AdaPool: A Diurnal-Adaptive Fleet Management Framework Using Model-Free Deep Reinforcement Learning and Change Point Detection

Marina Haliem, Vaneet Aggarwal, *Senior Member, IEEE*, and Bharat Bhargava, *Life Fellow, IEEE*

***Abstract*— This paper introduces an adaptive model-free deep reinforcement approach that can recognize and adapt to the diurnal patterns in the ride-sharing environment with car-pooling. Deep Reinforcement Learning (RL) suffers from catastrophic forgetting due to being agnostic to the timescale of changes in the distribution of experiences. Although RL algorithms are guaranteed to converge to optimal policies in Markov decision processes (MDPs), this only holds in the presence of static environments. However, this assumption is very restrictive. In many real-world problems like ride-sharing, traffic control, etc., we are dealing with highly dynamic environments, where RL methods yield only sub-optimal decisions. To mitigate this problem in highly dynamic environments, we (1) adopt an online Dirichlet change point detection (ODCP) algorithm to detect the changes in the distribution of experiences, (2) develop a Deep Q Network (DQN) agent that is capable of recognizing diurnal patterns and making informed dispatching decisions according to the changes in the underlying environment. Rather than fixing patterns by time of week, the proposed approach automatically detects that the MDP has changed, and uses the results of the new model. In addition to the adaptation logic in dispatching, this paper also proposes a dynamic, demand aware vehicle-passenger matching and route planning framework that dynamically generates optimal routes for each vehicle based on online demand, vehicle capacities, and locations. Evaluation on New York City Taxi public dataset shows the effectiveness of our approach in improving the fleet utilization, where less than 50% of the fleet are utilized to serve the demand of up to 90% of the requests, while maximizing profits and minimizing idle times.**

***Index Terms*— Ride-sharing, route planning, deep Q-networks, change point detection, non-stationary MDPs.**

I. Introduction

IN Q-LEARNING, there is a tight coupling between the learning dynamics (probability of choosing an action) and underlying execution policy (the effective rate of updating the Q value associated with that action). This coupling can cause performance degradation in dynamic noisy environments [2]. As the RL agent continues to build on its experiences in order to learn increasingly complex tasks, it should be able to quickly adapt while maintaining its acquired knowledge. However, once the i.i.d. assumption in the inherent Markov Decision Process (MDP) is violated, artificial neural networks have been shown to suffer from *catastrophic forgetting* [3], [4] due to erasing knowledge acquired from older data as the model gets trained on the new data. As an example, for an extraterrestrial rover mission, changes in the MDP may be a consequence of regular, predictable events such as intra-day or seasonal temperature variations, or may result from more complex phenomena that are difficult to predict (e.g., terrain changes due to wind) [5]. This paper proposes a novel approach to deal with the model changes in a model-free reinforcement learning setup.

We propose an adaptive model-free deep learning framework for ride-sharing with car-pooling that can learn different underlying contexts of the environments. Deep reinforcement learning methodologies are used for this adaptive modeling where transition probabilities are computed through Deep Q Neural Network (DQN). We utilizes the dispatch of idle vehicles using a Deep Q-learning (DQN) framework as in DeepPool [6], and we add the profit term in the reward function so that the output expected discounted rewards (Q-values) associated with each action, becomes a good reflection of the expected earnings gained from performing this action. To the best of our knowledge, our AdaPool framework is the first work that introduces an adaptive model-free approach for distributed matching and dispatching where agents are able to recognize various diurnal patterns, learn their corresponding models, detects the change points and adapts accordingly. Thus, influencing the decision making of ride-sharing platforms. We identify the following as our major contributions:

- We propose an adaptive model-free RL algorithm for handling non-stationary environments, where we adapt Deep Q-learning to learn optimal policies for different environment models. The proposed approach, in training, finds the set of models that divide the time-varying MDPs based on diurnal patterns, and uses the appropriate model to make decisions. To the best of our knowledge, this is the first work for model-free reinforcement learning that

Manuscript received December 10, 2020; revised March 31, 2021 and June 12, 2021; accepted August 27, 2021. Date of publication October 8, 2021; date of current version March 9, 2022. This work was supported in part by the National Science Foundation under Grant CNS-1618335, in part by the Defense Advanced Research Projects Agency (DARPA) under Award W911NF2020003, and in part by Cisco Systems, Inc. (CISCO). This work was presented in part at ACM BuildSys 2020 [1] [DOI: 10.1145/3385958.3430484]. The Associate Editor for this article was T.-H. Kim. *(Corresponding author: Vaneet Aggarwal.)*

Marina Haliem and Bharat Bhargava are with the Department of Computer Science, Purdue University, West Lafayette, IN 47907 USA (e-mail: mwadea@purdue.edu; bbshail@purdue.edu).

Vaneet Aggarwal is with the School of Industrial Engineering and the School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907 USA (e-mail: vaneet@purdue.edu).

This article has supplementary downloadable material available at https://doi.org/10.1109/TITS.2021.3109611, provided by the authors.

Digital Object Identifier 10.1109/TITS.2021.3109611

works for training policies accounting for diurnal patterns in any area.

- Using change point detection, the proposed algorithm switches between the MDP policies, and estimates policy for the new model or improves the policy learnt, if the model had been previously experienced. In this manner, our method avoids *catastrophic forgetting* [3], and utilizes the Q-values learnt using DQN for making the dispatch decisions.
- We propose a dynamic, demand-aware matching and route planning framework, that is scalable up to the maximum capacity per vehicle in the initial assignment phase. In the optimization phase, this algorithm takes into account the near-future demand in order to improve the route-planning by eliminating rides heading towards opposite directions and applying insertion operations to vehicles' current routes.
- Through experiments using real-word dataset of New York City's taxi trip records [7] (15 million trips), we simulate the ride-sharing system.[1] We show that the optimization problem of our novel AdaPool framework is formulated such that it enhances the overall acceptance rate, increases the profit margins of the fleet, minimizes the extra travel distance and the average idle time, when compared to non-adaptive RL approaches.

The rest of this paper is organized as follows: Section II describes the overall architecture of our adaptive deep RL framework: AdaPool. In Section III, we explain our dynamic, demand-aware approach for matching and route planning. Section IV, explains our adaptive DQN-based approach for dispatching vehicles. Simulation setup as well as experimental results are presented in Section V. Detailed related work is presented in Appendix A. Finally, Section VI concludes the paper with discussion on future directions.

Abbreviations and Acronyms: MoD: Mobility-on-Demand. RL: Reinforcement Learning. DQN: Deep Q-Network. ODCP: Online-parametric Dirichlet Change Point. Conv-Net: Convolutional Neural Network. OSRM: Open Source Routing Machine. MDP: Markov Decision Process.

II. AdaPool: Adaptive Matching and Dispatching Framework Using Deep RL

In this section, we provide details about each component in our model architecture, as well as explanations of model parameters and notations. We propose a novel adaptive framework for matching and dispatching in ride-sharing environments with car-pooling using DQN, where initial matchings (that are decided in a greedy fashion) are then optimized in a distributed manner (per vehicle) in order to meet the vehicle's capacity constraints as well as minimize customers' extra waiting time and driver's additional travel distance. We consider the scenario where the environment changes between models $1, 2, \cdots, k$ in a cyclic manner, dynamically as shown in Fig. 1. With the cyclic nature, the environment changes

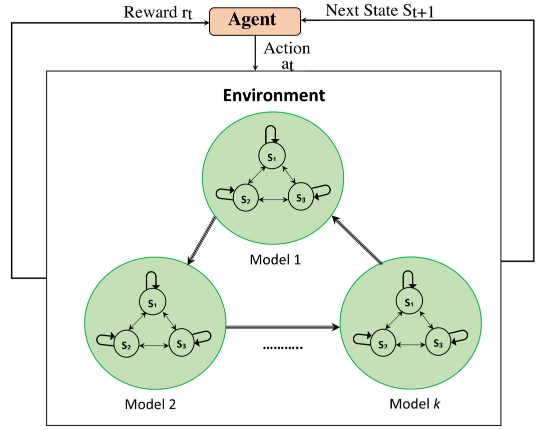


Fig. 1. Time varying MDPs in non-stationary environment.

from model n to model 1. Such cyclic repetition allows for learning the k models and use them in the run-time, where on detecting the change-point, next model can be used. The implication of the non-stationary environment is this: when the agent exercises a control a_t at time t, the next state s_{t+1} as well as the reward r_t are functions of the *active* environment model dynamics. In our approach, we assume that there are k environment models $M_1, M_2, \cdots, M_k$, through which the system cycles. However, neither the context information (or model parameters) of each model nor the change points $T_1, T_2, \cdots$ (when these model changes occur), are known to the RL agent. In this case, the agent can collect experience tuples while simultaneously following a model-free learning algorithm to learn an approximately optimal policy. Instead of assuming any specific structure, our model-free approach learns the Q-values dynamically using convolutional neural networks. Our method works in tandem with a change point detection algorithm, to get information about the changes in the environment. Then, it updates Q-values of the relevant model whenever a change is detected and does not attempt to estimate the transition and reward functions for the new model. Additionally, if the method finds that samples are obtained from a previously observed model, it updates the Q values corresponding to that model. Thus, in this manner, the information which was learnt and stored earlier (in the form of Q-values) is not lost.

Moreover, each vehicle learns the best future dispatch action to take at time step t, taking into consideration the locations of all other nearby vehicles, but without anticipating their future decisions. Vehicles' dispatch decisions are made in parallel, since it is unlikely for two drivers to take actions at the same exact time since drivers know the location updates of other vehicles in real time (e.g., GPS). Therefore, our algorithm learns the optimal policy for each agent independently as opposed to centralized-based approaches such as in [8].

A. Model Architecture

Figure 2 shows the basic components of our joint framework and the interactions between them. We assume that the control unit is responsible for: (1) making the initial matching decisions, based on the proximity of vehicles to ride requests,

[1]The code for this work is available at https://github.com/marina-haliem/AdaPool.

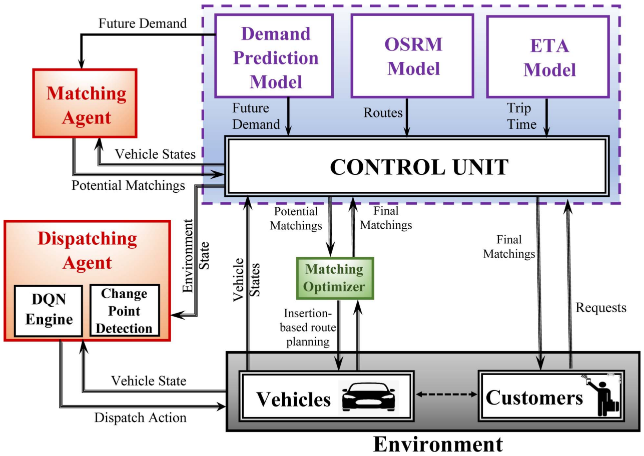


Fig. 2. Overall architecture of AdaPool framework.

(2) maintaining the states including current locations, current capacity, destinations, etc., for all vehicles. These states are updated in every time step based on the dispatching and matching decisions. (3) Control unit also has some internal components that help manage the ride-sharing environment such as: (a) the estimated time of arrival (ETA) model used to calculate and continuously update the estimated arrival time. (b) The Open Source Routing Machine (OSRM) model used to generate the vehicle's optimal trajectory, using [9], to reach a destination, and (c) the (Demand Prediction) model used to calculate the future anticipated demand in all zones. We adopt these three models from [8], [10] since they play a crucial role in enabling this framework; their details are provided in Appendix C. First, the ride requests are input to the system along with the heat map for supply and demand (which involves demand prediction in the near future). Then, based on the predicted demand, vehicles adopt a dispatching policy using DQN, where they get dispatched to zones with anticipated high demand. This step not only takes place when vehicles first enter the market (lines 3-5 in Algorithm 1), but also when they experience large idle durations (at the end of every time step, this gets checked for in lines 13-14, Algorithm 1). The control unit performs the initial vehicle-passenger(s) matching where each vehicle gets assigned all potential (one or more) requests in its vicinity. Then, each vehicle executes its matching optimizer module that performs an insertion-based route planning. In this step, vehicles reach their final n requests by dealing with their initial matchings list in the order of their proximity, performing an insertion operation to their current route plan (as long as this insertion satisfies the capacity, extra waiting time, and additional travel distance constraints to guarantee that serving this request would yield a profit).

Using the "Change Point Detection" module, vehicles are able to learn and switch between models $M_1, M_2, \cdots, M_k$ dynamically whenever a change is detected. Finally, a vehicle communicates with the control unit, as needed, to request new information of the environment (prior to making decisions) or update its own status (after any decision). Algorithm 1 presents the overall flow of our framework.

Algorithm 1 AdaPool Framework

1: **Initialize** vehicles' states X_0 at t_0.
2: **for** $t \in T$ **do**
3: **Fetch** all vehicles that entered the market in time slot t, V_{new}.
4: **Initialize** Vehicles' routes $S_{V_j} \leftarrow$ empty for each $V_j \in V_{new}$
5: **Dispatch** V_{new} to zones with anticipated high demand (Algo. 3).
6: **Fetch** all ride requests at time slot t, D_t.
7: **Fetch** all available vehicles at time slot t, V_t.
8: **for** each vehicle $V_j \in V_t \dots$ **do**
9: **Obtain** initial matching A_j using Algorithm 5.
10: **Perform** route planning $S_{V_j} \leftarrow$ GREEDY_INSERTION(A_j, S_{V_j})
11: **Retrieve** next stop from S_{V_j}.
12: **Head** to next stop (whether a pickup or a dropoff).
13: **Fetch** all idle vehicles with Idle_duration > 10 minutes, V_{idle}.
14: **Dispatch** V_{idle} to zones with anticipated high demand (Algo. 3).
15: **Update** the state vector s_t.
16: **procedure** GREEDY_INSERTION(A_j, S_{V_j})
17: **Initialize** $V^j_{\text{capacity}} = V^j_C$
18: **while** $V^j_{\text{capacity}} < C^{V_j}_{max}$ **do**
19: **for** each ride request $r_i \in A_j \dots$ **do**
20: **if** $(V^j_{\text{capacity}} + |r_i|) \leq C^{V_j}_{max}$ **then**
21: **Obtain** $(S'_{V_j}, \text{cost}(V_j, S'_{V_j})) \leftarrow$ ROUTE_PLANNING (V_j, S_{V_j}, r_i) using Algo. 2.
22: min_cost $\leftarrow min_{(r_i \in A_j)}(\text{cost}(V_j, S'_{V_j}[r_i]))$.
23: $r^* \leftarrow argmin_{(r_i \in A_j)}(\text{cost}(V_j, S'_{V_j}[r_i]))$.
24: **Update** trip time T_i based on $S'_{V_j}[r^*]$ using ETA model.
25: **Update** $S_{V_j} \leftarrow S'_{V_j}[r^*]$
26: **Increment** $V^j_{\text{capacity}} \leftarrow V^j_{\text{capacity}} + |r^*|$
27: **Remove** r^* from A_j
28: **if** A_j is *empty* **then**
29: ***break***
30: **Update** the state vector s_t.
31: **end procedure**

B. Model Parameters and Notations

We built a ride-sharing simulator to train and evaluate our framework. We simulate New York City as our area of operation, where the map is divided into multiple non-overlapping regions, a grid with each 1 square mile being taken as a zone. This allows us to discretize the area of operation and thus makes the action space —where to dispatch the vehicles—tractable. We use $m \in \{1, 2, 3, \dots, M\}$ to denote the city's zones, and n to denote the number of vehicles. A vehicle is marked as *available* if there is remaining seating capacity. Vehicles that are completely full or are not considering taking passengers are marked *unavailable*. Available vehicles in zone i at time slot t is denoted $v_{t,i}$. Only available vehicles are eligible to be dispatched. We optimize our algorithm over T time steps, each of duration Δt. The vehicles make decisions on where on the map to head-to to serve the demand at each time step $\tau = t_0, t_0 + \Delta t, t_0 + 2(\Delta t), \dots, t_0 + T(\Delta t)$ where t_0 is the start time. Below, we present the model parameters and notations:

- *Demand:* We denote the number of requests for zone m at time t as $d_{t,m}$. The future pick-up request demand in each zone is predicted through a historical distribution of trips across the zones [11], and is denoted by $D_{t:T} = (\overline{\boldsymbol{d}}_{\boldsymbol{t}}, \dots, \overline{\boldsymbol{d}}_{\boldsymbol{t+T}})$ from time t to $t + T$.
- *Supply:* At each time slot t, the supply of vehicles for each zone is projected to future time $\tilde{t}$. $d_{t,\tilde{t},m}$ is

the number of vehicles that are currently unavailable at time t but will become available at time $\tilde{t}$ as they will drop-off customer(s) at region m. This information can be ascertained using the ETA [6], [10] prediction for all vehicles. Consequently, for a set of dispatch actions at time t, we can predict the number of vehicles in each zone for T time slots ahead, from time t to time $t+T$, denoted by $V_{t:t+T}$ which serves as our predicted supply in each zone for T time slots ahead.

- *Vehicle Status:* We use $X_t = \{x_{t,1}, x_{t,2}, \ldots, x_{t,N}\}$ to denote the N vehicles' status at time t. $x_{t,n}$ tracks vehicle n's state variables at time step t including, (1) current location/zone V_{loc}, (2) current capacity V_C, (3) type V_T, (4) pickup time of each at passenger, (5) the destination of each passenger, and (6) the earnings till time t. A vehicle is considered available, if and only if $V_C < C^V_{max}$.

These variables change in real time according to the environment variations and demand/supply dynamics. However, our framework keeps track of all these rapid changes and seeks to make the demand, d_t, $\forall t$ and supply v_t, $\forall t$ close enough (i.e., mismatch between them is zero). Combining all this data, we define the state space, action space and reward function for our DQN agents:

1) *State Space:* we have defined a three tuple that captures the environment updates at time t to represent our state space as $s_t = (X_t, V_{t:t+T}, D_{t:t+T})$. When a set of new ride requests arrive at the system, we can retrieve from the environment all the state elements, combined in one vector s_t. Also, when a passenger's request becomes assigned, we append the customer's expected pickup time, source, destination and ride fare to s_t as well.
2) *Action Space:* a^n_t denotes the action taken by vehicle n at time step t. In our simulator, the vehicle can move (vertically or horizontally) at most 7 cells, and hence the action space is limited to these cells. A vehicle can move to any of the 14 vertical (7 up and 7 down) and 14 horizontal (7 left and 7 right). This results in a 15×15 action space $a_{t,n}$ for each vehicle as a vehicle can move to any of these cells or it can remain in its own cell. After the vehicles decides on which cell to go to using DQN (Section IV-A), it uses the shortest optimal route to reach its next stop.
3) *Reward:* Having explained all of the above factors, at every time step t, the DQN agent obtains a representation for the environment, s_t, and a reward r_t. Based on this information, the agent takes an action that directs the vehicle (that is either idle or recently entered the market) to different dispatch zone where the expected discounted future reward is maximized, i.e., $\sum_{j=t}^{\infty} \eta^{j-t} r_j(a_t, s_t)$, where $\eta < 1$ is a time discount factor. In our algorithm, we define the reward r_k as a weighted sum of different performance components that reflect the objectives of our DQN agent (explained in Section IV-C). The reward will be learnt from the environment for individual vehicles and then leveraged to optimize their decisions.

We note that this discount factor is a key that makes the change of model slow to learn. Thus, we use change point detection to learn the change and use the appropriate model. A detailed table of notations is provided in Appendix B.

III. Dynamic Demand-Aware Matching and Route-Planning Framework

This section provides details of our dynamic, demand-aware approach to solve the NP-Hard matching and route planning problems in ride-sharing environments. Our framework goes through two phases as explained in this section.

NP-Hardness: The ride-sharing assignment problem is proven to be NP-hard in [12] as it is a reduction from the 3-dimensional perfect matching problem (3DM). The authors of [12] provided an approximation algorithm that is 2.5 times the optimal cost for the case where at most two requests can share the same vehicle at a time. However, our approach is not limited to at most two requests per vehicle. The proposed approach is a different from that in [13] where pricing was used to dis-incentivize matching passengers going in opposite directions, while we will use a greedy approach in adding the customers to the route.

A. Initial Vehicle-Passenger(s) Assignment Phase

Note that the control unit for decision making knows the future demand $D_{t:t+T}$ at each zone, the vehicles' status vectors X_t including their current locations and destinations as well as the origin o_i and destination d_i locations for each request r_i. Each vehicle is assigned to up to 50 requests r_i in its vicinity (to reduce the computational power needed), that could potentially get served by it. At the end of this phase, each vehicle V_j has a list of initial matchings $A_j = [r_1, r_2, \ldots, r_k]$, where $k \le 50$ (Pseudo-code for this phase is given in Appendix E, Algorithm 5.

B. Optimization Phase: Greedy Insertion Cost

In this phase, we follow the idea of searching each route and locally optimally inserting new vertex (or vertices) into a route. In our problem, there are two vertices (i.e., origin o_i and destination d_i) to be inserted for each request r_i. We define the insertion operation as: given a vehicle V_j with the current route S_{V_j}, and a new request r_i, the insertion operation aims to find a new feasible route S'_{V_j} by inserting o_i and d_i into S_{V_j} with the minimum increased cost, that is the minimum extra travel distance, while maintaining the order of vertices in S_{V_j} unchanged in S'_{V_j}.

Specifically, for a new request r_i, the basic insertion algorithm checks every possible position to insert the origin and destination locations and return the new route such that the incremental cost is minimized. So, the vehicle reaches its final matchings (denoted M_j) list by greedily picking the top k' requests (where $k' \subset k$) with the minimal insertion cost while $k' \le C^{V_j}_{max}$ to satisfy its capacity constraint (see lines 16-24 in Algorithm 1). Assume the passenger count per request is $\mid r_i \mid$, and the vehicle V_j arrives at location z. Then, to check the capacity constraint in $O(1)$ time, we define vehicle V_j's current capacity $V^j_C[z]$ which refers to the total capacity of the requests that are still on the route of V_j when it arrives at that location z as follows:

$$V_C^j[z] = \begin{cases} V_C^j[z-1] + \mid r_i \mid & \text{if } z == o_i \\ V_C^j[z-1] - \mid r_i \mid & \text{if } z == d_i. \end{cases}$$

To present our cost function, we first define our distance metric, where given a graph G, we use our OSRM engine to pre-calculate all possible routes over our simulated city. Then, we derive the distances of the trajectories (i.e., paths) from location a to location b to define our graph weights. Thus, we obtain a weighted graph G with realistic distance measures serving as its weights. We extend the weight notation to paths as follows: $w(a_1, a_2, \ldots, a_n) = \sum_{i=1}^{n-1} w(a_i, a_{i+1})$. Thus, we define the cost associated with each new potential route/path $S'_{V_j} = [r_i, r_{i+1}, \ldots, r_k]$ to be the cost$(V_j, S'_{V_j}) = w(r_i, r_{i+1}, \ldots r_k)$ resulting from this specific ordering of vertices (origin and destination locations of the k requests assigned to vehicle V_j). The full insertion-based algorithm is presented in Algorithm 2.

Algorithm 2 Insertion-Based Route Planning

Input: Vehicle V_j, its current route S_{V_j}, a request $r_i = (o_i, d_i)$ and weighted graph G with pre-calculated trajectories using OSRM model.
Output: Route S'_{V_j} after insertion, with minimum cost(V_j, S'_{V_j}).
procedure ROUTE_PLANNING(V_j, S_{V_j}, r_i)
  if S_{V_j} is *empty* **then**
    $S'_{V_j} \leftarrow [loc(V_j), o_i, d_i]$.
    cost$(V_j, S'_{V_j}) = w(S'_{V_j})$.
    Return S'_{V_j}, cost(V_j, S'_{V_j})
  Initialize $S''_{V_j} = S_{V_j}$, $Pos[o_i] =$ NULL, cost$_{min} = +\infty$.
  for each x in 1 to $|S_{V_j}|$ **do**
    $S^x_{V_j}$:= **Insert** o_i at $x-th$ in S_{V_j}.
    Calculate cost$(V_j, S^x_{V_j}) = w(S^x_{V_j})$.
    if cost$(V_j, S^x_{V_j}) <$ cost$_{min}$ **then**
      cost$_{min} \leftarrow$ cost$(V_j, S^x_{V_j})$.
      $Pos[o_i] \leftarrow x$, $S''_{V_j} \leftarrow S^x_{V_j}$.
  $S'_{V_j} = S''_{V_j}$, cost$_{min} = +\infty$.
  for each y in $Pos[o_i] + 1$ to $|S''_{V_j}|$ **do**
    $S^y_{V_j}$:= **Insert** d_i at $y-th$ in S''_{V_j}.
    Calculate cost$(V_j, S^y_{V_j}) = w(S^y_{V_j})$.
    if cost$(V_j, S^y_{V_j}) <$ cost$_{min}$ **then**
      cost$_{min} \leftarrow$ cost$(V_j, S^y_{V_j})$.
      $S'_{V_j} \leftarrow S^y_{V_j}$, cost$(V_j, S'_{V_j}) \leftarrow$ cost$_{min}$.
  Return S'_{V_j}, cost(V_j, S'_{V_j})
end procedure

After vehicle V_j picks the route S'_{V_j} that has the minimum cost of inserting r_y into its current route, it follows the same procedure for every $r_i \in A_j$ as shown in *Greedy_Insertion* procedure in Algorithm 1. Repeatedly picking r^* that is the *argmin* of the minimum insertion cost among all potential requests in A_j, each vehicle ends up with k' final matchings that guarantees the optimal routing for the vehicle while serving all k' requests and satisfying its capacity constraint.

Finally, this phase works in a distributed fashion where each vehicle picks the top k' requests that minimizes its travel cost, following Algorithm 2, and satisfying its capacity constraint by following the *Greedy_Insertion* procedure in Algorithm 1.

Complexity Analysis: We note that the routes pre-calculation step done using our OSRM engine (with their associated costs), provides us with fast routing and constant-time computation $O(1)$, thus reducing the complexity of our algorithm from $O(n^3)$ to $O(n^2)$. In addition, we adopt the approach proposed in [13] for checking the route feasibility in $O(1)$ time to further reduce the computation needed (details provided in Appendix E.

IV. Adaptive DQN Dispatching Approach

In this section, we present our distributed adaptive approach for dispatching vehicles. This framework aims at re-balancing vehicles over the city to better serve the demand while accounting for the different diurnal patterns during the day. Utilizing DQNs along with a change point detection algorithms, individual agents (i.e., vehicles) are able to learn different underlying models of the environment that correspond to the different demand patterns and switch between them according to the observed state of the environment. We utilize a reinforcement learning framework, with which we can learn the probabilistic dependence between vehicle actions and the reward function thereby optimizing our objective function. We utilize this framework in order to re-balance vehicles over the city to better serve the demand. The fleet of autonomous vehicles were trained in a virtual spatio-temporal environment that simulates urban traffic and routing. In our simulator, we used the road network of the New York City Metropolitan area along with a realistic simulation of taxi pick-ups. This simulator hosts each deep reinforcement learning agent which acts as a delivery vehicle in the New York City area that is looking to maximize its reward defined by Eq. (3). The learning begins by obtaining experience tuples E_t according to the dynamics and reward function of current active model M_{θ_c}. The state and reward obtained are stored as experience tuples, since model information is not known.

A. Distributed Adaptive DQN

At every time step t, our adaptive DQN performs the change point detection algorithm described in Section IV-B. If it receives T^* signalling that a change has been detected, it increments the counter c and starts switching from its current model and updates (and takes action based on) a new model, it does not attempt to estimate the transition and reward functions for the new model. Instead, it starts to update the dynamics of this new model, where the Q values are updated. The full algorithm for this approach is in Algorithm 3, where as assume the knowledge of a pattern of change as in line 1, but without the knowledge of the context information of each model.

At every time step t, the DQN agent obtains a representation for the environment, $s_{t,n}$, and calculates a reward r_t associated with each dispatch-to location in the action space $a_{t,n}$ according to the dynamics and reward function of current active model M_{θ_c}, and updates Q-values of the relevant model. Based on the rewards associated with each cell of the vehicle's action space explained in Section II-B, the agent takes an action that directs the vehicle to different dispatch zone where

Algorithm 3 Distributed Dispatching Using Adaptive DQN

1: **Input 1:** Model Change Pattern, $M_{\theta_1} \rightarrow M_{\theta_2}, M_{\theta_2} \rightarrow M_{\theta_3}, \ldots, M_{\theta_{k-1}} \rightarrow M_{\theta_k}$, where $M_{\theta_i} \in \{M_1, M_2, \ldots, M_k\}$, and $\theta \in \Theta = \{1, 2, \ldots, k\}$
2: **Input 2:** $X_t, V_{t:t+T}, D_{t:t+T}$.
3: **Output:** Dispatch Decisions
4: **Fix** learning rate σ
5: **Initialize** context number, c = 1, Q values $Q(m, s, a) = 0, \forall\ m \in\ 1, \ldots, k$ $\forall (s, a) \in S$ x A.
6: **Construct** a state vector $s_{t,n} = (X_t, V_{t:t+T}, D_{t:t+T})$.
7: **Get** the best dispatch action $a_{t,n} = argmax[Q(s_{t,n}, a; \theta_c)]$ for all vehicles V_n using the Q-network of model M_{θ_c}.
8: **Get** the destination zone $Z_{t,j}$ for each vehicle $j \in V_n$ based on action $a_{t,j} \in a_{t,n}$
9: **Get** reward $r_{t,n}$ using Eq. 3 associated with model M_{θ_c}.
10: **Update** Q value associated with model M_{θ_c} (explained in Appendix H.
11: **Obtain** next state $s_{t+1,n}$ according to the environment dynamics.
12: $e_t \leftarrow (s_{t,n}, r_{t,n}, s_{t+1,n})$
13: **Update** dispatch decisions by adding $(j, Z_{t,j})$
14: $\tau \leftarrow$ DCP $(e_T = e_1, \ldots e_t)$, where T includes all $t \geq \tau^*$ at which model M_{θ_c} was active.
15: **if** τ is not Null **then**
16: **Increment** $c = mod\ (c + 1, k)$.
17: **if** $c == 0$ **then**
18: $c = k$
19: $\tau^* \leftarrow \tau$
20: **Return** $(n, Z_{t,n})$

the expected discounted future reward is maximized. In our algorithm, we define the reward r_k as a weighted sum of different performance components that reflect the objectives of our DQN agent (explained in Section IV-C). The architecture of our DQN is described in Appendix G.

The reward will be learnt from the environment for individual vehicles and then leveraged by the agent/optimizer to optimize its decisions. Through learning the probabilistic dependence between the action and the reward function that is explained further in Appendix H, we learn the Q-values according to the dynamics and reward function of current active model M_{θ_c} associated with the probabilities $P(r_t \mid a_t, s_t)$ over time by feeding the current states of the system. The Q-values are then used to decide on the best dispatching action to take for each individual vehicle. Looking at Figure 1, the DQN agent starts by learning model 1, where c = in line 5 of Algorithm 3. At each time step t, it receives the 3-tuple representation of the environment, calculates the reward (Q-values) according to the dynamics of that active model, and makes dispatch decisions accordingly (see lines 7 - 11) by picking the actions that yields the maximum expected discounted reward (Q-value). Besides, the agent stores experience tuples E_t, at each time step t, that consists of current state s_t, reward r_t, and next state s_{t+1} as shown in line 12. Further, after the learning step, the agent checks for change points using the ODCP algorithm (line 14) explained in Section IV-B. Once it detects a change, that is when the ODCP algorithm (Algorithm 4) returns T^*, it switches to next model $c + 1$ and continues its learning using the dynamics of the new active model (lines 14 - 19). Note that if the samples observed come from a model (i.e., policy) that has been learnt before, the DQN agent updates the Q-values of that previously seen model and continues learning building on its previous experience that is associated to that model. Also, after learning (where no new model is learnt), the different learnt models can be exploited along with change point detection for recognizing diurnal patterns.

Algorithm 4 Dirichlet Change Point Detection Algorithm

Input Time Window $[1 \ldots T]$, Data $[x_1 \ldots x_T]$.
Output T^*: Change Point (if there is a change).
procedure DCP($[x_1 \ldots x_T]$)
$Q_0 \leftarrow$ **Estimate** Drichlet Parameters for $[x_1 \ldots x_T]$ using Eq. 1
$LL_0 \leftarrow$ **Estimate** Log-Likelihood for $[x_1 \ldots x_T]$ under Q_0 (Eq. 2).
$(T^*, LL^*) \leftarrow$ ESTIMATE_2WINDOW($[x_1 \ldots x_T]$)
$Z^* \leftarrow LL^* - LL_0$
if $Z^* >$ threshold **then**
Return Change point at T^*.
else
No change, **Return**
end procedure
procedure ESTIMATE_2WINDOW($[x_1 \ldots x_T]$)
for $t \in\ 1 \ldots T - 1$ **do**
$Q_1 \leftarrow$ **Estimate** Drichlet Parameters for $[x_1 \ldots x_t]$ (Eq. 1).
$Q_2 \leftarrow$ **Estimate** Drichlet Parameters for $[x_{t+1} \ldots x_T]$ (Eq. 1).
$LL_t \leftarrow$ Log-Likelihood for $[x_1 \ldots x_t]$ under Q_1 + Log-Likelihood for $[x_{t+1} \ldots x_T]$ under Q_2 (Eq. 2).
$LL^* \leftarrow max_{(t \in\ 1 \ldots T-1)} LL(t)$
$T^* \leftarrow argmax_{(t \in\ 1 \ldots T-1)} LL(t)$
Return (T^*, LL^*)
end procedure

B. Online Dirichlet Change Point Detection

To detect points of change, our DQN agents analyze data from their experience memory. The samples can be analyzed for context changes in batch mode or online mode. If a change gets detected, then the counter c is incremented, signalling that the agent believes that context has changed. We adapt the online parametric Dirichlet change-point (ODCP) detection algorithm proposed in [14] for data consisting of experience tuples. Multiple change-points are detected by performing a sequence of single change-point detections. Although ODCP requires the multivariate data to be i.i.d. samples from a distribution, the justification in [15] explains the utilization of ODCP in the Markovian setting, where the data obtained does not consist of independent samples. The full algorithm for the Dirichlet change point detection algorithm is shown in Algorithm 4. In this algorithm, the maximum likelihood estimation of Dirichlet distribution parameters is calculated for the cumulative data (stored through experience tuples) using Eq. 1 below:

$$\alpha_i^* = argmax_{\alpha} log\Gamma(\sum_l \alpha_l) - \sum_l log\Gamma(\alpha_l) + \sum_l \left((\alpha_l - 1)(log\hat{x}_l)\right), \text{ where } \hat{x}_l = \frac{1}{T}\sum_i log(x_{i_l}) \quad (1)$$

Then, the log likelihood given distribution Q_0 is calculated using equation 2 below:

$$LL(x_1 \ldots x_T, Q) = \sum_{i=1}^{T} log(Q(x_i))$$
$$Q(x_i) = \frac{1}{B(\alpha)} \prod_{l=1}^{d} x_{i_l}^{\alpha_l - 1}, \text{ and } B(\alpha) = \frac{\prod_{l=1}^{d} \Gamma(\alpha_l)}{\Gamma(\sum_{l=1}^{d} \alpha_l)} \quad (2)$$

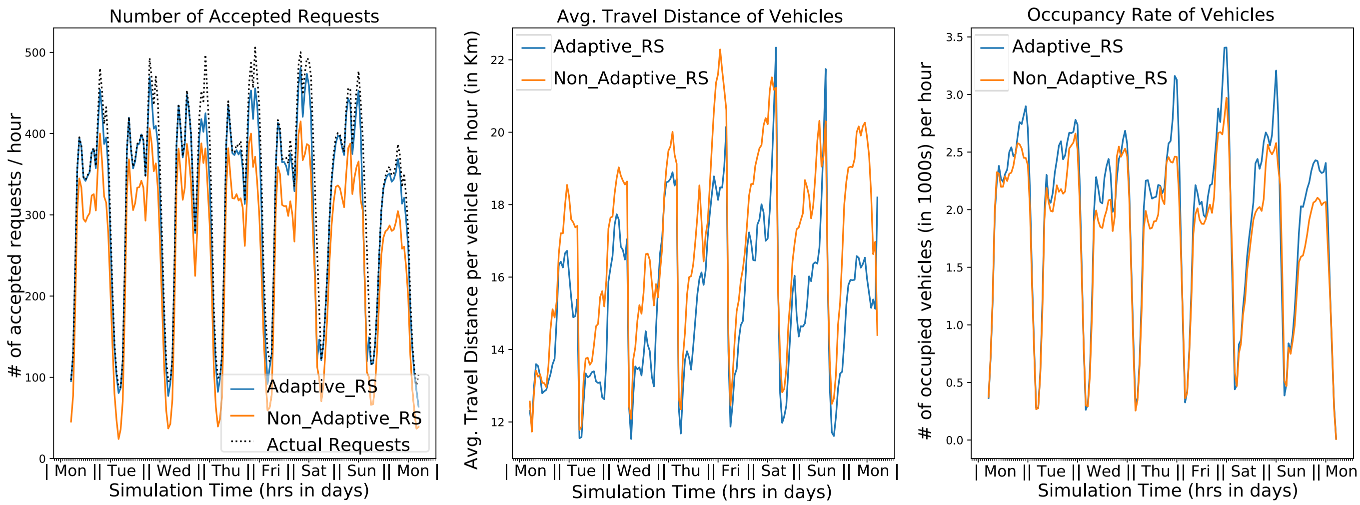


Fig. 3. Evaluation metrics for AdaPool and the non-adaptive baseline.

where d $=$ $|x_i|$ Dimensionality of x_i, $x_l \geq 0$, and $\sum_{l=1}^{d} x_l = 1$.

Then, at each time step t, that is seen as a potential change point, we split the data into two parts (prior and after this time step t), and we estimate the maximum likelihood as well as the sum of log likelihood for both partitions using the equations above. Finally, the algorithm returns the point in time T^* associated with the maximum log likelihood to be a potential change point. If the difference between this value and the log likelihood of our unsplit original data turns out to be greater than our threshold, then we declare that a change has been detected at time T^*.

C. DQN Dispatch Policy

In this section, we detail our system's global reward objective which allows efficient fleet dispatch in fulfilling service workloads. This global reward is optimized by our proposed algorithm in a distributed fashion as vehicles solve their own DQN to maximize rewards. At each time step t, each vehicle that is either idle or newly entered the market (i.e. vehicles that are marked available $\forall V_j \in v_{t,m}$) needs to make a dispatch decision of which zone m to be dispatched to at time slot t. To take this decision, each vehicle calculates the discounted reward (Q-value) associated with each potential action and picks the action that would yield the maximum future reward. The reward function, which drives the dispatch policy learner's objectives, is shaped in a manner which aims to (1) satisfy the demand of pick-up orders, thereby minimize the supply-demand mismatch: (diff_t), (2) minimize the dispatch time: T_t^D (i.e., expected travel time of vehicle V_j to go zone m at time t), (3) minimize the extra travel time a vehicle takes for car-pooling compared to serving one customer: Δt, (4) maximize the fleet profits P_t, and (5) minimize the number of utilized vehicles: e_t. These objectives are defined in Appendix F.

The overall objective of the system is optimized at each vehicle in the distributed transportation network. In this case, the reward $r_{t,n}$ for vehicle n at time slot t is represented in Eq. (3), where the objectives above are mapped to: (1) $C_{t,n}$: number of customers served by vehicle n at time t, (2) dispatch time: $T_{t,n}^D$, (3) extra travel time: $T_{t,n}^E$, (4) average profit for vehicle n at time t: $\mathbb{P}_{t,n}$, and (5) $\max(e_{t,n} - e_{t-1,n}, 0)$ that addresses the objective of minimizing the the number of vehicles at t to improve vehicle utilization. The reward function of each vehicle is defined as a weighted sum of these terms as:

$$r_{t,n} = r(s_{t,n}, a_{t,n}) = \beta_1 C_{t,n} - \left[\beta_2 T_{t,n}^D + \beta_3 T_{t,n}^E\right] + \beta_4 \mathbb{P}_{t,n} - \beta_5[\max(e_{t,n} - e_{t-1,n}, 0)] \quad (3)$$

where $\beta_1, \beta_2, \beta_3, \beta_4$ and β_5 depend on the weight factors of each of the objectives. Note that we maximize the discounted reward over a time frame. The negative sign here indicates that we want to minimize these terms. Further, the last term captures the status of vehicle n where $e_{t,n}$ is set to 1 if vehicle n was empty and then becomes occupied at time t (even if by one passenger), however, if it was already occupied and just takes a new customer, $e_{t,n}$ is 0. The intuition here is that if an already occupied vehicle serves a new user, the congestion and fuel costs will be less when compared to when an empty vehicle serves that user. Note that if we make β_3 very large, it will dis-incentivize passengers and drivers from making detours to serve other passengers, Thus, the setting becomes similar to the one in [8], where there is no carpooling. The overall optimization process includes a route planning and matching policy, and the DQN dispatch policy working in tandem with each other.

While the primary role of the DQN is to act as a means of dispatching idle vehicles, it contains useful signals on future anticipated demand that is utilized by other components of our method including the Demand Aware Matching and Route Planning. The additional profits term P_t integrated with the reward function makes the output expected discounted rewards (Q-values) associated with each possible move on the map, a good reflection of the expected earnings gained when heading to these locations. The Q-values are then used to decide on the best dispatching action to take for each individual vehicle. Since the state space is large, we don't use the full representation of s_t, instead a map-based input is used to alleviate this massive computing.

V. Experimental Results

A. Simulator Setup

In our simulator, we used the road network of the New York Metropolitan area along with a real public dataset of taxi trips in NY [7]. We used *Python* and *Tensorflow* to implement our framework. For each trip, we obtain the pick-up time, passenger count, origin location, drop-off location and ride fare. We use this trip information to construct travel requests and demand prediction model. We start by populating vehicles over the city, randomly assigning each vehicle a type and an initial location. According to the type assigned to each vehicle, we set the accompanying features accordingly such as: maximum capacity, mileage, and price rates (per mile of travel distance ω^1, and per waiting minute ω^3). We initialize the number of vehicles, to 8000. Note that, not all vehicles are populated at once, they are deployed incrementally into the market by each time step t. We also defined a reject radius threshold for a customer request. Specifically, if there is no vehicle within a radius of 5km to serve a request, it is rejected. This simulator hosts each deep reinforcement learning agent which acts as a ridesharing vehicle that aims to maximize its reward: Eq. (3).

B. DQN Training and Testing

The fleet of autonomous vehicles was trained in a virtual environment that simulates urban traffic. We consider the data of June 2016 for training and one week from July 2016 for evaluations. For each experiment, we trained our DQN neural networks using the data from the month of June 2016 for 20k epochs, which corresponds to a total of 14 days, and used the most recent 5000 experiences as a replay memory. Upon saving Q-network weights, after training, we retrieve the weights to run testing on an additional 8 days from the month of July which corresponds to 10k epochs. Thus, T = 8 × 24 × 60 steps, where $\Delta t = 1$ minute. To initialize the environment, we run the simulation for 20 minutes without dispatching the vehicles. Finally, we set $\beta_1 = 10, \beta_2 = 1, \beta_3 = 5, \beta_4 = 12, \beta_5 = 8$. Each vehicle has a maximum working time of 21 hours per day, after which it exits the market. Also, we perform hyper-parameter tuning to set k (the number of models to be learnt by our DQN) to 7, and the threshold for our ODCP algorithm to 5000. We show that our framework is able to recognize up to 7 different diurnal patterns throughout the day.

C. Computational Analysis

To provide more insight regarding the complexity of our AdaPool framework, we investigate:

1) Dispatch Decisions: We show in Fig. 4, the cumulative distribution function (cdf) plot for the time taken for dispatch decisions for each individual vehicle. We can obseve that with probability 1.0, it will take the vehicle < 0.2 seconds to make a dispatch decision of which location on the map to head to next in order to maximize its own reward.

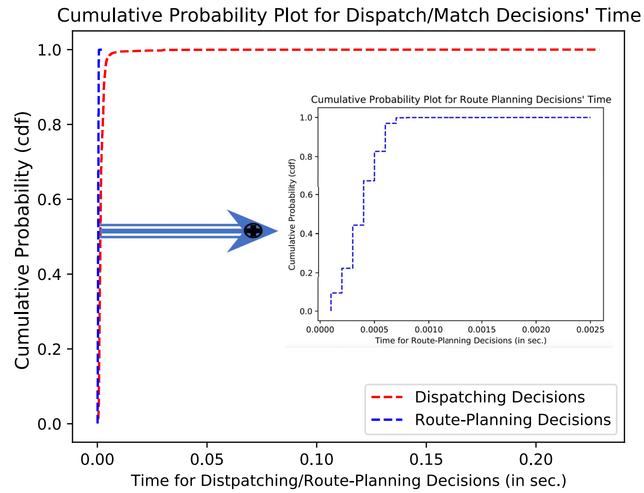


Fig. 4. CDF for dispatching and route-planning decisions' times.

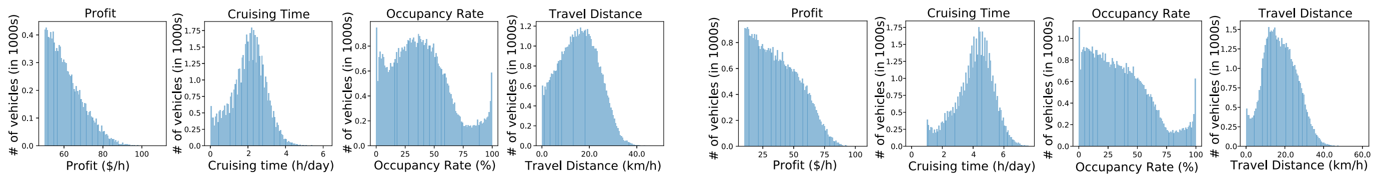
2) Matching and Route-Planning Decisions: We show in Fig. 4 the cdf plot for the time taken for route-planning decisions for each individual vehicle. This is the time taken by the vehicle to apply the greedy insertion operations and decide on the k' ride requests, which satisfy its capacity constraint and correspond to the minimum cost. We can conclude that with probability 1.0, it will take the vehicle < 2.5 milliseconds to make a dispatch decisions.

These results proves the viability and efficiency of our framework to be applied in large-scale real-world environments. Clearly, the time taken for route-planning is negligible when compared to the time taken for making dispatching decisions; however, both are very reasonable for real-time decision making scenarios.

D. Performance Metrics

We breakdown the reward, and investigate the performance of AdaPool against a non-adaptive baseline [13]. Recall that we want to minimize the components of our reward: Eq. (3).

- **Accepted Requests**: This is calculated as the total number of requests served by the fleet per working hour. The total number of customers served indicates how effectively the algorithm is able to minimize the supply demand gap and fulfill delivery requests.
- **Travel Distance:** This metric shows the total amount of distance traveled by each vehicle per hour of service, which gives a good reflection of the cost incurred by vehicles due to serving multiple ride requests. This distance is computed using the weights of the n edges that constitute the vehicle's optimal route from its current location to origin o_i and to destination d_i. This route is obtained through the insertion-operation, the route which minimizes the DARM cost function as shown in Algorithm 2.
- **Occupancy Rate:** This metric captures the utilization rate of the fleet of vehicles, it keeps track of how many vehicles are deployed from the fleet to serve the demand. This is calculated as the total number of vehicles that are carrying passengers per hour of service, while we also calculate the occupancy rate of vehicles (in Fig. 5),



(a) Performance Metrics of our AdaPool Framework

(b) Performance Metrics of the Non-Adaptive Baseline

Fig. 5. Histograms of performance metrics for the proposed AdaPool and the baseline.

which is defined as the percentage of time where vehicles are occupied out of their total working time. By minimizing the number of occupied vehicles, we achieve better utilization of individual vehicles in serving the demand. A lower occupancy rate indicates that a fleet is able to minimize the number of vehicles on the street to serve the requests.

- **Profit:** This represents the net profit accumulated by a vehicle over the course of a day, where the cost incurred by fuel consumption is subtracted from the revenue. The revenue is calculated by summing the trip fares from all customers served by this vehicle.
- **Cruising (idle) time:** This represents the time during which a vehicle is neither occupied nor gaining profit but still incurring gasoline cost. Lower cruising times therefore suggests a cost effective policy.
- **Detected points of change and the corresponding demand and hour in day:** This measure helps investigate the pattern of change in demand against the detected points of change in order to validate if our framework adapts accordingly.

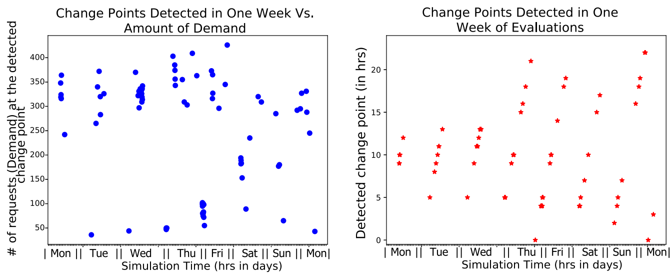
The proposed non-adaptive baseline aims to evaluate the effectiveness of the adaptive aspect of our framework as well as the demand aware matching and route planning component. Our proposed method incorporates both insertion-based route-planning and diurnal pattern adaptation. As compared to the non-adaptive baseline, we hypothesize that our AdaPool framework would be a more effective approach. With the capability of adapting to the demand pattern, we expect AdaPool to bring the supply/demand gap to a minimum and thus, minimize the cruising idle time and travel distance in addition to increasing the overall accept rate of requests. Given that the core intuition of our matching and route planning components is to group together rides that share route intersections to their destinations as opposed to rides heading to opposite-direction-destinations, we expect improvements in the number of rides served, profits, travel distance, and occupancy rate.

E. Results Discussion

From our simulation, we observe that the hypothesis for our baseline comparison has been supported for the most part by our experimental results. In Figure 3, we investigate the overall performance of our AdaPool framework. We show the actual number of requests as the dotted black line. We can observe that, over a week long of simulation, AdaPool consistently improves the overall acceptance rate of ride requests by around a $10-15\%$, while at the same time, significantly decreases the average travelling distance of the fleet. This proves the effectiveness of AdaPool in minimizing the supply/demand mismatch. Although, this comes at the cost of a slight increase in the number of utilized vehicles (≈ 300 extra vehicles) in the fleet, this outcome proves that - unlike the non-adaptive baseline, AdaPool does a successful job in re-balancing vehicles over the city, so extra vehicles would become occupied in order to serve extra demand (that was not served in the non-adaptive scenario) while at the same time achieving a decrease in the average travel distance of the fleet. This result points towards the effeciency of our insertion-based route-planning as it allows for serving more requests with the smallest possible travel distance. It is also worth noting that, even with a slight increase in the number of occupied vehicles, the total number of utilized vehicles stays below $3.5k$ which is less than half of the fleet (we set the maximum number of vehicles to 8000 in our simulator). This proves our hypothesis that AdaPool outperforms the non-adaptive baseline in the utilization of available resources. This is a positive outcome that points towards the viability of our proposed approach to learn diurnal patterns and adapt in a timely manner.

Besides, the extra number of utilized vehicles (which can be explained by the additional number of requests served) will -in turn- increase the average profits of the fleet which is also another desirable outcome. In Figure 5a, we can observe that AdaPool increases the average profits per vehicle in the fleet while minimizing their cruising idle time, which in turn, cuts down fuel consumption and environmental pollution. We note that, in AdaPool, more than 80% of vehicles in the fleet spend less than 3 hours of idle cruising per day. On the other hand, with the non-adaptive scenario, the plot is more skewed to the right where the average idle cruising time reaches more than 5 - 6 hours per day. This is mainly due to AdaPool dispatching vehicles according to the learnt demand pattern which makes the vehicles present at locations very close to the anticipated demand and thus minimizes their idle time and cruising time to get to the requests assigned to them. On the other hand, the non-adaptive baseline relies only on mobilizing vehicles according to the pickup locations of their requests as opposed to learning the changes of the supply-demand distribution of the city and mobilizing accordingly when they experience idle time.

In Figure 3, we observed the occupancy rate of the fleet (how many vehicles are occupied from the fleet); however in Figure 5, we look at the occupancy rate per vehicle. That is how much of the time, when the vehicle is in service, does it stay occupied. We note that, with AdaPool, more



(a) Detected change points and their corresponding demand

(b) Detected change points and their corresponding hour in day

Fig. 6. Analysis for Diurnal pattern adaptation.

than half of the fleet is occupied for more than 50% of their time in the market. Again, the occupancy rate plot is more skewed to the left with the non-adaptive baseline, which suggests less occupancy rate per vehicle as it leans towards $10\% - 30\%$. Finally, after we investigated the overall average travel distance for the whole fleet in Figure 3, we also take a closer look at the travel distance per vehicle in Figure 5. Clearly, AdaPool shows a considerably less travel distance per vehicle than the non-adaptive baseline. This further ascertains that, in AdaPool, the insertion-based route-planning in tandem with the demand pattern adaptaion achieves a great success in learning the demand pattern and re-balancing vehicles over the city accordingly. This significantly improves the utilization of each individual vehicle as well as the whole fleet.

Finally, to validate the adaptation aspect of AdaPool, Figure 6a shows the amount of demands at the detected points of change at which our approach switches contexts (i.e., models). We can, clearly, conclude that the detected change points are associated with either a sharp increase or decrease in demand which suggests a change in context; thus, our AdaPool switches to either learn a new model or to a previously learnt model (if the samples it observes has been learnt before). This behavior is consistent throughout the week of testing, except over the weekend where we can observe there has been some detected change points at intermediate demand. This could be attributed to other factors represented in our reward function, where the change might not be only dependent on the amount of demand but also on factors resulting from that such as larger customer's waiting times or larger cost incurred from fuel consumption. For future work, we plan to conduct more investigation on weekends, holidays, etc. and research on what other aspects could attribute to changing contexts (such as: unexpected weather conditions, etc.).

In Figure 6, we take a closer look at the pattern of the detected changes. We can observe that the change pattern is relatively consistent throughout the week of evaluations, excluding the weekend where the pattern varies slightly. In weekdays, AdaPool detects changes somewhere between 5-7 am, then between 11-noon, after that between 4-6 pm, and finally at night between 8-9 pm. On the other hand, for weekends there are additional detected points later in the night around 10-11 pm and at mid-night. Tying back to Figure 6a, we can conclude that these points correspond to peaks (e.g. 4-6 pm when people heading home from work, and students from school), decreasing demand (e.g., around 8-9 pm when traffic generally starts slowing down in weekdays), or rising peaks (e.g., around 5-7 am when people starts heading to work, or to school). This outcome ascertains that our approach is able to detect diurnal patterns as the contexts of the underlying environment change. In addition to this, AdaPool also adapts according to these changes, as it switches between models. Tracking these model switches, we observe that AdaPool exploits the same model around the same timing each day, and thus, models vary by the variation of demand as well. For instance, in weekdays, between 10 pm and 5 am, AdaPool utilizes the same model every day. We note that this time-frame characterizes the least amount of demand every weekday. Similarly, mornings between 5-9 am, the same model is also utilized every week day. This time-frame corresponds to a notably high demand as it signals the beginning of each business day. Therefore, AdaPool is able to adapt by switching to the model that corresponds to the diurnal pattern it has learnt, while it still is able to learn online any new unexpected changes as they take place within the environment.

VI. Conclusion and Future Work

In this paper, we detailed our novel approach—Distributed Adaptive Deep Q Learning for Ride-Sharing with car pooling, namely "AdaPool" framework and our Demand-Aware Matching and route planning approach—that generate ideal routes on-the-fly and adapts dynamically to changing environmental contexts. Agents' (i.e., vehicles) decision-making process is informed by a reward function that aims to achieve the maximum profit for drivers while accounting for fuel costs, waiting times, and supply-demand mismatch to compute the reward. This novel AdaPool methodology integrates a DQN-based dispatch algorithm, learns up to 7 different contextual models, and adapts accordingly by detecting the relevant change points. The learnt Q-values associated with the active model are then leveraged by each vehicle to make informed dispatch decisions independently. Experimental results show that AdaPool framework boosts the acceptance rate, while enhancing drivers' profits and decreasing their average travel distance. Given the maximum number of vehicles (8000) populated in the simulation, our framework utilizes less than 50% of the vehicles to serve the demand of up to 90% of the requests. Experiments also show that vehicle idle time (cruising without passengers) is reduced to below 2 hours per day and 40% - 70% of the vehicles are occupied most of the time. Our model-free AdaPool framework can be extended to large-scale ridesharing protocols due to the vehicles' distributed decision making that reduces the decision space significantly.

Application of the proposed approach for decision making in other environments with diurnal patterns will be considered in the future. Extension of this work to include capabilities of a joint delivery system for passengers and goods as in [16] and [17], multitrip deliveries within a certain time window as in [18], or using multi-hop routing of passengers as in [19] for efficient fleet utilization is left as future work.

REFERENCES

[1] M. Haliem, V. Aggarwal, and B. Bhargava, "AdaPool: An adaptive model-free ride-sharing approach for dispatching using deep reinforcement learning," in *Proc. 7th ACM Int. Conf. Syst. Energy-Efficient Buildings, Cities, Transp.*, Nov. 2020, pp. 304–305.

[2] S. Abdallah and M. Kaisers, "Addressing environment non-stationarity by repeating Q-learning updates," *J. Mach. Learn. Res.*, vol. 17, no. 1, pp. 1582–1612, 2016.

[3] R. Kemker, M. McClure, A. Abitino, T. Hayes, and C. Kanan, "Measuring catastrophic forgetting in neural networks," in *Proc. AAAI Conf. Artif. Intell.*, 2018, vol. 32, no. 1, pp. 3390–3398.

[4] M. McCloskey and N. J. Cohen, "Catastrophic interference in connectionist networks: The sequential learning problem," *Psychol. Learn. Motivat.*, vol. 24, pp. 109–165, Dec. 1989.

[5] J. R. Zimbelman, "Introduction to the planetary dunes special issue, and the aeolian career of ronald greeley," *Icarus*, vol. 230, pp. 1–4, Feb. 2014.

[6] A. O. Al-Abbasi, A. Ghosh, and V. Aggarwal, "DeepPool: Distributed model-free algorithm for ride-sharing using deep reinforcement learning," *IEEE Trans. Intell. Transp. Syst.*, vol. 20, no. 12, pp. 4714–4727, Dec. 2019.

[7] (2019). *NYC Taxi and Limousine Commission-Trip Record Data*. [Online]. Available: https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page

[8] T. Oda and C. Joe-Wong, "MOVI: A model-free approach to dynamic fleet management," in *Proc. IEEE Conf. Comput. Commun.*, Apr. 2018, pp. 2708–2716.

[9] (2018). *Openstreetmaps*. [Online]. Available: https://www.openstreetmap.org

[10] M. Haliem, G. Mani, V. Aggarwal, and B. Bhargava, "A distributed model-free ride-sharing algorithm with pricing using deep reinforcement learning," in *Proc. Comput. Sci. Cars Symp.*, Dec. 2020, pp. 1–10.

[11] D. C. Wyld, M. A. Jones, and J. W. Totten, "Where is my suitcase? RFID and airline customer service," *Marketing Intell. Planning*, vol. 23, no. 4, pp. 382–394, 2005.

[12] X. Bei and S. Zhang, "Algorithms for trip-vehicle assignment in ride-sharing," in *Proc. 32nd AAAI Conf. Artif. Intell.*, 2018, pp. 3–9.

[13] M. Haliem, G. Mani, V. Aggarwal, and B. Bhargava, "A distributed model-free ride-sharing approach for joint matching, pricing, and dispatching using deep reinforcement learning," *IEEE Trans. Intell. Transp. Syst.*, early access, Aug. 10, 2021, doi: 10.1109/TITS.2021.3096537.

[14] P. K. J., N. Singh, P. Dayama, A. Agarwal, and V. Pandit, "Change point detection for compositional multivariate data," *Int. J. Speech Technol.*, vol. 4, pp. 1–26, Jun. 2021.

[15] S. Padakandla, K. J. Prabuchandran, and S. Bhatnagar, "Reinforcement learning in non-stationary environments," *Appl. Intell.*, vol. 50, pp. 3590–3606, Oct. 2020.

[16] K. Manchella, A. K. Umrawal, and V. Aggarwal, "Flexpool: A distributed model-free deep reinforcement learning algorithm for joint passengers and goods transportation," *IEEE Trans. Intell. Transp. Syst.*, vol. 22, no. 4, pp. 2035–2047, Apr. 2021.

[17] K. Manchella, M. Haliem, V. Aggarwal, and B. Bhargava, "PassGood-Pool: Joint passengers and goods fleet management with reinforcement learning aided pricing, matching, and route planning," 2020, *arXiv:2011.08999*. [Online]. Available: http://arxiv.org/abs/2011.08999

[18] J. Wang, Y. Sun, Z. Zhang, and S. Gao, "Solving multitrip pickup and delivery problem with time windows and manpower planning using multiobjective algorithms," *IEEE/CAA J. Automatica Sinica*, vol. 7, no. 4, pp. 1134–1153, Jul. 2020.

[19] A. Singh, A. O. Al-Abbasi, and V. Aggarwal, "A distributed model-free algorithm for multi-hop ride-sharing using deep reinforcement learning," *IEEE Trans. Intell. Transp. Syst.*, early access, Jul. 8, 2021, doi: 10.1109/TITS.2021.3083740.

[20] J. Jung, R. Jayakrishnan, and J. Y. Park, "Dynamic shared-taxi dispatch algorithm with hybrid-simulated annealing," *Comput.-Aided Civil Infrastruct. Eng.*, vol. 31, no. 4, pp. 275–291, 2016.

[21] A. Tafreshian and N. Masoud, "Trip-based graph partitioning in dynamic ridesharing," *Transp. Res. C, Emerg. Technol.*, vol. 114, pp. 532–553, May 2020.

[22] J. Alonso-Mora, S. Samaranayake, A. Wallar, E. Frazzoli, and D. Rus, "On-demand high-capacity ride-sharing via dynamic trip-vehicle assignment," *Proc. Nat. Acad. Sci. USA*, vol. 114, no. 3, pp. 462–467, Jan. 2017.

[23] Y. Li and S. H. Chung, "Ride-sharing under travel time uncertainty: Robust optimization and clustering approaches," *Comput. Ind. Eng.*, vol. 149, Nov. 2020, Art. no. 106601.

[24] Y. Hu, Y. Yao, and W. S. Lee, "A reinforcement learning approach for optimizing multiple traveling salesman problems over graphs," *Knowl.-Based Syst.*, vol. 204, Sep. 2020, Art. no. 106244.

[25] A. Nilim and L. El Ghaoui, "Robust control of Markov decision processes with uncertain transition matrices," *Oper. Res.*, vol. 53, no. 5, pp. 780–798, Oct. 2005.

[26] E. Even-Dar, S. M. Kakade, and Y. Mansour, "Experts in a Markov decision process," in *Proc. Adv. Neural Inf. Process. Syst.*, 2005, pp. 401–408.

[27] T. Dick, A. Gyorgy, and C. Szepesvari, "Online learning in Markov decision processes with changing cost sequences," in *Proc. Int. Conf. Mach. Learn.*, 2014, pp. 512–520.

[28] R. Ortner, P. Gajane, and P. Auer, "Variational regret bounds for reinforcement learning," in *Proc. Uncertainty Artif. Intell.*, 2020, pp. 81–90.

[29] S.-J. Kim and R. Mowakeaa, "Kernel-based efficient lifelong learning algorithm," in *Proc. IEEE Data Sci. Workshop (DSW)*, Jun. 2019, pp. 507–515.

[30] K. James *et al.*, "Overcoming catastrophic forgetting in neural networks," *Proc. Nat. Acad. Sci. USA*, vol. 114, no. 13, pp. 3521–3526, Mar. 2017.

[31] Z. Li and D. Hoiem, "Learning without forgetting," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 40, no. 12, pp. 2935–2947, Dec. 2018.

[32] L. Liu and G. S. Sukhatme, "A solution to time-varying Markov decision processes," *IEEE Robot. Automat. Lett.*, vol. 3, no. 3, pp. 1631–1638, Jul. 2018.

[33] M. Ornik and U. Topcu, "Learning and planning for time-varying mdps using maximum likelihood estimation," *J. Mach. Learn. Res.*, vol. 22, pp. 1–35, Oct. 2021.

[34] X. Wang, Q. Kang, M. Zhou, L. Pan, and A. Abusorrah, "Multiscale drift detection test to enable fast learning in nonstationary environments," *IEEE Trans. Cybern.*, vol. 51, no. 7, pp. 3483–3495, Jul. 2021.

[35] J. Zhang, Z. Wei, Z. Yan, and A. Pani, "Collaborated online change-point detection in sparse time series for online advertising," in *Proc. IEEE Int. Conf. Data Mining*, Nov. 2015, pp. 1099–1104.

[36] L. Chen, X. Hu, W. Tian, H. Wang, D. Cao, and F.-Y. Wang, "Parallel planning: A new motion planning framework for autonomous driving," *IEEE/CAA J. Automatica Sinica*, vol. 6, no. 1, pp. 236–246, Jan. 2019.

[37] J. Wang *et al.*, "Demand-aware route planning for shared mobility services," *Proc. VLDB Endowment*, vol. 13, no. 7, pp. 979–991, Mar. 2020.

[38] L. Jiang, H. Huang, and Z. Ding, "Path planning for intelligent robots based on deep Q-learning with experience replay and heuristic knowledge," *IEEE/CAA J. Autom. Sinica*, vol. 7, no. 4, pp. 1179–1189, Jul. 2020.

[39] B. Cici, A. Markopoulou, and N. Laoutaris, "Designing an on-line ride-sharing system," in *Proc. 23rd SIGSPATIAL Int. Conf. Adv. Geographic Inf. Syst.*, Nov. 2015, pp. 1–4.

[40] P. Cheng, H. Xin, and L. Chen, "Utility-aware ridesharing on road networks," in *Proc. ACM Int. Conf. Manage. Data*, May 2017, pp. 1197–1210.

[41] M. Ota, H. Vo, C. Silva, and J. Freire, "STaRS: Simulating taxi ride sharing at scale," *IEEE Trans. Big Data*, vol. 3, no. 3, pp. 349–361, Sep. 2017.

[42] R. S. Thangaraj, K. Mukherjee, G. Raravi, A. Metrewar, N. Annamaneni, and K. Chattopadhyay, "Xhare-a-ride: A search optimized dynamic ride sharing system with approximation guarantee," in *Proc. IEEE 33rd Int. Conf. Data Eng. (ICDE)*, Apr. 2017, pp. 1117–1128.

[43] Y. Tong, Y. Zeng, Z. Zhou, L. Chen, J. Ye, and K. Xu, "A unified approach to route planning for shared mobility," *Proc. VLDB Endowment*, vol. 11, no. 11, p. 1633, 2018.

[44] X. Jin, H. Qin, Z. Zhang, M. Zhou, and J. Wang, "Planning of garbage collection service: An arc-routing problem with time-dependent penalty cost," *IEEE Trans. Intell. Transp. Syst.*, vol. 22, no. 5, pp. 2692–2705, May 2021.

[45] Y. Li, F. Chu, C. Feng, C. Chu, and M. Zhou, "Integrated production inventory routing planning for intelligent food logistics systems," *IEEE Trans. Intell. Transp. Syst.*, vol. 20, no. 3, pp. 867–878, Mar. 2019.

[46] L. Wang and J. Lu, "A memetic algorithm with competition for the capacitated green vehicle routing problem," *IEEE/CAA J. Automatica Sinica*, vol. 6, no. 2, pp. 516–526, Feb. 2019.