Multi-Agent Systems

Homework Assignment 6

Martynas Vaznonis (2701013)
m.vaznonis@student.vu.nl

1 Monte Carlo Tree Search (MCTS)

Questions:

1. I began the implementation by creating a `Node` class which is initialized recursively with a depth parameter ($d = 20$ in this case). Each of the two child nodes are assigned a name that is the parent's name appended with L or R. Leaf nodes are flagged as such. Every node is also given a pointer to the parent to make value propagation easier. A target is selected recursively by randomly choosing between children. All values can be immediately assigned with another recursive method, but this is expensive as most leaf nodes are never reached. This same method can be used during the search to simply initialize only a particular leaf node.

   On top of this I implemented the `MonteCarloTreeSearch` class. It is initialized with an instance of a tree, an exploration hyperparameter c, and a flag informing the search if the leaf nodes have already been initialized with values. The search begins by calling the `search` method. It takes as an argument n, the maximum limit a certain node is used as root. In other words, once the current root has had n rollouts backpropagated to it, its better child is selected as the next current root. Since this child will also have had some rollouts backpropagated to it, the number of iterations it will spend as the root node is less than n.

   The `search` method loops over the `iteration` method which selects the "snowcap" node and decides when to use the random rollout policy. The value received from the rollout (or leaf node) is then propagated back up the tree. Only one rollout per "snowcap" node is necessary before said node can be used in the search, "extending the snowcap."

   All of the experiments can then be run by initializing the MCTS class with the different values of c and letting it search with some computational budget. The exact implementation and the undiscussed minutia can be seen in `MonteCarloTreeSearch.ipynb`.

2. To rigorously analyze the effects of the hyperparameter c, I ran 100 searches per value of c with $c \in \{0.01, 0.1, 0.5, 1, 1.5, 2, 3, 5, 10, 100\}$ to cover a large range of values. The searches are best summarized in figure 1, which shows how the estimated value at the root node evolved on average for each of the values of c. With lower values of c, the estimate initially grows fast because the search exploits almost immediately. But such exploitation has a couple of negatives. The first is that the higher values found and exploited might only have been estimated as such because of variance, not because they are intrinsically closer to the target. This is reflected in figure 1 by the sharp decline inversely commensurate to the value of c which follows immediately after the quick rise.

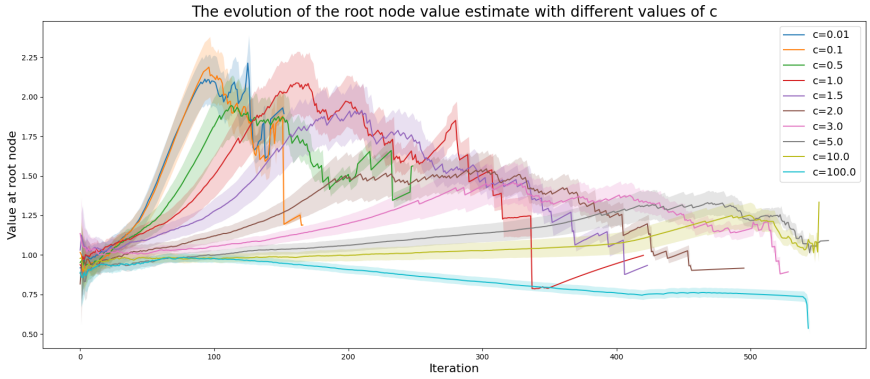


Figure 1: The value at the root node plotted for a range of values of exploration hyperparameter c averaged over 100 runs. The smaller the c, the shorter the search tends to be. Furthermore, smaller values of c lead to high initial root node values but also sharp drops as the search ends up in local minima. When c is really large (i.e., $c = 100$) no exploitation occurs at all and the value drops almost monotonically. The shaded area represents the standard deviation divided by 5.

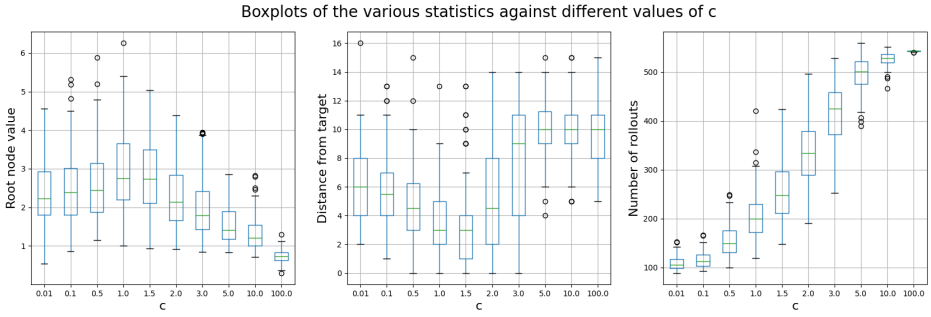
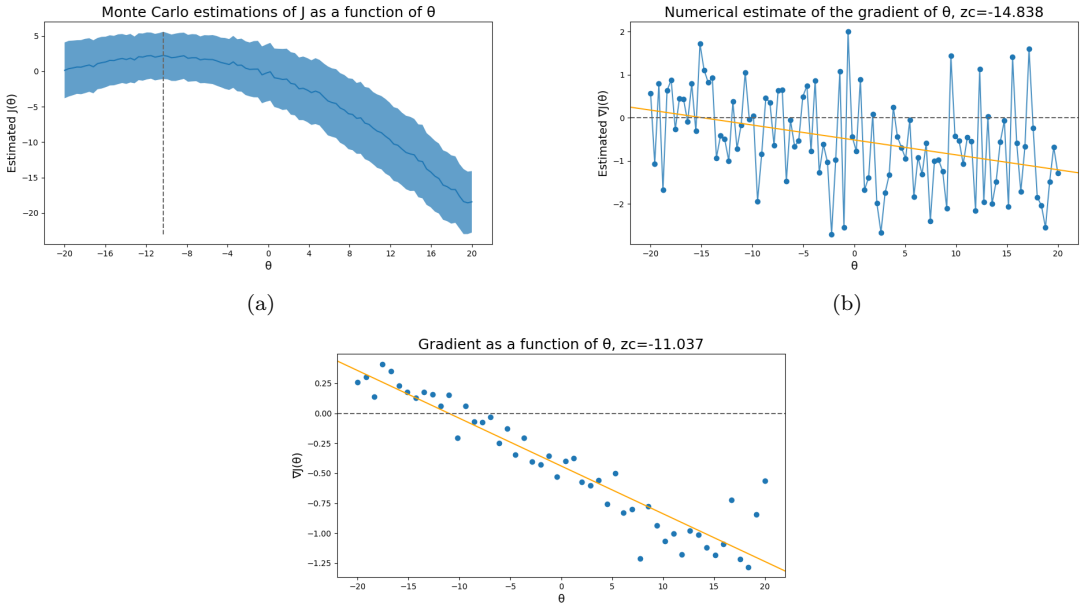


Figure 2: Summary statistics for the searches with differing values of c. The value of the root node at the end of the search (left). The distance from the target of the best identified leaf node (center). The number of rollouts per search (right). The number of rollouts differs because exploration propagates rollout values back to a greater diversity of nodes. That means that when a new root node is chosen, it will have observed fewer rollouts and thus remain the current root for longer.

The second issue arises because of the way MCTS is implemented here. When switching to a new current root, the number of rollouts previously recorded is maintained. That means that the more exploitative the search is, the quicker it will terminate. This is not inherently bad as searching is expensive and unpromising branches can remain unvisited. But in cases where c is too small, the search explores too little to find the target. This effect is especially clear in figure 2, on the right. A very strong correlation is seen between the value of c and the length of the search.

The optimal values of c I found for this problem with this implementation are $c = 1$ and $c = 1.5$. Neither distance from target nor root value are significantly different in these two cases (fig. 2, center and left). Moreover, they are significantly better than all other values of c in both distance and value. For the statistical tests see `MCTSfigs.ipynb`. This data suggests that a good value of c will balance the exploration and exploitation to maximize the odds of a good outcome.

Importantly, The smaller than optimal values of c outperform the larger than optimal ones. In other words, it seems that when a suboptimal c is selected, exploitation leads to better results. Thus, when uncertain about what value to choose, erring on the side of exploitation and picking a



(a) (b) (c)

Figure 3: Various strategies of finding θ. Estimating the value of a policy by averaging MC rollouts. 100 policies were evaluated with 5000 rollouts. The shaded area is standard deviation divided by 3 (a). Estimating the optimal value of θ numerically. For each value of θ, 1000 runs were averaged to reduce uncertainty (b). Calculating the gradient analytically gives a much better estimate of θ than numerical estimation. If the outliers are removed, the zero-crossing corresponds closely with the value found in question 1. (c). The zero-crossing in (b) and (c) was found with ordinary least squares regression.

smaller c will accord better results. This can be seen in figure 2. On the left, the root node values drop much less moving towards smaller cs than when moving towards larger ones. Similarly in the center plot, the distance from the target is much greater when exploration is preferred over exploitation. In fact, for $c \geq 5$ the distance from the target on average is as bad as random guessing.

2 Reinforcement Learning: Policy Gradient and the Actor-Critic algorithm

The implementation for this section consists of several parts. First, I implemented an `Environment` class that initializes states, provides state transitions given actions, and outputs rewards. All experiments in this assignment use a state space with $L = 20$. Then, I created a `Policy` class which has two methods, `sample_action` and `get_probs`. The first uses the latter to draw an action. The second is also useful in calculating the derivative. Finally, I created a plethora of utility functions, including `simulate_episode`, for creating MC rollouts, `compute_grad`, two functions for both regular and bootstrap policy gradient, `MC_prediction`, for evaluating a policy with MC rollouts, and some others for running experiments and making plots. The detailed implementation can be found at `PolicyGradient.ipynb`.

Questions:

1. Figure 3a shows the gradient as estimated with Monte Carlo rollouts under a hundred policies evenly spaced across the state space values (between -20 and 20). For each policy, 5000 rollouts

are used. The best θ found here is $\theta \approx -10.3$.

2. Figure 3b shows the estimation. The value of h was set to 0.5 as smaller values introduced too much variance. The exact same θ values were used as in the previous subquestion. Also the estimates were averaged over 1000 runs to further decrease the total variance seen in the plot. The line was fitted using linear regression with the zero-crossing in this case at around -14.266. The discrepancy with the previous answer likely originates from the very large value of h selected. Indeed on other runs I have observed varied values at the zero-crossing, such as -9.973, which is much closer to the value estimated in the previous subquestion.

3. To find $\nabla_\theta J$, $\nabla_\theta \log \pi_\theta(a_t|s_t)$ must first be calculated. Since no automatic differentiation library is used for this assignment, this step must be completed by hand. The derivation can be seen below.

$$\nabla_\theta \log \pi_\theta(a_t|s_t) = \frac{\nabla_\theta \pi_\theta(a_t|s_t)}{\pi_\theta(a_t|s_t)} \tag{1}$$

$$= \frac{\nabla_\theta \pi_\theta(a_t = R|s_t)}{\pi_\theta(a_t = R|s_t)} = \frac{-\frac{\nabla_\theta e^{-\beta(s-\theta)}}{(1+e^{-\beta(s-\theta)})^2}}{\pi_\theta(a_t = R|s_t)} = \frac{-\beta \frac{e^{-\beta(s-\theta)}}{(1+e^{-\beta(s-\theta)})^2}}{\pi_\theta(a_t = R|s_t)}$$

$$= \frac{-\beta \pi_\theta(a_t = R|s_t)(1 - \pi_\theta(a_t = R|s_t))}{\pi_\theta(a_t = R|s_t)} = \beta \pi_\theta(a_t = R|s_t) - \beta \tag{2}$$

$$= \frac{\nabla_\theta \pi_\theta(a_t = L|s_t)}{\pi_\theta(a_t = L|s_t)} = \frac{\nabla_\theta (1 - \pi_\theta(a_t = R|s_t))}{\pi_\theta(a_t = L|s_t)}$$

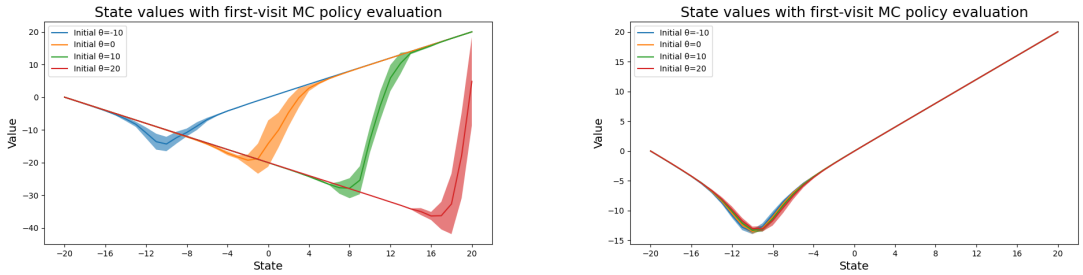
$$= \frac{\beta \pi_\theta(a_t = R|s_t)(1 - \pi_\theta(a_t = R|s_t))}{(1 - \pi_\theta(a_t = R|s_t))} = \beta \pi_\theta(a_t = R|s_t) \tag{3}$$

Equation 1 refers to the general derivative of the log probability for any action. Expression 2 is the solution for the specific action R. And similarly, expression 3 is the solution for action L.

Figure 3c shows the gradient values for each θ averaged over 1000 runs. Only 50 values of θ were used here, similarly evenly spaced across the state space. Again, linear regression was used to fit the line, and a zero-crossing is observed at around -11.037. This answer is more in line with the one in figure 3a because the analytical derivative is more accurate than the numerical estimate. Additionally, a few outliers can be seen on the right, without which the zero-crossing is even closer to the value found in the first subquestion at approximately -10.6.

4. Figure 4a shows the values of every state estimated for four policies. The different policies have $\theta_0 \in \{-10, 0, 10, 20\}$ and the value function was estimated using first-visit Monte Carlo prediction with the following update rule, $v(s) \leftarrow v(s) + \frac{1}{C(s)}[R - v(s)]$. Here, $C(s)$ maintains a count of how many updates have already occurred in state s and R is the cumulative reward received after state s in the episode. In short, $v(s)$ is estimated as the average of R over the MC rollouts. The number of rollouts was set to 100. Also, 10 repetitions for each value of θ_0 were run and averaged to ensure that the evaluations were stable.

5. After the 100 rollouts and an adequately estimated value function, the full Advantage Actor-Critic (A2C) algorithm was implemented. Figure 4b shows that regardless of the initial value of θ, the actor converges to the same final value, leading the critic to similarly converge on the same state value function. Again, the results are averaged over 10 runs, and the negligible standard deviation shows how stable the convergence is.



(a) (b)

Figure 4: State values pre- (a) and post- (b) convergence. The state values were estimated with first-visit MC policy evaluation. In (a) this algorithm used an iterative average update rule. In (b) it used a learning rate to reflect the nonstationary nature of on-policy improvement. Regardless of the starting θ, the A2C algorithm is robust enough to converge to the same optimal value. The data is averaged over 10 runs and the shaded area is the standard deviation.

One important change was made in the critic however. Namely, the update rule was changed to $v(s) \leftarrow v(s) + \alpha[R - v(s)]$, where α is the learning rate (set to 0.01). The reason for this is that the previous implementation never forgets. This is a good thing for simple policy evaluation as it is more data efficient. But it is a negative when the policy changes because it remembers data that is no longer applicable, introducing bias. A static learning rate circumvents this issue.

6. For the four policies outlined in subquestion 4., the A2C algorithm was applied. The convergence results can be seen in figure 5. All policies eventually reached the optimal θ value. The results in the figure are averaged over 10 learning trials for each starting θ. In summary, figures 5 and 4b show that convergence is achieved both by the actor and the critic. The final value of $\theta \approx -9.88$, and the value in question 1. was $\theta \approx -10.3$. An exact equivalence cannot be expected because of variance both in the estimate in question 1. and the variance from a stochastic optimization procedure. With that in mind, the two answers are likely statistically identical.

7. On-policy algorithms improve from the data generated with the target policy. That is, the behavior policy is the same as the target policy. This makes it somewhat data inefficient as it cannot reuse all the data it has generated every time that the policy changes. That is, any experience generated before a single update, may no longer be valid to update it further because it was meant for a different target policy. This prohibits it from using techniques like experience replay to speed up learning. It is also why I had to change the update rule in the MC prediction algorithm from $v(s) \leftarrow v(s) + \frac{1}{C(s)}[R - v(s)]$ to $v(s) \leftarrow v(s) + \alpha[R - v(s)]$.

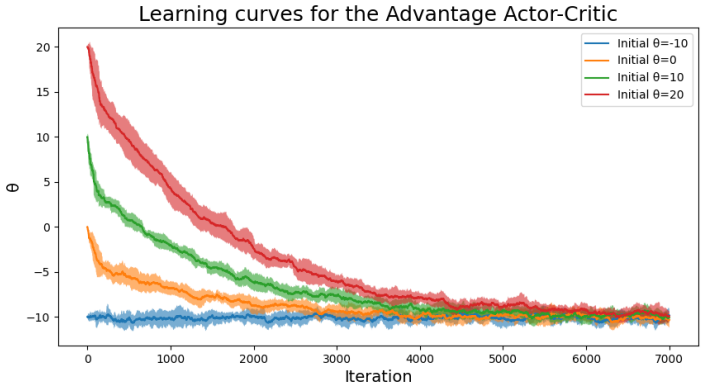


Figure 5: The learning curves for the policies with the different starting values of θ. All policies converge relatively quickly. The data is averaged over 10 runs and the shaded area is the standard deviation.