Reinforcement Learning I

Arnav Jain*

March 2023

1 Overview

Reinforcement Learning (RL) is a machine learning approach that relies on learning from trial and error. RL works by having an agent that explores a world/environment and learns how to make optimal decisions in order to maximize a reward. It is commonly used in fields like game theory and robotics, where the environment and reward function is relatively simple, but it has potential to help many other fields as well.

2 What Makes RL Unique

RL has four key aspects that distinguish it from other forms of machine learning:

- Optimization: Optimal way to make decisions
- Delayed Consequences: Decisions now can have impacts later
- Exploration: Learn about world/environment by making decisions
- Generalization: Being able to generalize learned experiences to new ones

Based on these four key aspects, we can see how other forms of learning falter. For example, supervised and unsupervised learning doesn't explore or account for delayed consequences.

3 The Basics

Before we dive into the math of RL, let's first define some key terms and symbols.

- s_t: s represents the state, which is what the observable environment looks like to the agent at timestep t (e.g. a chess board after move t).
- a: a represents the action, which is an action you perform to transition yourself from one state to the next (e.g. making a chess move).
- π: Pi represents the policy, which is essentially your strategy. It tells you what action to take given your current state.
- α: This represents the learning rate, or how much the policy will change/update when receiving new information.
- γ: Gamma represents the discount factor, which is essentially how we weight long-term vs short-term rewards. A value of 0 indicates not considering long-term rewards at all and a value of 1 represents valuing long-term rewards just as much as short-term.
- V^{π} This is the value function, and it is the cumulative sum of rewards one would obtain from following the policy π from a starting state s.

*Content based on Stanford's CS 234 class

4 Q-Learning

The most basic reinforcement learning technique is Q-Learning. This is a technique that will allow the agent to slowly formulate a good policy. We will define a function $Q(s_t, a_t)$ that tells us how good of a choice making action a_t in state s_t at time t is. We call the value this function produces the *Q-value*. Our policy will be to simply choose the action A that maximizes $Q(s_t, A)$. At the beginning, we will set $Q(s_t, a_t)$ equal to 0 for any state-action pair. There are two ways we can now create a policy: using a Monte Carlo learning method, or a TD Learning Method. The Monte Carlo learning method only updates the policy after one game, or *episode*, has finished, analyzing only the entire game and the total, cumulative reward to update the policy. The TD learning method updates the policy during the episode iteslf, taking into account the immediate reward resulting from a specific state-action pair and estimating the total reward at the end of the episode to update the policy.

4.1 TD Learning

We will first look at the Temporal Difference, or TD, learning method of Q-Learning:

$$Q(s_t, a_t) = Q(s_t, a_t) * (1 - \alpha) + \alpha \left(R_{t+1} + \gamma \max_a Q(s_{t+1}, a) \right)$$

This updates our function Q at every step of time based on the reward we got from the current time step. α is called the learning rate, some constant we define beforehand between 0 and 1 that determines how much we trust the new information we get in comparison to what we already know. High learning rate values will cause the policy to update rapidly, putting a lot of faith in the new information we get. γ is called the discount factor, a number between 0 and 1 that determines how much faith we put in our estimation of our future reward $\max_a Q(s_{t+1}, a)$. After a particular time step t, we analyze our reward and our new state and update the goodness of a particular state-action pair accordingly. We assume our old Q-value had some merit, so we take a weighted average of that old Q-value and our new value. Our new value is based on the reward we got - R_t as well as an estimation of our future reward. We approximate our future reward as the highest possible Q-value (goodness value) we could get in the next state. In general, however, we can't be too certain about the future, so we use a discount factor to represent our skepticism. So, at any given point of time, our agent will perform the action with the highest associated Q-value given the current state, observe the reward received from that action and the new state that it is in, and use that new information to update the Q function. The goal is for the Q-value for a state-action pair to approach the *expectation value* of $(R_{t+1} + \gamma \max_a Q(s_{t+1}, a))$. We assume that this value will not necessarily be the same for a given state-action pair all the time, taking randomness into account. That is why we give weight to both the old Q-value and the new calculated value - this way, the Q-value will approach the value $(R_{t+1} + \gamma \max_a Q(s_{t+1}, a))$ takes on average (i.e. its expectation value).

4.2 Monte Carlo Approach

However, we can put our agent through as many games/episodes as it wants until we are satisfied. Since we can do this, we can also choose to not update our policy mid-episode and use only the final, cumulative reward to update the Q function. This is the Monte Carlo method of Q-Learning, and the associated update formula is:

$$Q(s_t, a_t) = Q(s_t, a_t) * (1 - \alpha) + \alpha G_t$$

We *could* define G_t simply as the cumulative reward from the current time until the end of the game $\sum_{n=0}^{\infty} R_{t+n+1} = R_t + R_{t+1} + R_{t+2} + ...$ (we'll assume that $R_t = 0$ for any time t after the game has ended). But, we are skeptical about the future. Even though we observed the future first-hand in this case, especially with unpredictable environments, we can't be so sure that we will be guaranteed rewards very late in the game. So, we can incorporate out skepticism as a discount factor like so:

$$G_t = \sum_{n=0}^{\infty} \gamma^n R_{t+n+1}$$

Now, our strategy during a game is to simply always take the action with the highest associated Q-value given the current state. After the game finishes, we will go through every state-action pair we experienced and update $Q(s_t, a_t)$ for that state-action pair accordingly. Then, we will start anew. We can play as many games as we want until we are satisfied with our cumulative reward.

5 Gym

You are now ready to get your hands dirty with some actual code! Let's download Gym first.

```
pip install gym
pip install gym[atari]
```

(Add "sudo -H" to the beginning if it complains. Also make sure that you install Gym for the version of Python you are going to be coding with, whether that be Python 2 or 3. To specify Python 3, simply use "pip3" instead of "pip". The second line gives us all the Atari games.) Now, let's open up a file and start writing our first Gym program.

```
import gym
env = gym.make("CartPole-v0")
state = env.reset()
```

This just sets up the Gym environment. The first line imports gym. The second line initializes our environment, env as the Pacman environment. We need to set our state as the initial Pacman start-up state, which we can get from env.reset(). This is standard gym code we will need for pretty much any Gym program.

Here are a few useful methods/attributes of a Gym environment:

Code	Description
env.observation_space.n	returns the number of possible states the environment can be in
env.action_space.n	returns the number of actions the agent can possibly take
env.env.s from 0 to env.observation_space.n-1	returns the state the environment is in right now, represented by a number
env.render()	displays the current environment state in a new window
env.step(a)	updates the environment after the agent takes an action a - some number from 0 to env.action_space.n-1 also returns, as a 4-tuple: (the new state represented by a number, the reward represented by a number, whether or not the game has terminated represented by a boolean, debugging info represented by a dictionary)
env.action_space.sample()	returns a random action the agent could take
env.get_action_meanings	tells us what action each number from 0 to env.action_space.n-1 represents

Now we have all the tools to implement our Q-learning algorithm. Just to get a feel for Gym, you can try out this sample program that uses an agent that simply acts randomly, displaying the environment at each step:

```
done = False
while not done:
    state, reward, done, info = env.step(env.action_space.sample())
    env.render()
env.close()
```

This code is fairly self-explanatory; while the game has not ended, the agent will just continue taking a random action. Now let's try implementing our Q-learning algorithm. We will define Q as a matrix where the row index represents the state and the column index represents the action, commonly referred to as a Q table. Here is the entire program:

```
import gym
import numpy as np
Q = np.zeros([env.observation_space.n, env.action_space.n])
alpha = 0.5
discount = 0.8
for episode in range(1000):
    done = False
    totalreward = reward = 0
    state = env.reset()
    while not done:
        action = np.argmax(Q[state]) #action that maximizes Q for a particular state
        nextstate, reward, done, info = env.step(action)
        env.render()
        Q[state, action] = Q[state,action]*(1.0 - alpha) +
                    alpha*(reward + discount*np.max(Q[state2]))
        totalreward += reward
        state = nextstate
    if episode % 50 == 0:
        print('Episode {} Total Reward: {}'.format(episode,totalreward))
env.close()
```

This code comes straight from the Q-Learning equations outlined earlier on. The learning rate and discount factor were chosen arbitrarily here. You can toy with this code and try out different Gym environments to see how it does. For games

like Pacman, however, we run into an issue. The number of possible states is *huge*, and the agent will rarely experience the same exact state twice. This means that the classic Q-learning algorithm will not help, since (in all likelihood) every new state we encounter will have never been seen before and will thus have a Q-value of 0. But a Q-value about one state should help us determine the Q-value for another state, even if the two states are not exactly the same. For example, if two states differ by a few dots or if one of the ghosts' locations differ by a few pixels, the states are nonetheless very similar, so the Q-value associated with one should also be associated with the other. It would be nice if we could have some way to transform a state-action pair into a Q-value without using a handwritten formula - something more complex that could learn the similarities between only slightly different, but still distinct states on its own.

6 Deep Q-Networks

This leads us to consider using neural networks to predict the Q-value for a state-action pair - Deep Q-Networks (DQNs). We will structure our DQN to take in just the state and predict a Q-value for each possible action. (This is more efficient than having the network take in both the state and the action as inputs as this approach would require a pass through the network for each action which we need in order to decide which action to take, while having the network predict a Q-value for each possible action only requires one pass to give us enough information to make our decision.) We will have our network's target be the value we wanted the Q-value to approach in our original TD learning equation - $(R_{t+1} + \gamma \max_a Q(s_{t+1}, a))$. Since we can only get one state-action pair for our network at a time, we will have the target vector be the Q-value for the action we observe have elements equal to zero for the indices that correspond to other actions and have the element at the index that corresponds to the action that we *did* see equal to the Q-value for that state-action pair. Our policy will be based on the ϵ-greedy algorithm this time; in order to get some variety in the network's input diet, we want to add randomness when choosing an action. With probability ϵ, we will simply choose some random action. With probability $1 - \epsilon$, we will choose the action that maximizes the Q-value for the current state (according to the neural network). The probability ϵ will actually be a function of time; we will start it out very high so that our network quickly experiences all the available options for actions and then decrease it as time goes on so we can hone in on choosing the optimal actions. So, the algorithm for a single episode is:

1. Use ϵ-greedy to choose and perform an action; that is, with probability ϵ, choose a random action. Otherwise, feed the current state to the DQN and choose the action associated with the highest Q-value.
2. Observe the reward associated with that action and send this information to train the DQN.
3. Repeat until the episode is finished.

7 Sources

- Stanford CS 234 Reinforcement Learning Series
- Charlie Wu's Beginner RL Lecture