Bellman's Optimality Equation For Perlin Noise State Space

Mariana Ávalos Arce
13.06.2022

Reinforcement Learning

Reinforcement Learning is a technique of Machine Learning that requires both **states and reward** for the representation of a problem. It associates **actions** with states, with the objective of finding the sequence of actions (trajectory) that rewards the biggest benefit possible of a given problem. An **agent** is the program in charge of solving such problem, for which there are different techniques.

In Reinforcement Learning, there are a few fundamental concepts for the modelling of a problem:

- Set of states: represented by set $S = \{s_0, s_1, ..., s_n\}$, it is the set of all possible states or environment configurations the agent can be in.
- Set of actions: represented by set $A = \{a_0, a_1, ..., a_m\}$, it is the set of all possible actions or methods for transitioning that the agent has available in order to move from one state to another.
- Transition Model Function: represented as $s_f = f_{TM}(s, a)$ if it is a deterministic state space, or $s_f = P_{TM}(s_f|s, a)$ in the case of a non-deterministic state space. If an agent moves, by applying action a, from state s to state s_f with a **probability of arriving to s_f of 1.0**, the problem is deterministic. Otherwise, If an agent moves, by applying action a, from state s to many possible different states, the problem is non-deterministic.
- Reward Function: represented as $r = f_R(s, a, s_f)$, it is the function that returns a numeric reward for the agent when it transitions from state s to state s_f by applying action a. Usually a reward depends only on the arrival state, that is, $r = f_R(s_f)$.
- Policy: represented as $a = f_\pi(s)$ for a deterministic agent decision policy or as $a = P_\pi(a|s)$ for a non-deterministic agent decision policy. This refers to how the agent **takes an action** a while it is in state s, whereas the Transition Model Function refers to the **result** of an action a (already taken following $f_\pi(s)$) while the agent is in state s.

Once a problem is modelled by using the above concepts, there exists the question: how to train the agent to take the trajectory that gives the maximum reward possible? or, how to train the agent to solve the problem? One of the answers to such inquiries involves **Bellman's Opimality Equations**.

Bellman's Optimality Equations

Bellman's Optimality Equations are a set of n non-linear equations, where n is the number of states in the problem's environment, whose solution allows an **optimal policy** to be deduced that gives the agent the maximum reward possible, solving the problem. Bellman's Optimality Equations are defined as two versions: one for V and one for Q, depending on the amount of memory the program is allowed to use, where V version uses less memory for the solution of the non-linear system of n equations but more processing time for the policy deduction, and Q version uses more memory for the solution of the non-linear system of n equations but the deduction of the policy is basically an immediate computation. Due to the small scope of the current experiments, Q Learning is used.

Bellman's Optimality Equations for Q Learning are are expressed for each state s and action a as:

$$Q(s,a) = \sum_{s_f \in S} P_{MT}(s_f|s,a) \cdot (f_R(s,a,s_f) + \gamma \max_{a_f} Q(s_f,a_f)), \tag{1}$$

where the sum of the product of the probabilities given by $P_{TM}(s_f|s,a)$ with the reward term refers to the non-deterministic state space problems, which allows the special case of deterministic state space equations to basically be:

$$Q(s,a) = f_R(s,a,s_f) + \gamma \max_{a_f} Q(s_f,a_f), \tag{2}$$

given that for a deterministic state space, the probability of arriving to state s_f from state s by applying a is 1, or $P_{TM}(s_f|s,a) = 1$, and zero for all remaining states in S. A Bellman's Equation has to be calculated for each action a and state s, creating a matrix of $n \times m$ non-linear equations. It is crucial to note that γ is a constant term that determines the length of the trajectories considered to solve the problem, where $\gamma \in [0,1]$.

Perlin Noise State Space

Perlin Noise was used to create an $N \times N$ squared grid of cells, where each cell value is given by a Perlin Noise function for an octave size of 2. Perlin Noise grids have the common use of representing terrains where an agent can explore. The purpose of this brief document is to train an agent to travel though such terrain in order for it to reach the closest water pond (lowest pieces of terrain) with the trajectory that involves the least effort, meaning that the terrain's height at every coordinate inside the grid represents the **altitude**, and thus the larger the height, the more effort.

Therefore, by having a Perlin Noise grid, we can represent the *reward* for each arrival state s_f as the grid height value for that state but multiplied by -1. A negative reward is often considered a *punishment*, and thus we build the reward system for the agent: the height value of a state s, which is represented by a coordinate point inside the grid, multiplied by -1, gives us the reward the agent gets for arriving to state s, which in this case is a negative reward that naturally gets bigger as the coordinate or state is closer to a pond of water (lowest heights, biggest rewards). Figure 1, shows the grid generated for the present analysis. In other words, a Perlin Noise value at coordinate (x, y), also called state s_f, represents the height or the negative of the reward $f_R(s_f)$

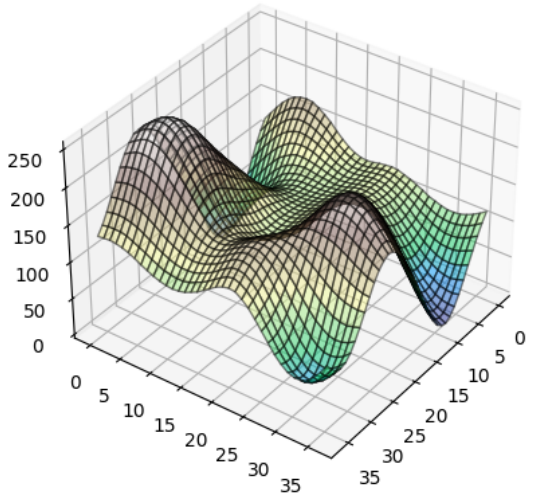

Figure 1: Grid generated by Perlin Noise.

given to the agent when arrival happens at such state s_f.

After generating all cells value for the grid, the computation of the reward values for all states was trival, by just multiplying the value by -1. Since the reward function in this problem depends only on which state does the agent arrive to, the reward function f_R is a uni-variate function, $r = f_R(s_f)$. However, since the reward values where computed from the grid noise values, in order for these rewards to constitute a vector, their 2-dimensional array coordinates need to be transformed into a 1-dimensional array coordinate, by making use of the following function:

```
def mat2line(x, y, dim):
  return (y * dim) + x
```

Now, by computing the array index s_f from a matrix cell position where the reward $r_{i,j}$ for state in (x, y) is, we can have:

$$f_R(s_f) = \begin{matrix} s_0 \\ s_1 \\ . \\ . \\ . \\ s_n \end{matrix} \begin{pmatrix} r_0 \\ r_1 \\ . \\ . \\ . \\ r_n \end{pmatrix}, \tag{3}$$

where n is the number of states, which in our grid state space, $n = N \times N$. The computed rewards for each s_f state arranged as a grid (original coordinates) are shown in Figure 2. The Transition Model Function also needs to be computed from the known information about the problem: by the transitions being deterministic, the Transition Model Function can be defined a two-dimensional array or matrix of n rows and m columns, as shown in Eq. (3).

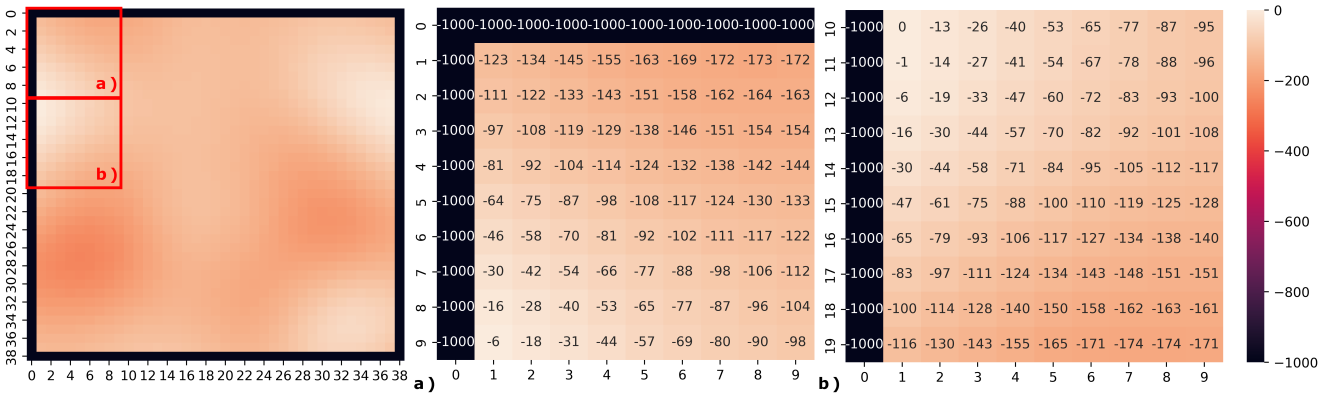

Figure 2: (Leftmost) Computed reward for all states arranged in a grid format. a) Close-up to the reward values of upper left corner. b) Close-up to the reward values of the block of 10×10 beneath the upper left corner.

$$f_{MT}(s,a) = \begin{array}{c} \\ s_0 \\ s_1 \\ . \\ . \\ . \\ s_n \end{array} \begin{array}{c} \begin{array}{cccc} a_0 & a_1 & ... & a_m \end{array} \\ \begin{pmatrix} s_f & s_f & ... & s_f \\ s_f & s_f & ... & s_f \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \\ s_f & s_f & ... & s_f \end{pmatrix} \end{array}, \tag{4}$$

where each s_f defined inside the matrix of size $n \times m$ is the arrival state index of the agent when it is located on state s_i and performs action a_j, given $i \in [0, n]$ and $j \in [0, m]$, respectively. It is important to note that in this problem, $0 \leq m \leq 7$, since there is a total of 8 actions available at each state s_i, that is, up, down, left and right, plus the four diagonal movements.

Method

Solving Bellman's Optimality Equations for Q Learning as stated in Eq. (2), involves solving a non-linear system of n equations. The system is non-linear since Eq. (2) involved a $max()$ function, whose domain is not continuous. Therefore, the computational solution for a non-linear system must be approximated using a numerical method, in this case Value Iteration. Thus, the algorithm to follow must include in each iteration the adjustment given by Eq. (5).

$$Q(s,a) \leftarrow f_R(s,a,s_f) + \gamma \max_{a_f} Q(s_f, a_f), \tag{5}$$

where $Q(s,a)$ initial value at the first iteration is zero. The overall algorithm is described in Alg. 1. The adjustment done at the end of each value iteration is given by Eq. (5), which is a Bellman Equation, meaning it is not continuous as any $max()$ function is just a point in space. This condition also means the adjustment can go on and on, as long as a $max()$ can be computed. This causes the policy to cycle over and over unless there is an explicit stop condition in the algorithm as presented in Alg. 1. Mathematically, the Q matrix of size $n \times m$ implies, at the time of convergence, the action that the agent must take when in state s, which assures that on every state s there exists an action a to be taken in order to get the maximum benefit, and this precisely

also assures that the sequence of actions is infinite if there is no stop condition known beforehand. There will always be a $max()$ value to be computed for each iteration and each equation inside the Q matrix, since the $max()$ is computed from a region that represents Q matrix column given by $Q(s_i, A)$, which can be visualized as shown in Fig.

Algorithm 1 Value Iteration for Q Matrix Solution

Require: $A, S, f_{MT}(s, a), f_R(s_f)$
  while True **do**
    check if *all* values in $deltas$ are less than ϵ to stop the loop
    for $a_i \in A$ **do**
      for $s_i \in S$ **do**
        $s_f = f_{MT}(s_i, a_i)$
        $r = f_R(s_f)$
        $terms = \{\}$
        for $a_{fi} \in A$ **do**
          $term = Q(a_{fi}, s_f)$
          add $term$ to $terms$
        end for
        update $deltas(a_i, s_i)$ with the absolute difference between new and old Q value
        $Q(a_i, s_i) = r + \gamma \max(terms)$ ▷ Q Adjustment in Eq. (5)
      end for
    end for
  end while