J.P. MORGAN EXERCISE PROJECT REPORT QUESTION 1: VARIATIONS ON TIC-TAC-TOE (ALSO CALLED NOUGHTS AND CROSSES)

Qihao Wu
The University of Hong Kong
qihaowu@connect.hku.hk

ABSTRACT

This project extends the classic Tic-Tac-Toe to a more complicated and stochastic two-player zero-sum game. Given this complexity, traditional traversal algorithms such as Minimax may fail due to the "curse of dimensionality". To address this challenge, I employ Deep Q Networks to derive the optimal policy under uncertainty. The reinforcement learning environments are well-designed for Part 1 and Part 2, respectively. The implementations of codes are tested following the testing plan and the performances can be evaluated with random and human-player policies.

1 INTRODUCTION

This project leverages the power of the TensorFlow (TF) Agent (Abadi et al., 2015; Guadarrama et al., 2019) to design and train reinforcement learning (RL) agents to play two variations of the classic game Tic-Tac-Toe (also known as Noughts and Crosses). TF Agent is an efficient, flexible, and easy-to-use library for developing and testing sophisticated RL algorithms.

In Part 1, the variant involves a larger 9 by 9 playing board where the win condition is altered such that a player must achieve 4 in a row, column, or diagonal to win, rather than 3 in the classic setting. The game involves two players taking turns to place their respective noughts and crosses on an empty square. However, after choosing a square, there is only a 1/2 chance that the piece is placed at the chosen square. If the player's choice is not accepted, the move is randomly determined from the 8 squares adjacent to the chosen one with an equal probability (i.e. 1/16). If the random choice is occupied or outside of the board, the player's turn is forfeited. In Part 2, the game variant introduces an energy concept. Each player begins the game with 10 energy points. During each move, a player can use any amount of energy between 0 and 1 until all the energy points are exhausted. The energy usage influences the probability of a player's move being accepted. If no energy is utilized, the acceptance probability of a move becomes only 1/9. If energy $E \in [0, 1]$ is used, the acceptance probability increases to $1/9 + 6/9 \times E$. If a chosen move is not accepted, the move is once again determined randomly as the setting in Part 1.

These variants pose multiple challenges. The first challenge arises from the expanded problem scale. The game board expands to a 9 by 9 grid, nine times larger than the classic Tic-Tac-Toe board. Additionally, the winning condition of aligning four pieces introduces further complexity. The second challenge comes from the uncertainty of moving. Players' moves are no longer deterministic, therefore, the game can be defined as a stochastic process. The third challenge is the intricate decision-making mechanism in Part 2, in which every move becomes a combination of position choice and energy usage. Designing an efficient agent to simulate, learn, and optimize the decision-making policy is a nontrivial task.

Given this project and its challenges, this report aims to outline feasible solutions from both theoretical and practical aspects. Section 2 presents a literature review of related studies in games and RL. Section 3 discusses the exact methods employed in this project. Section 4 showcases experimental results, followed by testing plans exhibited in Section 5. By the end, Section 6 summarizes this project report.

2 Literature Review

RL has been widely applied to various game scenarios, including classic games such as Tic-Tac-Toe and other board games. These applications serve as a proving ground for developing and testing RL algorithms (Sutton & Barto, 2018). This is because these games' mechanisms can be formulated as the Markov Decision Process (MDP) (Puterman, 1990), in which states, actions, transitions, and rewards can be explicitly defined. This advantage provides a solid environment for RL agents to learn and optimize.

Simple board games (i.e. classic Tic-Tac-Toe) can be optimally solved by traversal algorithms such as Minimax (Dem'yanov & Malozemov, 1990), dynamic programming (Bellman, 1966), etc. In the RL family, tabular methods such as Q-learning (Watkins & Dayan, 1992) and SARSA (Sutton & Barto, 2018) have been applied to Tic-Tac-Toe with notable success. These methods do not require any function approximation and hence can find an exact solution to the problem. However, the MDP of complex games (e.g. Go, chess) always has huge state and action spaces, as well as long transitions. Traversal algorithms may easily face the curse of dimensionality (Powell, 2007). RL with function approximation methods tend to excel in these cases.
With the help of neural networks as value function approximations, Mnih et al. (2013) introduced Deep Q Networks (DQN) to approximate the Q function (action-value function). This approach has been successfully applied to a variety of games, including board games. DeepMind's AlphaGo is one of the most notable successes of RL in games. Silver et al. (2016) employed deep RL to optimize the game of Go, in which RL agents defeated top human players. Merged with deep neural networks, Monte Carlo Tree Search (MCTS) (Świechowski et al., 2023) largely contributes to this achievement. This algorithm builds a search tree by performing numerous simulations (rollouts), balancing the exploration and exploitation with the Upper Confidence Bound (UCB) (Jamieson & Nowak, 2014).
Policy gradient methods such as REINFORCE (Williams, 1992) and Actor-Critic (AC) (Mnih et al., 2016) have also been used in the game scenarios. Williams (1992) introduced the REINFORCE algorithm, which directly optimizes the policy. AlphaZero, introduced by Silver et al. (2017), represents the state-of-the-art in RL for board games. Unlike its predecessor, AlphaGo, AlphaZero learns entirely from self-play and does not require any prior knowledge about the game. It uses the AC framework that combines a policy network to guide the search with a value network to evaluate positions, also with MCTS used for planning. Since the game environments are well-defined, model-based RL is another promising direction for board games. Racanière et al. (2017) introduced Imagination-Augmented Agents (I2As), which learn to interpret predictions from a learned environment model to construct a plan, significantly improving data efficiency. Schrittwieser et al. (2020) from DeepMind introduces the MuZero algorithm, which combines model-based and model-free aspects to achieve state-of-the-art performance on several complex board games.

On the other hand, RL in stochastic environments and combinatorial action spaces is a vibrant and challenging area of research. The challenge lies in the agent's ability to learn optimal policies while interacting with an uncertain environment and making decisions from combinations of action spaces. While compared with traversal algorithms and robust optimization methods (Gabrel et al., 2014), RL agents have demonstrated their advantages in decision-making under uncertainty (Sutton & Barto, 2018; Watkins & Dayan, 1992), especially in large-scale problems.
The main challenge when dealing with combinatorial action spaces is the size and complexity of the action space. For instance, in a routing problem, the number of possible routes grows factorially with the number of nodes, making it impossible to enumerate all actions. Pointer Networks (Vinyals et al., 2015) use an attention mechanism (Vaswani et al., 2017) to sort the elements of the input sequence, effectively learning a permutation of the sequence, which is a combinatorial action. This neural network architecture can be deployed in policy function approximation. In terms of value-based methods, Q-learning with Permutations (Bello et al., 2016) approximates the Q-function in a way that can handle combinatorial action. Furthermore, model-based methods can be highly effective in combinatorial action spaces by incorporating

domain knowledge into the model and avoiding the need to learn from scratch (Pascanu et al., 2017).

3 Methods

Based on the introduction and literature review, I employ value-based DQN to optimize this stochastic extended Tic-Tac-Toe game. Given that the Tic-Tac-Toe game is built on a zero-sum game, concepts in game theory (Bowling & Veloso, 2002) and multi-agent RL (Vinyals et al., 2019) are referred to enhance the formulation. In this project, player 1 and player 2 respectively play crosses (1) and noughts (2) in turn, and both of them are primarily modeled as two independent DQN agents without prior knowledge about the game. The RL environments for both Part 1 and Part 2 are then formulated.

3.1 RL environments

3.1.1 Part 1

The MDP of stochastic extended Tic-Tac-Toe in decision stages:

- *State:* The observable state for each player (agent) is the 9 by 9 Tic-Tac-Toe board with empties, noughts, and crosses on these 81 positions. In programs of RL agents, I use 0 to indicate empties, 1 for crosses, and 2 for noughts.
- *Action:* The action for each player is a dictionary that contains the position choice and the corresponding value (1 or 2) to be placed to identify the player.
- *Reward:* The reward for an agent's win is 10, and loss is -10. The rewards for the draw or not finished play are all 0. Here, I define a small penalty, -0.001, towards stochastic moves, when a player's choice is not accepted. Also, a large penalty, -0.5, is given towards illegal moves, when the position choice is successfully accepted, but has already been occupied.
- *Transition:* The transition follows the stochastic game mechanism, where there is only a 1/2 chance that a player can place at the chosen square (position). Otherwise, the move goes to nearby 8 squares (positions) with equal probability.

In this way, two independent DQN agents (players) and the MDP formulate sequential decision-making. In each decision stage, an agent observes the current state (board) and chooses an action (square to move). Then, the environment steps to the next state following defined transitions, and the agent receives the corresponding reward. Two agents take turns repeating this process sequentially till the end of the game episode. The sizes of state space and action space of the agent are both 81, which is the 9 by 9 game board. Given that there are two independent DQN agents engaging in a zero-sum game, I designate player 1 as the primary agent and set rewards for the win and loss of player 2 to be the negation of those allocated to player 1.

3.1.2 Part 2

Based on the above MDP of the Part 1 RL environment, the main differences of the Part 2 RL environment are located at the energy points and different stochastic probabilities in transitions. Therefore, the following modifications are listed:

- *State-2:* Besides the settings in Part 1, the remaining energy balances are integrated into state space for each agent.
- *Action-2:* The action for each agent extends to a dictionary that contains three items: square (position) choice, the corresponding value (1 or 2), and the energy used E for this move.
- *Reward-2:* The reward remains the same as the settings in Part 1.
- *Transition-2:* The probability of accepting the chosen square is dependent on energy used: $1/9 + 6/9 \times E$.

In contrast to the MDP presented in Part 1, the state space in Part 2 incorporates an additional dimension to represent energy balance, expanding the state space to 82. With respect to the action space, the chosen squares are always discrete. Consequently, I have opted to discretize the energy used for each decision into 11 evenly distributed options ranging from 0 to 1. This ensures consistency between the actions related to the choice of position and the energy used and allows them to be jointly encoded into an expanded action space comprising 891 possibilities.

3.2 DQN AGENT

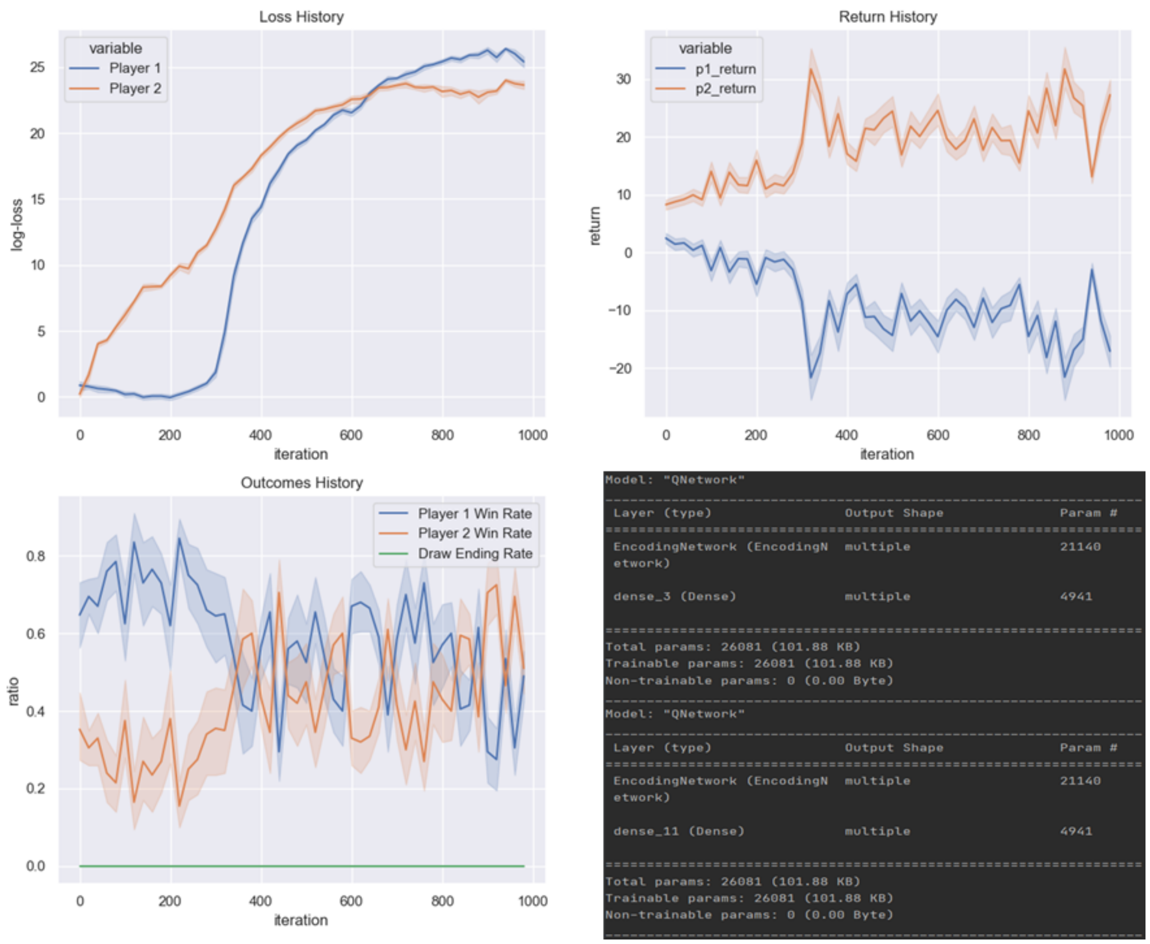
The method I choose is DQN since it has demonstrated human-level control in various game applications and has been widely applied in industry (Mnih et al., 2015). More importantly, in this project, the optimal policy of playing Tic-Tac-Toe consists of discrete actions of choosing squares. This is naturally consistent with value-based DQN.
Further, algorithmic improvements specifically designed for this project can be added to DQN in a more convenient manner. Although the penalty for illegal moves has been designed in the MDP, this may not be an efficient learning solution since the RL agent has to learn from unnecessary mistakes. In order to address this problem, I have designed action masking (Huang & Ontañón, 2020) for DQN to filter invalid actions (already occupied squares) based on the observations of the state. In this way, the learning efficiency can be optimized. The pseudocode is shown as follows:

```
Input: initialize replay memory $D$ to capacity $N$
Input: initialize action-value function $Q$ with random weights
Input: initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
Input: initialize samples $x$, discount rate $\gamma$
for episode = 1 to $M$ do
  Initialize state $s_1 = \{x\}$ and preprocessed state (observation) $\phi_1 = \phi(s_1)$
  for $t = 1$ to $T$ do
    With probability $\epsilon$ select a random action $a_t$, otherwise select $a_t = \arg\max_a Q(\phi(s_t), a; \theta)$

    Generate action masking $A(s_t)$, and valid action $a_t = a_t A(s_t)$
    Execute action $a_t$ in environment and receive reward $r_t$ and transition to $s_{t+1}$
    Preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
    Set $y_j = r_j$, for terminal $\phi_{j+1}$
    Set $y_j = r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-)$, for non-terminal $\phi_{j+1}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$, with respect to the network parameters $\theta$
    Every $C$ steps reset $\hat{Q} = Q$
  end for
end for
Output: Trained DQN agent
```

4 RESULTS

Referring to the implementations of codes, I have successfully conducted a series of experiments. Given that deep RL algorithms are very sensitive to hyperparameters' tuning, the models and neural networks can be flexibly updated according to different scenarios.

Figure 1 illustrates the experimental results of Part 1. The training process comprised 1000 iterations, with each iteration containing 10 episodes. A Multi-Layer Perceptron (MLP) served as the value-function approximator, with pertinent parameters detailed in the subplot located at the lower right corner of Figure 1. The subplot situated at the upper left corner provides a visual representation of the history of the training loss. Following 1000 iterations, convergence appears to be the trend for both agents. The subplot illustrating the return history exhibits two highly synchronous but opposite trends, which can be attributed to the opposing rewards set for the two players. In the subplot documenting the history of outcomes, no instances of draw endings are observed. This is likely due to the increased complexity of this stochastic, extended Tic-Tac-Toe game. In its classic counterpart, the game is deterministic and fully solved, implying that a player who takes the initiative to move and employs an optimal policy can, at the very least, secure a draw.



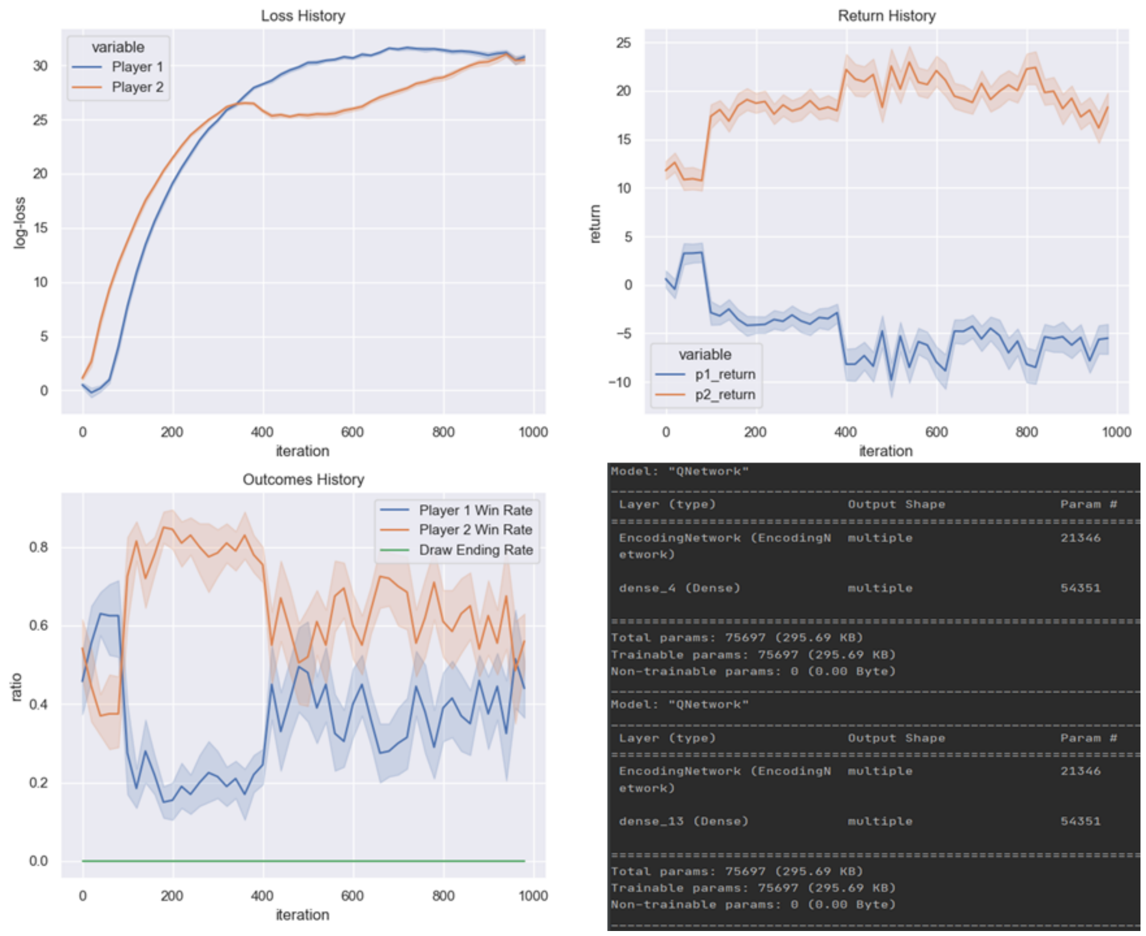
```
Model: "QNetwork"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 EncodingNetwork (EncodingN  multiple                  21140
 etwork)

 dense_3 (Dense)             multiple                  4941

=================================================================
Total params: 26081 (101.88 KB)
Trainable params: 26081 (101.88 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
Model: "QNetwork"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 EncodingNetwork (EncodingN  multiple                  21140
 etwork)

 dense_11 (Dense)            multiple                  4941

=================================================================
Total params: 26081 (101.88 KB)
Trainable params: 26081 (101.88 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Figure 1: Part 1 training results

Figure 2 depicts the experimental results of Part 2. The hyperparameters are set identically to those in Part 1, although the model is more complicated and stochastic, and subsequent optimization is added. The training loss history subplot demonstrates convergence of the two agents post 1000 iterations, thereby suggesting the efficacy of the training process. The return history subplot reveals contrasting returns for the two players as well, attributable to the opposing rewards setting of player 2 in the Part 2 experiments. Part 2 is even more complex than Part 1 and more likely leads to a non-stationary learning policy. Hence, careful examination and enhancement of the algorithms and neural networks is vital and needs to be accompanied by effective testing plans.

5 Tests and Validation

The testing plans are developed from two aspects. The first aspect is ensuring the correctness of codes, through unit tests, regression tests, and integration tests. The second aspect is to evaluate the trained model by comparing it with other policies.



```
Model: "QNetwork"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 EncodingNetwork (EncodingN  multiple                  21346
 etwork)

 dense_4 (Dense)             multiple                  54351

=================================================================
Total params: 75697 (295.69 KB)
Trainable params: 75697 (295.69 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
Model: "QNetwork"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 EncodingNetwork (EncodingN  multiple                  21346
 etwork)

 dense_13 (Dense)            multiple                  54351

=================================================================
Total params: 75697 (295.69 KB)
Trainable params: 75697 (295.69 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

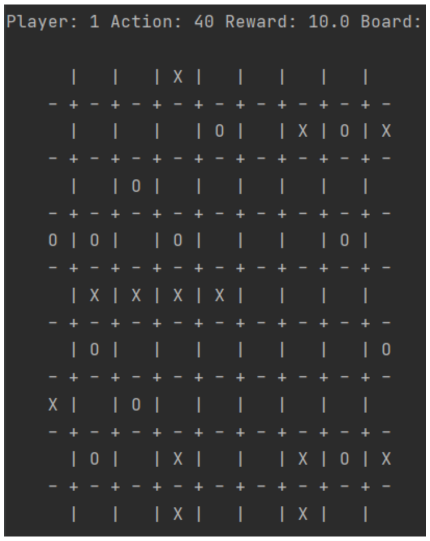
Figure 2: Part 2 training results

5.1 Testing plan

It is crucial to conduct various types of tests, including unit tests, regression tests, and integration tests to ensure the correctness and validity of codes' implementations. Also, all codes are well-commented for maintainability and understanding. Unit tests (Shaw, 2023) are used to verify the correctness of individual units of source code, such as functions or methods. Python's built-in *unittest* module or the *pytest* library (Okken, 2023) can be used for this. Regression tests (Okken, 2023) are performed to ensure that previously developed and tested software still performs correctly after the modification. Integration tests (Shaw, 2023) are designed to verify that different modules or services in an application can work together correctly. Following the pipeline of tests, each key function in this project has a corresponding unit test to cover all possible cases and inputs. Regression tests and integration tests have been implemented during the coding.

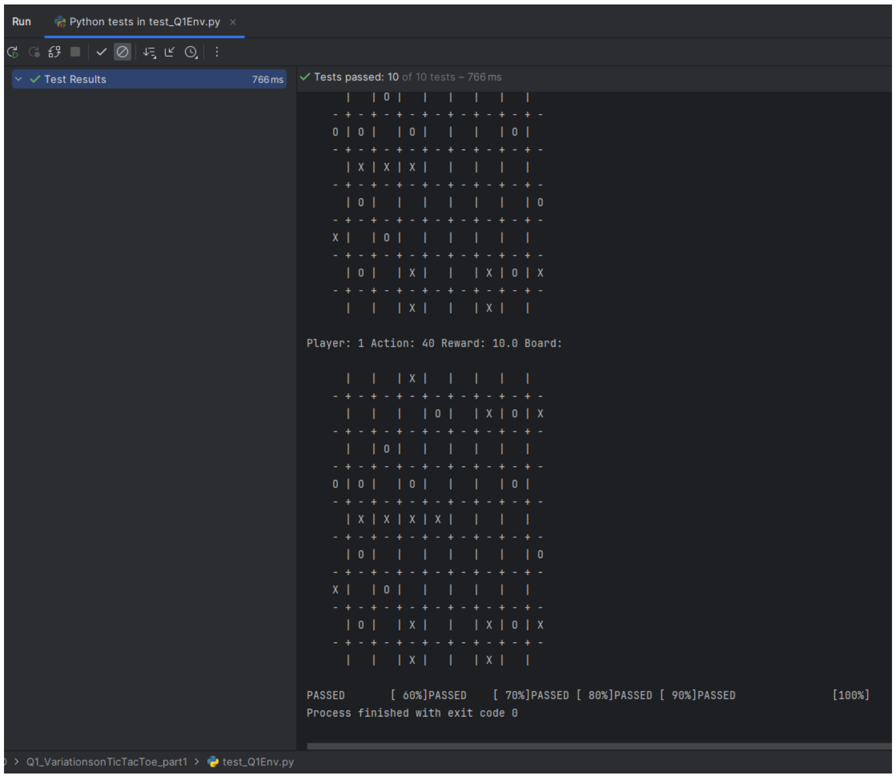
In detail, I design two groups of tests for both Part 1 and Part 2 separately. Each group has 10 tests, in which 6 unit tests can check the correctness of every function of the model, 2 regression tests check the players' move (choice on the board), and the status of the end of the episode, and then, 2 integration tests ensure the smooth play of the whole game and episode from the points of random trials (as depicted in Figure 3) and agent-based simulation.

There are a total of 20 tests (10 for Part 1 and 10 for Part 2) involved. Figure 4 and Figure 5 show that all these tests are successfully passed along with the visualizations of the boards of the Tic-Tac-Toe game. Therefore, these testing results can validate the correctness of my codes.



```
Player: 1 Action: 40 Reward: 10.0 Board:

  |   |   | X |   |   |   |   |  
- + - + - + - + - + - + - + - + -
  |   |   |   | O |   | X | O | X
- + - + - + - + - + - + - + - + -
  |   | O |   |   |   |   |   |  
- + - + - + - + - + - + - + - + -
O | O |   | O |   |   |   | O |  
- + - + - + - + - + - + - + - + -
  | X | X | X | X |   |   |   |  
- + - + - + - + - + - + - + - + -
  | O |   |   |   |   |   |   | O
- + - + - + - + - + - + - + - + -
X |   | O |   |   |   |   |   |  
- + - + - + - + - + - + - + - + -
  | O |   | X |   |   | X | O | X
- + - + - + - + - + - + - + - + -
  |   |   | X |   |   | X |   |  
```

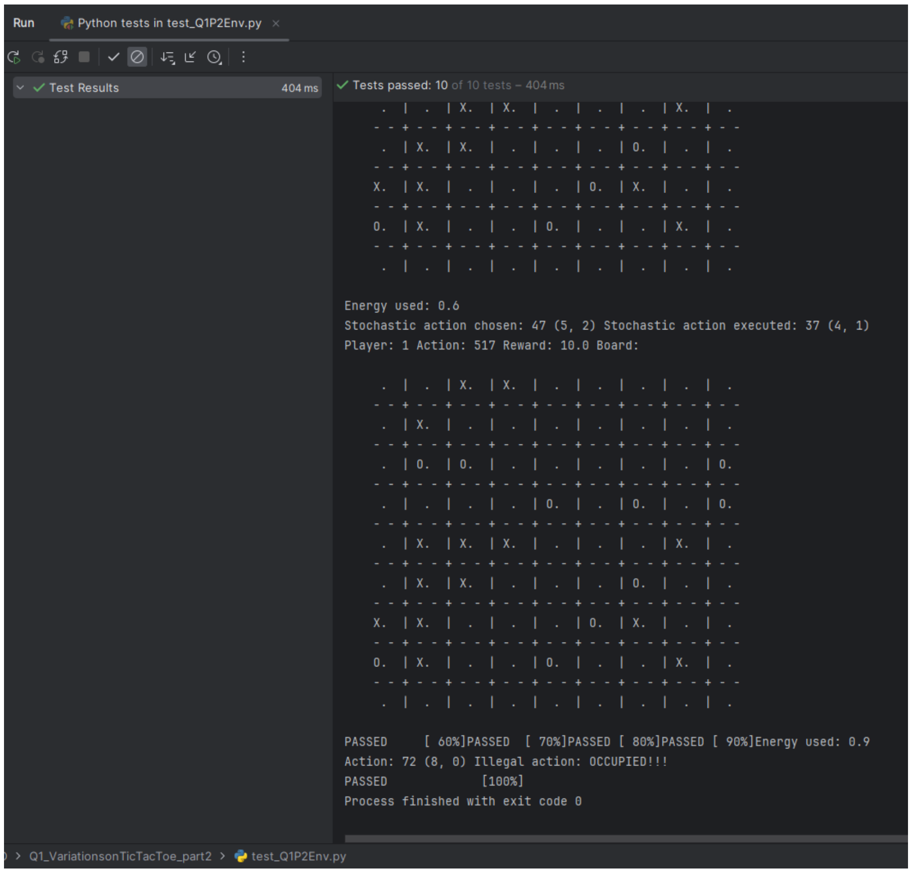
Figure 3: Outcome of simple test


Run
Python tests in test_Q1Env.py
Test Results 766 ms
Q1_VariationsonTicTacToe_part1 > test_Q1Env.py

Tests passed: 10 of 10 tests – 766 ms

```
  |   | O |   |   |   |   |   |  
- + - + - + - + - + - + - + - + -
O | O |   | O |   |   |   | O |  
- + - + - + - + - + - + - + - + -
  | X | X | X |   |   |   |   |  
- + - + - + - + - + - + - + - + -
  | O |   |   |   |   |   |   | O
- + - + - + - + - + - + - + - + -
X |   | O |   |   |   |   |   |  
- + - + - + - + - + - + - + - + -
  | O |   | X |   |   | X | O | X
- + - + - + - + - + - + - + - + -
  |   |   | X |   |   | X |   |  

Player: 1 Action: 40 Reward: 10.0 Board:

  |   |   | X |   |   |   |   |  
- + - + - + - + - + - + - + - + -
  |   |   |   | O |   | X | O | X
- + - + - + - + - + - + - + - + -
  |   | O |   |   |   |   |   |  
- + - + - + - + - + - + - + - + -
O | O |   | O |   |   |   | O |  
- + - + - + - + - + - + - + - + -
  | X | X | X | X |   |   |   |  
- + - + - + - + - + - + - + - + -
  | O |   |   |   |   |   |   | O
- + - + - + - + - + - + - + - + -
X |   | O |   |   |   |   |   |  
- + - + - + - + - + - + - + - + -
  | O |   | X |   |   | X | O | X
- + - + - + - + - + - + - + - + -
  |   |   | X |   |   | X |   |  

PASSED      [ 60%]PASSED    [ 70%]PASSED [ 80%]PASSED [ 90%]PASSED             [100%]
Process finished with exit code 0
```

Figure 4: Part 1 testing results


Run
Python tests in test_Q1P2Env.py
Test Results
404 ms
Q1_VariationsonTicTacToe_part2
test_Q1P2Env.py

```
Tests passed: 10 of 10 tests – 404 ms
  .  |  .  | X.  | X.  |  .  |  .  |  .  | X.  |  .
- - + - - + - - + - - + - - + - - + - - + - - + - -
  .  | X.  | X.  |  .  |  .  |  .  | O.  |  .  |  .
- - + - - + - - + - - + - - + - - + - - + - - + - -
 X.  | X.  |  .  |  .  |  .  | O.  | X.  |  .  |  .
- - + - - + - - + - - + - - + - - + - - + - - + - -
 O.  | X.  |  .  |  .  | O.  |  .  |  .  | X.  |  .
- - + - - + - - + - - + - - + - - + - - + - - + - -
  .  |  .  |  .  |  .  |  .  |  .  |  .  |  .  |  .

Energy used: 0.6
Stochastic action chosen: 47 (5, 2) Stochastic action executed: 37 (4, 1)
Player: 1 Action: 517 Reward: 10.0 Board:

  .  |  .  | X.  | X.  |  .  |  .  |  .  |  .  |  .
- - + - - + - - + - - + - - + - - + - - + - - + - -
  .  | X.  |  .  |  .  |  .  |  .  |  .  |  .  |  .
- - + - - + - - + - - + - - + - - + - - + - - + - -
  .  | O.  | O.  |  .  |  .  |  .  |  .  |  .  | O.
- - + - - + - - + - - + - - + - - + - - + - - + - -
  .  |  .  |  .  |  .  | O.  |  .  | O.  |  .  | O.
- - + - - + - - + - - + - - + - - + - - + - - + - -
  .  | X.  | X.  | X.  |  .  |  .  |  .  | X.  |  .
- - + - - + - - + - - + - - + - - + - - + - - + - -
  .  | X.  | X.  |  .  |  .  |  .  | O.  |  .  |  .
- - + - - + - - + - - + - - + - - + - - + - - + - -
 X.  | X.  |  .  |  .  |  .  | O.  | X.  |  .  |  .
- - + - - + - - + - - + - - + - - + - - + - - + - -
 O.  | X.  |  .  |  .  | O.  |  .  |  .  | X.  |  .
- - + - - + - - + - - + - - + - - + - - + - - + - -
  .  |  .  |  .  |  .  |  .  |  .  |  .  |  .  |  .

PASSED     [ 60%]PASSED  [ 70%]PASSED [ 80%]PASSED [ 90%]Energy used: 0.9
Action: 72 (8, 0) Illegal action: OCCUPIED!!!
PASSED              [100%]
Process finished with exit code 0
```

Figure 5: Part 2 testing results

5.2 Performances Evaluation

Upon verifying the correctness of the code implementation, the subsequent step of significance is performance evaluation. This process provides insight into the goodness of the designed agent in problem-solving. Within the scope of this project, the policy of the trained agent is compared against benchmark metrics such as a random policy from an untrained agent. Also, the creation of a user interface program to compare the trained agent policy with a human-level policy is another approach. The outcomes of these contests (i.e. win and loss) will be recorded as performance indicators for the DQN agents.
I have conducted 100 contests for Part 1 and Part 2, respectively. In Part 1, the winning rate of the DQN agent is $89 : 11$, while the winning rate in Part 2 is $83 : 17$. These results have reasonably evaluated the superior performance of the trained RL agent. Since Part 2 is more complicated with higher levels of uncertainty and game mechanisms. A lower winning rate is anticipated, and the RL's performance can be further improved with more advanced algorithms or finer-tuned hyperparameters.

6 Conclusion

In conclusion, this report delineates an optimal solution for the stochastic, extended Tic-Tac-Toe game presented in Question 1, encompassing a thorough review of potential methods from relevant literature. Considering the discrete nature of the action space inherent to the Tic-Tac-Toe game, I have developed an independent DQN based on the standard TF Agent. The results affirm the successful training and convergence of both independent DQN agents. After the training, comprehensive tests including unit tests, regression tests, and integration tests for both Part 1 and Part 2 are deployed to validate the correctness of codes' implementations. The performances of trained agents are also evaluated against untrained ones to show the effects. In a word, this project reports the whole process of solving the stochastic Tic-Tac-Toe game with TF Agent following software engineering principles from both theoretical and practical perspectives.

References

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

Richard Bellman. Dynamic programming. *Science*, 153(3731):34–37, 1966.

Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

Michael Bowling and Manuela Veloso. Multiagent learning using a variable learning rate. *Artificial intelligence*, 136(2):215–250, 2002.

Vladimir Fedorovich Dem'yanov and Vasiliĭ Nikolaevich Malozemov. *Introduction to minimax*. Courier Corporation, 1990.

Virginie Gabrel, Cécile Murat, and Aurélie Thiele. Recent advances in robust optimization: An overview. *European journal of operational research*, 235(3):471–483, 2014.

Sergio Guadarrama, Anoop Korattikara, Oscar Ramirez, Pablo Castro, Ethan Holly, Sam Fishman, Ke Wang, Ekaterina Gonina, Neal Wu, Efi Kokiopoulou, Luciano Sbaiz, Jamie Smith, Gábor Bartók, Jesse Berent, Chris Harris, Vincent Vanhoucke, and Eugene Brevdo. Tf-agents:

A reliable, scalable and easy to use tensorflow library for reinforcement learning, 2019. URL https://github.com/tensorflow/agents.

Shengyi Huang and Santiago Ontañón. A closer look at invalid action masking in policy gradient algorithms. *arXiv preprint arXiv:2006.14171*, 2020.

Kevin Jamieson and Robert Nowak. Best-arm identification algorithms for multi-armed bandits in the fixed confidence setting. In *2014 48th Annual Conference on Information Sciences and Systems (CISS)*, pp. 1–6. IEEE, 2014.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pp. 1928–1937. PMLR, 2016.

Brian Okken. *Python Testing with pytest: Simple, Rapid, Effective and Scalable*. Pragmatic Bookshelf, 2 edition, 2023.

Razvan Pascanu, Yujia Li, Oriol Vinyals, Nicolas Heess, Lars Buesing, Sebastien Racanière, David Reichert, Théophane Weber, Daan Wierstra, and Peter Battaglia. Learning model-based planning from scratch. *arXiv preprint arXiv:1707.06170*, 2017.

Warren B Powell. *Approximate Dynamic Programming: Solving the curses of dimensionality*, volume 703. John Wiley & Sons, 2007.

Martin L Puterman. Markov decision processes. *Handbooks in operations research and management science*, 2:331–434, 1990.

Sébastien Racanière, Théophane Weber, David Reichert, Lars Buesing, Arthur Guez, Danilo Jimenez Rezende, Adrià Puigdomènech Badia, Oriol Vinyals, Nicolas Heess, Yujia Li, et al. Imagination-augmented agents for deep reinforcement learning. *Advances in neural information processing systems*, 30, 2017.

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

Anthony Shaw. Getting started with testing in python, 2023. URL https://realpython.com/python-testing/.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

Maciej Świechowski, Konrad Godlewski, Bartosz Sawicki, and Jacek Mańdziuk. Monte carlo tree search: A review of recent modifications and applications. *Artificial Intelligence Review*, 56(3): 2497–2562, 2023.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. *Advances in neural information processing systems*, 28, 2015.

Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992.