Project 4 - RL and MCTS

Hoang-Dung Bui,
George Mason University
Fairfax, USA
hbui20@gmu.edu

I. Q Learning

The code of the Q-update is shown in Fig. 1. In this code, we get the new sets of possible actions *new_actions* of the *new_state*, and save the value of Q-function into a vector v_a. If the strategy is greedy, we determine the action by the best value function of the *new_state* among all the possible action in *new_actions*. If it is the ϵ one, we randomly select one of the valid actions. The policy is updated by the optimal action.

```
actions = env.get_actions_for_state(state)
v_a = []
for act in actions:
    q_a = Q_s_a[state, act]
    v_a.append(q_a)
if np.random.rand() > epsilon:
    indx = np.argmax(v_a)
    action = actions[indx]
else:
    indx = np.random.randint(0,len(actions))
    action = actions[indx]
r, new_state = env.execute_action(state, action)
# Update Q
maxQ = np.max(Q_s_a[new_state])
Q_s_a[state, action] += learning_rate*(r+gamma*maxQ -
                                Q_s_a[state, action])
```

Fig. 1: The code of Q-learning algorithm

We use the linear algebra to build the $evaluate\_policy$ function.

```
def evaluate_policy(env, policy, gamma=0.98):
    """Returns a list of values[state]."""
    P = np.zeros((len(env.states),len(env.states)))
    R = np.zeros(len(env.states))
    I = np.eye(len(env.states))
    for ii, state in enumerate(env.states):
        R[ii] = env.rewards[ii]
        action = policy[state]
        prob_vec = env.get_transition_probs(state, action)
        P[ii] += prob_vec
    V = np.linalg.solve(I-gamma*P, R)
    return V
```

Fig. 2: The code of evaluate_policy function

The average of the value function from Q-learning algorithm are listed following. As the learning rate increases from 0.001 to 0.02, the value function increases the values gradually. As the learning rate go over 0.02, the value function reduces significantly. Because as the learning rate is larger, amount of update for the action-value function will be huge, and make it jumping in the value space. As a consequence, it cannot gradually go to the optimal value.

- Q Learning Avg. Value (0.001): 213.4
- Q Learning Avg. Value (0.005): 257.7
- Q Learning Avg. Value (0.02): 233.2
- Q Learning Avg. Value (0.1): 207.8
- Q Learning Avg. Value (1.0): -49.5

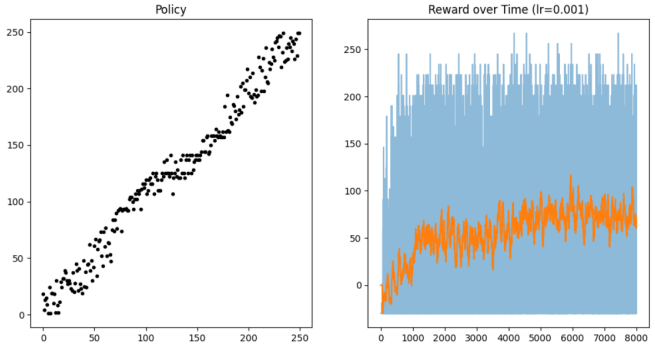


Fig. 3: The Policy and Reward as learning rate = 0.001

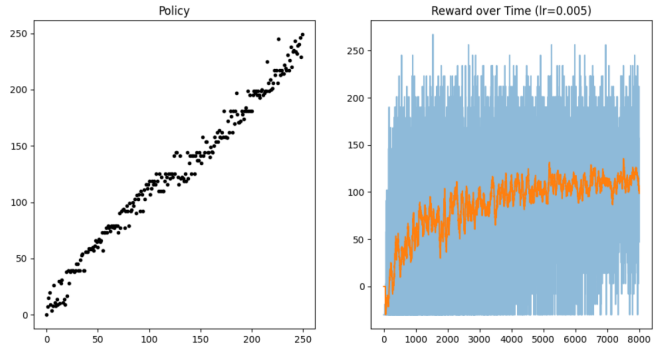


Fig. 4: The Policy and Reward as learning rate = 0.005

From Fig. 4 to Fig. 8 show the policy and the reward over time. The policies are similar with all the learning rate, however, the rewards are so different. The reward reaches maximum as *learning_rate* = 0.02, and reduce the *learning_rate* is too large. At *learning_rate* = 1.0, the rewards values are similar to random action selection.

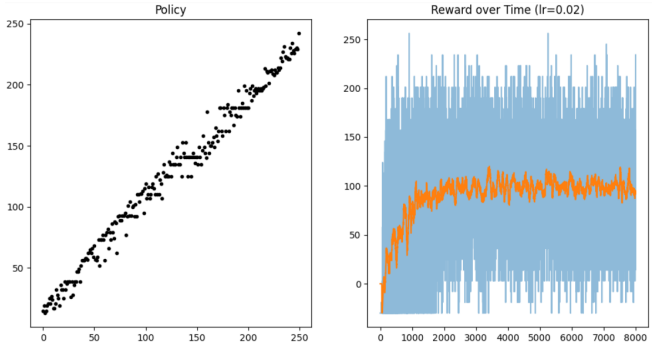


Fig. 5: The Policy and Reward as learning rate = 0.02

Question: The rate of convergence for Q learning is significantly slower than that of Value Iteration. What information does Value Iteration have access to (and indeed makes use of) that makes it converge faster?

Answer: The Value Iteration can access to the world model (via transition probability), that makes its convergence much faster than Q-learning. Q-learning does not know the transition probability function.

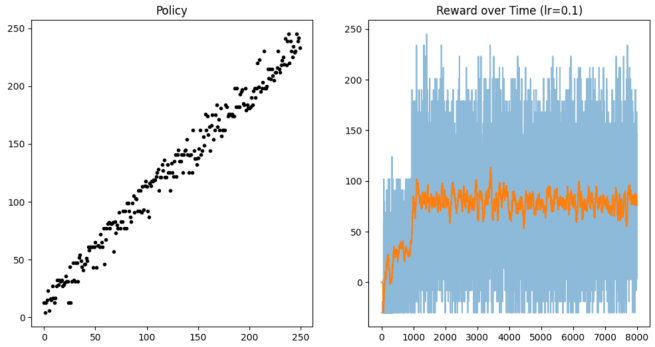


Fig. 6: The Policy and Reward as learning rate = 0.1

Question: When the learning rate is very low, the performance is not particularly good. From looking at the plots of the total reward over time, what is likely the cause? How would you fix this issue (without changing the learning rate)?

Answer: As learning rate is too low, the performance is not good. It is due to the value function and policy will be updated too slow and changed very little. It will require a lot iteration to make the policy convergence (This is the way to fix the small learning rate problem)

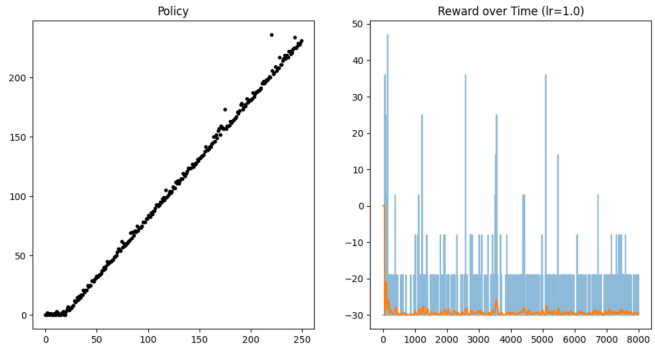


Fig. 7: The Policy and Reward as learning rate = 1.0

Question: When the learning is too high, the performance is also not very good. Why does this happen?

Answer: As the learning rate is too high, the performance is also bad. It is because the learning rate is high, the a mount of update for value function is huge and the function will be jumping in the value space and cannot converge to the optimal value.

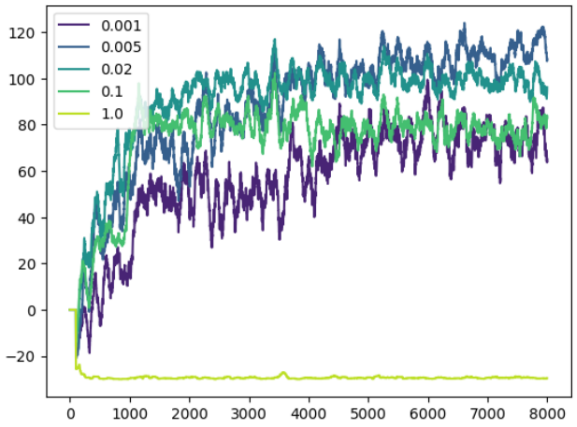


Fig. 8: Comparing among the learning rates

II. Deep Reinforcement Learning

Question: How does the algorithm perform? How does the final average return compare to the maximum possible value?

Answer: The algorithm performed good, and can reach the maximum returns after 500 episodes. However, I believe there are other algorithms which perform much faster than this.

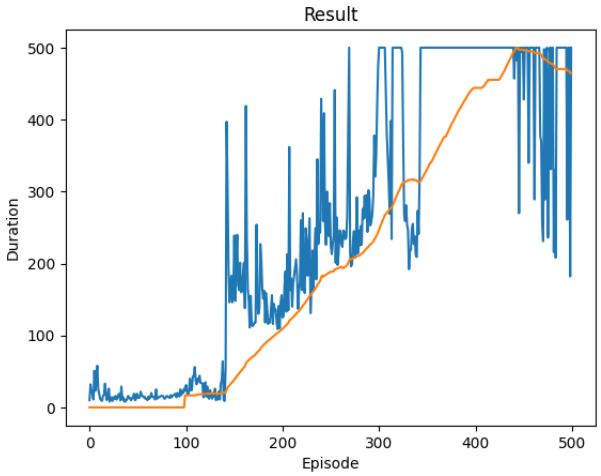


Fig. 9: Duration vs. Episode - learning-rate = 1e-4

The parameter set for this results:

- BATCH SIZE = 128
- GAMMA = 0.99
- EPS START = 0.9
- EPS END = 0.05
- EPS DECAY = 1000
- TAU = 0.005
- LR = 1e-4

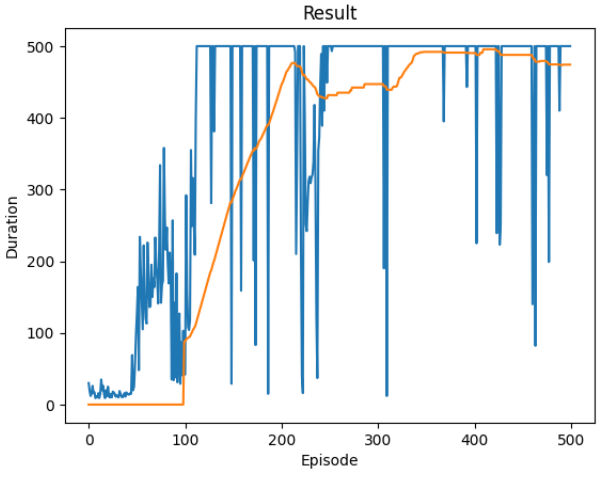


Fig. 10: Duration vs. Episode - learning rate = 5e-4

We changed the learning rate LR= 0.0005, and the network is trained much faster. We think bigger learning-rate will speed up the training process. At episode 120, the pendulum can be at the top position in 500 steps. The model is learned very fast from episode 120 to 220. From that, the performance of the network is fluctuated, but still will good return.

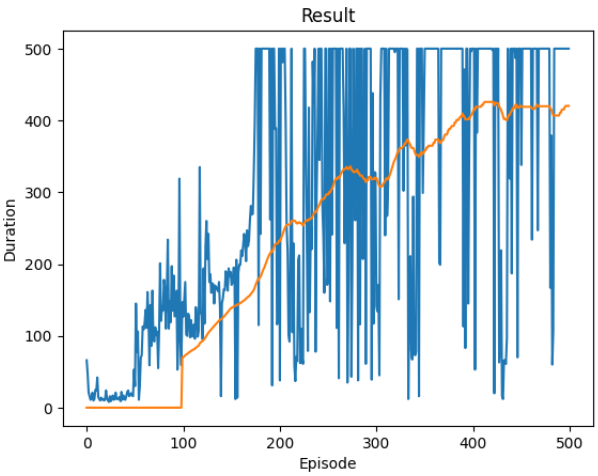


Fig. 11: Duration vs. Episode - learning rate = 2e-3

We keep increasing learning rate LR= 0.002, and to stable the learning process, the batch size is also increased to 256. However, the result is not worse. the learning progress is too slow. It cannot get 500 steps during all 500 training episodes.

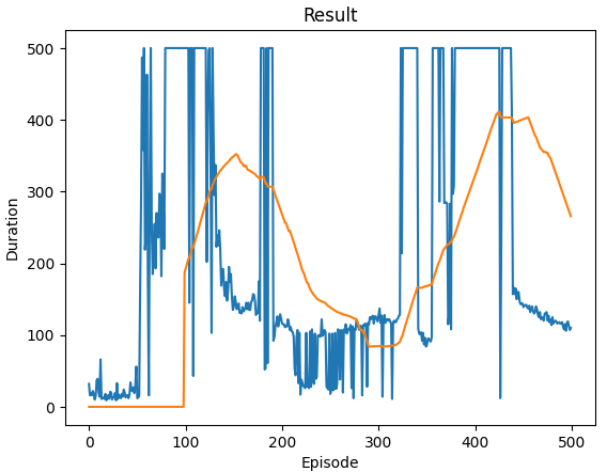


Fig. 12: Duration vs. Episode - learning rate = 1e-3

With learning rate LR= 0.001, the learning process is even more unstable.

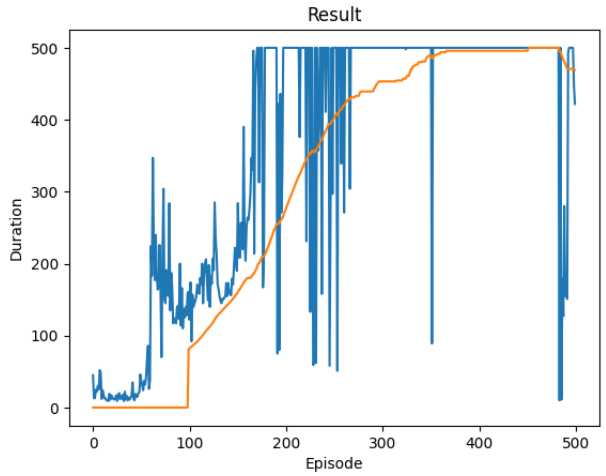


Fig. 13: Duration vs. Episode - learning rate = 5e-4 - batch-size 256

So, we tried with learning rate of 0.0005 and batch size of 256. the learning is very stable without much fluctuation.

Question: (3-6 sentences) Describe how the tuning went?. Include any surprises you encountered along the way

Answer: As adjusting the learning_rate = 1e-4 to 2e-3 and the batch-size of 128 and 256, we can see that the learning rate of 1e-3 is too large, and makes a huge effect on learning progress. Batch-size surely influence the training process, and right learning-rate can speed up the training. Interestingly, the batch-size of 256 combining with the learning rate of 2e-3 stabilizes the training process.

III. Connect Four and MCTS

A. Minimax

Question: What is the evaluation function being used to evaluate the goodness of a board state once the maximum depth is reached? How ”useful” is the value function I have provided?

Answer: the evaluation function being used in *minimax* algorithm is simple. If the board reaches a terminal state, it returns 1 for the winner and -1 for the loser. If not in a terminal, return 0. For simple games with small number of game steps, it could work. However, for complicated games with long step-horizon, it is not useful because it does not provide any hint to evaluate which state is better.

After playing 25 games:

- The Depth 5 wins 17 times
- The Dept 3 wins 6 times
- and Draw 2 times

Question: (1-3 sentences) You may notice that sometimes the depth-3 minimax search wins against the depth-5 minimax search. How is this possible?

Answer: Sometimes the depth-3 minimax wins over the depth-5 minimax because in this game, some actions are more crucial than others, especially the action at the middle of the board. At the first few iterations, randomly the depth-3 minimax takes actions at the middle of the board, its percentage to win increase significantly, and depth-5 minimax is no way to fight back.

B. Monte-Carlo Tree Search

The four functions *monte_carlo_tree_search*, *best_child*, *best_uct*, and *backpropagate* are shown in Fig. 14, 15, 16, and 17, respectively.

```
def monte_carlo_tree_search(start_state, num_iterations=1000):
    """MCTS core loop"""
    # Start by creating the root of the tree.
    root = Tree(start_state=start_state)
    for _ in range(num_iterations):
        leaf = traverse(root)
        sim_result = rollout(leaf, start_state)
        backpropagate(leaf, sim_result)
    return best_child(root)
```

Fig. 14: MCTS code

```
def best_child(node):
    """When done sampling, pick the child visited the most."""
    children = node.children
    m_max = 0
    for child in children:
        if child.n >= m_max:
            child_select = child
            m_max = child.n
    select_move = 0
    moves = node.state.get_moves()
    for move in moves:
        if (node.state.copy().play_move(move).board ==
                        child_select.state.board).all():
            select_move = move
    return select_move
```

Fig. 15: Best Child function code

```
def best_uct(node, C=5):
    """Pick the best action according to the UCB/UCT algorithm"""
    bandit_Q = []
    for jj, child in enumerate(node.children):
        Q_mean = np.average(child.values)
        ucb = C*np.sqrt(np.log(node.n)/child.n)
        val = Q_mean + ucb
        bandit_Q.append(val)
    bandit_ind = np.argmax(bandit_Q)
    return list(node.children)[bandit_ind]
```

Fig. 16: Best UCT function code

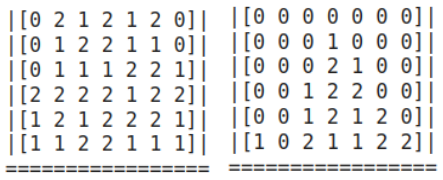
```
def backpropagate(node, simulation_result):
    """Update the node and its parent (via recursion)."""
    if node is None:
        return
    node.values.append(simulation_result)
    node.n += 1
    backpropagate(node.parent, simulation_result)
```

Fig. 17: Propagate function code

Question: For the given configuration (1000 iterations, C=5), which algorithm wins more often?

Answer: For the given configuration, MCTS algorithm dominates the Minimax depth-5 with results as following:

- Total Plays: 25
- MiniMax Wins: 6
- MCTS Wins: 15
- Draws: 4
- The running time of MCTS is from 4 to 8.5 seconds, and Minimax is from 2.5 to 6.5 seconds.

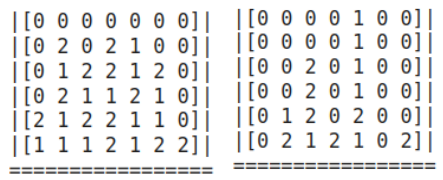


(a) (b)

Fig. 18: Two games with c=5

Question: Rerun the experiments with C=0.1 and C=25. Include the win rates; how well does MCTS perform when you change C?
Answer: With C = 0.1, the Minimax algorithm took advantages of MCTS algorithm. The win rates are shown as following:

- Total Plays: 25
- MiniMax Wins: 14
- MCTS Wins: 11
- Draws: 0
- The running time of MCTS is around 3.1-10.2 seconds, and Minimax is inside 2.6 -6.4 second.

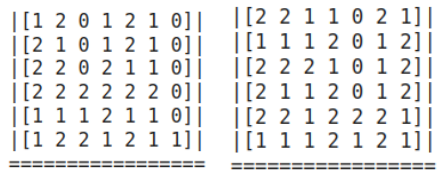


(a) (b)

Fig. 19: Two games with c=0.1 (a) Minimax win & (b) MCTS win

With C = 25, there is higher chance that MCTS over Minimax. The detail is shown as following:

- Total Plays: 25
- MiniMax Wins: 5
- MCTS Wins: 17
- Draws: 3
- The running time of MCTS is from 2.90 to 8.2 seconds, and Minimax is similar to the previous experiments.



(a) (b)

Fig. 20: Two games with c=25 - MCTS won in both games

Question: Describe how the behavior of the MCTS changes when you change the value of C. Pick a couple final board states. In your answer, you might consider discussing the types of ways MCTS wins/loses for different values of C. Be sure to label which value of C was used for each final board state you include in your writeup.
Answer: As we increase the value of C to a certain value, MCTS will work better than Minimax algorithm. In my opinion, as the C value increases, MCTS will spend more time to explore and find a better solution. , so it can find better optimal solution, and will dominate the Minimax.

We have tried looking the games where MTCS losing to Minimax, however, we cannot find any hint specifically pointing out how value of C affect to the losing of MCTS.

It is great if you can put the answer in the comment.