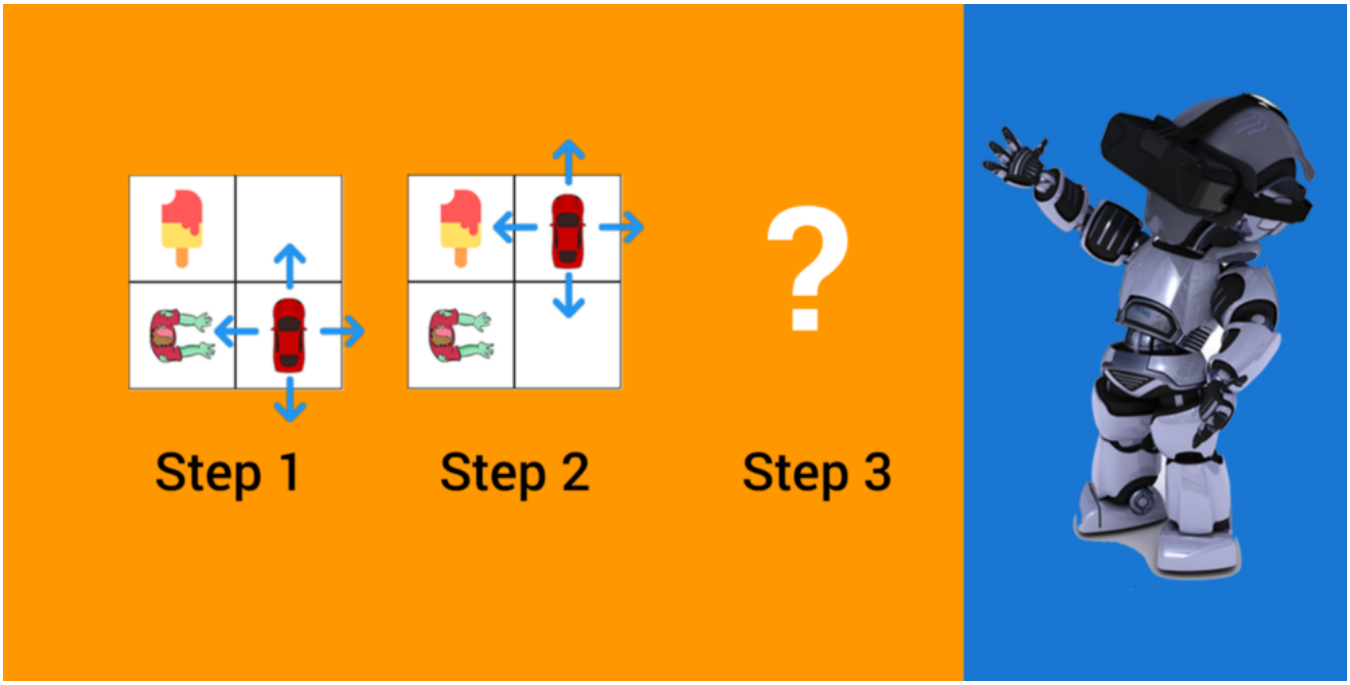
Venelin Valkov Follow
Adventures in Artificial Intelligence
Dec 10, 2017 · 9 min read



Robot image created by Kjpargeter — Freepik.com

Solving an MDP with Q-Learning from scratch — Deep Reinforcement Learning for Hackers (Part 1)

It is time to learn about value functions, the Bellman equation, and Q-learning. You will use all that knowledge to build an MDP and train your agent using Python. Ready to get that ice cream?

Here's an example of how well-trained agents can act in their environments given the proper incentive:

Discounted Future Reward

Why do we need the discount factor γ ? The total reward that your agent will receive from the current time step t to the end of the task can be defined as:

$$R_t = r_t + r_{t+1} + \ldots + r_n$$

That looks ok, but let's not forget that our environment is stochastic (the supermarket might close any time now). The discount factor allows us to value short-term reward more than long-term ones, we can use it as:

$$R_t = R_t + \gamma r_{t+1} + \ldots + \gamma^{n-t} r_n = r_t + \gamma R_{t+1}$$

Our agent would perform great if he chooses the action that maximizes the (discounted) future reward at every step.

Value function

It would be great to know how "good" a given state s is. Something to tell us: no matter the state you're in if you transition to state s

your total reward will be x, word! If you start from s and follow policy π. That would spare us from revisiting same states over and over again. The **value function** does this for us. It depends on the state we're in s and the policy π your agent is following. It is given by:

$$V^{\pi}(s) = \mathbb{E}(\sum_{t \geq 0} \gamma^t r_t) \quad \forall s \in \mathbb{S}$$

There exists an **optimal value function** that has the highest value for all states. It is given by:

$$V^{*}(s) = \max_{\pi} V^{\pi}(s) \quad \forall s \in \mathbb{S}$$

Q function

Yet, your agent can't control what state he ends up in, directly. He can influence it by choosing some action a. Let's introduce another function that accepts state and action as parameters and returns the expected total reward—the Q function (it represents the "quality" of a certain action given a state). More formally, the function $Q^{\pi}(s, a)$ gives the expected return when starting in s, performing aa and following π.

Again, we can define the optimal Q-function $Q^{*}(s, a)$ that gives the expected total reward for your agent when starting at s and picks action a. That is, the optimal Q-function tells your agent how good of a choice is picking a when at state s.

There is a relationship between the two optimal functions V^{*} and Q^{*}. It is given by:

$$V^*(s) = \max_a Q^*(s,a) \quad \forall s \in \mathbb{S}$$

That is, the maximum expected total reward when starting at `s` is the maximum of `Q*(s, a)` over all possible actions.

Using `Q*(s, a)` we can extract the optimal policy `π*` by choosing the action aa that gives maximum reward `Q*(s, a)` for state `s`. We have:

$$\pi^*(s) = \arg\max_a Q^*(s,a) \quad \forall s \in \mathbb{S}$$

There is a nice relationship between all functions we defined so far. You now have the tools to identify states and state-action pairs as good or bad. More importantly, if you can identify `V*` or `Q*`, you can build the best possible agent there is (for the current environment). But how do we use this in practice?

Learning with Q-learning

Let's focus on a single state `s` and action `a`. We can express `Q(s, a)` recursively, in terms of the Q value of the next state `s'`:

$$Q(s,a) = r + \gamma \max_{a'} Q(s',a')$$

This equation, known as the **Bellman equation**, tells us that the maximum future reward is the reward the agent received for entering the current state `s` plus the maximum future reward for the next state `s'`. The gist of Q-learning is that we can iteratively approximate `Q*` using the Bellman equation described above. The Q-learning equation is given by:

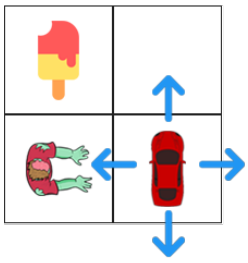
$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t))$$

where α is the learning rate that controls how much the difference between previous and new Q value is considered.

Can your agent learn anything using this? At first—no, the initial approximations will most likely be completely random/wrong. However, as the agent explore more and more of the environment, the approximated Q values will start to converge to Q_*.

Building the Environment

Okay, it is time to get your ice cream. Let's try a simple case first:



Simple MDP—4 possible states

The initial state looks like this:

```
ZOMBIE = "z"
CAR = "c"
ICE_CREAM = "i"
EMPTY = "*"

grid = [
    [ICE_CREAM, EMPTY],
    [ZOMBIE, CAR]
]

for row in grid:
    print(' '.join(row))
```

>

```
i *
z c
```

We will wrap our environment state in a class that holds the current grid and car position. Having a constant-time access to the car position on each step will help us simplify our code:

```
class State:

    def __init__(self, grid, car_pos):
        self.grid = grid
        self.car_pos = car_pos

    def __eq__(self, other):
        return isinstance(other, State) and self.grid ==
other.grid and self.car_pos == other.car_pos

    def __hash__(self):
        return hash(str(self.grid) + str(self.car_pos))

    def __str__(self):
        return f"State(grid={self.grid}, car_pos=
{self.car_pos})"
```

All possible actions:

```
UP = 0
DOWN = 1
LEFT = 2
RIGHT = 3

ACTIONS = [UP, DOWN, LEFT, RIGHT]
```

and the initial state:

```
start_state = State(grid=grid, car_pos=[1, 1])
```

Your agent needs a way to interact with the environment, that is, choose actions. Let's define a function that takes the current state with an action and returns new state, reward and whether or not the episode has completed:

```
from copy import deepcopy

def act(state, action):

    def new_car_pos(state, action):
        p = deepcopy(state.car_pos)
        if action == UP:
            p[0] = max(0, p[0] - 1)
        elif action == DOWN:
            p[0] = min(len(state.grid) - 1, p[0] + 1)
        elif action == LEFT:
            p[1] = max(0, p[1] - 1)
        elif action == RIGHT:
            p[1] = min(len(state.grid[0]) - 1, p[1] + 1)
        else:
            raise ValueError(f"Unknown action {action}")
        return p

    p = new_car_pos(state, action)
    grid_item = state.grid[p[0]][p[1]]

    new_grid = deepcopy(state.grid)

    if grid_item == ZOMBIE:
        reward = -100
        is_done = True
        new_grid[p[0]][p[1]] += CAR
    elif grid_item == ICE_CREAM:
        reward = 1000
        is_done = True
        new_grid[p[0]][p[1]] += CAR
    elif grid_item == EMPTY:
        reward = -1
        is_done = False
        old = state.car_pos
        new_grid[old[0]][old[1]] = EMPTY
        new_grid[p[0]][p[1]] = CAR
    elif grid_item == CAR:
        reward = -1
        is_done = False
    else:
        raise ValueError(f"Unknown grid item {grid_item}")
```

```
    return State(grid=new_grid, car_pos=p), reward, is_done
```

In our case, *one episode* is starting from the initial state and crashing into a Zombie or eating the ice cream.

Learning to drive

Ok, it is time to implement the Q-learning algorithm and get the ice cream. We have a really small state space, only *4 states*. This allows us to keep things simple and store the computed Q values in a table. Let's start with some constants:

```
import numpy as np
import random

random.seed(42) # for reproducibility

N_STATES = 4
N_EPISODES = 20

MAX_EPISODE_STEPS = 100

MIN_ALPHA = 0.02

alphas = np.linspace(1.0, MIN_ALPHA, N_EPISODES)
gamma = 1.0
eps = 0.2

q_table = dict()
```

We will decay the learning rate, `alpha`, every episode - as your agent explores more and more of the environment, he will "believe" that there is not that much left to learn. Additionally, limits for the number of training episodes and steps are defined.

Dicts in Python can be a bit clunky, so we're using a helper function `q` that gives the Q value for a state-action pair or for all actions, given a state:

```
def q(state, action=None):

    if state not in q_table:
        q_table[state] = np.zeros(len(ACTIONS))

    if action is None:
        return q_table[state]

    return q_table[state][action]
```

Choosing an action given the current state is really simple—act with random action with some small probability or the best action seen so far (using our `q_table`):

```
def choose_action(state):
    if random.uniform(0, 1) < eps:
        return random.choice(ACTIONS)
    else:
        return np.argmax(q(state))
```

Why your agent uses random actions, sometimes? Remember, the environment is unknown, so it has to be explored in some way—your agent will do so using the power of randomness.

Up next, training your agent using the Q-learning algorithm:

```
for e in range(N_EPISODES):

    state = start_state
    total_reward = 0
    alpha = alphas[e]

    for _ in range(MAX_EPISODE_STEPS):
        action = choose_action(state)
        next_state, reward, done = act(state, action)
        total_reward += reward

        q(state)[action] = q(state, action) + \
                alpha * (reward + gamma *
np.max(q(next_state)) - q(state, action))
        state = next_state
```

```
            if done:
                break
        print(f"Episode {e + 1}: total reward -> {total_reward}")
```

>

```
Episode 1: total reward -> 999
Episode 2: total reward -> 998
Episode 3: total reward -> 997
Episode 4: total reward -> 997
Episode 5: total reward -> 999
Episode 6: total reward -> 999
Episode 7: total reward -> 998
Episode 8: total reward -> -100
Episode 9: total reward -> -101
Episode 10: total reward -> 999
Episode 11: total reward -> 999
Episode 12: total reward -> 999
Episode 13: total reward -> 999
Episode 14: total reward -> 999
Episode 15: total reward -> 999
Episode 16: total reward -> 998
Episode 17: total reward -> 999
Episode 18: total reward -> 999
Episode 19: total reward -> 999
Episode 20: total reward -> 999
```

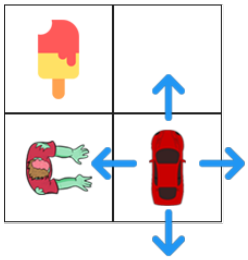
Here, we use all of the helper functions defined above to ultimately train your agent to behave (hopefully) kinda optimal. We start with the initial state, at every episode, choose an action, receive reward and update our Q values. Note that the implementation looks similar to the formula for Q-learning, discussed above.

You can clearly observe that the agent learns how to act efficiently, very quickly. Our MDP is really small and this might be just a fluke. Moreover, looking at some episodes. you can see that the agent hit a Zombie.

Did it learn something?

Let's extract the policy your agent has learned by selecting the action

with maximum Q value at each step, we will do that manually, like a boss. First up, the `start_state` :



Your agent stars here at every new episode

```
r = q(start_state)
print(f"up={r[UP]}, down={r[DOWN]}, left={r[LEFT]}, right=
{r[RIGHT]}")
```

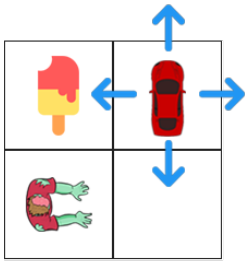
```
>
```

```
up=998.99, down=225.12, left=-85.10, right=586.19
```

UP seems to have the highest Q value, let's take that action:

```
new_state, reward, done = act(start_state, UP)
```

The new state looks like this:



Getting closer to the ice cream

What is the best thing to do now?

```
r = q(new_state)
print(f"up={r[UP]}, down={r[DOWN]}, left={r[LEFT]}, right=
{r[RIGHT]}")
```

```
>
```

```
up=895.94, down=842.87, left=1000.0, right=967.10
```

But of course, going left will get you the ice cream! Hooray! Your agent seems to know it's way around here.

Isn't this amazing? Your agent doesn't know anything about the "rules of the game", yet it manages to learn that Zombies are bad and ice cream is great! Also, it tries to reach the ice cream as quickly as possible. The reward seems to the ultimate signal that drives the learning process.

We're done here! You can now build complex agents that find optimal policies quickly. Except, maybe not. This was a very simple MDP. Next, we will find how Neural Networks fit into the Reinforcement Learning framework.

Want to learn more?

Guest Post (Part I): Demystifying Deep Reinforcement Learning - Intel Nervana

Two years ago, a small company in London called DeepMind uploaded their pioneering paper "Playing Atari with Deep...

www.intelnervana.com

CS234: Reinforcement Learning

To realize the dreams and impact of AI requires autonomous systems that learn to make good decisions. Reinforcement...

web.stanford.edu

Deep Reinforcement Learning Demysitifed (Episode 2) — Policy Iteration, Value Iteration and Q...

In previous two articles, we introduced reinforcement learning definition, examples, and simple solving strategies...

medium.com

Originally published at curiousily.com on December 8, 2017.

. . .