Reinforcement Learning for Quantum Tic-Tac-Toe Final Report

Martin Liu
mengdal@andrew.cmu.edu

Ai He
aihe@andrew.cmu.edu

Yi Yang
yiyang4@andrew.cmu.edu

Kinori Rosnow
krosnow@andrew.cmu.edu

Abstract

Quantum Tic Tac Toe (QTTT) is a more complex version of the classic Tic Tac Toe (TTT) game where players take turns marking Xs and Os till a player has 3 marks in a row, column or diagonal. QTTT has been considered as a potential teaching tool to help students understand quantum interactions as quantum phenomena are counter intuitive. This work aimed to build a machine agent that can play QTTT using Reinforcement Learning to explore possible strategies and to be competitive with human players. We experimented with multiple model types and the sarsa update algorithm to see the learning behavior and performance of various agents. The exploration yielded deep neural network with a wide first layer that progressively decreased in width with Rectified Linear Units (ReLU) activations as the best performing model. We created a "League" system that facilitated training and evaluation games among the agents. The agents were evaluated by win, loss and tie rates which were then combined to make an effective win rate. Future work is necessary to find potential improvements to our system.

1 Introduction

Quantum tic-tac-toe(QTTT) is the upgraded version of the classical tic-tac-toe(TTT). The basic rule of tic-tac-toe is very well known: two players, X and O, take turns marking on a 3x3 grid. The first player who has three marks in any horizontal, vertical or diagonal line wins the game. The winning condition of quantum tic-tac-toe is the same - three marks in one line, but the states of marks are different. In classical tic-tac-toe, marks only have one base state, while in QTTT, marks can be in superposition (can collapse to either position) or entanglement (one pair of O and X can switch position after collapsing) in addition to the base state.

In addition to being a fun game, QTTT has been discussed as a good tool to teach students about quantum interactions. Because humans only see macroscopic, classical interactions, learning about quantum mechanics can be hard to grasp for students. The stochastic nature of quantum scale interactions is counter-intuitive for people who think of physics "classically". QTTT of the game may provide an intuition for quantum interactions.

QTTT has more possible moves, and thus is more complicated and fun than classical TTT. Detailed rules are described thoroughly by Goff (2006) [1]. In this paper, he provided clear graphs to illustrate all possible outcomes of Xs and Os, and different collapse options under entanglement. In the classical game, each cell has either empty, a player1, or a player2 piece on it, so the total number of possible states is $3^9 = 19,683$. Of course, many states are not actually achievable since the game ends before that, the actual state values are fewer. Since the total number of states are limited, q-learning can perform decently. In the quantum version of the game of only adding superposition, each piece might be a firm X, soft X, firm O, soft O, and empty (Here, X is player1, O is player2, and firm/ soft mean probability of 1 and 0.5). In that sense, there are $5^9 = 1,953,125$ number of possible states.

However, we must also consider the superposition of the piece as connected to a specific other piece, meaning that a soft X has the possibility of being "entangled" to all the other soft X. A naive way to model this version might be adding all pairs on top of each cell. Basically, all pairs can be either entangled or not entangled. Since there are $\frac{9\times8}{2} = 36$ number of possible pairs, the total number of possible entanglements are $2^{36} = 68,719,476,736$! However, only pairs with the same uncertainty value (either 50% for player1 or 50% for player2) can be entangled, so most of the combinations here are not valid by definition. Another way to frame this is the number of possible entanglements in a certain board state. The maximum possible entanglements is when a player has 3 pairs of "soft" pieces, meaning it has 15 possible combinations of entanglements. Using this number as a cap, the total number of possible states is $15 \times 1,953,125 = 29,296,874$.

We have implemented the quantum rules in code and made it feasible for machines to learn the game. Our project aims to train an agent using reinforcement learning to play QTTT. We hope the agent discovers a variety of strategies that provide further insight into the nature of the game as well as a fun competitor with humans.

2 Related Work/Literature Review

[1] Goff writes a description of quantum tic-tac-toe's superposition rule and using the game as a teaching tool.

[2] Van Seijen, Harm, et. al discussed the potential of expected sarsa compared to the normal sarsa (2009). Expected sarsa uses the expected value of the next state to update the current state value, and in their research, they found the update method to converge faster, despite the theoretical condition of convergence being the same for both algorithms. We will experiment with both update methods and compare their results.

[3] Mnih, Volodymyr, et al talked about the integration of deep learning and reinforcement learning in their paper (2013). Their agent determines the action using only the raw pixels input, and the agent has to learn to interpret the image input and game rules. For QTTT, it is not necessary to read the input as raw pixels, but we can use deep learning to predict the value of the state, especially when the number of possible states becomes too large to handle. Additionally, deep learning may allow an agent to pick up on patterns to allow for better decisions when encountering unseen states. We explored deep learning in the second half of the project.

[4] Deepmind's (Google) blog article on creating learning agents to play the complex game Starcraft II. In the article the training techniques such as adversarial league training are described. We implemented a league that has many agents compete. Agents compete with each other and find the most effective and/or nonexploitable strategies. Multiple agents in a league allow each agent to see a variety of states. In addition we support forking where an agent can be copied to have two identical agents. Although initially identical the agents can begin to deviate strategies.

[5] Zhang's article in Towards Datascience describes implementation of tic-tac-toe and a simple learning agent that learns and plays. The article also provided us with our earliest baseline for the normal game and learning agent.

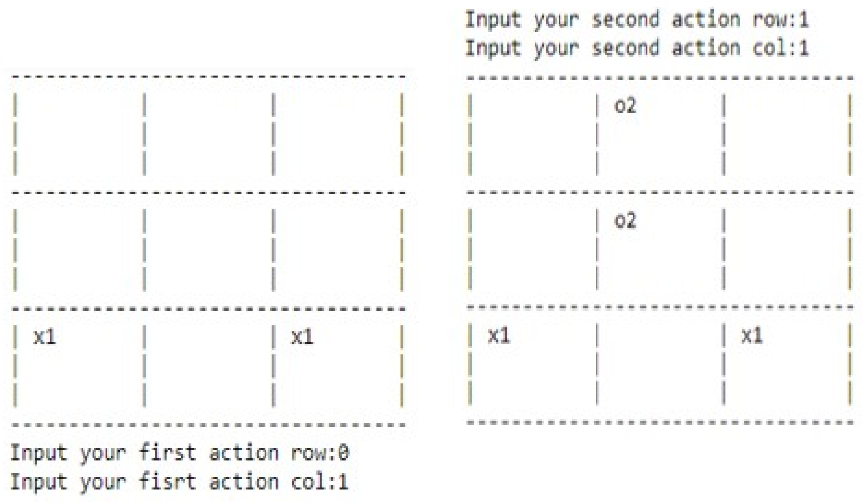
3 Work Description

Link to GitHub: `https://github.com/Gravellent/quantum_ttt.git`

3.1 Game Set Up

A simulated game will act as the dataset for this project. Each game between agents is run by a game interface that contains the board state, win conditions, and stores basic stats. At the start, each board state has the same default value, unless it is a winning state for either player. At each state, the acting player's algorithm evaluates all the available actions and their corresponding values, and then makes a decision. Based on the different decision and learning algorithms, the agent's decision making and learning can vary. These will affect the performance of the agent. The off-policy learning will be crucial for the quantum modes, since the probabilistic nature of the game requires many trials to find the optimal action which also may depend on rule variation. Finally, initial states or random explore actions may find local optima performances, but this is not yet known.

The team has built off the baseline normal TTT system adding quantum rules and experimenting with various Reinforcement Learning algorithms. The quantum version has two additional rules – superposition and entanglement. The superposition rule allows players to place two marks on the board at each time step. Figure 1 shows an example first move of a human player(o) against a bot player(x). These marks are called "spooky" marks and can be collapsed into different real states in the future.



```
---------------------------------
|         |         |         |
|         |         |         |
|         |         |         |
---------------------------------
|         |         |         |
|         |         |         |
|         |         |         |
---------------------------------
| x1      |         | x1      |
|         |         |         |
|         |         |         |
---------------------------------
Input your first action row:0
Input your fisrt action col:1
```

```
Input your second action row:1
Input your second action col:1
---------------------------------
|         | o2      |         |
|         |         |         |
|         |         |         |
---------------------------------
|         | o2      |         |
|         |         |         |
|         |         |         |
---------------------------------
| x1      |         | x1      |
|         |         |         |
|         |         |         |
---------------------------------
```

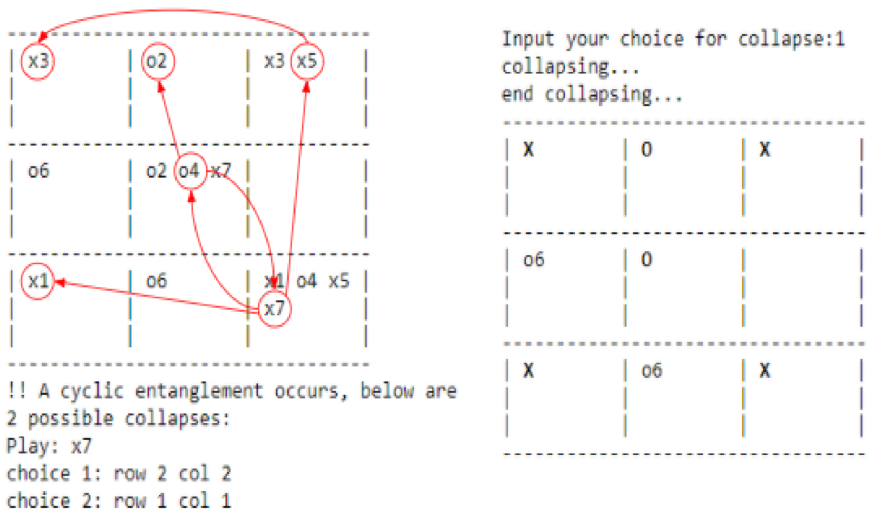
Figure 1: Human player plays quantum tic-tac-toe with bot.

Each pair of "spooky" marks played in the same time step are under the effect of entanglement, since the final placement of one mark will affect the other. What actually affects the game progression is the occurrence of a cyclic entanglement. Figure 2 shows an example of cyclic entanglement and a possible collapse. As indicated in the first board, if we choose to place x7 in the last square, x1, o4, and x5 will be forced to be placed in their counterpart's squares. This will cause a series of responsive reactions and a cycle is formed between the middle square and the last square, and this is a cyclic entanglement. There are always two possible collapses for a cyclic entanglement, and in our example, the two options are to place x7 in either the middle or the last square. The right board in figure 2 shows the final state after collapsing x7 to the last square. All the marks without a time step number are now collapsed into real marks, while the other marks that are not involved in the cyclic entanglement remain as "spooky".

After a collapse, if any player has three real marks in a line, the player wins the game, as shown in figure 3. The code implementation of the game can be found in our GitHub repository.

3.2 Baseline System

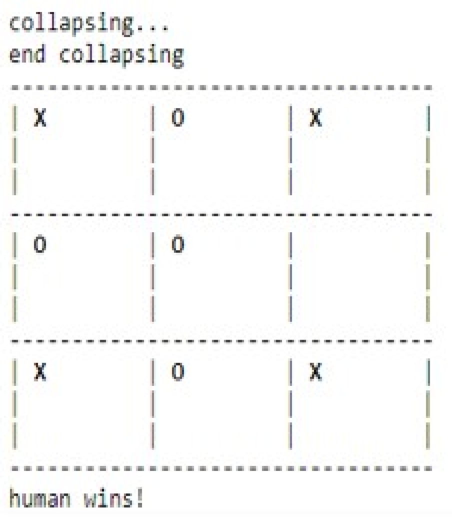
The team used the implementation of reinforcement learning on classical TTT, described in Jeremy Zhang's paper (2019) [5], as the base architecture. The original implementation uses the State–Action–Reward–State–Action (SARSA) algorithm with value iteration. The team tested with different algorithms and model structures after adding the quantum rules to the classical implementation. These will be described in detail in section 3.3, Reinforcement Learning Model.



```
------------------------------------
| x3       | o2       | x3 x5      |
|          |          |            |
|          |          |            |
------------------------------------
| o6       | o2 o4 x7 |            |
|          |          |            |
|          |          |            |
------------------------------------
| x1       | o6       | x1 o4 x5   |
|          |          | x7         |
|          |          |            |
------------------------------------
!! A cyclic entanglement occurs, below are
2 possible collapses:
Play: x7
choice 1: row 2 col 2
choice 2: row 1 col 1
```

```
Input your choice for collapse:1
collapsing...
end collapsing...
------------------------------------
| X        | O        | X          |
|          |          |            |
|          |          |            |
------------------------------------
| o6       | O        |            |
|          |          |            |
|          |          |            |
------------------------------------
| X        | o6       | X          |
|          |          |            |
|          |          |            |
------------------------------------
```

Figure 2: Human player chooses a collapse for a cyclic entanglement.



```
collapsing...
end collapsing
------------------------------------
| X        | O        | X          |
|          |          |            |
|          |          |            |
------------------------------------
| O        | O        |            |
|          |          |            |
|          |          |            |
------------------------------------
| X        | O        | X          |
|          |          |            |
|          |          |            |
------------------------------------
human wins!
```

Figure 3: Human player wins the game.

To evaluate QTTT agents, the team makes the agents play against the random agent which is used as a baseline to evaluate model performance. The team also designed a league in which agents with different model architectures play against each other to evaluate performance. The league and evaluation will be discussed in section 5, Results.

3.3 Model Description

3.3.1 Modification of Agent Learning Implementation for QTTT

For the quantum version, a few modifications are made. Since the states can no longer be represented in a 3x3 matrix with integer value in each cell, a list of entanglements are used to store the board state. For the q-learning agent, this representation would be sufficient, since it only needs to assign a unique hash for each different board state. For deep learning agents, it would require a different structure. We used half of a 9x9 matrix to represent the state.

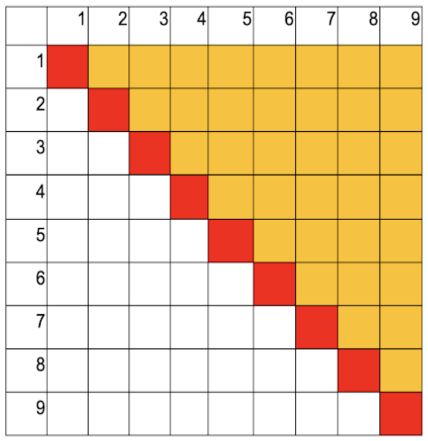


Figure 4: Board representation to allow QTTT gameplay.

As shown in the figure above, the red cells represent the "real" marks, and the yellow cells represent the "spooky" marks. For example, if there is an entanglement between the first and second cell by player 1, then the cell value for row 0, column 1 will be 1. This representation is similar to one-hot-encoding, since each cell only has 1 and -1 for its value. A potential issue with this is that the agent does not recognize the order where the pieces are placed. Since at the end of round, if after a collapse, two players both reach win condition, the order of play actually matters, this representation would cause the agent not to take that part into consideration. However, we believe recording the order of play would introduce too much variability into the state thus slowing down the learning process.

For reward, we change the reward of losing a game from 0 to -1. We found out that if the reward for losing is 0, sometimes a player would not be learning since it only gets 0s. Having a minus reward means the player would prefer to choose a random play instead of a "losing" play.

For action set-up, the quantum game introduces another action of choosing where to collapse. However, the logic is quite simple here, since it would only require a player to understand what happens after collapse under each decision. The logic is the same as choosing an action.

3.4 QTTT Agents

For the quantum version of the game, we introduce a variety of agents to compete against each other to understand each learning method's capabilities.

Random Agent:

Random agents choose action and collapse randomly. This player does not improve over time but it is good for other agents to train against, since the other agents would be able to encounter more novel states.

Q-Learning Agent:

Q-Learning Agent remembers all the states it has encountered and updates the state value pair based on the result of the game. Since the number of possible states are huge in the quantum version, a LRU cache is implemented as the memory backend. This is similar to what humans will do if it does not understand any rule of the game. For simplification purposes, we use a capacity of 100,000 so the number of stored states does not cause a significant slow down in training time for Q-learning agent.

Deep Learning Agent:

Deep learning agents employ a modelled approach to evaluate the value of board state. It takes the matrix representation of board and entanglement as input and generates a value based on the features. Number of unique input features are 36 + 9 since the matrix is symmetric along the top-left bottom-right diagonal.

For the model architecture, we explored vanilla neural networks with different activations, as well as CNN. The comparison will be explained in the result section.

3.5 Limitation of Q-Learning in QTTT

In order to further understand the limitations of q-learning, we decide to build a LRU cache with different levels of capacity and evaluate if they can capture enough states. The graph shows the relationship between training games and the percentage of hit rate (number of states found / number of queries in state values). Even with no capacity limit, the increase of the hit rate is still very slow, and the rate of increase is also decreasing. With capacity of 1,000 the hit rate starts decreasing after 10k games, while if the capacity is over 10k, it is still increasing. The effect is also related to the exploration preference of the agent. The exploration rate we use is 0.3, which is usually a good balance between exploitation and exploration. However, with such a rate, the speed of finding new states can be rather slow. Since the opponent is a random player, the agent will face a lot of new situations.

It is also worth noting that even if the hit rate does reach close to 100%, it will take time to learn the correct value, since early game states will have a higher number of possible next states to evaluate. Combined with the fact that the memory will be exceedingly large as the number of epochs increase, while q-learning provides a good theoretical optimum, the time of convergence increases exponentially as the complexity becomes higher, making it unsuitable for the quantum version of the game.

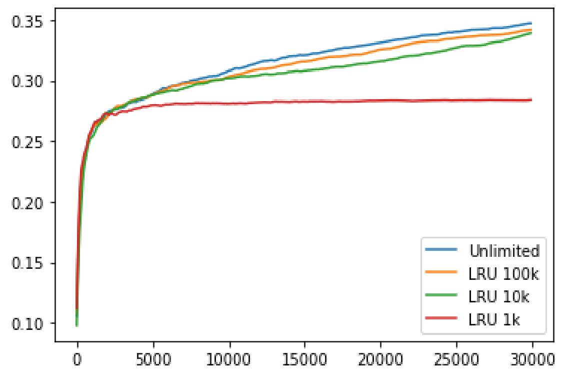


Figure 5: Varying LRU cache sizes for Q-learning. By limiting the number of states we only allow the agent to memorize a limited number of states at any time. We can see the effects of the effective win rate during learning.

3.6 League Participants

Group Stage

GroupA	GroupB
4 Layer NN	4 Layer NN
Random Agent	2 Layer NN
Q-Learning (100k capacity)	3 Layer NN
1 Layer	5 Layer NN

Neural Network Model Architectures

4 Layer NN	1 Layer	2 Layer NN	3 Layer NN	5 Layer NN
Linear(45, 256)	Linear(45, 1)	Linear(45, 32)	Linear(45, 128)	Linear(45, 128)
ReLU		ReLU	ReLU	ReLU
Linear(256, 128)		Linear(32, 1)	Linear(128, 16)	Linear(128, 64)
ReLU			ReLU	ReLU
Linear(128, 16)			Linear(16, 1)	Linear(64, 32)
ReLU				ReLU
Linear(16, 1)				Linear(32, 16)
				ReLU
				Linear(16, 1)

4 Results

4.1 Training and Evaluation Metrics: The League

We evaluated our various agents based on win, loss and tie percentage against each other and random decision agents. Since we will have a number of possible agents, we can have them play against each other and track their win rate. The main goal of the project is not only to train the best agent for the game, but to understand the relationship between different learning algorithms and their performance.

To facilitate training and evaluation of various learning methods we utilized a system inspired by Deepmind's AlphaStar where we use a "league" of agents to compete[4]. Similar to sports the league will have agents play a series of games against each other. The current implementation of the league allows for multiple rounds of play where every agent plays every other agent and playing first and second. Each round starts with every match up played for some specified number of games where the agents are learning. The round then finishes with every match up played for a specified number of evaluation games where the agents are no longer in a learning mode. The league has measures in place to address cold start issues such as warming up the agents against a random "dummy" player. The league also supports forking players allowing for multiple copies of the same player with different names. This would allow agents to then deviate their strategies creating more variability. Finally the league allows us to also eliminate poor performing models. Forking and elimination have not yet been fully automated so we would have to specify performance requirements for them.

The higher number of players makes evaluation more difficult with three metrics per agent. To simplify evaluation and visualizations we employed an "effective win rate" which is the percentage of wins combined with the percentage of ties weighted by $\frac{1}{2}$.

4.2 Neural Network Architectural Variations

Many architectures were tested in initial exploratory experiments. Various layering schemes, activation functions and losses proved the best model to be the 4 layer NN in our league's Group A. The following is some of the results we found from our experimentation.

4.2.1 Activation Functions

The three activations tested were ReLU, sigmoid and leaky-ReLU functions. First the same four layer model with ReLU and sigmoid loss function. We aimed to test a smooth function (sigmoid) against the linear ReLU that also zeros out negative outputs. Our hypothesis was that the sigmoid may perform better because it provides a smooth curve that does not just zero out negative outputs. In the following figure we can see that after several rounds, the model with ReLU just dominates the model with sigmoid.

We realized the linearity of the ReLU may be providing more information that the sigmoid as the sigmoid squashes outputs between 0 and 1 where anything not near the middle of the function would be close to 0 or 1. This may make certain outputs from the linear layer look similar when they should not. This inspired a test of a model with ReLU vs a model with leaky-ReLU as the leaky-ReLU has a linear portion for the negative values as well, which could provide more flexibility to the model.

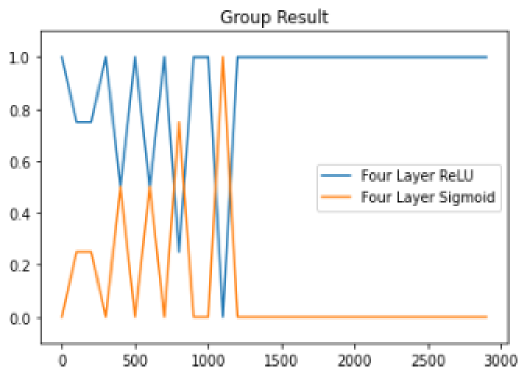


Figure 6: 4 Layer NN with ReLU vs 4 Layer NN with sigmoid. This revealed the ReLU model's superiority inspiring further experimentation.

As shown in the figure below, the ReLU still won over the leaky-ReLU, but not with as large of a margin. The evaluation effective win rate over the rounds for the ReLU model was $\sim 57\%$ while the leaky-ReLU only had an effective win rate of $\sim 43\%$. These results lead us to choose the ReLU activation function in our future experiments. however, it must be noted that there were still some fluctuations of performance as seen in Figure 8. Further experimentation must be done to conclusively state the ReLU is the best performing activation assuming sufficient training. This result only shows that ReLU performs better in the first 3000 games of learning.

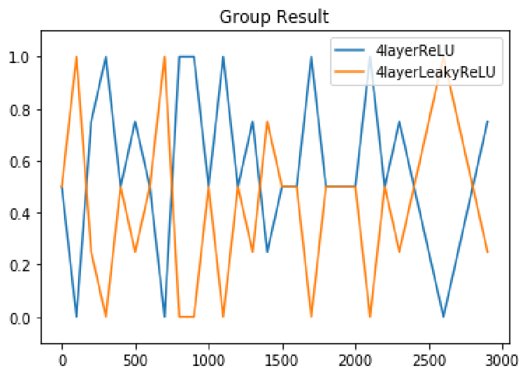


Figure 7: 4 Layer NN with ReLU vs 4 Layer NN with leaky ReLU. This experiment showed a slight advantage to the ReLU.

We also tested a Convolutional Neural Network architecture with 3 convolutional layers, each followed by a batchnorm, ReLU, and MaxPool layers. A final linear layer is appended to output the final state score. During testing, this model had very unstable performance against the random player and was outperformed by other deep learning models. A possible reason might be that the state representation, which is a 9x9 matrix, is too small for a convolutional architecture. It is also possible that the convolutional layers were not capturing interactions of nonadjacent states such as the upper left corner and the lower right corner. As a result, the team decided to exclude this model from the final league.

The number of linear layers was tested using the league system. The following section discusses the results of the league play among various types of agents and not just neural network based agents.

4.3 League Results

Our League contained 8 agents split into two groups of 4 to see which agent performs the best. We have the common four layer deep Q-learning model in both groups because we want to examine how different training conditions affect the same model. We used effective win rate as we described above to determine the winner. This was an elimination tournament style competition where each round had 3000 learning games and after every 100 learning games, 100 evaluation games were played where agents aren't learning or exploring states. The group stage results follow:

Group A Results

Rank	Name	Effective Win Rate
1	4 Layer NN	0.689
2	1 Layer NN	0.546
3	Q Learner	0.418
4	Random	0.347

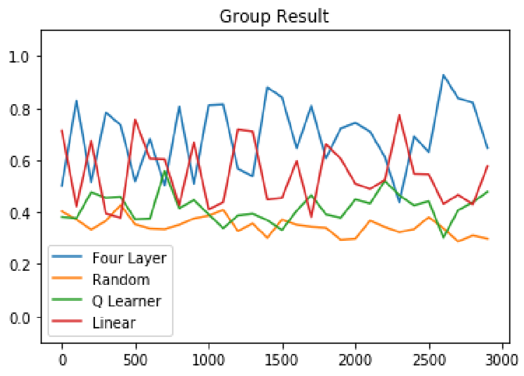


Figure 8: Group A results showing effective win rates for the 4 agents competing. The 4 layer deep agent was a clear winner.

Group B Results

Rank	Name	Effective Win Rate
1	4 Layer NN	0.611
2	3 Layer NN	0.558
3	2 Layer NN	0.428
4	5 Layer NN	0.403

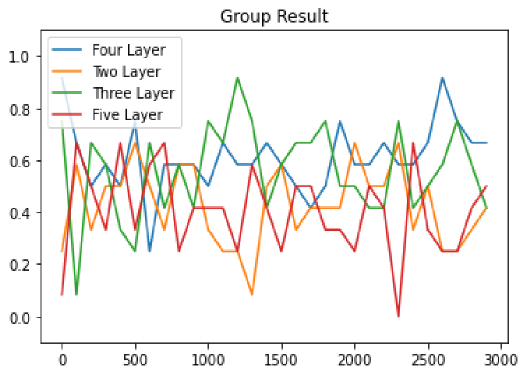


Figure 9: Group B results showing effective win rates for the 4 agents competing. Again, the 4 layer deep agent was a clear winner.

The top 2 agents from each group were sent to a semifinal round where the 1 seed from one group played the 2 seed from the other. The semifinal results follow and show a clear dominance of the the 4 layer NNs.

Semifinal: Group A 1st place vs Group B 2nd place

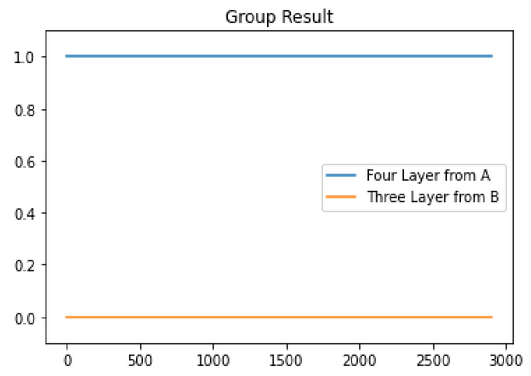


Figure 10: Semifinal 1 reveals the 4 layer NN winning.

Semifinal: Group B 1st place vs Group A 2nd place

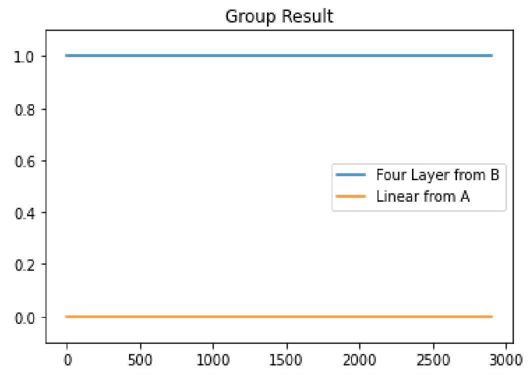


Figure 11: Semifinal 2 reveals the 4 layer NN winning.

With both the 4 layer NNs as the clear winners of their semifinal match-ups, we had them go to a championship. The results of the championship revealed the 4 layer NN from Group A winning, but not every time. The Group B 4 layer linear still has an effective win rate of about 0.25 means at the very least it was forcing some ties or wins.

Finals

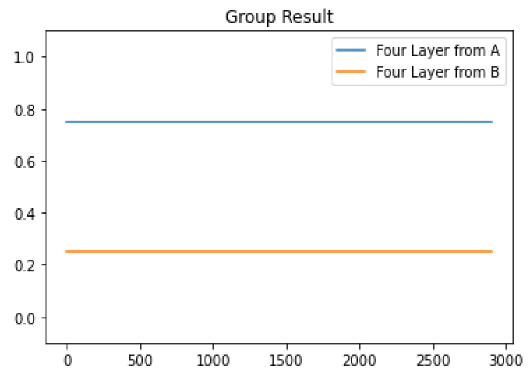


Figure 12: The championship showing the Group A 4 layer NN winning.

The Group A 4 layer NN had an effective win rate of 0.75 while the Group B agent had an effective win rate of 0.25. This was because the Group B agent only tied Group A 50% of the time. The Group A 4 layer NN winning by a significant amount may be evidence that the variety of agents in the group stages may have facilitated better learning. This would require further experimentation to confirm.

5 Conclusion

This paper implements reinforcement learning and analyzes results for the quantum tic-tac-toe game. While q-learning is sufficient for the traditional TTT game, it is no longer effective for learning QTTT. At 30,000 games level, the paper finds a vanilla four-layer neural network with ReLU activation to be the best value network. The result shows that the deep learning value network generalizes its state value model better than q-learning agents, and therefore learns at a much faster rate than its counterpart. The paper also finds that training against random agents tends to work better than training against other deep learning agents.

6 Future Work

This project was only an initial exploration of designing learning agents for QTTT. There is potential for further experimentation and improvement in many facets of the problem. Some of the possible future work is discussed below.

6.1 State of the Art Explore Rate

During most of the project, we use 0.3 as the exploration rate. The trade-off between exploration and exploitation can have a major impact on learning speed, since the higher exploration rate would lead to more states being explored. However, since the game has a huge number of states to begin with, a reasonable strategy is to focus on the winning strategy instead of exploring all the states. The complexity of the game can also have an impact on the optimal exploration rate. Further research can compare the optimal exploration rate for traditional TTT and QTTT.

6.2 More Games

https://www.overleaf.com/project/5f836123060d380001e844ad Due to time and hardware constraints, most of the models are trained for around 30,000 games. However, since reinforcement learning is very sample intensive, more games can easily lead to different results. Further research can replicate the models and train for longer time, so we can better understand what kind of model structure is truly optimal.

6.3 Effective Training Partners

One thing we notice in the result is that training against more random players tends to lead to better training overall. Random opponents mean that the players are more likely to discover new states. However, training against good players might help an agent develop understanding on the "winning strategies". Further research can introduce control groups and isolate the effect of different training partners.

6.4 Evaluation Against Other Agents

Our current evaluation is based on performance of models against the random agent as well as against each other in the league. To further evaluate our best-performing agent, we can make it compete with other existing trained agents (Android QTTT game app, the other QTTT team's agent).

6.5 Training and League Variations

League structure variations and diversity of agents has room for exploration. Our league provided possible evidence that a more diverse league can lead to better model training. This is probably because the agent is exposed to more states. Other league structures may provide better diversity.

Also ensemble agents or agent pairs like having one agent for playing first and one agent for playing second can be added functionality to the league.

6.6 Agent Evolutionary Improvement

The forking feature in the league implementation allows for agents to be copied and have the copies also compete. This would allow evolution like improvements where periodically best performers or "fittest" models continue to compete while bad performers are eliminated. Good performers can then be forked and "mutated" either through further training or manual perturbation to have copies begin to deviate to explore other strategies.

7 Division of Work

Kinori Rosnow:

Implemented the storage and retrieval of the evaluation metrics and subsequent visualizations. Implemented the league system and base script used to run the league, retrieve results and visualize. Ran tests on agent model architecture. Assisted in other areas such as debugging, agent tuning and running the league to get results.

Martin Liu:

Implementation of q-learning and deep-learning logic and integration with the state environment. Development of a variety of players and states.

Anna He:

Implementation of quantum tic-tac-toe game setup and integration with the training process. Development of the CNN model structure.

Yi Yang:

Implementation of superposition rule for quantum tic-tac-toe. Experiments on the effect of activation functions for the model performance. Generation of the graphs and results for the League.

8 References

[1] Quantum tic-tac-toe: A teaching metaphor for superposition in quantum mechanics: `https://perruchenautomne.eu/wordpress/wp-content/uploads/2015/05/QT3-AJP-10-20-06.pdf`

[2] Van Seijen, Harm, et al. "A theoretical and empirical analysis of Expected Sarsa." 2009 ieee symposium on adaptive dynamic programming and reinforcement learning. IEEE, 2009.

[3] Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." arXiv preprint arXiv:1312.5602 (2013).

[4] AlphaStar: Mastering the Real-Time Strategy Game StarCraft II: `https://deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii`

[5] Reinforcement Learning — Implement TicTacToe: `https://towardsdatascience.com/reinforcement-learning-implement-tictactoe-189582bea542`