RLdrive: Using Reinforcement Learning to make better Driving Decisions

Aditya Jain
IIIT-Delhi
New Delhi, India
aditya14129@iiitd.ac.in

Manasi Malik
IIIT-Delhi
New Delhi, India
manasi15146@iiitd.ac.in

Abstract

In this paper, we aim to solve the classical grid world problem using a Reinforcement Learning (RL) technique - Dynamic Programming (DP). We explore DP methods (policy evaluation, policy iteration, value iteration) to find the optimal paths. Standard techniques (like BFS, DFS, Dijkstra etc.) for finding shortest paths are used for verifying our results. By the end of the semester, we aim to evaluate if querying the state of the environment (non-local to the agent) helps the agent get better rewards. We would be doing this using DP and/or other RL techniques.

1. Introduction

1.1. Motivation

In a not so distant future, when we will have a connected network of autonomous cars on the roads, can we leverage the use of spatial and temporal information obtained from different vehicles to make better decisions regarding cruise speed, overtaking and lane-changing?

1.2. Problem Statement

With the above motivation, we are scaling down the problem to a grid-like environment where different cells represents their state (occupied/free of obstacle) and the agent has to travel from the start to the goal position (or cell) using RL techniques, thus maximizing the overall rewards. In addition to above:
1. Agent can query the state of (any/deterministic) k cells in the grid at each iteration and make decisions accordingly
2. Obstacles may also change their positions in the grid (just like cars on the road) at each iteration

With the above scenario, can the agent cover the distance in the shortest time as well as maximize the total reward?

2. Related Work

While lot of people have used RL techniques to solve various grid related problems with different constraints and optimizations, none have really looked into querying by agents and dynamic state of the environment. Galstyan in [1] discusses about resource sharing among multiple agents, [3] explores multi-agent RL approach for job scheduling in grid computing. [2] addresses the traveling salesman problem using RL techniques.

3. Methodology

To scale down the problem statement, we are solving the grid world problem (reach from start to goal position using the shortest path while maximizing the rewards) in [4] with our added constraints. The environment is defined using a 4x4 grid which gives total 16 cells. The general terminologies used are defined as:

agent - learner and decision maker; **environment -** everything outside the agent with which it interacts; ***s* -** current state of the agent i.e. position in the grid; ***r* -** reward which the agent collects while interacting with the environment ; ***a* -** action which the agent can take to go from one state to another (up, down, right and left in the grid problem); π **-** policy, decision-making rule; ***v*π*(s)* -** value of state s under policy π (expected return); ***q*π*(s,a)* -** value of taking action a in state s under policy π; **terminal state -** final goal position/state,state can't change once reached

In this version of the report, we have implemented Dynamic Programming (DP) methods of RL which are described below:

3.1. Policy Evaluation

Policy evaluation computes the state-value function vπ for a gven policy π. Policy evaluation is defined as **(figure 1)** in the DP literature.

Policy evaluation has to be iterated k times (k $\rightarrow \infty$) to get optimum state values. The pseudo code for the iterative policy evaluation is given in **figure 2**:

$$
\begin{aligned}
v_\pi(s) &\doteq \mathbb{E}_\pi\big[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \mid S_t = s\big] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \\
&= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\Big[r + \gamma v_\pi(s')\Big],
\end{aligned}
$$

Figure 1. State-value function

Input π, the policy to be evaluated
Initialize an array $V(s) = 0$, for all $s \in \mathcal{S}^+$
Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)
Output $V \approx v_\pi$

Figure 2. Iterative policy evaluation

3.2. Policy Improvement

Computing the value function for a policy helps find better policies. Suppose we have determined the value function $v\pi$ for an arbitrary deterministic policy π. For some state s we would like to know whether or not we should change the policy to deterministically choose an action $a \neq \pi(s)$. We know how good it is to follow the current policy from s –that is $v\pi(s)$ –but would it be better or worse to change to the new policy? One way to answer this question is to consider selecting a in s and thereafter following the existing policy, π. The key criterion is whether $q\pi(s,a)$ is greater than or less than $v\pi(s)$. If it is greater –that is, if it is better to select a once in s and thereafter follow π than it would be to follow π all the time –then one would expect it to be better still to select a every time s is encountered, and that the new policy would in fact be a better one overall.

3.3. Policy Iteration

Once a policy, π, has been improved using $v\pi$ to yield a better policy, π', we can then compute $v\pi$' and improve it again to yield an even better π". We can thus obtain a sequence **(figure 3)** of monotonically improving policies and value functions:

$$
\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_*,
$$

Figure 3. Iterative policy evaluation

The pseudo code for policy iteration is given in **figure 4**:

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
     $\Delta \leftarrow 0$
     For each $s \in \mathcal{S}$:
       $v \leftarrow V(s)$
       $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))\big[r + \gamma V(s')\big]$
       $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
     old-action $\leftarrow \pi(s)$
     $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
     If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

Figure 4. Policy Iteration (using iterative policy evaluation)

3.4. Value Iteration

One drawback to policy iteration is that each of its iterations involves policy evaluation, which may itself be a protracted iterative computation requiring multiple sweeps through the state set.

In fact, the policy evaluation step of policy iteration can be truncated in several ways without losing the convergence guarantees of policy iteration. One important special case is when policy evaluation is stopped after just one sweep (one backup of each state). This algorithm is called value iteration. **Figure 5** illustrates the pseudo code for the same:

Initialize array V arbitrarily (e.g., $V(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, $\pi \approx \pi_*$, such that
  $\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$

Figure 5. Value Iteration

4. Results and Analysis

The environment i.e. 4x4 grid **(figure 6)** is as follows: **No. of states** $\rightarrow$ 16; **Possible actions in each state** $\rightarrow$ (up,down,left,right); **Reward** $\rightarrow$ -1 for each step taken; **Terminal states** $\rightarrow$ state 1 and state 16; **Discount Rate** $\rightarrow$ 1.0 Actions that take the agent off the grid will leave the state unchanged.

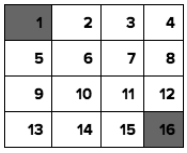

Figure 6. 4x4 Grid World

4.1. Policy Evaluation

We calculated the value function for a random policy **(figure 7)** i.e. all states are equiprobable:

```
Value Function for policy: all actions equiprobable:
[[  0.         -13.99887902 -19.99833891 -21.99814115]
 [-13.99887902 -17.99853668 -19.99835003 -19.99833891]
 [-19.99833891 -19.99835003 -17.99853668 -13.99887902]
 [-21.99814115 -19.99833891 -13.99887902   0.        ]]
```

Figure 7. Value function for random policy

When the maximum change in all state values is less than the threshold (kept as 0.0001), the loop is terminated and what we get is the optimal value function.

4.2. Policy Iteration

We then calculated the optimal policy and the corresponding value function using policy iteration (results in **figure 8**):

```
Best policy with Policy Iteration is
[[ 5.  2.  2.  1.]
 [ 0.  0.  0.  1.]
 [ 0.  0.  1.  1.]
 [ 0.  3.  3.  5.]]
Corresponding Value Function is
[[ 0. -1. -2. -3.]
 [-1. -2. -3. -2.]
 [-2. -3. -2. -1.]
 [-3. -2. -1.  0.]]

Index for actions:
0 : up
1 : down
2 : left
3 : right
5 : stay
```

Figure 8. Policy Iteration results

4.3. Value Iteration

The optimal value function and the corresponding policy was then calculated using value iteration (results in **figure 9**):

```
Best policy with Value Iteration is
[[ 5.  2.  2.  1.]
 [ 0.  0.  0.  1.]
 [ 0.  0.  1.  1.]
 [ 0.  3.  3.  5.]]
Corresponding Value Function is
[[ 0. -1. -2. -3.]
 [-1. -2. -3. -2.]
 [-2. -3. -2. -1.]
 [-3. -2. -1.  0.]]

Index for actions:
0 : up
1 : down
2 : left
3 : right
5 : stay
```

Figure 9. Value Iteration results

4.4. Analysis

Both policy and value iteration tend to give the same final policy and value function owing to the simplicity of the grid. But they vary greatly in terms of computational time taken:

Policy Iteration: **0.022844549927 sec**
Value Iteration: **0.00242858664957 sec**

4.5. Verification with Baseline Technique

We verified our results with standard techniques for finding shortest path, like BFS. The value function calculated for each state in the above cases gives the negation of the expected number of steps from that state until termination. To verify our results, we used BFS to find the shortest path length from each state to the closest termination state (figure 10):

```
Shortest Path Length (using BFS)
from each state to closest
terminal state:
   0  1  2  3
   1  2  3  2
   2  3  2  1
   3  2  1  0

Our Value Function:
[[ 0. -1. -2. -3.]
 [-1. -2. -3. -2.]
 [-2. -3. -2. -1.]
 [-3. -2. -1.  0.]]
```

Figure 10. Comparison with BFS

4.6. Challenges

Dynamic programming is limited in its usage because of two major reasons: it assumes a perfect model of the environment and is computationally expensive. Thus we will be experimenting with more complex RL techniques as part of our future work.

5. Future Work

- We will be adding obstacles in our grid, both deterministically placed and randomly placed.
- We will be implementing querying option. Right now the agent has information only for adjacent cells. We will check if information about other cells, help making better decisions.

Both the team members would be working on the above.

References

[1] K. C. Aram Galstyan and K. Lerman. Resource allocation in the grid using reinforcement learning, 2004.
[2] L. M. Gambardella and M. Dorigo. Ant-q: A reinforcement learning approach to the traveling salesman problem, 1995.
[3] P. Z. Jun Wu, Xin Xu and C. Liu. A novel multi-agent reinforcement learning approach for job scheduling in grid computing, 2010.
[4] R. Sutton and A. Barto. Reinforcement learning: An introduction, 2014.