Markov Decision Processes and Reinforcement Learning

Zheyuan Xu

Georgia Institute of Technology

Introduction

MDP(Markov Decision Processes) is a discrete time, stochastic control scheme, that resembles closely to Markov Chains except that the transition matrix depends on the action taken by the agent at each time step. The agent gets a reward (or punishment) based on the current state and possible actions taken. This report will use two reinforcement learning algorithms: value iteration and policy iteration to solve several MDPs, one with small number of states and another with larger. After the training their performance will be compared against several metrics.

Brief Review of MDP

Markov Decision Processes is an abstract mathematical modelling of a real-world processes and interactions. Essentially it contains several variables in the decision making process:

- S: a finite set of states, from which actions can be performed
- A: a finite set of actions, each of which can be performed on over S
- T(s,a,s'): a transition model that represents the probability of arriving at state a given previous state s' and action a. The model contains probability and entails randomness.
- R(s): an immediate reward function for the immediate value of being in a particular state
- **Π**(s): the policy for decision making, maps particular state to action

The MDP assumes that the probability of the future state is solely dependent on the present state of the system. In finding out the best policy which provides maximum total reward, the MDP is solved. This is called policy iteration.

Further assumptions include the ignorance of previous state as well as time elapsed (the only thing matters is the current state). Some utility measures use a discount factor to delayed award, enabling the extraction of utilities from infinite amount of states. The discounting feature allows MDP to have high-level heuristics when making decisions, and to be used in applications like motion planning as well as chess playing.

Value Iteration

The philosophy behind such measure is readily reasonable: iteratively finding out the highest possible utility.

Under the assumption of MDP, the previous states do not matter that much. The utility, consists of all immediate reward upon reaching that state and the discounted future rewards starting from that state. For a value iteration the steps are:

- Initialize: assigning an arbitrary utility to each state
- For each state s, update its new utility value based on the utility of its neighbours
- Bellman Equation can be used for updating:

$$U(s)_{t+1} = R(s) + \gamma \max_{a} \sum_{s'} T(s, a, s') U(s)_t$$

- Repeat until convergence

Policy Iteration

Value iterations are relatively time-consuming, and it focuses on finding the optimal utility value, not necessarily the optimal policy. Policy iteration, being an alternative, focuses directly on the goal (finding out the optimal policy), the steps are listed as follows:

- Initialize: generate a random policy, being a permutation of actions for all states in MDP
- Loop until convergence (policy gets fixed):
    - Compute utility for every state S according to the new policy chosen
    - Update utility for each state
    - Select the optimal policy for each state, updating current policy

MDP cases

Collision Avoidance

In this simple case, the python script written intends to avoid collisions between vehicles. There are two input files which contain the trajectory of the two vehicle in 2D map, and the output contains the solution which avoids collision between those two vehicles while still enabling them to reach their destination. The reward space is in the form of costs (negative rewards) for moving and collision. The destination reward is positive so it ensures both vehicles will not change their courses in the long run. This simple problem can have many applications in real life, such as traffic control, robot motion planning, etc. The Python script uses policy iteration in this case. If the input coordinates for vehicle 1 is:

```
1,1
3,5
7,7
6,1
7,4
3,9
9,1
6,8
9,8
5,3
8,5
6,5
4,5
7,2
4,9
3,0
1,0
5,2
4,7
4,7
9,5
```

And input coordinates for vehicle 2 is:

```
1,2
3,4
7,5
6,1
7,7
3,9
9,1
6,6
9,9
5,3
8,4
6,4
4,7
7,3
4,8
3,2
1,1
5,8
4,9
4,9
9,8
```

Then after running through the script, it would produce intended action for vehicle 1:

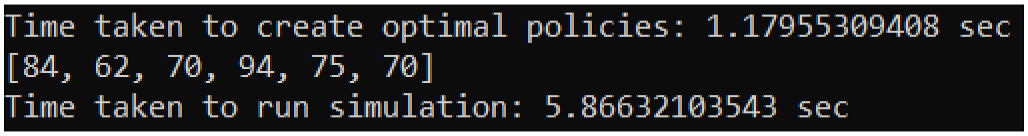
```
82
76
82
65
77
45
```

And the action for vehicle 2:

```
84
62
70
94
75
70
```

Notice that the output files contain coordinate appended together, without the separator (normally the comma). The output does not guarantee that the vehicles will each arrive at its intended destination, but it more of a compromise between achieving goals and avoiding unwanted scenarios.

In this case the number of states is not that large, resulting in trivial running time:



```
Time taken to create optimal policies: 1.17955309408 sec
[84, 62, 70, 94, 75, 70]
Time taken to run simulation: 5.86632103543 sec
```

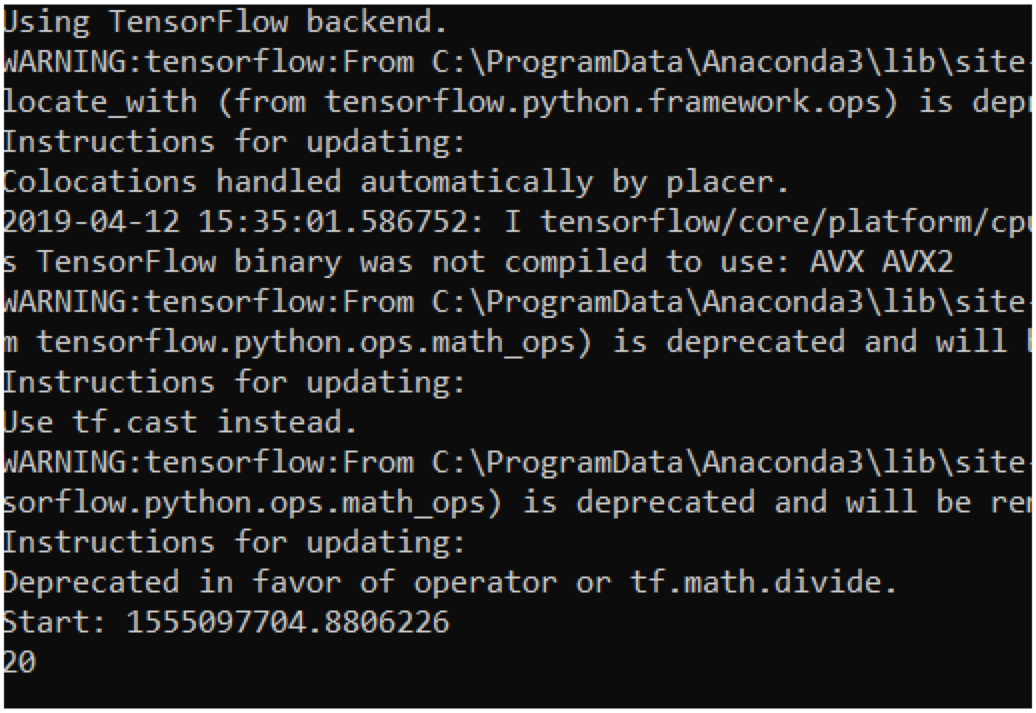
For cases in which a huge number of states are involve, it is more computationally expensive; one of which involves inferring chess board state according to the sensor placed on the board.

Recon Blind Multi-Chess(RBMC)

In this example, the main problem is to create an AI in playing blind chess. During each turn, each player can only see his/her own pieces; however, a sensor can also be placed on any legitimate locations of the board, and a 3 by 3 matrices of chess pieces will be revealed. The winning criterion is based on time, as well as checkmate of king. Due to the blind nature of this chess game, the win rate largely depends on whether or not the board state can be accurately reconstructed by only taking the sensor data and the limited amount of information given.

This problem has many states, considering that each square on the 8X8 board has 13 states in total (6 for white, 6 for black and 1 for empty state). In this problem, the python script written tries to find out an effective mapping between observations and real board state (whether a movement occurs or not). Note that in this case, due to the large number of states involved, the training time is also huge, and it would take several days for a standard PC to obtain the results. In this case, the sensor places are relatively fixed: 28, 24 and 31, and the initial move is also supposed to be standard. The association between observation and non-observed movement proves to be quite weak, though we can derive the relationship between them, by constructing a deep-learning neural network and train it. The code runs on Python 3.6.4 and its corresponding Tensorflow, as well as Keras machine learning library.

The belief_state_training script basically makes the player play against itself, and tries to place sensor on fixed locations iteratively. By feeding the observation to the network as input, it attempts to establish the association between partially observed results and the true board state.



```
Using TensorFlow backend.
WARNING:tensorflow:From C:\ProgramData\Anaconda3\lib\site
locate_with (from tensorflow.python.framework.ops) is depr
Instructions for updating:
Colocations handled automatically by placer.
2019-04-12 15:35:01.586752: I tensorflow/core/platform/cpu
s TensorFlow binary was not compiled to use: AVX AVX2
WARNING:tensorflow:From C:\ProgramData\Anaconda3\lib\site
m tensorflow.python.ops.math_ops) is deprecated and will 
Instructions for updating:
Use tf.cast instead.
WARNING:tensorflow:From C:\ProgramData\Anaconda3\lib\site
sorflow.python.ops.math_ops) is deprecated and will be re
Instructions for updating:
Deprecated in favor of operator or tf.math.divide.
Start: 1555097704.8806226
20
```

The training process is time consuming, and it saves the trained results in belief_state, move_policy as well as sense_policy files. The next time it can load the trained results from last time and continue the training. After the training, we can run play.py to examine the result (basically it outputs true board state matrices and estimated board state by feeding observation into the trained neural network).

The results may not seem as satisfactory, but the sensor now can detect surrounding moves accurately, even without being told to do so. By effectively capturing any possible movement made by the opponent, the player can translate the game into any other normal chess game, and apply MCTS (Monte-Carlo Tree Search) as well as Alphazero deep learning neural network, to achieve an insanely high win rate against other opponents.

The blind chess problem can also be used in applications like air traffic control, drone motion planning since any partially observable problems involving decision making in an unknown environment can be modelled into an RBMC problem.

Real World Problem in MDP and POMDP

Nowadays there are hundreds of consumer drones in the market. Whereas, their access to civilian airspace is strictly prohibited by FAA. The fact that most consumer drones do not have a stable enough collision-avoidance feature makes them susceptible to air crash with other aircraft or obstacles, or even humans. The propellers of large quadcopter is power enough to deliver the torque which rips apart human skin and flesh, causing unexpected casualties. Therefore, a collision avoidance system based on MDP as well POMDP(partially-observable MDP) is necessary to solve such challenges. POMDP, on the other hand, could be converted to continuous MDP problems by applying Bayes' rule.

Formulating the Problem

We use S to model state space of the drone, A to represent action space, and Ω to represent observation space. All space are assumed to be discrete. Transition function T: S x A → Π(S) takes in the current state and action taken and outputs the probability of next state. Observation function O: S x A → Π(S) determines the probability of observation o received after taking action resulting in state s'. The probability of receiving observation o after taking action a and landing in state s' is written as O(s', a, o).

The initial belief state is b0, and the probability of starting in state s is given as b0(s). The space of possible belief state is denoted as B. Belief state is initialized as b0, and updated with more observations, following Bayes' rule. The updating formula is given as:

$$\begin{aligned} b'(s) &= \Pr(s' \mid o, a, b) \\ &\propto \Pr(o \mid s', a, b) \Pr(s' \mid a, b) \\ &= \Pr(o \mid s', a) \sum_{s \in \mathcal{S}} T(s, a, s') b(s) \\ &= O(s', a, o) \sum_{s \in \mathcal{S}} T(s, a, s') b(s) \,. \end{aligned}$$

Given the current belief state, the goal is to find out an action that maximizes the expected discounted return, which is denoted as:

$$\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t)$$

As mentioned above we need to find out a policy π, which outputs an action given the current belief state which is continuously updating. Optimal policies have the highest discounted return at belief state b:

$$V(b) = \max_{\alpha \in \Gamma} (\alpha \cdot b)$$

In which the "alpha" vector contains action with optimal actions associated with each state.

Choosing the Algorithm

Finding out the alpha vector which consists the backbone of our policy can be difficult. Not until recent years did researchers came out of PBVI (point-based value iteration) which possesses the ability to solve problems that are orders of magnitude larger than was previously possible. In problems such as drone routing SARSOP (successive approximations of the reachable space under optimal policies performed better. A drone, in theory could have infinitely many states due to the randomness of real-world sensor data. In our case, the drone's sensor includes a Navio2 sensor hat, which has high-precision IMU sensors to measure the acceleration and position of the drone, GPS modules to measure the location of the drone (only works outdoor so far), voltage of the battery which measures the amount of charge remaining in the battery, as well as a camera which directly streams images online as well as giving an estimate of the speed of obstacles in front of the drone (by measuring the optical flow and their relative positional shift). It could also SONAR and LIDAR modules, which gives an accurate measure of the distance of obstacles, enabling in-door navigation. SARSOP outputs a policy file with a collection of alpha-vectors when the regret bounds fall below some preset values or the user interrupts the program.

CAS (Collision Avoidance System)

State Space:

The dimension of the discretized space is huge (26 according to the report) even in perfect sensing mode. There are, however, a huge amount of useless or marginally useless dimension that we can get rid of. The problem can be formulated to a 5-dimension one including:

- X: horizontal distance to another aircraft / obstacle
- Y: vertical distance to another aircraft / obstacle
- RelativeVx: horizontal closure rate
- OtherVy: vertical velocity of other aircraft / obstacle
- selfVy: vertical velocity of self

By reducing the dimension of state space from 26 to 5, large amount of computation can be saved. The vertical velocity is always known and included in the state bin, and the 5 state bins are split into "start" and "end" sets, in which "start" indicates the set of states before the encounter of obstacle / other aircraft, and "end" indicates the set of states after that encounter.

Action Space:

It consists all possible actions, majorly accelerations. It is actually restricted by the agility of the drone (the thrust-to-weight ratio) as well as the accuracy and sampling rate of the IMU sensors. The on-board coprocessor in this case is Raspberry Pi 3B+, with 1.4GHz.

Reward:

In this case the reward is chosen to be negative (costs). There are several reasons behind that choice:

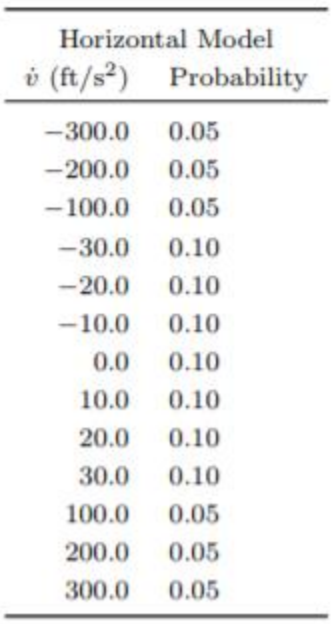
- Collision is strictly prohibited
- Separation distance is recommended
- Energy saving is critical due to limited amount of charge in the battery

Attempted Cost (negative reward):

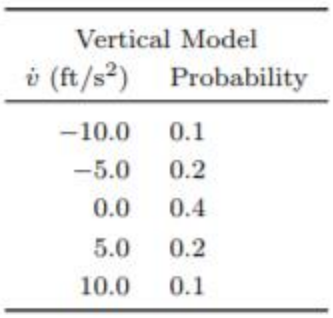
- 1000 for collision
- 500 for entering "danger zone"
- 100 for vertical speed other than 0 (this is to prevent crashing and over-maneuver)

State Transition:

The state transition from one state to another assumes that the other aircraft / obstacle is governed by a stochastic process (for computational ease), and depends on the parameters such as the amount of time required between successive MDP policy selection, magnitude of vertical acceleration, probability of staying in current state, probability of transitioning to other states, as well as the probability of other aircraft's horizontal and vertical acceleration models. For the last two attributes, we have to hardcode the problem to make sure the other aircraft / obstacles stay exactly in the category, since we have no obvious information of them.



Horizontal Model	
$\dot{v}$ (ft/s^2)	Probability
−300.0	0.05
−200.0	0.05
−100.0	0.05
−30.0	0.10
−20.0	0.10
−10.0	0.10
0.0	0.10
10.0	0.10
20.0	0.10
30.0	0.10
100.0	0.05
200.0	0.05
300.0	0.05



Vertical Model	
$\dot{v}$ (ft/s^2)	Probability
−10.0	0.1
−5.0	0.2
0.0	0.4
5.0	0.2
10.0	0.1

Given above are suggested models for other aircraft / obstacles in other UAV studies.

We then compute the joint probabilities for each possible vertical and horizontal accelerations. After that, we then derive the transitional probability for the drone and other aircraft / walls given their acceleration and action of our drone, by as well as our state (3-D box with vertices).

The update model is:

$$\Pr(s' \mid s, a) = \sum_{a_0} \Pr(s' \mid s, a, a_o) p_o$$

The model mentioned above simplifies the problem by cutting the 3D space surrounding the drone into many regular-shaped cubes. The collision model can then be simplified by making sure the aircraft stays away from certain Euclidean distance.

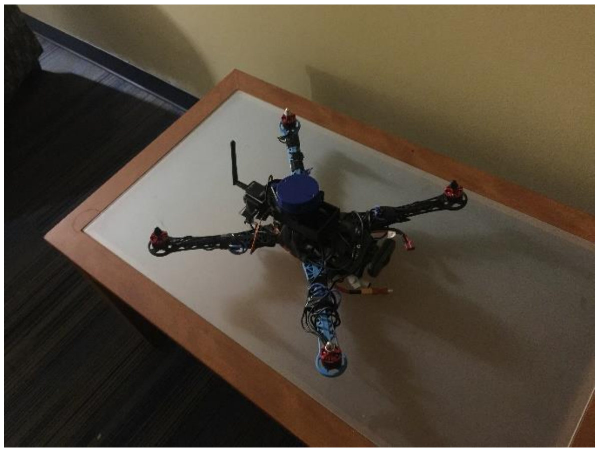


Figure 1. Black-Hornet MK. 2, the multi-purpose VR drone we made from scratch in one Hackathan

Control Problems:

During the testing we found that the sensor noise is incredibly high, that, without a responsive enough control system it is not even able to stably take off. The issues might be caused by the imperfect placement of the center of weight (which might not be an issue considering we did adjust the propellers thrust accounting that), violent vibration during takeoff (which might cause the IMU sensors to temporarily dysfunctional) or the complex electronic environment which causes the compass to produce erroneous results. After many trials, including re-tuning the PID parameters we are able to bring it to the air and execute some very simple maneuvers.

Observation Model:

We model the observation as Ovy, the vertical velocity and Od, the detection of the other aircraft / obstacle:

Pr(Ovy, Od | s) = Pr(Ovy | s) x Pr(Od | s)

(Bayes' rule)

Pr(Ovy | s) is always 1, assuming our vertical velocity measurement is correct. The second term, Pr(Od | s) is more complex due to false positives and false negatives. If there is a false positive detection, the model can be simplified into uniform probability over the space of values of Od.

Last but not least, if the obstacle / aircraft is within our drone's state cube, the sensor might not be able to detect it. It has the probability of Pr(O = NoObservation | s) = Pfn (false negative). If that is not the case, then the probability of successfully making a decision is 1 = Pfn.

When the distance to the obstacles / other aircraft is so large that it is highly likely that the sensor will not detect anything, Pr(Od = NoObservation | s) = 1 – Pfp. For other observations, Pr(Od = d | s, fp) = $|Od|^{-1}$, making it uniform over the space (when the distance is large, then it is close to 0 making the sensor less susceptible to noisy data, when the distance is short, then it is more likely for obstacle / aircraft to come near).

Pr(Od = d | s) = (1 – Pfp)*pi + Pfp*$|Od|^{-1}$

When the aircraft / obstacle detection happens inside the state cube, and

Pr(Od = d | s) = Pfp / $|Od|^{-1}$

According to the TCAS sensor rule, the margin added to 4 sides of X and Y rectangle is calculated as follows:

$$\begin{aligned} margin \quad = \quad & \text{Altitude quantization} + \\ & 3 \times \text{Range error standard deviation} + \\ & 3 \times \text{Altimetry error scale} \end{aligned}$$

Implementation:

To implement the model, we have to utilize the SDK provided by Navio2, which is directly attached on top of Raspberry Pi and does have high-speed datalink with the Pi. We have added a flight script under Python directory which utilized some of the other sensor functions. The check for APM presense is disabled, making the drone to take off and perform action without human interference. The project is still undergoing.

Future Improvement:

- Scan-while-tracking: It is highly effective that once our drone has detected a nearby target, it can focus on gathering more information about that target, thereby focusing more computational resources on the most emergent target. This is done by implementing a MCTS (Monte-Carlo tree search), which is more of a tradeoff between exploration and exploitation.
- Modification of state-space: which models the drone as a "X" shape rather than rectangle. This could potentially add more complexity but enables the drone to traverse narrow areas without touching the surrounding.

Conclusion

To find out the optimal policy, it requires more time for value iteration than policy iteration. The reason is due to the differing objectives of those two approaches: value iteration focuses on finding the maximal utility, which may not guarantee that the policy always converges. Whereas, policy iteration directly focuses on finding out optimal policy which is more sensitive to policy changes and can usually converge earlier in that regard. Policy iteration on the other hand, requires more computation power since the utilities of all states are looped in order to find out the better policy for updating. Whereas, value iteration only cares about the utilities and a loop through all states to find out the optimal value would be enough.

Reference

[1] Selim Temizer, Mykel J. Kochenderfer, Leslie P. Kaelbling, Tom´as Lozano-P´erez, and James K. Kuchark: Collision Avoidance for Unmanned Aircraft using Markov Decision Processes

[2] https://towardsdatascience.com/reinforcement-learning-demystified-markov-decision-processes-part-1-bf00dda41690

[3] Bob Givan, Ron Parr: An Introduction to Markov Decision Processes

[4] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran , Thore Graepel, Timothy Lillicrap, Karen Simonyan, Demis Hassabis: A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play

[5] Fatima Zahra Djiroun, Miloud Bagaa, Tarik Taleb: A Markov Decision Process-Based Collision Avoidance in IoT Applications

[6] Eric Mueller, Mykel J. Kochenderfer: Multi-Rotor Aircraft Collision Avoidance using Partially Observable Markov Decision Processes