Analyzing Iteration and Reinforcement Learning Techniques for Different MDP Inventory Space

Anna Smith

April 22, 2017

Purpose

Two inventory MDPs will be introduced, one with a small number of states, the other with a relatively large number of states. After defining the problems, each will be solved using value iteration and policy iteration. Analysis will be done on the run time, final solution and convergence of the solution. Then Q-learning, a reinforcement learning algorithm, will be used to solve the problems with special attention to parameter selection. The results of this models will be compared to the iteration policies.

Formulation

Markov Decision Process (MDP) is a model that contains a set of states S of which there are a set of possible actions A. After choosing the action to take in a time period, a reward is granted $R(s, a)$ and the next state is determined by the transition model $T(s, a, s')$, given the initial state was s and action a was taken. By the fundamental assumption of an MDP, the decision on the action a to take is only dependent on the current state s and predicted future reward, not any actions or states seen in history. Over an infinite time horizon, MDP aims to give an optimal policy π^* that maximizes expected utility of a state $U(s)$.

One of the most classic examples of an MDP in operations research and supply chain is the basic inventory problem. This problem will be simplified as to not distract from the purpose of analyzing solving and reinforcement techniques. Many more states, actions, parameters, and random variables could be added to make the problem more realistic and able to model real world inventory systems.

Suppose you work at a store that sells one type of shirt. Everyday, before customers come in, you start out with the inventory that you were left with the previous day. At this point, you must decide how many shirts to order O, at a cost c per shirt, to restock your store. After you place this order, shirt instantaneously arrive, and customers start coming in. Customer demand can be modeled by the probability distribution P, where $P[i]$ represents the probability of seeing demand i and $i \in [0, .5C]$, independent each day. You sell each shirt for a, and no customer will be turned down if inventory is still available. At the end of the day, whatever inventory is left incurs a storage fee w, but will be available at the start of the next day as inventory. Your goal is to maximize your profit by selling the most amount of shirts possible, but to minimize the costs associated with each transaction.

The First Problem: Small State Space

Assume all of the conditions from above, but in addition, you have a limited storage capacity C. That is, your inventory on hand (which includes anything you may have in storage from

the previous period as well as anything you order) must always follow $I + O \leq C$. The reward function is represented by the expected reward, and is as follows, given that the input is valid.

$$R(I, O) = \sum_{d=0}^{.5C} P[d]\Big[a \times \min(d, I + O) - c \times O - w \times \max(0, I + O - d)\Big]$$

For the purpose of training the model, when $I_x + O_x > C$, $R(I_x, O_x) = -\infty$. The transition model expresses the probability of having inventory I' at the end of the day, given you started with I inventory and ordered O additional shirts.

$$T(I, O, I') = P(I'|I, O) = \sum_{d=0}^{.5C} P[d] \quad \forall d \quad s.t. \quad I' = \max(0, I + O - d)$$

The Second Problem: Large State Space

Now assume all from the first problem, except that the inventory capacity and demand have increased by a factor of k where $k > 1$. That is, $I + O \leq kC$. Additionally, demand is scaled; the probability distribution $P[i]$ represents the probability of seeing demand i where $i \in [0, kC]$. The reward function will look very similar, except demand is scale by k

$$R(I, O) = \sum_{d=0}^{k\times.5C} P[d]\Big[a \times \min(d, I + O) - c \times O - w \times \max(0, I + O - d)\Big]$$

Similarly to the reward function scale, the transition model is expressed as

$$T(I, O, I') = \sum_{d=0}^{k\times.5C} P[d] \quad \forall d \quad s.t. \quad I' = \max(0, I + O - d)$$

Significance of Problem Selection

Given that $I, O \in \mathbb{Z}^+$, both problems have a finite possible number of state and action combinations. The second problem has k times as many states as the first, and k times as many ordering decisions. These problems will prove to be interesting because they are both very similar in concept, but vastly different in size. Stagnant parameters such as a, c, w will be held constant when evaluating both problems, as these will not effect the process of solving the MDP, rather they will only influence the optimal values and policy decisions. For example, an extremely warehousing fee w will skew the optimal policy to prefer minimizing the number items at the end of the day, even if this drastically cuts down on sales.

Because of its simplicity and linearity, a known solution of this problem can be expressed historically using a (s, S) policy, that is, when inventory falls below s, an order is placed so that inventory is up to S. Traditionally, policy iteration has been seen as the best solution to solving simple inventory MDPs when the reward and transition function is known [5]. As the scope of the problem is changed, we evaluate different iteration and reinforcement learning techniques.

Experiments

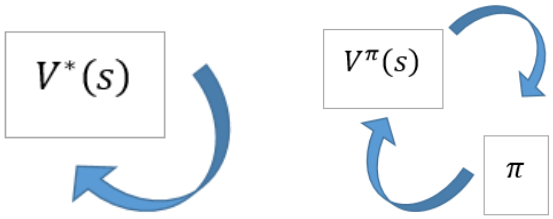
First, the problems will be run through value and policy iteration.

Value Iteration first initializes all of the state utilities to zero. Then it updates these values based on the reward at the the current state and expected value of going to nearby states. The utility values continue to be updated until they remain the same (by a significance of ϵ).

Using the expected utility at each state, an optimal policy can be obtained. Value Iteration will always converge and find the optimal solution. The expected sum of rewards is updated by the formula below.

$$V^*_{i+1}(s) \leftarrow max_a \sum_{s' \in S} T(s,a,s') \Big[R(s,a,s') + V^*_i(s') \Big] \quad \forall s \in S$$

The results from value iteration will be compared to policy iteration.



Value Iteration(left) vs. Policy Iteration(right)

Policy Iteration, on the other hand, does not evaluate the utilities at each state. This drastically will reduce the runtime of finding the optimal policy. Policy Iteration starts with random, non-optimal policies and improves them each iteration. Similarly to Value Iteration, it is guaranteed to converge (and at a much faster rate) and find the optimal solution. The expected policy to maximize long term reward is expressed below

$$V^\pi_{i+1}(s) \leftarrow \sum_{s' \in S} T(s,\pi(s),s') \Big[R(s,\pi(s),s') + \gamma V^\pi_i(s') \Big] \quad \forall s \in S$$

Using Python and iteration models from Russell's *Artificial Intelligence a Modern Approach*, objects for the problem and iteration methods have been created. Customer demand will be uniformly and independently distributed between 0 and $\frac{1}{2}k \times C$. The following constants will be set arbitrarily and consistently:

Experimental Constants		
Description	Constant	Value
unit cost	c	8
unit profit	a	10
unit warehousing cost	w	.5
base inventory size	C	10

Value and Policy Iteration will be run with the above with values of $k \in [1,5]$ to compare the two problems, $k = 1$ and $k > 1$ up to $k = 5$. The goal of these algorithms will be to maximize the expected profit over an extended time period. The expected value of each state and optimal policies will be compared as well as the amount of iterations required for the method to converge. Iterations are defined to be the number of repetitions for which all the states utilities are evaluated.

Iterative methods are effective in calculating true policy optima; however, it is unrealistic to assume we will always have such domain knowledge. Reinforcement Learning algorithms do not assume the reward or transition function of the problem and learn by balancing exploitative measures with exploitative results. For this application, Q-Learning will be focused on and compared to the iterative models. All of the above declared constants will hold.

Q-Learning is an off-policy learner for Temporal Difference Learning, meaning it learns the optimal policy independent of the actions taken. In contrast to SARSA (State-Action-Reward-State-Action) which is on-policy, Q-Learning can diverge and small changes to Q may drastically change the behavior of the method. Q-Learning is a well-known algorithm because it learns only the Q value, which means it can be easier tune. In each iteration of Q-learning, the Q function will be updated as below:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\Big[\overbrace{R(s,a,s') + \gamma \underbrace{\max_{a'} Q(s,a')}_{\text{future value}}}^{\text{learned value}} - \overbrace{Q(s,a)}^{\text{old value}}\Big]$$

The Q-learning algorithm is executed as follows through Python:

```
Initialize Q(s,a), s
Repeat:
        Choose a to take at s
        Observe r, s'
        Update Q(s,a)
        s <- s'
```

For this application of Reinforcement Learning, an ϵ-greedy policy for action selection will be used. Most of the time the action with the highest estimated reward is chosen(hence, greedy). With small probability ϵ, an action is selected uniformly random. This method ensures that if enough trials are done, each action will be tried an infinite number of times, balancing exploration with exploitation [1].

There are three main parameters to calibrate in Q learning involved in the update statement; the learning rate α, the discount factor γ, and the initial conditions Q_o.

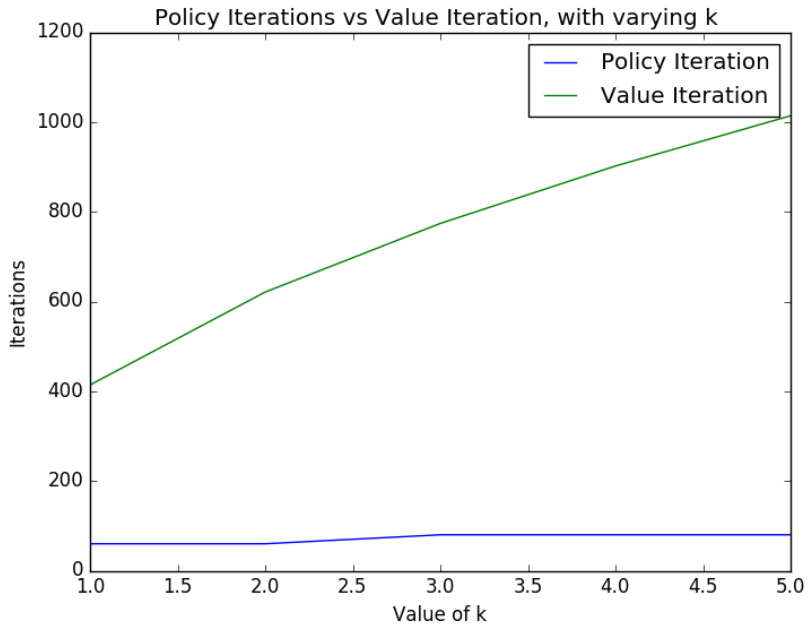
- *Learning Rate*: The learning rate will control the speed and step of which the Q value is updated.
- *Discount Rate*: The discount factor impacts the importance of future experiences. If $\gamma = 0$, the parameter is only concerned with maximizing immediate rewards. As $\gamma \to 1$, the model will tend to focus more on exploration of other state options.
- *Initial Values*: The initial values of Q must be set. If values are uniformly set high with a decently low learning rate, the model will explore more in the initial phases of the search.

These three parameters will be tunned to produce the most optimal Q-learning model for the inventory MDP with variable state space. As before, Python will be used to run the the Q-learning algorithm. The two problems will be tested, where $k = 1$ in the small state space problem and $k = 5$ in the large state space problem. The distribution of $Q(s,a)$ will be evaluated through heat-maps when the parameters are adjusted.

Analysis

Iterative Methods

The iterative methods varied significantly in efficiency. Value iteration was considerably slower as the amount of states k increased, while policy iteration was not significantly effected by the change in state size.



While both methods produced the same solution and polices (as discussed before, both methods will produce the optimal answer), it became evident that as the size of the states increased, Policy Iteration would converge to a policy much more efficiently.

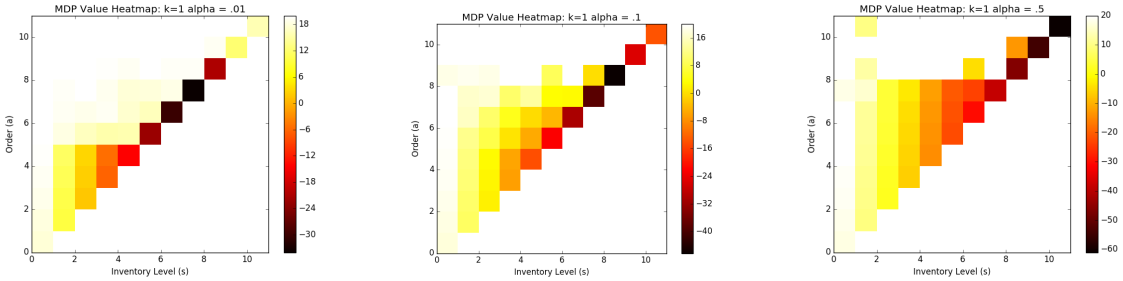
Reinforcement Learning

How effective was the Q-learning algorithm in learning the value at each state, not given the reward or transition function? This will be evaluated using a heat-map that maps the output $Q(s, a)$, and the known optimal policies through the iterative solutions. Below are the results on experiments for tuning the Q-learning parameters. The ordering decision (action) is along the y axis, and the inventory level (state) is along the x axis. Note that empty values are also represented by white space. For any value of inventory and order about $k \times C$, there is no value for $Q(s, a)$.

The optimal (s,S) policy, as determined from policy iteration, was to order when inventory got below half of the potential capacity ($\frac{1}{2}k \times C$). Anywhere above this inventory value, it was not efficient to make an order. Below this value, there should be a gradient of net positive reward; ordering will ensure a relative ordering cost, but the payoff with profit should make it a net positive decision. Ordering so that inventory is summed to $k \times C$ (this will be the linear line splitting the heat-map) will be a obvious poor decision.

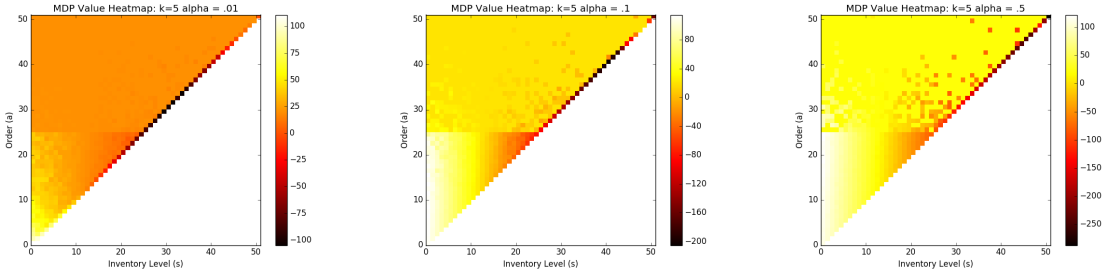
Learning Rate

Below are the results from varying alpha for the small inventory problem. From left to right are the value distribution of $Q(s, a)$ for $\alpha = .01, .1, .5$.



Increasing the learning rate effected the learned $Q(s, a)$ negatively, but increased the rate at which the solution was approached. When $\alpha = .5$, it was impossible to distinguish the reordering point, and the solution proposed that the value of ordering did not relate to the current state.

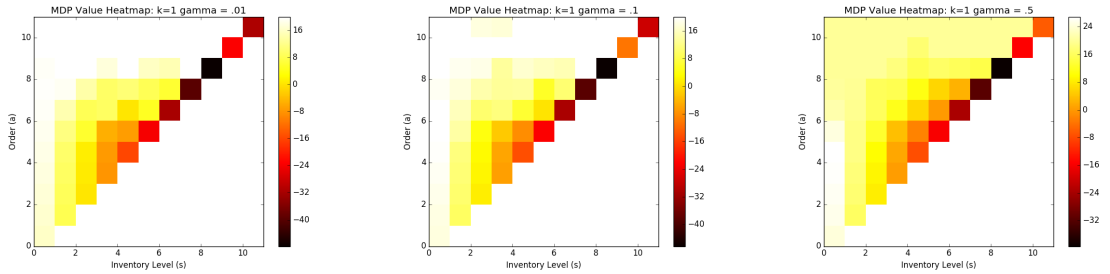
The result were similar the large inventory problem. Below are the results from varying alpha for the large inventory problem. From left to right are the value distribution of $Q(s, a)$ for $\alpha = .01, .1, .5$.



When $\alpha = .5$, the problem converged to a solution quickly. Inventory levels over $\frac{1}{2}k \times C$ were not sufficiently explored. A speckled area of negative values show that only a few points were evaluated multiple times. In contrast, $\alpha = .01$ looks like a more realistic distribution.

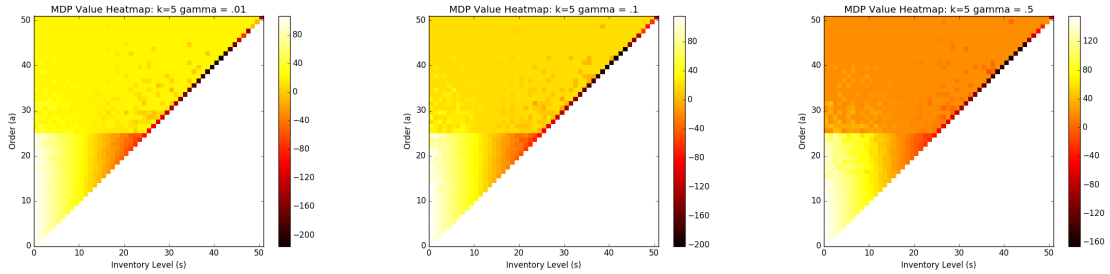
Discount Rate

The discount factor impacts the importance of future experiences. Below are the results from varying gamma for the small inventory problem. From left to right are the value distribution of $Q(s, a)$ for $\gamma = .01, .1, .5$.



Increasing the discount rate propagated the effects of large neighboring values. Since this problem did not have any large negative reward sites, changing the discount rate did not effect the problem significantly.

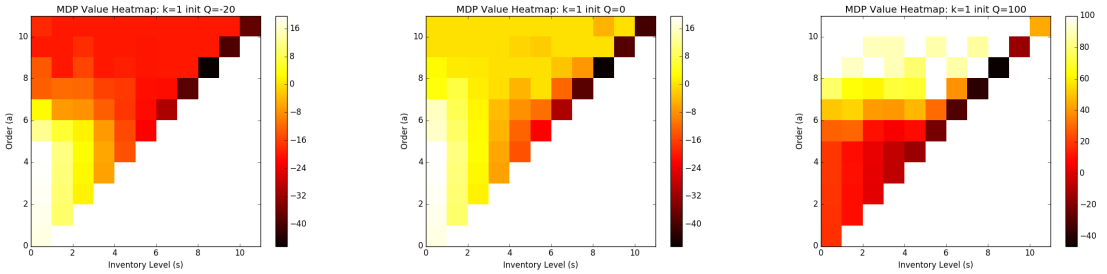
In contrast, the large inventory problem benefited in an increase in the discount rate. Values above the ordering point propagated negative, without running individual iterations for each state action pair. Because the size of space of this problem was considerably larger, increasing discount had a great impact on converging onto a solution faster. Below are the results from varying gamma for the large inventory problem. From left to right are the value distribution of $Q(s, a)$ for $\gamma = .01, .1, .5$.



Initial Q

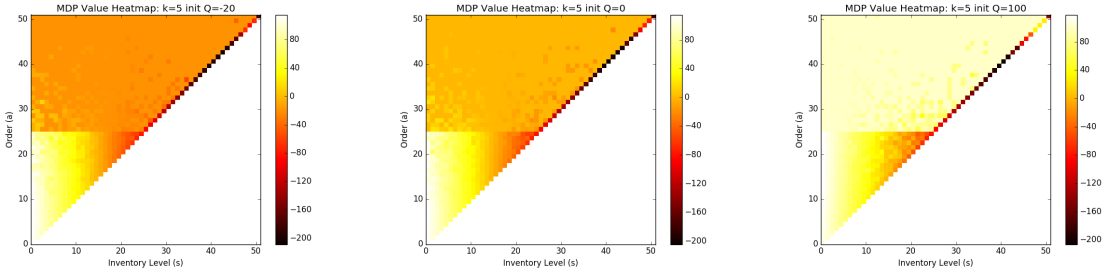
One of the most interesting parameters is how $Q(s, a)$ was initialized. By initializing all values below zero, all values were considered equally dangerous, and only by increasing expected utility at each state-action pair, was one state-action pair considered better. If initiating all values with a positive value, the model tended to be optimistic and get stuck in local maximals.

Below are the results from varying initial $Q(s, a)$ values for the small inventory problem. From left to right are the value distribution of $Q(s, a)$ for uniform initializations of $Q_o = -20, 0, 20$ where $Q(i, j) = \quad Q_o \forall i \in s, j \in a$



Changing the initialization drastically effected the small inventory problem and skewed the results. The Q matrix for $Q_o = -20$ and $Q_o = 20$ are vastly different and would propose two very different solutions.

In contrast, the large inventory problem was not severely effected by the change in Q initializations. Below are the results from varying initial $Q(s, a)$ values for the large inventory problem. From left to right are the value distribution of $Q(s, a)$ for uniform initializations of $Q_o = -20, 0, 20$ where $Q(i, j) = \quad Q_o \forall i \in s, j \in a$



Conclusion

It is evident that changing the size of the state space linearly increases the execution time and computational requirements for running value iteration and Q-learning. What is more interesting, is that parameters must be adjusted to suit the MDP size when tunning for Q-learning.

Overall, policy iteration is much more effective than value iteration. Q-learning and simple reinforcement methods are equivalently as effective, but contain much more error and required many more iterations to hope to converge to the optimal answer.

Further Exploration

As shown above, Q-learning can be easy to tune and runs relatively quickly. It can diverge with small change to input parameters. Generally, it is a state-of-the-art method because of its simplicity and ability to learn despite knowing the reward or transition. But, there are many other methods to explore in reinforcement learning:

- *SARSA*: SARSA (State-Action-Reward-State-Action) is another type of Temporal Difference learning, but uses an on-policy method. It is very similar to Q-learning, with slight improvement in run times.
- *REINFORCE*: REINFORCE (Reward Increment = Nonnegative Factor times Offset Reinforcement times Characteristic Eligibility) algorithms, in contrast, are always unbiased and tend not to diverge. They only need to learn the policy function and can be used if estimating value is difficult. While the can handle continuous action, the algorithm is very slow.
- *Actor Critic*: Actor-critic can also handle continuous action, but needs to learn two functions. Errors in one lead to errors or divergence in the other. For this reason, it can be difficult to tune.

References

[1] Eden, Tim, Anthony Knittel, and Raphael Van Uffelen. "Q-Learning." *Reinforcement Learning.* University of New South Wales Computer Science and Engineering, n.d. Web. 20 Apr. 2017.

[2] Goldberg, David A. "Inventory MDP & Machine Learning." ISYE 4232: Advanced Stochastic Systems. 21 Feb. 2017. Lecture.

[3] Ramaekers, Katrien, and Gerrit K. Janssens. "On the Choice of a Demand Distribution for Inventory Management Models." *European J. of Industrial Engineering 2.4* (2008): 479. Web.

[4] Russell, Stuart J., and Peter Norvig. *Artificial Intelligence a Modern Approach.* Third ed. Boston: Pearson, 2016. Print.

[5] Zheng, Yu-Sheng, and A. Federgruen. "Finding Optimal (s, S) Policies Is about as Simple as Evaluating a Single Policy." *Journal of Operations Research* 39.4 (1991): 655-65. Mar. 1989. Web.