Project 5 - Deep RL, Bandits, and MCTS

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

I. Deep Reinforcement Learning

Question: For this scenario, what is the maximum total reward for the Cartpole problem?

Answer: For this environment and its setup, there are 200 time steps in one iterations, and each step is rewarded +1 if the pole is stays up. Therefore, the maximum total reward is 200.

Before the machine learning algorithm is trained, the function compute_avg_return returns a value of the performance of a random policy, whose value is 22.6 (around 23 steps before falling down).

The result of the training algorithm is shown in Fig. 1. After 4000 iterations, the average returns reached maximum values: 200.

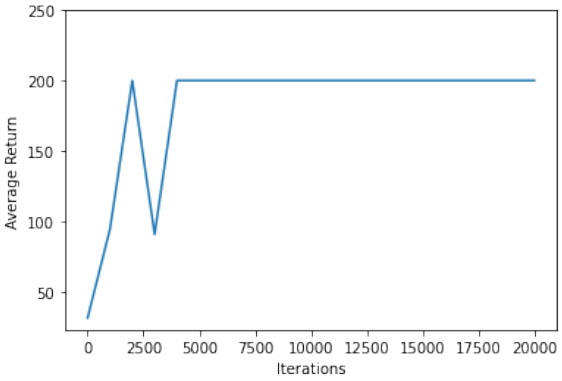


Fig. 1: Average Return vs. Iterations with learning_rate=1e-e

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

Answer: The algorithm performed good, and reaching the maximum returns after around 4000 iterations. However, I believe there are other algorithms which perform much faster than this. And the loss function values is weird because it increases gradually by iteration.

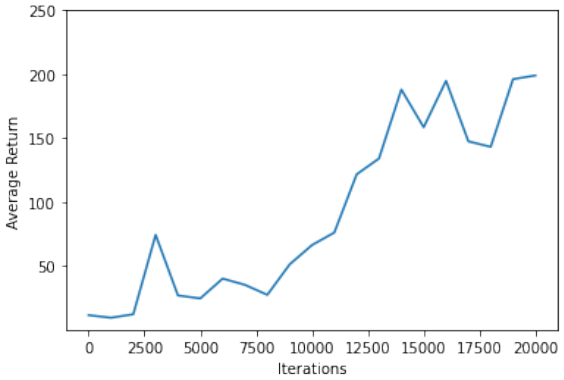


Fig. 2: Average Return vs. Iterations with learning_rate=1e-4

Question: (3-6 sentences) Describe how the learning rate changes performance. Does the final average return change? Does the rate of improvement change?
Answer: As adjusting the learning_rate = 1e-4, the average returns converge slower to 200 (Fig. 2). Until the iteration of 20000, the average returns reached 200 values only one time. However, the learning process is much more stable than the previous learning_rate. The loss value reached maximum value at 241.88, and in most of the iteration, the values is in range of $[10.0 : 30.0]$

II. Bandit Algorithms

A. *Epsilon Greedy Bandits*

In this algorithm, the selection is setup as following:

- If a random_number $>$ epsilon:
  - Pick the bandit which provide the best average return
- *Else:*
  - select a random bandit.

The result of *epsilon_greedy_algorithm* is shown in Fig. 3. With the epsilon = 0.05 (5% of bandit selection is randomly), the average rewards will reach the maximum of ≈ 3.0.

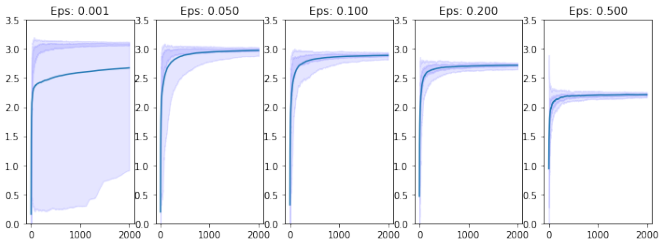


Fig. 3: Average rewards of epsilon_greedy_algorithm

The (average) percentage of optimal pulls for each value of epsilon is shown in Table I.

Epsilon	0.001	0.05	0.1	0.2	0.5
Optimal Pulls	78.59%	84.6%	75.75%	64.0%	45.12%

TABLE I: The epsilon vs Optimal Pulls

Question: For which epsilon is the peak "optimal pull percentage"? Why does the percentage of optimal pulls (and, relatedly the average reward) decrease for the higher values of epsilon?
Answer: At epsilon=0.05 (5 percent) of random selection of bandit, the optimal pulls reached optimal of 84.6%. Then the optimal pulls reduce as we increase the random bandit selection. The reduction can be explained as following. After taking random bandit, we have gained knowledge of the environment. The knowledge is not complete, however, it provides some hints about the environment features, which we can exploit of that. As the number of iteration goes up, the knowledge is more complete, and make us to decide better. If we still use a large mount of selection by random, it means we don't use the gained knowledge, and of course, will make the optimal pulls reduce.

B. UCB Bandits

The added code for UCB Bandit function is shown in Fig. 4. In the code, we set up an array of UCB values, which are determined by the average values and their upper bounds. The selected bandit is the one with the highest value.

```
# Pick one of the bandits by ucb
ucb = np.zeros(len(bandits))
for jj in range(len(bandits)):
  if num_pulls_per_bandit[jj] ==0:
     n_visit = 1
  else:
      n_visit = num_pulls_per_bandit[jj]

  ucb[jj] = tot_reward_per_bandit[jj]/n_visit + \
                  np.sqrt(c*np.log(ii+1)/n_visit)
bandit_ind = np.argmax(ucb)
```

Fig. 4: Selection of bandit by UCB

The average rewards with varied c values are shown in Fig. 5. As c value is in range (0.1 - 1.0), the average reward get the maximum, which is around 3.05.

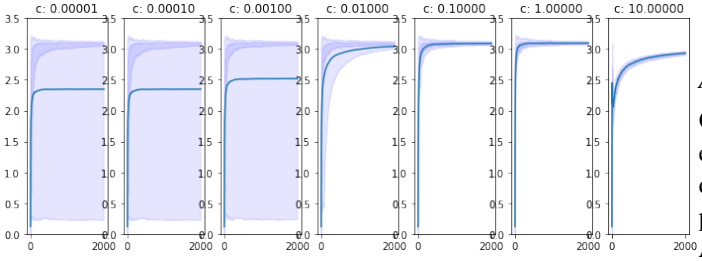


Fig. 5: Optimal Pulls with c values

The (average) percentage of optimal pulls for each value of C (the exploration parameter) is shown in Table II. The

C	1e-5	1e-4	1e-3	0.01	0.1	1.0	10.0
Opt. Pulls	73.86	73.86	79.82	98.05	99.73	99.87	57.15

TABLE II: The c values vs Optimal Pulls (%)

value of C from 0.1 to 1.0 provides the highest optimal Pulls (99.73% to 99.87 %).
Question: For which C is the peak "optimal pull percentage"? How does the percentage of optimal pulls for this value of C compare to the best epsilon from the epsilon-greedy bandit? Why?
Answer: At $C = 1.0$, the *optimal pull percentage* is the peak (99.87%). This value is much better than the epsilon-greedy bandit algorithm (with the peak of 84.6%). I think UCB algorithm can generate more knowledge about environment than epsilon-greedy algorithm in each iteration. It provides not only the average value, it also give us the boundary of that values, and then shrink it down by each iteration. This boundary make the convergence of optimal value faster.

Question: Why is the average performance of the algorithm poor for very low values of C? What will happen to the performance as the number of trials approach infinity?
Answer: The Upper Confidence Bound is :

$$a_t = argmax_{a \in A}\{Q(a) + \sqrt{C\frac{log(\sum n_{j,t})}{N_t(a)}}\} \quad (1)$$

As the C value is very low, it reduces the bound value, and the term inside the bracket will reduce to the average value of reward (similar to greedy algorithm). It means there is no exploration, only exploitation. As we increase the number of trials to infinity (N_t), the bound value will be smaller, and the exploitation will dominate the selection.

Question: Why is the average performance of the algorithm poor for very high values of C? What will happen to the performance as the number of trials approach infinity?
Answer: As the value of C is so large, the performance of the algorithm is also poor. Because large C in Eq. 1 will make the bound larger, and it performs more exploration. As a consequence, the algorithm converging very slow. As the number of trials goes to infinity (N_t), the bound in Eq. 1 will be smaller, and we will do more exploitation, then the algorithm will perform better.

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 greedy algorithm - taking the action which provide the highest reward value. If there are several actions with the same value, a random selection will be used. As the depth is larger, algorithm can get more knowledge about the state, and

the evaluate function make use of the knowledge about the reward's information and provide a better action selection.

After playing 25 games:

- The Depth 5 wins 15 times
- The Dept 3 wins 8 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 is more important than others especially the action at the middle of the board. At the first few iterations, if 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 win back.

B. Monte-Carlo Tree Search

The four functions *monte_carlo_tree_search*, *best_child*, *best_uct*, and *backpropagate* are shown in Fig. 6, 7, 8, and 9, respectively.

```
root = Tree(start_state=start_state)
if len(root.values) == 0:
    for jj in range(len(root.state.get_moves())):
        root.values.append(0.0)
# Loop through MCTS iterations.
for _ in range(num_iterations):
    # One step of MCTS iteration
    leaf = traverse(root)
    if len(leaf.values) == 0:
        for jj in range(len(leaf.state.get_moves())):
            leaf.values.append(0.0)
    simulation_result = rollout(leaf, root.state)
    backpropagate(leaf, simulation_result)
```

Fig. 6: MCTS code

```
best_action = None
max_visit = 0
for child in node.children:
    if child.n > max_visit:
        max_visit = child.n
        best_action = child.move
return best_action
```

Fig. 7: Best Child function code

```
ucb = np.zeros(len(node.values))
moves = node.state.get_moves()
for jj in range(len(node.values)):
    for child in node.children:
        if child.move == moves[jj]:
            n_visit = child.n
    ucb[jj] = node.values[jj]/n_visit + \
                    np.sqrt(C*np.log(node.n+1)/n_visit)
action = moves[np.argmax(ucb)]
for child in node.children:
    if action == child.move:
        return child
```

Fig. 8: Best UCT function code

```
node.n += 1
if node.parent is not None:
  actions = node.parent.state.get_moves()
  for ii in range(len(actions)):
    if actions[ii] == node.move:
      node.parent.values[ii] += simulation_result
backpropagate(node.parent, simulation_result)
```

Fig. 9: 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: 7
- MCTS Wins: 16
- Draws: 2
- The running time of MCTS is around 8.5 seconds, and Minimax is from 9.5 to 14.5 seconds.

```
|[0 2 1 2 1 2 0]|  |[0 0 0 0 0 0 0]|
|[0 1 2 2 1 1 0]|  |[0 0 0 1 0 0 0]|
|[0 1 1 1 2 2 1]|  |[0 0 0 2 1 0 0]|
|[2 2 2 2 1 2 2]|  |[0 0 1 2 2 0 0]|
|[1 2 1 2 2 2 1]|  |[0 0 1 2 1 2 0]|
|[1 1 2 2 1 1 1]|  |[1 0 2 1 1 2 2]|
=================  =================
```

(a) (b)

Fig. 10: 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: 17
- MCTS Wins: 6
- Draws: 2
- The running time of MCTS is around 4.0-8.2 seconds, and Minimax is still the same.

```
|[0 0 0 0 0 0 0]|  |[0 0 0 0 1 0 0]|
|[0 2 0 2 1 0 0]|  |[0 0 0 0 1 0 0]|
|[0 1 2 2 1 2 0]|  |[0 0 2 0 1 0 0]|
|[0 2 1 1 2 1 0]|  |[0 0 2 0 1 0 0]|
|[2 1 2 2 1 1 0]|  |[0 1 2 0 2 0 0]|
|[1 1 1 2 1 2 2]|  |[0 2 1 2 1 0 2]|
=================  =================
```

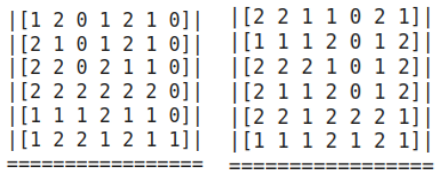
(a) (b)

Fig. 11: Two games with c=0.1 (a) Minimax win & (b) MCTS win

With C = 25, the MCTS algorithm win the Minimax one most of the cases. The detail is shown as following:

- Total Plays: 25
- MiniMax Wins: 4
- MCTS Wins: 18

- Draws: 3
- The running time of MCTS is from 6.0 to 9.0 seconds, and Minimax is similar to the previous experiments.



(a) (b)

Fig. 12: 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, MCTS will work better than Minimax algorithm. In my opinion, as the C value increases, MCTS will spend more time for exploration, so it can find better optimal solution, and will dominate the Minimax.