Universiteit Leiden
The Netherlands

Computer Science & Mathematics

Reinforcement Learning in Inventory Management: Addressing Finite Shelf Life and Lead Time in Infinite Horizon Problems

Arnaud Saint-Genez

Supervisors:
Dr. O. Kanavetas & Dr. D.M. Pelt & Prof.dr. F.M. Spieksma

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl
Mathematical Institute (MI)
www.math.leidenuniv.nl

31/01/2024

Abstract

The efficacy of conventional reinforcement learning methods on an infinite time horizon inventory management problem with finite shelf life and a lead time for orders is evaluated. Using optimizations, the effective size of $|\mathcal{S}|$ is greatly reduced, extending the boundaries of the parameters for which these methods perform well. By further analyzing the value propagation and state transition dynamics, a heuristic approach is developed that performs near optimally after a single pass. Still running into the complexity wall, a turn to DQN is made. This is shown to perform well and scale far beyond the limits of the conventional methods.

Contents

1 Introduction

1.1 Newspapers

One of the simplest inventory management problems is the newspaper vendor problem, where a vendor has to decide how many newspapers to order for the next day at cost c, which will be sold for price p. The vendor can order any number of newspapers a (for *action*), but any unsold newspapers are worthless at the end of the day, as there will be a new edition the next day. The *reward* R (i.e. profit) in a given day will be equal to

$$R = p \cdot \min\{a, D\} - c \cdot a,$$

as the vendor cannot sell more than the inventory a, nor more than the demand D. The price for stocking the chosen amount of inventory always has to be paid.

The question is then: how many newspapers should the vendor order? Assuming the vendor knows the probability distribution of the number of newspapers sold (for example, $D \sim \text{Pois}(\lambda)$ for some $\lambda > 0$), it is possible to calculate the expected reward for any given amount a they order. In the case of a Poisson distribution, for example, the expected reward is found to be

$$\begin{aligned}
\mathbb{E}(R(a)) &= \sum_{d=0}^{\infty} [p \cdot \min\{a, D\} \cdot \mathbb{P}(D = d)] - c \cdot a \\
&= \sum_{d=0}^{a} [p \cdot d \cdot \mathbb{P}(D = d)] + p \cdot a \cdot \mathbb{P}(D > a) - c \cdot a \\
&= \sum_{d=0}^{a} \left[p \cdot d \cdot \frac{\lambda^d}{d!} e^{-\lambda} \right] + \left(1 - \sum_{d=0}^{a} \frac{\lambda^d}{d!} e^{-\lambda} \right) \cdot p \cdot a - c \cdot a
\end{aligned}$$

which, although complicated at first sight, is simply a concave function in the variable a. An example of such a function is given in Figure 1 for $p = 3, c = 2, \lambda = 5$.

In Figure 1, the optimal solution is to order $a^* = \text{argmax}_a \, \mathbb{E}(R(a)) = 4$. Numerical methods excel at finding the maximum of such a function, even managing sub-linear time for concave functions, and the problem is easily solved. In addition, because every time period is independent, the optimal value a^* is the same for every day.

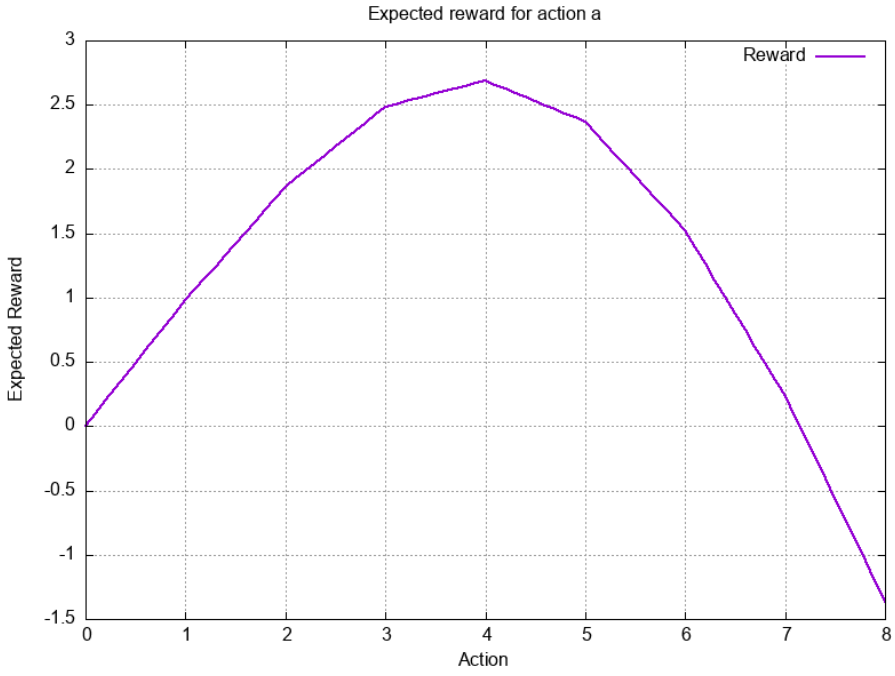


Figure 1: Expected reward for the newspaper vendor problem with $p = 3, c = 2, \lambda = 5$; due to the concave nature of the function, the optimal solution is easily found

1.2 Finite Time Horizons

Now consider a situation in which a vendor sells non-perishable products, but with a final day of sales, such as may be the case with holiday decorations. At the start of each period, the vendor may decide how many items to order, which will arrive only the next period. Any unsold goods, however, may be kept for future periods up to but excluding the final, terminal period T at a holding cost of h per period per product, after which they are worthless.

At the start of each time period, the vendor must now consider their current inventory I before deciding how many items to order. With a well-stocked inventory, ordering a lot of items may leave the vendor with large holding costs. With a low inventory, however, ordering too few items may lead to lost sales. As the terminal period T approaches, the amount of time to sell leftover inventory also decreases, potentially leaving the vendor with unsold goods.

The value of carrying a certain amount of inventory I at the start of time period t may be expressed using $V_t(I)$, which is the (discounted) maximum expected reward over the remaining time periods starting with an inventory I, assuming proper inventory management. At time period T, it holds that $V_T(I) \equiv 0$ for all inventory levels, as no more sales can be made. $V_{t-1}(I)$ may now be defined recurrently as follows:

$$V_{t-1}(I) = \max_a \left[\sum_{I'=0}^{\infty} \mathbb{P}(I'|I,a) \left(R(I,a,I') + \gamma V_t(I') \right) \right], \tag{1}$$

where $\mathbb{P}(I'|I,a)$ is the probability of ending up with I' items in inventory at the start of time period t given an inventory of I at the start of time period $t-1$, and subsequently ordering a items. $R(I,a,I')$ is the (expected) reward for transitioning from inventory I to inventory I' after ordering a items. Future profits may be discounted using γ, which may also be left at $\gamma = 1$ for now.

In most cases, a maximum inventory $I_{\max}$ will be imposed, either due to real-life constraints or for computational reasons. Because the set of all possible inventories is now finite, in addition to the finite number of time periods T, this problem is now exactly solvable using dynamic programming as in Equation 1.

This in turn gives rise to a *policy* $\pi_t(I)$, which maps an inventory I at the start of time period t to the number of items a to order. The optimal policy $\pi_t^* : I \mapsto a^*$ is the policy that maximizes the right-hand side (i.e. $a^* = \text{argmax}_a$) of Equation 1. Following this policy will maximize the expected long-term reward (profit) for the vendor and corresponds to ideal inventory management.

Evaluating this policy in the case described above, while imposing $I_{\max} = 15$, with the parameters $T = 16$, $p = 10$, $c = 6$, $h = 2$ and $\lambda = 8$, the policy can be showcased as is in Figure 2.

The behaviour as previously described is clearly visible: on the last time period ($t = 15$), the vendor orders zero items regardless of the inventory level, as any ordered items will be worthless in the next period. On the second-to-last time period and the preceding one ($t = 14$, $t = 13$), the vendor orders fewer items than all preceding time periods, as there are fewer periods left to sell the items, and as the inventory increases, fewer items are ordered.

For $t \leq 12$, the policy stabilizes. This behaviour will be considered next.

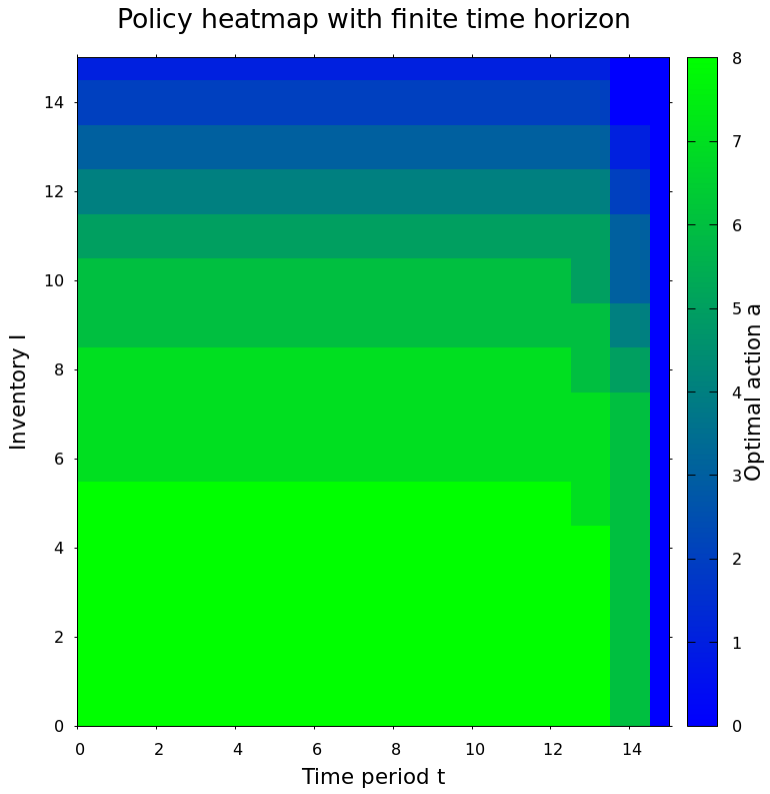


Figure 2: The optimal policy for the finite time horizon vendor problem with $I_{\max} = 15$, $T = 16$, $p = 10$, $c = 6$, $h = 2$ and $\lambda = 8$. Lighter green colors correspond to ordering more items, while darker blue colors correspond to ordering fewer items. The optimal policy is stable for $t \leq 12$, and is shown for $t = 0$ to $t = 15$.

1.3 Infinite Time Horizons

As $T \to \infty$, the optimal policy π_0^* at time $t = 0$ converges to a stationary policy π^*. This was visible in Figure 2, where π_t^* was already stable for $0 \leq t \leq 12$, or equivalently π_0^* for $T \geq 4$. The optimal policy is not necessarily unique, as multiple actions may exist with equal (optimal) expected values.

When generalizing the concept of an inventory level I to the broader concept of a *state* s in the state space $\mathcal{S}$ (assumed to be finite), such as the pair (I, t) in the previous section, with actions a in the action space $\mathcal{A}$, the value $V^*(s)$ of a state s under the optimal policy π^* satisfies the following equation:

$$V^*(s) = \max_{a \in \mathcal{A}} \left[\sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s,a) \left(R(s,a,s') + \gamma V^*(s') \right) \right]. \tag{2}$$

Equation 2 is known as the *Bellman equation*, of which Equation 1 was a special case. If any of the states are recurrent with non-zero rewards, $\gamma < 1$ is required for the above equation to be solvable. In that case, the solution is necessarily unique. Actually solving the Bellman equation is not always straightforward, and was only easily done in Section 1.2 due to the lack of cycles in the state space.

Various methods exist to numerically solve the Bellman equation. The rest of this document will discuss these and other methods in more detail on a broader class of inventory management problems.

Specifically, the primary goal is to evaluate the efficacy of these methods in finding an optimal policies for inventory models that incorporate finite shelf life and lead time within an infinite time horizon setting.

In this, the main focus will be placed on exploring and applying problem-specific optimizations to conventional methods such as policy iteration, value iteration and Q-learing, which will then be compared against a heuristic approach, as well as their ability to scale.

Finally, the ability of deep reinforcement learning to scale beyond the limits of conventional methods will be examined.

2 Extended Inventory Model

In Section 1, the concept of inventory models was introduced. The two main differentiating factors between problems were the Shelf Life (S) of inventory and Lead Time (L) of orders. In this section, a generalized model covering the broader class of these inventory models using an infinite time horizon will be introduced.

2.1 The Model

For the rest of this document, the following model will be used:

- A vendor selling a single product is considered. At the start of each time period t, the vendor may order any number of items $a \in \mathcal{A}$, which will arrive at the start of time period $t + L$ for some $L \in \mathbb{N}$, zero (instantaneous arrival) included.
- Any items that arrived at the start of time period t, yet remain unsold at the end of period $t + (S - 1)$ for $S > 0$ are discarded. $S = 0$, for computational reasons, is used to denote a product with indefinite shelf life, meaning no items are discarded unless $I_{\max}$ is exceeded.
- Products can be ordered at cost c, to be paid immediately upon ordering, and are sold at price p. The inventory may not exceed $I_{\max}$ items at any time; should items arrive while the inventory is full, the items with the shortest remaining shelf life are discarded. Items are also sold in the order they arrive, i.e. the oldest items are sold first.
- Inventory that is left over at the end of a time period and not discarded due to shelf life or exceeding $I_{\max}$ in the next time period after receiveing the order that is next due, is kept for the next time period. This incurs a holding cost of h per item per time period.
- Demand arrives during each time period t according to a Poisson distribution with parameter λ. The demand for each time period is independently and identically distributed.

As a practical consideration, the action space is limited to $\mathcal{A} = \{0, \ldots, I_{\max}\}$, as any extra items are guaranteed to be discarded. For reasons that will become apparent in Section 2.2, $I_{\max}$ is also limited to values of the form $2^n - 1$ for some $n \in \mathbb{N}$.

This model belongs to an even broader class of problems known as *Markov Decision Processes* (MDPs), the methods for solving which will be discussed in Section 3.

In implementing this model, there are some considerations to take into account beyond the simple rules of the model as outlined above.

2.2 Implementation Considerations

In all of the following sections, $\mathbb{N}_{\leq}$ will be used to denote the set of all natural numbers including zero, but less than or equal to $I_{\max}$.

The state spaces resulting from the introduction of lead time ($\mathcal{S}_L$) and shelf life ($\mathcal{S}_S$, which is simply I for $S = 0$) will also be considered seperately, resulting in a total state space of $\mathcal{S} = \mathcal{S}_L \times \mathcal{S}_S$.

2.2.1 Lead Time

The lead time L corresponds to the number of time periods between ordering and receiving inventory. For any value L (including $L = 0$), there may be L outstanding orders at any given time. Keeping track of these orders (which range from 0 to $I_{\max}$) leads to an order state space of $\mathcal{S}_L = \mathbb{N}_{\leq}^{L}$, with order states taking on the form $s_L = (O_L, \ldots, O_1)$, where O_i represents the number of items that will arrive in i time periods.

The order states may thus be stored in an array `unsigned int s[L]`, with V-values in an L-dimensional array. The methods described in Section 3, however, benefit greatly from the states having a single integer as representation too, as this allows for easy indexing of the state space. For this, it would be preferable to find an index function $i_L : \mathcal{S}_L \to \mathbb{N}$, where i_L is necessarily injective. If possible, it would also hold that $i_L(s) \in \{0, \ldots, |\mathcal{S}_L| - 1\}$ for all s_L, as this would fit exactly in an array of size $|\mathcal{S}_L|$ and be a bijection between $\mathcal{S} \leftrightarrow \mathbb{N}_{<|\mathcal{S}_L|}$, resulting in perfect memory utilization.

Due to the fact that $I_{\max} < \infty$, one such mapping is equivalent to the internal representation of the value tensor `V[I_max]...[I_max]` (with L dimensions), which is already stored linearly in memory. Seperating the inventory and outstanding orders,

the index function i_L may be defined as follows:

$$i_L(s_L) = \sum_{j=1}^{L} O_j \cdot (I_{\max} + 1)^{j-1}, \tag{3}$$

which evaluates to 0 for $L = 0$, and, with $S = 0$ for now, the general index function

$$i(s) = I + i_L(s) \cdot (I_{\max} + 1), \tag{4}$$

for $s = (O_L, \ldots, O_1, I)$, which is identical to the internal lookup methods used by processors, and in the model's case equivalent to concatenating all order and inventory fields into a single base-$(I_{\max}+1)$ number. For example, in the case of $L = 2$, $I_{\max} = 9$, the state $(O_2, O_1, I) = (2, 1, 3)$, when concatenated in base-10, would simply yield the index 213_{10}. Likewise, the same state with $I_{\max} = 15$ would yield the index $213_{16} = 531_{10}$.

Given the base-2 representation of integers in computer memory, the indices i_L of states with $I_{\max} = 2^n - 1$ for some $n \in \mathbb{N}$ are actually the values $O_L, \ldots, O_1$ stored in a bitfield. This means that, by storing order states not as an array `unsigned int S[L]` with the regular operations, but as a zero-padded bitstring embedded in an unsigned integer (such as `uint32_t` in C(++)), the state s_L may be mapped to an index $i_L(s_L)$ with a simple cast, and back. Advancing a time period no longer involves copying arrays, but simply bitshifting them by $\log_2(I_{\max}+1) = n$. Some bookkeeping has to be done for the inventory, which is unavoidable even with regular arrays.

For this reason, $I_{\max}$ will be limited to values of the form $2^n - 1$ for $n \in \mathbb{N}$ for the rest of this document.

It turns out this index has several nice properties, such as respecting the lexicographical ordering of states, meaning that $s_L <_{\text{lex}} s'_L \Leftrightarrow i_L(s_L) < i_L(s'_L)$. The fact that this holds for *all* states means it is possible to iterate over the entirery of $\mathcal{S}$ in this lexicographical order using a simple `for (uint s = 0; s < |S|; s++)`-loop, which will prove itself to be very useful in Section 4.

For $S > 0$, this becomes more complicated, but all the more important.

2.2.2 Shelf Life

For $S = 0$, the inventory may simply be stored as an integer, or equivalently `unsigned int s[1]`. In cases where $S > 0$, the array representation of inventory states is

`unsigned int s[S]`, but for $S > 1$ the state space is not exactly equal to $\mathbb{N}_{\leq}^{S}$, as the additional requirement that $\sum_k I_k \leq I_{\max}$ must hold, which means $\mathbb{N}_{\leq}^{S}$ contains many invalid states.

As a result, simply casting the base-2-embedded state to an integer does not map $\mathcal{S}_S \to \mathbb{N}_{<|\mathcal{S}_S|}$ in cases where $S > 1$. Using the stars-and-bars method, where, in addition to the $S > 1$ fields representing the inventory, an additional 'bin' to represent the unassigned inventory is added, a total of

$$|\mathcal{S}_S| = \binom{I_{\max} + (S+1) - 1}{(S+1) - 1} = \binom{I_{\max} + S}{S} = \frac{(I_{\max} + S)!}{S! \cdot I_{\max}!}$$

possible valid states are found to exist. Compared to $\mathbb{N}_{\leq}^{S}$, this quickly becomes a very small fraction of the allocated memory. For example, with $I_{\max} = 15$, and $S = 3$, a memory efficieny of

$$\frac{\binom{15+3}{3}}{(15+1)^3} \approx 0.199,$$

is found, i.e. only 20% of the memory is used to store valid states, which deteriorates very quickly as S and $I_{\max}$ increase due to the difference in complexity between the enumerator and denominator. For $I_{\max} = 31$ and $S = 4$ (i.e. one step up for each), this already drops to about 5%, with a further drop to about 1.1% for $S = 5$.

This quickly becomes the limiting factor in deciding the practical upper limit of $I_{\max}$ and S. For this reason, a comparable index function i_S is desirable, preferably respecting the lexicographical ordering as well.

To achieve this, the index of an inventory state s_S should be equal to the number of states that precede it in lexicographical ordering. Consider the following inventory state with $S = 3$ and $I_{\max} = 7$:

3	2	1

Changing the first field to either 2, 1 or 0 would, in lexicographical ordering, result in a smaller state regardless of the values of the other fields. If set to 2, up to $7 - 2 = 5$ items can be distributed over the remaining two fields to the right. Their current values of 2 and 1 are irrelevant. This means that there are $\binom{5+2}{2} = 21$ possible states of the form $(2, \ldots)$ smaller than the state above. Similarly, there are $\binom{6+2}{2} = 28$ states of the form $(1, \ldots)$ that are smaller (as there are now 6 items to distribute over the

two fields to the right).

Using this method, all of the following states must be iterated over:

2	?	?
1	?	?
0	?	?
3	1	?
3	0	?
3	2	0

while keeping in mind that in cases such as $(3, 1, ?)$ there are $7 - 3 - 1 = 3$ items that may be assigned to the single remaining field (for a total of 4 states that are smaller).

It is possible to express this for all S as

$$i_S(s_S) = \sum_{k=2}^{S} \sum_{i=0}^{I_k - 1} \binom{I_{\max} - (i + \sum_{j=k+1}^{S} I_j) + (k-1)}{k-1} + I_1 \tag{5}$$

where the outer sum goes through the fields I_k of the inventory, and the inner sum goes through the possible (lower) values i of that field. $I_{\max} - (i + \sum_{j=k+1}^{S} I_j)$ is the number of items that can be distributed over the remaining fields, and $(k-1)$ is the number of remaining fields. For $S = 0$, it holds that $I_1 = I$.

When implemented, as shown in Figure 3, the complexity is not as bad as three summations would suggest due to the fact that $\sum_k I_k \leq I_{\max}$, and the ability to accumulate $\sum_{j=k+1}^{S} I_j$ if the outer sum is evaluated from $k = S$ to $k = 2$.

In addition, one may even use the fact that $\binom{n+k-1}{k} = \binom{n+k}{k} \cdot \frac{n}{n+k}$ to reduce the number of operations even further (not shown). Furthermore, the sum evaluates to 0 for $S = 0$, as well as $S = 1$.

This provides us with the sought-after bijection i_S with respect to the number of shelf life states. The two indices may be combined to an updated and generalized

```
i_S(s):
  count = 0
  already_assigned = 0

  for k = S, ..., 2:
    for i = 0, ..., I_k - 1:
      distributable = I_max - already_assigned - i
      leftover_fields = k - 1
      count += binom(distributable + leftover_fields,
                     leftover_fields)

    already_assigned += I_k

  count += I_1  # edge case for last field

  return count
```

Figure 3: Pseudocode for the index function i_S

version of Equation 4:

$$i(s) = i_L(s_L) \cdot |\mathcal{S}_S| + i_S(s_S), \tag{6}$$

where, for any state $s = (O_L, \ldots, O_1, I_S, \ldots, I_1)$, $s_L = (O_L, \ldots, O_1)$ and $s_S = (I_S, \ldots, I_1)$ or I.

It is possible to contrive a method of inversing this index for $S \geq 2$ (with $S = 0$, $S = 1$ being trivial to inverse, as $i_S(I) = I$), although this suffers from inherent complexity issues due to the many loops required for the inverse algorithm, as the state must be built up field by field. For a one-off memory cost of $O(|\mathcal{S}|)$, which is already matched or exceeded in memory complexity by, for example, the value function, it is possible to store a reverse lookup table for i^{-1}, which can be initialized with a single pass and from then on allows for a constant-time lookup of the state s given $i(s)$.

This may be reduced even further by storing only the lookup table for i_S^{-1}, and recognizing that, under integer division, $s_L = i(s)/|\mathcal{S}_S|$, which can simply be left-shifted $n \cdot \max\{S, 1\}$ bits for $I_{\max} = 2^n - 1$, and added to $s_S = i_S^{-1}(s \bmod |\mathcal{S}_S|)$ to recover the original state s in constant time with a memory complexity of $O(|\mathcal{S}_S|)$. For $L > 0$, this is a small (often tiny) fraction of the memory required for the value function.

Last but not least, for $L > 0$, a significant reduction in computational and memory complexity can be achieved by considering which states are actually observed.

2.2.3 Observable States

Although, up until now, L values have been assigned to keeping track of the outstanding orders due to lead time, in the case of $L > 0$, the most significant value O_L will always be *observed* to be 0 in practice, as shown in Figure 4.

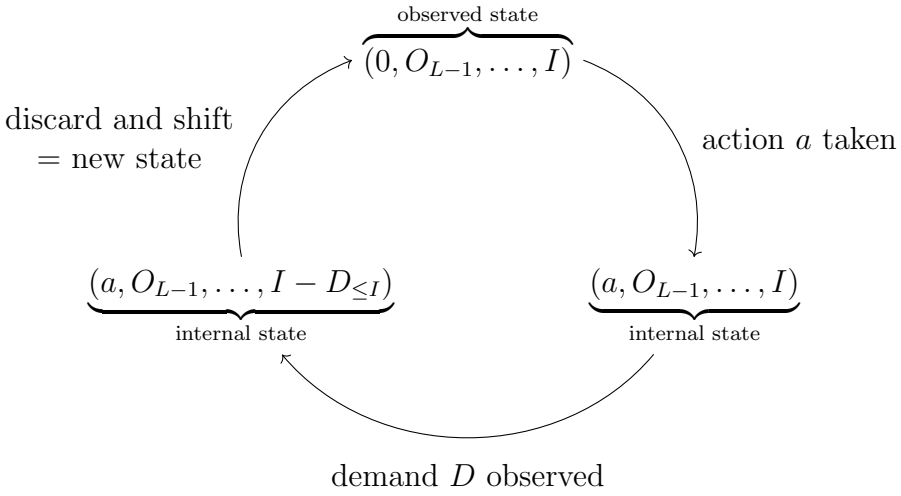


Figure 4: The internal cycle of states during every time period in the case of $L > 0$; only the observed state is encountered in practice.

Because of this, the policy for a large subset of states is irrelevant. Nothing prevents the calculation of $V(s)$ for states in which $O_L > 0$, with an associated policy selecting an action that increases the order to $O_L + a$, but as these states are never observed, and never result from taking any action a in any state s (except internally), these do not influence $V^*(s)$ and may safely be left out of the equation.

This means that, for $L > 0$, the practical order state space is reduced from $\mathcal{S}_L = \mathbb{N}_{\leq}^{L}$ to $\mathbb{N}_{\leq}^{L-1}$, resulting in a speedup of a factor $I_{\max}+1 = |\mathcal{A}|$, as well as an equal reduction in memory footprint, which in many cases will be more than an order of magnitude. This speedup is not affected by the value of S.

Internally, however, the order state space is still represented as $\mathcal{S}_L = \mathbb{N}_{\leq}^L$, as all outstanding orders, especially after taking an action, are relevant for the transition dynamics.

3 Conventional Methods

In Section 1, application of dynamic programming to the finite time horizon problem was discussed. Due to the acyclic nature of the state space, only a single pass was required to find an exact solution. The infinite time horizon inventory problem suffers from cycles, and thus requires numerical methods when using dynamic programming.

As an introduction to future sections, three of these methods will briefly be discussed in this section, based on the work of Sutton and Barto [6]: two dynamic programming methods (*policy iteration* and *value iteration*), each of which requires knowledge of the distributions and rewards, as well as *Q-learning*, which learns through interactions with the model and requires no knowledge.

The function that maps a state s to the set of all states that can be reached from s in one time step after taking action a, or in other words, the *forward states* of s under a such that $\mathbb{P}(s'|s,a) > 0$, will be denoted as $\text{fw}(s,a) : \mathcal{S} \to \mathcal{P}(\mathcal{S})$.

Due to the different complexities of the methods, the number of iterations will not be representative of the actual computational complexity. Instead, the number of array references (be it Q-values or V-values) during the iterations will be used as a measure of complexity, as this accurately represents the number of operations required, properly scaling with the different parameters.

The default parameters, unless otherwise specified, are $p = 10$, $c = 6$, $h = 1.5$ and $\lambda = \frac{|\mathcal{A}|}{2} = \frac{I_{\max}+1}{2}$, with a discount rate of $\gamma = 0.99$. Values (such as the V-table) are also initialized to zero.

Although γ must be < 1, higher values of γ will result in a policy that maximizes the total reward over a longer time horizon, albeit at a slower rate of convergence. Lower values of γ lead to greedy behaviour (rewards 'now' > rewards 'later'), which may manifest itself in unwanted ways. For example, under the default parameters, the maximum revenue for an ordered item is $p = 10$, but for $L > 4$ the present value of this revenue is less than $0.9^5 \approx 0.59$, which means the optimal policy is to never order anything, even though a reward of $p - c = 4$ may potentially be obtained.

A good approximation for the average performance of the optimal policy under these parameters will be $\approx 75\% \cdot \lambda(p - c) = 3\lambda$. This approximation will show itself to be quite accurate in the following sections.

3.1 Policy Iteration

Policy iteration solves the Bellman equation (Equation 2) by iteratively improving upon a policy π_k until convergence. Given any policy π_k, one may calculate the value function $V^{\pi_k}(s)$ for all states $s \in \mathcal{S}$ by iteratively applying the following update rule

$$\forall s \in \mathcal{S}: \quad V^{\pi_k}(s) \leftarrow \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s, \pi_k(s)) \left(R(s, \pi_k(s), s') + \gamma V^{\pi_k}(s')\right) \tag{7}$$

until sufficient convergence is achieved, often taken to be the point where the largest update difference Δ is smaller than a certain threshold ε. This phase is also known as *policy evaluation*.

One may then proceed to determine an updated policy π_{k+1} by running through the whole state space once more, also known as *policy improvement*, using the update rule

$$\forall s \in \mathcal{S}: \quad \pi_{k+1}(s) = \operatorname*{argmax}_a \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s, a) \left(R(s, a, s') + \gamma V^{\pi_k}(s')\right). \tag{8}$$

In practice, only $s' \in \mathrm{fw}(s, a)$ have to be considered, as $\mathbb{P}(s'|s, a) = 0$ for all other s'. In implementation, the associated rewards and probabilities can often be calculated efficiently while determining the forward states.

It is a known result that the chain

$$\pi_0 \to V^{\pi_0} \to \pi_1 \to V^{\pi_1} \to \ldots \to \pi_k \to V^{\pi_k} \to \ldots$$

converges to the optimal policy π^* and value function V^* in a finite number of steps for finite MDPs. For the case $L = 2$, $S = 0$ and $I_{\max} = 15$ this process can be seen in Figure 5 using the default parameters.

In the inventory management model, $|\mathrm{fw}(s, a)|$ is bounded by $I_{\max} + 1 = |\mathcal{A}|$, resulting in a complexity of $O(|\mathcal{S}| \cdot |\mathcal{A}|)$ for policy evaluation, and $O(|\mathcal{S}| \cdot |\mathcal{A}|^2)$ for policy improvement. The total complexity depends on the choice of ε.

For the case of $L = 2$, $S = 3$ and $I_{\max} = 15$, the number of array references required for policy iteration to converge for a logarithmically wide range of ε values is shown in Figure 6.

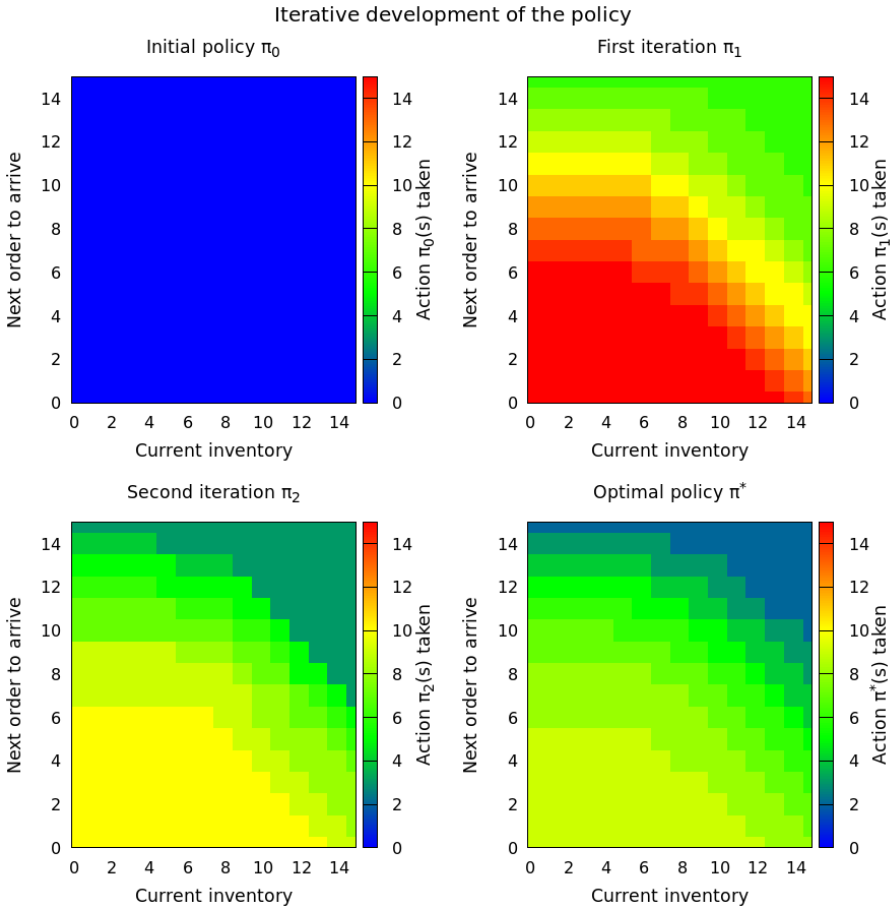


Figure 5: The initial policy with $L = 2$, $S = 0$, as well as the policy after 1 and 2 policy improvements, in addition to the optimal policy. Blue corresponds to ordering zero items, green/yellow to ordering around 8 items, and red to ordering the maximum amount of items.

In total, these differ by no more than a factor of 4, with the optimal value lying somewhere between $\varepsilon = 10^{-1}$ and $\varepsilon = 10^{-2}$. Note that, for ϵ large enough, the policy upon termination of the algorithm may not be optimal. Because of this, $\varepsilon = 10^{-4}$ will be used whenever full convergence is required for tests depending on the optimal policy.

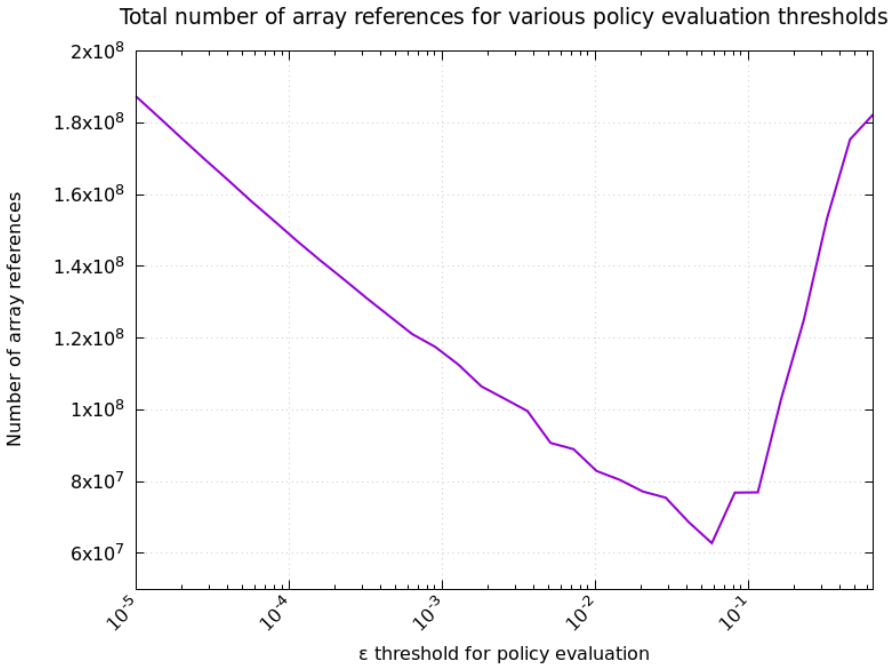


Figure 6: The number of array references required for policy iteration to converge for different values of ε with $L = 2$, $S = 3$ and $I_{\max} = 15$. Note the logarithmic x-axis, as well as the fact that convergence to the optimal policy is not guaranteed for large ε.

3.2 Value Iteration

Like policy iteration, value iteration is a numerical method for iteratively calculating the value function $V^*(s)$ for all states $s \in \mathcal{S}$, but instead of periodically updating the policy, the optimal action is evaluated on the fly. This is known as *value iteration*. The update rule is given by

$$\forall s \in \mathcal{S} : V(s) \leftarrow \max_{a \in \mathcal{A}} \sum_{s' \in \mathrm{fw}(s,a)} \mathbb{P}(s'|s,a) \left(R(s,a,s') + \gamma V(s')\right) \tag{9}$$

until sufficient convergence ($\Delta < \varepsilon$) is achieved. This approaches the fixed point of Equation 2 iteratively.

The policy may be evaluated at any time using the same method as in Equation 8.

Compared to policy iteration, value iteration has a complexity of $O(|\mathcal{S}| \cdot |\mathcal{A}|^2)$ as the optimal action has to be evaluated for every state in every iteration. Consequently,

value iteration converges in fewer iterations than policy iteration, but each iteration is more expensive.

3.3 Q-Learning

Whereas the dynamic programming methods described above require knowledge of the transition probabilities $\mathbb{P}(s'|s,a)$ and the associated rewards $R(s,a,s')$ to evaluate the best action, *Q-learning* does not. To do this, all state-action values $Q(s,a)$ are stored, which represent the future value of a state after taking the specific action a, and improved upon through repeated observations. After choosing any action a in state s, the value $Q(s,a)$ is updated according to the rule

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha \left[R(s,a) + \gamma \max_{a' \in \mathcal{A}} Q(s',a') \right], \tag{10}$$

where α is the learning rate, and s' is the state reached after taking action a in state s. Essentially, an exponentially weighted moving average is taken of the observed reward $R(s,a)$ and the discounted value $V(s')$ of the next state s' (calculated using the known $Q(s',a')$-values). Given enough iterations, this will converge to the optimal value.

A common policy, as will be used in this case, is

$$\pi_\varepsilon(s) = \begin{cases} \operatorname{argmax}_a Q(s,a) & \text{with probability } 1-\varepsilon \\ \text{random action} & \text{with probability } \varepsilon \end{cases} \tag{11}$$

where ε is a parameter that determines the probability of taking a random action for the purpose of exploration, commonly equal to 0.1 or 0.2. During performance evaluation, ε will be set to nearly 0, which chooses the best-known action almost every time, but prevents the algorithm from potentially remaining in the zero state.

The memory complexity of Q-learning is $O(|\mathcal{S}| \cdot |\mathcal{A}|)$, as the Q-values must be stored for every state-action pair. The computational complexity is variable, depending on the amount of convergence required. The most visited state-action pairs will be updated many orders of magnitude more often than the least visited ones.

Full convergence of the Q-values is generally not required for good performance, and often also computationally infeasible. Because of this, some state-action pairs may be

visited rarely (or even never), and the Q-values for these pairs after any reasonable finite amount of iterations may differ significantly from the true values. Checking policy equality between Q-learning and the dynamic programming methods will thus result in a negative result with a probability of practically 1 in all but the most simple models.

A brief comparison between the performance of the three methods for a given amount of array references will be made in the next section, showcasing the previously mentioned dynamics.

3.4 Comparison

For reasons that will become apparent in Section 4, the order in which the states are updated is important. In Section 2.2, the lexicographical ordering of the states was discussed. For now, in the case of value and policy iteration, the states will be updated from the highest to the lowest index. The impact of this will be discussed later on.

Figure 7 shows the performance of the three methods up to 10^8 array references using $I_{\max} = 15$, $L = 3$ and $S = 2$. The fact that policy iteration improves in larger steps as the policy is updated, in addition to the larger number of iterations required per improvement is clear.

In addition, it should be noted that Q-learning iterates faster. While the number of array references is used as a proper measure of complexity, this may differ by a linear factor in actual performance between methods, especially between Q-learning and value/policy iteration. As such, Q-learning may seem to significantly underperform based on the array references, while being more comparable in actual performance when Q-learning is allowed to run for the same amount of processor cycles, which may for example result in $2 \cdot 10^8$ or $3 \cdot 10^8$ references.

The convergence to $\approx 3\lambda = 3 \cdot 8 = 24$ is also visible.

In the next section, the deeper dynamics of transitions, convergence and information propagation will be considered, with the ultimate goal of deriving a method that outperforms the conventional methods discussed above.

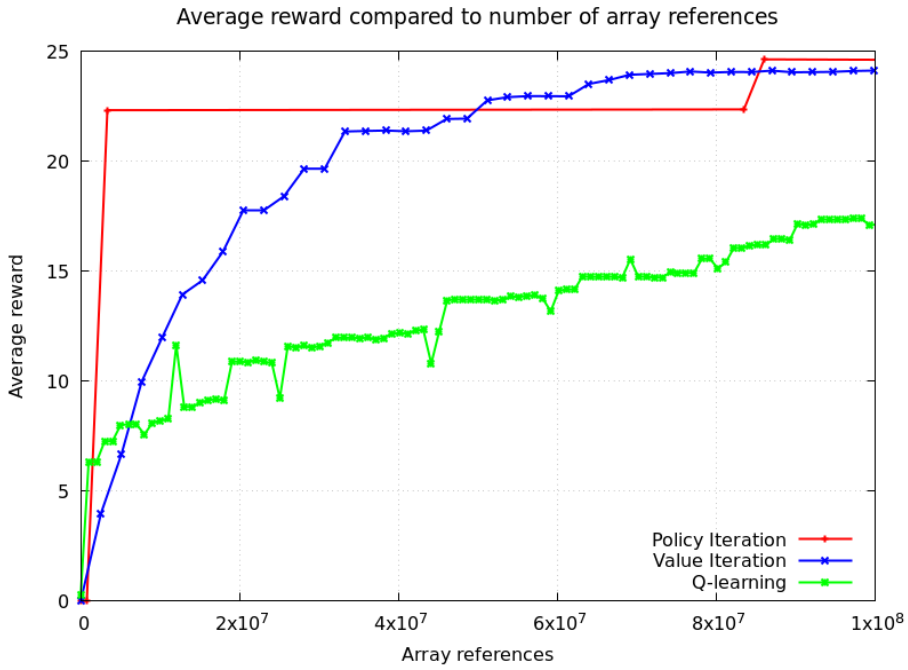


Figure 7: The performance of policy iteraton, value iteration and Q-learning for $I_{\max} = 15$, $L = 3$ and $S = 2$. The discrete nature of policy iteration is clearly visible, compared to the more continuous nature of value iteration and Q-learning. Note that Q-learning requires less time per array reference and may thus perform better than shown when given the same compute budget. Based on the rule of thumb, optimal performance is expected to be ≈ 24.

4 Heuristical Optimization

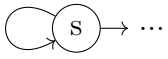
The methods described in the previous sections are all general methods developed for solving any Markov Decision Process. In some cases, however, knowledge of the problem may be used to significantly improve performance, be it in the rate of convergence or even a reduction in complexity.

Two of these methods - disregarding the states that are never observed, and mapping $\mathcal{S} \to \mathbb{N}_{\leq |\mathcal{S}|}$ - have already been discussed in Section 2.2. For $L > 0$, this has resulted in a speedup of a factor $I_{\max} + 1 = |\mathcal{A}|$ for dynamic programming methods with an equal reduction in memory for all methods, and a massive reduction in memory footprint for $S > 0$, also for all methods. In this section, the transition dynamics within the state space will be analyzed in an attempt to further improve performance.

4.1 Overcoming Cycles

Circular dependencies, that is to say $\mathbb{P}(s_t = s|s_0 = s) > 0$ for some $t \geq 1$, were the root of the problem when trying to solve cyclic graphs using dynamic programming. In section 3, this problem was solved numerically by iterating through the state space until the values had converged 'enough'.

In transient cases (i.e. $\lim_{t\to\infty} \mathbb{P}(s_t = s|s_0 = s) = 0$), however, it may still be possible to solve the problem analytically in simple situations. For example, consider the following simple state space graph for a transient state s:



Assuming an action has already been chosen for now, the value of s is given by the Bellman equation 2 as

$$\begin{aligned} V(s) &= \sum_{s'} \mathbb{P}(s'|s) \left(R(s,s') + \gamma V(s')\right) \\ &= \sum_{s' \neq s} \mathbb{P}(s'|s) \left(R(s,s') + \gamma V(s')\right) + \mathbb{P}(s|s) \left(R(s,s) + \gamma V(s)\right) \\ &= \frac{\mathbb{P}(s|s)R(s,s) + \sum_{s' \neq s} \mathbb{P}(s'|s) \left(R(s,s') + \gamma V(s')\right)}{1 - \gamma \mathbb{P}(s|s)} \end{aligned} \tag{12}$$

which does not depend on the value of $V(s)$ anymore. Recoginizing the geometric series (as $\gamma\mathbb{P}(s|s) < 1$), this may even be written as

$$= \sum_{n=0}^{\infty} \gamma^n \mathbb{P}(s|s)^n \left[\mathbb{P}(s|s)R(s,s) + \sum_{s' \neq s} \mathbb{P}(s'|s) \left(R(s,s') + \gamma V(s') \right) \right]$$

to see that this indeed sums over every possible number of times one may remain in s and collect $R(s,s)$, after which one moves on to a different state s', never to return.

Equation 12 may thus be used to solve for $V(s)$ if $V(s)$ is transient with the only path to itself being a trivial loop. As a final necessary remark, Equation 12 also works for a recurrent state with the only path being a simple loop ($\mathbb{P}(s|s) = 1$), as this results in

$$V(s) = \frac{1 \cdot R(s,s) + 0}{1 - \gamma \cdot 1} = \frac{R(s,s)}{1-\gamma} = \sum_{n=0}^{\infty} \gamma^n \cdot R(s,s),$$

which is well-defined for $\gamma < 1$, and once again corresponds to the future discounted rewards.

For some types of more complicated loops (longer paths, multiple paths, etc.) it is possible to extend equation 12, although this becomes significantly more complicated when there exists a cycle that bypasses s entirely. In this case, the above will suffice.

4.2 State Space Analysis

Given that some simple cases may be solved analytically, it is worth investigating whether the state space of the inventory management problem has any such cases.

First, consider the case where $L = S = 0$. In this case, the state space is simply $\mathcal{S} = I = \mathbb{N}_{\leq}$. If the action $a = 0$ is taken (nothing is ordered), the situation as described in Section 4.1 occurs, where the state s is transient with the only path to itself being a trivial loop in the case of zero demand, except for the zero state, which is recurrent with only a path to itself. This has been visualized in Figure 8.

It is thus possible to calculate the intrinsic value of the inventory (that is to say, what would happen if one were to stop ordering and only sell the current inventory, which is not $V^*(s)$) for all s analytically using equation 12 and dynamic programming,

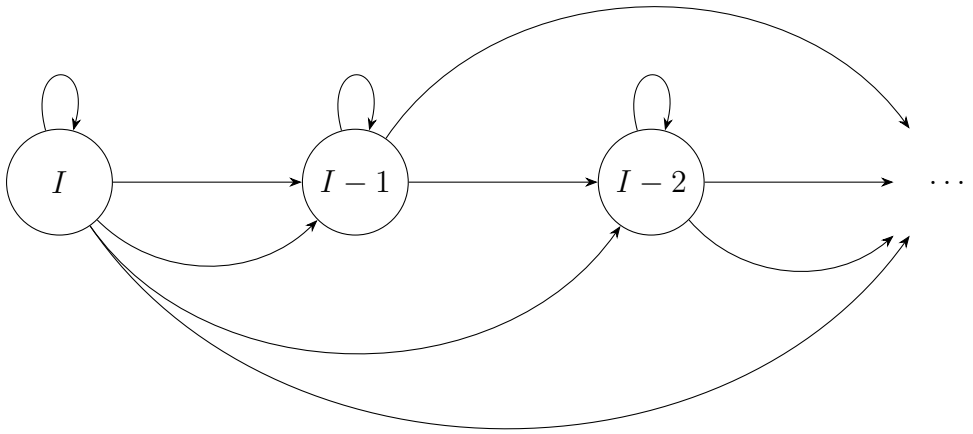


Figure 8: State space transitions for $L = S = 0$ under $a = 0$. All states, except the zero state, are transient with the only path to itself being a trivial loop. The zero state is recurrent with a trivial loop to itself.

starting from $I = 0$ and working up to $I = I_{\max}$.

Due to the FIFO nature of the model, it is now possible to do the reverse: given that the net present value of the current inventory I is known, as well as the net present value of the inventory $I + a$ for any $a \geq 0$, a can be chosen in such a way that the inventory value of the next state, minus the associated cost of $a \cdot c$, is maximized. This will be considered further after handling the lead time and shelf life.

Having considered $L = S = 0$, the case with $L > 0$, $S = 0$ may be considered next. This results in a state space of $\mathcal{S} = \mathcal{S}_L \times I$. To reason about the transition dynamics, it is easiest to look at only the classes of states as far as transitions are concerned, disregarding the actual values. These classes are represented by binary strings of length $L + 1$, with a 1 representing the presence of a non-zero order or inventory, and a 0 representing the absence of an order or any inventory. For example, with $L = 2$, the state $(O_2, O_1, I) = (3, 0, 1)$ would become 101.

Due to the right-shift that happens every time period (with a possible carry of the least significant bit if there is any inventory left over), the state space transitions under the 0-action closely resemble a binary tree, except for the potential carry.

For $L = 3$, while disregarding the unobservable states, this has been drawn in Figure 9.

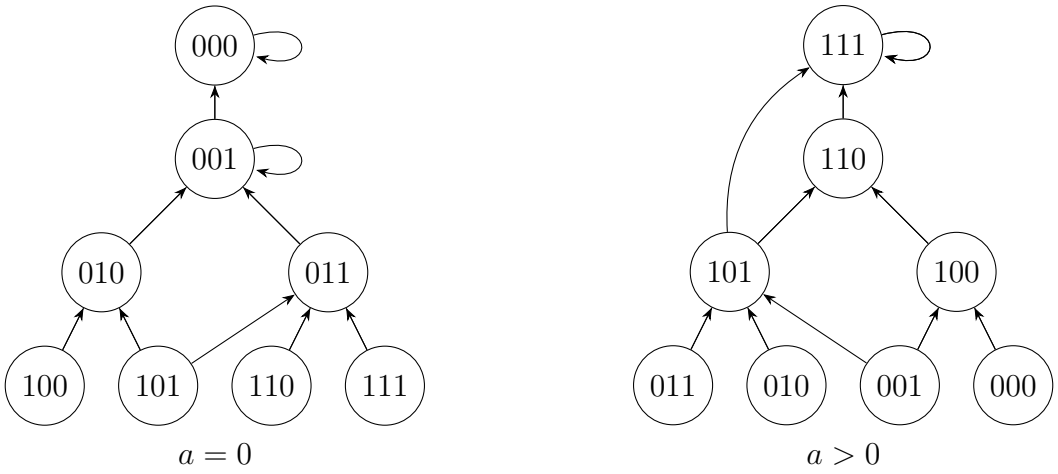


Figure 9: Transition graphs for the state classes with $L = 3$, $S = 0$. On the left, the transition graph under $a = 0$, and on the right the transitions under $a > 0$. The upper (unobservable) states `1000`, ..., `1111` have been omitted.

In the figure it can be seen that the state space forms a directed acyclic graph under the action $a = 0$, with the exception of the `000` and `001` nodes. The dynamics in these nodes corresponds exactly to the $L = 0$ case, as nothing is ordered, which is again equivalent to the case shown in Figure 8.

Because the first two nodes (`000` and `001`) are known to be solvable, with the rest of the state space forming a directed acyclic graph, it is once again possible to solve the entire state space analytically using dynamic programming. This may be extended to any arbitrary value of L, as the right-shift guarantees a smaller most-significant non-zero bit, except in the first two nodes, which have been covered seperately.

It is important to note that, under the zero action, the acyclicality of all nodes aside from the first two is independent of the inventory dynamics. This means that, irrespective of S, if $O_i > 0$ for some i, the fact that $i(s') < i(s)$ holds for all $s' \in \text{fw}(s, 0)$. If the first two nodes are solvable, this implies that the rest of the state space is too.

The next question is whether $S > 0$ influences the solvability of the first two nodes. Because $\mathcal{S} = \mathcal{S}_L \times \mathcal{S}_S$, the transition dynamics of the inventory states occur in parallel to the transition dynamics of the order states. For $S = 3$ these dynamics (now only

for $\mathcal{S}_S$) have been drawn in Figure 10.

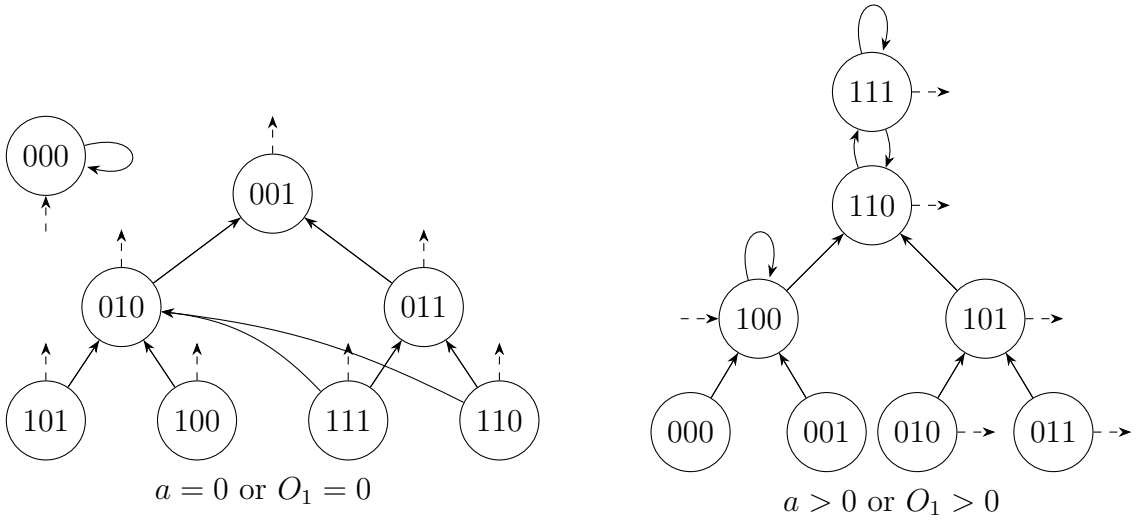


Figure 10: Transition graphs for the inventory state classes with $S = 3$. On the left, the transition graph under $a = 0$ or $O_1 = 0$, and on the right the transitions under $a > 0$ or $O_1 > 0$. The upper (unobservable) states `1000`, ..., `1111` have been omitted. All dashed outgoing arrows correspond to the single incoming dashed arrow.

The way to interpret this in the broader state space $\mathcal{S}$ is as a subgraph replacing the least significant bit in Figure 9, with the order state transitions still applying.

For $L \in \{0, 1\}$, with the exception of `000` (which contains only the zero state), all other states form a directed acyclic graph under the zero action. This is once again due to the right-shift, this time without the ability to carry the least significant bit to the next state. The zero state, once again, is known to be solvable.

Because an outstanding order may arrive even after taking the zero action for $L > 2$, the situation on the right may occur even after taking $a = 0$. Recall, however, only the solvability of the first two nodes in Figure 9 was required to solve the problem. These two nodes contain no outstanding orders, which means that the transition dynamics on the left of Figure 10 still apply in these nodes for $a = 0$, and that these are once again solvable, as $i(s') < i(s)$ for all $s' \in \mathrm{fw}(s, 0)$ and all $s \in \mathcal{S}$ such that $s \neq 0$, regardless of the value of L, with $s = 0$ already being solvable. For $L > 2$, this is because, even if the inventory substates may contain cycles under the zero action, the order substates are decreasing in index. Referring back to

the index function in Equation 6, this always results in a lower index irrespective of $\mathcal{S}_S$.

The next thing to wonder is what use the solvability of the discounted value of the current inventory is. For $L = S = 0$, it was previously discussed that this allows a to be chosen such that the inventory value of the next state (minus the cost of ordering) is maximized. Defining '$s + a$' as the state that results from taking action a, but still within the same time period, which for $L > 0$ and $a > 0$ is one of the previously discussed unobservable states, with $s + a$ being the state on the bottom right in Figure 4, it may more generally be stated that the optimal state-action value function $Q^*(s, a)$ for the inventory model satisfies

$$Q^*(s, a) = Q^*(s + a, 0) - a \cdot c, \tag{13}$$

as the action incurs an immediate cost of $a \cdot c$, which 'lifts' the current (intermediate) state to $s + a$, from where demand will be observed (corresponding to selling the current inventory without taking any other action). It has now become more apparent why, even though these states are not 'observable' for $L > 0$, the V^*-values are well-defined. In addition, because $\text{fw}(s+a, 0) = \text{fw}(s, a)$ and $r(s+a, 0, s') - ac = r(s, a, s')$, these states still do not have to be considered in the implementation, preserving the previously obtained computational and memory savings for $L > 0$.

Using this, for all L, S, the state-action value function $Q^0(s, a)$, representing the present value of $s + a$ after paying $a \cdot c$ for the order, is given by

$$Q^0(s, 0) = \frac{\mathbb{P}(s|s, 0)R(s, 0, s) + \sum_{s' \neq s} \mathbb{P}(s'|s, 0) \left(R(s, 0, s') + \gamma Q^0(s', 0)\right)}{1 - \gamma \mathbb{P}(s|s, 0)}, \tag{14}$$

$$\begin{aligned} Q^0(s, a) &= Q^0(s + a, 0) - ac \\ &= \sum_{s' \in \text{fw}(s,a)} \mathbb{P}(s'|s, a) \left(R(s, a, s') + \gamma Q^0(s', 0)\right). \end{aligned} \tag{15}$$

As previously discussed, because $\mathbb{P}(s'|s, 0) > 0$ and $s' \neq s$ implies $i(s') < i(s)$, with $Q(0, 0)$ also being well-defined, it follows that $Q^0(s, 0)$ is well-defined for all $s \in \mathcal{S}$, which in turn means $Q^0(s, a)$ is well-defined for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$. The initial policy π^0 is then given by

$$\pi^0(s) = \underset{a \in \mathcal{A}}{\text{argmax}}\, Q^0(s, a). \tag{16}$$

Q^0 may be evaluated using a single pass of dynamic programming from the lowest index to the highest, which has a complexity of $O(|\mathcal{S}| \cdot |\mathcal{A}|)$. From there, a single pass

of policy improvement may be performed to obtain π^0, with the usual complexity of $O(|\mathcal{S}| \cdot |\mathcal{A}|^2)$.

This yields a policy that, from any state, takes the action that maximizes the future discounted rewards, taking the immediate cost of the action into account. As a result of the definition of $Q^0(s, a)$, this goes beyond optimizing the next time period's expected immediate reward and takes into account the future rewards (and losses, such as the probability of discarding inventory) over multiple time periods.

It should be noted that this policy is not necessarily optimal. For one, the introduction of a new factor such as a fixed cost associated with placing an order easily breaks the premise of optimizing the next time period's inventory value, as it may now be optimal to order a large amount of items in one go, for which a look-ahead of more than one time period is required.

Additionally, the implicitly defined value function

$$V^0(s) = \max_{a \in \mathcal{A}} Q^0(s, a)$$

does not satisfy the Bellman equation 2, as this requires *all* future rewards to be discounted. Although the policy may (potentially) still be optimal without the value function from which it is derived satisfying the Bellman equation, as long as the relative differences between the V-values are maintained, this suffers from the same kinds of problems as heuristic approaches to integer programming due to the discrete nature of the inventory model.

Once a near-optimal policy has been obtained, this may be iterated upon using the conventional methods discussed in Section 3.

Next, this policy, as well as the iteration upon this policy, will be compared.

4.3 Application

Knowing the importance of the order in which the states are updated, the heuristic performance will be compared against three variations on value iteration: one iterating from the lowest state to the highest, one iterating from the highest state to the lowest, and one iterating in a random order. The first two are deterministic, and the third

will be averaged over 20 runs, each with different evaluation orders.

The heuristic approach will first perform a single pass of dynamic programming from the lowest index to the highest, storing the values as initial values in the V-table, followed by a single pass of policy improvement, at which point the performance will be measured. This policy will then be iterated upon.

This is done using a model with $I_{\max} = 15$, $L = 3$, $S = 2$ with the default parameters. A final performance of $\approx 3 \cdot 8 = 24$ may thus be expected. Convergence is defined as $\Delta < 10^{-4}$.

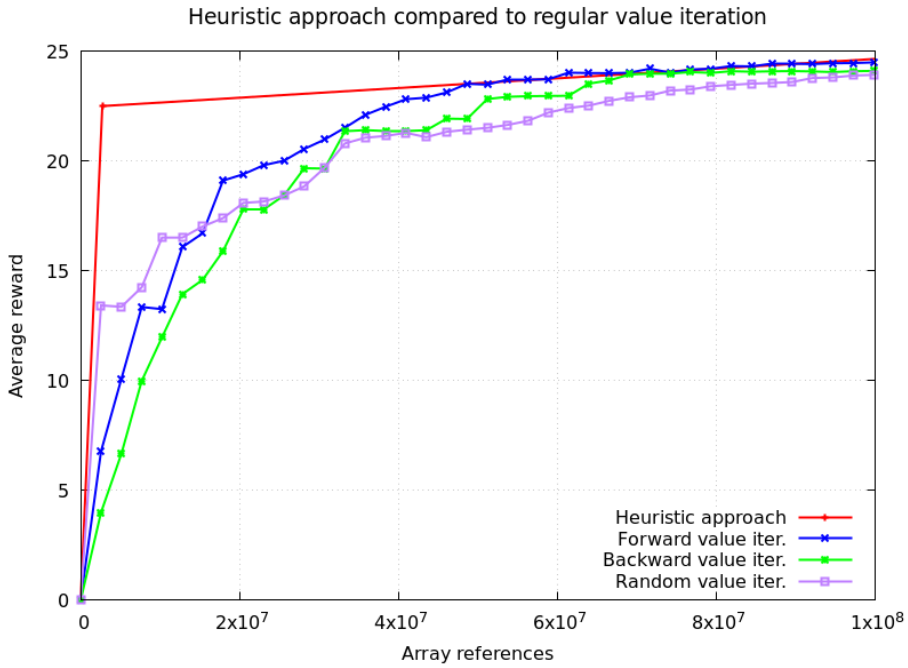


Figure 11: The performance of the heuristic approach compared to the three variations on value iteration, evaluating in forward, backward and random (averaged over 20 runs) order. The heuristic approach performs significantly better after the first pass, but still requires a large amount of policy evaluation iterations to converge to the actual optimum. This further increases performance from ≈ 22.5 to ≈ 24.9 after full convergence.

The results of this are shown in Figure 11. The heuristic approach performs significantly

better than the other three methods after the first pass, but then requires a large amount of policy evaluation iterations to converge. This is due to the fact that the initial V-values are initialized on the order of the expected final performance, i.e. 24, with some values being higher and some being lower. The final V-values, being the discounted future rewards, however, lie around

$$\frac{24}{1-\gamma} = \frac{24}{1-0.99} = 2400$$

which means the policy has to be evaluated until the values have converged to a factor 100 higher than the initial values, with a delta of 10^{-4}. The figure was cut off at 10^8 references, but about three times this amount ($\approx 3.2 \cdot 10^8$) was required to converge. The performance, however, barely improved, increasing from ≈ 22.5 using the heuristic, to ≈ 24.7 after the first (normal) policy improvement, and finally ≈ 24.9 after full convergence. This shows the efficacy of the single pass of the heuristic.

The value iteration methods also showcase the information propagation dynamics. Broadly speaking, going 'down' with regards to the index realizes rewards (which means value propagates upwards, as rewards resulting from actions are only realized after L time periods), and going 'up' accumulates *future* rewards, corresponding to ordering inventory. This is precisely the reason a single low to high pass for the heuristic performs as well as it does.

Updating the states from the highest index to the lowest index results in the realization of the reward only being evaluated after the evaluation of the action, which in the case of ordering > 0 items always leads to a leaf in Figure 9, in turn requiring L iterations for the reward to reach the this state, as it has to be propagated through the tree-like structure, which hapens one level at a time when going from the highest index to the lowest index.

Inversely, updating the states from the lowest index to the highest, which implicitly contains aspects of the heuristic, results in the fastest convergence. Due to the actual calculation of V-values instead of a policy based on the differences between discounted inventories, this is not instant and still requires gradual convergence. The value propagation, however, happens faster.

Interestingly enough, the random order actually showcases this the best. Consider a lower state s_l where value is realized, and a higher state s_h where future value is accumulated, with a set of intermediate states $\{s_i\}$ with many different connections

to s_l and s_h. A pass from low to high will instantly propagate realized value from s_l to s_h, and a high to low pass will require depth($\{s_i\}$) iterations to obtain this realized value, but then propagate it back to s_l in a single iteration when s_l is evaluated.

A random order of $\{s_i\}$, however, will contain many of the paths from $s_l \to s_h$, as well as many of the paths from $s_h \to s_l$, which means value may be propagated back *and* forth within a single iteration, possible even multiple times, resulting in very fast inital value propagation. This is visible in 11, where the random order outperforms both other orders by a wide margin after a single iteration.

As the values converge and the policy starts to stabilize, however, the individual paths start mattering more as the vast majority will not be taken. Because these individual paths are evaluated in a random order, propagation in neither direction is instant, resulting in the worst possible performance. This is also visible in the figure, as the random order consistently underperforms the other two once more convergence iterations have been performed.

Although the figure may suggest that random order converges to a lower value, it does eventually converge after $\approx 1.6 \cdot 10^9$ references. This does not differ significantly from the backward pass ($\approx 1.5 \cdot 10^9$), but both are slower than the forward pass ($\approx 0.97 \cdot 10^9$).

Last but not least, this behaviour may be explicitly shown by performing policy evaluation a varying number of times with a policy $\pi \equiv 0$ (preventing inter-state value development), ranging from one iteration to 10. This is done from the highest index to the lowest. After performing this number of iterations, policy improvement is performed, and the resulting policy's average performance is evaluated. This is restarted for every tested number of iterations. Doing this, in turn, for $L = 2, \ldots 5$ yields Figure 12.

Here, the fact that it requires L iterations for the reward to reach the leaf states is clearly visible, after which a very well-performing policy is obtained. Because $\pi \equiv 0$, this is very similar to the heuristic aside from some minor details such as not handling self-transitions and requiring L iterations instead of one.

Having considered the value dynamics and the resulting heuristic approach, the limit of what is practically possible with conventional methods has been reached. Convergence using the parameters used above takes minutes in the worst cases, which will blow up very quickly as $I_{\max}$, L and S are increased. Because of this, a neural

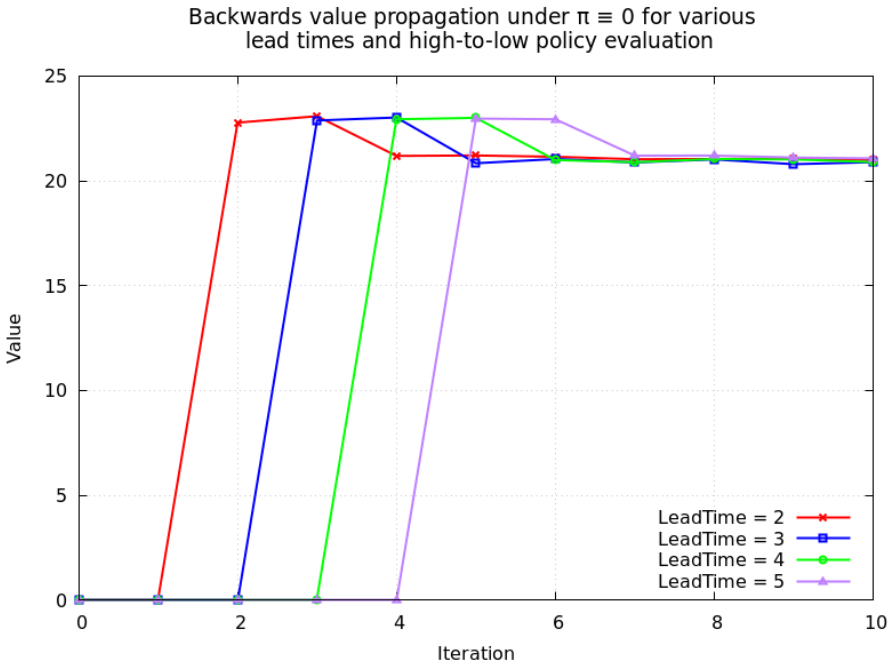


Figure 12: The performance of policy iteration after a varying number of top-to-bottom policy evaluation iterations, with the policy being evaluated using a policy $\pi \equiv 0$. After performing this number of iterations, policy improvement is performed, and the resulting policy's average performance is evaluated. Various values of L are compared. The arrival of information in the leaf states after precisely L iterations is clearly visible, resulting in a sudden surge in performance.

approach will be considered next.

5 Deep Reinforcement Learning

The step from exact methods like policy iteration and value iteration into more random methods like Q-learning may be taken further by taking a small step into the field of deep learning. Instead of storing exact values for every state-action pair, a neural network may be used to approximate these Q-values.

Deep Q-networks (DQN) are one such method, and a brief look into their performance will be given in this section.

5.1 Motivation

The methods discussed in Section 3 had various complexities. Policy iteration had a complexity of $O(|\mathcal{S}| \cdot |\mathcal{A}|)$ for policy evaluation, and $O(|\mathcal{S}| \cdot |\mathcal{A}|^2)$ for the occasional policy improvement, although this required more total iterations than value iteration, which had a complexity of $O(|\mathcal{S}| \cdot |\mathcal{A}|^2)$ for each iteration. The both of them have a memory complexity of $O(|\mathcal{S}|)$.

Q-learning no longer strictly required obervation of every state-action pair, but instead had a memory complexity of $O(|\mathcal{S}| \cdot |\mathcal{A}|)$, and a computational complexity that depended on the amount of convergence required.

Considering the size of $\mathcal{S}$, the problem starts to become clear very quickly. Seperating $|\mathcal{S}|$ into $|\mathcal{S}_L| \cdot |\mathcal{S}_S|$ again, and using the fact that $I_{\max} + 1 = |\mathcal{A}|$, the observable size of each is given by

$$|\mathcal{S}_L| = |\mathcal{A}|^{\max\{L-1,0\}}, \quad |\mathcal{S}_S| = \begin{cases} \binom{S+|\mathcal{A}|-1}{|\mathcal{A}|-1} & \text{if } S > 0 \\ |\mathcal{A}| & \text{if } S = 0 \end{cases}. \tag{17}$$

Even with $S = 0$, value iteration and policy improvement's complexity would become $O(|\mathcal{A}|^{L-1} \cdot |\mathcal{A}| \cdot |\mathcal{A}|^2) = O(|\mathcal{A}|^{L+2})$ for $L > 0$. For even small values of L, this blows up *very* quickly as $I_{\max}$ is increased. Most tests were performed with $|\mathcal{A}| = 16$ and converged in a matter of seconds for a value like $L = 3$. Increasing $|\mathcal{A}|$ to 64, however, already increases the number of steps to be taken per iteration by a factor of $4^{3+2} = 1024$, now likely requiring hours to converge. Additionally increasing S from 0 to 2 replaces one factor of 64 by $\binom{2+64-1}{2} = 2080$, or $4^{3+1} \cdot 2080 = 532480$ times as many operations as the $I_{\max} = 15$, $L = 3$ and $S = 0$ case, potentially requiring

months to converge.

Even the heuristical method described in Section 4 suffers from this, as it provided a mostly linear speedup by reducing the number of convergence iterations, but not the number of steps per iteration, with the first step still requiring forward policy evaluation, followed by an immediate policy improvement, which resulted in a strong initial policy. Further improvements on this also fall back on regular policy iteration with the associated complexity, as could be seen in Figure 11.

Recalling Figure 1 from the newsvendor problem, the behaviour of the expected future reward was actually rather simple. Although the pipeline of inventory resulting from the introduction of lead time and shelf life significantly complicates the state space, the reward dynamics remain largely the same: below a certain threshold, ordering more inventory results in a higher expected reward, until this threshold is passed and the holding costs and risk of discarding inventory start to outweigh additional sales.

In fact, a figure analogous to Figure 1 could be drawn for every state in the state space. The optimal policy is already obtained by taking the argmax of this figure for every state, and the value function is simply the maximum of this figure. For three random states with $L = 2$, $S = 3$ and $I_{\max} = 15$, this has been done in Figure 13.

The concave nature that was first observed with the newsvendor problem is still present, albeit in a higher dimensional space. Even simple neural networks are capable of approximating such a function with a high degree of accuracy, and thus it may be expected that a neural network can be used to approximate the Q-values for every state-action pair to a high enough degree.

In addition, neural networks are capable of generalizing over similar states, which was not possible with the previous methods. For example, whereas Q-learning stored and updated the state action value $Q(s, a)$ for every state-action pair seperately, a neural network may observe two Q-values $Q(s, a_1)$ and $Q(s, a_2)$ and, for some a_3 inbetween a_1 and a_2, infer the value of $Q(s, a_3)$ to be roughly between the two without ever observing $Q(s, a_3)$ directly. Given the nature of the $Q(s, a)$-curves as seen in Figure 13, this may be expected to perform well.

To make matters worse with regards to the complexity, or better with regards to the potential performance of a neural network, *very few* of the states in $\mathcal{S}$ are actually reachable under the optimal policy, even after disregarding the outright unobservable

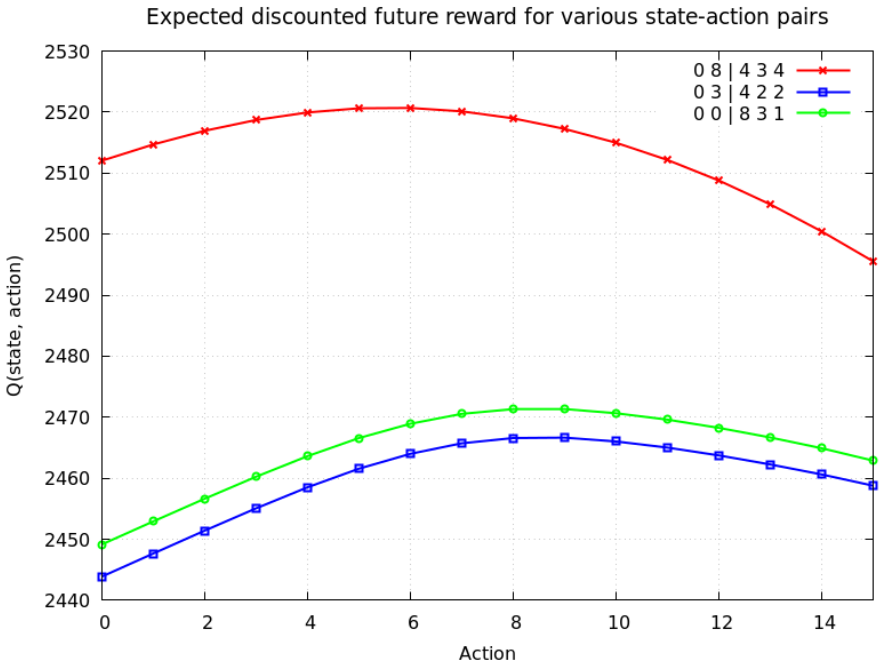


Figure 13: The Q-values for three random states with $L = 2$, $S = 3$ and $I_{\max} = 15$. The optimal action is the one that maximizes the Q-value. The simple concave nature of the Q-values, even with the addition of lead time and shelf life, is clearly visible.

states. On top of that, out of the probabilistically reachable states, the distribution of the occurrences of each state is highly skewed, with the vast majority of reachable states being visited very rarely.

This occurrence distribution (also known as stationary distribution) is shown in Figure 14 for $L = 2$, $S = 0$, $I_{\max} = 31$, based on 10^9 iterations. On the left, it can be seen that a small cluster of states represent the majority of the observed states. In fact, the top 10 states add up to $\approx 60\%$ of the total observations, with the top 20 representing more than 70%.

On the right, the logarithm of the occurrence is shown. Here, it can be seen that significantly fewer than half of the states are actually visited at all, with most of those that are reached occurring about four to five orders of magnitude less often than the most visited state. For $I_{\max} = 15$, $L = 2$, $S = 3$, using depth-first-search to mark all reachable forward states starting from an initial state obtained by running

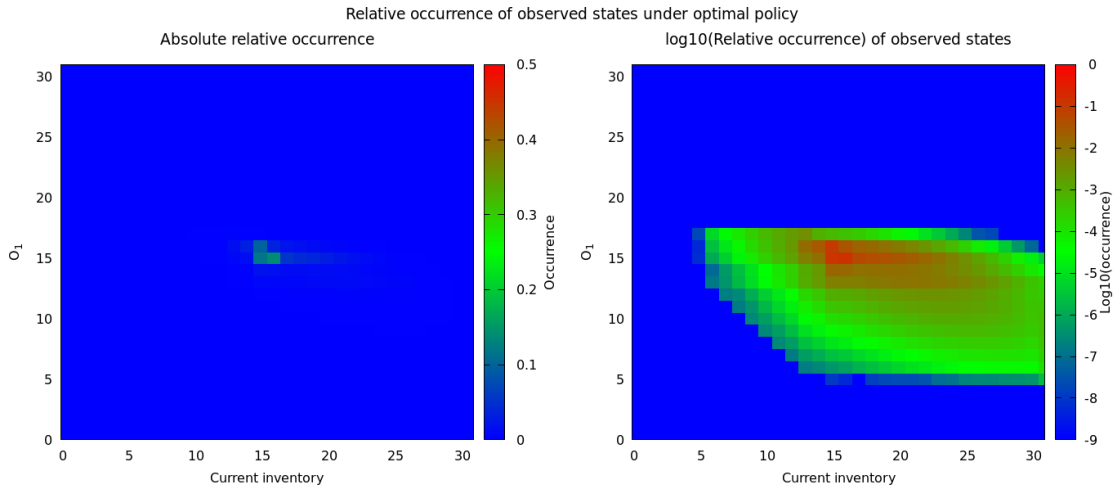


Figure 14: Distribution of observed states during 10^9 iterations under the optimal policy for $L = 2$, $S = 0$, $I_{\max} = 31$. On the left, the absolute (relative) occurrences are shown, and on the right the logarithm of the occurrences. Many states are never visited, and of the states that are visited, the majority of visits are concentrated in a small number of states.

the (optimal) policy for 100 iterations, a mere 1196 out of 208896 observable states, or $\approx 0.57\%$, are found to be reachable at all.

To conclude things, DQN will be applied on a state space that is unmanageable for the conventional methods as a proof of concept.

5.2 Deep Q-Learning

A Deep Q-Network replaces the Q-table with a neural network that takes the state as input and outputs the Q-values for every action. The network is trained by observing the reward $R(s, a, s')$ and the next state s' after taking action a in state s, and updating the network weights according to the rule.

To do this, two networks are used. One policy network is used to choose actions based on its own predicted Q-values, and one target network is used to calculate the target Q-values. In Equation 10, the updated Q-value would be the policy network's prediction of $Q(s, a)$, and the target network would be used to calculate $\max_{a'} Q(s', a')$.

The policy network actively learns the Q-values based on the target network's predictions. The policy network's weights are occasionally copied to the target network. Because of this, the target network is always slightly behind the policy network; this improves stability by preventing the policy network from chasing a moving target.

In addition, a replay memory is used to store observed transitions, which are sampled randomly to train the network. This decorrelates the training data and allows for more stable convergence. A pseudocode for DQN is given in Figure 15.

To implement this, `pybind11` (Jakob et al., [2]) is used to interface with the C++ code, and `pytorch` (Padzke et al., [3]) is used to implement the neural network. The code was largely based on the example code provided by the PyTorch Foundation (Padzke and Towers, [4]).

In Section 5.1, a very rough estimate of the time required to converge for $L = 3$, $S = 2$, $I_{\max} = 63$ using value iteration was given as months. With an observable $|\mathcal{S}| = 64^2 \cdot 2080 = 8519680$ states, and $|\mathcal{A}| = 64$ actions, the Q-table would have 545259520 ($\approx 5 \cdot 10^8$) entries. Using `sizeof(double)` $= 8$, this alone requires just over 4 GiB of memory (or 2 GiB if using `float`). Evaluating every state-action pair, even if only once, adds one final factor of 64, requiring $\approx 3.5 \cdot 10^{10}$ array references.

The DQN maps $\mathbb{N}^{L+\max\{1,S\}} \to \mathbb{R}^{|\mathcal{A}|}$, using a linear layer (with biases), followed by a single hidden layer with 256 neurons and a ReLU activation function, followed by another linear layer (with biases). For the case at hand, there are $|\mathcal{A}| = 64$ actions, and $L + \max\{1, S\} = 5$ inputs, resulting in $5 \cdot 256 + 256 \cdot 64 = 17664$ weights, and $256 + 64 = 320$ biases, for a total of 17984 parameters; a reduction of more than five orders of magnitude compared to the Q-table.

ε is initially set to 0.9 and linearly decreases to 0.01 over the whole training process. The discount factor γ is set to 0.9 to prevent the Q-values from having to converge to massive values, and the Huber loss is used as the loss function, which is more conservative than the mean squared error. In addition, gradients are clipped to a maximum norm of 0.5. The common optimizer Adam is used with a learning rate of 0.0005. Training runs for 10000 epochs, each with 100 episodes, which in turn perform one step and backpropagate on a random batch of 128 transitions. The memory stores the last 10000 transitions, and the target net is updated only every 10 epochs.

```
# two networks; policy_net learns the values and target_net
# serves as a stable base for target values
policy_net = DQN()
target_net = policy_net

# epsilon-greedy policy
select_action(state):
  if rand() > epsilon:
    return argmax(policy_net.forward(state))
  else:
    return random_action()

for epoch in 1, ..., N:

  for epsidode in 1, ..., M:

    # add a transition to the memory
    action = select_action(state)
    transition = environment.step(action)
    memory.store(transition)

    # backpropagate a random batch from the memory
    replay_batch = memory.sample(batch_size)
    for transition in replay_batch:
      q_value  = policy_net.forward(state)[action]   # predicted Q(s, a)
      q_next   = max(target_net.forward(next_state)) # max Q(s', a')
      target_q = reward + gamma * q_next             # target Q(s, a)
      loss     += loss_function(target_q, q_value)

    loss.backpropagate(policy_net)

  # transfer learned information to target_net
  target_net = policy_net
```

Figure 15: Pseudocode for DQN. The policy network is updated every episode using a random batch of previously observed transitions, and used to choose actions. The target network, used to calculate the target Q-values, is updated to equal the policy network every epoch. This prevents the policy network from chasing a moving target.

All of this is very conservative, and the rate of convergence could significantly be improved by using more aggressive settings, but these settings provide more stable convergence for demonstrative purposes.

The average reward, evaluated every 40 epochs for 30000 iterations, is shown in Figure 16. This has in turn been averaged over 10 training runs.

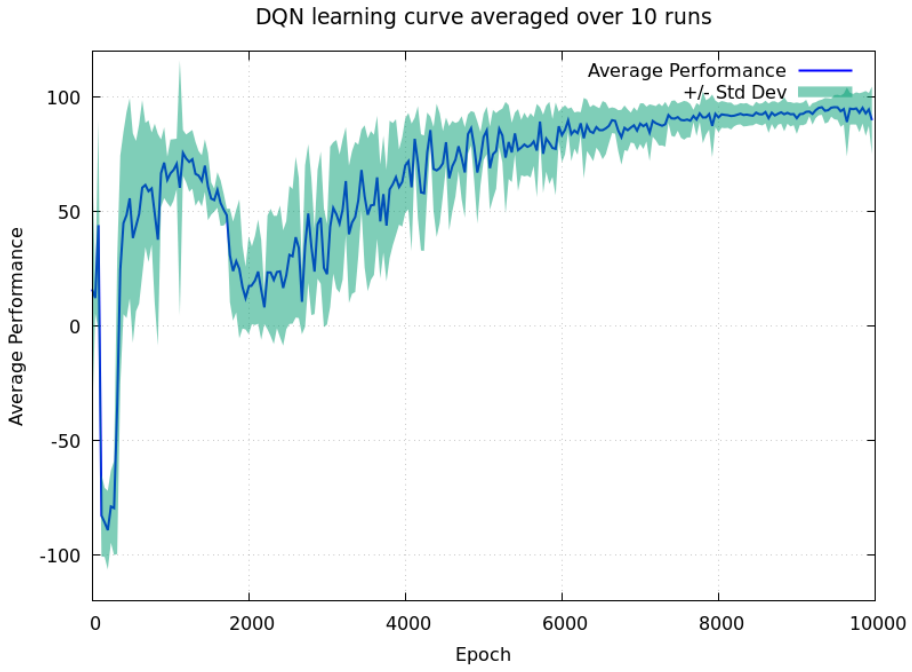


Figure 16: The reward of the DQN averaged over 10 training runs of 10000 epochs each with $I_{\max} = 63, L = 3, S = 2$, evaluated every 40 epochs. The final average reward is $\approx 3 \cdot 32 = 96$, which matches the rule of thumb for the optimal average reward.

The final average reward can be seen to match the rule of thumb of the optimal average reward being approximately $3\lambda = 3 \cdot 32 = 96$. Even on a non-AVX-512 CPU using a single thread, the training time is on the order of one to two hours per run. More aggressive settings may reduce this by a linear factor, and using a GPU can be expected to provide an additional massive linear increase in performance on the scale of orders of magnitude, as the batches and matrix multiplications may be processed in parallel.

Given the nature of the inventory model and the resulting Q-values, as was shown in Figure 13, the value of $I_{\max}$ has relatively little effect on performance, as this simply scales the range of inputs but not the general shape of the Q-values, something neural networks have no trouble doing too.

Increasing L has the largest impact on the rate of convergence, as rewards for actions taken are pushed further into the future, requiring longer to propagate. This may be one of the explanations for the initial performance in Figure 16.

Furthermore, increasing S has little effect on performance, as any inventory is likely to be sold within two periods under the current parameters. Attempting to extend this by lowering λ will simply scale the optimal policy down by the same factor. In addition, given the FIFO inventory policy, the vast majority of observed inventory states will still have the unsold inventory in the first fields.

Although no further analysis of DQN for inventory management will be performed, the current state of deep reinforcement learning in inventory management is covered well by Boute et al. [1], which serves as an excellent starting point for further reading.

While lacking the elegance of a proper mathematical solution, the inevitable outperformance of general, computation-driven methods over specialized, model-based approaches has not gone unnoticed. A fitting quote to end this thesis is the bitter lesson from Richard Sutton [5]:

> *We have to learn the bitter lesson that building in how we think we think does not work in the long run. The bitter lesson is based on the historical observations that 1) AI researchers have often tried to build knowledge into their agents, 2) this always helps in the short term, and is personally satisfying to the researcher, but 3) in the long run it plateaus and even inhibits further progress, and 4) breakthrough progress eventually arrives by an opposing approach based on scaling computation by search and learning. The eventual success is tinged with bitterness, and often incompletely digested, because it is success over a favored, human-centric approach.*

6 Conclusion

After a brief introduction of simple inventory models, a more general model incorporating finite shelf life and a lead time for orders was introduced for infinite time horizon. Three conventional methods (policy iteration, value iteration and Q-learning) had no trouble solving these model for small values of L, S and $I_{\max}$, but started running into a complexity wall as these values were increased.

Optimizations allowed this limit to be pushed one step further by reducing the state space by a factor of $|\mathcal{A}|$, as well as a significant reduction in memory footprint for $S > 0$. This allowed the model to be solved for values of L and S that were previously unmanageable, but only slightly shifted the ultimate limit.

Based on an analysis of the structure of the model, a heuristic approach was introduced. This provided a high quality initial policy using a single pass of dynamic programming to calculate the discounted intrinsic value of the inventories, followed by a single pass of policy improvement to yield a near-optimal policy, which greatly outperformed conventional methods after the first iteration. Still depending on the policy improvement, however, led to the same complexity issues as the conventional methods.

Finally, a neural approach was taken, which proved itself to be capable of solving the model for values of L, S and $I_{\max}$ that were previously unmanageable for conventional methods with ease, scaling well with increases in these values.

References

[1] Robert N. Boute, Joren Gijsbrechts, Willem van Jaarsveld, and Nathalie Vanvuchelen. Deep reinforcement learning for inventory control: A roadmap. *European Journal of Operational Research*, June 2021. Available at SSRN: https://ssrn.com/abstract=3861821 or http://dx.doi.org/10.2139/ssrn.3861821.

[2] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. pybind11 – seamless operability between c++11 and python, 2017. https://github.com/pybind/pybind11.

[3] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8026–8037, 2019.

[4] Adam Paszke and Mark Towers. Reinforcement learning (dqn) tutorial. PyTorch Tutorials, 2017.

[5] Richard S. Sutton. The bitter lesson. `http://www.incompleteideas.net/IncIdeas/BitterLesson.html`, 2019.

[6] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 2 edition, 2018.