CS 3600: Reinforcement Learning Project Hints

Siyan Li
Osama Sakhi
Satwik Mekala

Georgia Institute of Technology

Contents

1 Notation:

Here we'll define the terms used within the project and how they may relate to concepts from the course.

1.1 `reward`

Rewards are state-specific indications of how good a certain state is. These are often arbitrary. A positive reward means that this state is good, while a negative reward indicates that this state is bad.

1.2 `utility`

Utilities, similar to rewards, also measure how good a state is. However, they are different as in the reward only accounts for the current moment, whereas the utility accounts for future rewards. During value iterations (will be detailed later), only the utilities of the states will be updated, while the rewards of the states remain the same.

1.3 `mdp`

A Markov Decision Process (MDP) provides the best action for every state in a fully-observable and action-stochastic environment. An MDP is defined by:

- S: set of states
- S_0: the initial state, $S_0 \in S$
- A: set of actions
- $T(s, a, s')$: the transition function that returns the probability of ending up in state s' from state s if action a is chosen. This is equivalently $P(s'|s, a)$.
- $R(s)$: the reward function, $R(s) \in \mathbb{R}$

Over time, the MDP will learn an `optimal policy` π^*,

$$\pi^*(s) = \underset{a \in A}{\operatorname{argmax}} \sum_{s' \in S} T(s, a, s')U(s')$$

which gives the best action in every state.

1.4 `answerDiscount`

This corresponds to the discount factor, γ.

1.5 `answerNoise`

This corresponds to the probability of **not** going in the intended direction. In other words, this affects our transition function $T(s, a, s')$. The higher your `answerNoise`, the more likely you agent is to go *any* direction other than your intended direction.

1.6 `answerLiving`

This corresponds to a reward for non-terminal states. It's a reward for staying "alive", meaning for being in a state other than the states with payoffs like $+1$ or -10.

2 Value Iteration

Value iteration is a process that updates the utilities of all the states. After a large number of iterations, these values should be close enough to the "real" values, and should therefore be able to supply an optimal policy.

2.1 Initializing Utilities

We can initialize the first set of utilities to 0, i.e. $U^0(s) = 0$ for all states s. If you initialize the utilities to $R(s)$, the rewards, you'll just have an off-by-one error for this autograder (as in, you'll end up doing one more iteration of value iteration than what the autograder expects), so keep that in mind.

2.2 Updating Utilities: Bellman Update Equation

This part of value iteration updates the set of values from iteration i to iteration i + 1. The process is based on the `Bellman Update Equation`:

$$U^{(i+1)}(s) = R(s) + \gamma \max_{a \in A} \sum_{s' \in S} T(s, a, s') U^i(s') \tag{1}$$

This process is repeated for all of the states, and the utilities of these states will converge eventually. It is important to note that the set of utilities from the **previous iteration** is used to update the set of utilities in the current iteration.

2.3 In Code

We can think of each state s as a key to some table U^i which has the utility values for the i-th iteration. As we compute the values for the new table $U^{(i+1)}$, we'll want to make sure that we don't overwrite the old table. We just need to use the values from the previous table. This means you'll want a secondary table that you hold all new utility values in (U^{i+1}), while you reference the old table (U^i).

You'll find that to implement the `__init__()` function, your helper functions `computeActionFromValues()` and `computeQValueFromValues()` will be very helpful if you've already implemented them. Think of `computeActionFromValues()` as telling you the argmax of that summation in Equation 1, and the `computeQValueFromValues()` telling you the value of summation itself.

3 Discount Factors, Noise, and Rewards for Living

3.1 Problem Setup

In Q2 and Q3, you'll be setting up the rewards and discount factors in a way that will allow the agent to take the desired path. As you try different values for `answerDiscount` and `answerNoise` and `answerLiving`, think of how each variable individually will affect the agent.

3.2 Varying `answerDiscount`

Keeping all other variables fixed, let's think about `answerDiscount`. Does decreasing this value make the agent prefer terminal states that are closer or further? What happens if the reward is greater than 1?

3.3 Varying `answerNoise`

Keeping all other variables fixed, let's think about `answerNoise`. As we increase this value, how does the agent behave? Is the agent more willing or less willing to go near the "cliff"?

3.4 Varying `answerLiving`

Keeping all other variables fixed, let's think about `answerLiving`. As we increase this value, how does the agent's path change? Is it more willing to take a risky path since it gets some payoff from just staying alive?

4 Q Learning

4.1 A Whole New World

We use Q-Learning in a *model-free environment*, meaning an environment that we don't have a model for. In other words, this is no longer an MDP. Specifically, we don't have a Transition model, $T(s, a, a')$ or even a reward function $R(s)$. It's up to our agent to discover how to navigate this world and understand the rewards. In that regard, it's as if we're exploring a whole new world.

4.2 Learning From Examples

In this type of environment, we learn how to navigate our world by learning from experiences. Those experiences come in the form: (s, a, s', r):

- s - state we started at
- a - action we took
- s' - state we ended up at
- r - the reward we obtained

Tuples of this form are given as input to our agent, and the agent then computes Q-values for each s and a in each tuple. Q-values are defined for pairs of states and actions, i.e. $Q(s, a)$, and this can be understood as the "quality" of this state-action pair in getting some future reward. Using the Q-values, our agent can then come up with a policy for navigating in this world.

4.3 Initializing Q Values

Think of a Q-learning agent as being a clean slate, it has no notion of where to go or how rewarding anything is. So as a starting point, the Q value for all state-action pairs can just be 0. You'll need to initialize your Q-table in the `__init__()` function.

4.4 Updating Q Values

Remember, to update the Q-values, we are given some new sample of the form: (s, a, s', r). Using this example, we can then update our existing Q-values with the following rule:

$$Q(s, a) = (1 - \alpha)\, Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a')) \tag{2}$$

Note: You have a lot of helper functions within the `QLearningAgent` class, and those will be really useful in completing Q4 and Q5.

4.5 Epsilon-Greedy Policy

Epsilon-Greedy Strategy is a way to combine random and on-policy (picking the action that provides the best Q value all the time) strategies. The main idea is this:

- The best action for a given state can be determined via the Q-values: i.e

$$\texttt{bestAction} = \operatorname*{argmax}_a Q(s, a) \tag{3}$$

- Instead of *always* picking `bestAction`, with some probability ϵ, we will actually choose a random action. This will allow us to "explore" more of the grid instead of always going with what we think is the current best, i.e "exploitation".

This problem of Exploration versus Exploitation is a common theme in Reinforcement Learning, and you can learn more about it here.