Creating a Reinforcement Learning AI to play Connect Four

Brandon Leung

Section 1: Introduction

Reinforcement learning (RL) is a subfield of machine learning which allows one to solve Markov Decision Processes (MDPs) even without a known motion model $p_f(x'|x,u)$ and known cost function $\ell(x,u)$. Instead, they are inferred by access to examples of system transitions and incurred costs. RL techniques can be categorized into two groups: deep reinforcement learning, which uses neural networks, and classical reinforcement learning, which do not. In recent years, deep RL has seen great success – however, there are some trade-offs, such as computational (i.e. GPU) complexity. On the other hand, classical reinforcement learning has been studied for many decades, and more is known about it (in terms of theoretical guarantees), but at times is outperformed by deep RL methods.

In this report, we focus on studying two classical reinforcement learning algorithms: Q-learning and Monte-Carlo policy iteration. These techniques are applied to a two-player game called Connect Four, which is a game similar to tic-tac-toe, in order to learn a policy which will allow an AI agent to play the game at a high level. In Section 2, we formally describe the game as an MDP. Then, in Section 3 we describe our RL algorithms and how it can be applied to the MDP formulation. This includes the generation of episodes using self-play, which we will use to update our policy. Finally, in Section 4 the final results are presented on a variety of different opponents. We also give some insight into the affect of various parameters and give some visualizations of the results.

Figure 1. *Left: A connect four game where the boy (red) is about to win. Right: A situation where the game ends in a tie*

Section 2: Problem Formulation

Connect Four is a game similar to tic-tac-toe, except for three main modifications:

- The board is larger (officially, it is 7 columns, 6 rows though this can be modified without changing the other rules)
- The player needs 4 in a row (either diagonally, horizontally, or vertically) to win
- The player picks a column, and their marker always falls to the lowest row of that column (ie, there is "gravity" involved in the game)
- The markers are black/red instead of x/o (this is just a cosmetic change)

Formally, the game can be described as a finite-horizon MDP as follows:

- The state space $\mathcal{X}$ is a subset of all possible, legal configurations of the board. Since any given position can either be red, black, or empty, a crude upper bound for a 7x6 board is $|\mathcal{X}| < 42^3$. However, this is an overapproximation because it also contains many impossible states, such as a board with all red entries. Having all red entries is impossible for two reasons: because the game would have ended by then, and also, the number of red markers differ from the number of black markers by at most 1 (since players take turns). It is nontrivial to calculate the exact size of the state space. However, this was computationally calculated by Tromp [1] to be 4,531,985,219,092 in the case of a 7x6 board.
    - The reason why $\mathcal{X}$ is a subset of all legal board configurations is because for our purposes, it is more helpful to assume the position of a player (eg, the red player, whose turn is first). Then from this perspective, $\mathcal{X}$ becomes all the states that the red player can see and immediately take their turn. For instance, assuming red goes first, all the states in $\mathcal{X}$ will have an even number of markers.

- The control space $\mathcal{U}$ represents all the columns in the board. For example, if the board's size is 7 columns, 6 rows and we denote placing the marker in numbered columns starting from 0 from the left, then $\mathcal{U} = \{0,1,2,3,4,5,6\}$.

- $\ell(x,u) = 0 \ \ \forall\, x \in \mathcal{X}, \forall\, u \in \mathcal{U}$. Since there is no obvious way to set the stage costs, we simply set them to all be zero, and rely on the terminal cost instead.

- $q(x)$ is the state cost. It is set to -1 if the player wins (ie, $q(x) = -1$ if $x \in \mathcal{X}$ is a winning state with four in a row), 1 if the player loses, -0.5 if the game ends in a tie, and 0 otherwise. Note that these values can be changed depending on one's interpretation of a tie (for example, if one views ties as very undesirable, then it could be set to a higher number).

- Because Connect Four is a two-player game, the transition model $p_f(\cdot|x_t, u_t)$ depends on the player's opponent. It represents a probability distribution over the next board configuration that a player will be able to act upon and make an input, given their current board configuration x_t and its next input u_t. The main unknown factor is what the player's opponent will do during their turn before returning control to the current player. Some examples of different opponents are discussed in Section 3b.

Assuming all the above are known, ideally we would like to find $V^*(x), \pi^*(x)$ such that the Bellman equation is satisfied:

$$V^*(x) = \min_{u \in \mathcal{U}(x)} \left(q(x) + \gamma \sum_{x' \in \mathcal{X}} p_f(x'|x,u) V^*(x') \right), \forall x \in \mathcal{X}$$

$$\pi^*(x) = \arg\min_{u \in \mathcal{U}(x)} \left(q(x) + \gamma \sum_{x' \in \mathcal{X}} p_f(x'|x,u) V^*(x') \right), \forall x \in \mathcal{X}$$

Then, $\pi^*(x)$ can be used to make optimal control decisions given a state. This Bellman equation states the necessary condition for optimality. Intuitively, it asserts that in the optimal case, the expected long-term cost at a certain point in time $V(x)$ can be written as the payoff from some initial choices $\ell(x, u_{\min})$ and the discounted, expected long-term cost of the remaining decision problem $V(x')$. In order to be optimal, this property must be true for all $x \in \mathcal{X}$, and since the time horizon is infinity, $V(x)$ and the choice of $u \in \mathcal{U}(x)$ should be time-invariant. It can be shown that the solution to the Bellman equation is unique, and in totality, the minimizing control $u \in \mathcal{U}(x)$ for each $x \in \mathcal{X}$ gives a stationary optimal policy. Thus, at the end for each $x \in \mathcal{X}$ we should have an optimized expected long-term cost $V^*(x)$ as well as a corresponding $u \in \mathcal{U}(x)$ which specifies the optimal control.

However, solving this exactly is impossible since in our RL formulation we don't know $p_f(\cdot|x_t, u_t)$. Instead, we assume that we have can have access to many episodes of the form:

$$\rho_i = x_0, u_0, x_1, u_1, \dots, x_{T-1} u_{T-1}, x_T \sim \pi$$

where some policy function $\pi: \mathcal{X} \to \mathcal{U}$ is used to give us the control as function of a state input. Here, T is not a constant – it can vary depending on $p_f(\cdot|x_t, u_t)$ and π. However, it is always a finite number in the case of Connect Four. Towards solving this MDP, in the next section we describe a few RL algorithms which can provide an estimate for $\pi^*(x)$ using our episodes ρ_i, even without explicit knowledge of $p_f(\cdot|x_t, u_t)$. We also discuss a method called self-play which can online, automatically generate a dataset of episodes $\{\rho_i\}$.

Section 3a: Technical Approach – Solving MDPs with RL

There are many different RL algorithms which have been proposed. In this report, we study two popular ones: First-Visit Monte Carlo Policy Iteration (MC-PI) and Q-Learning, both using an ϵ-soft policy to encourage exploration. We first take a look at MC-PI; the algorithm can be derived as follows. First, recall that if we have $p_f(\cdot|x_t, u_t)$, then the policy iteration algorithm will converge to satisfy the Bellman equation:

Psuedocode 1: Policy Iteration Algorithm

***Given**: initial random guess $V_0(x), \pi_0(x), \forall x \in \mathcal{X}$, and some ϵ to determine if the Bellman equation is satisfied*

$i = 0$

While True:

$$V_{i+1}(x) = \ell\big(x, \pi_i(x)\big) + \gamma \sum_{x' \in \mathcal{X}} p_f\big(x' \big| x, \pi_i(x)\big) V_i(x') \,, \forall x \in \mathcal{X}$$

$$\pi_{i+1}(x) = \underset{u \in \mathcal{U}(x)}{arg\ min} \left(\ell(x,u) + \gamma \sum_{x' \in \mathcal{X}} p_f(x'|x,u) V_i(x') \right), \forall x \in \mathcal{X}$$

$i = i + 1$

If $|V_i(x) - V_{i-1}(x)| < \epsilon, \forall x \in \mathcal{X}$*:*

Break

We can see that this first involves a policy evaluation step, where we are given a policy and evaluate V^π . Next, given V^π, we obtain a new and improved stationary policy π'. Theoretically, it can be proven that each new π is at least as good as the policy before it, and overall, the policy iteration algorithm converges to an optimal policy. However, the execution of this algorithm requires knowing p_f . This can be avoided with the following observations:

- Note that that $V_{i+1}(x) = \mathbb{E}_{\rho \sim \pi}[L_\tau(\rho)|x_\tau = x] \approx \frac{1}{k}\sum_{k=1}^{K} L_\tau(\rho^k)$, where $L(x_\tau, u_\tau, \dots, x_{T-1}u_{T-1}, x_T) = q(x_T) + \sum_{t=\tau}^{T-1} \ell(x_t, u_t)$. That is, V_{i+1} is essentially the expected long-term cost, over episodes ρ.
    - Thus, by the law of large numbers if we had many such episodes, we could compute the cost of those episodes and average them to estimate V_{i+1}.

- We can generalize the expression from $V^\pi(x)$ to $Q^\pi(x,u) = \ell(x,u) + \gamma E_{x' \sim p_f(\cdot|x,u)}[Q^\pi\big(x', \pi(x')\big)]$. Thus, we allow the first input to be anything, and then follow policy π afterwards. With this modification, it follows that $\pi'(x) = \arg\min_{u \in \mathcal{U}(x)} Q^\pi(x,u)$.

From the points above, we can adapt Policy Iteration to the MC-PI algorithm:

Psuedocode 2: First-Visit Monte Carlo Policy Iteration Algorithm (MC-PI)

Given: *initial random guess* $Q(x,u), \forall x \in \mathcal{X}, u \in \mathcal{U}$ *, some learning rate* α

Until convergence:

Define π *to be an* ϵ*-greedy policy derived from the current* $Q(x,u)$ *function*
Somehow generate an episode $\rho = x_0, u_0, x_1, u_1, \dots, x_{T-1}u_{T-1}, x_T$ *using* π
For each $(x_\tau, u_\tau) \in \rho$ *do:*

$$L(x_\tau, u_\tau, \dots, x_{T-1}u_{T-1}, x_T) = q(x_T) + \sum_{t=\tau}^{T-1} \ell(x_t, u_t)$$
$$Q(x_\tau, u_\tau) \leftarrow Q(x_\tau, u_\tau) + \alpha\big(L - Q(x_\tau, u_\tau)\big)$$

For MC-PI, note the following:

- In the first step, $\pi(x)$ is defined to be a stochastic policy which chooses the natural, optimal choice $\arg\min_{u \in \mathcal{U}(x)} Q(x,u)$ most $(1-\epsilon) * 100$ percent of the time, and some other control $u \in \mathcal{U}(x)$ for the remaining $\epsilon * 100$ percent of the time. Since $\epsilon \in [0,1]$, when $\epsilon = 0$ we are back at a deterministic optimal policy and when $\epsilon = 1$ we are picking the control uniformly randomly.
    - This introduction of stochasticity encourages more exploration of the state/input space for the Q function
- After we have π, we need to somehow generate ρ from it. This is non-trivial for Connect Four because it is a two-player game. We discuss the "self-play" method of generating episodes ρ given π in Section 3b.
- The average of the policies' long-term costs is written as an online, running average weighted by a learning rate α.
- It can be shown the MC-PI converges if the sequence of ϵ-greedy policies is "Greedy in the Limit with Infinite Exploration" (GLIE). This essentially means that ϵ eventually becomes zero.

Now that we have established MC-PI, Q-learning easily follows if we use bootstrapping to approximate the long term cost:

Psuedocode 3: Q-Learning

Given: *initial random guess* $Q(x,u), \forall x \in \mathcal{X}, u \in \mathcal{U}$ *, some learning rate* α

Until convergence:

Define π *to be some policy*
Somehow generate an episode $\rho = x_0, u_0, x_1, u_1, \dots, x_{T-1}u_{T-1}, x_T$ *using* π
For each $(x_\tau, u_\tau, x_{\tau+1}) \in \rho$ *do:*

$$Q(x_\tau, u_\tau) \leftarrow Q(x_\tau, u_\tau) + \alpha\left(\ell(x_\tau, u_\tau) + \gamma \min_{u' \in \mathcal{U}(x_{\tau+1})} Q(x_{\tau+1}, u') - Q(x,u)\right)$$

For Q-learning, note the following:

- It can be shown that this converges even if π is arbitrary; it does not have to be GLIE.
- As for MC-PI, generating ρ from π is non-trivial for Connect Four because it is a two-player game. We discuss the "self-play" method of generating episodes ρ given π in Section 3b.
- γ is the discount factor, and determines the importance of future rewards. For finite-horizon cases where there will always be a terminal state reached (as in the case of Connect Four), we can set $\gamma = 1$.

Section 3b: Technical Approach – Generating Meaningful Episodes

As mentioned in Section 2, Connect Four is a two-player game so the transition model $p_f(\cdot|x_t, u_t)$ depends on the player's opponent. This arguably makes the creation of meaningful, informative episodes more complicated than a fully observable one player game (like the Atari game Breakout), or physics tasks like pendulum balancing. The easiest way to solve this problem, assuming that we want our Connect Four AI to do well against human players, is to obtain a very large dataset of real, human played Connect Four episodes at many different skill levels. Then, MC-PI or Q learning can be applied in a straightforward way. However, this is impractical. A more feasible method would be to somehow have simulate the opponents move. At a high level, some example opponents could be:

- A trivial, deterministic opponent which always places their marker into the leftmost valid column
- A random opponent which choses valid columns to place their marker in a uniformly random way
- An opposing "beginner", "medium", or "advanced" AI opponent

Clearly, the first choice would not lead to meaningful progress, since our trained AI would overfit to this opponent playstyle. The second choice is slightly more compelling, but could only ever be guaranteed to do well on random opponents. The third choice is the most desirable, and we can emulate this behavior by using self-play. In essence, we will also have the opponent derive their moves from the latest Q-function. This means that we will need to track episodes for the opponent and also run MC-PI or Q learning on those opponent episodes to update their Q function. This is because the state space for the opponent is different from the AI. For example, if our AI goes first, then the AI always applies inputs to board states with an even number of markers while for the opponent, it is always an odd number. In this way, the algorithm learns the game completely from scratch with no outside help. We detail we detail the results of generating episodes through random opponents versus self-play in Section 4.

Section 3c: Technical Approach – Implementation Choices

So far we have focused on mathematically characterizing Connect Four as an MDP and presented two algorithms to approximate an episode dataset based solution with RL. However, as is often the case are several additional important details to note when actually implementing the algorithms:

- As mentioned in Section 2, the state space from a 7x6 board is enormous (it's 4,531,985,219,092 states). As a result, for the purposes of this report all experiments have been performed on a 5x4 board instead. This is due to computational and time limitations.
    - In principle, the method could also be extended to 7x6 boards. In terms of the theory, this transition to a larger board is trivial. One option could be to use a function approximation, but this is usually best suited if one were to use a neural network for the approximator (this report focuses on classical RL, not deep RL). A linear function approximation is also possible, but would lead to increased complexity and likely decreased performance, at least in the 5x4 case, since sampling episodes is cheap. It is also likely that at deployment, a table-based method (as presented so far) is fastest, if the table is implemented as a hash-map.
- The initializations for policy/value iteration V_0, π_0 are set to be a vector of zeros.
- All code was implemented in Python, and all calculations are as vectorized as possible for efficiency.

Section 4a: Connect Four Play Results

In this section we report the results of our Connect Four RL AI. We show the results for Q-Learning and MC-PI for several different parameters of ϵ (for the ϵ-greedy policy) and learning rate α. Since Connect Four is finite-horizon (there is always a terminal state reached), we always set $\gamma = 1$. In order to evaluate the results, we test our AI every 100,000 iterations (an iteration is composed of generating a new episode and updating the Q function). It is tested against three different opponents: a random opponent, MCTS_25, and MCTS_50. Their descriptions are as follows:

- The random opponent, choses valid columns to place their marker in a uniformly random way
- The MCTS opponents are based on an algorithm called Monte-Carlo Tree Search (MCTS), which works by exhaustively trying different possible moves it can perform in simulations against itself. Publicly available code was used for this part [2] (I did not implement it), since MCTS has little to do with RL and is outside the scope of this report. For our purposes, we can treat it as a "black-box" opponent to test our RL methods against.
    - One advantage of MCTS is that it can be tuned to play at different skill levels. For example, MCTS_50 refers to a MCTS algorithm which choses a move after simulating 50 games, while MCTS_25 choses a move after simulating 25 games.
    - Note that the MCTS opponent is used only for testing purposes; it is not used for generating any episodes to train on.

Table 1 below shows the results for Q-learning, over 6,000,000 iterations. The y-axis demotes the rate of winning, averaged over 150 games each 100,000 iterations. We can see that in general, having a learning rate of 0.9 and an epsilon in the range [0.05,0.15] leads to the best performance. Otherwise, other parameter configurations (such as a higher/lower epsilon not in the range) leads to bad performance against the MCTS opponents. However, all parameter configurations do reasonably well on the random opponent. Next, Table 2 shows Q-

learning results for random-play (where episodes are generated by a random opponent instead of self-play). We can see that the performance against a random opponent is high, but the performance against the MCTS opponents suffer because the episodes are not rich enough. Therefore, this validates the use of self-play as an effective strategy to generate new episodes to train on. Finally, in Table 3, we show results for MC-TS. Experimentally, I found that Q-learning outperformed MC-TS. This could be because Q learning seems to require less conditions for convergence compared to MC-TS.

Q-Learning Self-Play Results

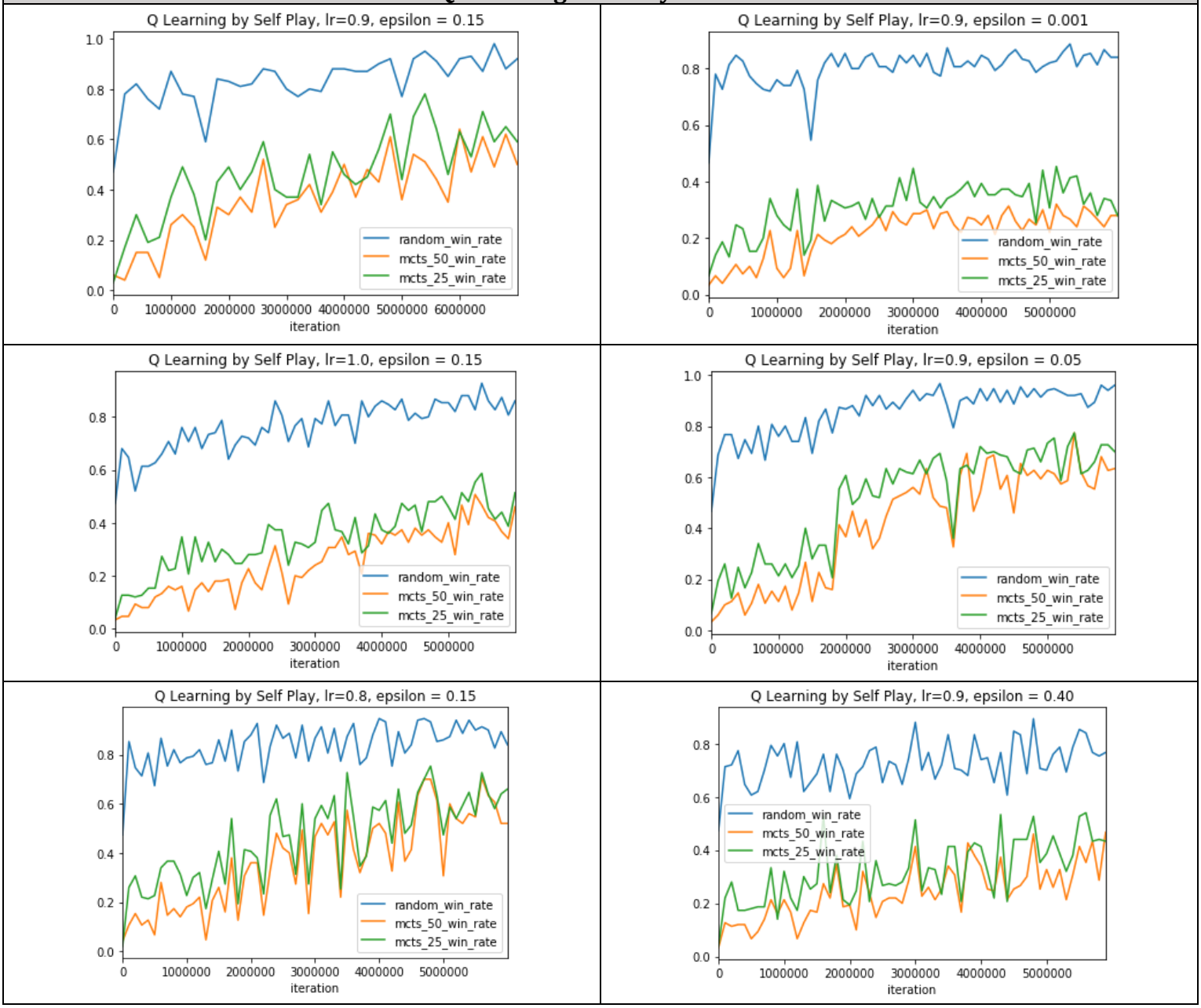


Table 1. *Training results for Q-Learning with self-play generated episodes, for a few different parameter choices.*

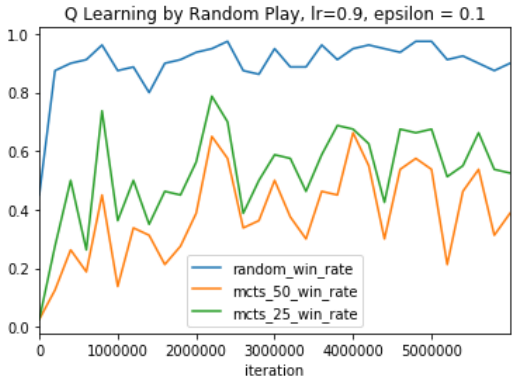
Q-Learning Random-Play Results


Table 2. *Training results for Q-Learning with random-play generated episodes.*

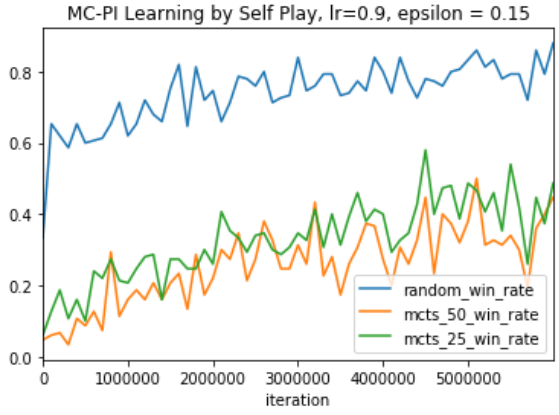
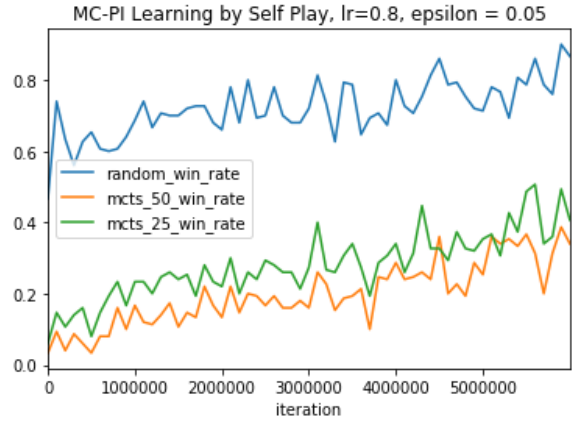
MC-PI Results	
	

Table 3. *Training results for MC-PI learning with self-play generated episodes.*

Section 4b: Visualizations & Example Games

In this section, we provide a few example games against a random opponent (Table 4). Only relatively short games are shown, to save space. Our trained RL agent always goes first, and is the 'x' player. The opponent always goes second, and is the 'o' player. When it is the RL agent's turn, we show a heatmap for values of the learned $Q(x,u)$ function, where x is the current board state and $u \in \mathcal{U}(x)$. Lighter shades of red denote lower values, while darker shades for red denote higher values. The heatmap values are for the valid input columns, from left to right. This intuitively gives a sense of which columns are more advantageous to place the next move. The Q function is learned with Q-Learning, where the learning rate is 0.9 and epsilon is 0.15, since this performs well (as shown in Section 4a).

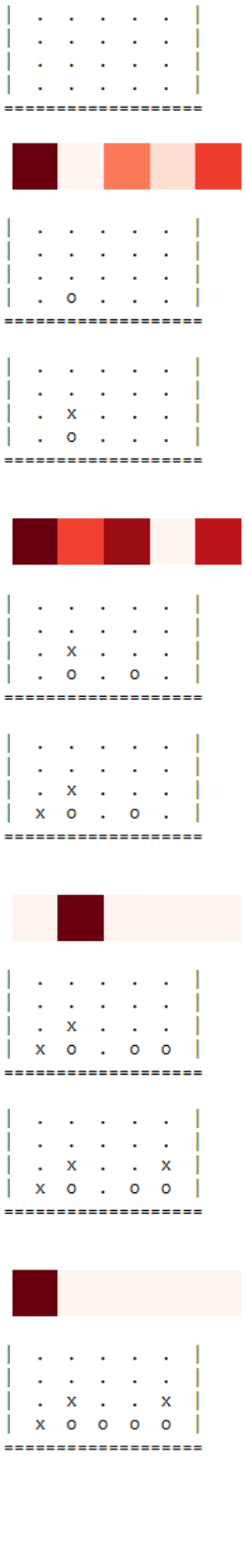
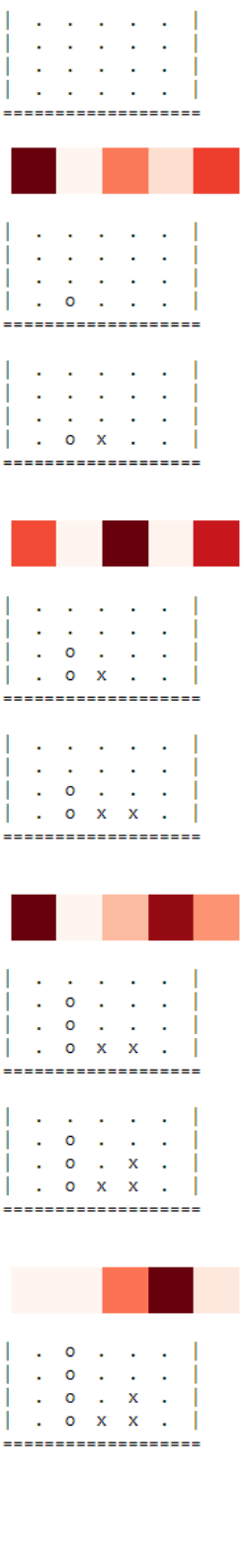
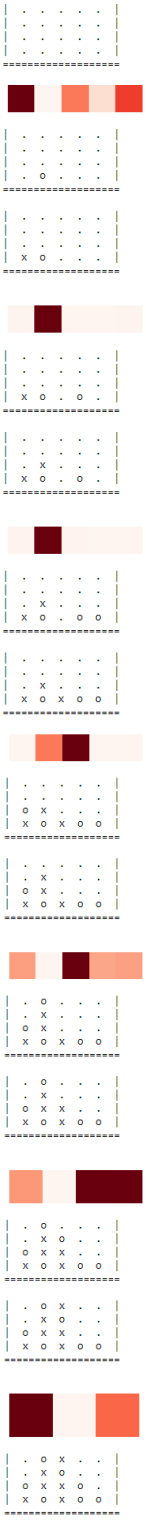
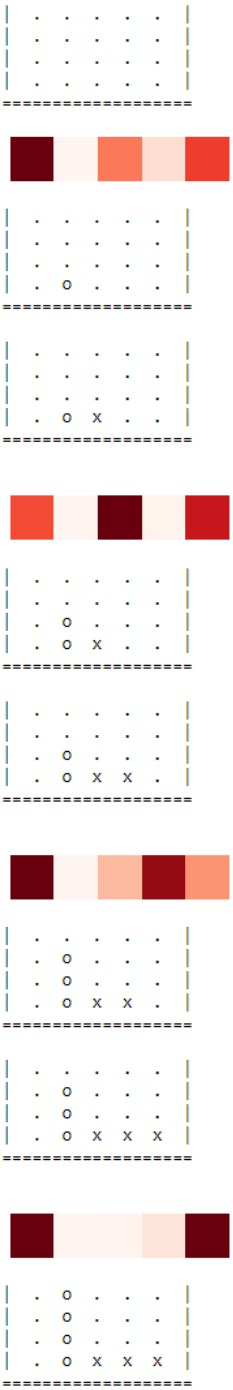
Random Game 1	Random Game 2	Random Game 3	Random Game 4	Random Game 5
				

***Table 4**: Examples of simulated games against a random opponent, with the $Q(x,u)$ function heatmaps shown.*

Section 5: References

[1] Tromp, J.: Solving connect-4 on medium board sizes. ICGA Journal 31:1, 110–112 (2008)
[2] https://github.com/codebox/connect4