# Dynamic courier capacity acquisition in rapid delivery systems: a deep Q-learning approach

Ramon Auad[*1, 2], Alan Erera[†1], and Martin Savelsbergh[‡1]

[1]School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta GA

[2]Departamento de Ingeniería Industrial, Universidad Católica del Norte, Antofagasta, Chile

**Abstract**

With the recent boom of the gig economy, urban delivery systems have experienced substantial demand growth. In such systems, orders are delivered to customers from local distribution points respecting a delivery time promise. An important example is a restaurant meal delivery system, where delivery times are expected to be minutes after an order is placed. The system serves orders by making use of couriers that continuously perform pickups and deliveries. Operating such a rapid delivery system is very challenging, primarily due to the high service expectations and the considerable uncertainty in both demand and delivery capacity. Delivery providers typically plan courier shifts for an operating period based on a demand forecast. However, because of the high demand volatility it may at times during the operating period be necessary to adjust and dynamically add couriers. We study the problem of dynamically adding courier capacity in a rapid delivery system and propose a deep reinforcement learning approach to obtain a policy that balances the cost of adding couriers and the cost of service quality degradation because of insufficient delivery capacity. Specifically, we seek to ensure that a high fraction of orders is delivered on time with a small number of courier hours. A computational study in the meal delivery space shows that a learned policy outperforms policies representing current practice and demonstrates the potential of deep learning for solving operational problems in highly stochastic logistic settings.

**Keywords**: Logistics; Rapid delivery; Capacity management; Last-mile delivery; Reinforcement learning; Deep Q-learning

[*]Corresponding author, ramon.auad@gmail.com

[†]alan.erera@isye.gatech.edu

[‡]martin.savelsbergh@isye.gatech.edu

# 1 Introduction

In recent years, the explosive development of technology and e-commerce have substantially impacted retail and logistics. Primarily due to Amazon and its early innovations in last-mile delivery (Hausman et al., 2014), the expectation and behavior of customers about delivery notoriously shifted toward faster and more effective delivery (Mabe, 2017), ultimately serving as the foundation for today's rapid delivery platforms. E-grocery sales (*e.g.*, Walmart, Target, Whole Foods) reached $125 billions in 2020 and are expected to account for over 20% of the US total grocery sales by 2025 (Mercatus, 2017). Similarly, demand in the meal delivery industry (*e.g.*, Doordash, Grubhub, Uber Eats) during the last decade has been growing steadily with the US food delivery market being worth $11 billions in 2015 and predicted to reach a value of $210 billion in 2022 (Morgan Stanley Research, 2017). Furthermore, the worldwide online food delivery market revenue reached $82.7 billion in 2018 and is expected to double by 2024 (Statista Report, 2019). Moreover, the COVID-19 pandemic has prompted people to increase their use of delivery services, causing sales and prospective growth in the industry to skyrocket (Alfonso et al., 2021; Mercatus, 2017; Popper, 2020; U.S. Food and Drug Administration, 2020).

In rapid delivery systems, customers dynamically place orders through a digital platform with one of the available vendors and receive a delivery time promise. The platform coordinates the delivery of the orders using a set of (active) couriers that repeatedly travel between facilities and customer locations to deliver orders.

We focus our work on rapid delivery systems, such as meal delivery systems, where the delivery time promise is typically less than an hour after the order placement. Many aspects of meal delivery are highly stochastic, making it a source of the most challenging logistic problems. On the demand side, information about orders is only revealed at the moment they are placed and each order can only be dispatched after being prepared. Furthermore, it is critical to deliver orders as soon as possible to minimize the loss of food freshness. On the supply or delivery capacity side, the couriers – mostly independent contractors – are available only during certain period of times, or *blocks*, which are scheduled in advance based on a forecast of order placements; such scheduled couriers will also be referred to as *base couriers*. However, since the timing and volume of the demand for a day is difficult to predict (*e.g.*, due to changes in weather and traffic conditions), delivery providers may need to adjust delivery capacity in real-time by adding additional

couriers to the available pool. We will refer to such added delivery personnel as *on-demand couriers*. For an in-depth discussion on the challenges and opportunities associated with the use of crowdsourced delivery capacity, see Savelsbergh and Ulmer (2022).

This paper focuses on the problem of dynamically adjusting the delivery capacity of a meal delivery system by adding on-demand couriers in response to some demand and supply signals. We view and model this problem as a sequence of *real-time* opportunities to decide whether or not to enlarge the pool of delivery resources, *i.e.*, exploiting the option to add on-demand couriers for short shifts of varying lengths. Due to the inherent uncertainty in this setting, evaluating the impact of such a decision can be difficult. In practice, delivery companies typically seek to adjust their delivery capacity based on a relatively simple workload metric. One such metric used in practice is the *order-per-courier ratio* (OPC), which divides the number of active orders by the number of on-duty couriers at a given time. Although simple to compute, making a capacity decision solely on OPC may ignore additional available information about current and future system operations that, if leveraged, could lead to significantly better solutions. Therefore, we propose and explore approaches for dynamically adjusting delivery capacity using higher-dimensional system state information that utilize deep Q-learning, a reinforcement learning (RL) technique that approximates the value of state-action pairs using a neural network; in this context, such an approach may typically be referred to as a deep Q-network (DQN). This methodology is well suited for the studied environment; the DQN can be trained offline and then queried efficiently to support real-time decision-making, and a well-trained DQN can factor in the future impact of current actions as well as system state features such as time of the day, order placement rate, number of available couriers, and order delivery promises. These characteristics make this type of methodology attractive for real-time decision-making in complex logistics systems.

The main contributions of this research can be summarized as follows:

- We present, to the best of our knowledge, the first study looking at dynamic fleet sizing in rapid delivery operations, and one of the first to use deep Q-learning for a dynamic transportation problem;
- We propose a deep RL framework to configure a policy to decide whether or not to add delivery capacity, and, if so, what type of delivery capacity, in an environment with uncertain, highly fluctuating demand considering order volume, order urgency, and active delivery capacity;
- We demonstrate the effectiveness of the devised policy by performing a series of experiments in

which we compare the policy's performance against the performance of policies representing current practice; and

- We analyze the sensitivity of the performance of learned policy to algorithmic configuration decisions and context-specific settings.

The remainder of the paper is organized as follows. Section 2 provides a brief survey of the related literature. Section 3 defines the problem and introduces the RL-based solution approach. Section 4 provides empirical evidence of the effectiveness of the solution approach. Section 5 summarizes the work and suggests future research directions.

## 2 Literature review

Rapid delivery problems can be classified as dynamic vehicle routing problems as routing decisions are made as new orders are placed. The papers by Pillac et al. (2013) and Psaraftis et al. (2016) provide excellent reviews of this class of problems. More precisely, meal delivery operations fall under the scope of dynamic pickup and delivery problems (dPDP), where couriers are dynamically assigned to retrieve orders from different vendors and drop them off at customer locations. The related literature in this area is vast, and a thorough survey on dPDP is provided by Berbeglia et al. (2010).

The existing work on dPDP covers a wide range of applications that include general same-day delivery (Arslan et al., 2019; Dayarian and Savelsbergh, 2020; Klapp et al., 2018; Ulmer et al., 2019; Ulmer and Streng, 2019; Voccia et al., 2019), ride-sharing (Agatz et al., 2012; Chen et al., 2020; Wang et al., 2018), and meal delivery (Auad et al., 2021; Reyes et al., 2018; Ulmer et al., 2021; Yildiz and Savelsbergh, 2019). However, most of that research focuses on routing decisions assuming a given set of courier blocks (*i.e.*, known delivery capacity) for the entire operating period. Auad et al. (2022) and Auad and Van Hentenryck (2021) study versions of the fleet-sizing problem in the contexts of meal delivery and ride-sharing, respectively, although in settings where information about order arrivals is known in advance and these papers do not consider dynamically expanding delivery capacity. In contrast, here we specifically focus on dynamically expanding delivery capacity based on observed order arrivals, *i.e.*, augmenting the base courier fleet when demand is higher than expected, a problem that to date has not been explored in dynamic delivery settings.

A related line of research can be found in the vehicle scheduling literature, which studies problems in which

one seeks an optimal number of vehicles to serve a set of requests. A comprehensive survey on this line of research can be found in Bunte and Kliewer (2009). The most basic setting of the vehicle scheduling problem, the deterministic single-depot variant, was first solved by Saha (1970). This single-depot variant can by solved in polynomial time, whereas the multi-depot extension is NP-hard (Bertossi et al., 1987). Huisman et al. (2004) considers a setting where travel times are not deterministic, potentially causing delays in the start time of serving requests; the authors propose to solve a sequence of optimization problems, each considering multiple scenarios for future travel times. Other authors further extend this setting by considering other potential disruptions (*e.g.*, vehicle breakdowns and traffic congestion) and evaluate the benefit of corrective actions that modify the schedules of the vehicles (Guedes and Borenstein, 2018; Uçar et al., 2017). However, none of this research considers an environment in which requests arrive in real-time and information about a request is revealed only when it is placed, as is the case in the environment studied in this chapter.

Our methodological approach is based on deep Q-learning, a reinforcement learning algorithm first proposed by Mnih et al. (2013). Adoption of deep learning and Q-learning techniques in the field of transportation has recently been on the rise, but there are significant opportunities for further research in the use of these methods. Qin et al. (2020) address the problem of trip-driver assignments in ride-hailing contexts. They improve the traditional linear assignment model by refining the matching weights using CVNet, a driver-decentralized deep Q-network (DQN) that estimates the long-term value of each feasible assignment. The authors also describe the deployment process of the algorithm for the ride-hailing company DiDi Chuxing and report significant benefits in multiple business metrics when compared to the traditional linear optimization model. Chen et al. (2019) use deep-Q learning to solve a single depot dynamic delivery routing problem using a fleet comprising vehicles and drones. A DQN is employed to determine whether or not an order is served and if so, whether it is served by a vehicle or a drone. The authors make routing decisions using heuristic methods. Lin et al. (2018) explore the effectiveness of DQN and actor-critic algorithms in learning policies for fleet management, via courier repositioning in online ride-hailing operations. The authors make use of contextual information to explicitly incorporate coordination between numerous drivers in the learning process and explore the effect of multiple modeling and algorithmic choices.

The use of machine learning methodologies in logistics is still in an early stage of development, but may offer great potential for improving scalability and for solving more complex problems (Capalbo et al., 2021;

Woschank et al., 2020), and the research results we present in this chapter offer another example of this potential.

# 3 Methodology

## 3.1 Problem description

Let $\mathcal{H} = [0, H]$ be an operating period for a rapid delivery system (and $H > 0$ the horizon), typically representing a day of operations. At different times during the operating period, delivery orders are placed at one of $N_{pickup}$ pickup locations. For each order $o$, let $t_o$ be the time when it is placed. Information about an order $o$ becomes known only at $t_o$, *i.e.*, its pickup and dropoff locations $p_o$ and $d_o$, respectively, and its ready time $e_o$ at the pickup location. When an order is placed, the customer placing the order receives a promised delivery time $m_o$; if the order cannot be delivered by this promised time, the order is assumed lost. No orders are placed after time $H_0 < H$.

Orders are continuously assigned to on-duty couriers to be delivered at their dropoff location. Each courier has a start time $t_{start} \in \mathcal{H}$ and a number of working hours $c \in \mathcal{C}$, where $\mathcal{C}$ is the set of working period lengths for couriers. Orders can be assigned for delivery to a courier only at or after $t_{start}$ and at or before $\min\{H, t_{start} + 60c\}$; the actual delivery of an assigned order can take place after $t_{start} + 60c$. In the sequel, we refer to couriers that work for $c \in \mathcal{C}$ hours as *c-hour couriers*. At any time $t \in \mathcal{H}$, unassigned orders are sequentially assigned to available couriers in a greedy manner based on the remaining time until the order becomes lost, i.e., giving priority to orders closer to their promised delivery time. A courier can only be assigned to deliver a given order $o$ if (i) the resulting order *pickup* time is no later than the end of the courier working period (a courier can complete its last delivery after the end of its working period); and (ii) the resulting order *delivery* time is no later than the promised delivery time, $m_o$. Consequently, an order $o$ becomes lost if it is still unassigned at its latest pickup time, *i.e.*, the pickup time that results in the order being delivered exactly at its due time, $m_o$.

If a courier $q$ arrives at $p_o$ at time $t$ to pick up order $o$, we assume $q$ picks it up at a time $t_{pickup} = \max\{r_o, t + s_p\}$, where $s_p$ is the time it takes $q$ to walk from his vehicle to the pickup location; and we further assume that $q$ starts driving towards the dropoff location $d_o$ at time $t_{pickup} + s_p$. Likewise, when $q$ arrives at dropoff location $d_o$ at time $t'$, we assume that $o$ is effectively delivered at time $t_{dropoff} = t' + s_d$, where $s_d$ is the time

it takes $q$ to reach the customer and handover the order; and that $q$ resumes its duties at time $t_{dropoff} + s_d$. At time $t_{dropoff} + s_d$, if $q$ has already been assigned another order, the courier immediately starts heading towards that order's pickup location. Otherwise, we assume that $q$ starts repositioning towards the closest pickup location from its current location. However, if at any point during the repositioning $q$ is assigned to a new order, the courier immediately starts moving towards the order's (possible different) pickup location.

Prior to the start of the operating period, a set of base couriers is scheduled for the day based on expected demand information (expected order volume and placement times). For simplicity, the start location of each base courier is assumed to be the centroid of the pickup location locations. However, the inherent uncertainty of order placements might result in the system's delivery capacity being surpassed by the order volume during the day of operation, at which point the decision maker may request and receive additional on-demand couriers to ease the workload. In the remainder, we will refer to the decision maker as the *agent*. The start location of on-demand couriers added at time $t$ is assumed to be the pickup location with largest number of active unassigned orders at time $t$. Moreover, at time $t$, if a decision is made to add on-demand couriers, then the effective start time of these couriers is $t + \delta$, where $\delta$ is a fixed show-up delay. On-demand couriers will also have a working period length; the working period length, $c$, of an on-demand courier implies an associated cost. We assume that on-demand couriers are always available to be added when needed.

The agent's decision problem is then to decide when and how many on-demand couriers to add with the objective of minimizing the expected overall system cost, which captures the cost of missing orders' delivery time promises, and the cost of dynamically enlarging the system's delivery capacity via on-demand couriers.

### 3.2 Markov decision process formulation

Next, we formulate the problem as a Markov decision process. This formulation is at the core of our deep Q-learning framework.

**Decision epoch.** A time at which the agent decides whether or not to expand delivery capacity by adding one or more on-demand couriers. We assume the agent makes a decision every $\Delta$ time units and that the last epoch occurs at $H_0$. Hence, the set of decision epochs is $T_{action}(\Delta) \doteq \{i\Delta : i \in \{0, 1, \ldots, \frac{H_o}{\Delta}\} \subseteq \mathcal{H}$.

**State.** The state at a epoch $t \in T_{action}(\Delta)$, denoted by $s_t$, encodes the features of the system that we believe

are relevant to make a decision. The features include:

- the time remaining until the end of the operating period, *i.e.*, $H - t$, to capture anticipated temporal demand patterns during the operating period.
- the number of active couriers, $q_t^{courier}$, i.e., the number of couriers currently on-duty plus the number of couriers scheduled to start at time $t$.
- the number of active orders in the system, $q_t^{orders}$, i.e., the number of placed orders that have not been delivered (both assigned and unassigned orders).
- a $j_1$-dimensional vector $\Theta_t^1$ encoding the scheduled changes in the number of on-duty couriers during the time windows $\{(t, t+k_1], (t+k, t+2k_1], \ldots, (t+(j_1-1)k_1, t+j_1k_1]\}$, which captures (allows computing) the available delivery capacity in the near future.
- a $j_2$-dimensional vector $\Theta_t^2$ encoding the number of orders placed during the time windows $\{(t - j_2k_2, t-(j_2-1)k_2], \ldots, (t-2k_2, t-k_2], (t-k_2, t]\}$, which captures (allows predicting) the workload in the near future (which, in turn, allows predicting whether additional delivery capacity will be needed). If for some $j \leq j_3$ we have $t-(j-1)k_2 < 0$, then the count corresponding to $(t-jk_2, t-(j-1)k_2]$ is set to 0.
- a $j_3$-dimensional vector $\Theta_t^3$ encoding the number of orders that will become late during time windows $\{(t, t+k_3], (t+k_3, t+2k_3], \ldots\ (t+(j_3-1)k_3, t+j_3k_3]\}$ if no additional delivery capacity is added, which captures (allows determining) the near-term delivery capacity needs.
- another $j_3$-dimensional vector $\Theta_t^4$ encodes the average completion time at time $t$ of orders that will become late during time windows $\{(t, t+k_3], (t+k_3, t+2k_3], \ldots\ (t+(j_3-1)k_3, t+j_3k_3]\}$ if no additional delivery capacity is added, complementing the information encoded by $\Theta_t^3$.

Thus, at time $t \in \mathcal{H}$, we represent the state as $s_t = (H-t, q_t^{couriers}, q_t^{orders}, \Theta_t^1, \Theta_t^2, \Theta_t^3, \Theta_t^4)$.

**Action.** At decision epoch $t$, the agent chooses an action $a_t$ which specifies for each working period length $c \in \mathcal{C}$ how many couriers with that working period length to add, with the restriction that at most $\overline{m}_c$ couriers of type $c$ can be added (not adding any courier is also a possible action). Any on-demand couriers added at $t$ enter the system at time $t+\delta$, where $\delta > 0$ is a delay (in minutes).

**Reward.** Adding an on-demand $c$-type courier results in a negative reward $K_c < 0$, and a lost order results in a (negative) reward $K_{lost} < 0$. An order is lost as soon as it can be determined that it cannot be delivered on time, and we ignore it from that point on. (At time $t$, determining whether an order is lost takes into account any couriers added at time $t$.) Let $n_{t_1,t_2}$ be the number of orders lost during $[t_1, t_2)$, and let $t^+$ denote the time of the decision epoch following the decision epoch at time $t$. Then an action $a_t$ at time $t$ that adds $k_t^c$ on-demand couriers of type $c$ for $c \in \mathcal{C}$, results in an *immediate* reward at time $t$ of $r_t(s_t, a_t) = K_{lost} \cdot n_{t+\delta,t^+ +\delta} + \sum_{c\in\mathcal{C}} K_c \cdot k_t^c$. Observe that the lost orders associated with action $a_t$ do not include orders that are lost prior $t + \delta$, the start time of any on-demand couriers added at time $t$, as these on-demand couriers cannot prevent orders being lost before $t + \delta$.

**Transition.** The state is updated to reflect decisions at epoch $t \in T_{action}(\Delta)$, any order placements since the last update, and any order – courier assignments since the last update.

- At time $t$, $q_t^{couriers}$ is updated to reflect any couriers ending their duty and any couriers starting their duty at $t$, and $q_t^{orders}$ to reflect any orders placed and any orders delivered at $t$. Likewise, the corresponding counts in $\Theta_t^1, \Theta_t^2$ and $\Theta_t^3$ are updated accordingly.
- If action $a_t$ implies adding on-demand couriers, then $\Theta_t^1$ is updated accordingly, by modifying the change in the number couriers to occur for the time window containing $t + \delta$.
- If an order is placed and it is not immediately assigned to a courier, the order increases the count of one entry of $\Theta_t^3$, since the order may become late in the future.
- Given the assignment logic, once an order is assigned to a courier, it is no longer at risk of being lost and therefore the late order count in $\Theta_t^3$ is updated correspondingly.

**Objective.** Let $\Pi$ be the space of policies $\pi$ that map each possible state to a feasible action. The optimal policy is then specified by

$$\pi^* = \underset{\pi\in\Pi}{\arg\max}\ \mathbb{E}\left[\sum_{t\in T_{action}(\Delta)} r_t(s_t, a_t)\middle| s_0\right]$$

namely, a policy that maximizes the total expected reward over the decision horizon.

### 3.3 Deep Q-learning

Formulating the problem as a Markov decision process allows it, in theory, to be solved using the Bellman equation (1) and backward recursion, where $V(s_t)$ is the value function of state $s_t$, and $A_t$ is the action space at time $t \in T_{action}(\Delta)$:

$$V(s_t) = \max_{a \in A_t} \left\{ r(s_t, a) + \mathbb{E}[V(s_{t+\Delta})|s_t] \right\}, \qquad \forall t \in T_{action}(\Delta). \tag{1}$$

However, the size of the state space in this problem is too large to enumerate. Furthermore, even trying to approximate the value function using tabular RL methods (*e.g.*, tabular Q-learning) is not practically feasible, as storing the value of every possible state-action pair incurs prohibitive memory consumption and accurately learning the value of every pair would require a massive exploration in order to observe all possible state-action combinations. In contrast, deep Q-learning allows to mitigate these practical issues since neural networks can learn the relationship between different state and actions, and use this information to extrapolate the value from explored state-action pairs to the value of unexplored pairs.

The Q-learning algorithm learns the value of taking an action at each given state. The so-called *Q-value* of a state-action pair is learned by a neural network, the DQN. Let $\theta$ be the current weights characterizing the DQN that models the agent, and for a state-action pair $(s_t, a_t)$, let $Q_\theta(s_t, a_t)$ be the Q-value that the DQN of weights $\theta$ associates with that state-action pair. The Q-values approximate Equation (1) as follows

$$V(s_t) \approx \max_{a \in A_t} Q_\theta(s_t, a). \tag{2}$$

Moreover, by performing the learning phase of the DQN offline and storing only its weights $\theta$, the agent is later able to quickly compute, for any state $s_t$, the Q-value $Q_\theta(s_t, a_t)$ for every possible action $a_t$, thus easily identifying the action with the largest expected reward. At decision epoch $t \in T_{action}(\Delta)$, the action $a_t$ selected by $\theta$ to be executed corresponds to the one that maximizes the future expected rewards, *i.e.*,

$$a_t = \arg\max_{a \in A_t} Q_\theta(s_t, a), \quad \forall t \in T_{action}(\Delta) \tag{3}$$

**DQN architecture.** We model the agent as a multi-layer perceptron, a fully connected neural network

characterized by an input layer; $N_{layers}$ hidden layers, each with $N_{nodes}$ nodes, and an output layer. Given an observed state $s_t$, the input layer is fed with the encoded information and is passed to the nodes in the first hidden layer. Each hidden layer receives the output from the previous layer, where each of the $N_{nodes}$ nodes in the layer first computes the dot product between the weights of the inbound connections and the values passed through each connection from the previous layer, and then an activation function is applied on the resulting value before it is passed to each node of the next layer. Our model uses the rectified linear unit function (ReLU) as the activation function for each hidden layer. The last hidden layer passes its output to the output layer, which estimates, for each action $a \in A_t$, the corresponding Q-value $Q_\theta(s_t, a)$.

**Training settings.** The weights $\theta$ of the DQN are learned in a training phase. This phase consists of simulating a total of $N_{episodes}$ instances (simulating an instance is referred to as an episode), each representing a day of operations, *i.e.*, a realization of order placements and a schedule of base couriers.

To make the agent more robust and able to handle multiple scenarios, at the beginning of an episode a daily order placement profile $p$ describing the order placement rate throughout the day is randomly sampled from a set of possible daily order profiles $P$; then, the number of orders placed during the day and their placement times are sampled from the sampled pattern $p$. During an episode, the agent evaluates the system state $s_t$ at every decision epoch $t \in T_{action}(\Delta)$ using the current weights $\theta$, and chooses an action $a_t$, *i.e.*, whether or not to increase the delivery capacity and if yes, how to do so. Given the action, it then collects the associated reward $r_t(s_t, a_t)$ and reaches the next decision epoch at a post-decision state $s'_t$ (see Figure 1 for an illustration of this process). Once the post-decision state is observed, an *experience tuple*

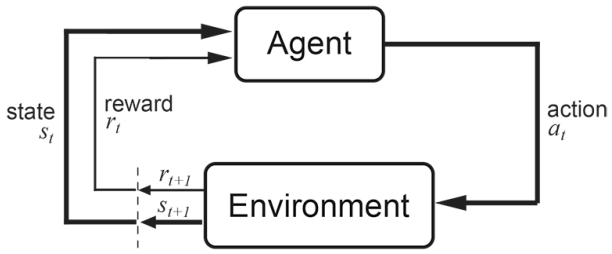



Figure 1: Standard training process in a RL setting (Sutton et al., 1998)

$(s_t, a_t, r_t(s_t, a_t), s'_t)$ is stored to later be used in the update of the DQN parameters.

In the training phase, at each decision epoch $t$, the agent typically selects the action $a_t$ that maximizes the total expected reward associated with the current state $s_t$ (see Equation (3)). This greedy approach is known

as *exploitation*. However, training solely using exploitation may prevent the agent from observing most of the state-action space, thus not being able to find potentially better actions for given states and ultimately getting stuck at a local optima. To mitigate this effect, the agent is occasionally forced to take a random action; this is known as *exploration*. In particular, we adopt an $\varepsilon$-greedy exploration policy during the training phase: at every decision epoch $t$ the agent selects an action uniform randomly from the action space $A_t$ with probability $\varepsilon$, and selects an action based on Equation (3) with probability $1 - \varepsilon$. After completing an episode, the $\varepsilon$ parameter is slightly decreased to gradually increase the likelihood of making decisions based on the Q-values estimated by the agent toward the end of the training phase.

At the end of an episode, instead of immediately training the DQN with the experience tuples just collected, these are first placed in a memory storage, and then the DQN weights $\theta$ are updated using a batch of $b$ tuples randomly sampled from that storage, a process known as experience replay (Lin, 1992). Experience replay makes use of a memory with finite storage capacity $M > b$; once the memory becomes full, newest stored experience tuples overwrite the oldest stored tuples. By using experience tuples from both present and past episodes, this practice seeks to train the DQN using less correlated observations, ultimately stabilizing the parameter updates and improving the training speed of the DQN.

Algorithm 1 illustrates the DQN training process. Line 1 initializes the DQN parameters $\theta$ and the exploration parameter $\varepsilon$. A set of preliminary episodes is run in order to completely initialize the memory with $M$ experience tuples (lines 2 - 5). Then for each of the $N_{episode}$ episodes, the algorithm simulates a complete day of operations gathers $|T_{action}|$ experience tuples and stores them in the experience replay memory, overwriting the oldest stored tuples (lines 7 and 9). At the end of the episode, line 10 samples a batch of size $b$ uniformly at random from the memory. For each sampled tuple, a target Q-value $Q_{\theta,i}^{target}$ is computed, assuming for non-terminal post-decision states $s_i'$ that the best action is the one that maximizes the expected total reward given that the system is in state $s_i'$, and for terminal states $s_i'$ that the reward is simply $r_i$ (lines 11 - 15). Consequently, the algorithm computes the total loss $L(\theta)$ comparing the target values $Q_{\theta,i}^{target}$ with the observed Q-values $Q_\theta(s_t, a_t)$ in line 16, and the loss is then used by the optimizer method $O$ to update the DQN weights $\theta$ in line 17. After updating $\theta$, the algorithm updates the exploration parameter $\varepsilon$ in line 18. Once all the episodes have been simulated, the algorithm returns the trained DQN weights.

Let $\theta^*$ be the DQN weights trained by Algorithm 1. Then for a given state $s_t$, our deep Q-learning policy selects the action $a$ that maximizes the Q-value $Q_{\theta^*}(s_t, a)$, as in Equation (3). We denote this policy as

**Algorithm 1** DQN Training Phase

**Input:** Loss function $L$, discount factor $\gamma$, initial exploration rate $\varepsilon_0$, minimum exploration rate $\varepsilon_{min}$, epsilon decay $\varepsilon_{decay}$, number of episodes $N_{episodes}$, set of decision epochs $T_{action}$, action space $A_t$, batch size $b$, initial DQN weights $\theta_0$, memory size $M$, optimizer method $O$.

**Output:** Trained DQN weights $\theta$.

$\theta \leftarrow \theta_0, \varepsilon \leftarrow \varepsilon_0$
**for** $e \in \left\{1, \ldots, \left\lceil \frac{M}{|T_{action}|} \right\rceil \right\}$ **do**
    **for** $t \in T_{action}$ **do**
        Select an action $a_t \in A_t$ using the current $\varepsilon$-greedy policy.
        Observe and store the experience tuple $(s_t, a_t, r_t, s'_t)$ into the experience replay memory.
**for** $e \in \{1, \ldots, N_{episodes}\}$ **do**
    **for** $t \in T_{action}$ **do**
        Select an action $a_t \in A_t$ using the current $\varepsilon$-greedy policy.
        Observe and store the experience tuple $(s_t, a_t, r_t, s'_t)$ into the experience replay memory.
    Collect a uniformly-random sample a batch of $b$ experience tuples $(s, a, r, s')$ from the memory.
    **for** $i \in \{1, \ldots, b\}$ **do**
        **if** $s'_i$ is terminal **then**
            $Q_{\theta,i}^{target} = r_i$
        **else**
            $Q_{\theta,i}^{target} = r_i + \gamma \max_a Q_\theta(s'_i, a)$
    Compute loss $L(\theta)$
    Update $\theta$ using optimizer $O$ and total loss $L(\theta)$.
    $\varepsilon \leftarrow \max\{\varepsilon - \varepsilon_{decay}, \varepsilon_{min}\}$
**return** $\theta$

$\pi^{DQN}$.

### 3.4 Baseline policies

To assess the performance of the learned policy $\pi^{DQN}$, we propose a set of baseline policies for comparison. These policies react only to the observed order-per-courier ratio mimic how on-demand courier addition decisions are made in practice. We then use these baseline policies as straw man to benchmark against $\pi^{DQN}$.

The baseline policies compare an observed value of OPC to a threshold value and if the observed value exceeds the threshold then on-demand couriers are added until a target OPC is achieved. Let $opc(t)$ be the observed OPC level at time $t \in \mathcal{H}$, let $opc_{max}$ be the maximum allowed OPC level, and let $opc_{target}$ be the target OPC level ($opc_{target} \leq opc_{max}$). Then for each type of courier $c \in \mathcal{C}$, we define the base policy $\pi^c(opc_{max}, opc_{target})$ as follows: at every decision epoch $t \in T_{action}(\Delta)$ (and *immediately after* the couriers scheduled to start and end their block at $t$ do so) we observe state $s_t = opc(t)$ and check if it exceeds $opc_{max}$. If that is the case, then $\pi^c(opc_{max}, opc_{target})$ performs action $a_t$ that adds an extra number $m_t^c$ of couriers of type $c$ to the system until $opc(t)$ falls below $opc_{target}$. More precisely, if $o(t)$ and $z(t)$ are the number of active orders and couriers in the system at time $t$, respectively, then the number of couriers added by $\pi^c(opc_{max}, opc_{target})$, is

$$m_t^c = \begin{cases} 0 & \text{if } opc(t) \leq opc_{max} \\ \\ \lceil \frac{o(t)}{opc_{target}} \rceil - z(t) & \text{otherwise} \end{cases} \tag{4}$$

and $opc(t)$ is updated as

$$opc(t) = \frac{o(t)}{z(t)+m_t^c}. \tag{5}$$

In addition, we consider a hybrid policy $\pi^{hybrid}(opc_{max}, opc_{target})$ that adds on-demand courier also based on Equations (4) and (5) but may add couriers of multiple types $c \in \mathcal{C}$, where the added type depends on the decision epoch $t \in T_{action}(\Delta)$. More precisely, this policy considers a partition of the operating period $\mathcal{H}$ into the collection of subsets $\{\mathcal{H}_c\}_{c\in\mathcal{C}}$, and any action taken at decision epoch $t \in \mathcal{H}_c$ for some $c \in \mathcal{C}$ only

involves couriers of a single type $c$.

### 3.5 Performance metrics

We measure the performance of each considered policy using three metrics: (i) the fraction of orders delivered (on-time), which represents the service level attained by the policy, (ii) the total number of courier-hours added, which measures the additional resources required by the policy to achieve its service level, and (iii) the total reward (without discount), which captures the trade-off between customer service and the cost of expanding delivery capacity. More precisely, let $(\bar{s}_t, \bar{a}_t)$ be the state-action tuples observed by the agent at each decision epoch $t \in T_{action}(\Delta)$, with $\bar{s}_t$ being the observed state at $t$ (for $\pi^{DQN}$, the state $s_t$ is defined as in Section 3.2, for $\pi^c$ it is simply $opc(t)$, and for $\pi^{hybrid}$ it is $t$ and $opc(t)$), let $\bar{a}_t$ the corresponding action taken; and let $n_0^H$ be the number of orders lost during $\mathcal{H}$ of a total of $n$ placed orders. Then the total reward is computed as $\sum_{t \in T_{action}(\Delta)} r_t(\bar{s}_t, \bar{a}_t)$, and the service level corresponds to $\frac{n - n_0^H}{n}$.

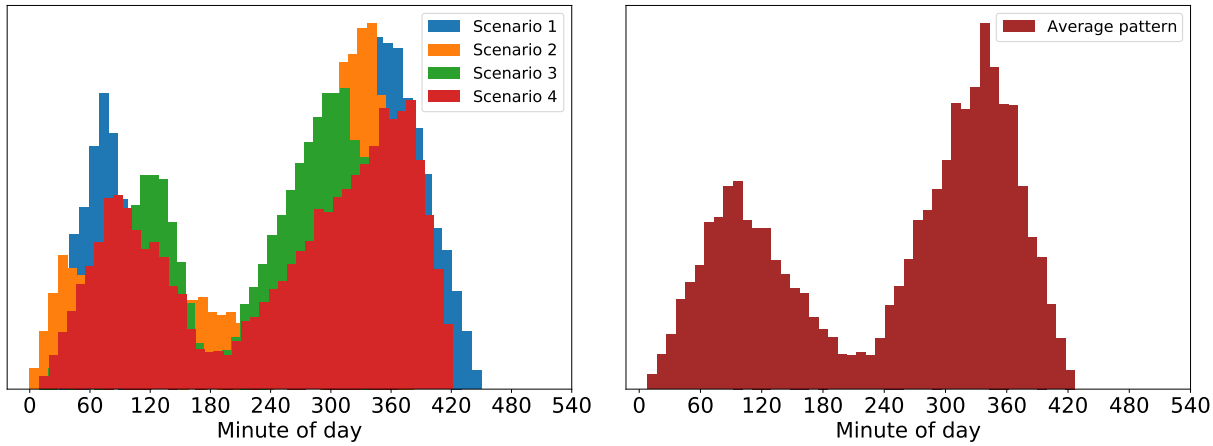
## 4 Experimental results

This section shows the evolution of the learning that occurs during the training of the DQN, and presents the results of a comparison between the performance of two learned policies and the three baseline policies defined in Section 3.4 in terms of total reward, service quality, and courier hours acquired on-demand.

### 4.1 Experimental settings

Each episode covers an operating period of $H = 540$ minutes during which orders are dynamically placed at a set of $N_{pickup} = 16$ pickup locations. Orders are placed during the first $H_0 = 450$ minutes. Service times are set to $s_p = s_d = 4$ minutes. For an order $o$, the ready time is set to $e_o = t_o + 10$ and the promised delivery time is set to $m_o = t_o + 40$.

At the beginning of each episode, the number of orders placed at each pickup location $d$, $n_d$, is drawn from a uniform distribution, i.e. $n_d \sim \mathcal{U}[10, 20]$ for $d \in \{1, \ldots, N_{pickup}\}$; the total number of orders, $n$, therefore, is $n = \sum_{d=1}^{N_{pickup}} n_d$. An order placement time pattern $p$ is then uniformly sampled from a set of patterns $P$ comprising four different scenarios; a pattern $p$ corresponds to a probability mass function as depicted in Figure 2a. Given a selected pattern, each order's placement time is sampled using the mass function for pattern $p$. Each of the scenarios in $P$ has a moderate peak in order placements around lunch time, a large




(a) Graphical representation of the set of order placement patterns $P$

(b) Average order placement pattern obtained from the patterns in $P$, used to schedule base courier capacity

Figure 2: Order placement patterns used to create episodes.

peak in order placements around dinner time, and an off-peak period with relatively few order placements between the two peaks, reflecting what is observed in meal delivery operations in practice. However, there are differences between the four scenarios that reflect external conditions that impact demand, such as special events or changes in weather.

Both base and on-demand couriers work for either 1 or 1.5 hours (*i.e.*, $\mathcal{C} = \{1, 1.5\}$), so we model each action as a two-dimensional vector $(a_t^1, a_t^{1.5})$ where $a_t^c$ is the number of $c$-hour couriers added at time $t$. Moreover, there is a decision epoch, a time at which the agent takes an action, every $\Delta = 5$ minutes. At each decision epoch, we also limit the agent to add up to two on-demand couriers of each type, *i.e.*, $A_t = \{0, 1, 2\} \times \{0, 1, 2\}$.

**Base capacity schedule** The number of base $c$-hour couriers, $D_c$ for $c \in \mathcal{C}$, to be scheduled at the beginning of an episode is randomly sampled from a discrete uniform distribution, i.e., $D_c \sim \mathcal{U}[\ell_c, u_c]$; we run experiments for two different initial base capacity levels: a lower capacity setting with $(\ell_1, u_1; \ell_{1.5}, u_{1.5}) = (20, 30; 10, 20)$, and a higher capacity one with $(\ell_1, u_1; \ell_{1.5}, u_{1.5}) = (30, 40; 20, 30)$. The scheduling is based on the average order placement profile $\overline{p}$ shown in Figure 2b (obtained by averaging the placement patterns in $P$). The start times of the couriers are set as follows:

- For the 1-hour couriers, $\lceil \frac{D_1}{6} \rceil$ of them are set to start at time 60, and another $\lceil \frac{D_1}{6} \rceil$ couriers to start at time 120. Furthermore, another $\lceil \frac{D_1}{3} \rceil$ couriers are set to start at time 270, and the remaining couriers

Table 1: Tested and selected values for training parameters. Final values are selected based on best average reward obtained in the testing phase.

| Parameter | Tested values | Selected value |
|---|---|---|
| $N_{episodes}$ | $\{1, 2, 3, 6\} \cdot 10^4$ | $\{3, 6\} \cdot 10^4$ |
| $N_{layers}$ | $\{1, 2, 3, 4\}$ | $\{3, 4\}$ |
| $N_{nodes}$ | $\{16, 32, 64\}$ | 64 |

are set to start at time 330.

- For the 1.5-hour couriers, $\lceil \frac{D_{1.5}}{6} \rceil$ of them are set to start at time 0, and another $\lceil \frac{D_{1.5}}{6} \rceil$ couriers to start at 120. Furthermore, another $\lceil \frac{D_{1.5}}{3} \rceil$ couriers are set to start at time 240, and the remaining couriers are set to start their working period at 360.

- If it is a training episode, then each courier's start time is perturbed by adding a realization of the discrete uniform random variable $u \sim \mathcal{U}[-20, 20]$ (a perturbed start time that occurs before the beginning of the horizon is replaced by 0).

The reason for having a random number of base couriers and perturbed start times of the base couriers is to facilitate exploration of more state-action pairs during training (by having scenarios in which the system is under-staffed and over-staffed) so that the policy learned by the agent is able to take good decisions in a variety of settings.

**Training settings** We model the DQN as a multi-layer perceptron (*i.e.*, a fully-connected neural network) with $N_{layers}$ hidden layers, each with $N_{nodes}$ nodes and ReLU activation function, and an output layer with $|A_t|$ nodes. The results we present in this section correspond to a policy trained with the selected values specified in Table 1, which were selected based on the average reward obtained by the corresponding trained policy over the test instances.

We experiment training a DQN using two different state vectors that differ in their definition of $\Theta_t^1, \Theta_t^2, \Theta_t^3$ and $\Theta_t^4$. In particular, we consider:

- An interval-based state vector that at a given decision epoch $t$, its components $\Theta_t^1, \Theta_t^2, \Theta_t^3$ and $\Theta_t^4$ are defined such that $(j_1, k_1) = (4, 30), (j_2, k_2) = (6, 5)$, and $(j_3, k_3) = (4, 10)$. That is, the scheduled changes to the pool of on-duty couriers after time $t$, $\Theta_t^1$, are captured using time windows $\{(t, t + 30], (t + 30, t + 60], (t + 60, t + 90], (t + 90, t + 120]\}$, which are the most relevant windows for an on-demand courier scheduled at $t$ since a courier works for up to 1.5 hours in these experiments

(we include an extra window to capture potential benefits of looking ahead the 1.5-hour duration); the number of placed orders before time $t$, $\Theta_t^2$, are captured using time windows $\{(t-30, t-25], (t-25, t-20], \ldots, (t-10, t-5], (t-5, t]\}$; and the number of unassigned orders at time $t$, $\Theta_t^3$, whose latest pickup time falls in the time windows $\{(t, t+10], (t+10, t+20], (t+20, t+30], (t+30, t+40]\}$, which covers all active orders since none of them may spend more than 40 minutes unassigned before being lost. Given that this definition results in a 21-dimensional state vector, we denote this state and the policy trained under this definition by $s_t^{21}$ and $\pi_{21}^{DQN}$, respectively.

- A value-based state vector that aggregates the information in $s_t^{21}$, *i.e.*, at a given decision epoch $t$, its components $\Theta_t^1, \Theta_t^2, \Theta_t^3$ and $\Theta_t^4$ are defined such that $(j_1, k_1) = (1, 120), (j_2, k_2) = (1, 30)$, and $(j_3, k_3) = (1, 40)$. In this case, $\Theta_t^1$ counts the aggregated scheduled changes to the pool of on-duty couriers after time $t$ over the next two hours, *i.e.*, for the time window $(t, t+120]$; similarly, $\Theta_t^2$ now captures the aggregated number of placed orders before time $t$, this is, only for the time window $(t-30, t]$; and $\Theta_t^3$ and $\Theta_t^4$ now respectively encode the number of unassigned orders and their average completion time at time $t$ that will become late . As this definition results in a 7-dimensional state vector, we denote this state and the corresponding trained policy by $s_t^7$ and $\pi_7^{DQN}$, respectively.

Prior to feeding the state vector as input to the DQN, the encoded information is normalized using a min-max scaler to give more accurate relative importance between the considered features (Bishop, 1995).

Given that the agent takes action $a_t$, on-demand couriers associated with $a_t$ start their working period after a delay of $\delta = 5$ minutes (i.e. they start at $t + 5$). The unit reward of adding a courier of each type are $(K_1, K_{1.5}) = (-0.2, -0.25)$. Note that the per-hour cost of a 1.5-hour courier is less than the per-hour cost of a 1-hour courier; this is justified by the fact that 1.5-hour couriers have more flexibility to receive orders due to the assignment feasibility conditions, thus having the potential of serving more orders. The unit reward for lost order is $K_{lost} = -1$ so service quality degradation receives a heavier penalization than adding an extra unit of courier capacity. We set discount factor employed in our experiments to $\gamma = 0.99$.

We train the agent using an $\varepsilon$-greedy approach to effectively explore the state space during training, starting with a value of $\varepsilon_0 = 1$ and linearly decreasing it to $\varepsilon_{min} = 0.01$ over the considered $N_{episodes}$ episodes, thus $\varepsilon_{decay} = \frac{\varepsilon_0 - \varepsilon_{min}}{N_{episodes}}$. The loss function we consider to train the DQN is the *mean-square error*, and the optimization of the DQN weights $\theta$ is performed using the Adam optimizer (Kingma and Ba, 2014), with

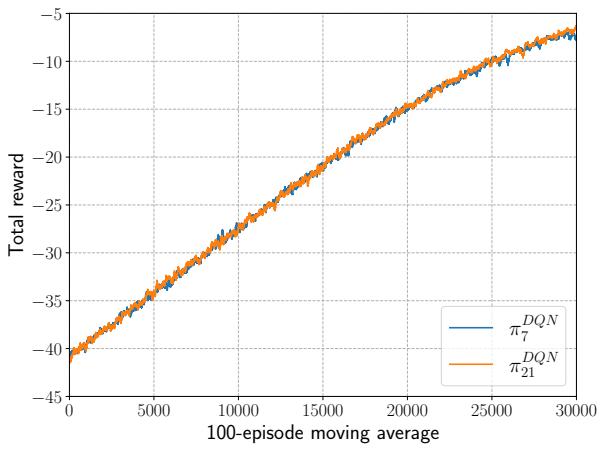
a learning rate of $10^{-3}$, and parameter values $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\varepsilon = 10^{-3}$. Our implementation of experience replay considers a memory size of $M = 10,000$ tuples, and after each episode the network weights are trained using a random minibatch of size $b = 1,000$.

### 4.2 Learning $\pi^{DQN}$

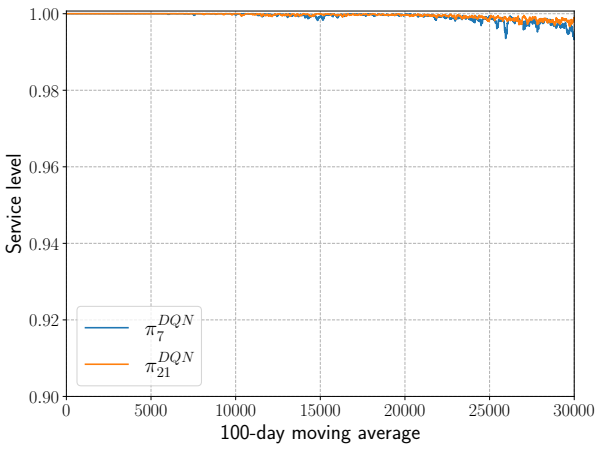
Next, we investigate how the performance achieved by the policy during training improves over time, and the corresponding changes in the number of added on-demand couriers of each type.

Figure 3 illustrates the evolution of the average total rewards, the average service level, and the average number of added on-demand 1-hour and 1.5-hour couriers, as the agent learns $\pi^{DQN}$ over $N_{episodes} = 30,000$ episodes, and assuming base courier fleet sampling parameter values $(\ell_1, u_1; \ell_{1.5}, u_{1.5}) = (30, 40; 20, 30)$. In particular, each data point is the average of 100 consecutive episodes, and each policy is modeled as a neural net with $N_{layers} = 3$ hidden layers of $N_{nodes} = 64$ nodes each. Both trained policies follow a similar learning process through the entire training phase. Initially, the agent follows a policy heavily focused on exploration (i.e. high frequency of random actions in the $\varepsilon-$greedy exploration) that is able to serve 100% of the orders, but at the expense of incorporating a large number of on-demand couriers as eight out of the nine possible actions result in adding extra delivery capacity to the system, thereby greatly increasing the total costs. However, we observe that the average reward attained by the agent steadily increases during the training phase, as illustrated in Figure 3a. We can also see the agent's transition from exploration to exploitation; after episode 20,000 the average reward improvement slows down due to a lower exploration rate $\varepsilon$.

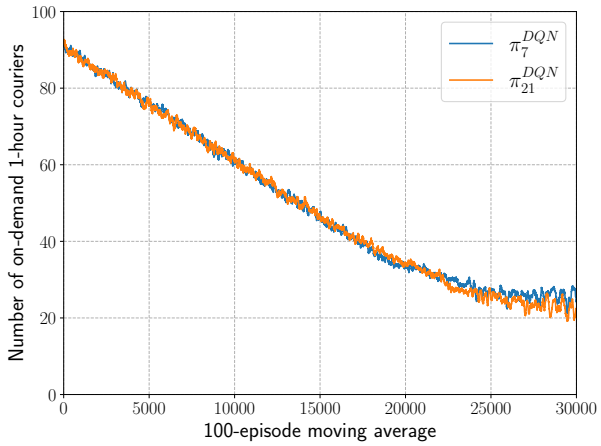
This is also reflected in the evolution of the average service level and the average number of on-demand 1-hour and 1.5-hour couriers added, shown, respectively, in Figures 3b, 3c, and 3d. The number of 1-hour and 1.5-hour on-demand couriers added by either policy shows a steady decrease over the first 20,000 episodes. After this point, the number of on-demand 1-hour couriers added by the agent starts to converge, observing a final average of 24.7 1-hour couriers added per episode by $\pi_7^{DQN}$, while $\pi_{21}^{DQN}$ reaches an average of 21.7. Interestingly, the agent shows "less appreciation" for the lower per-hour cost of adding on-demand 1.5-hour couriers; by the end of the training phase $\pi_7^{DQN}$ converges to a per-episode average of 4.7 1.5-hour additional couriers, while $\pi_{21}^{DQN}$ to a slightly higher average of 6.6. This is likely due to the base fleet scheduling and the order placement patterns: in most situations, the periods during which extra couriers



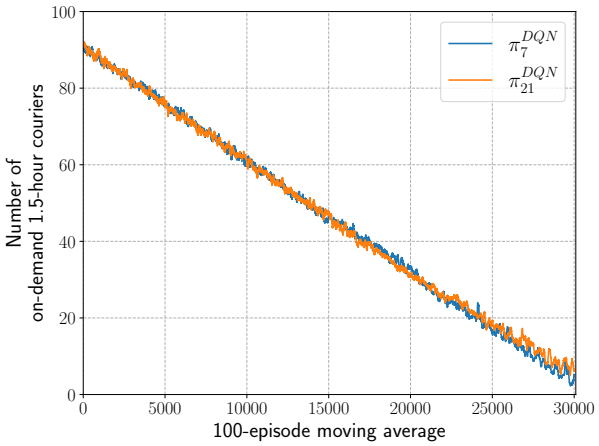

(a) Total rewards




(b) Service level



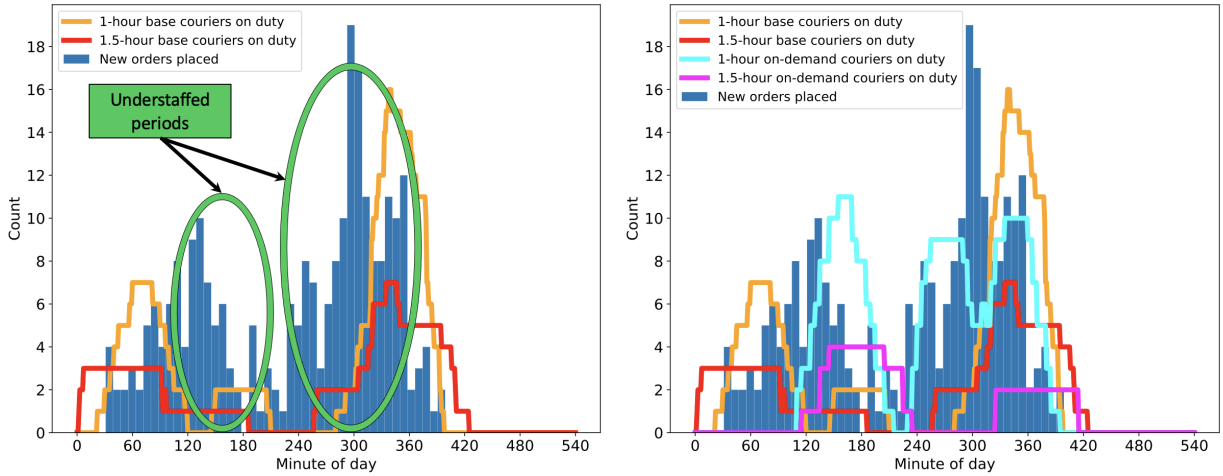

(c) 1-hour on-demand couriers




(d) 1.5-hour on-demand couriers

Figure 3: Evolution of performance metrics during training episodes; 100-episode moving average. Each policy shown was trained using $(N_{episodes}, N_{layers}, N_{nodes}) = (30000, 3, 64)$, and assuming $D_1 \sim \mathcal{U}[30, 40], D_{1.5} \sim \mathcal{U}[20, 30]$

are needed are relatively short, hence deciding to use the more expensive 1.5-hour couriers has to done judiciously. We see that near episode 20,000 the service level achieved by both policies slightly decreases as the number of on-demand couriers added reaches a level at which serving all orders is no longer trivial. Interestingly, the agent is in general able to identify market signals that indicate when additional courier capacity is required, thereby preventing a capacity shortage when demand deviates from the average order placement pattern, as illustrated in Figure 4. Ultimately, this allows an average reduction of 85% in the number of on-demand couriers added per episode by the end of the training phase (relative to the first set of training episodes) while successfully serving an average of over 99.5% of the orders.




(a) Order coverage *without* on-demand couriers

(b) Order coverage *with* on-demand couriers

Figure 4: Order coverage in scenarios with and without the capability of adding on-demand courier capacity. Adding couriers in an on-demand basis makes possible to prevent capacity shortage when realized order placement deviates from the average pattern

### 4.3 Performance

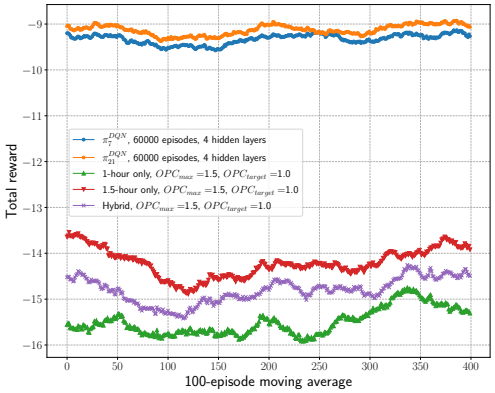
In this section, we compare the performance of the trained policies $\pi_7^{DQN}$ and $\pi_{21}^{DQN}$ to the performance of the baseline policies $\pi^1$, $\pi^{1.5}$ and $\pi^{hybrid}$. More specifically, we produce 500 independent instances using the order placement profiles in $P$ and report the results of each of the policies on these instances. We consider five configurations of each of the baseline policies, by setting the hyperparameter $opc_{max}$ to every element of the set of values $\{1.0, 1.5, 2.0, 2.5, 3.0\}$ and setting the target OPC to $opc_{target} = opc_{max} - 0.5$, and report the results for the configuration that yields the best average reward. Table 2 shows for each baseline policy, the parameter configuration that yields the highest average reward. Also, for $\pi^{hybrid}$ it is assumed that $\mathcal{H}_1 = [60, 120] \cup [300, 360]$ and $\mathcal{H}_{1.5} = \mathcal{H} \setminus \mathcal{H}_1$. To select each trained policy for the benchmark,

every 50 episodes of its training process we evaluate the average reward obtained by the agent in the last 50 episodes, and we save the weights of the current DQN if the computed average reward is strictly greater than the one observed for the previous saved DQN; the final policy that results from the training process is given by the the last DQN saved in this fashion.

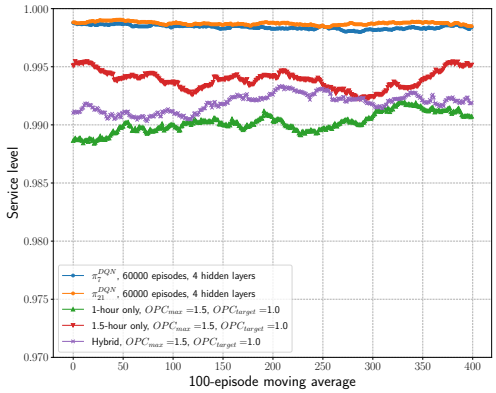
Table 2: Tuned values of $opc_{max}$ and $opc_{target}$ for baseline policies

| $(\ell_1, u_1; \ell_{1.5}, u_{1.5})$ | Policy | $opc_{max}$ | $opc_{target}$ |
|---|---|---|---|
| $(20, 30, 10, 20)$ | $\pi^1$ | 1.5 | 1.0 |
| | $\pi^{1.5}$ | 1.5 | 1.0 |
| | $\pi^{hybrid}$ | 1.5 | 1.0 |
| $(30, 40, 20, 30)$ | $\pi^1$ | 1.5 | 1.0 |
| | $\pi^{1.5}$ | 1.5 | 1.0 |
| | $\pi^{hybrid}$ | 1.5 | 1.0 |

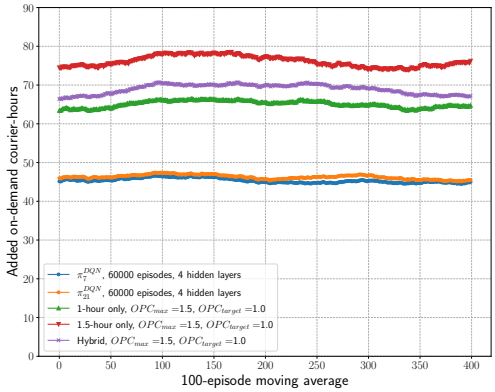
We evaluate the performance of the policies using three metrics: the average total reward, the average service level, and the average number of on-demand courier hours added (with the average taken over the 500 instances). Results for the setting $(N_{episodes}, n_{layers}) = (60000, 4), D_1 \sim \mathcal{U}[20, 30], D_{1.5} \sim \mathcal{U}[10, 20]$ are shown in Figure 5, where each curve depicts the evolution of the 100-instance moving average of the corresponding metric, i.e., each data point represents the average of 100 instances (for a total of 400 data points). Furthermore, Table 3 reports the average and standard deviation of the values of the metrics for each tested policy over the 500 test instances. We provide plots showing the performance results for all learned and base policies in Appendix 5.



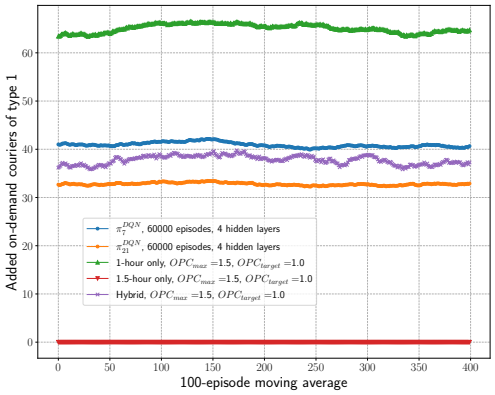

(a) Total reward



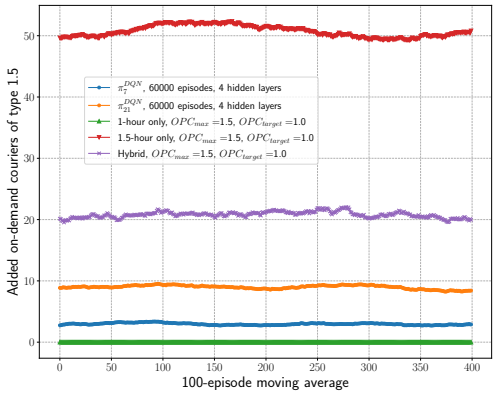

(b) Service level




(c) On-demand courier hours




(d) On-demand 1-hour couriers




(e) On-demand 1.5-hour couriers

Figure 5: Policy benchmark over 500 test instances; 100-episode moving average - $(N_{episodes}, N_{layers}) = (60000, 4), D_1 \sim \mathcal{U}[20, 30], D_{1.5} \sim \mathcal{U}[10, 20]$

Table 3: Per-policy average and standard deviation of performance metrics over the 500 test instances

| $(\ell_1, u_1; \ell_{1.5}, u_{1.5})$ | $(N_{episodes}, N_{layer})$ | Policy | Total reward | | Service level | | On-demand couriers added | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | # 1-hour | | # 1.5-hour | | Total hours | |
| | | | Avg. | Std. | Avg. | Std. | Avg. | Std. | Avg. | Std. | Avg. | Std. |
| $(20, 30; 10, 20)$ | $(30000, 3)$ | $\pi_7^{DQN}$ | -9.1 | 1.20 | 99.9% | 0.19% | 37.4 | 4.54 | 5.7 | 2.65 | 46.0 | 5.35 |
| | | $\pi_{21}^{DQN}$ | -9.6 | 1.53 | 99.7% | 0.32% | 33.1 | 4.67 | 8.9 | 3.47 | 46.4 | 6.18 |
| | $(30000, 4)$ | $\pi_7^{DQN}$ | -9.6 | 1.50 | 99.7% | 0.36% | 37.6 | 4.91 | 5.3 | 3.56 | 45.5 | 5.99 |
| | | $\pi_{21}^{DQN}$ | -9.2 | 1.47 | 99.8% | 0.26% | 38.6 | 5.00 | 4.5 | 2.69 | 45.3 | 6.14 |
| | $(60000, 3)$ | $\pi_7^{DQN}$ | -9.4 | 1.41 | 99.8% | 0.30% | 36.9 | 4.67 | 6.0 | 2.55 | 46.0 | 5.77 |
| | | $\pi_{21}^{DQN}$ | -9.4 | 1.48 | 99.8% | 0.26% | 35.3 | 4.90 | 7.3 | 3.31 | 46.3 | 6.12 |
| | $(60000, 4)$ | $\pi_7^{DQN}$ | -9.3 | 1.45 | 99.8% | 0.26% | 40.9 | 4.82 | 2.9 | 2.00 | 45.3 | 5.81 |
| | | $\pi_{21}^{DQN}$ | -9.1 | 1.45 | 99.9% | 0.23% | 32.8 | 4.59 | 8.9 | 2.94 | 46.2 | 6.25 |
| | - | $\pi^{1}$ | -15.5 | 2.92 | 99.0% | 0.90% | 64.9 | 8.70 | - | - | 64.9 | 8.70 |
| | | $\pi^{1.5}$ | -14.1 | 2.50 | 99.4% | 0.71% | - | - | 50.6 | 8.19 | 76.0 | 12.28 |
| | | $\pi^{hybrid}$ | -14.8 | 2.92 | 99.2% | 0.90% | 37.8 | 17.10 | 20.6 | 11.34 | 68.6 | 9.64 |
| $(30, 40; 20, 30)$ | $(30000, 3)$ | $\pi_7^{DQN}$ | -7.2 | 1.65 | 99.7% | 0.33% | 28.5 | 4.84 | 3.1 | 2.28 | 33.2 | 6.46 |
| | | $\pi_{21}^{DQN}$ | -7.1 | 1.68 | 99.7% | 0.30% | 26.6 | 4.96 | 4.3 | 2.77 | 33.1 | 7.03 |
| | $(30000, 4)$ | $\pi_7^{DQN}$ | -7.0 | 1.77 | 99.8% | 0.34% | 31.1 | 5.54 | 1.0 | 1.48 | 32.6 | 6.16 |
| | | $\pi_{21}^{DQN}$ | -7.4 | 1.73 | 99.6% | 0.34% | 23.5 | 4.22 | 7.1 | 3.22 | 34.1 | 7.18 |
| | $(60000, 3)$ | $\pi_7^{DQN}$ | -7.4 | 2.35 | 99.7% | 0.63% | 30.4 | 5.85 | 1.8 | 1.82 | 33.0 | 6.21 |
| | | $\pi_{21}^{DQN}$ | -7.0 | 1.66 | 99.8% | 0.29% | 25.4 | 4.98 | 5.3 | 3.01 | 33.4 | 7.00 |
| | $(60000, 4)$ | $\pi_7^{DQN}$ | -6.8 | 1.50 | 99.9% | 0.25% | 29.7 | 5.43 | 2.2 | 2.16 | 33.0 | 6.27 |
| | | $\pi_{21}^{DQN}$ | -6.9 | 1.72 | 99.8% | 0.30% | 27.2 | 5.36 | 3.9 | 2.33 | 33.0 | 6.99 |
| | - | $\pi^{1}$ | -11.8 | 2.89 | 99.3% | 0.85% | 49.6 | 10.09 | - | - | 49.6 | 10.09 |
| | | $\pi^{1.5}$ | -11.7 | 2.81 | 99.5% | 0.77% | - | - | 41.3 | 8.94 | 62.0 | 13.41 |
| | | $\pi^{hybrid}$ | -11.8 | 2.98 | 99.3% | 0.83% | 28.4 | 16.66 | 17.3 | 11.91 | 54.3 | 11.19 |

We observe that both $\pi_7^{DQN}$ and $\pi_{21}^{DQN}$ greatly outperform the base policies $\pi^1, \pi^{1.5}$ and $\pi^{hybrid}$ in terms of average reward: for both low and high base courier capacity, the learned policies not only improve the baselines rewards by $36\%$ and $40\%$, respectively, but are also more robust with 48% and $40\%$ lower standard deviation, respectively. This shows that the trained policies more accurately assess when the system needs additional delivery capacity, which not only results in a better and more consistent service level, but does so with around 34% and 40% fewer on-demand couriers for the scenarios with low and high base courier capacity, respectively. This is especially clear when contrasting $\pi_7^{DQN}$ with $(N_{episodes}, N_{layers}) = (30000, 4)$ with $\pi^1$ in the high base courier capacity setting: even though $\pi_7^{DQN}$ almost exclusively adds 1-hour on-demand couriers (as does $\pi^1$), $\pi_7^{DQN}$ is able to serve slightly more orders with 34% fewer courier hours, significantly outperforming $\pi^1$ in terms of total reward. The learned policy exploits the information embedded in the state vector to properly time the addition of on-demand capacity, thereby outperforming the purely OPC threshold-based policy. Another interesting observation is the fact that the policy $\pi^{1.5}$, which only adds 1.5-hour on-demand couriers, consistently attains the best performance among the base policies, contrasting to the actions followed by the learned policies, which favor the addition of 1-hour on-demand couriers.

Comparing the learned policies $\pi_7^{DQN}$ and $\pi_{21}^{DQN}$, we observe that they tend to perform similarly in terms of total reward, attaining comparable service levels and number of on-demand courier hours. Interestingly, the actions taken differ somewhat: while $\pi_7^{DQN}$ shows a clear preference towards 1-hour couriers, the additional information in the state vector used by $\pi_{21}^{DQN}$ makes it follow a more balanced set of actions, resulting in the addition of consistently more 1.5-hour couriers. The difference is most apparent for the cases $(N_{episodes}, N_{layers}) = (60000, 4)$ and $(30000, 4)$ with lower and higher base courier capacity, respectively, where $\pi_{21}^{DQN}$ acquires between three to seven times the amount of 1.5-hour couriers than $\pi_7^{DQN}$; in practice, this feature can be advantageous if couriers had diverse preferences on the length of their operating blocks.

We also study the effect of varying the learning parameters $(N_{episodes}, N_{layers})$ and the findings depend on the base courier capacity. For a small base capacity, the agent will need to add many on-demand couriers to achieve a proper service level. As 8 out of the 9 possible actions involve adding some type of courier, we believe it is easier for a policy to learn the correct actions in this setting; this is evidenced by the fact that in the most basic learning configuration $(N_{episodes}, N_{layers}) = (30000, 3)$, the simpler policy $\pi_7^{DQN}$ already attains its best performance, and either adding more hidden layers to the DQN or stretching the training

phase does not result in any improvements (in fact, we observe a deterioration in total rewards, possibly due to overfitting). On the contrary, $\pi_{21}^{DQN}$ achieves its worst performance in the basic configuration, but it slightly improves when the DQN receives an extra hidden layer and is trained for more episodes; the more complex state requires extending the training phase to better learn the importance of each component of the state, while the extra layer gives the DQN more modeling capabilities. On the other hand, for a large base courier capacity, the agent will need to add few on-demand couriers, and therefore will also need to learn when not to acquire additional courier capacity. As there is only one action that does not add any on-demand couriers, learning how to do so is more challenging. This is in line with our results: with respect to the most basic DQN configuration $(N_{episodes}, N_{layers}) = (30000, 3)$, just training $\pi_7^{DQN}$ for a longer period deteriorates its performance due to overfitting; however, also adding more modeling power by incorporating an extra hidden layer results in higher rewards. By contrast, $\pi_{21}^{DQN}$ improves its performance when trained for a longer period, suggesting that the DQN's number of parameters are suitable for the more complex state vector, and moreover, adding an extra layer and extending the training phase boosts its rewards further (making it comparable to $\pi_{21}^{DQN}$); however, only adding an extra hidden layer results in a performance degradation due to overfitting.

# 5 Conclusion

Our research considers the application of deep reinforcement learning techniques to manage courier capacity in a meal delivery system. Such a system is highly dynamic and highly uncertain, which means that effectively managing courier capacity is extremely challenging. The use of a deep Q-learning network makes it possible to effectively consider many aspects of a meal delivery environment when deciding courier capacity, ultimately making it possible to derive a cost-efficient policy that outperforms standard practices in the industry.

A natural next step is to develop a deeper understanding of the actions dictated by the obtained policy, specifically in terms of the courier type preference and the timing of capacity expansions, and to consider improving the baseline policies to further justify the use of machine learning algorithms in this context. Another key aspect to study is the robustness of the policy when applied to instances that significantly deviate from the ones used during training, with respect to both demand distribution and values of application-specific parameters.

In terms of the RL framework, it may be worth considering potential enhancements such as the use of target networks, double DQN and prioritized experience replay, which in various other settings have further improved the performance of the resulting policy. The positive results obtained in this work using DQN also suggests that this methodology may be useful to study other aspects of meal delivery. In particular, a similar framework could be used to improve the results on dynamic zone management (see, e.g., Auad et al. (2021)) and to learn courier-order assignment policies that optimize service quality (see, e.g., Reyes et al. (2018)).

## References


Agatz, N., Erera, A., Savelsbergh, M., and Wang, X. (2012). Optimization for dynamic ride-sharing: A review. *European Journal of Operational Research*, 223(2):295–303. https://doi.org/10.1016/j.ejor.2012.05.028.

Alfonso, V., Boar, C., Frost, J., Gambacorta, L., and Liu, J. (2021). E-commerce in the pandemic and beyond. *BIS Bulletin*, 36.

Arslan, A. M., Agatz, N., Kroon, L., and Zuidwijk, R. (2019). Crowdsourced delivery—a dynamic pickup and delivery problem with ad hoc drivers. *Transportation Science*, 53(1):222–235. .

Auad, R., Erera, A., and Savelsbergh, M. (2021). Courier satisfaction in rapid delivery systems using dynamic operating regions. *Optimization Online*.

Auad, R., Erera, A., and Savelsbergh, M. (2022). Capacity requirements and demand management strategies in meal delivery. *Optimization Online*.

Auad, R. and Van Hentenryck, P. (2021). Ridesharing and fleet sizing for on-demand multimodal transit systems. *arXiv preprint arXiv:2101.10981*.

Berbeglia, G., Cordeau, J.-F., and Laporte, G. (2010). Dynamic pickup and delivery problems. *European journal of operational research*, 202(1):8–15. .

Bertossi, A. A., Carraresi, P., and Gallo, G. (1987). On some matching problems arising in vehicle scheduling models. *Networks*, 17(3):271–281.

Bishop, C. M. (1995). *Neural networks for pattern recognition*. Oxford university press.

Bunte, S. and Kliewer, N. (2009). An overview on vehicle scheduling models. *Public Transport*, 1(4):299–317.

Capalbo, V., Ghiani, G., and Manni, E. (2021). The role of optimization and machine learning in e-commerce logistics in 2030. *International Journal of Economics and Management Engineering*, 15(3):290–294. .

Chen, X., Ulmer, M. W., and Thomas, B. W. (2019). Deep q-learning for same-day delivery with a heterogeneous fleet of vehicles and drones. *arXiv preprint arXiv:1910.11901*. .

Chen, X. M., Zheng, H., Ke, J., and Yang, H. (2020). Dynamic optimization strategies for on-demand ride services platform: Surge pricing, commission rate, and incentives. *Transportation Research Part B: Methodological*, 138:23–45. .

Dayarian, I. and Savelsbergh, M. (2020). Crowdshipping and same-day delivery: Employing in-store customers to deliver online orders. *Production and Operations Management*, 29(9):2153–2174. .

Guedes, P. C. and Borenstein, D. (2018). Real-time multi-depot vehicle type rescheduling problem. *Transportation Research Part B: Methodological*, 108:217–234.

Hausman, L., Herrmann, N.-A., Krause, J., and Netzer, T. (2014). Same-day delivery: The next evolutionary step in parcel logistics. Retrieved from https://www.mckinsey.com/industries/travel-logistics-and-infrastructure/our-insights/same-day-delivery-the-next-evolutionary-step-in-parcel-logistics#. Accessed May 2021.

Huisman, D., Freling, R., and Wagelmans, A. P. (2004). A robust solution approach to the dynamic vehicle scheduling problem. *Transportation Science*, 38(4):447–458.

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

Klapp, M. A., Erera, A. L., and Toriello, A. (2018). The dynamic dispatch waves problem for same-day delivery. *European Journal of Operational Research*, 271(2):519–534. https://doi.org/10.1016/j.ejor.2018.05.032.

Lin, K., Zhao, R., Xu, Z., and Zhou, J. (2018). Efficient large-scale fleet management via multi-agent deep reinforcement learning. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1774–1783.

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

Mabe, J. (2017). The Amazonification of Logistics. Retrieved from https://www.techgistics.net/blog/2017/6/4/the-amazonification-of-logistics-amazon. Accessed May 2021.

Mercatus (2017). The Evolution of the Grocery Customer. Retrieved from https://www.mercatus.com/blog/grocery-retail-innovation/. Accessed May 2021.

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

Morgan Stanley Research (2017). Is online food delivery about to get "amazoned?". https://www.morganstanley.com/ideas/online-food-delivery-market-expands/. [Online; accessed 24-December-2018].

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

Popper, N. (2020). Americans Keep Clicking to Buy, Minting New Online Shopping Winners. https://www.nytimes.com/interactive/2020/05/13/technology/online-shopping-buying-sales-coronavirus.html.

Psaraftis, H. N., Wen, M., and Kontovas, C. A. (2016). Dynamic vehicle routing problems: Three decades and counting. *Networks*, 67(1):3–31. https://doi.org/10.1002/net.21628.

Qin, Z., Tang, X., Jiao, Y., Zhang, F., Xu, Z., Zhu, H., and Ye, J. (2020). Ride-hailing order dispatching at didi via reinforcement learning. *INFORMS Journal on Applied Analytics*, 50(5):272–286. https://doi.org/10.1287/inte.2020.1047.

Reyes, D., Erera, A., Savelsbergh, M., Sahasrabudhe, S., and O'Neil, R. (2018). The meal delivery routing problem. *Optimization Online*.

Saha, J. (1970). An algorithm for bus scheduling problems. *Journal of the Operational Research Society*, 21(4):463–474.

Savelsbergh, M. and Ulmer, M. (2022). Challenges and opportunities in crowdsourced delivery planning and operations. *4OR-Q J Oper Res*. https://doi.org/10.1007/s10288-021-00500-2.

Statista Report (2019). Online food delivery. https://www.statista.com/outlook/374/100/online-food-delivery/worldwide#market-arpu.

Sutton, R. S., Barto, A. G., et al. (1998). *Introduction to reinforcement learning*, volume 135. MIT press Cambridge.

Uçar, E., Birbil, Ş. İ., and Muter, İ. (2017). Managing disruptions in the multi-depot vehicle scheduling problem. *Transportation Research Part B: Methodological*, 105:249–269.

Ulmer, M., Thomas, B., and Mattfeld, D. (2019). Preemptive depot returns for dynamic same-day delivery. *EURO Journal on Transportation and Logistics*, 8(4):327–361. https://doi.org/10.1007/s13676-018-0124-0.

Ulmer, M. W. and Streng, S. (2019). Same-day delivery with pickup stations and autonomous vehicles. *Computers & Operations Research*, 108:1–19. https://doi.org/10.1016/j.cor.2019.03.017.

Ulmer, M. W., Thomas, B. W., Campbell, A. M., and Woyak, N. (2021). The restaurant meal delivery problem: dynamic pickup and delivery with deadlines and random ready times. *Transportation Science*, 55(1):75–100. https://doi.org/10.1287/trsc.2020.1000.

U.S. Food and Drug Administration (2020). COVID-19 Frequently Asked Questions. https://www.fda.gov/emergency-preparedness-and-response/coronavirus-disease-2019-covid-19/covid-19-frequently-asked-questions#food.

Voccia, S. A., Campbell, A. M., and Thomas, B. W. (2019). The same-day delivery problem for online purchases. *Transportation Science*, 53(1):167–184. .

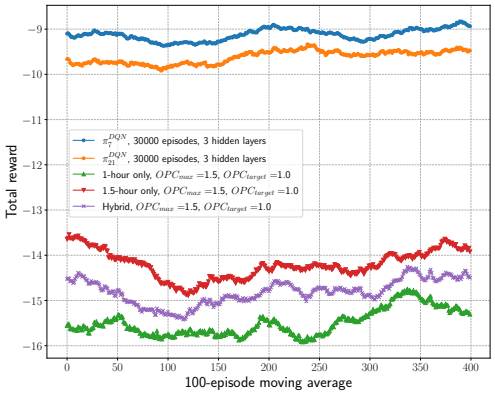
Wang, X., Agatz, N., and Erera, A. (2018). Stable matching for dynamic ride-sharing systems. *Transportation Science*, 52(4):850–867. .

Woschank, M., Rauch, E., and Zsifkovits, H. (2020). A review of further directions for artificial intelligence, machine learning, and deep learning in smart logistics. *Sustainability*, 12(9):3760. .

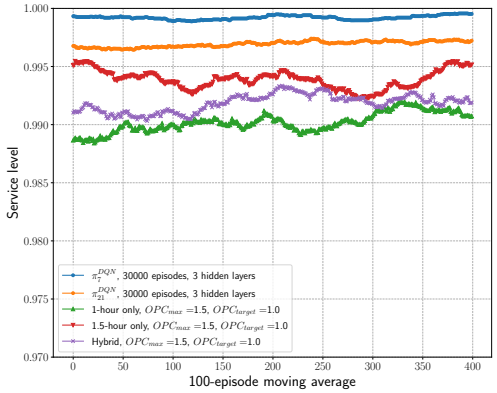
Yildiz, B. and Savelsbergh, M. (2019). Provably high-quality solutions for the meal delivery routing problem. *Transportation Science*, 53(5):1372–1388. https://doi.org/10.1287/trsc.2018.0887.

# Appendix. Performance results for different settings

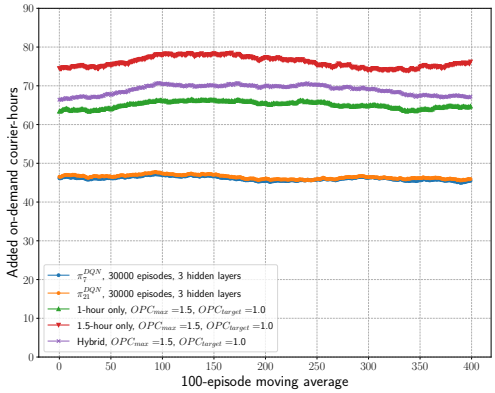
(1) $(N_{episodes}, N_{layers}) = (30000, 3), D_1 \sim \mathcal{U}[20, 30], D_{1.5} \sim \mathcal{U}[10, 20]$



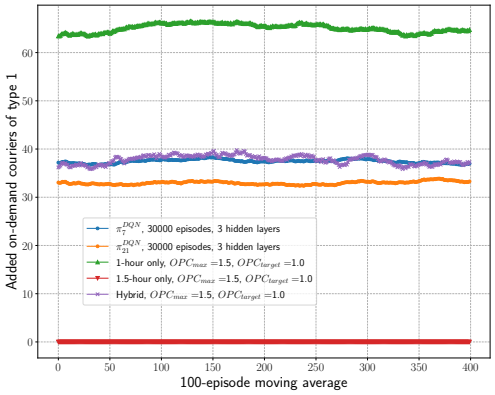

(a) Total reward



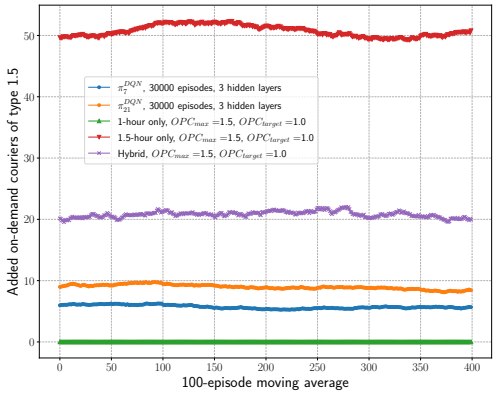

(b) Service level




(c) On-demand courier hours



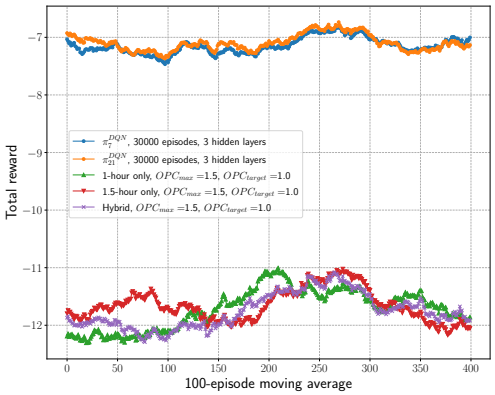

(d) On-demand 1-hour couriers



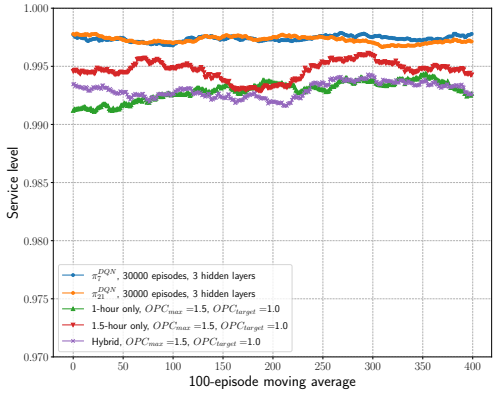

(e) On-demand 1.5-hour couriers

Figure 6: Policy benchmark over 500 test instances; 100-episode moving average

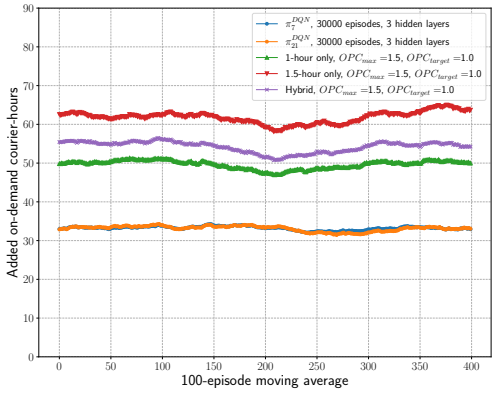
(2) $(N_{episodes}, N_{layers}) = (30000, 3), D_1 \sim \mathcal{U}[30, 40], D_{1.5} \sim \mathcal{U}[20, 30]$



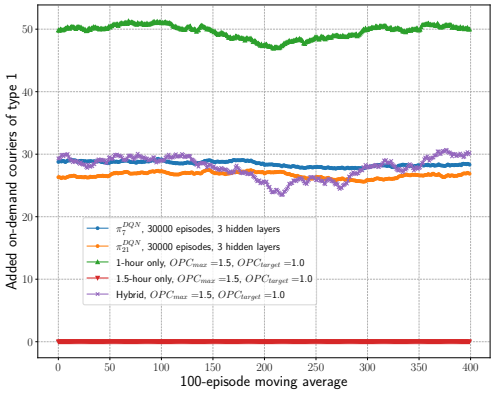

(a) Total reward



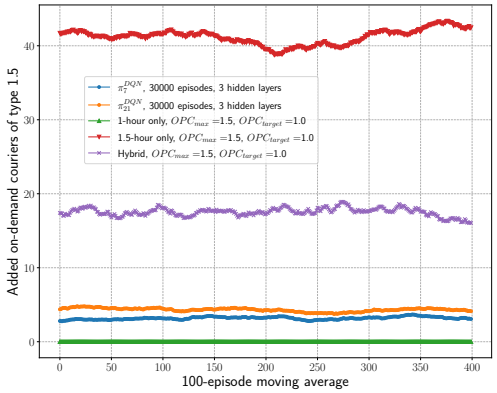

(b) Service level




(c) On-demand courier hours



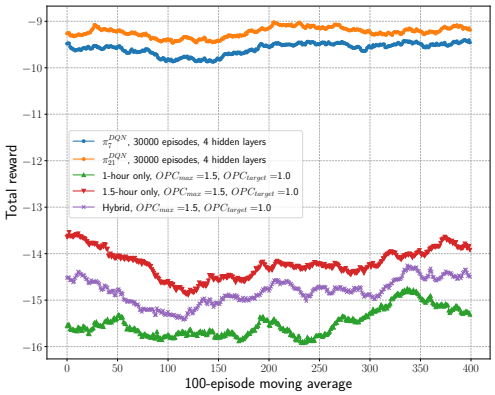

(d) On-demand 1-hour couriers



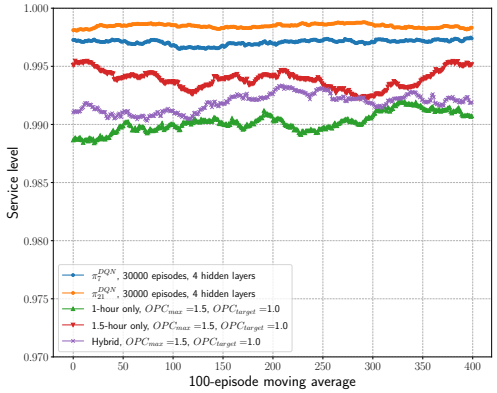

(e) On-demand 1.5-hour couriers

Figure 7: Policy benchmark over 500 test instances; 100-episode moving average

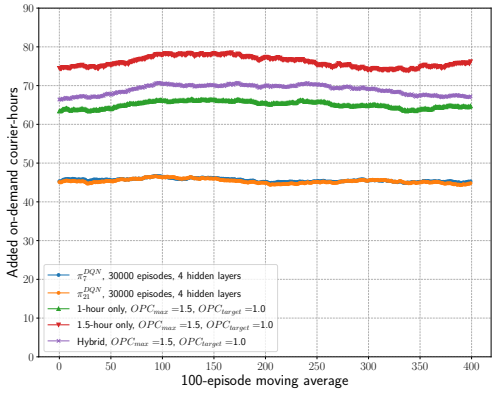
(3) $(N_{episodes}, N_{layers}) = (30000, 4), D_1 \sim \mathcal{U}[20, 30], D_{1.5} \sim \mathcal{U}[10, 20]$



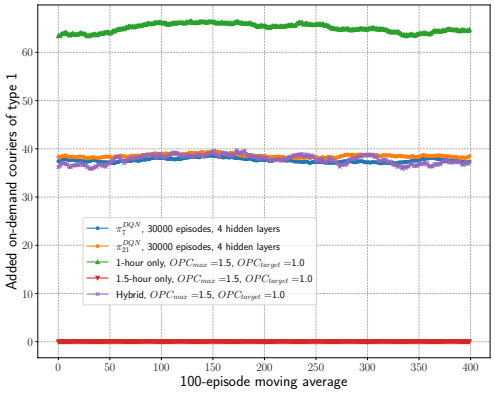

(a) Total reward



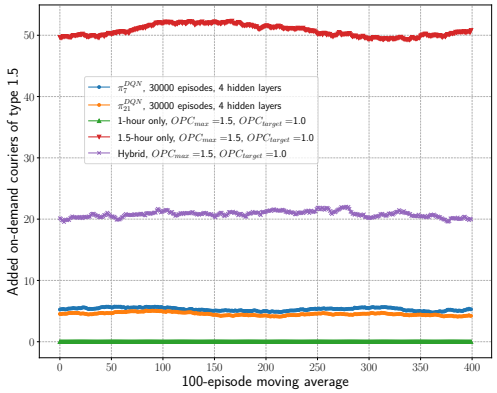

(b) Service level




(c) On-demand courier hours



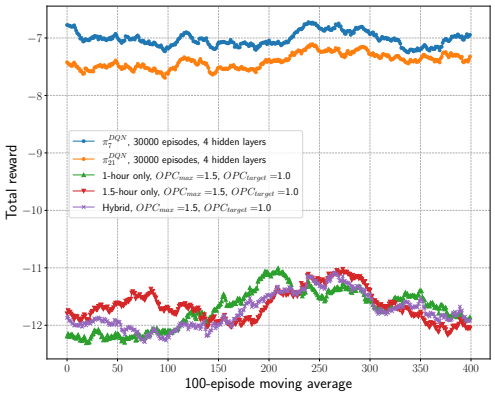

(d) On-demand 1-hour couriers



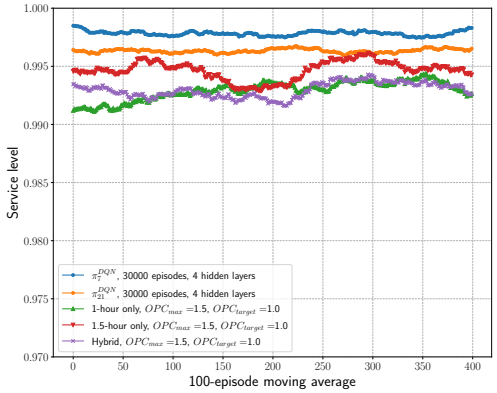

(e) On-demand 1.5-hour couriers

Figure 8: Policy benchmark over 500 test instances; 100-episode moving average

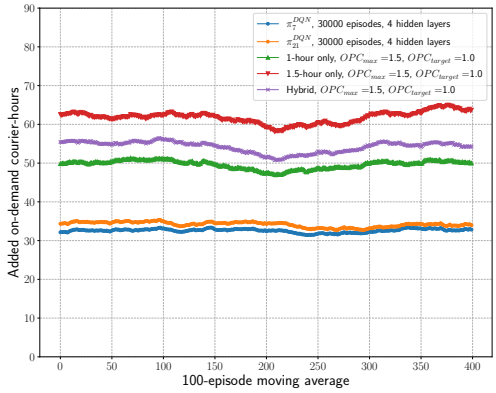
(4) $(N_{episodes}, N_{layers}) = (30000, 4), D_1 \sim \mathcal{U}[30, 40], D_{1.5} \sim \mathcal{U}[20, 30]$



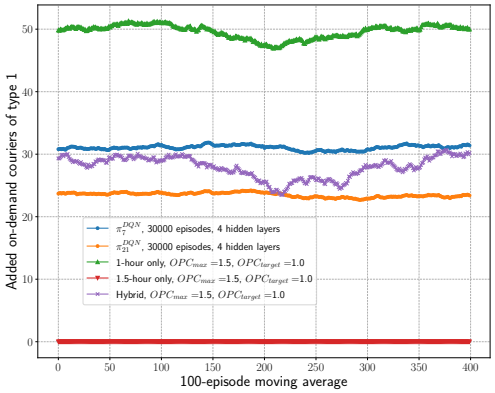

(a) Total reward



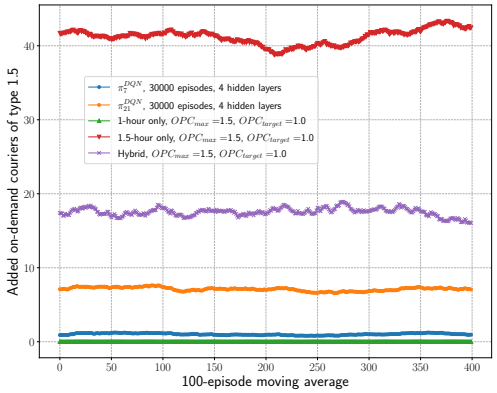

(b) Service level




(c) On-demand courier hours



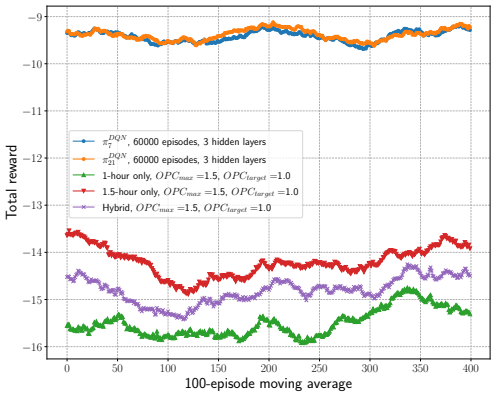

(d) On-demand 1-hour couriers



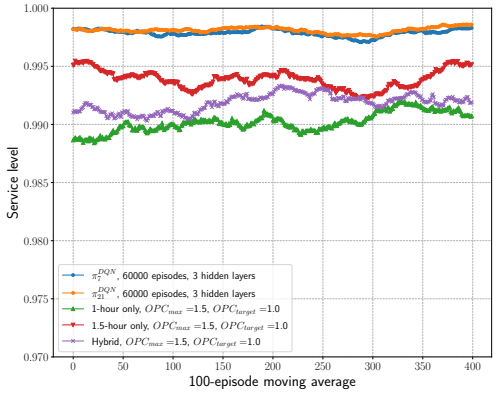

(e) On-demand 1.5-hour couriers

Figure 9: Policy benchmark over 500 test instances; 100-episode moving average

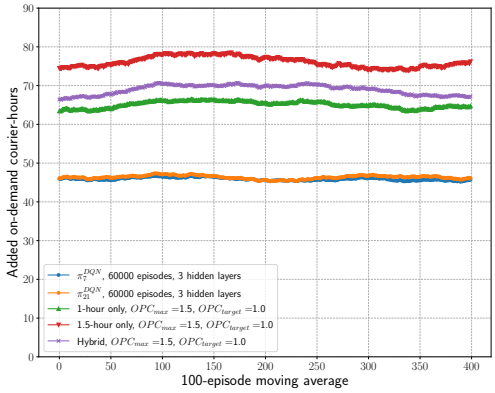
(5) $(N_{episodes}, N_{layers}) = (60000, 3), D_1 \sim \mathcal{U}[20, 30], D_{1.5} \sim \mathcal{U}[10, 20]$



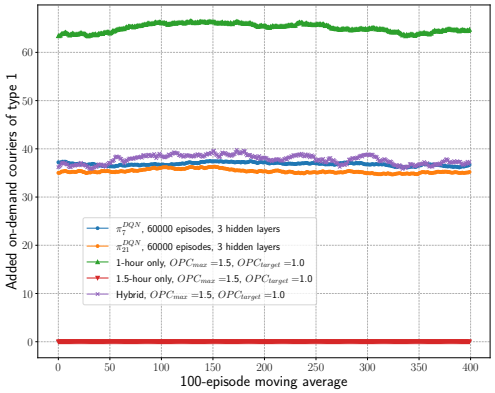

(a) Total reward



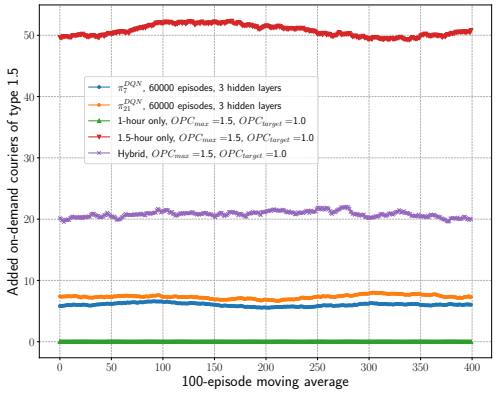

(b) Service level




(c) On-demand courier hours



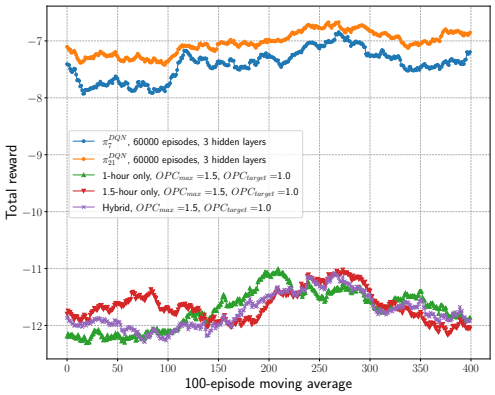

(d) On-demand 1-hour couriers



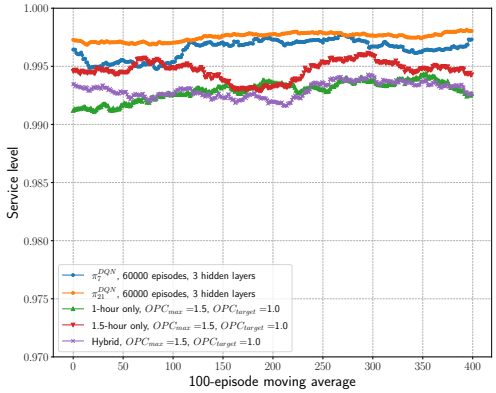

(e) On-demand 1.5-hour couriers

Figure 10: Policy benchmark over 500 test instances; 100-episode moving average

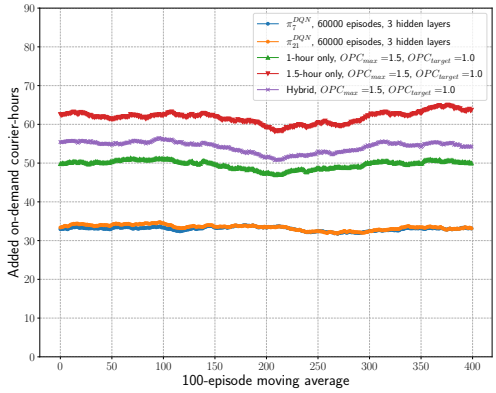
(6) $(N_{episodes}, N_{layers}) = (60000, 3), D_1 \sim \mathcal{U}[30, 40], D_{1.5} \sim \mathcal{U}[20, 30]$



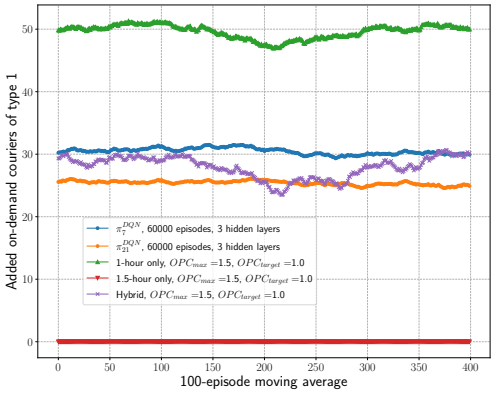

(a) Total reward



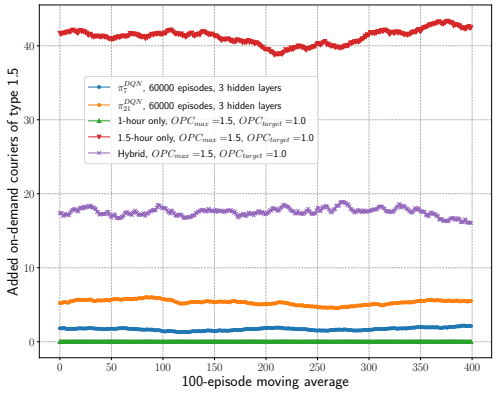

(b) Service level




(c) On-demand courier hours



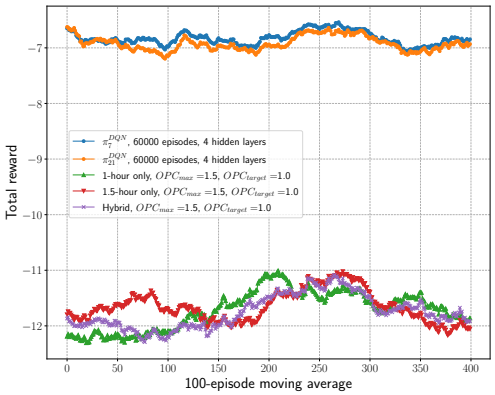

(d) On-demand 1-hour couriers



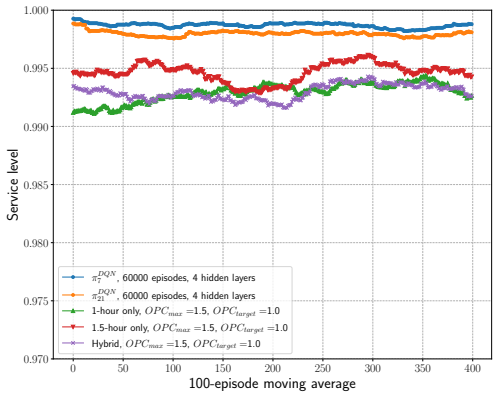

(e) On-demand 1.5-hour couriers

Figure 11: Policy benchmark over 500 test instances; 100-episode moving average

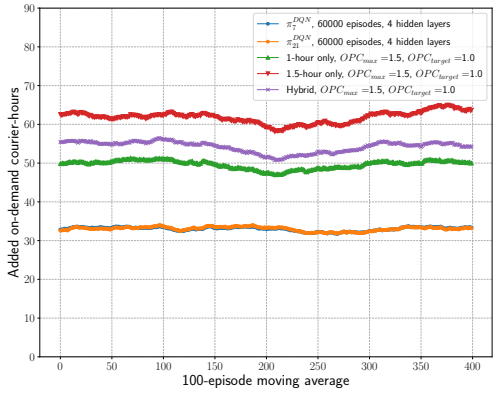
(7) $(N_{episodes}, N_{layers}) = (60000, 4), D_1 \sim \mathcal{U}[30, 40], D_{1.5} \sim \mathcal{U}[20, 30]$



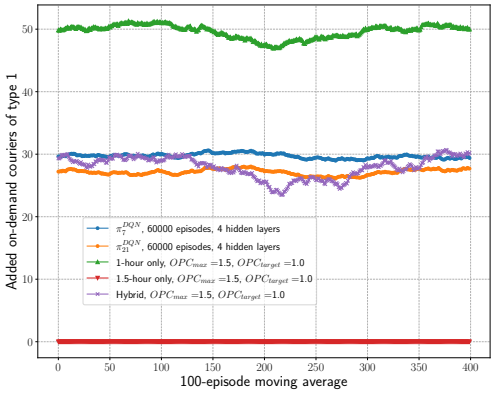

(a) Total reward



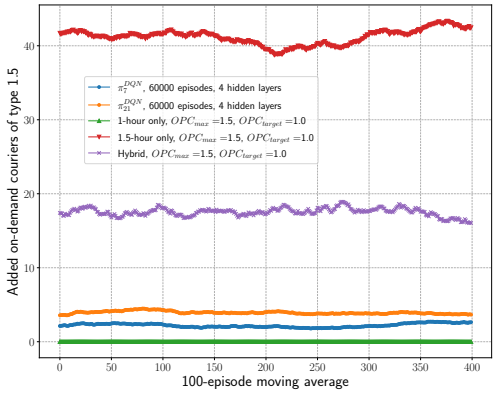

(b) Service level




(c) On-demand courier hours




(d) On-demand 1-hour couriers




(e) On-demand 1.5-hour couriers

Figure 12: Policy benchmark over 500 test instances; 100-episode moving average