Pacman – Part 2: Learning

Welcome to the second part of the project for building a Pacman AI. In this installment, you will build two reinforcement learning algorithms (*value iteration* and *Q-learning*), and you will test them on Gridworld, Crawler, and Pacman. Detailed instructions on how to do this are provided in Section 1 of the present document. Before getting started, however, it is highly recommended that you read the background material in Section 2, which gives a short introduction to the concepts needed for understanding this project.

Pacman, now with ghosts.
Should he eat or should he run?

1 Reinforcement learning – Implementation

For starters, run the command below to play Gridworld with the arrow keys. You will see a maze with two exits marked by +1 and -1, where the agent is the blue dot, and its goal is to reach an exit. When you press an arrow key, the agent only actually moves into the desired direction 80% of the time (use `-n` to change the probability of going the wrong way). Such is the life of a Gridworld agent!

```
python -m gridworld
```

The default agent moves randomly. Run the following command to see the random agent bounce around the grid until it happens upon an exit. Not the finest hour for an AI agent.

```
python -m gridworld -a random
```

Look at the console output that accompanies the graphical output. You will be told about each transition the agent experiences (use `-q` to turn it off). As in Pacman, positions are represented by (x,y)-coordinates, with "north" being the direction of increasing the y-coordinate, "right" being the direction of increasing the x-coordinate, and so forth. By default, transitions to non-terminal states get a reward of 0 (use `-r` to change it), and the expected return is computed with a discount rate of 0.9 (use `-d` to change it).[1] Moreover, you can change the maze with the option `-g` (possible choices are Cliff, Cliff2, Discount, Bridge, Book, Maze).

The provided project consists of several files, the majority of which you can ignore, except for a few ones you might want to read and understand at some later time. In `learning.py`, you will find several fully implemented agents, which estimate the Q-values of all state-action pairs admissible in the world, and then execute the resulting policy step-by-step. The algorithms for computing the Q-values are not implemented: **that's your job**! Specifically, all of your algorithms need to return a dictionary (indexed by states) of nested dictionaries (indexed by actions): this is called a **Q-table**.

[1] The Gridworld MDP is such that you first must enter a pre-terminal state (the double boxes shown in the GUI), and then take the special 'exit' action before the episode actually ends (which is not shown in the GUI). If you run an episode manually, your expected return (i.e., the total sum of rewards) may be less than you expected, due to the discount rate.

1.1 Value iteration

In this exercise, you are going to work on the file `ex08.py`. Your job is to implement the algorithm called **value iteration**, which is described in Section 2.3, and whose pseudocode is given below. This algorithm takes three inputs: an MDP describing the states and the actions of the world, the number of iterations it should run (use `-i` to change it), and the discount rate. Value iteration is then used by `ValueIterationAgent`, an offline planner already implemented for you.

- **Inputs**: *MDP, iterations, discount*
- Create the **Q** table as a dictionary of nested dictionaries *> Use Python "defaultdict"*
- Create the **V** table as a dictionary *> Use Python "defaultdict"*
- **Repeat** for the given number of iterations
  1. **For each** *state* in the MDP
     a) **If** *state* is terminal, **then V**[*state*] = 0
     b) **Else V**[*state*] = max of **Q**[*state*][*action*] **for each** *action* available in *state*
  2. **For each** *state* in the MDP
     a) **For each** *action* available in *state*
        i. *q* = 0
        ii. **For each** *successor* of (*state, action*) pair
            - *R* = reward of (*state*, *action*, *successor*) triplet
            - *q* **+=** probability of *successor* * (*R* + discount * **V**[*successor*])
        iii. **Q**[*state*][*action*] = *q*

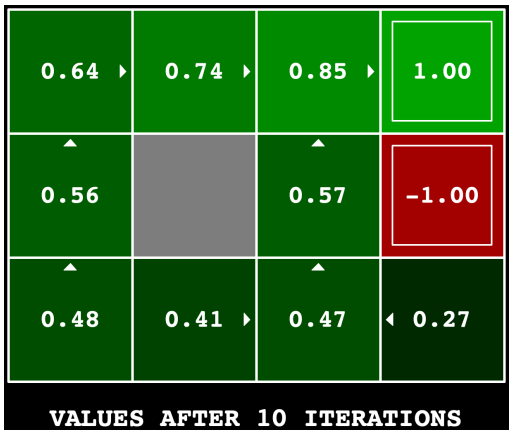
Here are some tips to help you with the implementation.

- The **Q**-table is indexed by states and actions, while the **V**-table is indexed by states alone.
- **V**[*state*] is equal to the max of **Q**[*state*][a_1], **Q**[*state*][a_2], …, **Q**[*state*][a_n], where $\mathbf{a_1}$, $\mathbf{a_2}$, …, $\mathbf{a_n}$ are the actions available in a given *state* of the MDP. You may need a loop to compute this.
- Update the **Q**-table **only after** you computed the **V**-table. Don't merge the loops 1 & 2.

When you are finished, run the following command to launch GridWorld.

```
python -m gridworld -a value
```

In the GridWorld GUI, press a key to cycle through the **V**-table (optimal value for each state), the **Q**-table (optimal value for each state-action pair), and the simulation (where the agent follows the optimal policy). On the default "BookGrid", value iteration should give you the following output.



1.2 Analysis of GridWorld MDP: uncertainty, reward, discount

Optimal policies in GridWorld are affected by three parameters:

1. **Probability of going in the wrong direction**, which can be modified with the **`-n`** option;
2. **Reward of non-terminal actions**, which can be changed with the **`-r`** option;
3. **Discount of long-term rewards**, which can be altered with the **`-d`** option.

Let us analyze the "Bridge" world map, which presents a low-reward terminal state (on the left) and a high-reward terminal state (on the right) 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 the default parameters (reward=0, discount=0.9, noise=0.2), the optimal policy does not cross the bridge, because there is a large enough probability of "falling off" and getting a negative reward (-100). Hence, the average return is maximized by safely reaching the closest exit. Run the following command to compute the optimal policy, and execute it for 10 times. The average return is -17.28, because the agent falls off the bridge a few times.

```
python -m gridworld -a value -i 100 -k 10 -g Bridge
```

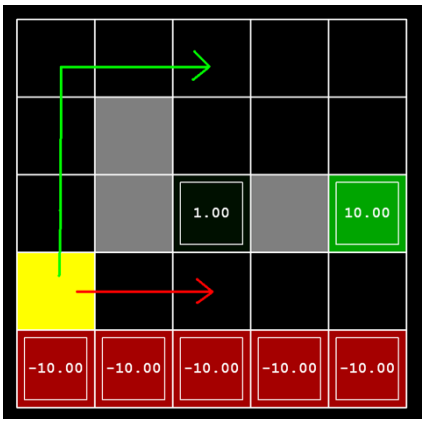
- By reducing the chance of going in the wrong direction (noise=0.01), the optimal policy causes the agent to attempt to cross the bridge. Indeed, as the probability of falling off the bridge is very low, the average return is maximized by going to the high-reward exit. Run the following command to compute the optimal policy, and execute it for 10 times. The average return is 2, due to the default discount of 0.9.

```
python -m gridworld -a value -i 100 -k 10 -g Bridge -n 0.01
```

Now, let's consider the "Discount" world map, which presents a close exit with payoff +1, a distant exit with payoff +10, and a "cliff" region with negative payoff -10; the starting state is the yellow square in the figure. We distinguish between two types of paths.

(1) Paths that **"risk the cliff"** by traveling near the bottom of the grid. They are shorter, but risk earning a large negative payoff. Those are represented by the red arrow in the figure.

(2) Paths that **"avoid the cliff"** by traveling along the top of the grid. They are longer, but are less likely to incur huge negative payoffs. Those are represented by the green arrow.



In this exercise, you are going to work on the file `ex09.py`. Your job is to choose the parameters (discount, noise, reward) for the "DiscountGrid" MDP to produce optimal policies of several types.

a) Policy that prefers the close exit (+1), risking the cliff (-10).
b) Policy that prefers the close exit (+1), but avoiding the cliff (-10).
c) Policy that prefers the distant exit (+10), risking the cliff (-10).
d) Policy that prefers the distant exit (+10), avoiding the cliff (-10).

Your setting should have the property that, if your agent followed its optimal policy without being subject to any noise, it would exhibit the given behavior. Run the command below with the desired parameters, and check your policies in the GUI. For example, using a correct answer to (a), the arrow in (0,1) should point east, the arrow in (1,1) should also point east, and in (2,1) should point north.

```
python -m gridworld -a value -i 100 -k 3 -g Discount -n ?? -d ?? -r ??
```

When you are finished, modify the function `mdp_parameter` so that it returns your parameters for each type of policy. Then, run the autograder to check the correctness of your work.

1.3 Q-learning

The 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. This distinction may be subtle in a simulated environment, but it's very important in the real world, where MDPs are rarely available.

In this exercise, you are going to work on the file `ex10.py`. Your job is to write a Q-learning agent, which does very little on construction, but instead learns by trial and error from interactions with the environment by sampling (state, action, next-state, reward) tuples. A stub of a Q-learner is specified in `QLearningAgent` class, and you must implement its methods as follows.

- **`__init__`** – The constructor creates the **Q-table** as an attribute of the class. Recall that the **Q**-table is a dictionary of nested dictionaries, which you can define exactly as in `ex08.py`. In python, you can create an attribute by writing `self.q_table = <initialization>`, and then you can reference it from any other class method by writing `self.q_table`.
- **`getQValue`** – It returns the Q-value of the state-action pair passed as input. *Note that pairs that your agent have not seen before still have a Q-value, specifically a Q-value of zero.*
- **`computeValueFromQValues`** – It returns the max Q-value over all the legal actions in the state passed as input. For states with no legal actions (e.g., terminal states), it returns zero. *Make sure that you only access Q-values by calling the method `getQValue`.*
- **`computeActionFromQValues`** – It returns the action corresponding to the max Q-value over all the legal actions in the state passed as input, or `None` if no legal action is available. *You should break ties randomly for better behavior (use the `random.choice()` function). Make sure that you only access Q-values by calling the method `getQValue`.*
- **`getAction`** – It returns the action to take in the state passed as input. Specifically, a coin is flipped to decide whether to explore by selecting a random action, or to exploit by taking the best action according to the current Q-values. Exploration is chosen with a probability indicated by the attribute `self.epsilon`, in which case an action is selected uniformly at random among all the legal actions. *Make sure to return `None` if no legal action is available.*
- **`update`** – It modifies a Q-value using the (state, action, next-state, reward) tuple passed as input, according to the formula described in Section 2.5 and recalled below.

$$\boldsymbol{Q}[\text{state}][\text{action}] = (1-\alpha)\,\boldsymbol{Q}[\text{state}][\text{action}] + \alpha\left(\text{reward} + \gamma \max_{\mathbf{a}} \boldsymbol{Q}[\text{next state}][\mathbf{a}]\right)$$

  The parameters α and γ are stored in the attributes `self.alpha` and `self.discount`.

With the Q-learning in place, you can watch your agent learn while being under your control. Run the following command to train the agent by yourself using the keyboard, over several episodes. Watch how the agent "leaves a trace of learning" on all the Q-values that are in its path to an exit.

```
python -m gridworld -a q -k 10 -m
```

Run the following command to train the agent without your intervention. Your final Q-values should resemble those of your value iteration agent, especially along well-traveled paths.

```
python -m gridworld -a q -k 100
```

When you are finished, run the autograder to check the correctness of your work.

1.4 Approximate Q-learning

Time to play some Pacman! Pacman will play games in two phases: training and testing. In the first phase, 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, where exploration is disabled so as 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 the following command, where the `PacmanQAgent` is defined for you in terms of the `QLearningAgent` you've already written.[2]

`python -m pacman -p PacmanQAgent -x 2000 -n 2010 -l smallGrid`

During training, you will see output every 100 games with statistics about how Pacman is faring. Exploration is enabled during training, so Pacman will play poorly even after having learned a good policy, because he occasionally makes a random exploratory move into a ghost. Make sure you understand what is happening here: *the MDP state is the board configuration facing Pacman, with the transitions describing all the possible future configurations after both Pacman and the ghost have moved*. Once Pacman is done training, he exploits his learned policy and wins at least 90% of test games. However, you will find that training the same agent on larger layouts does not work well, because he has no way to generalize that running into a ghost is bad for all positions!

In this exercise, you are going to work on the file `ex11.py`. Your job is to implement an approximate Q-learning agent that learns weights for state features, where many states might share the same features. You will write your implementation in `ApproximateQAgent` class. In this regard, recall that approximate Q-learning assumes the Q-values take the following form

$$Q(s,a) = w_1\, f_1(s,a) + \cdots + w_n\, f_n(s,a),$$

where each weight w_i is associated to a feature $f_i(s,a)$. The function for extracting the features is already implemented for you, and it returns a dictionary. *In your code, you should store the weights in a dictionary, using the same keys as the feature dictionary*. Then, you will update your weights similarly to how you updated a Q-value using a (s, a, s', r) tuple:

$$[\forall i \in \{1,\dots,n\}] \qquad w_i \leftarrow w_i + \alpha\, f_i(s,a)\left(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right).$$

By default, `ApproximateQAgent` uses `IdentityExtractor` as feature extractor, which makes it work identically to `PacmanQAgent`. You can test this by running the following command.[3]

`python -m pacman -p ApproximateQAgent -x 2000 -n 2010 -l smallGrid`

Once you're confident that your agent works correctly, use `SimpleExtractor` on a larger layout. If you have no errors, the agent should win almost every, even with only 50 training games.

`python -m pacman -p ApproximateQAgent -e Simple -x 50 -n 60 -l mediumClassic`

When you are finished, run the autograder to check the correctness of your work.

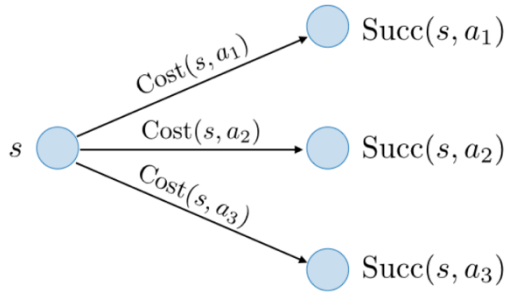
---

[2] If your Q-learning agent works for GridWorld, but it does not seem to be learning a good policy for Pacman on the "smallGrid", it may be because `getAction` and/or `computeActionFromQValues` methods in `QLearningAgent` class do not properly consider unseen actions. Beware of the argmax function from util.Counter if you used it in your code.

[3] `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.

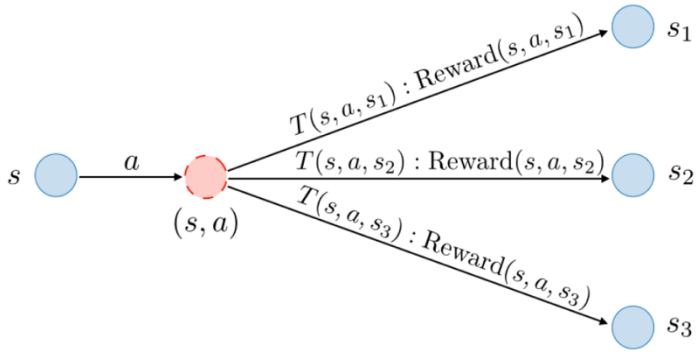
2 Reinforcement learning – Background

In the previous installment, we talked about **deterministic search problems**, and how they can be modeled as state-space graphs, where nodes represent search states and edges represent actions. Central to this model is the **successor function "Succ(s,a)"**, which determines the future state of the world after an action is executed in some state. This is depicted in the figure below.



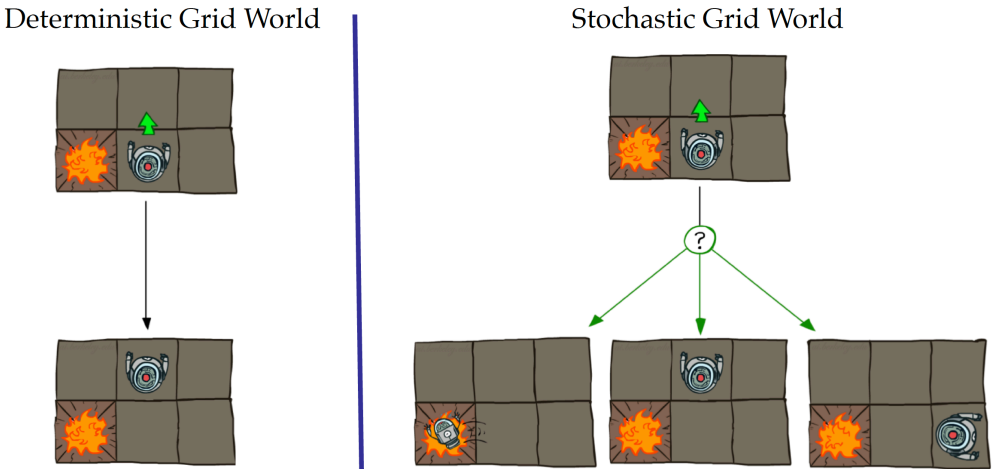
*In deterministic search problems, an action **"a"** executed in a state **"s"** always leads to the same future state **s' = succ(s,a)**. Each state-action pair is associated to a cost.*

However, graphs are not able to describe worlds where outcomes are partly random. This is, in fact, the case of many card games such as poker or blackjack, where there exists an inherent uncertainty from the randomness of card dealing. The problems where the world poses a degree of uncertainty are known as **stochastic search problems**. In such environments, the agent's actions are no longer deterministic, which means that there are multiple possible successor states that can result from an action taken in some state. This is illustrated in the figure below (more details later).



*In stochastic search problems, an action **"a"** executed in a state **"s"** leads to one of the possible future states **s'**, each occurring with probability **T(s,a,s')**. Every state-action-state triplet is associated to a reward.*

To illustrate the difference between deterministic and stochastic search problems, let us consider the example of GridWord. This is a problem where the agent lives in a maze with "bonuses" and "hazards". The agent's goal is to maximize the rewards cumulated by collecting the bonuses while avoiding the hazardous locations. The difficulty is that actions do not always go as planned: e.g., the action North takes the agent North 80% of the time, West 10% of the time, or East 10% of the time. The uncertainty in the movements prevents "graph search" algorithms from solving this problem.

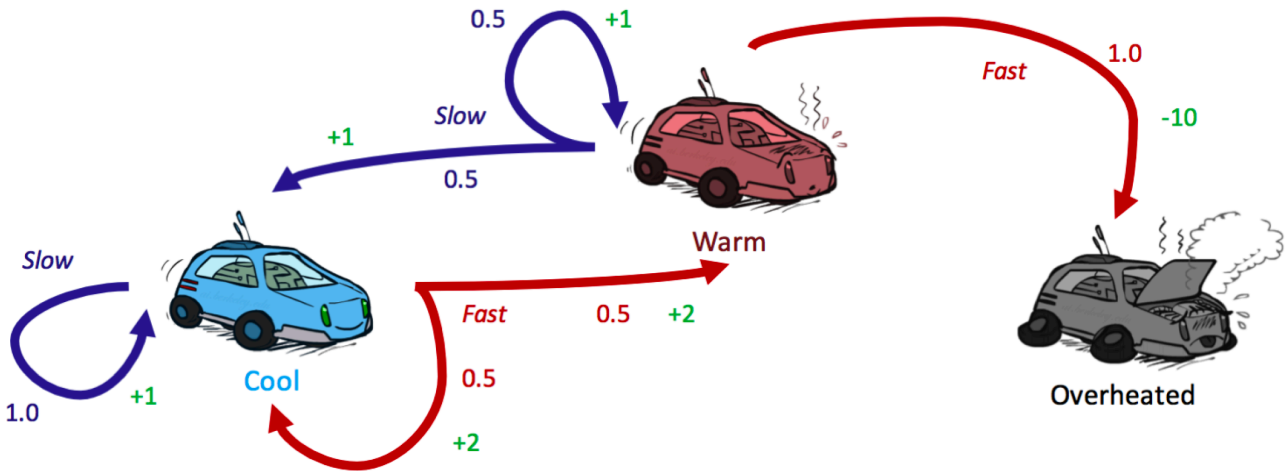


2.1 Markov decision process

A stochastic search problem can be formally described by a **Markov decision process (MDP)**, which is defined by the following properties.

1. **State space** – Set of all states that are possible in the world. A state represents the picture of the world at a certain point along the progression of the agent's plan to reach its goal.
2. **Action space** – Set of all actions that are possible in the world.
3. **Transition function T(s, a, s')** – Function that takes in a state-action-successor triplet, and returns the probability of ending up in the successor state **s'** when the action **a** is executed in the state **s**. *This generalizes the notion of successor function in state-space graphs.*
4. **Reward function R(s, a, s')** – Function that takes in a state-action-successor triplet, and returns the reward for ending up in the given successor state **s'** when the given action **a** is executed in the given state **s**. *This generalizes the notion of cost in state-space graphs.*
5. **Terminal states** – The states where the agent stops to interact with the world.

Constructing an MDP for a stochastic search problem is quite similar to constructing a state-space graph for a deterministic search problem, with a couple additional caveats. To illustrate the concept, let us consider the example of a racecar that wants to travel far, quickly. The racecar can go slow or fast. Going fast gets double reward, but also has a chance to get the racecar warm, and eventually overheated, which is a terminal state (no further actions are possible in this state). The corresponding MDP is illustrated below, where each of the three states is represented by a node, with edges representing actions. Notably, for nondeterministic actions, there are multiple edges representing the same action from the same state with differing successor states. Each edge is annotated with the action it represents, a transition probability, and the corresponding reward.



More specifically, the above MDP is defined by the following properties.

- **States**: cool, warm, overheated (terminal).
- **Actions**: slow, fast.
- **Transition T(s, a, s')**: In the state "cool", the action "slow" always keeps the state "cool", whereas the action "fast" may change the state to "warm" with 50% probability. In the state "warm", the action "slow" may change the state to "cool" with 50% probability, whereas the action "fast" always leads to the terminal state "overheated".
- **Reward R(s, a, s')**: The action "slow" is rewarded +1 in "cool" and "warm" states. Conversely, the action "fast" is rewarded +2 in the state "cool", and -10 in the state "warm".

2.2 Solving a Markov decision process

The agent interacts with an MDP by following a **policy "π(s)"** that indicates which action to take at every state. This interaction is broken down into discrete timesteps, where at each step **t**:

- the MDP is in some state $\mathbf{s_t}$,
- the agent takes an action $\mathbf{a_t = \pi(s_t)}$,
- the MDP moves to a new state $\mathbf{s_{t+1}}$ with probability $\mathbf{T(s_t, a_t, s_{t+1})}$,
- the agent receives a reward $\mathbf{r_t = R(s_t, a_t, s_{t+1})}$.

The agent-MDP interaction is thus described by a trajectory $\boldsymbol{\tau} = (\boldsymbol{s_1}, \boldsymbol{a_1}, \boldsymbol{r_1}, \boldsymbol{s_2}, \boldsymbol{a_2}, \boldsymbol{r_2}, \dots)$. Solving an MDP means finding an optimal policy that, if followed by the agent, maximizes the sum of rewards $\boldsymbol{r_1} + \boldsymbol{r_2} + \cdots$, also called **return**. Due to uncertainty in the MDP, an optimal policy must actually maximize the **average return** over all the trajectories generated by following that policy. For example, the racecar MDP is solved by the policy **π(cool) = fast** and **π(warm) = slow**, because it allows the agent to maximize the average return received after a finite number of timesteps.[4]

2.3 Value iteration

The optimal policy for an MDP can be found through a variety of methods. Dynamic programming is a family of such methods, which require knowledge of both the transition/reward functions. The basic algorithm, called **value iteration**, works by repeatedly updating the numerical values of a table indexed by states and actions, until those values stop changing. Specifically, it works as follows.

1. **Initialize** a table with enough values for every state-action pair, and set them to zero:

$$[\forall (s,a) \in \mathcal{S} \times \mathcal{A}] \qquad Q_0(s,a) = 0$$

   where $\mathcal{S}$ is the set of all states, and $\mathcal{A}$ is the set of all actions.

2. **Repeat** the following update until convergence:

$$[\forall \boldsymbol{s} \in \mathcal{S}] \qquad V_k(\boldsymbol{s}) = \max_{\boldsymbol{a} \in \mathcal{A}} Q_k(\boldsymbol{s}, \boldsymbol{a})$$

$$[\forall (\boldsymbol{s}, \boldsymbol{a}) \in \mathcal{S} \times \mathcal{A}] \qquad Q_{k+1}(\boldsymbol{s}, \boldsymbol{a}) = \sum_{\boldsymbol{s'} \in \mathcal{S}} T(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s'}) \big(R(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s'}) + \gamma\, V_k(\boldsymbol{s'})\big)$$

   where $\gamma \in \,]0,1]$ is the discount rate.

3. **Extract** the optimal policy from the Q-values:

$$[\forall \boldsymbol{s} \in \mathcal{S}] \qquad \pi_*(\boldsymbol{s}) = \operatorname*{argmax}_{\boldsymbol{a} \in \mathcal{A}} Q_*(\boldsymbol{s}, \boldsymbol{a}).$$

The value iterations converge to the "correct answer" for any $\gamma < 1$ ($\gamma \leq 1$ if the MDP graph is acyclic). Specifically, at each iteration **k**, the value $\boldsymbol{V_k(s)}$ is equal to the average discounted return received when the agent starts from state **"s"** and acts optimally for a horizon of **k** steps.

[4] Note that the number of timesteps can be finite or infinite. A finite horizon is often used when the interaction between the agent and the MDP breaks naturally into **episodes**. In this case, the horizon is the maximum number of steps that the agent is allowed to take for reaching a terminal state. Conversely, if the interaction does not break into identifiable episodes, but goes on continually without limit (like in the racecar example), the horizon is infinite. In this case, a **discount rate γ < 1** must be introduced in the sum of rewards to exponentially decrease the value of future actions.

Let's see a few updates of value iteration in practice by revisiting the racecar MDP from earlier. Since the MDP graph contains loops, we will choose $\gamma = 1/2$ as a discount factor.

- **Initialization**. We begin by creating a Q-table filled with zeros.

Q_0(state, action)		state		
		cool	**warm**	**overheated**
action	**slow**	0	0	0
	fast	0	0	0

- **1st update.** In our first iteration, we update each Q-value according to the formula.

Q_1(state, action)		state		
		cool	**warm**	**overheated**
action	**slow**	1	1	0
	fast	2	-10	0

$$Q_1(\text{cool}, \text{slow}) = \underbrace{T(\text{cool}, \text{slow}, \text{cool})}_{1}\left(\underbrace{R(\text{cool}, \text{slow}, \text{cool})}_{1} + \underbrace{\gamma}_{0.5}\max_{a' \in \mathcal{A}} \underbrace{Q_0(\text{cool}, a')}_{0}\right) = 1$$

$$Q_1(\text{cool}, \text{fast}) = \underbrace{T(\text{cool}, \text{fast}, \text{cool})}_{0.5}\left(\underbrace{R(\text{cool}, \text{fast}, \text{cool})}_{2} + \underbrace{\gamma}_{0.5}\max_{a' \in \mathcal{A}} \underbrace{Q_0(\text{cool}, a')}_{0}\right)$$
$$+ \underbrace{T(\text{cool}, \text{fast}, \text{warm})}_{0.5}\left(\underbrace{R(\text{cool}, \text{fast}, \text{warm})}_{2} + \underbrace{\gamma}_{0.5}\max_{a' \in \mathcal{A}} \underbrace{Q_0(\text{warm}, a')}_{0}\right) = 2$$

$$Q_1(\text{warm}, \text{slow}) = \underbrace{T(\text{warm}, \text{slow}, \text{cool})}_{0.5}\left(\underbrace{R(\text{warm}, \text{slow}, \text{cool})}_{1} + \underbrace{\gamma}_{0.5}\max_{a' \in \mathcal{A}} \underbrace{Q_0(\text{cool}, a')}_{0}\right)$$
$$+ \underbrace{T(\text{warm}, \text{slow}, \text{warm})}_{0.5}\left(\underbrace{R(\text{warm}, \text{slow}, \text{warm})}_{1} + \underbrace{\gamma}_{0.5}\max_{a' \in \mathcal{A}} \underbrace{Q_0(\text{warm}, a')}_{0}\right) = 1$$

$$Q_1(\text{warm}, \text{fast}) = \underbrace{T(\text{warm}, \text{fast}, \text{over.})}_{1}\left(\underbrace{R(\text{warm}, \text{fast}, \text{over.})}_{-10} + \underbrace{\gamma}_{0.5}\max_{a' \in \mathcal{A}} \underbrace{Q_0(\text{over.}, a')}_{0}\right) = -10$$

- **Convergence.** We repeat the update several times, until the Q-values stop changing.

Q*(state, action)		state		
		cool	**warm**	**overheated**
action	**slow**	2	1.75	0
	fast	3.125	-10	0

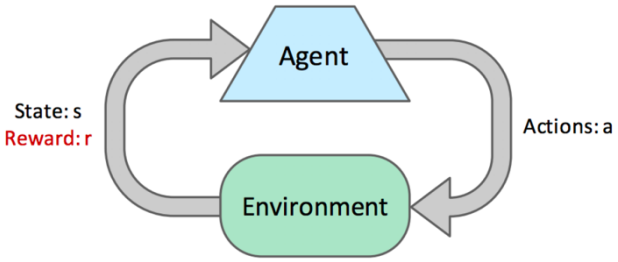
- **Optimal policy.** Finally, we extract the policy from the Q-values.

$$\pi(\text{cool}) = \text{argmax}\left\{\underbrace{Q_*(\text{cool}, \text{slow})}_{2}; \underbrace{Q_*(\text{cool}, \text{fast})}_{3.125}\right\} = \text{fast}$$

$$\pi(\text{warm}) = \text{argmax}\left\{\underbrace{Q_*(\text{warm}, \text{slow})}_{1.75}; \underbrace{Q_*(\text{warm}, \text{fast})}_{-10}\right\} = \text{slow}$$

2.4 Reinforcement learning

Solving an MDP is called **offline planning**, because the agent has full knowledge of both the transition/reward functions, which is all the information they need to precompute optimal actions in the world described by the MDP. By contrast, **online planning** refers to the case when the agent has no prior knowledge of rewards or transitions in the world (still represented as an MDP). In online planning, an agent must try **exploration**, during which it performs actions and receives feedback in the form of the successor states it arrives in, and the corresponding rewards it reaps. The agent uses this feedback to estimate an optimal policy through a process known as **reinforcement learning**, before using this estimated policy for **exploitation**, during which it maximizes the average return.



At each timestep during reinforcement learning, an agent starts in a state **s**, then takes an action **a**, and ends up in a successor state **s'**, attaining some reward **r**. Each **(s, a, r, s')** tuple is a sample of the underlying MDP. An agent continues to take actions and collect samples in succession, until arriving at a terminal state. Such a collection of samples is known as an **episode**. Agents typically go through many episodes during exploration, in order to collect sufficient data needed for learning. There are many types of reinforcement learning methods to process the collected data. In the following, we will focus on model-free learning, where the goal is to estimate the Q-values of state-action pairs, without ever attempting to construct a model of the rewards and transitions in the (unknown) MDP.

2.5 Q-learning

Value iteration computes the Q-values of an MDP by iterating the following update

$$[\forall(\boldsymbol{s},\boldsymbol{a}) \in \mathcal{S} \times \mathcal{A}] \qquad Q_{k+1}(\boldsymbol{s},\boldsymbol{a}) = \sum_{\boldsymbol{s'} \in \mathcal{S}} T(\boldsymbol{s},\boldsymbol{a},\boldsymbol{s'}) \left(R(\boldsymbol{s},\boldsymbol{a},\boldsymbol{s'}) + \gamma \max_{\boldsymbol{a'} \in \mathcal{A}} Q_k(\boldsymbol{s'},\boldsymbol{a'}) \right).$$

Q-learning tries to estimate the above average without having knowledge of the weights **T(s,a,s')**, cleverly doing so with an exponential moving average. Specifically, it works as follows.

- **Initialize:** Begin by setting $Q(s,a) \leftarrow 0$ for every $(s,a) \in \mathcal{S} \times \mathcal{A}$.
- **Repeat:** At each timestep, collect a sample $(\boldsymbol{s},\boldsymbol{a},\boldsymbol{r},\boldsymbol{s'})$ by interacting with the environment, and then perform the following update

  $$Q(\boldsymbol{s},\boldsymbol{a}) \leftarrow (1-\alpha)\, Q(\boldsymbol{s},\boldsymbol{a}) + \alpha \left(\boldsymbol{r} + \gamma \max_{\boldsymbol{a'} \in \mathcal{A}} Q(\boldsymbol{s'},\boldsymbol{a'}) \right),$$

  where $\alpha \in [0,1]$ is the **learning rate**.
- **Exploit:** After enough iterations, extract the optimal policy from the Q-values.

Note that a Q-value is updated after each action is taken, so as to learn something at each timestep. The learning rate typically starts at $\alpha = 1$, and then it slowly shrinks to $\alpha \to 0$, at which point all subsequent samples will be zeroed out, and stop affecting the learned Q-values. Moreover, the exponential moving average gives less importance to older samples, potentially less accurate. As long as the algorithm spends enough time in exploration, and decreases the learning rate at an appropriate pace, it learns the optimal Q-values. This is what makes Q-learning so revolutionary: it can learn the optimal policy directly, even by taking suboptimal or random actions.

2.6 Exploration vs exploitation

Q-learning can learn an optimal policy only if "sufficient" exploration is performed. This is known as **exploration-exploitation tradeoff** in reinforcement learning, and refers to how the time must be distributed between exploration and exploitation to ensure that the optimal policy is learned. We'll now cover two ways to ensure sufficient exploration: e-greedy policies and exploration functions.

Agents following an **"epsilon-greedy" policy** can decide either to explore by acting randomly with probability $\varepsilon < 1$, or to exploit by following their current established policy with probability $\mathbf{1-\varepsilon}$. This is a very simple policy to implement, yet can still be quite difficult to handle. If a large value for the probability ε is selected, then even after learning the optimal policy, the agent will still behave mostly randomly. Similarly, selecting a small value for the probability ε means the agent will explore infrequently, making Q-learning to infer the optimal policy very slowly. To get around this, the parameter ε must be manually tuned and lowered over time to see results.

The issue of manually tuning the parameter ε can be avoided by exploration functions, which modify the Q-value update to give some preference to visiting less-explored states. The update is as follows:

$$Q(s,a) \leftarrow (1-\alpha)\, Q(s,a) + \alpha \left(r + \gamma \max_{\boldsymbol{a'} \in \mathcal{A}} \boldsymbol{F}(s', \boldsymbol{a'}) \right),$$

where **F** denotes an exploration function. There exists some degree of flexibility in designing an exploration function, but a common choice is to use $\boldsymbol{F}(s,a) = Q(s,a) + \frac{k}{N(s,a)}$, where **k** is some predetermined value, and **N(s, a)** is the number of times the Q-state (s,a) has been visited. Agents in a state always select the action that has the highest **F**-value in that state, and hence never have to make a probabilistic decision between exploration and exploitation. Instead, exploration is automatically encoded by the exploration function, as the ratio $\frac{k}{N(s,a)}$ gets larger and larger for infrequently-taken actions, such that they are selected over actions with higher **Q**-values. As time goes on and states are visited more frequently, this ratio decreases towards 0 and thus **F**-values regress towards Q-values, making exploitation more and more exclusive.

2.7 Approximate Q-learning

While Q-learning is an incredible learning technique that continues to sit at the center of reinforcement learning, it still has some room for improvement. As it stands, Q-learning just stores all Q-values in tabular form, which is not particularly efficient given that most applications have several thousand, or even millions, of states. This means we can't visit all states during training.

Figure 1

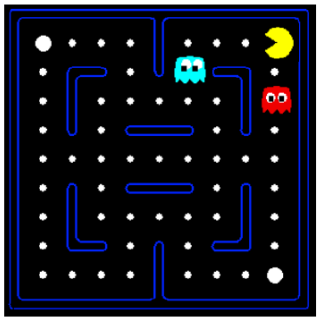


Figure 2

In the illustration above, if Pacman learned that Figure 1 is unfavorable after running vanilla Q-learning, it would still have no idea that Figure 2 is unfavorable as well. **Approximate Q-learning** tries to account for this by learning about a few general situations, and extrapolating to many similar situations. The key to generalize learning experiences is the feature-based representation of states, which represents each state as a small vector. For example, a feature vector for Pacman may encode

- the distance to the closest ghost,
- the distance to the closest food pellet,
- the number of ghosts,
- 0 or 1 whether Pacman is trapped or not.

With feature vectors, we can treat Q-values as **linear value functions**

$$Q(s,a) = w_1\, f_1(s,a) + \cdots + w_n\, f_n(s,a),$$

where $f(s,a) = [f_1(s,a), \dots, f_n(s,a)]$ represents the feature vector for a state-action pair, and $w = [w_1, \dots, w_n]$ denotes the weight vector shared by all state-action pairs. Approximate Q-learning works identically to Q-learning, with the only difference being that after collecting a $(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{r}, \boldsymbol{s}')$ sample through the interaction with the environment, the shared weights are updated as follows:

$$[\forall i \in \{1, \dots, n\}] \qquad w_i \leftarrow w_i + \alpha\, f_i(s,a)\left(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right).$$

Rather than storing Q-values, approximate Q-learning only requires a single weight vector, and computes Q-values on-demand as needed. This gives us a more generalized version of Q-learning, which is also significantly more efficient in terms of memory.