Proceedings of the 2025 Winter Simulation Conference
E. Azar, A. Djanatliev, A. Harper, C. Kogler, V. Ramamohan, A. Anagnostou, and S. J. E. Taylor, eds.

BRIDGING THE GAP: A PRACTICAL GUIDE TO IMPLEMENTING DEEP REINFORCEMENT LEARNING SIMULATION IN OPERATIONS RESEARCH WITH GYMNASIUM

Konstantinos Ziliaskopoulos[1], Alexander Vinel[1], and Alice E. Smith[1]

[1]Dept. of Industrial and Systems Engineering, Auburn University, Auburn, AL, USA

ABSTRACT

Deep Reinforcement Learning (DRL) has shown considerable promise in addressing complex sequential decision-making tasks across various fields, yet its integration within Operations Research (OR) remains limited despite clear methodological compatibility. This paper serves as a practical tutorial aimed at bridging this gap, specifically guiding simulation practitioners and researchers through the process of developing DRL environments using Python and the Gymnasium library. We outline the alignment between traditional simulation model components, such as state and action spaces, objective functions, and constraints, and their DRL counterparts. Using an inventory control scenario as an illustrative example, which is also available online through our GitHub repository, we detail the steps involved in designing, implementing, and integrating custom DRL environments with contemporary DRL algorithms.

1 INTRODUCTION

Deep Reinforcement Learning (DRL) has emerged as a powerful technique capable of solving complex sequential decision-making problems across many domains, including gaming, autonomous navigation, robotics, and increasingly, Simulation and Operations Research (OR). The structured and analytical framework of these problems makes them particularly suitable for reinforcement learning. In fact, many such models, which are characterized by explicit objective functions, constraints, and state transitions, are naturally formed as Markov Decision Processes (MDPs), which easily translate to DRL. Objective functions from traditional models correspond closely to reward functions in DRL, while constraints can be effectively captured through carefully designed simulation dynamics.

While OR and DRL share conceptual foundations, collaboration between these two fields is still developing. Powell (2019) describes the fragmentation of the optimization community, especially in stochastic problems. Simulation practitioners in industry and academia typically use simulation software to develop their discrete-event models, such as Simio (Pegden 2007) or Anylogic (Borshchev 2014). Although these tools are proven and robust simulation packages, their integration with the fast-moving field of deep learning and DRL agents is currently lacking. Efforts to bridge this gap exist but often encounter significant barriers due to the rapid evolution of DRL technologies, sparse documentation, and fragmented implementations across numerous high-level and low-level programming libraries. Practitioners aiming to integrate DRL into simulation projects face a steep learning curve, frequently having to navigate outdated frameworks or conflicting methodologies.

This tutorial aims explicitly to address this gap, providing accessible guidance tailored to simulation practitioners and researchers who have basic familiarity with Python programming. We present a structured overview of the process of designing and implementing custom RL environments, elucidating how key components, such as state and action spaces, reward functions, and environment dynamics, align with traditional OR models. Furthermore, we offer practical advice for integrating these environments with state-of-the-art DRL algorithms, leveraging standard implementations or customized approaches adapted specifically to practitioner needs.

The remainder of this paper is organized as follows: In Section 2, we provide the context and discuss prior applications of DRL within OR/simulation. Section 3 outlines the key steps involved in creating DRL simulation environments. Section 4 demonstrates the implementation process with a detailed illustrative example based on an inventory control problem. Section 5 covers integration with existing DRL algorithms, concluding with guidance on leveraging these methods effectively in practice. Finally, in Section 6, we present some different avenues of future research in the field that are especially relevant to OR researchers and simulation practitioners.

2 BACKGROUND

DRL consists of two main components: MDPs, which serve as the model formulation language, and neural networks, which approximate both problem formulations and solutions (Boute et al. 2022). MDPs and dynamic programming (DP) are already critical components in OR methodologies often tackled in simulation. DRL simply addresses the curse of dimensionality in RL by providing scalability and flexibility through neural networks, rather than introducing new fundamental modeling concepts.

Traditionally, Reinforcement learning (RL) builds upon the dynamic programming framework, where the primary goal is to identify an optimal policy for a given sequential decision-making problem (Sutton and Barto 2018). An agent interacts with an environment over discrete time steps, receiving observations (states, s) and rewards (r), and taking actions (a). The agent's behavior is defined by its policy, π, which maps states to actions or distributions over actions, typically denoted as $\pi(a|s) = P[A_t = a|S_t = s]$. The objective is to find an optimal policy, π^*, that maximizes the expected cumulative discounted reward, often represented by value functions. The state-value function, $V_\pi(s)$, estimates the expected return starting from state s and following policy π: $V_\pi(s) = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|S_t = s]$, where $\gamma \in (0,1]$ is the discount factor. Similarly, the action-value function, $Q_\pi(s,a)$, estimates the expected return after taking action a in state s and subsequently following policy π: $Q_\pi(s,a) = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|S_t = s, A_t = a]$. RL algorithms iteratively apply principles related to the Bellman equations of dynamic programming (Bellman 1984) to find optimal value functions (V^*, Q^*) and thereby the optimal policy π^*. For instance, the optimal action-value function Q^* satisfies the Bellman optimality equation: $Q^*(s,a) = \mathbb{E}[r_{t+1} + \gamma \max_{a'} Q^*(S_{t+1}, a')|S_t = s, A_t = a]$.

For problems with large or continuous state and/or action spaces, representing value functions or policies explicitly (e.g., in tables) becomes intractable. The integration of neural networks with reinforcement learning, commonly referred to as deep reinforcement learning (DRL), addresses this by using neural networks as powerful function approximators (Bertsekas and Tsitsiklis 1996). In DRL, the value functions or the policy itself are parameterized by the network's weights, θ. For example, we might approximate the action-value function as $Q(s,a;\theta) \approx Q^*(s,a)$ or the policy as $\pi(a|s;\theta) \approx \pi^*(a|s)$. Different DRL algorithms leverage these approximations in various ways:

- *Value-based* methods (like Deep Q-Networks, DQN (Mnih et al. 2013)) train a network to approximate $Q(s,a;\theta)$ and derive the policy implicitly (e.g., by choosing the action with the highest Q-value). Training typically minimizes the difference between the current Q-value estimate and a target value derived from the Bellman equation (Temporal Difference learning).
- *Policy-based* methods (like REINFORCE (Williams 1992)) directly train a network to represent the policy $\pi(a|s;\theta)$ and update θ to increase the probability of actions leading to higher returns (Policy Gradient methods).
- *Actor-Critic* methods (like Soft Actor Critic, SAC (Haarnoja et al. 2018) or Proximal Policy Optimization, PPO (Schulman et al. 2017)) combine both, using separate networks (or shared networks with different output heads) for the policy (the actor) and a value function (the critic), which helps evaluate the actor's actions and guide learning. For example, in PPO, the actor is updated using a clipped surrogate objective L^{CLIP} that prevents large, divergent policy updates, while the critic is trained to minimize the mean squared error between the predicted and actual returns.

We discuss these algorithms in more detail in Section 5.

As computational resources have become more accessible and cost-effective, DRL has seen rapid expansion across various fields. Significant achievements in DRL include DeepMind's AlphaGo, which successfully surpassed human champions in the game of Go (Silver et al. 2017). In the realm of autonomous vehicles, recent developments have produced autonomous agents capable of rivaling expert drone pilots in both speed and maneuverability (Kaufmann et al. 2023). Reinforcement learning has also contributed significantly to breakthroughs in large language models; for example, researchers developing DeepSeek used reinforcement learning to improve reasoning and problem-solving capabilities in their models (Guo et al. 2025).

Parallel to advancements in other domains, reinforcement learning has garnered substantial interest within operations management. A closely related concept, more familiar to Operations Research (OR) practitioners, is approximate dynamic programming (ADP). Powell, in his work on ADP (Powell 2011), categorizes several core problem classes suitable for ADP, including budgeting, asset acquisition, resource allocation, storage management, shortest path, and dynamic assignment—problems central to OR. Beyond these categories, DRL has successfully addressed various other OR-related challenges, including combinatorial optimization (Mazyavkina et al. 2021), inventory management (Boute et al. 2022), and supply chain optimization (Oroojlooyjadid et al. 2017).

Although tools have emerged to bridge the gap between OR practice and RL, such as the OR-Gym library (Hubbs et al. 2020) or the in development RL4CO library (Berto et al. 2023), the rapidly evolving nature of DRL often leaves these implementations obsolete, insufficiently documented, or difficult to implement. Addressing this gap, the Farama Foundation introduced Gymnasium, a Python library intended as the natural successor to OpenAI's Gym (Towers et al. 2024). Despite similarities, Gymnasium differs in parts from the original implementation, requiring DRL practitioners either to rely on deprecated tools compatible with legacy systems or to navigate the complexities of adapting their code to the newer standard.

Although one could theoretically train a DRL agent with any simulation model, there exist significant benefits in using the community standard in simulation environments. Software such as AnyLogic or Simio excels at detailed industrial simulations, but their integration with DRL would require substantial custom code to handle action execution, space representation, and reward calculation. Gymnasium, on the other hand, is currently integrated with most major software packages in the DRL ecosystem. The tools developed by the leading research teams, for example JAX from Google (Frostig et al. 2018) or TorchRL from Meta AI (Bou et al. 2023), provide integrations with Gymnasium environments. Another benefit of using Gymnasium environments over other established simulation tools for DRL training is the ability to vectorize the simulation models. Typically, an agent will require millions of interactions with a simulation to converge to an optimal policy. Gymnasium provides tools to run multiple simulation environments *in parallel*, with which the DRL algorithm can interact simultaneously, constrained only by the modeler's hardware.

Since notation differs wildly between the OR and DRL fields (Powell 2019), we establish consistent terminology to help the reader navigate the rest of the tutorial. We define the following:

- $\mathscr{A}$ = action space
- $\mathscr{O}$ = observation space
- $\mathscr{S}$ = state space
- r = reward function
- θ = Trained parameters of the DRL agent
- π = Policy distribution of the DRL agent

This tutorial provides a structured, step-by-step approach to implementing a discrete-event DRL simulation environment in Python using the Gymnasium library. Designed to bridge the existing gap between OR and DRL, the guide is designed to provide clarity and accessibility. The illustrative example

used in this tutorial focuses on an inventory control problem; however, the principles, methods, and insights presented can be readily generalized to a wide array of simulation situations.

3 CONSTRUCTING THE ENVIRONMENT

As many simulation practitioners are aware, the first step in developing a simulation model is to define it conceptually. In fact, many of the lessons from Robinson (2008) still hold true here. Our simulation environment still needs to be built in tandem with stakeholder and company objectives. We need to decide on both the breadth of the model and the level of detail required. We need to answer questions about our system, like: *What are the assumptions and simplifications we are willing to make? What do we want the inputs and outputs of our final model to be?* Of course this is also an iterative process, as we work on the model implementation we often revisit our conceptual model and make additions or adjustments. Nevertheless, we assume the reader has some familiarity in simulation model construction so we do not go in depth on this topic. Readers unfamiliar with the concept of conceptual modeling in simulation are encouraged to read Robinson (2008) for a more in-depth discussion on the topic.

3.1 Limitations and Considerations for Translating Simulation Models from Specialized Software

Although Gymnasium provides a powerful platform for developing DRL compatible simulation environments, it is important to recognize inherent limitations when translating simulations from specialized software. Commercial discrete-event simulation tools offer easy-to-use tools, with graphical interfaces and domain-specific functionality that simplifies model development, particularly in applications with complex simulation logic or entity interaction. Translating such models to Gymnasium (or Python in general) requires manually re-implementing many underlying process logic and data structures, which can be time-consuming and error-prone. Additionally, not all features in commercial software have a clear analog in Gymnasium; for instance, any custom visualization capabilities have to be implemented manually. While there is no alternative for practitioners aiming to use state-of-the-art DRL models with their simulation environments, a consequence of the fast-paced nature of the field is rough edges in implementation. Therefore, our methodology emphasizes practical guidelines for conceptual translation and simplification, rather than a feature-by-feature replication. We focus on capturing critical decision-making dynamics relevant to reinforcement learning and provide further material for simulation and modeling extensions in Section 6

3.2 Defining the Components

After defining our model conceptually, we can clarify the basic structure of our simulation environment. Here, the terminology differs slightly from classic simulation literature. Depending on the literature, the set of *decision variables, control variables,* or *policy inputs* are called here **the action space**, $\mathscr{A}$. The set of *measurements, observables, input data,* or *exogenous information* are called **the observation space**, $\mathscr{O}$. The set of *system states* or *state variables* are defined here as **the state space**, $\mathscr{S}$. Finally, as DRL environments are built as simulation models to be optimized, we need a function that quantifies the performance of our training agent. Therefore, the *objective function, cost function,* or *utility function* of the model is called **the reward function**, $r(a|s)$.

The action space is the set of decision variables available to our agent and the values those variables take at each timestep of our model. The observation space is the set of variables that accurately describe the current state of our system to our agent. OR practitioners may be familiar with the concept of Partially Observed Markov Decision Processes (POMDP), (Kaelbling et al. 1998). As in POMDPs, the observation space of the agent is not necessarily equivalent to the state space of the simulation model. In order to model real-world conditions, we might want to obscure certain state variables from our DRL agent to better mimic actual decision making processes. We do this by providing a subset of the overall information to our training agent. The state space that defines the dynamics of our model can be obscured from the agent and still be extracted from the environment for model observability and validation. Finally, the reward

function is the metric that our agent is trying to optimize. While this is often a difficult and iterative process in other DRL domains, like robotics and autonomous vehicles, where the specific metric we are trying to optimize is not always clear, in OR models the reward function tends to be clear. After all, if the problem can theoretically be defined by a mathematical model, a cost/reward function already exists; we simply need to adapt it to our implementation.

3.3 Transition Dynamics and Simulation Logic

After defining the components of our environment, we need to define the transition dynamics of our model, or the implementation logic of our simulation. Again, while this is often a complex and uncertain process in many DRL domains, often involving neural network function approximators to estimate unknown or complex dynamics, in OR models the transition dynamics are often closed form and well-defined. We demonstrate this in our example in Section 4.

3.4 Implementation in Gymnasium

Every Gymnasium environment has 3 mandatory sections to properly function as a DRL environment: an initialization (`__init__`) function, a reset (`reset()`) function and a step (`step()`) function. Besides these, the modeler is free to add as many auxiliary functions as needed to properly define the environment.

- The `__init__` function initializes the environment and sets the parameters of the simulation. Here, the modeler should define the parameter input the model requires, decide how to handle optional inputs, as well as setting the environment components. At minimum, the `__init__` function should define the action space and the observation space as environment class attributes, `self.action_space` and `self.observation_space`. In `gymnasium`, observation and action spaces are defined as `gym.spaces` objects. Depending on the values and the structure of the action and observation spaces, the modeler can choose from different types of spaces, as shown in Table 1. It is important to note that some reinforcement learning algorithms require specific types of action and observation spaces. For example, DQN requires discrete action spaces, while PPO can handle continuous action spaces. In general, the modeler should strive to design observations and actions as simply as possible, as composite spaces can also preclude certain optimizers and algorithms. We discuss this in more detail in Section 5.
- The `reset()` function initializes the environment at the beginning of each episode. This function should reset the environment to its initial state and return the initial observation. In our inventory control example, the reset function would set the inventory levels to the starting inventory, the pipeline inventory to 0, and the demand ratios to the expected demand. It would then return the stacked vertices of these variables as the initial observation. As an example, we reset the internal timestep counter to 0 and inventory levels to the starting inventory. Other state attributes, such as the pipeline inventory vector, are set to their initial values. The function then returns the initial observation.
- The `step()` function is the most important method of our environment. It is where the simulation logic takes place, but is also how we define the agent interaction with our environment. Every time the `step()` function is called, the simulation environment progresses one time period, and allows the agent to take another action. It is important to note here that this does not mean our time steps have to be equal in length. We could define each time step as the time until the next event occurs, as in traditional discrete-event simulation, provided we program the environment as such. A better way to think about the time step is the desired time between action opportunities for our agent. For example, if we want our agent to be able to interact with our environment daily, then it makes sense to have time steps of equal length, where each call of the `step()` function would simulate the transition dynamics for one day. On the other hand, if we want our agent to interact with the environment depending on the time spent on his previous action, for example, then we would

simply change our simulation logic and increment the environment class attributes differently. In OR problems the time steps tend to be equal but the modeler should keep in mind that they are not precluded from a different implementation, if it suits their problem structure better.

The `step()` function takes an action matrix as input, updates the environment based on the action, and returns five variables: the next observation, the reward, a boolean indicating whether the episode has reached a terminal state, i.e., is terminated, a boolean indicating whether the episode is truncated, and key-value pairs (in Python, a dictionary) of additional information. The definition of *terminated* and *truncated* conditions is up to the modeler, but there are some common conventions. In general, a terminating condition should reflect an agent completing or failing its task. A truncating condition, on the other hand, ends an episode prematurely, but usually due to some external limit, like a maximum number of time steps. In OR problems, the difference is subtle. In some cases, like the classic news vendor problem, we can define *terminated* as an end of a single period or multi-period scenario, or when a predefined limit of stock-outs has been reached. We could truncate an episode based on random parameters, signifying external disruptions, or if the profit margins fall below viability thresholds and the episode is not worth continuing. In other problem categories, such as in a Traveling Salesman Problem, it is clear when to assign a *terminated* condition but not a truncated condition.

It is possible to design a simulation environment without a truncated condition, but it can help avoid degenerate cases where the agent learns to exploit the environment and help the agent extrapolate in unseen scenarios. In general, once an episode truncates, the agent can no longer accrue rewards. Therefore, it will learn to avoid truncating behavior, if possible, allowing the modeler to better shape its behavior, at the risk of introducing bias. For example, in our inventory control environment, we terminate an episode when the number of timesteps reaches the maximum. Meanwhile, during training, there is a small percentage chance each step that the episode will truncate. This is to avoid the agent learning to exploit the maximum number of timesteps of the environment, which could lead to reckless behavior later in the simulation.

Besides the two boolean variables, our example computes the new inventory levels, pipeline inventory, and demand ratios based on the action matrix and defined transition dynamics. We return the next step's observation matrix, as well as any additional information we want to extract from the model for observability, validation, and monitoring purposes: accrued cost, cumulative stock-outs, and orders placed. Finally, we also compute and return the reward for the step.

3.4.1 Reward Function

As mentioned earlier, the reward function is intrinsically linked with the respective problem's objective function. In essence, it is the objective function for the reinforcement learning agent. Through it, it will learn to better estimate value functions and select more effective policies. The modeler also has the option, however, to add *shaping* rewards to the reward function. Shaping rewards are typically used in complex simulation environments with delayed or sparse reward signals to encourage the agent towards a desired behavior. Shaping rewards tend to reduce convergence time by eliminating trial-and-error from the training process, but it has the downside of introducing bias to the agent. For example, in robotics, we might want a cobot to learn to approach a specific workstation. We might reward the agent only on achieving its goal, or we may give it partial rewards for approaching it. This might help the cobot learn faster. On the other hand, it might also lead to reward hacking, where the agent learns to exploit the easy intermediate rewards instead of searching for the intended goal, or cause the agent to overfit on intermediate steps. It requires additional foresight and care by the modeler to properly design the shaping rewards and gradually phase them out as the agent better learns the environment.

3.4.2 Auxiliary Functions

Besides the core three methods that are necessary to implement in every simulation environment, we can also add auxiliary functions to help implement our simulation logic and data collection. Some common

auxiliary functions to consider are functions to collate the action and observation spaces, as that logic tends to be reused often. Besides those, our example includes functions to generate daily orders and demand for each location from the user inputted demand parameters, a function to clip the action matrix to ensure that the agent does not order more than the available inventory, and finally a function to compute the expected demand each day to provide to the agent through the observation space.

3.5 Verifying and Monitoring the Environment

Of course, practitioners should be familiar with the concepts of verification and validation in simulation. Verification is the process of ensuring that the simulation model is implemented correctly and that it accurately represents the intended design, our conceptual model. Validation, on the other hand, is the process of ensuring that the simulation model accurately fulfills the intended purpose and that it is a good representation of the real-world system. Validation is usually done through comparison with real-world data, other models, or by consulting experts and stakeholders. While extremely important, validation is outside the scope of this tutorial.

Verification in `gymnasium` environments can be done a variety of ways. Researchers familiar with Python programming and best practices will be familiar with testing and debugging processes. While constructing the simulation logic, the modeler can write tests beforehand that initially will fail, and will only succeed with a proper implementation of the model. This paradigm is called test-driven development (TDD) and is a common practice in software engineering (Beck 2022). Besides tests, the model can also be verified through monitoring key output metrics during different phases of the simulation. For example, one can monitor the average reward per episode, the sensitivity of the state to different input actions, and if the environment is behaving as expected. Gymnasium action and state spaces provide a `sample()` method that can be used to generate random samples from the space to quickly iterate through a simulation and check environment behavior.

Since the environment is built as a tool for the DRL agent to learn from, we can also monitor the agent's performance during training. Irregular training patterns or episodic rewards can indicate that the environment is not behaving as expected. Please note that this does not necessarily mean that the environment is incorrect. DRL agents are notoriously difficult to train properly and can be sensitive to hyperparameters and training data. While poor performance may indicate an issue with the environment, it may also indicate that the observation space is insufficient for proper decision making, or that the chosen hyperparameters are not appropriate for the problem. The modeler is encouraged to rely on multiple streams of monitoring data, from both the environment and the agent, to verify the process. A good practice in training is to periodically evaluate the agent through a single episode, without exploration, and compare the results with both other agent behavior and the intended simulation outputs. Tools such as TensorBoard (Abadi et al. 2016), can be used to visualize the training process in real time and help modelers identify issues with the simulation. Further discussion on potential pitfalls with training DRL agents in Section 5.

4 PRACTICAL EXAMPLE: INVENTORY CONTROL - OPTIMIZING FILL RATE

In our example, which we also provide publicly on our GitHub repository (Ziliaskopoulos 2025), we are interested in simulating the inventory control process in a network of locations. While the overall system has many inherent complexities (i.e. truck capacity, storage layouts, etc.), we focus on the metrics that are important to our bottom line: inventory levels, lead times, stock-outs, and costs. We decide on modeling a network of critical infrastructure, and we are trying to identify the optimal order frequency, direction, and amount in a network of locations with a single, critical SKU, whose main objective is minimizing stock-outs and maximizing order fulfillment. This could be for a network of health facilities during an epidemic, military locations during conflict, or first responders during a crisis, for instance, where stock-outs can be devastating and cost is secondary. We will design the environment with a lost sales model, and a fully connected network with deterministic lead times. The action space, therefore, is defined as the set

Table 1: Gymnasium spaces types and their OR examples.

Name	Type	Description	OR Example
Box	Fundamental, Continuous/ Discrete	An n-dimensional box of bounded or unbounded float or int values. Can be continuous or discrete depending on dtype.	Order quantities shipped from n warehouses to m retailers in a supply chain.
Discrete	Fundamental, Discrete	A single integer space from ‘start‘ to ‘start + n - 1‘.	Choice of a machine to assign a job in job-shop scheduling.
MultiBinary	Fundamental, Binary	An n-shape binary space where each element is either 0 or 1.	Whether to activate a set of production facilities or not.
MultiDiscrete	Fundamental, Multi-discrete	A Cartesian product of multiple discrete spaces.	Allocation of shifts to workers where each shift has multiple options.
Text	Fundamental, Text	A space of strings with characters from a charset and bounded length.	Encoding routing codes for delivery paths.
Dict	Composite, Structured	A dict of spaces, enabling structured observations or actions.	Combining vehicle location (Box) and demand status (Discrete) in vehicle routing.
Tuple	Composite, Cartesian	A fixed-length tuple of other spaces. Useful for ordered heterogeneous inputs.	A (mode, quantity) action pair in inventory control.
Sequence	Composite, Variable-Length	A variable-length sequence of elements from a space.	A sequence of customer arrivals or delivery stops.
Graph	Composite, Graph	A graph structure with node and optional edge features.	Modeling a transportation or power network.
OneOf	Composite, Exclusive-Choice	A space representing a choice between different types of actions.	Choosing between transport modes (drive, rail, ship), each with unique parameters.

of possible order amounts per time step, $\mathbf{A} \in \mathbb{R}^{n \times n}$, where n is the number of locations in our network, $\mathbf{A}$ is our action matrix, $a_{i,j}$ is the quantity ordered from location i by location j, and $a_{i,j} = 0 \; \forall i = j$. The observation space is defined as the set of inventory levels at each location, $\mathbf{s} \in \mathbb{R}^n$, the pipeline inventory for each location of the next incoming order, $\mathbf{p} \in \mathbb{R}^n$, and the ratio of expected demand to inventory levels at each location, $\mathbf{d} \in \mathbb{R}^n$. The reward function, as shown in Figure 1, is defined as the overall systemic difference between filled orders and stockouts per time step,

$$R_t(\mathbf{A}) = \sum_{i=1}^{n} (o_{t,i} - \delta_{t,i}) \tag{1}$$

where:

- R_t is the reward at time step t,

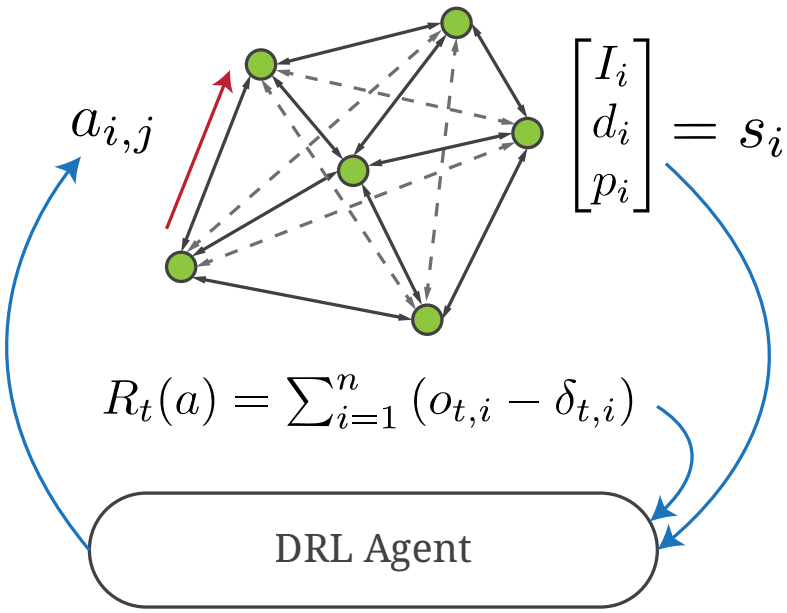

Figure 1: Conceptual diagram of the inventory control environment during a single timestep transition.

- $o_{t,i}$ is the number of orders filled at location i at time step t, and
- $\delta_{t,i}$ is the number of stockouts at location i at time step t.

In the code, we use a composite observation space of `Dict`, containing 3 `Box` subspaces. While in this case using a composite space is not necessary as our observation space is homogeneous, we wanted to showcase the composite spaces and their implementation. Our action space is also a `Box` space, here with $n \times n$ dimensions and a lower bound of 0.

As for our transition dynamics, we define the simulation logic in the `step()` function. The simulation is run in discrete time steps, where each step represents a day. We first generate incoming orders and demand for each location based on the user-defined parameters. The incoming orders are translated to demands per unit and per location, and the overall distribution of demand is defined by user input in the `init()` method and a custom auxiliary function `_generate_demand()`. We decide to use a composite demand function, where the demand is defined as a sum of uniform and binomial distributions, where the parameters are themselves also random variables. During each timestep, we first decrement the time until a pipeline order arrives. If the counter reaches 0, we increase the inventory level by the order amount and remove it from the pipeline. Then, we update the inventory levels at each location based on incoming demand. If the demand exceeds the inventory level, we record a stock-out and decrease the reward function according to Equation (1). Finally, we place orders in the network based on the action matrix. We update pipeline inventory and inventory levels based on outgoing orders and increment the reward function accordingly based on Equation (1). We repeat this process for each timestep until a predefined stopping condition is met.

Finally, as shown in Figure 2, we can simultaneously collect metrics and monitor both the simulation environment and the DRL agent to debug and identify the next steps. As an illustrative example, we configure

the inventory control environment with four locations, randomly assigning one location a significantly higher inventory level. We then train a PPO DRL agent to minimize shortages throughout the network.

On the right side of Figure 2, we see the monitoring outputs of the DRL agent's training process. The reward function converges to the maximum achievable level under the current episodic parameters, resulting in no shortages and a 100% fill rate. Additionally, the neural network used for value estimation converges to an almost zero total loss, indicating that the agent effectively understands the value of each state in relation to its reward function. The Kullback-Leibler Divergence (KL Divergence), named after Kullback and Leibler (1951), is a widely used component of loss functions in deep learning, measuring the difference between two probability distributions. In this context, it indicates the extent of change in the agent's policy distribution: a lower value signifies less change, meaning the agent is converging toward an optimal policy.

On the right side of the figure, we see the simulation outputs through the inventory levels of the four locations at 3 different snapshots of the PPO agent: after training for 50,000, 200,000, and 500,000 time-steps. Initially, the agent struggles to manage shortages, placing sparse, low-quantity orders once the initial stock is depleted. After 200,000 iterations, the agent begins to recognize the relationship between maintaining inventory levels and fulfilling orders, though performance remains inconsistent. However, after 500,000 time-steps, the agent consistently places orders at every time-step, prioritizing high inventory levels. This progression indicates that we can now incorporate penalties for large inventories and frequent ordering into the reward function, gradually integrating cost considerations without compromising the fill rate.

5 INTEGRATING WITH DEEP REINFORCEMENT LEARNING ALGORITHMS

While this tutorial focuses on the implementation of the DRL environment, the integration with DRL algorithms is inextricably linked to the environment design (see Figure 3). The choice of DRL algorithm can significantly impact the performance and efficiency of the training process. Conversely, some design choices in the environment can preclude the use of certain algorithms. For example, some algorithms, like DQN, require discrete action spaces, while others, like PPO, can handle continuous action spaces. There are many different DRL algorithms available, and many different implementation packages available. For someone newer to DRL implementations, we recommend using the `stable_baselines3` Python package (Raffin et al. 2021). While not as easy to modify as other packages, `stable_baselines3` is a well-documented and widely used package that provides easy access to many of the most popular DRL algorithms. It also integrates very well with `gymnasium` environments. In Section 6, we provide some alternatives to `stable_baselines3` and some additional resources for customizing and modifying the stock algorithms.

5.1 Hyperparameters in DRL

As mentioned in Section 3, DRL agents are very fickle during training. They are sensitive to hyperparameters, the training data, and the environment. For this reason, the majority of well-known DRL algorithms tend to implement some form of clipping or bounding method to prevent the agent from diverging too far from the training data. While the modeler can control the extent of this clipping through hyperparameters, each algorithm has its own set of hyperparameters and there are rarely one-size-fits-all solutions. Nevertheless, we recommend starting with default hyperparameter configurations and gradually tuning them based on performance. Some hyperparameters are counterintuitive. For example, batch size might be a familiar concept to the reader, and in traditional deep learning it is often a hyperparameter where larger values lead to higher stability. However, recent research (Obando Ceron et al. 2023) has identified that in many cases smaller batch sizes lead to both faster training times and better stability and performance.

Another counterintuitive yet very important hyperparameter is the function approximator neural network architecture. Again, in typical deep learning, large, deep networks tend to perform best, with commercial

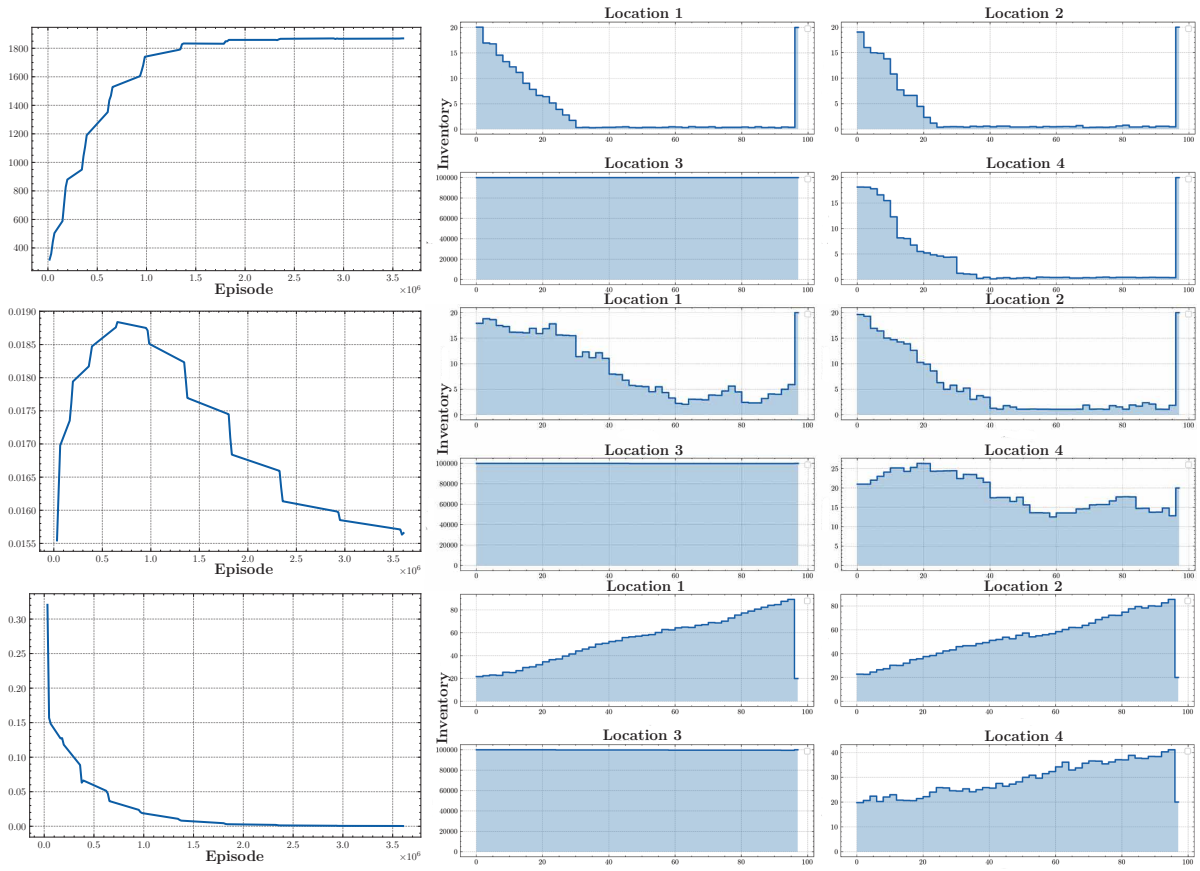

Figure 2: Showcase of monitoring and observability during simulation runs. On the left, 3 DRL agent metrics: Average episodic reward achieved, Approximate KL divergence and average value loss. On the right, inventory levels during 3 evaluation periods: after training for 50,000 time-steps, 200,000 time-steps, and 500,000 time-steps.

large language models (LLMs) having billions of parameters and vision models trained with near a hundred layers. However, in DRL, the opposite is often true. In typical deep learning models the model is trained supervised, with millions of labeled examples to learn from. In DRL, the model *generates its own data*, which can often be correlated, with sparse reward signals. A large network can easily overfit on the training noise, while a smaller network tends to generalize better. Another reason is the training stability of the agent, as larger networks introduce higher variance and divergence in the process. Smaller networks with less parameters are simply easier to stabilize than their larger counterparts. Finally, a crucial reason is the computational cost of training and speed of the interaction. As mentioned in Section 2, DRL agents require millions of interactions with the environment to properly learn and optimize the system dynamics. A larger network would impede both training and inference times, especially when many different training iterations are often necessary to fine tune the hyperparameters of the agent.

As for the other hyperparameters, there are ways to methodically tune them, from simpler methods, like a grid search, to more complex methods, such as Bayesian optimization or adaptive optimization algorithms. The reader is encouraged to look into Eimer et al. (2023) and Bakshy et al. (2018) for further information on the topic. However, simply running the agent with default settings and slowly iterating in

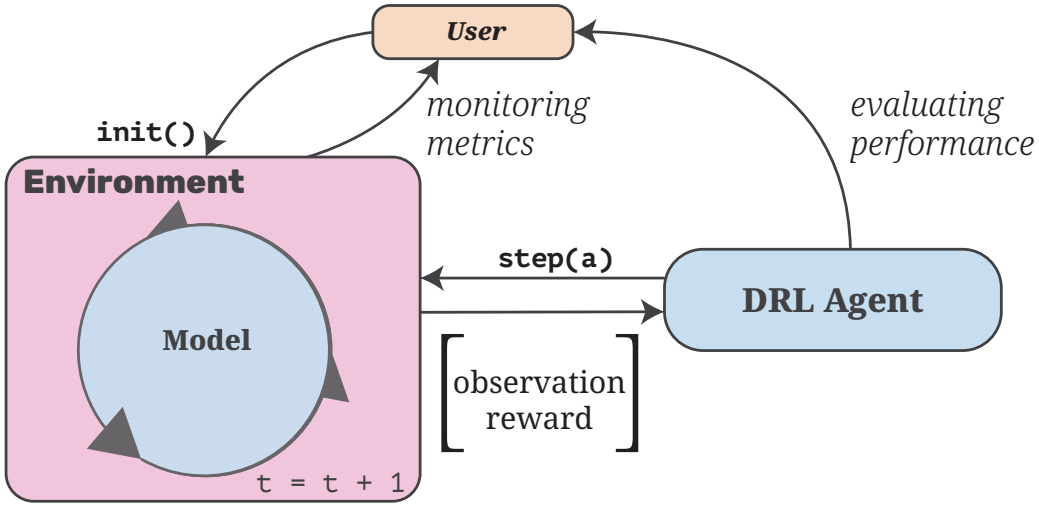

Figure 3: High-level overview of the DRL environment and its interaction with the agent and the user.

small implementations can often suffice, provided there is adequate monitoring of the environment and the training process.

6 FURTHER READING

For those interested in getting a foundational understanding of RL from both a theoretical and algorithmic perspective, we highly recommend the textbook by Sutton and Barto (2018). It is widely considered a seminal text in the field and provides a comprehensive overview of the inner workings of the RL and DRL algorithms used today. For a more practical introduction to DRL, we recommend the hands-on tutorials by the OpenAI team, (Achiam 2018), and the HuggingFace team, (Simonini T., and S. Omar 2023). Both tutorials give a brief theoretical overview but focus on practical implementations of DRL algorithms, environments, and examples.

A great resource for researchers looking to graduate from the stock implementations in `stable-baselines3`, or for practitioners looking to better understand the inner workings of a specific algorithm is the CleanRL GitHub repository and accompanying paper, (Huang et al. 2022). There the authors provide clean and well-documented, single file implementations of many popular DRL algorithms. We already mentioned PPO (Schulman et al. 2017), DQN (Mnih et al. 2013), and SAC (Haarnoja et al. 2018), which are provided, but the repository also contains implementations of other state-of-the-art algorithms, such as Deep Deterministic Policy Gradient (DDPG) (Lillicrap et al. 2015), Delayed Deep Deterministic Policy Gradient (TD3) (Fujimoto et al. 2018), and more. Researchers are encouraged to look into both the repository and accompanying papers for a deeper understanding of individual algorithms and their implementations.

Of course the field is currently evolving rapidly, and new packages and tools are being developed all the time. Researchers familiar with the field of Deep Learning might have heard of the `pytorch` package, which is an open-source library maintained by the PyTorch team at Meta AI and is currently one of the state-of-the-art toolboxes for deep learning implementations (Paszke 2019). The same team is currently also developing a package called `torchrl`, which is a low-level toolkit for designing DRL algorithms and handling data pipelines between agents and environments. It provides PyTorch-native abstractions and data types that are optimized computationally for DRL workloads, while also easily integrating into `gymnasium`

environments. While we do not recommend using `torchrl` for beginners, for researchers and practitioners looking for non-trivial DRL implementations (perhaps with large-scale experiments, custom architectures, multi-agent systems etc.) we encourage you to look into it further (Bou et al. 2023).

There are also many researchers working between the fields of OR and DL, either in theoretical or practical applications. Mazyavkina et al. (2021) provide a comprehensive literature review of the applications of DRL to combinatorial optimization (CO) problems, such as bin packing, minimum vertex cover, maximum independent set, and more. The authors also introduce the concept of combining DRL with graph neural networks (GNNs) to better model combinatorial problems. With the widespread use of networks in OR problems and the emerging field of GNNs, we encourage researchers to look into this intersection further. Readers interested in applications of GNNs in CO problems are also encouraged to look into Peng et al. (2021), or Ward et al. (2022) for a more introductory look into GNNs. Many researchers have also looked into the challenges of DRL implementations and how the models translate to real-world applications. Dulac-Arnold et al. (2021) formally define the challenges of real-world DRL applications, implement simulation environments for each challenge and benchmark state-of-the-art DRL agents on them, showcasing the fragility of learned policies to perturbations.

As a way to mitigate model uncertainty, Wachi et al. (2024) provide a comprehensive literature review of the field of safe reinforcement learning, or the use of DRL in constrained environments. Many OR problems are inherently constrained, and the authors provide a formal taxonomy on constraint formulation in reinforcement learning. In a similar vein, Moos et al. (2022) provide a literature review of the current state of the field of robust reinforcement learning, or how to cope with uncertainty in system dynamics. They provide an extensive view of the field for both single and multi-agent paradigms, and discuss the concept of Nash equilibriums in adversarial multi-agent systems to force agents to adapt to uncertain environments. Pinto et al. (2017) discuss this concept further in depth. Specifically in the field of inventory management, Boute et al. (2022) provide a recent comprehensive literature review of the field, research gaps and emerging topics.

Finally, for those interested in current applications and examples of DRL in OR and CO, we provide a short list of interesting recent papers. We mentioned Hubbs et al. (2020) in Section 1. Here the authors implement a set of environments for classic OR problems, such as knapsack, traveling salesman, and inventory management. In their paper, the authors provide the mathematical models, the pseudo-code algorithms, and the output metrics from each implementation. Similarly for CO, Berto et al. (2023) give example simulation environments, benchmarks and implementations for CO problems. Dai and Gluzman (2022) utilize PPO in queueing networks, and show that the agent would often converge to the optimal policy. In the field of supply chains, Oroojlooyjadid et al. (2017) implement multi-agent reinforcement learning to simulate a decentralized beer game and showcase transfer learning to mitigate the computational cost of training multiple agents by training one and transferring the knowledge to the others. Kaynov et al. (2024) benchmark DRL in inventory management, specifically in lost sales, dual source, and multi-echelon systems. Finally, Li et al. (2022) tackle the pickup and delivery problem, a variant of the vehicle routing problem, with DRL. They integrate the transformer attention mechanism in the agents and use that to develop an agent that both follows constraints and learns to optimize the routing while achieving good results.

7 CONCLUSIONS

This tutorial highlights the potential and practical approaches for incorporating DRL in OR by clearly mapping conventional simulation and OR components to DRL environment constructs. By demonstrating a structured process using Gymnasium, we addressed existing integration barriers caused by fragmented and evolving DRL frameworks. The inventory control example underscores DRL's suitability for real-world OR problems characterized by explicit objectives, measurable outcomes, and structured transition dynamics. Future work should focus on further simplifying DRL integration, with targeted one-to-one translations from commercial software, improving compatibility across libraries in the DRL ecosystem, and exploring

topics such as safe and robust reinforcement learning to enhance applicability in dynamic and uncertain operational environments.

REFERENCES

Abadi, M., A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, *et al*. 2016. "Tensorflow: Large-Scale Machine Learning on Heterogeneous Distributed Systems". *arXiv preprint arXiv:1603.04467*.

Achiam, J. 2018. *Spinning Up in Deep Reinforcement Learning*. San Francisco, CA: OpenAI. https://spinningup.openai.com/.

Bakshy, E., L. Dworkin, B. Karrer, K. Kashin, B. Letham, A. Murthy *et al*. 2018. "AE: A Domain-Agnostic Platform for Adaptive Experimentation". In *Conference on Neural Information Processing Systems*, 1–8.

Beck, K. 2022. *Test Driven Development: By Example*. Addison-Wesley Professional.

Bellman, R. 1984. *Dynamic Programming*. Princeton University Press.

Berto, F., C. Hua, J. Park, L. Luttmann, Y. Ma, F. Bu, *et al*. 2023. "RL4CO: an Extensive Reinforcement Learning for Combinatorial Optimization Benchmark". *arXiv preprint arXiv:2306.17100*.

Bertsekas, D., and J. N. Tsitsiklis. 1996. *Neuro-Dynamic Programming*. Athena Scientific.

Borshchev, A. 2014. *Multi-method modelling: AnyLogic*, 248–279. Wiley Online Library.

Bou, A., M. Bettini, S. Dittert, V. Kumar, S. Sodhani, X. Yang, *et al*. 2023. "TorchRL: A Data-Driven Decision-Making Library for PyTorch". *arXiv preprint arXiv:2306.00577*.

Boute, R. N., J. Gijsbrechts, W. Van Jaarsveld, and N. Vanvuchelen. 2022. "Deep Reinforcement Learning for Inventory Control: A Roadmap". *European Journal of Operational Research* 298(2):401–412.

Dai, J. G., and M. Gluzman. 2022. "Queueing Network Controls via Deep Reinforcement Learning". *Stochastic Systems* 12(1):30–67.

Dulac-Arnold, G., N. Levine, D. J. Mankowitz, J. Li, C. Paduraru, S. Gowal *et al*. 2021. "Challenges of Real-World Reinforcement Learning: Definitions, Benchmarks and Analysis". *Machine Learning* 110(9):2419–2468.

Eimer, T., M. Lindauer, and R. Raileanu. 2023. "Hyperparameters in Reinforcement Learning and How to Tune Them". In *International Conference on Machine Learning*, 9104–9149. PMLR.

Frostig, R., M. J. Johnson, and C. Leary. 2018. "Compiling Machine Learning Programs via High-Level Tracing". *Systems for Machine Learning* 4(9):1–3.

Fujimoto, S., H. Hoof, and D. Meger. 2018. "Addressing Function Approximation Error in Actor-Critic Methods". In *International Conference on Machine Learning*, 1587–1596. PMLR.

Guo, D., D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, *et al*. 2025. "DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning". *arXiv preprint arXiv:2501.12948*.

Haarnoja, T., A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, *et al*. 2018. "Soft Actor-Critic Algorithms and Applications". *arXiv preprint arXiv:1812.05905*.

Huang, S., R. F. J. Dossa, C. Ye, J. Braga, D. Chakraborty, K. Mehta *et al*. 2022. "CleanRL: High-Quality Single-File Implementations of Deep Reinforcement Learning Algorithms". *Journal of Machine Learning Research* 23(274):1–18.

Hubbs, C. D., H. D. Perez, O. Sarwar, N. V. Sahinidis, I. E. Grossmann, and J. M. Wassick. 2020. "OR-Gym: A Reinforcement Learning Library for Operations Research Problems". *arXiv preprint arXiv:2008.06319*.

Kaelbling, L. P., M. L. Littman, and A. R. Cassandra. 1998. "Planning and Acting in Partially Observable Stochastic Domains". *Artificial Intelligence* 101(1-2):99–134.

Kaufmann, E., L. Bauersfeld, A. Loquercio, M. Müller, V. Koltun, and D. Scaramuzza. 2023. "Champion-Level Drone Racing Using Deep Reinforcement Learning". *Nature* 620(7976):982–987.

Kaynov, I., M. Van Knippenberg, V. Menkovski, A. Van Breemen, and W. Van Jaarsveld. 2024. "Deep Reinforcement Learning for One-Warehouse Multi-Retailer Inventory Management". *International Journal of Production Economics* 267:109088.

Kullback, S., and R. A. Leibler. 1951. "On Information and Sufficiency". *The Annals of Mathematical Statistics* 22(1):79–86.

Li, J., L. Xin, Z. Cao, A. Lim, W. Song, and J. Zhang. 2022. "Heterogeneous Attentions for Solving Pickup and Delivery Problem via Deep Reinforcement Learning". *IEEE Transactions on Intelligent Transportation Systems* 23(3):2306–2315.

Lillicrap, T. P., J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, *et al*. 2015. "Continuous Control with Deep Reinforcement Learning". *arXiv preprint arXiv:1509.02971*.

Mazyavkina, N., S. Sviridov, S. Ivanov, and E. Burnaev. 2021. "Reinforcement Learning for Combinatorial Optimization: A Survey". *Computers & Operations Research* 134:105400.

Mnih, V., K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra *et al*. 2013. "Playing Atari with Deep Reinforcement Learning". *arXiv preprint arXiv:1312.5602*.

Moos, J., K. Hansel, H. Abdulsamad, S. Stark, D. Clever, and J. Peters. 2022. "Robust Reinforcement Learning: A Review of Foundations and Recent Advances". *Machine Learning and Knowledge Extraction* 4(1):276–315.

Obando Ceron, J., M. Bellemare, and P. S. Castro. 2023. "Small Batch Deep Reinforcement Learning". *Advances in Neural Information Processing Systems* 36:26003–26024.

Oroojlooyjadid, A., M. Nazari, L. Snyder, and M. Takáč. 2017. "A Deep Q-Network for the Beer Game: A Reinforcement Learning Algorithm to Solve Inventory Optimization Problems". *arXiv preprint arXiv:1708.05924*.

Paszke, A. 2019. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". *arXiv preprint arXiv:1912.01703*.

Pegden, C. D. 2007. "Simio: A new simulation system based on intelligent objects". In *2007 Winter Simulation Conference (WSC)*, 2293–2300 https://doi.org/10.1109/WSC.2007.4419867.

Peng, Y., B. Choi, and J. Xu. 2021. "Graph Learning for Combinatorial Optimization: a Survey of State-of-the-Art". *Data Science and Engineering* 6(2):119–141.

Pinto, L., J. Davidson, R. Sukthankar, and A. Gupta. 2017. "Robust Adversarial Reinforcement Learning". In *International Conference on Machine Learning*, 2817–2826. PMLR.

Powell, W. B. 2011. *Approximate Dynamic Programming: Solving the Curses of Dimensionality*. First ed. Wiley Series in Probability and Statistics. Wiley.

Powell, W. B. 2019. "A Unified Framework for Stochastic Optimization". *European Journal of Operational Research* 275(3):795–821.

Raffin, A., A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann. 2021. "Stable-baselines3: Reliable reinforcement learning implementations". *Journal of Machine Learning Research* 22(268):1–8.

Robinson, S. 2008. "Conceptual Modelling for Simulation Part I: Definition and Requirements". *Journal of the Operational Research Society* 59(3):278–290.

Schulman, J., F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. 2017. "Proximal Policy Optimization Algorithms". *arXiv preprint arXiv:1707.06347*.

Silver, D., J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, *et al*. 2017. "Mastering the Game of Go without Human Knowledge". *Nature* 550(7676):354–359.

Sutton, R. S., and A. G. Barto. 2018. *Reinforcement Learning: An Introduction*. Second ed. Adaptive Computation and Machine Learning Series. The MIT Press.

Simonini T., and S. Omar 2023. "The Hugging Face Deep Reinforcement Learning Class". https://github.com/huggingface/deep-rl-class.

Towers, M., A. Kwiatkowski, J. Terry, J. U. Balis, G. De Cola, T. Deleu, *et al*. 2024. "Gymnasium: A standard interface for reinforcement learning environments". *arXiv preprint arXiv:2407.17032*.

Wachi, A., X. Shen, and Y. Sui. 2024. "A Survey of Constraint Formulations in Safe Reinforcement Learning". *arXiv preprint arXiv:2402.02025*.

Ward, I. R., J. Joyner, C. Lickfold, Y. Guo, and M. Bennamoun. 2022. "A Practical Tutorial on Graph Neural Networks". *ACM Computing Surveys (CSUR)* 54(10s):1–35.

Williams, R. J. 1992. "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning". *Machine Learning* 8:229–256.

Ziliaskopoulos, K. 2025. "Example DRL Inventory Environment". https://doi.org/10.5281/zenodo.16746925.

AUTHOR BIOGRAPHIES

KONSTANTINOS ZILIASKOPOULOS is a Ph.D. candidate in the Dept. of Industrial & Systems Engineering at Auburn University. His research interests include the intersection of data analytics and artificial intelligence with the fields of Operations Research and Combinatorial Optimization. His email address is kzz0034@auburn.edu.

ALEXANDER VINEL is an Associate Professor in the Industrial & Systems Engineering Dept. of Auburn University. He holds a doctorate degree in industrial engineering from the University of Iowa. His research interests are in the areas of stochastic optimization and risk-averse decision making, with applications in transportation systems and data analytics. He is an Area Editor of Computers & Operations Research. His email address is alexander.vinel@auburn.edu.

ALICE E. SMITH is the Joe W. Forehand, Jr. Distinguished Professor of the Industrial & Systems Engineering Dept. at Auburn University. Her research focus is analysis, modeling, and optimization of complex systems with emphasis on computation inspired by natural systems integrated with OR and statistics. She holds one U.S. and several international patents and has authored publications with over 18,000 citations. Dr. Smith has been a principal investigator on over $12 million of sponsored research with funding by Dept. of Homeland Security, NASA, U.S. Dept. of Defense, Missile Defense Agency, National Security Agency, NIST, U.S. Dept. of Transportation, Frontier Technologies Inc., Lockheed Martin, Adtranz, the Ben Franklin Technology Center of Western Pennsylvania, and U.S. National Science Foundation, from which she has been awarded 18 distinct grants including a CAREER grant and an ADVANCE Leadership grant. Dr. Smith is a member of the National Academy of Engineering (NAE), a Life Fellow of the IEEE, a Fellow of INFORMS, a Fellow of the IISE, a senior member of the Society of Women Engineers, a member of Tau Beta Pi, and a Registered Professional Engineer. Her email address is smithae@auburn.edu.