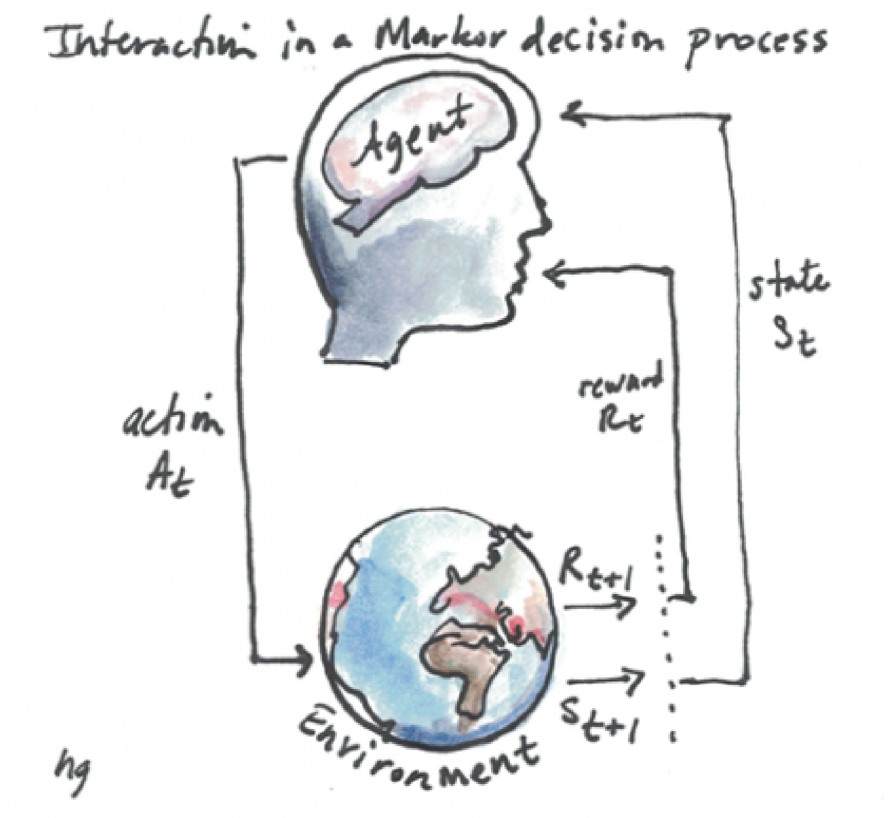
Week 1 - Reinforcement Learning - Markov Decision Processes

Submitted by hollygrimm on Fri, 06/08/2018 - 07:50



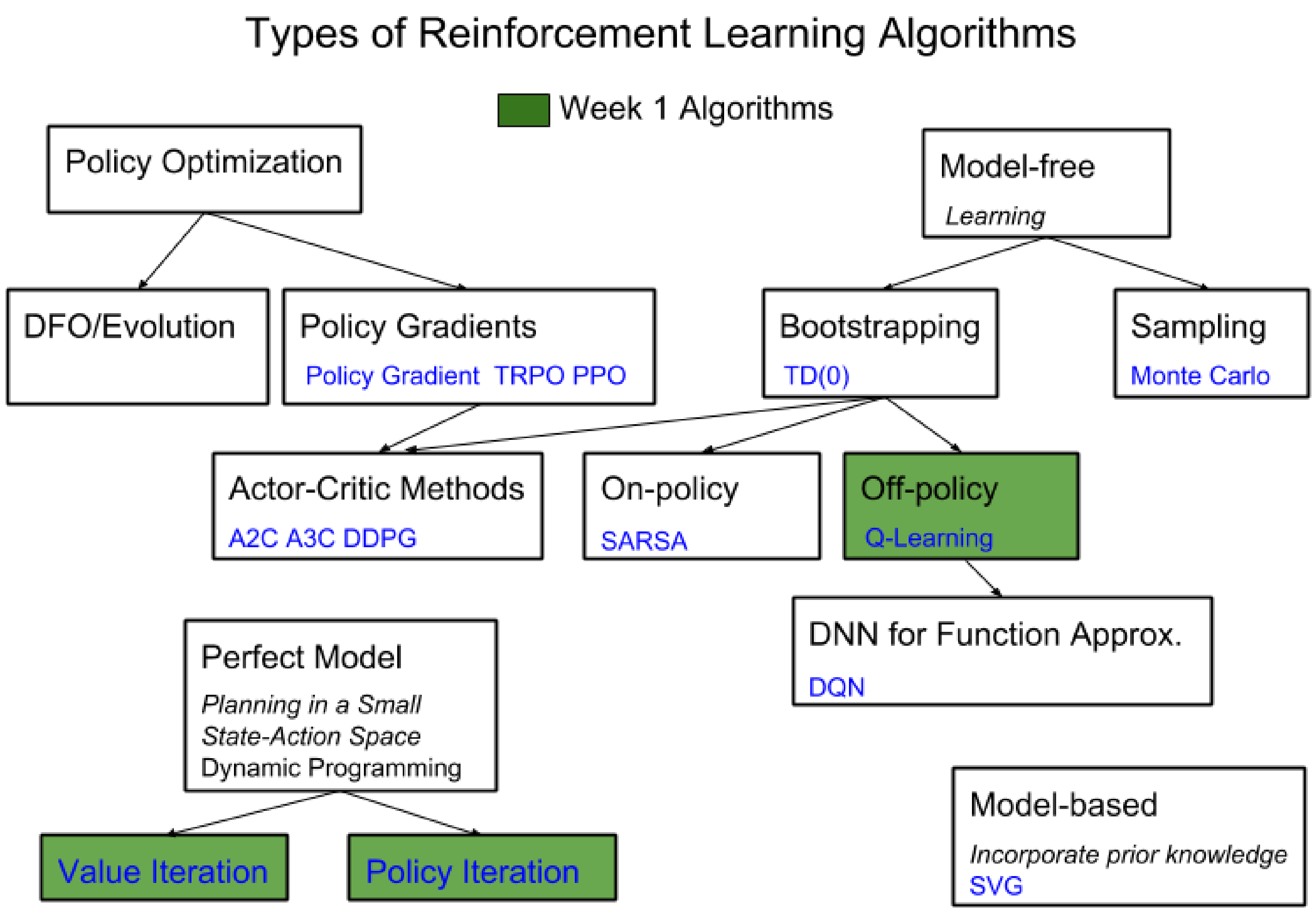
I'm happy to be a member of the inaugural group of OpenAI Scholars. Every Friday for the next three months, I'll be writing a blog post about my Machine Learning studies, struggles, and successes.

I'm a Native American (Navajo) software engineer and painter living in Santa Fe, NM. As a long-time developer, I've built many software systems that would have benefited from Machine Learning (ML) modules. I realize that ML is not a fix for all software development challenges, but I believe it would have helped in several situations that I experienced: 1) a rule-based system that was difficult to create and maintain 2) a large dataset that was overwhelming to sift through and 3) finding patterns in features that weren't easily identifiable.

Last week, all the scholars visited the OpenAI office and met with the OpenAI teams. My mentor is Christy Dennison who is part of the Dota team. During the 2017 competition of Dota players, the OpenAI bot beat several top players in 1v1 matches. We also met with the Robotics team, the Multi-agent team, and the AI Safety team.

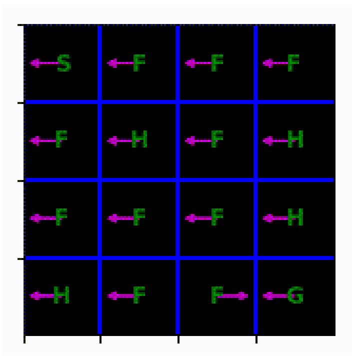
The scholars cohort is a great group with diverse interests and background. I look forward to collaborating with them while I pursue my individual goals.

For the next two months, I'll be doing a deep dive into Reinforcement Learning (RL). I'd like to obtain an understanding of the common RL algorithms and apply them to toy projects running on OpenAI's Gym.



Here is my full syllabus.

My first week (GitHub repo) was spent learning Markov decision processes (MDP). At the beginning of this week, I implemented Value Iteration and Policy Iteration on a finite MDP, the FrozenLake environment. It's considered finite because the entire dynamics of the model is defined. Here is an animation of value iteration:



Value Iteration in Python:

```python
def value_iteration(mdp, gamma, nIt):
    Vs = [np.zeros(mdp.nS)] # list of value functions contains the initial value function V^{(0)}, which is zero
    pis = []
    for it in range(nIt):
        Vprev = Vs[-1] # V^{(it)}
        V = np.zeros(mdp.nS)
        pi = np.zeros(mdp.nS)
        for state in mdp.P: # for all the states in the finite MDP
            maxv = 0 # track the max value across all the actions in the current state
            for action in mdp.P[state]: # for all the actions in current state
                v = 0
                for probability, nextstate, reward in mdp.P[state][action]:
                    v += probability * (reward + gamma * Vprev[nextstate]) # update the value
                if v > maxv: # if value is largest across all the actions, set the policy to that action
                    maxv = v
                    pi[state] = action
            V[state] = maxv # set the value function to the max value across all the actions
        Vs.append(V)
        pis.append(pi)
    return Vs, pis
```

valueiteration.py hosted with ♡ by GitHub view raw

Function parameters:

- mdp - state transition probabilities and rewards, number of actions, number of states
- gamma - discount factor
- nIt - number of iterations through the MDP

In most cases, though, the entire transition model will not be available. Instead, sample-based Q-learning is used to train the agent. Q-learning is an off-policy algorithm, meaning the q values are updated and improved by selecting the next state's value using the greedy policy (deterministic), whereas the agent's action is sampled using the epsilon-greedy policy.

```python
def eps_greedy(q_vals, eps, state): # random action with probability of eps; argmax Q(s, .) with probability of (1-eps)
    import random
    if random.random() < eps:
        action = np.random.choice(len(q_vals[state])) # randomly select action from state
    else:
        action = np.argmax(q_vals[state]) # greedily select action from state
    return action

def q_learning_update(gamma, alpha, q_vals, cur_state, action, next_state, reward): # implement one step of Q-learning
    target = reward + gamma * np.max(q_vals[next_state]) # use the next state's maximum Q value
    q_vals[cur_state][action] = (1 - alpha) * q_vals[cur_state][action] + alpha * target
```

q-learning.py hosted with ♡ by GitHub view raw

Function parameters:

- q_vals - q value table by state and action
- eps - epsilon used in the epsilon-greedy function; encourages exploration
- state - state to select action from
- gamma - discount factor
- alpha - learning rate
- cur_state - current state whose q value will be updated
- action - action taken in current state that resulted in next_state and reward
- next_state - next state as a result of current state and action, used to update the current state's q value
- reward - reward when moving from current state to next state

Here is a video of OpenAI’s Crawler robot attempting to walk with random actions:

0:17 / 0:17

A video of the same Crawler robot after it has been trained for 30,000 steps with a Q-learning algorithm:

0:09 / 0:15

This week I was able to learn and work with the basic concepts of reinforcement learning using simple environments provided by OpenAI Gym. I look forward to expanding on this with Monte Carlo methods using Gym's Blackjack environment next week.

Tags
Reinforcement Learning OpenAI