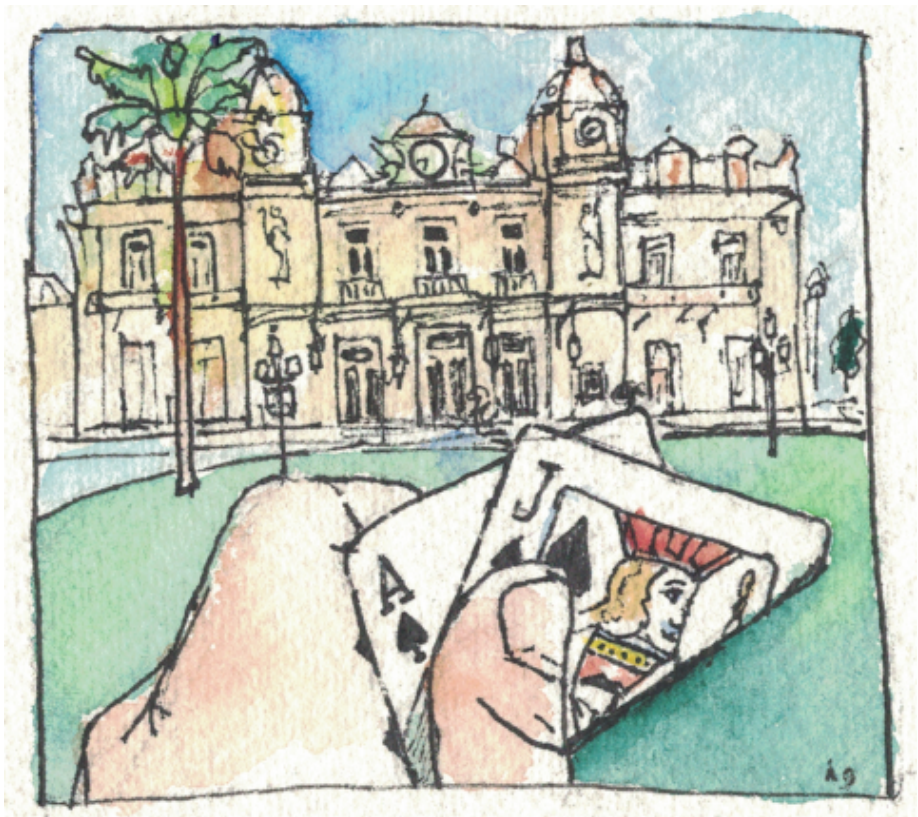
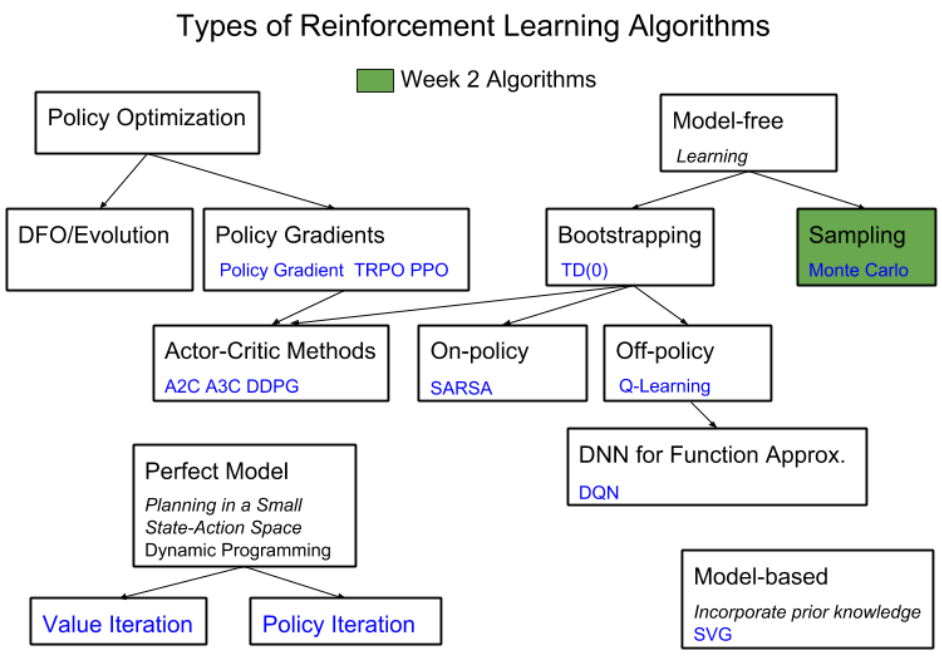
Week 2 - Reinforcement Learning - Monte Carlo Methods and OpenAI Gym's Blackjack

Submitted by hollygrimm on Sat, 06/16/2018 - 05:51



This week I learned about the Reinforcement Learning algorithms called Monte Carlo (MC) methods. Most of my instruction came from Chapter 5 of *Reinforcement Learning: An Introduction* by Sutton and Barto.

MC methods are used on episodic tasks where the model is not known (model-free). The algorithm learns the perfect policy from experience by averaging the sample returns at the end of each episode. Below, I've highlighted MC methods in my RL Algorithms diagram:



Model-free

Last week, I worked with the FrozenLake environment, a "Perfect Model" where all the transition probabilities between states are defined. Alternatively, MC methods learn probability distributions by generating sample transitions over many episodes. A common toy game to test out MC methods is Blackjack.

Episodic Tasks

MC methods work only on episodic RL tasks. These are tasks that will always terminate. The action-value function is updated at the end of each episode.

OpenAI Gym's Blackjack-v0

To play Blackjack, a player obtains cards that total as close to 21 without going over. Face cards (K, Q, J) are each worth ten points. An Ace can be counted as either 1 or 11 points. If 11, it's considered a usable ace.

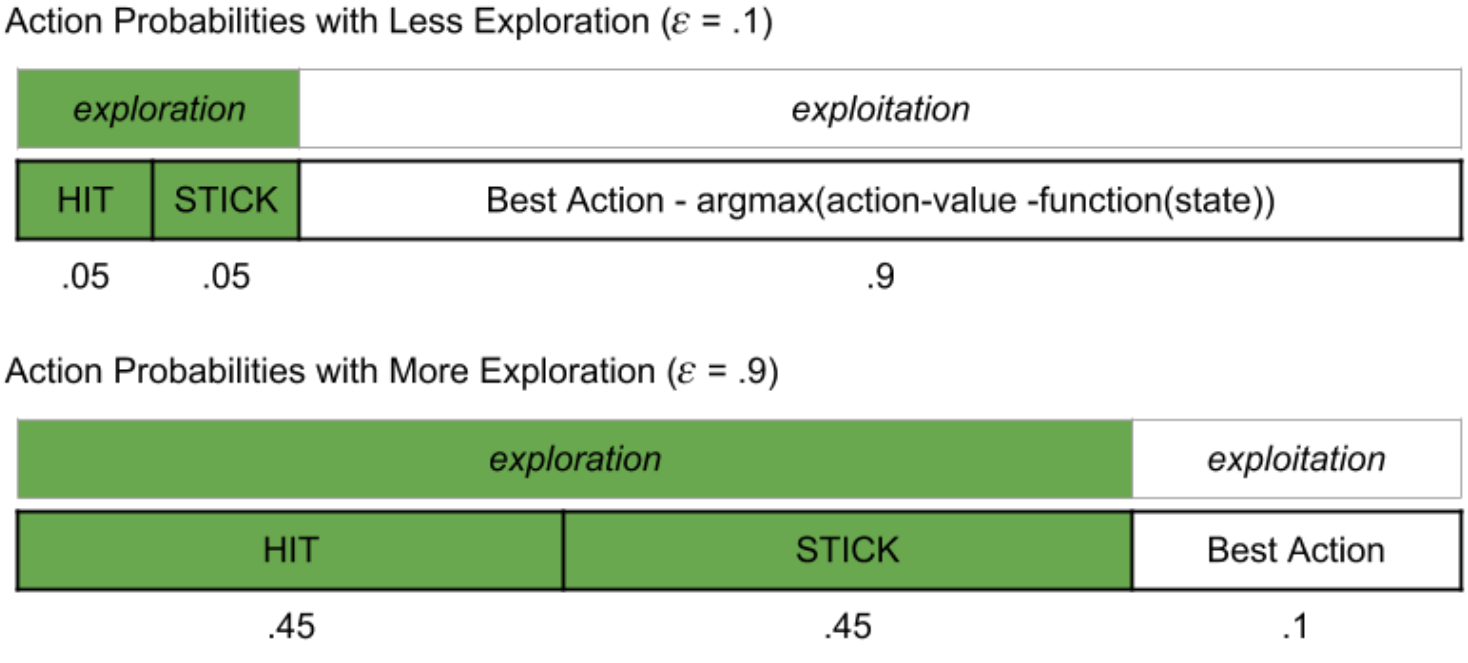
The game starts with the player and dealer each receiving two cards, with one card face up. The player can choose to get more cards by asking for a HIT, or stop getting more cards by asking to STICK, or they may BUST by going over 21. After the player has decided to STICK, the dealer will then draw cards until reaching a sum of 17 or greater.

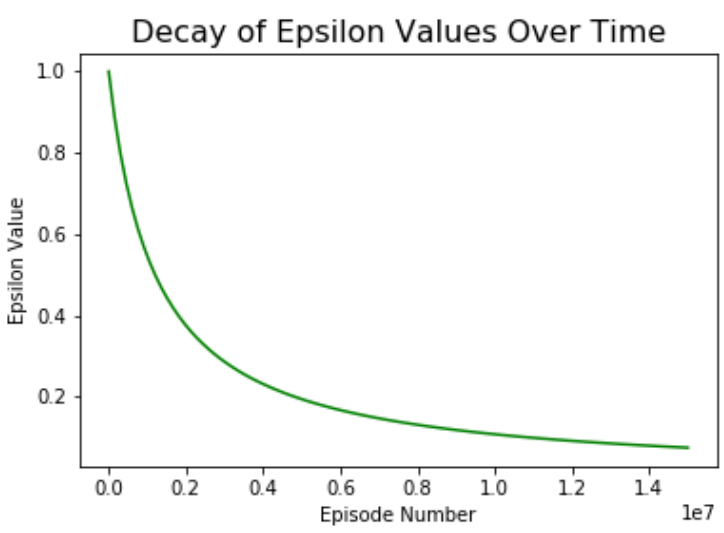
In OpenAI's Gym a state in Blackjack has three variables: the sum of cards in the player's hand, the card that the dealer is showing, and whether or not the player has a usable ace. The cards are dealt from an infinite deck.

Whoever is closer to 21 when the game is over is the winner. Rewards are +1 if the player wins, 0 if there is a draw, and -1 if the player loses.

Action Selection

For action selection, I used the epsilon-greedy approach. When selecting the next action in an episode, an epsilon value is used to increase the exploration within an episode instead of selecting the same greedy action over and over. When using epsilon decay, as more games are played, the epsilon value becomes smaller and the agent begins to exploit rather than explore new actions in a state. Using a higher epsilon value results in more exploration.





Visit

Every occurence of a state/action pair is a *visit* to that state/action pair. In the example below, the same state/action pair state: (20, 1, 1) and action: HIT was visited in Game 1 and Game 2:

Game 1:	20, 1, 1 -> HIT	21, 1, 1 -> STICK	WIN Reward +1
Game 2:	14, 1, 1 -> HIT	20, 1, 1 -> HIT	BUST Reward -1

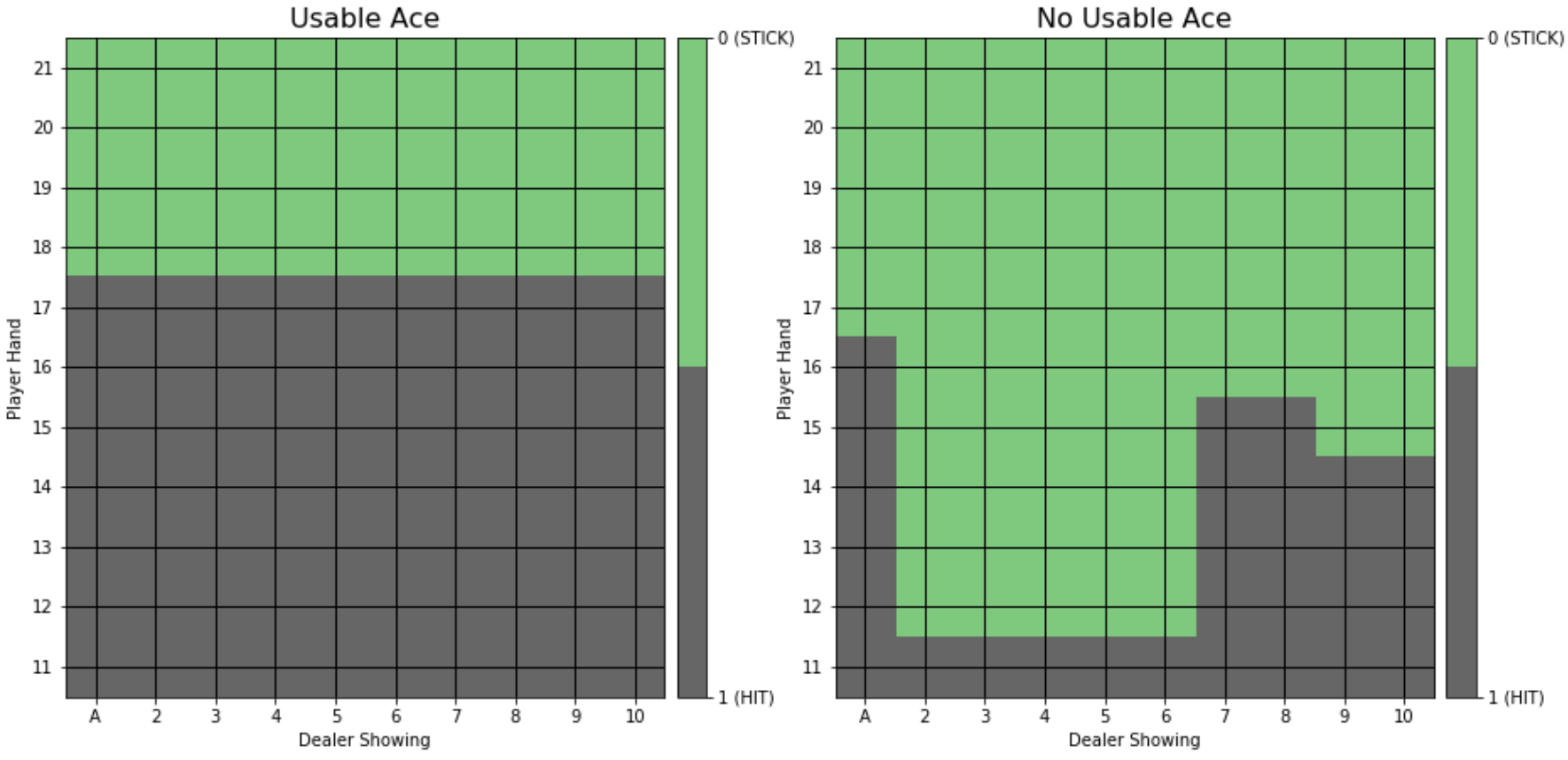
Averaging Sample Returns

As an episode is played, all the states, actions, and rewards are appended to a list.

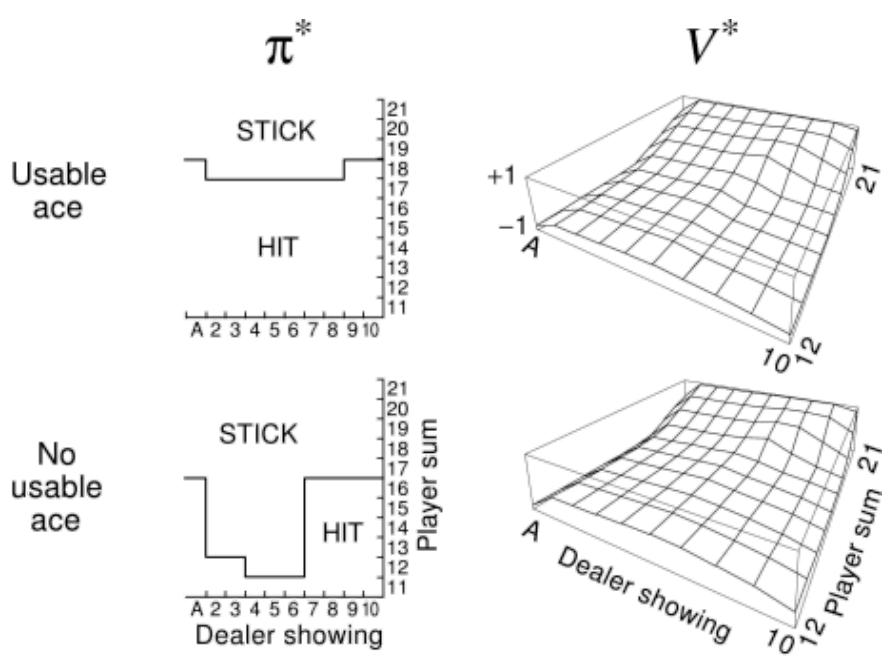
When the episode is complete, the expected return is calculated by summing all the rewards after a state has been observed in the episode. Then the action-value function is updated by taking the sum and dividing it by the number of appearances of that state.

Results

Below is the policy generated after 10,000,000 episodes



Sutton and Barto's optimal policy looks like this:



The policy generated is more conservative then Sutton and Barto's when there is not a Usable Ace, and less conservative when there is a Usable Ace.

Coding Blackjack Monte Carlo

For this project, I created a BlackjackAgent class with the following methods:

- __init__(action_space, epsilon_decay)
- get_policy_for_observation(Q_s, epsilon) calculates action probabilities for a particular observation (state) using an epsilon-greedy method
- choose_action(observation, epsilon) if observation is not yet in the action-value function, sample randomly, otherwise make a random choice using the action probabilities
- update_action_value_function(episode) using the observations, actions, and rewards for one episode, update the action-value function and N count dictionaries
- generate_episode(env) start a new episode and step until done
- log_observation(observation)
- log_done(observation, done)

The main function parses command line arguments, learns an optimal policy by playing multiple episodes, scores the policy, and plots the policy. Logging is currently done to the console.

Testing

For an example, I created a simple unit test to make sure the epsilon greedy probabilities were properly calculated. The python guide has more information on the different test packages in Python: http://docs.python-guide.org/en/latest/writing/tests/

Next Week

I'll be continuing my deep dive into RL with Policy Gradients and Mujoco.

Tags
Reinforcement Learning OpenAI Blackjack Monte Carlo Methods