:

Player:	Q1 (.1 lr)	Q2 (.01 lr)	Q3 (.02 lr)	Deep Learning
Rounds Won:	5	25	68	7

Figure 5: Results of Experiment Two.

The above demonstrates that our original player (Q2) and our .02 lr player (Q3), using a very low learning rate, have better performance than the .1 lr player (Q1).

For further testing (see extended_LR_testing.ipynb or the document ”Extended Testing”) we opted to incorporate a Rule-Based player, as this agent also performed fairly well in Liu’s initial testing. In these tests we set Q Learning players against each other at learning rates of 0.01, 0.02, 0.03 and a rule-based player in a group of 4, then did the same at rates of 0.04, 0.05, 0.06 and another group of 0.07, 0.08, and 0.09. In repeated testing the .01 player appears to emerge the most victorious, although occasionally the .02 player will earn the most victories. The other groups (.03 - .09) did not perform well.

We also ran head-to-head competitions of .01 vs .02, .01 vs Rule-based, and .02 vs Rule-based, .01 and .02 generally take turns defeating each other; however, the Rule-based agent defeats both .01 and .02 in their head-to-head matchups.

From these initial tests, we can conclude that a low learning rate (.01 to .02) is generally optimal for a standard Q Learning agent. However, we are not seeing the improvements we are looking for (we want to defeat Liu’s Q Learning agent, Deep Learning agent, and Rule-based agent).

3.2 Adjusted Reward Function

Our other tests involved adjusting the reward function. In our first test we based the reward on (score / 10) instead of +1 (or +2, +3, and so forth in multiplayer) for a win, -1 for a loss. The results showed that this was not optimal:

Player:	Q1 (Reward)	Q2 (Reward)	Q3 (Q, .02 lr)	Deep Learning
Rounds Won:	0	0	95	5

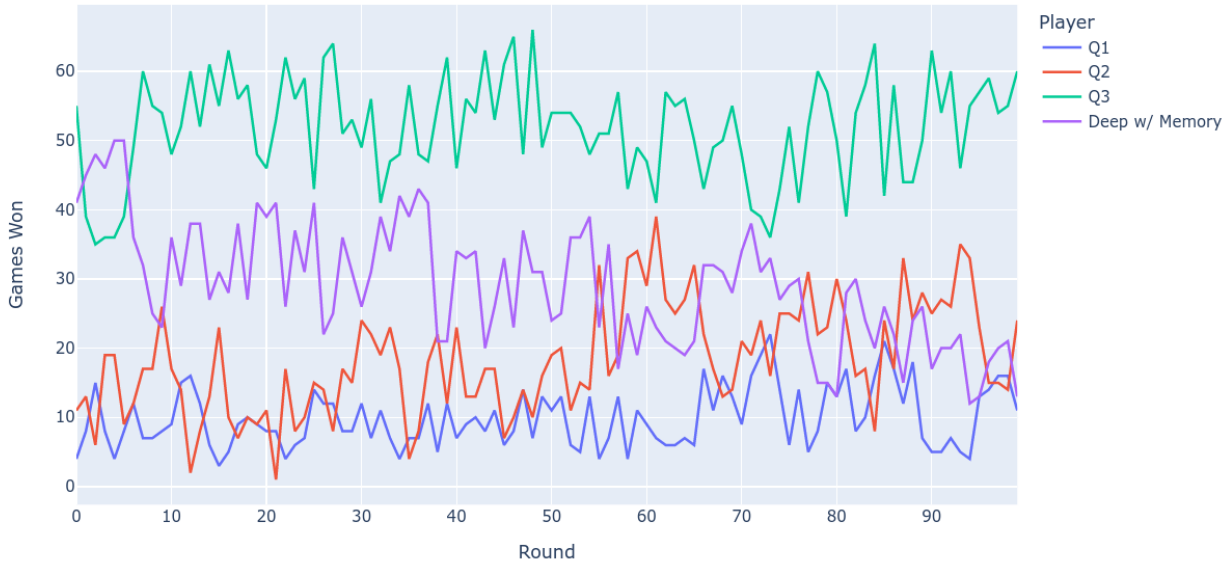

Figure 6: Results of Experiment Three.

As neither reward-based player was able to win a round, we adjusted the reward function, to (score / 10) plus or minus the original reward.

The results with the adjusted function were equally poor:

Player:	Q1 (Modified Reward)	Q2 (Modified Reward)	Q3 (Q, .02 lr)	Deep Learning
Rounds Won:	0	0	96	4

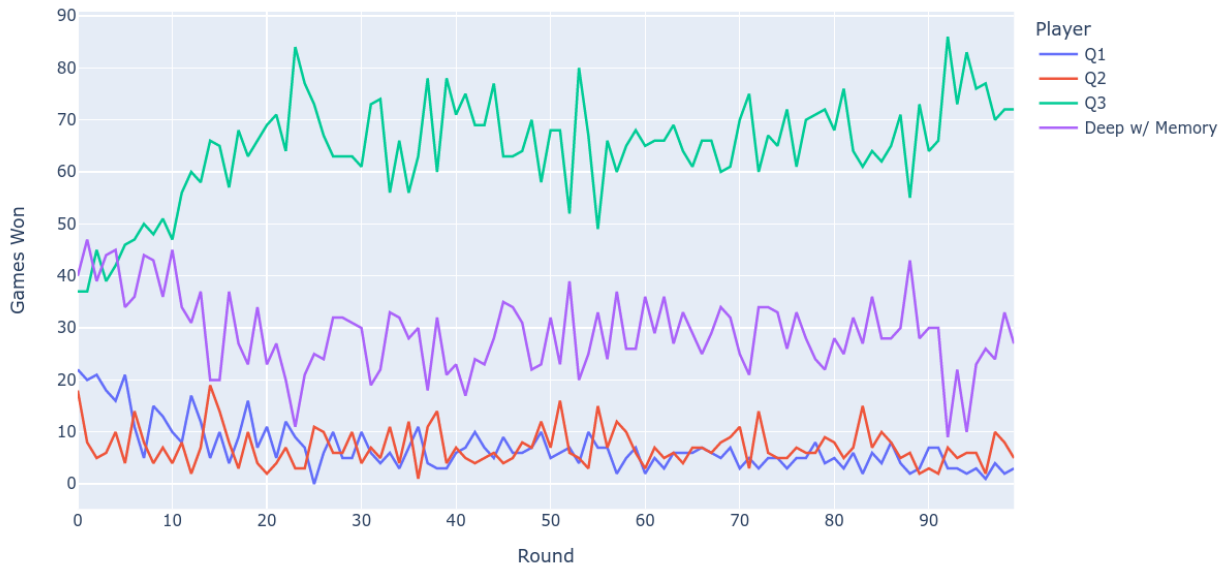

Figure 7: Results of Experiment Four.

As these initial results were so poor for the modified reward functions, it was fairly clear that the original reward function was optimal at this stage. While we ran individual games (observing the state and Q Value for each state) and determined that the modified reward functions were running as we intended (in other words, they were correctly recording the modified Q Values we designed), their poor performance indicates that further investigation into the code itself may be needed.

3.3 Q Learning with Database

In Liu's implementation, the Q Player starts outperforming other players after about 170 sets of 200 games (roughly 34,000 games). We decided to do a warm-up of our implementation of Q Player with a database by playing individual games against Liu's Q Player and Deep Learning Player. In the initial set of 500 games, Deep Player won the maximum games and our Q DB Player performed the worst. This however gave our player enough knowledge of the states. We then decided to play our DB Player against the Rule Player, Original Q Player and Deep Player.

The results showed that once the Q Player gets a knowledge base, it becomes the strongest player. After playing just 500 games, the Q DB Player started outperforming all other players from the very beginning as opposed to the original Q Player's 34,000 games. Another interesting thing to note is that our Q Player reached a threshold at about 25000 states. We ran an additional 100 sets of 100 training 100 evaluation games, when there were already 25,216 state/value pairs in the database. Only 392 more state/value pairs were added to the database. This means that we are very close to determining the finite set of all states in this game.

Player:	Rule Player	Q DB Player	Original Q Player	Deep Learning
Rounds Won:	10	78	14	0

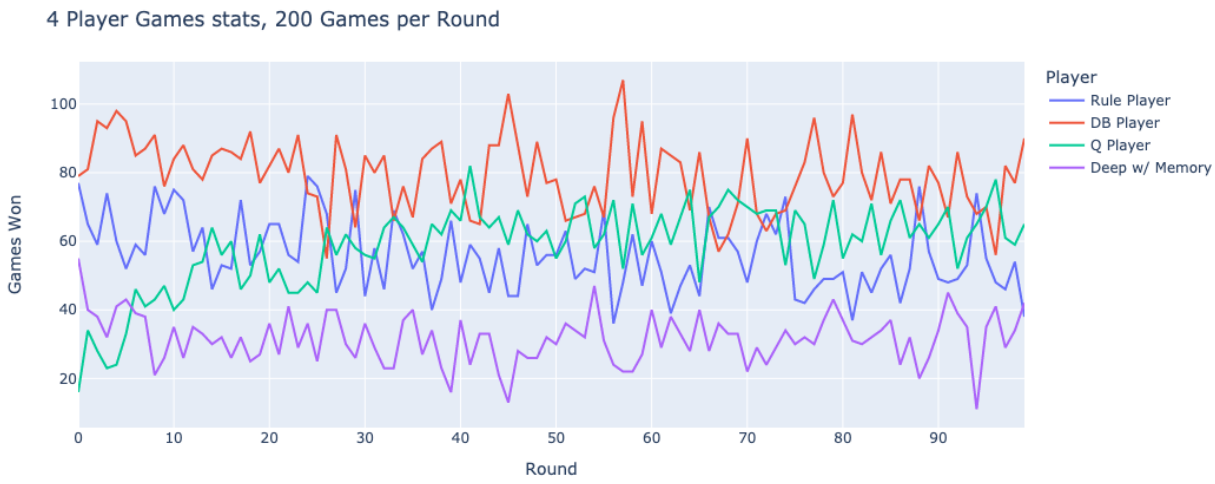

Figure 8: Results of Experiment Five.

By the end of this experiment, our knowledge base (Q-Table) had 25,216 unique state/value pairs.

```
postgres=# select count(*) from q_table;
 count
-------
 25216
(1 row)

postgres=# select * from q_table;
                state                |         q_value
-------------------------------------+-------------------------
 [1, 0, 1, 1, 0, 0, 0, 0, 2, 0, 11] | 0.1
 [0, 0, 2, 0, 0, 1, 0, 0, 3, 1, 4]  | 0.1
 [1, 1, 0, 2, 0, 0, 0, 0, 1, 1, 6]  | -0.1
 [3, 0, 2, 0, 0, 0, 0, 1, 2, 0, 3]  | 0.1
 [1, 0, 4, 1, 0, 0, 0, 0, 1, 1, 3]  | -0.1
 [1, 0, 0, 0, 0, 0, 0, 0, 5, 2, 4]  | 0.1
 [0, 0, 2, 0, 0, 0, 0, 1, 2, 1, 4]  | -0.008099999999999996
 [2, 1, 0, 0, 0, 0, 0, 1, 3, 1, 3]  | 0.1
 [2, 1, 0, 0, 0, 0, 0, 1, 3, 1, 2]  | 0.09000000000000001
 [1, 1, 0, 0, 0, 0, 0, 1, 3, 1, 2]  | 0.08100000000000002
 [2, 1, 1, 0, 0, 0, 0, 1, 1, 1, 5]  | -0.1
 [2, 1, 1, 0, 0, 0, 0, 1, 1, 0, 5]  | -0.09000000000000001
 [1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 4]  | -0.0729
 [1, 0, 3, 1, 0, 0, 0, 0, 0, 1, 3]  | -0.3889441
 [0, 0, 2, 1, 0, 0, 0, 0, 0, 1, 3]  | -0.4905300045905561
 [2, 0, 1, 0, 0, 0, 0, 0, 3, 1, 5]  | -0.1
 [0, 0, 1, 1, 0, 0, 1, 0, 2, 1, 3]  | 0.5541680399798613
 [1, 1, 3, 0, 0, 0, 0, 0, 0, 1, 10] | -0.1
 [1, 1, 3, 0, 0, 0, 0, 0, 0, 0, 10] | -0.09000000000000001
 [0, 1, 3, 0, 0, 0, 0, 0, 0, 0, 10] | -0.08100000000000002
 [0, 1, 3, 0, 0, 0, 0, 0, 0, 0, 8]  | -0.0729
 [0, 1, 1, 0, 0, 0, 0, 0, 2, 0, 10] | -0.1
 [1, 1, 1, 1, 0, 0, 0, 0, 1, 2, 7]  | -0.1
 [1, 1, 1, 0, 0, 0, 0, 0, 1, 2, 7]  | -0.09000000000000001
 [0, 0, 2, 0, 0, 0, 0, 0, 0, 2, 12] | 0.1
```

Figure 9: Database.

3.4 Adjusted State (with Database)

We created another table to the database to store the new state/value pairs, with the state adjusted to include the cards in the player's hand in addition to the cards on a player's board. We played our adjusted state player against Liu's Q Player and our Q Player with DB. The results were disappointing.

Player:	Original Q Player	Q DB Player	Adjusted State Player with DB
Rounds Won:	50	53	0

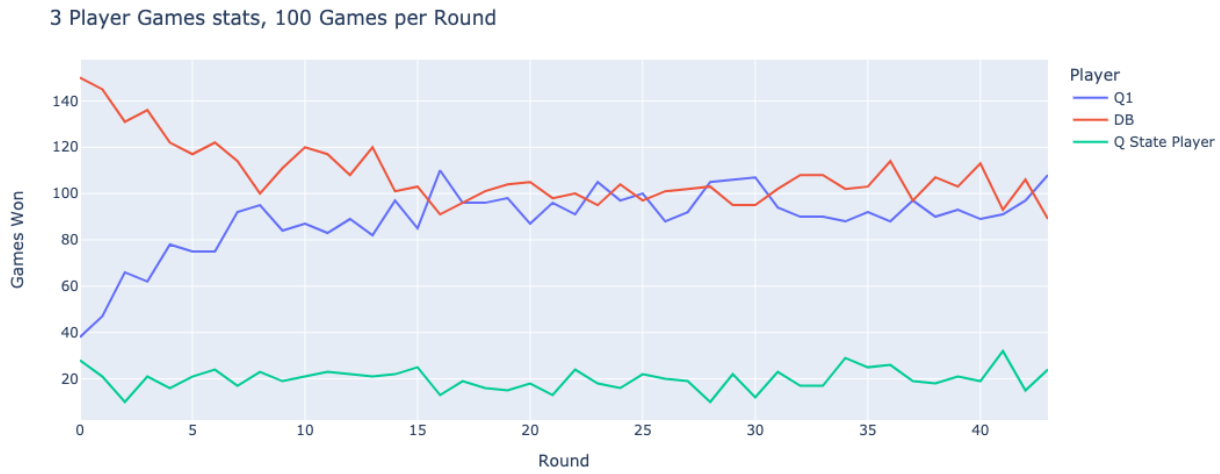

Figure 10: Results of Experiment Six.

In an additional run/test sequence, three state players ran games at the same time, contributing to the database a total of 717,795 unique state/value pairs. This was time-consuming computationally, and did not yield improved results when set against the below agents:

Player:	Rule Player	State Player	Original Q Player	Random
Rounds Won:	100	0	0	0

Figure 11: Results of Experiment Seven.

As shown in Figure 11, the updated state player (in red) is playing at the same level as the random player. This is because of the sheer number of variable states that can arise when the player's hand is added to state.

4. CONCLUSION

Our experiments using modified learning rates and reward functions did not yield successful results. However, setting the database allowed our Q Learning Agent to outperform the original Q Learning agent in just 500 games. After about a 1500 more games, it left behind the Deep Player. This goes on to show that in a game like Sushi Go, adding a repository to keep a track of states and their q-values, that are updated frequently, goes a long way towards improving the agent. This further indicates that storing a Q-Table and continuously improving it should be the first priority in building an agent for a game that has a high state variability.

Furthermore, although the adjusted state player performed poorly, it still has a lot of potential. If the agent has a memory of the cards in hand and the cards it has passed on, it can make more informed decisions. This particular feature requires more work in future, with either an improved algorithm, more computing power, or both.

REFERENCES

Deeplizard. (n.d.). *Q-learning explained - a reinforcement learning technique.* Retrieved from `https://deeplizard.com/learn/video/qhRNvCVVJaA`

Liu, M. (2020). Reinforcement learning & sushi go! *Towards Data Science*. Retrieved from `https://towardsdatascience.com/reinforcement-learning-sushi-go-238ad9bd7311`

Soen, A. (2019). Making tasty sushi using reinforcement learning and genetic algorithms. *Scientific Reports*.