35_Mirage Mazes: Q-Learning

Created	@Nov 19, 2020 8:55 AM

The Ideas and The Intuitions

We have seen the use of solving algorithms, from BFS and DFS up to A^* and UCS. When we are faced with the task of added complexity in the terms of new mirage walls, all of the above search algorithm fail. Not because they aren't good enough; but because their basic assumptions are destroyed. What looks as a wall, may not be one, if you touch it. And, this simple addition to the building process, of adding a few mirage walls, is enough to break the algorithms we know and love. So, what do we do?

Well the answer is easy to think of. That is, think like a human.

...?

Let me explain. What will happen if you are placed in a gigantic maze, walls much taller than yourselves and now, with the added dread of fake-ness? The first answer is, obviously we'll go and touch all the walls. I don't want to be stopped just one step away from the gold!

But now, what if there is a penalty to touch the walls? Say, you get a teeny-tiny electric shock. Touching all the walls doesn't seem like a good idea anymore. Can we do better?

One thing you are free to assume is that you have perfect memory. Every cell of the maze is perfectly stored in your head. What would you do?

Well the best idea is to start slow. Keep touching and checking walls as you go, but if there is a path, take it. At the end of the maze, you are allowed to 'reset'. You get back to your original position, but now you have additional information.

For the first few iterations, that is reaching the end of the maze, is purely random. You end up taking larger paths or end up touching too many walls. For each cell, a good idea is to remember how good taking that edge of the cell is; that is, is taking step in North direction better, did it give better results than the last time?

One clear approach that emerges is, that greedily picking the next best cell of the grid is helpful. But, what about new cells? The cells you have not explored yet? How do you balance this exploration versus (what we call) exploitation? This way of picking is your strategy, what we call: the policy.

At the same time, even if we have a strategy, how do we know it is the optimal strategy? Is there not a shorter path? Can we minimize the shocks we get? Lets take a deeper dive into the world of Reinforcement Learning, and in that way, our own Minds.

Notations

The Language of Reinforcement Learning

Aa Symbol/Label	Description
S	The set of all the States. Each state is a position an agent can be in.
A	The set of all the actions. Note: Not all actions are valid for all the states. it depends where can an agent move from this current state

Symbol/Label	Description
R	The reward for any action taken, form a given state. This is something we have to define.
\pi	The policy function, maps $S \rightarrow A$. Explained in depth just below.
\gamma	The discount factor. How uncertain is the future and how much of it do we want to consider?
Episode	One iteration through the entire maze; from the starting point to the ending point.
Q	The Q value function. Takes input as $(state, action)$, returns the *q-value* at that point, for that action.

Policy

What is a policy? Its is the **strategy** that is used by the agent(you) to get to the optimal solution. More elaborately, policy dictates a mapping from the states that an agent can be in, to the actions that agent can take. In fact, policy is represented as a function, $\pi : S \rightarrow A$. That is, a mapping from all the possible state space, to the action space.
For example, the agent can be in state 0, the initial state. How does the agent then decide where to move? It can obviously take random steps, but we can do better.

One of the policies talked about in Q Learning is a type of **deterministic** policy. What does that mean? It means that for a given state, you will always have a fixed action as the output of the policy function. It is consistent, given the same state. Since we want to maximize our rewards, we will always pick the best action; the one that gives the highest reward, **greedily**.

This kind of policy will certainly work if we know what steps to take, but then ... , how do we get there? If you already know the optimal steps for each and every action, there is no need for the algorithm itself!

Okay. Let's take a step back, and recall that one of the Reinforcement Learning's several central concepts is Exploration versus exploitation. Sure, towards the end, we would want to use the knowledge and exploit the most. But for the beginning, it makes more sense to try out new paths and actions. Since we do not know when to stop exploring, it is evident that this is a battle between the two ideas.

Now, lets come to a compromise. For the most of the part, we want to exploit the rewards, and only for some, we want to explore. This leads us to a rather simple policy: **The $\epsilon - greedy$ policy.**

Simply put, with $P(\epsilon)$ we explore a random action, and, with a probability $P(1-\epsilon)$ we will exploit the previously known actions. For the purposes of this Project, we have kept $\epsilon = 0.10$, moreover, it it fixed. We'll explore this later again, but can you yourself think why decaying ϵ with time is a good idea?

To wrap this up, remember that this is just a strategy. It could be *anything* else in the world.

```
def greedy(state, ACTIONS, evaluation_function):
    best_action = ACTIONS[0]
    best_score = evaluation_function(state, best_action)
    for action in ACTIONS:
        if evaluation_function(state, action) > best_score:
            best_score = evaluation_function(state, action)
            best_action = action
    return best_action
```

```
def e_greedy(state, ACTIONS, evaluation_function, epsilon):
    choice_array = [0]*(int(1/epsilon))
    choice_array[0] = 1

    if random.choice(choice_array):
        return random.choice(ACTIONS)
    else:
        return greedy(state, ACTIONS, evaluation_function)
```

Reward

So, once we have the strategy figured out, its time to structure the rewards for the agent. The agent's goal is the maximize the total rewards. We split this totaling of the rewards to as-per-episode basis.

It comes as no surprise to keep the rewards for the last state; the end of the maze as highly rewarding. At the same time, we punish the agent even more heavily if it decides to cross a wall. The important part to note is that, say for example, we reward the end-goal as $+1000$ points, we want to keep the penalty for crossing the walls as $-10,000$.

Why so? Well, we simply do not want the agent to be able to walk through walls and all that negative score to then be neutralized and made positive by the end-goal score. That is, say the score for walls is -10 only. In this case, the agent can walk though ≈ 100 walls, and still end up with a non-negative score. That will essentially destroy the whole point of the maze!

Okay. For normal mazes this will work just fine. But, what about mirage mazes? there are fake walls spread all over the maze with the probability of wall being fake given as p. The issue is, how much do we reward the agent for finding a fake wall? We can't possibly give a high score, since the agent will the favor exploiting *all* the fake walls and only *then* keep the end-goal, as second priority. At the same time, we *want* the agent to explore out and test walls for fake-ness, since there may be a solution shorter than the current one, just waiting to be explored. Lets keep it in the middle, say 100, with respect to walls being -10000 and end-goal being 1000, for now. We will touch on this score later as well.

So, what prevents the agent for going and checking every single wall? clearly, its always favorable to just check every wall! Lets introduce a penalty too; this time, the additive inverse of the reward for getting a fake wall right. This time, it makes more sense to keep an equal positive and negative reward for the mirage walls.

```
# The following imports are the
interface betwwen what we see: the maze
# and what the algorithm understands.
from helper import state_to_cell, get_next_state

INF = float('inf')

# the directions we can go in
ACTION_MAPPER = {
        0: 'N',
        1: 'S',
        2: 'W',
        3: 'E'
    }

def reward(maze, state, action):
    # pretends to take the action, does NOT take it in the simulation
    next_state = get_next_state(maze, state, action)
    x, y = state_to_cell(maze, state)
    # the agent decides to go though a wall. ouch!
    if maze.grid[x][y].neighbors[ACTION_MAPPER[action]] == INF:
        return -10000
    # the agent reaches the last state!
    elif next_state == (maze.num_rows*maze.num_columns - 1):
        return 1000
    else:
        return 0
```

```
from mirage_helper import check

# note how we have to check for mirage walls.
# actions 4, 5, 6, 7 check the walls N, S, W, E, respectively
def reward_mirage(maze, mirage, state, action):
    if action >= 4:
        x, y = state_to_cell(maze, state)
        direction = ACTION_MAPPER[action-4]

        # the algorithm says its a mirage wall,
        # and we check against the original maze
        if check(mirage, maze, x, y, direction):
            return 100
        else:
            return -100
    else:
        return reward(maze, state, action)
```

Code

```
# the following the hyperparameters for the algorithm
DISCOUNT = 0.8
LEARNING_RATE = 0.5

# constants
N = M = 8
ACTIONS = [i for i in range(8)]
ACTION_MAPPER = {
        0: 'N',
        1: 'S',
        2: 'W',
        3: 'E'
    }
```

```
def reset_q_table():
    """
    initialises/resets the q table

    NOTE: Many sources claim using random initializations,
    but we found that using prior knowledge that forces
    exploitation where needed and exploitation where needed,
    giving a better result. Not just that, the algorithm converges
    quicly as compared to random initializations.
    """
    # setup the q table
    global q_table
    q_table = []

    for i in range(N):
        for j in range(M):
            temp = []
            for value in maze.grid[i][j].neighbors.values():
                if value == INF:
                    # we HIGHLY discourage going through of walls
                    temp.append(-1000)
                else:
```

```
                # we favor exploration for unseen states
                # NOTE the use of states.
                # We dont want to be exploring all the actions,
                # but all the states
                temp.append(10)

        # decision to chech for mirage wall is neutral
        for _ in range(4):
            temp.append(0)

        # temp now has the values for ONE state.
        q_table.append(temp)
```

```
def execute_episode():
    """
    Executes one single episode.
    """
    # the reward of the entire episode
    total_cumulative_reward = 0

    # iterating through each and every state
    for state in range(N*M):
        # taking the action, according to the policy
        # as described above
        action = e_greedy(
            state,
            get_valid_actions(maze, state, ACTIONS),
            Q, epsilon=0.1)

        # getting the reward for taking that action
        R = reward_mirage(maze, mirage, state, action)

        # the next state where we land up
        next_state = get_next_state(maze, state, action)

        # taking the best of the next state
        next_best = max(q_table[next_state])

        # reward for the current state, and,
        # discounted reward for for next state
        td_target = R + DISCOUNT * next_best

        # THE CENTRAL IDEA:
        # how off were our predictions about the future
        td_diff = td_target - q_table[state][action]

        # updating the q-table
        q_table[state][action] += LEARNING_RATE * td_diff
        q_table[state][action] = round(q_table[state][action], 3)

        # adding the reward
        total_cumulative_reward += R

    # returning the total reward for the entire episode
    return total_cumulative_reward
```

Analysis

Reaching a suitable reward for finding a mirage wall proved to be a challenge in itself. We assigned the arbitrary value of 1000 to the destination node, and had to ensure that finding a mirage wall didn't overshadow this reward. However we also wanted the solving agent to try and discover mirage walls and not completely ignore them (because the reward is too low). To reach this fine balance, we came up with the simple heuristic "***if a solving agent finds all the mirage walls, it gets a reward equal to the destination reward***". Keeping this in mind, it is easy to formulate the equation we used to assign the reward to each mirage wall.

$$R_m = \frac{1000}{4pN}$$

- $R_m denotes\ the\ reward\ for\ finding\ a\ mirage\ wall$
- $p\ is\ the\ probability\ that\ a\ wall\ is\ a\ mirage\ wall$
- $N\ is\ the\ total\ number\ of\ nodes$

We know that the total number of possible walls in the maze is $4N$, and that each wall is a mirage wall with probability p, hence our formula matches our heuristic (at least roughly without knowledge of the exact number of mirage walls).

Okay, so we have our Reward function setup. Lets now talk about the Policy. For the greedy policy, we have to look at the 4 edges of each of the node. This takes constant time, or, $O(1)$. For the epsilon greedy policy, we have created an array of size $int(1/\epsilon)$. We then randomly pick one element of that array, again in constant time, $O(1)$.

As for the Q Function itself, for each episode, we take N iterations, to update each node. Within the loop, each of the operations is $O(1)$, thus Q Learning takes a time of $O(EN)$, where E is the number of episodes and N is the number of nodes.

Proof

Our aim is to prove that we can reach Q^*, which is the optimal Q value function; the function which knows all the answers. To do this, recall the Q Learning Update Formula

$$Q_t(s_t,a_t) = Q_t(s_t,a_t) + \alpha_t[r_t + \max_{a'} Q_t(s_{t+1},a_t) - Q_t(s_t,a_t)]$$

To get an accurate idea of the optimal Q function, Q^*, we introduce $\mathbf{H}$, which is a contraction operator; mapping Q^* to a single point.

$$(\mathbf{H}Q)(s_t,a_t) = \sum_{s_{t+1}\in S} P_a(s_t,s_{t+1}) \times [r_t + \gamma \max_{a'} Q(s_{t+1},a')]$$

Uh, hold on, from where did that P come from? What is it?
So, if you look at this same problem from another perspective, we have a bunch of states and a bunch of actions. Our goal is to maximize the reward, by taking the appropriate actions. What we implicitly assumed is that by taking action a, from state s_t, we reach s_{t+1}. For more complex problems, it helps to assume P_a as a probability transition function, which is what guides us from one state to the next.

Okay. So we use the contraction mapping. What space does this lie in?
This operator is in the max-norm space. And we make use of this like so:

$$\begin{aligned}
||\mathbf{H}Q_1 - \mathbf{H}Q_2||_\infty = \max_{s,a} | &\sum_{s_{t+1}\in S} P_a(s_t,s_{t+1}) \times [r_t + \gamma \max_{a'} Q_1(s_{t+1},a') - r_t + \gamma \max_{a'} Q_2(s_{t+1},a')]| \\
&\leq \gamma \times \max_{s,a} |P_a(s_t,s_{t+1}) \times [\max_{a'} Q_1(s_{t+1},a') - \max_{a'} Q_2(s_{t+1},a')]| \\
&\leq \gamma \times \max_{s,a} P_a(s_t,s_{t+1}) \times |\max_{a'} Q_1(s_{t+1},a') - \max_{a'} Q_2(s_{t+1},a')| \\
&\leq \gamma \times \max_{s,a} P_a(s_t,s_{t+1}) \times \max_{a'} |Q_1(s_{t+1},a') - Q_2(s_{t+1},a')| \\
&\leq \gamma \times \max_{s,a} P_a(s_t,s_{t+1}) \times ||Q1 - Q2||_\infty \\
&\leq \gamma ||Q1 - Q2||_\infty
\end{aligned}$$

A lot has happened here. We basically want to prove that we can use a contraction mapping here. Thus, it is clearly valid, since we get that function back in terms. Now, since we need to prove convergence and we have a stochastic process in hand, lets go to *stochastic approximations*.

$$\Delta_{t+1}(x) = (1 - \alpha t(x))\Delta t(x) + \alpha \times t(x) F_t(x)$$

In the above, we have x as s, state. We can also see direct substitutions, into the Q Learning Update formula:

$\Delta_t(s,a) = Q_t(s,a) - Q^*(s,a)$

$F_t(s_t,a_t) = r_t + \gamma \max_{a' \in A}[Q_t(s_{t+1},a') - Q^*(s_{t+1},a')]$

Okay, this fits! But here is the thing. We need to confirm a few assumptions before we can say that we can apply the result of stochastic approximations.

1. $\sum_t \alpha_t = \infty$
2. $\sum_t \alpha_t^2 < \infty$
3. $||E[F_t(s)|F_t]||_\infty \leq \gamma||\Delta_t||_\infty \;\; with \;\; \gamma < 1$
4. $var[F_t(s)|F_t] \leq C(1 + ||\Delta t||^2_\infty), \;\; for \;\; C > 0$

1 and 2 are by the virtue of the fact that we always choose α, the learning rate, from $[0,1)$. 0 means that the new value is not important at all and there is no learning happening, and 1 means that the previous value is not important at all; only the current one is, in both the cases, no learning happens.

To understand the third, we use the contraction operator, as such:

$$\begin{aligned} E[F_t(s_t,a_t)|F_t] &= \sum_{s_{t+1} \in S} P_a(s_t,s_{t+1}) \times [r_t + \gamma \max_{a'} Q(s_{t+1},a') - Q^*(s_{t+1},a')] \\ &= (\mathbf{H}Q_t)(s,a) - Q^*(s,a) \\ &= (\mathbf{H}Q_t)(s,a) - (\mathbf{H}Q^*)(s,a) \; [from\ above] \\ &\leq \gamma||Q - Q^*||_\infty = ||\Delta_t||_\infty \end{aligned}$$

Alright, almost done. Let's prove the fourth!

$$\begin{aligned} var[F_t(s_t,a_t)|F_t] &= E[(r_t + \gamma \max_{a'} Q(s_{t+1},a') - Q^*(s_{t+1},a') - E[F_t(s_t,a_t)])^2] \\ &= E[(r_t + \gamma \max_{a'} Q(s_{t+1},a') - Q^*(s_{t+1},a') - (\mathbf{H}Q_t)(s,a) + Q^*(s,a))^2] \\ &= E[(r_t + \gamma \max_{a'} Q(s_{t+1},a') - (\mathbf{H}Q_t)(s,a))^2] \\ &= var(r_t + \gamma \max_{a'} Q(s_{t+1},a') \mid (\mathbf{H}Q_t)(s,a)) \end{aligned}$$

Thus, since reward is bounded, we can conclude 4 also holds.

And we are done! What we have proved up-till now is simply that the Q function can be modeled in terms of Stochastic Approximations. So, what's next? Well, the result of *any* such stochastic is that this function **converges, with a probability of 1.** Thus, we are done! We have proved that the Q Learning converges to an optimal answer!

For the average reader who can't understand the complicated equations above, it is well known that Q-learning performs well when we have finite action and state spaces.

- **State Space -** The set of all nodes is the state space for a mirage maze. We can only exist in one node at any point in time, and any actions we take moves us between nodes.
- **Action Space -** The decisions to move in a direction (N, S, E, W) or check a wall in a direction is the action space for a mirage maze.

As we can see both of these are finite, and from the formal proof above we can conclude that Q-learning will converge to the optimal solution. Another powerful perk of using Q-learning is it's ability to morph to handle new updates (stretching into Dyna-Q) to our mirage maze. This is perfect for a situation when the mirage maze also has a dynamic aspect.