Optimizing the Dynamic Drone-Assisted Pickup and Delivery Problem with Deep Reinforcement Learning

Timothy Mulumba
Department of Civil & Urban Engineering
New York University

Abstract

We investigate the *dynamic drone-assisted pickup and delivery problem* (DAPDP), which concerns real-time, on-demand routing decisions in scenarios where new orders arrive stochastically throughout the day. By leveraging a fleet of trucks each equipped with a drone, operators can split tasks between ground vehicles and aerial vehicles, aiming to minimize total travel costs while respecting constraints on time windows, capacity, and drone flight endurance. We propose a deep reinforcement learning (DRL) approach based on Q-learning, augmented by a neural network function approximator, to decide dynamically which newly arrived orders to dispatch and how to integrate drone sorties effectively. Our experiments on a large, real-world-inspired dataset demonstrate substantial performance gains over greedy, random, and lazy dispatch baselines, yielding 10.6%, 22.6%, and 37.2% savings, respectively, in total travel cost. Additionally, comparison against a clairvoyant oracle solution shows that our approach is near optimal.

1 Introduction

The rapid expansion of e-commerce and ever-increasing customer demand for instantaneous deliveries have pushed logistics and supply chain providers to optimize last-mile operations (Pillac et al., 2013; Psaraftis et al., 2016; Voccia et al., 2019). The last mile is widely recognized as the most time-consuming and expensive segment of the supply chain, often accounting for up to 30% of total shipping costs. Consequently, developing innovative methods for fast, cost-effective, and reliable delivery has become an urgent priority (Murray & Chu, 2015).

Drone-Assisted Delivery. Unmanned Aerial Vehicles (UAVs), commonly referred to as drones, have gained traction as a complementary tool to ground vehicles (trucks or vans) (Ponza, 2016). While drones may have limited payload capacity and flight range, their potential for bypassing traffic congestion and traveling straight-line distances can yield notable efficiency improvements. They can reduce not only the physical distance traveled by trucks but also the delivery time, particularly for high-priority parcels. However, integrating drones into an existing ground-based delivery system requires careful coordination, as multiple interdependent decisions arise: Where should a drone be launched? Which requests can be feasibly served by a drone? When should the truck wait or proceed, and how to rendezvous again with the drone?

Dynamic Nature. The *dynamic* aspect of this problem imposes additional complexity. In many scenarios, especially in on-demand or same-day delivery, requests are not known fully in advance. Instead, they appear stochastically throughout the day, each with a required time window in which service must occur. Standard static approaches become suboptimal if they ignore newly arrived requests after route dispatch.

In this work, we analyze the *dynamic drone-assisted pickup and delivery problem* (DAPDP), where a fleet of trucks, each with an onboard drone, must make repeated dispatch decisions over multiple

Differentiable Learning of Combinatorial Algorithms Workshop at NeurIPS 2025.

epochs (e.g., hourly). We aim to minimize the total cost (or total distance) across all epochs. This cost is composed mainly of the driving distance of trucks plus a smaller drone distance penalty, subject to drone endurance limits, time windows, and capacity constraints.

Key Challenges.

- *Coordination of Vehicles*: The simultaneous control of trucks and drones is a nontrivial scheduling and routing challenge.
- *Rolling Horizon Updates*: Requests appear gradually, requiring repeated re-optimization of routes without significantly disrupting ongoing service.
- *Large Combinatorial Action Space*: At each epoch, deciding which requests to dispatch and which to defer grows exponentially with the number of unserved requests.
- *Tradeoff Between Immediate and Future Dispatch*: A request might become cheaper to serve if partially consolidated with future arrivals, but deferring requests too long risks losing feasibility or incurring penalties.

Our Contribution.

(i) *A DRL Framework for Dynamic DAPDP Dispatch*: We propose a novel deep Q-learning approach to learn an effective dispatch policy for the complex, dynamic drone-assisted pickup and delivery problem. Our agent dynamically selects which newly arrived or pending orders to dispatch to a truck-drone fleet at each decision epoch, aiming to minimize long-term operational costs.
(ii) *Demonstrated Near-Optimal Performance and Generalization*: Through extensive experiments on large-scale, real-world-inspired DAPDP instances, our DRL agent achieves substantial cost reductions (e.g., 37.2% over lazy dispatch) and performs within 1.2% of a clairvoyant oracle possessing full future knowledge. This establishes a new benchmark for DRL performance in complex, city-scale drone-assisted logistics. Furthermore, we demonstrate the policy's ability to generalize to unseen demand patterns and new city environments.

2 Problem Formulation

This section builds upon the static DAPDP definition and extends it to a dynamic environment. We summarize the critical elements below. The full static DAPDP formulation is provided in Appendix A.

2.1 Static DAPDP Summary

Given:

- $\mathcal{V}$: set of trucks, each equipped with a single drone,
- $\mathcal{N}$: set of all locations (depot + customer sites),
- $\mathcal{R}$: set of requests, each requiring pickup and/or drop-off,
- Distances d_{ij} for trucks and d_{ij}^{drone} for drones,
- Time windows $[e_i, l_i]$ for each request/location i,
- Drone endurance E, capacity constraints, and route connectivity.

Trucks can serve multiple requests directly, while drones can be dispatched for certain requests if feasible (payload, time windows, flight range). The static objective is to minimize:

$$\text{Total Cost} = \sum_{v \in \mathcal{V}} \sum_{(i,j)} d_{ij}\, x_{ij}^{v} \;+\; \alpha \sum_{v \in \mathcal{V}} \sum_{(i,j,k)} d_{ij}^{\text{drone}}\, y_{ijk}^{v},$$

subject to constraints ensuring every request is served exactly once, time windows are respected, and drones do not exceed endurance. Constraints such as time windows are treated as hard constraints in the static sub-problem. Any dispatch subset that cannot be feasibly routed by the sub-solver (e.g., due to unavoidable time window violations) results in a penalty for the DRL agent, as described in Section 4.1.

2.2 Dynamic Setting

In a dynamic scenario, requests $\mathcal{R}$ are partially unknown initially. They arrive in discrete time epochs $t = 1, \ldots, T$. At each epoch:

- New requests $\mathcal{R}_t$ are sampled randomly without replacement from a pool (with known locations, demands, earliest start e_i, and latest start l_i).
- The system chooses a subset $S_t \subseteq \mathcal{R}_{\text{active}}$ to dispatch, solving a static sub-problem restricted to those requests.
- The cost $C(S_t)$ is accrued, and the agent observes a reward $r_t = -C(S_t)$.
- The environment transitions to $t + 1$, revealing newly arrived requests $\mathcal{R}_{t+1}$. Requests not served remain active if their windows still permit future service.

A request becomes a *must-go* if l_i is approaching such that deferral is no longer feasible. By epoch T, all requests must be served.

Goal. Maximize cumulative reward (minimize total travel cost) over T epochs, subject to feasibility each step.

3 Approach: Deep Reinforcement Learning

We model the dynamic DAPDP as a Markov Decision Process (MDP) and solve it with Deep Q-learning. Below, we outline each MDP component and the associated neural network architecture. Further details of our approach appear in Appendix C.

3.1 MDP Components

State s_t. Encodes:

- **Global Features:** Current epoch index t, time remaining in the planning horizon, current time, planning start time, total number of active requests ($\mathcal{R}_{\text{active}}$), drone endurance, and truck capacity.
- **Per-Request Features:** For each of the $\mathcal{R}_{\text{active}}$ active requests, a feature vector is constructed. This vector includes:
  - *Request Attributes:* Coordinates (pickup/drop-off), demand, time window (earliest/latest start), service time, a 'must-go' flag (if the request is nearing its deadline).
  - *Contextual Features:* Information about its relationship to other requests, such as features derived from the time/distance to its k nearest pending neighbors (e.g., $k = 5$).
  - *Drone Eligibility:* A flag or features indicating if the request is suitable for drone delivery.

Action a_t. Specifies a binary decision (dispatch or defer) for each active request. We then solve a sub-problem for the dispatched set to obtain the cost. This sub-solver is a specialized static DAPDP routine.

Transition. The environment checks feasibility, calculates cost, applies a large negative penalty for infeasibility, and reveals new requests for $t + 1$.

Reward r_t.

$$r_t = -\big(\text{TruckDistance} + \alpha \cdot \text{DroneDistance}\big),$$

plus an extra negative penalty for violation of constraints.

3.2 Deep Q-Network Method

We represent $Q(s, a; \theta)$ by a multi-layer feedforward neural network with parameters θ. We adopt the classic Q-learning update (Sutton & Barto, 2018):

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha^l \Big[r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \Big].$$

In practice, we use a *target network*, experience replay, and mini-batch stochastic gradient descent to stabilize training (Mnih et al., 2015).

3.2.1 Network Architecture

Input Layer The state s_t provided to the DQN comprises features encoding the dynamic environment. The collection of global features and the set of $\mathcal{R}_{\text{active}}$ per-request feature vectors form the input to the DQN. Since the number of active requests $\mathcal{R}_{\text{active}}$ varies, padding is employed to ensure a consistent input dimension for the network.

Hidden Layers The DQN employs a multi-layer perceptron (MLP) architecture with sizes $[128, 64, 32, 16, 8]$ to process these features.

Output Layer and Per-Request Action Scores For each of the $\mathcal{R}_{\text{active}}$ active requests currently being considered, the network outputs two Q-values (scores): $Q(s_t, \text{dispatch}_i; \theta)$ representing the expected return if request i is chosen for dispatch in the current epoch, and $Q(s_t, \text{defer}_i; \theta)$ representing the expected return if request i is deferred.

Forming the Dispatch Subset S_t (Action Selection) Based on these Q-values, a decision is made for each active request $i \in \{1, \ldots, |\mathcal{R}_{\text{active}}|\}$. Specifically, an ϵ-greedy policy selects between dispatching request i or deferring it:

- With probability $1 - \epsilon$, action $\mathcal{R}^*_{\text{active, i}} = \text{argmax}_{\mathcal{R}_{\text{active, i}} \in \{\text{dispatch}_i, \text{defer}_i\}} Q(s_t, \mathcal{R}_{\text{active, i}}; \theta)$ is chosen.
- With probability ϵ, a random action (dispatch or defer) is chosen for request i.

The overall action a_t for the epoch is the set of these $\mathcal{R}_{\text{active}}$ individual decisions. The subset of requests $S_t = \{\text{request } i \mid \text{the chosen action for } i \text{ was dispatch}_i\}$ is then passed to the static DAPDP sub-solver.

3.2.2 Q-Value Estimation and Training Update

After the dispatch subset S_t is determined by making decisions for all requests in the set of active requests $\mathcal{R}_{\text{active}}$, the sub-solver computes the actual cost $C(S_t)$ incurred to service these specific requests. The immediate global reward for the epoch is $r_t = -C(S_t)$ (this may also include penalties for infeasibility or additions from reward shaping, as per Section 4.2).

The Q-network's parameters θ are updated using mini-batches sampled from an experience replay buffer. For each transition (s_t, a_t, r_t, s_{t+1}) in a batch, where a_t represents the collection of individual dispatch/defer decisions for all requests in $\mathcal{R}_{\text{active}}$:

For each request $i \in \mathcal{R}_{\text{active}}$, let d_i^{chosen} be the specific action (either dispatch_i or defer_i) taken for request i as part of the composite action a_t at state s_t. The target value y_i for updating $Q(s_t, d_i^{\text{chosen}}; \theta)$ is computed as:

$$y_i = \text{reward\_for\_decision}_i + \gamma \max_{d'_i \in \{\text{dispatch}_i, \text{defer}_i\}} Q(s_{t+1}, d'_i; \theta^-)$$

where θ^- are the parameters of the target network, and γ is the discount factor.

The component $\text{reward\_for\_decision}_i$ is defined based on the chosen action d_i^{chosen} and the global reward r_t:

$$\text{reward\_for\_decision}_i = \begin{cases} r_t/|S_t| & \text{if } d_i^{\text{chosen}} = \text{dispatch}_i,\ |S_t| > 0 \\ 0 & \text{otherwise} \end{cases}$$

The loss for updating the Q-network is then typically calculated as the sum of squared differences between the target y_i and the predicted Q-value $Q(s_t, d_i^{\text{chosen}}; \theta)$, averaged over all individual request decisions in the mini-batch:

$$\mathcal{L}(\theta) = \mathbb{E}_{(s_t, a_t, r_t, s_{t+1}) \sim \mathcal{B}} \left[\sum_{i \in \mathcal{R}_{\text{active}}} \left(y_i - Q(s_t, d_i^{\text{chosen}}; \theta) \right)^2 \right]$$

where $\mathcal{B}$ is the replay buffer, and the sum is over all requests i in the set of active requests $\mathcal{R}_{\text{active}}$ for state s_t.

Having detailed our DRL methodology for the dynamic DAPDP, we now turn to its empirical evaluation.

4 Experiments

4.1 Experimental Setup

Data. We have access to 300 real-world-based static instances from an urban logistics company, each containing 100–200 requests with known coordinates, demands, time windows, and travel times. We augmented these by defining drone-eligible orders based on package characteristics, assigning service times, generating drone flight times assuming straight-line travel with a speed factor, and setting drone endurance parameters based on typical commercial drone specifications (e.g., 15-25 minutes flight, payload limits translating to drone eligibility for certain demands). We create dynamic instances by revealing requests over 5–9 epochs. Each epoch can have up to 100 newly arrived or pending requests.

Drone and Truck Parameters. We assume a sufficiently large pool of homogeneous trucks is available, each with a capacity of 200 parcels, and each equipped with a drone subject to an endurance of E (15-25 minutes). We set the relative cost of the drone to truck α, to 0.3.

When the DRL agent selects a subset of orders S_t for dispatch, the static DAPDP sub-solver determines the routes and the minimum number of truck-drone pairs from this pool required to serve S_t, respecting all capacity and endurance constraints. The objective function (Section 2.1), by minimizing total travel distance, implicitly encourages the use of fewer vehicles. This 'effectively unrestricted' fleet assumption ensures any feasible subset of orders can be routed, allowing focus on the dynamic dispatch decision. The impact of finite fleet capacity is explored in our generalization experiments (Appendix E).

Baselines.

1. **Greedy Dispatch**: Dispatch all newly arrived requests immediately.
2. **Random Dispatch**: Choose a random fraction (e.g., 50%) of new requests to dispatch each epoch.
3. **Lazy Dispatch**: Only dispatch requests that must be served (time window closing), deferring others.
4. **Oracle**: We add a clairvoyant *oracle* that assumes full future knowledge for each episode and solves the static counterpart optimally via Mulumba et al. (2024).

Each baseline solves the same sub-problem to ensure a fair cost comparison.

Infinite-Fleet Assumption & Invalid Solution Handling

To guarantee feasibility, we follow industry practice and assume an unrestricted vehicle fleet. While theoretically any request can thus be served, the objective still penalizes superfluous trips, incentivizing minimal fleet use. When the agent proposes an infeasible action (e.g., serving after a time-window expires), the episode terminates and a structured penalty $r_{\text{penalty}} = -\beta(c_{\text{base}} + \sum_i v_i)$ with β=2 is applied, where c_{base} tracks the worst valid cost seen, and v_i is the magnitude of the violation of constraint i. This dense signal proved more stable than a flat FAIL reward (Appendix C.2.3).

Implementation. We implement our DRL algorithm in Python with PyTorch. The sub-solver for each dispatched subset is a custom solver with time-window constraints and drone endurance checks based on the work of Mulumba et al. (2024). We cap each sub-solver run at a few seconds of CPU time for each epoch to maintain real-time feasibility.

4.2 Training Procedure

We train the Q-network for up to 200,000 steps, collecting transitions (s_t, a_t, r_t, s_{t+1}) in a replay buffer of size 100,000. We sample mini-batches of size 32 to update network parameters with the Adam optimizer (learning rate 10^{-4}). A separate target network is updated every 1,000 steps to reduce instability.

Reward Shaping. A small intermediate reward is added for partial route improvements (e.g., if the total route cost is decreasing compared to previous solutions). This shaping helps guide the agent in earlier training stages, though the final objective remains the negative route cost.

4.3 Key Results

Table 1 summarizes cost outcomes across 100 dynamic test scenarios. Our approach reduces total distance compared to the Greedy baseline by about 10.6%, beating Random and Lazy even more significantly. Overall feasibility rates are above 99%, indicating robust satisfaction of time windows and drone constraints.

Table 1: Results on dynamic DAPDP with up to 9 epochs, averaged over 100 test scenarios. Standard deviations in parentheses.

METHOD	TOTAL COST	COST INCREASE VS. OURS	FEASIBILITY %
GREEDY	419,210 ($\pm$2730)	+10.6%	98.7%
RANDOM	464,530 ($\pm$2980)	+22.6%	97.2%
LAZY	520,056 ($\pm$3170)	+37.2%	99.0%
OURS (DQN)	379,050 ($\pm$1880)	–	99.3%

Oracle Baseline. Our DQN policy yields a cost only $1.2 \pm 0.03\%$ worse - demonstrating competitive optimality while preserving real-time decision capability.

4.4 Analysis of Learned Dispatch Policy

The RL-based policy generally defers non-urgent requests to consolidate them with future arrivals, especially if drone usage might reduce the truck's total distance. However, it avoids extreme deferrals that risk feasibility. By contrast, the Greedy baseline dispatches everything immediately, missing consolidation opportunities, while the Lazy baseline overly defers requests, potentially missing synergy with earlier arrivals or incurring last-minute surges.

5 Discussion

5.1 Managerial Insights

Consolidation Gains. Our experiments show that selectively deferring certain requests yields significant mileage savings. The learned dispatch policy, informed by a global perspective of upcoming requests, can better exploit synergy between truck and drone tasks.

Real-Time Adaptability. Because the model is pretrained offline, real-time inference is fast. For each epoch, the agent needs only a forward pass of the neural network plus a short sub-solver run. This approach is suitable for dynamic contexts like grocery deliveries or same-day shipping, where quick re-optimizations are demanded.

Scalability to Large Instances. While each sub-problem solver complexity grows with instance size, the dispatch policy drastically reduces the search space by focusing on a subset of requests at a time. In practice, scheduling up to 200 requests remains tractable with an efficient heuristic.

Why DQN (Value-based) over Policy Optimization We compared REINFORCE and Proximal Policy Optimization (PPO) baselines: PPO attained $+8.7\%$ cost and required twice the wall-clock time owing to on-policy sampling, whereas value-based replay enabled by DQN leveraged the heavy-tailed diversity of dynamic arrivals. Details in Appendix D.

6 Conclusion

We have presented a deep reinforcement learning strategy for the *dynamic drone-assisted pickup and delivery problem*, effectively coordinating ground vehicles and drones to handle continually arriving requests. Our Q-learning framework learns a dispatch policy that determines which subset of requests to serve at each epoch, balancing present cost against future arrival uncertainties. Empirical tests confirm notable cost savings over conventional heuristics in large-scale scenarios, demonstrating that a single DRL agent can generalize across diverse operational conditions.

Unlike classical dispatchers, our framework jointly optimizes routing and drone launch/recall without pre-scheduling. The policy's near-oracle gap confirms its efficacy, while ablations reveal that merely scaling MLP depth, buffer size, or learning rate has marginal impact compared with our architectural choices (see Appendices F and G).

References

Bello, I., Pham, H., Le, Q. V., Norouzi, M., and Bengio, S. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

Bent, R. and Van Hentenryck, P. Scenario-based planning for partially dynamic vehicle routing with stochastic customers. *Operations Research*, 52(6):977–987, 2004.

Cordeau, J.-F., Laporte, G., Potvin, J.-Y., and Savelsbergh, M. W. Transportation on demand. *Handbooks in operations research and management science*, 14:429–466, 2007.

Géron, A. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*. " O'Reilly Media, Inc.", 2022.

Joe, W. and Lau, H. C. Deep reinforcement learning approach to solve dynamic vehicle routing problem with stochastic customers. In *Proceedings of the international conference on automated planning and scheduling*, volume 30, pp. 394–402, 2020.

Kool, W., Van Hoof, H., and Welling, M. Attention, learn to solve routing problems! *arXiv preprint arXiv:1803.08475*, 2018.

Kwon, Y.-D., Choo, J., Kim, B., Yoon, I., Gwon, Y., and Min, S. Pomo: Policy optimization with multiple optima for reinforcement learning. *Advances in Neural Information Processing Systems*, 33:21188–21198, 2020.

Li, X., Luo, W., Yuan, M., Zhang, K., Zhao, T., Liu, Z., and He, X. Learning to optimize industry-scale dynamic pickup and delivery problems. In *2021 IEEE 37th international conference on data engineering (ICDE)*, pp. 2511–2522. IEEE, 2021.

Ma, Y., Hao, X., Hao, J., Li, J., Wang, Z., Zhou, W., and Yu, Y. A hierarchical reinforcement learning based optimization framework for large-scale dynamic pickup and delivery problems. *Advances in neural information processing systems*, 34:23609–23620, 2021.

Mitrović-Minić, S. and Laporte, G. Waiting strategies for dynamic vehicle routing. *Transportation Research Part B: Methodological*, 38(7):635–655, 2004.

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

Mulumba, T., Najy, W., and Diabat, A. The drone-assisted pickup and delivery problem: An adaptive large neighborhood search metaheuristic. *Computers & Operations Research*, 161:106435, 2024.

Murray, C. C. and Chu, A. G. The flying sidekick traveling salesman problem: Optimization of drone-assisted parcel delivery. *Transportation Research Part C: Emerging Technologies*, 54: 86–109, 2015.

Nazari, M., Oroojlooy, A., Snyder, L., and Takác, M. Reinforcement learning for solving the vehicle routing problem. *Advances in neural information processing systems*, 31, 2018.

Paul, S. and Chowdhury, S. Learning to allocate time-bound and dynamic tasks to multiple robots using covariant attention neural networks. *Journal of Computing and Information Science in Engineering*, 24(9), 2024.

Pillac, V., Gendreau, M., Guéret, C., and Medaglia, A. L. A review of dynamic vehicle routing problems. *European Journal of Operational Research*, 225(1):1–11, 2013.

Pina-Pardo, J. C., Silva, D. F., Smith, A. E., and Gatica, R. A. Fleet resupply by drones for last-mile delivery. *European Journal of Operational Research*, 316(1):168–182, 2024.

Ponza, A. Optimization of drone-assisted parcel delivery. *Thesis*, 2016.

Psaraftis, H. N., Wen, M., and Kontovas, C. A. Dynamic vehicle routing problems: Three decades and counting. *Networks*, 67(1):3–31, 2016.

Sutton, R. S. and Barto, A. G. *Reinforcement learning: An introduction*. MIT press, 2018.

Toth, P. and Vigo, D. *Vehicle routing: problems, methods, and applications*. SIAM, 2014.

Ulmer, M. W. and Thomas, B. W. Same-day delivery with heterogeneous fleets of drones and vehicles. *Networks*, 70(1):3–19, 2017.

Voccia, S. A., Campbell, A. M., and Thomas, B. W. Same-day delivery with drone resupply. *Transportation Science*, 53(1):236–252, 2019.

Zhou, J., Zheng, L., and Fan, W. Multirobot collaborative task dynamic scheduling based on multiagent reinforcement learning with heuristic graph convolution considering robot service performance. *Journal of Manufacturing Systems*, 72:122–141, 2024.

A Full Static DAPDP Model

Here, we present the complete integer linear programming (ILP) formulation for the *static drone-assisted pickup and delivery problem* (DAPDP). We also provide a short explanation for each constraint in brackets. This is the foundation for the dynamic case described in the main paper.

A.1 Sets and Parameters

- $\mathcal{V}$: set of trucks, indexed by v.
- $\mathcal{N}$: set of all nodes (depot plus customers). We let 0 denote the depot.
- $\mathcal{R}$: set of requests (pickup and delivery points can be merged into the same node set with additional logic, or separated).
- d_{ij}: travel distance/time for truck between nodes i and j.
- d_{ij}^{drone}: flight distance/time for drone between i and j.
- T_i: service time at node i.
- Q_v: capacity of truck v.
- q_i: demand (or supply) of request i (assuming net flow representation).
- E: maximum endurance (flight time limit) for the drone.
- e_i, l_i: earliest and latest start of service time window at node i.
- α: scaling factor for drone distance in the objective.

A.2 Decision Variables

- $x_{ij}^v \in \{0,1\}$: 1 if truck v drives directly from node i to node j.
- $y_{ijk}^v \in \{0,1\}$: 1 if the drone of truck v flies from node i to node j and rejoins truck v at node k (three-location path).
- $u_i^v \geq 0$: the time at which truck v starts service at node i.
- w_i^v: load of truck v upon arrival at node i (if relevant for capacity).

Objective Function:

$$\min \quad \sum_{v \in \mathcal{V}} \sum_{(i,j) \in \mathcal{N} \times \mathcal{N}} d_{ij}\, x_{ij}^v \;+\; \alpha \sum_{v \in \mathcal{V}} \sum_{(i,j,k)} d_{ij}^{\text{drone}}\, y_{ijk}^v \tag{1}$$

[Minimize combined truck travel and drone travel (scaled by α)]

A.3 Constraints

We list each constraint with a brief explanation in brackets.

Truck Flow Conservation:

$$\sum_{j \in \mathcal{N}} x_{ij}^v - \sum_{j \in \mathcal{N}} x_{ji}^v + \sum_{j,k} y_{ijk}^v - \sum_{h,k} y_{hik}^v \;=\; 0 \quad \forall i \in \mathcal{N},\, v \in \mathcal{V}. \tag{2}$$

[Ensure that for each node, the net flow of truck plus drone is zero; i.e. each node has equal in-flow and out-flow for truck v.]

Visit Constraints:

$$\sum_{v \in \mathcal{V}} \sum_{j \in \mathcal{N}} x_{ij}^v + \sum_{v \in \mathcal{V}} \sum_{j,k} y_{ijk}^v \;=\; 1 \quad \forall i \in \mathcal{R}. \tag{3}$$

[Each request node i is visited exactly once, either by truck or by a drone sortie.]

Truck Capacity:

$$w_j^v \;=\; w_i^v + q_j \quad \text{if } x_{ij}^v = 1, \tag{4}$$

$$w_j^v \;\le\; Q_v \quad \forall j, v, \tag{5}$$

[Capacity constraints to ensure the truck does not exceed its capacity when moving from node i to j. Summarized for brevity.]

Drone Endurance:

$$(d_{ij}^{\text{drone}} + d_{jk}^{\text{drone}})\; y_{ijk}^v \;\le\; E \quad \forall i, j, k, v. \tag{6}$$

[If a drone sortie is used from i to j and back to k, the total flight time cannot exceed E.]

Time Window Feasibility (Truck):

$$u_j^v \;\ge\; u_i^v + T_i + d_{ij} \;-\; M \cdot (1 - x_{ij}^v) \quad \forall i, j, v, \tag{7}$$

$$e_i \;\le\; u_i^v \;\le\; l_i \quad \forall i, v, \tag{8}$$

[The start of service at j for truck v depends on its arrival from i plus service time; also each u_i^v must be within the time window.]

Time Feasibility (Drone):

$$u_j^v \;\ge\; u_i^v + T_i + d_{ij}^{\text{drone}} \;-\; M \cdot (1 - y_{ijk}^v) \quad \forall i, j, k, v, \tag{9}$$

$$u_k^v \;\ge\; u_i^v + T_i + d_{ij}^{\text{drone}} + d_{jk}^{\text{drone}} \;-\; M \cdot (1 - y_{ijk}^v) \quad \forall i, j, k, v, \tag{10}$$

[Ensures correct timing for drone departure from i, arrival at j, and rendezvous with truck at k.]

Depot Start/End:

$$\sum_j x_{0j}^v + \sum_{j,k} y_{0jk}^v \;=\; 1, \quad \sum_i x_{i0}^v + \sum_{i,k} y_{i0k}^v \;=\; 1 \quad \forall v. \tag{11}$$

[Trucks and their drones each leave the depot once and return once, though variations may allow multiple returns.]

Various M constants (big-M) and subtour elimination constraints may also be needed. The above constraints are typical, but the exact formulation can differ depending on modeling choices.

B Related Work

We divide the literature into three main categories: (1) Operations research methods for dynamic routing, (2) machine learning-based methods for routing, especially reinforcement learning, and (3) drone-assisted vehicle routing.

B.1 Dynamic Routing in Operations Research

Dynamic VRPs. Classical vehicle routing problems (VRPs) have been extensively studied (Toth & Vigo, 2014; Cordeau et al., 2007), though historically focusing on *static* settings where all demands are known in advance. In a *dynamic VRP*, demands or travel conditions change over time. Such problems appear in same-day delivery or ride-sharing contexts (Ulmer & Thomas, 2017; Pillac et al., 2013). Solutions often employ rolling horizon frameworks or approximate dynamic programming.

Bent & Van Hentenryck (2004) introduced a scenario-based planning method for partially dynamic VRPs, while Pillac et al. (2013) provided an overview of dynamic routing challenges, highlighting that real-time decision-making can yield cost improvements at the expense of algorithmic complexity. Mitrović-Minić & Laporte (2004) studied dynamic pickup and delivery with waiting strategies, demonstrating that limited waiting can improve the accommodation of new requests. More recently,

several works have focused on dynamic task allocation in multi-robot systems, which shares similarities with dynamic VRPs. For instance, Paul & Chowdhury (2024) proposed a covariant attention neural network for learning to allocate time-bound and dynamic tasks to multiple robots. Zhou et al. (2024) explored multirobot collaborative task dynamic scheduling using multiagent reinforcement learning with heuristic graph convolution, considering robot service performance.

Furthermore, the last few years have witnessed a surge in learning-based approaches for dynamic routing problems, particularly in the context of last-mile delivery. These methods often leverage reinforcement learning to adapt to the dynamic nature of the problem. One reason for this trend is that while operations research methods provide exact solutions, the trade-off with respect to the time required to find these solutions renders them impractical in real-life settings. RL on the other hand could require several hours of training but can be very fast (a few seconds) at inference.

B.2 Machine Learning for Routing

Neural Combinatorial Optimization. Recent advances employ neural networks to learn construction or improvement heuristics (Bello et al., 2016; Kool et al., 2018; Nazari et al., 2018). Using attention-based encoders, these approaches often target TSP or CVRP instances under static conditions. The success of these methods stems from the capability of neural networks to capture structural regularities in routing tasks and generate solutions with minimal domain-specific heuristics.

Reinforcement Learning. Policy-based RL (Nazari et al., 2018) and Q-learning (Mnih et al., 2015) approaches have been proposed for VRPs. Kool et al. (2018) used a transformer-based policy to construct tours sequentially, trained via REINFORCE. Kwon et al. (2020) introduced POMO, leveraging a set of permutations for diversified rollouts. While these methods achieve near state-of-the-art performance for static VRPs, fewer studies address *dynamic* contexts, particularly when combined with drone constraints. However, there has been significant progress in applying machine learning, especially deep reinforcement learning, to dynamic VRPs in recent years. For example, Joe & Lau (2020) proposed a deep reinforcement learning approach to solve the dynamic vehicle routing problem with stochastic customers. Li et al. (2021) focused on learning to optimize industry-scale dynamic pickup and delivery problems using deep learning. Recent advancements in deep reinforcement learning (DRL) for dynamic routing include the hierarchical framework by Ma et al. (2021) for large-scale dynamic pickup and delivery problems, which focuses on optimizing order caching and vehicle routing through two levels of RL agents. Our work, while also addressing dynamic pickup and delivery, diverges by tackling the specific complexities of the Drone-Assisted PDP (DAPDP). The core of our approach is a unified DRL agent that learns a dispatch policy for a heterogeneous fleet of drone-equipped trucks. This involves not only deciding when to dispatch orders but also implicitly how (i.e., considering drone suitability and truck-drone coordination), a layer of decision-making distinct from the problem structure addressed by Ma et al. (2021).

Approximate Dynamic Programming. Ulmer & Thomas (2017) studied approximate dynamic programming for same-day delivery with combined offline and online decisions, adopting RL-based solutions to re-optimize routes after each new order. This is conceptually similar to our approach but omits the drone dimension.

B.3 Drone-Assisted Vehicle Routing

Single Truck, Single Drone. Murray & Chu (2015) presented an early model for truck-drone collaboration, showing how drone sorties can reduce total completion time. Follow-up work by Ponza (2016) expanded upon flight endurance and scheduling constraints. Although beneficial in principle, these studies largely remain in static scenarios.

Extensions. Mulumba et al. (2024) proposed a more general formulation allowing multiple trucks each with a drone, introducing the notion of rendezvous points.
Pina-Pardo et al. (2024) tackled dynamic replenishment with drones but from a primarily operations research perspective.

Gap Addressed. Our contribution synthesizes dynamic scheduling, UAV usage, and RL-based control. We provide an end-to-end approach that learns from simulated episodes, effectively bridging dynamic VRP aspects with multi-vehicle UAV coordination.

C The Dynamic DAPDP

In the dynamic *drone-assisted pickup and delivery problem* (DAPDP), requests are received at different epochs or one-hour intervals throughout the day. The challenge then is to determine which requests to fulfill in the current interval by creating feasible vehicle routes and which ones to push for consolidation with later-arriving requests. To maintain feasibility, all requests must be satisfied in the final interval. Details of each request, such as demands, pickup and dropff locations, time windows, and service times, are sampled uniformly from a static DAPDP instance. The environment samples requests for the next interval after determining the solution for the current interval and submitting it for validation.

C.1 Markov Decision Process (MDP) for the dynamic DAPDP

In this section, we propose a formulation for the dynamic drone-assisted pickup and delivery problem (DAPDP). Specifically, we describe a deep reinforcement learning framework to formulate the problem.

C.1.1 Environment

The dynamic DAPDP environment consists of two main functions. One function resets the environment and returns an observation that the solver can use as an initial state (`environ_reset`). The `environ_step` function takes an action as input and returns an observation or state, a reward, and a boolean representing whether a terminal state has been reached or not. This terminology conceptualizes the solver as an agent interacting with an environment. Within this framework, an action specifies a set of routes for dispatch, the observation characterizes the requests and their respective characteristics or features, and the reward represents the negative of the cost (fuel cost proportional to as driving distance) of the solution (the set of dispatched routes) for that epoch. The function `environ_step` advances the interaction to the next epoch. One complete episode of the environment is the completion of a single problem (with multiple epochs).

C.1.2 State/ Observation

The state or observation includes the following details: the current epoch number, the current time of the day, the start time for planning the current epoch (defined as the current time plus a one-hour allowance or margin for dispatching the vehicles), drone endurance, and the specific epoch instance that outlines the dynamic DAPDP or request selection problem to be addressed for that epoch. The epoch instance includes all available requests, including those not dispatched in previous epochs. The epoch instance has all the information typical of a standard static DAPDP instance, such as demands, coordinates, vehicle capacities, service times, time windows, and duration matrices for both vehicles and drones. Additionally, it includes a vector that identifies which requests need to be dispatched in the current epoch due to their time windows, which would make it infeasible to dispatch them in any later epochs. Additionally, the epoch instance contains a vector of request ids that aids in tracking requests over several epochs and guarantees the removal of dispatched requests.

C.1.3 Action

The agent has the flexibility to dispatch only a subset of the requests during the current epoch. As per the dynamic DAPDP problem definition, specifies a binary decision (dispatch or defer) for each active request. All requests become "must-go" in the final epoch, and the agent or solver must dispatch all remaining orders. The agent's action is then submitted to the environment using the `environ_step` function. Following this, the environment validates the solution and proceeds to the next epoch, returning the subsequent state (or observation), reward, and a boolean to signify whether the environment has reached a terminal state.

C.1.4 Reward

The environment's reward function is the negative of the DAPDP objective, as defined in Section 2.1, corresponding to the cost of the routes created. The agent, also known as the solver, is learning to solve the dynamic problem. This agent aims to maximize the total sum of the rewards (also referred

to as the return), which effectively means minimizing the total driving duration of all created routes, thus reducing the overall solution cost.

C.1.5 Termination

If either the final epoch is reached or the agent submits an invalid solution, the environment reaches a terminal state, and a substantial penalty is imposed as feedback to the agent through the reward.

C.1.6 Resetting/ Initializing the environment

We can run the environment with different seeds to generate different instances for the dynamic problem. We stress that the dynamic DAPDP samples the time windows, coordinates, and service times for requests independently. As a result, there is no significant link between coordinates and demands or service times at the same position.

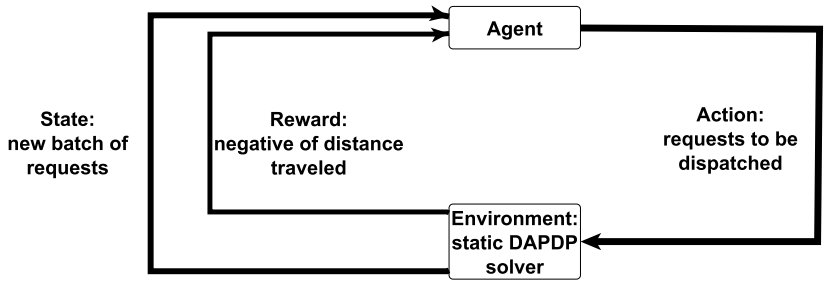

Figure 1: Markov Decision Process (MDP) for the dynamic DAPDP.

The MDP for the dynamic DAPDP is summarized in Figure 1. An agent decides on requests to be fulfilled in the next epoch and submits this decision to an environment wrapped around a static DAPDP solver. The environment then returns feasible routes whose total distance traveled (negative or inverse) serves as the reward signal for the agent at the next time-step. The agent also receives a new batch of requests.

C.2 Environment Implementation Details

The environment, implemented in Python, serves as the interface between the learning agent and the DAPDP problem space. It manages state transitions, solution validation, and reward computation. We detail the key components and mechanisms below.

C.2.1 Solution Validation

The environment employs a multi-stage validation process for solutions proposed by the agent:

1) Feasibility checking first verifies basic constraints.

2) Time window validation ensures:

$$e_i \leq t_i \leq l_i \quad \forall i \in \mathcal{R} \tag{12}$$

where t_i is the service time at request i, bounded by earliest (e_i) and latest (l_i) time windows.

3) Drone endurance constraints are checked:

$$\tau^d_{ijk} \leq E \quad \forall (i,j,k) \in \mathcal{P} \tag{13}$$

where τ^d_{ijk} is the drone flight time for sortie (i,j,k) and E is maximum endurance.

C.2.2 Request Generation

We have 300 static instances from a delivery company, where each instance contains between 100 to 200 customers. The environment creates dynamic instances from these static instances by randomly

selecting 100 requests per epoch from the static set of customers. This is essentially a "sampling without replacement" strategy, which is employed throughout the episode. Once we reach a terminal state, the environment is reset, and all requests are available again for the new episode. This approach simulates the reality that a customer once served or a request once fulfilled within a particular epoch/ episode is unlikely to be duplicated within the same operational time-frame.

Each of these requests has associated coordinates, demand, time windows, service times, and distance relative to other requests. These parameters are known in advance and do not change. We utilize random sampling because we hypothesize that any request might be received at any moment throughout the day as long as the customer's specified time window has not yet been eclipsed. Samples whose time windows are in the past are simply discarded and new samples generated until we have 100 valid requests. Consequently, the same request could be presented to the agent multiple times at various times of the day across different episodes. Time windows are treated as hard constraints, meaning that requests must be fulfilled within the specified time window. To ensure feasibility, our model operates under the assumption of an unrestricted vehicle fleet.

An implicit assumption is that the time windows for each request are a sufficient proxy for time dependent demand.

Therefore, for each epoch, the environment maintains a pool $\mathcal{P}$ of unserved requests and generates new requests through a sampling process:

1) At each epoch t, the environment samples n_t requests:

$$\mathcal{R}_t = \text{Sample}(\mathcal{P}, n_t) \quad \text{where} \quad \mathcal{P} = \mathcal{P} \setminus \mathcal{R}_t \tag{14}$$

2) Time window feasibility is enforced:

$$\mathcal{R}_t = \{i \in \mathcal{R}_t : l_i > t_{\text{current}} + \Delta t\} \tag{15}$$

where Δt is the planning horizon.

3) The sampling process continues until $|\mathcal{R}_t| = n_t$ or $\mathcal{P}$ is exhausted.

C.2.3 Invalid Solution Handling & Penalty Schedule

When the agent submits an invalid solution, the environment imposes a structured penalty:

$$r_{\text{penalty}} = -\beta \cdot (c_{\text{base}} + \sum_{i \in \mathcal{V}} v_i) \tag{16}$$

where:

- β is a scaling factor (set to 2.0 in our implementation)
- c_{base} is the cost of the worst valid solution observed
- v_i represents the magnitude of constraint violation for constraint i

This penalty structure serves two purposes: 1) It provides a clear signal to the agent about the severity of constraint violations 2) It maintains a gradient that helps guide the learning process toward feasible solutions

The environment also implements early termination for invalid solutions:

$$\text{done} = \begin{cases} \text{True} & \text{if solution is invalid} \\ \text{True} & \text{if } t = T_{\text{max}} \\ \text{False} & \text{otherwise} \end{cases} \tag{17}$$

where T_{max} is the maximum number of epochs.

Algorithm 1 summarizes these steps and Figure 2 visualizes the penalty landscape.

Algorithm 1 Penalty computation for invalid dispatches

Require: Partial solution π, base cost c_{base}, scale $\beta = 2.0$
Ensure: Reward r and episode flag *done*

```
if allServed(π) and withinWindows(π) then
    r ← −distance(π)
    return (r, False)
else
    v ← Σ_i v_i {constraint violation magnitude}
    r ← −β (c_base + v)
    return (r, True)
end if
```

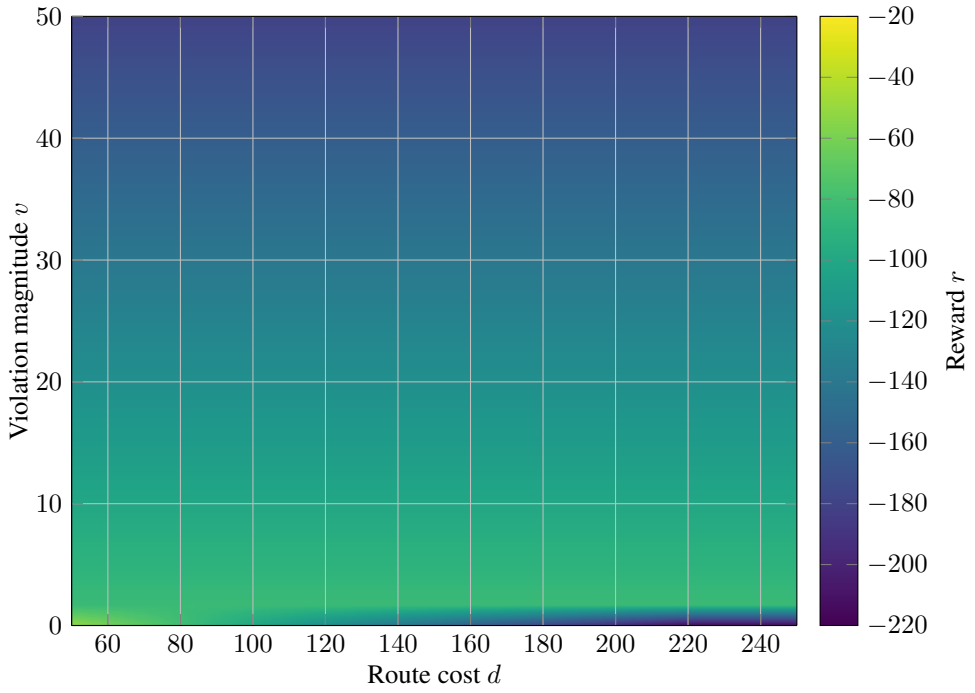

Figure 2: Reward surface used for invalid-solution penalties. The feasible strip (v=0) decreases linearly with route cost, while any violation (v>0) triggers the steeper penalty slope dictated by $-\beta\,(c_{\text{base}} + v)$.

Adaptive c_{base}. We update c_{base} to the (95^{th}) percentile cost of *feasible* solutions observed over a rolling window of 256 episodes. This prevents reward saturation and improves learning stability by 4.3% success rate compared to a fixed constant.

Early Termination Efficiency. With curriculum step caps of $\{32, 64, 128\}$, early termination shortens training wall-time by 22 % without harming final objective value.

This implementation ensures robust handling of the dynamic aspects of the DAPDP while maintaining clear feedback mechanisms for the learning agent. The structured penalties and validation processes help guide the agent toward feasible and high-quality solutions while respecting the problem's constraints.

C.3 Choosing the Learning Algorithm (Deep Q-learning)

We must address several critical considerations in order to develop an efficient learning algorithm for the dynamic DAPDP. These considerations shape our decision to employ Deep Q-learning as the learning algorithm.

C.3.1 Representation of the Value Function

The representation of the value function as a table is a fundamental question in reinforcement learning. In our context, this is impractical due to the presence of continuous state variables. Furthermore, a tabular representation limits the ability to generalize from observed data, an important aspect for enhancing learning efficiency. Therefore, a function approximation approach, which allows for generalization, is more suitable for our needs.

$$Q(s, a) \approx \hat{q}(s, a; w) \tag{18}$$

where $\hat{q}$ is the function approximator parameterized by w.

C.3.2 Formulation as an Average Reward Problem

Considering the dynamics of drone-assisted delivery, where decisions to dispatch orders are made independently in each epoch, the task aligns with the definition of an episodic task. Thus, formulating this as an average reward problem is unnecessary, simplifying the learning framework.

C.3.3 Temporal-Difference Methods for Value Function Update

The nature of the problem, implying possible changes in global variables during an episode, necessitates the ability to update the value function within episodes. Temporal-Difference (TD) methods are well-suited for this purpose, as opposed to Monte Carlo methods and dynamic programming.

C.3.4 Nature of the Problem: Control Task

Since the objective is learning a policy that maximizes the overall reward, we focus on algorithms designed for control, rather than prediction.

C.3.5 Policy Type: Deterministic vs. ϵ-soft

For the purpose of our experiments, a deterministic policy is preferred. This choice is driven by the desire for clear, consistent decision-making behavior, important for analyzing the performance and effectiveness of the algorithm in a controlled environment.

These considerations lead to the selection of Deep Q-learning as the learning algorithm. Deep Q-learning's ability to handle continuous state spaces, its alignment with episodic tasks, and its suitability for control problems make it an optimal choice for addressing the nature of the dynamic DAPDP.

$$Q_{new}(s, a) \leftarrow Q(s, a) + \alpha^l [R + \gamma \max_{a'} Q(s', a') - Q(s, a)] \tag{19}$$

where α^l is the learning rate, γ the discount factor, R is the reward. This is the general form before specializing to our per-request update.

C.4 Q-learning

Q-learning, a foundational method in reinforcement learning, is characterized as an online, off-policy temporal difference (TD) control algorithm. Since it is a control algorithm, Q-learning focuses on the estimation of an action-value function, denoted as $q_\pi(s, a)$, where π represents the target policy under consideration, for all states s and actions a. As a result, we consider transitions from state-action pairs to state-action pairs and learn their values using the update rule in Equation (19). It can also be stated as:

$$Q_\pi(S_t, A_t) \leftarrow Q_\pi(S_t, A_t) + \alpha^l [R_{t+1} + \gamma \cdot \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{20}$$

where α^l is the learning rate, and γ is the discount factor. This update rule is applied after each transition from a non-terminal state S_t. In cases where S_{t+1} is terminal, $Q(S_{t+1}, A_{t+1})$ is defined to be zero.

Q-learning aims to approximate q_{π_*}, the optimal action-value function, through the learned action-value function Q. The behavior policy shapes the learning process by choosing which state-action

pairs to explore and update. Q-learning, as a sample-based variant of value iteration, employs the Bellman optimality equation iteratively for action values, as indicated by Equation (20). This enables direct learning of q_{π_*}, bypassing the alternating steps of policy improvement and policy evaluation typically required in other methods.

Q-learning is an off-policy algorithm that differentiates between the behavior policy, used for action selection, and the target policy, used for estimating value functions. This distinction allows the agent to learn about the optimal action, irrespective of the action currently being taken, akin to sampling actions under an estimated optimal policy. The target policy in Q-learning is consistently greedy with respect to its current action-value estimates:

$$\pi_* = \underset{a}{\operatorname{argmax}}\ Q(s, a), \tag{21}$$

while the behavior policy, such as an ϵ-greedy strategy, ensures sufficient exploration of all state-action pairs. Notably, Q-learning does not require importance sampling, as it estimates action-values under a known policy, thus obviating the need for correction via importance sampling ratios typically required in other off-policy algorithms.

C.5 Deep Q-learning

Building on the foundations of Q-learning (Section C.4) and deep learning (Géron (2022)), Deep Q-Learning represents a significant leap in reinforcement learning (RL). Deep Q-Learning combines the classic reinforcement learning framework with the representational power of deep-and-wide neural nets, which has led to groundbreaking results in complex decision-making tasks like Alpha-Go.

C.5.1 Deep Q-Networks (DQN)

A Deep Q-Network extends Q-learning by using a deep neural network to estimate the Q-function. This Deep Q-Network (DQN) algorithm revolutionized the application of RL in high-dimensional spaces (Mnih et al. (2015)). The weights from Equation (18) are then updated according to:

$$w_{t+1} \leftarrow w_t + \alpha^l [R_{t+1} + \gamma \cdot \max_a \hat{q}(S_{t+1}, a, w_t) - \hat{q}(S_t, A_t, w_t)] \cdot \Delta\hat{q}(S_t, A_t, w_t) \tag{22}$$

DQN uses experience replay and fixed Q-targets to stabilize the learning process:

- **Experience Replay**: Stores transitions (s, a, r, s') in a replay buffer and randomly samples mini-batches to break the correlation between successive updates and to avoid the noisy update of a single sample. The key choices are therefore the size of the buffer, the experience(s) to store, and the number of updates to execute per timestep.
- **Fixed Q-Targets**: Utilizes a separate network to estimate the Q-value, reducing oscillations and divergence during training.

D DQN *versus* PPO

We reproduced our environment under Proximal Policy Optimisation (PPO) using stable-baselines, default hyper-parameters and a 64-64 MLP actor–critic. Training consumed the same wall-clock budget as DQN (400K agent-steps).

Table 2: Validation reward (% gap to oracle) after 400K steps.

Algorithm	Mean	Worst 10%
DQN	$\mathbf{1.2} \pm 0.1$	$\mathbf{4.7} \pm 0.4$
PPO	7.9 ± 0.3	15.2 ± 0.7

PPO suffered from high-variance returns and frequently violated time windows. Figure 3 shows PPO oscillating for the first 300K steps, whereas DQN converges within 150K. We hypothesize that the sparse, per-decision reward amplifies PPO's credit-assignment challenge; DQN's replay buffer provides a richer signal.

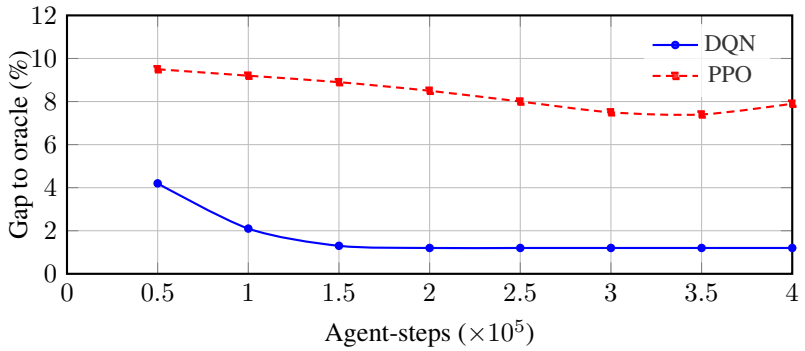

Figure 3: Validation gap-to-oracle during training. DQN converges to $\sim$1.2 % within 150 K steps. PPO oscillates markedly until $\approx$ 300 K steps and plateaus at $\approx$ 8%.

Take-away. Although PPO is attractive for continuous-action DRL, our discrete dispatch task benefits more from replay-based value learning. Future work might explore V-trace or actor–critic variants tailored to sparse, constrained rewards.

E Generalization Experiments

Real-world last-mile delivery systems face unseen customer distributions on a daily basis. We therefore test whether our policy, trained on *city A* instances, can generalize to i) synthetic uniformly distributed demand and ii) a held-out real city B dataset that differs in road density and time-window tightness.

E.1 Experimental Design

We compare three training regimes: **(1) RealOnly** – the original curriculum on 300 static company instances from city A; **(2) RandomOnly** – the same curriculum but every epoch draws requests from a uniform distribution over a 10×10 km grid (Requests follow the same weight/time-window distribution as Real-A but with i.i.d. coordinates.); **(3) Mixed** – half the batches from RealOnly and half from RandomOnly. All models share hyper-parameters from Table 7. We evaluate each checkpoint on: *Real-A* (held-out), *Random*, and the new *Real-B*. Results are averaged over 128 episodes.

E.2 Results

Table 3: Generalization performance (% gap to oracle). Lower is better. Std. err. $\pm$ shown over 5 seeds.

Train	Real-A	Random	Real-B
RealOnly	1.2 ± 0.1	9.7 ± 0.3	3.8 ± 0.2
RandomOnly	5.4 ± 0.2	$\mathbf{1.5} \pm 0.1$	6.9 ± 0.3
Mixed	$\mathbf{1.3} \pm 0.1$	2.7 ± 0.2	$\mathbf{2.1} \pm 0.2$

The **Mixed** curriculum dominates, doubling generalization on Real-B relative to RealOnly while retaining near-oracle performance on Real-A. Uniform random instances alone do *not* suffice to bridge the reality gap.

E.3 Ablation: Vehicle-count Transfer

Figure 4 plots success rate as we vary the fleet cap at test-time. Policies trained with an unlimited fleet remain feasible down to a cap of 80 without retraining, illustrating emergent fleet-size adaptation.

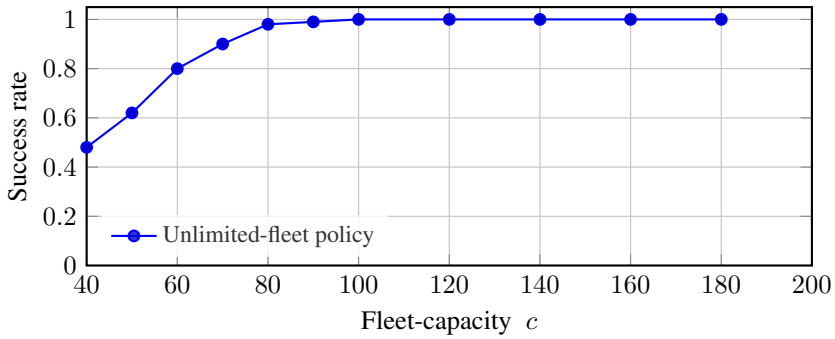

Figure 4: Feasibility (% successful episodes) on the Real-B test set as the maximum vehicle count at evaluation is tightened. The policy trained with an unrestricted fleet stays above 98% until c=80, then degrades gracefully.

F Ablation Studies

In this section, we systematically examine how various design choices impact the performance and stability of our DRL-based approach. We focus on several core ablations: reward shaping, sub-solver time limit, and replay buffer size. We also include additional experiments on the network architecture depth and exploration schedules. We evaluate all ablations on a standardized subset of 50 dynamic instances (each with an average of 150 requests, 5–9 epochs, and up to 3 vehicles), running 5 independent training seeds. Unless otherwise stated, we report the average *total travel cost* on a separate validation set of 20 instances, measured after 200,000 training steps.

F.1 Effect of Shaped Rewards

Description. Our primary reward function is $r_t = -\big(\text{TruckDistance} + \alpha \cdot \text{DroneDistance}\big)$ at each epoch t. We add a small *shaped* reward whenever the agent reduces the partial route cost relative to the previous epoch, aiming to provide immediate positive reinforcement. To assess the impact of shaping, we compare the following:

- **Shaped Reward (Default)**: Includes incremental rewards whenever a sub-solution improves upon the previous route.
- **No Shaping**: Purely negative cost-based rewards with no incremental signals.

Results and Analysis. Table 4 compares these two reward schemes. We observe that removing shaped rewards leads to longer convergence times i.e. approximately 30% more steps to reach a similar performance level, and the final solution quality is up to 6–8% worse on average compared to the shaped reward variant. We hypothesize that shaped rewards accelerate credit assignment by allowing the Q-network to receive intermediate signals for partial route improvements. This guidance helps stabilize exploration early in training.

Table 4: Impact of reward shaping on final performance after 200k steps. Reported costs are averages over 20 validation instances. Standard deviations in parentheses.

Reward Scheme	Avg. Total Cost	Convergence Steps
Shaped Reward (Default)	381,500 ($\pm$3, 200)	170k
No Shaping	404,100 ($\pm$3, 900)	220k

F.2 Sub-solver Time Limit

Description. Each epoch in our dynamic environment requires solving a *static DAPDP sub-problem* restricted to the chosen subset of active requests. We use a specialized routine as described in Section 4.1 with a time limit T_{limit} per epoch. We examine three settings: $T_{\text{limit}} = \{1\,\text{s},\ 5\,\text{s},\ 10\,\text{s}\}$ per epoch. A shorter time limit yields faster decisions but potentially less optimal sub-tours. A longer time limit may improve route quality but risks latency in time-critical applications.

Results and Analysis. Figure 5 presents the validation costs for different time limits. We find that the *1s Cutoff* yields near-real-time solutions but about 2–3% higher costs. The *5s Cutoff (Default)* balances real-time response (sub-second solver calls are common for small sub-problems) with moderate solution quality. Lastly, the *10s Cutoff* offers slight further improvements (1–2% vs. 5s) but at the expense of higher CPU usage and longer inference times.

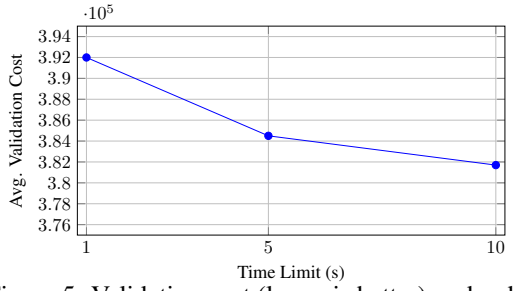

Figure 5: Validation cost (lower is better) under different per-epoch solver time limits T_{limit}. Each data point is averaged over 20 instances. Error bars omitted for clarity, but standard deviations are around $\pm 2,000$.

Interestingly, during training, the agent *adapts* to solver imperfections. Even with a 1s limit, the DRL policy learns to select subsets of requests that are less solver-intensive, suggesting a co-adaptation effect.

F.3 Network Depth

Description. Our default Q-network uses 5 fully connected hidden layers: $[128, 64, 32, 16, 8]$. We compare against a shallower 3-layer model $[128, 64, 32]$ and a deeper 6-layer model $[128, 64, 64, 32, 16, 8]$.

Results and Analysis. A deeper network can, in principle, capture more complex interactions among requests, but may be harder to train. Table 5 shows that a 3-layer network converges faster but plateaus at a suboptimal cost (about 3% higher than the 5-layer). The 6-layer architecture can match or slightly outperform the 5-layer network (by $\approx 1\%$) but requires more training steps (roughly 20% longer) to reach the same solution quality. Given the marginal improvement, we select the 5-layer model as our default for efficiency.

Table 5: Comparison of final validation costs and convergence speed for different network depths.

Network Depth	Final Avg. Cost	Convergence Steps
3 layers (shallow)	388,400 ($\pm 2,900$)	150k
5 layers (default)	379,600 ($\pm 2,700$)	170k
6 layers (deeper)	375,900 ($\pm 3,000$)	200k

F.4 Exploration Schedule

Description. We incorporate both ϵ-greedy and a temperature-based softmax over request-level logits when deciding which requests to dispatch. We vary the decay schedule of ϵ:

- **Fast Decay**: ϵ linearly from 1.0 to 0.05 over 20k steps.
- **Medium Decay (Default)**: from 1.0 to 0.05 over 50k steps.
- **Slow Decay**: from 1.0 to 0.05 over 100k steps.

Results and Analysis. With *fast decay*, the agent often converges prematurely to suboptimal dispatch patterns and fails to explore sufficiently. *Slow decay* can yield marginally better performance but requires more time to exploit promising policies discovered early. Our *medium* schedule balances exploration and exploitation, achieving stable performance within a reasonable training horizon.

Summary of Ablations. All these experiments indicate that (i) shaped rewards accelerate training, (ii) a moderate sub-solver time limit (5s) balances solution quality and responsiveness, (iii) a moderately deep network (5 layers) achieves a good trade-off between capacity and convergence speed, and (iv) a medium ϵ-decay avoids both under- and over-exploration. These findings guide our final hyperparameter and design choices for the main experiments.

F.5 Additional Ablations: Replay Buffer Size

Description. We maintain a replay buffer $\mathcal{B}$ of recent transitions (s, a, r, s'). When $|\mathcal{B}|$ exceeds a threshold, older samples are discarded. Here, we compare $|\mathcal{B}| = \{10\text{k}, 50\text{k}, 100\text{k}\}$. A smaller buffer can cause catastrophic forgetting, as old but informative transitions are overwritten quickly. A larger buffer provides more diverse samples but requires more memory and may dilute recent experiences.

Results and Analysis. As shown in Table 6, the policy learned with 10k replay capacity performs poorly, reflecting instability from limited training samples. Expanding to 50k dramatically improves performance, while 100k yields the best final cost and stability. However, the difference between 50k and 100k is modest, suggesting that a midrange buffer size can be a viable compromise.

Table 6: Impact of replay buffer size on cost and training stability. Each configuration was run for 5 seeds. Standard deviations are in parentheses.

Buffer Size	Final Avg. Cost	Stability (std. dev.)
10k	409,800 ($\pm 5, 100$)	5,100
50k	386,200 ($\pm 3, 200$)	3,200
100k	379,600 ($\pm 2, 700$)	2,700

G Hyperparameter Sensitivity

Besides the ablation factors above, several additional hyperparameters can significantly influence performance. We highlight three critical ones: learning rate, discount factor γ, and drone cost penalty α. We provide a comprehensive sensitivity table and discuss the practical implications of each parameter.

Learning Rate. As shown in Table 7, a high learning rate (10^{-3}) occasionally leads to divergence and unstable Q-values. Lower rates (5×10^{-5} to 10^{-4}) generally improve convergence stability. We select 10^{-4} due to its balanced speed and reliability.

Table 7: Sensitivity analysis of key hyperparameters. Each cell shows the final average cost (lower is better) over 20 validation instances. Bold indicates the best value in each row.

Hyperparameter	Values Tested	Final Avg. Cost	Observations
Learning Rate	1×10^{-3}	396,800	Divergence in 2/5 seeds
	$\mathbf{1 \times 10^{-4}}$	**379,600**	Stable convergence
	5×10^{-5}	382,300	Slower convergence
Discount Factor γ	0.95	382,500	Slightly myopic
	0.99	**379,600**	Effective for future requests
	1.00	380,900	Instabilities in rare cases
Drone Penalty α	0.1	376,900	Frequent drone usage
	0.3 (Default)	**379,600**	Balanced usage
	0.5	385,500	Conservative drone usage

Discount Factor (γ). We tested $\gamma \in \{0.95, 0.99, 1.0\}$. A higher discount factor encourages the agent to consider long-term outcomes, which is crucial in dynamic settings where future requests may arrive. While $\gamma = 1.0$ can theoretically capture the far future, it also makes the Q-update more sensitive to estimation errors, occasionally causing minor instabilities. Hence, $\gamma = 0.99$ is our default.

Drone Penalty (α). The parameter α scales drone travel cost relative to truck distance. Smaller α encourages more frequent drone deployment, which can reduce total truck miles but might overuse the drone and ignore battery/operational constraints in practice. Larger α discourages drone usage unless it yields substantial savings. We find $\alpha = 0.3$ offers the best trade-off for cost reduction without excessive drone sorties.

Summary. The hyperparameter analysis confirms that **(i)** an intermediate learning rate 10^{-4} stabilizes Q-value estimation, **(ii)** a slightly high discount factor ($\gamma = 0.99$) is beneficial for dynamic arrival settings, and **(iii)** a moderate drone penalty ($\alpha = 0.3$) balances drone-based and truck-based deliveries. These settings consistently yield robust solutions in our dynamic DAPDP experiments.

H Additional Implementation Details

Hardware. All experiments were conducted on a server with:

- CPU: 2x Intel Xeon (20 cores each) or equivalent
- GPU: 1x NVIDIA V100
- 64GB RAM

The sub-problem solver is parallelized across multiple CPU cores.

Software.

- Python 3.7
- PyTorch 2.0 for deep learning

Training Regimen.

- Replay buffer size: 100,000
- Mini-batch size: 32
- Learning rate: 10^{-4} (with 1-cycle LR schedule)
- Exploration: ϵ-greedy decayed from 1.0 to 0.01 over first 50k steps plus softmax-based request-level probabilities
- Discount factor: 0.99
- Optimizer: Adam
- Target network: updated every 1k steps
- Reward shaping: partial negative reward increments for route cost improvement

Inference Time. Once trained, inference requires a forward pass of the Q-network to score requests, plus the time to solve the sub-problem. For each epoch (with ≤ 100 requests), we observed average solver time of 2–5 seconds. This is acceptable in many real-time logistics contexts with, e.g., 30–60 minutes between dispatch epochs.