Week 9 Homework

Bookmark this page

Week 9 Homework

(1.0 points possible)

A code directory that may be useful for doing this homework can be found here. You should implement your code for Problem 1 (State Machines) in `sm.py` and Problem 5 (MDP Implementations) in `mdp.py`. There are unit tests available for you run in `tests.py`. Download this to your computer, or alternatively, use the colab notebook. The colab notebook will also contain unit tests defined after each code question.

1) State Machines

We will implement state machines as sub-classes of the `SM` class, which specifies the `start_state`, `transition_fn`, and `output_fn`.

```
class SM:
    start_state = None  # default start state

    def transition_fn(self, s, x):
        '''s:       the current state
           x:       the given input
           returns: the next state'''
        raise NotImplementedError

    def output_fn(self, s):
        '''s:       the current state
           returns: the corresponding output'''
        raise NotImplementedError

    def transduce(self, input_seq):
        '''input_seq: the given list of inputs
           returns: list of outputs given the inputs'''
        raise NotImplementedError
```

An example of a sub-class is the `Accumulator` state machine, which adds up (accumulates) its input and outputs the sum. Convince yourself that the implementation works as expected before moving on.

```
class Accumulator(SM):
    start_state = 0

    def transition_fn(self, s, x):
        return s + x

    def output_fn(self, s):
        return s
```

An example case of using `Accumulator` (with `transduce`)

```
sm = Accumulator()

# Output should = [-1, 1, 4, 2, 7, 13]
output = sm.transduce([-1, 2, 3, -2, 5, 6])
```

You can find these definitions in the `sm.py` file.

1.1) Transduce

Implement the `transduce` method for the `SM` class. It is given an input sequence (a list) and returns an output sequence (a list) of the outputs of the state machine on the input sequence. Assume `self.transition_fn` and `self.output_fn` are defined.

```
class SM:
    start_state = None

    def transduce(self, input_seq):
        '''input_seq: a list of inputs to feed into SM
           returns:   a list of outputs of SM'''
        outputs = []
        cur_state = self.start_state
        for input in input_seq:
            cur_state = self.transition_fn(cur_state, input)
            outputs.append(self.output_fn(cur_state))
        return outputs

```

Run Code Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

Your score on your most recent submission was: 100.00%

Show/Hide Detailed Results

1.2) Binary Addition

Implement a `Binary_Addition` state machine that takes in a sequence of pairs of binary digits (0,1) representing two **reversed** binary numbers and returns a sequence of digits representing the reversed sum. For instance, to sum two binary numbers `100` and `011`, the input sequence will be `[(0, 1), (0, 1), (1, 0)]`. You will need to define `start_state`, `transition_fn`, and `output_fn`. Note that when transduced, the input sequence may need to be extended with an extra (0,0) to output the final carry. The inputs in the test cases have already been extended with the extra (0, 0).

An example case of using `Binary_Addition` is

```
sm = Binary_Addition()

# Output should = [0, 0, 1]
output = sm.transduce([(1, 1), (1, 0), (0, 0)])
```

```
class Binary_Addition(SM):
    start_state = [0, 0] # Change

    def transition_fn(self, s, x): # x: (1, 1)
        # Your code here
        return [int(((x[0] + x[1]) + s[1]) % 2), int(((x[0] + x[1]) + s[1]) / 2)]

    def output_fn(self, s):
        # Your code here
        return s[0]

```

Run Code Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

Your score on your most recent submission was: 100.00%

Show/Hide Detailed Results

1.3) Reverser

Implement a state machine that reverses a sequence. The input is a list of the form:

```
sequence1 + ['end'] + sequence2
```

`+` refers to concatenation. `sequence1` is a list of strings, the `'end'` string indicates termination, and `sequence2` is arbitrary. The machine reverses `sequence1`: for each entry in the `sequence1`, the machine outputs `None`. For the `'end'` input and each entry in the second sequence, an item from the reversed `sequence1` is output, or `None` if no characters remain.

To view test cases, try submitting your code. An example case of using `Reverser`

```
sm = Binary_Addition()

# Output should = [None, None, None, 'bar', ' ', 'foo', None, None, None]
output = sm.transduce(['foo', ' ', 'bar'] + ['end'] + list(range(5)))
```

```
class Reverser(SM):
    start_state = 0 # stage
    reversed_seq1 = []
    reversed_seq1_id = 0

    def transition_fn(self, s, x): # x: a string
        # Your code here
        if x == 'end':
            s = 1
            return s
        if s == 0:
            self.reversed_seq1.insert(0, x)
            return s
        if s == 2:
            return s
        if self.reversed_seq1_id == len(self.reversed_seq1):
            return 2



    def output_fn(self, s):
        # Your code here
```

Run Code Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

Your score on your most recent submission was: 100.00%

Show/Hide Detailed Results

1.4) RNN

A neural network is a (learned) function that maps input vectors to output vectors. A **recurrent** neural network (RNN) is a (learned) state machine that maps input sequences to output sequences.

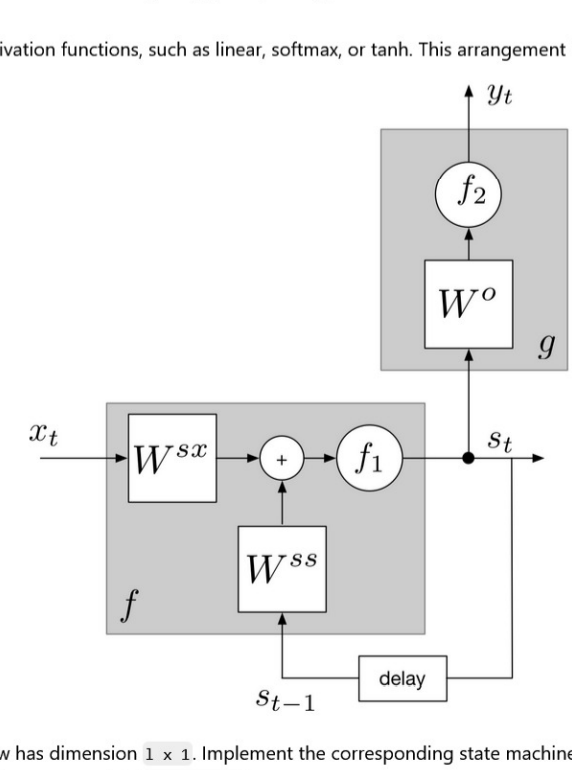
In particular, an RNN has a transition function and an output function, each of which is defined in terms of weight matrices, offset vectors and activation functions, analogously to standard neural networks.

- The inputs x are $l \times 1$ vectors
- The states s are $m \times 1$ vectors
- The outputs y are $n \times 1$ vectors

The behavior is defined as follows:

$$s_t = f_1(W^{ss}s_{t-1} + W^{sx}x_t + W_0^{ss})$$
$$y_t = f_2(W^{o}s_t + W_0^{o})$$

where f_1 and f_2 are two activation functions, such as linear, softmax, or tanh. This arrangement is shown below.



Note that each input `i` below has dimension `1 x 1`. Implement the corresponding state machine, where the weights are given in `__init__`. Make sure to set an appropriate `start_state`.

```
class RNN(SM):

    def __init__(self, Wsx, Wss, Wo, Wss_0, Wo_0, f1, f2):
        # Your code here
        self.Wss = Wss
        self.Wsx = Wsx
        self.Wss_0 = Wss_0
        self.Wo = Wo
        self.Wo_0 = Wo_0
        self.f1 = f1
        self.f2 = f2
        self.start_state = np.zeros((Wss_0.shape[0], 1))

    def transition_fn(self, s, i):
        # Your code here
        return self.f1(self.Wss @ s + self.Wsx @ i + self.Wss_0)

    def output_fn(self, s):
        # Your code here
        return self.f2(self.Wo @ s + self.Wo_0)
```

Run Code Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

Your score on your most recent submission was: 100.00%

Show/Hide Detailed Results

1.5) Accumulator Sign RNN

Enter the parameter matrices and vectors for an instance of the `RNN` class such that the output is 1 if the cumulative sum of the inputs is positive, -1 if the cumulative sum is negative and 0 if otherwise. Make sure that you scale the outputs so that the output activation values are very close to 1, 0 and -1. Note that both the inputs and outputs are 1x1.

Hint: `np.tanh` may be useful. Remember to convert your python lists to `np.array`.

```
Wsx =    np.array([[1]])
Wss =    np.array([[1]])
Wo =     np.array([[100]])
Wss_0 =  np.array([[0]])
Wo_0 =   np.array([[0]])
f1 =     lambda x : x
f2 =     np.tanh
acc_sign = RNN(Wsx, Wss, Wo, Wss_0, Wo_0, f1, f2)
```

Run Code Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

Your score on your most recent submission was: 100.00%

Show/Hide Detailed Results

1.6) Autoregressive RNN

Enter the parameter matrices and vectors for an instance of the `RNN` class such that it implements the following autoregressive model:

$$y_t = 1y_{t-1} - 2y_{t-2} + 3y_{t-3}$$

when $x_t = y_{t-1}$.

Note: unlike in the lab, as a result of your call to `RNN`, your initial (start) state vector in the RNN will be initialized to all zeros (i.e., our `RNN` implementation enforces this requirement). Instead, you should assume that the initial value for x will be provided, e.g., $x_1 = 5$ in the lab example, and that each successive $x_t = y_{t-1}$ is also provided to the `transduce` method.

```
Wsx =    np.array([[1], [0], [0]])
Wss =    np.array([[0, 0, 0],
                   [1, 0, 0],
                   [0, 1, 0]])
Wo =     np.array([[1, -2, 3]]) # shape=(1, 3)
Wss_0 =  np.array([[0], [0], [0]])
Wo_0 =   np.array([[0]])
f1 =     lambda x: x
f2 =     lambda x: x
auto = RNN(Wsx, Wss, Wo, Wss_0, Wo_0, f1, f2)
```

Run Code Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

Your score on your most recent submission was: 100.00%

Show/Hide Detailed Results

2) MDP

We consider the same tiny MDP as in the exercises, with states (0, 1, 2, 3) and actions ('b', 'c'). The reward function is:

$$R(s,a) = \begin{cases} 1 & \text{if } s = 1 \\ 2 & \text{if } s = 3 \\ 0 & \text{otherwise} \end{cases}$$

You get the reward associated with a state on the step when you exit that state. The transition function for each action is below, where $T[i, x, j]$ is the $P(s_{t+1} = j | a = x, s_t = i)$ (note rows correspond to the input states, and columns correspond to the output states).

$$T(s_t, \text{'b'}, s_{t+1}) = \begin{bmatrix} 0.0 & 0.9 & 0.1 & 0.0 \\ 0.9 & 0.1 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.1 & 0.9 \\ 0.9 & 0.0 & 0.0 & 0.1 \end{bmatrix}$$

$$T(s_t, \text{'c'}, s_{t+1}) = \begin{bmatrix} 0.0 & 0.1 & 0.9 & 0.0 \\ 0.9 & 0.1 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.1 & 0.9 \\ 0.9 & 0.0 & 0.0 & 0.1 \end{bmatrix}$$

Note that the only effect of the action is to change the transition probability from state 0.

2.1) Limited horizons

Consider two policies: one that always takes action 'b' in state 0 and one that always takes action 'c'.

2.1.A) Which policy is best if you're starting from state 0 with horizon 2?

Taking action 'b'

Ask for Help 100.00%

You have infinitely many submissions remaining.

Solution: Taking action 'b'

Explanation:

We compare the values from each of the policies.

Horizon 2 values from taking action 'b': $[0 + (0.9 \cdot 1), 1 + (0.1 \cdot 1), 0 + (0.9 \cdot 2), 2 + (0.1 \cdot 2)] = [0.9, 1.1, 1.8, 2.2]$

Horizon 2 values from taking action 'c': $[0 + (0.1 \cdot 1), 1 + (0.1 \cdot 1), 0 + (0.9 \cdot 2), 2 + (0.1 \cdot 2)] = [0.1, 1.1, 1.8, 2.2]$

Since the $0.9 > 0.1$, taking action 'b' is the better policy.

2.1.B) Which policy is best if you're starting from state 0 with horizon 3?

Taking action 'c'

Ask for Help 100.00%

You have infinitely many submissions remaining.

Solution: Taking action 'c'

Explanation:

Now, we compute the best policy from starting at state 0, building off of the values found in question 1.

Expected value from taking action 'b': $0 + (0.9 \cdot 1.1) + (0.1 \cdot 1.8) = 1.17$

Expected value from taking action 'c': $0 + (0.1 \cdot 1.1) + (0.9 \cdot 1.8) = 1.73$

Since $1.73 > 1.17$, the best policy for starting from state 0 with horizon 3 is taking action 'c'.

2.1.C) What if we start in state 0 with horizon 5, take action 'c', land in state 2 with horizon 4, land in state 3 with horizon 3 (and get reward = 2!), and then land in state 0 with horizon 2? What action should we take now?

Action 'b'

Ask for Help 100.00%

You have infinitely many submissions remaining.

Solution: Action 'b'

Explanation:

The previous actions have already yielded their rewards. Now that we're in state 0 with horizon two, the best action is 'b', as we have previously determined.

2.2) At a discount

Now, in the same tiny MDP, we are interested in the infinite-horizon discounted value, with discount factor 0.9. For a given policy, we can write down a set of linear equations characterizing these values, in the form:

$$v_0 = r_0 + c_{00}v_0 + c_{01}v_1 + c_{02}v_2 + c_{03}v_3$$
$$v_1 = r_1 + c_{10}v_0 + c_{11}v_1 + c_{12}v_2 + c_{13}v_3$$
$$v_2 = r_2 + c_{20}v_0 + c_{21}v_1 + c_{22}v_2 + c_{23}v_3$$
$$v_3 = r_3 + c_{30}v_0 + c_{31}v_1 + c_{32}v_2 + c_{33}v_3$$

2.2.A) Enter the vector of r_i values, in the order shown above.

Enter the matrix as a list of 4 numbers: [0, 1, 0, 2]

Ask for Help 100.00%

You have infinitely many submissions remaining.

Solution: `[0, 1, 0, 2]`

Explanation:

Thinking of v as a vector, we can write it as

$$v = r + Cv$$

. Now, the only constant term guaranteed is the reward at each state (i.e. that is independent of the values at other states), while the rest of the value is tied to values of other states. Thus, the constant part should be equal to the reward.

2.2.B) Enter the matrix of c_{ij} values, in the order shown above, for the policy "always take action 'c'".

Enter the matrix as a list of 4 lists of numbers: [[0.0, 0.09, 0.81, 0.0], [0.81, 0.09, 0.0, 0.0], [0.0, 0.0

Ask for Help 100.00%

You have infinitely many submissions remaining.

Solution: `[[0.0, 0.09, 0.81, 0.0], [0.81, 0.09, 0.0, 0.0], [0.0, 0.0, 0.09, 0.81], [0.81, 0.0, 0.0, 0.09]]`

Explanation:

Consider v_0, in other words the value from starting in state 0. We know that after the first action, we are guaranteed the reward defined by the reward function, 0. For future steps, it depends on where we transitioned to. Thus, the coefficients on the right hand side of the equation for each of the values v_0 thru v_3 are the discount factor times probability of transitioning to each of those states. The values for r_i are the immediate rewards at each state.

$$v_0 = 0 + 0.9(0v_0 + 0.1v_1 + 0.9v_2 + 0v_3)$$
$$v_1 = 1 + 0.9(0.9v_0 + 0.1v_1 + 0v_2 + 0v_3)$$
$$v_2 = 0 + 0.9(0v_0 + 0v_1 + 0.1v_2 + 0.9v_3)$$
$$v_3 = 2 + 0.9(0.9v_0 + 0v_1 + 0v_2 + 0.1v_3)$$

Notice that you can also think of v as the one steady solution to the equation

$$v_t = r + \gamma T v_{t-1}$$

i.e. if you have value v, then applying the transition linear operation over the value distribution, scaling it by γ and adding the constant reward gives you the same values as before.

Solve for the value function under this policy by using `numpy.linalg.solve` to solve these equations. For example, if you have a set of linear equations of the form:

```
1 v0 + 2  v1 + 3  v2 = 4
5 v0 + 6  v1 + 7  v2 = 8
9 v0 + 10 v1 + 11 v2 = 12
```

Then you can solve as follows:

```
A = np.matrix([[1, 2, 3], [5, 6, 7], [9, 10, 11]])
b = np.matrix([[4], [8], [12]])
v = np.linalg.solve(A,b)  # A v = b
```

2.2.C)

Enter a Python list of 4 numbers, accurate to at least 3 decimal places: [6.053, 6.487, 6.752, 7.586]

Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

3) Tiny Q-value iteration

In the same example as above, with an infinite horizon and a discount factor of 0.9, compute three iterations of value iteration. Don't assume a particular policy. Assume that:

- All the value estimates start at 0 (meaning, at iteration 0, $Q(s, a) = 0$ for all s, a pairs), and
- You operate synchronously (that is, on iteration t of value iteration, you only use values that were computed on iteration $t - 1$).

We recommend you compute the Q-value iteration by hand to get a better understanding of the algorithm.

For each iteration enter 8 numbers corresponding to:

$$[Q(0,b), Q(0,c), Q(1,b), Q(1,c), Q(2,b), Q(2,c), Q(3,b), Q(3,c)]$$

at that iteration, accurate to at least 3 decimal places.

3.A) Iteration 1

Enter a Python list of 8 numbers: [0, 0, 1, 1, 0, 0, 2, 2]

Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

3.B) Iteration 2

Enter a Python list of 8 numbers: [0.81, 0.09, 1.09, 1.09, 1.62, 1.62, 2.18, 2.18]

Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

3.C) Iteration 3

Enter a Python list of 8 numbers: [1.0287, 1.4103, 1.7542, 1.7542, 1.9116, 1.9116, 2.8

Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

3.D) After the third iteration, what action would you select in state 0?

Action 'c'

Ask for Help 100.00%

You have infinitely many submissions remaining.

Solution: Action 'c'

Explanation:

Q(0,'c') > Q(0,'b')

4) One D Grid

Consider a domain where states correspond to squares on a 1 by 10 grid. The state is the position of a robot on the board, i, which ranges from 1 to 10. The robot can move one square East or West. In this question, assume there is no randomness in the robot's motion. It cannot move off the board.

4.1) Negative Reward

There is a "goal" state i_g. Assume that the reward for taking any action from the goal state is 0, and that, once in the goal state, the robot "sticks" there forever. Further, assume the reward for all other actions is -1. H is the horizon.

4.1.A) If the goal is 10, $H = 5$ and $\gamma = 1$, what is $V^*(1)$?

Enter a number: -5

Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

4.1.B) If the goal is 10, $H = 15$ and $\gamma = 1$, what is $V^*(1)$?

Enter a number: -9

Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

4.1.C) If the goal is 10, $H = \infty$ and $\gamma = 1$, what is $V^*(1)$?

Enter a number: -9

Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

4.1.D) If the goal is 10 and $H = \infty$, what is $V^*(1)$ in terms of γ, if $0 < \gamma < 1$? Recall that $\sum_{i=0}^{n} \gamma^i = \frac{1-\gamma^{n+1}}{1-\gamma}$. You can use `gamma` in your expression.

$V^*(1) =$ -1 * (1 - gamma**9) / (

Check Syntax Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

Your entry was parsed as:

$$\frac{-1 \times (1 - \gamma^9)}{1 - \gamma}$$

4.1.E) Is there a value of $0 < \gamma < 1$ for which the optimal policy differs from the optimal policy for $\gamma = 0.9$?

No

Ask for Help 100.00%

You have infinitely many submissions remaining.

Solution: No

Explanation:

No matter the value of the positive fractional discount γ, the only way we can maximize reward is by heading towards the goal of 10, which will give us a nonnegative reward.

4.2) Positive Reward

Now, we will consider a different reward function: The reward for any action in the goal state is 1; reward for any action in any other state is 0; once in the goal state, the robot "sticks" there forever. H is the horizon.

4.2.A) If the goal is 10, $H = 5$ and $\gamma = 1$, what is $V^*(1)$?

Enter a number (or '+infty', '-infty'): 0

Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

4.2.B) If the goal is 10, $H = 15$ and $\gamma = 1$, what is $V^*(1)$?

Enter a number (or '+infty', '-infty'): 6

Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

4.2.C) If the goal is 10, $H = \infty$ and $\gamma = 1$, what is $V^*(1)$?

Enter a number (or '+infty', '-infty'): '+infty'

Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

4.2.D) If the goal is 10 and $H = \infty$, enter an expression for $V^*(1)$ in terms of γ, if $0 < \gamma < 1$? Recall that $\sum_{i=0}^{\infty} \gamma^i = \frac{1}{1-\gamma}$. You can use `gamma` in your expression.

$V^*(1) =$ gamma ** 9 / (1 - gam

Check Syntax Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

Your entry was parsed as:

$$\frac{\gamma^9}{1 - \gamma}$$

5) MDP Implementations

We'll be using a couple of simple classes to represent MDPs and probability distributions. The minimal API is shown below, the detailed definition can be found in the lab code file.

```
# Discrete distribution represented as a dictionary.  Can be
# sparse, in the sense that elements that are not explicitly
# contained in the dictionary are assumed to have zero probability.
class DDist:
    def __init__(self, dictionary):
        # Initializes dictionary whose keys are elements of the domain
        # and values are their probabilities

    def prob(self, elt):
        # Returns the probability associated with elt

    def support(self):
        # Returns a list (in any order) of the elements of this
        # distribution with non-zero probabability.

    def draw(self):
        # Returns a randomly drawn element from the distribution

    def expectation(self, f):
        # Returns the expected value of the function f over the current distribution

def uniform_dist(elts):
    # Returns DDist with uniform distribution over a given finite set of elts

def argmax(l, f):
    # l is a list of items. f is a procedure that maps an item into a numeric score
    # returns the element of l that has the highest score

class MDP:
    def __init__(self, states, actions, transition_model, reward_fn,
                     discount_factor = 1.0, start_dist = None):
        # states: list or set of states
        # actions: list or set of actions
        # transition_model: function from (state, action) into DDist over next state
        # reward_fn: function from (state, action) to real-valued reward
        # discount_factor: real, greater than 0, less than or equal to 1
        # start_dist: optional instance of DDist, specifying initial state dist
        #    if it's unspecified, we'll use a uniform over states

    def terminal(self, s):
        # Given a state, return True if the state should be considered to
        # be terminal.  You can think of a terminal state as generating an
        # infinite sequence of zero reward.

    def init_state(self):
        # Return an initial state by drawing from the distribution over start states.

    def sim_transition(self, s, a):
        # Simulates a transition from the given state, s and action a, using the
        # transition model as a probability distribution.  If s is terminal,
        # use init_state to draw an initial state.  Returns (reward, new_state)
```

We will be using a class to store the Q function, for now this is basically a dictionary mapping `(s, a)` pairs into their Q values. In the homework next week we will generalize this to use neural nets.

```
class TabularQ:
    def __init__(self, states, actions):
        self.actions = actions
        self.states = states
        self.q = dict([((s, a), 0.0) for s in states for a in actions])

    def copy(self):
        q_copy = TabularQ(self.states, self.actions)
        q_copy.q.update(self.q)
        return q_copy

    def set(self, s, a, v):
        self.q[(s,a)] = v

    def get(self, s, a):
        return self.q[(s,a)]
```

You can find these definitions in the files `dist.py`, `util.py` and `mdp.py`.

5.1) Working with MDPs

Recall that given a Q_π for any policy π, then $V_\pi(s) = \max_a Q_\pi(s, a)$.

5.1.A) Write the `value` method, which takes a Q function (an instance of TabularQ) and a state and returns the value $Q^*(s, a)$ of an action that maximizes Q function stored in `q`.

```
def value(q, s):
    """ Return Q*(s,a) based on current Q

    >>> q = TabularQ([0,1,2,3],['b','c'])
    >>> q.set(0, 'b', 5)
    >>> q.set(0, 'c', 10)
    >>> q_star = value(q,0)
    >>> q_star
    10
    """
    max_q = -1
    for a in q.actions:
        if q.get(s, a) > max_q:
            max_q = q.get(s, a)
    return max_q

```

Run Code Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

Your score on your most recent submission was: 100.00%

Show/Hide Detailed Results

5.1.B) Write the `greedy` method, which takes a Q function (an instance of TabularQ) and a state and returns the action a determined by the policy that acts *greedily* with respect to the current value of `q`.

```
def greedy(q, s):
    """ Return pi*(s) based on a greedy strategy.

    >>> q = TabularQ([0,1,2,3],['b','c'])
    >>> q.set(0, 'b', 5)
    >>> q.set(0, 'c', 10)
    >>> q.set(1, 'b', 2)
    >>> greedy(q, 0)
    'c'
    >>> greedy(q, 1)
    'b'
    """
    # Your code here
    max_q = -1
    max_a = ''
    for a in q.actions:
        if q.get(s, a) > max_q:
            max_q = q.get(s, a)
            max_a = a
    return max_a

```

Run Code Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

Your score on your most recent submission was: 100.00%

Show/Hide Detailed Results

5.1.C) Write the `epsilon_greedy` method, which takes a state `s` and a parameter `epsilon`, and returns an action. With probability `1 - epsilon` it should select the greedy action and with probability `epsilon` it should select an action uniformly from the set of possible actions.

- You should use `random.random()` to generate a random number to test against eps.
- You should use the `draw` method of `uniform_dist` to generate a random action.
- You can use the `greedy` function defined earlier.

```
def epsilon_greedy(q, s, eps = 0.5):
    """ Returns an action.

    >>> q = TabularQ([0,1,2,3],['b','c'])
    >>> q.set(0, 'b', 5)
    >>> q.set(0, 'c', 10)
    >>> q.set(1, 'b', 2)
    >>> eps = 0.
    >>> epsilon_greedy(q, 0, eps) #greedy
    'c'
    >>> epsilon_greedy(q, 1, eps) #greedy
    'b'
    """
    if random.random() < eps:  # True with prob eps, random action
        # Your code here
        return uniform_dist(q.actions).draw()
    else:
        # Your code here
        return greedy(q, s)

```

Run Code Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

Your score on your most recent submission was: 100.00%

Show/Hide Detailed Results

5.2) Implement Q-Value Iteration

Provide the definition of the `value_iteration` function. It takes an `MDP` instance and a `TabularQ` instance. It should terminate when

$$\max_{(s,a)} |Q_t(s,a) - Q_{t-1}(s,a)| < \epsilon$$

that is, the biggest difference between the value functions on successive iterations is less than input parameter `eps`. This function should return the final TabularQ instance. It should do no more that `max_iters` iterations.

The pseudocode for this algorithm is in the lecture notes. Read the API specification at the beginning of this problem to use the `MDP` instance `mdp` and `TabularQ` instance `q`.

- **Make sure to copy the Q function between iterations, e.g. `new_q = q.copy()`.**
- The `q` parameter contains the initialization of the Q function.
- The `value` function is already defined.
- Use `mdp` class definitions to get the reward function, discount factor, transition model, and expectation of the Q-values over a distribution.

```
def value_iteration(mdp, q, eps = 0.01, max_iters=1000):
    # Your code here
    def v(s):
        return value(q,s)
    for it in range(max_iters):
        new_q = q.copy()
        delta = 0
        for s in mdp.states:
            for a in mdp.actions:
                new_q.set(s, a, mdp.reward_fn(s, a) + mdp.discount_factor * \
                          mdp.transition_model(s, a).expectation(v))
                delta = max(delta, abs(new_q.get(s, a) - q.get(s, a)))
        if delta < eps:
            return new_q
        q = new_q
    return q
```

Run Code Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

Your score on your most recent submission was: 100.00%

Show/Hide Detailed Results

5.3) Receding-horizon control and online search

Value iteration depends on the dynamic programming principle to efficiently solve for a value function over all the states in the domain, which allows very fast determination of the optimal action for each state. In some cases, the state space is huge, but the set of states that are reachable in a reasonable number of steps from the current state is relatively small. In that case, it's actually more efficient to select actions by performing an "expectimax" tree search from the current state to find the best action, taking the action, seeing what state results, and performing another tree search.

We will need to fix the horizon for the tree search; it is possible to choose a finite horizon that will guarantee bounded "regret" (optimal value minus value actually obtained) based on the discount factor, but for this problem we will just assume we are given horizon `h`.

Consider a function $q_{em}(s, a, h)$ that computes the horizon-h Q value for state s and action a by using the definition of the finite-horizon Q function in the notes (**but including a discount factor**). This is sometimes called "expectimax" search, because you can think of it as building a search tree, in which some nodes are evaluated by taking an expectation (we take an expectation over possible outcomes of an action in a state) and some nodes are evaluated by taking a maximum (we take a maximum over the Q values for the possible actions in a state).

Consider again our "tiny" MDP as in Section 2 with states (0, 1, 2, 3) and actions ('b', 'c'). Assume no discounting (i.e., discount factor = 1.0). The reward function is:

$$R(s,a) = \begin{cases} 1 & \text{if } s = 1 \\ 2 & \text{if } s = 3 \\ 0 & \text{otherwise} \end{cases}$$

You get the reward associated with a state on the step when you exit that state. The transition function for each action is below, where $T[i, x, j]$ is the $P(s_{t+1} = j | a = x, s_t = i)$.

$$T(s_t, \text{'b'}, s_{t+1}) = \begin{bmatrix} 0.0 & 0.9 & 0.1 & 0.0 \\ 0.9 & 0.1 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.1 & 0.9 \\ 0.9 & 0.0 & 0.0 & 0.1 \end{bmatrix}$$

$$T(s_t, \text{'c'}, s_{t+1}) = \begin{bmatrix} 0.0 & 0.1 & 0.9 & 0.0 \\ 0.9 & 0.1 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.1 & 0.9 \\ 0.9 & 0.0 & 0.0 & 0.1 \end{bmatrix}$$

Note that the only effect of the action is to change the transition probability in state 0.

5.3.A) What is $q_{em}(0, \text{'b'}, 0)$?

Enter a number: 0

Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

5.3.B) What is $q_{em}(0, \text{'b'}, 1)$?

Enter a number: 0

Ask for Help 100.00%

You have infinitely many submissions remaining.

Solution: `0`

Explanation:

In this case, we transition out of state $s = 0$ to state $s = 1$ with probability 0.9, and to state $s = 2$ with probability 0.1 (and are then done). However, transitioning out of state $s = 0$ earned us $R = 0$ reward, for our reward function.

5.3.C) What is $q_{em}(0, \text{'b'}, 2)$?

Enter a number: 0.9

Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

5.3.D) Write the procedure `q_em(mdp, s, a, h)` that computes the horizon-`h` Q value for state `s` and action `a`. This can be written as a relatively simple recursive procedure with a base case (what is the Q value when horizon is 0?) and a recursive case that computes the horizon-h values assuming we can (recursively) get horizon h-1 values.

```
f q_em(mdp, s, a, h):
# Your code here
if h == 0:
    return 0
next_q_sum = 0.0
for (next_s, next_prob) in mdp.transition_model(s, a).d.items():
    next_q_sum += next_prob * max([q_em(mdp, next_s, next_a, h - 1) for next_a in mdp.
return mdp.reward_fn(s, a) + mdp.discount_factor * next_q_sum
```

Run Code Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

Your score on your most recent submission was: 100.00%

Show/Hide Detailed Results

This page was last updated on Saturday January 18, 2020 at 01:16:11 PM (revision `043d9ae`).

Sec None Release 1900-01-01 00:00:00 Due 9999-12-31 23:59:59

Powered by CAT-SOOP v14.0.0.dev144.

CAT-SOOP is free/libre software, available under the terms of the GNU Affero General Public License, version 3.
Download Source Code
Javascript License Information

© All Rights Reserved

© Massachusetts Institute of Technology, 2022