COMS 4771 HW4 (Spring 2020)

Due: Apr 24, 2020 at 11:59pm

This homework is to be done **alone**. No late homeworks are allowed. To receive credit, a typesetted copy of the homework pdf must be uploaded to Gradescope by the due date. You must show your work to receive full credit. Discussing possible solutions for homework questions is encouraged on piazza and with your peers, but you must write their own individual solutions. You should cite all resources (including online material, books, articles, help taken from specific individuals, etc.) you used to complete your work.

1 Reinforcement Learning

Reinforcement learning (often abbreviated RL) is an area of machine learning that explores how an agent should interact with its environment in order to maximize some notion of total reward. RL algorithms were behind DeepMind's AlphaGo and AlphaGo Zero systems that beat Go masters in 2017. The goal of an RL algorithm is typically to learn a **policy** $\pi(state)$ (a function that determines what action(s) to take in a given state) such that the total (expected) future reward $\mathbb{E}\left[\sum_k^\infty R_{t+k}\right]$ is maximized.

For example, the challenge of controlling a robot can be formulated as an RL problem in which rewards are given for successfully picking up an object, states are the position/velocity of the robot, and the learned **policy** predicts what torques (actions) should be applied to each joint in order to move closer to achieving the goal. The aim of RL is to learn an optimal policy π_* purely from exploration and experimentation in the environment.

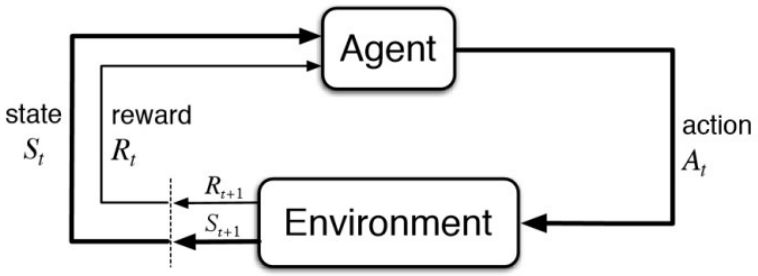

Figure 1: A Markov Decision Process (MDP), the theoretical framework of an RL problem.

Figure 1 shows the general setup for a reinforcement learning problem. At each time step t, an agent in a state S_t chooses an action A_t, and transitions (sometimes stochastically) into a new state S_{t+1} and receives a reward R_{t+1}. This framework is called a Markov Decision Process, or MDP, defined by:

- a set of states $\mathcal{S}$ (e.g. the possible configurations of the robot and the environment, i.e. the state space)
- a set of actions $\mathcal{A}$ (e.g. the possible torques the robot can apply to various joints)
- a transition function $P(s' \mid s, a)$ which gives the probability of transitioning from state s to state s' when taking action a (in this case, a distribution over possible positions the robot can end up in after applying torques to imperfect motors)
- a reward function $R_a(s, s')$, the reward received when transitioning to from state s to state s' after taking action a (in this case, usually 1 if the robot succeeds at picking up the object, and 0 otherwise). This can be random or deterministic. For this problem, we will assume it is deterministic (this is in contrast to a k-armed bandits problem when you are sampling from certain actions to better understand the reward distribution).

Our usual goal is to learn a policy $\pi : \mathcal{S} \to \mathbb{P}(\mathcal{A})$, a function that gives a distribution over possible actions in a given state that maximizes the expected future (discounted) reward

$$G_t = \mathbb{E}\left[\sum_{k=1}^{\infty} \gamma^{k-1} R_{t+k}\right]$$

where the expectation is taken over $P(s' \mid s, a)$. The discount factor γ is usually a small number $0 \leq \gamma \leq 1$. If $\gamma = 0$, the reward is greedy and only considers the next immediate reward, while if $\gamma = 1$, the algorithm weighs all rewards equally far into the future); a middle ground is usually preferable in which you consider some future rewards but decrease their importance as they become more distant uncertain. Note that G_t is still a random variable with respect to the actions taken by the policy π.

1. **Finding the value of a state under a policy**

   Define the **value** of a state under a policy π to be $v_\pi(s) = \mathbb{E}_\pi [G_t \mid S_t = s]$. Intuitively, this gives the expected reward achieved starting in a certain state and choosing actions according to the policy π. While not all RL algorithms use this notion of value, a value function can be extremely useful and intuitive if the set of states is small. Using this definition, show the following recursive definition for the value function:

   $$v_\pi(s) = \sum_a \pi(a \mid s) \sum_{s'} P(s' \mid s, a) \left[R_a(s, s') + \gamma v_\pi(s')\right]$$

   Hint: $G_t = R_{t+1} + \gamma G_{t+1}$

2. **Solving for a value function using linear algebra**

   The result from the previous section gives us a recursive formula for defining the value function. For every state, we can express $v(s)$ as a weighted sum over the values of possible subsequent states $v(s')$ (with a bias term). If our state space is finite, this gives us a system of linear equations $v(s_i) = \sum a_j v(s_j)$ for an appropriate choice of a_j.

   Assume all transitions are deterministic, i.e. $p(s' \mid s, a) = \mathbb{1}[s' = next(s, a)]$ for some deterministic function $next : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$. Using your result from (1), write a system of linear equations for the value function $v(s_i)$ in the form $A \cdot v = b$ where $v = [v(s_1), ..., v(s_n)]$.

3. **Finding the value of states "in the real world"**

   Now we will use the linear equation from (ii) to solve these equations in a "real" example. Figure 2 shows a gridworld environment, a kind of maze where different actions receive different rewards.

   Each grid cell represents a state, and in each cell four actions are possible: `north`, `south`, `east`, and `west`. Actions that move the agent off the board leave its position unchanged but incur a penalty of -1. All other actions move the agent in the chosen direction and receive no reward or penalty, except for actions that move the agent out of the special cells A and B. If the agent takes any action while in cell A, it receives a reward of 10 and moves to position A'. If the agent takes any action while in cell B, it receives a reward of 5 and moves to position B'.

   Note: the following alternate interpretation of this question is also fine: if you take an action that would move you into A, you receive a reward and move to A' instead (same for B). If you use this version, you will not lose points.

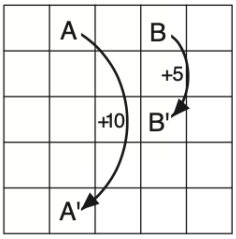

Figure 2: The gridworld environment with special cells A and B

Using this model of an MDP with a deterministic state function and rewards and a uniform random policy $\pi(a|s) = 0.25$ for all actions and all states, use the linear equation from the previous section to solve for the value function under this policy. You may use a linear equation solver like `numpy.linalg.solve`. What about for a non-uniform policy $\pi(\texttt{up}|s) = 0.7$, $\pi(\text{other actions}) = 0.1$? Plot these values in a grid like figure 2. Do these values make sense? For your solution, choose a starting state and check if the recursive definition holds. Try a few other policies and see if you can find a good policy that gives high values to more states.

4. **Finding an optimal value function**

   (a) Note that the previous set of equations does not give you a formula for the optimal value function (the value under the optimal policy). It merely allows you to efficiently find the value of a state (the expected reward from that state) under a **specific** policy π. Now the question is *how to find a good policy in the first place*?

   For the optimal value function $v_*(s)$ which maximizes $v_*(s)$ among all possible π, show that

$$v_*(s) = \max_a \sum_{s'} p(s' \mid s, a) \left[R_a(s, s') + \gamma v_*(s') \right] \tag{1}$$

   Note that this equation is independent of π.

(b) Given an optimal value function $v_*(s)$ in a finite environment with access to the model $P(s' \mid s, a)$ and the reward function $R_a(s, s')$, explain specifically how to find the optimal policy. Prove that this is indeed optimal. Is this policy random or deterministic?

(c) In some settings, it's more useful to talk about the value of a (state, action) pair instead of a single state. Just like the value function, we define the q function $q : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$ which gives the expected reward from taking an action a in a state s.

$$q_\pi(s, a) = \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right]$$

Extend your result from $4a$ to show that q_π satisfies

$$q_*(s, a) = \sum_{s'} p(s' \mid s, a) \left[R_a(s, s') + \gamma \max_{a'} q_*(s', a') \right] \tag{2}$$

5. **Finding the optimal policy using iterative methods**

While equation 1 is no longer a system of linear equations, for a finite environment like gridworld, it gives a system of non-linear equations that can be solved using a variety of iterative algorithms that converge over time to the optimal solution. We will explore one method that iterates on the value function, and prove that it converges to the optimal value function.

Here is the algorithm:

Algorithm 1 Value iteration algorithm

for $s_i \in \mathcal{S}$ **do**
  $V(s_i) \leftarrow 0$ (initialize value to zero)
end for
while not converged **do**
  for $s_i \in \mathcal{S}$ **do**
    $V(s_i) \leftarrow \max_a \sum_{s'} p(s' \mid s, a) \left[R_a(s, s') + \gamma V(s') \right]$
  end for
end while

The intuition behind this algorithm is simple: you are repeatedly applying the definition of the optimal value function from 4 to improve your estimates for the value of each state. Now it is your task to show that this algorithm converges to the optimal value function $v_*(s)$ as the number of iterations $\to \infty$. Assume $0 < \gamma < 1$. Assume also that the state and action spaces $(\mathcal{S}, \mathcal{A})$ are finite, and let $V_t(S)$ be the vector of values assigned to each state at iteration t. Prove that

$$\|V_t(S) - V_*(S)\|_\infty \leq \gamma^t \|V_0(S) - V_*(S)\|_\infty$$

Show that this implies convergence as $t \to \infty$.

6. **Find the optimal value function for gridworld**

Returning to the gridworld environment from part 3, use this algorithm to find the optimal value of each state. Show the value of each state in a grid/table like Figure 2. Pick a few random positions in the grid and describe the resulting trajectory under the optimal policy (see part 4b). Does it make sense that this is the optimal policy?

7. **A model-free approach**

So far, we have talked about model-based Reinforcement Learning. The agent is already aware of all the possible states and actions in its environment, so it can solve the problem without taking a single action because of its prior knowledge about the environment (concretely, this means $P(s' \mid s, a)$ and $R_a(s, s')$ are known a priori). However, what if the agent has no prior information about the environment? It will need to *explore* to learn about the environment. Consider the MDP shown below. In states A through E, one can either move left or right. The reward of each transition is marked in the below diagram:

If we randomly drop an agent into this MDP without telling it anything about the rewards and transitions in the environment, it will need to explore the environment itself in order to learn the optimal solution (moving all the way to the right).

One of the most common approaches to this problem involves estimating the q function (from 4c) instead of the value function (from 4a). To learn this state-value function, we initialize the value of each (state, action) pair to zero, and each time the agent completes a simulation in the environment, we update our estimate of the value of each pair using the following rule:

$$Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma \max_a Q(S', A) - Q(S, A) \right] \tag{3}$$

where the learning rate is given by parameter α, and R is the reward. We run many simulations, initially taking actions completely at random, and gradually improve our estimates of the true value while also improving our policy at the same time. This strategy is known as an *ϵ-greedy* policy. The agent takes the currently optimal action with probability $1-\epsilon$, but takes a random action with probability ϵ at each step ($0 \leq \epsilon \leq 1$). This is essentially a value iteration algorithm like (5), but we have to estimate the transition functions instead of using them in our estimates. This is a practical algorithm to solving a whole host of important control problems.

i) Intuitively, why would you expect this to work (converge to the optimal state action function)? *Hint: value iteration.* How does the Q estimate change as large or smaller rewards are encountered?

ii) Implement this update rule to train an agent to solve the Cartpole problem, whose gym environment is available through OpenAI open source (https://gym.openai.com/envs/CartPole-v1/). The environment name is 'Cartpole-v1'. Information and Documentation can be found through the above link. Your agent should be able to keep the pole up for at least 200 frames after training for enough episodes. Write a short (1-paragraph) report describing your implementation (how you decided to keep track of your q estimates, what hyperparameter settings you used, how your agent improved over time, etc.) and include a performance graph that plots the number of frames per episode.

Some Tips:

- You'll need to formulate Cartpole as an MDP, which entails discretizing the state space (the position of the cart and the angle of the cart) into a finite grid, since the default observation space is continuous.

- You will need to keep track of your q estimates for each state-action pair. You should initialize the q values for each state-action pair to 0, and use the reward information given from `env.step()` to update your q value estimations as you run simulations.
- You can tune your model by changing your values of γ, α, and ϵ. In general, it is common practice to decay the learning rate α and the epsilon-random rate ϵ as your training progresses, since as your algorithm converges, smaller update steps and less randomness tend to be more beneficial. You may also wish to experiment with different grid sizes for your observation/state space.
- Consider having your agent use an ϵ-greedy policy rather than a random policy (your algorithm might converge faster). As your Q values get more accurate, you may wish to reduce ϵ so fewer random actions are made. For example, you might want to start with a fully random policy ($\epsilon = 1$) and then reduce ϵ to 0.1 or 0.01 over the course of one or two thousand iterations.

Extra Credit

An even simpler update rule than (3) is the following: We initialize the value of each state to zero, and each time the agent completes a simulation in the environment, we update the value of each state using the following rule (under a fixed exploration policy):

$$V_\pi(S_t) \leftarrow V_\pi(S_t) + \alpha \left[G_t - V_\pi(S_t) \right] \tag{4}$$

where the learning rate is given by parameter α, and G_t is the discounted reward received starting at time t from one specific episode ($G_t = \sum \gamma^{n-1} R_{t+n}$).

In order to use this update rule, we need to simulate an entire episode to the end before updating the state values (so we know G_t). What if we want to make updates based on intermediate rewards received before the episode has finished?

i) Estimate G_t using only the reward R_{t+1} and V_{t+1} ("one-step-ahead" estimate) instead of the discounted sum of rewards R_{t+n} for all $n \geq 1$. Rewrite the recursive update rule (2) using this estimate of G_t. What about estimating G_t using rewards received from the next *two* timesteps? Write the recursive update rule for this as well.

ii) Let's generalize this to an arbitrary estimate of G_t. Estimate for G_t using the rewards received up to n steps into the future. Write the recursive update rule (2) using this "n-steps-ahead" estimate.

iii) What happens to the variance of our estimate of G_t for larger choices of n? What about the bias of the estimate? You do not need to calculate the bias or variance, but you must explain your reasoning fully. When would we prefer a small n vs. a large n in our estimate? Why?

Now, what if we don't want to choose a value for n? We can use all of the n at once! Let $G_t^{(n)}$ denote the value of G_t using an n-steps-ahead estimation (from (iii)). We can estimate G_t as a convex combination of **all** the estimates $G_t^{(n)}$, where larger values of n are worth exponentially less than smaller ones. Consider a parameter value λ (the exponential discount factor). We can write our new estimation rule as:

$$G_t^{(\lambda)} = \sum_{n=1}^{T} a_n G_t^{(n)}$$

$$a_n = O(\lambda^n)$$

iv) Given that $\sum_n a_n = 1$ ($G_t^{(\lambda)}$ is a convex combination of all the possible $G_t^{(n)}$), find the explicit expression for a_n, and write the recursive update rule (4) for this algorithm.

v) How often will this algorithm make updates to $V(S_t)$, and why?

vi) What happens to the variance of $G_t^{(\lambda)}$ for larger choices of λ? What about the bias? Again, you do not need to calculate the bias or variance, but you must explain your reasoning fully. When would we prefer a small λ vs. a large λ in our estimate? Why?