Implementation of 4x4 Tic-Tac-Toe using Q Learning in Python

Nathan A. Ruprecht and Xiangnan Zhong
Department of Electrical Engineering, University of North Texas, USA

***Abstract*—In today's world of ever evolving technology, we look more into how to make computers more intelligent or able to think for themselves. Inside Artificial Intelligence, we have a subcategory that is Machine Learning. During class, we mentioned a number of techniques to implement a learning algorithm for a machine to make a decision. One such technique is the focus of this project - Reinforcement Learning. The focus of this paper is to reflect the student's understanding of the topic through applying reinforcement learning to a well-known children's game. We will cover the specific primary and secondary objectives, implementation in Python 3, and results of our game.**

***Index Terms*—Reinforcement Learning, Q Learning, Tic-Tac-Toe**

I. Introduction

Reinforcement learning was the basis of this project where the student was to implement a 4×4 sized game of Tic-Tac-Toe. Most kids know the traditional 3×3 game where two players play against each other to make 3 in a row of their mark; player 1 who goes first is usually 'X', while the opponent is 'O'. It turns into a game of both defense to block the opponent and offense to take advantage and get 3 in a row to win the game. With a 3×3, there are 8 ways to win: 3 in a row on each row, or each column, or the diagonals. For our 4×4, there are 10 ways to win. With that, there are $3^{16} = 43,046,721$ possible game boards - 'X', 'O', and ' ' as possible spaces and 16 spaces. The primary objective of this project is to:

- Use a programming language of our choice to implement a Tic-Tac-Toe game
- Game board size of 4×4
- Use Q learning algorithm

I took on secondary objectives for this project being:

- Implement board sizes of 3, 4, and 5
- Compare how much training is "enough" training for the computer
- Vary a parameter with the 4×4 to see how it affects computer learning capabilities

With that in mind, this paper is organized as follows: the next section covers Q learning theory to prove the author has learned the foundation, then Section III covers the implementation of turning theory to code, Results in Section IV will show code within the while-loop that generated data followed by the actual tables/figures, then a discussion of what I thought went well or wrong and how future work can better the results, and finally a conclusion of the project as a whole in Section VI.

II. Q Learning Theory

The idea of reinforcement learning is that the learner will have some type of reward for completing a task. Whether it's a positive or negative reward depends on if the learner successfully completes said task. This approach is used most notably with children: using allowance to incentivize children doing chores, or play time to reward completion of homework or studying, et cetera. Breaking down the pieces is how we apply it to machines.

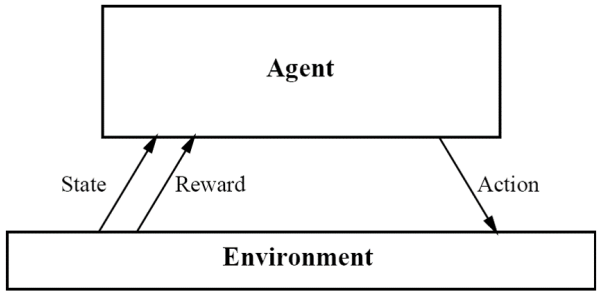

Fig. 1. Reinforcement Learning Breakdown

Looking at Figure 1, we see that the Agent (learner) interacts with the Environment (chores or homework). The Agent does some action, and a resulting reward and next state is fed back. So the child bags the trash, the next state is to take it out and no reward yet. If they continue on that path, they take out the trash with no next state (they are done) and the reward of an allowance. This flow is better shown by:

$$s_0 \xrightarrow[a_0]{r_0} s_1 \xrightarrow[a_1]{r_1} ... \xrightarrow[a_{n-1}]{r_{n-1}} s_n, \quad (1)$$

where s, a, and r are the state, action, and reward index respectively. If the child is new and not used to it, there are many numbers of steps to take. Those possible actions and paths create a flow of Q values with a machine learning those paths by:

$$Q(s) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} ... \quad (2)$$

with γ being a discount factor. A low value for γ would have the computer playing the short term for the more immediate reward, while a high γ plays for long term.

In the case of the child, they want to maximize their reward in the fewest steps possible - take out the trash as fast as possible to get it over with and receive their allowance. Without knowing it, the child is already looking as many steps into the future to decide the best path. This path is what we call a policy, π. From the possible Q values, the best path is defined by Equation 3 and the path chosen by the machine given by Equation 4.

$$\hat{Q}(s,a) = r + \gamma \max_{a'} \hat{Q}(s', a') \quad (3)$$

$$\pi^*(s) = \max_a Q(s,a) \quad (4)$$

With that, creating the Q table is an iterative process of the computer trying different paths to find its reward, and update the optimal path to it. We can give the computer either a positive reward for completing the task, a negative for failing to complete, or zero if there is a middle ground (such as a tie game).

III. Theory to Code Implementation

Going off of Equation 3, we can add another parameter that adjusts how much the Q table will move each step, a learning rate value shown by α. A low α will mean small adjustments to the Q table relying more on previous values, while a large will give more weight to the updated value. Thus Equation 3 can be shown as:

$$\hat{Q}(s,a) = (1-\alpha)\hat{Q}(s,a) + \alpha[r + \gamma \max_{a'} \hat{Q}(s', a')] \quad (5)$$

which is shown in Python by the below code where if the next move will end the game, the expected reward is the final reward, otherwise we use the above equation.

```
def learn_Q(self, move):
    state_key = Trainer.
        make_and_maybe_add_key(self.
        board, self.current_player.mark
        , self.Q)
    next_board = self.board.
        get_next_board(move, self.
        current_player.mark)
    reward = next_board.give_reward()
    next_state_key = Trainer.
        make_and_maybe_add_key(
        next_board, self.other_player.
        mark, self.Q)
    if next_board.over():
        expected = reward
    else:
        next_Qs = self.Q[
            next_state_key]
        if self.current_player.mark ==
            "X":
            expected = reward + (self.
                gamma * min(next_Qs.
                values()))
        elif self.current_player.mark
            == "O":
            expected = reward + (self.
                gamma * max(next_Qs.
                values()))
    change = self.alpha * (expected -
        self.Q[state_key][move])
    self.Q[state_key][move] += change
```

Then Equation 4 is implemented with:

```
def get_move(self, board):
    if np.random.uniform() < self.
        epsilon: #Exploration!
        moves = board.available_moves
            ()
        if moves:
            return moves[np.random.
                choice(len(moves))]
    else: #No exploration...
        state_key = Trainer.
            make_and_maybe_add_key(
            board, self.mark, self.Q)
        Qs = self.Q[state_key]
        if self.mark == "X":
            return Trainer.
                stochastic_argminmax(Qs
```

```
            , max)
        elif self.mark == "O":
            return Trainer.
                stochastic_argminmax(Qs
                , min)
```

We introduce another parameter which controls how much the computer explores different paths, ϵ. With a higher value, the computer will explore more to try different paths. This is also a good time to point out the reward system. If the computer with 'X' wins, it receives a +1 reward. If 'O' wins, it receives a -1 reward. If it's a 'Cat' then the reward would be slightly positive at 0.1. With that, it makes sense in the above code that if the player is 'X' then they should go for the policy that maximizes the reward in order to achieve +1, while 'O' would minimize it to a goal of -1.

During training, we have the computer play against itself with a high valued ϵ, then during the test games that matter, we set $\epsilon = 0$ so that both players take the shortest path to victory. How we structure the while-loop is having the computer "train" itself for a number of games (NUM_TRAINING).

```
while NUM_TRAINING < 2**21:
    if NUM_TRAINING != 1:
        filename = "Q_final_{}.p".
            format(int(NUM_TRAINING/2))
    Q = pickle.load(open(filename, "rb
        "))
    p1 = Trainer(mark="X",epsilon =
        epsilon)
    p2 = Trainer(mark="O",epsilon =
        epsilon)
    game = Game(root, p1, p2, Q=Q)
    start = time.time()
    for i in range(int(NUM_TRAINING/2)
        , NUM_TRAINING):
        print(i, "out of",
            NUM_TRAINING)
        game.play()
        game.reset()
    end = time.time()-start
    Q = game.Q
    filename = "Q_final_{}.p".format(
        NUM_TRAINING)
    pickle.dump(Q, open(filename, "wb"
        ))
```

Then, the computer will "test" itself for 100 games as to show a histogram of whether player 1 wins, player 2 wins, or a tie game represented as a "Cat."

```
Q = pickle.load(open(filename, "rb"))
p1 = Trainer(mark="X", epsilon=0)
p2 = Trainer(mark="O", epsilon=0)
game = Game(root, p1, p2, Q=Q)
game.reset()
for i in range(100):
    game.play()
if game.board.winner() is None:
    cats += 1
elif game.board.winner() is "X":
    x += 1
elif game.board.winner() is "O":
    o += 1
game.reset()
print("Trial", NUM_TRAINING, "\tTime
    {0:.4f}".format(end/60), "\tCats",
    cats, "P1", x, "P2", o)
NUM_TRAINING = NUM_TRAINING*2
```

In the above case, we would increment NUM_TRAINING by using powers of 2 up to 2^{20}. To avoid starting over at 1 for each training value, we just remember the past value used and compensate. This is best shown by a screenshot of the Python console shown in Figure 2.

```
0 out of 1
Trial 1         Time 0.0008     Cats 42.0 P1 27.0 P2 31.0
1 out of 2
Trial 2         Time 0.0003     Cats 35.0 P1 34.0 P2 31.0
2 out of 4
3 out of 4
Trial 4         Time 0.0008     Cats 36.0 P1 38.0 P2 26.0
4 out of 8
5 out of 8
6 out of 8
7 out of 8
Trial 8         Time 0.0013     Cats 44.0 P1 25.0 P2 31.0
8 out of 16
9 out of 16
10 out of 16
11 out of 16
12 out of 16
13 out of 16
14 out of 16
15 out of 16
Trial 16        Time 0.0025     Cats 40.0 P1 25.0 P2 35.0
16 out of 32
17 out of 32
18 out of 32
19 out of 32
20 out of 32
```

Fig. 2. Python Console of Results

With that, the true time it takes to train the computer is actually the sum of that current trial along with all

previous trials. The Q table is saved to a file after each iteration so we can go back and play the computer as if it were trained a certain number of times. Again, the primary objective of the project is to successfully implement a 4×4 game, while secondary objectives were to compare the minimum level of computer training for a 3×3 versus a 4×4, and 5×5, while at the same time varying ϵ within a 4×4 to see how it changed the performance of the computer.

IV. Results

The first 3 figures shown is varying the size of the game board, while the second set of 3 figures will be showing the 4×4 with a varying ϵ. In both cases, it's a shared y-axis plotting where the left hand side shows the histogram of who won while the right side shows the time it took in hours to train the computer to that level of competency.

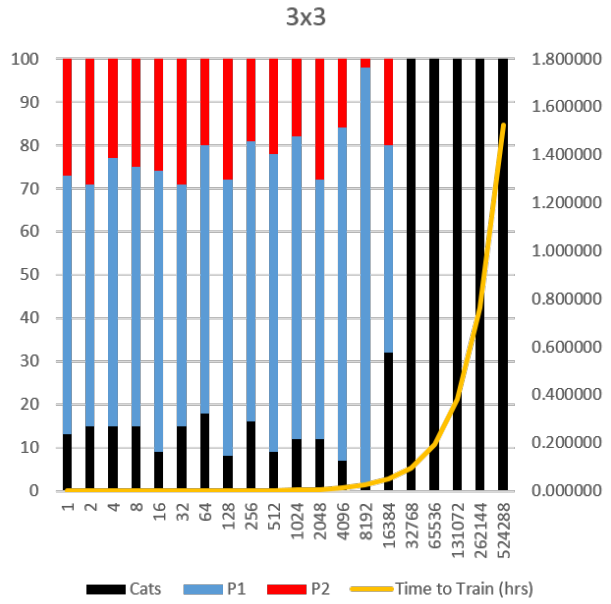

Fig. 3. 3x3 Game

With varying size shown in Figures 3, 4, and 5, we confirm our assumption that it takes significantly longer to train the computer to be an "expert" in that the computer cannot beat itself. Understandably so since we move from 3^9 possible games, to 3^{16}, to 3^{25}.

With varying ϵ shown in Figures 6, 7, and 8 we do not see much change. There is slight change in time it takes to train, most notable when $\epsilon = 0.9$, it takes significantly less time to train as many games to its counterparts. Figure 8 is the same as Figure4 since we held epsilon constant when varying size.

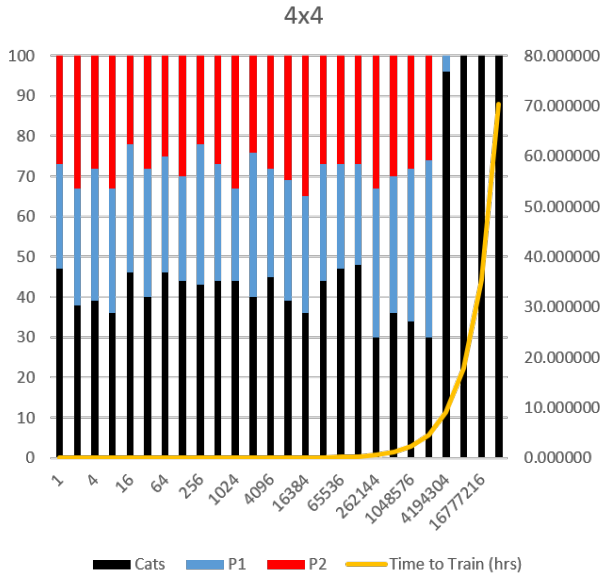

Fig. 4. 4x4 Game

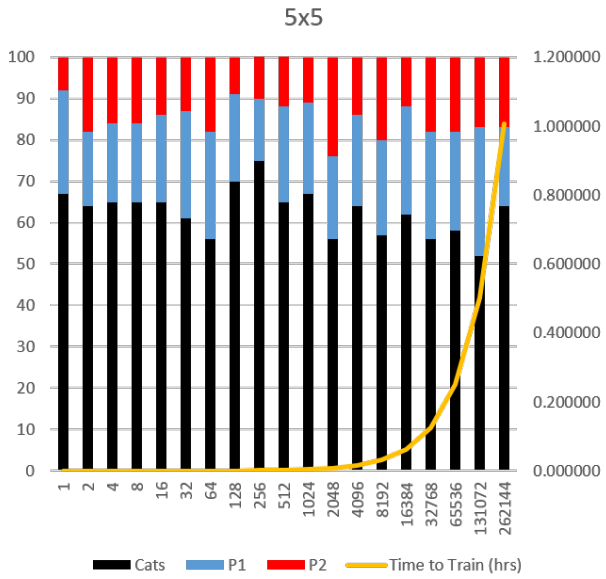

Fig. 5. 5x5 Game

V. Discussion and Future Work

I did not get to see how many games are needed to be enough training for a 5×5. The limiting factor was without a doubt the machine used to train. It seems, under these circumstances, that 2^{15} is enough training for a computer to be an expert for a 3×3 game of Tic-Tac-Toe, while it takes 2^{23} games to train it for a 4×4. Unfortunately, I did not train it enough to know how much training is enough for a 5×5 game.

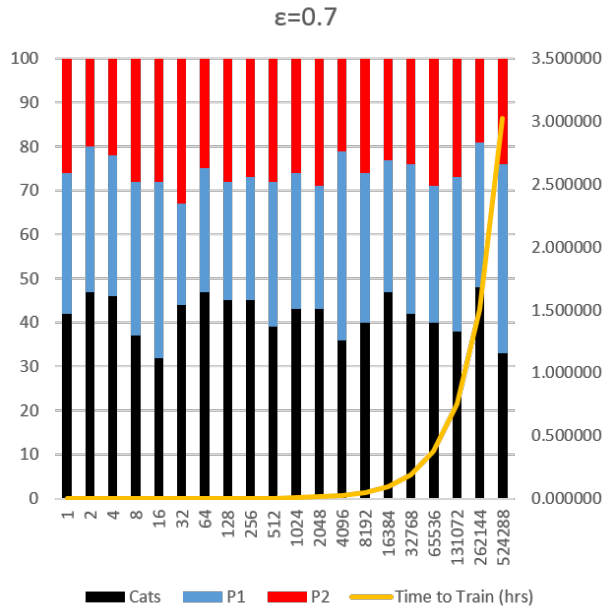

Fig. 6. $\epsilon = 0.7$

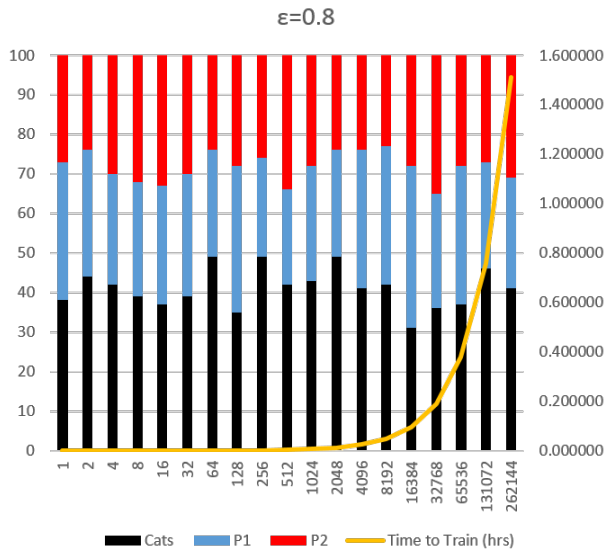

Fig. 7. $\epsilon = 0.8$

In future work, I would like to see that training boundary for size=5, along with varying other parameters besides ϵ to see how they affect both time and competency. Again, a huge factor that posed a problem was the specifications of the machine that was running the program. I will most definitely be upgrading my desktop to specialize in machine learning simulations in foresight of doctoral research.

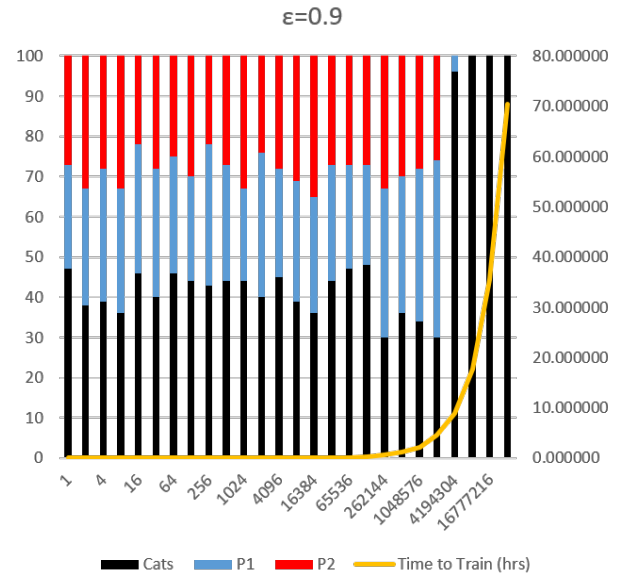

Fig. 8. $\epsilon = 0.9$

VI. Conclusion

Overall, I grasped the basic foundation for reinforcement learning and more specifically Q learning. To recap on the primary objectives of this project, I did successfully implement a 4×4 Tic-Tac-Toe game based on Q learning. For personal goals, I was able to make size a global variable as to implement a 3×3, 4×4, and 5×5 board size along with a varying ϵ to observe changes in learning capabilities. I am still in the process of uploading files to my GitHub so that Q tables can be downloaded and ran along with complete source code.

GitHub url: https://github.com/NathanRuprecht/EENG5940CompIntel/tree/master/Project

References

[1] Tom M Mitchell. *Machine Learning*. McGraw-Hill Componances, Inc, Massachusetts, 1997.

[2] Khpeek. *Q learning Tic Tac Toe*. https://github.com/khpeek/Q-learning-Tic-Tac-Toe. GitHub, 2017.