Instituto Tecnológico de Costa Rica
School of Computing Engineering
Costa Rica

A Deep Reinforcement Learning Approach to Multistage Stochastic Network Flows for Distribution Problems

Javier Fabio Porras Valenzuela

Committee:
Advisor
Ignacio Trejos Zelaya, M.Sc.
Scientific Advisor
Luis Leopoldo Pérez Pérez, Ph.D.

A thesis
submitted in partial fulfillment of the
requirements for the degree of
Master of Science in Computing (Computer Science concentration)

May 27th, 2022

Instituto Tecnológico de Costa Rica
Escuela de Computación
Unidad de Posgrado

ACTA DE APROBACION DE TESIS

A Deep Reinforcement Learning Approach to Multistage Stochastic Network Flows for Distribution Problems

Por: Javier Fabio Porras Valenzuela

TRIBUNAL EXAMINADOR

[illegible]

MSc. Ignacio Trejos Zelaya
Profesor Asesor

[illegible]

Dr.-Ing. José Mario Carranza Rojas
Profesor Lector

[illegible]

Dr. Luis Leopoldo Pérez Pérez
Lector Externo

[illegible]

Dra.-Ing. Lilliana Sancho Chavarría
Presidente, Tribunal Evaluador Tesis

SINAES ACREDITADO

13 de junio, 2022

Abstract

Distribution networks are a crucial part of supply chains that often entail highly complex optimization problems. An NP-hard example is minimizing the long-term transportation cost of multiple kinds of goods over a time horizon from suppliers to customers, considering order consolidation restrictions for shipments and demand uncertainty. This work presents a novel instance of such a distribution problem called the Shipping Point Assignment (SPA) problem, formulated as a multistage stochastic multicommodity network flows problem with additional nonlinear constraints, where the decision is to which warehouse to assign to the delivery of incoming orders to minimize inventory movements. Inspired by recent advances in combinatorial optimization using reinforcement learning and graph neural networks, we propose a deep Q learning agent with a GCN-based Value Function Approximator. We compare this agent with MLP-based, deterministic and greedy approaches over different simulations scenarios of the SPA problem. While the results do not suggest that the deep Q learning agent finds better policies than the reference agents, interesting avenues of future research were identified to enable reinforcement learning agents to learn from stochastic optimization problems with a graph structure.

Keywords: deep reinforcement learning, graph neural networks, distribution networks, stochastic optimization, network flows.

Resumen

Las redes de distribución son una parte crucial de las cadenas de suministro que frecuentemente implican problemas de optimización altamente complejos. Un ejemplo NP-duro es minimizar el costo de transporte al largo plazo de multiples tipos de bienes en un horizonte de tiempo, de proveedores a clientes, considerando restricciones de consolidación de embarques para las órdenes y la incertidumbre de la demanda. Este trabajo presenta una instancia novedosa de este problema de distribución, llamado el problema Shipping Point Assignment (SPA), formulado como un problema de flujos de redes multimercancía estocástico multietapa con restricciones no lineales adicionales, donde la decisión es a qué almacén asignar la entrega de órdenes entrantes para minimizar movimientos de inventario. Inspirados por avances en optimización combinatoria con aprendizaje por refuerzo y redes neuronales de grafos, proponemos un agente de deep Q learning con una función de aproximación de valor basada en GCN. Comparamos este agente con alternativas basadas en MLP, deterministicas y greedy en diferences escenarios de simulación del problema SPA. A pesar de que los resultados no sugieren que el agente de deep Q learning encontrara mejores políticas que los agentes de referencia, interesantes avenidas de investigación futuras fueron identificadas para abilitar que agentes de aprendizaje por refuerzo aprendan de problemas de optimización estocástica con estructura de grafo.

Palabras clave: aprendizaje por refuerzo profundo, redes neuronales de grafos, redes de distribución, optimización estocástica, flujos de redes.

To my mother, who arrived in this country 29 years ago to pursue a master's degree. Here it is, mom.

Contents

List of Figures

List of Tables

Acknowledgements

First and foremost, my infinite gratitude to my advisors, Ignacio Trejos Zelaya and Luis Leopoldo Pérez Pérez, this work is what it is thanks to their expert insight. I am very fortunate to be able to have worked with such talented individuals. Thank you for all the support and patience over many years.

I would like to thank the Costa Rica Institute of Technology, the school of Computer Science, and the defense committee comprised of Lilliana Sancho Chavarría and José Mario Carranza Rojas, for making this research possible.

The work of this thesis was inspired by my experience on supply chain optimization projects carried during my years in Singularities. I am forever grateful to Carlos Araya for all the guidance, as well as for granting me the opportunity to work in the most challenging projects of my career.

Several people kindly offered their feedback regarding the supply chain and operations research aspects of this work. Special thanks to Natalia Mesén, who took much of her time to validate the use case and provided me with valuable contacts. Thanks also to Sebastián Urbina, Marcos Moya Navarro, and Harold Schoenbeck.

Thank you to the people who took the time to read the drafts, listen to presentation rehearsals or offer feedback: Amber McKenzie, José Mario Marín, René Reyes, and Miguel Porras. Thank you also to Barnum Castillo and Randall Jiménez, for supporting the initial exploration of the topics during the Tema Selecto course.

Finally, thank you to my parents for being a steady source of love and support along the years. Thanks to my grandmother Helvia Pérez, for believing in me when not even I did. Thank you also to all my dear friends and family, for supporting me, and for putting up with me for so long when I would talk about nothing else but my thesis.

1 Introduction

Supply chains are a critical part of modern economies. A supply chain is defined by Snyder [4] as the activities and infrastructure dedicated to moving products from where they are produced to where they are consumed. They are expensive operations, accounting for \$1.5 trillion in costs for companies in the United States [5]. As supply chains are high scale operations, even marginal improvements in operational costs have very significant benefits for companies and consumers. Besides their high costs, having efficient supply chains give companies a competitive advantage. Supply chain operations include a variety of tasks such as demand forecasting, production planning, inventory planning, warehouse location decisions, among many others. One important cost driver is the daily decision-making involved with handling the distribution of goods.

The problem of distributing goods with minimal cost has been of interest to academics since the 1940's, with the proposal of the transportation problem [6]. Since then, massive progress in mathematical modeling and optimization have resulted in a mature field of supply chain management and optimization. However, distribution problems in the real world are still computationally very challenging due to their inherent uncertainty and high number of potential scenarios that must be considered. Additionally, existing algorithmic approaches to solving problems with uncertainty are brittle and do not hold in general cases. Instead, they typically exploit characteristics that are specific to each problem.

Distribution networks can be modeled as network flows programs, which in the simpler scenarios are special cases of linear programs [7]. When considering additional variables such as multiple commodities or flows across time, the computational complexity of the problem increases [8]. And even considering all these factors that add complexity, optimizing deterministically – that is, ignoring the inherent randomness of certain variables, such as demand – inevitably leads to inefficient decision making in real world problems. Stochastic optimization is a field that addresses this limitation with great success in linear programming with stochastic variables, typically by exploiting characteristics of the stochastic formulations. However, stochastic integer programs, which may resemble more closely many real world scenarios, are more difficult to solve due to the large number of NP-hard scenarios that must be considered on problems with few exploitable characteristics, except on certain special cases [9].

An alternative could lie in deep learning and reinforcement learning methods. The deep learning approach is to use statistical techniques called neural networks to automatically learn chained combinations of functions automatically in order to perform a task. By removing human design as much as possible, deep learning has enabled breakthroughs in diverse domains such as image recognition [10], machine translation [11] and speech recognition [12]. Reinforcement learning, which is focused on how agents can learn not through patterns but rather through experience by interacting with an environment, has also enabled great progress on tasks when coupled with deep learning, such as learning to play Atari games [13], and beating the Go world champion [14]. More recently, deep learning in the form of graph neural networks and reinforcement learning approaches has made significant progress in combinatorial optimization tasks, performing better than previous machine learning strategies [15][16][17]. These results motivate the exploration of deep learning methods to automatically exploit the graph structure of distribution problems in order to handle their inherent uncertainty and high dimensionality, instead of the manual analysis of problem structure in order to exploit mathematical properties, which has been the general direction of the research community.

This work presents a novel formulation of a distribution problem called the *shipping point assignment problem*. It is inspired in real world characteristics such as demand uncertainty, multiple commodities, order consolidation requirements and the consideration of the movement of goods across time. These characteristics make it highly dimensional and difficult to solve even for deterministic optimizers. The problem is formulated as a Markov decision process, and a novel deep reinforcement learning-based agent is proposed to derive a policy that can optimize long term efficiency in distribution planning decisions by choosing the most efficient shipping point for customer orders. The proposed agent is compared against deterministic optimization-based and greedy heuristic-based policies in order to evaluate its performance.

The document is structured as follows. The first chapter is the introduction. Chapter 2 is the conceptual framework, which provides an overview of network flows optimization, reinforcement learning, stochastic optimization and deep learning techniques. Chapter 3 is a brief description of related work: reinforcement learning approaches to supply chain, dynamic resource allocation problems that may be related to distribution networks, and graph neural network applications to network flows. In chapter 4, the shipping point assignment problem is stated. It is a kind of multistage, stochastic network flows problem that simulates the behavior of a distribution network. Chapter 5 presents motivations and justifications of the proposed problem. Chapter 6 states the hypothesis. Chapter 7 delineates scope and limitations of the presented work. Chapter 8 formulates the research objectives. Chapter 9 is a summary of the deliverables that provide evidence of the work reported in this thesis. In Chapter 10, methodological aspects are addressed, such as specifications of the implementation of the shipping point assignment problem as a Markov decision process, a simulator to represent the environment, a set of agents to derive policies, metrics, parameters as well as the experimentation plan. Chapter 11 shows the results and analysis of the aforementioned experiments. Chapter 12 summarizes the work and findings. Chapter 13 proposes future research directions.

2 Conceptual framework

2.1 Network flows

Network flows problems are optimization problems with an underlying network structure where the objective is to move an entity across the network efficiently. This kind of problem arises naturally in many different settings: manufacturing, transportation networks, energy gridlines, hydraulic systems, etc. The network in question could be a vehicle transportation grid, a system of water pipes or the paths between airports in various cities. In those examples, the things to move through the network are cars, water flow and airplanes, respectively. Network flows are interesting because their graph structure allows for specialized algorithms to solve them more efficiently than with general linear programming techniques.

The most common network flows problems fall in one of three categories:

1. **Shortest paths:** The shortest (or least cost) way to traverse a network from point A to point B.
2. **Maximum flows:** Given a network with fixed maximum capacities on the number of items that can flow through each arc, maximize the amount of flow between two points in the network.
3. **Minimum cost flows:** Given a graph with demands of a commodity on certain nodes and supply on others. The nodes are connected through arcs, each associated with a cost and a capacity restriction. Find the amount of units of the commodity that should flow through each arc in order to satisfy the demand requirements with the minimum possible cost.

These three categories of problems can all actually be formulated as linear programming problems. However, due to their graph nature, the resulting programs would be represented with very sparse constraint matrices, and the algorithms would not be able to leverage the network structure to handle problems with a large number of variables and constraints efficiently. In this work, the focus is mainly on minimum cost flows, which will be used to model the distribution networks in a supply chain environment.

2.1.1 Minimum cost flows optimization

Minimum cost flows are related to moving commodities from supply nodes to demand nodes to satisfy their demand by passing through the network with the lowest cost possible. It is intuitive to think of it in the context of transportation networks: moving physical goods from supply warehouses to consumer destinations while minimizing transportation costs. However, they are applicable to many other fields, such as energy grids, water pipelines, manufacturing and traffic networks.

Formally, the network is defined as a directed graph $G = (N, A)$, where N is a set of n nodes and A a set of m directed arcs $(i, j) \in A$. Arcs have an associated capacity u_{ij} and a cost c_{ij}. Some notations also include a lower bound l_{ij}, which for the purposes of this work this will always be zero. Each node i has an associated balance $b(i)$ that represents either supply if positive or demand if negative. Finally, the decision variable x_{ij} represents the flow going through arc (i, j) The minimum cost flows problem is then formulated as: [18]

$$\begin{aligned}
&\text{minimize} \quad \sum_{j:(i,j)\in A} c_{ij}x_{ij} \\
&\text{subject to} \\
&\sum_{j:(i,j)\in A} x_{ij} - \sum_{j:(j,i)\in A} x_{ji} = b(i) \qquad \forall i \in N, \\
&l_{ij} \leq x_{ij} \leq u_{ij} \qquad \forall (i,j) \in A
\end{aligned} \tag{1}$$

The first set of constraints are called *mass balance constraints.* They specify that the balance of a node is equal to the sum of all flows from incoming arcs minus the sum of all flows from outgoing arcs. The second set of constraints is called the *flow bound constraints* and they model the physical limitations of the network.

Alternatively, in matrix notation:

$$\begin{aligned}
&\text{minimize} \quad cx \\
&\text{subject to} \\
&\mathbf{N}x = b, \\
&l \leq x \leq u \qquad \forall (i,j) \in A
\end{aligned} \tag{2}$$

Where $\mathbf{N}$ is the node-arc incidence matrix for the minimum cost flows problem, with $a + 1$ for inbound nodes and $a - 1$ for outbound nodes, and zeros elsewhere. The vector b is the balance for each node, x is the vector of flow for each arc, l and u are the vectors with the lower and upper bounds for each node.

The following assumptions must hold true when working with minimum cost flows: [19]

1. **The *integrality assumption***, which means that arc capacities, costs and node supplies and demands are always integers. This does not limit expressive power, as rational values can be converted to integers by multiplying them by a sufficiently large number.

2. **The network must be a directed graph.** An undirected graph can be trivially generated by adding two arcs for each undirected arc.

3. **The sum of all node balances must be equal to zero** ($\sum_{i\in N} b(i) = 0$), and a feasible solution to the problem must exist. The first part of this assumption can be guaranteed by creating artificial "dummy" nodes with balance equal to the difference between supplies and demands of the original problem. The feasible solution part can be obtained by solving

a max flow problem on this artificial formulation and ensuring no flow is assigned to artificial arcs.

4. **There exists uncapacitated directed paths between every pair of nodes.** This can be satisfied by creating artificial arcs with an arbitrarily large cost and infinite capacity between all unconnected nodes. However, these arcs would only be used if there is no feasible solution to the original problem, so, in implementation, they do not need to be present.

5. **Arc costs are nonnegative.** To satisfy this also only takes a trivial transformation, simply sum the greatest negative arc cost to all arcs to transpose.

2.1.2 Transportation problem

An important special case of the minimum cost flows problem occurs naturally in distribution settings, called the *transportation problem.* A classic problem in operations research and management science literature first formulated by Hitchcock [6], its objective is to minimize the cost of moving goods from their origins, called sources, to their required locations, called sinks. A more general formulation is called the *transshipment problem*, in which intermediate nodes are allowed to be used to satisfy a given demand [20]. The transshipment and transportation problems are equivalent and both can be represented as minimum cost flows. The following example is taken from Ahuja [7] to illustrate an application of the transshipment problem using networks.

Given a set of p plants with known supplies and q warehouses with known demands, the goal is to find the flows on the arcs that satisfy the demands at the warehouses while respecting the supplies and minimizing transportation costs, to obtain an optimal production and shipping plan. Consider a car manufacturer that produces several car models, which are then shipped to retailers in various locations. The problem can be modeled as follows:

Generate four kinds of nodes:

- **Plant nodes:** Represent the plants that produce the models.
- **Plant-model nodes:** Represent the models produced at the plant.
- **Retailer-model nodes:** Represent the reception of a specific model at a retailer.
- **Retailer nodes:** The nodes that aggregate the demand.

and three kinds of arcs:

- **Production arcs:** From plant nodes to plant-model nodes. The cost represents the cost of producing a unit of this model at this plant, and capacities may be used to control the limits of production capacities for each model.

- **Transportation arcs:** From plant-model to retail-model nodes. The cost of the arc is the cost of shipping that model to a specific retailer, and the capacities associated may reflect contractual limitations or distribution capacities.

- **Demand arcs:** From plant-model to retail-model. They can have zero costs and lower bounds equal to the demand for the model.

After solving this problem, the flows in the arcs will represent the optimal production plan at each plant, and the optimal shipping strategy from each plant to each retailer. Notice that this model has an evident limitation: imagine that the transportation channels are shared for all models and have a limited capacity. With this formulation, it is not possible to represent such a constraint, since the capacities are independent for each plant-model to retail-model arc. This issue will be addressed in a subsequent section when discussing multicommodity flows.

2.1.3 Network simplex algorithm

Network simplex methods are one of the most commonly used methods for optimizing minimum cost problems. Inspired by the simplex algorithm to solve linear programs, discovered by George Dantzig in 1947 [21], network simplex methods exploit the graph structure of network flow problems while using the general simplex framework. The computation is performed by using the concept of spanning-tree solutions, analogous to basic feasible solutions in linear programming.These solutions are iteratively improved by moving from one spanning tree solution to another, each time introducing a new non-tree arc into the solution and removing another one, an action analogous to pivoting in linear programming [22].

To understand the network simplex algorithm, it is necessary to explain the concepts of cycle-free solutions and spanning tree solutions. A *cycle-free solution* is a feasible solution to the optimization problem that contains no cycle with free arcs. A free arc in this case is an arc (i, j) such that $0 < x_{ij} < u_{ij}$, whereas a restricted arc has a flow of either 0 or u_{ij}. A solution is a *spanning tree solution* if every arc in the graph not in the solution is a restricted arc. A property of minimum cost flows is that there always exists an optimal cycle-free spanning tree solution. This property allows the search for optimality to be limited to spanning tree solutions only [22].

Formally, a spanning tree solution can be viewed as a partitioning on the arc space: (T, L, U), where for all $(i, j) \in L$, $x_{ij} = 0$ and similarly, for all $(i, j) \in U$, $x_{ij} = u_{ij}$. The mass balance constraints define the flow on the arcs in T. The optimality conditions are intrinsically related to this partitioning.

The optimality condition for a shortest path problem is $d(j) \leq d(i) + c_{ij}$, where $d(i)$ represents the shortest path cost from a source node s to node i. This holds true because if it were not, then the arc c_{ij} would be part of the shortest path and the shortest path cost of $d(j)$ would be lower. This condition can be rewritten as $c^d_{ij} = d(j) - d(i) + c_{ij}\ \forall (i, j) \in A$, with the interpretation that c^d_{ij}

is the *reduced cost* of the arc (i, j), which means that it is the cost of that arc relative to the optimal shortest path distances between i and j. Inspired by this optimality condition of shortest path problems, we can define the *potential* of a node as $\pi(i)$ and the reduced cost of an arc as $c^{\pi}_{ij} = c_{ij} + \pi(j) - \pi(i)$. These node potentials also happen to be the dual variables in the linear programming formulation of the minimum cost flows problem. [22]

With an understanding of node potentials, we can now state the so called complementary slackness optimality conditions for a minimum cost flow problem, which state that a feasible solution x^* is optimal if and only if the following conditions hold true for some set of node potentials π: [22]

$$\begin{gathered}\text{If } c^{\pi}_{ij} > 0 \text{ then } x^*_i j = 0 \\ \text{If } 0 < x^*_{ij} < u_{ij} \text{ then } c^{\pi}_{ij} = 0 \\ \text{If } c^{\pi}_{ij} < 0 \text{ then } x^*_i j = u_{ij}\end{gathered}$$

These conditions can also be used with spanning tree structures. We can say that a spanning tree structure is optimal if it is feasible and the following conditions hold true for some set of node potentials π:

$$\begin{gathered}\text{If } c^{\pi}_{ij} = 0 \text{ then } (i, j) \in T \\ \text{If } c^{\pi}_{ij} \geq 0 \text{ then } (i, j) \in L \\ \text{If } c^{\pi}_{ij} \leq 0 \text{ then } (i, j) \in U\end{gathered}$$

An economic interpretation of reduced costs in minimum cost flows is that, given a starting node with potentials $\pi(1) = 0$, the reduced cost of a nontree arc $(i, j) \in L$ c^{π}_{ij} represents the change in the cost of the flow incurred by sending one unit of flow from node 1 to i and then to j through (i, j) and finally back into node i through the rest of the tree. Therefore, a positive reduced cost c^{π}_{ij} implies that it is not efficient to add this arc to the spanning tree solution.

Network simplex begins by obtaining a feasible spanning tree, and *pivoting* from one spanning tree to the other. The pivoting operation is performed by iteratively finding an arc that violates this optimality condition, adding it to the spanning tree solution creating a negative cycle, sending the maximum possible flow through this arc until the flow of another arc in the cycle reaches its lower or upper bound, and removing that arc from the spanning tree. When no other arcs violate the optimality condition, we have arrived at the optimal spanning tree. Algorithm 1 shows the pseudocode for network simplex [7].

2.1.4 Multicommodity flows

As previously mentioned while reviewing the car manufacturing example, minimum cost flows are limited in the sense that flow is assumed to be of a single

Algorithm 1 Network Simplex Algorithm

```
function NetworkSimplex
Precondition: $G = (N, A)$ is the graph for a minimum cost flow problem
  satisfying the assumptions.
    $(T, L, U) \leftarrow generateInitialFeasibleSolution()$
    $X \leftarrow$ Flow associated to $(T, L, U)$
    $\pi \leftarrow$ Node potentials associated to $(T, L, U)$
    while $nonTreeArcViolatesOptimality(T, L, U)$ do
        $(k, l) \leftarrow findEnteringArcViolatingOptimality(T, L, U)$
        $(p, q) \leftarrow findLeavingArc(T, L, U, (k, l))$
        $(T, L, U, X, \pi) \leftarrow updateTreeAndSolutions(T, L, U, (k, l), (p, q))$
```

type of commodity. If the problem requires that goods of different kinds flow across the same network, sharing the arcs' capacities, a different formulation is required. Using vector notation, the formulation for the multicommodity flow problem is: [19]

$$\begin{aligned}
&\text{minimize} \quad \sum_{1 \le k \le K} c^k x^k \\
&\text{subject to} \\
&\sum_{1 \le k \le K} x_{ij}^k \le u_{ij} && \forall (i, h) \in A, \\
&\mathcal{N} x^k = b^k && \forall k \in 1, 2, ..., K, \\
&0 \le x_{ij}^k \le u_{ij}^k && \forall k \in 1, 2, ..., K,
\end{aligned} \tag{3}$$

where c^k and x^k refer to the vectors of costs and flows respectively for each arc on commodity k, $\mathcal{N}$ is the node-arc incidence matrix, b^k is the vector of node balances for commodity k and u_{ij}^k is the upper bound of flow for commodity k on arc (i, j).

The capacity constraints are called *bundle constraints* because they make sure the combined flows of all commodities across each arc don't exceed the total arc capacities. It is also possible to impose capacities on the arcs if necessary.

The first difference in this formulation with respect to the single commodity version is that costs, flows and balances are now expressed as a vector for each commodity k. Which, if not for the bundle constraints, would mean having $|K|$ parallel single-commodity linear programs. This is a property exploited by many solution schemes for multicommodity flows approximation.

Another important distinction is that multicommodity flows may not always have integer solutions. In the maximum flow problem example with three commodities, sources and sinks in Figure 1 [7], the optimal solution would be to send 0.5 units of each commodity to share the 1 unit arc capacities between nodes 1,2 and 3, for a total flow of 1.5. If an integer solution is sought, only one unit from one of the commodities can be delivered for a total max flow of 1 and without satisfying all the demand constraints.

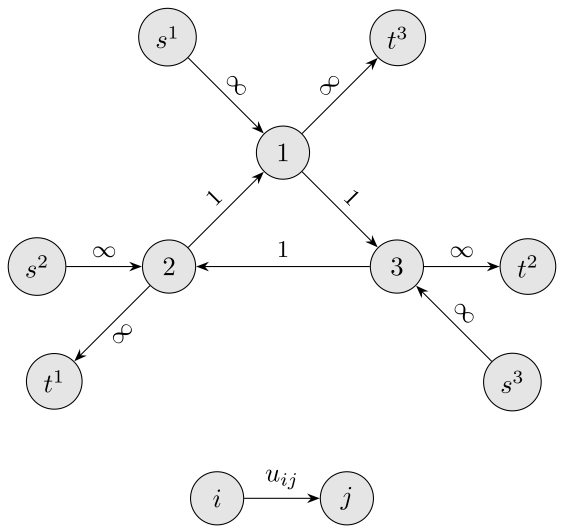


Figure 1: Maximum flow problem without integer solution

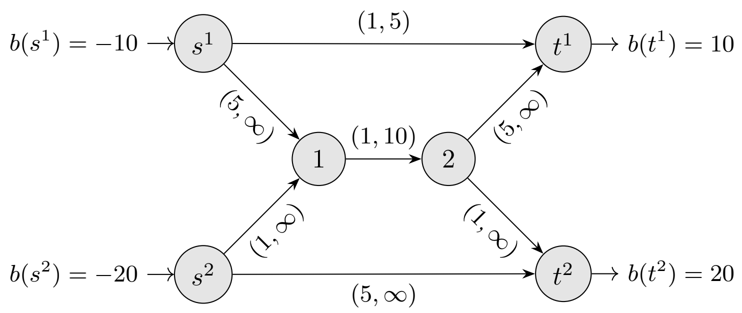
There are two main scenarios where it makes sense to formulate a problem as multicommodity flows:

- Different kinds of goods or commodities that share a common network. For example, a distribution network with fixed capacities and different kinds of products.
- The same kind of good but with multiple pairs of source and destinations that need to send the good to each other. For example, in telecommunications, many messages with a source and a destination share the same physical network.

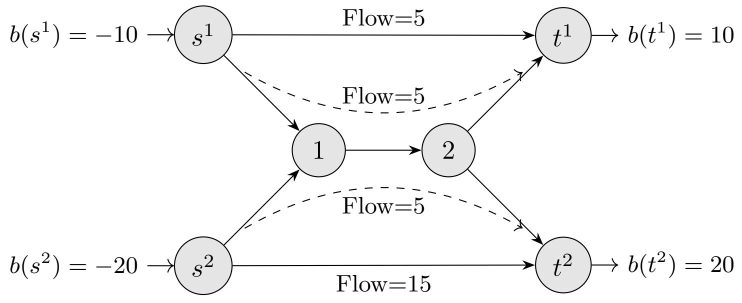
The following example from Ahuja [7] illustrates a multicommodity flow network. In Figure 2a, there are two commodities, $k = 1$ and $k = 2$. All supply from $k = 1$ comes from node s^1 and is required at t^1, and all supply from $k = 2$ comes from s^2 and is required at t^2. All but two arcs are uncapacitated: (s^1, t^1) with a capacity of 5, and $(1, 2)$ with a capacity of 10. The demand for commodity $k = 2$ could be satisfied by sending all the flow directly through (s^2, t^2). However, the arc $(1, 2)$ with limited capacity has a lower cost, but it must be shared by both commodities, because commodity $k = 1$ cannot satisfy its demand by just using (s^1, t^1). Figure 2b shows the optimal solution, where the shared arc is used by sending 5 units of each commodity, while the rest of the flow is sent through the direct (s^1, t^1) and (s^2, t^2) arcs.

2.1.5 Flows over time

Flows in the real world typically happen over a period of time. A significant limitation of the traditional network flows model is precisely that it does not



(a) Problem



(b) Solution

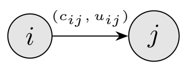

Notation:

Figure 2: Multicommodity min cost flow example with two commodities

consider flows as a function of time. Network flows over time models, also called dynamic flows, seek to overcome this limitation. The first models, introduced by Ford and Fulkerson [18], consider a maximum flow problem with a time τ_{ij} associated with the arcs as well as its capacities, and is solved by an iterative static flow computation over the network.

The most popular approximation is for this problem is obtained by transforming the network into a time expanded network (TEN), which is essentially copying the network nodes for each discrete time step and adding arcs that connect nodes through time based on the transportation time τ_{ij}. This approach has an important limitation: the size of the resulting TEN grows linearly with both the size of the original network and with the number of time steps (as opposed to growing logarithmically with the number of timesteps), which is why this kind of solution's complexity is *pseudopolynomial.* [23]

Although the minimum cost flow problem over time is proven to be weakly NP-hard [24], several special cases and approximation techniques have been developed to tackle these kinds of problems.

Most ways of handling flows over time involve exploiting problem-specific qualities or that may simplify the computation. For example, Klinz [24] proposes a variant called quickest flows over time, where the objective is satisfy the demand in the shortest possible time. Quickest flows with one commodity can be solved in polynomial time [25]. Fleischer expands by proposing fully polynomial time approximation schemes based on converting temporally repeated solutions to paths. Also, time expanded networks can be solved approximately with reasonable precision in polynomial time by coarsening the discretization of time and then translating the coarse solution to the original network [23].

Other special cases to be noted are when transit times are zero [26] or when transit times are uniform across arcs in a path [23]. In both scenarios, the time expanded networks can be simplified and solved in polynomial time due to the fact that all paths share the same transit time.

2.1.6 Multicommodity min cost flows over time

The combination of multicommodity flows and flows over time results in a distinctly more difficult problem. In this case, multiple commodities flow across a common network with transportation times associated to the arcs. An important condition that doesn't arise in static flows is that prohibiting the storage of flows at intermediate nodes results in a strongly NP-hard problem. Although evidently harder than single-commodity flows over time, the NP-hardness of multicommodity flows over time remained an open problem for many years until proven by Hall, Hippler and Skutella in 2007 [8].

Efficient algorithms for approximating multicommodity min cost flows over time also mostly leverage exploitable characteristics of the problem. For example, multicommodity flows over time with uniform path lengths can be solved as a static multicommodity flow problem in polynomial time, and quickest multicommodity flows over time with the same characteristics can be decomposed into a logarithmic set of static multicommodity flow computations [8]. Another useful approximation technique was developed by Groß and Skutella for multicommodity flows over time in the case where intermediate storage is not allowed [27], which has been mentioned previously to be a complicating restriction on flows over time problems. The method relies on condensing the time expanded network into an intermediate between an arc-based and a path-based formulation, by means of solving the *separation problem* (finding an arc sequence of minimum cost connecting two nodes, with the length of the sequence is restricted to lie on a given interval). It is developed first for maximum multicommodity flows over time and then generalized to any multicommodity flows over time problem.

In all the approximation scenarios laid out so far, not only are the solution strategies heavily reliant on specific structural properties of each optimization problem, but they also all work exclusively with linear optimization objectives. Therefore, they are not suitable as general purpose tools for solving nonlinear network flow problems. Additionally, the introduction of stochasticity in these NP-hard flow problems has not been studied much to the best of our knowledge.

2.1.7 Limitations of traditional flow models

To illustrate with an example, consider a consumer goods company that seeks to distribute their products from their factories to satisfy customer orders. The orders are composed of multiple kinds of products that all share the same distribution and inventory storage capacity limitations, and a customer's order is required to be delivered in full by a single truck, which means that the required supply must be guaranteed to be available before shipment begins. Even ignoring that customer demand is a random variable that influences the problem constraints, the single-origin requirement is sufficient to render traditional network flows models useless in this scenario, since it would require the introduction of integer variables. The company in question would then be forced to make a decision: relax some of the nonlinear constraints and make suboptimal decisions, or incur in a costly combinatorial optimization that may not even be guaranteed to yield a feasible solution in time to execute the operation.

Multicommodity min cost flows over time offer a way of modeling dynamic problems using algorithms from static flows. But as pointed out by Powell [28], using deterministic models for dynamic problems brings several complications: the resulting time-expanded models may be too large, they may be too sensitive to demand forecast scenarios, and they may have inferior performance as compared to directly considering stochastic scenarios.

2.2 Reinforcement learning

Reinforcement learning is a field of study that focuses on how autonomous agents make decisions while interacting with an environment in order to achieve a goal through a reward function. An autonomous agent is a decision-making entity that perceives the environment through sensors and performs actions in order to achieve some goal, and a reward function is a numerical feedback that tells the agent whether its actions were beneficial towards the goal given the current state of the environment. In a reinforcement learning framework, an agent is not given explicit instructions about how to operate besides the “rules of the game”: the constraints in the environment. Instead, it is expected to discover the best way to take actions based on experience and its current situation in order to maximize its reward over time. The goal of a reinforcement learning agent, informally, is to maximize its expected reward in the long term. The strategy with which an agent generates actions in a given state is called a policy. Inspired by insights from animal psychology about the way organisms interact with reality and adapt, reinforcement learning seeks to model intelligent behavior through simple, general principles that can be used to act autonomously. It also draws out ideas from the field of optimal control, which studies the behavior of a system over time when its dynamics are known. In this review of the theory of reinforcement learning, the notation and formulas of Sutton and Barto [29] will be used unless stated otherwise.

Reinforcement learning should be considered a separate machine learning framework, neither supervised nor unsupervised. The goal of supervised learning is to learn how to extrapolate predictions from a set of labeled examples. Although supervised learning may be used in some specific contexts in reinforcement learning, it is a fundamentally different kind of learning with distinct underlying objectives. The objective of reinforcement learning is to maximize a reward signal in all kinds of situations possible for an environment, which may be impractical to generalize based solely on specific labeled examples. On the other hand, it is also not unsupervised learning. Unsupervised learning tries to find underlying hidden structure in unlabeled data, which also does not equate optimizing for a specific reward signal. Using experience to drive learning is unique to reinforcement learning and it sets it apart from supervised and unsupervised learning [29].

Another fundamental distinction of reinforcement learning with other machine learning approaches is the arisal of the exploration vs. exploitation tradeoff. In its path to maximize its reward, an agent constantly faces a choice: to leverage what it already knows and obtain a certain reward, or sacrifice short-term gains to explore more about its environment in the hopes of discovering higher reward-yielding situations. This dilemma clearly does not exist in supervised or unsupervised learning, as it is the result of an agent interacting with the environment [29]. A number of techniques in reinforcement learning have been developed to address the exploration-exploitation tradeoff and some will be covered later.

2.2.1 Markov decision processes

To arrive to a formal definition of reinforcement learning problems, we must first introduce the concept of Markov decision processes (MDP). MDPs are a mathematical formulation for a sequential decision making scenario where limited information about an environment is made available to an agent. In an MDP, the *agent* is an entity that selects actions in an interaction with an *environment*, which in turn gives feedback by taking the agent into a new *state* and presenting it with a reward for its previous action. The objective of the agent is to maximize this reward signal given by the environment over time. [30] An important assumption is that the latent distributions of state transitions and rewards of the environment are stationary to some degree, meaning that they do not change for at least some period of time in order for the agent to be able to learn it successfully.

We define S as a discrete set of states provided by the environment, A as the set of possible actions and R as the set of possible rewards, and assume these three to be finite. The agent-environment interaction occurs over discrete timesteps $t = 0, 1, 2, ...$ At each timestep t, the agent senses its current state $S_t \in S$ and selects an action $A_t \in A(s)$. The environment, in response to the action, returns a real-valued reward on the next timestep R_{t+1} and takes the agent to a new state $S_{t+1} \in S$ [30].

For the random variables S_t and R_t, a conditional probability distribution can be drawn given the previous state and action in the MDP. This is called the dynamics function, and is defined as follows:[29]

$$p(s', r|s, a) \coloneqq P\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\}$$

In simple terms, this reads as: the probability of obtaining the reward r and arriving to state s' given that the agent was previously in state s and chose action a. Having access to the probability function p, it is possible to compute an optimal policy to achieve the path of highest reward. However, in most relevant problems, the function p is either not known or practically impossible to evaluate, so we have to develop methods that approximate the value of the states without having access to p.

The goal of reinforcement learning was previously defined informally as maximizing the expected return in the long term. In the simple case where the interaction with the environment is episodic (the interaction happens in subsequences of finite length called episodes), then the return can be defined as:

$$G_t \coloneqq R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T$$

Where R_T corresponds to the reward obtained after arriving to a terminal state S^+, which is the state at which the episode ends upon arriving. In a setting where the interaction between the agent and the environment is continuous, which means there are no terminal states, optimizing to the simple sum of all episodes would make the goal itself tend to infinity. To handle these cases, the exponentially decaying average of the rewards is best suited:

$$G_t \coloneqq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

The hyperparameter γ is called the *discount rate*. Values of γ closer to zero will bias an agent to give a higher weight to rewards in the near future, while a γ closer to one will make rewards longer in the horizon more valuable [30].

Notice that this goal definition can be unified to define both discounted and undiscounted tasks by allowing γ to be equal to 1 but limiting the timesteps to being finite.

The function that maps states to their expected return while following a given policy is called a *state-value function* and is defined as:

$$v_\pi(s) := \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s]$$

Similarly, the function that maps states and actions to their expected return is called action-value function and is defined as:

$$q_\pi(s, a) := \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a]$$

Value functions are useful because they define how valuable it is for an agent to be in a particular situation (state or state-action) in terms of the reinforcement learning goal. They can be expressed recursively in terms of their successor states and the environment's dynamics function p using Bellman's equations: [31]

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma v_\pi(s')] \forall s \in S$$

$$q_\pi(s, a) = \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s', a')] \forall s \in S, a \in A$$

It is always possible to find at least one optimal policy π_*. Although there may be multiple optimal policies, they all share the same optimal state-value function which can be defined as $v_*(s) := max_\pi v_\pi(s)$, and likewise for the action-value function. The optimal value functions can be expressed independently of any policy in the form of the Bellman optimality equations (first presented by Bellman by the name of "basic functional equation" [32]), here presented in Sutton and Barto's unifying notation [29]:

$$v_*(s) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')]$$

$$q_*(s, a) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma \max_{a'} q_*(s', a')]$$

As illustrated by these equations, it is simple to derive the optimal policy if the state-value function is known: the agent only needs to compute the value of all possible next states and greedily choose the action that maximizes it. The action-value function makes it even easier, no look-ahead evaluation is required.

2.2.2 Dynamic programming

Dynamic programming, in the context of reinforcement learning, refers to algorithms that rely on computing and storing a value function from a perfect model of the environment, with finite and discrete state and action spaces [31]. Evidently, these conditions are of little use in scenarios where the dynamics of the environment are almost always unknown. Also, the memory requirements for the value function of an exponentially large state space with respect to the number of state variables make dynamic programming computationally infeasible to solve exactly, a phenomenon coined by Bellman as the *curse of dimensionality* [31]. However, dynamic programming techniques are still worth mentioning briefly due to their theoretical value; they serve as the foundation for more sophisticated methods.

In reinforcement learning, there are two kinds of tasks that need to be solved: prediction and control. The prediction task is to find the value function of a given policy. In other words, if the agent keeps acting the same way forever, we want to discover what the expected value will be for each state. On the other hand, control is how to change the existing policy in order to improve the agent's performance. Solving the prediction task helps solve control, since having knowledge of the value functions enables the agents to find better policies. At the same time, control influences the prediction task, since changing the policy also implies changing the value function.

The first method for solving the prediction task in dynamic programming is called policy evaluation. The strategy is to use Bellman's equation as an update rule for the value function:

$$v_{k+1}(s) \leftarrow \sum_{a} \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma v_k(s')]$$

This update rule illustrates the essential concept in dynamic programming called *bootstrapping*, which is the process of iteratively updating a value function based on incomplete approximations of it. The prediction task can also be solved as a system of linear equations. However, linear programming-based methods become impractical in size faster than their dynamic programming counterparts [29].

Next, it is necessary to have a way of gauging whether a particular policy is better than another one, in order to know whether it is worth changing the policy. It is possible to do so thanks to the policy improvement theorem, which states that given two deterministic policies π and π' such that:

$$q_\pi(s, \pi'(s)) \geq v_\pi(s) \forall s \in S$$

Then,

$$v_{\pi'}(s) \geq v_\pi(s) \forall s \in S$$

That is, if following policy π most of the time and then following π' leads to a higher value state for all states, then it can be said that π' is a better policy than

π. That means that if we create a policy that selects the best action greedily with respect to the current estimates of $q_\pi(s, a)$ all the time, it is guaranteed to be better than the original policy. If the value functions remain equal, it means that it is the optimal value function and both π and π' are optimal policies [29]. This improvement of a policy by acting greedily with respect to its value function is called *policy improvement*. From this point on, when we refer to the greedy policy, we mean greedy with respect to the approximation of the value function under a given policy π.

The process of combining steps of policy improvement and policy evaluation is called *policy iteration*. Since the policy improvement theorem guarantees that the new policy will be a better one, and a finite MDP has a finite number of policies, this method is guaranteed to converge to the optimal policy. The convergence of this method can be improved substantially in some cases by doing one sweep of value updates instead of a full policy evaluation. This method, called value iteration, is a special case of Generalized Policy Iteration (GPI), a framework that describes the interaction between the processes of updating a policy and bringing the value function closer to the true value of the current policy. GPI is a general framework that can be used to describe most reinforcement learning algorithms [29].

2.2.3 Monte Carlo methods

Since having a perfect model of the environment's dynamics is usually not a reasonable assumption, the next question that arises is how to estimate state values by learning from experience generated by interacting with the environment. Monte Carlo methods provide a solution: to calculate the expected returns per state by sampling in an episode-by-episode basis.

In the case of the prediction task, there are two ways of calculating the state-value function: first-visit Monte Carlo (averaging the returns after the first visit to the state s on each episode) or every-visit Monte Carlo (averaging the returns on all visits to s). They both converge to the expected value by law of large numbers, but they differ in that every-visit Monte Carlo presents theoretical properties which are related to eligibility traces, a concept not relevant for the purposes of this work. These methods can be applied analogously for estimating state-action values as well.

An important difference with Dynamic Programming methods is that in Monte Carlo, the updates of one state do not influence other states; that is, there is no bootstrapping. Another key difference is that the value updates are made at the end of each episode, as opposed to every step in Dynamic Programming. Although both prediction methods converge, Monte Carlo methods tend to be better when only a subset of the state space is relevant. This is typically the case if the state space is large but some states are very unlikely to occur [29].

In order to do control with Monte Carlo, Monte Carlo prediction for action values is important. The reason is that, without a model, state values do not carry enough information to structure a policy. However, this technique suffers from an issue of "maintaining exploration": since in evaluating a deterministic policy some actions will never be selected for a particular state, their values will never be updated. This can be solved by using a technique called “exploring starts”, in which the first state-action pair is randomly chosen and all state-action pairs have nonzero probability of being chosen as initial. However, it is not always a reasonable requirement, as one cannot always choose the starting state and action [29].

In order to do Monte Carlo Control with GPI, one needs to collect the action values for a full episode, perform policy evaluation for the observed returns, and then improve the policy for the visited episodes [29]. Some way of guaranteeing that all states are visited is necessary to prove convergence. Exploring starts is an option, another one is using an ϵ-greedy policy. An ϵ-greedy policy may choose the non greedy action with probability $\frac{\epsilon}{|A(s)|}$, and the greedy action with probability $1 - \epsilon + \frac{\epsilon}{|A(s)|}$. ϵ-greedy policies are a way to tackle the exploration-exploitation problem described earlier in a more flexible way than exploring starts, which is not always feasible.

Methods that use a single policy to evaluate and optimize are called on-policy. ϵ-greedy policies are an example of on-policy methods. In contrast, off-policy methods have separate mechanisms for optimization and gaining knowledge of the environment. Typically, this setting involves two policies, the policy to be optimized, called the target policy, and the one that drives exploration, called the behavior policy. [29] This brings the challenge of how to evaluate the target policy while acting according to the exploratory behavior policy. The solution is to calculate a weighted average of the returns by using the probability of a trajectory occurring in the target policy relative to the trajectory in the behavior policy, which is called the importance-sampling ratio:

$$\rho_{t:T-1} \coloneqq \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}$$

It has the useful property of not depending upon the dynamics of the MDP because they are the same for both policies. With the importance sampling ratio, the expected value can be transformed from $\mathbb{E}[G_t|S_t = s] = v_b(s)$ to $E[\rho_{t:T-1}G_t|S_t = s] = v_\pi(s)$. To do Monte Carlo with importance sampling, all that is needed is to calculate the weighted sum of the return values weighted by the importance-sampling ratio, and average over the count of visits to each particular state. This is called ordinary importance sampling, as opposed to weighted importance sampling which is biased towards $v_b(s)$ but tends to have lower variance and is more used in practice, according to Sutton and Barto. Dealing with high variance in Monte Carlo methods is an ongoing area of research. [29]

2.2.4 Temporal-difference learning

Temporal-difference (TD) methods draw ideas from both dynamic programming and Monte Carlo. Like Monte Carlo methods, TD is able to learn state values by sampling from experience without requiring a model of the environment.

And like DP, TD keeps track and updates state-value estimates, in other words, it performs bootstrapping. TD differs from Monte Carlo methods because TD does not need to wait until the end of the episode to update the values.

The simplest TD method, called one-step TD, updates the value of each state after it is visited using the rule

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)],$$

where the parameter $0 \leq \gamma \leq 1$ is called a trace decay parameter and $0 \leq \alpha \leq 1$ is the learning rate. The trace decay parameter γ controls how much weight to assign to earlier state values with respect to older ones. A value of $\gamma = 1$ will result in an algorithm equivalent to Monte Carlo methods. The part of the equation in brackets is also called the TD Error, which means the error incurred in the estimate of the value function and is important in theoretical reinforcement learning:

$$\delta_t \coloneqq R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$$

Besides the evident advantages over DP methods such as not depending on a model of the environment, TD methods may also be more practical than Monte Carlo in continuous settings or where episodes are too long and updating after every episode would be very inefficient. Also, though not a guarantee, TD methods tend to converge faster than Monte Carlo methods in practice as illustrated by Sutton and Barto on simulations using Markov Reward Processes (MRP) for benchmarking the prediction task [29].

One possible explanation for this is that while Monte Carlo methods optimized the expected sampled return, which is equivalent to minimizing Mean Squared Error on the observed data, whereas TD methods optimize according to the maximum-likelihood estimation of the underlying MDP. That is, TD methods calculate the transition probability times the expected reward based on the sample average, also known as the certainty-equivalence estimate, because it is the estimate if we assume that the process is known with certainty rather than approximated [29].

2.2.5 SARSA for on-policy TD control

Applying the concept of TD for control only requires to change from state-value to action-value function. Since this requires the 5-tuple of states and actions given by $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, this method is referred to as SARSA. The update rule is:

$$Q(S_t, A_t) \longleftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

To obtain an on-policy control algorithm, we can use an ϵ-greedy policy with respect to our approximation of Q.

2.2.6 Q-learning for off-policy TD control

Q-learning is like SARSA, except that it approximates q_* directly, regardless of the policy being used. The values are updated with the maximum value for all possible actions: [30]

$$Q(S_t, A_t) \longleftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \text{max}_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

A control algorithm with this update rule is off-policy, because the value being predicted (q_*) is different from the actual value of the policy being followed (which could be an ϵ-greedy policy, for example).

2.2.7 Expected SARSA

Expected SARSA is a variant of SARSA where the TD error is calculated with respect to the weighted probability of each Q value while following policy π. It is computationally more expensive than SARSA and Q-learning, but it tends to converge faster. The reason for this is that expected SARSA has the same expected value than SARSA, but with lower variance due to the weighted average over the actions. Here, the update rule is: [29]

$$Q(S_t, A_t) \coloneqq Q(S_t, A_t) + \alpha[R_{t+1} + \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)]$$

Although typically used on-policy, it is worth noting that it can be made an off-policy algorithm by setting a behavior policy. In fact, if the target policy is the greedy policy, expected SARSA is equivalent to Q-learning [29].

2.2.8 Value function approximations

So far all the methods that have been presented are feasible only under the assumption that the state space is both discrete and of limited cardinality so as to be computationally tractable to keep the values of each state in memory. However, this is not always feasible, or even desirable. It could be the case that the state space is continuous, or that it is discrete but many states share similar properties and it makes sense to generalize an agent's behavior when encountering them, so that updating the value of one state affects all similar states. This is called value function approximation (VFA). Instead of having exact values for each state, we aim to approximate the value of multiple states at once.

For the prediction task, the objective must change from converging to the true value of a state to minimizing the error in the estimation of all states. Moreover, some state's values may be more important than others, for example, the values of the most frequently visited states. The objective of the prediction task, to put an example, can be defined as minimizing the weighted Mean Squared Error: [29]

$$VE(w) \coloneqq \sum_{s \in S} \mu(s)[v_\pi(s) - \hat{v}(s,w)]^2$$

where $\mu(s)$ is a distribution of the relevance of the states, $\sum_s u(s) = 1$ and $\hat{v}(s,w)$ is the value approximation of state s with respect to parameters w.

Regarding the question of which method of value function approximation to choose, it would be tempting to take any statistical approximation method from the supervised learning framework and apply it here. However, as stated by Sutton and Barto, there are key differences between the supervised and reinforcement learning frameworks that rule out some methods. Some supervised algorithms learn over a whole training set and must be retrained from scratch if the training data changes, which means that the training objective must be non-stationary. Reinforcement learning requires that the training be done online, while the agent is capturing data from the environment and the policy π influencing the value estimates may be changing as well. Even if the policy does not change, a bootstrapping method is constantly modifying the target values. For this reason, non-stationary methods should be ruled out [29].

A well-suited alternative is to use methods based on stochastic gradient descent (SGD). The idea is to represent the value function with a differentiable approximation $\hat{v}(s,w)$, where w is a real-valued vector.

This value function estimator is approximated using stochastic gradient descent in order to minimize the mean squared error of the difference between the estimator and the true value of the state:

$$w_{t+1} \leftarrow w_t \alpha [v_\pi(S_t) - \hat{v}(S_t, w_t)] x(S_t)$$

Notice that $v_\pi(S_t)$ is typically not known at the time of the update, therefore an approximation is necessary, so to perform the update we replace with an approximation U_t, which could be a noise-ridden approximation of $v_\pi(S_t)$, or the TD bootstrapping approximation of its value up to that point:

$$w_{t+1} \leftarrow w_t \alpha [U_t - \hat{v}(S_t, w_t)] \delta \hat{v}(S_t, w)$$

The above equation requires U_t to be an unbiased estimate ($\mathbb{E}[U_t|S_t = s] = v_\pi(S_t)$) to guarantee convergence with decreasing α. This is true for Monte Carlo estimates, which are generated by sampling. For TD methods, however, this does not hold due to the bootstrapping. Given the bias that bootstrapping introduces by the dependency on w_t, TD methods that rely on stochastic gradient descent are not *true* gradient descent. Rather, they are called semi-gradient methods. They do not have as robust convergence guarantees in all scenarios as true stochastic gradient descent, but they work well enough in some cases, such as when the VFA is a linear model [29].

A useful strategy to get better convergence performance on semi-gradient methods is to keep two copies of the model, one that is temporarily fixed for computing the gradient and another one that is updated with the gradient of the former. This approach helps stabilize the target value by not having it change with every exploration step [13].

Sutton and Barto put forth linear methods for approximation due to their convergence guarantees and computational efficiency. In this case, the parameterized value function is a linear combination of a set of features x with the weight vectors w, defined as:

$$\hat{v} := w^T x(s) = \sum_{i=1}^{d} w_i x_i(s)$$

with its respective update rule:

$$w_{t+1} \leftarrow w_t \alpha [U_t - \hat{v}(S_t, w_t)] x(S_t)$$

Their simplicity, computational efficiency and mathematical properties have made linear-based models very popular for function approximation. The effectiveness of linear methods rely on good constructions of features that enable the model to generalize over the state space. Since this work focuses on exploring new nonlinear approximations of value functions, the present methods are outside the scope. For a detailed overview of the most popular ones in reinforcement learning, see Sutton and Barto [29].

2.3 Stochastic optimization

Stochastic optimization is the field of study concerned with sequential decision making under uncertainty. After the community of mathematical programming realized that many real world optimization problems are influenced by stochastic variables, the community started developing techniques to handle them. These techniques were developed in parallel as the reinforcement learning community started growing, so many of them overlap with reinforcement learning strategies but with different names and conventions. For example, methods that can handle multidimensional state variables in reinforcement learning fall into the umbrella of approximate dynamic programming in the stochastic community. For an unifying view of the stochastic optimization and reinforcement learning literature, see Powell (2021) [33].

2.3.1 The farmer problem

To illustrate the motivation for stochastic optimization, consider the farmer problem, adapted from Birge and Louveaux [9]. A farmer plants wheat, corn and sugar beets. Each year she must decide how many of her 500 acres of land to allocate to each crop in order to maximize profit. She must meet a quota of wheat and corn to satisfy her farm's requirements, and can sell the rest of the yield. Sugar beets are the most profitable crop and there are no minimum requirements, but if she sells more than 6000 tons of sugar beets, government regulations force her to sell for a lower price. This is a simple linear program given by the following formulation:

	Wheat	Corn	Sugar beets
Yield per acre (T/acre)	2.5	3	20
Planting cost($/acre)	150	230	260
Selling price($/T)	170	150	36 (if under 6000T)
			10 (if above 6000T)
Purchase price($/T)	238	210	-
Minimum requirements(T)	200	240	-
Available land: 500 acres			

Table 1: Data for the farmer's problem

	Wheat	Corn	Sugar beets
Surface (acres)	120	80	300
Yield (acres)	300	240	6000
Sales (T)	100	-	6000
Purchases (T)	-	-	-

Table 2: Solution for the farmer's problem

$$
\begin{aligned}
\text{minimize} \quad & 150x_1 + 230x_2 + 260x_3 + 238y_1 - 170w_1 \\
& + 210y_2 - 150w_2 - 36w_3 - 10w_4 \\
\text{subject to} \\
& x_1 + x_2 + x_3 \leq 500, \\
& 2.5x_1 + y_1 - w_1 \geq 200, \\
& 3x_2 + y_2 - w_2 \geq 240, \\
& w_3 + w_4 \leq 20x_3, \\
& w_3 \leq 6000, \\
& x_1, x_2, x_3, y_1, y_2, y_3, w_1, w_2, w_3, w_4 \geq 0
\end{aligned}
\tag{4}
$$

Table 1 shows the meanings of the variables alongside constraints. After using a linear programming optimizer, the solution for the program is given in Table 2. This would be sufficient if the crop yield was the same every year. However, the farmer has observed that there is a $\pm 20\%$ variance in yield over the years due to external force such as weather variations. In order to evaluate if this uncertainty can alter her profits significantly, she runs two more optimizations with the worst and best case scenario of crop yields (assuming all crops will have the same variation in yield) and realizes that the profits may vary in the range from \$59,950 and \$167,667. If the yields are low, more land is required to plant the favorable beets, but if they end up being high, the risk is selling beets at an unfavorable price.

Notice that this stochastic version of the problem is composed of two stages. In the first stage, a decision must be made about the allocation of the land. In the second, the yield random variable is observed and the optimization problem of how to buy and sell crops must be solved.

One possible solution is to create an optimization with the constraints of all three scenarios, assuming they are all equally likely to happen. The solution of such a program would sacrifice meeting the maximum sugar beet quota in order to reduce the risk of having to sell crops at an unfavorable price. This kind of trade off would not need to happen if a perfect forecast of the yields existed.

Two important quantities arise from this scenario. Imagine that the yields are still random, but the farmer has knowledge on the yields before planting, allowing her to choose the best of the three allocation scenarios. The total profit in this condition is called the expected value of perfect information (EVPI). The second quantity, called the value of the stochastic solution (VSS), is the marginal value of optimizing this stochastic program versus the expected value solution, which is obtained by optimizing using the expected value of the yields. EVPI measures the value of knowing the future with certainty and VSS measures the value of knowing distributions on future outcomes.

2.3.2 General two-stage model

In the stochastic programming community, a decision taken before a random event is called a first stage decision, represented with x. A decision taken after the realization of a random vector ξ is a second stage decision, represented with $\mathbf{y}$. The boldface notation denotes a random vector, different from their realizations. The following example can be generalized to the form

$$\begin{aligned}
&\text{minimize} \quad c^T x + \mathbb{E}_\xi\, Q(x, \xi) \\
&\text{subject to} \\
&\mathbf{A}x = b, \\
&x \geq 0
\end{aligned} \tag{5}$$

where $Q(x, \xi) = \min\{\mathbf{q^T} y | W\mathbf{y} = \mathbf{h} - \mathbf{T}x, y \geq 0\}$ is the value of the second stage for a given realization, and ξ is the vector formed by the random components $\mathbf{q^T}$, $\mathbf{h^T}$ and $\mathbf{T}$. Q here is analogous to the action value function in reinforcement learning. In other words, minimize the value of the cost constraints in the first stage , plus the expected cost in the second stage after the realization of the random variables. In the farmer example, the random variable is the vector $\xi = (\mathbf{t_1}, \mathbf{t_2}, \mathbf{t_3})$ formed by the yields of the three crops. This notation can be simplified by defining $\mathcal{L}(x) = \mathbb{E}_\xi\, Q(x, \xi)$, called the value or recourse function.

Until now, the assumption has been that the random variable takes discrete values. This assumptions can be lifted and the values of the yields can be continuous. By deriving the recourse functions for each crop in this scenario, one can find that they are piecewise linear, convex, continuous and differentiable. Such calculations are not displayed here for brevity. These properties mean that the optimization problem is convex and a global minimum can be found.

The literature mentions the concept of multistage optimization problems [34]. Multistage refers to the fact that the second stage solution become the variables of the next stage of the problem, in contrast to each pair of stages being independent of each other, such as in the case of the farmer problem. For example, in a distribution network, the decisions made to fulfill the orders in one timestep affect the inventory that will be the starting point for the next round of orders. In a multistage problem, the recourse function is recursive, since it must consider all decision variables up until that point in time.

2.3.3 Solution methods

The naive way of solving this problem is to use a finite number of second stage realizations in order to form the fully deterministic problem. This approach is inefficient because the full deterministic form of the problem becomes quite large. The structure of the stochastic program must be exploited for more efficient methods.

The most common solution, called the L-shaped method [35] to solve this problem is to create a linearization of the objective function and solve the first stage problem plus the linearization. The linearization is typically done by a Dantzig-Wolfe decomposition [36] or a Benders decomposition [37].

2.3.4 Stochastic integer programming

Stochastic integer programming problems are more difficult to solve, and fewer theoretical results have been obtained. For example, adding integer restrictions to the constraints on the second stage problem significantly increases the complexity, because any first stage decision and outcome results in a different integer program which is already NP-hard [38].

There are few properties that can be exploited to develop general, efficient solution methods [9]. Instead, the literature focuses on special cases, such as when the first stage variables are binary [38] or when the second stage decisions are integer and the random variables come from a discrete distribution [39]. In such cases, a branch and bound framework is prescribed by performing optimality cuts based on the conditions.

2.4 Deep learning

Deep learning is a sub field of machine learning that has gained massive popularity in the last decade due to the success of neural network-based algorithms in a variety of supervised learning tasks such as image recognition [10], machine translation [11] and speech recognition [12], as well as reinforcement learning tasks such as learning to play Atari games [13], and beating the Go world champion [14]. The deep learning strategy is to automatically learn which features of an input space are required to perform a task, instead of relying on manually-provided features as in traditional machine learning algorithms. This is done via a series of nonlinear transformations on the input space that allow neural networks to compose complex representations from simpler ones. This approach of learning features is called representation learning [40]. The name deep learning refers to the fact that a neural network that composes many of such transformation graphically gives the impression of being deep.

Although artificial neural networks exist since the 1950s with the perceptron [41], and have been applied to some tasks in the late 80s such as handwritten text recognition [42], for many years they were believed to be unsuitable to be applied to larger problems. In recent years, due to a combination of algorithmic improvements, advances in computational capabilities and the availability of large enough datasets, neural networks have regained popularity as they achieve state of the art performance in various domains [43]. The initial groundbreaking results were obtained in image processing, where convolutional neural networks were first applied to drastically improve the accuracy in image classification tasks [44].

While there is still much to be researched about why deep learning has been so effective, there are a variety of empirical arguments that may justify using deep models. The most salient one is their ability to deal with highly dimensional data. The number of possible configurations grows exponentially with the number of dimensions, a phenomenon known as the *curse of dimensionality* [45]. Neural networks are statistically efficient in the sense that they are able to generalize well in these highly dimensional spaces, even when the number of examples is low relative to the number of possible configurations, thanks to the distributed representations that they learn [40]. Conventional machine learning algorithms generally struggle with the curse of dimensionality, because they typically depend on an implicit local constancy prior, which in essence means working under the assumption that examples that are close together in the input space of a function should not change much. Taking K-nearest neighbors as an example, a region containing a set of points that all have the same neighbors will predict a constant value. This is useful if the function varies in just a few dimensions, however it poses challenges when moving to higher-dimensional spaces [1].

To understand how the local constancy prior complicates generalization ability in highly dimensional problems, consider a checkerboard. In order to predict the color of a point in the board, it would be necessary to have at least one example in each square, even though it evidently has a structure that could be extrapolated. Another approach would be to introduce another explicit, stronger prior such as saying that the function is periodic. However, this way of introducing domain knowledge is evidently not useful to generalize to a wide array of problems.

It is indeed possible to represent highly-dimensional functions efficiently and to have an approximation that generalizes well non-locally. This is due to the fact that a $O(2^k)$ space can be represented with $O(k)$ examples, given additional, generic assumptions about the data generating distribution [46]. One such assumption that deep learning makes is that the data generating distribution is the result of a composition of features, possibly in many levels.

Another key idea that may help tackle the curse of dimensionality is related

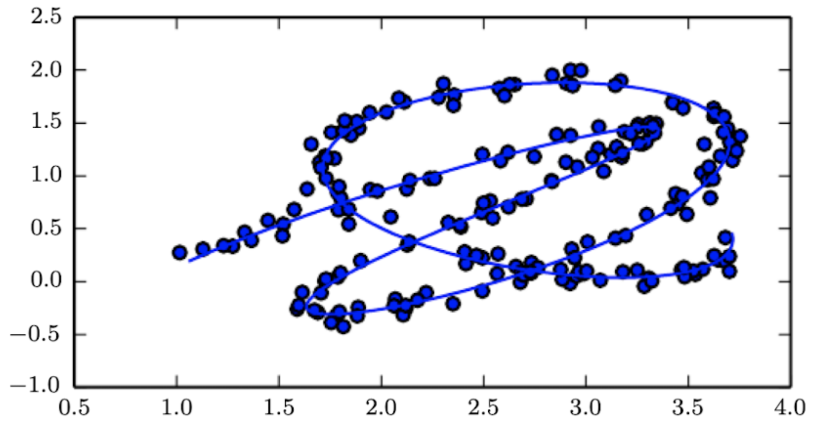


Figure 3: Example of a manifold with one degree of freedom. Taken from Bengio et al. [1]

to manifolds. A manifold, as viewed in machine learning, is a set of points in space connected through a neighborhood of points with limited degrees of freedom. Consider the example in Figure 3. The points are defined in a two-dimensional space, but the data generating distribution is actually a manifold with one degree of freedom. Similarly, when learning on manifolds, the assumption is that most of the inputs in a $\mathbb{R}^n$ space are invalid, and that the learned function would only vary within directions in the manifold. In other words, that the probability mass is highly concentrated across few dimensions. This assumption about the data generating distribution is called the *manifold hypothesis* [47] [48]. An informal intuitive example to consider why the manifold hypothesis seems reasonable, imagine that most randomly generated images are perceived as noise, and most randomly generated strings of characters are not real language. There is a non-zero probability of getting a picture of a dog, or a real sentence, but it is infinitely small in the input space. Another informal argument is that we can imagine transformations between valid examples, for example rotating an image, or transforming a sentence via synonyms or grammatical restructuring.

For these reasons, it is compelling to utilize deep learning to solve tasks that traditional machine learning algorithms have struggled with in the past.

2.4.1 Multilayer perceptrons

Multilayer perceptrons, also called feedforward neural networks, are the simplest neural network architecture. They are function approximators that map $y = f(x;\theta)$, where θ is the set of parameters that best maps x to the output y. In a multilayer perceptron, the input is passed through a series of nonlinear transformations that generate vector-valued representations. In figure 4, an

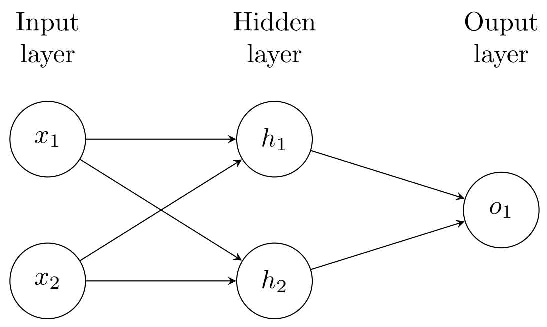


Figure 4: Feedforward neural network with one hidden layer for the XOR task.

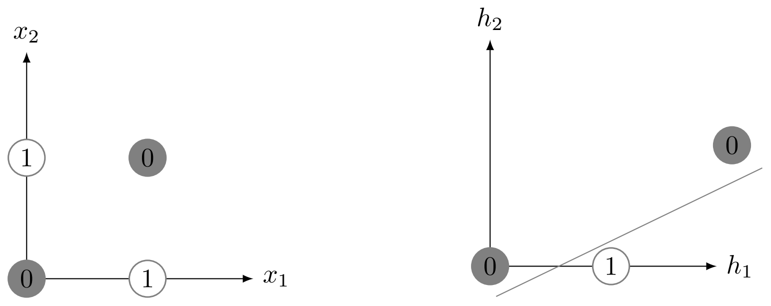


Figure 5: XOR function. On the left, the plot of the XOR function, showing that a linear model cannot completely represent it. On the right, the hidden feature space has transformed the function to a linearly separable mapping. Reproduced from [1]

example of a multilayer perceptron is presented, with its main parts: input layer, hidden layers and output layers. So if $f^{(1)}$,$f^{(2)}$,$f^{(3)}$ are the input layer, one hidden layer and an output layer, the approximation would be computed as $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$. These functions can be, for example, a linear transformation coupled with some nonlinearity, such as a sigmoid, hyperbolic tangent or a rectified linear unit. The number of hidden layers is referred to as the depth of the model, and the number of parameters in the hidden layers is called the width. It is useful to think of the hidden layers as a way of computing an alternative, nonlinear feature representation for $\phi(x)$ that allows the neural network to represent a richer set of functions than, for example, a linear model. In fact, neural networks are universal function approximators, which means that they are capable of representing any Borel measurable function (any continuous function on a closed subset of $\mathbb{R}^n$), given a sufficiently large network [49][50].

An important caveat to the universal approximation theorem is that being able to represent the function does not imply that it is guaranteed that the training algorithm will be able to learn the function; it could fail to converge or learn a different function due to overfitting.

As a concrete example, consider a multilayer perceptron with one hidden layer used to solve the XOR task, which should be able to create a nonlinear mapping. In Figure 5, we can see an example of the XOR function, where the original x space is nonlinear, but it can be transformed into an alternate h space that a linear model can be used to solve the task. The model contains two sets of parameters, W and w, and two functions $h = f^{(1)}(x; W, c)$ and $y = f^{(2)}(h; w, b)$, where b and c are the intercept terms. The full model would be $y = f^{(2)}(f^{(1)}(x))$. $f^{(2)}$ is a linear model. $f^{(1)}$ cannot be a linear model, because otherwise the whole model would become a linear model, as it could be rewritten as $f(x) = x^T w'$ in vector notation, where $w' = W^T w$. The solution is to perform a linear combination of the input space and the parameter vector, followed by some nonlinear transformation function, which is known as the activation function. The most common choice for activation functions is ReLU (Rectified Linear Unit) [51], defined as $g(z) = max(0, z)$. With that, the full model becomes, in vector notation,

$$f(x; W, c; w; b) = w^T \max(0, W^T x + c) + b.$$

The objective is to minimize the loss function in a regression problem, which can be mean squared error (MSE), defined as

$$J(\theta) = \frac{1}{4} \sum_{x \in X} (f^*(x) - f(x; \theta))^2$$

The following vectors yield a solution to the XOR problem:

$$W = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}; c = \begin{bmatrix} 0 - 1 \end{bmatrix}; w = \begin{bmatrix} 1 - 2 \end{bmatrix}; b = 0$$

The input matrix for XOR:

$$X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$

After applying the model with the specified values, the resulting vector would be:

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

which is the correct mapping for the XOR problem. It is worth noting that this is not the only set of values for W, w, c and b that result in a zero error solution for the XOR task.

The question now becomes how to arrive at the parameters that minimize the error. Since the model is not linear, we are unable to solve this equation in closed form. Instead, the function must be approximated using non-convex optimization techniques, such as gradient-based methods.

2.4.2 Training and backpropagation

Training a neural network means finding appropriate values for parameters θ that approximate the distribution of the process that generated the training data. However, in contrast with mathematical optimization, where the only goal is to minimize the objective error, here the minimization of the error with respect to the training data is being used as a proxy for approximating the true data generating distribution p_{data}, which means that we may not necessarily be interested in directly finding the absolute minimum for $J(\theta)$. The cost function can be written as

$$J(\theta) = \mathbb{E}_{(x,y)} \;\; \hat{p}_{data} L(F(x;\theta), y)$$

where $\hat{p}_{data}$ is the empirical distribution observed by the training data, with labels y assuming a supervised training scenario. We optimize this as a proxy for approaching the minimum of the true cost function,

$$J^*(\theta) = \mathbb{E}_{(x,y)} \;\; p_{data} L(F(x;\theta), y).$$

Since it is not possible to observe p_{data} directly, the goal in machine learning tasks is to reduce the expected generalization error $J^*(\theta)$, also known as minimizing the risk. The way this is achieved is by minimizing the empirical risk, defined over the training set as:

$$\mathbb{E}_{(x,y)} \;\; \hat{p}_{data(x,y)} L[F(x;\theta), y] = \frac{1}{m}\sum_{i=1}^{m} L(f(x^{(i)};\theta), y^{(i)})$$

where m is the number of training examples. Due to the fact that empirical risk minimization leads to overfitting and that sometimes the direct loss function does not have a useful derivative (for example when using 0-1 loss), a surrogate function is used instead. A common choice for a surrogate loss function is negative log-likelihood of the correct class. Another advantage of having surrogate loss functions is that generalization error can be further reduced even once 0-1 loss has reached zero on the training set, by further separating the boundary between classes. It is common to halt training not when the surrogate loss function reaches zero, but rather at a point where a metric based on a validation set reaches an optimal point, after which overfitting tends to occur [1].

The algorithm used to update the parameters of the layers of the neural network is called backpropagation. Essentially, backpropagation is applying the calculus chain rule to calculate the gradient of the error function and propagate that information from the output layer into the hidden layers, causing them to generate representations of meaningful features to fit the data [52]. The algorithm used to approximate the parameters that minimize generalization error is called stochastic gradient descent. In gradient descent, the intuition is to use the derivative of the function given by the model to update the parameters in small steps and thus reducing the value of the cost function iteratively, using an update rule such as

$$\theta \leftarrow \theta - \frac{1}{m}\alpha\nabla_\theta \sum_i L(f(x^{(i)};\theta), y^{(i)})$$

Algorithm 2 Stochastic Gradient Descent

```
Precondition: Randomly initialized parameters $\theta$
Precondition: Learning rate $\epsilon_k$
  while convergence not met do
      $miniBatch \leftarrow \{x^{(1)}, x^{(2)}, ..., x^{(m)}\} \sim X$
      $\hat{g} \leftarrow \frac{1}{m}\nabla_\theta \sum_i L(f(x^{(i)};\theta), y^{(i)})$
      $\theta \leftarrow \theta - \epsilon\hat{g}$
```

In stochastic gradient descent, a subset of the training data is used at each update to approximate the gradient instead of the whole dataset [53]. The full algorithm can be observed in Algorithm 0.

Consider finding the parameters that optimize maximum likelihood estimation:

$$\theta_{ML} = \text{argmax}_\theta \sum_{i=1}^{m} \log p_{model}(x^{(i)}, y^{(i)}; \theta)$$

which is equivalent to maximizing the expectation over the empirical distribution:

$$J(\theta) = \mathbb{E}_{(x,y)}\ \hat{p}_{data} \log p_{model}(x, y; \theta).$$

to minimize this, an optimization over the full training set must be performed:

$$\nabla_\theta J(\theta) = \mathbb{E}_{(x,y)}\ \hat{p}_{data} \log p_{model}(x, y; \theta).$$

Given that the gradient computation can be very expensive, and that the training set can be very large, stochastic gradient descent approximates the gradient instead of computing it exactly. Since the standard error of the mean is $\sigma/\sqrt{n}$, this means that there are less than linear returns from computing the gradient with more examples, so it is more statistically efficient to compute the gradient based on a sample of the training set. Methods that compute the gradient based on a sample are called mini batch gradient methods, not to be confused with batch gradient methods, which use the whole training set. Another method is online training, which uses one example at a time. It results in high variance of the gradient approximation which must be controlled via a lower learning rate, but often results in the lowest generalization error [54].

Since neural network training is a non-convex optimization problem, it is highly likely that a large number of local minima exist. Moreover, due to neural networks suffering from the model identifiability problem [55], there are potentially infinite parameter settings that result in the same local minimum cost function values. However, this is only a problem if the local minimum's values are high with respect to the true global optimum. Proving this is an open area of research, but there is evidence to suggest that most local minima have a low cost function value, and since it is not necessary to arrive at the global optimum, most of the time it is not an issue [56]. Additionally, there is theoretical work that proves that the number of saddle points in a high dimensional function grows exponentially with respect to the number of local minima, and empirical work that suggests that gradient descent tends to be able to escape saddle points most of the time for large enough networks, despite the small gradient norm at these points. [57].

An important technique in regularizing neural network models is an approximate bagging method called dropout [58]. The idea is to train the ensemble of all sub-networks created by removing non-output units. However, there can be exponentially many of these networks, and training a large ensemble of neural networks quickly becomes computationally intractable. Instead, the model is trained by randomly multiplying the output of some units by zero. When training using mini batch gradient descent, this equates to performing Bernoulli sampling on all the units of the network at each training step to select the subset that will be trained in that step. Formally, we are approximating, for a mask vector μ of which units to include and the cost function of the model defined by μ and the parameters θ, the expectation $\mathbb{E}_\mu J(\theta, \mu)$. In inference time, while a typical ensemble would calculate the arithmetic mean of the predictions, namely

$$\frac{1}{k}\sum_{i=1}^{k} p^{(i)}(y|x),$$

in dropout the predictions are given as the mean over all masks:

$$\sum_{\mu} p(\mu)p(y|x,\mu).$$

This sum is clearly intractable. An approximation can be attained by applying the weight scaling inference rule, proposed by Hinton et al. [59]. The idea is to approximate the normalized harmonic mean by evaluating $p(y|x)$ once on the model with all units, and multiply the weights by the probability of including each unit.

Another challenge in training neural networks, especially in recurrent neural networks, is the steep cliff-like regions that create exploding gradients values, as seen on figure 6. This may cause numerical overflow or straying very far from the optimal values. Based on the intuition that the gradient does not suggest the optimal magnitude but rather only the correct direction within an infinitesimal region, this can be addressed by applying gradient clipping. As long as the sign of the gradient is correct and the magnitude is not too large, learning will occur, however slowly. Whenever the gradient becomes too steep, gradient clipping reduces the learning rate to reduce the possibility of stepping too far off from the region.

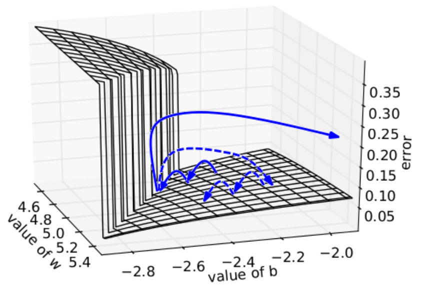


Figure 6: Gradient cliffs, common in recurrent neural networks, that make the gradient descent update perform abrupt jumps. Taken from Pascanu [2].

2.4.3 Convolutional neural networks

One of the most popular neural network architectures has been convolutional neural networks (CNNs) [42]. CNNs have been widely successful in tasks with data that present a grid-like structure, such as two-dimensional image data [44], spectral audio data [60] or time-series data [61]. They do so by means of processing data via the convolution operation instead of the typical matrix multiplication in fully connected neural networks, which allows for several benefits to automatic feature extraction and generalization.

The convolution operation is denoted as $s(t) = (x * w)(t)$, where x is a real-valued function referred to as the input, the function w is the kernel or filter, and the output is called the feature map. Although formally it is defined as an integral, here we focus on the linear transformation described by the discrete case,

$$ws(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a),$$

with the implicit assumption that the filter is defined as zero wherever values are not stored, so it is only necessary to compute the values where x and w are defined.

In image processing theory, a similar function called cross-correlation is commonly used. It is equivalent to convolution, but without flipping the kernel, and is sometimes called convolution regardless. The two-dimensional cross-correlation for an image I with a kernel K, common in image processing is defined as

$$s(t) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n).$$

To understand the motivation behind using convolutional operations in neural networks, consider processing images with a fully connected neural network. The input layer would have dimensionality equal to the number of pixels, with each pixel being an independent input to the network. If the image size is 256x256, that would be a vector of size 65,536. The next hidden layer would have a neuron connecting to each of those inputs. If the hidden layer is of size 50, the number of parameters would 3,276,800. The first evident drawback is that the network will be very large, with high data requirements and prone to overfitting. Another problem is that the input layer will be invariant to the ordering in the pixels: pixels that are close together will be independent from each other. Finally, the network would be very sensitive to positional shifts to the object of interest in the images [42].

Now consider a convolution with a kernel comprised of learnable parameters K. By making the filters of smaller size than the input images, the parameters connect sparsely to the input, instead of densely, like in a fully connected layer. This means that a given input interacts only with a subset of the parameters, and that an output has a limited receptive field (is only affected by a subset of the input). Indirectly, however, deeper layers may interact with the entirety of the input by means of hidden layers. In practice, this means higher statistical efficiency due to having less parameters, and more computational efficiency, as a matrix multiplication that results in an $m \times n$ matrix requires $O(m * n)$ computations, while a filter K with orders of magnitude less dimensions than m can perform a convolution with $O(m * k)$ steps.

Another advantage is that the filter parameters are shared between multiple inputs. In a fully connected network, a parameter is used exactly once, whereas, in a CNN, the same parameter is used to compute features in multiple parts of the input, which also means that more features can be learned with fewer parameters.

CNNs do not suffer from positional shifts in the input in the same way as fully connected networks. This is thanks to a property called translation equivariance, which means that if the input shifts, the output will shift as well. So if a part of an image containing a feature, for example an edge in an image, is moved a certain number of pixels, the output of the convolution with the transformed image will be shifted in the same amount as the transformed image.

Finally, constructing features with convolutions introduces the useful prior that the inputs are locally correlated. In the context of image processing, this means that pixels that are close together are likely to be correlated and may represent features. Fully connected networks make no such topological assumptions about the input dimensions, which makes it more difficult in practice to extract meaningful features from spatial and temporal data [62].

After the linear transformation performed by the convolution, the next step is to apply some nonlinearity, such as ReLU or hyperbolic tangent, and then pass the output to a pooling function, which will summarize the outputs over each region. One common operation is max pooling [63], which outputs the maximum over each $k \times k$ region. Pooling has two major advantages. First, it helps further reduce translation invariance, as it highlights the presence or absence of a feature, regardless of exactly where it was found. Second, the summarization of the output of a layer means a dimensionality reduction for the next layer, which supports statistical efficiency.

CNNs are usually combined with fully connected layers to create a classifier. An example of a CNN architecture that leverages convolutions, pooling and fully connected layers is AlexNet [44]. Its architecture consists of five convolutional layers, with pooling components in the first, second and last layers, followed by two fully connected layers into a softmax-activated output layer for outputting label probabilities.

2.4.4 Recurrent neural networks

Recurrent neural networks (RNN) [52] are networks that can process sequential data – such as time series, sentences, DNA sequences – by applying a recurrent function. The input to an RNN is a sequence of vectors $x^{(1)}, x^{(2)}, \ldots, x^{(\tau)}$ of arbitrary length τ. RNNs are capable of scaling to longer sequences than other neural network architectures thanks to parameter sharing and also to the fact that the outputs of the RNN are a function of previous outputs. Most RNN architectures can also process variable length sequences.

Although CNNs can also be used to process time series data by applying a 1-D kernel, for example as in Waibel et. al. [64], this approach only allows for sequences that are locally correlated in the size of the filter. In contrast, RNNs share parameters by means of unfolding a deep computational graph that relates each output with all previous outputs [52].

A computational graph is a representation of a series of computations in the form of a directed acyclic graph. For example, consider a dynamical system with a hidden state and driven by an external signal $x^{(t)}$,

$$s^{(t)} = f(h^{(t-1)}, x^{(t)}; \theta).$$

In this model, the hidden state $h^{(t)}$ encodes information observed up until timestep t, which can be interpreted as a lossy summary of the sequence history, because it encodes the information of an arbitrary length sequence into a fixed length vector. The operation that maps a circuit to a directed acyclic graph is known as unfolding the computational graph, which makes it a fixed length graph dependent on the length of the sequence. For example, for a sequence of length 3, the computation can also be denoted as a recurrent application of the function f,

$$s^3 = f(s(2); \theta) = f(f(s(1); \theta); \theta).$$

This formulation is what allows the input to the model to have the same length while being able to process sequences of arbitrary length. The same parameters are also reused at each step.

Since any parameterized function that involves recurrence can in theory be called an RNN, different kinds of RNNs can be identified. One architecture is a network that produces an output at each timestep and is recurrent from the output to the hidden state. In this architecture, each output $o^{(t)}$ is independent from each other. This kind of network is more limited in the functions that it can express, unless the output is highly dimensional and contains all the necessary information about the past. However, since the outputs are independent, backpropagation can be computed in parallel for each timestep.

A second recurrent network architecture produces an output at each timestep and is recurrent in the connections to the hidden states. A neural network with this architecture is capable of computing any function computable by a Turing machine, in an asymptotically linear number of timesteps [65][66]. The formulation of this network can be as follows, assuming a hyperbolic tangent activation function:

$$a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)} \tag{6}$$

$$h^{(t)} = \tanh(a^{(t)}) \tag{7}$$

$$o^{(t)} = c + Vh^{(t)} \tag{8}$$

$$\hat{y}^{(t)} = \text{softmax}(o^{(t)}) \tag{9}$$

where U, V, W are parameter matrices for input-to-hidden, hidden-to-output and hidden-to-hidden connections, and b and c are the bias terms for input-to-hidden and hidden-to output layers. The loss is given by the sum of the losses at each step:

$$L(\{x^{(1),\ldots,x^{(\tau)}}\}, \{y^{(1),\ldots,y^{(\tau)}}\}) = -\sum_t \log p_{model}(y^{(t)}|\{x^{(1),\ldots,x^{(\tau)}}\}), \tag{10}$$

where p_{model} is the probability of predicting $y^{(t)}$ given the previous observations $x^{(1),\ldots,x^{(\tau)}}$

Since in these equations, values of the output are dependent on previous values of the hidden state $h^{(t)}$, the computation of the gradient with respect to the parameters is an expensive $O(\tau)$ operation called back-propagation through time (BPTT). To compute this, the gradient is calculated recursively from the end of the sequence until the first step [67].

A third network architecture would have recurrent connections between hidden units and produce a single output value for sequence-level prediction. This architecture would also require BPTT to calculate the gradients.

If the model has connections from the outputs back into the model, a strategy called *teacher forcing* [68] may be used. In teacher forcing, instead of feeding back the output into the model, the labels may be used. This allows us to remove the time dependency if the model doesn't have hidden-to-hidden connections, which means that BPTT can be avoided.

In the cases where the prediction task depends on the whole of the sequence, for example in machine translation, where the output may depend on future values as much as on past values, the causal structure of the RNNs seen so far might not be sufficient. Bidirectional RNNs [69] handle this by having two RNNs, one in the forward direction. In this case, $h^{(t)}$ is the hidden state going forward and $g^{(t)}$ is the hidden state going backward.

RNN architectures can also be made deep by stacking blocks of parameters either in the hidden-to-hidden, hidden-to-output or input-to-hidden connections. The evidence clearly suggests that deep RNNs using these techniques outperform shallow ones [2].

One challenge particular to RNNs is handling long term dependencies. These are hard because of the problem of vanishing and exploding gradients [70][71]. Consider a scalar weight w, multiply it by itself multiple times, $\prod_t w^{(t)}$. Depending on the magnitude of w, this value will tend to either vanish or explode. This happens during RNN training due to the time dependency and can rapidly reduce the probabilities of successfully completing the optimization. Although this problem is still open, several approaches exist to try to avoid it. Susillo [72] argues that appropriate initialization of the weights to aim for a certain variance can avoid exploding or vanishing gradients. Echo state networks [73] and liquid state machines [74] try to avoid learning the recurrent hidden state weights by setting them in a way that they can represent the history and instead only learn the output weights. These techniques have also been found to be useful for initializing the weights to improve fully trainable RNNs [75] Another idea proposed by Lin et al. [76] is implementing skip connections in the unfolded computational graph to provide a shorter path to propagate gradient information.

One of the most successful strategies for solving this problem has been gated RNNs such as long short-term memory (LSTM) [77] and Gated Recurrent Units (GRUs) [78] [79]. In Gated RNNs, the model is allowed to accumulate information over many timesteps, and forget it once it has been of use, by using gate components that pass or eliminate information. This allows the potential timescale for the gradient computation to be dynamic, and thus support longer length dependencies.

2.4.5 Attention

An important idea in recent years for deep learning has been the concept of attention, first proposed by Bahdanau et al. for neural machine translation [11], but it has transcended into other domains such as computer vision, recommender systems and interpretability [80] and some graph neural network architectures such as GaAN [81]. The objective is to have a memory about previous inputs that an attention mechanism can selectively draw from. Before attention, most neural machine translation models used the encoder-decoder [78] approach, in which the model encoded the source sentence into a fixed-length context vector c using an RNN, and using that context to predict the words in the translation. The major drawback of this approach is that encoding a variable length sequence into a fixed length vector results in deteriorating performance with longer sequences. Using attention, the model generates a probability α_{ij} that reflects the importance of the hidden state h_j to the next word to be predicted y_i. The context vectors for each timestep are then calculated as

$$c_i = \sum_{j=1}^{T} \alpha_{ij} h_j$$

and used by the decoder in order to signal which hidden states to direct attention to.

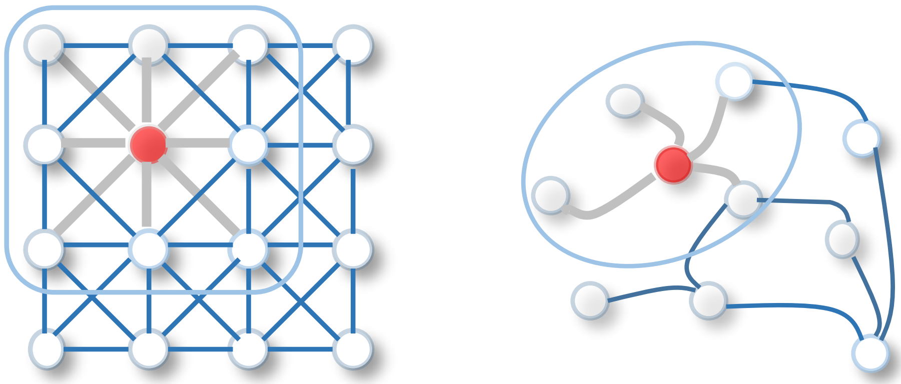


Figure 7: 2D Convolution vs. Graph Convolution. A visual representation of the analogy between 2D Convolutions in an image-like grid and a graph convolution. The 2D convolution takes the weighted average of pixels surrounding the red one. The proposed solution to a graph convolution is to take the average of neighbor's nodes features. Taken from Wu et al. [3]

Transformers [82] take attention one step further. Instead of having an attention mechanism coupled to a recurrent or convolutional network, transformers rely solely on attention and multilayer perceptrons, resulting in a more parallelizable model structure. Transformers are massively scalable, as proven by the 175 billion parameter language model, GPT-3 [83].

2.4.6 Graph neural networks

An area of research that has significantly grown in interest in the last ten years is graph neural networks. Deep learning has shown great success in applications using Euclidean data, that is, data that can be mapped into an Euclidean $\mathbb{R}^n$ space, such as images, text or audio. Inspired by such advances in network architectures and training techniques, the idea of applying neural networks in non-Euclidean domains, where data is represented as graphs, has grown stronger.

Graph data presents additional complications for neural network architectures such as CNNs and RNNs. First, a node's neighbors have no particular order and nodes have a variable number of neighbors, which complicate the calculation of a convolution operation. Second, graphs violate the core assumption in these frameworks that every example is independent, since nodes may be related by their arcs.Although there has been work on applying neural networks in graphs since the 1990s, recent advances in a new type of architecture called Graph Convolutional Networks (ConvGNN) have sparked renewed interest in GNNs [84]. The central concept in ConvGNNs is a generalization of the convolution operation as understood in the two-dimensional context. One can consider 2D convolutions as a special case of a graph convolution. An image is a grid-like graph with every pixel connected through arcs to all surrounding pixels, and the convolution operation calculates the weighted average of a pixel and its neighbors. Likewise, a graph convolution for any kind of graph aggregates the values of the features for each node with its neighbors. An example of this intuition can be visualized in Figure 7.

The first application of neural networks on graph data was on directed acyclic graphs by Sperduti et. al. [85]. The concept of Graph Neural Networks, however, started with Gori et. al [86], and resulted in a family of methods now known as recurrent graph neural networks (RecGNNs). RecGNNs typically apply a single set of parameters to nodes on a graph to generate node hidden representations. For instance, Scarselli's Graph Neural Network [87] calculates,for each node v,

$$h_v^{(t)} = \sum_{u \in N(v)} f(x_v, x^e_{(v,u)}, x_u, h_u^{(t-1)}),$$

until convergence is reached. Here, f is the neural network applied to the concatenation of the features of v, the features of each edge $x^e_{(u,v)}$ and the features of the neighbors x_u and the previous hidden state $h_u^{(t-1)}$. This is applied to all neighbors of v and then summed to give a new representation $h_v^{(t)}$. RecGNNs research was mostly focused on directed acyclic graphs using limited computational resources, with the objective of extracting node representations assuming that an equilibrium will be reached. [85]

In parallel, other approaches called convolutional graph neural networks (ConvGNNs) were developed. Instead of applying the same set of parameters recurrently until convergence, ConvGNNs have a fixed number of parameter sets (convolutional layers) that are applied sequentially.

Wu et. al. identify two families of ConvGNNs [84]. The first one is spectral-based ConvGNNs. They stem from techniques of graph signal processing theory, assume an undirected graph, and represent the graph with a normalized Laplacian matrix $L = A_n - D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$ where D is the degree matrix and A is the adjacency matrix. The Laplacian normalized matrix can be decomposed as $L = U \Lambda U^T$, with U being the eigenvalues and Λ the diagonal matrix of eigenvectors. The features are represented in terms of the inverse graph Fourier transform, defined as $\mathscr{F}^{-1}(\hat{x}) = U\hat{x}$, in order to represent the input as $x = \sum_i \hat{x}_i u_i$. The graph convolution operation, for a given signal x and a filter g_θ, is then defined as

$$x *_G g_\theta = U g_\theta U^T x, \tag{11}$$

and the different spectral-based methods vary on the implementation of the parameterized filter g_θ.

The second family is spatial-based ConvGNNs. Based on the aforementioned intuition that an image is a graph of spatially related pixels, spatial ConvGNNs create a convolution of a node's representations with its neighbors' representation. They borrow the idea of message passing of nodes to their neighbors from RecGNNs. For example, Neural Network for Graphs (NN4G) [88] calculates the hidden states as $H^{(k)} = f(XW^{(k)} + \sum_{i=1}^{k-1} AH^{(k-1)}\Theta^{(k)})$, where f is an activation function, $W^{(k)}$ is the parameter matrix for layer k for the node features and $\Theta^{(k)}$ is a diagonal matrix of parameters for the adjacency matrix. The usage of $H^{(k-1)}$ allows the hidden representations of nodes to be propagated into the neighbors in further convolutions.

Message Passing Neural Networks (MPNN) turns NN4G's approach into a general framework for spatial-based ConvGNNs by defining the convolution operation as

$$h_v^{(k)} = U_k(h_v^{(k-1)}, \sum_{U \in N(v)} M_k(h_v^{(k-1)}, h_u^{(k-1)}, x_{uv}^e)),$$

where $h_v^{(0)} = x_v$ and U_k and M_k are functions with learnable parameters [89].

Currently, spatial-based ConvGNNs are often preferred to spectral-based ones for a variety of reasons. First, spectral-based ConvGNNs are computationally less efficient due to having to compute the eigenvector or requiring to compute the whole graph at once, which means they are more difficult to scale to larger graphs. Second, they tend to generalize poorly to new graph structures, since any change to the graph means a change in eigenbasis. Finally, spectral-based graphs assume an undirected graph, which in many cases is not desirable, whereas spatial-based models are more flexible in the kind of information that can be incorporated into the aggregation function [84].

A special case is GCN [90], which is technically spectral-based, but instead of computing the expensive eigendecomposition of the graph Laplacian, it uses a first-order approximation using Chebyshev polynomials. The spectral graph convolution equation can be rewritten as

$$x *_G g_\theta = \theta(I_N + D^{-\frac{1}{2}} A D^{-\frac{1}{2}})x. \tag{12}$$

This approximation of the Laplacian is dependent only on the neighborhood instead of the full graph. For this reason, [3] points out that the composable GCN layer can be reinterpreted as a spatial method, by rewriting it as

$$h_v = f(\Theta^T(\sum_{u \in N(v) \cup v} \bar{A}_{v,u} x_u)) \quad \forall v \in V \tag{13}$$

where $\bar{A} = \tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}}$, with $\tilde{A} = A + I_N$ and $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$. This definition of $\bar{A}$ is a renormalization trick to improve numerical stability.

After convolutions are applied a downsampling operation called graph pooling is used. Graph pooling is useful for dimensionality reduction, permutation invariance and regularization, while avoiding the Fourier Transform required in spectral-based methods [91]. The operation is simply

$$h_G = (\mathbf{mean}|\mathbf{max}|\mathbf{sum})(h_1^{(K)}, h_2^{(K)}, ..., h_n^{(K)}),$$

that is, the mean, max or sum of the hidden representations of all the nodes in the last convolutional layer. This operation is also called graph readout when pooling all the node representations in order to obtain a representation that will be attached to an output layer for graph-level predictions. [91]

Graph pooling, however effective for dimensionality reduction, is still inefficient, since the embedding is of a fixed size regardless of graph size. One solution is Set2Set, which generates a variable size memory, passes it through an LSTM to encode order-dependent information and then aggregates it. [92]. Other approaches sort the nodes into a meaningful order [93] [94], add parameters to the pooling mechanism [95] or, use self-attention to learn the pooling mechanism [96].

For graphs that present variability not just in their structure but also in the nodes features over time, Spatial-temporal graph neural networks (STGNNs) have been developed. Two kinds of methods exist as of this writing: RNN-based and CNN-based methods. RNN-based methods, such as Graph WaveNet [3] combine a spatial ConvGNN with an RNN in the following way:

$$H^{(y)} = \sigma(\text{Gconv}(X^{(t)}, A; W) + \text{Gconv}/(H^{(t-1)}, A; U) + b).$$

Another framework used to capture the temporal aspect of the graph in both nodes and graphs is Structural-RNN [97]. Two RNNs are used, one for the nodes and one for the edges, with the node-RNN taking the outputs of the edge-RNN as input to represent the spatial information.

RNN-based methods have all the same issues of RNNs: high computational complexity due to the time-dependent backpropagation, and vanishing and exploding gradients. To address this, CNN-based methods instead pass the output of a spatial-convGNN to a one-dimensional CNN layer., thus removing the temporal dependency on the gradient. For example, CGCN stacks a CNN, followed by a graph convolutional layer followed by another CNN layer [98].

In order to learn latent dynamic spatial dependencies, for instance, when travel time between roads in a traffic network depends on traffic conditions, GaAN uses attention to learn dynamic spatial dependencies using an RNN-based approach. The attention mechanism works on the edges based on the input of the current state of its two nodes [81]. ASTGCN includes both a spatial and a temporal attention mechanism to learn latent dynamic spatial dependencies and temporal dependencies [99].

One application were graph neural networks have been successful is in combinatorial optimization. Gasse et. al. [15] was able to improve upon state-of-the-art branch and bound solvers by training a spatial ConvGNN on the bipartite graph representing the optimization problem with the labels from a slow expert. Dai et. al [16] uses a reinforcement learning approach with graph embeddings in order to learn to solve combinatorial problems over graphs. It does so by reducing several NP-hard problems such as max cut and travelling salesman problems into minimum vertex cut problems, and training a struct2vec embedding network, which is then use as a value function approximation on the state space for the deep RL agent, being on par with commercial greedy solvers. Li et. al. [17] also manage to surpass other deep learning methods for graph NP-hard problems by combining a Graph Convolutional Network with guided tree search applied on problems reduced to Maximal Independent Set (MIS).

2.4.7 Deep reinforcement learning

Experience replay [100] is a reinforcement learning technique by which an agent stores and continually learns from past experiences to gain stability and learn more efficiently from the interaction with the environment. DQN [101] leverages this technique along with neural networks value function approximation to improve upon online Q learning. In DQN, a replay buffer is updated sequentially with the last n steps of interaction with the environment. In order to decorrelate this sequentially derived experience, a random sample of the experience buffer is taken to create the minibatch. This is also beneficial for neural network training, as a batch of experiences will be more stable during training than single point experiences and will make better use of computational resources. To further improve training stability, DQN uses a separate network, called the target network, to calculate the target values of the Q learning update rule. The target network is only updated periodically, therefore making the target values more stable than if they were computed using the same network.

3 Related work

3.1 Reinforcement learning approaches to supply chain

Reinforcement learning has been investigated on supply chain scenarios, but mostly on inventory management, for example [102] or the RL agents that learn to play the Beer Game [103] [104]. In these scenarios, multiple RL agents collaborate in an environment with a single commodity and demands that do not necessarily match supply. Instead, when a demand is not satisfied by the available inventory, an additional cost is incurred. The action space is the amount to request to other agents in order to replenish inventory.

Stockheim [105] proposes a multi agent approach to supply chain optimization, where the agent's action space is to accept an incoming job or not, based on the job price, due date and expectation of other jobs in the future. It does not give the agents the option of manipulating the due dates of incoming jobs nor reassigning jobs from one agent to another.

Portrandolfo [106] proposes another pure RL model based on an average reward algorithm (SMART) for a global supply chain management that also focuses on inventory optimization but considers transportation and inventory costs in its reward function.

3.2 Dynamic resource allocation problems

Topaloglu and Powell [107] work on a time-staged multicommodity flow problem in the context of airline fleet allocation. In this case, the commodities are the planes themselves and the allocation decision is whether to take a particular trip, awaiting the realization of the next demand at the trip's destination. They address this case by using linear and piecewise linear value function approximations, with the objective of maximizing expected profit. They are able to do this efficiently due to the network structure of their particular problem, which can be solved via flow augmenting paths.

3.3 Network flows problems with graph neural networks

Very little research has focused on exploiting the graph structure of a network flows problem using neural network approaches. Recently, Busch et. al. [108] have applied a message passing graph neural network to a network flows graph to classify malware and spam. The graph is a communication network graph and the flows are the content passed between each pair of sources. No mathematical or stochastic optimization is involved.

Cheung and Powell [34] explore a stochastic distribution problem called the multilocation inventory problem, where decisions must be made anticipating an uncertain customer demand forecast. They also use a multistage, multicommodity min cost network flows formulation. However, hard constraints to ship customer orders from a single warehouse are not imposed. Additionally, they analyze the effect of redundancy in the number of warehouses that can serve each customer. They find that, unlike standard distribution networks which assign one warehouse per customer, assigning at least two warehouses per customer implies a significant cost reduction in distribution.

4 Problem statement

In most supply chains, an important part of a distribution network is transporting the goods from warehouses to customers in an efficient and timely manner. A common requirement is that all of the items of an order must be shipped from the same location in one shipment. Given that customers tend to order multiple products at a time and that products may be scattered across warehouses due to the positioning of production plants, inventory management becomes hard to manage as the number of commodities and customer facilities increases. If there is more than one warehouse that can satisfy a customer's orders with its available inventory, it could be possible to reduce transportation costs by carefully assigning the best warehouse to be the shipping point of each order, in order to minimize the costly inventory relocation from a warehouse to another, known as interplant movements.

This work considers a distribution problem with uncertain demands and the characteristics described above, and will be referred to as the *shipping point assignment problem*.

The mathematical formulation is a multistage, stochastic optimization problem, with a second stage network flows optimization problem that may contain thousands of variables depending on the number of commodities being managed. Additional binary variables will be added to the traditional multicommodity min cost flows in order to enforce the consolidation of all the commodities required to satisfy an order. The constraints that use the binary variables will be referred to as order consolidation constraints. The underlying optimization problem is already NP-hard [8], and the stochasticity makes it an even more difficult problem. This kind of problem is typically approached either as a deterministic optimization problem, thus ignoring stochasticity, optimized based on a forecast, which does not directly optimize long-term expected reward, or otherwise is subjected to a relaxation in order to solve it as a convex optimization problem with random variables.

The aim of this work is to design an algorithm that can optimize a policy in an environment that simulates the shipping point assignment problem, using neural networks as value function approximators. The proposed solution should be benchmarked against state-of-the-art deterministic and greedy heuristic approaches to assess whether it is superior in terms of computational performance and average optimization cost.

4.1 Shipping point assignment problem

The shipping point assignment problem is a multicommodity version of a transportation problem with random demand variables and order consolidation constraints. The goal is to satisfy the demand balance constraints in a multicommodity flows network with minimal long-term operational costs over a number of timesteps.

This formulation is derived from the empirical observation that on some supply chains, there is more than one warehouse that can satisfy an order.

This results in a combinatorial set of decisions that must be considered at each timestep for all outstanding orders, since there are multiple order-to-warehouse assignments that may satisfy the supply and demand constraints. To make matters more complex, the solution to the combinatorial problem should be optimal not only for the visible planning horizon, but also for the long term objective of minimizing the operational costs for future orders.

More specifically, the distribution network in question is comprised of a set of warehouses W, a set of customers C and a set of commodities K. Each customer $c \in C$ is associated to an $F_c \subseteq W$ that contains the warehouses which are allowed to fulfill their orders. Each day, customers will issue new orders according to their demand probability distribution parameters. When an order is generated, it will be considered an *open* order, since it does not have a warehouse assigned as a shipping point yet. Orders are composed of a positive number of units of different commodities from a random, nonempty subset of K with a random number of units for each selected commodity, which are expected to be available completely from a shipping point in the timestep of departure to the customer. The demand distribution is assumed to be stationary. More details about the demand generating distribution will be specified in chapter 11.

A certain number of new open orders appear each day, and shipping decisions must be taken on these orders on the same day that they arrive. For simplicity, all orders are assumed to arrive at the first working hour of each day, and decisions need to be made before the end of day, before the orders of the next day arrive. When an agent assigns a warehouse as the shipping point of an order, that order becomes a committed order, and the decision cannot be changed in the future. The order consolidation constraints require that, if the flow x^k_{wc} is positive for an arc from warehouse w to customer c for commodity k, arcs $x^{k'}_{wc}$ for all other $k' \in K$ must also be positive. All other arcs to that customer's order must be zero.

Inventory can be moved between warehouses to ensure that each warehouse has the necessary materials to fulfill all customer orders, albeit with a high cost relative to inventory storage costs. To maintain supply-demand balance, inventories for each commodity are replenished according to a distribution at every step. Inventory movements are not modeled directly as decision variables in this scenario, rather they are automatically issued to satisfy the demand balance constraints at each warehouse as quickly as possible; they happen as a consequence of deciding the shipping point of each order. An agent that finds a policy for the shipping point assignment problem must choose shipping points for the orders generated each day so that service levels are as high as possible and with minimal inventory movements, thus reducing operational costs.

In chapter 11, the shipping point assignment problem will be formulated as a Markov decision process in order to train reinforcement learning agents and evaluate policies on it. Its implementation will be referred to as the shipping point assignment environment.

4.2 Mathematical formulation

The shipping point assignment can be expressed as an mixed integer linear program with a linear objective by modifying the traditional multicommodity network flows formulation to account for the order consolidation constraints. In order to achieve this, it is required to introduce indicator variables for the presence of positive flow on customer-incident arcs. Then, additional constraints are needed to model the interaction between arcs to the same customer. This formulation assumes the physical network for the problem has been expanded and duplicated for each commodity. First, if an arc from a warehouse to a customer has positive flow on any commodity, the solution to the problem should also have positive flow on the rest of the commodities for that warehouse-customer pair, and vice versa if the arc has zero flow. Second, if any of the arcs for a given warehouse has positive flow, the rest of the arcs corresponding to the other warehouses to that customer should have zero flow.

$$
\begin{aligned}
&\underset{x}{\text{minimize}} && \sum^{(i,j)\in A} x_{ij}^k c_{ij}^k && \text{(14a)}\\
&\text{s. t.} && && \text{(14b)}\\
& & \sum^{i} x_{in}^k - \sum^{j} x_{ni}^k = b_n^k && \forall n \in N; k \in K && \text{(14c)}\\
& & x_{ij}^k \leq x_{ij}^k l_{ij}^k && \forall (i,j) \in A_C; k \in K && \text{(14d)}\\
& & x_{ij}^k \geq l_{ij}^k && \forall (i,j) \in A_C; k \in K && \text{(14e)}\\
& & l_{ij}^k = l_{ij}^{k'} && \forall (i,j) \in A_C k; k' \in K && \text{(14f)}\\
& & \sum^{k} l_{ic}^k = k l_{ic}^0 && \forall (i,c) \in A_C; k \in K && \text{(14g)}\\
& & x_{ij}^k \geq 0, && \forall (i,j) \in A; k \in K && \text{(14h)}\\
& & l_{ij}^k \in \{0,1\} && \forall (i,c) \in A_C; k \in K && \text{(14i)}
\end{aligned}
$$

The equation above describes the optimization program for the shipping point assignment problem. The constraint set 14c corresponds to the mass balance constraints. The constraints in 14d and 14e bind the indicator variables. The constraints on 14f restrict the indicator variables from a warehouse-customer pair to be equal to all other commodity arcs corresponding to that pair. Finally, the 14g constraints specify that, for each warehouse-customer pairs, the sum of all indicator variables equals to the number of commodities if that warehouse-customer pair is present in the solution, and is zero for all other warehouse-customer pairs. Without this last constraint, multiple warehouses could incorrectly send flow to a customer node, so long as each contributes at least one unit of each commodity. Constraints 14h ensure flows are positive and 14i declares indicators as binary variables, only for arcs going to customers.

4.2.1 Linearization of quadratic terms

The integer program, as formulated on 14, is problematic for MILP solvers as the constraints defined in 14d, which are required to bind the indicator variables to the x_{ij}^k flows, contain a quadratic term between an integer variable and a binary variable. As a result, the program cannot be effectively relaxed into a convex linear program. Fortunately, this can be trivially resolved by rewriting the constraint in terms of the upper bound of the flows,

$$\begin{aligned} x_{ij}^k &\leq \bar{x} l_{ij}^k \\ x_{ij}^k &\geq l_{ij}^k \end{aligned}$$

where $\bar{x}$ is an upper bound on the flow. An easy way to calculate an upper bound in this case is to sum the balances of all the supply nodes. In practice, no arc will ever receive all flow, and thus a more sophisticated per-node upper bound could be calculated and potentially be more efficient to solve. In the present work, this was not implemented. However, any gains in performance for a branch and bound optimizer are uncertain and beyond the scope of this work.

4.2.2 High density demand assumption

For conciseness, an assumption is made in the previous formulation of the shipping point assignment problem. All orders must be comprised of at least one unit of each commodity in order for the sum of indicator variables to equal to k. In practice, however, it is extremely common for companies to have highly sparse product portfolios and customers to submit orders from only a subset of that portfolio. This assumption does not limit the method, it is only necessary to replace the k constant with the count of commodities presenting nonzero demand for each order. In this implementation, the demand generator respects the high density assumptions and only creates order vectors with positive values on all entries.

4.2.3 Simplification from other multicommodity flows problems

For the sake of comparing agents only in terms of their ability to choose shipping points that reduce transportation costs, this work incurs in some simplifications from a typical multicommodity flows problem. First, all arc capacities are set to a high enough number such that they are never a constraining factor. Also, inventory and demand generators are expected to be in sync so that the optimization problem is always perfectly balanced. Finally, all costs of the same kind (inventory, interplant and delivery) are set to the same value, instead of varying per warehouse-customer pair or other conditions that typically vary on a distribution network.

5 Motivation and justification

Work on multistage multicommodity flow problems has been scarce and, as revealed by the literature review, the problem of shipping point assignment, as posited here, has not been treated previously. More work on this area could be of benefit to supply chain management practices.

5.1 Innovation

Research related to multistage nonlinear optimization problems is limited due to the computational challenge of generating experience on NP-hard problems [38], and thus it is worth investigating new methods that *learn* better policies in these environments with good sample efficiency.

Concretely, the innovating factors of this work are:

1. The proposal of a novel, multistage stochastic optimization problem referred to as the shipping point assignment problem.
2. The implementation of an environment to simulate the shipping point assignment problem that enables the training and benchmarking of reinforcement learning agents.
3. The design of a relaxation of the underlying NP-hard network flow problem related to the shipping point assignment environment. This relaxation can be used as a reward signal to train a reinforcement learning agent.
4. The proposal of a novel reinforcement learning agent with a neural network as a value function approximator that can optimize a policy in the shipping point assignment environment.
5. The usage of a graph convolutional network architecture in the context of a stochastic nonlinear optimization problem.
6. A benchmark of greedy heuristics and state-of-the-art deterministic optimization methods against the proposed reinforcement learning agent in the multistage shipping point assignment environment.

5.2 Impact

Supply chains are ubiquitous in our modern economy. Large scale supply chains are challenging to design, maintain and manage, and typically require large teams of people using a variety of information management and optimization tools to be able to maintain service levels while keeping costs as low as possible. Specifically, in the case of distribution optimization, the large number of decision variables, coupled with the inherent uncertainty of demand requirements, and the short windows of action, force supply chain teams to make suboptimal operational decisions. Furthermore, the stochasticity and high dimensionality of the scenarios make it difficult to evaluate how much the decisions had cost in the long run.

Improving the toolset with which supply chain teams can automatically create strategies to fulfill demand with the available supply in the long run could help companies with large supply chains save time spent planning, reduce the costs of suboptimal decisions, and become more competitive by means of better service levels. All of these would have a direct impact in the retail cost of consumer goods. The environmental footprint of maintaining a supply chain network may also be reduced as a result. Additionally, having an environment to simulate an order management scenario with which to perform benchmarks can shed light on the magnitude of the potential gains and motivate further research.

5.3 Depth

The activities required to create a reinforcement learning agent for the specified environment are:

1. Design the environment that simulates the shipping point assignment problem, which includes: choosing the data generating distributions, defining the dimensionality of the network (number of commodities, warehouses, customers, length of the time window), determining the connections between sources and destinations, and determining the parameters for the distributions to be used.

2. Explore a number of candidate architectures for the value function approximation network, including linear models, multilayer perceptrons and graph convolutional networks.

3. Implement deterministic optimizers to represent short-sighted agents operating in the shipping point assignment environment.

4. Compare the novel agent with the deterministic and stochastic optimizers in a suite of experiments on different environment configurations in terms of their computational efficiency and average optimization cost.

6 Hypothesis

A deep reinforcement learning agent that leverages a graph neural network for value function approximation should be able to learn a policy on the shipping point assignment environment that is statistically superior to greedy heuristic-based policies and deterministic optimization-based policies, as measured by average cost and time per action.

7 Research objectives

7.1 General objective

To propose a novel reinforcement learning agent that learns a policy for the shipping point assignment environment and is statistically superior in terms of long term reward and time per action when compared to state-of-the-art deterministic optimizers and greedy heuristic-based policies on the same environment.

7.2 Specific objectives

- Design an environment that simulates a realistic scenario of an order management distribution problem.
- Review the existing literature for strategies used to approximate policies on environments with similar characteristics.
- Design novel methods based on neural network models that can make decisions in the proposed shipping point assignment environment.
- Implement baseline deterministic agents that can make decisions in the proposed shipping point assignment environment based on the deterministic time window.
- Implement baseline greedy heuristic-based agents that can make decisions in the proposed shipping point assignment environment in a short amount of time.
- Execute a suite of experiments to be able to empirically assess the performance of the novel, neural network-based agent with respect to the baseline agents.
- Perform an analysis of the experiment suite results in order to derive conclusions about the best suited method to optimize an environment with the characteristics of the order management scenario proposed.

8 Scope and limitations

It is within the scope of this work:

1. The study of the performance of different implementations of decision agents in a multistage, multicommodity order management scenario assuming stationary demand generation distributions and fully balanced network flow problems.

2. Experimentation with deep neural network architectures, especially graph neural networks as potential value function approximators for multistage stochastic optimization problems.

3. The design and implementation of a deep reinforcement learning agent that may be used to optimize policies in an environment with a non-convex, expensive-to-compute reward signal.

4. The comparison of state-of-the-art deterministic and stochastic optimization methods as decision-making agents in non-convex stochastic optimization environments.

It is not within the scope of this work:

1. To study convex stochastic optimization problems that can be solved as linear programs with stochastic variables.

2. To study convex or non-convex stochastic optimization problems that are not represented by a network flow structure.

3. To study multistage stochastic optimization problems where the random variables present non-stationary distributions.

4. To consider similar supply chain scenarios and environments that do not share the same assumptions as the one proposed: orders must be delivered in full, there is a limited visibility horizon, there are no fractional flows, etc.

5. To use of stochastic integer programming techniques to solve the shipping point assignment problem.

6. To consider other supply chain scenarios such as: inventory optimization or vehicle routing or network design.

7. To apply non deep learning-based value function approximation methods.

8. To assess the feasibility of the proposed reinforcement learning method in a real world scenario.

9. To consider non-stationary demand generating distributions.

9 Deliverables

9.1 Literature review

A non-exhaustive overview of reinforcement learning and stochastic optimization methods for solving non-convex optimization problems with random variables and high-dimensionality in the decision variables. Furthermore, the following questions will be addressed:

1. What are the major challenges in optimizing multistage, non-convex stochastic optimization problems?
2. What are the most common approaches in the literature to optimize similar problems?
3. How can advances in deep learning and reinforcement learning methods help address the challenges in the field of stochastic optimization?

9.2 Optimization environment

The design and implementation of a novel reinforcement learning environment based on a multistage, multicommodity flows optimization problem with which to benchmark reinforcement learning agents. The environment in question must solve the challenge of efficiently generating a meaningful reward signal without exactly solving the underlying NP-hard optimization problem.

9.3 Reinforcement learning agent implementation

The design and implementation of reinforcement learning agents that uses a deep learning architecture for value function approximation of the proposed optimization environment.

9.4 Deterministic agents implementation

The implementation of baseline agents based on deterministic optimization to compare against the proposed reinforcement learning agent.

9.5 Experiments

A set of experiments comparing the proposed reinforcement learning agent with the two baseline agents. The experiments should be able to conclusively determine which agent is better in scenarios that vary in:

1. The parameters of the data generating distributions.
2. The size of the underlying network flow architecture.
3. The number of degrees of freedom in the decision variables.

In order to correctly assess this, the experiments should measure:

1. **Average reward over time:** The reward signal accumulated by the agent, averaged over a number of episodes.
2. **Time per action:** The time taken by the agent to generate an action for the current state.

10 Methodology

In this chapter, we present the methodological aspects of the design and implementation of a set of experiments on an instance of the SPA problem. The chapter is structured as follows. Section 10.1 describes the simulation environment. Section 10.2 explains how agents interact with the environment. Section 10.3 presents types of agents used for modeling and simulating different approaches for solving the SPA. Section 10.4 frames the SPA as a Markov decision process (MDP). Section 10.5 enumerates the metrics for the experiment. Section 10.6 contains the environment and agent parameters. Section 10.6 presents the experimental technique. Section 10.7 presents the three parameter settings for each experiment scenario. Section 10.8 describes various design decisions and their motivation. Finally, section 10.8 describes the computing platform where the experiments were executed.

10.1 Environment

Algorithm 3 Shipping Point Assignment Environment

```
Precondition: The physical network $P$ with associated actions $A_P$
Precondition: Demand generator $DemandGen(P)$
Precondition: Inventory generator $InvGen(P, Inv, Open, Committed, o)$
Precondition: Min cost flows optimizer function $MCF(G)$
Precondition: Time expanded network generator from a state $TEN(S_t)$
Precondition: A decision agent $Agent$
  function Simulate
    $o_1 \sim DemandGen(P)$
    $Open \leftarrow \{o_1\}$
    $Committed \leftarrow \{\}$
    $Inventory \sim InvGen(P, \{\}, Open, Committed, o_1)$
    $S_0 \leftarrow \{Open||Committed||Inventory||P\}$
    $A_1 \leftarrow Random(A_P)$
    for $t = 1..T$ do
      for $j = 1..DailyOrders$ do
        $A_{t+1} \leftarrow Agent.act(S_t, A_t)$
        $o_t \sim DemandGen(P)$
        $Inventory \leftarrow InvGen(P, Inventory, Open, Committed, o_t)$
        $Committed \leftarrow Committed :: \{(o_{t-1}, A_t)\}$
        $Open.remove(o_{t-1})$
        $Open \leftarrow Open :: \{o_t\}$
        $S_{t+1} \leftarrow \{Open||Committed||Inventory||P\}$
        $R_{t+1} \leftarrow MCF(TEN(S_{t+1}))$
        $Agent.feedback(S_t, R_{t+1}, A_{t+1})$
```

The shipping point assignment environment is a simulation of a distribution network that follows the constraints of the shipping point assignment problem, where a decision agent relays the allocation choices made for a set of orders generated according to a distribution and then moves forward in time. This simulation is necessary, not only because distribution network data is typically proprietary, but also because of the multistage nature of the decision problem: actions taken at $t = 1$ affect the state of the network at $t = 2$ and so on. It is impossible to recreate a scenario based solely on historical data, because states are dependent on an agent's previous decisions.

Instead of solving the full multicommodity flow problem at timestep t to derive the current cost of the shipping plan, it will generate a proxy problem that is easier to optimize. More specifically, it will generate $|K|$ time expanded networks containing all previously planned orders with their respective shipping warehouses, and the new order o_t scheduled on the current timestep with the warehouse A_{o_t} assigned by the agent.

This problem is simpler to solve on two accounts. First, the resulting optimization is decomposed into $|K|$ linear minimum cost flows, which can be solved in parallel and are significantly easier to solve than the full NP-hard multicommodity flows over time. Second, optimizing on orders where the warehouse has been decided means that the arcs from all the other warehouses to the order can be removed, thus removing constraints from the optimization problem.

The selected arc for each order will have a fixed delivery cost c_D if it is a valid warehouse for that order, or an arbitrarily large cost M for all other arcs to that order. If a decision agent chooses an invalid warehouse, in other words, a warehouse not in the customer's set of authorized warehouses, that arc to the customer will have cost M. Valid warehouses are determined at the beginning of the simulation for each customer. The environment then runs $|K|$ parallel min cost flows over time network optimizations assuming all shipping decisions are now fixed. The result of this optimization will be used as the reward signal for the agent. The pseudocode for the environment's simulation is presented in Algorithm 3.

This section presents a deep dive on the implementation details of the environment, including the simulation loop, the demand generator, the valid warehouse generator, the inventory generator and the process to create time expanded networks.

10.1.1 Simulation loop

The shipping point assignment environment implements the OpenAI Gym environment interface, composed of the step and reset functions. The reset function runs at the start of each episode and initializes all the environment data structures, such as the inventory vector, the open and fixed order lists, time counter and statistics accumulators.

The step function executes all the reactions of the environment to an agent's action to the last state. Concretely, the process it follows is:

1. Assign the warehouse chosen as an action by the agent to the next open order in the open order queue.

2. Move the open order to the fixed orders list.
3. Generate the k TENs and execute the min cost flows optimizations.
4. If there are no more open orders, call the step update routine.
5. Generate the next state data structure and statistics.
6. Return reward, current state and statistics.

The step update routine is called at the end of a timestep, which happens when no more open orders are available. It executes all the actions necessary to move to the next timestep by removing orders that are now in the past, generating new open orders, and generating inventory to preserve the mass balance constraints. The routine is as follows:

1. Calculate the consumed inventory, the inventory of the orders that will be delivered in the current timestep t.
2. Call the order generator to generate new open orders.
3. Call the inventory generator for the new open orders, ensuring that the balance of the problem remains zero.
4. Update the inventory vector as

$$Inventory = Inventory + newInventory - consumedInventory$$

5. Set $t = t + 1$

10.1.2 Demand generator

The demand generator is designed to generate orders that have correlation across commodities for each customer. It uses a Gaussian copula to generate the parameters for a geometric distribution, which will yield order vectors of size K with correlated demands for each commodity.

The full algorithm for the order generator is shown in Algorithm 4. First, we generate covariance matrices for each customer as a $(|C|, K, K)$ cube using an inverse Wishart distribution, as well as a vector of commodity means sampled from a Poisson with a parameter $\lambda/|K|$, where λ is set by the user beforehand. Therefore, λ controls the total volume of the demand across all commodities. Then, at each step of generation, a subset of Φ customers are chosen uniformly to generate orders. A multivariate normal is sampled with the covariances for each customer and a vector of zeros for the means. The probability density of the outcome is calculated from a normal distribution with mean zero and variance equal to the square root of the diagonal of the covariance matrix. Finally, this probability is passed into a geometric distribution with parameters $p = pz * (1 - pz)$, where $pz = 1/commodityMeans$ in order to obtain the demand vector of size K. Finally, the initial shipping point of the order is chosen uniformly and the order is scheduled to be delivered at the last day of the planning horizon, to simulate its appearance in the agent's visibility.

Algorithm 4 Order Generator

```
  function OrderGen
Precondition: $K$,$W$,$C$,$F_c$,$\lambda$,
    $customerCovariances \sim \mathbf{InvWishart}(K, [1\ \forall\ k \in K])$
    $\lambda \leftarrow DemandMean/K$
    $commodityMeans \sim [\mathbf{Poisson}(\lambda) \forall k \in K]$
    while $Simulating$ do
        $C' \leftarrow \mathbf{UniformSubset}(\mathbf{C})$
        while $c \in C'$ do
            $Cov \leftarrow customerCovariances[c]$
            $ZeroMu \leftarrow [0\ \forall\ k \in K]$
            $mnx \leftarrow \mathbf{Multinorm}(\mu = ZeroMu, \sigma = Cov)$
            $px \leftarrow \mathbf{NormalCDF}(\mu = 0, \sigma = \sqrt{\mathbf{Diag}(Cov)})(mnx)$
            $pz \leftarrow \frac{1}{commodityMeans}$
            $OrderDemand \leftarrow \mathbf{GeomPPF}(p = pz * (1 - pz))(px)$
            $InitialShipping \leftarrow \mathbf{Uniform}(\mathbf{Valid}(\mathbf{W}, \mathbf{c}))$
            $Delivery \leftarrow t + HorizonLength - 1$
```

10.1.3 Valid warehouse generator

The valid warehouse generator determines which warehouse can serve a specific customer. At the beginning of the simulation, it generates a matrix of dimensions $(|C|, |W|)$, with F_c elements per row set to 1 and the rest to 0. For each customer row, entries with a 1 represent valid warehouses for that customer.

10.1.4 Inventory generator

The inventory generator is based on a Dirichlet distribution $\mathbf{Dir}(\alpha)$. A total of $|K|$, with α being a $|W|$ dimensional vector calculated as:

$$\alpha = \frac{|W|}{[1, 2, \ldots, |W| + 1]}.$$

Thus, the generator will yield $|K|$ samples of $|W|$, each summing to one. The choice of α is meant for some warehouses to have a higher concentration of a commodity than others, thus simulating that some warehouses are closer to the commodities' production plants than others. This α favors concentration towards the lower ID warehouses, but since the samples are independent, the warehouse of highest concentration may differ from one commodity to the next. This clear inventory allocation strategy makes the inventory generator more realistic and ensures that there is a clear pattern for the RL agent to identify. As an illustration, the following is a sample for a 10 commodity setting with five warehouses:

$$\begin{bmatrix} 0.5000 & 0.1300 & 0.0300 & 0.0800 & 0.2500 \\ 0.3100 & 0.1700 & 0.3500 & 0.1700 & 0.0018 \\ 0.4700 & 0.4000 & 0.0680 & 0.0540 & 0.0087 \\ 0.3700 & 0.3900 & 0.0780 & 0.1200 & 0.0360 \\ 0.3300 & 0.3900 & 0.0260 & 0.1800 & 0.0740 \\ 0.4100 & 0.1400 & 0.1400 & 0.0830 & 0.2200 \\ 0.5000 & 0.0880 & 0.2400 & 0.0290 & 0.1400 \\ 0.5300 & 0.2400 & 0.0790 & 0.1500 & 0.0019 \\ 0.6100 & 0.0390 & 0.0270 & 0.2700 & 0.0490 \\ 0.6400 & 0.0800 & 0.1900 & 0.0740 & 0.0190 \end{bmatrix}.$$

In this example, seven out of the ten commodities will be more concentrated on the first warehouse, two will have more inventory on the second, and one on the third warehouse. Also, the degree of skew between the warehouses varies between commodities.

The Dirichlet generator is called once at the beginning of the simulation to generate the proportions of each commodity in each warehouse. When the inventory generator is called, it takes the sum of units for each commodity generated by the last call of the demand generator and distributes them according to the established proportions.

10.1.5 Graph generation

The physical network, depicted in Figure 8, is comprised of the warehouse and customer facilities, as well as the valid arcs between them. The information of the current state of the distribution network can be converted into a time expanded network to create a multicommodity network flow problem representing the transportation of goods in a planning horizon to satisfy current demands, as seen in figure 9. The environment is responsible of generating $|K|$ single-commodity min cost flow graphs from the multicommodity TEN so that the min cost flow solver can assess the current state of the network given the allocation plan.

For simplicity, all costs on transportation and inventory arcs are fixed, with inventory costs significantly lower than transportation costs. This implies that it is desirable to deliver the orders with as little relocation of inventory as possible, since allocation on the wrong warehouse will create flow between transportation arcs and drive costs up.

Scheduled orders in the generated graph will have a single incoming arc from the chosen warehouse. If the scheduled warehouse belongs to the subset of warehouses F_c, then the cost of this arc will be a fixed low delivery cost, otherwise, the cost of the arc is a sufficiently large M to represent an illegal movement (or an unfulfilled order). This is to motivate an agent to learn which warehouses are desirable without being explicitly aware of the set F_c. As an illustration, the TEN in Figure 9 corresponds to the physical network of Figure 8 with some orders. Warehouse nodes are replicated for timesteps 1, 2 and 3.

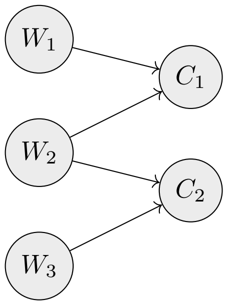


Figure 8: Physical Network Example

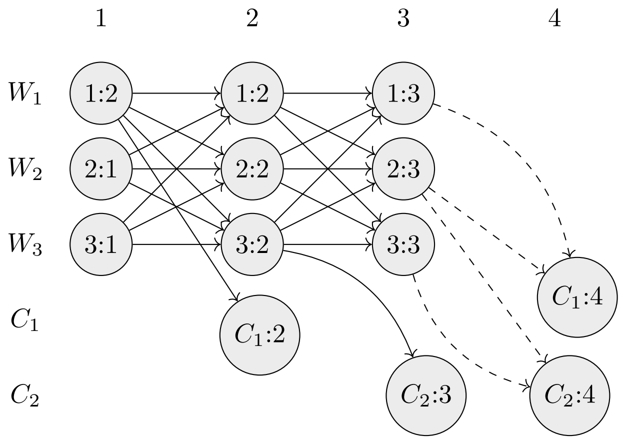


Figure 9: Time expanded network

C_1 has an order due on timestep 2, which is already allocated to be delivered from W_1 on timestep 1. C_2 has an order due on timestep 3, allocated to be shipped from W_3 in timestep 2. Finally, both customers have open orders due on timestep 4, which may be allocated to either of the warehouses as specified in the physical network of Figure 8 (W_1 and W_2 for C_1, and W_2 and W_3 for C_2).

10.2 Interaction with the environment

The agent that interacts with this environment is required to allocate each order to any of the existing warehouses, at the given timestep. This means that the action space is all the possible permutations of warehouse-customer allocation pairs possible for the generated orders.

The cardinality of the action space depends on how many orders are generated per timestep. The agent acts on the first order from the list of open orders by selecting a warehouse. This action is then communicated to the environment, which advances to the next state, calculates the changes in the network and inventories, and emits a reward.

10.3 Agents

In this section, the agents for the SPA environment will be presented. In total, six agents were implemented: a random baseline, a greedy heuristic agent, two agents that use deterministic optimizers, and two that use neural network agents.

Throughout this work, when referring to deterministic agents, it is understood to refer to B&B and Lookahead. Likewise, when referring to neural network or deep reinforcement learning agents, it is in reference to the MLP and GNN agents.

10.3.1 Random

The Random agent will simply make a random decision at each timestep from the list of valid warehouses for each customer. It is meant to be a baseline to see how better the other agents are when compared to a coin toss.

10.3.2 Best Fit

The Best Fit agent will follow the heuristic of choosing the warehouse that has the highest availability of the products required to fulfill the current order. This is meant to represent a typical behavior of a human making decisions in a supply chain scenario. It does not have any regard for the future impact of that decision, the expectation is that warehouses with more inventory will be less likely to cause problems later.

10.3.3 Lookahead

The Lookahead agent uses the same min cost flows solver as the environment to evaluate each possible warehouse for the current customer. It solves a total of $|F_c|$ minimum cost flows per action. This can be interpreted as a heuristic to avoid the complexity of exhaustively considering all possible combinations of warehouse and customer for all open orders. This combination of a deterministic optimization with a relaxation heuristic is a tradeoff between very fast heuristic agents that do not directly optimize a metric, and more expensive deterministic optimizers that solve the full problem on a planning horizon.

10.3.4 Branch and bound

The branch and bound (B&B) agent uses a general purpose branch & bound solver for Mixed Integer Linear Programs (MILPs). It uses the MILP formulation of the SPA in order to arrive at the optimal solution for the current planning horizon, without taking into account any stochasticity.

This deterministic optimization problem quickly becomes hard to solve exactly, as for every timestep with orders, potentially all combinations of orders and shipping point assignments must be considered. Even then, at every timestep, the optimization algorithm will only decide locally based on the visibility of the order demand window. It is possible that the linear relaxation of the order consolidation constraints are useful enough that not all combinations of warehouses and orders need to be evaluated, which would reduce overall runtimes.

Given that B&B is exhaustive, this agent is expected to outperform Lookahead in terms of average cost when the number of orders per day Φ is greater than one. It is also expected to be significantly slower because of two reasons: the exhaustive search and the fact that the B&B solver is not a specialized solver for minimum cost flows.

The problem will be solved using the Google OR Tools optimization library [109] with the SCIP solver for Mixed Integer Programs[110]. The problem's constraints are the same as in classical multicommodity flow formulation, with added nonlinear constraints for restricting customer demands to be satisfied using the same warehouse arcs for all commodities.

10.3.5 Multilayer perceptron (MLP)

The first deep reinforcement learning agent uses Q learning with a multilayer perceptron (MLP) for value function approximation, hence its name. The MLP consumes a feature vector that describes the current state of the MDP, such as the inventory levels and the next order to decide upon, and has an output shape of $|W|$ for the Q values. This agent leverages advances in deep reinforcement learning techniques to make learning more efficient. More details about the network's architecture, feature design and deep RL techniques applied are discussed later in the design decisions section.

This agent is meant to serve as a baseline for other reinforcement learning agents, as it is unclear that the simplistic representation of the environment via the hand-crafted feature vector is sufficient to derive useful policies for the SPA environment.

10.3.6 Graph neural network (GNN)

The GNN agent is similar to MLP, but instead of using a Multilayer Perceptron for Value Function Approximation, it generates graph-level embeddings using a graph convolutional network (GCN) [90]. The graph convolutional layers generate node level embeddings, which are then pooled and passed to a fully connected layer with output size $|W|$, corresponding to the Q values for each possible action.

10.4 Markov decision process

Here we express the shipping point assignment scenario as a Markov decision process, in order to be able to solve it with a reinforcement learning approach.

State: A state is described by the information of the physical network, the available inventory at each warehouse at timestep t, the open orders at timestep t and the committed orders at timestep t.

Actions: The agent acts by choosing a shipping point from the set of warehouses W for the next upcoming open order o_t. For this reason we use the notation a_{o_t} to refer to the action for the order o_t Notice that, according to this action space definition, it is technically possible that the agent chooses to ship the order from an unacceptable warehouse for that customer, which would result in a high cost penalty, also known as a Big M cost. In the implementation, however, most agents will have mechanisms that prevent them from choosing such actions.

Reward: The negative cost of the solution found for the $|K|$ linear min cost flow optimizations with the fixed shipping points, delivery dates and arc capacity distributions specified by the agent's actions.

As portrayed in Figure 10, the environment contains three components that comprise the state abstraction: a physical network (such as the one on Figure 8), an order generator that creates the next step's new orders and a time expanded network representation of the current state. For calculating the R_{t+1}, the environment runs $|K|$ min cost flow optimizations, which are equivalent to $|K|$ linear programs.

10.5 Metrics

10.5.1 Average cost

The most direct way of comparing the agents is by analyzing the average optimization cost per timestep over a series of simulations for each agent. The deep RL agents do not optimize directly against this metric for reasons of numerical stability, but a better long-term policy should reflect a decrease in optimization cost.

10.5.2 Time per action

Time per action (TPA) is the time in milliseconds taken by the agent to generate an action for the next open order. With a sufficiently large network flows problem, the deep RL agent should always outperform deterministic agents in this metric, since the latter has to run a full branch & bound or a large linear optimization for each action. This metric does not include the time it takes for the single commodity optimization done by the environment to generate the reward function.

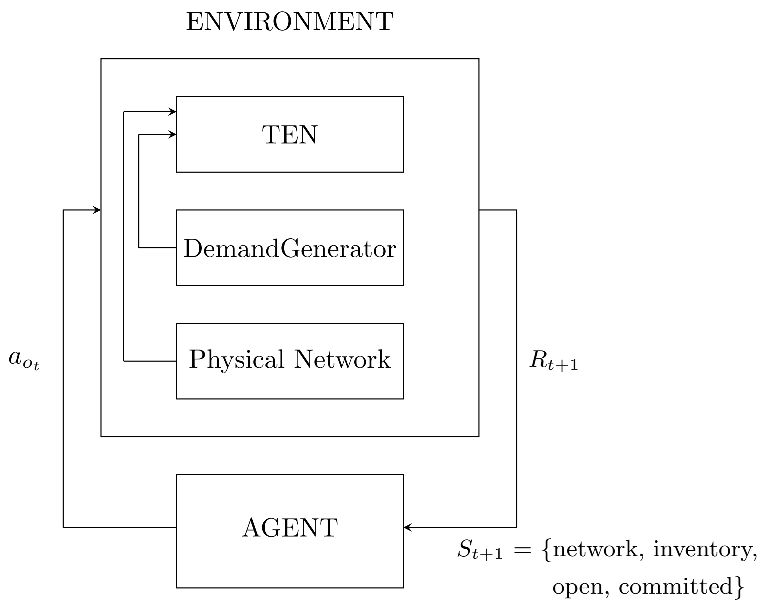


Figure 10: A graphical representation of the Markov Decision process. The agent selects a warehouse to ship the next order o_t based on the current state, the environment simulates its impact on the network by optimizing the TEN that represents the distribution problem, and then communicates the reward R_{t+1} to the agent.

10.5.3 Average reward

The reward function acts as a proxy signal to give feedback to the RL algorithms of the quality of their actions. It can be useful to observe this metric to compare each RL agent, and evaluate the effectiveness of the reward function choice. Similarly to the cost metric, the expectation is that deep RL agents outperform deterministic optimizers and greedy policies in this regard, due to effect of the long-term cost optimization.

10.5.4 Interplant movements

A proxy indicator that an agent is making decisions that consider future scenarios is interplant movements. Interplant movements refer to the number of items moved from a warehouse to another in a timestep. If an agent doesn't plan for future orders, it may use up inventory in warehouses that will need it in the future, requiring to incur in additional interplant movements that could be avoided. Although this measure is indirectly explained by the reward, it is more interpretable since it shows improvement in terms of material units moving through the network.

10.6 Parameters

10.6.1 Environment parameters

Physical network parameters. These are the parameters that influence the physical characterization of the distribution network. It includes:

- W: The set of warehouses
- C: The set of customers
- K: The set of commodities
- F_c: The set of valid warehouses per customer
- l: Lead time per order
- Φ: Number of orders per day
- H: Planning horizon length in timesteps
- ι: Interplant arcs cost
- κ: Warehouse to customer arc cost
- ζ: Inventory storage cost

Distribution parameters. The parameters to the data generating distributions for the demand component of the environment:

- λ: Poisson parameter for the customer means.
- α: Dirichlet parameter to distribute the inventories in the inventory generator.

Reinforcement Learning parameters. Parameters related to a reinforcement learning optimization, including:

- Number of episodes
- Number of steps per episode

10.6.2 Agent parameters

Greedy heuristic agents do not have any parameters to configure. The parameters in the B&B optimizer were left to the defaults of the library and thus are not presented. In the case of the Q learning-based reinforcement learning agents there is the following set of tuneable hyperparameters:

- Learning rate
- Discount factor
- Warmup steps
- Batch size
- Replay buffer length

10.7 Experimental technique

The experiments that will be proposed should be sufficient to answer the following questions about each of the proposed agents in the order management scenario:

1. Does the size of the physical network affect the agent's performance?
2. Does the number of commodities affect the agent's performance?
3. Does the total number of possible actions (decision variables) affect the agent's performance?
4. Do agents that optimize against a long term goal perform better than a short-sighted, deterministic agent?
5. How do the agents perform in intractably large optimization problems?
6. If the agent learns, how quickly can it start making reasonable decisions in the environment?

Each of the proposed experiments will modify the environment construction parameters to simulate different realistic supply chain scenarios: number of warehouses, number of customers, number of new orders generated per day, number of distinct commodities, number of available warehouses for delivery per customer. This will make the underlying optimization problem progressively larger in terms of constraints and variables. In order to reliably evaluate the performance of the agents, their average performance after 250 episodes of training will be reported. The expectation is that as problems become larger, deterministic optimization becomes progressively slower. Learning agents might also begin to struggle when the number of random variables increase.

The data generating distribution's parameters will be held constant across all episodes on all experiments. Although non-stationary processes are important in real world scenarios, they are outside the scope of this work.

The environment, agents, and simulation environments will be implemented using Python, with PyTorch leveraged for gradient-based methods, and Numpy for data generation.

10.8 Experiments

Description	$\mathbf{\|W\|}$	$\mathbf{\|C\|}$	$\mathbf{\|K\|}$	$\mathbf{\Phi}$	$\mathbf{F_c}$	$\mathbf{w^o}$
Small	3	16	1	1	2	1
Medium	8	32	8	3	2	9
Large	8	64	16	5	3	125
Huge	16	128	32	7	4	2401

Table 3: Variable experiment parameter settings

Parameter	**Parameter kind**	**Value**
Episodes	RL	250
Steps per episode	RL	30
Learning rate	RL	0.009
Discount rate	RL	0.90
Warmup steps	RL	32
Batch size	RL	32
Buffer length	RL	90
Planning horizon	Network	7
Demand mean	Network	500
Demand variance	Network	150
Big M Factor	Network	10000

Table 4: Constant experiment parameters

In this section, each experiment scenario will be described, along with its design motivations. Four scenarios with increasing sizes in parameters will be evaluated. Table 3 shows the varying parameters: number of warehouses $|W|$, number of customers $|C|$, number of commodities $|K|$, new orders appearing per day Φ, warehouses per customer F_c. An important number to have a notion of the complexity of the problem is the number of shipping assignment variations, which depends on the total number of warehouses and the number of orders to be considered at each timestep, denoted in the table by w^o.

Originally, the number of commodities on the largest scenario was planned to reach 64, however, this proved to be too expensive to compute for the linear optimizer, and the execution time for 30 runs of 5 different agents in such a scenario proved to be infeasibly long for the project at hand. The parameters Φ and F_c were also reduced in size in order to keep runtimes in a reasonable range.

It is also worth noting that, although the largest scenario is termed "huge" due to its computational complexity, a product portfolio of 32 is actually considered very small. Many companies have portfolios of hundreds, even thousands of SKUs. Such environment configurations would be tractable if the linear program were to be solved once per day, as in real world scenarios. However, for simulating runs over many days, as in this experiment setting, it is infeasibly long.

Table 4 shows the set of hyperparameters that remain constant, for reinforcement learning training or network problem graph generation.

10.8.1 Small

This is the smallest possible scenario, with only one commodity, three warehouses, sixteen customers and only one customer arriving per day, with two possible warehouses to choose from. The goals of this scenario are to have a baseline of run times when constraints are low and to understand whether it is possible to learn long term assignment policies in a low complexity scenario.

10.8.2 Medium

The medium scenario, with eight commodities, eight warehouses and three orders per day, is at least eight times larger in the linear optimization program to solve at each time window. The number of customers also increases, which will result in higher overall demand variance. This scenario will showcase the impact of having multiple commodities enter into play and its impact on the deterministic optimizers. In the case of branch and bound, the optimizer would, in the worst case, visit nine nodes in order to arrive at the optimal deterministic solution.

10.8.3 Large

The large scenario doubles the number of commodities again, and increases the number of orders per day to five. There are also three warehouses out of eight to choose for each customer. With 125 possible combinations of orders to warehouse assignments, the branch and bound and lookahead optimizers are expected to receive a performance hit. The linear optimizer from the environment simulator should also start seeing significant slowdowns.

10.8.4 Huge

The huge scenario doubles the commodities, increases the total number of warehouses to 16, and the valid warehouses per customer to 4. There is very high variability in the demand due the 128 customers. The linear optimizer of the environment also becomes a bottleneck for this experiment, as it is required to solve 32 linear programs of considerable size on every simulation step. The deterministic optimizers have to visit 2401 possible combinations on the worst case to arrive at a solution, which means that their TPA is expected increase significantly or they may not be able to reach a solution at all in reasonable time.

10.9 Design decisions

This section covers several design decisions that had to be addressed before implementing and executing the experiment suite.

10.9.1 Handling of Big M Actions

For implementation purposes, the action space for all orders is given by all the warehouses in the system. However, each agent can only be realistically served from a subset of these. For instance, serving to a customer hundreds of kilometers away would be infeasibly expensive. These actions will be referred to as "illegal actions" for a customer. From an optimization standpoint, this is managed using Big M costs. Illegal actions are no problem for heuristic-based agents or deterministic optimizers, as they will naturally avoid these actions.

For a deep reinforcement learning agent, however, the output layer is the same for orders of all customers, so it must be of size $|W|$.

Originally, it was intended to leave it to the reinforcement learning agent to learn to avoid illegal warehouse-customer pairings by means of very low rewards. Preliminary experiments showed that even with careful reward function design, which will be covered on the next section, this approach led to poor results, close to random choice. The reason is that the neural network has to handle two scales of rewards: when there are Big Ms in the system and where there are not. The neural network had to first explore the space of possible warehouses and learn which ones to avoid due to Big Ms, and by the point that it had learned it, the variations in cost were so trivial relative to big Ms that it tended to only choose one valid warehouse per customer, instead of adapting to the circumstance at hand.

The solution was to introduce a mask after the output layer to penalize the Q values, ensuring that a Q learning-based agent never chose an illegal action,

$$Q(S, A)_{masked} = Q(S, A) - F_c * BigM$$

where F_c is a vector of size $|W|$ with ones for valid warehouses of customer c. The Q Values for illegal actions will always be low enough that they will never be chosen by $\text{argmax}_a Q(S, a)$.

10.9.2 Choice of reward function

The reward function must convey information about the state of the physical network and the impact that an action has on it. The most intuitive way to model this is by taking the sum of the cost of the relaxed linear programs, since it summarizes the totality of all the flows going through the network and how good were the decisions that resulted in those flows. However, some challenges arise with this approach.

First, there are reasons unrelated to an action for why the optimization cost changes. For example, at every timestep, the planning horizon shifts, new orders appear into the picture, orders shipped in the previous step disappear from view, resulting in a completely different cost function every time. It is unclear if a reinforcement learning agent will be able to tease out the fluctuations in the cost due to the optimization problem's constraints from the impact that actions have on the long term reward gain.

Consider the following scenario to illustrate the above issue. On Figure 11, two time expanded networks are shown for a planning horizon of three, with two warehouses and a single commodity. On timestep one, W_1 serves an order of five units to C_1 on the current timestep, and another one on timestep three to C_2, also for five units. W_1 is also sending three units to W_2 to satisfy the requirements for C_3 on timestep three. On timestep two, the optimization window shifts, the order to C_1 is already in the past, as well as the three units moved from W_1 and W_2.

The shift from the planning window on $t = 1$ to $t = 2$ implies a change in the optimization cost for multiple reasons. Assuming costs per unit of $\kappa = 1$,

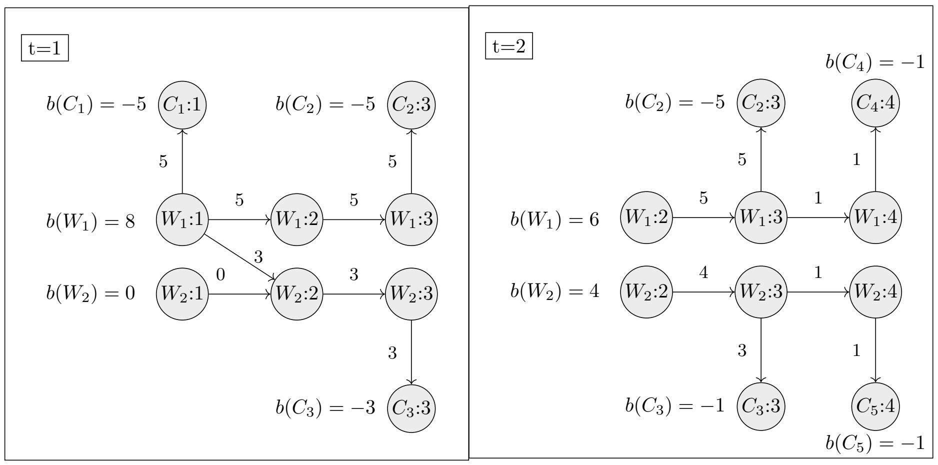


Figure 11: Sample Time Expanded Network for two warehouses, two customers in a time horizon of length 3

$\iota = 10$ and $\zeta = 1$ for warehouse-to-customer, interplant and storage, the total cost on $t = 1$ is given by $Cost_{t=1} = 5\kappa + 3\iota + 13\zeta$, while the cost at $t = 2$ is $Cost_{t=2} = 10\kappa + 0\iota + 8\zeta$. The cost increased simultaneously due to a decrease in total number of total order units, a decrease of interplants in the horizon, and a decrease in total storage costs. Out of these three factors affecting the total cost in a time window, the agent is only directly responsible for the interplant part. This poses a challenge for a reinforcement learning agent: distinguishing the causal component of its actions on a reward function affected by multiple factors.

A second challenge is that the cost-based metric might have issues of numerical stability. With a larger number of commodities or a larger number of average units generated per timestep, the reward values will vary over a larger range.

Given these challenges, five different reward functions were developed to compare the performance of Deep RL agents on them and choose the best one. When referring to the cost, the sum of the optimization cost for all commodities

$$Cost_t = \sum^{k} Cost_k(TEN(S_t))$$

is used as a shorthand.

1. **Simple negative cost:** The simplest reward function, with the drawbacks described above.

$$R_t = -Cost_t$$

2. **Diff negative cost:** As an attempt to have a reward that emphasizes the impact of an action on the optimization cost, the diff reward will give a

positive result if cost went down as a result of the last action, and negative otherwise.

$$R_t = (Cost_{t-1} - Cost_t)$$

3. **Negative log cost:** Log transform of the negative cost to have a more stable reward.

$$R_t = -log(Cost_t)$$

4. **Negative log cost or Big M (NLCOM)**: When a Big M action is chosen, it is carried over the horizon for many steps. This function attempts to avoid it by only including Big M costs on the penalty if they correspond to the latest order.

$$f(x) = \begin{cases} -log(Cost_t), & \text{if LastOrderBigM} \\ -log(Cost_t - BigMCosts), & \text{otherwise} \end{cases}$$

5. **Negative Log Cost Minus Big M (NLCMM):** This function intends to accentuate the benefit of having a Big M unit free solution. When the number of Big M units is zero, the function enters a higher reward space by means of $log(\epsilon)$, where epsilon is a small enough number.

$$-log(cost + \epsilon) - log(BigMUnits + \epsilon)$$

The functions described above were tested on single runs with the same parameters and RNG seed, before implementing the Big M mask filter described in the previous section. It was informally observed that NLCMM was most effective not only at reducing Big Ms, but also interplants and total cost . The simple negative cost, diff cost and log cost functions all performed significantly worse. The interpretation for this outcome is that, besides being effective for reducing Big Ms (in the absence of the mask described in the previous section) the reward space for zero big Ms, on the plateau of the log function, makes learning more stable. For these reasons, NLCMM was chosen for experimentation.

Although most reward functions tested successfully reduced Big Ms, the high variance between attained rewards with and without Big Ms hinders learning later on. This motivated the usage of the Big M mask filter in conjunction with the NLCMM reward function.

10.9.3 Choice of cost parameters

The cost parameters ι,κ and ζ have two purposes: to drive flows correctly from warehouses to customers and to convey information to the reinforcement learning agent via the reward function. In order to satisfy customer demands, all that is needed is to satisfy the condition that transport costs are higher than storage costs, that is, $\iota \geq \kappa$. However, for the reinforcement learning agent, it is helpful for the proportion $\frac{\iota}{\kappa}$ to be significantly high in order to incentivize reducing interplants as much as possible. For this reason, the values $\iota = 150$, and $\kappa = 1$ were chosen. $\zeta = 10$ just to represent the realistic assumption that transports to customers are relatively more expensive than storing inventory, but this has no impact on the decisions made by the optimizer.

A more realistic cost scheme was purposefully avoided, for example by sampling the costs for each warehouse-customer pair to represent variance in distances and transportation routes, in order to avoid confusions on the decisions of the optimizer and agent. It is worth noting that an advantage of this reward function/cost parameter framework is that it is possible to model any kind of scenario and decision tradeoff to convey to the agents by manipulating the cost parameters.

10.9.4 Episodic vs. continuous reinforcement learning

The shipping assignment problem lends itself naturally to be framed as a continuous RL problem. The experiments described previously were designed as en episodic task, with each task comprised of a certain number of simulation steps. However, this episodic simulation is only done to reset the order generation process every few steps, in case the agent falls into a series of wrong actions (analogous to the pole falling on the cartpole example). The Q learning agents presented, however, incorporate the $\gamma = 0.9$ discount rate on their formulas, so they are essentially treating all episodes in a simulation as one continuous stream of agent interaction steps.

The number of episodes per run was decided to be 250 based on informal observations from preliminary executions of neural network agents. The training loss typically stabilizes after 100 episodes on any environment size, with some slight variations in the range of 100 episodes until 250 episodes. After 250 episodes, no significant variation in agent's performance is expected to happen. Given that the first episodes of the neural network agents tend to perform poorly while the network adapts to the environment, they were trained for 275 episodes and the comparisons with other agents were made from episode 25 to episode 275.

10.9.5 Normalization of reported cost

Given that the demand generator is initialized at each run with different customer means, runs of the same environment can vary greatly in the average total demand observed. This makes run metrics not directly comparable. The cost metrics reported in the results section are normalized by the total demand observed in that step of the simulation:

$$NormalizedCost_t = \frac{Cost_t}{CustomerFlows_t},$$

where $Cost_t$ refers to the unnormalized cost observed at timestep t, and

Features	**Dimension**
Valid warehouse mask	$\|W\|$
Inventory consumption	$\|W\| * \|K\|$
Total	$\|W\| + \|W\| * \|K\|$

Table 5: Description of the feature vector for the multilayer perceptron VFA

$CustomerFlows_t$ is the sum of all flows inbound to a customer node in the TEN. Whenever not specified, when referring to cost in the results section, it is assumed that the metric is $NormalizedCost_t$. A similar treatment is given to the interplants metric.

The normalized cost can be interpreted as "the optimization cost of the whole system to satisfy the demand of one unit". In the case of interplants, how many units must be moved from a warehouse to another in order to satisfy the demand of one unit.

For illustration purposes, the reward metric is presented without any normalization. This highlights that the differences in the reward function when neural network agents are relatively small when compared to deterministic optimizers, even though the difference in real cost is significant.

10.9.6 Feature design

The input features used to feed the neural network VFAs varied depending on the agent. MLPs required a feature set that was rich enough to represent the whole state space, while GNNs already implicitly encode graph-based information, and the features are at the level of nodes, which represent warehouses and customers.

For MLPs, the feature set has two parts, the first will be referred to as the "valid warehouse mask" and the second one is the "consumption vector". The valid warehouse mask is a vector of size $|W|$ with a one indicating that W_i is valid and a zero if it is not a valid warehouse for the current customer. The consumption vector, of size $|W| * K$, is computed by subtracting the current order's demands to the current inventory vector, thus representing what the inventory would look like if the current order were served from that warehouse. The motivation behind the consumption vector is to illustrate which warehouses have currently deficient inventories for the current order's demands. Since the mask and consumption vectors have different scales, it is important to normalize them before passing them into the network. Before settling on the consumption vector as part of the feature set, having demands and inventories separated was also attempted. An example of the feature set can be visualized on Figure 5

GNNs do not require to encode information about valid warehouses, because they operate on the physical network graph, and the arcs already represent that information. Therefore, the feature set for GNNs is only a vector of size K representing the inventory on warehouse nodes and the demand on customer nodes. Additionally, two indicator features are added: one to indicate whether

Layer	**Activation**	**Input dimensions**	**Output dimensions**
Noisy Linear	Tanh	$\|W\| + \|W\| * \|K\|$	512
Noisy Linear	Tanh	512	256
Linear	None	256	$\|W\|$

Table 6: Multilayer perceptron architecture

Features	**Dimension**
Balance	$\|K\|$
Node type indicator	1
Current order indicator	1
Total	$\|K\| + 2$

Table 7: Description of the feature fector for each node in the graph neural network VFA.

a node is a warehouse or a customer, and another one to signal to which customer node the current open order belongs. A graph example with a feature set can be observed in Figure 7.

10.9.7 MLP architecture

The architecture for the MLP agent can be seen in Table 6. The network is a multilayer perceptron comprised of two hidden noisy linear layers with hyperbolic tangent activations and a linear layer, decreasing the layer size by a factor of two, with an output size of $|W|$. Some other architectures that were explored were a single layer linear model, a one layer wide neural network, and a three layer neural network. The two and three layer networks performed better than the other models tried, and so the simplest model was preferred. Sigmoid activations were also tried and observed to perform similarly.

10.9.8 GNN architecture

GCN was chosen for the graph convolutional layers of the GCN value function approximation because of its properties as they relate to the structure of the network graphs it consumes. The graphs across time are completely regular in structure and only vary in the node features, so the issue of changing eigenbasis is not a concern. Additionally, GCN provides good numerical stability due to its renormalization trick, which is important for the reinforcement learning use case. Although GCN is a spectral-based method, it can be reinterpreted as a spatial method due to its first-order approximation of the graph Laplacian matrix being equivalent to a node communicating its information with its neighbors.

The value function approximation of the GNN agent is a graph-level GNN with two GCN layers as defined in equation 13, with ReLu activations. The GCN layers are followed by a global max pooling operator and finally, a linear

Layer	Activation	Level	Input dimensions	Output dimensions
GCN	ReLu	Node	$\|K\|$	1024
GCN	ReLu	Node	1024	512
Pooling	None	Graph	$512 * (\|W\| + \|C\|)$	512
MLP	None	Graph	512	$\|W\|$

Table 8: Graph Convolutional architecture

output layer to transform the embeddings into the $|W|$ sized Q values, as seen in Table 8.

The global max pooling operator aggregates the node features into a single graph embedding by taking the channel-wise maximum per node feature. The max operation was chosen because of the ReLu activations. Since the graph consumed by the network is a bipartite graph, having more than two graph convolutional layers yields no benefit. Finally, no performance improvement was observed from adding more dense layers after the graph pooling.

10.9.9 Hyperparameter optimization

In order to decide the hyperparameters for the MLP and GNN agents, a grid search was performed to tune the learning rate, discount factor and number of units on the first layer on the GNN-based agent.

10.9.10 Deep reinforcement learning implementation details

A number of Deep RL specific optimizations were incorporated into the implementation for neural network based agents, inspired by DQN [101]. Specifically:

1. **Experience replay:** A replay buffer is kept with the experiences from the last 90 steps. This buffer is populated at the beginning of training by means of a random sample.

2. **Target network:** As in DQN, the targets are calculated using a separate copy of the neural network, which in this case is updated every 30 steps (every episode end).

3. **Noisy dense layers:** On MLP and GNN agents, all fully connected layers are implemented using noisy linear layers to improve learning stability and encourage exploration.

4. **Huber loss:** Instead of the typical MSE loss, Huber loss was chosen to mitigate the impact of outliers, which were expected to be common given the variance between the average customer demand sizes.

5. **Epsilon greedy exploration:** Regular ϵ-greedy exploration was used. Some preliminary experiments were made by attempting have a decaying ϵ parameter, but no significant benefit was observed.

10.9.11 Training algorithm

For the training of neural network agents, an Adam optimizer, with no tuning on the Adam parameters except for the learning rate. Additionally, a decaying learning rate scheduler with a decay rate of 0.999992 per epoch was incorporated. This was implemented after observing that the loss values tend to become unstable after several training episodes.

10.10 Execution platform

All non neural-network-based experiments were run using Google Cloud Platform's n1-highcpu-64 instances, which have Intel Xeon E5 processors with base frequencies of 2.3Ghz and 57.6GB of memory. Experiments were run on batches of 30 parallel processes, corresponding to all runs of an agent in one environment size.

Due to limitations provisioning instances with GPUs, neural network experiments had to be run on n1-standard-4 instances, which have the four cores of the same same Intel Xeon E5 processors, with 16GB of memory and an NVIDIA K80 GPU with 8vCPUs and 12GB GDDR5. Processes were limited to two per instance to avoid thrashing.

Although experiments were run on different machine configurations and different number of parallel processes, which could potentially affect execution time benchmarks, those differences should be negligible due to the fact that time is being measured using Python's time.process_time() function, which only measures time the process is actively using the CPU.

	Reward		Cost		TPA	
Agent	Mean	STD	Mean	STD	Mean	STD
Lookahead	9.41	0.78	6.94	1.55	4.50	2.48
B&B	9.39	0.80	7.17	1.66	20.34	127.66
MLP	9.26	0.82	8.05	2.08	4.67	2.22
Best Fit	9.24	0.82	8.12	1.94	0.22	0.10
GNN	9.15	0.83	8.84	2.31	5.81	1.76
Random	9.04	0.85	10.45	2.44	0.20	0.12

Table 9: Average metrics per agent on 30 runs of the small environment

11 Experiment results

In this section, each agent's performance will be evaluated for each of the environment sizes tested. The performance will be reported in tables showing the average reward, time and cost with one standard deviation over 30 instances of the simulation. Additionally, plots will be presented to illustrate the distribution of the runs. Finally, an informal exploration on the impact of environment size on the TPA performance of each agent will be presented.

11.1 Small environment

Table 9 shows the average metrics for 30 runs of the agents on the small environment. Figure 12 contains box plots to further explore the distribution of the metrics for each agent. Lookahead achieved the lowest average cost of all agents, followed by B&B, then MLP, Best Fit, GNN, and finally Random. Lookahead is very close to the B&B solution, as expected. This is because B&B optimizes for the only open order visible each day, which is equivalent to what Lookahead does as a heuristic. MLP, Best Fit, and GNN can be regarded as similar in terms of average cost. All of the aforementioned agents performed better than random.

With respect to time, Lookahead took almost a fifth of the time of B&B. On such a small environment, this can be explained because of the additional time that B&B takes to solve the linear programs using a generic solver instead of a network flows-specific algorithm, like Lookahead does. B&B also shows a high variance in TPA, which suggests that some of the linear programs generated may be simpler to solve than others.

Since MLP and GNN were not able to outperform the deterministic optimizers in this scenario, the results do not suggest that these reinforcement learning agents were not able to derive better long term policies that the short-sighted strategies of deterministic optimizers and greedy heuristic policies.

However, with such a small problem setting, it would not be advisable to derive any meaningful conclusions from the performance of the agents, neither with respect to time or overall cost.

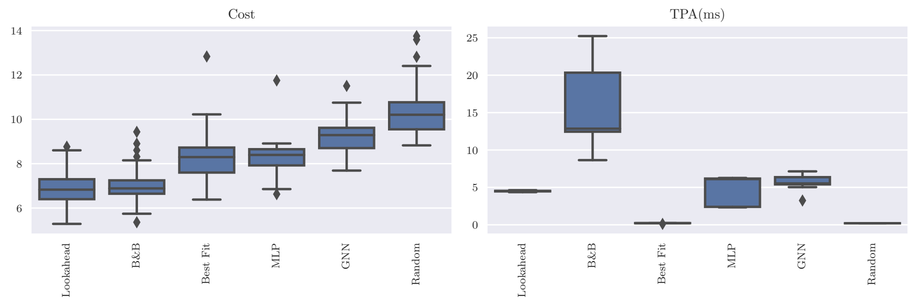


Figure 12: Box plots for cost and TPA per agent on 30 runs of the small environment

11.2 Medium environment

Table 10 and the box plots on Figure 13 show the average performance for each agent on the medium scenario. In terms of average reward and cost, Lookahead is the most effective agent, followed by B&B, then Best Fit. GNN outperforms MLP and Random, but is not near the results of Best Fit. MLP had an average performance worse than Random.

In terms of time, however, B&B and Lookahead are the slowest agents, with GNN being faster than B&B by almost two orders of magnitude. Best Fit, being a greedy heuristic agent, is of course faster than neural network agents, almost as fast as Random while achieving a good performance in overall cost. The neural network agents are at least an order of magnitude slower than Best Fit. Lookahead is the best agent both in terms of cost and time, although the small difference in cost with B&B might be attributed to random variations. Even then, it achieves so in almost a fifth of the execution time.

It is observed that the differences in average reward between the agents is negligible when considering the variance. This may be due to the log transformation to the cost applied by the reward function that was chosen, and might be a possible explanation why agents struggle to find better policies after achieving certain level of reward.

In terms of long term reward optimization, the reinforcement learning agents used in this scenario cannot be said to have performed better than any greedy or deterministic-based optimizer.

11.3 Large environment

The results for the large environment can be seen in Table 11 and the box plots of 14. In this case, B&B slightly outperformed Lookahead, which was expected, given the exhaustive evaluation of Lookahead. Next, Best Fit with a significantly higher cost, followed by Random, then MLP and finally GNN. It was unexpected to find MLP and GNN perform worse than Random in this environment. A possible explanation is that the hyperparameter tuning was

	Reward		Cost		TPA	
Agent	Mean	STD	Mean	STD	Mean	STD
Lookahead	7.97	0.11	7.73	0.57	74.68	13.21
B&B	7.94	0.10	7.98	0.70	349.24	160.78
Best Fit	7.83	0.11	9.00	0.82	0.27	0.02
GNN	7.77	0.11	9.64	0.97	4.10	0.52
Random	7.72	0.10	9.84	0.81	0.19	0.01
MLP	7.69	0.13	10.32	1.13	2.33	0.14

Table 10: Average metrics per agent on 30 runs of the medium environment

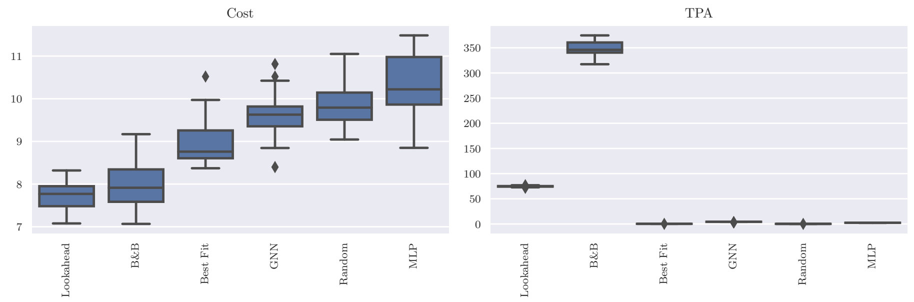


Figure 13: Box plots for cost and TPA per agent on 30 runs of the medium environment

	Reward		Cost		TPA	
Agent	Mean	STD	Mean	STD	Mean	STD
B&B	7.65	0.08	6.29	0.32	1366.75	534.37
Lookahead	7.64	0.08	6.38	0.32	370.65	106.90
Best Fit	7.38	0.10	8.43	0.62	0.20	0.01
Random	7.24	0.09	9.47	0.58	0.16	0.02
MLP	7.19	0.13	10.07	1.14	2.32	0.11
GNN	7.17	0.13	10.29	1.20	5.48	0.81

Table 11: Average metrics per agent on 30 runs of the large environment

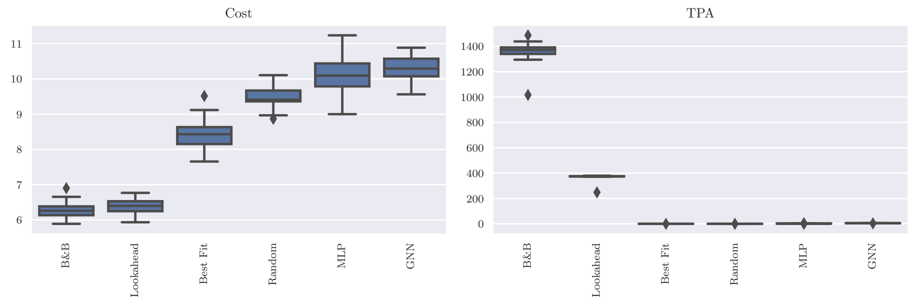


Figure 14: Box plots for cost and TPA per agent on 30 runs of the large environment

performed on simulations of the medium scenario, which has a different cost scale because of only having half the commodities. Training on a non-optimized set of hyperparameters resulted in instability during training.

In terms of TPA, as expected, greedy agents and neural network agents performed similarly as on previous environment. Lookahead and B&B solve times increased significantly with the larger optimization problem, with Lookahead taking 730ms per action on average, and B&B upwards of 3.6 times that, at 1366ms per action taken. The standard deviation of B&B runs is also very elevated: this happens because some simulation steps end up being easy to solve for the optimizer, while others require a more exhaustive search, resulting in high variability in solve times.

Although B&B was able to outperform Lookahead in this environment, it is difficult to justify the marginal reduction in cost for the massive increase in computation time. Best Fit is a feasible heuristic that achieves good performance for a minimal amount of computational effort.

Once again, the results do not suggest that the proposed neural network VFAs are able to learn a good long term policy for the shipping point assignment scenario.

	Reward		Cost		TPA	
Agent	Mean	STD	Mean	STD	Mean	STD
Lookahead	7.27	0.1	6.57	0.29	6043.83	137.81
Best Fit	7.03	0.09	8.34	0.47	0.34	0.01
Random	6.82	0.06	9.88	0.41	0.19	0.04
GNN	6.73	0.11	11.23	1.08	6.38	1.32
MLP	6.68	0.12	11.56	1.06	2.55	0.21
B&B	OOM	OOM	OOM	OOM	OOM	OOM

Table 12: Average metrics per agent on 30 runs of the huge environment

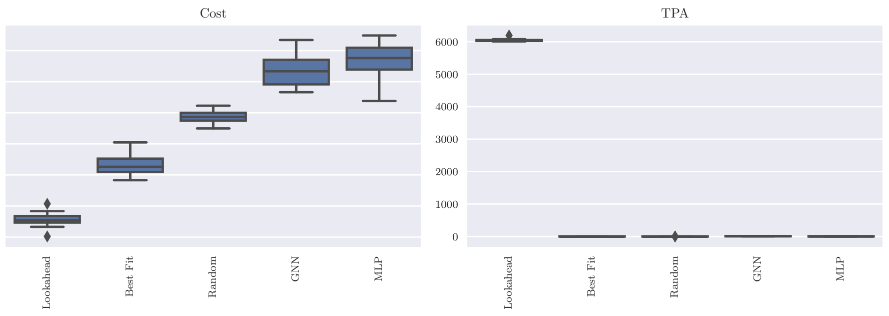


Figure 15: Box plots for cost and TPA per agent of the huge environment, considering only runs of at least 15 episodes without program failure.

11.4 Huge environment

The results for the huge environment are presented in Table 12 and in the box plots of Figure 15. However, these results are incomplete due to execution challenges associated to the scale of the experiment. Therefore, they must be analyzed with reservation.

The first execution challenge was associated to the environment itself. Some experiment runs on all agents failed. While the root cause is unknown, a likely explanation is that they are associated to a memory or garbage collection issue from the large number of objects generated during the environment's optimization step. Each run takes approximately five minutes per episode for greedy agents, and twenty six minutes for Lookahead. This translates to 20.8 and 108 hours of run time, respectively, making the experiments of the huge environment prohibitively costly to repeat. Thus, the decision was made to present partial results instead, with runs not containing the full 520 episodes. Additionally, for the runs to be more comparable, the results were reported on a cutoff of 100 episodes, because only a handful of runs managed to finish through the full 250 episodes.

The second execution challenge was that it was impossible to generate any data about the behavior of B&B. Informally, it was observed that some runs failed with out of memory errors, while others did not return results for even one simulation step after over four hours of execution. For this reason, the results on Table 12 show the legend “OOM” for the B&B agent.

The consequence of the partial results is that the reported means and standard deviations in reward, cost and TPA cannot be regarded as representative. With that limitation in mind, some clear trends that continue from previous experiments can be observed: Lookahead is the best performing agent in terms of cost, beating Best Fit by around 21%. Best Fit is also clearly superior to a random choice. Meanwhile, RL agents strayed further away from the random choice performance, relative to the results of the medium and large environments. It cannot be determined with certainty if GNN significantly outperforms MLP in average cost.

In terms of TPA, Lookahead is three orders of magnitude slower than all of the other agents. Best Fit and Random take under a millisecond to generate a decision, consistent with previous environments. MLP and GNN show only a slight increase in TPA. GNNs unsurprisingly require a slightly longer inference time than MLPs in this environment, as it scales with the number of nodes and arcs in the physical network graph.

11.5 Assessment of neural network VFAs

As observed on the results of small, medium and large environments , GNN agents can outperform random choice. On huge environments, both GNN and MLP perform similarly to a random choice. This evidence suggests that the NN-based VFAs may be encoding valuable information about the dynamics of the network, but that it struggles to scale as the optimization problem becomes more complex. The results on the large environment cast doubts about the robustness of the learning of the RL agents with respect to hyperparameter tuning. The results encourage further research to discover learning strategies using GNNs to solve challenging optimization problems.

It is important to highlight that agents that leveraged neural network VFAs struggled to surpass not only short-sighted deterministic optimizers, but even very simple greedy heuristics such as Best Fit. In the case of MLP, its performance can be said to be worse than a random choice. There are several factors that could have affected the neural network agents.

First, the results show that, due to the logarithmic scaling of the reward function, the performance of all agents in terms of average reward was virtually indistinguishable, as can be appreciated in the box plot of Figure 16 . This suggests that although the choice of reward function might have helped to stabilize learning, it made it increasingly difficult for the NN-based agents to distinguish the impact of one action from another on a given state. A better choice of a reward function should offer both numerical stability as well a good signal-to-noise ratio of the impact of an action on the environment.

Second, the fact that neural networks in the experiments do not manage to outperform deterministic optimizers in any environment, suggests that neural networks are struggling to learn both the constraints of the optimization problem

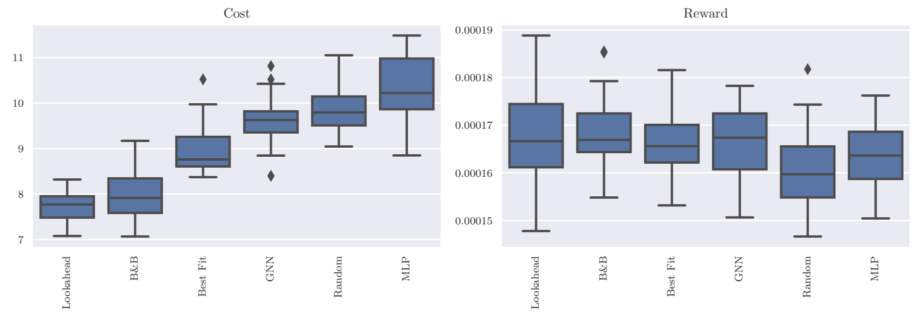


Figure 16: Comparison of box plots for cost and reward for the medium environment. While the difference in average cost is clear between agents, there is virtually no distinguishable difference in terms of average reward.

alongside the dynamics of the stochastic variables embedded in it. Although the network structure is encoded into the GNN, it still does not seem to be a sufficient representation to learn both the structure and the random variable distributions simultaneously. It must be necessary, via specialized layers or some other mechanism, to feed the prior information of the optimization problem into a neural network, so that the reinforcement learning agent can focus only on learning the optimal decision given the random variables.

Third, a more exhaustive treatment of neural network architectures and feature design might be useful to find better policies. For example, a few unexplored ideas are: node level embeddings instead of graph level GNNs, Graph Attention Networks, and using the greedy heuristic methods to generate neural network features.

Finally, the marginal improvements in cost from using Lookahead to B&B even on larger scenarios might be an indicator that the environments that were designed do not offer much room for long term optimization of cost. It remains an open question whether this phenomenon is a property of distribution networks in general or only of this particular simulation environment. Several improvements to the simulator are proposed in the Future Work section to shed light on this question.

11.6 Impact of environment size on time per action

As seen on each environment, TPA varies at different rates for each agent. Figure 17 and Table 13 illustrate how this growth happens at an exponential rate for deterministic optimizers, in line with expectations. Best Fit and Random's TPA remains constant with environment size. MLP also shows a relatively constant TPA. GNN's TPA grows slightly from medium through large. This is an expected consequence of the GCN convolution operation, which depends on the size of the physical network graph. There is a small, unexpected decrease in TPA for MLP and GNN when comparing the small and medium environments,

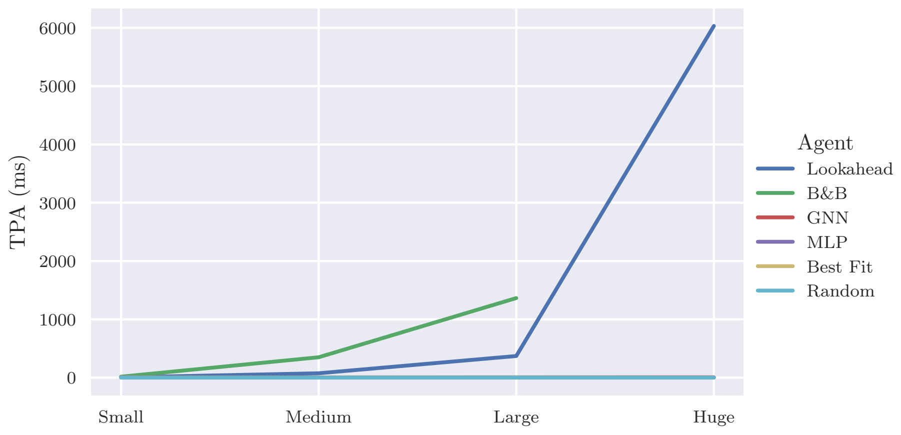


Figure 17: Growth of average episode time with environment size for each agent. B&B and Lookahead grow at a much higher rate than the rest of the agents.

Agent	Small	Medium	Large	Huge
Random	0.20	0.19	0.16	0.19
Best Fit	0.22	0.27	0.20	0.34
MLP	4.69	2.35	2.32	2.67
GNN	5.84	4.12	5.49	6.47
Lookahead	4.50	74.57	370.67	6032.39
B&B	16.35	349.34	1364.47	OOM

Table 13: Comparison of average TPA per agent on all environments

possibly due to random variations in memory transfer times between GPU and CPU. It is worth noting that B&B was not able to generate any results at all on the huge environment, hence the missing data point.

This view of the results highlights the relevance of finding good heuristic methods in order to solve larger scale instances of the SPA problem, as deterministic optimizers are evidently unable to scale at this rate to real world problems, while RL and heuristic-based methods present constant time.

11.7 Deterministic allocation environment

The environment simulator introduces variance to the optimization cost via two of its mechanisms: the inventory generator, and the demand generator. At the same time, an agent's policy is also a source of variance, as is illustrated by the difference in cost variances between agents: on the large environment, variance for B&B and Lookahead were much lower, and MLP showed an even higher variance than random. The high variance observed in average costs across all runs of neural network agents motivates the question: How much of this observed variance can be attributed to these two sources of uncertainty, and how much is due to the agent's actions? Furthermore, could the neural network agents' poor performance be attributed to some particular aspect of this demand generator?

To address these questions, a new scenario was designed by using the small environment parameters and modifying the generators to ensure that a deterministic mapping of orders to warehouses exists. Such a scenario enables a clearer comparison of agent performance, especially between random choice and the RL agents. Recall that the inventory generator distributes all of the available inventory based on a fixed Dirichlet distributed vector of allocations per warehouse and commodity. In the new scenario, the inventory generator deterministically allocates all inventory required to fulfill arriving orders into the valid warehouse with the lowest warehouse identifier, thus functioning as an oracle of the best action. This setting removes one source of uncertainty and guarantees that the optimal policy with minimal interplant movements on all windows should be easy to learn by always allocating each customer's order to the same warehouse. Notice that in this deterministic mapping, the decision is between the lower and upper bound on the cost for each optimization window: if orders are allocated to the optimal warehouse, the cost to fulfill that order will be $u * P_D$ and the interplants resulting from it will be 0, where u is the total number of units and P_D is the cost across all arcs required for that order to be delivered. On the other hand, any warehouse other than the optimal would in theory result in a 100% of the order's units being transported from a warehouse to another, for a total cost of $u * P_D + u * P_I$, where P_I is the cost of the path of interplants the inventory would take to satisfy that order. In practice, however, the interplants will not equal exactly 100% at any timestep, because the inventory is shared across all orders, and inventory of future orders can be used to satisfy part of the demand. Another consequence of this setting is that optimization of the long term cost is no longer relevant, since every order has a deterministic and locally optimal action to that time window that reduces costs

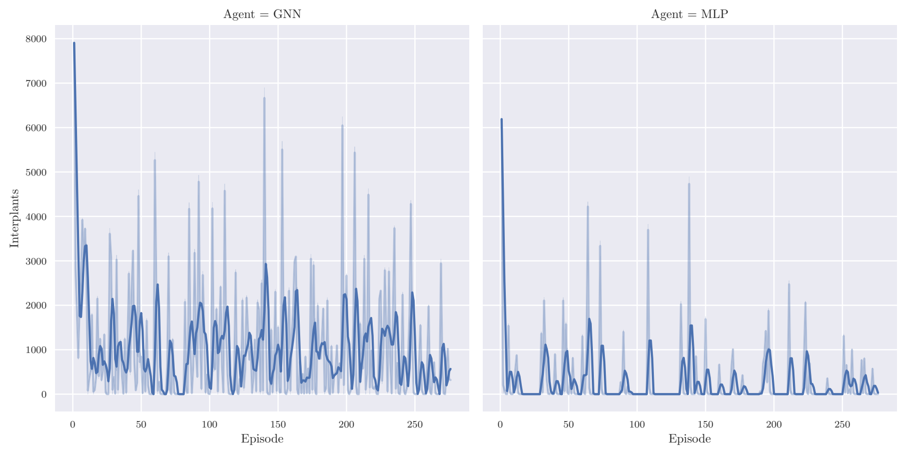


Figure 18: GNN and MLP's interplants from a sample run over 250 episodes of the deterministic allocation environment

as much as possible.

Table 14 and Figure 19 show the results for deterministic allocation with parameters equivalent to the small environment. In this case, MLP averages a normalized cost of 2.54 per unit, which corresponds to 3.92 times better than random choice, while GNN's normalized cost of 3.78 is 2.63 times better than random choice. It is clear from this result that the RL agents are able to understand the dynamics of this simple environment.

Regarding the difference in cost between deterministic and RL agents, it can be explained by the ϵ-greedy search that the NN agents use to maintain exploration. On 1% of the steps, the RL agents will select a random action that can potentially result in unnecessary interplants.

There is a noticeable difference between GNN and MLP. It can be observed on Figure 18 that MLP tends to stay at zero interplants besides some sporadic spikes due to ϵ-greedy, while MLP diverges more frequently. This difference in stability is likely due to MLP having access to the valid mask explicitly on the feature vector, which enables it to more quickly associate a specific input neuron with the best action for the current order. GNN, on the other hand takes longer to converge to this policy, since it needs to propagate that information from the warehouse nodes to the appropriate customer node.

The variance in average cost is observed to be clearly reduced on all agents with respect to the regular small environment. Additionally, the random choice provides a reference on how much variance can be introduced by a policy. We observe that while Lookahead and B&B show a lower average cost with lower variance than the RL agents, the difference between the agents' performance is notably reduced.

These results, when contrasted against the results of the regular small environment, confirm that the RL agents are successfully learning useful policies

	Reward		Cost		TPA	
Agent	Mean	STD	Mean	STD	Mean	STD
Lookahead	10.46	0.21	2.17	0.10	2.92	0.43
B&B	10.45	0.21	2.17	0.10	8.94	1.29
MLP	10.39	0.25	2.54	0.98	2.43	0.20
Best Fit	10.30	0.27	2.93	0.97	0.16	0.02
GNN	10.19	0.37	3.78	2.20	5.44	0.67
Random	9.22	0.34	9.97	2.59	0.13	0.01

Table 14: Average metrics per agent on 30 runs of the deterministic allocation environment

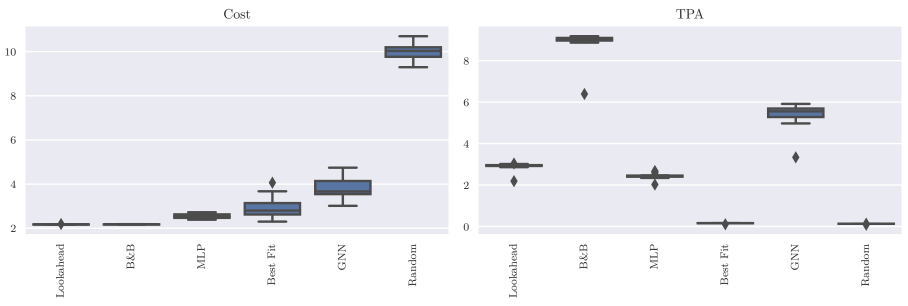


Figure 19: Box plots for cost and TPA per agent on 30 runs of the deterministic allocation environment

when the environment's variance sources are reduced. They also confirm that on deterministic settings, that RL-based agents can perform at least similarly to the deterministic optimizers.

Given that the inventory allocation is the only component of the environment that enables optimization outside the current optimization window, the question that remains is whether the demand generator is too stringent to allow for significant long term optimization. This question is harder to explore, as it would require rewriting the demand generator and optimizer to support generating excess inventory to be used in future optimization windows.

11.8 Limitations of the experiments

The evidence from the experiments on the regular SPA environments show that none of the implemented agents was able to obtain an average reward superior to that of a deterministic optimizer over many episodes, which is the goal of a reinforcement learning agent. These results raise doubt as to whether it is possible to find a better policy than the short-sighted optimization of deterministic agents, given the current network, supply and demand generating distributions.

Besides the previously discussed challenges of having a stable and informative reward function, there is also the question of whether the way the inventories are generated allow for long term optimization strategies. In theory, even with perfectly balanced inventories, the possibility of long term optimization should exist; the inventory units generated for one order within the horizon could be used to satisfy another that appears a few timesteps after, and await for future inventory generations with better allocation to satisfy outstanding orders. However, it is possible that these opportunities are marginal to the overall cost given the current design. The possibilities to reduce inventory costs using perfectly balanced problems would certainly be significantly less frequent than in a scenario where surplus inventories were allowed to exist. By simplifying the simulator to avoid having to deal with imbalanced problems and edge cases, we may have effectively removed the possibility of policies that optimize long term cost. Having new inventory arrivals be periodic and not tightly bound to orders in a horizon would also enable RL agents to learn to hold off shipping certain orders with inefficient allocations, something that a deterministic agent could not do. This would align more closely with a real world scenario, where production plants periodically create new goods and ship them to warehouses.

Another fundamental limitation of experiments done with this framework is the scalability of the problem. It is prohibitively expensive to simulate real world scenarios with the current simulator, especially due to the problem size scaling by a factor of K, as evidenced by the growth in TPA with the size of the environment. Whether the results obtained with a small K would apply to larger distribution networks with hundreds of thousands of commodities cannot be determined at the moment.

12 Conclusions

In this work, a novel multistage stochastic optimization problem, called the shipping point assignment (SPA) problem, was presented. It represents a distribution network with order consolidation constraints that give rise to the decisions of how to assign order shipments to warehouses. The SPA problem is challenging to handle with existing optimization techniques due to its high dimensionality and high computational complexity. It was formulated as a mathematical optimization program, as well as a Markov decision process (MDP), in order to optimize it using reinforcement learning techniques.

A simulator design and implementation was presented with the purpose of gaining a better understanding of the kind of policies that are effective to minimize expected cost over time on the SPA problem. The simulator generates instances of distribution networks with different structures and demand distributions that balance representing a real world distribution network with computational scalability. Agents are tasked with assigning orders to warehouses during the simulation, with the objective of minimizing costs incurred from inventory mismatches. The performance of a given policy may be assessed in terms of average cost, time per action (TPA) and number of interplant movements and average reward.

Six agents with different policies for the SPA were compared. One was a random choice baseline. Two used deterministic optimizers: a branch & bound-based integer programming solver and a linear programming-based lookahead solver. One was a greedy heuristic policy that consisted in finding the best fit according to the current inventory availability. The last two used Q learning with different neural networks for value function approximation: one using a multilayer perceptron and the other used GCN. Additionally, a random choice agent was included as a baseline. For the reinforcement learning agents, a reward function was designed in order to give feedback about the impact of the agent's actions in the environment's state. A novel contribution of this work was the use of a GCN network as a value function approximation of one of the agents in order to leverage the graph structure of the stochastic optimization program. The hypothesis that was presented stated that an agent leveraging the graph structure of the distribution network by means of the aforementioned GCN-based VFA should be superior in terms of long-term optimization cost and time per action, when compared against the greedy policy and the deterministic optimizers.

A set of experiments was performed in order to compare the proposed agents in terms of their average cost and time elapsed per action on multiple instances of the SPA with increasing optimization sizes. The experiments confirmed that deterministic agents' TPA increased exponentially with environment size, while greedy and neural network agents remain at a virtually constant TPA. Furthermore, the B&B agent was entirely unable to enact a policy on the largest environment. However, the hypothesis that the reinforcement learning agents would outperform greedy heuristic and deterministic optimizers in terms of cost was rejected for all sizes of the SPA environment. An encouraging finding was that GCN-based VFAs perform better than a random choice on certain scenarios. This suggests that the agents were successful in extracting some useful information from the graph structure of the optimization problem. The failure to observe this same behavior on the larger environments highlights the importance of hyperparameter tuning when the scale of demand changes.

The results obtained suggest that the best policies for the experiment scenarios are the ones based on deterministic optimizers. The two deterministic optimizers, Lookahead and B&B, perform similarly, although B&B seems to gain a marginal advantage in average cost as the optimization problem becomes more complex, at the expense of a much higher computational overhead. The marginal advantage of B&B over Lookahead also suggests that the potential improvement of a policy that successfully optimizes cost in the long-term might also be quite small.

Additionally, the results observed using the deterministic allocation setting shed light on the shortcomings of the current design of the inventory and demand generators. The results on this simpler environment showed dramatic differences in cost of RL agents with respect to random choice, which confirm the correctness in the implementation of the RL agents. It is possible that RL agents are suitable for this problem, but the environment is too stringent to showcase significant improvements on long-term optimization cost. Further work is necessary in order to make deep reinforcement learning agents competitive for stochastic optimization problems.

Finally, it is suspected that the reason why RL agents struggle to match even the performance of deterministic optimizers is because they must learn both the optimization problem constraints, as well as the stochastic variable dynamics, in order to generate decisions that maximize long-term reward attainment. In the future work section, improvements to the neural network architectures are proposed to address this issue.

13 Future work

The results obtained in this work should motivate further research on the combination of deep reinforcement learning techniques as a viable alternative to improve upon the state of the art on challenging stochastic optimization problems that cannot be exploited by properties such as convexity of the underlying optimization problem. Two large areas of future work can be identified. The first is to try to improve the results attained by existing agents. The second is to make modifications to the simulator to make it more scalable and to have demand generators that represent different scenarios. In this section, some unexplored ideas on these two areas are presented.

13.1 Improvement ideas for SPA agents

With respect to improving the results of the current agents, several improvement ideas were left out of the scope of this work. The most important area of improvement is exploring how to encode information of the optimization problem into neural networks. Other improvements include: using different reinforcement learning algorithms, changing the architectures for the VFAs and modifying the reward function.

13.1.1 Encoding the optimization problem into a neural network

The evidence strongly suggests that the neural networks are struggling to learn the stochastic variable dynamics alongside the constraints of the optimization problem. For this reason, the most promising direction of future work is to explore encoding information about the optimization problem as prior knowledge to the network. Recently, it has been shown that certain subclasses of convex optimization problems can be differentiated through by treating them as a function that maps problem data to solutions [111]. Moreover, Agrawal e.t al. [112] have proposed a framework that allows to integrate differentiable convex optimization layers for a subset of convex programs called Disciplined Convex Programs (DCP). They implement this as a layer that can be integrated with differentiable programming frameworks. This is achieved generally for all DCPs by differentiating through the solutions of the cone form of a DCP (all DCPs can be rewritten into cone form, which can be achieved by feeding the problem in its natural mathematical form into a Domain Specific Language for rewriting convex problems such as Cvxpy [113][114]). It might be promising to attempt to encode the deterministic optimization window of the current state into this framework and add it as a differentiable layer to the neural network VFA, allowing an RL agent to focus mainly on learning the stochastic variable dynamics to find the best policy. The question that remains for this approach to be feasible is whether the SPA problem can be represented as a DCP. While Integer Programs are by definition not convex, a convex approximation may be achieved in some cases by relaxing the binary variables into a continuous $[0, 1]$ space. Formally demonstrating that the SPA problem is indeed a DCP requires extensive work in convex analysis. However, there are convex optimization DSLs that can be used to write an optimization problem in its natural mathematical notation in order to analyze the convexity of its variables and constraints. We informally observed that a sample optimization window based on the SPA constraints is indeed a DCP by writing it in the Cvxpy DSL.

13.1.2 VFA architectures

Many different and promising VFA architectures are left to explore. For example, a Graph Attention Network (GAT) [115], might be useful to bring attention to certain nodes of the distribution network that are more relevant to the current window of the optimization window. Since the graph structures generated by this problem are fairly stable (a fixed-length planning horizon with different nodes only for the demand nodes), a spatial-temporal graph network could be used to encode the state information from the graphs and the state transitions simultaneously. Using Message Passing Networks, only graph-level embeddings were used to generate decisions, but node-level embeddings could also be generated and then extract the Q values from the embeddings of the order node for which the decision will be made. Finally, only graph embeddings on the physical network graph were explored, but an interesting strategy could be to create embeddings from the TEN network, which encodes both spatial and temporal information and is the graph that most closely resembles the actual network flows problem.

13.1.3 Different reinforcement learning algorithms

This work focused more heavily on the design of VFA networks and thus, all reinforcement learning agents used Q learning. No other reinforcement learning algorithms such as Actor-Critic, Double DQN, Dueling DQN, were evaluated. While these algorithms could potentially improve learning stability and convergence, we consider that it is much more pressing to ensure find a VFA that actually reduces short and long term cost before venturing into different learning techniques.

13.1.4 Modifying the reward function

With respect to reward functions, the results obtained suggest that the chosen reward function did not offer a strong enough signal for reinforcement learning agents to distinguish a good policy from a bad one, since the variance between average rewards for all agents fell under one standard deviation of each other. One idea of a reward function that was not explored consists of evaluating the interplant cost relative to the total overall cost,

$$R_t = -\frac{InterplantCost_t}{Cost_t},$$

where $InterplantCost_t$ is the portion of the optimization cost related to interplant flows. This reward function has the advantage of being numerically stable and always between zero and one, which could be advantageous in terms of scale for neural networks. Also, since it is a proportion of parts of the cost, this function should be resilient to changes in total demand observed.

13.2 Modifications to the environment simulator and demand generators

The implementation of the SPA environment is valuable because it enables the evaluation of different policies to minimize long term cost, but it was evidenced by experiments that it is limited by problem sizes, which scale with the number of commodities, planning horizon length and number of orders per timestep. Further technical improvements can be done to the simulator in order to scale to larger optimization problems and be able to simulate environments that more closely resemble real world scenarios.

Since it is not uncommon to find supply chains that are concerned with the fulfillment of hundreds, even thousands of SKUs at the same time, making more realistic scenarios requires supporting a high number of commodities. One improvement in this direction is to parallelize the linear optimization routine. Since the relaxation of fixing the shipping point of all orders means that the optimizer runs k independent min cost flows problems, these could be solved in parallel. For a network with high dimensionality in the number of commodities, this would allow the simulation to scale horizontally by distributing the separate subproblems into machines on a cluster.

Another improvement to reduce optimization times would be to solve simulation steps incrementally. Currently, the simulator constructs and optimizes new minimum cost flows from scratch after each action. This is inefficient, since much of the nodes and flows from one optimization step to the other will remain the same from one timestep to another. By using this information to provide hints of the optimal values of the decision variables, the overall optimization time could be reduced to facilitate running larger experiments.

In order to focus on the comparison between deterministic and reinforcement learning agents on the shipping point assignment, several simplifications to the distribution network were imposed in this work. For example, in order to make the generated problems always have solutions, capacities were always set to infinite and supply always perfectly matches demand. It would be more realistic to have limited capacities, allow for inventory generators to produce an inventory surplus and support having supply-demand mismatches by using dummy nodes to capture any unsatisfied demand and impose a cost on unfulfilled orders. The evidence provided by the deterministic inventory allocator suggests that this modification might give agents more degrees of freedom to make decisions that optimize in the long term. Another simplification mentioned in the problem statement was that costs were held constant for each kind of arc (inventory, interplant, delivery), while on a real world scenario, costs would be modeled by the distance to a warehouse, or vary according to route conditions or time of day.

Finally, some simplifications of the demand generating distributions could be removed. Currently, the generator assumes demand to be dense, which means that all commodities are being requested on all orders for all customers. It is more realistic to assume that customers order from a subset of the overall commodity set, which would be generated from a vector of probabilities that customer c orders commodity k. Another simplification is that a fixed set of orders is generated per day Φ. This could be replaced by sampling a Poisson to generate a certain number of daily orders.

References

[1] Aaron Courville, Ian Goodfellow, and Joshua Bengio. Deep Learning. In *MIT Press*, volume 29. 2016.

[2] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training Recurrent Neural Networks. *30th International Conference on Machine Learning, ICML 2013*, (PART 3):2347–2355, 11 2012.

[3] Zonghan Wu, Shirui Pan, Guodong Long, Jing Jiang, and Chengqi Zhang. Graph WaveNet for Deep Spatial-Temporal Graph Modeling. *IJCAI International Joint Conference on Artificial Intelligence*, 2019-Augus:1907–1913, 5 2019.

[4] Lawrence V Snyder and Zuo-Jun Max Shen. *Fundamentals of Supply Chain Theory, Second Edition*. Wiley, 2019.

[5] Council of Supply Chain Management Professionals. 29th Annual State of Logistics Report. Technical report, Council of Supply Chain Management Professionals, Washington D.C., 2018.

[6] Frank L. Hitchcock. The Distribution of a Product from Several Sources to Numerous Localities. *Journal of Mathematics and Physics*, 20(1-4):224–230, 4 1941.

[7] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. Network flows. *Handbooks in Operations Research and Management Science*, 50(1):99, 1989.

[8] Alex Hall, Steffen Hippler, and Martin Skutella. Multicommodity flows over time: Efficient algorithms and complexity. *Theoretical Computer Science*, 2007.

[9] John R. Birge and François Louveaux. *Introduction to Stochastic Programming*. Springer, second edi edition, 2011.

[10] Weibo Liu, Zidong Wang, Xiaohui Liu, Nianyin Zeng, Yurong Liu, and Fuad E. Alsaadi. A survey of deep neural network architectures and their applications. *Neurocomputing*, 234:11–26, 4 2017.

[11] Dzmitry Bahdanau, Kyung Hyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*. International Conference on Learning Representations, ICLR, 9 2015.

[12] Jayashree Padmanabhan and Melvin Jose Johnson Premkumar. Machine learning in automatic speech recognition: A survey. *IETE Technical Review (Institution of Electronics and Telecommunication Engineers, India)*, 32(4):240–251, 2015.

[13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. *CoRR*, 12 2013.

[14] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 1 2016.

[15] Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. In *Advances in Neural Information Processing Systems*, volume 32, pages 15580–15592, 2019.

[16] Hanjun Dai, Elias B. Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning Combinatorial Optimization Algorithms over Graphs. *Advances in Neural Information Processing Systems*, 2017-Decem:6349–6359, 4 2017.

[17] Zhuwen Li, Qifeng Chen, and Vladlen Koltun. Combinatorial Optimization with Graph Convolutional Networks and Guided Tree Search. *Advances in Neural Information Processing Systems*, 2018-Decem:539–548, 10 2018.

[18] L. R. Ford and D. R. Fulkerson. *Flows in networks.* RAND Corporation, 2015.

[19] T. C. Hu. Minimum-cost flows in convex-cost networks. *Naval Research Logistics Quarterly*, 13(1):1–9, 3 1966.

[20] Alex Orden. The Transhipment Problem. *Management Science*, 2(3):276–285, 4 1956.

[21] G.B. Dantzig. Problems for the Numerical Analysis of the Future. In *Problems for the Numerical Analysis of the Future*, number v. 15-16 in Applied mathematics series, chapter Chapter 4, page 18. U.S. Government Printing Office, 1951.

[22] George Bernard Dantzig. *Linear Programming and Extensions*, volume 53. RAND Corporation, Santa Monica, CA, 1963.

[23] Lisa Fleischer and Martin Skutella. Quickest flows over time. *SIAM Journal on Computing*, 2007.

[24] Bettina Klinz and Gerhard J. Woeginger. Minimum-cost dynamic flows: The series-parallel case. *Networks*, 2004.

[25] Rainer E Burkard, Karin Dlaska, and Bettina Klinz. The Quickest Flow Problem. Technical report, 1993.

[26] Lisa Fleischer, James B. Orlin, and Working Paper. Optimal rounding of instantaneous fractional flows over time. *SIAM Journal on Discrete Mathematics*, 13(August):145–153, 8 1999.

[27] Martin Groß and Martin Skutella. Maximum multicommodity flows over time without intermediate storage. In *Lecture Notes in Computer Science*, volume 7501 LNCS, pages 539–550. Springer, Berlin, Heidelberg, 2012.

[28] Warren B. Powell, Patrick Jaillet, and Amedeo Odoni. Stochastic and dynamic networks and routing. *Handbooks in Operations Research and Management Science*, 8(C):141–295, 1 1995.

[29] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction.* 2nd edition, 2018.

[30] R. S. Sutton, Andrew G Barto, and C. J. C. H. Watkins. Learning and Sequential Decision Making. In Michael Gabriel and John Moore, editors, *Learning and Computational Neuroscience, Foundations of Adaptive Networks*, number September 1989, pages 539–602. MIT Press, 1990.

[31] Richard Bellman. The Theory of Dynamic Programming. *Bulletin of the American Mathematical Society*, 1954.

[32] Richard Bellman and A Markovian Decision Process. A Markovian Decision Process. *Indiana University Mathematics Journal*, 1957.

[33] Warren B. Powell. *Reinforcement Learning and Stochastic Optimization: A unified framework for sequential decisions.* 2021.

[34] Raymond K-M Cheung and Warren B Powell. Models and Algorithms for Distribution Problems with Uncertain Demands. Technical report, 1996.

[35] R.M. Van Slyke and Roger Wets. L-Shaped Linear Programs With Applications To Optimal Control and Stochastic Progrqmming. *SIAM Journal on Applied Mathematics*, 17(4):638–663, 7 1969.

[36] George B. Dantzig and Philip Wolfe. Decomposition Principle for Linear Programs. *Operations Research*, 8(1):101–111, 2 1960.

[37] J. F. Benders. Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik*, 4(1):238–252, 12 1962.

[38] Claus C. Carøe and Jørgen Tind. L-shaped decomposition of two-stage stochastic programs with integer recourse. *Mathematical Programming, Series B*, 83(3):451–464, 11 1998.

[39] Shabbir Ahmed, Mohit Tawarmalani, and Nikolaos V. Sahinidis. A finite branch-and-bound algorithm for two-stage stochastic integer programs. *Mathematical Programming*, 100(2):355–377, 6 2004.

[40] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1798–1828, 2013.

[41] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 11 1958.

[42] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation*, 1(4):541–551, 12 1989.

[43] Samira Pouyanfar, Saad Sadiq, Yilin Yan, Haiman Tian, Yudong Tao, Maria Presa Reyes, Mei Ling Shyu, Shu Ching Chen, and S. S. Iyengar. A survey on deep learning: Algorithms, techniques, and applications, 8 2018.

[44] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. Technical report, 2012.

[45] Richard Bellman. Dynamic programming. *Science*, 153(3731):34–37, 7 1966.

[46] Yoshua Bengio and Martin Monperrus. Non-Local Manifold Tangent Learning. Technical report, 2004.

[47] Lawrence Cayton. Algorithms for manifold learning. 2005.

[48] Hariharan Narayanan and Sanjoy Mitter. Sample complexity of testing the manifold hypothesis. In *Advances in Neural Information Processing Systems 23: 24th Annual Conference on Neural Information Processing Systems*, 2010.

[49] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1 1989.

[50] G Cybenkot. Mathematics of Control, Signals, and Systems Approximation by Superpositions of a Sigmoidal Function*. *Math. Control Signals Systems*, 2(3):303–314, 1989.

[51] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve Restricted Boltzmann machines. Technical report, 2010.

[52] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[53] Léon Bottou. On-line Learning and Stochastic Approximations. In *On-Line Learning in Neural Networks*, pages 9–42. Cambridge University Press, 2 1998.

[54] D. Randall Wilson and Tony R. Martinez. The general inefficiency of batch training for gradient descent learning. *Neural Networks*, 16(10):1429–1451, 12 2003.

[55] Geoffrey Roeder, Luke Metz, Google Brain, and Diederik P Kingma. On Linear Identifiability of Learned Representations. 7 2020.

[56] Ian J. Goodfellow, Oriol Vinyals, and Andrew M. Saxe. Qualitatively characterizing neural network optimization problems. *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, 12 2014.

[57] Yann N. Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Advances in Neural Information Processing Systems*, volume 4, pages 2933–2941. Neural information processing systems foundation, 6 2014.

[58] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. Technical report, 2014.

[59] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. 7 2012.

[60] Ossama Abdel-Hamid, Abdel Rahman Mohamed, Hui Jiang, Li Deng, Gerald Penn, and Dong Yu. Convolutional neural networks for speech recognition. *IEEE Transactions on Audio, Speech and Language Processing*, 22(10):1533–1545, 10 2014.

[61] Tim Hill, Marcus O'Connor, and William Remus. Neural network models for time series forecasts. *Management Science*, 42(7):1082–1092, 7 1996.

[62] Y Bengio. Convolutional Networks for Images, Speech, and Time-Series Unsupervised Learning of Speech Representations View project Parsing View project. Technical report, 1997.

[63] Y. T. Zhou and R. Chellappa. Computation of optical flow using a neural network. pages 71–78. Publ by IEEE, 1988.

[64] Alexander Waibel, Toshiyuki Hanazawa, Geoffrey Hinton, Kiyohiro Shikano, and Kevin J. Lang. Phoneme Recognition Using Time-Delay Neural Networks. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(3):328–339, 1989.

[65] Hava T. Siegelmann and Eduardo D. Sontag. On the computational power of neural nets. *Journal of Computer and System Sciences*, 50(1):132–150, 2 1995.

[66] D Sontag, Hava T. Siegelmann, and Eduardo D. Sontag. Turing computability with neural nets. *Applied Mathematics Letters*, 4(6):77–80, 1 1991.

[67] Paul J. Werbos. Backpropagation Through Time: What It Does and How to Do It. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[68] Ronald J. Williams and David Zipser. A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. *Neural Computation*, 1(2):270–280, 6 1989.

[69] Mike Schuster and Kuldip K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[70] Yoshua Bengio, Paolo Frasconi, and Patrice Simard. Problem of learning long-term dependencies in recurrent networks. In *1993 IEEE International Conference on Neural Networks*, pages 1183–1188. Publ by IEEE, 1993.

[71] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning Long-Term Dependencies with Gradient Descent is Difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

[72] David Sussillo and L. F. Abbott. Random Walk Initialization for Training Very Deep Feedforward Networks. 12 2014.

[73] Herbert Jaeger. Adaptive nonlinear system identification with Echo State networks. In *Advances in Neural Information Processing Systems*, 2003.

[74] Wolfgang Maass, Thomas Natschläger, and Henry Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural Computation*, 14(11):2531–2560, 11 2002.

[75] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. Technical report, 5 2013.

[76] Tsungnan Lin, Bill G Horne, Peter Tii No, and C Lee Giles. Learning long-term dependencies is not as diicult with NARX recurrent neural networks. Technical report, 10 1998.

[77] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 11 1997.

[78] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *EMNLP 2014 - 2014 Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference*, pages 1724–1734. Association for Computational Linguistics (ACL), 6 2014.

[79] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. 12 2014.

[80] Sneha Chaudhari, Varun Mithal, Gungor Polatkan, and Rohan Ramanath. An Attentive Survey of Attention Models. 4 2019.

[81] Jiani Zhang, Xingjian Shi, Junyuan Xie, Hao Ma, Irwin King, and Dit-Yan Yeung. GaAN: Gated Attention Networks for Learning on Large and Spatiotemporal Graphs. *34th Conference on Uncertainty in Artificial Intelligence 2018, UAI 2018*, 1:339–349, 3 2018.

[82] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 2017-Decem, pages 5999–6009. Neural information processing systems foundation, 6 2017.

[83] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. 5 2020.

[84] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S Yu. A Comprehensive Survey on Graph Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–21, 2020.

[85] Alessandro Sperduti and Antonina Starita. Supervised neural networks for the classification of structures. *IEEE Transactions on Neural Networks*, 8(3):714–735, 1997.

[86] Marco Gori, Gabriele Monfardini, and Franco Scarselli. A new model for learning in graph domains. In *Proceedings of the International Joint Conference on Neural Networks*, volume 2, pages 729–734, 2005.

[87] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[88] Alessio Micheli. Neural network for graphs: A contextual constructive approach. *IEEE Transactions on Neural Networks*, 20(3):498–511, 2009.

[89] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural Message Passing for Quantum Chemistry. Technical report, 7 2017.

[90] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*. International Conference on Learning Representations, ICLR, 2017.

[91] Mikael Henaff, Joan Bruna, and Yann LeCun. Deep Convolutional Networks on Graph-Structured Data. 6 2015.

[92] Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. Order Matters: Sequence to sequence for sets. *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, 11 2015.

[93] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. *Advances in Neural Information Processing Systems*, pages 3844–3852, 6 2016.

[94] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. An End-to-End Deep Learning Architecture for Graph Classification. Technical Report 1, 4 2018.

[95] Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L. Hamilton, and Jure Leskovec. Hierarchical Graph Representation Learning with Differentiable Pooling. *Advances in Neural Information Processing Systems*, 2018-Decem:4800–4810, 6 2018.

[96] Junhyun Lee, Inyeop Lee, and Jaewoo Kang. Self-attention graph pooling. In *36th International Conference on Machine Learning, ICML 2019*, volume 2019-June, pages 6661–6670. PMLR, 5 2019.

[97] Ashesh Jain, Amir R Zamir, Silvio Savarese, and Ashutosh Saxena. Structural-RNN: Deep Learning on Spatio-Temporal Graphs. Technical report, 2016.

[98] Bing Yu, Haoteng Yin, and Zhanxing Zhu. Spatio-Temporal Graph Convolutional Networks: A Deep Learning Framework for Traffic Forecasting. *IJCAI International Joint Conference on Artificial Intelligence*, 2018-July:3634–3640, 9 2017.

[99] Shengnan Guo, Youfang Lin, Ning Feng, Chao Song, and Huaiyu Wan. Attention based spatial-temporal graph convolutional networks for traffic flow forecasting. In *33rd AAAI Conference on Artificial Intelligence, AAAI 2019, 31st Innovative Applications of Artificial Intelligence Conference, IAAI 2019 and the 9th AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019*, volume 33, pages 922–929. AAAI Press, 7 2019.

[100] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning 1992 8:3*, 8(3):293–321, 5 1992.

[101] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2 2015.

[102] Ilaria Giannoccaro and Pierpaolo Pontrandolfo. Inventory management in supply chains: A reinforcement learning approach. *International Journal of Production Economics*, 78(2):153–161, 2002.

[103] Afshin Oroojlooyjadid, MohammadReza Nazari, Lawrence Snyder, and Martin Takáč. A Deep Q-Network for the Beer Game: A Deep Reinforcement Learning algorithm to Solve Inventory Optimization Problems. 8 2017.

[104] S. Kamal Chaharsooghi, Jafar Heydari, and S. Hessameddin Zegordi. A reinforcement learning model for supply chain ordering management: An application to the beer game. *Decision Support Systems*, 45(4):949–959, 11 2008.

[105] Tim Stockheim, Michael Schwind, and Wolfgang Koenig. A Reinforcement Learning Approach for Supply Chain Management. *1st European Workshop on MultiAgent Systems*, 40(6):1299–1317, 2002.

[106] P. Pontrandolfo, A. Gosavi, O. G. Okogbaa, and T. K. Das. Global supply chain management: A reinforcement learning approach. *International Journal of Production Research*, 40(6):1299–1317, 4 2002.

[107] Huseyin Topaloglu and Warren B Powell. Dynamic Programming Approximations for Stochastic, Time-Staged Integer Multicommodity Flow Problems. Technical report, 2006.

[108] Julian Busch, Anton Kocheturov, Volker Tresp, and Thomas Seidl. NF-GNN: Network Flow Graph Neural Networks for Malware Detection and Classification. 12(18), 3 2021.

[109] Google Developers. OR-Tools — Google Developers, 2021.

[110] Gerald Gamrath, Daniel Anderson, Ksenia Bestuzheva, Wei-Kun Chen, Leon Eifler, Maxime Gasse, Patrick Gemander, Ambros Gleixner, Leona Gottwald, Katrin Halbig, Gregor Hendel, Christopher Hojny, Thorsten Koch, Pierre Le Bodic, Stephen J Maher, Frederic Matter, Matthias Miltenberger, Erik Mühmer, Benjamin Müller, Marc E Pfetsch, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Christine Tawfik, Stefan Vigerske, Fabian Wegscheider, Dieter Weninger, and Jakob Witzig. The SCIP Optimization Suite 7.0. Technical report, Optimization Online, 3 2020.

[111] Brandon Amos and J. Zico Kolter. OptNet: Differentiable Optimization as a Layer in Neural Networks. *34th International Conference on Machine Learning, ICML 2017*, 1:179–191, 3 2017.

[112] Akshay Agrawal, Brandon Amos, Shane Barratt, Stephen Boyd, Steven Diamond, and J. Zico Kolter. Differentiable Convex Optimization Layers. *Advances in Neural Information Processing Systems*, 32, 10 2019.

[113] Akshay Agrawal, Robin Verschueren, Steven Diamond, and Stephen Boyd. A rewriting system for convex optimization problems. *JOURNAL OF CONTROL AND DECISION*, 5(1):42–60, 2018.

[114] Steven Diamond and Stephen Boyd. CVXPY: A Python-Embedded Modeling Language for Convex Optimization. *Journal of Machine Learning Research*, 17:1–5, 2016.

[115] Petar Veličković, Arantxa Casanova, Pietro Liò, Guillem Cucurull, Adriana Romero, and Yoshua Bengio. Graph Attention Networks. *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, 10 2017.