towards data science

Follow 563K Followers

Learning To Win Blackjack With Monte Carlo Methods

Donal Byrne Nov 7, 2018 · 9 min read

Image can be found here

What will I learn?

This article will take you through the logic behind one of the foundational pillars of reinforcement learning, Monte Carlo (MC) methods. This classic approach to the problem of reinforcement learning will be demonstrated by finding the optimal policy to a simplified version of blackjack.

By the end of this article I hope that you will be able to describe and implement the following topics. The full code can be found on my GitHub.

- Monte Carlo Prediction

- Monte Carlo Control
- Firs Visit vs Every Visit
- Q Values
- Discounted Rewards

What are Monte Carlo methods?

The city of Monte Carlo, Source

MC is a very simple example of model free learning that only requires past experience to learn. It does this by calculating the average reward of taking a specific action A while in a specific state S over many games. If you are not familiar with the basics of reinforcement learning I would encourage you to quickly read up on the basics such as the agent life cycle. My previous article goes through these concepts and can be found here . Also if you are unfamiliar with the game of blackjack checkout this video .

Using Monte Carlo In blackjack

If you have ever played blackjack seriously (or in my case seen the movie 21) then you will probably have heard of “basic strategy”. This is simply a table containing each possible combination of states in blackjack (the sum of your cards and the value of the card being shown by the dealer) along with the best action to take (hit, stick, double or split) according to probability and statistics. This is an

example of a policy.

	2	3	4	5	6	7	8	9	10	A
17+	ST	ST	ST	ST	ST	ST	ST	ST	ST	ST
16	ST	ST	ST	ST	ST	H	H	H	H	H
15	ST	ST	ST	ST	ST	H	H	H	H	H
14	ST	ST	ST	ST	ST	H	H	H	H	H
13	ST	ST	ST	ST	ST	H	H	H	H	H
12	H	H	ST	ST	ST	H	H	H	H	H
11	D	D	D	D	D	D	D	D	D	H
10	D	D	D	D	D	D	D	D	H	H
9	H	D	D	D	H	H	H	H	H	H
5-8	H	H	H	H	H	H	H	H	H	H

Simple version of the basic strategy policy developed by Edward O. Thorp, image taken from here

In our example game we will make it a bit simpler and only have the option to hit or stick. As well as this, we will divide our state logic into two types, a hand with a usable ace and a hand without a usable ace.

In blackjack an ace can either have the value of 1 or 11. If we can have an ace with a value of 11 without going bust we call that a “usable ace”.

Prediction

Source

Lets say that we have been given a very simple strategy (even simpler than the basic strategy above).

Policy: if our hand ≥ 18, stick with a probability of 80% else hit with a probability of 80%

This isn't an amazing policy but it is simple and will still be able to win some games. Now lets say that we want to know the value of holding a hand of 14 while the dealer is showing a 6. This is an example of the prediction problem.

To solve this, we are going to use First Visit Monte Carlo. This method has our agent play through thousands of games using our current policy. Each time the agent carries out action A in state S for the first time in that game it will calculate the reward of the game from that point onwards. By doing this, we can determine how valuable it is to be in our current state.

This is the opposite of Every Visit Monte Carlo, which calculates the reward every time it sees that state/action pair. Both of these methods provide similar results. The steps to implement First Visit Monte Carlo can be seen here.

Algorithm 9: First-Visit MC Prediction (*for action values*)

Input: policy π, positive integer *num_episodes*
Output: value function Q ($\approx q_\pi$ if *num_episodes* is large enough)
Initialize $N(s,a) = 0$ for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
Initialize $returns\_sum(s,a) = 0$ for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
for $i \leftarrow 1$ ***to*** *num_episodes* **do**
  Generate an episode $S_0, A_0, R_1, \ldots, S_T$ using π
  for $t \leftarrow 0$ ***to*** $T-1$ **do**
    if (S_t, A_t) *is a first visit (with return* G_t*)* **then**
      $N(S_t, A_t) \leftarrow N(S_t, A_t) + 1$
      $returns\_sum(S_t, A_t) \leftarrow returns\_sum(S_t, A_t) + G_t$
  end
end
$Q(s,a) \leftarrow returns\_sum(s,a)/N(s,a)$ for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
return Q

First Visit Monte Carlo algorithm for prediction, taken from here

Now if you aren't used to reading these algorithms this can look a bit

over whelming, but it is actually quite simple. Lets go through the steps to implement this algorithm.

1. Provide a policy pi
2. Create empty dictionaries N, returns_sum and Q to hold the amount of times a State/Action pair has been visited, the amount of returns that State/Action pair has received and finally the value of that State/Action pair.
3. Play a game of Blackjack using our current policy for X games
4. Loop through each turn in the game and check if the current State/Action pair has been seen in that game yet
5. If this is the first time we have seen that pair we increase the count for that pair and add the discounted rewards of each step from that turn onwards to our returns_sum dictionary
6. Finally update the Q values with the average of the returns and amount of each State/Action pair.

One last thing that I want to quickly cover before we get into the code is the idea of discounted rewards and Q values.

Discounted Rewards

The idea of discounted rewards is to prioritise immediate reward over potential future rewards. Much like expert chess players, our agent isn't just looking at taking a pawn this turn, they are looking at how to win 12 moves from now. This is why when calculating action values we take the cumulative discounted reward (the sum of all rewards after the action) as opposed to just the immediate reward.

The discount factor is simply a constant number that we multiply our reward by at each time step. After each time step we increase the power to which we multiply our discount factor. This gives more priority to the immediate actions and less priority as we get further away from the action taken.

$$v_\pi(s) = \mathbb{E}_\pi \left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots \mid S_t = s \right]$$

the discounted reward of an action over time, image taken from here

This gives more priority to the immediate actions and less priority as we get further away from the action taken .Choosing the value of our discount factor depends on the task at hand, but it must always be between 0 and 1. The larger the discount factor the higher importance of future rewards and vice versa for a lower discount factor. In general most a discount factor of 0.9 is a good starting point.

Q Values

Q values refer to the value of taking action A while in state S. We store these values in a table or dictionary and update them as we learn. Once we have completed our Q table we will always know what action to take based on the current state we are in.

Implementation

Below is a jupyter notebook with the code to implement MC prediction. Each section is commented and gives more detail about what is going on line by line.

As you can see there is not much to implementing the prediction algorithm and based on the plots shown at the end of the notebook we can see that the algorithm has successfully predicted the values of our very simple blackjack policy. Next up is control.

Control

Source

This is the more interesting of the two problems because now we are going to use MC to learn the optimal strategy of the game as opposed to just validating a previous policy. Once again we are going to use the First Visit approach to MC.

Algorithm 11: First-Visit Constant-α (GLIE) MC Control

Input: positive integer *num_episodes*, small positive fraction α, GLIE $\{\epsilon_i\}$
Output: policy π ($\approx \pi_*$ if *num_episodes* is large enough)
Initialize Q arbitrarily (e.g., $Q(s,a) = 0$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$)
for $i \leftarrow 1$ ***to*** *num_episodes* **do**
  $\epsilon \leftarrow \epsilon_i$
  $\pi \leftarrow \epsilon$-greedy(Q)
  Generate an episode $S_0, A_0, R_1, \ldots, S_T$ using π
  for $t \leftarrow 0$ ***to*** $T-1$ **do**
    if (S_t, A_t) *is a first visit (with return* G_t*)* **then**
      $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(G_t - Q(S_t, A_t))$
  end
end
return π

First Visit MC Control with Constant Alpha, image taken from here

This algorithm looks a bit more complicated than the previous prediction algorithm, but at its core it is still very simple. Because this is a bit more complicated I am going to split the problem up into sections and explain each.

1. Initialise our values and dictionaries

2. Exploration
3. Update policy
4. Generate episodes with new policy
5. Update the Q values

1) Initialising values

This is similar to the last algorithm except this time we only have 1 dictionary to store our Q values. This is because we are using a new update function, we'll talk about that later

2) Update the policy and exploration

People learn by constantly making new mistakes. Our agent learns the same way. In order to learn the best policy we want to have a good mix of carrying out what good moves we have learned and exploring new moves. This is known as the exploration/exploitation problem. In this case we will use the classic epsilon-greedy strategy which works as follows:

1. Set a temporary policy to have equal probability to select either action, [.5,.5]
2. Get the current best policy for the current state Q[s]
3. Get the best action based on the best policy, argmax(Q[s])
4. Set the probability of selecting the best action to 1-epsilon + the temporary policy value (50%)

At the start epsilon will be large meaning that for the most part the best action will have a probability of .5 (random) as the game goes on, epsilon will decrease and the probability of taking the best action will increase

3) Generate episodes with new policy

This is almost the exact same as our previous algorithm, however instead of choosing our actions based on the probabilities of our hardcoded policy we are going to alternate between a random action and our best action. This is the epsilon greedy strategy that we

discussed previously. As we go through we record the state, action and reward of each episode to pass to our update function.

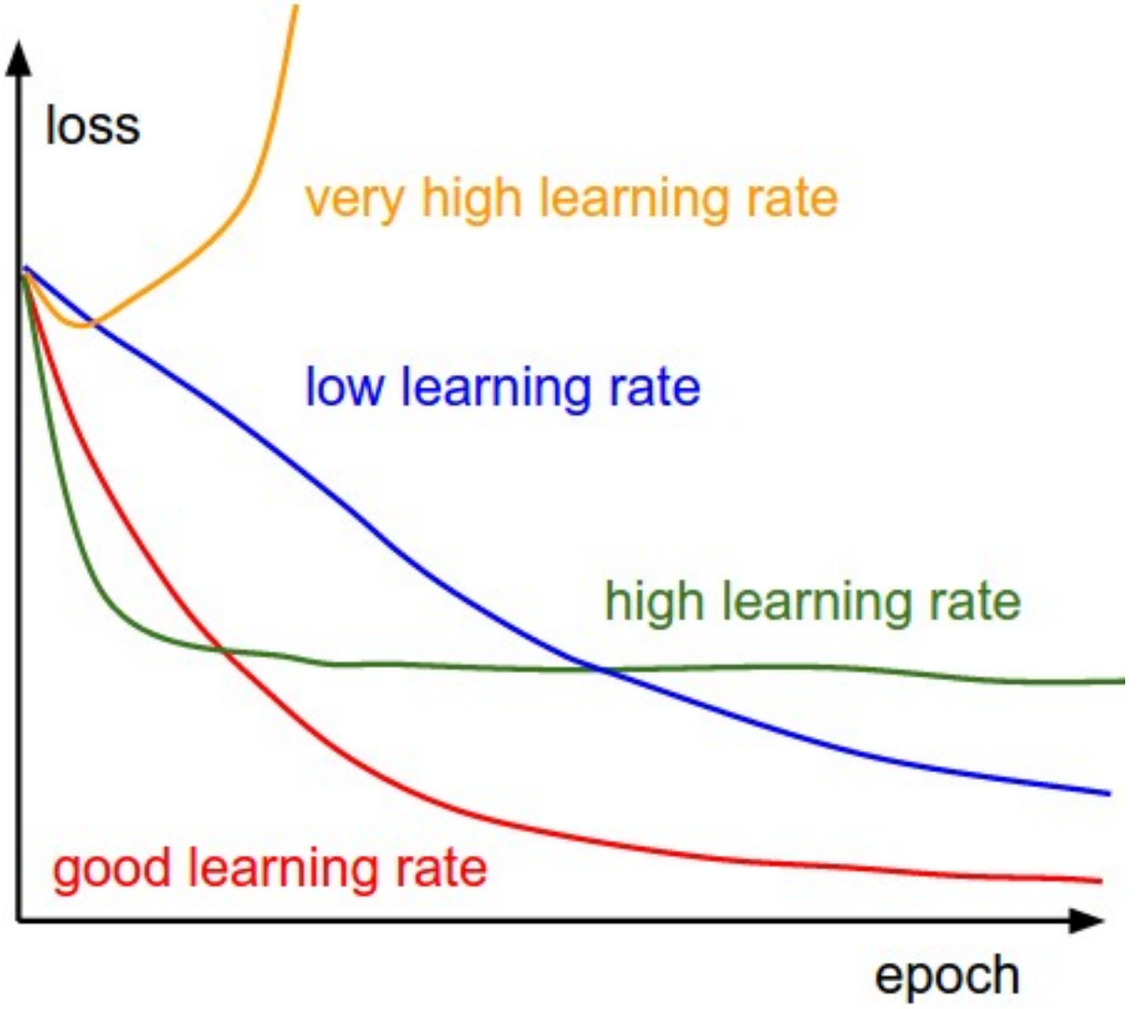
4) Update Q Values

This is the important part of the algorithm. Here we implement the logic for how our agent learns. Instead of simply getting the average returns of a State/Action pair over time, we are going to use an update function that will improve our policy over time. The function looks like this.

```
Q[s][a] = Q[s][a] + alpha*(G - Q[s][a])
```

All we are doing here is taking our original Q value and adding on our update. The update is made up of the cumulative reward of the episode (G) and subtracting the old Q value. This is then all multiplied by alpha.

In this case alpha acts as our learning rate. A large learning rate will mean that we make improvements quickly, but it runs the risk of making changes that are too big.



Although it will initially make progress quickly it may not be able to figure out the more subtle aspects of the task it is learning. On the other hand if the learning rate is too small, the agent will learn the task, but it could take a ridiculously long time. As like most things in machine learning, these are important hyper parameters that you will have to fine tune depending on the needs of your project.

Implementation

Now that we have gone through the theory of our control algorithm, we can get stuck in with code.

Now we have successfully generated our own optimal policy for playing blackjack. You will notice that the plots of the original hard coded policy and our new optimal policy are different and that our new policy reflects Thorps basic strategy.

Conclusion

We now know how to use MC to find an optimal strategy for blackjack. Unfortunately you wont be winning much money with just this strategy any time soon. The real complexity of the game is knowing when and how to bet. An interesting project would be to combine the policy used here with a second policy on how to bet correctly.

I hope you enjoyed the article and found something useful. Any feedback or comments is always appreciated. The full code can be found on my GitHub

References:

Sutton R. and Barto A. — *Reinforcement Learning: An Introduction*, MIT Press, 1998

Udacities RL repository: https://github.com/udacity/deep-reinforcement-learning

Denny Britz repository: https://github.com/dennybritz/reinforcement-learning/blob/master/MC/MC%20Prediction%20Solution.ipynb

Note from Towards Data Science's editors: *While we allow independent authors to publish articles in accordance with our rules and guidelines, we do not endorse each author's contribution. You should not rely on an author's works without seeking professional advice. See our Reader Terms for details.*