CS440 – Reinforcement Learning

Purpose: In this lab, you will implement value iteration and Q-learning. You will test your agents first on Gridworld (from class), then apply them to Pacman.

Getting Started: Download the reinforcement.zip files and complete each of the following assignments to be submitted for grading. Each should be done individually but you can consult with a classmate to discuss your strategies or if you get an error message that you do not understand.

Assignment 1 – Value Iteration:

Recall the value iteration formula:

$$V_{k+1}(s) = \max_a \sum_{s'} T(s,a,s') \Big[R(s,a,s') + \gamma V_k(s') \Big]$$

Write a value iteration agent in `ValueIterationAgent`, which has been partially specified for you in `valueIterationAgents.py`. Your value iteration agent is an offline planner, not a reinforcement learning agent, and so the relevant training option is the number of iterations of value iteration it should run in its initial planning phase. ValueIterationAgent takes an MDP on construction and runs value iteration for the specified number of iterations before the constructor returns.

Value iteration computes k-step estimates of the optimal values, V_k. In addition to running value iteration, implement the following methods for ValueIterationAgent using V_k.

1. `computeActionFromValues(state)` computes the best action according to the value function given by self.values.
2. `computeQValueFromValues(state, action)` returns the Q-value of the (state, action) pair given by the value function given by self.values.

These quantities are all displayed in the GUI: values are numbers in squares, Q-values are numbers in square quarters, and policies are arrows out from each square.

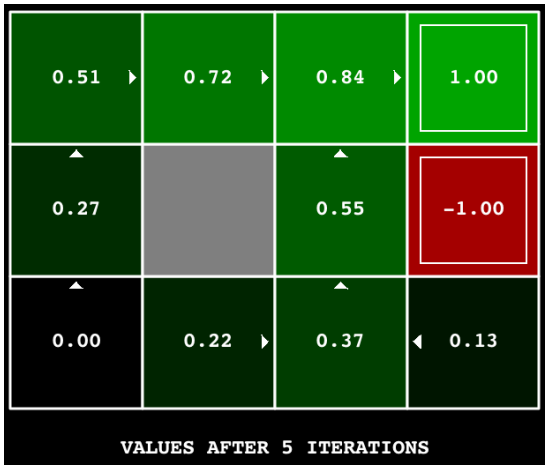
Important: In class we discussed the "batch" version of value iteration where each vector $V_k$is computed from a fixed vector V_{k-1} . This means that when a state's value is updated in iteration k based on the values of its successor states, the successor state values used in the value update computation should be those from iteration k-1 (even if some of the successor states had already been updated in iteration k). Use this version!

Criteria for Success:

Use the autograder with q1 to verify everything is working.

You can also run your code by running `gridworld.py` with the command options:
`-a value -i 5`

After five iterations you should get:


0.51
0.72
0.84
1.00
0.27
0.55
-1.00
0.00
0.22
0.37
0.13
VALUES AFTER 5 ITERATIONS

Assignment 2 – Bridge Crossing Analysis:

BridgeGrid is a grid world map with the a low-reward terminal state and a high-reward terminal state separated by a narrow "bridge", on either side of which is a chasm of high negative reward. The agent starts near the low-reward state. With a discount of 0.9 and the default noise of 0.2, the optimal policy does not cross the bridge. You can see the result by running `gridworld.py` with the options:
`-a value -i 100 -g BridgeGrid --discount 0.9 --noise 0.2`

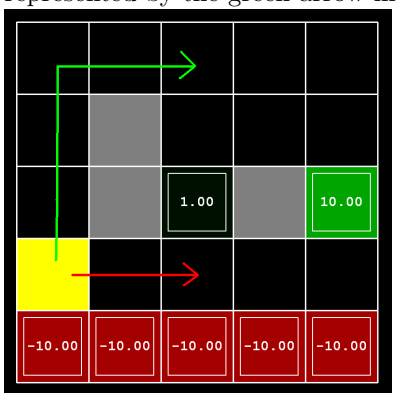
Change only ONE of the discount and noise parameters so that the optimal policy causes the agent to attempt to cross the bridge. Put your answer in `question2()` of `analysis.py`. (Noise refers to how often an agent ends up in an unintended successor state when they perform an action.) The default corresponds to:

Criteria for Success:

Use the autograder with q2 to verify everything is working. The autograder checks that you only changed one of the given parameters, and that with this change, a correct value iteration agent should cross the bridge.

Assignment 3 – Policies:

Consider the DiscountGrid layout, shown below. This grid has two terminal states with positive payoff (in the middle row), a close exit with payoff +1 and a distant exit with payoff +10. The bottom row of the grid consists of terminal states with negative payoff (shown in red); each state in this "cliff" region has payoff -10. The starting state is the yellow square. We distinguish between two types of paths: (1) paths that "risk the cliff" and travel near the bottom row of the grid; these paths are shorter but risk earning a large negative payoff, and are represented by the red arrow in the figure below. (2) paths that "avoid the cliff" and travel along the top edge of the grid. These paths are longer but are less likely to incur huge negative payoffs. These paths are represented by the green arrow in the figure below.



Choose settings of the discount, noise, and living reward parameters for this MDP to produce optimal policies of several different types. Your setting of the parameter values for each part should have the property that, if your agent followed its optimal policy without being subject to any noise, it would exhibit the given behavior. If a particular behavior is not achieved for any setting of the parameters, assert that the policy is impossible by returning the string 'NOT POSSIBLE'.
Here are the optimal policy types you should attempt to produce:

1. Prefer the close exit (+1), risking the cliff (-10)
2. Prefer the close exit (+1), but avoiding the cliff (-10)
3. Prefer the distant exit (+10), risking the cliff (-10)
4. Prefer the distant exit (+10), avoiding the cliff (-10)
5. Avoid both exits and the cliff (so an episode should never terminate)

Criteria for Success:
Use the autograder with q3 to verify that the desired policy is created for each case. In `analysis.py`, question3a() through question3e() should each return a 3-item tuple of (discount, noise, living reward).

Assignment 4 – Q-Learning:

Note that your value iteration agent does not actually learn from experience. Rather, it ponders its MDP model to arrive at a complete policy before ever interacting with a real environment. When it does interact with the environment, it simply follows the precomputed policy (e.g. it becomes a reflex agent). This distinction may be subtle in a simulated environment like a Gridword, but it's very important in the real world, where the real MDP is not available.

You will now write a Q-learning agent, which does very little on construction, but instead learns by trial and error from interactions with the environment through its `update(state, action, nextState, reward)` method. A stub of a Q-learner is specified in `QLearningAgent` in `qlearningAgents.py`. For this question, you must implement the `update`, `computeValueFromQValues`, `getQValue`, and `computeActionFromQValues` methods.

For `computeActionFromQValues`, you should break ties randomly for better behavior. The `random.choice()` function will help. In a particular state, actions that your agent hasn't seen before still have a Q-value, specifically a Q-value of zero, and if all of the actions that your agent has seen before have a negative Q-value, an unseen action may be optimal.

Make sure that in your `computeValueFromQValues` and `computeActionFromQValues` functions, you only access Q values by calling `getQValue` . This abstraction will be useful later when you override `getQValue` to use features of state-action pairs rather than state-action pairs directly.

Criteria for Success:
The autograder with q4 will run your Q-learning agent and check that it learns the same Q-values and policy as the reference implementation when each is presented with the same set of examples.

Assignment 5 – Q-Learning and Pacman:

Time to play some Pacman! Pacman will play games in two phases. In the first phase, training, Pacman will begin to learn about the values of positions and actions. Because it takes a very long time to learn accurate Q-values even for tiny grids, Pacman's training games run in quiet mode by default, with no GUI (or console) display. Once Pacman's training is complete, he will enter testing mode. When testing, Pacman's `self.epsilon` and `self.alpha` will be set to 0.0, effectively stopping Q-learning and disabling exploration, in order to allow Pacman to exploit his learned policy. Test games are shown in the GUI by default. Without any code changes you should be able to run Q-learning Pacman for very tiny grids by running `pacman.py` with the command options:
`-p PacmanQAgent -x 2000 -n 2010 -l smallGrid`

Note that `PacmanQAgent` is already defined for you in terms of the QLearningAgent you've already written. PacmanQAgent is only different in that it has default learning parameters that are more effective for the Pacman problem (epsilon=0.05, alpha=0.2, gamma=0.8). If your QLearningAgent works for gridworld.py but does not seem to be learning a good policy for Pacman on smallGrid, it may be because your `getAction` and/or `computeActionFromQValues` methods do not in some cases properly consider unseen actions. In particular, because unseen actions have by definition a Q-value of zero, if all of the actions that have been seen have negative Q-values, an unseen action may be optimal. Beware of the `argmax` function from `util.Counter`!

Criteria for Success:
Use the autograder with q5 to check your code. If you want to watch 10 training games to see what is going on, run `pacman.py` with the command option:
`-p PacmanQAgent -n 10 -l smallGrid -a numTraining=10`

Once Pacman is done training, he should win very reliably in test games (at least 90% of the time), since now he is exploiting his learned policy. However, you will find that training the same agent on the seemingly simple mediumGrid does not work well. In my implementation, Pacman's average training rewards remain negative throughout training. At test time, he plays badly, probably losing all of his test games. Training will also take a long time, despite its ineffectiveness.

Pacman fails to win on larger layouts because each board configuration is a separate state with separate Q-values. He has no way to generalize that running into a ghost is bad for all positions. Obviously, this approach will not scale.

Assignment 6 – Approximate Q-Learning:

Implement an approximate Q-learning agent that learns weights for features of states, where many states might share the same features. Write your implementation in the `ApproximateQAgent` class in `qlearningAgents.py`, which is a subclass of `PacmanQAgent`.

Approximate Q-learning assumes the existence of a feature function $f(s, a)$ over state and action pairs, which yields a vector $f_1(s, a)..f_i(s, a)..f_n(s, a)$ of feature values. Feature functions are provided for you in `featureExtractors.py`. Feature vectors are `util.Counter` (like a dictionary) objects containing the non-zero pairs of features and values; all omitted features have value zero.

The approximate Q-function takes the following form

$$Q(s, a) = \sum_{i=1}^{n} f_i(s, a) w_i$$

where each weight w_i is associated with a particular feature $f_i(s, a)$. In your code, you should implement the weight vector as a dictionary mapping features (which the feature extractors will return) to weight values. You will update your weight vectors similarly to how you updated Q-values:

$$w_i \leftarrow w_i + \alpha \cdot difference \cdot f_i(s, a)$$
$$difference \leftarrow (r + \gamma \max_{a'} Q(s', a')) - Q(s, a)$$

Note that the $difference$ term is the same as in normal Q-learning, and r is the experienced reward.

`ApproximateQAgent` is a subclass of `QLearningAgent`, and it therefore shares several methods like `getAction`. Make sure that your methods in QLearningAgent call `getQValue` instead of accessing Q-values directly, so that when you override `getQValue` in your approximate agent, the new approximate q-values are used to compute actions.

Criteria for Success:
By default, `ApproximateQAgent` uses the IdentityExtractor, which assigns a single feature to every (state,action) pair. With this feature extractor, your approximate Q-learning agent should work identically to PacmanQAgent. You may test this by running `pacman.py` with the command options:
`-p ApproximateQAgent -x 2000 -n 2010 -l smallGrid`

Once you're confident that your approximate learner works correctly with the identity features, run your approximate Q-learning agent with the custom feature extractor, which can learn to win with ease:
`-p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l mediumGrid`

Even much larger layouts should be no problem for your ApproximateQAgent. (warning: this may take a few minutes to train)
`-p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l mediumClassic`

If you have no errors, your approximate Q-learning agent should win almost every time with these simple features, even with only 50 training games.

The autograder with q6 will run your approximate Q-learning agent and check that it learns the same Q-values and feature weights as the reference implementation when each is presented with the same set of examples.

Submit your code in Canvas for grading.