A Neuro-Dynamic Programming Approach to Retailer Inventory Management[1]

Benjamin Van Roy[†‡]
Dimitri P. Bertsekas[‡]
Yuchun Lee[†]
John N. Tsitsiklis[‡]

[†]Unica Technologies, Inc.
Lincoln North
Lincoln, MA 01773

and

[‡]Laboratory for Information and Decision Systems
Massachusetts Institute of Technology
Cambridge, MA 02139

[1]This material is based upon work supported by the National Science Foundation under award number 9561500. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

ABSTRACT

We present a model of two-echelon retailer inventory systems, and we cast the problem of generating optimal control strategies into the framework of dynamic programming. We formulate two specific case studies, for which the underlying dynamic programming problems involve thirty-three and forty-six state variables, respectively. Because of the enormity of these state spaces, classical algorithms of dynamic programming are inapplicable. To address these complex problems, we develop *approximate* dynamic programming algorithms. The algorithms are motivated by recent research in artificial intelligence involving simulation–based methods and neural network approximations, and they are representative of algorithms studied in the emerging field of neuro-dynamic programming. We assess performance of resulting solutions relative to optimized s–type ("order–up–to") policies, which are generally accepted as reasonable heuristics for the types of problems we consider. In both case studies, we are able to generate control strategies substantially superior to the heuristics, reducing inventory costs by approximately ten percent.

1 Introduction

Many important problems in operations research involve sequential decision-making under uncertainty, or *stochastic control*. Dynamic programming (Bertsekas, 1995) provides an omnipotent framework for studying such problems, as well as a suite of algorithms for computing optimal decision policies. Unfortunately, the overwhelming computational requirements of these algorithms render them inapplicable to most realistic problems. As a result, complex stochastic control problems that arise in the real world are usually addressed using drastically simplified analyses and/or heuristics.

An exciting new alternative that is more closely tied to the sound framework of dynamic programming is being developed in the emerging field of neuro-dynamic programming (Bertsekas and Tsitsiklis, 1996). This approach makes use of ideas from artificial intelligence involving simulation–based algorithms and functional approximation techniques such as neural networks. The outcome is a methodology for approximating dynamic programming solutions without demanding the associated computational requirements.

Over the past few years, neuro-dynamic programming methods have generated several notable success stories. Examples include a program that plays Backgammon at the world champion level (Tesauro, 1992), an elevator dispatcher that is more efficient than several heuristics employed in practice (Crites and Barto, 1996), and an approach to job shop scheduling (Zhang and Dietterich, 1996). Additional case studies reported by Bertsekas and Tsitsiklis (1996) further demonstrate significant promise for neuro-dynamic programming. However, neuro-dynamic programming is a young field, and the algorithms that have been most successful in applications are not fully understood at a theoretical level. Furthermore, there is a large conglomeration of algorithms proposed by researchers in the field, and each one is complicated and parameterized by many values that must be selected by a user. It is unclear which algorithms and parameter settings will work on a particular problem, and when a method does work, it is still unclear which ingredients were actually necessary for success. Because of this, application of neuro-dynamic programming often requires trial and error, in a long process of parameter tweaking and experimentation.

In this paper, we describe work directed towards developing a streamlined neuro-dynamic programming approach for optimizing performance of retailer inventory systems (Nahmias and Smith, 1993). This is the problem of ordering and positioning retailer inventory at warehouses and stores in order to meet customer demands while simultaneously minimizing storage and transportation costs. This problem can also be viewed as a simple example from the broad class of multi-echelon inventory control problems that has received significant attention in the field of supply-chain management (Lee and Billington, 1993).

The remainder of this paper is organized as follows. The next section provides an overview of the research and results obtained. Section 3 describes in detail the model of retailer inventory systems used in this study, while Section 4 discusses the use of s–type policies in this model. Dynamic programming and neuro-dynamic programming are presented in Sections 5 and 6, respectively. Rather than presenting the methodologies in full generality, the material in these sections is customized to the purposes of retailer inventory management. Experimental results are discussed in Section 7. Section 8 contains some concluding remarks. Finally, formal state equations are provided in the appendix as a mathematically precise description of our retailer inventory system model.

2 Overview of Research

Rather than simply describing the final results obtained at the end of this research, we attempt in this paper to present some of the obstacles that were encountered and how they were overcome. We hope that this will lead to a better perspective on the state-of-the-art in neuro-dynamic programming, as well as the potential difficulty of applying the methodology. In this section, we overview the steps taken in this research and the results obtained at the end of the process. Subsequent sections provide a far more detailed account of what we discuss here.

In formulating a model of retailer inventory systems, we attempted to reflect the complexities that make the problem difficult. However, we did not attempt to capture all aspects of the problem that may be required for modeling a real-world retailer inventory system. The objective of this study was to establish the viability of neuro-dynamic programming as an effective approach to optimizing retailer inventory systems. We expect that success demonstrated on the models we present will generalize to more realistic models.

To gauge the efficacy of the neuro-dynamic programming algorithms developed in this study, performance was compared with optimized s–type policies (i.e., "order-up-to" policies), which have been the most commonly adopted approach in past research concerning optimization of retailer inventory systems (Nahmias and Smith, 1993; Nahmias and Smith, 1994). The details of this heuristic method are discussed in Section 4.

In choosing and customizing neuro-dynamic programming algorithms for the purpose of retailer inventory management, an effort was made to minimize the complexity of the methods. Only ingredients deemed critical to success were included, and we avoided many of the "frills" that make for additional parameters to be tweaked. We initially selected two neuro-dynamic programming algorithms and specialized them for the purposes of retailer inventory management. The two algorithms were approximate policy iteration and an on-line temporal-difference method (Bertsekas and Tsitsiklis, 1996).

In solving a stochastic control problem like that of retailer inventory management, neuro-dynamic programming algorithms approximate the problem's cost-to-go function by tuning parameters of an approximation architecture. Appropriate approximation architectures must be chosen for the problem at hand. In this research, we chose to employ linear and multilayer perceptron (neural network) architectures coupled with the extraction of features pertinent to the problem. We describe the chosen features and architectures in more detail later in this paper.

In an initial set of experiments, we analyzed only a very simple retailer inventory system. This system consisted of one warehouse and one store, and the underlying stochastic control problem involved only three state variables. A multilayer perceptron architecture, using the raw state variables as input features, was coupled with the neuro-dynamic programming algorithms to solve the problem. Unfortunately, the neuro-dynamic programming algorithms led to control strategies that performed far worse than an optimized s–type policy.

Extensive experimentation and a study of where the neuro-dynamic programming algorithms were failing on this simple problem led to a modification in the on-line temporal-difference algorithm. This involved adding an element of active exploration to the workings of the algorithm, the details of which will be described in a later section. The resulting algorithm generated approximations using the multilayer perceptron architecture that yielded performance essentially equivalent to the heuristic approach.

It is not surprising that the heuristic and the neuro-dynamic programming approach delivered similar levels of performance on the first problem. In particular, since the problem was so simple, both methods were probably nearoptimal. A second set of experiments was performed on a more complex case study. This new problem involved a central warehouse and ten stores with substantial transportation delays. The underlying stochastic control problem this time involved 33 state variables. A set of 24 features was selected based on the given problem, and a feature-based linear architecture was tried with the on-line temporal-difference algorithm. This combination was successful, generating policies that reduced costs by about ten percent over the cost of an optimized heuristic policy.

To see if this result could be further improved, we tried two variations of the approach. First, we replaced the feature-based linear architecture with a feature-based multilayer perceptron architecture (using the same features). However, this did not lead to performance superior to the linear case. A second variation involved increasing the degree of exploration in the on-line temporal difference method. Again, this did not improve performance.

We tested the neuro-dynamic programming approach on an additional problem of even greater complexity. This time, the underlying stochastic control problem involved 46 state variables. Again, the combination of a feature-based linear architecture and the on-line temporal difference method (with active exploration) led to costs about ten percent lower than the heuristic.

In the final analysis, neuro-dynamic programming proved to be successful in solving two complex retailer inventory management problems. To summarize the results obtained, we reiterate some key points:

1. The straightforward approximate policy iteration algorithms that were employed in the initial stages of this research do not work in their initial simple form.

2. The on-line temporal difference method with active exploration, coupled with a feature–based linear architecture, consistently cut costs by about ten percent (over a well–accepted heuristic approach) on two different complex retailer inventory management problems.

3. Given the chosen features, using a multilayer perceptron instead of the linear architecture did not lead to improved performance.

4. Increasing the degree of exploration in the temporal-difference algorithm did not lead to improved performance (just a small amount of exploration seemed to be required to make the huge difference in performance over having no exploration at all).

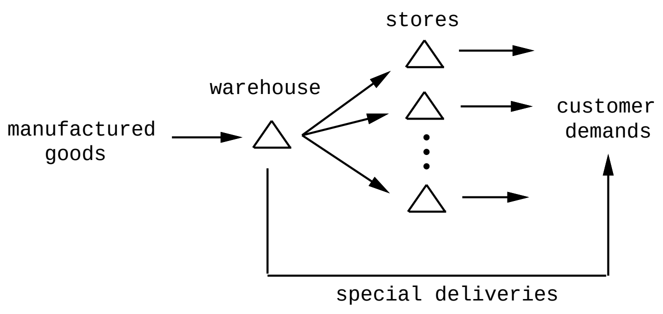

Figure 1: Schematic diagram of a retailer inventory system.

3 A Model of Retailer Inventory Systems

In this section, we describe the model of retailer inventory systems used in this study. The characteristics of this model are largely motivated by the studies of (Nahmias and Smith, 1993) and (Nahmias and Smith, 1994). The general structure is illustrated in Figure 1 and involves several stages:

1. Transportation of products from manufacturers
2. Packaging and storage of products at a central warehouse
3. Delivery of products from the warehouse to stores
4. Fulfillment of customer demands using either store or warehouse inventory

Demands materialize at each store during each time period. Each unit of demand can be viewed as a customer request for the product. If inventory is available at the store, it is used to meet ongoing demands. In the event of a shortage, the customer will, with a certain probability, be willing to wait for a special delivery from the warehouse. If the customer is in fact willing to wait, the demand is filled by inventory from the warehouse (if it is available).

At the end of each day, the warehouse orders additional units of inventory from the manufacturers, and the stores place orders to the warehouse. The warehouse manager fills store orders as much as possible given current levels of inventory. As materials travel from manufacturers to the warehouse and from the warehouse to the stores, they are delayed by transportation times.

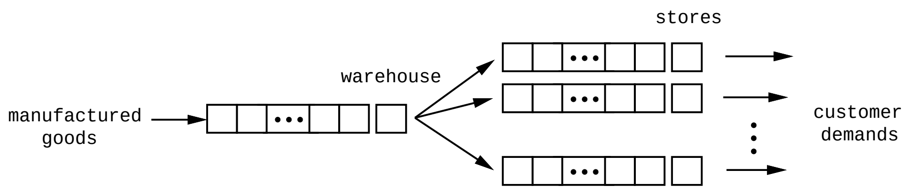

Figure 2: An illustration of the buffers in the retailer inventory system.

Coupled with the uncertainty of future demands, these delays create the need for storage of inventory at stores.

The differing impact of inventory at the warehouse on costs and service performance makes it desirable to also maintain stock there. For example, inventory stored at the warehouse provides a greater degree of flexibility than that maintained at a single store. In particular, inventory stored at the warehouse can be used to fill special orders made by customers at any store (for individual customers who are willing to wait), and can also be sent to any store in the advent of a shortage of goods. On the other hand, a surplus of inventory at one store cannot be used to compensate for a shortage at another. Furthermore, storage costs at stores are often higher than at the warehouse.

In the remainder of this section, we present some technical details concerning aspects of the model we have described. In particular, we further discuss the dynamics of inventory flow, the nature of the stochastic demand process, and the cost structure.

3.1 Dynamics of Inventory Flow

As illustrated in Figure 1, inventory is stored at two stages. The warehouse holds reserves in anticipation of special orders and shipments to stores, which make up the second stage of inventory storage. There are delays in the transportation of stock from one stage to the next, and to simplify our discussion, we consider the delays to be multiples of a fixed unit of time which we will take to be a day. Hence, the model involves a dynamic system that evolves in discrete time.

The delays in the inventory system are illustrated in Figure 2. Each square represents a buffer where goods may be located at a particular point in time.

The movement of goods between buffers is synchronized by a single clock that "ticks" once per day. Goods enter and exit buffers only at the clock ticks. The row of buffers to the left of the warehouse buffer are associated with delays in the transportation of goods from a manufacturer to the warehouse. At each clock tick, goods located in any one of these buffers moves one buffer to the right. Similarly, the row of buffers to the left of each store buffer is associated with delays in transporting goods from the warehouse to the store. Again, at each clock tick, goods proceed one buffer to the right, as transportation progresses.

The entrance of goods into the system and the movement of goods from the warehouse to transportation buffers are controlled by decisions of the inventory manager, which are made just prior to each tick. At each tick, a specified quantity of goods (the warehouse order) enters the system at the leftmost buffer. This quantity is limited by a production capacity as well as the warehouse capacity. The amount ordered at any one time cannot exceed the production capacity, and the total quantity of goods currently at and on-route to the warehouse cannot exceed the warehouse capacity.

Also at each tick, a specified quantity of goods are transferred from the warehouse to the leftmost transportation buffers of specified stores. Of course, the total quantity of goods transfered here must be less than the amount available at the warehouse prior to the tick. Furthermore, at any time, the total quantity of goods currently at and on-route to any particular store can be no greater than the store capacity.

Goods exit the system upon customer demand. At each tick, such demands arise at each of the stores. If the amount demanded at a particular store is less than or equal to the quantity of inventory available just prior to the tick, the store's inventory level is reduced by the demanded quantity. Otherwise, the store's inventory is completely depleted, and each unsatisfied customer is allowed the option to request a special delivery from the warehouse. In our model, each individual customer makes such a request with a given probability. If a customer whose demand has not been satisfied by the store does make such a request, and a unit of inventory is available, then the warehouse inventory is decremented by one.

To be completely precise, we must specify the ordering of events that occur at each clock tick. First, goods ordered by the warehouse enter into the system. Second, goods are transferred from the warehouse to the appropriate transportation buffers. Then, demands are filled as needed. Finally, goods in transportation buffers progress towards the right.

3.2 Demand Process

In the model, stochasticity arises from the uncertainty of future demands. In this research, the demands were modeled as random variables that are independent and identically distributed through time and among different stores. Each sample was generated as follows:

1. sample from a normal distribution with a given mean and a given standard deviation;

2. round off this value to the closest integer;

3. take the maximum of zero and the resulting value.

3.3 Cost Structure

At each clock tick, a cost of operation is incurred by the retailer inventory system. The objective of the firm is to minimize these costs, on average. The cost can be broken down into three categories: storage cost, shortage cost, and transportation cost. In this section, we describe how each of these costs are computed.

Storage costs are incurred at both the warehouse and the stores. At each clock tick, the total quantity of inventory at stores is multiplied by a store cost of storage, and the quantity of inventory at the warehouse is multiplied by a warehouse cost of storage. The sum of these two products is the storage cost for that day.

Shortage costs are costs associated with unfulfilled demands. A customer's demand may be satisfied by inventory either at the store where the customer is located or the warehouse (if the customer opts for a special delivery). Any customer whose demands are not filled by either of these two places incurs a shortage cost to the system.

Transportation costs in our model are associated only with special deliveries. In particular, each special delivery made to a customer incurs a particular cost. Note that for special deliveries to be profitable, the cost of an unfulfilled unit of demand (i.e., shortage cost) must be greater than that of a special delivery.

3.4 Model Parameters

Now that we have described the dynamics of the model in detail, it may be useful to enumerate the model parameters. Values must be assigned to these parameters in order to make the model behavior commensurate with that of a specific manifestation of a retailer inventory system. The list follows:

1. Number of stores
2. Delay to stores
3. Delay to warehouse
4. Production capacity
5. Warehouse capacity
6. Store capacity
7. Probability of customer waiting
8. Cost of special delivery
9. Warehouse storage cost
10. Store storage cost
11. Mean demand
12. Demand standard deviation
13. Shortage cost

4 Heuristic Policies

A heuristic policy for controlling the retailer inventory system was implemented and used as a baseline for comparison against neuro-dynamic programming approaches. The type of heuristic used is known as an s-type, or "order-up-to," policy and is accepted as a reasonable approach to problem formulations that have independent identically distributed demands, like the one we have proposed. Examples of research where such policies are the focus of study are discussed in (Nahmias and Smith, 1993) and include (Nahmias and Smith, 1994).

The s–type policy we implemented is parameterized by two values: a warehouse order-up-to level and a store order-up-to level. Essentially, at each time step the inventory manager tries to order inventory such that all inventory at and expected to arrive at the warehouse is equal to the warehouse order-up-to level and all the inventory at or expected to arrive at any particular store is equal to the store order-up-to level.

Although the main idea is simple, the details of how store orders are generated by the heuristic policy are tedious. First, a "desired order" equal to the difference between the store order-up-to level and the total of inventory currently at the store and inventory currently on-route to the store is computed for each store. If all desired orders can be filled by the inventory currently available at the warehouse, then they are. Otherwise, all inventory at the warehouse is sent to stores, and the preference among stores is decided in a way that always maximizes the minimum among stores of the total of current inventory and inventory on-route to a store.

Once the store orders have been computed, we compute the total of inventory currently at the warehouse and all inventory on-route to the warehouse, less the total of store orders. If the difference between this quantity and the warehouse order-up-to level is less than the production capacity, then the warehouse order is set equal to this difference. Otherwise, the warehouse order is equal to the production capacity.

Note that we have discussed only how the heuristic policy works given specified order-up-to levels, but not how the order-up-to levels are to be determined. In this research, the best order-up-to levels were determined by an exhaustive search, where the average cost associated with each pair of order-up-to levels was assessed in a lengthy simulation. Note that an exhaustive search of this type would be computationally prohibitive if we allowed the stores to have different order-up-to levels, which would be called for if the stores had independent attributes (e.g., different transportation delays).

5 Dynamic Programming

Dynamic programming (DP) offers a very general framework for stochastic control problems (Bertsekas, 1995). In this section, we present a DP framework that is a bit different from the standard. In particular, our setting is somewhat specialized to the retailer inventory problem and leads to more efficient computational approaches in the context of neuro-dynamic programming (NDP). We also present in detail the way in which we formulated the retailer inventory management problem in terms of this DP framework. To make the exposition of DP both brief and precise, we only discuss the case involving systems that evolve over finite state spaces and in discrete time.

Let S be the state space of a system of interest (each element corresponds to a particular combination of inventory levels). We associate two states $x_t, y_t \in S$ to any nonnegative integer time t. We refer to x_t as the "pre-decision state" and y_t as the "post-decision state." Furthermore, a decision u_t that influences the system is selected from a finite set U at each time step. The state evolves according to two difference equations: $x_{t+1} = f_1(y_t, w_t)$ and $y_t = f_2(x_t, u_t)$, where f_1 and f_2 are some functions describing the system dynamics and w_t is a random noise term taken from a fixed distribution, independent from all information available up to time t. There is a cost $g(y_t, w_t)$ associated with the system affected by a noise term w_t while the post-decision state is y_t.

A *policy* is a mapping $\mu : S \mapsto U$ that determines a decision as a function of pre-decision state, i.e., $u_t = \mu(x_t)$. The goal in stochastic control is to select an optimal policy (i.e., one that minimizes long-term costs). We express the long-term cost to be minimized as the expectation of a discounted infinite sum of future costs, as a function of an initial post-decision state, i.e.,

$$J^{\mu}(y) = E\left[\sum_{t=0}^{\infty} \alpha^t g(y_t, w_t) | y_0 = y, \mu\right].$$

Here, $\alpha \in (0,1)$ is a discount factor and $J^{\mu}(y)$ denotes the expected long-term cost given that the system starts in post-decision state y and is controlled by a policy μ. An optimal policy μ^* is one that minimizes J^{μ} simultaneously for all initial post-decision states, and the function J^{μ^*}, known as the value function, is denoted by J^*.

A well known result in dynamic programming is that the value function satisfies Bellman's equation, which in our formulation, takes on the form

$$J^*(y) = E_w\left[g(y,w) + \alpha \bar{J}(f_1(y,w))\right],$$

where $\bar{J}$ is given by

$$\bar{J}(x) = \min_{u \in U} J^*(f_2(x,u)).$$

Furthermore, a policy μ^* is optimal if and only if it satisfies

$$\mu^*(x) = \arg\min_{u \in U} J^*(f_2(x,u)).$$

Note that, using this expression, we can generate an optimal policy based on a value function J^* that is defined only over the post–decision states. If this function is available, there is no need for the function $\bar{J}$, which defines values for pre–decision states.

In principle, an optimal policy can be found by first numerically solving Bellman's equation and then computing the optimal policy using the resulting value function. However, this requires computation and storage of $J^*(y)$ for each post-decision state, which is generally infeasible given the enormity of state spaces for practical problems.

We now describe how the retailer inventory management problem was formulated in terms of the DP framework we have described. First of all, the state of the retailer system is described by a vector in which each component corresponds to a buffer (see Figure 2). At any time, each state variable takes a value equal to the quantity of goods currently located at the corresponding buffer. Hence, the number of state variables (components of the state vector) is equal to one plus the warehouse transportation delay plus the number of stores times one plus the store transportation delay. Note that the size of the state space here grows exponentially with the number of state variables, and therefore quickly becomes intractable.

Each decision u_t corresponds to a vector of store and warehouse orders during the tth time step. The decision u_t must be made on the basis of the pre-decision state x_t. Given the pre-decision state x_t and the decision u_t, the post-decision state y_t is generated deterministically. This involves the entrance of goods ordered by the warehouse into the leftmost buffer and the transition of goods ordered by stores from the warehouse buffer to appropriate transportation buffers.

The post decision state y_t is transformed as customer demands are fulfilled and transportation progresses. The result of these transformations is the next pre–decision state x_{t+1}. Note that the transition from y_t to x_{t+1} is influenced by stochastic demands. In the context of our DP formulation, demands correspond to the random noise term w_t. The dynamics of the state transitions can be inferred from the description of our retailer inventory system model, provided in Section 3. Nevertheless, to enhance clarity, we present formal state equations in the appendix.

Costs $g(y_t, w_t)$ are computed in a fairly straightforward manner as described in Section 3. The discount factor used in our formulation was 0.99. The reason is that policies will be evaluated in terms of average costs and setting the discount factor close to one makes the discounted problem resemble the average cost problem.

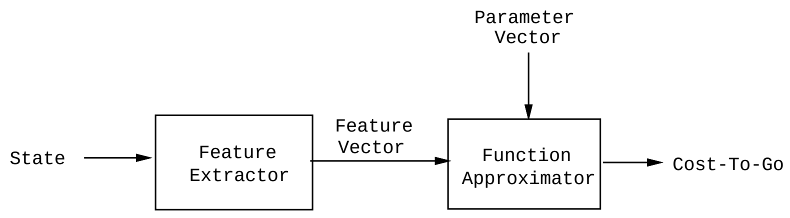

Figure 3: A feature–based approximation architecture.

6 Neuro-Dynamic Programming

Dynamic programming offers a suite of algorithms for generating optimal control strategies. However, the overwhelming computational requirements associated with these algorithms render them inapplicable in practical situations. Due to a lack of other systematic approaches for dealing with such problems, simplified problem-specific analyses and heuristics have become the norm. Such analyses and heuristics often ignore much information that is important to effective decision-making, leading to control policies that are far from optimal. The recent emergence of neuro-dynamic programming puts forth an exciting new possibility. New and highly promising approaches to addressing complex stochastic control problems have been developed in this field. These approaches focus on approximating solutions that would be generated by dynamic programming, except in a computationally feasible manner.

The main idea in neuro-dynamic programming is to approximate the mapping $J^* : S \mapsto \Re$ using an approximation architecture. An approximation architecture can be thought of as a function $\tilde{J} : S \times \Re^k \mapsto \Re$. NDP algorithms try to find a parameter vector $r \in \Re^k$ such that the function $\tilde{J}(\cdot, r)$ closely approximates J^*.

In general, choosing an appropriate approximation architecture is a problem dependent task. In this research, we designed approximation architectures involving two stages: a feature extractor and a function approximator (see Figure 3). The feature extractor uses the post-decision state y_t to compute a feature vector z_t. The components of z_t are values that we thought were natural for capturing key information concerning states of the retailer inventory management problem. This feature vector was used as input to a second stage, which involved a generic function approximator parameterized by a vector r. Two types of function approximators were employed in this research: linear approximators and the multilayer perceptron neural network with a linear output node. In the case of the linear approximator, all but one component of the parameter vector r correspond to coefficients that are multiplied by individual components of the feature vector. The remaining component is a scalar offset term. In the case of the multilayer perceptron, the parameter vector r stores the weights of the network connections.

In the remainder of this section, we describe the NDP algorithms that were used to tune parameters of our approximation architectures. The features we used are described in Section 7. The function approximators (linear and multilayer perceptron) are well known, and we omit any detailed discussion about them.

6.1 Approximate Policy Iteration

Approximate policy iteration is a generalization of policy iteration, a classical algorithm in dynamic programming. The policy iteration algorithm generates a sequence μ_i of improving policies. The initial policy μ_0 is usually chosen to be some reasonable heuristic, and the cost function J^{μ_0} associated with the policy is computed (one value is computed for each state). Then, a new policy μ_1 is generated according to the equation

$$\mu_1(x) = \arg\min_{u \in U} J^{\mu_0}(f_2(x, u)).$$

The same procedure is iterated to generate subsequent policies. It is well-known that for problems with a finite number of policies, μ_i is equal to μ^* and J^{μ_i} is equal to J^* for sufficiently large i.

In approximate policy iteration, instead of computing the cost function J^{μ_i} exactly at each iteration, the function is approximated by some architecture $\tilde{J}(\cdot, r_i)$, where r_i is a parameter vector chosen to make $\tilde{J}(\cdot, r_i)$ close to J^{μ_i}. The subsequent policy is then generated via

$$\mu_{i+1}(x) = \arg\min_{u \in U} \tilde{J}(f_2(x, u), r_i).$$

There have been many methods used for approximating J^{μ_i} in each ith policy iteration. A comprehensive survey is provided in (Bertsekas and Tsitsiklis, 1996). In this research, we chose to use the on-line TD(1) method, which at each iteration, effectively computes the parameter vector r_i that minimizes

$$\sum_{x \in S} \pi_i(x) \left(J^{\mu_i}(x) - \tilde{J}(x, r_i) \right)^2,$$

where $\pi_i(x)$ stands for the relative frequency of occurrence of state x when the system is controlled by policy μ_i. We refer the reader to (Bertsekas and Tsitsiklis, 1996) for a detailed discussion of this method.

6.2 An On-Line Temporal-Difference Method

Variants of the temporal-difference algorithm (Sutton, 1988; Tsitsiklis and Van Roy, 1996) have been applied successfully to several large scale applications of NDP. Examples include a Backgammon player (Tesauro, 1992), an elevator dispatcher (Crites and Barto, 1996), and a job shop scheduling method (Zhang and Dietterich, 1996). The variants used in these applications bear significant differences, and in this research project, we tried to use a simple algorithm that possessed what we felt were the most important properties. In this section, we present the algorithm in its initial form. This algorithm may be viewed as an extreme form of "optimistic approximate policy iteration," as discussed in (Bertsekas and Tsitsiklis, 1996). As mentioned in Section 2, this algorithm was not successful until we added active exploration, which is discussed in the next section.

The algorithm updates the parameter vector of an approximation architecture during each step of a single endless simulation. In particular, we start with an arbitrary parameter vector r_0 and generate a sequence r_t using the following procedure:

1. Given the initial pre–decision state x_0 of the simulator, generate a control u_0 by letting
$$u_0 = \arg\min_{u \in U} \tilde{J}(f_2(x_0, u), r_0);$$
2. Run the simulator using control u_0 to obtain the first post-decision state
$$y_0 = f_2(x_0, u_0);$$
3. More generally, at time t, run the simulator using control u_t to obtain the next pre-decision state
$$x_{t+1} = f_1(y_t, w_t),$$
and the cost $g(y_t, w_t)$;
4. Generate a control u_{t+1} by letting
$$u_{t+1} = \arg\min_{u \in U} \tilde{J}(f_2(x_{t+1}, u), r_t);$$

5. Run the simulator using control u_{t+1} to obtain the post-decision state

$$y_{t+1} = f_2(x_{t+1}, u_{t+1});$$

6. Update the parameter vector via

$$r_{t+1} = r_t + \gamma_t \Big(g(y_t, w_t) + \tilde{J}(y_{t+1}, r_t) - \tilde{J}(y_t, r_t) \Big) \nabla_r \tilde{J}(y_t, r_t),$$

   where γ_t is a small step size parameter;

7. Return to step (3).

6.3 Active Exploration

As we mentioned in the introduction, it was only after we added active exploration to the workings of the temporal-difference method that it performed well. Note that the algorithm described in the previous section always updates the parameter vector to tune the approximate values $\tilde{J}(x, r)$ at states x visited by the current policy, which in turn are determined by the parameter vector r. In some sense, the exploration here is *passive*, i.e., only states that naturally occur on the basis of the current approximation to the value function are visited. By active exploration, we refer to a mechanism that brings about some tendency to visit a larger range of states.

Except for the steps involving generation of control decisions, the temporal-difference algorithm that we used *with* active exploration follows the same routine as that *without* active exploration. In particular, the algorithm can be described by the steps enumerated in the previous section, except with the equations of Steps (1) and (4) replaced by

$$u_0 = n_0 + \arg\min_{u \in U} \tilde{J}(f_2(x_0, u), r_0),$$

and

$$u_{t+1} = n_t + \arg\min_{u \in U} \tilde{J}(f_2(x_{t+1}, u), r_t),$$

respectively, where each n_t is a noise term. Note that the only difference is the addition of a noise term. The structure of the noise term is described on a case-by-case basis in the next section.

7 Results With the NDP Approach

In this section, we present the results obtained from applying the NDP approaches we developed to the retailer inventory management problem. Through our development, much of which occurred in the process of experimentation, we arrived at an approach that was successful relative to the heuristic s-type policies. To the extent of our experimentation, the method also proved to be robust to changes in problem-specifics.

In this section, we present a relatively detailed account of our experimental results. Experiments were conducted using three different problems as test beds, each described in the following subsections with its associated experiments.

7.1 Initial Experiments With a Simple Problem

The first set of experiments involved optimization of a very simple retailer inventory system. The purpose of these experiments was to debug the software packages developed in the initial stages of research and also to ensure that the NDP methodologies worked reasonably well on a simple problem, before moving to complex situations.

The system included only one store in addition to the warehouse. There was no delay for goods ordered by the warehouse, and there was a delay of only one time unit between the warehouse and the store. There were therefore only three state variables involved (each corresponding to the quantity of goods within a buffer). The list of parameter settings for this problem are provided in the table below. Note that we also have provided the true mean and standard deviation of demands (recall from Section 3 that the mean and standard deviation parameters do not correspond to the true mean and standard deviations of the resulting stochastic demands).

number of stores	1
delay to stores	1
delay to warehouse	0
production capacity	10
warehouse capacity	50
store capacity	50
probability of customer waiting	1
cost of special delivery	10
warehouse storage cost	1
store storage cost	2
mean demand (true mean)	5 (6.2)
demand stdev (true stdev)	8 (6.2)
shortage cost	50

As a baseline for comparison, we developed an s-type policy, optimizing the order-up-to levels associated with the warehouse and store. Figure 4 illustrates how varying the order-up-to levels affects the average cost of the policy. Each point on the graph is computed by averaging costs over a lengthy simulation. The optimal order-up-to levels turned out to be 10 for the warehouse and 16 for the store. The corresponding average cost was 51.7.

Several NDP algorithms were tried with a single approximation architecture. The architecture consisted of a multilayer perceptron with ten hidden nodes in a single hidden layer. Three features were used as input to the network, each a normalized version of one of the state variables. In particular, if the buffer levels at a given point in time were b_i, $i = 1, 2, 3$, then the ith input feature was given by

$$c_i = \frac{b_i - \bar{b}_i}{\sigma_i}.$$

The $\bar{b}_i$'s and σ_i's were computed prior to execution of the NDP algorithm as follows. A large collection of (post-decision) states sampled while simulating the heuristic policy (with the optimal order-up-to levels) was collected. From this data, each $\bar{b}_i$ was set equal to the sample mean of the ith state variable, and each σ_i was set equal to the sample standard deviation of the ith state variable.

The approximate policy iteration algorithm, in the form described in Section 6, was tried on the problem. The s–type policy (with optimized order-up-to levels) was used as the initial policy. This algorithm consistently generated a second policy that was far worse than the initial one.

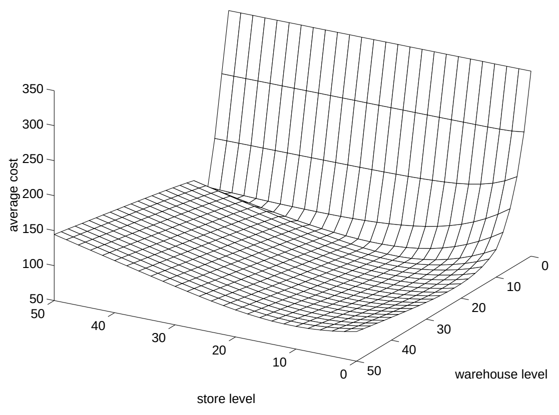

Figure 4: Performance of the heuristic as a function of order-up-to levels. The optimum levels was 10 for the warehouse and 16 for the store. With these levels, the average cost was 51.7.

Upon failure of the approximate policy iteration algorithm, experiments were conducted using the on-line temporal-difference method. Again, only policies that were far worse than the heuristic were generated.

Next, we tried adding a small degree of exploration to the on-line temporal difference method. In particular, each time a decision was generated using the approximation architecture $\tilde{J}$, a noise term was added to the decision. Recall that there are two decision variables: the warehouse order and the store order. The noise term was generated by adding a unit normal random variable to each decision variable, rounding off to the closest integer in each case, and then making sure the decision variables stayed within their limits. That is, if the noise term made a variable negative, the variable was set to zero, and if the noise term made a variable too large (e.g., having a warehouse order greater than the production capacity), then the variable was set to its maximum allowable value.

With the extra exploration term, the on-line temporal-difference method essentially matched the performance of the heuristic. Figure 5 displays the evolution of average cost as the algorithm progresses in tuning the parameters of the multilayer perceptron, in experiments performed both with and without active exploration. In both cases, a step size $\gamma_t = 0.01$ was used for the first 2×10^7 time steps, and a step size $\gamma_t = 0.001$ was used for the next 2×10^7

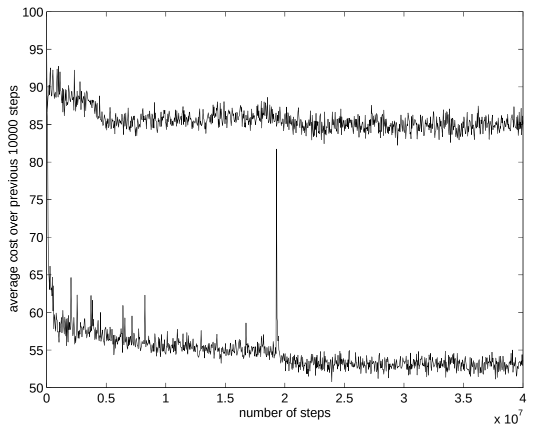

Figure 5: A demonstration of the importance of exploration. The two plots show the evolution of average cost using the on-line temporal-difference method with (lower plot) and without (higher plot) exploration. Each point represents cost averaged over ten thousand consecutive time steps during the execution of an algorithm.

time steps.

Note that in the graph of Figure 5 associated with the exploratory version of the algorithm, the average cost is computed during the execution of the algorithm, and is thus affected by the active exploration. In particular, a policy based on the final approximate value function without any exploratory term should perform better than the policy with active exploration (the exploration is there to improve the "learning and discovery" process that the algorithm goes through, rather than to improve performance of a policy at any given time). Indeed, a simulation employing a non-exploratory policy based on the final approximate value function generated an average cost of 52.6, which was slightly better than average costs sampled during execution of the exploratory on-line algorithm.

7.2 Case Study 1

With the success of the on-line temporal-difference method on a simple problem, a subsequent set of experiments was conducted on a more complex test bed. The parameters used for the retailer inventory management problem of this case study are given in the table below.

number of stores	10
delay to stores	2
delay to warehouse	2
production capacity	100
warehouse capacity	1000
store capacity	100
probability of customer waiting	0.8
cost of special delivery	0
warehouse storage cost	3
store storage cost	3
mean demand (true mean)	5 (8.6)
demand stdev (true stdev)	14 (9.8)
shortage cost	60

Once again, an s–type heuristic policy was developed by optimizing over order-up-to levels. Since the properties of all stores were identical, we assumed that the order up-to-levels of all stores should be the same, and there were again only two variables to optimize: a warehouse order-up-to level and a store order-up-to level. Figure 6 shows how the average cost of the system varies with these two variables. Each value in the graph was computed from a lengthy simulation. The optimal order-up-to levels were 330 for the warehouse and 23 for each store. The corresponding average cost was 1302.

In the simple problem of the previous section, there were only two inventory sites for which orders had to be placed. In the more complex problem of this section, on the other hand, there are eleven inventory sites, and exhausting all possible combinations of orders that can be made for these eleven sites would take too long. In particular, the minimizations carried out in steps (1) and (4) of the on-line temporal-difference method would be essentially impossible to carry out. Because of this, we constrained the decision space to a more manageable subset.

First of all, we represented decisions in terms of two variables: a warehouse order and a store order-up-to level. Given particular values for the two variables, the individual store orders would be set to exactly what the s–type policy described in Section 4 would set them to given the store order-up-to level. Note, however, that unlike the case of the heuristic s–type policy, the store order-up-to-level here is chosen at each time step, rather than taken to be a fixed constant.

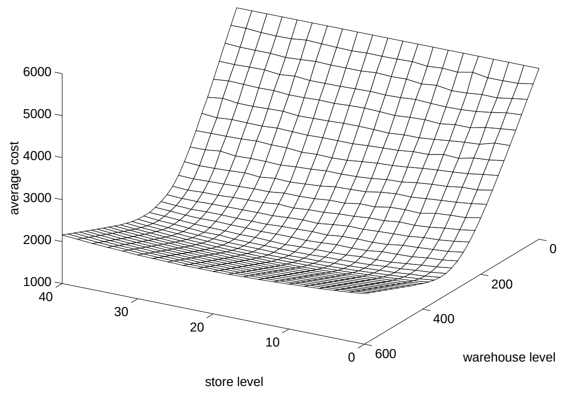

Figure 6: Performance of the heuristic as a function of order-up-to levels. The optimum levels were 330 for the warehouse and 23 for each of the stores. With these levels, the average cost was 1302.

To further accelerate execution of the on-line temporal-difference method, we limited the space of decisions considered at each time step to the set involving warehouse orders ranging from 50 to 100 in increments of 10 and store order-up-to levels ranging from 0 to 40 in increments of 5. There were therefore a total of 60 possible decisions considered at each time step. The minimization in Step (4) of the temporal-difference algorithm was carried out by exhaustive enumeration of these 60 possible decisions.

The approximation architectures employed in this case study involved the use of the following 22 features:

(1) total inventory at stores
(2) total inventory to arrive at stores in one time step
(3) total inventory to arrive at stores in two time steps
(4) inventory at warehouse
(5) inventory to arrive at warehouse in one time step
(6) inventory to arrive at warehouse in two time steps
(7) inventory to arrive at warehouse in three time steps
(8)-(14) the squares of (1)-(7)
(15) variance among stores of inventory levels

(16) variance among stores of inventory levels plus inventory to arrive in one time step
(17) variance among stores of inventory levels plus inventory to arrive within two time steps
(18) the product of (1) and (4)
(19) the product of (4) and the sum of (1) through (3)
(20) the sum of (4) through (7) times the sum of (1) through (3)
(21) the sum of (4) through (6) times the sum of (1) through (3)
(22) the product of (3), (4), and (7)

By variance among stores (as in features (15) through (17)), we mean the average among stores of the square of the difference between quantities associated with each store and the average of such quantities over the stores.

The feature values were normalized using an approach analogous to that used in the context of the simple problem from the previous section. In particular, a data set was collected from simulations using the s-type policy, and this data was used to compute means and standard deviations associated with each feature. These mean and standard deviation values were then used as normalization parameters just as in the previous section.

For active exploration, noise terms were added to the decisions generated using the approximate value function at each step of the temporal-difference algorithm. The way noise terms were added is completely analogous to the method employed in the previous section, except that this time the noise term added to the warehouse order involved a normal random variable with a mean of zero and a standard deviation of five. Furthermore, the noise terms added to the store orders were independent from one another.

We began by using a feature-based linear architecture with the features described. We experimented with different step sizes to better understand how the temporal-difference method was working with this problem. We found that the performance tended to diverge with larger step sizes, and improved at an extremely slow rate when the step sizes were reduced enough to prevent divergence. Upon investigation of feature values, it was found that feature (17) was taking on values far larger than those that appeared when the system was controlled by the s-type policy. Hence, we increased the standard deviation parameter associated with this feature to scale its values down significantly. Once this was done, performance improved at a much faster rate upon execution of the temporal-difference algorithm.

Two variations on the initial architecture/algorithm were explored. One involved replacing the linear function approximator with a multilayer per-

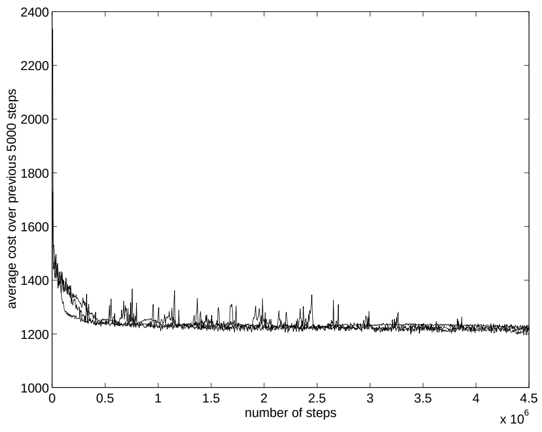

Figure 7: Evolution of average cost using the on-line temporal-difference method with a linear architecture and a small degree of exploration, a single hidden layer, 10 hidden node, multilayer perceptron and a small degree of exploration, and a linear architecture with a greater degree of exploration (all three cases generated very similar plots). Each point represents cost averaged over five thousand consecutive time steps during the execution of an algorithm. The final average costs associated with the three approximation schemes after this training were 1179, 1209, and 1181, respectively (versus 1302 for the heuristic).

ceptron with a single hidden layer of ten nodes. The other variation used the original linear architecture, but with an increased degree of exploration. Here the random noise term added to the warehouse order involved a normal random variable with a standard deviation of ten, and that added to the store orders involved a normal random variable with a standard deviation of two. Figure 7 charts the evolution of average cost during the execution of the temporal-difference algorithm in all three of these cases. In the two cases involving linear architectures, the step size was maintained at $\gamma_t = 0.0001$, while with the multilayer perceptron-based architecture, the step size was $\gamma_t = 0.0001$ during the first 1.5 million steps and $\gamma_t = 0.00001$ thereafter. These step size schedules were chosen after some trial and error.

All three variants of on-line temporal-difference methods generated policies superior to the heuristic. In particular, the linear architectures generated policies with average costs of 1179 (less active exploration) and 1181 (more

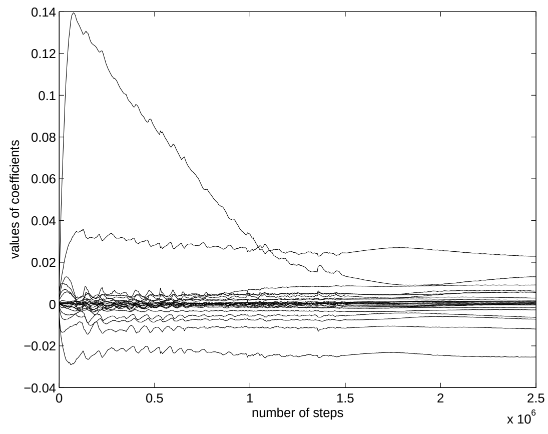

Figure 8: Evolution of coefficient values during training using the on–line temporal –difference method with a linear architecture and a small degree of exploration.

active exploration), while the multilayer perceptron architecture led to an average cost of 1209. Hence, the best policy cut costs by about ten percent relative to the heuristic. Figure 8 charts the evolution of parameter values in the linear feature-based architecture as they were tuned by the on-line temporal difference method with the lesser degree of exploration.

7.3 Case Study 2

We tested the on–line temporal difference algorithm on an additional problem of even greater complexity than the previous one. The parameters for this new problem are given in the table below.

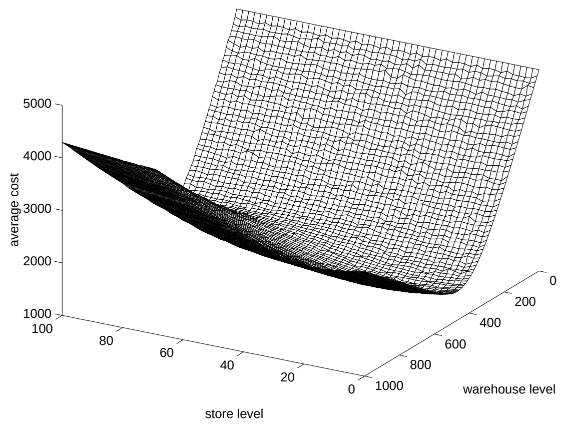

Figure 9: Performance of the heuristic as a function of order-up-to levels. The optimum levels were 460 for the warehouse and 22 for each of the store. With these levels, the average cost was 1449.

number of stores	10
delay to stores	3
delay to warehouse	5
production capacity	100
warehouse capacity	1000
store capacity	100
probability of customer waiting	0.8
cost of special delivery	0
warehouse storage cost	3
store storage cost	3
mean demand (true mean)	0 (8.0)
demand stdev (true stdev)	20 (11.6)
shortage cost	60

Once again, an s–type heuristic policy was developed by optimizing over order-up-to levels. Figure 9 shows how the average cost of the system varies with the two order-up-to levels. Each value in the graph was computed from a lengthy simulation. The optimal order-up-to levels were 460 for the warehouse and 22 for each store. The corresponding average cost was 1449.

As in the previous section, decisions were represented in terms of two variables: a warehouse order and a store order-up-to level. This time, the decision space was limited to warehouse orders ranging from 50 to 100 in increments of 10 and store order-up-to levels ranging from 0 to 50 in increments of 5, for a total of 72 possible decisions considered at each time step.

The approximation architectures employed in this case study involved the use of the following 29 features:

(1) total inventory at stores
(2) total inventory to arrive at stores in one time step
(3) total inventory to arrive at stores in two time steps
(4) total inventory to arrive at stores in three time steps
(5) inventory at warehouse
(6) inventory to arrive at warehouse in one time step
(7) inventory to arrive at warehouse in two time steps
(8) inventory to arrive at warehouse in three time steps
(9) inventory to arrive at warehouse in four time steps
(10) inventory to arrive at warehouse in five time steps
(11)-(20) the squares of (1)-(10)
(21) variance among stores of inventory levels
(22) variance among stores of inventory levels plus inventory to arrive in one time step
(23) variance among stores of inventory levels plus inventory to arrive within two time steps
(24) variance among stores of inventory levels plus inventory to arrive within three time steps
(25) the product of (1) and (5)
(26) the product of (5) and the sum of (1) through (4)
(27) the sum of (5) through (10) times the sum of (1) through (4)
(28) the sum of (5) through (8) times the sum of (1) through (4)
(29) the product of (4), (5), and (10)

The feature values were normalized using the same approach as in the previous case study, except that this time, the scaling parameter associated with feature (4) was the one that needed to be increased.

Noise terms were once again added to the orders during execution of the temporal-difference algorithm. The noise terms used here were exactly the same as the smaller noise terms of the two tried in the previous section.

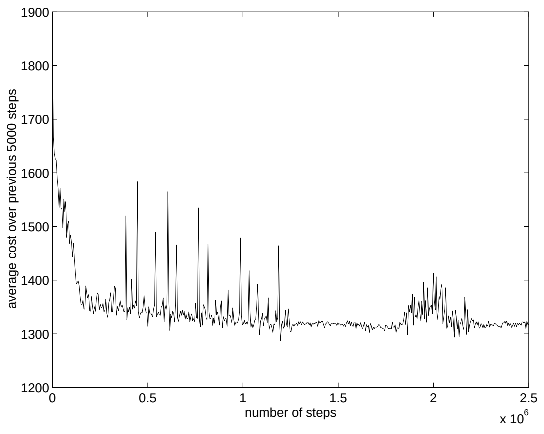

Figure 10: Evolution of average cost using the on-line temporal-difference method with a linear architecture and a small degree of exploration. Each point represents cost averaged over five thousand consecutive time steps during the execution of an algorithm. The final average cost associated with the approximation scheme after this training (without exploration) was 1318 (versus 1449 for the heuristic).

We used a feature-based linear architecture with the features we have described. Figure 10 charts the evolution of average cost during the execution of the temporal-difference algorithm. The step size was $\gamma_t = 0.0001$ during the first million steps and $\gamma_t = 0.00001$ thereafter.

Once again, the on-line temporal-difference method generated a policy superior to the heuristic. The average cost was 1318, a savings of almost ten percent relative to the heuristic's average cost of 1449.

8 Conclusions

Through this study, we have demonstrated that NDP can provide a viable approach to advancing the state-of-the-art in retailer inventory management. The method we have developed outperformed a well-accepted heuristic approach in two case studies.

Though the problems we solved in this research were truly complex from a technical standpoint, not much effort was directed at ensuring that the models reflected all the practical issues inherent in real-world retailer inventory systems. Further research is required to translate the methods we have developed into those that could be truly beneficial in a real-world application.

Finally, it may be possible to improve the results obtained in this research. Alternative choices of architectures and algorithms may lead to further reductions in inventory costs. Also, since performance is measured in terms of average cost, formulating the problem in terms of average–cost (rather than discounted cost) dynamic programming and employing NDP algorithms that directly address such formulations may enhance performance. Such algorithms are discussed in (Bertsekas and Tsitsiklis, 1996) and analyzed in (Abounadi et al., 1996) and (Tsitsiklis and Van Roy, 1996).

Acknowledgments

We would like to thank colleagues at Unica and the Laboratory for Information and Decision Systems for useful discussions, feedback, proof–reading, and help with various other matters. In particular, special thanks go to Ruby Kennedy, Bob Crites, and Steve Patek.

A State Equations

In this appendix, we formalize the retailer inventory system model by providing explicit state equations. The purpose here is to make our description precise using mathematical notation. We do not intend to generate a better intuitive understanding of the model dynamics, which have already been discussed at length in Section 3.

Recall that, for each time t, there are associated pre–decision and post–decision states, denoted by x_t and y_t, respectively. Each post–decision state is given by $y_t = f_2(x_t, u_t)$, for some function f_2, where u_t is a decision representing orders placed at time t. On the other hand, each pre–decision state is given by $x_{t+1} = f_1(y_t, w_t)$, for some function f_2, where w_t is a random variable representing demands that arise at time t. To formally define the dynamics of the model, we will describe the structure of the state vectors x_t and y_t, the decisions u_t, the random variables w_t, and the system functions f_1 and f_2.

A.1 State Vectors

Both pre–decision and post–decision states represent quantities of inventory contained in buffers of the system (see Figure 2). Suppose we have K stores indexed by $i = 1, \ldots, K$. Let $q_{0,T}$ be the quantity of inventory that is currently being transported and will arrive at the warehouse in T days. Similarly, let $q_{i,T}$ be the quantity to arrive at the ith store in T days. We use $q_{0,0}$ and $q_{1,0}, \ldots, q_{K,0}$ here to represent the current levels of inventory at the warehouse and the stores. Let D_w and D_s be the delays for transportation of goods to the warehouse and from the warehouse to the stores. Then, a vector

$$x = \left(q_{0,0}, \ldots, q_{0,D_w}, q_{1,0}, \ldots, q_{1,D_s}, \ldots, q_{K,0}, q_{K,D_s}\right)$$

captures all relevant information concerning current inventory levels. The vectors x_t and y_t take on this general structure.

A.2 Decisions

We represent decisions by vectors of the form

$$u_t = \left(a_0, a_1, \ldots, a_K\right),$$

where a_0 denotes a warehouse order and $a_1, \ldots, a_k$ denote the store orders. In order to enforce that orders and inventory levels are positive and that storage and production capacities are not exceeded, several constraints are placed on the decision space. Given a current pre–decision state

$$x_t = \left(q_{0,0}, \ldots, q_{0,D_w}, q_{1,0}, \ldots, q_{1,D_s}, \ldots, q_{K,0}, q_{K,D_s}\right),$$

the constraints on u_t are captured by the following inequalities:

$$\begin{array}{rcll} a_i & \geq & 0, & \forall i \in \{0, \ldots, K\}, \\ a_0 & \leq & C_p, & \\ \sum_{i=1}^{K} a_i & \leq & q_{0,0}, & \\ a_0 & \leq & C_w - \sum_{T=0}^{D_w} q_{0,T} + \sum_{i=1}^{K} a_i, & \\ a_i & \leq & C_s - \sum_{T=0}^{D_s} q_{i,T}, & \forall i \in \{1, \ldots, K\}, \end{array}$$

where C_p denotes the production capacity, C_w denotes the warehouse capacity, and C_s denotes the store capacity.

A.3 Random Variables

The vectors w_t reflect all random factors that can influence the system during the given time period. This includes the demands that arise at each store as well as the willingness of each customer to place a special order in the event of a shortage. We employ a representation of the form

$$w = \Big(d_1, \ldots, d_K, b\Big),$$

where each d_i is the demand that arises at the ith store on a given day and b is a scalar in $[0, 1)$ that we will interpret as a string of bits by taking the binary representation. Each d_i is generated according to

$$d_i = \left\lceil z_i - \frac{1}{2} \right\rceil^+,$$

where each z_i is independently sampled and normally distributed with

$$z_i \sim \mathrm{N}(\mu, \sigma),$$

where μ and σ are the mean and standard deviation parameters used in defining the model.

The bit string $b = (b_1, b_2, \ldots)$ provides information about the willingness of individual customers to wait for special deliveries. Each bit b_j is an independent Bernoulli sample that is equal to 1 with probability P_w, where P_w is the probability that a customer is willing to wait. We denote by H a hashing function that associates to each of the $\sum_{i=1}^{K} d_i$ units of demand an index $H(i, j)$, where i is the index of a particular store and j is the index of a customer arriving at that store on the given day. We associate with $b_{H(i,j)} = 1$ the fact that the customer would be willing to wait for a special delivery. We do not elaborate the details of this hashing function since they are inconsequential so long as the function is one–to–one (i.e., each customer gets mapped to a different index).

A.4 System Functions

To complete our model description, we must define the two system functions f_1 and f_2. Let us start by defining the transformation $y_t = f_2(x_t, u_t)$ from pre–decision to post–decision states. Suppose that a vector x_t is given by

$$x_t = \Big(q_{0,0}, \ldots, q_{0,D_w}, q_{1,0}, \ldots, q_{1,D_s}, \ldots, q_{K,0}, q_{K,D_s}\Big).$$

In terms of the buffer inventory levels, the decisions bear immediate consequences upon the the quantities q_{0,D_w}, $q_{0,0}$, and $q_{1,D_s}, \ldots, q_{K,D_s}$. In particular, given a decision

$$u_t = \Big(a_0, a_1, \ldots, a_K\Big),$$

the new quantities are given by

$$\begin{aligned}
\bar{q}_{0,D_w} &= q_{0,D_w} + a_0, \\
\bar{q}_{0,0} &= q_{0,0} - \sum_{i=1}^{K} a_i, \\
\bar{q}_{i,D_s} &= q_{i,D_s} + a_i, \qquad \forall i \in \{1, \ldots, K\},
\end{aligned}$$

where the post–decision state is

$$y_t = \Big(\bar{q}_{0,0}, \ldots, \bar{q}_{0,D_w}, \bar{q}_{1,0}, \ldots, \bar{q}_{1,D_s}, \ldots, \bar{q}_{K,0}, \bar{q}_{K,D_s}\Big).$$

Now let

$$\begin{aligned}
y_t &= \Big(q_{0,0}, \ldots, q_{0,D_w}, q_{1,0}, \ldots, q_{1,D_s}, \ldots, q_{K,0}, q_{K,D_s}\Big), \\
w_t &= \Big(d_1, \ldots, d_K, b\Big), \\
x_{t+1} &= \Big(\bar{q}_{0,0}, \ldots, \bar{q}_{0,D_w}, \bar{q}_{1,0}, \ldots, \bar{q}_{1,D_s}, \ldots, \bar{q}_{K,0}, \bar{q}_{K,D_s}\Big).
\end{aligned}$$

In order to simplify the equations involved, we break our definition of the transformation $x_{t+1} = f_1(y_t, w_t)$ into three steps, using $\hat{q}_{i,0}, i \in \{0, \ldots, K\}$, as intermediate variables. First, demands are filled by stores according to

$$\hat{q}_{i,0} = [q_{i,0} - d_i]^+, \qquad \forall i \in \{1, \ldots, K\}.$$

Second, special orders are filled by the warehouse according to

$$\hat{q}_{0,0} = \left[q_{i,0} - \sum_{i=1}^{K} \sum_{j=1}^{[d_i - q_{i,0}]^+} b_{H(i,j)} \right]^+ .$$

Finally, transportation of goods progresses according to

$$\begin{aligned}
\bar{q}_{0,0} &= \hat{q}_{0,0} + q_{0,1}, \\
\bar{q}_{0,T} &= q_{0,T+1}, \qquad \forall T \in \{1, \ldots, D_w - 1\}, \\
\bar{q}_{0,D_w} &= 0, \\
\bar{q}_{i,0} &= \hat{q}_{i,0} + q_{i,1}, \qquad \forall i \in \{1, \ldots, K\}, \\
\bar{q}_{i,T} &= q_{i,T+1}, \qquad \forall i \in \{1, \ldots, K\}, T \in \{1, \ldots, D_s - 1\}, \\
\bar{q}_{i,D_s} &= 0.
\end{aligned}$$

References

Abounadi, J., Bertsekas, D.P., and Borkar, V.S. (1996) "ODE Analysis for Q–Learning Algorithms," Lab for Information and Decision Systems Draft Report, Massachusetts Institute of Technology, Cambridge, MA.

Bertsekas, D. P. (1995) *Dynamic Programming and Optimal Control*, Athena Scientific, Bellmont, MA.

Bertsekas, D. P., and Tsitsiklis, J. N. (1996) *Neuro-Dynamic Programming*, Athena Scientific, Bellmont, MA.

Crites, R. H., and Barto, A. G (1996) "Improving Elevator Performance Using Reinforcement Learning," *Advances in Neural Information Processing Systems 8*, Touretzky, D. S., Mozer, M. C., and Hasselmo, M. E., eds., MIT Press, Cambridge, MA.

Lee, H. L., and Billington, C. (1993) "Material Management in Decentralized Supply Chains," *Operations Research*, vol. 41, no. 5, pp. 835-847.

Nahmias, S., and Smith, S. A. (1993) "Mathematical Models of Inventory Retailer Systems: A Review," *Perspectives on Operations Management, Essays in Honor of Elwood S. Buffa*, Sarin, R., editor, Kluwer Academic Publishers, Boston, MA, pp. 249-278.

Nahmias, S., and Smith, S. A. (1994) "Optimizing Inventory Levels in a Two Echelon Retailer System with Partial Lost Sales," *Management Science*, Vol. 40, pp. 582-596.

Sutton, R. S. (1988) "Learning to Predict by the Methods of Temporal Differences," *Machine Learning*, vol. 3.

Tesauro, G. J. (1992) "Practical Issues in Temporal–Difference Learning," *Machine Learning*, vol. 8.

Tsitsiklis, J. N., and Van Roy, B. (1996) "An Analysis of Temporal–Difference Learning with Function Approximation," to appear in *IEEE Transactions on Automatic Control.*

Tsitsiklis, J. N., and Van Roy, B. (1996) "Average Cost Temporal–Difference Learning," working paper.

Zhang, W., and Dietterich, T. G. (1995) "A Reinforcement Learning Approach to Job Shop Scheduling," *Proceedings of the IJCAI.*