2019

Applications of Machine Learning in Supply Chains

Afshin Oroojlooy
Lehigh University, oroojlooy@gmail.com

Applications of Machine Learning in Supply Chains

by

Afshin Oroojlooy

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Doctor of Philosophy

in

Industrial and Systems Engineering

Lehigh University

January 2019

Approved and recommended for acceptance as a dissertation in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Date

Dissertation Advisor

Accepted Date

Committee Members:

Lawrence V. Snyder, Committee Chair

Martin Takáč

Frank E. Curtis

Ioannis Akrotirianakis

Acknowledgments

First and foremost, I would like to express my greatest gratitude to my advisor, Professor Larry Snyder, in providing me guidance and support from the beginning of my PhD. I appreciate his patience, expertise, and great help through the last four years of my life. Without his guidance and persistent help, this dissertation would not have been possible.

I would like to thank my committee members, Professor Curtis, Dr. Akrotirianakis, and Professor Takáč for all the helpful ideas that they shared with me. I especially appreciate all the help, and assistance that Professor Takáč provided at all levels of my research projects. I am also thankful to Kathy Rambo, Rita Frey, and Ana Quiroz, who always help me to go through all administration challenges.

I also acknowledge the generous funding by NSF grants CCF:1618717, CMMI:1663256, and CCF:1740796 that supported this dissertation.

I would like to thank my family for the support that they provided me through my entire life. I am grateful to my mother, father, and my sister and I would not have made it this far without them. Finally, I thank my wife, and my best friend. Without her patience, help, and support I would not have finished my dissertation.

Contents

List of Tables

List of Figures

Abstract

Advances in new technologies have resulted in increasing the speed of data generation and accessing larger data storage. The availability of huge datasets and massive computational power have resulted in the emergence of new algorithms in artificial intelligence and specifically machine learning, with significant research done in fields like computer vision. Although the same amount of data exists in most components of supply chains, there is not much research to utilize the power of raw data to improve efficiency in supply chains. In this dissertation our objective is to propose data-driven non-parametric machine learning algorithms to solve different supply chain problems in data-rich environments. Among wide range of supply chain problems, inventory management has been one of the main challenges in every supply chain. The ability to manage inventories to maximize the service level while minimizing holding costs is a goal of many company. An unbalanced inventory system can easily result in a stopped production line, back-ordered demands, lost sales, and huge extra costs. This dissertation studies three problems and proposes machine learning algorithms to help inventory managers reduce their inventory costs.

In the first problem, we consider the newsvendor problem in which an inventory manager needs to determine the order quantity of a perishable product to minimize the sum of shortage and holding costs, while some feature information is available for each product. We propose

a neural network approach with a specialized loss function to solve this problem. The neural network gets historical data and is trained to provide the order quantity. We show that our approach works better than the classical separated estimation and optimization approaches as well as other machine learning based algorithms. Especially when the historical data is noisy, and there is little data for each combination of features, our approach works much better than other approaches. Also, to show how this approach can be used in other common inventory policies, we apply it on an (r, Q) policy and provide the results. This algorithm allows inventory managers to quickly determine an order quantity without obtaining the underling demand distribution.

Now, assume the order quantities or safety stock levels are obtained for a single or multi-echelon system. Classical inventory optimization models work well in normal conditions, or in other words when all underlying assumptions are valid. Once one of the assumptions or the normal working conditions is violated, unplanned stock-outs or excess inventories arise. To address this issue, in the second problem, a multi-echelon supply network is considered, and the goal is to determine the nodes that might face a stock-out in the next period. Stock-outs are usually expensive and inventory managers try to avoid them, so stock-out prediction might results in averting stock-outs and the corresponding costs. In order to provide such predictions, we propose a neural network model and additionally three naive algorithms. We analyze the performance of the proposed algorithms by comparing them with classical forecasting algorithms and a linear regression model, over five network topologies. Numerical results show that the neural network model is quite accurate and obtains accuracies in $[0.92, 0.99]$ for the hardest to easiest network topologies, with average of 0.950 and standard deviation of 0.023, while the closest competitor, i.e., one of the proposed naive algorithms, obtains accuracies in $[0.91, 0.95]$ with average of 9.26 and standard deviation of .0136.

Additionally, we suggest conditions under which each algorithm is the most reliable and additionally apply all algorithms to threshold and multi-period predictions.

Although stock-out prediction can be very useful, any inventory manager would like to have a powerful model to optimize the inventory system and balance the holding and shortage costs. The literature on multi-echelon inventory models is quite rich, though it mostly relies on the assumption of accessing a known demand distribution. The demand distribution can be approximated, but even so, in some cases a globally optimal model is not available. In the third problem, we develop a machine learning algorithm to address this issue for multi-period inventory optimization problems in multi-echelon networks. We consider the well-known beer game problem and propose a reinforcement learning algorithm to efficiently learn ordering policies from data. The beer game is a serial supply chain with four agents, i.e. retailer, wholesaler, distributor, and manufacturer, in which each agent replenishes its stock by ordering beer from its predecessor. The retailer satisfies the demand of external customers, and the manufacturer orders from external suppliers. Each of the agents must decide its own order quantity to minimize the summation of holding and shortage cost of the system, while they are not allowed to share any information with other agents. For this setting, a base-stock policy is optimal, if the retailer is the only node with a positive shortage cost and a known demand distribution is available. Outside of this narrow condition, there is not a known optimal policy for this game. Also, from the game theory point of view, the beer game can be modeled as a decentralized multi-agent cooperative problem with partial observability, which is known as a NEXP-complete problem. We propose an extension of deep Q-network for making decisions about order quantities in a single node of the beer game. When the co-players follow a rational policy, it obtains a close-to-optimal solution, and it works much better than a base-stock policy if the other

agents play irrationally. Additionally, to reduce the training time of the algorithm, we propose using transfer learning, which reduces the training time by one order of magnitude. This approach can be extended to other inventory optimization and supply chain problems.

Chapter 1

Introduction

Classical supply chain models require the decision maker to make assumptions about the probability distributions of the demands, lead times, and other random elements. Fortunately, today's supply chains capture huge volumes of data about these parameters. However, the prevalent approach for utilizing this data, both in research and in practice, involves two stages: First, we use statistics, machine learning (ML), or another tool to estimate each random parameter (often as only a point forecast); and second, we plug those estimates into a classical supply chain model, as though the estimates were perfectly accurate. In our opinion, this approach can be *improved*.

Instead, we propose an integrated approach that combines the data-analysis and supply-chain-optimization stages into a single ML algorithm. We apply this idea in three problems, which are each presented in one chapter of this dissertation. We have applied this approach to the newsvendor problem in chapter 2: We assume we have historical data with no knowledge of the demand distribution's shape or parameters. Rather than the two-stage approach (which uses ML to estimate the mean and/or standard deviation of the demand from the data, then plugs those into the classical newsvendor problem to obtain an order

quantity), we have designed a ML algorithm that is trained to use the historical data to choose the order quantity directly, without generating an explicit demand forecast. Our results show that our approach gives better results than the two-stage approach or other data-driven newsvendor algorithms in the literature. Additionally, we apply this approach to an (r, Q) policy to show the generalizability of our method to other supply chain problems.

In chapter 3, we address stock-out prediction in multi-echelon networks. Stock-outs are expensive and common in supply chains and companies utilize different approaches to minimize the corresponding costs. Most of approaches are designed to target a given stock-out percentage; however, very few provide information about when they may happen. There is very little research on stock-out prediction in single node systems and even less in multi-echelon systems. In multi-echelon systems the performance of a given node is heavily affected by other nodes of the system and it is too complicated to predict it using state-of-the-art approaches, like probabilistic models. We propose a ML approach which uses the corresponding historical data and predict stock-outs for all nodes of the system.

Finally, in chapter 4, we provide a reinforcement learning algorithm to make inventory decisions in multi-echelon systems with several periods. We tackle the beer game problem, which is a widely used in-class game that is played in supply chain management classes to demonstrate a phenomenon known as the bullwhip effect. The game is a decentralized, multi-agent, cooperative problem that can be modeled as a serial supply chain network in which agents cooperatively attempt to minimize the total cost of the network even though each agent can only observe its own local information. We develop a ML algorithm to solve this problem. Our results show that the algorithm works well when an agent that follows our approach plays with other rational or irrational agents. Unlike most algorithms in the literature, our algorithm does not have any limits on the parameter values, and it provides

good solutions even if the agents do not follow a rational policy. Moreover, it does not make any assumption about the probability distribution of the demand, and it works with any data set, even if the form of the demand distribution is unknown. Finally, in order to reduce the training time, we propose using transfer learning and show that it can reduce the training time by one order of magnitude. The algorithm can be extended to other decentralized multi-agent cooperative games with partially observed information, which is a common type of situation in supply chain problems.

Chapter 2

The Multi-Feature Newsvendor Problem

The newsvendor problem is one of the most basic and widely applied inventory models. There are numerous extensions of this problem. If the probability distribution of the demand is known, the problem can be solved analytically. However, approximating the probability distribution is not easy and is prone to error; therefore, the resulting solution to the newsvendor problem may be not optimal. To address this issue, we propose an algorithm based on deep learning that optimizes the order quantities for all products based on features of the demand data. Our algorithm integrates the forecasting and inventory-optimization steps, rather than solving them separately, as is typically done, and does not require knowledge of the probability distributions of the demand. Numerical experiments on real-world data suggest that our algorithm outperforms other approaches, including data-driven and machine learning approaches, especially for demands with high volatility. Finally, in order to show how this approach can be used for other inventory optimization

problems, we provide an extension for (r, Q) policies.

2.1 Introduction

The newsvendor problem optimizes the inventory of a perishable good. Perishable goods are those that have a limited selling season; they include fresh produce, newspapers, airline tickets, and fashion goods. The newsvendor problem assumes that the company purchases the goods at the beginning of a time period and sells them during the period. At the end of the period, unsold goods must be discarded, incurring a *holding cost*. In addition, if it runs out of the goods in the middle of the period, it incurs a *shortage cost*, losing potential profit. Therefore, the company wants to choose the order quantity that minimizes the expected sum of the two costs described above. The problem dates back to Edgeworth [1888]; see Porteus [2008] for a history and Zipkin [2000], Porteus [2002], and Snyder and Shen [2019], among others, for textbook discussions.

The optimal order quantity for the newsvendor problem can be obtained by solving the following optimization problem:

$$\min_y C(y) = E_d \left[c_p (d - y)^+ + c_h (y - d)^+ \right], \tag{2.1}$$

where d is the random demand, y is the order quantity, c_p and c_h are the per-unit shortage and holding costs (respectively), and $a^+ := \max\{0, a\}$. In the classical version of the problem, the shape of the demand distribution (e.g., normal) is known, and the distribution parameters are either known or estimated using available (training) data. If $F(\cdot)$ is the cumulative density function of the demand distribution and $F^{-1}(\cdot)$ is its inverse, then the

optimal solution of (2.1) can be obtained as

$$y^* = F^{-1}\left(\frac{c_p}{c_p + c_h}\right) = F^{-1}(\alpha), \tag{2.2}$$

where $\alpha = c_p/(c_p + c_h)$ (see, e.g., Snyder and Shen [2019]).

Extensions of the newsvendor problem are too numerous to enumerate here (see Choi [2012] for examples); instead, we mention two extensions that are relevant to our model. First, in real-world problems, companies rarely manage only a single item, so it is important for the model to provide solutions for multiple items. (We do not consider substitution, demand correlation, and complementarity effects as Bassok et al. [1999] and Nagarajan and Rajagopalan [2008] do for the multi-product newsvendor problem.) Second, companies often have access to some additional data—called *features*—along with the demand information. These might include weather conditions, day of the week, month of the year, store location, etc [Rudin and Vahn, 2013]. The goal is to choose today's base-stock level, given the observation of today's features. We will call this problem the multi-feature newsvendor (MFNV) problem. In this chapter, we propose an approach for solving this problem that is based on deep learning, specifically, deep neural networks (DNN).

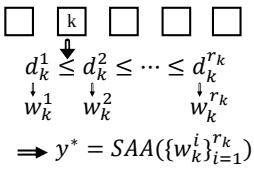
The remainder of this chapter is structured as follows. A brief summary of the literature relevant to the MFNV problem is presented in Section 2.2. Section 2.3 presents the details of the proposed algorithm. Numerical experiments are provided in Section 2.4. Section 2.5 introduces an extension of the approach for (r, Q) policies, and the conclusion and a discussion of future research complete the chapter in Section 2.6.

2.2 Literature Review

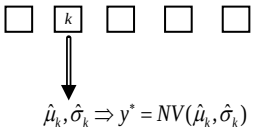
2.2.1 Current State of the Art

Currently, there are five main approaches in the literature related to MFNV. The first category, which we will call the *estimate-as-solution* (EAS) approach, suggests forecasting the demand and then using it for the order quantity. Although EAS cannot be compared to an actual MFNV solution—like the latter four approaches—it is common in practice. This approach involves first clustering the demand observations, then forecasting the demand, and then simply treating the point forecast as a deterministic demand value, i.e., setting the newsvendor solution equal to the forecast. (See Figure 2.1e, which shows cluster k and the order quantity, which is simply the forecast.) By clustering, we mean that all demand observation that have same feature values are put together in a set, called a cluster. For example, when there are 100 demand records for two products in two stores, there are four clusters, and on average each cluster has 25 records. The forecast may be performed in a number of ways, some of which we review in the next few paragraphs.

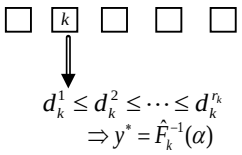
This approach ignores the key insight from the newsvendor problem, namely, that we should not simply order up to the mean demand, but rather choose a level that strikes a balance between underage and overage costs using the distribution of the demand. Nevertheless, the approach is common in the literature. For example, Yu et al. [2013] propose a support vector machine (SVM) model to forecast newspaper demands at different types of stores, along with 32 other features. Wu and Akbarov [2011] use a weighted support vector regression (SVR) model to forecast warranty claims; their model gives more priority to the most recent warranty claims. Chi et al. [2007] propose an SVM model to determine the replenishment point in a vendor-managed replenishment system, and a genetic algorithm is



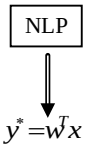
(a) K-nearest neighbors (KNN) and random forest (RF).



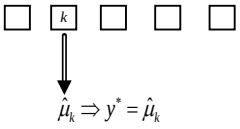
(b) Separated estimation and optimization (SEO).



(c) Empirical quantile (EQ).



(d) Linear machine learning (LML).



(e) Estimate-as-solution (EAS).

Figure 2.1: Approaches for solving MFNV problem. Squares represent clusters.

used to solve it. Carbonneau et al. [2008] present a least squares SVM (LS-SVM) model to forecast a manufacturer's demand. They compare it with standard forecasting methods such as average, moving average, trend, and multiple linear regression, as well as neural network and recurrent neural network algorithms. According to their numerical experiments, the recurrent neural network and the LS-SVM algorithm have the best results. Ali and Yaman [2013] forecast grocery sales, with datasets containing millions of records, and for each record there are thousands of features. They reduce the number of features and data and use an SVM to solve the problem. Since general SVM methods are not able to solve such a large problem, they propose an algorithm to reduce the number of rows and columns of the datasets with a small loss of accuracy. Lu and Chang [2014] propose an iterative algorithm to predict sales. They use independent component analysis (ICA) to obtain hidden features of their datasets, k-mean clustering to cluster the sales data, and finally SVR to provide the prediction. Viaene et al. [2000] propose an LS-SVM classifier model with 25 features

to predict whether a direct mail offer will result in a purchase. Since a huge dataset was available, an iterative approach based on the Hestenes–Stiefel conjugate gradient algorithm was proposed to solve the model.

Classical parametric approaches for forecasting include ARIMA, TRANSFER, and GARCH models [Box et al., 2015, Shumway and Stoffer, 2010]; these are also used for demand forecasting (see Cardoso and Gomide [2007], Shukla and Jharkharia [2011]). Similarly, Taylor [2000] uses a normal distribution to forecast demand one or more time steps ahead; however, his model does not perform well when demands are correlated over time and when the demands are volatile. These and other limitations have motivated the use of DNN to obtain demand forecasts. For example, Efendigil et al. [2009] propose a DNN model to forecast demand based on recent sales, promotions, product quality, and so on. Vieira [2015] proposes a deep learning algorithm to predict online activity patterns that result in an online purchase. Taylor [2000], Kourentzes and Crone [2010], Cannon [2011], and Xu et al. [2016] use DNN for quantile regression, with applications to exchange rate forecasting, for example. For reviews of the use of DNN for forecasting, see Ko et al. [2010], Kourentzes and Crone [2010], Qiu et al. [2014b], and Crone et al. [2011].

The common theme in all of the papers in the last two paragraphs is that they provide only a forecast of the demand, which must then be treated as the solution to the MFNV or other optimization problem. This is the EAS approach.

The second approach for solving MFNV-type problems, which Rudin and Vahn [2013] refer to as *separated estimation and optimization* (SEO), involves first estimating (forecasting) the demand distribution and then plugging the estimate into an optimization problem such as the classical newsvendor problem. The estimation step is performed similarly as in the EAS approach except that we estimate more than just the mean. For example, we might

estimate both mean (μ_k) and standard deviation (σ_k) for each cluster, which we can then use in the optimization step. (See Figure 2.1b.) Or we might use the σ that was assumed for the error term in a regression model. The main disadvantage of this approach is that it requires us to assume a particular form of the demand distribution (e.g., normal), whereas empirical demand distributions are often unknown or do not follow a regular form. A secondary issue is that we compound the data-estimation error with model-optimality error. Rudin and Vahn [2013] show that for some realistic settings, the SEO approach is provably suboptimal. This idea is used widely in practice and in the literature; a broad list of research that uses this approach is given by Turken et al. [2012]. Rudin and Vahn [2013] analyze it as a straw-man against which to compare their solution approach.

The third approach was proposed by Bertsimas and Thiele [2005] for the classical newsvendor problem. Their approach involves sorting the demand observations in ascending order $d_1 \leq d_2 \leq \cdots \leq d_n$ and then estimating the αth quantile of the demand distribution, $F^{-1}(\alpha)$, using the observation that falls $100\alpha\%$ of the way through the sorted list, i.e., it selects the demand d_j such that $j = \lceil n\frac{c_p}{c_p+c_h} \rceil$. This quantile is then used as the base-stock level, in light of (2.2). Since they approximate the αth quantile, we refer to their method as the *empirical quantile* (EQ) method. (See Figure 2.1c.) Importantly, EQ does not assume a particular form of the demand distribution and does not approximate the probability distribution, so it avoids those pitfalls. However, an important shortcoming of this approach is that it does not use the information from features. In principle, one could extend their approach to the MFNV by first clustering the demand observations and then applying their method to each cluster. However, similar to the classical newsvendor algorithm, this would only allow it to consider categorical features and not continuous features, which are common in supply chain demand data, e.g., Ali and Yaman [2013] and Rudin and Vahn

[2013]. Moreover, even if we use this clustering approach, the method cannot utilize any knowledge from other data clusters, which contain valuable information that can be useful for all other clusters. Finally, when there is volatility among the training data, the estimated quantile may not be sufficiently accurate, and the accuracy of the EQ approach tends to be worse.

In the newsvendor problem, the optimal solution is a given quantile of the demand distribution. Thus, the problem can be modeled as a quantile regression problem, in a manner similar to the empirical quantile model of Bertsimas and Thiele [2005]. Taylor [2000] was the first to propose the use of neural networks as a nonlinear approximator of the quantile regression to get a conditional density of multi-period financial returns. Subsequently, several papers used quantile-regression neural networks to obtain a quantile regression value. For example, Cannon [2011] uses a quantile-regression neural network to predict daily precipitation; El-Telbany [2014] uses it to predict drug activities; and Xu et al. [2016] uses a quantile autoregression neural network to evaluate value-at-risk. One can consider our approach as a quantile-regression neural network for the newsvendor problem. However, our approach is much more general and can be applied to other inventory optimization problems, provided that a closed-form cost function exists. To demonstrate this, in Section 2.5 we extend our approach to solve an inventory problem that does not have a quantile-type solution, namely, optimizing the parameters of an (r, Q) policy.

A fourth approach for solving MFNV-type problems can be derived from the method proposed by Bertsimas and Kallus [2014], which applies several ML methods on a general optimization problem given by

$$z^*(x) = \underset{z}{\operatorname{argmin}}\, \mathbb{E}\left[c(z, y)|x\right], \tag{2.3}$$

where $\{(x_1, y_1), \ldots, (x_N, y_N)\}$ are the available data—in particular, x_i is a d-dimensional vector of feature values and y_i is the uncertain quantity of interest, e.g., demand values—and z is the decision variable. They test five algorithms to optimize (2.3): k-nearest neighbor (KNN), random forest (RF), kernel method, classification and regression trees (CART), and locally weighted scatterplot smoothing (LOESS). They use sample average approximation (SAA) as a baseline, and each algorithm provides substitute weights for the SAA method. For example, KNN identifies the set of k nearest historical records to the new observation x such that

$$\mathcal{N}(x) = \left\{ i = 1, \ldots, n : \sum_{j=1}^{n} \mathbb{I}\{||x - x_i|| \geq ||x - x_j||\} \leq k \right\}.$$

Bertsimas and Kallus [2014] assign weights $w_i = 1/k$ for all $i \in \mathcal{N}(x)$ (and zero otherwise) and call a weighted SAA; for example, if applied to the newsvendor problem, the SAA might take the form

$$q = \inf \left\{ d_j : \sum_{i=1}^{j} w_i \geq \frac{c_p}{c_p + c_h} \right\}, \tag{2.4}$$

where d_j are the ascending sorted demands (see Figure 2.1a). Similarly, in RF, there are T trees. The weight of each observation is obtained using

$$w_i = \frac{1}{T} \sum_{t=1}^{T} \frac{\mathbb{I}\{R^t(x) = R^t(x_i)\}}{|\{j : R^t(x_j) = R^t(x^i)\}|},$$

where $R^t(x)$ is the region of tree t that observation x is in. In other words, the RF algorithm counts all trees in which the new observation x is in the same region as historical observation x_i, $i = 1, \ldots, n$, and normalizes them over all observations in tree t that have the same region. Finally, it normalizes the weights over all trees. Using these weights, the method of Bertsimas and Kallus [2014] as applied to the newsvendor problem calls the weighted SAA

(2.4) to get the order quantity. Bertsimas and Kallus [2014] discuss asymptotic convergence of their methods and compare their performance with that of SAA.

The fifth approach for the MFNV, and the one that is closest to our proposed approach, was introduced by Rudin and Vahn [2013]; we refer to it as the *linear machine learning* (LML) method. They postulate that the optimal base-stock level is related to the demand features via a linear function; that is, that $y^* = w^T x$, where x is the vector of features and w is a vector of (unknown) weights.

They estimate these weights by solving the following nonlinear optimization problem, essentially fitting the solution using the newsvendor cost:

$$\begin{aligned} &\min_w \ \tfrac{1}{n}\textstyle\sum_{i=1}^{n}\left[c_p(d_i - w^T x_i)^+ + c_h(w^T x_i - d_i)^+\right] + \lambda||w||_k^2 \\ &\text{s.t. } (d_i - w^T x_i)^+ \geq d_i - w_1 - \sum_{j=2}^{p} w_i x_i^j; \quad \forall i = 1,\ldots,n \\ &\qquad (w^T x_i - d_i)^+ \geq w_1 + \sum_{j=2}^{p} w_i x_i^j - d_i; \quad \forall i = 1,\ldots,n \end{aligned} \tag{2.5}$$

where n is the number of observations, p is the number of features, and $\lambda||w||_k^2$ is a regularization term. The LML method avoids having to cluster the data, as well as having to specify the form of the demand distribution. Rudin and Vahn [2013] comprehensively analyze the effects of adding nonlinear combination of features into the feature space, as well as the effects of regularization and of overfitting. (For more theoretical details on these concepts, see Smola and Schölkopf [2004].) However, this model does not work well when $p > n$ and its learning is limited to the current training data. In addition, if the training data contains only a small number of observations for each combination of the features, the model learns poorly. Finally, it makes the strong assumption that x and y^* have a linear relationship. We drop this assumption in our model and instead use DNN to quantify the

relationship between x and y^*; see Section 2.3. Rudin and Vahn [2013] also propose a kernel regression (KR) model to optimize the order quantity, in which weighted historical demands are used to build an empirical cumulative distribution function (cdf) of the demand. The weights are proportional to the distance between the newly observed feature value and the historical feature values, i.e.,

$$w_i = \frac{K(x - x_i)}{\sum_{j=1}^{n} K(x - x_j)},$$

where $K(u) = \exp(-||u||_2^2/2h)/\sqrt{2\pi}$ and h is the kernel bandwidth, which has to be tuned. Then they call weighted SAA (2.4) to obtain the order quantity. In addition, they provide a mathematical analysis of the generalization errors associated with each method.

There is a large body of literature on data-driven inventory management that assumes we do not know the demand distribution and instead must directly use the data to make a decision. Besbes and Muharremoglu [2013] consider censored data (in which some demands cannot be observed due to stockouts) in the newsvendor problem. The paper proposes three models and algorithms to minimize the regret when real, censored, and partially censored demand are available. They propose an EQ-type algorithm (discussed above) for observable demand. For censored and partially censored demand, they propose two algorithms, as well as lower and upper bounds on the regret value for all algorithms. Burnetas and Smith [2000] propose an adaptive model to optimize price and order quantity for perishable products with an unknown demand distribution, assuming historical data of censored sales are available. They assume that the demand is continuous and propose two algorithms, one for a fixed price and another for the pricing/ordering problem. Their algorithm for choosing the order quantity provides an adaptive policy and works even when there is nearly no historical information, so it is suitable for new products. It starts from an arbitrary point q_0 and

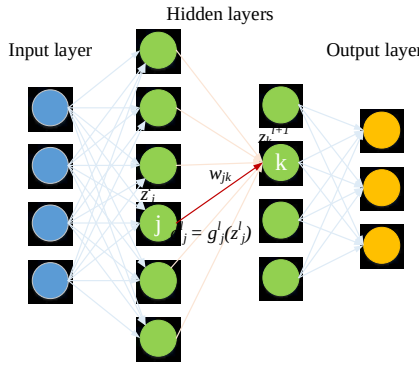
iteratively updates it with some learning rate and information about whether or not the order quantity q_t was sufficient to satisfy the demand in period t.

None of these two papers use features, which is the key aspect of our problem. One data-driven approach that does use features is by Ban et al. [2017], who propose a model to choose the order quantity for new, short-life-cycle products from multiple suppliers over a finite time horizon, assuming that each demand has some feature information. They propose a data-driven algorithm, called the residual tree method, which is an extension of the scenario tree method from stochastic programming, and prove that this method is asymptotically optimal as the size of the data set grows. Their approach has separate steps for estimation (using regression) and optimization (using stochastic linear programming). Although their problem has some similarities to ours, it is not immediately applicable since it is designed for finite-horizon problems with multiple suppliers.

2.2.2 Deep Learning

In this chapter, we develop a new approach to solve the newsvendor problem with data features, based on deep learning. Deep neural networks (DNN), is a branch of machine learning that aims to build a model between inputs and outputs. Deep learning has many applications in image processing, speech recognition, drug and genomics discovery, time series forecasting, weather prediction, and—most relevant to our work—demand prediction. On the other hand, one major criticism of deep learning (in non-vision-based tasks) is that it lacks interpretability—that is, it is hard for a user to discern a relationship between model inputs and outputs; see, e.g. Lipton [2016]. In addition, it usually needs careful hyper-parameter tuning, and the training process can take many hours or even days. We provide only a brief overview of deep learning here; for comprehensive reviews of the algorithm and

Figure 2.2: A simple deep neural network.



its applications, see Goodfellow et al. [2016], Schmidhuber [2015], LeCun et al. [2015], Deng et al. [2013], Qiu et al. [2014a], SHI et al. [2015], and Längkvist et al. [2014].

DNN uses a cascade of many layers of linear or nonlinear functions to obtain the output values from inputs. A general view of a DNN is shown in Figure 2.2. The goal is to determine the weights of the network such that a given set of inputs results in a true set of outputs. A loss function is used to measure the closeness of the outputs of the model and the true values. The most common loss functions are the hinge, logistic regression, softmax, and Euclidean loss functions. The goal of the network is to provide a small loss value, i.e., to optimize:

$$\min_{w} \frac{1}{n} \sum_{i=1}^{n} E(\theta(x_i; w), y_i) + \lambda R(w),$$

where E is the loss function, w is the matrix of the weights, x_i is the vector of the inputs from the ith instance, $\theta(\cdot)$ is the DNN function, and $R(w)$ is a regularization function with weight λ. The regularization term prevents over-fitting and is typically the ℓ_1 or ℓ_2 norm of the weights. (Over-fitting means that the model learns to do well on the training set but does not extend to the out-of-training samples; this is to be avoided.) Finaly, y_i is the target value that DNN wants to predict, and in the context of the newsvendor problem, it

is the optimal order quantity. Though the optimal order quantity is not known, we provide a way to learn it.

In each node j ($j = 1, \ldots, n$) of a layer l ($l = 1, \ldots, L$), the input value

$$z_j^l = \sum_{i=1}^{n} a_i^{l-1} w_{i,j} \tag{2.6}$$

is calculated and the value of the function $g_j^l(z_j^l)$ provides the output value of the node. The function $g_j^l(\cdot)$ is called the activation function; the value of $g_j^l(z_j^l)$ is called the activation of the node, and is denoted by a_j^l. Typically, all nodes in the network have similar $g_j^l(\cdot)$ functions. The most commonly used activation functions are the ReLU ($\max(0, x)$) sigmoid ($1/(1 + e^{-z_j^l})$) and tanh ($(1 - e^{-2z_j^l})/(1 + e^{-2z_j^l})$) functions, which add non-linearity into the model (see more details about them in LeCun et al. [2015], Goodfellow et al. [2016]). The activation function value of each node is the input for the next layer, and finally, the activation function values of the nodes in the last layer determine the output values of the network. The general flow of the calculations between two layers of the DNN, with a focus on z_j^l, a_j^l, w_{jk}, and z_j^{l+1}, is shown in Figure 2.2.

In each DNN, the number of layers, the number of nodes in each layer, the activation function inside each node, and the loss function have to be determined. After selecting those characteristics and building the network, DNN starts with some random initial solution. In each iteration, the activation values and the loss function are calculated. Then, the back-propagation algorithm obtains the gradient of the network and, using one of several optimization algorithms [Rumelhart et al., 1988], the new weights are determined. The most common optimization algorithms are gradient descent, stochastic gradient descent (SGD), SGD with momentum, and the Adam optimizer (for details on each optimization

algorithm see Goodfellow et al. [2016]). This procedure is performed iteratively until some stopping condition is reached; typical stopping conditions are (a) reaching a maximum number of iterations and (b) attaining $||\nabla_w \ell(\theta(x_i; w), y_i)|| \leq \epsilon$ through the back-propagation algorithm.

Since the number of instances, i.e., the number of training records, is typically large, it is common [Goodfellow et al., 2016, Bottou, 2010] to use a stochastic approximation of the objective function. That is, in each iteration, a mini-batch of the instances is selected and the objective is calculated only for those instances. This approximation does not affect the provable convergence of the method. For example, in networks with sigmoid activation functions in which a quadratic loss function is used, the loss function asymptotically converges to zero if either gradient descent or stochastic gradient descent are used [Tesauro et al., 1989, Bottou, 2010].

2.2.3 Our Contribution

To adapt the deep learning algorithm for the newsvendor problem with data features, we propose a revised loss function, which considers the impact of inventory shortage and holding costs. The revised loss function *allows the deep learning algorithm to obtain the minimizer of the newsvendor cost function directly, rather than first estimating the demand distribution and then choosing an order quantity.*

In the presence of sufficient historical data, this approach can solve problems with known probability distributions as accurately as (2.2) solves them. However, the real value of our approach is that it is effective for problems with small quantities of historical data, problems with unknown/unfitted probability distributions, or problems with volatile historical data—all cases for which the current approaches might fail.

2.3 Deep Learning Algorithm for Newsvendor with Data Features

In this section, we present the details of our approach for solving the newsvendor problem with data features. Assume there are n historical demand observations for m products. Also, for each demand observation, the values of p features are known. That is, the data can be represented as

$$\left\{(x_i^1, d_i^1), \ldots, (x_i^m, d_i^m)\right\}_{i=1}^n,$$

where $x_i^q \in \mathbb{R}^p$ and $d_i^q \in \mathbb{R}$ for $i = 1, \ldots, n$ and $q = 1, \ldots, m$. The problem is formulated mathematically in (2.7) for a given period i, $i = 1, \ldots, n$, resulting in the order quantities $y_i^1, \ldots, y_i^m$:

$$E_i = \min_{y_i^1, \ldots, y_i^m} \frac{1}{m} \left[\sum_{q=1}^{m} c_h (y_i^q - d_i^q)^+ + c_p (d_i^q - y_i^q)^+ \right], \tag{2.7}$$

where E_i is the loss value of period i and $E = \frac{1}{n} \sum_{i=1}^n E_i$ is the average loss value. Since at least one of the two terms in each term of the sum must be zero, the loss function (2.7) can be written as:

$$\begin{aligned} E_i &= \sum_{q=1}^{m} E_i^q \\ E_i^q &= \begin{cases} c_p (d_i^q - y_i^q) \ , & \text{if } y_i^q < d_i^q, \\ c_h (y_i^q - d_i^q) \ , & \text{if } d_i^q \leq y_i^q. \end{cases} \end{aligned} \tag{2.8}$$

We set equation (2.7) as the goal for the DNN, i.e. it will find the variables $y_i^1, \ldots, y_i^m$ that obtain the minimum average cost. In other words, for each input, the DNN obtains a single output that is the order quantity for the corresponding input feature. Note that the

variables of the model are the weights of the neural network, i.e., $\{w_{jk}\}$ for all $j = 1, \ldots, nn_l$, $k = 1, \ldots, nn_k$, and $l = 1, \ldots, L$, which connect the inputs and all nodes of the networks to each other. Then, the order quantity y_i^q can explicitly be written as a function of those weights, such that the output of the network, i.e., a_0^L, is the order quantity. To get the optimal order quantity, the DNN iteratively updates the weights of the network to minimize the loss function (2.7) which is the total cost of the newsvendor problem, while the order quantities are also the output of the DNN.

As noted above, there are many studies on the application of deep learning for demand prediction (see SHI et al. [2015]). Most of this research uses the Euclidean loss function (see Qiu et al. [2014b]). However, the demand forecast is an estimate of the first moment of the demand probability distribution; it is not, however, the optimal solution of model (2.7). Therefore, another optimization problem must be solved to translate the demand forecasts into a set of order quantities. This is the separated estimation and optimization (SEO) approach described in Section 2.2.1, which may result in a non-optimal order quantity (Rudin and Vahn [2013]). To address this issue, we propose two loss functions, the newsvendor cost function (2.7) and a revised Euclidean loss function, so that instead of simply predicting the demand, the DNN minimizes the newsvendor cost function. Thus, running the corresponding deep learning algorithm gives the order quantity directly.

We found that squaring the cost for each product in (2.7) sometimes leads to better solutions, since the function is smooth, and the gradient is available in the whole solution space. Therefore, we also test the following revised Euclidean loss function:

$$E_i = \min_{y_i^1, \ldots, y_i^m} \frac{1}{m} \left[\sum_{q=1}^{m} \left[c_p (d_i^q - y_i^q)^+ + c_h (y_i^q - d_i^q)^+ \right]^2 \right] \tag{2.9}$$

which penalizes the order quantities that are far from d_i much more than those that are close. Then we have

$$E_i^q = \begin{cases} \frac{1}{2}||c_p(d_i^q - y_i^q)||_2^2 \ , & \text{if } y_i^q < d_i^q, \\ \frac{1}{2}||c_h(y_i^q - d_i^q)||_2^2 \ , & \text{if } d_i^q \leq y_i^q. \end{cases} \tag{2.10}$$

The two propositions that follow provide the gradients of the loss functions with respect to the weights of the network. In both propositions, i is one of the samples, w_{jk} represents a weight in the network between two arbitrary nodes j and k in layers l and $l+1$,

$$a_j^l = g_j^l(z_j^l) = \frac{\partial(z_k^l)}{\partial w_{jk}} \tag{2.11}$$

is the activation function value of node j, and

$$\begin{aligned} \delta_j^l &= \frac{\partial E_i^q}{\partial z_j^l} \\ &= \frac{\partial E_i^q}{\partial a_j^l}\frac{\partial a_j^l}{\partial z_j^l} \\ &= \frac{\partial E_i^q}{\partial a_j^l}(g_j^l)'(z_j^l). \end{aligned} \tag{2.12}$$

Also, let

$$\begin{aligned} \delta_j^l(p) &= c_p(g_j^l)'(z_j^l) \\ \delta_j^l(h) &= c_h(g_j^l)'(z_j^l) \end{aligned} \tag{2.13}$$

denote the corresponding δ_j^l for the shortage and excess cases, respectively. Proofs of both propositions are provided in Appendix A.

Proposition 1. *The gradient with respect to the weights of the network for loss function* (2.8) *is:*

$$\frac{\partial E_i^q}{\partial w_{jk}} = \begin{cases} a_j^l \delta_j^l(p) & \text{if } y_i^q < d_i^q, \\ a_j^l \delta_j^l(h) & \text{if } d_i^q \leq y_i^q. \end{cases} \tag{2.14}$$

Proposition 2. *The gradient with respect to the weights of the network for loss function* (2.10) *is:*

$$\frac{\partial E_i^q}{\partial w_{jk}} = \begin{cases} (d_i^q - y_i^q) a_j^l \delta_j^l(p), & \text{if } y_i^q < d_i^q \\ (y_i^q - d_i^q) a_j^l \delta_j^l(h), & \text{if } d_i^q \leq y_i^q. \end{cases} \tag{2.15}$$

Our deep learning algorithm uses gradient (2.14) and sub-gradient (2.15) under the proposed loss functions (2.8) and (2.10), respectively, to iteratively update the weights of the networks. In order to obtain the new weights, an SGD algorithm with momentum is called, with a fixed momentum of 0.9. This gives us two different DNN models, using the linear loss function (2.8) and the quadratic loss function (2.10), which we call DNN-ℓ_1 and DNN-ℓ_2, respectively.

In order to obtain a good structure for the DNN network, we follow the HyperBand algorithm [Li et al., 2016]. In particular, we generate 100 fully connected networks with random structures. In each, the number of hidden layers is randomly selected as either two or three (with equal probability). Let nn_l denote the number of nodes in layer l; then nn_1 is equal to the number of features. The number of nodes in each hidden layer is selected randomly based on the number of nodes in the previous layer. For networks

with two hidden layers, we choose $nn_2 \in [0.5nn_1, 3nn_1]$, $nn_3 \in [0.5nn_2, nn_2]$, and $nn_4 = 1$. Similarly, for networks with three hidden layers, $nn_2 \in [0.5nn_1, 3nn_1]$, $nn_3 \in [0.5nn_2, 2nn_2]$, $nn_4 \in [0.5nn_3, nn_3]$, and $nn_5 = 1$. The nn_l values are drawn uniformly from the ranges given. For each network, the learning rate and regularization parameters are drawn uniformly from $[10^{-2}, 10^{-5}]$. In order to select the best network among these, following the HyperBand algorithm, we train each of the 100 networks for one epoch (which is a full pass over the training dataset), obtain the results on the test set, and then remove the worst 10% of the networks. We then run another epoch on the remaining networks and remove the worst 10%. This procedure iteratively repeats to obtain the final best networks.

2.4 Numerical Experiments

In this section, we discuss the results of our numerical experiments. In addition to implementing our deep learning models (DNN-ℓ_1 and DNN-ℓ_2), we implemented the EQ model by Bertsimas and Thiele [2005], modifying it so that first the demand observations are clustered according to the features and then EQ is applied to each cluster. We also implemented the LML and KR models by Rudin and Vahn [2013] and the KNN and RF models by Bertsimas and Kallus [2014], as well as the SEO approach in which we obtained the mean by training a DNN over the feature values and then assuming a normally distributed error term to use formula (2.2). We trained the DNN with both ℓ_1 and ℓ_2 regularizations since we do the same for our DNN approach, and we denote the corresponding results as SEO-ℓ_1 and SEO-ℓ_2. Additionally, we provide the results of another version of the SEO approach by calculating the classical solution from (2.2) with parameters μ and σ set to the mean and standard deviation of the training data in each data cluster. The corresponding results are denoted by

parametric SEO (PSEO). We do not include results for EAS since it is dominated by PSEO: PSEO uses the newsvendor solution based on estimates of μ and σ, whereas EAS simply sets the solution equal to the estimate of μ. In order to compare the results of the various methods, the order quantities were obtained with each algorithm and the corresponding cost function

$$\text{cost} = \sum_{i=1}^{n} \sum_{q=1}^{m} \left[c_p (d_i^q - y_i^q)^+ + c_h (y_i^q - d_i^q)^+ \right]$$

was calculated.

All of the deep learning experiments were done with TensorFlow (Abadi et al. [2016]) in Python. Note that the deep learning, LML, KR, KNN, and RF algorithms are scale dependent, meaning that the tuned parameters of the problem for a given set of cost coefficients do not necessarily work for other values of the coefficients. Thus, we performed a separate tuning for each set of cost coefficients. In addition, we translated the categorical data features to their corresponding binary representations (using one-hot encoding). These two implementation details improve the accuracy of the learning algorithms. All computations were done on 16-core machines with cores of 1.8 GHz computation power and 32 GB of memory.

In what follows, we demonstrate the results of the seven algorithms in three separate experiments. First, in Section 2.4.1, we conduct experiments on a very small data set in order to illustrate the differences among the methods. Second, the results of the seven algorithms on a real-world dataset are presented in Section 2.4.2. Finally, in Section 2.4.3, to determine the conditions under which deep learning outperforms the other algorithms on larger instances, we present the results of the seven approaches on several randomly generated datasets.

Table 2.1: Demand of one item over three weeks.

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Week 1	1	2	3	4	3	2	1
Week 2	6	10	12	14	12	10	10
Week 3	3	6	8	9	8	6	5

2.4.1 Small Data Set

Consider the small, single-item instance whose demands are contained in Table 2.1.

In order to obtain the results of each algorithm, the first two weeks are used for training data and the third week is used for testing. To train the corresponding deep network, a fully connected network with one hidden layer is used. The network has eight binary input nodes for the day of week and item number. The hidden layer contains one sigmoid node, and in the output layer there is one inner product function. Thus, the network has nine variables.

Table 2.2 shows the results of the seven algorithms. The first column gives the cost coefficients. Note that we assume $c_p \geq c_h$ since this is nearly always true for real applications. The table first lists the actual demand for each day, repeated from Table 2.1 for convenience. For each instance (i.e., set of cost coefficients), the table lists the order quantity generated by each algorithm for each day. The last column lists the total cost of the solution returned by each algorithm, and the minimum costs for each instance are given in bold.

First consider the results of the EQ algorithm. The EQ algorithm uses c_h and c_p and returns the historical data value that is closest to the αth fractile, where $\alpha = c_p/(c_h + c_p)$. In this data set, there are only two observed historical data points for each day of the week. In particular, for $c_p/c_h \leq 1$, the EQ algorithm chose the smaller of the two demand values as the order quantity, and for $c_p/c_h > 1$, it chose the larger value. Since the testing data vector is nearly equal to the average of the two training data vectors, the difference between EQ's output and the real demand values is quite large, and consequently so is the cost. This

Table 2.2: Order quantity proposed by each algorithm for each day and the corresponding cost. The bold costs indicate the best newsvendor cost for each instance.

		Day & Demand							
(c_p, c_h)	Algorithm	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Cost
	True demand	3	6	8	9	8	6	5	
	DNN-ℓ_2	3.5	6.0	7.5	9.0	7.5	6.5	5.5	2.5
	DNN-ℓ_1	4.6	6.0	8.5	9.0	8.6	5.6	5.6	2.9
(1,1)	EQ	1.0	2.0	3.0	4.0	3.0	2.0	1.0	29.0
	LML	1.3	2.2	3.1	4.0	4.9	5.8	6.7	20.3
	PSEO	3.5	6.0	7.5	9.0	7.5	6.5	5.5	2.5
	SEO-ℓ_1	3.2	6.0	5.3	6.4	6.4	5.8	5.0	7.3
	SEO-ℓ_2	3.6	6.2	7.9	9.1	7.8	6.7	5.3	**2.0**
	KR	4.0	6.0	6.0	6.0	6.0	4.0	4.0	11.0
	KNN	6.0	6.0	6.0	6.0	6.0	6.0	6.0	11.0
	RF	6.0	6.0	6.0	6.0	6.0	6.0	6.0	11.0
	DNN-ℓ_2	5.8	7.3	8.9	9.7	8.9	8.1	7.0	10.7
	DNN-ℓ_1	6.0	6.0	7.9	9.3	9.3	6.4	6.1	**5.9**
(2,1)	EQ	6.0	10.0	12.0	14.0	12.0	11.0	10.0	30.0
	LML	6.0	7.0	8.0	9.0	10.0	11.0	12.0	18.0
	PSEO	5.0	8.4	10.2	12.0	10.2	9.2	8.2	18.5
	SEO-ℓ_1	5.3	7.0	7.8	8.5	7.8	7.3	7.5	8.9
	SEO-ℓ_2	4.7	6.8	8.8	10.1	8.8	7.5	6.3	8.0
	KR	6.0	10.0	12.0	14.0	12.0	11.0	10.0	30.0
	KNN	10.0	6.0	10.0	10.0	10.0	10.0	10.0	25.0
	RF	6.0	10.0	10.0	10.0	11.0	11.0	11.0	24.0
	DNN-ℓ_2	5.6	9.3	11.2	13.1	11.2	10.2	9.2	24.6
	DNN-ℓ_1	6.9	10.2	10.2	10.2	10.2	10.2	10.2	**22.8**
(10,1)	EQ	6.0	10.0	12.0	14.0	12.0	11.0	10.0	30.0
	LML	8.0	10.0	12.0	14.0	16.0	18.0	20.0	53.0
	PSEO	8.2	13.6	16.0	18.4	16.0	15.0	14.0	56.2
	SEO-ℓ_1	5.8	9.5	11.5	13.5	11.5	10.5	9.5	26.8
	SEO-ℓ_2	5.8	9.5	11.5	13.5	11.5	10.5	9.5	26.8
	KR	6.0	10.0	12.0	14.0	12.0	11.0	10.0	30.0
	KNN	6.0	10.0	12.0	12.0	12.0	11.0	10.0	39.0
	RF	12.0	12.0	12.0	14.0	12.0	12.0	12.0	41.0
	DNN-ℓ_2	5.8	9.6	11.6	13.5	11.6	10.6	9.6	**27.2**
	DNN-ℓ_1	6.1	11.3	11.3	11.3	11.3	11.3	11.3	28.6
(20,1)	EQ	6.0	10.0	12.0	14.0	12.0	11.0	10.0	30.0
	LML	8.0	10.0	12.0	14.0	16.0	18.0	20.0	38.0
	PSEO	9.4	15.4	18.1	20.8	18.1	17.1	16.1	70.1
	SEO-ℓ_1	7.2	11.3	15.8	15.8	13.6	12.5	11.6	40.6
	SEO-ℓ_2	7.2	11.3	15.8	16.0	13.6	12.5	11.6	40.8
	KR	10.0	12.0	14.0	14.0	14.0	12.0	11.0	42.0
	KNN	14.0	14.0	14.0	14.0	14.0	14.0	14.0	53.0
	RF	14.0	14.0	14.0	14.0	14.0	14.0	14.0	53.0

is an example of how the EQ algorithm can fail when the historical data are volatile.

Consider the KNN algorithm. Since there are only two weeks of historical data, we opt to use all possible historical records without any validation and set $k = 14$. KNN gets the k historical records that are nearest to the new observation, each with a weight of $\frac{1}{k}$, and then chooses the point that weighted SAA selects. The demand of that point is the order quantity. So, as c_p/c_h increases, it selects larger values. However, the demands during the third week (the testing set) are close to the mean demand of the first two weeks (the training set); therefore, the increased order quantity chosen by KNN turns out to be too large. Similarly, in RF we select 2000 forests, and in KR we select $h = 0.5$ and use all data from the two weeks of the training set. Since both algorithms work with sorted demands, as c_p/c_h increases, they select larger demands from the training sets. Therefore, RF and KR also results in large cost values, for similar reasons as KNN.

Now consider the results of all versions of the SEO algorithm. For the case in which $c_h = c_p$ (which is not particularly realistic), SEO-ℓ_2 gets the best result; however, SEO-ℓ_1 does not perform well. Also, PSEO's output is approximately equal to the mean demand, which happens to be close to the week-3 demand values. This gives PSEO a cost of 2.5, which ties DNN-ℓ_2 for second place. For all other instances, however, the increased value of c_p/c_h results in an inflated order quantity and hence a larger cost.

Finally, both DNN-ℓ_1 and DNN-ℓ_2 outperform the LML algorithm by Rudin and Vahn [2013], because LML uses a linear kernel, while DNN uses both a linear and non-linear kernel. Also, there are only two features in this data set, so LML has some difficulty to learn the relationship between the inputs and output. Finally, the small quantity of historical data negatively affects the performance of LML.

This small example shows some conditions under which DNN outperforms the other

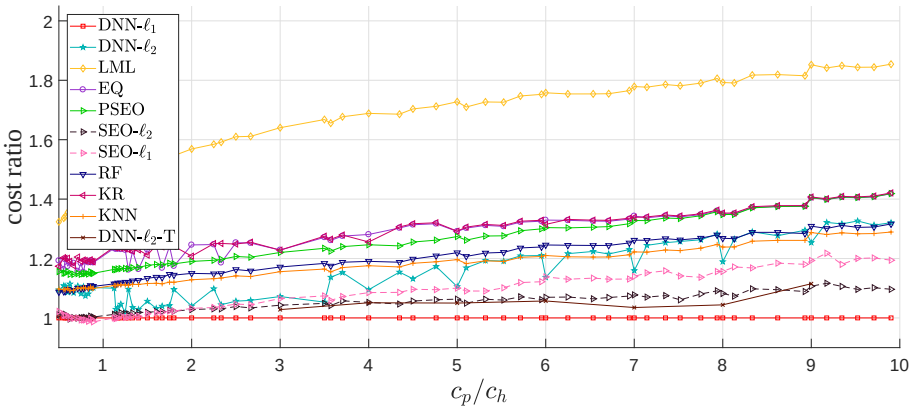


Figure 2.3: Ratio of each algorithm's cost to DNN-ℓ_1 cost on a real-world dataset.

three algorithms. In the next section we show that similar results hold even for a real-world dataset.

2.4.2 Real-World Dataset

We tested the seven algorithms on a real-world dataset consisting of basket data from a retailer in 1997 and 1998 from Pentaho [2008]. There are 13170 records for the demand of 24 different departments in each day and month, of which we use 75% for training and validation and the remainder for testing. The categorical data were transformed into their binary equivalents, resulting in 43 input features.

The results of each algorithm for 100 values of c_p and c_h are shown in Figure 2.3. In the figure, the vertical axis shows the normalized costs, i.e., the cost value of each algorithm divided by the corresponding DNN-ℓ_1 cost. The horizontal axis shows the ratio c_p/c_h for each instance. As before, most instances use $c_p \geq c_h$ to reflect real-world settings, though a handful of instances use $c_p < c_h$ to test this situation as well.

As shown in Figure 2.3, for this data set, the DNN-ℓ_1 and DNN-ℓ_2 algorithms both outperform the other three algorithms for every value of c_p/c_h. Among the three remaining algorithms, the results of SEO-ℓ_2 and SEO-ℓ_1, and then the KNN and RF algorithms, are

the closest to those of DNN. On average, their corresponding cost ratios are 1.04, 1.08, 1.15, and 1.16, whereas the ratios for EQ, LML, KR, and PSEO are 1.26, 1.53, 1.16, 1.26, and 1.23, respectively. The average cost ratio of DNN-ℓ_2 is 1.13. However, none of the other approaches are stable; their cost ratios increase with the ratio c_p/c_h.

DNN-ℓ_2 requires more tuning than DNN-ℓ_1, but the DNN-ℓ_2 curve in Figure 2.3 does not reflect this additional tuning. The need for additional tuning is suggested by the fact that DNN-ℓ_2's loss value increases as c_p or c_h increase, suggesting that it might need a smaller learning rate (to avoid big jumps) and a larger regularization coefficient λ (to strike the right balance between cost and over-fitting). Thus, tuning DNN-ℓ_2 properly would require a larger search space of the learning rate and λ, which would make the procedure harder and more time consuming. In our experiment, we did not expend this extra effort; instead, we used the same procedure and search space to tune the network for both DNN-ℓ_1 and DNN-ℓ_2, in order to compare them fairly.

Nevertheless, it is worth investigating how the performance of DNN-ℓ_2 could be improved if it is tuned more thoroughly. To that end, we selected integer values of $c_p/c_h = 3, \ldots, 9$, and for each value, we applied more computational power and tuned the parameters using a grid search. We fixed the network as $[43, 350, 100, 1]$, tested it with 702 different parameters, and selected the best test result among them. The grid search procedure is explained in detail in Appendix B. The corresponding result is labeled as DNN-ℓ_2-T in Figure 2.3. As the figure shows, this approach has better results than the original version of DNN-ℓ_2; however, DNN-ℓ_1 is still better.

The DNN algorithms execute more slowly than some of the other algorithms. For the basket dataset, the PSEO and EQ algorithms each execute in about 10 seconds. The DNN algorithm requires about 50 seconds (on a relatively large network, e.g., $[43, 90, 150, 56, 1]$)

Table 2.3: Summary of hyper-parameter (HP) tuning process for each method. Times reported are approximate training times for a single problem instance.

	# HP	# HP Values Tested	HP Values Tested	Approx. Avg. Training Time per HP (sec)	Approx. Total Training Time (sec)
SEO	–	–			10
EQ	–	–			10
LML	1	30	$2^h, h \in \{-20, \ldots, 10\}$	40	1200
KR	1	7	$\{10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 0.05, 0.1, 0.25\}$	15	105
KNN	1	6	$\{5, 10, 15, 50, 100, 150\}$	5	30
RF	1	5	$\{10, 20, 50, 100, 150\}$	4 (per tree)	1320
DNN	4	100	(see Section 2.3)	600	44,050

for each epoch of training, while the LML, KR, KNN, and RF algorithms require on average, respectively, about 40 seconds (per regularization value), 15 seconds (per bandwidth), 5 seconds (for a given k), and 4 seconds (per tree) for training for a given c_p and c_h. As the size of the search space for hyper-parameter tuning increases, so does the training time for DNN, LML, KR, RF, and KNN. For LML, we tested 30 different bandwidths—$2^h, h \in \{-20, \ldots, 10\}$—which resulted in 1200 seconds of training, on average. For KR, we tested bandwidth values of $10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 0.05, 0.1$, and 0.25, with a total time of 110 seconds on average. KNN needs to tune k, for which we tested six values—$5, 10, 15, 50, 100$, and 200—which took 30 seconds on average. Similarly, for RF we tested five forest sizes—$10, 20, 50, 100$, and 150—which resulted in 1320 seconds of training on average. For DNN-ℓ_1 DNN-ℓ_2, SEO-ℓ_1, and SEO-ℓ_2 we used the HyperBand algorithm to tune the network. We tested several different values of each of the hyper-parameters (as explained at the end of Section 2.3), resulting in a total of 881 epochs, which took 12.25 hours of training on average. The best network runs for 16 epochs, which took 600 seconds on average. Table 2.3 summarizes the hyper-parameters used during the tuning process for each method, and their approximate computation times. Note that the times reported in the table are for one instance of the basket dataset, i.e., one value of c_p/c_h.

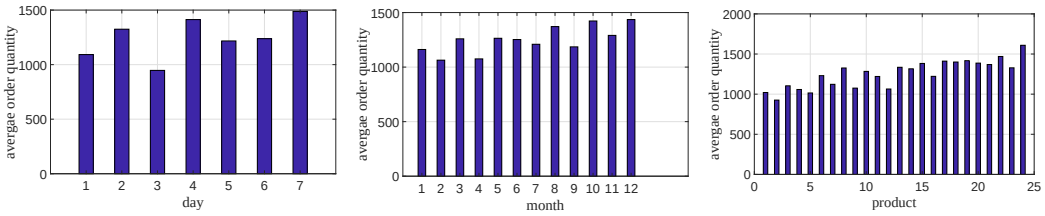
On the other hand, DNN, SEO, and LML algorithms execute in less than one second,

i.e., once the network is trained, the methods generate order quantities for new instances very quickly. In contrast, KR, KNN, and RF required approximately 15, 5, and $4t$ seconds, respectively, for inference, where t is the number of trees that is selected.

Since tuning the DNN hyper-parameters can be time-consuming, in Appendix C we propose a simple tuning-free network for the newsvendor problem.

Finally, we performed a small experiment to provide some intuition about which features have the most impact on the order quantity. In particular, we calculated the order quantity for each of the $7 \times 12 \times 24 = 2016$ possible combinations of the feature values, using the DNN model tuned for a uniform distribution with 100 clusters. For each individual feature value, we calculated the average order quantity; these are plotted in Figure 2.4. From the figure it is evident that—for this data set—the order quantity is affected most strongly by the product category, then by the day of the week, and then by the month of the year. The average order quantity ranges (max − min) for the product, day, and month are 682.9, 540.7, and 371.9, respectively.

This sort of approach could be used to analyze the results of the DNN algorithm for any set of categorical features. The results could be useful to managers attempting to decide whether to use a feature-based approach—including DNN or the other models discussed here—rather than treating the entire data set as a single cluster. For example, if the supply chain manager for the supermarket data set did not have access to product labels, a feature-based optimization approach would be less valuable, since the day and month features provide less differentiation in the order quantities; in this case, ignoring the features and treating the entire data set as a single cluster would result in less error than it would if product labels were available. Of course, these insights pertain only to this data set. We are not claiming that product is a stronger differentiator than month in general, but rather



(a) (b) (c)

Figure 2.4: The effect each feature on the order quantity for uniformly distributed data with 100 clusters.

illustrating how the DNN model can be used to generate such insights.

2.4.3 Randomly Generated Data

In this section we report on the results of an experiment using randomly generated data. This experiment allows us to test the methods on many more instances; however, the disadvantage is that these data are much cleaner than those typically encountered in real supply chains, i.e., they come from a single probability distribution with no noise. This should be kept in mind when interpreting these results. In short, the results in this section indicate that, when the data are non-noisy, all of the methods perform more or less similarly, with some exceptions. In all cases, DNN's performance is competitive with, if not better than, the other methods; and since it also performs better on messier data sets (e.g., the real-world data set in Section 2.4.2), we recommend its use in general. We now present a more detailed discussion of this experiment.

We conducted tests using five different probability distributions for the demand (normal, lognormal, exponential, uniform, and beta distributions). For each distribution, we generated 257,500 records. The parameters for the five demand distributions are given in Table 2.4; these parameters were selected so as to provide reasonable demand values. All demand values are rounded to the nearest integer. Each group of 257,500 records is divided into

Table 2.4: Demand distribution parameters for randomly generated data.

	Number of Clusters			
Distribution	1	10	100	200
Normal	$\mathcal{N}(50, 10)$	$\mathcal{N}(50i, 10i)$	$\mathcal{N}(50i, 5i)$	$\mathcal{N}(50i, 5i)$
Lognormal	$\ln\mathcal{N}(2, 0.5)$	$\ln\mathcal{N}(1 + 0.1(i + 1),$ $0.5 + 0.1(i + 1))$	$\ln\mathcal{N}(0.05(i + 1),$ $0.01(i + 1))$	$\ln\mathcal{N}(0.02(i + 1),$ $0.005(i + 1))$
Exponential	$\exp(10)$	$\exp(5 + 2(i + 1))$	$\exp(5 + 0.2(i + 1)$	$\exp(5 + 0.05(i + 1)$
Beta	$20\mathcal{B}(1, 1)$	$100\mathcal{B}(0.6(i + 1),$ $0.6(i + 1))$	$100\mathcal{B}(0.1(i + 1),$ $0.1(i + 1))$	$100\mathcal{B}(0.07(i + 1),$ $0.07(i + 1))$
Uniform	$\mathcal{U}(1, 21)$	$\mathcal{U}(5(i + 1,$ $15 + 5(i + 1))$	$\mathcal{U}((i + 1),$ $15 + (i + 1))$	$\mathcal{U}(0.5(i + 1),$ $15 + 0.5(i + 1))$

training and validation (10,000 records) and testing (99 sets, each 2,500 records) sets.

In each of the distributions, the data were categorized into clusters, each representing a given combination of features. Like the real-world dataset, we considered three features: the day of the week, month of the year, and department. We varied the number of clusters (i.e., the number of possible combinations of the values of the features) from 1 to 200 while keeping the total number of records fixed at 257,500; thus, having more clusters is the same as having fewer records per cluster. In this experiment, an "instance" refers to a given combination of demand distribution (normal, exponential, ...) and number of clusters (1, 10, ...).

Each problem was solved for $c_p/c_h = 5$ using all seven algorithms (including both loss functions for DNN), without assuming any knowledge of the demand distribution. We conducted additional tests using additional c_p/c_h ratios; the results and conclusions were similar, so they are omitted here in the interest of conciseness.

In part, this experiment is designed to model the situation in which the decision maker does not know the true demand distribution. To that end, our implementations of the SEO and PSEO algorithm assumes the demands come from a normal distribution (regardless of the true distribution for the dataset being tested), since this distribution is used frequently as

the default distribution in practice. The other algorithms (DNN, LML, EQ, KNN, KR, and RF) do not assume any probability distribution. Additionally, since we know the underlying demand distributions, we also calculated and reported the optimal solution in each case. The average times required to tune or execute each of the algorithms, per instance, are similar to those in Table 2.3.

Figure 2.5 plots the average cost ratio (cost divided by optimal cost) for the five distributions. Each point on a given plot represents the average cost (over 99 testing sets) for one instance. Figure 2.6 contains magnified versions of the plots in Figure 2.5 for three of the distributions. From the plots, we can draw the following conclusions:

- If there is only a single cluster, then all seven algorithms produce nearly the same results. This case is essentially a classical newsvendor problem with 7,500 data observations, for which all algorithms do a good job of providing the order quantity in the test sets.

- As the number of clusters increases, i.e., the number of training samples in each cluster decreases, the methods begin to differentiate somewhat. In particular:

- DNN-ℓ_1, SEO-ℓ_2, PSEO, EQ, KR, and KNN perform the best and have roughly equal performance.

- SEO performs well when the demands are normally distributed but less well otherwise. This is because one has to assume a demand distribution in order to use SEO, and we assumed normal. If the demands happen to come from a normal distribution, therefore, SEO works well. In practice, however, the demand distribution is usually unknown and often non-normal.

- SEO-ℓ_2 and EQ perform relatively well in general in this experiment because, when the data are non-noisy, it is easier to estimate a quantile. However, for both the small data set (Section 2.4.1) and the real-world data set (Section 2.4.2), which are noisier, SEO-ℓ_2 EQ do not perform as well as in the simulated data.

- The performance of DNN-ℓ_2 is quite good *except* in the case of normal demands with 100 or 200 clusters. In these cases, the method would benefit from further tuning (similar to the additional tuning that we did for the basket data set in Section 2.4.2).

- LML and RF are nearly always worse than the other methods because there is not enough data for them to learn the distribution well. (As a result, we have omitted them from Figure 2.6.)

To confirm these findings statistically, Figures 2.7 and 2.8 plot 95% confidence intervals for each algorithm for normally and uniformly distributed demands (respectively). The confidence intervals are calculated using the mean and standard error of the cost ratio over the 99 test data sets. When two confidence intervals are non-overlapping, we can conclude that the performance of the two corresponding methods is statistically different. If a given method is excluded from a plot, it means that the method is much worse than the methods that are plotted. From these figures, we can draw the following conclusions:

- DNN-ℓ_1 is statistically better than all other methods for some cases (e.g., uniform demands with 100 clusters); is in statistical second place to PSEO for normal demands with 200 clusters and to DNN-ℓ_2 for uniform demands with 100 and 200 clusters; and is tied for first place in all other cases.

- PSEO is statistically better than all other methods for normal demands with 200 clusters and statistically worse than all other methods for uniform demands with any

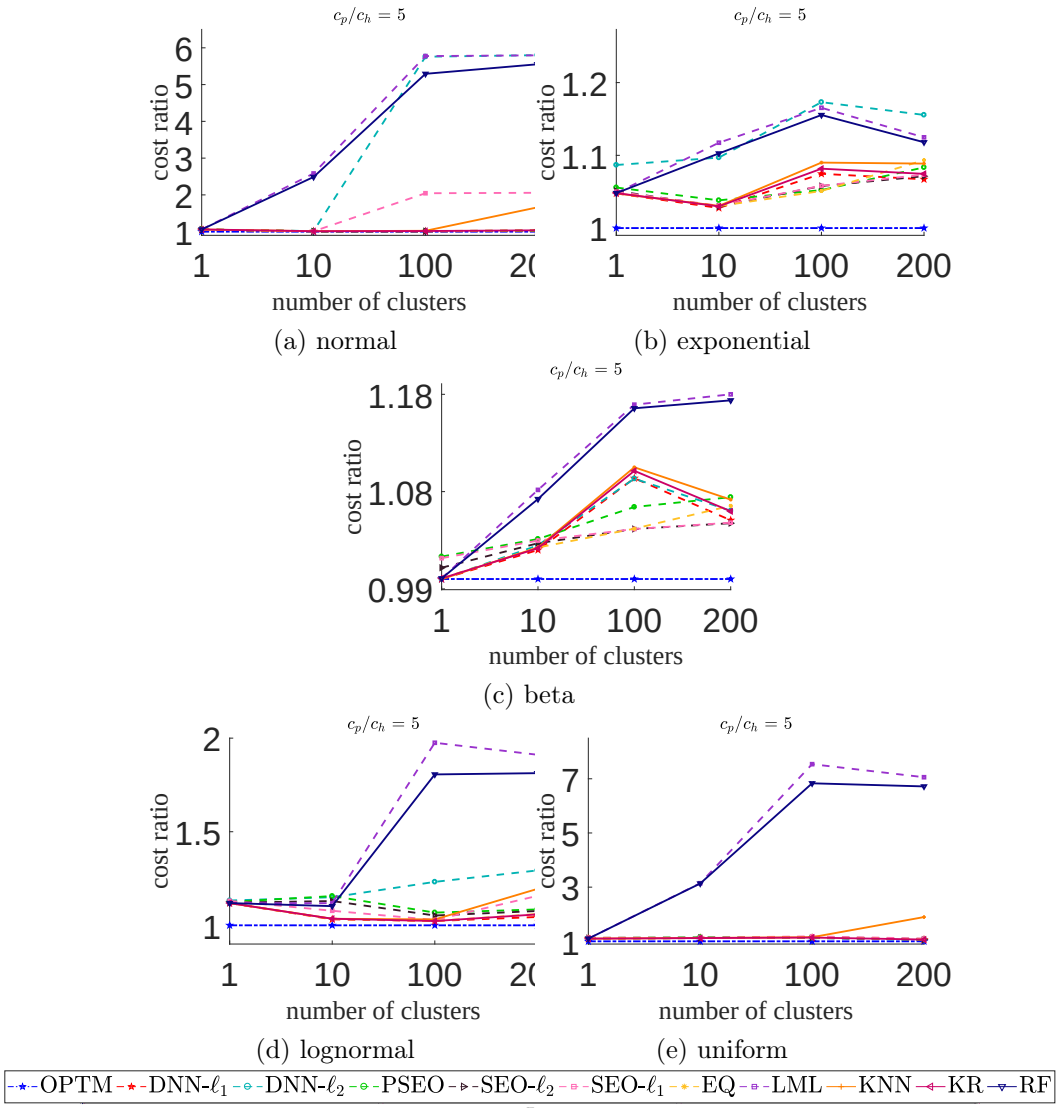


Figure 2.5: Ratio of each algorithm's cost to optimal cost on randomly generated data from each distribution.

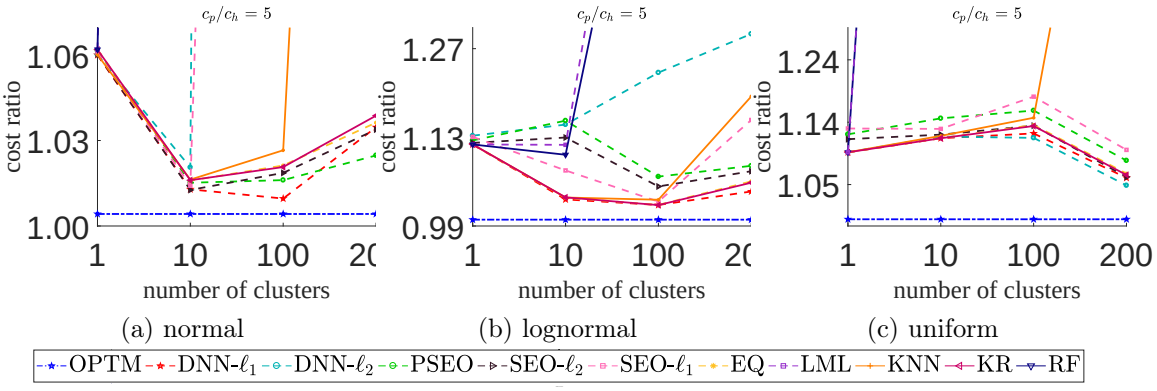
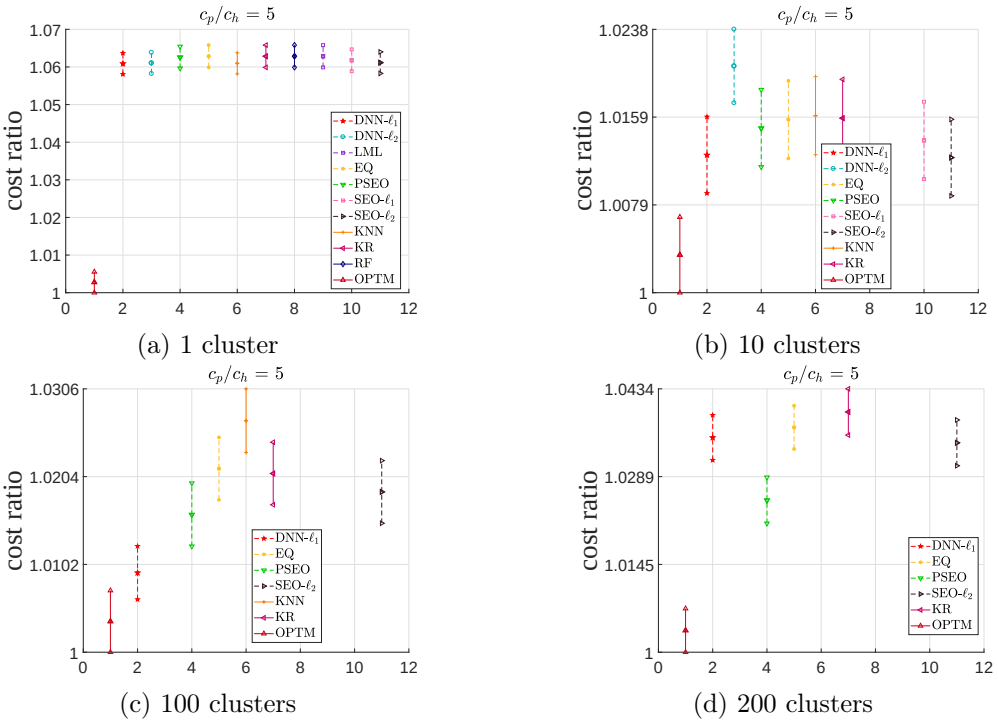


Figure 2.6: Magnified results for normal, lognormal, and uniform distributions.



(a) 1 cluster (b) 10 clusters

(c) 100 clusters (d) 200 clusters

Figure 2.7: Confidence intervals for each algorithm for normally distributed demands.

number of clusters. It is tied with other methods for most other instances.

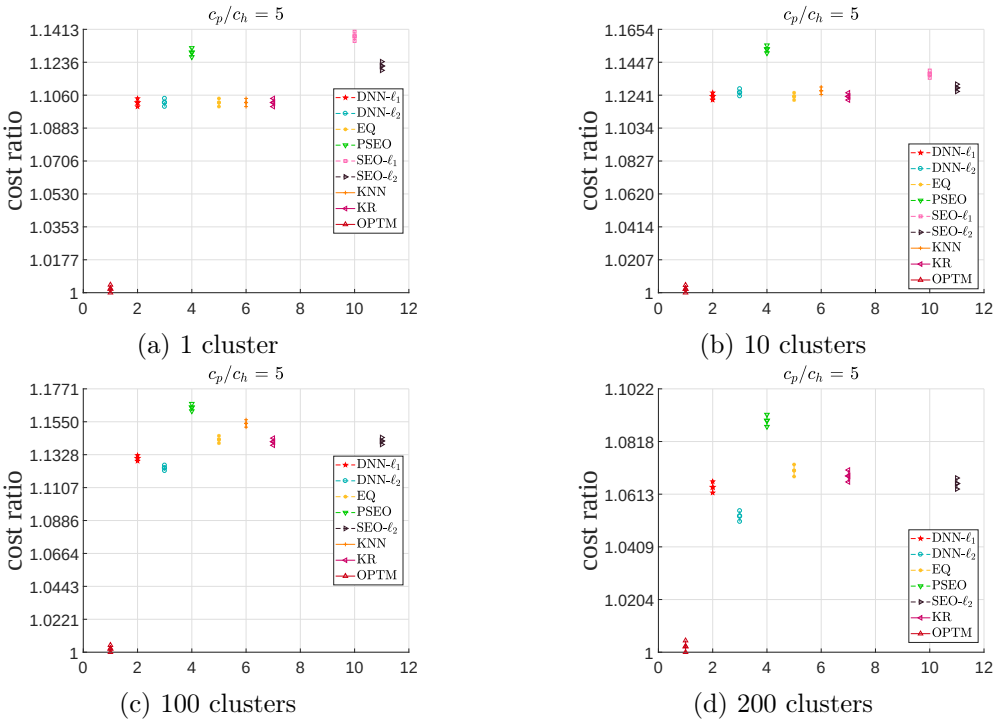
- SEO-ℓ_2 in most cases is in a statistical tie with DNN-ℓ_1 except for uniform demands with 10 and 100 clusters, and normal demands with 100 clusters.

- DNN-ℓ_2, EQ, KNN, and KR are, in most cases, in a statistical tie.

- LML, SEO-ℓ_1, and RF are statistically worse than all other methods, except in the case of normal demands with 1 cluster.

- In nearly every instance, no method obtains solutions that are statistically equal to the optimal solution. The exception is normal demands with 100 clusters, for which DNN-ℓ_1 is statistically tied with the optimal solution.

Suppose we take a naive approach toward the MFNV problem and ignore the data features, optimizing the inventory level as though there were only a single cluster. How



(a) 1 cluster (b) 10 clusters

(c) 100 clusters (d) 200 clusters

Figure 2.8: Confidence intervals for each algorithm for uniformly distributed demands.

significant an error is this? To answer this question, we solved the problem using DNN-ℓ_1, grouping all of the data into a single cluster. (Note that this data set is different from the 1-cluster data sets discussed above. The data sets above assume there *is* only a single cluster, i.e., all demand records have identical feature values, whereas the data set here has multiple sets of feature values, but we are ignoring them to emulate the naive approach.) Figure 2.9 plots the ratio between the cost of the resulting solution and the cost of the DNN-ℓ_1 solution that accounts for the clusters, for the five probability distributions and for data sets with 10, 100, and 200 clusters. Clearly, the error resulting from this naive approach can be significant: They range from 5.6% (for the exponential distribution with 200 clusters) to 677.9% (for the uniform distribution with 100 clusters). In general these errors will change with the probability distributions and their parameters, but it is clear that it is important to consider clusters when faced with featured data, and costly to ignore

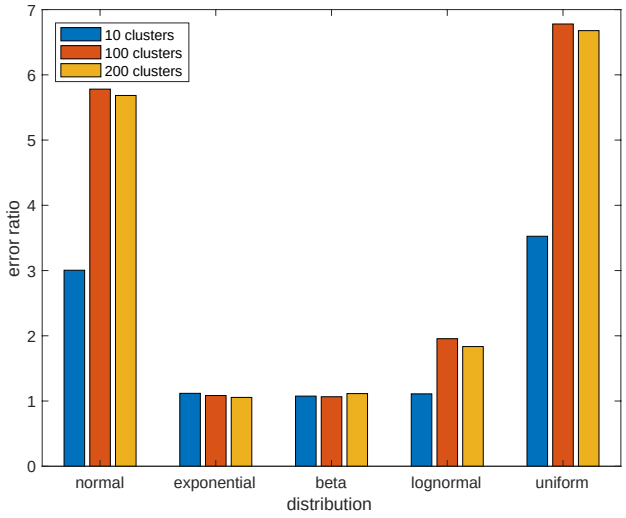


Figure 2.9: Error ratio from ignoring clusters when solving MFNV.

them.

2.4.4 Numerical Results: Summary

Our recommendations for which method to use are as follows. If the data set is noisy, like most real-world data sets, our experiments show that DNN is the most reliable algorithm, with the caveat that careful hyperparameter tuning is required. If the data are non-noisy (they come from a single probability distribution) and the number of historical samples is small (say, fewer than 10 records per combination of features), DNN tends to outperform the other methods. As the number of historical records begin to increase, either EQ, SEO, DNN, KR, RF, or KNN is a reasonable choice. Finally, if there are a large number of non-noisy historical demand records for each combination of features (say, at least 10,000), then the algorithms all work roughly equally well, and it may be best to choose EQ or SEO, since they do not need any hyperparameter tuning.

2.5 Extension to (r, Q) Policy

In this section, we extend our DNN approach to optimize the parameters of an (r, Q) inventory policy, in order to demonstrate that the method can be adapted to other inventory problems, and especially to problems that cannot be solved simply by estimating the quantile of a probability distribution. Consider a continuous-review inventory optimization problem with stochastic demand, such that the mean demand per unit time is λ. Placing an order incurs a fixed cost K, and the order arrives after a deterministic lead time of $L \geq 0$ time units. Unmet demand is backordered. We assume the firm follows an (r, Q) inventory policy: Whenever the inventory position falls to r, an order of size Q is placed. The aim of the optimization problem is to determine r and Q.

If we know the true demand distribution, the optimal r and Q can be obtained by solving a convex optimization problem; see Hadley [1963] or Zheng [1992]. However, heuristic approaches are commonly used to obtain approximate values for r and Q; for a discussion of these, see Snyder and Shen [2019]. We use the so-called expected-inventory level (EIL) approximation, which is arguably the most common approximation for the (r, Q) optimization problem. The EIL approximates the expected cost function as

$$g(r, Q) = c_h \left(r - \lambda L + \frac{Q}{2} \right) + \frac{K\lambda}{Q} + \frac{c_p \lambda n(r)}{Q}, \tag{2.16}$$

where

$$n(r) = \int_r^\infty (d - r) f(d) dd$$

and $f(d)$ is the demand distribution. The cost function (2.16) can be optimized through an iterative algorithm proposed by Hadley [1963], again assuming that the demand distribution

is known.

Of course, in practice, the demand distribution is often not known, which is where DNN becomes a useful approach. In order to use DNN to obtain the policy parameters, we propose a DNN network similar to that used for the newsvendor problem, except that it has two outputs, r and Q. We use the cost function (2.16) as the loss function for the DNN, and in place of $n(r)$ we use the unbiased estimator $\frac{1}{m}\sum_{i=1}^{m}(d_i - r_i)^+$. In addition, in order to avoid negative values for r and Q, we use r^+ and Q^+ in the DNN loss function, and also add a penalty for negative values of r and Q into the DNN loss function:

$$l(r,Q) = c_h\left(r^+ - \lambda L + \frac{Q^+}{2}\right) + \frac{K\lambda}{Q^+} + \frac{c_p\lambda n\left(r^+\right)}{Q^+} + \eta_Q Q^- + \eta_r r^-,$$

where η_r and η_Q are the penalty coefficients for negative r and Q, respectively.

Additionally, we use a KNN approach as machine learning based benchmark. We use SAA for approximating $n(r)$, i.e.

$$n(r) = \frac{1}{k}\sum_{i\in\mathbb{N}_x}(d_i - x)^+, \tag{2.17}$$

for a given feature value x. Then, to obtain (r,Q) we modify the EIL algorithm as it is:

2.5.1 Numerical Experiments

In order to see the effectiveness of the proposed algorithm for the (r,Q) optimization problem, we tested both algorithms on a problem with $K = 20$, $\lambda = 1200$, $c_p = 10$, $c_h = 1$, and $L = \mu/\lambda$, where μ is the annual demand of a given product. We used the iterative algorithm by Hadley [1963] to obtain the optimal r and Q that minimize (2.16). (We will refer to this as the *EIL algorithm.*) Since the algorithm needs the demand distribution, similar to the

KNN-SAA

procedure GET (r, Q)
  Set $Q = \sqrt{\frac{2K\lambda}{h}}$
  while `True` **do**
    Get r_{new} by optimizing $g(r, Q)$ using current Q.
    Set $Q = \sqrt{\frac{2\lambda\left[K+\frac{1}{k}p\sum_{i\in\mathbb{N}_x}(d_i-x)^+\right]}{h}}$
    if $|Q_{new} - Q| < \epsilon$ **then**
      if $|r_{new} - r| < \epsilon$ **then**
        Break
      end if
    end if
    $Q = Q_{new}$
  end while
end procedure

approach in Section 2.4.3, we fit a normal distribution to each cluster and use it to obtain (r, Q) for the corresponding cluster.

When testing the DNN algorithm on this problem, we performed the same level of hyper-parameter tuning that we did on the newsvendor problem. All of the neural networks used the `Relu` activation function, where $\texttt{Relu}(x) = x^+$. We used the Adam optimizer [Kingma and Ba, 2014] to optimize the weights of the network with random learning rate, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 1e-8$, a batch size of 128, and exponential decay with rate 0.96.

In what follows, we demonstrate the results of three algorithms on six datasets that we used when testing the newsvendor problem: the basket data set, which is presented in Section 2.5.1.1, and the five randomly generated datasets, presented in Section 2.5.1.2.

2.5.1.1 Basket Dataset

We obtained (r, Q) values using both algorithms. The solution found by the EIL algorithm incurs a cost of 1,650,214, KNN results in 1,392,643, while that obtained by DNN has a cost of 1,322,568, 19.9% and 5.0% better than EIL and KNN. At first this may seem surprising, since the EIL algorithm is an exact algorithm to optimize the cost function (2.16) (though of

course (2.16) is itself an approximation of the exact cost function). However, recall that the basket dataset is noisy and contains few historical observations (between 1 and 9) per cluster, but the EIL algorithm assumes the demands are normally distributed. This assumption is inaccurate for the basket dataset. Also, KNN works well when there is a large enough number of neighbors for each sample. On the other hand, DNN considers the feature values and in three epochs optimizes the weights of the network, and in doing so is able to learn better (r, Q) values to minimize the objective.

2.5.1.2 Randomly Generated Data

In order to further explore the performance of both algorithms, we tested their performance on the randomly generated datasets in Section 2.4.3. Just as in the newsvendor problem, we assume we do not know the demand distribution and instead approximate a normal distribution in each cluster to obtain the solution using EIL. The results of all demand distributions are shown in Figure 2.10, in which the cost of KNN, DNN, and EIL are divided by the corresponding cost of the EIL algorithm. As shown in the figure, when the data are generated from a normal distribution, EIL finds smaller costs than DNN, though the DNN solution is close, with around a 1.1% gap, on average, for four clusters. DNN provides a smaller cost for the other distributions, such that the average cost across all distributions is 1.6% smaller than EIL. On the other hand, KNN works well in the simulated dataset, specially in the uniform and lognormal demand distributions, and on average it obtains 3.8% smaller cost than EIL. The reason is that there are at least 37 neighbors for each test record so that KNN can get a reasonable approximation for the demand.

Let us more closely examine one instance, the normally distributed dataset, for which the EIL solution is optimal. When there is only one cluster, the optimal solution from EIL is

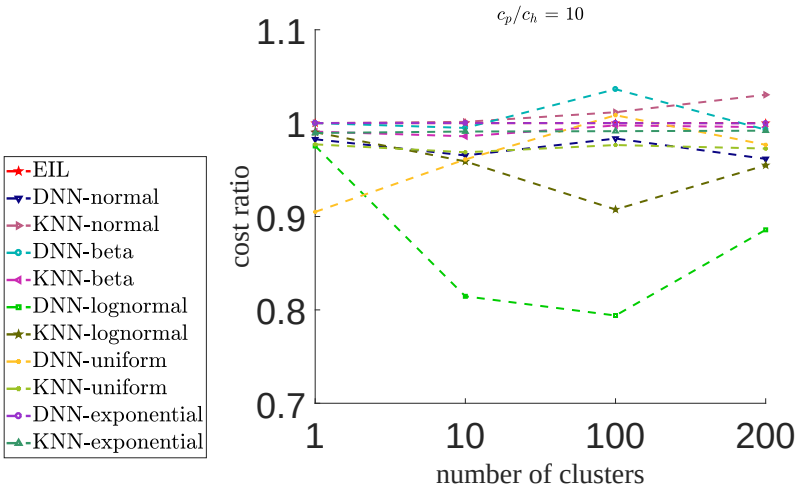


Figure 2.10: The results for randomly generated datasets for the (r, Q) model.

Table 2.5: EIL and DNN values of (r, Q) for the normally distributed dataset with 10 clusters.

Cluster	1	2	3	4	5	6	7	8	9	10
EIL r	70.41	141.47	212.07	277.74	350.43	416.84	485.73	555.94	626.89	697.91
DNN r	69.76	138.83	209.16	278.86	354.69	424.88	480.12	556.31	640.10	700.25
EIL Q	222.73	226.57	230.24	233.39	238.05	241.86	245.16	250.17	253.13	259.53
DNN Q	211.19	215.68	224.30	230.53	233.85	246.88	247.78	249.10	250.44	262.03

$(r, Q) = (70.00, 222.50)$, whereas DNN obtains $(r, Q) = (70.24, 222.70)$, which is quite close. Similarly, when there are 10 clusters, the DNN (r, Q) is quite close to the optimal solutions, as shown in Table 2.5. As a result, the costs of the solutions obtained by the two algorithms are almost equal. Similar results also emerge from the instances with 100 and 200 clusters.

To summarize, if the true distribution is available, our DNN method and the classical EIL approach work almost equally well. However, EIL's performance deteriorates when the true demand distribution is not known, even if there is a relatively large amount of historical data. In contrast, DNN works well when the true demand distribution is unknown, even if the historical dataset is small and/or noisy.

2.6 Conclusion

In this chapter, we consider the multi-feature newsvendor (MFNV) problem. If the probability distribution of the demands is known for every possible combination of the data features, there is an exact solution for this problem. However, approximating a probability distribution is not easy and produces errors; therefore, the solution of the newsvendor problem also may be not optimal. Moreover, other approaches from the literature might not work well when the historical data are scant and/or volatile.

To address this issue, we propose an algorithm based on deep learning to solve the MFNV. The algorithm does not require knowledge of the demand probability distribution and uses only historical data. Furthermore, it integrates parameter estimation and inventory optimization, rather than solving them separately. Extensive numerical experiments on real-world and random data demonstrate the conditions under which our algorithm works well compared to the algorithms in the literature. The results suggest that when the volatility of the demand is high, which is common in real-world datasets, deep learning works very well. When the data can be represented by a well-defined probability distribution, in the presence of enough training data, a number of approaches, including DNN, have roughly equivalent performance.

Furthermore, we extend our DNN approach to the (r, Q) inventory optimization problem, to demonstrate that our approach is applicable in more general settings, especially those that cannot be solved by estimating a quantile. Our computational results show that the DNN approach works well when the historical data are noisy and/or sparse, and that it often outperforms the "exact" algorithm when the true demand distribution is unknown (since the exact algorithm must make an assumption about the distribution).

Motivated by the results of deep learning on both newsvendor and (r, Q) problems, we suggest that this idea can be extended to other supply chain problems. For example, since general multi-echelon inventory optimization problems are very difficult, deep learning may be a good candidate for solving these problems. Another direction for future work could be applying other machine learning algorithms to exploit the available data in the newsvendor problem.

Chapter 3

Stock-Out Prediction in Multi-Echelon Networks

In multi-echelon inventory systems, the performance of a given node is affected by events that occur at many other nodes and in many other time periods. For example, a supply disruption upstream will have an effect on downstream, customer-facing nodes several periods later as the disruption "cascades" through the system. There is very little research on stock-out prediction in single-echelon systems and (to the best of our knowledge) none on multi-echelon systems. However, in the real world, it is clear that there is significant interest in techniques for this sort of stock-out prediction. Therefore, our research aims to fill this gap by using deep neural networks (DNN) to predict stock-outs in multi-echelon supply chains. We test our approach on several types of multi-echelon networks and compare its performance to that of several naive approaches. We find that our approach outperforms the other algorithms, and we suggest conditions under which it is most reliable. Finally, we extend the algorithms to handle threshold prediction, multi-period-ahead prediction, and

multi-item prediction.

3.1 Introduction

A multi-echelon supply chain is a network of nodes that aims to provide a product or service to its customers. Each network consists of production and assembly lines, warehouses, transportation systems, retail processes, etc., and each of them is connected at least to one other node. The most downstream nodes of the network face the customers, which usually present an external stochastic demand. The most upstream nodes interact with third-party vendors, which offer an unlimited source of raw materials and goods. An example of a multi-echelon network is shown in Figure 3.1, which depicts a distribution network, e.g, a retail supply chain.

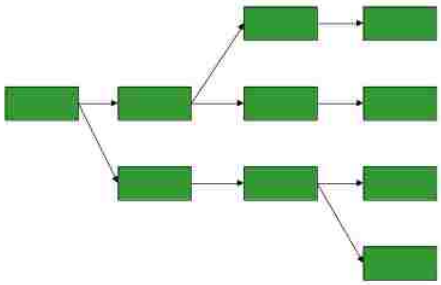


Figure 3.1: A multi-echelon network with 10 nodes

The supply chain manager's goal is to find a compromise between the profit and service level (the fraction of the customer's orders that are satisfied on time) to its customers. For example, a retail network may decide to change the number of retail stores to increase its service availability and create more sales, which also results in a higher cost for the system. In this case, the relevant decisions are how many, where, and when they should be opened/closed to maximize the profit. Facility location and network design are the common mathematical programming problems to provide the optimal decision in those questions.

Similarly, the problems in production and inventory systems are where, when, how, and how much to produce or order of which item. Scheduling and capacity management are common problems in this area. Also, distribution systems must decide when, where, how, and how much of which item should be moved. The transportation problem is the most famous problem that answers these questions. In well-run companies, there are multiple systems that optimize those problems to provide the best possible balance between service level and profit. In this chapter, we focus on inventory management systems to provide an algorithm that answers some of the questions in an environment with stochastic demand.

Balancing between the service level and profit in an inventory system is equivalent to balancing the stock-out level and holding safety stock. (For simplicity we ignore order cost.) Stock-outs are expensive and common in supply chains. For example, distribution systems face $6\% - 10\%$ stock-outs for non-promoted items and $18\% - 24\%$ for promoted items [Gartner, 2011]. Stock-outs result in significant lost revenue for the supply chain. When a company faces a stock-out, roughly 70% of customers do not wait for inventory to be replenished, but instead, purchase the items from a competitor [Bharadwaj et al., 2002]. Thus, in order to not lose customers and maximize profit, companies should have an inventory management system to provide high service level at a small cost.

Supply chains have different tools to balance between service level and cost. Classical inventory models usually solve an optimization problem to balance between holding and stock-out costs, or to minimize holding cost subject to a constraint on service levels. Our models take a similar approach, except that our aim is to predict stock-outs for a given inventory policy rather than to optimize inventory. However, we note that the literature also contains discussions of alternate approaches to modeling and managing service levels in a supply chain. For example, Gruson et al. [2017] considers both backorders (the unmet

demand in the current period) and backlogs (all pending unmet demands) and proposes different service levels for them. Rather than considering only the total cost, Lu et al. [2018] considers the stock-out risk under different criteria such as time, percentage, frequency, and volume. They minimize the sum of holding, stock-out, and production costs while maintaining the stock-out rate within a given risk tolerance. Liao et al. [2014] provide a model to manage the situation when a stock-out happens. They propose inventory policies to choose the order quantity by either transshipments or emergency orders, to satisfy a stocked-out order. For a review of papers on lateral transshipments, see Paterson et al. [2011].

One category of models for multi-echelon inventory optimization is called the Stochastic Service Model (SSM) approach, which considers stochastic demand and stochastic lead times due to upstream stockouts. The optimal base-stock level can be found for serial systems without fixed costs by solving a sequence of single-variable convex problems [Clark and Scarf, 1960]. Similarly, by converting an assembly system (in which each node has at most one successor) to an equivalent serial system, the optimal solution can be achieved [Rosling, 1989]. For more general network topologies, no efficient algorithm exists for finding optimal base-stock levels, and in some cases the form of the optimal inventory policy is not even known [Zipkin, 2000].

Another approach for dealing with multi-echelon problems is the Guaranteed Service Model (GSM) approach. GSM assumes the demand is bounded above, or equivalently the excess demand can be satisfied from outside of the system, e.g., by a third party vendor. It assumes a Committed Service Time (CST) for each node, which is the latest time that the node will satisfy the demand of its successor nodes. The GSM model minimizes the expected holding cost using the CSTs as its decision variables, but this is equivalent to

optimizing the base-stock level of each node. This approach can handle more general supply chain topologies, typically using either dynamic programming [Graves, 1988, Graves and Willems, 2000] or MIP techniques [Magnanti et al., 2006].

For a review of GSM and SSM Models see Eruguz et al. [2016], Simchi-Levi and Zhao [2011], and Snyder and Shen [2019].

The sense among (at least some) supply chain practitioners is that the current set of inventory optimization models are sufficient to optimize most systems as they function normally. What keeps these practitioners up at night is the deviations from "normal" that occur on a daily basis and that pull the system away from its steady state. In other words, there is less need for new inventory optimization models and more need for tools that can help when the real system deviates from the practitioners' original assumptions.

Our algorithm takes a snapshot of the supply chain at a given point in time and makes predictions about how individual components of the supply chain will perform, i.e., whether they will face stock-outs in the near future. We assume an SSM-type system, i.e., a system in which demands follow a known probability distribution, and stages within the supply chain may experience stock-outs, thus generating stochastic lead times to their downstream stages. The stages may follow any arbitrary inventory policy, e.g., base-stock or (s, S). Classical inventory theory can provide long-term statistics about stock-out probabilities and levels (see, e.g., Snyder and Shen [2019], Zipkin [2000]), at least for certain network topologies and inventory policies. However, this theory does not make predictions about specific points in time at which a stock-out may occur. Since stock-outs are expensive, such predictions can be very valuable to companies so that they may take measures to prevent or mitigate impending stock-outs.

Note that systems whose base-stock levels were optimized using the GSM approach may

also face stock-outs, even though the GSM model itself assumes they do not. The GSM approach assumes a bound on the demand value; when the real-world demand exceeds that bound, it may not be possible or desirable to satisfy the demand externally, as the GSM model assumes. Therefore, stock-outs may occur in these systems, and stock-out prediction can be useful for users of both SSM and GSM approaches.

In a single-node network, one can obtain the stock-out probability and make stock-out predictions if the probability distribution of the demand is known (see Appendix D). However, to the best of our knowledge, there are no algorithms to provide stock-out predictions in multi-echelon networks. To address this need, in this chapter, we propose an algorithm to provide stock-out predictions for each node of a multi-echelon network, which works for any network topology (as long as it contains no directed cycles) and any inventory policy.

The remainder of chapter is organized as follows. In Section 3.2, we introduce our algorithm. Section 3.3 describes five naive algorithms to predict stock-outs. To demonstrate the efficiency of the proposed algorithm in terms of solution quality, we compare our results with the best naive algorithms in Section 3.4. Finally, Section 3.5 concludes the chapter and proposes future studies.

3.2 Stock-out Prediction Algorithm

We develop an approach to provide stock-out predictions for multi-echelon networks with available data features. Our algorithm is based on deep learning, or deep neural networks (DNN). DNN is a non-parametric machine learning algorithm, meaning that it does not make strong assumptions about the functional relationship between the input and output variables. In the area of supply chain, DNN has been applied to demand prediction [Efendigil

et al., 2009, Vieira, 2015, Ko et al., 2010] and quantile regression [Taylor, 2000, Cannon, 2011, Xu et al., 2016]. We successfully applied it to the newsvendor problem with data features in chapter 2, and has also been applied to credit scoring [Malhotra and Malhotra, 2003], predicting the functional status of patients in organ transplant operations [Misiunas et al., 2016], and stock index forecasting [Wang et al., 2012]. For time series prediction, Crone and Kourentzes [2010] propose using filter and wrapper approaches to improve the neural network results, where the patterns are obtained by classical forecasting techniques, like auto-regressive models. The basics of deep learning are available in Goodfellow et al. [2016].

Consider a multi-echelon supply chain network with n nodes, with arbitrary topology. For each node of the network, we know the history of the inventory level (IL), i.e., the on-hand inventory minus backorders, and of the inventory-in-transit (IT), i.e., the items that have been shipped to the node but have not yet arrived. The values of these quantities at node j in period i are given by IL_i^j and IT_i^j, respectively, and the vectors of all IL and IT values at time i are given by IL_i and IT_i, respectively. In addition, we know the stock-out status for the node, given as a `True` or `False` Boolean, where `True` indicates that the node experienced a stock-out. (We use 1 and 0 interchangeably with `True` and `False`.) The historical stock-out information is not used to make predictions at time t but is used to train the model. The demand distribution can be known or unknown; in either case, we assume historical demand information is available. The goal is to provide a stock-out prediction for each node of the network for the next period.

The available information that can be provided as input to the DNN algorithm includes the values of the p available features (e.g., day of week, month of year, weather information), along with the historical observations of IL and IT at each node. Therefore, the available

information for node j at time t can be written as:

$$[f_t^1, \ldots, f_t^p, [IL_i^j, IT_i^j]_{i=1}^t], \tag{3.1}$$

where $f_t^1, \ldots, f_t^p$ denotes the value of the p features at time t.

However, DNN algorithms are designed for inputs whose size is fixed; in contrast, the vector in (3.1) changes size at every time step. Therefore, we only consider historical information from the k most recent periods instead of the full history. Although this omits some potentially useful information from the network, it unifies and reduces the input size, which has computational advantages, and selecting a large enough k provides a good level of information about the system. Additionally, by not keeping all historical data, the effect of any outlier observations will be ignored after k periods. Therefore, the input of the DNN is:

$$[f_t^1, \ldots, f_t^p, [IL_i, IT_i]_{i=t-k+1}^t]. \tag{3.2}$$

The output of the DNN is the stock-out prediction for time $t+1$, for each node of the supply chain network, denoted $y_t = [y_t^1, \ldots, y_t^n]$, a vector of length n. Each of the y_t^j, $j = 1, \cdots, n$, equals 1 if the node in period t is predicted to have a stock-out and 0 otherwise.

A DNN is a network of nodes, beginning with an input layer (representing the inputs, i.e., (3.2)), ending with an output layer (representing the y_t vector), and one or more layers in between. Each node uses a mathematical function, called an activation function, to transform the inputs it receives into outputs that it sends to the next layer, with the ultimate goal of approximating the relationship between the overall inputs and outputs. In a fully connected network, each node of each layer is connected to each node of the next layer through some coefficients, called weights, which are initialized randomly. "Training" the

network consists of determining good values for those weights, typically using nonlinear optimization methods. (A more thorough explanation of DNN is outside the scope of this dissertation; see, e.g., Goodfellow et al. [2016].)

A loss function is used to evaluate the quality of a given set of weights. The loss function measures the distance between the predicted values and the known values of the outputs. We consider the following loss functions, which are commonly used for binary outputs such as ours:

- Hinge loss function
- Euclidean loss function
- Soft-max loss function

The hinge and Euclidean loss functions are reviewed in Appendix F. The soft-max loss function uses the soft-max function, which is a generalization of logistic regression (also reviewed in Appendix F) and is given by

$$P(z^u) = \frac{e^{z^u}}{\sum_{v=1}^{U} e^{z^v}}; \quad \forall u = 1, \ldots, U, \tag{3.3}$$

where U is the number of possible categories (in our case, $U = 2$), $z^u = \sum_{i=1}^{M_{L-1}} a_i^{L-1} w_{i,u}$, L is the number of layers in the DNN network, a_i^{L-1} is the activation value of node i in layer $L-1$, $w_{i,u}$ is the weight between node i in layer $L-1$ and node u in layer L, and M_{L-1} represents the number of nodes in layer $L-1$. Note that $P(z^u)$ is the probability of observing the uth category when we have observed input vector $[f_t^1, \ldots, f_t^p, [IL_i, IT_i]_{i=t-k+1}^t]$. Once we have these probabilities, we can calculate the loss value. Then the soft-max loss function

is given by

$$E = -\frac{1}{M}\sum_{i=1}^{M}\sum_{u=1}^{U}\mathbb{I}\{y_i = u-1\}\log\frac{e^{z_i^u}}{\sum_{v=1}^{U}e^{z_i^v}}, \tag{3.4}$$

where M is the total number of training samples, $\mathbb{I}(\cdot)$ is the indicator function, and E is the loss function value, which evaluates the quality of a given classification (i.e., prediction). In essence, the loss function (3.4) penalizes the corresponding prediction to y_i by the logarithm of the probability (3.3). So, the model tries to maximize the probability of selecting a correct label to minimize the loss value.

The hinge and soft-max function provide a probability distribution over U possible classes; we then take the `argmax` over them to choose the predicted class. In our case there are $U = 2$ classes, i.e., `True` and `False` values, as required in the prediction procedure. On the other hand, the Euclidean function provides a continuous value, which must be changed to a binary output. In our case, we round Euclidean loss function values to their nearest value, either 0 or 1.

Choosing weights for the neural network involves solving a nonlinear optimization problem whose objective function is the loss function and whose decision variables are the network weights. Therefore, we need gradients of the loss function with respect to the weights; these are usually obtained using back-propagation or automatic differentiation. The weights are then updated using a first- or second-order algorithm, such as gradient descent, stochastic gradient descent (SGD), SGD with momentum, LBFGS, etc. Our procedure repeats iteratively until one of the following stopping criteria is met:

- The loss function value is less than `Tol`
- The number of passes over the training data reaches `MaxEpoch`

`Tol` and `MaxEpoch` are parameters of the algorithm; we use `Tol`$= 10^{-6}$ and `MaxEpoch`$= 3$.

So, once the loss value drops below `Tol` or the number of passes over the training dataset goes over `MaxEpoch`, we stop the training.

The loss function provides a measure for monitoring the improvement of the DNN algorithm through the iterations. However, it cannot be used to measure the quality of prediction, and it is not meaningful by itself. Since the prediction output is a binary value, the test error—the number of wrong predictions divided by the number of samples—is an appropriate measure. (See Appendix G for further discussion of this issue.) Moreover, statistics on false positives (type I error, the incorrect rejection of a true null hypothesis) and false negatives (type II error, the failure to reject a true null hypothesis) are helpful, and we use them to get more insights about how the algorithm works.

The DNN algorithm provides one prediction, in which the false positive and negative errors are weighted equally. However, the modeler should be able to control the likelihood of a stock-out prediction, i.e., the balance between false positive and false negative errors. To this end, we would benefit from a loss function that can provide control over the likelihood of a stock-out prediction, since the DNN's output is directly affected by its loss function.

The loss functions mentioned above do not have any weighting coefficient, and place equal weight between selecting 0 (predicting no stock-out) and 1 (predicting stock-out). To correct this, we propose weighing the loss function value that is incurred for each output, 0 and 1, using weights c_n and c_p, which represent the costs of false positive and negative errors, respectively. In this way, when $c_p < c_n$, the DNN tries to have a smaller number of cases in which it returns `False` but in fact $y_i = 0$, so it predicts more stock-outs to result in a smaller number of false negative errors and a larger number of false positive errors. Similarly, when $c_p > c_n$, the DNN predicts fewer stock-outs to avoid cases in which it returns `True` but in fact $y_i = 1$. Therefore, it makes a smaller number of false positive errors and a

larger number of false negative errors. If $c_n = c_p$, our revised loss function works similarly to the original loss functions.

Using this approach, the weighted hinge, weighted Euclidean, and weighted soft-max loss functions are as follows.

Hinge:

$$E = \frac{1}{N}\sum_{i=1}^{N} E_i \tag{3.5a}$$

$$E_i = \begin{cases} c_n \max(0, 1 - y_i\hat{y}_i) \text{ , if } \ y_i = 0 \\ \\ c_p \max(0, 1 - y_i\hat{y}_i) \text{ , if } \ y_i = 1, \end{cases} \tag{3.5b}$$

Euclidean:

$$E = \frac{1}{N}\sum_{i=1}^{N} E_i \tag{3.6a}$$

$$E_i = \begin{cases} c_n||y_i - \hat{y}_i||_2^2 \text{ , if } \ y_i = 0 \\ \\ c_p||y_i - \hat{y}_i||_2^2 \text{ , if } \ y_i = 1, \end{cases} \tag{3.6b}$$

Soft-max:

$$E = -\frac{1}{N}\sum_{i=1}^{N}\sum_{u=1}^{U} w_u \mathbb{I}\{y_i = u - 1\} \log \frac{e^{z_i^u}}{\sum_{v=1}^{U} e^{z_i^v}}, \tag{3.7}$$

where $U = 2$, $w_1 = c_n$, and $w_2 = c_p$. Thus, these loss functions allow one to manage the number of false positive and negative errors. Hereinafter, we use WDNN to denote the DNN that uses a weighted loss function.

3.3 Naive Approaches

In this section, we propose five naive approaches to predict stock-outs. These algorithms are used as baselines for measuring the quality of the DNN algorithm. They are easy to implement, but they do not consider the system state at any nodes other than the node for which we are predicting stockouts. (The proposed DNN approach, in contrast, uses the state at all nodes to provide a more effective prediction.)

In the naive algorithms, we use IP_t to denote the inventory position in period t. Also, v and u are the numbers of the training and testing records, respectively, and $d = [d_1, d_2, \cdots, d_v]$ is the demand of the customers in each period of the training set. Finally, the function `approximator(`s`)` takes a list s of numbers, fits a normal distribution to it, and returns the corresponding parameters of the normal distribution.

Algorithm 1 Naive Algorithm 1

procedure NAIVE-1
  given α as an input;
  $s = \{IP_t | y_{t+1} = 1, t = 1, \ldots, v\}$; ▷ Training procedure
  $\mu_s, \sigma_s =$ `approximator(`s`)`;
  $\eta_\alpha = \mu_s + \Phi_\alpha^{-1}(\sigma_s)$;
  for $t = 1 : u$ **do** ▷ Testing procedure
    if $IP_t < \eta_\alpha$ **then**
      prediction$(t) = 1$;
    else
      prediction$(t) = 0$;
    end if
  end for
  return prediction
end procedure

Naive Algorithm 1 first determines all periods in the training data in which a stock-out occurred and builds a list s of the inventory positions in the preceding period for each. Then it fits a normal distribution $\mathcal{N}(\mu_s, \sigma_s)$ to the values in s and calculates the αth quantile of

that distribution, for a given value of α. Finally, it predicts a stock-out in period $t+1$ if IP_t is less than that quantile. The value of $\alpha \in (0,1)$ is determined by the modeler.

Naive Algorithm 2 groups the inventory positions into a set of ranges, calculates the frequency of stock-outs in the training data for each range, and then predicts a stock-out in period $t+1$ if the range that IP_t falls into experienced stock-outs more than γ fraction of the time in the training data.

Naive Algorithm 3 uses classical inventory theory, which says the inventory level in period $t+L$ equals IP_t minus the lead-time demand, where L is the lead time [Zipkin, 2000, Snyder and Shen, 2019]. The algorithm estimates the lead-time demand distribution by fitting a normal distribution based on the training data, then predicts a stockout in period $t+1$ if IP_t is less than or equal to the αth quantile of the estimated lead-time demand distribution, where α is a parameter chosen by the modeler.

The value of α (and hence η_α) in Naive Algorithms 1 and 3 and the value of γ in Naive Algorithm 2 are selected by the modeler. A small value of α results in a small η_α so that the algorithm predicts fewer stock-outs. The same is true for a small γ. Generally, as α or γ decreases, the number of false positive errors decreases compared to the number of false negative errors, and vice versa. Thus, selecting an appropriate value of α or γ is important and directly affects the output of the algorithm. Indeed, the value of α or γ has to be selected according to the preferences of the company running the algorithm. For example, a company may have very expensive stock-outs. So, it may choose a very large α or γ so that the algorithm predicts frequent stock-outs, along with many more false positive errors, and then checks them one by one to prevent the stock-outs. In this situation the number of false positive errors increases; however, the company faces fewer false negative errors, which are

Algorithm 2 Naive Algorithm 2

procedure NAIVE-2
  $l = \min_{t=1}^{v}\{IP_t\}$; $u = \max_{t=1}^{v}\{IP_t\}$;
  given γ as an input;
  Divide $[l, u]$ into k equal intervals $[l_i, u_i], \forall i = 1, \cdots, k$;
  $SO_i = NSO_i = 0 \ \forall i = 1, \cdots, k$;
  for $t = 1 : v$ **do** ▷ Training procedure
    $s(t) = i$ such that $IP_t \in [l_i, u_i]$
    if $y_{t+1} = 1$ **then**
      $SO_{s(t)} += 1$;
    else
      $NSO_{s(t)} += 1$;
    end if
  end for
  for $t = 1 : u$ **do** ▷ Testing procedure
    $s(t) = i$ such that $IP_t \in [l_i, u_i]$
    if $SO_{s(t)} * \gamma > NSO_{s(t)}$ **then**
      prediction$(t) = 1$;
    else
      prediction$(t) = 0$;
    end if
  end for
  return prediction
end procedure

Algorithm 3 Naive Algorithm 3

procedure NAIVE-3
  $\mu_d, \sigma_d =$ `approximator(`$\{d_t\}_{t=1}^{v}$`)`; ▷ Training procedure
  given α as an input;
  $\eta_\alpha = \mu_d + \Phi_\alpha^{-1}(\sigma_d)$;
  for $t = 1 : u$ **do** ▷ Testing procedure
    if $IP_t < \eta_\alpha$ **then**
      prediction$(t) = 1$;
    else
      prediction$(t) = 0$;
    end if
  end for
  return prediction
end procedure

costly. In order to determine an appropriate value of α or γ, the modeler should consider the costs of false positive and negative errors, i.e., c_p and c_n, respectively.

The last two naive algorithms are simply classical forecasting methods. Naive Algorithm 4 uses exponential moving average (EMA) [Montgomery et al., 2015] while Naive Algorithm 5 uses linear regression. In both algorithms, we predict a stock-out if the value predicted by the forecasting method is greater than c_n/c_p.

3.4 Numerical Experiments

In order to check the validity and accuracy of our algorithm, we conducted a series of numerical experiments. Since there is no publicly available data of the type needed for our algorithm, we built a simulation model of a multi-echelon inventory system. Our simulation assumes that each node follows a base-stock policy, and that a node can make an order only if its predecessor has enough stock to satisfy it; this means that only the retailer nodes face stock-outs. The simulation records several state variables for each of the n nodes and for each of the T time periods. Figure 3.2 shows the flowchart of the simulation algorithm used.

To see how our algorithm works with different network topologies, we conducted multiple tests on five supply chain network topologies, ranging from a simple series system to complex networks containing (undirected) cycles and little or no symmetry. These tests are intended to explore the robustness of the DNN approach on simple or very complex networks. The five supply chain networks we used are:

- Serial network with 11 nodes.
- One warehouse, multiple retailer (OWMR) network with 11 nodes.

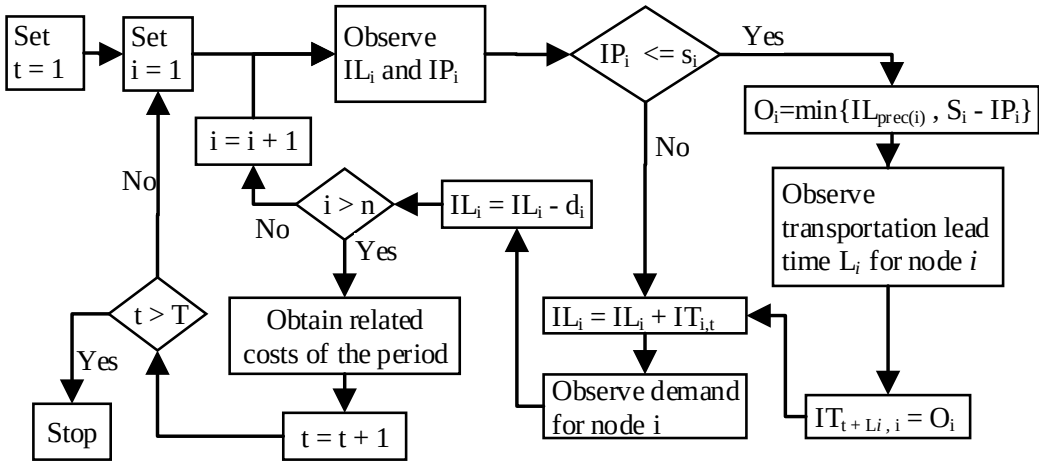


Figure 3.2: The simulation algorithm used to simulate a supply network

- Distribution network with 13 nodes.
- Complex network I with 11 nodes, including one retailer and two warehouses.
- Complex network II with 11 nodes, including three retailers and one node at the farthest echelon upstream (which we refer to as a warehouse).

We simulated each of the networks for 10^6 periods, with 75% of the resulting data used for training (and validation) and the remaining 25% for testing. For all of the problems we used a fully connected DNN network with 350 and 150 sigmoid nodes in the first and second layers, respectively. The inputs are the inventory levels and on-order inventories for each node from each of the $k = 11$ most recent periods (as given in (3.2)), and the output is the binary stock-out predictor for each of the nodes. Figure 3.3 shows a general view of the DNN network. Among the loss functions reviewed in Section 3.2, the soft-max loss function had the best accuracy in initial numerical experiments. Thus, the soft-max loss function was selected and its results are provided. To this end, we implemented the weighted soft-max function and its gradient (see Appendix E) in the DNN computation framework Caffe [Jia et al., 2014], and all of the tests were done on machines with 16 AMD cores and 32 GB of

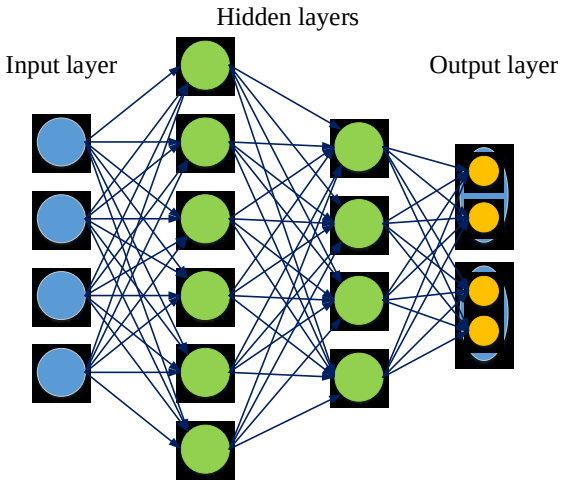


Figure 3.3: A network used to predict stock-outs of two nodes. For each of the networks, we used a similar network with n soft-max outputs.

memory. In order to optimize the network, the SGD algorithm—with batches of 50—with momentum is used, and each problem is run with `MaxEpoch=3`. Each epoch defines one pass over the training data. Finally, we tested 99 values of $\alpha \in \{0.01, 0.02, \ldots, 0.99\}$ and 118 values of (c_n, c_p), such that $c_p, c_n \in [0.3, 15]$. More details are provided in Appendix I, and the effects of changes in α, c_p, and c_n are explored in Section 3.4.8.

In the exponential moving average algorithm (EMA) (Naive Algorithm 4), we use a smoothing coefficient of $\frac{1}{11}$. For Naive Algorithm 5, we use the `scikit-learn` library [Pedregosa et al., 2011] in Python to perform the linear regression.

The DNN algorithm is scale dependent, meaning that the algorithm hyper-parameters (such as γ, learning rate, momentum, etc.; see Goodfellow et al. [2016]) are dependent on the values of c_p and c_n. Thus, a set of appropriate hyper-parameters of the DNN network for a given set of cost coefficients (c_p, c_n) does not necessarily work well for another set (c'_p, c'_n). This means that, ideally, for each set of (c_p, c_n), we should re-tune the DNN hyper-parameters, i.e., re-train the network. However, the tuning procedure is computationally expensive, so in our experiments we tuned the hyper-parameters for $c_p = 2$ and $c_n = 1$

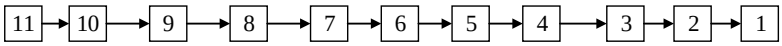
Table 3.1: The hyper-parameters used for each network

Network	lr	γ	λ
Serial	0.001	0.0005	0.0001
Distribution	0.0005	0.001	0.0005
OWMR	0.001	0.0005	0.0005
Complex-I	0.05	0.000005	0.000005
Complex-II, $(c_p, c_n) = (2, 1)$	0.05	0.05	0.05
Complex-II, $(c_p, c_n) = (1, 11)$	0.005	0.005	0.005

and used the resulting value for other sets of costs, in all network topologies. However, in complex network II, we did not get good convergence using this method, so we tuned the network for another set of cost coefficients to make sure that we get a non-diverging DNN for each set of coefficients. All of the resulting hyper-parameters are given in Table 3.1. To summarize, our experiments use minimal tuning (except for complex network II), far less than the amount typically used for a complete hyper-parameter tuning, but even so, the algorithm performs very well. Of course, additional tuning could further improve our results.

In what follows, we demonstrate the results of the DNN and compare them with those of the five naive algorithms in seven experiments. Sections 3.4.1–3.4.5 present the results of the serial, OWMR, distribution, complex I, and complex II networks, respectively. Section 3.4.7 extends these experiments: Section 3.4.7.1 discusses threshold prediction, Section 3.4.7.2 analyzes the results of a distribution network with multiple items with dependent demand, Section 3.4.7.3 discusses the effect of the supply chain network size, and Section 3.4.7.4 shows the results of predicting stock-outs multiple periods ahead in a distribution network. In each of the network topologies, we plot the false positive vs. false negative errors for all algorithms to compare their performance. In addition, two other figures in each section show the accuracy vs. false positive and negative errors to provide better insights

Figure 3.4: The serial network



into the way that the DNN algorithm (weighted and unweighted) works compared to the naive algorithms.

3.4.1 Results: Serial Network

Figure 3.4 shows the serial network with 11 nodes. The training dataset is used to train all five algorithms and the corresponding results are shown in Figures 3.5 and 3.6. Figure 3.5 plots the log-scaled false-negative errors vs. the false-positive errors for each approach and for a range of α values and a range of weights for the naive approaches and the weighted DNN approach. Points closer to the origin indicate more desirable solutions. Since there is just one retailer, the algorithms each make 2.5×10^5 stock-out predictions (one in each of the 2.5×10^5 testing periods); therefore, the number of errors in both figures should be compared to 2.5×10^5.

The DNN approach always dominates the naive approaches, with the unweighted version providing a slightly better accuracy but the weighted version providing more flexibility. For any given number of false-positive errors, the numbers of false-negative errors of the DNN and WDNN algorithms are smaller than those of the naive approaches, and similarly for a given number of false-negative errors. The results of the naive approaches are similar to each other, with Naive-1 and Naive-3 outperforming Naive-5 for most α values, and Naive-2 and Naive-4 not performing well at all. Similarly, Figure 3.6 plots the errors vs. the accuracy of the predictions and shows that for a given number of false positives or negatives, the DNN approaches attain a much higher level of accuracy than the naive approaches do. In conclusion, the naive algorithms perform similar to each other and worse than DNN, since

they do not use the available historical information. In contrast, DNN learns the relationship between state inputs and stock-outs and can predict stock-outs very well.

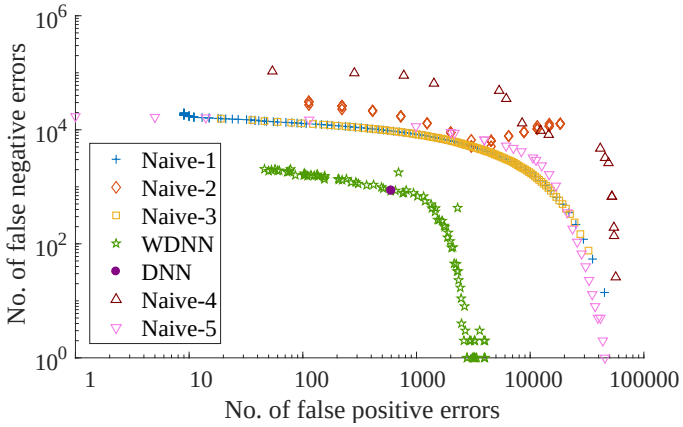


Figure 3.5: False positives vs. false negatives for the serial network

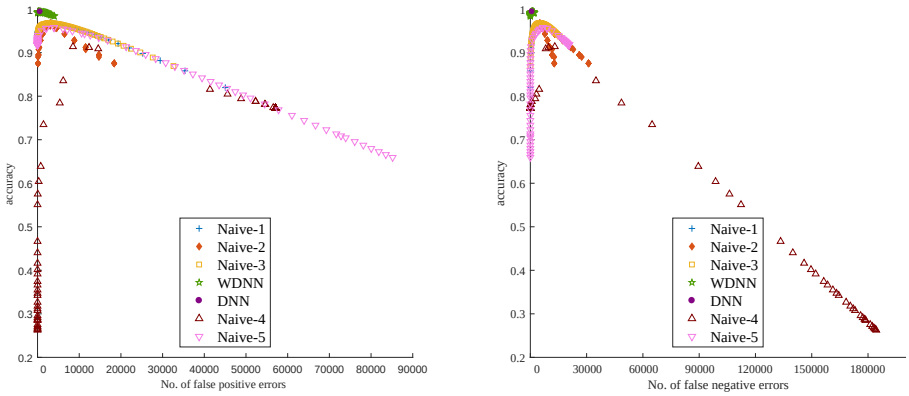


Figure 3.6: Accuracy of each algorithm for the serial network

3.4.2 Results: OWMR Network

Figure 3.7 shows the OWMR network with 11 nodes and Figures 3.8 and 3.9 present the experimental results for this network. Since there are 10 retailers, prediction is more challenging than for the serial network, as the algorithms each make 2.5×10^6 stock-out predictions; the number of errors in both figures should be compared to 2.5×10^6.

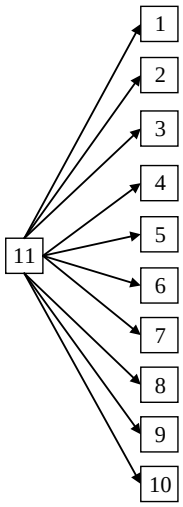


Figure 3.7: The OWMR network

Figure 3.8 shows the log-scaled false-negative errors vs. the false-positive errors for each approach and for a range of α values a range of weights for the naive approaches and the weighted DNN approach. DNN and weighted DNN dominate the naive approaches. Naive-1, Naive-3, and Naive-5 provide similar results to WDNN for a few (c_p, c_n) values; although, on average, Naive-1 and Naive-3 provide higher accuracy than the other naive approaches. Finally, Naive-2 and Naive-4 are somewhat worse than the other three. Figure 3.9 plots the errors vs. the accuracy of the predictions and confirms that DNN can attain higher accuracy levels for the same number of errors than the naive approaches. It is also apparent that all methods are less accurate for the OWMR system than they are for the serial system since there are many more predictions to make. However, DNN still provides better accuracy compared to the naive approaches.

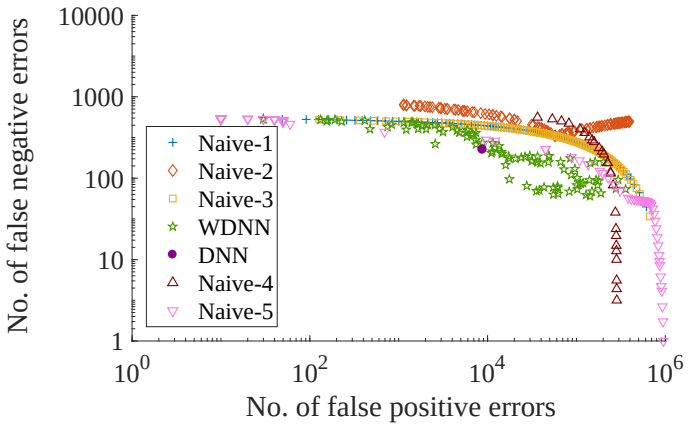


Figure 3.8: False positives vs. false negatives for the OWMR network

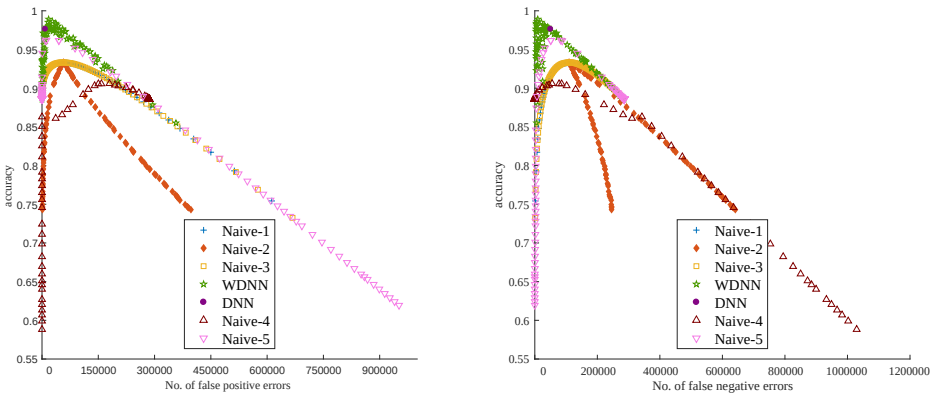


Figure 3.9: Accuracy of each algorithm for the OWMR network

3.4.3 Results: Distribution Network

Figure 3.10 shows the distribution network with 13 nodes, and Figure 3.11 provides the corresponding results of the five algorithms.

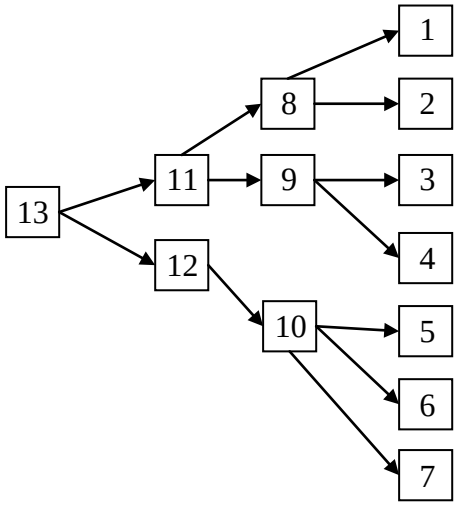


Figure 3.10: The distribution network

As Figure 3.11 shows, the DNN approach mostly dominates the naive approaches. However, it does not perform as well as in the serial or OWMR networks; that occurs because of the tuning of the DNN network hyper-parameters. Among the naive approaches, Naive-3 dominates Naive-1, since the demand data comes from a normal distribution without any noise, and the algorithm also approximates a normal distribution, which needs around 12 samples to get a good estimate of the mean and standard deviation. Therefore, the experiment is biased in favor of Naive-3. For a few (c_p, c_n) values, Naive-5 provides better results than WDNN; although, on average, Naive-1, Naive-2, and Naive-3 provide higher accuracy than Naive-5. Naive-4 provides the worst results. Plots of the errors vs. the accuracy of the predictions are similar to those in Figure 3.9; they are omitted to save space.

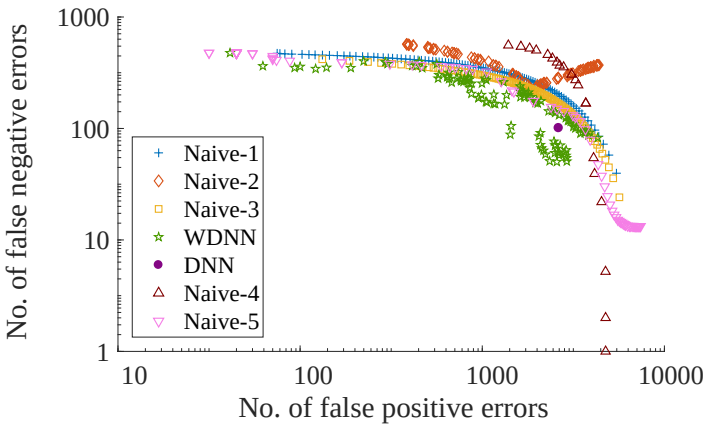


Figure 3.11: False positives vs. false negatives for the distribution network

Compared to the OWMR network, the distribution network includes fewer retailer nodes and therefore fewer stock-out predictions; however, the network is also more complex, and as a result the DNN is less accurate than it is for the OWMR network. We conclude that the accuracy of the DNN depends more on the number of echelons in the system than it does on the number of retailers.

3.4.4 Results: Complex Network I

Figure 3.12 shows a complex network with two warehouses (i.e., two nodes at the farthest echelon upstream), and Figure 3.13 presents the corresponding results of the five algorithms.

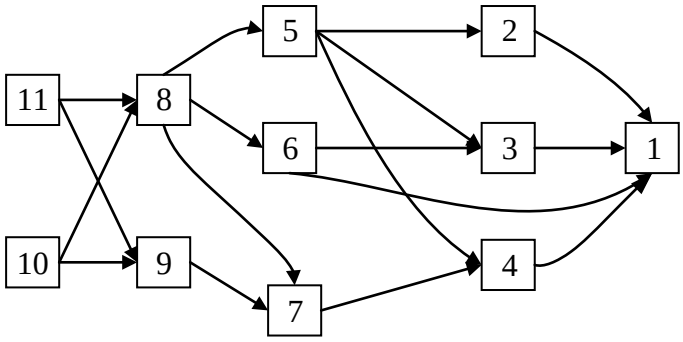


Figure 3.12: The complex network, two warehouses

Figure 3.13 plots the log-scaled false-negative errors vs. the false-positive errors for each

approach and for a range of α values and a range of weights for the naive approaches and the weighted DNN approach. The DNN approach dominates the naive approaches for most cases, but does worse when false-positives are tolerated in favor of reducing false-negatives. Additionally, the average accuracy rates for this system are 91% for WDNN and 97% for DNN, which show the importance of hyper-parameter tuning for each weight of the weighted DNN approach. Tuning it for each weight individually would improve the results significantly (but increase the computation time). Among the naive approaches, Naive-3 obtains the best accuracy on average, even though at a few (c_p, c_n) values, Naive-5 dominates all other algorithms, including DNN and WDNN. Plots of the errors vs. the accuracy of the predictions are similar to those in Figure 3.9; they are omitted to save space.

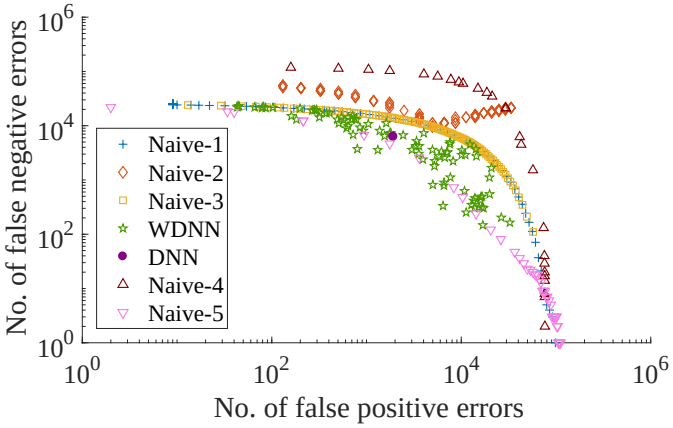


Figure 3.13: False positives vs. false negatives for complex network I

As in the serial network, there is just one retailer node; however, since the network is more complex, DNN produces less accurate predictions for complex network I than it does for the serial network, or for the other tree networks (OWMR and distribution). The added complexity of this network topology has an effect on the accuracy of our model, though the algorithm is still quite accurate.

3.4.5 Results: Complex Network II

Figure 3.14 shows the complex network with three retailers and Figure 3.15 presents the corresponding results of each algorithm.

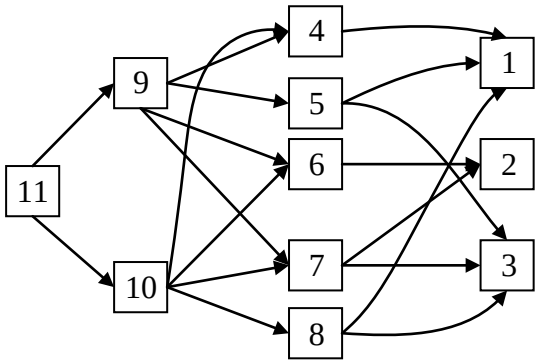


Figure 3.14: The complex network, three retailers

Figure 3.15 plots the log-scaled false-negative errors vs. the false-positive errors for each approach and for a range of α values and a range of weights for the naive approaches and for the weighted DNN approach. Figure 3.16 plots the errors vs. the accuracy of the predictions. As we did for the other network topologies, for complex network II we tuned the DNN network hyper-parameters for the case of $c_p = 2$ and $c_n = 1$ and used the resulting hyper-parameters for all other values of (c_p, c_n). However, the hyper-parameters obtained in this way did not work well for 46 sets of (c_p, c_n) values, mostly those with $c_p = 1$. In these cases, the training network did not converge, i.e., after 3 epochs of training, the network generally predicted 0 (or 1) for every data instance, even in the training set, and the loss values failed to decrease to an acceptable level. Thus, we also tuned the hyper-parameters for $c_p = 1$ and $c_n = 11$ and used them to obtain the results for these 46 cases. The hyper-parameters obtained using $(c_p, c_n) = (2, 1)$ and $(c_p, c_n) = (1, 11)$ are all given in Table 3.1. We used the first set of hyper-parameters for 72 of the 118 combinations of (c_p, c_n) values and the second set for the remaining 46 combinations. Additional hyper-parameter tuning would

result in further improved dominance of the DNN approach.

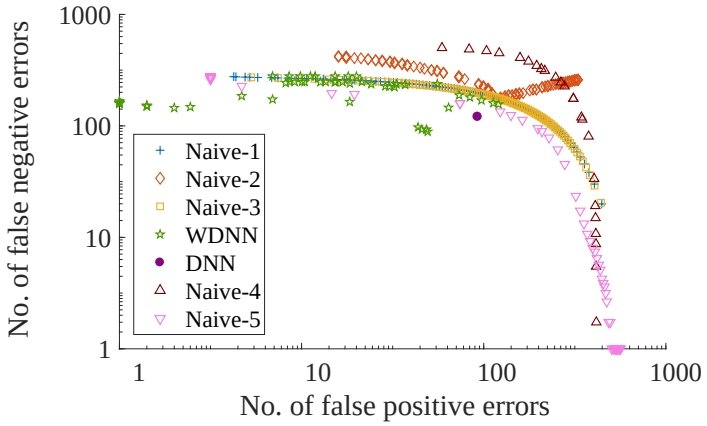


Figure 3.15: False positives vs. false negatives for complex network II

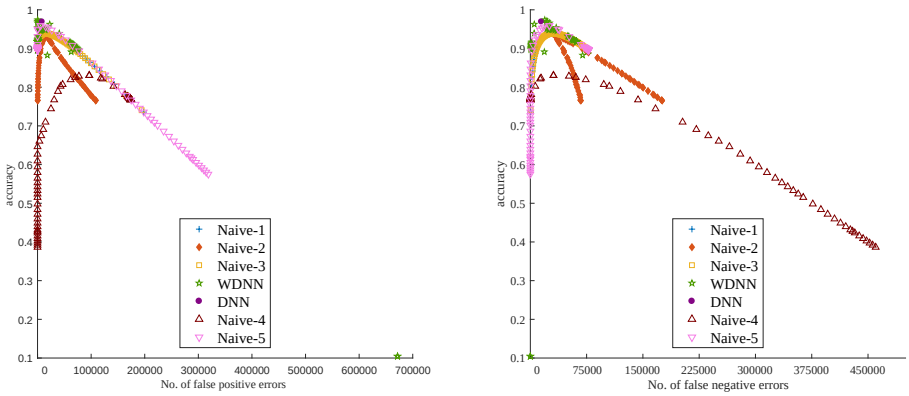


Figure 3.16: Accuracy of each algorithm for complex network II

Complex network II is the most complex network among all the networks we analyzed, since it is a non-tree network with multiple retailers. As Figure 3.16 shows, WDNN performs worse than the naive approaches for a few values of the weight, which shows the difficulty of the problems and the need to tune the network's hyper-parameters for each set of cost coefficients.

3.4.6 Results: Comparison

In order to get more insight, the average accuracy of each algorithm for each of the networks is presented in Table 3.2. The average is taken over all instances of a given network type, i.e., over all cost parameters. In the column headers, N1–N5 stand for the Naive-1 through Naive-5 algorithms. The corresponding hyper-parameters that we used to obtain these results are also presented in Table 3.1.

DNN provides the best accuracy compared to the other algorithms. Among the naive algorithms, the first three outperform the classical forecasting methods (Naive-4 and -5). WDNN is equally good for the serial and OWMR networks and slightly worse for the distribution and complex II networks. The difference is larger for complex I; this is a result of the fact that we did not re-tune the DNN network for each value of the cost parameters, as discussed in Section 3.4.4. We conclude that DNN is the method to choose if the user wants to ensure high accuracy; and WDNN is useful if the user wants to control the balance between false positive and false negative errors.

The column labeled N3 < N1 shows the number of cost-parameter values in which one of Naive-3's predictions has fewer false positive and fewer false negative errors than at least one of the predictions of Naive-1. This happens often for some networks, since the simulated data are normally distributed and since Naive-3 happens to assume a normal distribution. We would expect the method to work worse if the simulated data were from a different distribution. Finally, the last column shows a similar comparison for the Naive-3 and WDNN algorithms. In particular, Naive-3 never dominates WDNN in this way.

Generally, DNN with hyper-parameter tuning has its best performance in the serial, OWMR, then in complex I, and complex II networks and does a little bit worse in the

Table 3.2: Average accuracy of each algorithm

Network	N1	N2	N3	N4	N5	WDNN	DNN
Serial	0.94	0.92	0.95	0.68	0.89	0.99	0.99
Distribution	0.91	0.88	0.93	0.83	0.86	0.95	0.95
OWMR	0.91	0.84	0.91	0.85	0.85	0.95	0.98
Complex I	0.86	0.86	0.92	0.63	0.84	0.92	0.97
Complex II	0.91	0.85	0.92	0.71	0.85	0.94	0.97

distribution networks. The reason is that in the serial and OWMR networks, there is only node that affects the inventory level and the arriving shipments to the retailer nodes. Thus, DNN is better able to learn the situations in which the predecessor node may not be able to satisfy the retailers' demand. Although in complex I there are a few middle echelons, since there is only one retailer node it can learn how the three predecessor nodes affect the retailer. In complex II and the distribution network there are many middle nodes and many retailers so that stock-out prediction is not as easy as in other three networks. Thus, the performance of DNN is affected by (i) the number of middle echelons in the network, (ii) the number of nodes in the middle echelons, and (iii) the number of retailer nodes. When these three elements in a network increase, stock-out prediction becomes harder and the DNN performance becomes slightly worse.

Last, one might think of training n different neural networks for each of the n nodes of the multi-echelon network. Although this would be a quite expensive approach, it might improve the accuracy of the system. In order to see how this idea might work, we tested it on the distribution network, so that we trained 13 neural networks, with the best hyper-parameters obtained from tuning the distribution network. This approach resulted in an accuracy of 96.39%, a bit higher than the 95% accuracy obtained by the single DNN network. Thus, training n separate neural network helps improve the accuracy; however, it requires n times the computation power.

3.4.7 Extended Results

In this section we present results on some extensions of our original model and analysis. In Section 3.4.7.1, we examine the ability of the algorithms to predict whether the inventory level will fall below a given threshold that is not necessarily 0. In Section 3.4.7.2, we apply our method to problems with dependent demands. Finally, in Section 3.4.7.4, we explore multiple-period-ahead predictions.

3.4.7.1 Threshold Prediction

The models discussed above aim to predict whether a stock-out will occur; that is, whether the inventory level will fall below 0. However, it is often desirable for inventory managers to have more complete knowledge about inventory levels; in particular, we would like to be able to predict whether the inventory level will fall below a given threshold that is not necessarily 0. In order to see how well our proposed algorithms perform at this task, in this section we provide results for the case in which we aim to predict whether the inventory level will fall below 10.

A similar procedure is applied to achieve the results of all algorithms. In particular, we changed the way that the data labels are applied so that we assign a label of 1 when $IL < 10$ and a label of 0 otherwise. We exclude the results of the DNN and Naive-2 algorithms, since they are dominated by the WDNN and Naive-3 algorithms. Figures 3.17–3.21 present the results of the serial, OWMR, distribution, complex I, and complex II networks. As before, WDNN outperforms the naive algorithms. Table 3.3 provides the overall accuracy of all algorithms and the comparisons among them; the columns are the same as those in Table 3.2. As before, WDNN performs better than or equal to the other algorithms for all networks. The accuracy figures for this case are provided in Appendix J.

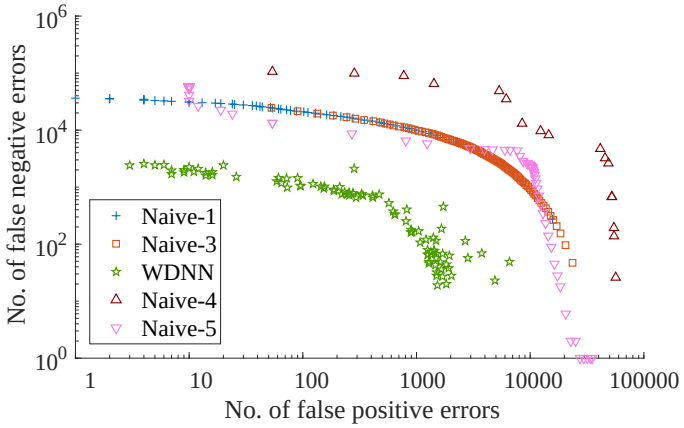


Figure 3.17: False positives vs. false negatives for serial network

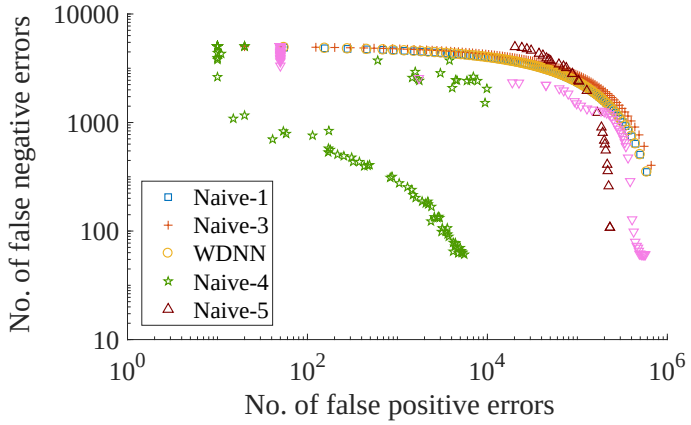


Figure 3.18: False positives vs. false negatives for OWMR network

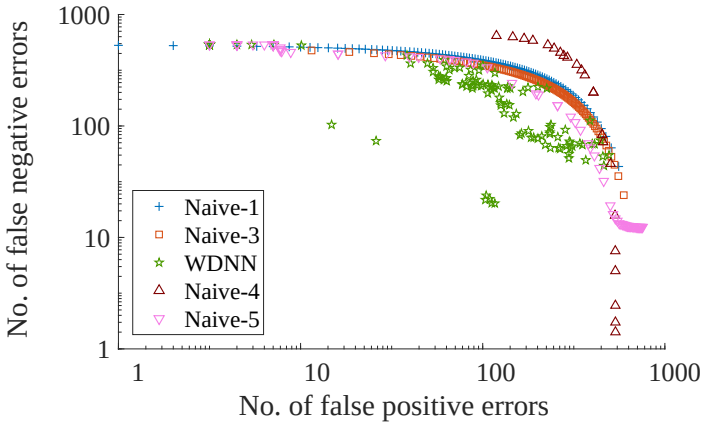


Figure 3.19: False positives vs. false negatives for distribution network

Table 3.3: Average accuracy of each algorithm for predicting inventory level less than 10

Network	N1	N3	N4	N5	WDNN
Serial	0.88	0.96	0.68	0.85	0.99
Distribution	0.90	0.92	0.79	0.83	0.93
OWMR	0.91	0.92	0.89	0.90	0.96
Complex I	0.85	0.87	0.63	0.73	0.97
Complex II	0.82	0.87	0.71	0.76	0.96

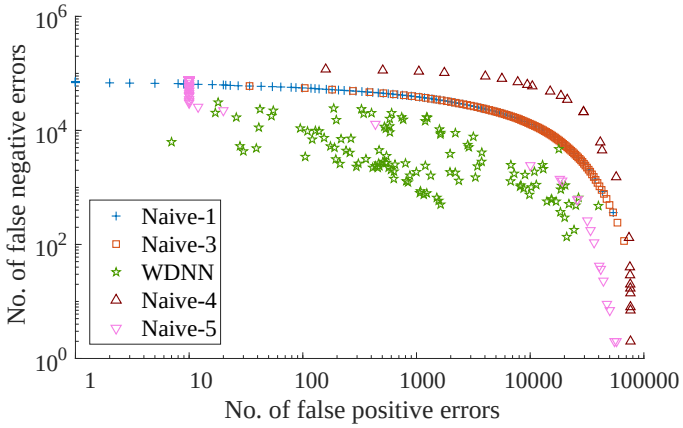


Figure 3.20: False positives vs. false negatives for complex network I

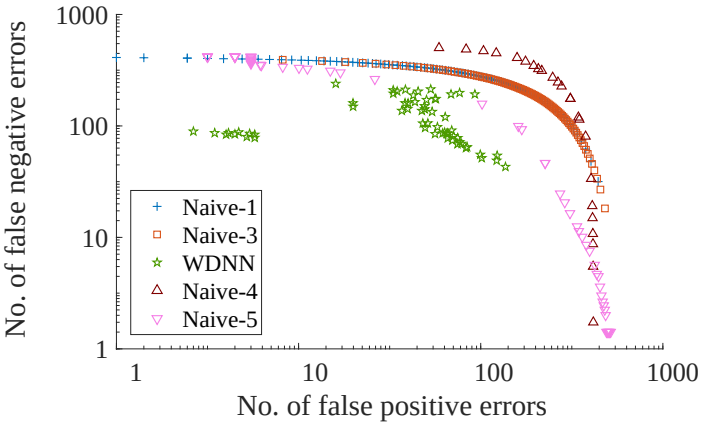


Figure 3.21: False positives vs. false negatives for complex network II

3.4.7.2 Multi-Item Dependent Demand Multi-Echelon Problem

The data sets we have used so far assume that the demands are statistically independent. However, in the real world, demands for multiple items are often dependent on each other.

Moreover, this dependence information provides additional information for DNN and might help to provide more accurate stock-out predictions. To analyze this, we generated the data for seven items with dependent demands, some positively and some negatively correlated. The mean demand of the seven items for seven days of a week is shown in Figure 3.22. For more details see Appendix H, which provides the demand means and standard deviations for each item and each day.

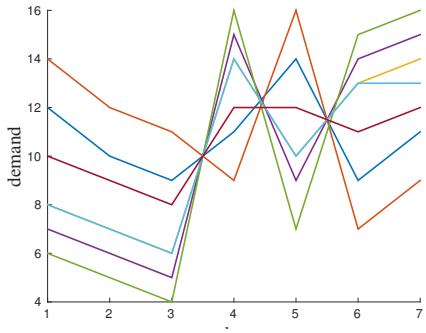


Figure 3.22: The demand of seven items in each day

We tested this approach using the distribution network (Figure 3.10). Figure 3.23 plots the false-negative errors vs. the false-positive errors for each approach and for a range of α values and a range of weights for the naive approaches and the weighted DNN approach. WDNN produces an average accuracy rate of 99% for this system, compared to 95% for the independent-demand case, which shows how DNN is able to make more accurate predictions by taking advantage of information it learns about the demand dependence. Finally, Figure 3.24 plots the errors vs. the accuracy of the predictions. DNN and WDNN provide much more accurate predictions than the naive methods.

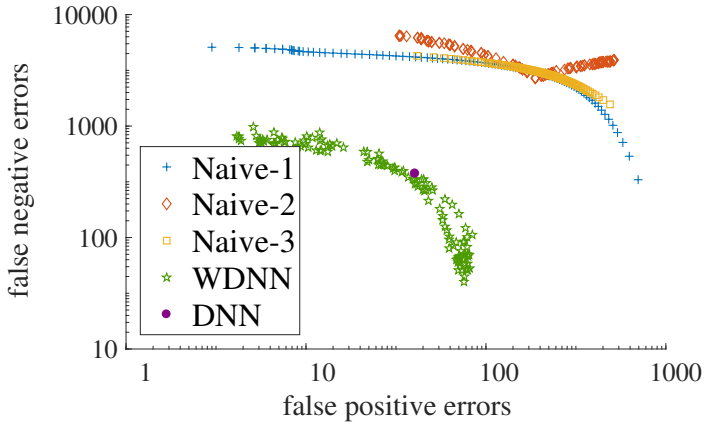


Figure 3.23: False positives vs. false negatives for distribution network with multi-item dependent demand

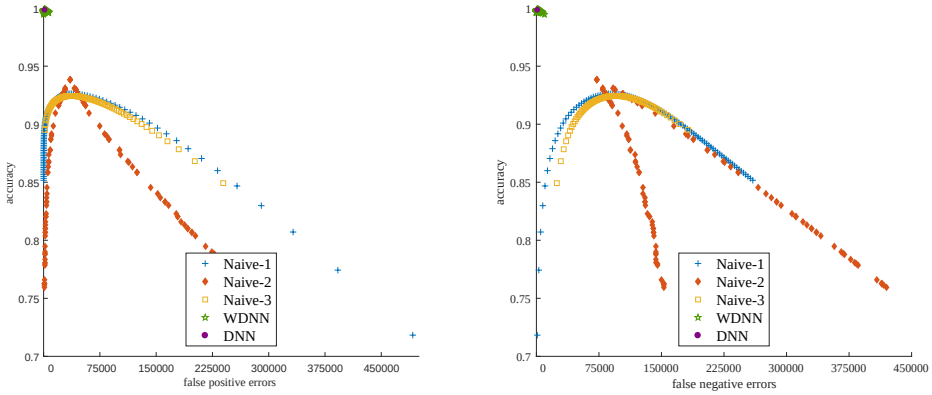
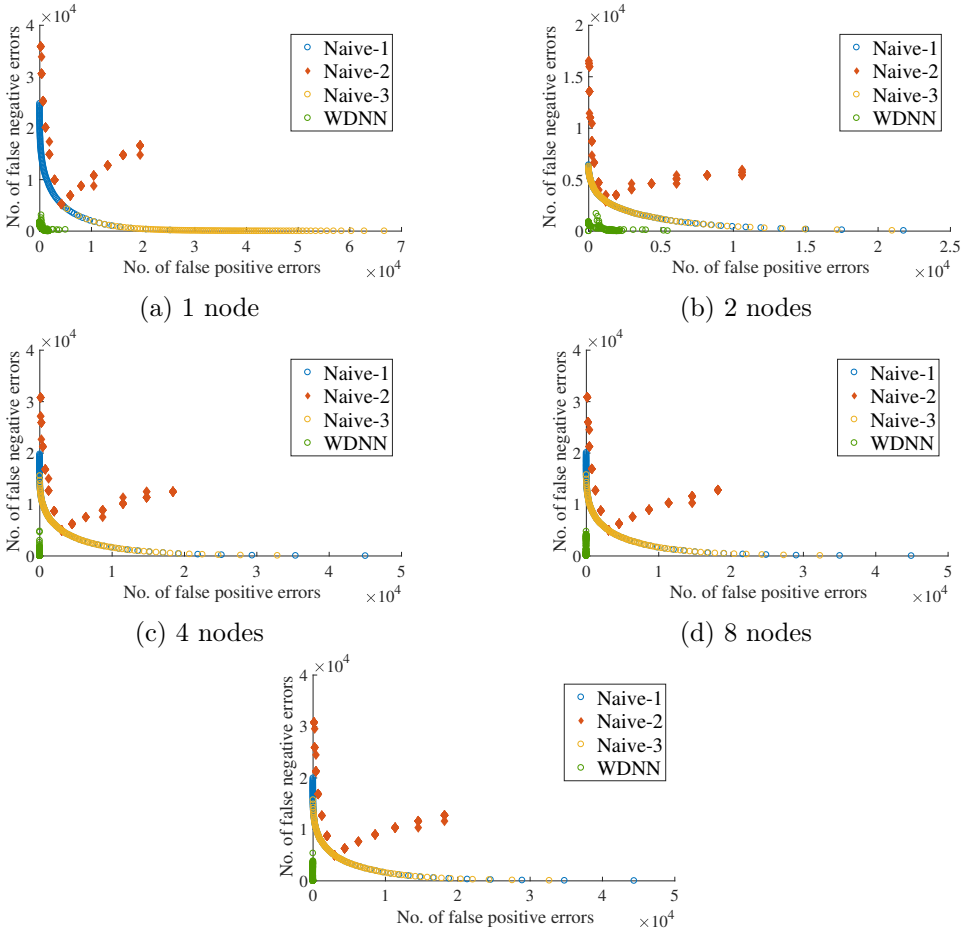


Figure 3.24: Accuracy of each algorithm for distribution network with multi-item dependent demand

3.4.7.3 Effect of Network Size

In this section, we examine the performance of DNN compared with the naive algorithms as the supply chain network size increases. In particular, Figure 3.25 provides the results of a serial network with 1, 2, 4, 8, and 16 nodes.

In order to get the results of the WDNN, we used the same hyper-parameters as in Table 3.1. As is shown in the figure, the naive algorithms perform the same on networks with different numbers of nodes, while the performance of WDNN slightly deteriorates as the number of nodes increases. Following a hyper-parameter tuning procedure would improve



(a) 1 node

(b) 2 nodes

(c) 4 nodes

(d) 8 nodes

(e) 16 nodes

Figure 3.25: False positives vs. false negatives for serial networks with 1, 2, 4, 8, and 16 nodes.

the accuracy, although as the number of nodes increases, prediction becomes harder.

3.4.7.4 Multi-Period Prediction

In order to see how well our algorithm can make stock-out predictions multiple periods ahead, we revised the DNN structure, such that there are $n \times q$ output values in the DNN algorithm, where q is the number of prediction periods. We tested this approach using the distribution network (Figure 3.10).

We tested the algorithm for three different problems. The first predicts stock-outs for each of the next two days; the second and third do the same for the next three and seven days, respectively. The accuracy of the predictions for each day are plotted in Figure 3.26. For example, the blue curve shows the accuracy of the predictions made for each of the next 3 days when we make predictions over a horizon of 3 days. The one-day prediction accuracy is plotted as a reference.

Not surprisingly, it is harder to predict stock-outs multiple days in advance. For example, the accuracy for days 4–7 is below 90% when predicting 7 days ahead. Moreover, when predicting over a longer horizon, the predictions for earlier days are less accurate. For example, the accuracy for predictions 2 days ahead is roughly 99% if we use a 2-day horizon, 95% if we use a 3-day horizon, and 94% if we use a 7-day horizon. Therefore, if we wish to make predictions for each of the next q days, it is more accurate (though slower) to run q separate DNN models rather than a single model that predicts the next q days.

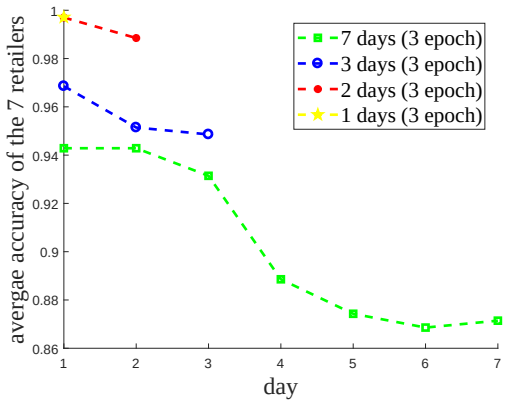


Figure 3.26: Average accuracy over seven days in multi-period prediction

3.4.8 Effect of α, c_p, and c_n on Accuracy

In this section, we briefly explore the prediction accuracy of the various algorithms as the parameters α and (c_p, c_n) change. Figures 3.28–3.31 demonstrate the accuracy for the threshold prediction case, in which we predict whether the inventory level will fall below 10. (Recall that we ruled out Naive-2 for threshold prediction since it was dominated by the other methods.)

First, we note that, although WDNN is more accurate than the other methods in nearly all cases, its accuracy is highest when $c_p/c_n = 2$ and falls slightly as c_p/c_n increases or decreases, especially for the distribution and complex networks, which are harder networks to analyze. The reason for this decline in accuracy is that we tuned the neural network for $(c_p, c_n) = (2, 1)$; as a result, problems with $c_p/c_n \approx 2$ have high accuracy, while other values of c_p/c_n give problems that are quite different from the problem that the hyper-parameters were tuned for. In general, we would expect the accuracy of WDNN to be good for any value of c_p/c_n, as long as the hyper-parameters are tuned appropriately for that value.

Next, the Naive-4 and Naive-5 algorithms work best near $c_p/c_n = 1$ and then plateau at larger values. Recall that these algorithms predict a stock-out if the output of a forecasting

model is greater than c_p/c_n. Both forecasting models produce outputs that can be greater than 1—say, in the range $[0, 1+\delta]$ for some δ. Once $c_p/c_n \geq 1+\delta$, the prediction is basically the same for all (c_p, c_n) values, which explains the plateau.

Naive-3 attains its highest accuracy around $\alpha = 0.5$, which is logical since this method predicts a stock-out if the inventory position IP_t is less than or equal to the estimate of the αth quantile of the lead-time demand distribution. As long as the estimate of the quantile is reasonably good, we would expect $\alpha = 0.5$ to produce the most accurate results, since classical inventory theory tells us that a stockout will occur if IP_t is less than the lead-time demand.

Finally, Naive-1 is skewed to the right. This algorithm predicts a stockout when IP_t is less than the αth quantile of the estimated distribution of inventory positions that resulted in stock-outs. One would expect that most of these inventory positions will tend to result in stock-outs, and therefore that using a value of α close to 1 will result in the highest accuracy.

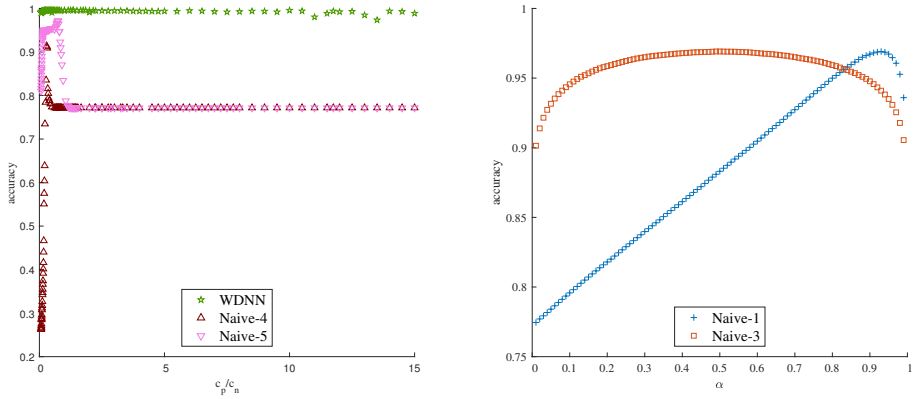


Figure 3.27: Effect of algorithm parameters on accuracy for serial network

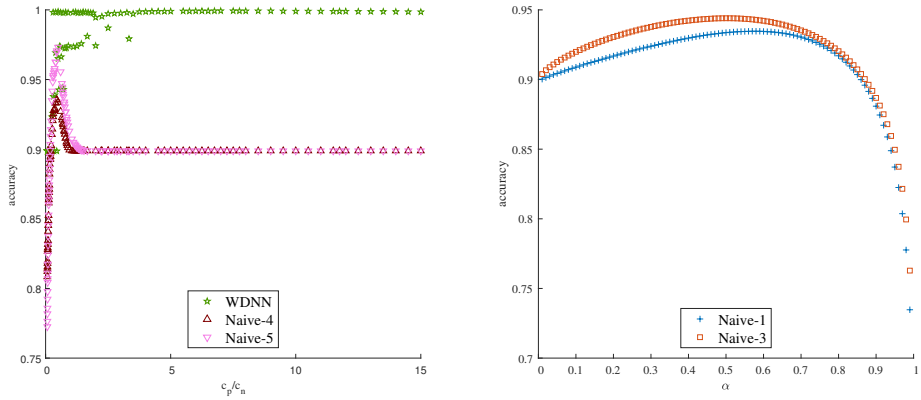


Figure 3.28: Effect of algorithm parameters on accuracy for OWMR network

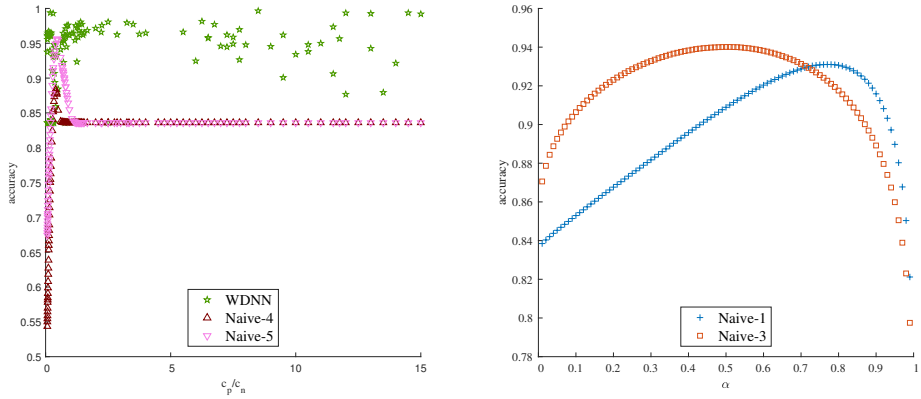


Figure 3.29: Effect of algorithm parameters on accuracy for distribution network

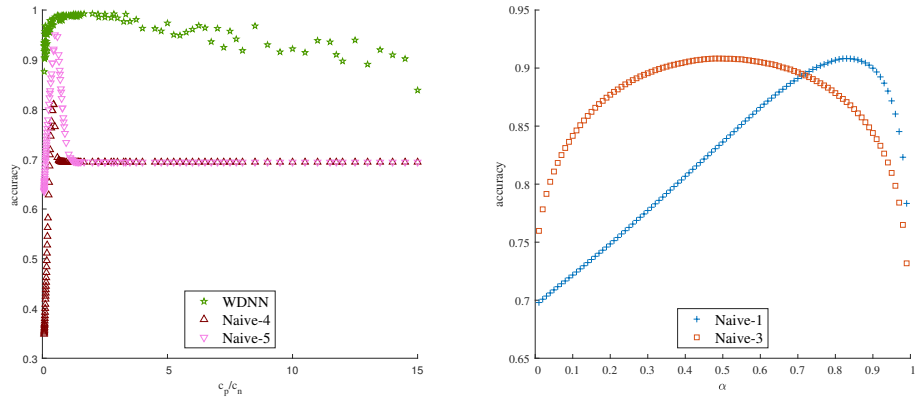


Figure 3.30: Effect of algorithm parameters on accuracy for complex network I

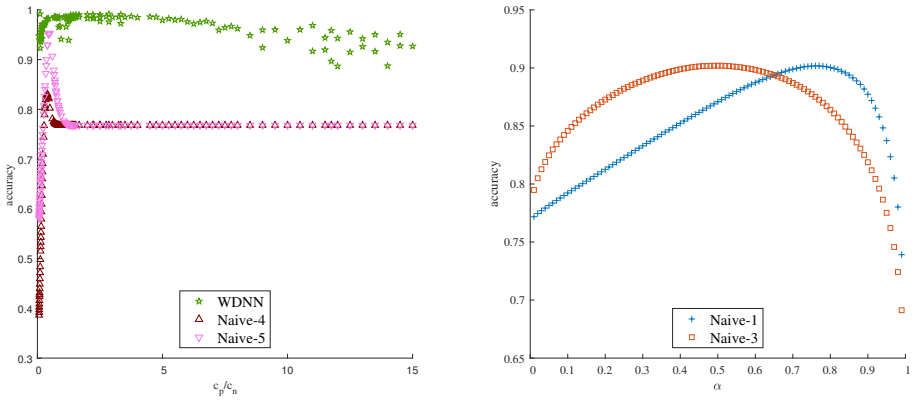


Figure 3.31: Effect of algorithm parameters on accuracy for complex network II

3.5 Conclusion and Future Work

We studied stock-out prediction in multi-echelon supply chain networks. In single-node networks, classical inventory theory provides tools for making such predictions when the demand distribution is known. However, to the best of our knowledge, there are no algorithms to predict stock-outs in multi-echelon networks. To address this need, we proposed an algorithm based on deep learning. We also introduced several naive algorithms to provide benchmarks for stock-out prediction. None of the algorithms requires knowledge of the demand distribution; they use only historical data.

Extensive numerical experiments show that the DNN algorithm works well compared to the naive algorithms. The results suggest that our method holds significant promise for predicting stock-outs in complex, multi-echelon supply chains. It obtains an average accuracy of 99% in serial networks and 95% for OWMR and distribution networks. Even for complex, non-tree networks, it attains an average accuracy of at least 91%. It also performs well when predicting inventory levels below a given threshold (not necessarily 0), making predictions when the demand is correlated, and making predictions multiple periods ahead.

Several research directions are now evident, including expanding the current approach

to handle other types of uncertainty, e.g., lead times, supply disruptions, etc. Improving the model's ability to make accurate predictions for more than one period ahead is another interesting research direction. Our current model appears to be able to make predictions accurately up to roughly 3 periods ahead, but its accuracy degrades quickly after that. Additionally, the current model uses the 11 previous periods for the input to capture the history of the network. Instead, utilizing LSTM could help to remember the history and get more accurate results. Finally, the model can be extended to take into account other supply chain state variables in addition to current inventory and in-transit levels.

Chapter 4

Application of Reinforcement Learning to the Beer Game

The beer game is a widely used in-class game that is played in supply chain management classes to demonstrate the bullwhip effect and the importance of supply chain coordination. The game is a decentralized, multi-agent, cooperative problem that can be modeled as a serial supply chain network in which agents cooperatively attempt to minimize the total cost of the network, even though each agent can only observe its own local information. Each agent chooses order quantities to replenish its stock. Under some conditions, a base-stock replenishment policy is known to be optimal. However, in a decentralized supply chain in which some agents (stages) may act irrationally (as they do in the beer game), there is no known optimal policy for an agent wishing to act optimally.

We propose a machine learning algorithm, based on deep Q-networks, to play the beer game. When playing with teammates who follow a base-stock policy, our algorithm obtains near-optimal order quantities. More importantly, it performs much better than a base-stock

policy when the other agents use a more realistic model of human ordering behavior. Unlike most other algorithms in the literature, our algorithm does not have any limits on the beer game parameter values. Like any deep learning algorithm, training the algorithm can be computationally intensive, but this can be performed ahead of time; the algorithm executes in real time when the game is played. Moreover, we propose a transfer learning approach so that the training performed for one agent and one set of cost coefficients can be adapted quickly for other agents and costs. Our approach can be extended to more general inventory and supply chain optimization problems, especially those in which supply chain partners act in irrational or unpredictable ways, i.e., to decentralized multi-agent cooperative games with partially observed information.

4.1 Introduction

The beer game consists of a serial supply chain network with four agents—a retailer, a warehouse, a distributor, and a manufacturer—who must make independent replenishment decisions with limited information. The game is widely used in classroom settings to demonstrate the *bullwhip effect*, a phenomenon in which order variability increases as one moves upstream in the supply chain, as well as the importance of communication and coordination in the supply chain. The bullwhip effect occurs for a number of reasons, some rational [Lee et al., 1997] and some behavioral [Sterman, 1989]. It is an inadvertent outcome that emerges when the players try to achieve the stated purpose of the game, which is to minimize costs. In this chapter, we are interested not in the bullwhip effect but in the stated purpose, i.e., the minimization of supply chain costs, which underlies the decision making in every real-world supply chain. For general discussions of the bullwhip effect, see, e.g., Lee

et al. [2004], Geary et al. [2006], and Snyder and Shen [2019].

The agents in the beer game are arranged sequentially and numbered from 1 (retailer) to 4 (manufacturer), respectively. (See Figure 4.1.) The retailer node faces stochastic demand from its customer, and the manufacturer node has an unlimited source of supply. There are deterministic transportation lead times (l^{tr}) imposed on the flow of product from upstream to downstream, though the actual lead time is stochastic due to stockouts upstream; there are also deterministic information lead times (l^{in}) on the flow of information from downstream to upstream (replenishment orders). Each agent may have nonzero shortage and holding costs.

In each period of the game, each agent chooses an order quantity q to submit to its predecessor (supplier) in an attempt to minimize the long-run system-wide costs,

$$\sum_{t=1}^{T} \sum_{i=1}^{4} c_h^i (IL_t^i)^+ + c_p^i (IL_t^i)^-, \tag{4.1}$$

where i is the index of the agents; $t = 1, \ldots, T$ is the index of the time periods; T is the time horizon of the game (which is often unknown to the players); c_h^i and c_p^i are the holding and shortage cost coefficients, respectively, of agent i; and IL_t^i is the inventory level of agent i in period t. If $IL_t^i > 0$, then the agent has inventory on-hand, and if $IL_t^i < 0$, then it has backorders, i.e., unmet demands that are owed to customers. The notation x^+ and x^- denotes $\max\{0, x\}$ and $\max\{0, -x\}$, respectively.

The standard rules of the beer game dictate that the agents may not communicate in any way, and that they do not share any local inventory statistics or cost information with other agents until the end of the game, at which time all agents are made aware of the system-wide cost. In other words, each agent makes decisions with only partial information

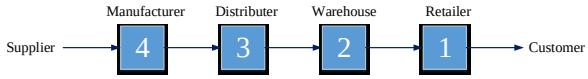


Figure 4.1: Generic view of the beer game network.

about the environment while also cooperating with other agents to minimize the total cost of the system. According to the categorization by Claus and Boutilier [1998], the beer game is a decentralized, independent-learners (ILs), multi-agent, cooperative problem.

The beer game assumes the agents incur holding and stockout costs but not fixed ordering costs, and therefore the optimal inventory policy is a *base-stock policy* in which each stage orders a sufficient quantity to bring its inventory position (on-hand plus on-order inventory minus backorders) equal to a fixed number, called its base-stock level [Clark and Scarf, 1960]. When there are no stockout costs at the non-retailer stages, i.e., $c_p^i = 0,\ i \in \{2, 3, 4\}$, the well known algorithm by Clark and Scarf [1960] (or its subsequent reworkings by Chen and Zheng [1994], Gallego and Zipkin [1999]) provides the optimal base-stock levels. To the best of our knowledge, there is no algorithm to find the optimal base-stock levels for general stockout-cost structures, e.g., with non-zero stockout costs at non-retailer agents. More significantly, when some agents do not follow a base-stock or other rational policy, the form and parameters of the optimal policy that a given agent should follow are unknown.

In this chapter, we propose a new algorithm based on deep Q-networks (DQN) to solve this problem. Our algorithm is customized for the beer game, but we view it also as a proof-of-concept that DQN can be used to solve messier, more complicated supply chain problems than those typically analyzed in the literature.

The remainder of this chapter is as follows. Section 4.2 provides a brief summary of the relevant literature and our contributions to it. The details of the algorithm are introduced in Section 4.3. Section 4.4 provides numerical experiments, and Section 4.5 concludes the

chapter.

4.2 Literature Review

4.2.1 Current State of Art

The beer game consists of a serial supply chain network. Under the conditions dictated by the game (zero fixed ordering costs, no ordering capacities, linear holding and backorder costs, etc.), a base-stock inventory policy is optimal at each stage [Lee et al., 1997]. If the demand process and costs are stationary, then so are the optimal base-stock levels, which implies that in each period (except the first), each stage simply orders from its supplier exactly the amount that was demanded from it. If the customer demands are iid random and if backorder costs are incurred only at stage 1, then the optimal base-stock levels can be found using the exact algorithm by Clark and Scarf [1960]; see also Chen and Zheng [1994], Gallego and Zipkin [1999]. This method involves decomposing the serial system into multiple single-stage systems and solving a convex, single-variable optimization problem at each. However, the objective function requires numerical integration and is therefore cumbersome to implement and computationally expensive. An efficient and effective heuristic method is proposed by Shang and Song [2003]. See also Snyder and Shen [2019] for a textbook discussion of these models.

There is a substantial literature on the beer game and the bullwhip effect. We review some of that literature here, considering both independent learners (ILs) and joint action learners (JALs) [Claus and Boutilier, 1998]. (ILs have no information about the other agent's current states, whereas JALs may share such information.) For a more comprehensive review, see Devika et al. [2016]. See Martinez-Moyano et al. [2014] for a thorough history of the

beer game.

In the category of ILs, Mosekilde and Larsen [1988] develop a simulation and test different ordering policies, which are expressed using a formula that involves state variables such as the number of anticipated shipments and unfilled orders. In their problem, there is one period of shipment and information lead time. They assume the customer demand is 4 in each of the first four periods, and then 7 per period for the remainder of the horizon. Sterman [1989] uses a similar version of the game in which the demand is 8 after the first four periods. (Hereinafter, we refer to this demand process as $C(4, 8)$ or the *classic* demand process.) Sterman [1989] does not allow the players to be aware of the demand process. He proposes a formula (which we call the *Sterman formula*) to determine the order quantity based on the current backlog of orders, on-hand inventory, incoming and outgoing shipments, incoming orders, and expected demand. His formula is based on the anchoring and adjustment method of Tversky and Kahneman [1979]. In a nutshell, the Sterman formula attempts to model the way human players over- or under-react to situations they observe in the supply chain such as shortages or excess inventory. Note that Sterman's formula is not an attempt to optimize the order quantities in the beer game; rather, it is intended to model typical human behavior. There are multiple extensions of Sterman's work. For example, Strozzi et al. [2007] consider the beer game when the customer demand increases linearly after four periods and proposes a genetic algorithm (GA) to obtain the coefficients of the Sterman model. Other behavioral beer game studies include Kaminsky and Simchi-Levi [1998a], Croson and Donohue [2003] and Croson and Donohue [2006a]. Also, Van Ackere et al. [1993] discuss how business process redesign can help to reduce costs. They propose four scenarios and analyze them through simulation. Similarly, Hieber and Hartel [2003] propose seven strategies and tested their performance with simulation.

Most of the optimization methods described in the first paragraph of this section assume that every agent follows a base-stock policy. The hallmark of the beer game, however, is that players do not tend to follow such a policy, or *any* policy. Often their behavior is quite irrational. There is comparatively little literature on how a given agent should optimize its inventory decisions when the other agents do not play rationally [Sterman, 1989, Strozzi et al., 2007]—that is, how an individual player can best play the beer game when her teammates may not be making optimal decisions.

Some of the beer game literature assumes the agents are JALs, i.e., information about inventory positions is shared among all agents, a significant difference compared to classical IL models. For example, Kimbrough et al. [2002] propose a GA that receives a current snapshot of each agent and decides how much to order according to the $d + x$ *rule*. In the $d + x$ rule, agent i observes d_t^i, the received demand/order in period t, chooses x_t^i, and then places an order of size $q_t^i = d_t^i + x_t^i$. In other words, x_t^i is the (positive or negative) amount by which the agent's order quantity differs from his observed demand. (We use the same ordering rule in our algorithm.) Giannoccaro and Pontrandolfo [2002] consider a beer game with three agents with stochastic shipment lead times and stochastic demand. They propose a reinforcement learning (RL) algorithm to make decisions, in which the state variable is defined as the three inventory positions, which are each discretized into 10 intervals. The agents may use any actions in the integers on $[0, 30]$. Chaharsooghi et al. [2008] consider the same game and solution approach as Giannoccaro and Pontrandolfo [2002] except with four agents and a fixed length of 35 periods for each game. In their proposed RL, the state variable is the four inventory positions, which are each discretized into nine intervals. Moreover, their RL algorithm uses the $d + x$ rule to determine the order quantity, with x restricted to be in $\{0, 1, 2, 3\}$. Note that these RL algorithms assume that

real-time information is shared among agents, whereas ours adheres to the typical beer-game assumption that each agent only has local information.

Additionally, Machuca and del Pozo Barajas [1997], Kaminsky and Simchi-Levi [1998b], Coakley et al. [1998], Goodwin and Franklin [1994], Jacobs [2000], Ravid and Rafaeli [2000], Chen and Samroengraja [2000], Martin et al. [2004], and Day and Kumar [2010] provide educational studies or procedures to teach the effect of sharing information, bullwhip effect, centralization effect, lead time effect, etc. With the same idea, Chen and Samroengraja [2000] describe a different version of the game for education purposes, named the stationary beer game, in which the customer demand in different periods is independently and identically distributed and all players know the demand distribution. Each player has a holding cost and only the retailer has a stock-out cost, and the inventory position of all agents is shared. They also has propose another variation of the game in which each agent is supposed to minimize its own cost, in two cases, whether they have the information about the demand in each period or not.

4.2.2 Reinforcement Learning

Reinforcement learning (RL) [Sutton and Barto, 1998] is an area of machine learning that has been successfully applied to solve complex sequential decision problems. RL is concerned with the question of how a software agent should choose an action in order to maximize a cumulative reward. RL is a popular tool in telecommunications [Al-Rawi et al., 2015], elevator scheduling [Crites and Barto, 1998], robot control [Finn and Levine, 2017], and game playing [Silver et al., 2016], to name a few.

Consider an agent that interacts with an environment. In each time step t, the agent observes the current state of the system, $s_t \in S$ (where S is the set of possible states),

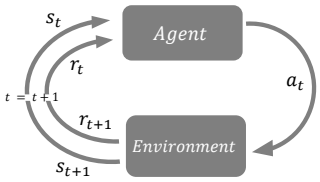


Figure 4.2: A generic procedure for RL.

chooses an action $a_t \in A(s_t)$ (where $A(s_t)$ is the set of possible actions when the system is in state s_t), and gets reward $r_t \in \mathbb{R}$; and then the system transitions randomly into state $s_{t+1} \in S$. This procedure is known as a *Markov decision process* (MDP) (see Figure 4.2), and RL algorithms can be applied to solve this type of problem.

The matrix $P_a(s, s')$, which is called the *transition probability matrix*, provides the probability of transitioning to state s' when taking action a in state s, i.e., $P_a(s, s') = \Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$. Similarly, $R_a(s, s')$ defines the corresponding reward matrix. In each period t, the decision maker takes action $a_t = \pi_t(s)$ according to a given policy, denoted by π_t. The goal of RL is to maximize the expected discounted sum of the rewards r_t, when the systems runs for an infinite horizon. In other words, the aim is to determine a policy $\pi : S \to A$ to maximize $\sum_{t=0}^{\infty} \gamma^t E\left[R_{a_t}(s_t, s_{t+1})\right]$, where $a_t = \pi_t(s_t)$ and $0 \leq \gamma < 1$ is the discount factor. For given $P_a(s, s')$ and $R_a(s, s')$, the optimal policy can be obtained through dynamic programming (e.g., using value iteration or policy iteration), or linear programming [Sutton and Barto, 1998].

Another approach for solving this problem is *Q-learning*, a type of RL algorithm that obtains the policy π for any $s \in S$ and $a = \pi(s)$, i.e.:

$$Q(s, a) = E\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots \mid s_t = s, a_t = a; \pi\right]. \tag{4.2}$$

The Q-learning approach starts with an initial guess for $Q(s, a)$ for all s and a and then

proceeds to update them based on the iterative formula

$$Q(s_t, a_t) = (1 - \alpha_t)Q(s_t, a_t) + \alpha_t \left(r_{t+1} + \gamma \max_a Q(s_{t+1}, a) \right), \forall t = 1, 2, \ldots, \tag{4.3}$$

where α_t is the learning rate at time step t. In each observed state, the agent chooses an action through an ϵ-greedy algorithm: with probability ϵ_t in time t, the algorithm chooses an action randomly, and with probability $1 - \epsilon_t$, it chooses the action with the highest cumulative action value, i.e., $a_{t+1} = \text{argmax}_a Q(s_{t+1}, a)$. The random selection of actions, called exploration, allows the algorithm to explore the solution space more fully and gives an optimality guarantee to the algorithm if $\epsilon_t \to 0$ when $t \to \infty$ [Sutton and Barto, 1998].

All of the algorithms discussed so far (dynamic programming, linear programming, and Q-learning) guarantee that they will obtain the optimal policy. However, due to the curse of dimensionality, these approaches are not able to solve MDPs with large state or action spaces in reasonable amounts of time. Many problems of interest (including the beer game) have large state and/or action spaces. Moreover, in some settings (again, including the beer game), the decision maker cannot observe the full state variable. This case, which is known as a *partially observed MDP* (POMDP), makes the problem much harder to solve than MDPs.

In order to solve large POMDPs and avoid the curse of dimensionality, it is common to approximate the Q-values in the Q-learning algorithm [Sutton and Barto, 1998]. Linear regression is often used for this purpose [Melo and Ribeiro, 2007]; however, it is not powerful or accurate enough for our application. Non-linear functions and neural network approximators are able to provide more accurate approximations; on the other hand, they are known to provide unstable or even diverging Q-values due to issues related to non-

stationarity and correlations in the sequence of observations [Mnih et al., 2013]. The seminal work of Mnih et al. [2015] solved these issues by proposing *target networks* and utilizing *experience replay memory* [Lin, 1992]. They proposed a *deep Q-network* (DQN) algorithm, which uses a deep neural network to obtain an approximation of the Q-function and trains it through the iterations of the Q-learning algorithm while updating another target network. This algorithm has been applied to many competitive games, which are reviewed by Li [2017]. Our algorithm for the beer game is based on this approach.

The beer game exhibits one more characteristic that differentiates it from most settings in which DQN is commonly applied, namely, that there are multiple agents that cooperate in a decentralized manner to achieve a common goal. Such a problem is called a decentralized POMDP, or Dec-POMDP. Due to the partial observability and the non-stationarity of the local observations of each agent, Dec-POMDPs are extremely hard to solve and are categorized as NEXP-complete problems [Bernstein et al., 2002].

The beer game exhibits all of the complicating characteristics described above—large state and action spaces, partial state observations, and decentralized cooperation. In the next section, we discuss the drawbacks of current approaches for solving the beer game, which our algorithm aims to overcome.

4.2.3 Drawbacks of Current Algorithms

In Section 4.2.1, we reviewed different approaches to solve the beer game. Although the model of Clark and Scarf [1960] can solve some types of serial systems, for more general serial systems neither the form nor the parameters of the optimal policy are known. Moreover, even in systems for which a base-stock policy is optimal, such a policy may no longer be optimal for a given agent if the other agents do not follow it. The formula-based beer-game

models by Mosekilde and Larsen [1988], Sterman [1989], and Strozzi et al. [2007] attempt to model human decision-making; they do not attempt to model or determine optimal decisions.

A handful of models have attempted to optimize the inventory actions in serial supply chains with more general cost or demand structures than those used by Clark and Scarf [1960]; these are essentially beer-game settings. However, these papers all assume full observation or a centralized decision maker, rather than the local observations and decentralized approach taken in the beer game. For example, Kimbrough et al. [2002] use a genetic algorithm (GA), while Chaharsooghi et al. [2008], Giannoccaro and Pontrandolfo [2002] and Jiang and Sheng [2009] use RL. However, classical RL algorithms can handle only a small or reduced-size state space. Accordingly, these applications of RL in the beer game or even simpler supply chain networks also assume (implicitly or explicitly) that size of the state space is small. This is unrealistic in the beer game, since the state variable representing a given agent's inventory level can be any number in $(-\infty, +\infty)$. Solving such an RL problem would be nearly impossible, as the model would be extremely expensive to train. Moreover, Chaharsooghi et al. [2008] and Giannoccaro and Pontrandolfo [2002], who model beer-game-like settings, assume sharing of information, which is not the typical assumption in the beer game. Also, to handle the curse of dimensionality, they propose mapping the state variable onto a small number of tiles, which leads to the loss of valuable state information and therefore of accuracy. Thus, although these papers are related to our work, their assumption of full observability differentiates their work from the classical beer game and from our work.

As we discussed in Section 4.2.2, the beer game is a Dec-POMDP. The algorithm proposed by Xuan et al. [2004] for general Dec-POMDPs cannot be used for the beer game since they allow agents to communicate, with some penalty; without the communication,

there is no way for the agents to learn the shared objective function. Similarly, Seuken and Zilberstein [2007] and Omidshafiei et al. [2017] propose algorithms to solve multi-agent problems under partial observability while assuming there is a reward shared by all agents that is known by all agents in every period, but in the beer game the agents do not learn the full reward until the game ends. For a survey of research on ILs with shared rewards, see Matignon et al. [2012].

Another possible approach to tackle this problem might be classical supervised machine learning algorithms. However, these algorithms also cannot be readily applied to the beer game, since there is no historical data in the form of "correct" input/output pairs. Thus, we cannot use a stand-alone support vector machine or deep neural network with a training data-set and train it to learn the best action (like the approach used in Chapters 2 and 3 to solve some simpler supply chain problems). Based on our understanding of the literature, there is a large gap between solving the beer game problem effectively and what the current algorithms can handle. In order to fill this gap, we propose a variant of the DQN algorithm to choose the order quantities in the beer game.

4.2.4 Our Contribution

We propose a Q-learning algorithm for the beer game in which a DNN approximates the Q-function. Indeed, the general structure of our algorithm is based on the DQN algorithm [Mnih et al., 2015], although we modify it substantially, since DQN is designed for single-agent, competitive, zero-sum games and the beer game is a multi-agent, decentralized, cooperative, non-zero-sum game. In other words, DQN provides actions for one agent that interacts with an environment in a competitive setting, and the beer game is a cooperative game in the sense that all of the players aim to minimize the total cost of the system in a

random number of periods. Also, beer game agents are playing independently and do not have any information from other agents until the game ends and the total cost is revealed, whereas DQN usually assumes the agent fully observes the state of the environment at any time step t of the game. For example, DQN has been successfully applied to Atari games [Mnih et al., 2015], but in these games the agent is attempting to defeat an opponent (human or computer) and observes full information about the state of the system at each time step t.

One naive approach to extend the DQN algorithm to solve the beer game is to use multiple DQNs, one to control the actions of each agent. However, using DQN as the decision maker of each agent results in a competitive game in which each DQN agent plays independently to minimize its own cost. For example, consider a beer game in which players 2, 3, and 4 each have a stand-alone, well-trained DQN and the retailer (stage 1) uses a base-stock policy to make decisions. If the holding costs are positive for all players and the stockout cost is positive only for the retailer (as is common in the beer game), then the DQN at agents 2, 3, and 4 will return an optimal order quantity of 0 in every period, since on-hand inventory hurts the objective function for these players, but stockouts do not. This is a byproduct of the independent DQN agents minimizing their own costs without considering the total cost, which is obviously not an optimal solution for the system as a whole.

Instead, we propose a unified framework in which the agents still play independently from one another, but in the training phase, we use a feedback scheme so that the DQN agent learns the total cost for the whole network and can, over time, learn to minimize it. Thus, the agents in our model play smartly in all periods of the game to get a near-optimal cumulative cost for any random horizon length.

In principle, our framework can be applied to multiple DQN agents playing the beer game simultaneously on a team. However, to date we have designed and tested our approach only for a single DQN agent whose teammates are not DQNs, e.g., they are controlled by simple formulas or by human players. Enhancing the algorithm so that multiple DQNs can play simultaneously and cooperatively is a topic of ongoing research.

Another advantage of our approach is that it does not require knowledge of the demand distribution, unlike classical inventory management approaches [e.g., Clark and Scarf, 1960]. In practice, one can approximate the demand distribution based on historical data, but doing so is prone to error, and basing decisions on approximate distributions may result in loss of accuracy in the beer game. In contrast, our algorithm chooses actions directly based on the training data and does not need to know, or estimate, the probability distribution directly.

The proposed approach works very well when we tune and train the DQN for a given agent and a given set of game parameters (e.g., costs, lead times, action spaces, etc.). Once any of these parameters changes, or the agent changes, in principle we need to tune and train a new network. Although this approach works, it is time consuming since we need to tune hyper-parameters for each new set of game parameters. To avoid this, we propose using a *transfer learning* approach [Pan and Yang, 2010] in which we transfer the acquired knowledge of one agent under one set of game parameters to another agent with another set of game parameters. In this way, we decrease the required time to train a new agent by roughly one order of magnitude.

To summarize, our algorithm is *a variant of the DQN algorithm for choosing actions in the beer game.* In order to attain near-optimal cooperative solutions, we develop *a feedback scheme as a communication framework.* Finally, to simplify training agents with new cost

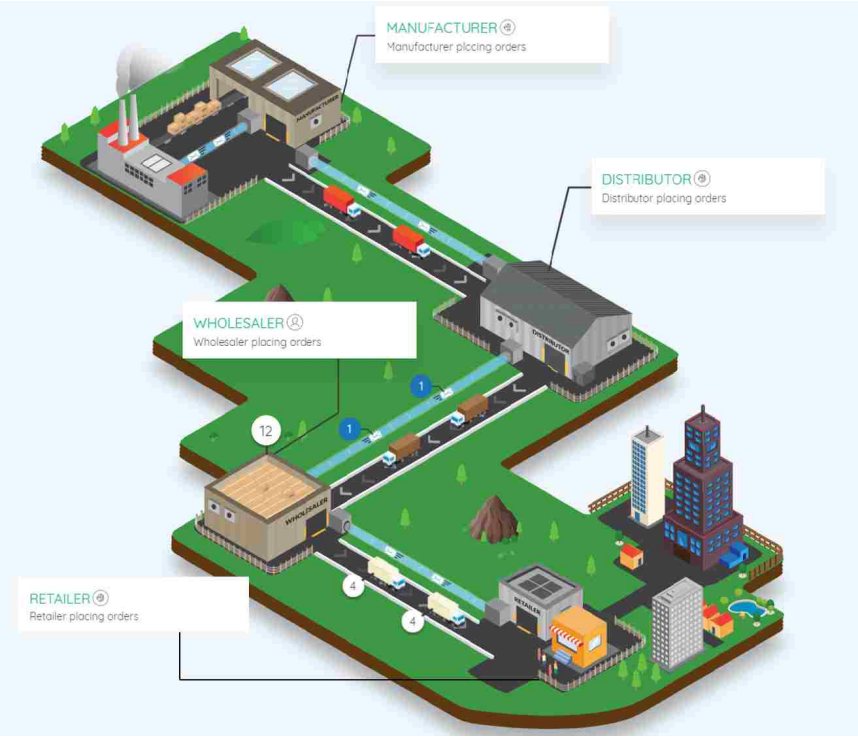


Figure 4.3: Screenshot of Opex Analytics online beer game integrated with our DQN agent

parameters, we use *transfer learning* to efficiently make use of the learned knowledge of trained agents. In addition to playing the beer game well, we believe our algorithm serves as a proof-of-concept that DQN and other machine learning approaches can be used for real-time decision making in complex supply chain settings.

Finally, we note that we have integrated our algorithm into a new online beer game developed by Opex Analytics (`http://beergame.opexanalytics.com/`); see Figure 4.3. The Opex beer game allows human players to compete with, or play on a team with, our DQN agent.

4.3 The DQN Algorithm

In this section, we first present the details of our DQN algorithm to solve the beer game, and then describe the transfer learning mechanism.

4.3.1 DQN for the Beer Game

In our algorithm, a DQN agent runs a Q-learning algorithm with DNN as the Q-function approximator to learn a semi-optimal policy with the aim of minimizing the total cost of the game. Each agent has access to its local information and considers the other agents as parts of its environment. That is, the DQN agent does not know any information about the other agents, including both static parameters such as costs and lead times, as well as dynamic state variables such as inventory levels. We propose a feedback scheme to teach the DQN agent to work toward minimizing the total system-wide cost, rather than its own local cost. The details of the scheme, Q-learning, state and action spaces, reward function, DNN approximator, and the DQN algorithm are discussed below.

State variables: Consider agent i in time step t. Let OO_t^i denote the on-order items at agent i, i.e., the items that have been ordered from agent $i+1$ but not received yet; let AO_t^i denote the size of the arriving order (i.e., the demand) received from agent $i-1$; let AS_t^i denote the size of the arriving shipment from agent $i+1$; let a_t^i denote the action agent i takes; and let IL_t^i denote the inventory level as defined in Section 4.1. We interpret AO_t^1 to represent the end-customer demand and AS_t^4 to represent the shipment received by agent 4 from the external supplier. In each period t of the game, agent i observes IL_t^i, OO_t^i, AO_t^i, and AS_t^i. In other words, in period t agent i has historical observations $o_t^i = \left[((IL_1^i)^+, IL_1^i)^-, OO_1^i, AO_1^i, RS_1^i), \ldots, ((IL_t^i)^+, IL_t^i)^-, , OO_t^i, AO_t^i, AS_t^i)\right]$ and does not have any information about the other agents. Thus, the agent has to make its decision with partially observed information of the environment. In addition, any beer game will finish in a finite time horizon, so the problem can be modeled as a POMDP in which each historic sequence o_t^i is a distinct state and the size of the vector o_t^i grows over time, which is difficult

for any RL or DNN algorithm to handle. To address this issue, we capture only the last m periods (e.g., $m = 3$) and use them as the state variable; thus the state variable of agent i in time t is $s_t^i = \left[((IL_j^i)^+, IL_j^i)^-, OO_j^i, AO_j^i, RS_j^i)\right]_{j=t-m+1}^{t}$.

DNN architecture: In our algorithm, DNN plays the role of the Q-function approximator, providing the Q-value as output for any pair of state s and action a. There are various possible approaches to build the DNN structure. One natural approach is to provide the state s and action a as the input of the DNN and then get the corresponding $Q(s, a)$ from the output. Another approach is to include the state s in the DNN's input and get the corresponding Q-value of all possible actions in the DNN's output so that the DNN output is of size $|A|$. The first approach requires more, but smaller, DNN networks, while the second requires fewer, larger ones. The second approach is much more efficient in the sense that it requires less training overall (even though the network is larger), so we use this approach in our algorithm. Thus, we provide as input the m previous state variables into the DNN and get as output $Q(s, a)$ for every possible action $a \in A(s)$.

Action space: In each period of the game, each agent can order any amount in $[0, \infty)$. Since our DNN architecture provides the Q-value of all possible actions in the output, having an infinite action space is not practical. Therefore, to limit the cardinality of the action space, we use the $d + x$ rule for selecting the order quantity: The agent determines how much more or less to order than its received order; that is, the order quantity is $d + x$, where x is in some bounded set. Thus, the output of the DNN is $x \in [a_l, a_u]$ $(a_l, a_u \in \mathbb{Z})$, so that the action space is of size $a_u - a_l + 1$.

Experience replay: The DNN algorithm requires a mini-batch of input and a corresponding set of output values to learn the Q-values. Since we use a Q-learning algorithm as our RL engine, we have information about the new state s_{t+1} along with information about the

current state s_t, the action a_t taken, and the observed reward r_t, in each period t. This information can provide the required set of input and output for the DNN; however, the resulting sequence of observations from the RL results in a non-stationary data-set in which there is a strong correlation among consecutive records. This makes the DNN and, as a result, the RL prone to over-fitting the previously observed records and may even result in a diverging approximator [Mnih et al., 2015, Foerster et al., 2017, de Bruin et al., 2015]. To avoid this problem, we follow the suggestion of Mnih et al. [2015] and use *experience replay* [Lin, 1992], taking a mini-batch from it in every training step. In this way, agent i has experience memory E^i, which holds the previously seen states, actions taken, corresponding rewards, and new observed states. Thus, in iteration t of the algorithm, agent i's observation $e_t^i = (s_t^i, a_t^i, r_t^i, s_{t+1}^i)$ is added to the experience memory of the agent so that E^i includes $\{e_1^i, e_2^i, \ldots, e_t^i\}$ in period t. Then, in order to avoid having correlated observations, we select a random mini-batch of the agent's experience replay to train the corresponding DNN (if applicable). This approach breaks the correlations among the training data and reduces the variance of the output [Mnih et al., 2013]. Moreover, as a byproduct of experience replay, we also get a tool to keep every piece of the valuable information, which allows greater efficiency in a setting in which the state and action spaces are huge and any observed experience is valuable. However, in our implementation of the algorithm we keep only the last M observations due to memory limits.

Reward function: In iteration t of the game, agent i observes state variable s_t^i and takes action a_t^i; we need to know the corresponding reward value r_t^i to measure the quality of action a_t^i. The state variable, s_{t+1}^i, allows us to calculate IL_{t+1}^i and thus the corresponding shortage or holding costs, and we consider the summation of these costs for r_t^i. However, since there are information and transportation lead times, there is a delay between taking

action a_t^i and observing its effect on the reward. Moreover, the reward r_t^i reflects not only the action taken in period t, but also those taken in previous periods, and it is not possible to decompose r_t^i to isolate the effects of each of these actions. However, defining the state variable to include information from the last m periods resolves this issue; the reward r_t^i represents the reward of state s_t^i, which includes the observations of the previous m steps.

On the other hand, the reward values r_t^i are the intermediate rewards of each agent, and the objective of the beer game is to minimize the total reward of the game, $\sum_{i=1}^{4}\sum_{t=1}^{T} r_t^i$, which the agents only learn after finishing the game. In order to add this information into the agents' experience, we revise the reward of the relevant agents in all T time steps through a feedback scheme.

Feedback scheme: When any episode of the beer game is finished, all agents are made aware of the total reward. In order to share this information among the agents, we propose a penalization procedure in the training phase to provide feedback to the DQN agent about the way that it has played. Let $\omega = \sum_{i=1}^{4}\sum_{t=1}^{T} \frac{r_t^i}{T}$ and $\tau^i = \sum_{t=1}^{T} \frac{r_t^i}{T}$, i.e., the average reward per period and the average reward of agent i per period, respectively. After the end of each episode of the game (i.e., after period T), for each DQN agent i we update its observed reward in all T time steps in the experience replay memory using $r_t^i = r_t^i + \frac{\beta_i}{3}(\omega - \tau^i)$, $\forall t \in \{1, \ldots, T\}$, where β_i is a regularization coefficient for agent i. With this procedure, agent i gets appropriate feedback about its actions and learns to take actions that result in minimum total cost, not locally optimal solutions. This feedback scheme gives the agents a sort of implicit communication mechanism, even though they do not communicate directly.

Determining the value of *m*: As noted above, the DNN maintains information from the most recent m periods in order to keep the size of the state variable fixed and to address the issue with the delayed observation of the reward. In order to select an appropriate value

for m, one has to consider the value of the lead times throughout the game. First, when agent i takes a given action a_t^i at time t, it does not observe its effect until at least $l_i^{tr} + l_i^{in}$ periods later, when the order may be received. Moreover, node $i+1$ may not have enough stock to satisfy the order immediately, in which case the shipment is delayed and in the worst case agent i will not observe the corresponding reward r_t^i until $\sum_{j=i}^{4}(l_j^{tr} + l_j^{in})$ periods later. However, the Q-learning algorithm needs the reward r_t^i to evaluate the action a_t^i taken. Thus, ideally m should be chosen at least as large as $\sum_{j=1}^{4}(l_j^{tr} + l_j^{in})$. On the other hand, this value can be large and selecting a large value for m results in a large input size for the DNN, which increases the training time. Therefore, selecting m is a trade-off between accuracy and computation time, and m should be selected according to the required level of accuracy and the available computation power. In our numerical experiment, $\sum_{j=1}^{4}(l_j^{tr} + l_j^{in}) = 15$ or 16, and we test $m \in \{5, 10\}$.

The algorithm: Our algorithm to get the policy π to solve the beer game is provided in Algorithm 4. The algorithm, which is based on that of Mnih et al. [2015], finds weights θ of the DNN network to minimize the Euclidean distance between $Q(s, a; \theta)$ and y_j, where y_j is the prediction of the Q-value that is obtained from target network Q^- with weights θ^-. Every C iterations, the weights θ^- are updated by θ. Moreover, the actions in each training step of the algorithm are obtained by an ϵ-greedy algorithm, which is explained in Section 4.2.2.

In the algorithm, in period t agent i takes action a_t^i, satisfies the arriving demand/order AO_{t-1}^i, observes the new demand AO_t^i, and then receives the shipments AS_t^i. This sequence of events results in the new state s_{t+1}. Feeding s_{t+1} into the DNN network with weights θ provides the corresponding Q-value for state s_{t+1} and all possible actions. The action with the smallest Q-value is our choice. Finally, at the end of each episode, the feedback scheme

Algorithm 4 DQN for Beer Game

procedure DQN
  for $Episode = 1 : n$ **do**
    Initialize Experience Replay Memory, $E_i = [\,]$, $\forall i$
    Reset IL, OO, d, AO, and AS for each agent
    for $t = 1 : T$ **do**
      for $i = 1 : 4$ **do**
        With probability ϵ take random action a_t,
        otherwise set $a_t = \underset{a}{\operatorname{argmin}}\, Q(s_t, a; \theta)$
        Execute action a_t, observe reward r_t and state s_{t+1}
        Add $(s_t^i, a_t^i, r_t^i, s_{t+1}^i)$ into the E_i
        Get a mini-batch of experiences (s_j, a_j, r_j, s_{j+1}) from E_i
        Set $y_j = \begin{cases} r_j & \text{if it is the terminal state} \\ r_j + \min\limits_{a} Q(s, a; \theta^-) & \text{otherwise} \end{cases}$
        Run one forward and one backward step on the DNN with loss function
        $(y_j - Q(s_j, a_j; \theta))^2$
        Every C iterations, set $\theta^- = \theta$
      end for
    end for
    Run feedback scheme, update experience replay of each agent
  end for
end procedure

runs and distributes the total cost among all agents. The details of beer game simulation steps are provided in Appendix O.

Evaluation procedure: In order to validate our algorithm, we compare the results of our algorithm to those obtained using the heuristic for base-stock levels in serial systems by Shang and Song [2003] (and, when possible, the optimal solutions by Clark and Scarf [1960]), as well as models of human beer-game behavior by Sterman [1989]. (Note that none of these methods attempts to do exactly the same thing as our method. The methods by Shang and Song [2003] and Clark and Scarf [1960] optimize the base-stock levels assuming all players follow a base-stock policy—which beer game players do not tend to do—and the formula by Sterman [1989] models human beer-game play, but they do not attempt to optimize.) The details of the training procedure and benchmarks are described in Section 4.4.

4.3.2 Transfer Learning

Transfer learning has been an active and successful field of research in machine learning and especially in image processing (see Pan and Yang [2010]). In transfer learning, there is a *source* dataset `S` and a trained neural network to perform a given task, e.g. classification, regression, or decisioning through RL. Training such networks may take a few days or even weeks. So, for similar or even slightly different *target* datasets `T`, one can avoid training a new network from scratch and instead use the same trained network with a few customizations. The idea is that most of the learned knowledge on dataset `S` can be used in the target dataset with a small amount of additional training. This idea works well in image processing (e.g. Razavian et al. [2014], Rajpurkar et al. [2017]) and considerably reduces the training time.

In order to use transfer learning in the beer game, we first train a fixed-size network for a given agent $i \in \{1, 2, 3, 4\}$ with a given set of game parameters $P_1^i = \{|A_1^i(s)|, c_{p_1}^i, c_{h_1}^i\}$. ($P_1^i$ includes the size of agent i's action space as well as its costs, but in principle one could also include lead times and other game parameters.) Suppose we have trained agent i assuming that the customer's demand was generated from a given distribution, call it D_1, and that the three co-players followed a given policy, call it π_1. Assume that we wish to apply this learned knowledge to other agents, with other game parameters. For those agents, we construct a new DNN network in which the input values, as well as the learned weights in the first layer(s), are similar to the values from the fully trained agent, i. As we get closer to the final layer, which provides the Q-values, the weights become less similar to agent i's and more specific to each agent. Thus, the acquired knowledge in the first k hidden layer(s) of the neural network belonging to agent i is transferred to agent j, with $P_2^j \neq P_1^i$, where k is a tunable parameter.

To be more precise, assume there exists a source agent $i \in \{1, 2, 3, 4\}$ with trained network S_i, parameters $P_1^i = \{|A_1^j(s)|, c_{p_1}^j, c_{h_1}^j\}$, observed demand distribution D_1, and co-player policy π_1. The weight matrix W_i contains the learned weights such that W_i^q denotes the weight between layers q and $q+1$ of the neural network, where $q \in \{0, \ldots, nh\}$, and nh is the number of hidden layers. The aim is to train a neural network S_j for target agent $j \in \{1, 2, 3, 4\}$, $j \neq i$. We set the structure of the network S_j the same as that of S_i, and initialize W_j with W_i, making the first k layers not trainable. Then, we train neural network S_j with a small learning rate.

In Section 4.4.3, we test the use of transfer learning in six cases to transfer the learned knowledge of source agent i to:

1. Target agent $j \neq i$ in the same game.
2. Target agent j with $\{|A_1^j(s)|, c_{p_2}^j, c_{h_2}^j\}$, i.e., the same action space but different cost coefficients.
3. Target agent j with $\{|A_2^j(s)|, c_{p_1}^j, c_{h_1}^j\}$, i.e., the same cost coefficients but different action space.
4. Target agent j with $\{|A_2^j(s)|, c_{p_2}^j, c_{h_2}^j\}$, i.e., different action space and cost coefficients.
5. Target agent j with $\{|A_2^j(s)|, c_{p_2}^j, c_{h_2}^j\}$, i.e., different action space and cost coefficients, as well as a different demand distribution D_2.
6. Target agent j with $\{|A_2^j(s)|, c_{p_2}^j, c_{h_2}^j\}$, i.e., different action space and cost coefficients, as well as a different demand distribution D_2 and co-player policy π_2.

Unless stated otherwise, the demand distribution and co-player policy are the same for the source and target agents.

Transfer learning could also be used when other aspects of the problem change, e.g., lead times, state representation, and so on. This avoids having to tune the parameters of the neural network for each new problem, which considerably reduces the training time. However, we still need to decide how many layers should be trainable, as well as to determine which agent can be a base agent for transferring the learned knowledge. Still, this is computationally much cheaper than finding each network and its hyper-parameters from scratch.

4.4 Numerical Experiments

In Section 4.4.1, we discuss a set of numerical experiments that uses a simple demand distribution and a relatively small action space:

- $d_0^t \in \mathbb{U}[0,2]$, $A(s_t) = \{-2,-1,0,1,2\}$.

After exploring the behavior of our algorithm under different co-player policies, in Section 4.4.2 we test the algorithm using three well-known cases from the literature, which have larger possible demand values and action spaces:

- $d_0^t \in \mathbb{U}[0,8]$, $A(s_t) = \{-8,\ldots,8\}$ [Croson and Donohue, 2006b]
- $d_0^t \in \mathbb{N}(10,2^2)$, $A(s_t) = \{-5,\ldots,5\}$ [adapted from Chen and Samroengraja, 2000, , who assume $\mathbb{N}(50,20^2)$]
- $d_0^t \in C(4,8)$, $A(s_t) = \{-8,\ldots,8\}$ [Sterman, 1989].

As noted above, we only consider cases in which a single DQN plays with non-DQN agents, e.g., simulated human co-players. In each of the cases listed above, we consider three types of policies that the non-DQN co-players follow: (i) base-stock policy, (ii) Sterman formula, (iii)

random policy. In the random policy, agent i also follows a $d + x$ rule, in which $a_i^t \in A(s_i^t)$ is selected randomly and with equal probability, for each t.

After analyzing these cases, in Section 4.4.3 we provide the results obtained using transfer learning for each of the six proposed cases.

In the training, the rewards (costs) are normalized by dividing them by 200, which helps to reduce the loss function values and produce smooth training. We test values of m in $\{5, 10\}$ and $C \in \{5000, 10000\}$. (Recall that m is the number of periods of history that are stored in the state variable, and C is the number of iterations after which the weights θ^- are updated.) Our DNN network is a fully connected network, in which each node has a ReLU activation function. The input is of size $5m$, and there are three hidden layers in the neural network. There is one output node for each possible value of the action, and each of these nodes takes a value in $\mathbb{R}$ indicating the Q-value for that action. Thus, there are $a_u - a_l + 1$ output nodes, and the neural network has shape $[5m, 180, 130, 61, a_u - a_l + 1]$.

In order to optimize the network, we used the Adam optimizer [Kingma and Ba, 2014] with a batch size of 64. Although the Adam optimizer has its own weight decaying procedure, we used exponential decay with a stair of 10000 iterations with rate 0.98 to decay the learning rate further. This helps to stabilize the training trajectory. We trained each agent on at most 60000 episodes and used a replay memory E equal to the one million most recently observed experiences. Also, the training of the DNN starts after observing at least 500 episodes of the game. The ϵ-greedy algorithm starts with $\epsilon = 0.9$ and linearly reduces it to 0.1 in the first 80% of iterations. The algorithm is implemented in Python using TensorFlow [Abadi et al., 2016].

In the feedback mechanism, the appropriate value of the feedback coefficient β_i heavily depends on τ_j, the average reward for agent j, for each $j \neq i$. For example, when τ_i is

one order of magnitude larger than τ_j, for all $j \neq i$, agent i needs a large coefficient to get more feedback from the other agents. Indeed, the feedback coefficient has a similar role as the regularization parameter λ has in the lasso loss function; the value of that parameter depends on the ℓ-norm of the variables, but there is no universal rule to determine the best value for λ. Similarly, proposing a simple rule or value for each β_i is not possible, as it depends on τ_i, $\forall i$. For example, we found that a very large β_i does not work well, since the agent tries to decrease other agents' costs rather than its own. Similarly, with a very small β_i, the agent learns how to minimize its own cost instead of the total cost. Therefore, we used a similar cross validation approach to find good values for each β_i.

4.4.1 Basic Cases

In this section, we test our approach using a beer game setup with the following characteristics. Information and shipment lead times, l_j^{tr} and l_j^{in}, equal 2 periods at every agent. Holding and stockout costs are given by $c_h = [2, 2, 2, 2]$ and $c_p = [2, 0, 0, 0]$, respectively, where the vectors specify the values for agents $1, \ldots, 4$. The demand is an integer uniformly drawn from $\{0, 1, 2\}$. Additionally, we assume that agent i observes the arriving shipment AS_t^i when it chooses its action for period t. We relax this assumption later. We use $a_l = -2$ and $a_u = 2$; so that there are 5 outputs in the neural network. i.e., each agent chooses an order quantity that is at most 2 units greater or less than the observed demand. (Later, we expand these to larger action spaces.)

We consider two types of simulated human players. In Section 4.4.1.1, we discuss results for the case in which one DQN agent plays on a team in which the other three players use a base-stock policy to choose their actions, i.e., the non-DQN agents behave rationally. See `https://youtu.be/gQa6iWGcGWY` for a video animation of the policy that the DQN

learns in this case. Then, in Section 4.4.1.2, we assume that the other three agents use the Sterman formula (i.e., the anchoring-and-adjustment formula by Sterman [1989]), which models irrational play.

For the cost coefficients and other settings specified for this beer game, it is optimal for all players to follow a base-stock policy, and we use this policy (with the optimal parameters as determined by the method of Clark and Scarf [1960]) as a benchmark and a lower bound on the base stock cost. The vector of optimal base-stock levels is $[8, 8, 0, 0]$, and the resulting average cost per period is 2.0705, though these levels may be slightly suboptimal due to rounding. This cost is allocated to stages 1–4 as $[2.0073, 0.0632, 0.03, 0.00]$, i.e., the retailer bears the most significant share of the total cost. In the experiments in which one of the four agents is played by DQN, the other three agents continue to use their optimal base-stock levels.

4.4.1.1 DQN Plus Base-Stock Policy

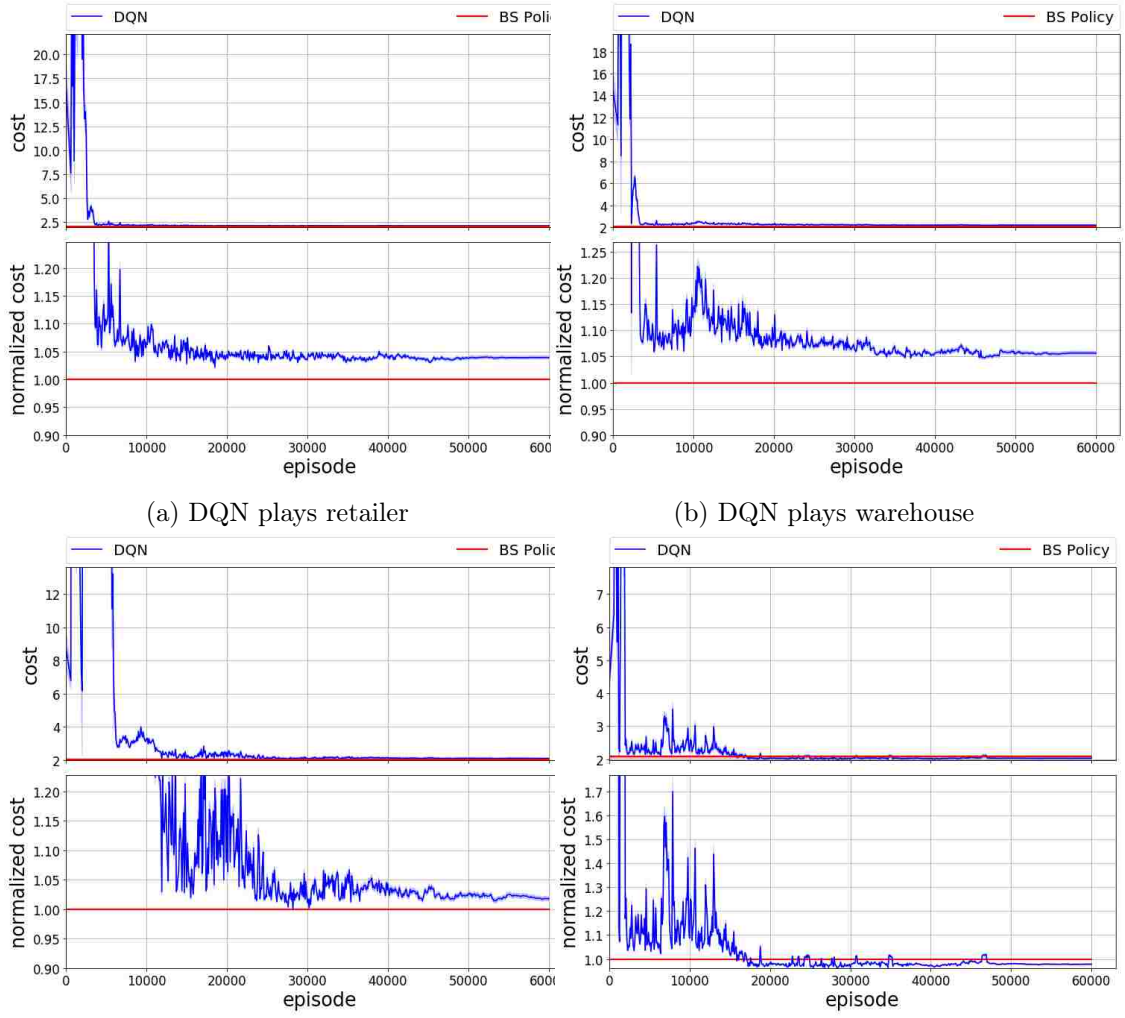
We consider four cases, with the DQN playing the role of each of the four players and the co-players using a base-stock policy. We then compare the results of our algorithm with the results of the case in which all players follow a base-stock policy, which we call `BS` hereinafter.

The results of all four cases are shown in Figure 4.4. Each plot shows the training curve, i.e., the evolution of the average cost per game as the training progresses. In particular, the horizontal axis indicates the number of training episodes, while the vertical axis indicates the total cost per game. After every 100 episodes of the game and the corresponding training, the cost of 50 validation points (i.e., 50 new games), each with 100 periods, are obtained and their average plus a 95% confidence interval are plotted. (The confidence intervals,

which are light blue in the figure, are quite narrow, so they are difficult to see.) The red line indicates the cost of the case in which all players follow a base-stock policy. In each of the sub-figures, there are two plots, the upper plot shows the cost, while the lower plot shows the normalized cost, in which each cost is divided by the corresponding `BS` cost; essentially this is a "zoomed-in" version of the upper plot. The confidence intervals in the lower sub-figures are also calculated based on the normalized costs. We trained the network using values of $\beta \in \{5, 10, 20, 50, 100, 200\}$, each for at most 60000 episodes. Figure 4.4 plots the results from the best β_i value for each agent; we present the full results using different β_i, m and C values in Appendix M.

The figure indicates that DQN performs well in all cases and finds policies whose costs are close to those of the `BS` policy. After the network is fully trained (i.e., after 60000 training episodes), the average gap between the DQN cost and the `BS` cost, over all four agents, is 2.31%.

Figure 4.5 shows the trajectories of the retailer's inventory level (IL), on-order quantity (OO), order quantity (a), reward (r), and order up to level (OUTL) for a single game, when the retailer is played by the DQN with $\beta_1 = 50$, as well as when it is played by a base-stock policy (`BS`), and the Sterman formula (`Strm`). The base-stock policy and DQN have similar IL and OO trends, and as a result their rewards are also very close: `BS` has a cost of $[1.42, 0.00, 0.02, 0.05]$ (total 1.49) and DQN has $[1.43, 0.01, 0.02, 0.08]$ (total 1.54, or 3.4% larger). (Note that `BS` has a slightly different cost here than reported on page 120 because those costs are the average costs of 50 samples while this cost is from a single sample.) Similar trends are observed when the DQN plays the other three roles; see Appendix K. This suggests that the DQN can successfully learn to achieve costs close to `BS` when the other agents also play `BS`. (The OUTL plot shows that the DQN does not quite *follow* a



(a) DQN plays retailer

(b) DQN plays warehouse

(c) DQN plays distributor

(d) DQN plays manufacturer

Figure 4.4: Total cost (upper figure) and normalized cost (lower figure) with one DQN agent and three agents that follow base-stock policy

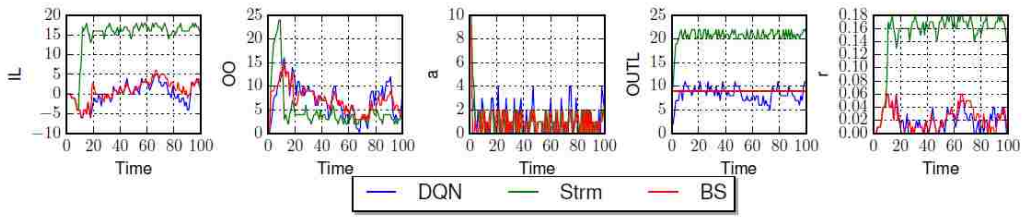


Figure 4.5: IL_t, OO_t, a_t, r_t, and $OUTL$ when DQN plays retailer and other agents follow base-stock policy

base-stock policy, even though its costs are similar.)

4.4.1.2 DQN Plus Sterman Formula

Figure 4.6 shows the results of the case in which the three non-DQN agents use the formula proposed by Sterman [1989] instead of a base-stock policy. (See Appendix L for the formula and its parameters.) We train the network using values of $\beta \in \{1, 2, 5, 10, 20, 50, 75, 100\}$, each for 40000 episodes, and report the best result among them. For comparison, the red line represents the case in which the single agent is played using a base-stock policy and the other three agents continue to use the Sterman formula, a case we call `Strm-BS`.

From the figure, it is evident that the DQN plays much *better* than `Strm-BS`. This is because if the other three agents do not follow a base-stock policy, it is no longer optimal for the fourth agent to follow a base-stock policy, or to use the same base-stock level. In general, the optimal inventory policy when other agents do not follow a base-stock policy is an open question. This figure suggests that our DQN is able to learn to play effectively in this setting.

Table 4.1 gives the cost of all four agents when a given agent plays using either DQN or a base-stock policy and the other agents play using the Sterman formula. From the table, we can see that the DQN produces similar (but slightly smaller) costs than a base-stock policy when used by the retailer, and significantly smaller costs than a base-stock policy when

Table 4.1: Average cost under different choices of which agent uses DQN or `Strm-BS`.

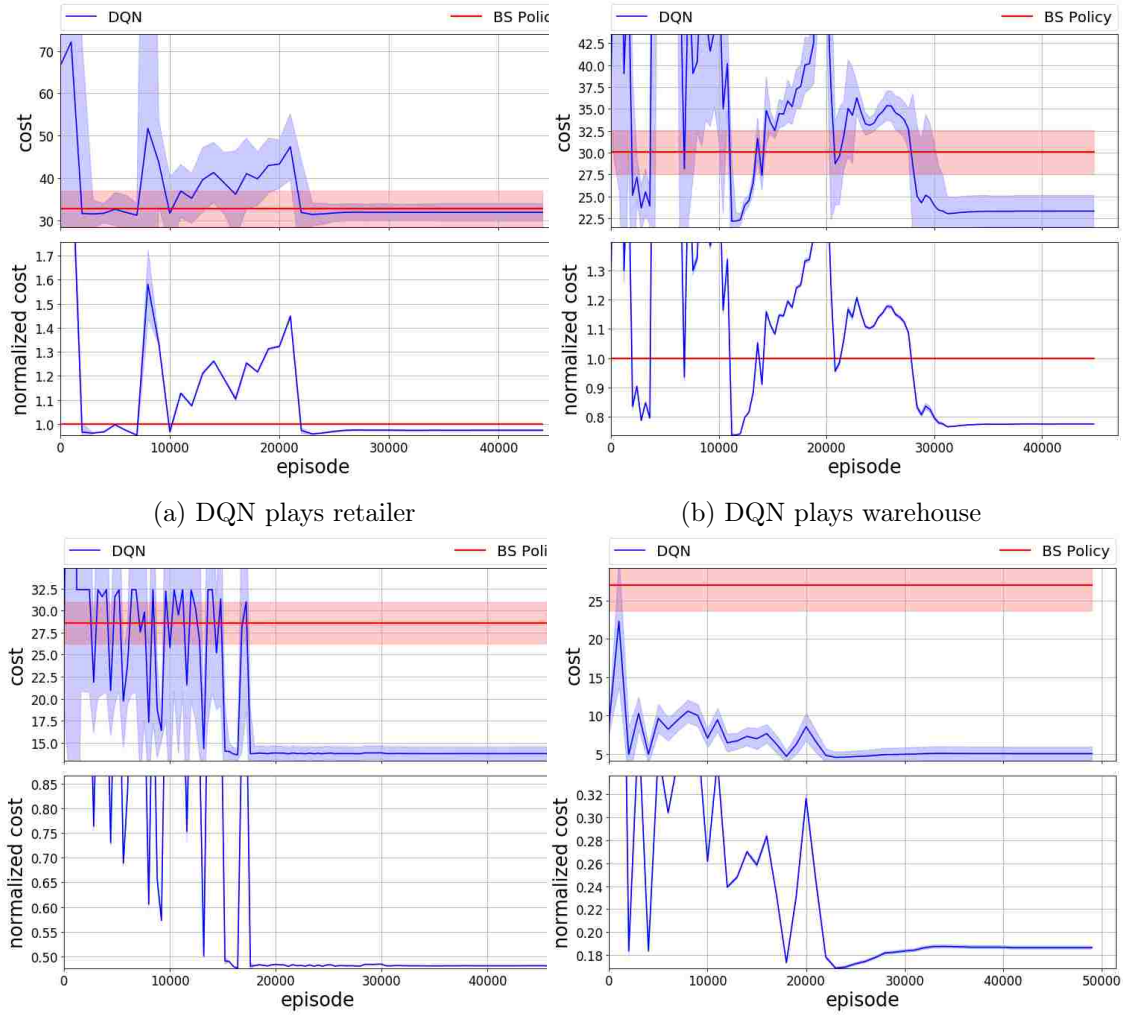
	Cost (DQN, `Strm-BS`)				
DQN Agent	Retailer	Warehouse	Distributer	Manufacturer	Total
Retailer	(0.89, 1.89)	(10.87, 10.83)	(10.96, 10.98)	(12.42, 12.82)	(35.14, 36.52)
Warehouse	(1.74, 9.99)	(0.00, 0.13)	(11.12, 10.80)	(12.86, 12.34)	(25.72, 33.27)
Distributer	(5.60, 10.72)	(0.11, 9.84)	(0.00, 0.14)	(12.53, 12.35)	(18.25, 33.04)
Manufacturer	(4.68, 10.72)	(1.72, 10.60)	(0.24, 10.13)	(0.00, 0.07)	(6.64, 31.52)

used by the other agents. Indeed, the DQN learns how to play to decrease the costs of the other agents, and not just its own costs—for example, the retailer's and warehouse's costs are significantly lower when the distributor uses DQN than they are when the distributor uses a base-stock policy. Similar conclusions can be drawn from Figure 4.6. This shows the power of DQN when it plays with co-player agents that do not play rationally, i.e., do not follow a base-stock policy, which is common in real-world supply chains. Finally, we note that when all agents follow the Sterman formula, the average cost of the agents is [10.81, 10.76, 10.96, 12.6], for a total of 45.13, much higher than when any one agent uses DQN.

Finally, Figure 4.7 shows the game details for the manufacturer when the manufacturer is played by the DQN with $\beta_4 = 100$, when it uses a base-stock policy (`Strm-BS`), and when it uses the Sterman formula (`Strm`); the other three agents all use the Sterman formula. The green trajectory represents the case in which all agents use the Sterman formula. Similar trends are observed when the DQN plays the other three roles; see Appendix K.

4.4.2 Literature Benchmarks

We next test our approach on beer game settings from the literature. These have larger demand-distribution domains, and therefore larger plausible action spaces, and thus represent harder instances to train the DQN for. In all instances in this section, $l^{in} = [2, 2, 2, 2]$ and $l^{tr} = [2, 2, 2, 1]$. Shortage and holding cost coefficients and the base-stock levels for each



(a) DQN plays retailer

(b) DQN plays warehouse

(c) DQN plays distributor

(d) DQN plays manufacturer

Figure 4.6: Total cost (upper figure) and normalized cost (lower figure) with one DQN agent and three agents that follow the Sterman formula

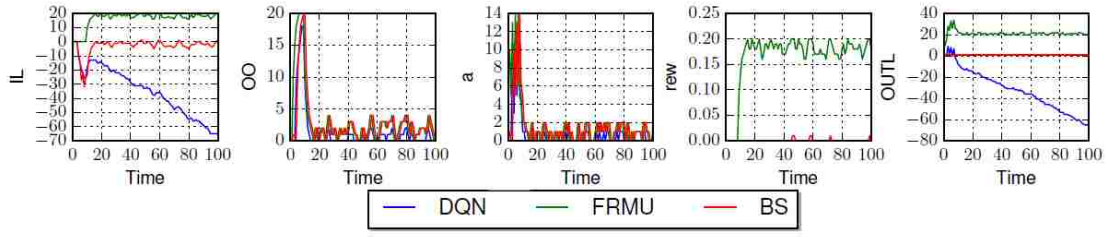


Figure 4.7: IL_t, OO_t, a_t, r_t, and OUTL when DQN plays manufacturer and other agents use Sterman formula

Table 4.2: Cost parameters and base-stock levels for instances with uniform, normal, and classic demand distributions.

demand	c_p	c_h	BS level
$\mathbb{U}[0,8]$	[1.0,1.0,1.0,1.0]	[0.50,0.50,0.50,0.50]	[19,20,20,14]
$\mathbb{N}(10,2^2)$	[10.0,0.0,0.0,0.0]	[1.00,0.75,0.50,0.25]	[48,43,41,30]
$C(4,8)$	[1.0,1.0,1.0,1.0]	[0.50,0.50,0.50,0.50]	[32,32,32,24]

instance are presented in Table 4.2.

Note that the Clark–Scarf algorithm assumes that stage 1 is the only stage with non-zero stockout costs, whereas the $\mathbb{U}[0,8]$ instance has non-zero costs at every stage. Therefore, we used a heuristic approach based on a two-moment approximation, similar to that proposed by Graves [1985], to choose the base-stock levels; see Snyder [2018]. In addition, the $C(4,8)$ demand process is non-stationary—4, then 8—but we allow only stationary base-stock levels. Therefore, we chose to set the base-stock levels equal to the values that would be optimal if the demand were 8 in every period.

Finally, in the experiments in this section, we assume that agent i observes AS_t^i *after* choosing a_t^i, whereas in Section 4.4.1 we assumed the opposite. Therefore, the agents in these experiments have one fewer piece of information when choosing actions, and are therefore more difficult to train.

Tables 4.3, 4.4, and 4.5 show the results of the cases in which the DQN agent plays with co-players who follow base-stock, Sterman, and random policies, respectively. In each group of columns, the first column ("DQN") gives the average cost (over 50 instances) when one agent (indicated by the first column in the table) is played by the DQN and the co-players are played by base-stock (Table 4.3), Sterman (Table 4.4), or random (Table 4.5) agents. The second column in each group ("`BS`", "`Strm-BS`", "`Rand-BS`") gives the corresponding cost when the DQN agent is replaced by a base-stock agent (using the base-stock levels given in Table 4.2) and the co-players remain as in the previous column. The third column ("Gap")

gives the percentage difference between these two costs. For example, in Table 4.4, if the wholesaler is played by the DQN and the co-players are Sterman agents, the average cost is 5.90; the cost increases to 9.53 if the wholesaler is instead played by a base-stock agent and the co-players are still Sterman agents.

As Table 4.3 shows, when the DQN plays with base-stock co-players under uniform or normal demand distributions, it obtains costs that are reasonably close to the case when all players use a base-stock policy, with average gaps of 12.58% and 5.80%, respectively. These gaps are not quite as small as those in Section 4.4.1, due to the larger action spaces in the instances in this section. Since a base-stock policy is optimal at every stage, the small gaps demonstrate that the DQN can learn to play the game well for these demand distributions. For the classic demand process, the percentage gaps are larger. To see why, note that if the demand were to equal 8 in every period, the base-stock levels for the classic demand process will result in ending inventory levels of 0 at every stage. The four initial periods of demand equal to 4 disrupt this effect slightly, but the cost of the optimal base-stock policy for the classic demand process is asymptotically 0 as the time horizon goes to infinity. The absolute gap attained by the DQN is quite small—an average of 0.49 vs. 0.34 for the base-stock cost—but the percentage difference is large simply because the optimal cost is close to 0. Indeed, if we allow the game to run longer, the cost of both algorithms decreases, and so does the absolute gap. For example, when the DQN plays the retailer, after 500 periods the discounted costs are 0.0090 and 0.0062 for DQN and `BS`, respectively, and after 1000 periods, the costs are 0.0001 and 0.0000 (to four-digit precision).

Similar to the results of Section 4.4.1.2, when the DQN plays with co-players who follow the Sterman formula, it performs far better than `Strm-BS`. As Table 4.4 shows, DQN performs 34% better than `Strm-BS` on average. Finally, when DQN plays with co-players

Table 4.3: Results of DQN playing with co-players who follow base-stock policy.

	Uniform			Normal			Classic		
	DQN	BS	Gap (%)	DQN	BS	Gap (%)	DQN	BS	Gap (%)
R	904.88	799.20	13.22	881.66	838.14	5.19	0.50	0.34	45.86
W	960.44	799.20	20.18	932.65	838.14	11.28	0.47	0.34	36.92
D	903.49	799.20	13.05	880.40	838.14	5.04	0.67	0.34	96.36
M	830.16	799.20	3.87	852.33	838.14	1.69	0.30	0.34	-13.13
Average			12.58			5.80			41.50

Table 4.4: Results of DQN playing with co-players who follow Sterman policy.

	Uniform			Normal			Classic		
	DQN	Strm-BS	Gap (%)	DQN	Strm-BS	Gap (%)	DQN	Strm-BS	Gap (%)
R	6.88	8.99	-23.45	9.98	10.67	-6.44	3.80	13.28	-71.41
W	5.90	9.53	-38.10	7.11	10.03	-29.06	2.85	8.17	-65.08
D	8.35	10.99	-23.98	8.49	13.83	-38.65	3.82	20.07	-80.96
M	12.36	13.90	-11.07	13.86	15.37	-9.82	15.80	19.96	-20.82
Average			-24.15			-20.99			-59.57

who use the random policy, for all demand distributions DQN learns very well to play so as to minimize the total cost of the system, and on average obtains 8% better solutions than Rand-BS.

To summarize, DQN does well regardless of the way the other agents play, and regardless of the demand distribution. The DQN agent learns to attain near-BS costs when its co-players follow a BS policy, and when playing with irrational co-players, it achieves a much smaller cost than a base-stock policy would. Thus, when the other agents play irrationally, DQN should be used.

Table 4.5: Results of DQN playing with co-players who follow random policy.

	Uniform			Normal			Classic		
	DQN	Rand-BS	Gap (%)	DQN	Rand-BS	Gap (%)	DQN	Rand-BS	Gap (%)
R	31.39	28.24	11.12	13.03	28.39	-54.10	19.99	25.88	-22.77
W	29.62	28.62	3.49	27.87	35.80	-22.15	23.05	23.44	-1.65
D	30.72	28.64	7.25	34.85	38.79	-10.15	22.81	23.53	-3.04
M	29.03	28.13	3.18	37.68	40.53	-7.02	22.36	22.45	-0.42
Average			6.26			-23.36			-6.97

4.4.3 Faster Training through Transfer Learning

We trained a DQN network with shape $[50, 180, 130, 61, 5]$, $m = 10$, $\beta = 20$, and $C = 10000$ for each agent, with the same holding and stockout costs and action spaces as in section 4.4.1, using 60000 training episodes, and used these as the base networks for our transfer learning experiment. (In transfer learning, all agents should have the same network structure to share the learned network among different agents.) The remaining agents use a `BS` policy.

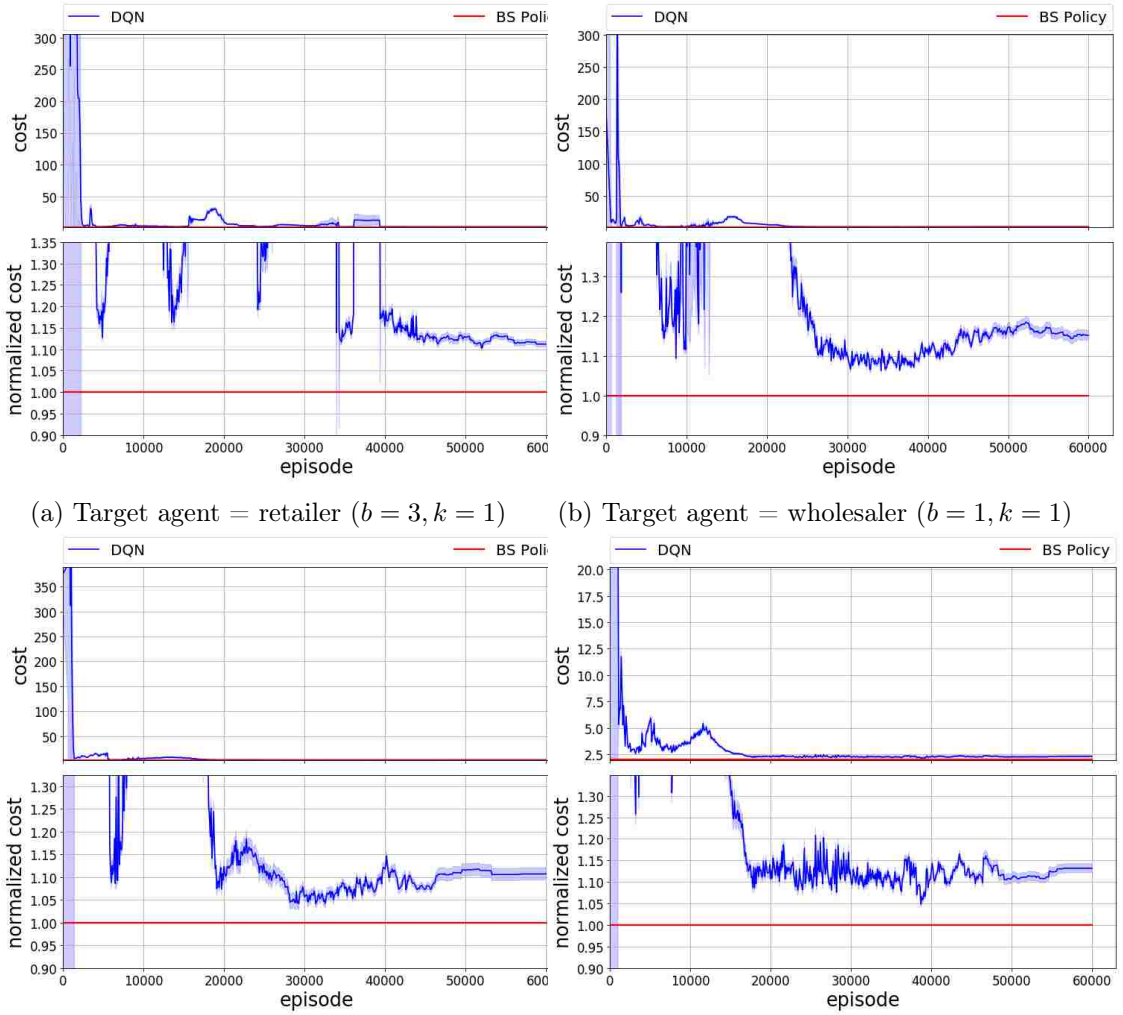
Table 4.6 shows a summary of the results of the six cases discussed in Section 4.3.2 (different agent, same parameters; different agent and costs, same action space; etc.). The first set of columns indicates the holding and shortage cost coefficients, the size of the action space, as well as the demand distribution and the co-players' policy for the base agent (first row) and the target agent (remaining rows). The "Gap" column indicates the average gap between the cost of the resulting DQN and the cost of a `BS` policy; in the first row, it is analogous to the 2.31% average gap reported in Section 4.4.1.1. The average gap is relatively small in all cases, which shows the effectiveness of the transfer learning approach. Moreover, this approach is efficient, as demonstrated in the last column, which reports the average CPU times for one agent. In order to get the base agents, we did hyper-parameter tuning and trained 140 instances to get the best possible set of hyper-parameters, which resulted in a total of 28,390,987 seconds of training. However, using the transfer learning approach, we do not need any hyper-parameter tuning; we only need to check which source agent and which k provides the best results. This requires only 12 instances to train and resulted in an average training time (across cases 1-4) of 1,613,711 seconds—17.6 times faster than training the base agent. Additionally, in case 5, in which a normal distribution is used, full hyper-parameter tuning took 20,396,459 seconds, with an average gap of 4.76%, which means

Table 4.6: Results of transfer learning when π_1 is `BS` and D_1 is $\mathbb{U}[0,2]$

	(Holding, Shortage) Cost Coefficients				$\|A\|$	D_2	π_2	Gap (%)	CPU Time (sec)
	R	W	D	M					
Base agent	(2,2)	(2,0)	(2,0)	(2,0)	5	$\mathbb{U}[0,2]$	`BS`	2.31	28,390,987
Case 1	(2,2)	(2,0)	(2,0)	(2,0)	5	$\mathbb{U}[0,2]$	`BS`	6.06	1,593,455
Case 2	(5,1)	(5,0)	(5,0)	(5,0)	5	$\mathbb{U}[0,2]$	`BS`	6.16	1,757,103
Case 3	(2,2)	(2,0)	(2,0)	(2,0)	11	$\mathbb{U}[0,2]$	`BS`	10.66	1,663,857
Case 4	(10,1)	(10,0)	(10,0)	(10,0)	11	$\mathbb{U}[0,2]$	`BS`	12.58	1,593,455
Case 5	(1,10)	(0.75,0)	(0.5,0)	(0.25,0)	11	$\mathbb{N}(10,2^2)$	`BS`	17.41	1,234,461
Case 6	(1,10)	(0.75,0)	(0.5,0)	(0.25,0)	11	$\mathbb{N}(10,2^2)$	`Strm`	-38.20	1,153,571
Case 6	(1,10)	(0.75,0)	(0.5,0)	(0.25,0)	11	$\mathbb{N}(10,2^2)$	`Rand`	-0.25	1,292,295

transfer learning was 16.6 times faster on average. We did not run the full hyper-parameter tuning for the instances of case-6, but it is similar to that of case-5 and should take similar training time, and as a result a similar improvement from transfer learning. Thus, once we have a trained agent i with a given set P_1^i of parameters, demand D_1 and co-players' policy π_1, we can efficiently train a new agent j with parameters P_2^j, demand D_2 and co-players' policy π_2. (Note that we used different computing clusters to train these cases. In order to compare the training times among different clusters, we trained some instances on all clusters and then obtained conversion coefficients among each pair of clusters. The times reported are normalized so that they can be interpreted as though all instances were trained on the same cluster.)

In order to get more insights about the transfer learning process, Figure 4.8 shows the results of case 4, which is a quite complex transfer learning case that we test for the beer game. The target agents have holding and shortage costs (10,1), (10,0), (10,0), and (10,0) for agents 1 to 4, respectively; and each agent can select any action in $\{-5,\ldots,5\}$. Each caption reports the base agent (shown by `b`) and the value of k used. Compared to the original procedure (see Figure 4.4), i.e., $k = 0$, the training is less noisy and after a few thousand non-fluctuating training episodes, it converges into the final solution. The resulting agents obtain costs that are close to those of `BS`, with a 12.58% average gap compared to



(a) Target agent = retailer ($b = 3, k = 1$) (b) Target agent = wholesaler ($b = 1, k = 1$)

(c) Target agent = distributor ($b = 3, k = 2$) (d) Target agent = manufacturer ($b = 4, k = 2$)

Figure 4.8: Results of transfer learning for case 4 (different agent, cost coefficients, and action space)

the BS cost. (The details of the other cases are provided in Appendix N.1—N.5.

Finally, Table 4.7 explores the effect of k on the tradeoff between training speed and solution accuracy. As k increases, the number of trainable variables decreases and, not surprisingly, the CPU times are smaller but the costs are larger. For example, when $k = 3$, the training time is 46.89% smaller than the training time when $k = 0$, but the solution cost is 17.66% and 0.34% greater than the BS policy, compared to 4.22% and -11.65% for $k = 2$.

To summarize, transferring the acquired knowledge between the agents is very efficient.

Table 4.7: Savings in computation time due to transfer learning. First row provides average training time among all instances. Third row provides average of the best obtained gap in cases for which an optimal solution exists. Fourth row provides average gap among all transfer learning instances, i.e., cases 1–6.

	$k = 0$	$k = 1$	$k = 2$	$k = 3$
Training time	185,679	126,524	118,308	107,711
Decrease in time compared to $k = 0$	—	37.61%	41.66%	46.89%
Average gap in cases 1–4	2.31%	4.39%	4.22%	17.66%
Average gap in cases 1–6	—	-15.95%	-11.65%	0.34%

The target agents achieve costs that are close to those of the `BS` policy (when co-players follow `BS`) and they achieve smaller costs than `Strm-BS` and `Rand-BS`, regardless of the dissimilarities between the source and target agents. The training of the target agents starts from relatively small cost values, the training trajectories are stable and fairly non-noisy, and they quickly converge to a cost value close to that of the `BS` policy or smaller than `Strm-BS` and `Rand-BS`. Even when the action space and costs for the source and target agents are different, transfer learning is still quite effective, resulting in a 12.58% gap compared to the `BS` policy. This is an important result, since it means that if the settings change—either within the beer game or in real supply chain settings—we can train new DQN agents much more quickly than we could if we had to begin each training from scratch.

4.5 Conclusion and Future Work

In this chapter, we consider the beer game, a decentralized, multi-agent, cooperative supply chain problem. A base-stock inventory policy is known to be optimal for special cases, but once some of the agents do not follow a base-stock policy (as is common in real-world supply chains), the optimal policy of the remaining players is unknown. To address this issue, we propose an algorithm based on deep Q-networks. It obtains near-optimal solutions when playing alongside agents who follow a base-stock policy and performs much better

than a base-stock policy when the other agents use a more realistic model of ordering behavior. Furthermore, the algorithm does not require knowledge of the demand probability distribution and uses only historical data.

To reduce the computation time required to train new agents with different cost coefficients or action spaces, we propose a transfer learning method. Training new agents with this approach takes less time since it avoids the need to tune hyper-parameters and has a smaller number of trainable variables. Moreover, it is quite powerful, resulting in beer game costs that are close to those of fully-trained agents while reducing the training time by an order of magnitude.

A natural extension of this work is to apply our algorithm to supply chain networks with other topologies, e.g., distribution networks. Another important extension is to consider a larger state space, which will allow more accurate results. This can be done using approaches such as convolutional neural networks that help to efficiently reduce the size of the input space. Finally, developing algorithms capable of handling continuous action spaces will improve the accuracy of our algorithm.

Bibliography

M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA*, 2016.

H. A. Al-Rawi, M. A. Ng, and K.-L. A. Yau. Application of reinforcement learning to routing in distributed wireless networks: a review. *Artificial Intelligence Review*, 43(3): 381–416, 2015.

Ö. G. Ali and K. Yaman. Selecting rows and columns for training support vector regression models with large retail datasets. *European Journal of Operational Research*, 226(3): 471–480, 2013.

G.-Y. Ban, J. Gallien, and A. Mersereau. Dynamic procurement of new products with covariate information: The residual tree method. *Social Science Research Network, Rochester, NY. URL https://papers. ssrn. com/abstract*, 2926028, 2017.

Y. Bassok, R. Anupindi, and R. Akella. Single-period multiproduct inventory models with substitution. *Operations Research*, 47(4):632–642, 1999.

J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

D. S. Bernstein, R. Givan, N. Immerman, and S. Zilberstein. The complexity of decentralized control of markov decision processes. *Mathematics of operations research*, 27(4):819–840, 2002.

D. Bertsimas and N. Kallus. From predictive to prescriptive analytics. *arXiv preprint arXiv:1402.5481*, 2014.

D. Bertsimas and A. Thiele. A data-driven approach to newsvendor problems. Technical report, Massechusetts Institute of Technology, Cambridge, MA, 2005.

O. Besbes and A. Muharremoglu. On implications of demand censoring in the newsvendor problem. *Management Science*, 59(6):1407–1424, 2013.

S. Bharadwaj, T. W. Gruen, and D. S. Corsten. Retail Out of Stocks: A Worldwide Examination of Extent, Causes, and Consumer Responses. 2002.

L. Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

G. E. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung. *Time series analysis: forecasting and control*. John Wiley & Sons, 2015.

A. N. Burnetas and C. E. Smith. Adaptive ordering and pricing for perishable products. *Operations Research*, 48(3):436–443, 2000.

A. J. Cannon. Quantile regression neural networks: Implementation in R and application to precipitation downscaling. *Computers & geosciences*, 37(9):1277–1284, 2011.

R. Carbonneau, K. Laframboise, and R. Vahidov. Application of machine learning techniques for supply chain demand forecasting. *European Journal of Operational Research*, 184(3): 1140–1154, 2008.

G. Cardoso and F. Gomide. Newspaper demand prediction and replacement model based on fuzzy clustering and rules. *Information Sciences*, 177(21):4799–4809, 2007.

S. K. Chaharsooghi, J. Heydari, and S. H. Zegordi. A reinforcement learning model for supply chain ordering management: An application to the beer game. *Decision Support Systems*, 45(4):949–959, 2008.

F. Chen and R. Samroengraja. The stationary beer game. *Production and Operations Management*, 9(1):19, 2000.

F. Chen and Y. Zheng. Lower bounds for multi-echelon stochastic inventory systems. *Management Science*, 40:1426–1443, 1994.

H.-M. Chi, O. K. Ersoy, H. Moskowitz, and J. Ward. Modeling and optimizing a vendor managed replenishment system using machine learning and genetic algorithms. *European Journal of Operational Research*, 180(1):174–193, 2007.

T.-M. Choi, editor. *Handbook of Newsvendor Problems*. Springer, New York, 2012.

A. J. Clark and H. Scarf. Optimal policies for a multi-echelon inventory problem. *Management science*, 6(4):475–490, 1960.

C. Claus and C. Boutilier. The dynamics of reinforcement learning in cooperative multiagent systems. *AAAI/IAAI*, 1998:746–752, 1998.

J. R. Coakley, J. A. Drexler Jr, E. W. Larson, and A. E. Kircher. Using a computer-based

version of the beer game: Lessons learned. *Journal of Management Education*, 22(3): 416–424, 1998.

R. H. Crites and A. G. Barto. Elevator group control using multiple reinforcement learning agents. *Machine Learning*, 33(2-3):235–262, 1998.

S. F. Crone and N. Kourentzes. Feature selection for time series prediction–a combined filter and wrapper approach for neural networks. *Neurocomputing*, 73(10-12):1923–1936, 2010.

S. F. Crone, M. Hibon, and K. Nikolopoulos. Advances in forecasting with neural networks? empirical evidence from the NN3 competition on time series prediction. *International Journal of Forecasting*, 27(3):635–660, 2011.

R. Croson and K. Donohue. Impact of POS data sharing on supply chain management: An experimental study. *Production and Operations Management*, 12(1):1–11, 2003.

R. Croson and K. Donohue. Behavioral causes of the bullwhip effect and the observed value of inventory information. *Management Science*, 52(3):323–336, 2006a.

R. Croson and K. Donohue. Behavioral causes of the bullwhip effect and the observed value of inventory information. *Management science*, 52(3):323–336, 2006b.

J. M. Day and M. Kumar. Using sms text messaging to create individualized and interactive experiences in large classes: A beer game example. *Decision Sciences Journal of Innovative Education*, 8(1):129–136, 2010.

T. de Bruin, J. Kober, K. Tuyls, and R. Babuška. The importance of experience replay database composition in deep reinforcement learning. In *Deep Reinforcement Learning Workshop, NIPS*, 2015.

L. Deng, J. Li, J.-T. Huang, K. Yao, D. Yu, F. Seide, M. Seltzer, G. Zweig, X. He, J. Williams, et al. Recent advances in deep learning for speech research at Microsoft. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8604–8608. IEEE, 2013.

K. Devika, A. Jafarian, A. Hassanzadeh, and R. Khodaverdi. Optimizing of bullwhip effect and net stock amplification in three-echelon supply chains using evolutionary multi-objective metaheuristics. *Annals of Operations Research*, 242(2):457–487, 2016.

T. Domhan, J. T. Springenberg, and F. Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *IJCAI*, pages 3460–3468, 2015.

F. Edgeworth. The mathematical theory of banking. *Journal of Royal Statistical Society*, 51:113–127, 1888.

T. Efendigil, S. Önüt, and C. Kahraman. A decision support system for demand forecasting with artificial neural networks and neuro-fuzzy models: A comparative analysis. *Expert Systems with Applications*, 36(3):6697–6707, 2009.

K. Eggensperger, M. Feurer, F. Hutter, J. Bergstra, J. Snoek, H. Hoos, and K. Leyton-Brown. Towards an empirical foundation for assessing bayesian optimization of hyperparameters. In *NIPS workshop on Bayesian Optimization in Theory and Practice*, volume 10, 2013.

M. E. El-Telbany. What quantile regression neural networks tell us about prediction of drug activities. In *Computer Engineering Conference (ICENCO), 2014 10th International*, pages 76–80. IEEE, 2014.

A. S. Eruguz, E. Sahin, Z. Jemai, and Y. Dallery. A comprehensive survey of guaranteed-service models for multi-echelon inventory optimization. *International Journal of Production Economics*, 172:110–125, 2016.

C. Finn and S. Levine. Deep visual foresight for planning robot motion. In *Robotics and Automation (ICRA), 2017 IEEE International Conference on*, pages 2786–2793. IEEE, 2017.

J. Foerster, N. Nardelli, G. Farquhar, T. Afouras, P. H. Torr, P. Kohli, and S. Whiteson. Stabilising experience replay for deep multi-agent reinforcement learning. *arXiv preprint arXiv:1702.08887*, 2017.

G. Gallego and P. Zipkin. Stock positioning and performance estimation in serial production-transportation systems. *Manufacturing & Service Operations Management*, 1:77–88, 1999.

J. R. Gardner, M. J. Kusner, Z. E. Xu, K. Q. Weinberger, and J. P. Cunningham. Bayesian optimization with inequality constraints. In *ICML*, pages 937–945, 2014.

I. Gartner. Improving On-Shelf Availability for Retail Supply Chains Requires the Balance of Process and Technology, Gartner Group. `https://www.gartner.com/doc/1701615/improving-onshelf-availability-retail-supply`, 2011. Accessed: 2016-08-04.

S. Geary, S. M. Disney, and D. R. Towill. On bullwhip in supply chains—historical review, present practice and expected future impact. *International Journal of Production Economics*, 101(1):2–18, 2006.

I. Giannoccaro and P. Pontrandolfo. Inventory management in supply chains: A reinforcement

learning approach. *International Journal of Production Economics*, 78(2):153 – 161, 2002. ISSN 0925-5273. doi: http://dx.doi.org/10.1016/S0925-5273(00)00156-0.

I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

J. S. Goodwin and S. G. Franklin. The beer distribution game: using simulation to teach systems thinking. *Journal of Management Development*, 13(8):7–15, 1994.

S. C. Graves. A multi-echelon inventory model for a repairable item with one-for-one replenishment. *Management Science*, 31(10):1247–1256, 1985.

S. C. Graves. Safety stocks in manufacturing systems. *Journal of Manufacturing and Operations Management*, 1:67–101, 1988.

S. C. Graves and S. P. Willems. Optimizing strategic safety stock placement in supply chains. *Manufacturing and Service Operations Management*, 2(1):68–83, 2000.

M. Gruson, J.-F. Cordeau, and R. Jans. The impact of service level constraints in deterministic lot sizing with backlogging. *Omega*, 2017.

G. W. Hadley. Analysis of inventory systems. Technical report, 1963.

R. Hieber and I. Hartel. Impacts of scm order strategies evaluated by simulation-based'beer game'approach: the model, concept, and initial experiences. *Production Planning & Control*, 14(2):122–134, 2003.

F. R. Jacobs. Playing the beer distribution game over the internet. *Production and Operations Management*, 9(1):31, 2000.

Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093*, 2014.

C. Jiang and Z. Sheng. Case-based reinforcement learning for dynamic inventory control in a multi-agent supply-chain system. *Expert Systems with Applications*, 36(3):6520–6526, 2009.

P. Kaminsky and D. Simchi-Levi. A new computerized beer distribution game: Teaching the value of integrated supply chain management. In H. L. Lee and S.-M. Ng, editors, *Global Supply Chain and Technology Management*, volume 1, pages 216–225. POMS Society Series in Technology and Operations Management, 1998a.

P. Kaminsky and D. Simchi-Levi. A new computerized beer game: A tool for teaching the value of integrated supply chain management. *Global supply chain and technology management*, 1(1):216–225, 1998b.

S. O. Kimbrough, D.-J. Wu, and F. Zhong. Computers play the beer game: Can artificial agents manage supply chains? *Decision support systems*, 33(3):323–333, 2002.

D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

M. Ko, A. Tiwari, and J. Mehnen. A review of soft computing applications in supply chain management. *Applied Soft Computing*, 10(3):661–674, 2010.

N. Kourentzes and S. Crone. Advances in forecasting with artificial neural networks. 2010.

M. Längkvist, L. Karlsson, and A. Loutfi. A review of unsupervised feature learning and deep learning for time-series modeling. *Pattern Recognition Letters*, 42:11 – 24,

2014. ISSN 0167-8655. doi: http://dx.doi.org/10.1016/j.patrec.2014.01.008. URL http://www.sciencedirect.com/science/article/pii/S0167865514000221.

Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

H. L. Lee, V. Padmanabhan, and S. Whang. Information distortion in a supply chain: The bullwhip effect. *Management Science*, 43(4):546–558, 1997.

H. L. Lee, V. Padmanabhan, and S. Whang. Comments on "Information distortion in a supply chain: The bullwhip effect". *Management Science*, 50(12S):1887–1893, 2004.

L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *arXiv preprint arXiv:1603.06560*, 2016.

Y. Li. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*, 2017.

Y. Liao, W. Shen, X. Hu, and S. Yang. Optimal responses to stockouts: Lateral transshipment versus emergency order policies. *Omega*, 49:79–92, 2014.

L.-J. Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8(3-4):293–321, 1992.

Z. C. Lipton. The mythos of model interpretability. *arXiv preprint arXiv:1606.03490*, 2016.

C.-J. Lu and C.-C. Chang. A hybrid sales forecasting scheme by combining independent component analysis with K-Means clustering and support vector regression. *The Scientific World Journal*, 2014, 2014.

H. Lu, J. J. Shi, et al. Stockout risk of production-inventory systems with compound poisson demands. *Omega*, 2018.

J. A. Machuca and R. del Pozo Barajas. A computerized network version of the beer game via the internet. *System Dynamics Review*, 13(4):323–340, 1997.

T. L. Magnanti, Z.-J. M. Shen, J. Shu, D. Simchi-Levi, and C.-P. Teo. Inventory placement in acyclic supply chain networks. *Operations Research Letters*, 34:228–238, 2006.

R. Malhotra and D. K. Malhotra. Evaluating consumer loans using neural networks. *Omega*, 31(2):83–96, 2003.

M. K. Martin, C. Gonzalez, and C. Lebiere. Learning to make decisions in dynamic environments: Act-r plays the beer game. 2004.

I. J. Martinez-Moyano, J. Rahn, and R. Spencer. The Beer Game: Its History and Rule Changes. Technical report, University at Albany, 2014.

L. Matignon, G. J. Laurent, and N. Le Fort-Piat. Independent reinforcement learners in cooperative Markov games: A survey regarding coordination problems. *The Knowledge Engineering Review*, 27(01):1–31, 2012.

F. S. Melo and M. I. Ribeiro. Q-learning with linear function approximation. In *International Conference on Computational Learning Theory*, pages 308–322. Springer, 2007.

N. Misiunas, A. Oztekin, Y. Chen, and K. Chandra. Deann: A healthcare analytic methodology of data envelopment analysis and artificial neural networks for the prediction of organ recipient functional status. *Omega*, 58:46–54, 2016.

V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing Atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

D. C. Montgomery, C. L. Jennings, and M. Kulahci. *Introduction to time series analysis and forecasting*. John Wiley & Sons, 2015.

E. Mosekilde and E. R. Larsen. Deterministic chaos in the beer production-distribution model. *System Dynamics Review*, 4(1-2):131–147, 1988.

M. Nagarajan and S. Rajagopalan. Inventory models for substitutable products: optimal policies and heuristics. *Management Science*, 54(8):1453–1466, 2008.

S. Omidshafiei, J. Pazis, C. Amato, J. P. How, and J. Vian. Deep decentralized multi-task multi-agent reinforcement learning under partial observability. *arXiv preprint arXiv:1703.06182*, 2017.

S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010.

C. Paterson, G. Kiesmüller, R. Teunter, and K. Glazebrook. Inventory models with lateral transshipments: A review. *European Journal of Operational Research*, 210(2):125–136, 2011.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Pentaho. Foodmart's database tables. `http://pentaho.dlpage.phi-integration.com/mondrian/mysql-foodmart-database`, 2008. Accessed: 2015-09-30.

E. L. Porteus. *Foundations of Stochastic Inventory Theory*. Stanford University Press, Stanford, CA, 2002.

E. L. Porteus. The newsvendor problem. In D. Chhajed and T. J. Lowe, editors, *Building Intuition: Insights From Basic Operations Management Models and Principles*, chapter 7, pages 115–134. Springer, 2008.

X. Qiu, L. Zhang, Y. Ren, P. N. Suganthan, and G. Amaratunga. Ensemble deep learning for regression and time series forecasting. In *Computational Intelligence in Ensemble Learning (CIEL), 2014 IEEE Symposium on*, pages 1–6, Dec 2014a. doi: 10.1109/CIEL.2014.7015739.

X. Qiu, L. Zhang, Y. Ren, P. N. Suganthan, and G. Amaratunga. Ensemble deep learning for regression and time series forecasting. In *IEEE Symposium Series on Computational Intelligence*, pages 21–26, 2014b.

P. Rajpurkar, J. Irvin, K. Zhu, B. Yang, H. Mehta, T. Duan, D. Ding, A. Bagul, C. Langlotz, K. Shpanskaya, et al. Chexnet: Radiologist-level pneumonia detection on chest x-rays with deep learning. *arXiv preprint arXiv:1711.05225*, 2017.

G. Ravid and S. Rafaeli. Multi player, internet and java-based simulation games: Learning and research in implementing a computerized version of the" beer-distribution supply chain game". *Simulation Series*, 32(2):15–22, 2000.

A. S. Razavian, H. Azizpour, J. Sullivan, and S. Carlsson. CNN features off-the-shelf: An

astounding baseline for recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 806–813, 2014.

K. Rosling. Optimal inventory policies for assembly systems under random demands. *Operations Research*, 37(4):565–579, 1989.

C. Rudin and G.-y. Vahn. The big data newsvendor : practical insights from machine learning analysis. *Cambridge, Mass. : MIT Sloan School of Management*, 2013. URL `http://hdl.handle.net/1721.1/85658`.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61: 85–117, 2015.

S. Seuken and S. Zilberstein. Memory-bounded dynamic programming for DEC-POMDPs. In *IJCAI*, pages 2009–2015, 2007.

K. H. Shang and J.-S. Song. Newsvendor bounds and heuristic for optimal policies in serial supply chains. *Management Science*, 49(5):618–638, 2003.

X. SHI, Z. Chen, H. Wang, D.-Y. Yeung, W.-k. Wong, and W.-c. WOO. Convolutional LSTM network: A machine learning approach for precipitation nowcasting. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 802–810. Curran Associates, Inc., 2015. URL `http://papers.nips.cc/paper/5955-convolutional-lstm-network-a-machine-learning-approach-for-precipitation-nowcastin` `pdf`.

M. Shukla and S. Jharkharia. ARIMA models to forecast demand in fresh supply chains. *International Journal of Operational Research*, 11(1):1–18, 2011.

R. H. Shumway and D. S. Stoffer. *Time series analysis and its applications: with R examples*. Springer Science & Business Media, 2010.

D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

D. Simchi-Levi and Y. Zhao. Performance evaluation of stochastic multi-echelon inventory systems: A survey. *Advances in Operations Research*, 2012, 2011.

A. J. Smola and B. Schölkopf. A tutorial on support vector regression. *Statistics and computing*, 14(3):199–222, 2004.

J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

L. V. Snyder. Multi-echelon base-stock optimization with upstream stockout costs. Technical report, Lehigh University, 2018.

L. V. Snyder and Z.-J. M. Shen. *Fundamentals of Supply Chain Theory*. John Wiley & Sons, 2nd edition, 2019.

J. D. Sterman. Modeling managerial behavior: Misperceptions of feedback in a dynamic decision making experiment. *Management Science*, 35(3):321–339, 1989.

F. Strozzi, J. Bosch, and J. Zaldivar. Beer game order policy optimization under changing customer demand. *Decision Support Systems*, 42(4):2153–2163, 2007.

R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT Press, Cambridge, 1998.

J. W. Taylor. A quantile regression neural network approach to estimating the conditional density of multiperiod returns. *Journal of Forecasting*, 19(4):299–311, 2000.

G. Tesauro, Y. He, and S. Ahmad. Asymptotic convergence of backpropagation. *Neural Computation*, 1(3):382–391, 1989.

N. Turken, Y. Tan, A. J. Vakharia, L. Wang, R. Wang, and A. Yenipazarli. The multi-product newsvendor problem: Review, extensions, and directions for future research. In *Handbook of Newsvendor Problems*, pages 3–39. Springer, 2012.

A. Tversky and D. Kahneman. Judgment under uncertainty: Heuristics and biases. *Science*, 185(4157):1124–1131, 1979.

A. Van Ackere, E. R. Larsen, and J. D. Morecroft. Systems thinking and business process redesign: an application to the beer game. *European management journal*, 11(4):412–423, 1993.

S. Viaene, B. Baesens, T. Van Gestel, J. A. Suykens, D. Van den Poel, J. Vanthienen, B. De Moor, and G. Dedene. Knowledge discovery using least squares support vector machine classifiers: A direct marketing case. In *Principles of Data Mining and Knowledge Discovery*, pages 657–664. Springer, 2000.

A. Vieira. Predicting online user behaviour using deep learning algorithms. *Computing*

Research Repository - arXiv.org, abs/1511.06247, 2015. URL http://arxiv.org/abs/1511.06247.

J.-J. Wang, J.-Z. Wang, Z.-G. Zhang, and S.-P. Guo. Stock index forecasting based on a hybrid model. *Omega*, 40(6):758–766, 2012.

S. Wu and A. Akbarov. Support vector regression for warranty claim forecasting. *European Journal of Operational Research*, 213(1):196–204, 2011.

Q. Xu, X. Liu, C. Jiang, and K. Yu. Quantile autoregression neural network model with applications to evaluating value at risk. *Applied Soft Computing*, 49:1–12, 2016.

P. Xuan, V. Lesser, and S. Zilberstein. Modeling cooperative multiagent problem solving as decentralized decision processes. *Autonomous Agents and Multi-Agent Systems*, 2004.

X. Yu, Z. Qi, and Y. Zhao. Support vector regression for newspaper/magazine sales forecasting. *Procedia Computer Science*, 17:1055–1062, 2013.

Y.-S. Zheng. On properties of stochastic inventory systems. *Management science*, 38(1): 87–103, 1992.

P. H. Zipkin. *Foundations of Inventory Management*. McGraw-Hill, Irwin, 2000.

Appendix A

Proofs of Propositions 1 and 2

These proofs are based on the general idea of the back-propagation algorithm and the way it builds the gradients of the network. For further details, see LeCun et al. [2015]. **Proof of Proposition 1.** To determine the gradient with respect to the weights of the network, we first consider the last layer, L, which in our network contains only one node. Note that in layer L, $y_i^q = a_1^L$. So, we first obtain the gradient with respect to w_{j1}, which connects node j in layer $L-1$ to the single node in layer L, and then recursively calculate the gradient with respect to other nodes in other layers.

First, consider the case of excess inventory ($d_i^q \leq y_i^q$). Recall from (2.12) that $\delta_j^l = \frac{\partial E_i^q}{\partial a_j^l}(g_j^l)'(z_j^l)$. Then $\delta_1^L = c_h(g_1^L)'(z_1^L)$, since $E_i^q = c_h(a_1^L - d_i^q)$. Then:

$$
\begin{aligned}
\frac{\partial E_i^q}{\partial w_{j1}} &= c_h \frac{\partial (y_i^q - d_i^q)}{\partial w_{j1}} \\
&= c_h \frac{\partial a_1^L}{\partial w_{j1}} \quad \text{(since } d_i^q \text{ is independent of } w_{j1}\text{)} \\
&= c_h \frac{\partial g_1^L(z_1^L)}{\partial w_{j1}} \\
&= c_h \frac{\partial g_1^L(z_1^L)}{\partial z_1^L} \frac{\partial z_1^L}{\partial w_{j1}} \quad \text{(by the chain rule)} \\
&= c_h (g_1^L)'(z_1^L) a_j^{L-1} \quad \text{(by (2.11))} \\
&= \delta_1^L(h) a_j^{L-1} \quad \text{(by (2.13))}.
\end{aligned}
\tag{A.1}
$$

Now, consider an arbitrary layer l and the weight w_{jk} that connects node j in layer l and node k in layer $l+1$. Our goal is to derive $\delta_j^l = \frac{\partial E_i^q}{\partial z_j^l}$, from which one can easily obtain $\frac{\partial E_i^q}{\partial w_{jk}}$, since

$$
\frac{\partial E_i^q}{\partial w_{jk}} = \frac{\partial E_i^q}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{jk}} = \delta_j^l a_j^l \tag{A.2}
$$

using similar logic as in (A.1). To do so, we establish the relationship between δ_j^l and δ_k^{l+1}.

$$
\begin{aligned}
\delta_j^l &= \frac{\partial E_i^q}{\partial z_j^l} \\
&= \sum_k \frac{\partial E_i^q}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} \\
&= \sum_k \delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_j^l}
\end{aligned}
\tag{A.3}
$$

Also, from (2.6), we have

$$
z_k^{l+1} = \sum_j w_{jk} a_j^l = \sum_j w_{jk} g_j^l(z_j^l)
$$

Therefore,

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{jk}(g_j^l)'(z_j^l). \tag{A.4}$$

Plugging (A.4) into (A.3), results in (A.5).

$$\delta_j^l = \sum_k w_{jk}\delta_k^{l+1}(g_j^l)'(z_j^l). \tag{A.5}$$

We have now calculated δ_j^l for all $l = 1, \ldots, L$ and $j = 1, \ldots, nn_l$. Then, substituting (A.5) in (A.2), the gradient with respect to any weight of the network is:

$$\frac{\partial E_i^q}{\partial w_{jk}} = a_j^l \sum_k w_{jk}\delta_k^{l+1} g_j'^l(z_j^l). \tag{A.6}$$

Similarly, for the shortage case in layer L, we have:

$$\begin{aligned}
\frac{\partial E_i^q}{\partial w_{j1}} &= -c_p \frac{\partial(d_i^q - y_i^q)}{\partial w_{j1}} \\
&= c_p \frac{\partial(a_1^L)}{\partial w_{j1}} \\
&= c_p \frac{\partial(g_1^L(z_1^L))}{\partial w_{j1}} \\
&= c_p \frac{\partial(g_1^L(z_1^L))}{\partial z_1^L} \frac{\partial(z_1^L)}{\partial w_{j1}} \\
&= c_p a_j^{L-1}(g_1^L)'(z_1^L) \\
&= \delta_1^L(p) a_j^{L-1}.
\end{aligned} \tag{A.7}$$

Using the chain rule and following same procedure as in the case of excess inventory, the gradient with respect to any weight of the network can be obtained. Summing up (A.1),

(A.6) and (A.7), the gradient with respect to the w_{jk} is:

$$\frac{\partial E_i^q}{\partial w_{jk}} = \begin{cases} a_j^l \delta_j^l(p) & \text{if } y_i^q < d_i^q, \\ a_j^l \delta_j^l(h) & \text{if } d_i^q \leq y_i^q. \end{cases}$$

□

Proof of Proposition 2. Consider the proposed revised Euclidean loss function defined in (2.10). Using similar logic as in the proof of Proposition 1, we get that the gradient of the loss function at the single node in layer L is

$$\begin{aligned} \frac{\partial E_i^q}{\partial w_{j1}} &= c_h(y_i^q - d_i^q)\frac{\partial(y_i^q - d_i^q)}{\partial w_{j1}} \\ &= (y_i^q - d_i^q)a_j^{L-1}\delta_1^L(h). \end{aligned} \tag{A.8}$$

in the case of excess inventory and

$$\begin{aligned} \frac{\partial E_i^q}{\partial w_{j1}} &= -c_p(d_i^q - y_i^q)\frac{\partial(d_i^q - y_i^q)}{\partial w_{j1}} \\ &= (d_i^q - y_i^q)a_j^{L-1}\delta_1^L(p). \end{aligned} \tag{A.9}$$

in the shortage case. Again following the same logic as in the proof of Proposition 1, the gradient with respect to any weight of the network can be obtained:

$$\frac{\partial E_i^q}{\partial w_{jk}} = \begin{cases} (d_i^q - y_i^q)a_j^l \delta_1^l(p) & \text{if } y_i^q < d_i^q \\ (y_i^q - d_i^q)a_j^l \delta_1^l(h) & \text{if } d_i^q \leq y_i^q. \end{cases}$$

□

Appendix B

Grid Search for Basket Dataset

In this appendix, we discuss our method for performing a more thorough tuning of the network for DNN-ℓ_2, as discussed in Section 2.4.2. We used a large, two-layer network with 350 and 100 nodes in the first and second layer, respectively. In order to find the best set of parameters for this model, a grid search is used. We considered three parameters, lr, λ, and γ; λ is the regularization coefficient, and lr and γ are parameters used to set the learning rate. In particular, we set lr_t, the learning rate used in iteration t, using the following formula:

$$lr_t = lr \times (1 + \gamma \times t)^{-0.75}.$$

We considered parameter values from the following sets:

$$\begin{aligned}
\gamma \in &\quad \{0.01, 0.005, 0.001, 0.0001, 0.0005, 0.00005\} \\
\lambda \in &\quad \{0.01, 0.005, 0.001, 0.0005, 0.0001, 0.00005\} \\
lr \in &\quad \{0.001, 0.005, 0.0005, 0.0001, 0.00005, 0.00003, 0.00001, 0.000009, 0.000008, 0.000005\},
\end{aligned}$$

The best set of parameters among these 360 sets were $\gamma = 0.00005, \lambda = 0.00005$, and $lr = 0.000009$. These parameters were used to test integer values of $c_p/c_h \in \{3, \ldots, 9\}$ in Figure 2.3, for the series labeled DNN-ℓ_2-T.

Appendix C

A Tuning-Free Neural Network

To tune the hyper-parameters of the DNN in Section 2.4, we used an extension of the random search algorithm [Bergstra and Bengio, 2012] called HyperBand [Li et al., 2016]—in particular, to determine the network structure, learning rate, and regularization coefficient. However, a user of our model might not have the time, resources, or expertise to follow a similar procedure. Even cheaper procedures like Bayesian optimization [Snoek et al., 2012, Gardner et al., 2014] are still too time consuming and too complex to implement. To address this issue, in this section we propose a computationally cheap approach to set up a network structure without extensive tuning. Our approach provides quite good results on a wide range of problem parameters.

The network structure should have a direct relation with the number of training samples n, the number of features p, and the range r_i that feature f_i, $i = 1, \ldots, p$, can take values from. For example, a feature f_i which represents the day of week takes values between 1 and 7, and the one-hot-encoded version is a categorical feature with 7 categories; so, $r_i = 7$. For a continuous feature like the sales quantity, r_i may be an interval such as $[0, \infty]$. These characteristics—the number of features and the range of values for each feature—affect both

the number of layers in the network and the numbers of nodes in each layer. For instance, if the number of features is small and the features take on only a few values, a trained DNN returns a solution that minimizes the average loss value. In this case a small network can provide quite good results. On the other hand, when the number of features is relatively large and each feature can take values from a large range or set, the DNN must be able to distinguish among a large number of cases. In this event, the DNN network must be relatively large.

Now, consider the newsvendor problem with p features. In the datasets that we considered, the features are quite simple, e.g., receipt date and item category. However, we wish to propose a general structure for prospective users of our model, so we assume one may use more complex features, either categorical or continuous. (However, we assume the input cannot be an image, so we do not need a convolutional network Goodfellow et al. [2016].) Thus, we propose a three-layer network in which the number of nodes in the first, second, and third hidden layers equals aq, bq, and cq, respectively, where a, b, and c are constants (by default we use $a = 1.5$, $b = 1$, and $c = 0.5$), and where q is defined as follows. Let q_v be the number of continuous features, let $P_c \subseteq \{1, \ldots, p\}$ be the set of categorical features, and let

$$q_u = \min \left\{ \sum_{i \in P_c} r_i, \prod_{i \in P_c} r_i \right\}.$$

In words, q_u is the smallest number that can represent all combinations of categories. Let $q = q_u + q_v$. Finally, the number of input nodes also equals q, and the output layer includes a single node.

Using this approach, if the number of features is small, the number of DNN weights to optimize is small, and if the number of features is large, the number of weights is large.

Using the default values of a, b, and c, the proposed network has $m = \frac{1}{2}q(7q+1)$ weights, which should be smaller than the number of training records. If $m > n$, there is a chance of over-fitting, and if $m \gg n$, the DNN over-fits the training data with high probability, in which case the number of DNN variables must be reduced. In this case one should select smaller values of the coefficients a, b, and c to reduce the number of nodes in each layer. Finally, using the default coefficient values, the resulting network has size $[q, 1.5q, q, 0.5q, 1]$, so the number of nodes in the first hidden layer is larger than the number of features, and with a high probability the DNN is able to capture the information of the features and transfer them through the network. Setting the number of nodes in the first hidden layer smaller than that in the input layer may result in losing some input information.

We continue training until we meet one of the following criteria:

- the point-wise improvement in loss function value is less than 0.01%, or
- the number of passes over the training data reaches `MaxEpoch`.

(We set `MaxEpoch`=100.)

Of course, we cannot guarantee that this approach will produce an optimal network structure, but it eliminates the work of determining the structure, and our experiments suggest that it performs well. We also note that one still must follow an approach to determine a suitable learning rate and regularization parameter (see Snoek et al. [2012], Eggensperger et al. [2013], Domhan et al. [2015], Bergstra and Bengio [2012]).

In order to see how well the fixed-size network works, we ran the same experiments as in Section 2.4.3. In these tests, we fixed the network structure to $[q, 1.5q, q, 0.5q, 1]$ with learning rate = 0.001 and $\lambda = 0.005$ for all demand distributions. In all cases except normally distributed demand, the network obtained near-optimal costs after at most 10

Table C.1: Results of 100 and 200 training epochs.

	100 epochs				300 epochs			
clusters	1	10	100	200	1	10	100	200
normal	0.000	0.004	2.006	3.083	0.000	0.004	0.005	3.083
lognormal	0.003	0.006	0.129	0.006	0.000	0.004	0.126	0.011
uniform	0.001	0.012	0.020	0.134	0.000	0.001	0.020	-0.004
beta	0.029	0.003	0.014	0.023	0.027	-0.006	0.007	0.021
exponential	0.000	0.071	0.008	0.019	0.000	0.001	0.006	0.018
average	0.0067	0.0192	0.4353	0.6531	0.0054	0.0008	0.0329	0.6260

epochs (which, on average, took 10 minutes to train), when improvement stopped. For normally distributed demands, the algorithm ran for at least 50 epochs to get a converged network. Table C.1 shows the results of the test datasets for all demand distributions, in which we provide the gap between the results of the fixed network and the results from the HyperBand algorithm. As provided in the table, when we train for 100 epochs, the fixed network obtains costs that are very close to those obtained using the HyperBand algorithm. For 1, 10, 100, and 200 clusters, it obtains average gaps of 0.67%, 1.9%, 43.5%, and 65.3% compared to the results of networks obtained by HyperBand algorithm.

In order to see the effect of training length, we ran all experiments for 300 epochs to see whether the solutions improve; these are provided in the right side of Table C.1. The average gaps decreased to 0.5%, 0.08%, 3.29%, and 62.6% for 1, 10, 100, and 200 clusters, respectively. Therefore, running the DNN for longer training periods can help to get smaller cost values.

In sum, setting the network size using this approach is much cheaper than any extension of random search or Bayesian optimization, and it can provide near-optimal results for the newsvendor problem when there is a sufficiently large number of historical records. (In our experiment, this corresponds to having fewer clusters.) When there is insufficient historical data available, additional tuning and/or training is required in order to obtain good results.

Appendix D

Stock-Out Prediction for Single-Stage Supply Chain Network

Consider a single-stage supply chain network. The goal is to obtain the stock-out probability and as a result make a stock-out prediction, i.e., we want to obtain the probability:

$$P(IL_t < 0),$$

where IL_t is the ending inventory level in period t. Classical inventory theory (see, e.g., Snyder and Shen [2019], Zipkin [2000]) tells us that

$$IL_t = IP_{t-L} - D_L,$$

where L is the lead time, IP_{t-L} is the inventory position (inventory level plus on-order inventory) after placing a replenishment order in period $t - L$, and D_L is the lead-time demand. Since we know IP_{t-L} and we know the probability distribution of D_L, we can

determine the probability distribution of IL_t and use this to calculate $P(IL_t < 0)$. Then we can predict a stock-out if this probability is larger than α, for some desired threshold α.

Appendix E

Gradient of Weighted Soft-max Function

Let

$$p_j = \frac{e^{z_j}}{\sum_{u=1}^{U} e^{z_u}}.$$

Then the gradient of the soft-max loss function (3.4) is:

$$\frac{\partial E}{\partial z_j} = p_j - y_j$$

and the gradient of weighted soft-max loss function (3.7) is:

$$\frac{\partial E_w}{\partial z_j} = w_j(p_j - y_j).$$

Appendix F

Activation and Loss Functions

The most common loss functions are the hinge (F.1), logistic regression (F.2), and Euclidean (F.3) loss functions, given (respectively) by:

$$E = \max(0, 1 - y_i \hat{y}_i) \tag{F.1}$$

$$E = -\log(1 + e^{y_i \hat{y}_i}) \tag{F.2}$$

$$E = ||y_i - \hat{y}_i||_2^2, \tag{F.3}$$

where y_i is the observed value of sample i, and $\hat{y}_i$ is the output of the DNN. The hinge loss function is appropriate for $0, 1$ classification. The logistic loss function is also used for $0, 1$ classification; however, it is a convex function, which is easier to optimize than the hinge function. The Euclidean loss function minimizes the difference between the observed and calculated values and penalizes closer predictions much less than farther predictions.

Each node of the DNN network has an activation function. The most commonly used activation functions are sigmoid, tanh, and inner product, given (respectively) by:

$$\text{Sigmoid}(z) = \frac{1}{1 + e^{1+z}} \tag{F.4}$$

$$\text{Tanh}(z) = \frac{2e^z - 1}{2e^z + 1} \tag{F.5}$$

$$\text{InnerProduct}(z) = z \tag{F.6}$$

Appendix G

Loss vs. Accuracy

The loss function alone cannot be used to measure the quality of prediction; we need an additional criterion. The reason for this is that the loss function does not measure the same thing as the desired objective function for the problem. For example, in our problem the objective is to make accurate predictions, but the loss function instead measures the distance between 1 and the probability $P(z^u)$ assigned to the correct label u. These two quantities tend to move in the same direction—when the DNN assigns probabilities close to 1 for the correct label u, the model as a whole also tends to make accurate predictions—but they are not equivalent to each other.

In fact, it is possible that the loss function and the accuracy can move in opposite directions. For example, assume we are trying to predict stock-outs for a single node, and we have three samples, each consisting of a set of feature values. The true label is 1 for each of the three samples. Table G.1 provides the DNN output values z^{u_0} and z^{u_1} for two different hypothetical DNN networks. The table also gives the probability $P(z^{u_1})$ that the model assigns to the label being 1, the resulting prediction, and the resulting loss value, using the softmax function. For example, in case 1, sample 1 has

output values of $(z^{u_0}, z^{u_1}) = (1.10, 1.00)$; from (3.3), we get $P(z^{u_1}) = 0.5249$. Since $P(z^{u_1}) > P(z^{u_0}) = 1 - P(z^{u_1})$, we assign a prediction of $y = 1$, which results in a loss function value of 0.28, from (3.4). The average loss value for the three samples is 0.29 for DNN network 1 and 0.20 for network 2. On the other hand, the predictions made by DNN network 1 are more accurate than those made by network 2 (2/3 vs. 1/3). Therefore, the network with the larger loss value actually has the better accuracy.

This is not the typical situation—usually a larger loss value indicates a worse accuracy. But the fact that this can happen argues for the use of another measure to evaluate the performance of the model. A second reason is that the loss function values reported in the table (0.28, etc.) are much less meaningful to a decision maker than accuracy values (e.g., 1/3). For these reasons, we use the accuracy to measure the quality of the predictions, rather than the loss value.

Table G.1: Comparison of loss and accuracy for two DNN networks.

		(z^{u_0}, z^{u_1})	$P(z^{u_1})$	Prediction	Loss
DNN Network 1	Sample 1	(1.10, 1.00)	0.5249	1	0.28
	Sample 2	(1.30,1.20)	0.5249	1	0.28
	Sample 3	(0.09, 0.10)	0.4975	0	0.30
	Average				0.29
DNN Network 2	Sample 1	(1.00, 1.01)	0.4975	0	0.30
	Sample 2	(1.00, 2.01)	0.4975	0	0.30
	Sample 3	(12.00,4.00)	0.9996	1	0.00
	Average				0.20

Appendix H

Dependent Demand Data Generation

This section provides the details of data generation for dependent demands. In the case of dependent demand, there are seven items, and the demand mean of each item is different on different days of the week. Tables H.1 and H.2 provide the mean (μ) and standard deviation (σ) of the normal distribution of for each item in each day of week.

Table H.1: Mean demand (μ) of each item on each day of the week.

Item	Mon	Tue	Wen	Thu	Fri	Sat	Sun
1	12	10	9	11	14	9	11
2	14	12	11	9	16	7	9
3	8	7	6	14	10	13	14
4	7	6	5	15	9	14	15
5	6	5	4	16	7	15	16
6	8	7	6	14	10	13	13
7	10	9	8	12	12	11	12

Table H.2: Standard deviation (σ) of each item on each day of the week.

Item	Mon	Tue	Wen	Thu	Fri	Sat	Sun
1	3	2	4	1	2	3	2
2	4	3	4	1	3	2	1
3	1	1	2	2	2	4	3
4	1	1	1	3	1	4	3
5	1	1	1	2	1	3	3
6	2	1	1	3	1	3	3
7	3	2	4	1	2	3	2

Appendix I

Experimental Details

This section provides additional details of the experiments. In order to compare the WDNN with the naive algorithms, we do not have a one-to-one mapping between α, which controls the false positive versus false negative errors in the Naive-1, Naive-2, and Naive-3 algorithms, and (c_p, c_n), which do the same in WDNN, Naive-4, and Naive-5 algorithms. So, to make apples-to-apples comparisons, we selected some c_p and c_n values that result in similar numbers of false positive errors as in the naive algorithms. The values that we selected are: $c_p = 1, c_n \in \{0.3, 0.35, 0.4, 0.45, 0.5, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 1.15, 1.2, 1.25, 1.3, 1.35, 1.4, 1.45, 1.5, 2.25, 2.5, 2.75, 3, 3.25, 3.5, 3.75, 4, 4.5, 4.75, 5, 5.25, 5.5, 5.75, 6, 6.25, 6.5, 6.75, 7, 7.25, 7.5, 7.75, 8, 8.5, 9, 9.5, 10, 10.5, 11, 11.5, 11.75, 12, 12.5, 13, 13.5, 14, 14.5, 15\}$ and $c_n = 1, c_p \in \{0.3, 0.35, 0.4, 0.45, 0.5, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 1.15, 1.2, 1.25, 1.3, 1.35, 1.4, 1.45, 1.5, 2.25, 2.5, 2.75, 3, 3.25, 3.5, 3.75, 4, 4.5, 4.75, 5, 5.25, 5.5, 5.75, 6, 6.25, 6.5, 6.75, 7, 7.25, 7.5, 7.75, 8, 8.5, 9, 9.5, 10, 10.5, 11, 11.5, 11.75, 12, 12.5, 13, 13.5, 14, 14.5, 15\}$, which results in 118 cases.

For each combination of c_p and c_n, we trained DNN networks for each of the five supply chain networks. Each DNN was trained until it reached `MaxEpoch=3` or the loss value was

Table I.1: Average training time (in seconds) for each supply network.

network	serial	distribution	OWMR	complex-I	complex-II
time	30319	132039	113524	124025	98392

smaller than 10^{-6}. Training for the serial supply chain terminated after one epoch, while the others ran for three epochs. Table I.1 shows the average training time (in seconds) for each supply chain network.

Also, to train the linear regression, each (c_p, c_n) took approximately 550 seconds. Other naive algorithms run for less then a second for each (c_p, c_n) or α. Additionally, the inference time of all algorithms is less than one second.

Appendix J

Results of Threshold-Prediction Case

This section provides the accuracy results for the problem described in Section 3.4.7.1, in which we wish to predict whether the inventory level will fall below 10. Figures J.1–J.5 show the results for the serial, OWMR, distribution, complex I, and complex II networks, respectively.

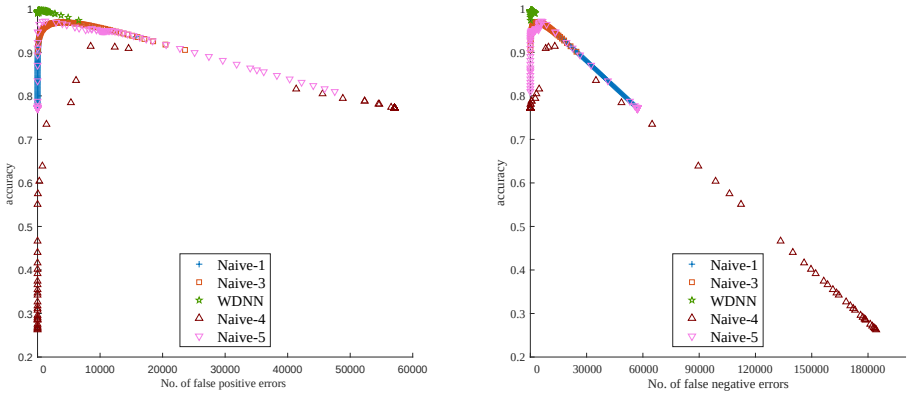


Figure J.1: Accuracy of each algorithm for serial network

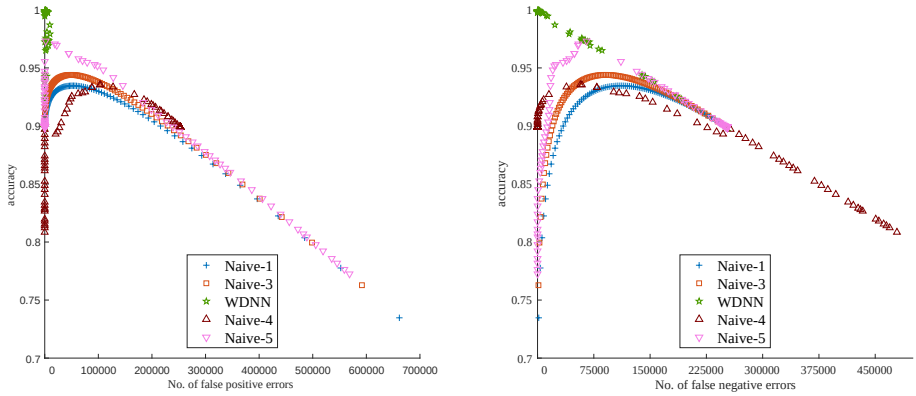


Figure J.2: Accuracy of each algorithm for OWMR network

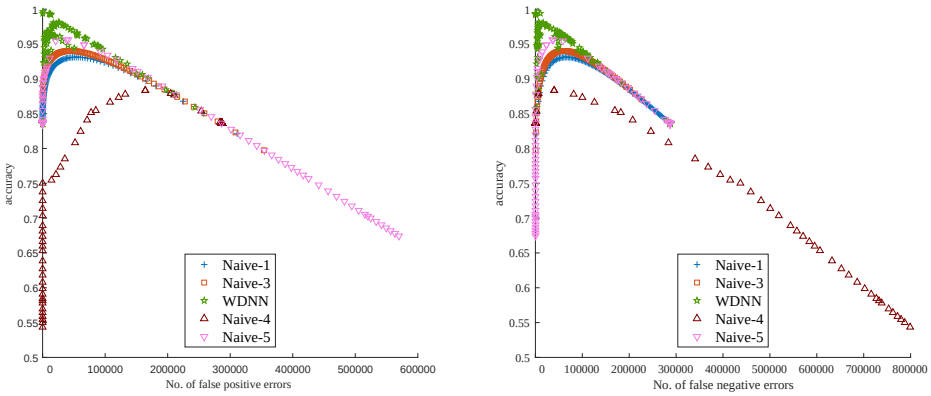


Figure J.3: Accuracy of each algorithm for distribution network

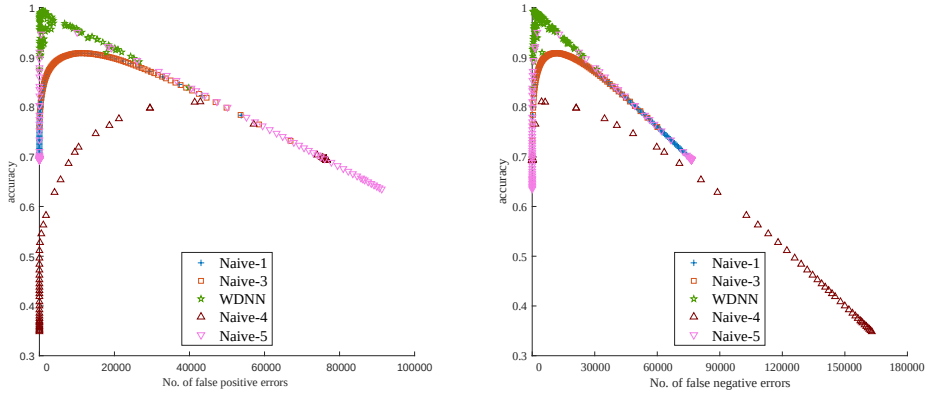


Figure J.4: Accuracy of each algorithm for complex network I

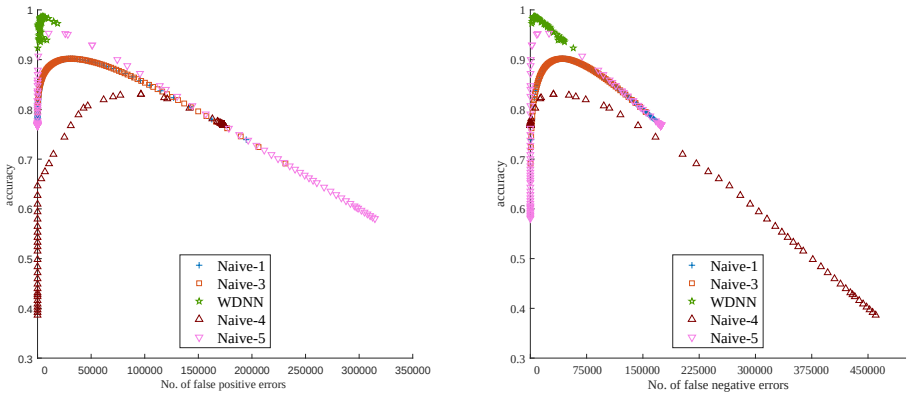


Figure J.5: Accuracy of each algorithm for complex network II

Appendix K

Extended Numerical Results

This appendix shows additional results on the details of play of each agent. Figure K.1 provides the details of IL, OO, a, r, and OUTL for each agent when the DQN retailer plays with co-players who use the `BS` policy. Clearly, DQN attains a similar IL, OO, action, and reward to those of `BS`. Figure K.2 provides analogous results for the case in which the DQN manufacturer plays with three `Strm` agents. The DQN agent learns that the shortage costs of the non-retailer agents are zero and exploits that fact to reduce the total cost. In each of the figures, the top set of charts provides the results of the retailer, followed by the warehouse, distributor, and manufacturer.

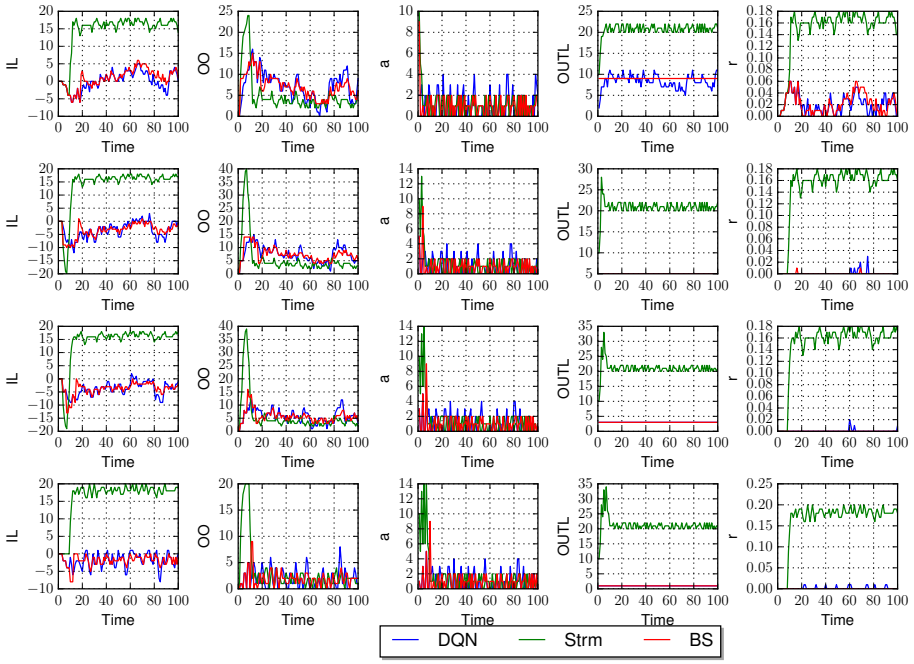


Figure K.1: IL_t, OO_t, a_t, and r_t of all agents when DQN retailer plays with three `BS` co-players

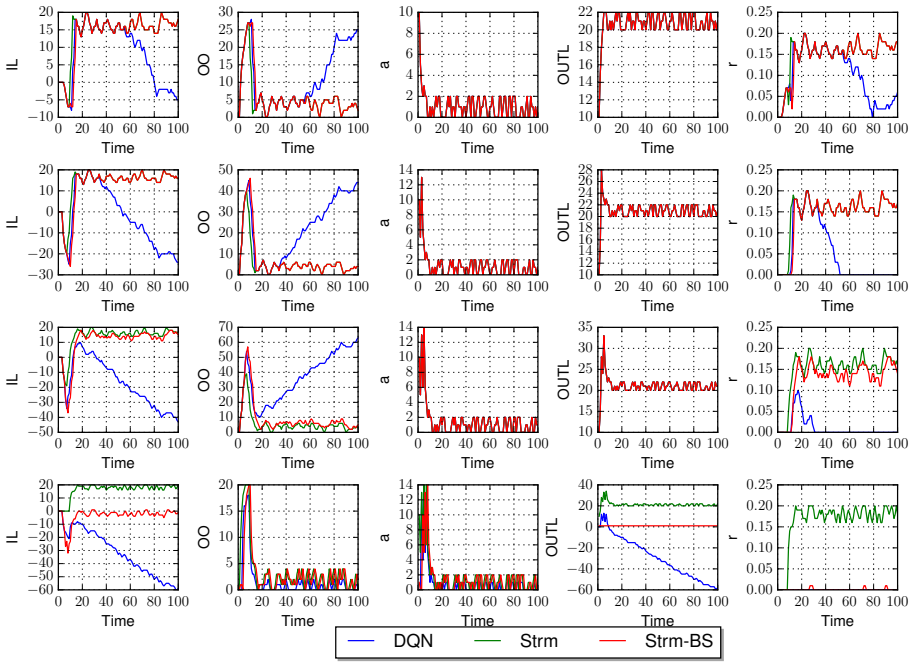


Figure K.2: IL_t, OO_t, a_t, and r_t of all agents when DQN manufacturer plays with three `Strm-BS` co-players

Appendix L

Sterman Formula Parameters

The computational experiments that use `Strm` agents calculate the order quantity using formula (L.1), adapted from Sterman [1989]:

$$q_t^i = \max\{0, AO_{t+1}^{i-1} + \alpha^i(IL_t^i - a^i) + \beta^i(OO_t^i - b^i)\}, \tag{L.1}$$

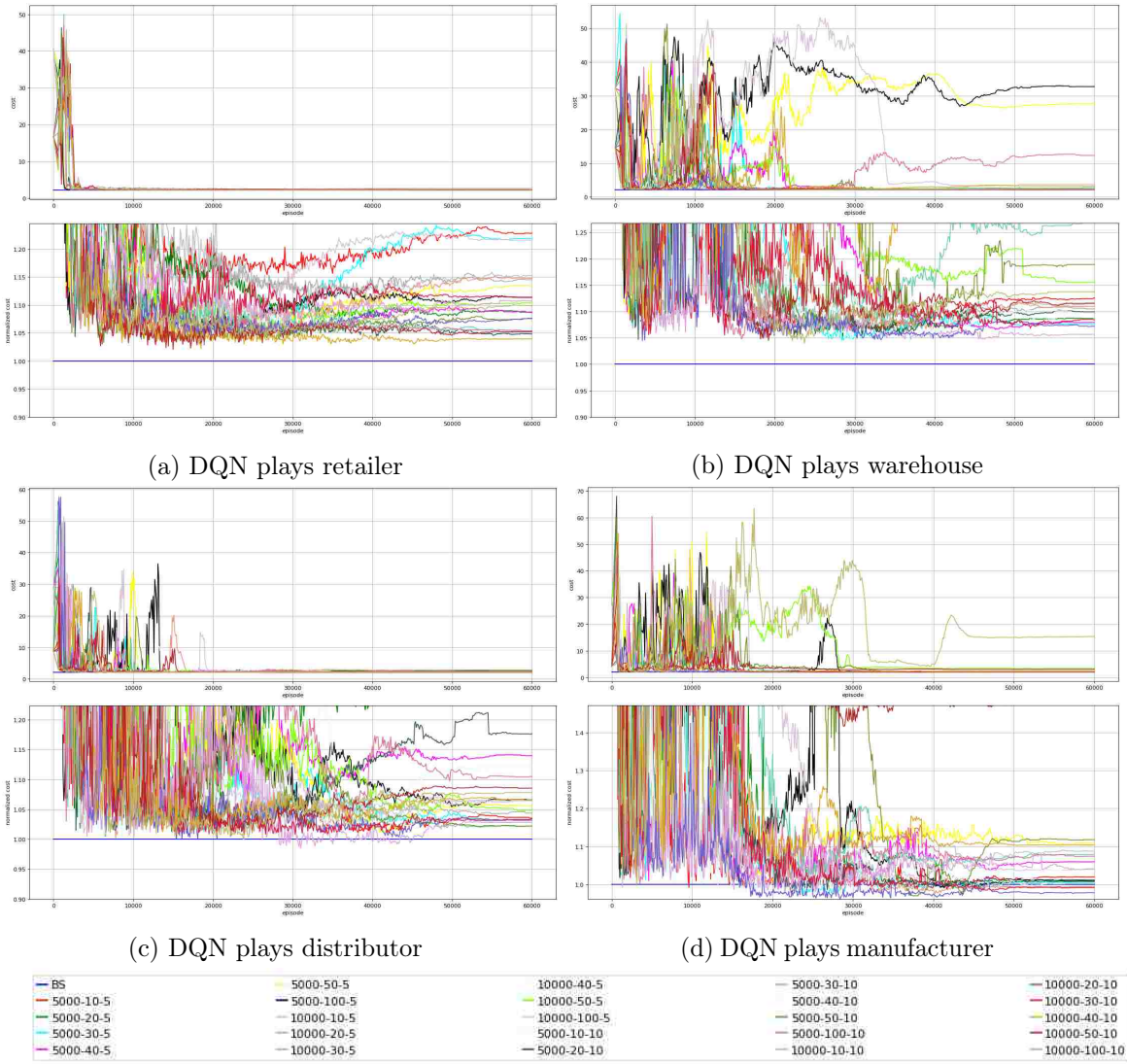
where α^i, a^i, β^i, and b^i are the parameters corresponding to the inventory level and on-order quantity. The idea is that the agent sets the order quantity equal to the demand forecast plus two terms that represent adjustments that the agent makes based on the deviations between its current inventory level (resp., on-order quantity) and a target value a^i (resp., b^i). We set $a^i = \mu_d$, where μ_d is the average demand; $b^i = \mu_d(l_i^{fi} + l_i^{tr})$; $\alpha^i = -0.5$; and $\beta^i = -0.2$ for all agents $i = 1, 2, 3, 4$. The negative α and β mean that the player over-orders when the inventory level or on-order quantity fall below the target value a_i or b_i.

Appendix M

The Effect of β on the Performance of Each Agent

Figure M.1 plots the training trajectories for DQN agents playing with three `BS` agents using various values of C, m, and β. In each sub-figure, the blue line denotes the result when all players use a `BS` policy while the remaining curves each represent the agent using DQN with different values of C, β, and m, trained for 60000 episodes with a learning rate of 0.00025.

As shown in Figure M.1a, when the DQN plays the retailer, $\beta_1 \in \{20, 40\}$ works well, and $\beta_1 = 40$ provides the best results. As we move upstream in the supply chain (warehouse, then distributor, then manufacturer), smaller β values become more effective (see Figures M.1b–M.1d). Recall that the retailer bears the largest share of the optimal expected cost per period, and as a result it needs a larger β than the other agents. Not surprisingly, larger m values attain better costs since the DQN has more knowledge of the environment. Finally, larger C works better and provides a stable DQN model. However, there are some combinations for which smaller C and m also work well, e.g., see Figure M.1d, trajectory



(a) DQN plays retailer

(b) DQN plays warehouse

(c) DQN plays distributor

(d) DQN plays manufacturer

Figure M.1: Total cost (upper figure) and normalized cost (lower figure) with one DQN agent and three agents that follow base-stock policy

5000-20-5.

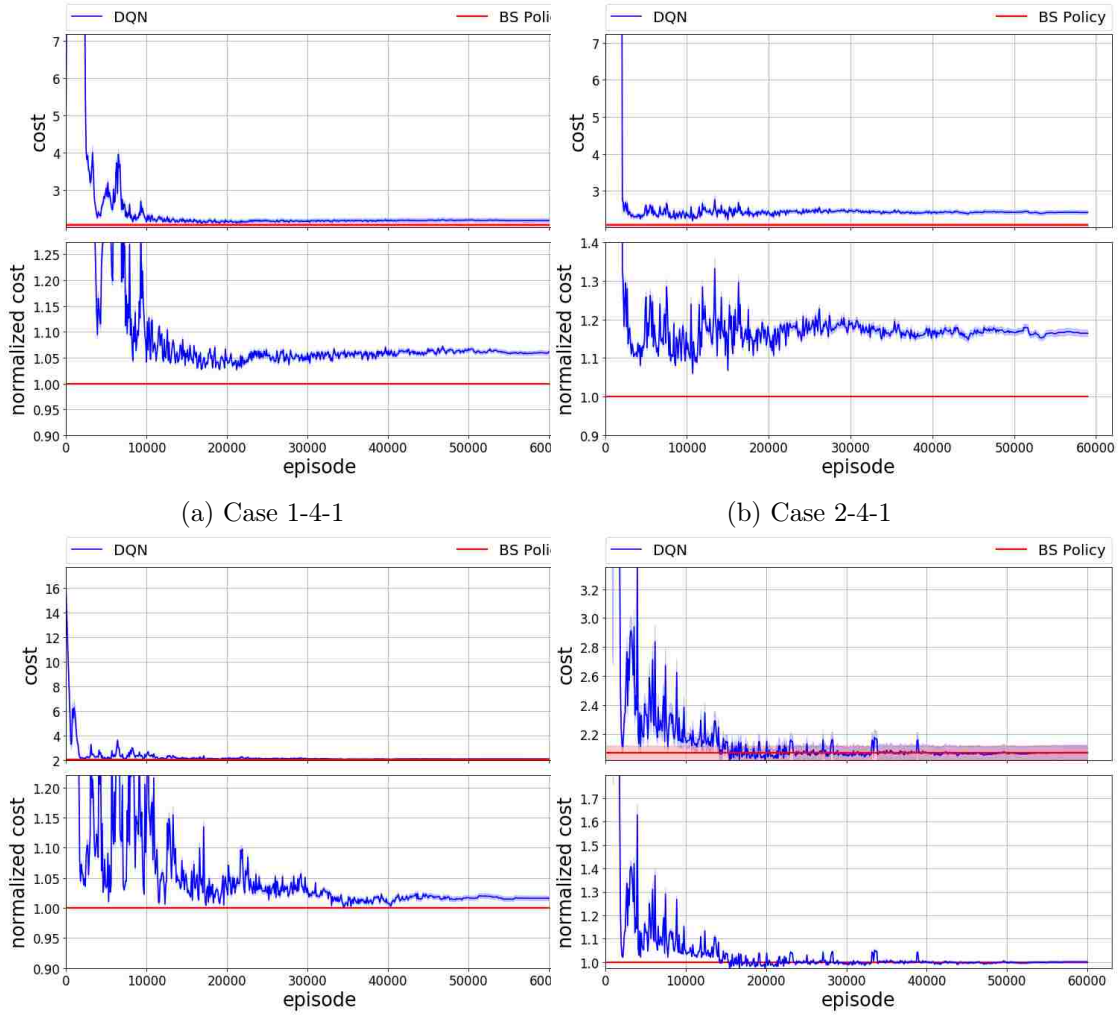
Appendix N

Extended Results on Transfer Learning

N.1 Transfer Knowledge Between Agents

In this section, we present the results of the transfer learning method when the trained agent $i \in \{1, 2, 3, 4\}$ transfers its first $k \in \{1, 2, 3\}$ layer(s) into co-player agent $j \in \{1, 2, 3, 4\}$, $j \neq i$. For each target-agent j, Figure N.1 shows the results for the best source-agent i and the number of shared layers k, out of the 9 possible choices for i and k. In the sub-figure captions, the notation j-i-k indicates that source-agent i shares weights of the first k layers with target-agent j, so that those k layers remain non-trainable.

Except for agent 2, all agents obtain costs that are very close to those of the `BS` policy, with a 6.06% gap, on average. (In Section 4.4.1.1, the average gap is 2.31%.) However, none of the agents was a good source for agent 2. It seems that the acquired knowledge of other agents is not enough to get a good solution for this agent, or the feature space that agent 2 explores is different from other agents, so that it cannot get a solution whose cost is close to



(a) Case 1-4-1

(b) Case 2-4-1

(c) Case 3-1-1

(d) Case 4-2-1

Figure N.1: Results of transfer learning between agents with the same cost coefficients and action space

the BS cost.

In order to get more insight, consider Figure 4.4, which presents the best results obtained through hyper-parameter tuning for each agent. In that figure, all agents start the training with a large cost value, and after 25000 fluctuating iterations, each converges to a stable solution. In contrast, in Figure N.1, each agent starts from a relatively small cost value, and after a few thousand training episodes converges to the final solution. Moreover, for agent 3, the final cost of the transfer learning solution is smaller than that obtained by training the

network from scratch. And, the transfer learning method used one order of magnitude less CPU time than the approach in Section 4.4.1.1 to obtain very close results.

We also observe that agent j can obtain good results when $k = 1$ and i is either $j - 1$ or $j + 1$. This shows that the learned weights of the first DQN network layer are general enough to transfer knowledge to the other agents, and also that the learned knowledge of neighboring agents is similar. Also, for any agent j, agent $i = 1$ provides similar results to that of agent $i = j - 1$ or $i = j + 1$ does, and in some cases it provides slightly smaller costs, which shows that agent 1 captures general feature values better than the others.

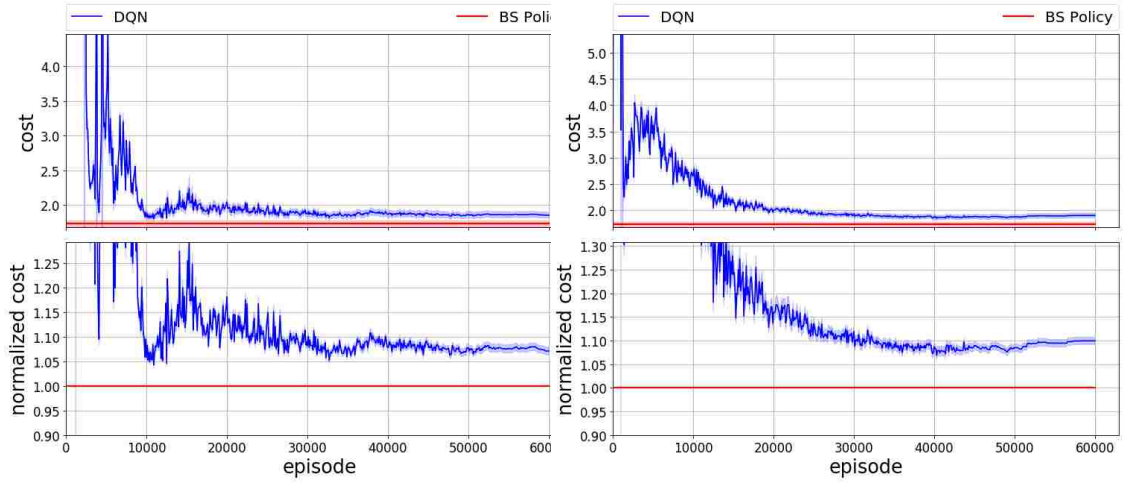
N.2 Transfer Knowledge for Different Cost Coefficients

Figure N.2 shows the best results achieved for all agents, when agent j has different cost coefficients, $(c_{p_2}, c_{h_2}) \neq (c_{p_1}, c_{h_1})$. We test target agents $j \in \{1, 2, 3, 4\}$, such that the holding and shortage costs are (5,1), (5,0), (5,0), and (5,0) for agents 1 to 4, respectively. In all of these tests, the source and target agents have the same action spaces. All agents attain cost values close to the `BS` cost; in fact, the overall average cost is 6.16% higher than the `BS` cost.

In addition, similar to the results of Section N.1, base agent $i = 1$ provides good results for all target agents. We also performed the same tests with shortage and holding costs (10,1), (1,0), (1,0), and (1,0) for agents 1 to 4, respectively, and obtained very similar results.

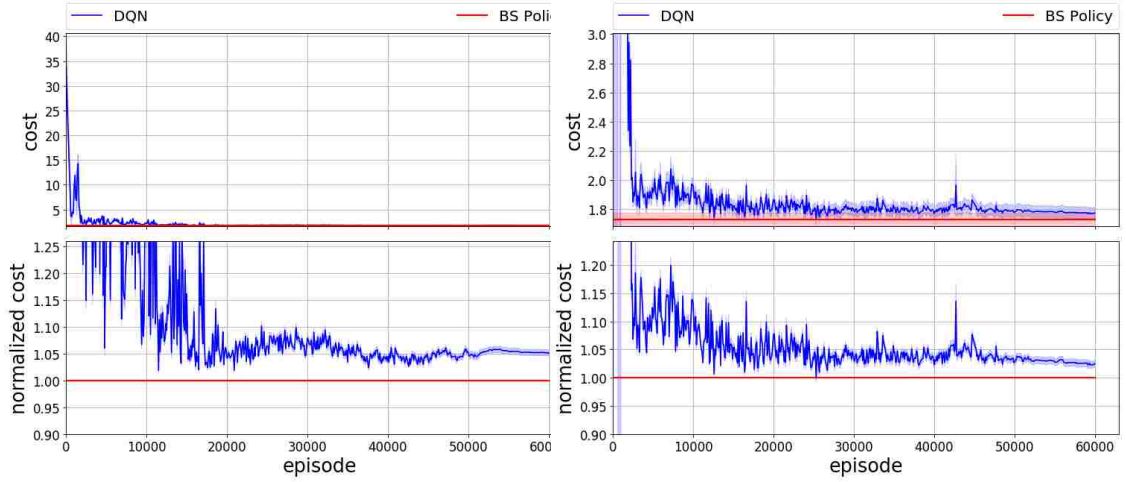
N.3 Transfer Knowledge for Different Size of Action Space

Increasing the size of the action space should increase the accuracy of the $d + x$ approach. However, it makes the training process harder. It can be effective to train an agent with a



(a) Case 1-4-1

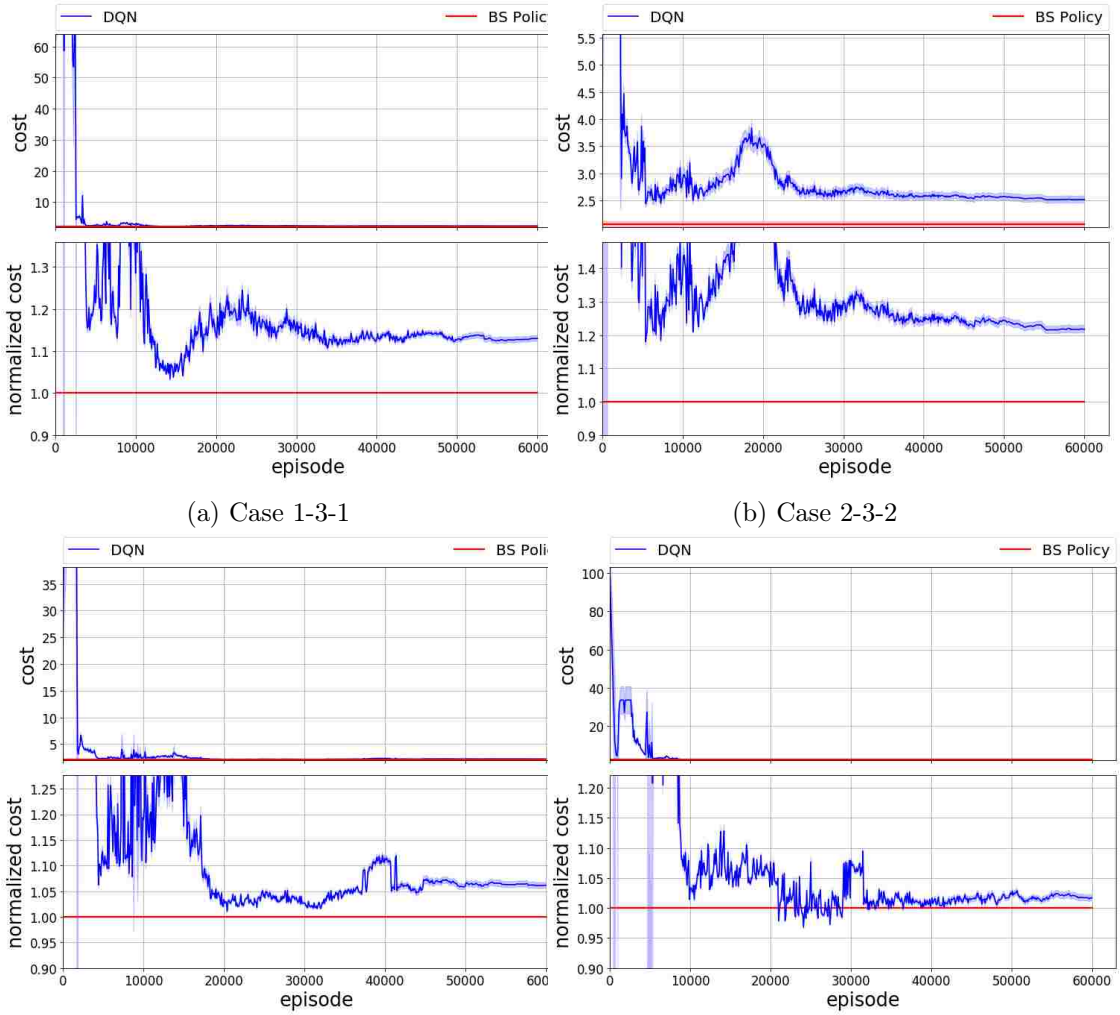
(b) Case 2-3-3



(c) Case 3-1-1

(d) Case 4-4-2

Figure N.2: Results of transfer learning between agents with different cost coefficients and same action space



(a) Case 1-3-1

(b) Case 2-3-2

(c) Case 3-4-2

(d) Case 4-2-1

Figure N.3: Results of transfer learning between agents with same cost coefficients and different action spaces

small action space and then transfer the knowledge to an agent with a larger action space. To test this, we test target-agent $j \in \{1, 2, 3, 4\}$ with action space $\{-5, \ldots, 5\}$, assuming that the source and target agents have the same cost coefficients.

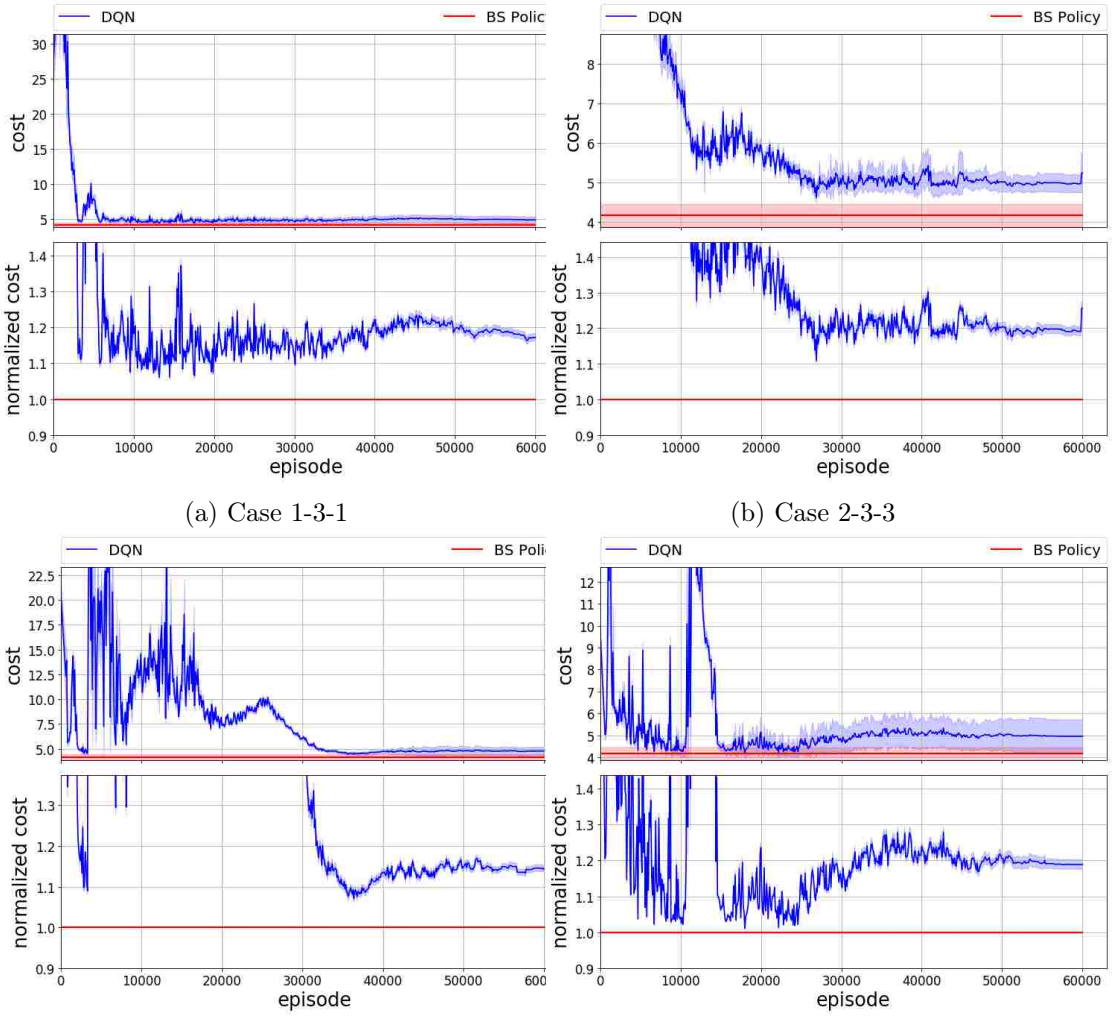
Figure N.3 shows the best results achieved for all agents. All agents attained costs that are close to the `BS` cost, with an average gap of approximately 10.66%.

N.4 Transfer Knowledge for Different Action Space, Cost Coefficients, and Demand Distribution

This case includes all difficulties of the cases in Sections N.1, N.2, N.3, and 4.4.3, in addition to the demand distributions being different. So, the range of demand, IL, OO, AS, and AO that each agent observes is different than those of the base agent. Therefore, this is a hard case to train, and the average optimality gap is 17.41%; however, as Figure N.4 depicts, the cost values decrease quickly and the training noise is quite small.

N.5 Transfer Knowledge for Different Action Space, Cost Coefficients, Demand Distribution, and π_2

Figures N.5 and N.6 show the results of the most complex transfer learning cases that we tested. Although the DQN plays with non-rational co-players and the observations in each state might be quite noisy, there are relatively small fluctuations in the training, and for all agents after around 40,000 iterations they converge.



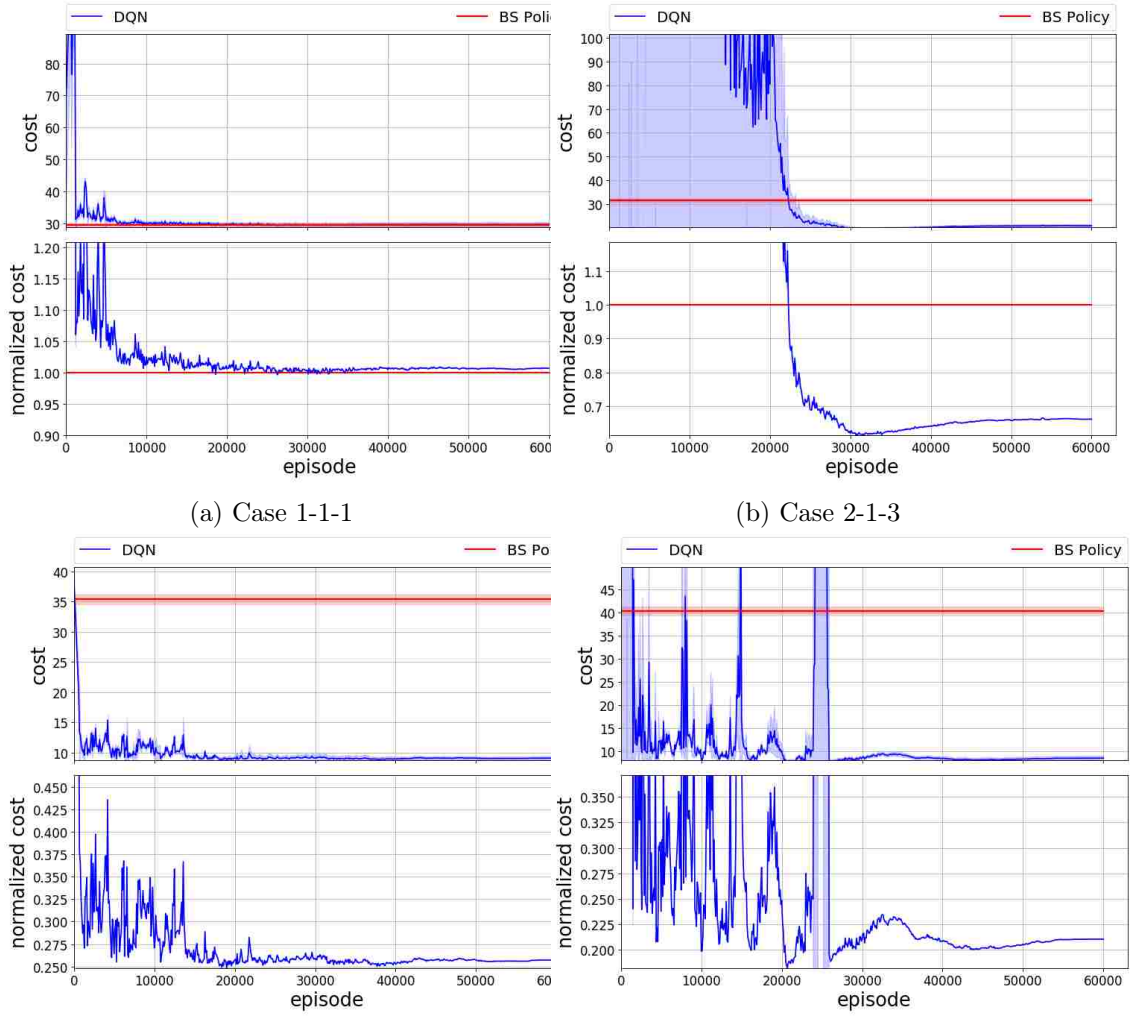
(a) Case 1-3-1

(b) Case 2-3-3

(c) Case 3-2-1

(d) Case 4-3-2

Figure N.4: Results of transfer learning between agents with different action space, cost coefficients, and demand distribution



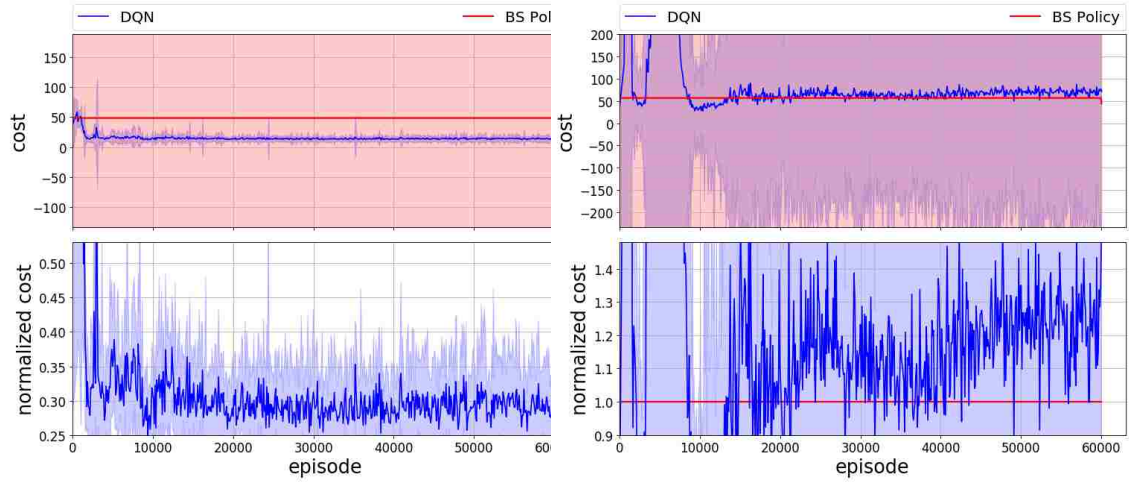
(a) Case 1-1-1

(b) Case 2-1-3

(c) Case 3-1-1

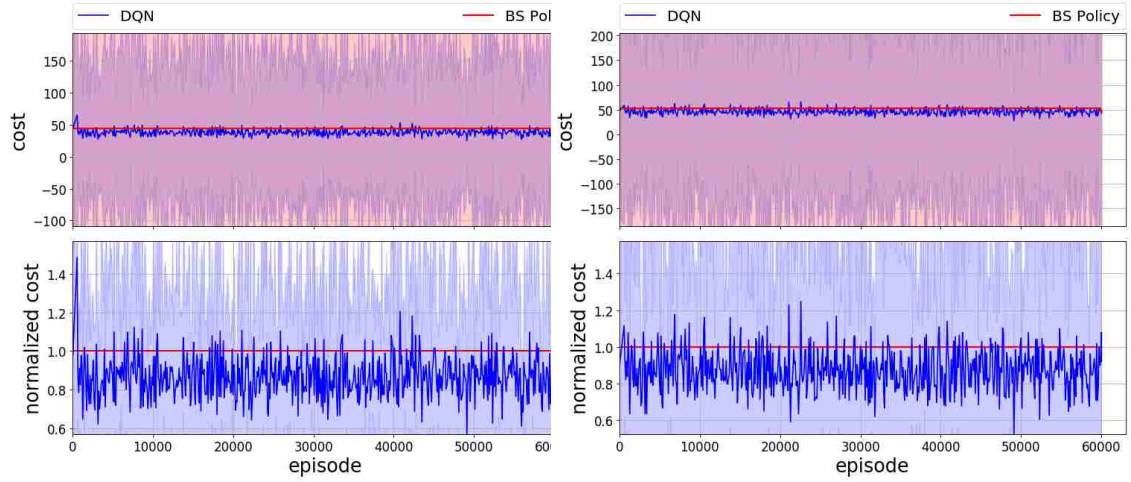
(d) Case 4-1-1

Figure N.5: Results of transfer learning between agents with different action space, cost coefficients, demand distribution, and π_2



(a) Case 1-2-1

(b) Case 2-1-2



(c) Case 3-3-3

(d) Case 4-1-1

Figure N.6: Results of transfer learning between agents with different action space, cost coefficients, demand distribution, and π_2

Appendix O

Pseudocode of the Beer Game Simulator

The DQN algorithm needs to interact with the environment, so that for each state and action, the environment should return the reward and the next state. We simulate the beer game environment using Algorithm 5. In addition to the notation defined earlier, the algorithm also uses the following notation:

d^t: The demand of the customer in period t.

OS_i^t: Outbound shipment from agent i (to agent $i-1$) in period t.

Algorithm 5 Beer Game Simulator Pseudocode

procedure playGame
  Set T randomly, and $t = 0$
  Initialize IL_i^0 for all agents
  $AO_i^t = 0, AS_i^t = 0, \forall i, t$
  while $t \leq T$ **do**
    # set the retailer's arriving order to external demand
    $AO_i^{t+l_i^{fi}} += d^t$
    for $i = 1 : 4$ **do**
      # choose order quantity
      get action a_i^t
      # propagate order upstream
      $OO_i^{t+1} = OO_i^t + a_i^t$
      $AO_{i+1}^{t+l_i^{fi}} += a_i^t$
    end for
    # set manufacturer's arriving shipment to its order quantity
    $AS_4^{t+l_4^{tr}} += a_4^t$
    # loop through stages upstream to downstream
    for $i = 4 : 1$ **do**
      # receive inbound shipment
      $IL_i^{t+1} = IL_i^t + AS_i^t$
      $OO_i^{t+1} -= AS_i^t$
      # determine outbound shipment
      current_Inv = $\max\{0, IL_i^{t+1}\}$
      current_BackOrder = $\max\{0, -IL_i^t\}$
      $OS_i^t = \min\{$ current_Inv, current_BackOrder + AO_i^t $\}$
      # propagate order downstream
      $AS_{i-1}^{t+l_i^{tr}} += OS_i^t$
      # update IL_i and calculate cost
      $IL_i^{t+1} -= AO_i^t$
      $c_i^t = c_i^p \max\{-IL_i^{t+1}, 0\} + c_i^h \max\{IL_i^{t+1}, 0\}$
    end for
    $t += 1$
  end while
end procedure

Biography

Afshin Oroojlooy is a doctoral candidate in the department of Industrial and Systems Engineering at Lehigh University. He obtained his Bachelor and Master degrees in Industrial Engineering in 2010 and 2012, respectively, from Isfahan University of Technology and Sharif University of Technology, Iran. His research interest revolves around reinforcement learning, multi-agent problems, and their applications in supply chain. He is a Rossin Doctoral fellow since 2017, and he also served as CORL admin from summer 2016. He spent the Summer 2017 and 2018 as Machine Learning intern at SAS Institute, while working remotely in between. His research focuses on reinforcement learning with applications to supply chains and specially inventory optimization. After completing his Ph.D., Afshin will join the Artificial Intelligence and Machine Learning division at SAS Institute as a Reinforcement Learning Researcher/Developer.