11-785 Preliminary Report
Reinforcement Learning in Quantum Tic Tac Toe

Mingze Tang
Electrical and Computer Engineering
Carnegie Mellon University
Mountain View, CA 94035
mingzet@andrew.cmu.edu

Yanjun Zhou
Information Networking Institute
Carnegie Mellon University
Mountain View, CA 94035
yanjunz@andrew.cmu.edu

Yehan Zhang
Electrical and Computer Engineering
Carnegie Mellon University
Mountain View, CA 94035
yehanz@andrew.cmu.edu

Yueqing Li
Heinz College
Carnegie Mellon University
Pittsburgh, PA 15213
yueqingl@andrew.cmu.edu

Abstract

Reinforcement learning has been widely used for artificial intelligence in playing games and proven having the ability to defeat human players in many cases. In this essay we built an AlphaZero for the Quatumn Tic Tac Toe (QTTT) which is a game with more than 10^8 possible piece arrangements. A naive reinforement learning algorithm, temperoal difference learning (TD(0)) is also implemented. The 2 agents are tested aganist AIs with multiple difficulty levels in the QTTT game application available on the google app store. Preliminary result shows that TD(0) barely dominates the AI with easy level and the AlphaZero can reach the medium hardness level.

1 Introduction

Since the 1950s, human researchers have been working on training computer programs to play games, especially board games, against humans or other computers through reinforcement learning. Reinforcement learning is one of the fundamental subfields of artificial intelligence, a set of goal-oriented methods to learn which action to take that maximizes its long-term reward in a given environment. As a combination of deep learning and reinforcement learning, deep reinforcement learning focuses more on leveraging the powerful representational capabilities of deep neural networks to solve the continuous action space problem and explore more on the uncharted territory.

In recent years, the research into implementing board games based on deep reinforcement learning has been flourishing, including the familiar AlphaGo, which beat the human champion. Based on existing research and techniques, we would like to explore games with more complex rules, such as Quantum Tic Tac Toe, which is a "quantum version" of the traditional Tic Tac Toe game. In this project, we will focus on applying Deep Reinforcement Learning methods and the Alphazero model to simulate the operation of the Quantum Tic Tac Toe and compete with the program we designed.

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

2 Literature reviews

2.1 Quantum Tic Tac Toe

Tic Tac Toe is a board game that we are familiar with, but what is Quantum Tic Tac Toe? Quantum Tic Tac Toe is a game developed in part to provide both a metaphor and an educational activity for students to have better intuition about the nature of the quantum systems[1]. The rules of Quantum Tic Tac Toe are basically the same as in traditional Tic Tac Toe chess, but there are several additional rules: first, each grid in Tic Tac Toe can be placed with many pieces, no matter circles or crosses. If more than one piece exists on a grid at the same time, the state of that grid is called a "superposition" state, which means that the grid on which the opponent has placed a piece can still be placed by the players. Second, if the "superposition" of multiple grids form a loop, these grids will instantly collapse into a unique state, and which state to collapse into in the game requires the player to make a choice based on the position that benefits them best. All these rules add great complexity and increase the size of possible arrangements of pieces dramatically.

2.2 Reinforcement learning on relevant fields

Temporal difference learning as a classic model-free reinforcement learning algorithm, bootstraps from the current estimate of the state value function[2]. Unlike the naive Monte Carlo method where we cannot get the estimate of a state until we reach the end of the episode, TD learning updates the current state with state values learned from the last episode, which is similar to dynamic programming. TD method is a family of algorithms and the simplest one is TD(0), where the current state value is propagated from the value of next immediate states. TD(λ) is smarter as it uses the notion of eligibility trace, a weighted coefficient that combines the heuristic of recency as well as frequency of a state when doing the state value propagation. For example, if an agent gets some reward r_t at timestep t, then this reward is propagated back to previous states s_i with reward = eligibility trace $e_i * r_t$, where e_i is positively related to how frequent s_i shows up in the history and how close are those s_i to the current timestep t. Compared with TD(0), TD(λ) allows a much faster update propagation. There are also some other algorithms including Q-learning and Sarsa algorithms.

However, they all have the problem that updates are done in a look-up-table flavor, where a table is maintained to conduct all the updates and state evaluation. This becomes extremely harder when applying to chess games where the size of the state space could be more than a billion. On one hand it might not be possible to store the gigantic table in a capacity-limited machine, also it is not uncommon to encounter a state that AI never gets a chance to explore, which leads to bad estimation no better than random move. Therefore, lookup tables are replaced with much stronger function approximators, namely, the deep neural network.

The AlphaZero algorithm involves 2 parts[3]: a convolution neural network f_θ combining state-value function and action-value function and a general-purpose Monte Carlo Tree Search(MCTS) for both a policy evaluation and a policy improvement.

f_θ takes the current state s as input and outputs a probabilistic policy $P(a|s)$ as well as the state value v. Policy improvement starts with a neural network policy, executes an MCTS based on that policy's recommendations, and then projects the (much stronger) search policy back into the function space of the neural network, where before each move hundred rounds of simulations are performed based on the network recommendations, making the final decision much stronger than the initial estimation of the network. These projection steps are achieved by training the neural network parameters to match the search probabilities and self-play game outcome respectively[4].

The loss function comprises 3 parts: the square error between the estimated state value and the final reward of the episode, the second term derives from the cross entropy loss between the policy recommended by the network and the policy that projects back from the Monte Carlo Tree Search. Finally a regularization term is added to avoid over-fitting.

$$(\overrightarrow{p}, v) = f_\theta(s)$$

$$loss = (z - v)^2 - \pi^T log\overrightarrow{p} + c||\theta||^2$$

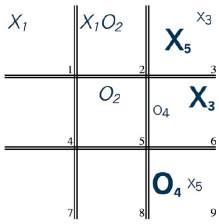

Figure 1: An example for QTTT board [1]

3 Data set

An reinforcement learning agent is learning through interacting with the environment instead of external big data input. In the case of AlphaZero, an agent generates data through self-play with the help of Monte Carlo Tree Search, where before making a move, hundred round of simulation are done with the MCTS and the prediction of the network. As a result, MCTS provides a much stronger estimation for both the policy and the state value. The network is trained on the enhanced data derived from the MCTS to improve the prediction for both, the updated network is used for data generation in the new iteration.

4 Evaluation metrics

The trained model would act as a human player to compete with the machine in an existing QTTT game [1] on Google Play. The game provides different difficulty levels, including Random Move, Easy, Medium, Hard and Optimal(difficulty increases in order). A well trained agent should be able to win Random Move and Easy modes easily and might beat machine players in Hard and Optimal modes.

5 Working baseline

We trained a TD(0) model in which two agents learn the policy by competing with each other during the training. In each episode, the program repeats the following steps till the end of the game:

- Game environment tells which agent will take its turn.
- The selected agent chooses an action based on the epsilon greedy policy
- Agent calls the game API to take the action and gets a reward G_t
- Agent updates its policy via the TD(0) learning formula

Let's look into the elements in each step.

The TD-agent uses epsilon greedy policy to make a decision between exploitation and exploration. A value smaller than the epsilon threshold leads to an exploration where valid actions are picked randomly while the opposite results in an exploitation where actions that leads to the highest state value is returned. The epsilon will decay with a fixed rate in each episode. In the exploration, actions are taken randomly, while in the exploitation, all the actions are enumerated and the next state with the maximum or minimum state value is selected.

A game tree is built for the storage of the state value and the 2 TD-agents share the same game tree. The min-max decision rule is applied to make this possible. Positive and negative rewards are assigned correspondingly to the winning of the 2 agents, which infers the 2 agents will choose the action to the maximum or minimum node respectively as their greedy action. The state is defined as a combination of 9 tuples, each of the tuples contains the marks with numbers ranging from 0 to 9 where odd numbers for the first player, even for the second and zero indicates a collapsed grid. For example, a board in the Figure1 has its state as ((1,), (1,2), (5,0), (), (2,), (3, 0), (), (), (4,0)).

The state value is updated according to the TD(0) learning formula below

$$V(S_t) = V(S_t) + \alpha[R_t + \gamma V(S_{t+1}) - V(S_t)]$$

[1] https://play.google.com/store/apps/details?id=com.gmail.smanis.konstantinos.QTTT&hl=en_US&gl=US

where S_t is the state before action, S_{t+1} is the state after the action is taken. R_t refers to the reward after taking the action, α stands for the learning rate and γ stands for the discount rate.

In the experiment, we find that the number of states blows up as we increase the number of episodes. Specifically, we will get more than 10 million states after 100,000 episodes, with most of which only being visited once and keeping its initial state value. As a result, TD(0) agent is only slightly better than the random move during the experiment.

6 AlphaZero

The convolution neural network and the Monte Carlo Tree Search are the 2 main components of AlphaZero. We will introduce them respectively.

6.1 Neural Network

6.1.1 Network Architecture

The architecture of the network is shown in Figure 2 to 5. Different from Go where the only type of action is dropping a piece, QTTT requires the player to collapse the chess board when a cyclic entanglement takes place before players' pieces are deployed. To tackle this problem, we designed a special network architecture where cases are handled in a uniform manner no matter a collapse is required or not.

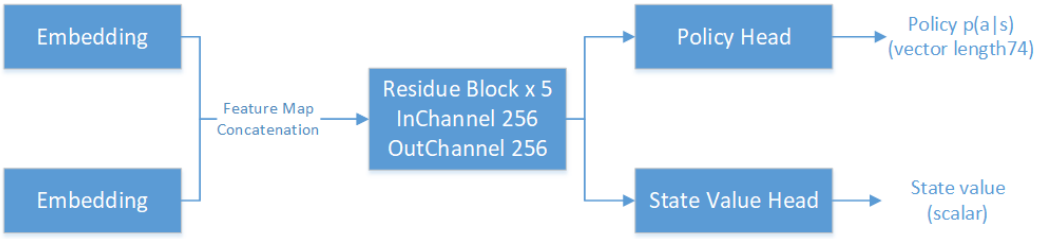

Figure 2: Main architecture

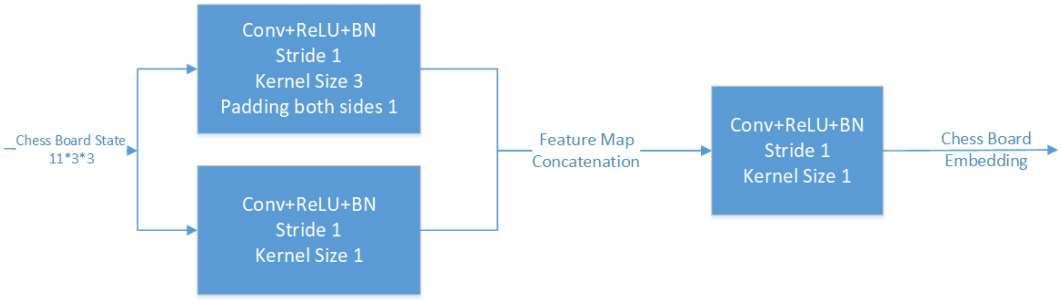

Figure 3: Embedding layer

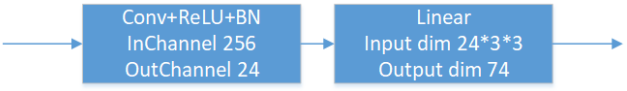

Figure 4: Policy head

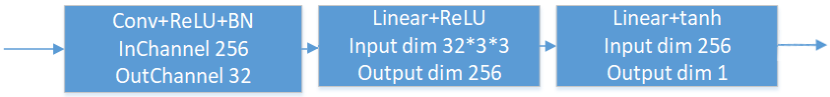

Figure 5: State value head

Collapsing an entangled chess board state is effectively making a choice between 2 possible collapsed states. The difference between a state that requires a collapse and a state otherwise would be that the former one has 2 different choices while the later offers 2 identical choices, where the identical choices is the original chess board itself, since nothing would change if we try to collapse a chess board where none of the blocks can collapse.

With this idea in mind, the network is designed to take a pair of chess board states as inputs, which are the 2 collapsed choices of the chess board. First a convolution layer functions as the embedding layer for the chess board states, then the embedding of the 2 chess boards go through layers of residue blocks, which is followed by a policy head that outputs policy $p(a|s)$, and the state value head which outputs the state value given the input.

In practice there are 74 possible actions given a pair of chess board states, which are composed of 3 parts:

1. 0-35: Take the first chess board state of the chess board pair input as the next state to play with, where each number encodes a pair of chess blocks at which pieces will be deployed. There are 9 blocks available at maximum, therefore, at most C_9^2 possible actions in total.
2. 36-71: Take the second one of the chess board pair input as the collapsed chess board to play with. Each number indicates the same meaning as the case above.
3. 72-73: There are cases where once an entangled state collapses, the game concludes and no piece is allow to dropped. As a result, 2 more options are provided here for the purpose of choosing the collapsed chess board without dropping pieces. 72 means choosing the first input state, 73 means choosing the second input state.

The policy head of the network effectively outputs a vector of probabilities, where every possible actions are encoded by the index of the elements, which are the numbers mentioned above. The elements themselves represent the probability of taking those actions. In order to avoid invalid moves, all elements whose indexes correspond to invalid actions are masked to 0, while the rest are normalized to 1.

6.1.2 Data Augmentation through exploiting symmetry

It is noted that given an terminal state of the game, which can be a tie or one of the players wins, rotating or transposing the chess board will not change the result, indicating the fact that given a piece arrangement, there are actually 8 equivalent piece arrangements in total, all of which should share the same state value and similar action probabilities $p(s|a)$.

Based on the observation above, each time we get a collapsed QTTT pairs for the current chess board, say $state_1$, $state_2$, and its corresponding action probability vector $p(s|a)$, we exploit the symmetry of the game where $state_1$, $state_2$ are applied with the same chess board transformation and action probability vector $p(s|a)$ is also applied the counterpart transformation. Transformations for the chess board includes: rot0, rot90, rot180, rot270 and ones followed by a transpose operation: rot0Transpose, rot90Transpose, rot180Transpose, rot270Transpose. For each chess board transformation, there is a corresponding transformation for the probability vector $p(s|a)$, which simply rearranges the elements of $p(s|a)$ in some specific way. Thus given chess board state pair ($state_1$, $state_2$) and $p(s|a)$, we apply 8 transformations to bulk the size of the data set. It is a very practical data augment technique. Compared with data derived from these transformations, it is extremely computationally expensive to get a data point from self-play considering hundred rounds of simulations inside a MCTS is required per data point.

When making predictions with the neural net, instead of using the real input $state_1$, $state_2$, we will randomly choose a board transformation mentioned before to get the probability vector p', and apply another probability vector transformation to p' whose corresponding board transformation is the

inverse transformation of the one we applied to the input. Thus we get the action probability vector for the original input.

6.2 Monte Carlo Tree Search

The MCTS serves the purpose of evaluating and improving the network prediction on the action policy and the state value. Before taking the final action, MCTS will roll out hundred rounds of simulations based on the recommended policy of the network and project back with a much stronger policy and state value as the training data for the network.

MCTS can be treated as a strategy improvement operator. For each state s, the original neural network will output an action probability distribution $p(s|a)$. Then the MCTS searches for a probability distribution $\pi(s|a)$ based on that network. The new probability distribution $\pi(s|a)$ provide a better action probability. Also, MCTS can be treated as a state value evaluation operator. At the end of a simulated game, MCTS gives out a stronger state value evaluation $z \in \{-1, +1\}$, which will be used as the target state value to train the network. The outputs of the MCTS are always providing better strategies compared with the outcome of the neural network due to tons of simulations. Later the network parameters θ will be updated to approximate the output value of MCTS. Let's take a closer look at how MCTS enhance the policy and state value prediction.

6.2.1 MCTS Policy Iteration

Each node s in a search tree contains edges (s, a) for all legal actions $a \in A(s)$. Each edge stores a set of statistics, $\{N(s, a), Q(s, a), P(s, a)\}$, where $N(s, a)$ is the number of times we take action a at state s, $Q(s, a)$ is the averaged action value given state s and action a, $P(s, a)$ is the prior probability of taking action a at state s, which is the output of the policy head of the network.

Select: Each simulation starts at the root node s, which is the current state of the chess board. The tree keep selecting the action a with the highest score which is characterized by score $Q(s, a) + U(s, a)$, traverse to a new node, until we reach a leaf node, where

$$U(s, a) = c_{prod} p(s, a) \frac{\sqrt{\sum_b N(s, b)}}{N(s, a)}$$

$Q(s, a)$ represents the exploitation score, whose value is initialized by the network prediction and further updated during the simulations process. On the other side, $U(s, a)$ represents the exploration score, where the fewer you tried an action a, the higher $U(s, a)$ will be among all other valid actions given state s. The coefficient $p(s, a)$ acts as the network recommendation which sheds light on the prior evaluation of each action value. Finally c_{prod} is a constant which quantifies the comparative importance between the exploration and exploitation.

Expand and Evaluate: A leaf node contains a chess board state that has never been visited before. In this case, a prior probability and predicted state value are generated using a neural network, which is used to initialize the statistics of the node before adding it to the search tree.

$$(p(s), v(s)) = f_\theta(s)$$
$$N(s, a) = 0, P(s, a) = p(s), Q(s, a) = 0$$

Backup: Starting from the leaf node, the state value $v(s)$ is propagated back to parent nodes, and at each node, parameters are updated as follows:

$$Q(s, a) = \frac{(Q(s, a) + v(s)) * N(s, a)}{N(s, a) + 1},\ N(s, a) = N(s, a) + 1$$

Play: After we finish a given rounds of simulation defined by the process of select, expand, evaluate and backup, MCTS will output the enhanced policy and state value for the root state:

$$p(s|a) = \frac{N(s, a)}{\sum_b N(s, b)},\ v(s) = Q(s, a)$$

We will either choose actions based on its probability or adopt a greedy policy choosing the action with the largest probability. Now we get a tuple (root state s, $p(s|a), v(s)$) and it will be used as the training data to make the network make better predictions.

Once an action is performed, MCTS will set the state after action as the new root node, the sub tree root the state is kept while other nodes are discarded.

6.3 Pipeline of training through self-play

The general training pipeline is shown in Algorithm 1.

Algorithm 1: Pipeline of training through self-play

```
curr_network = NN();
for each iteration, a new network is trained do
    training_data = [];
    while training_data.length < program_configuration.training_dataset_size do
        # gather training data through self-play;
        training_data += run_one_epsiode(curr_network);
    end
    # train a competitor network;
    competitor = curr_network.copy();
    train_new_network(competitor, training_data);
    # 2 networks play 100 rounds of game, if the competitor win rate is over 55%, use it to
      generate training data for the next round;
    win_rate = compete(competitor, curr_network);
    if win_rate > program_configuration.win_rate_threshold then
        curr_network = competitor;
        curr_netowork.save();
    end
end
for each episode do
    training_data = [];
    2 Monte Carlo Tree Search for 2 players;
    mcts1, mcts2;
    while game not done do
        state, player = game.get_state();
        mcts = mcts1 if player 1 else mcts2;
        mcts.simulate(100, state);
        p_enhanced = mcts.get_policy(state);
        training_data.append([state, p_enhanced]);
        action = pick_action(p_enhanced);
        reward = game.act(action);
        mcts.act(action);
    end
    append_reward_to_training_data(training_data, reward);
    return training_data;
end
```

During the self-play, we will randomly choose actions based on it probability derived from the MCTS, also Dirichlet noise are added to the root node as $P(s, a) = (1 - \mu)P(s, a) + \mu Dir(0.03)$ to ensure enough exploration is made. When the newly trained network competing against the current network, no exploration is made, namely we always choose the action with the highest probability to ensure the networks makes their strongest play.

The number of MCTS simulations is set to 400 during training and old-new network competition. The loop of the self-play won't finish until we accumulate enough training samples. In our experiment this threshold is set to 28,000. For efficiency, we always keep 30% stale data from the last iteration so that we don't need to generate that many samples from scratch.

7 TD(0) Test Result and Analysis

For our TD(0) model, Table 1 shows its performance in ten rounds for each mode. The number of wins and probability of winning are shown after the round number. Since there is no API off-the-shelf that enables us to automate the process, we do this manually.

Table 1: TD(0) model performance in ten rounds for each mode

Difficulty Level(Mode)	Number of rounds	Number of wins	Probability of winning
Random Move	10	4	0.4
Easy	10	3	0.3
Medium	10	1	0.1
Hard	10	0	0
Optimal	10	0	0

As shown above, the TD(0) agent barely dominates the agent even in the easiest level. We checked the agent policy and it turns out that most state values are 0 and states listed inside the policy is about 10^7, which is still at least a magnitude smaller than all the possible piece arrangements. This can be explained by the look-up-table update strategy where agents can only tackle cases that are recorded during training with zero generalization ability. What's more, in order to fully evaluate a state, repeated visit of the same state is required, which is almost impossible confronted with state space with such a big size.

8 AlphaZero Test Result and Analysis

For our AlphaZero model, Table 2 shows its performance in ten rounds for each mode. The number of wins, ties and probability of winning are shown after the round number. Again, since there is no API off-the-shelf that enable us to automate the process, we do this manually.

Table 2: AlphaZero model performance in ten rounds for each mode

Difficulty Level (Mode)	Number of rounds	Number of wins	Number of ties	Probability of winning
Random Move	10	10	0	1
Easy	10	4	1	0.4
Medium	10	4	0	0.4
Hard	10	0	3	0
Optimal	10	0	2	0

It is noted that there is improvement compared with the naive TD(0) agent. But we still end up with an agent with very limited intelligence, despite the fact that most part of the system is followed by the suggestions from the AlphaZero essay. There are several possible factors that could lead to this problem:

1. Insufficient training: It takes about 8 days of training before we finally evaluate the model, when only fewer than 100 epochs had been finished. It is typical that one epoch takes 4 to 5 hours to finish, where most of the time is spent collecting the training data. It is noted that the entire game environment is implemented in python, which is notoriously lack of performance. As a result, we cannot afford enough rounds of training. We once considered to re-implement the entire game environment with Cython or leveraging multi-thread programming, but considering the time we left and the potential workload, we finally decide to stick to what we have. At the time we do the evaluation, the agent is still generating getting stronger every 2 to 4 epochs, showing no evidence of saturating.
2. Trapped in the local optima: A small bug is spotted at the fifth day of training, where Dirichlet noise was not added to the state of the blank chess board, which is the very initial stage of the game. It is possible that even though we fixed the bug immediately and the training continues, the model could have already been trapped inside a local optima, and the

noise added later is insufficient for it to jump out of it, finding a brand new way of starting the game. The first move matters a lot compared with all subsequent moves. During our manual test, the model failed to show that it learns the optimal way of making the first move, which, according to the AI of the optimal hardness level provided by the game application, is to place the piece along the corners of the diagonal.

3. Parameter tuning: There are many parameters that we can tune, including c_{prod}, number of simulations, win rate threshold, stale data percentage, etc.
4. Network architecture design: Considering QTTT chess board is very small compared with Go and there are multiple types of actions in QTTT(QTTT sometimes requires state collapsing, so the network has to take 2 chess board states as input simultaneously), We have to design our own network. There might be flaws of our network design where strategies learnt before can be forgotten by the network due to the lack of capacity.

During the implementation, the MCTS part is built upon the code base of [5], which helps us to gain a basic understanding of the Monte Carlo tree search algorithm. But some features mentioned in the AlphaZero essay is missing in this implementation including the Dirichlet noise, where we have to work on our own. Also many adaptations are needed to fit the code into the context of our project .

9 Conclusion and Future Work

We will try to increase the noise to see if we can get a better first move. Also when making predictions with the network, instead of randomly choosing one transformation, we can apply all transformations to get 8 different state values and action policies and average them to get the final prediction, which could give us more accurate prediction of the state and the action to take.

For further improvements, the one top on the list would be speeding up the training process, implementing the game environment API with C or multi-threading the process of self-play are strongly suggested. What's more, the symmetry of the game can be further exploited, for example, instead of making the network and the search tree to cover all possible symmetric equivalence, why not eliminate those choices inside the game environment? Valid moves that are symmetrically equivalent will not be provided as valid moves, instead the game environment itself will record which version of the symmetric we choose. Thus the possible piece arrangements is greatly reduced (by factor of 8 at most), and a better agent performance is expected.

10 Division of work

Yehan Zhang:

- Implemented main logic for QTTT game and TD(0) agent training pipeline
- Implemented the main logic for the AlphaZero
- Unit test for multiple QTTT APIs
- AlphaZero integration test
- MCTS integration test
- AlphaZero paramter tunning

YueQing Li:

- Investigated and implemented MCTS

Yanjun Zhou:

- Implemented TD(0) and tested baseline model
- Implemented the network module for the AlphaZero

Mingze Tang:

- Implemented and testing of QTTT
- Implemented the feature of data augmentation through chess board symmetry exploitation

11 Deliverables:

Git repo: `https://github.com/yehanz/Qttt_RL`.

For runnable TD(0) agent, please refer the master branch

For code of AlphaZero, please refer the AlphaZero_Qttt package located at branch yehanz

References

[1] Goff, A. (2006). Quantum Tic Tac Toe: A teaching metaphor for superposition in quantum mechanics. *American Journal of Physics*, 74(11), 962-973. doi:10.1119/1.2213635

[2] Temporal difference learning. (2020, October 14). Retrieved November 09, 2020, from https://en.wikipedia.org/wiki/Temporal_difference_learning

[3] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., ... & Lillicrap, T. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*.

[4] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., … Hassabis, D. (2017). Mastering the game of Go without human knowledge. *Nature*, 550(7676), 354–359. https://doi.org/10.1038/nature24270

[5] Suragnair. (2020, May 05). Suragnair/alpha-zero-general. Retrieved December 08, 2020, from https://github.com/suragnair/alpha-zero-general/blob/master/MCTS.py