Deep Reinforcement Learning with Knowledge Transfer for Online Rides Order Dispatching

Zhaodong Wang*† Zhiwei (Tony) Qin*‡ Xiaocheng Tang*‡ Jieping Ye§ Hongtu Zhu§
*Equal contribution
†Washington State University, Pullman, WA Email: zhaodong.wang@wsu.edu
‡DiDi Research America, Mountain View, CA Email: {qinzhiwei,xiaochengtang}@didiglobal.com
§DiDi Chuxing, Beijing, China Email: {yejieping,zhuhongtu}@didiglobal.com

***Abstract*—Ride dispatching is a central operation task on a ride-sharing platform to continuously match drivers to trip-requesting passengers. In this work, we model the ride dispatching problem as a Markov Decision Process and propose learning solutions based on deep Q-networks with action search to optimize the dispatching policy for drivers on ride-sharing platforms. We train and evaluate dispatching agents for this challenging decision task using real-world spatio-temporal trip data from the DiDi ride-sharing platform. A large-scale dispatching system typically supports many geographical locations with diverse demand-supply settings. To increase learning adaptability and efficiency, we propose a new transfer learning method Correlated Feature Progressive Transfer, along with two existing methods, enabling knowledge transfer in both spatial and temporal spaces. Through an extensive set of experiments, we demonstrate the learning and optimization capabilities of our deep reinforcement learning algorithms. We further show that dispatching policies learned by transferring knowledge from a source city to target cities or across temporal space within the same city significantly outperform those without transfer learning.**

***Index Terms*—Ride dispatching, Deep reinforcement learning, Transfer learning, Spatio-temporal mining**

I. Introduction

As GPS-enabled applications are widely used in the ride-sharing market nowadays, massive amount of trip data could be collected, offering huge opportunities for providing more intelligent service and leading to a surge of interest in research fields such as demand prediction [1], [2], driving route planning [3], [4], and order dispatching [5], [6]. Our work focuses on order dispatching. There are two major challenges in building an intelligent dispatching system for a ride-sharing platform.

The first challenge is to improve the dispatching efficiency. Previous work like [7] focused on how to match a ride-sharing driver with the passenger at the least cost of travel distance or pick-up time. The work in [8] aims to improve the success rate of the global order matches, by involving the combinatorial optimization problem [9]. A higher success rate would deliver better user experience, but it should not be the only metric to be optimized. Another previous research [10] proposed a revenue optimization method for cruising taxi drivers. It leverages reinforcement learning to help the drivers make decisions on their cruising routes. After training on the historical trajectory logs, the cruise efficiency could be improved. The cruise trajectories (training data in [10]) only consider location states when the taxi is idle (no customer on board). However, since passenger trips change drivers' locations, there is still limitation on the global optimization if the training data only contains idle cruise logs.

The other challenge is scalability. For a large-scale platform that supports many cities, it is common to decompose the order dispatching problem, often by natural geographical boundaries, and focus on individual cities instead, given the complexity of optimizing the entire ride-sharing platform in one shot and the diverse supply-demand settings across different cities. However, building a new model for every city on the platform is not computationally efficient and is hardly scalable because of the size of each optimization problem and the large number of cities. Although traffic patterns are usually different in different cities, they may still share some common properties. For example, we could imagine the rush-hour traffic demand between downtown and uptown could possibly be similar across different cities. Instead of treating each city as a fresh problem, it is crucial to enable knowledge transfer and sharing across cities such that learning happens in a global context instead of being siloed in one narrow area. This is often referred to as Transfer Learning [11], [12], which has been successfully applied in many domains, such as multitask learning [13], [14], deep reinforcement learning [15]–[17], and representation learning [18], [19].

To tackle the above challenges, we consider the order dispatching problem from a single driver perspective, where the driver is assigned a sequence of trip requests with the objective of maximizing the total revenue throughout the day. Recent work in [20] has shown great success in this regard by using a learning and planning approach based on passenger demand and taxi supply pattern from historical data. In particular, [20] models this problem as a Markov Decision Process (MDP), and performs policy evaluation in a discrete tabular state space using dynamic programming. The limitations of this approach, however, are mainly in three folds. First, the state value can vary a lot based on different real-time factors such as transportation supply and demand, but it is generally difficult for a tabular model to incorporate and respond to such contextual information in real time. In other words, the agent has to be able to generalize beyond the historical data according to online conditions. Second, trips in different cities constitute different

MDPs that may share many common structures. The tabular approach treats each MDP separately and does not provide a viable mechanism for transferring knowledge across cities. Finally, the policy improvement in [20] is a separate step that has to be performed on-line for at least one day. To converge it usually takes dozens of evaluation-improvement iterations which can take weeks. Ideally, we would like a off-policy method which learns more efficiently, while avoiding the risk of learning directly online.

Our contribution in this paper is a deep reinforcement learning approach to overcome those limitations. We build upon the recent progress in model-free RL and propose an order dispatching framework based on Q-learning [21], [22]. The key to produce an optimal policy that governs the decision-making at each step is to estimate the *state-action value function* of the driver. This function tells us how good a decision made at a particular location and time under given supply-demand context with respect to the long-term objective. Specifically, our approach falls within the deep Q-network (DQN) framework, with additional capability of carrying out action search. Our approach effectively combines historical data and simple synthetic auxiliary data for training a deep reinforcement learning agent and works even when a full simulation environment is not available. The resulting Q-network can be used as a component in a multi-driver dispatching system. DQN has shown its learning power through reaching and exceeding human-level playing of the various Atari games [22]. We show in this paper that it is equally powerful in learning strategies for drivers and the dispatching system.

Compared with tabular state value functions, using deep networks also has the advantages of leveraging trained knowledge and speed up learning across different cites. Considering that reinforcement learning usually suffers from slow learning speed at the beginning, transferring correlated prior knowledge [23] provides an efficient solution to boosting the learning process. We propose a novel transfer learning method for order dispatching to leverage knowledge transfer from a source city, demonstrating that reusing prior models could improve the training performance in the target cities. The improvement includes three aspects: higher jump-start (i.e. better initial solution), faster learning towards convergence, and higher convergence performance.

In what follows, we will formulate our MDP in Section II and describe our deep Q-network approach tailored to large-scale order dispatching problems on a ride-sharing platform in Section III. In Section IV, we propose a new transfer learning method, along with two existing methods, which allows us to transfer knowledge in both spatial and temporal spaces. We demonstrate the learning and optimization capabilities of our deep reinforcement learning approach and the advantages of the proposed transfer learning method in learning speed through an extensive set of experiments in Section V using real trip data from the DiDi platform. We close the paper with a few concluding remarks in Section VI.

II. MDP FORMULATION

Our basic MDP formulation follows that of [20], with the major distinction that our approach uses more granular state and action information with supply-demand contextual features and learns a state-action value function. The agent is defined from a driver's perspective. A trip transition consists of order pick-up and completion: The driver is matched to a trip order and travels to the trip origin location. The trip moves the driver to the destination. The driver earns an immediate reward (trip fee) from this transition. A transition can also be an idle driver movement. For the rest of the paper, we will consider an idle driver movement as a zero-reward trip. We list the key elements of our MDP formulation below.

State, s is the geo-coordinates of the driver and time-of-day (in seconds) when the driver is dispatched for a trip order, i.e. $s := (l, t)$, where l is the GPS coordinates pair (latitude, longitude) and t is time. Note that it could be different from the actual origin of the trip where the passenger stands at. Moreover, s may contain additional contextual features at (l, t), such as statistics of demand, supply, and order fulfillment within the vicinity of (l, t), denoted as f. In this case, s can be extended from (l, t) to (l, t, f). We also differentiate the time for weekday and weekend. For the rest of the paper, we denote the l and t components of a state s by s_l and s_t respectively.

Action, a is the assignment of a particular trip to the driver, which is simply defined by the trip destination and drop-off time. Let the current state $s_0 := (l_0, t_0, f_0)$ be the driver's location, time and the context when the trip is assigned, and the next-state $s_1 := (l_1, t_1, s_1)$ is the drop-off location, time and context. Then, the action is $a = (l_1, t_1)$. The space of all eligible actions is denoted by $\mathcal{A}$.

Reward, r is the total fee collected for the trip and is a function of s and a.

An **episode** is one complete day, from 0:00am to 23:59pm. Hence, a terminal state is a state with t component corresponding to 23:59pm. We set s_1 in all those transitions where the trip crosses midnight to be terminal state.

State-action value function, $Q(s, a)$ is expected cumulative reward that the driver will gain till the end of an episode if he/she starts at state s and takes an action a. Mathematically, $Q(s, a) := E\left[\sum_{t=0}^{T} \gamma^t R(S_t, A_t) | S_0 = s, A_0 = a\right]$, where S, A, and R are stochastic variable version of s, a, and r respectively; T is the number of transition steps till the terminal state, and γ is the discount factor for the future rewards. We discretize the time space into steps of 10 minutes and γ is multiple powers of the time steps that an order strides across.

Policy, $\pi(a|s)$ is a function that maps a state s to a distribution over the action space (stochastic policy) or a particular action (deterministic policy). The greedy policy with respect to a learned $Q(s, a)$ is given by $\pi(s) := \arg\max_a Q(s, a)$.

State value function, $V(s)$: expected cumulative reward that the driver will gain till the end of an episode if he/she starts at state s and follows a policy π. Assuming that a greedy policy w.r.t. the Q function is used, the state value $V(s) := Q(s, \pi(s)) = \max_{a \in \mathcal{A}} Q(s, a)$.

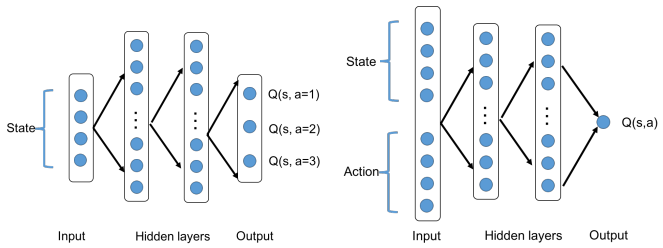


Fig. 1: Left: Q-network in [22]. Right: Q-network in this paper.

III. Deep Q-network with action search

To solve the MDP formulated in Section II, we adopt the model-free approach and use the DQN framework proposed in [22]. The deviations from [22] lie in two folds: First, on the neural network architecture, due to the continuous action space, we use both state and action, (s, a) as the network input and the Q-value as the single output, whereas vanilla DQN assumes a small discrete action space and uses only the state as input and multiple outputs corresponding to the action value for each action. The hidden layers in both cases can be either fully-connected layers or convolution layers, depending on the specific application. Figure 1 illustrates the structure contrast. Second, since the action space within our MDP formulation for order dispatching is technically continuous, we develop an action search mechanism (see Sections III-B and III-C) to define a discrete action space for each transition. The pseudo code of the algorithm is shown in Algorithm III.2.

In the DQN framework, the mini-batch update through back-propagation is essentially a step for solving a bootstrapped regression problem with the following loss function

$$\Big(Q(s_0, a|\theta) - r(s_0, a) - \gamma \max_{a' \in \mathcal{A}} Q(s_1, a'|\theta')\Big)^2, \qquad (1)$$

where θ' is the weights for the Q-network of the previous iteration and $\mathcal{A}$ is the action space.

To improve training stability, we use Double-DQN proposed in [24]. Specifically, a target Q-network $\hat{Q}$ is maintained and synchronized periodically with the original Q-network. The targets in (1) is modified so that the argmax is evaluated by the target network:

$$r(s_0, a) + \gamma \hat{Q}(s_1, \arg\max_{a' \in \mathcal{A}} Q(s_1, a'|\theta')|\hat{\theta}'). \qquad (2)$$

A. Model training

Building a realistic order dispatching environment that simulates city-scale driver-passenger dynamics is very challenging, leaving alone applying a value function learned from a simulator to real settings. On the other hand, a large amount of historical trip data is usually available in the data warehouse. Therefore, we choose to use historical trip data for training. This distinguishes our method from conventional reinforcement learning approaches, in that there is no explicit simulation environment involved in training. We instead train our agent on historical data generated by some existing behavior policy and augmented with simple synthetic data, which we will describe in Sections III-B and III-C. Each trip x defines a transition of the agent's states (s_0, a, r, s_1). These transition experiences are retrieved from the data warehouse one by one (as if they were generated from a simulator) and are stored in a replay memory, similar to [22]. Each iteration then samples a mini-batch from this replay memory. We state our DQN-based training algorithm in Algorithm III.2 and explain the various algorithmic elements in the subsequent sections.

B. Action search

Recall from Section II that an action takes the form (l, t). Since both the GPS coordinates and time are continuous, computing the max-Q term in (1) exactly is not tractable. In addition, the t-component has dependency on the l-component as it reflects the duration of the trip. Random sampling from the action space is thus not appropriate. Hence, we develop an approximation scheme for computing this term by constructing an approximate feasible space for the actions, $\tilde{\mathcal{A}}(s)$. This notation makes explicit the dependency of the action space on the state s where the search starts. Instead of searching through all valid actions, we search within the historical trips originating from the vicinity of s:

$$\tilde{\mathcal{A}}(s) := \{x_{s_1} | x \in \mathcal{X}, B(x_{s_0}) = B(s)\}, \qquad (3)$$

where $\mathcal{X}$ is the set of all trips, and $B(s)$ is discretized spatio-temporal bin that s falls into. For spatial discretization, we use the hexagon bin system, where in our case here, a hexagon bin is represented by its center point coordinates. x_{s_0} and x_{s_1} are the s_0 and s_1 components of the trip x respectively. The larger the search space is, the more computation is required for evaluating the value network at each action point. We set the number of actions allowed in the action search space as a tuning parameter and do random sampling without replacement if necessary. The same search procedure is used for policy evaluation, where we simulate the driver's trajectory during the day using historical trip data.

C. Expanded action search

Due to training data sparsity in certain spatio-temporal regions, e.g. some remote area in early morning, the above action search may return an empty set. In this case, we perform an expanded action search in both spatial and temporal spaces. The first search direction is to stay at the last drop-off location and wait for a period of time, which corresponds to keeping the l-component, s_l constant and advancing s_t, till one of the following happens (s' is the searched state.): 1) If $\tilde{\mathcal{A}}(s')$ is non-empty, then return $\tilde{\mathcal{A}}(s')$. 2) If a terminal state is reached, then return the terminal state. 3) If s'_t exceeds the wait-time limit, then return s'. The second search direction is through spatial expansion by searching the neighboring hexagon bins of s in a layered manner. See Figure 2 for an illustration. For each layer L of hexagon bins, we search within the appropriate time interval to take into account the travel time required to reach the target hexagon bin from s. The travel time estimation can be obtained from a map service, for example, and is beyond the scope of this paper. We denote the layer L neighboring

spatio-temporal bins of s by $B(s, L)$ and the set of historical trips originating from any of the bins in $B(s, L)$ by

$$\tilde{\mathcal{A}}(s, L) := \{x_{s_1} | x \in \mathcal{X}, B(x_{s_0}) \in B(s, L)\}. \quad (4)$$

We stop increasing L when $\tilde{\mathcal{A}}(s, L)$ is non-empty and return $\tilde{\mathcal{A}}(s, L)$. Otherwise, we return $B(s, L_{\max})$, i.e. the hexagon bins' center points and their associated time components. $L_{\max}$ basically controls the size of the action space. Algorithm III.1 summarizes the full action search. We can view action search as bridging a spatio-temporal bin to another bin by keeping the driver waiting or moving the driver without a passenger to another location to get dispatched for an order.

Algorithm III.1 Spatio-temporal action search

1: Given $s = (l_0, t_0)$; t' is search time limit.
2: $\mathcal{A} \leftarrow \{\}$
3: $T_{\max} \leftarrow \min(T, t_0 + t')$
4: **if** $\tilde{\mathcal{A}}(s) \neq \emptyset$ **then**
5: $\quad \mathcal{A} \leftarrow \mathcal{A} + \tilde{\mathcal{A}}(s)$
6: **else**
7: $\quad$ **for** $t = t_0, t_0 + 1, \cdots, T_{\max}$ **do**
8: $\quad\quad s' \leftarrow (l_0, t)$
9: $\quad\quad$ **if** $\tilde{\mathcal{A}}(s') \neq \emptyset$ **then**
10: $\quad\quad\quad \mathcal{A} \leftarrow \mathcal{A} + \tilde{\mathcal{A}}(s')$
11: $\quad\quad\quad$ break
12: $\quad\quad$ **end if**
13: $\quad$ **end for**
14: $\quad$ **if** $\mathcal{A}$ did not change from line 6 **then**
15: $\quad\quad \mathcal{A} \leftarrow \mathcal{A} + s'$
16: $\quad$ **end if**
17: $\quad$ **for** $L = 1, \cdots, L_{\max}$ **do**
18: $\quad\quad$ **if** $B(s, L) \neq \emptyset$ **then**
19: $\quad\quad\quad \mathcal{A} \leftarrow \mathcal{A} + \tilde{\mathcal{A}}(s, L)$
20: $\quad\quad\quad$ break
21: $\quad\quad$ **end if**
22: $\quad$ **end for**
23: $\quad$ **if** $\mathcal{A}$ did not change from line 17 **then**
24: $\quad\quad \mathcal{A} \leftarrow \mathcal{A} + B(s, L_{\max})$
25: $\quad$ **end if**
26: **end if**
27: **return** $\mathcal{A}$

D. Terminal state values

From the definition of a terminal state in Section II, it is clear that $Q(s, a)$ with s_t near the end of the episode horizon should be close to zero regardless the location s_l. Following the idea of the dynamic programming algorithm, we add transitions with s_1 being a terminal state to the replay buffer at the very beginning of training. We find that it helps getting the terminal state-action values right early in training. This is important because the target values for the states s_0's in the mini-batch updates of DQN (i.e. (2)) are computed through bootstrapping on the values of states that are temporally after them. Since the training samples with a terminal state form a very small percentage of the entire data set, a uniform sampling of mini-batches would result in the terminal state values being learned without enough supervision, which causes the values of many states far away from the terminals to be fitted with incorrect targets, hence slowing down the learning process.

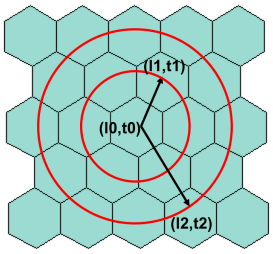


Fig. 2: Action search. The red circle lines cover the first two layers of neighboring hexagon bins of l_0. The arrows represent searches to hexagon bin centered at l_1 (first layer) at time $t_1 > t_0$ and hexagon bin centered at l_2 at time $t_2 > t_1$. The spatio-temporal bins covered by the inner red circle are $B((l_0, t_0), 1)$.

.

E. Experience augmentation

The original training data is the experience generated by the given historical policy, which may not have sufficiently explored the trajectory space. In particular, when the driver is at a state where historically there were few trips originating from there, our action search may require the driver to wait a long period of time or cruise without a passenger before a trip starts. However, there might be very few such transitions in the data. If the agent were trained on the original trip data, it would not learn to make good decisions in situations where the driver go into a rare state. The way we mitigate this problem is to supplement the original training experience with transitions obtained through action search. Specifically, we add action search experiences generated by following Sections III-B and III-C during mini-batch update (2) to the replay memory to supplement the existing training data.

F. Evaluating Q^π

To learn the average value of a driver in a given state, one approach is to learn Q^π of the policy π that generated the training data. To do that, we simply replace the 'argmax' in (2) with 'mean'. The resulting algorithm is similar to Expected SARSA [25]. The value network is trained on targets that represent the average values that drivers would have over all possible actions under policy π. Dispatching policy that is greedy or collectively greedy (see Section III-G) with respect to Q^π is one-step policy improvement.

G. Deployment in multi-driver dispatching environment

The real environment is intrinsically multi-agent, since multiple drivers fulfill passenger orders at the same time. Our learned single-driver value function can nevertheless be

Algorithm III.2 Double-DQN with spatio-temporal action search

Given: historical trips pool $\mathcal{P}$, a constant C.
Initialize replay memory $\mathcal{M}$ to capacity N and insert the terminal transitions set $\{x : x_{s_1} \text{ is terminal}\}$.
Initialize the state-action value network Q with random weights θ_0.
Initialize the target state-action value network $\hat{Q}$ with weights $\hat{\theta}_0$.
for $t = 1, 2, \cdots, T$ **do**
  Remove a trip sample (s_0, r, s_1) from $\mathcal{P}$, where $s_0 = (l_0, t_0)$, and $s_1 = (l_1, t_1)$.
  Extract action a from s_1. Store transition (s_0, a, r, s_1) in $\mathcal{M}$.
  Sample a random mini-batch $\{(s_j, a_j, r_j, s_{j+1})\}$ from $\mathcal{M}$.
  For each s_{j+1}, perform action search in Algorithm III.1 and get $\tilde{\mathcal{A}}$.
  if s_{j+1} is a terminal state **then**
    $y_j \leftarrow r_j$
  else
    $y_j \leftarrow r_j + \gamma \hat{Q}(s_{j+1}, \underset{a' \in \tilde{\mathcal{A}}}{\operatorname{argmax}} Q(s_{j+1}, a'|\theta_{t-1})|\hat{\theta}_{t-1})$
  end if
  Perform a gradient descent step on θ per loss function $\big(y_j - Q(s_j, a_j|\theta)\big)^2$ and get θ_t.
  if $t \mod C = 0$ **then**
    $\hat{\theta}_t \leftarrow \theta_t$
  end if
  Perform experience augmentation described in Section III-E.
end for
return Q

deployed in a multi-agent environment in a similar way as in [20]. At each decision point (dispatching), we assign the orders collected within a dispatching window to a set of drivers to maximize the total value of the assignments.

$$\arg\max_{a \in \mathcal{A}'} \sum_{s \in \mathcal{S}} Q(s, a(s)), \tag{5}$$

where $a(s)$ is an assignment function that assigns an order from the pool to a driver s; $\mathcal{A}'$ is the space of all assignment functions for the order pool; and $\mathcal{S}$ is the set of available free drivers (and their states). The matching part can be solved by a standard matching algorithm, such as the Hungarian Method (a.k.a. KM algorithm). Specifically, we use the single-driver value function to compute the edge weights of the bi-partite graph for the KM algorithm. We call such a dispatching policy *collectively greedy* w.r.t. Q.

H. State values (V-values) in tabular form

In [20], a tabular-form state value function $V(\cdot)$ is learned to compute the edge weights corresponding to the trip assignment $x = (s, a, r, s')$, where s and s' are without contextual features. As $r + V(B(s'))$ is a sample approximation of $Q(s, a)$, $A_x := r + V(B(s')) - V(B(s))$ is the advantage associated with the trip assignment x and is used as the edge weights. We recall that $B(s)$ is the spatio-temporal bin associated with s.

Our method is able to directly leverage the above framework. We can generate the tabular V-function from the learned Q-function as follows: For every spatio-temporal cell $B(s)$ with cell center s, $V^*(B(s)) := \max_{a \in \tilde{\mathcal{A}}} Q^*(s, a)$, for Q^* learned by Algorithm III.2, and $V^\pi(B(s)) := \text{mean}_{a \in \tilde{\mathcal{A}}} Q^\pi(s, a)$, for Q^π obtained by following III-F.

IV. Multi-city Transfer

The dispatching system has to take charge of the orders across a large number of cities. Training a single dispatching agent that covers all cities is computationally prohibitive and has limited flexibility in deployment. Furthermore, if we treat the entire dispatching as a set of independent optimization problems for different cities, then the computation cost is also quite demanding. For example, with a 6-core CPU, single GPU computing resource setting, optimizing a dispatching policy model for a mid-sized city in China using one month of data will require around 30 hours to converge. Under such scenarios, transferring prior knowledge (i.e. reusing previously trained models) may be a feasible solution. More importantly, because of the non-convex property, deep learning would suffer from local optima, and if the training starts from a better initial point or follows a better gradient direction, which shall come from the knowledge transfer, it would probably reach a higher convergence performance.

Traffic patterns would not be exactly the same across different cities because of the urban structures, development levels and many other factors. However, they could possibly share some common characteristics, such as the rush hours of morning and evening traffic and trip demands from residential areas to commercial districts. Taking advantages of the approximation function (i.e. non-linear network) of the DQN, we know how to construct the policy model. In other words, the formulation of DQN gives us many flexible ways of transferring learned policies. This is also an advantage over [20] For a tabular form value function, no function approximator is involved, so the learned policy of each city is so unique that it is not applicable for knowledge transfer.

We consider three methods to improve the training on the target cities, including fine-tuning, progressive network, and correlated-feature progressive transfer (CFPT). The common idea of these transfer methods is to employ trained network weights learnt from the source city. Network structures are shown in Figures 3 and 4.

Fine-tuning [26] remains a popular method for transfer learning. After training the network on the source city, we transfer the weights to the target city's network. As shown in Figure 3, we initialize the weights of all fully-connected layers with the weights learnt from the source city, and leave them trainable on the target city data. Then, the network is fine-tuned via backpropagation.

Progressive network [15] leverages the trained weights via lateral connections to the target network. The connection function is defined as:

$$h_i^{(t)} = f\left(W_i^{(t)} h_{i-1}^{(t)} + U_i^{(c)} h_{i-1}^{(s)}\right), \tag{6}$$

where $W_i^{(t)}$ denotes the weight matrix of layer i of the target network, and $U_i^{(c)}$ denotes the lateral connection weight matrix from the network of the source tasks. $h_i^{(t)}$ and $h_i^{(s)}$ are the outputs of layer i in the target network and the source network, respectively. $f(\cdot)$ is the activation function. To perform the weights transfer, we first train a source network as in Figure 1, and then connect it to the target network as in Figure 3. We also prone the output layer (the semi-transparent circle) of the source network during transfer.

CFPT: Due to the variety of state space, not all state elements are adaptive across different cities. Instead of using a fully-connected network which takes all state elements as an entirety during training, we build and train a parallel progressive structure as shown in Figure 4 for the source city, and the connection is the same as that in 6. In addition, the network input is also separated into two parts: s denotes those elements intuitively not adaptable for the target city, and f denotes those adaptable. During training for the source city, all the weights of the network are trainable. For the target city, we build the network with the same structure and particularly reuse the weights of the progressive part (shown as the green blocks in Figure 4) from the source city's model that takes f as the direct input. The major novelty of CFPT is: during the training stage of the source city, we already split the network into to parallel flows, where the bottom flow (for future transfer) in Figure 4 only takes care of the input f. The number of neurons within each same-level layers of the two flows is half of the original fully-connected network, and this will substantially reduce the number of total trainable parameters.

In this work, we define the correlated feature input f as the concatenation of spatio-temporal displacement vector and real-time contextual features. Using the notations in Section II, the 3-tuple spatio-temporal displacement vector is computed as $(s_1 - s_0)$. The 5-tuple contextual feature vector contains real-time event counts, such as real-time number of idle drivers, and real-time number of orders created in the past 1 minute. Compared with the absolute GPS locations, the above correlated features are related to the statistics of dispatching demand, supply, and order fulfillment. Generally speaking, they could be more adaptive as inputs across different cities, and that is why we split the original network input into two sub-spaces. Figure 3 illustrates the input space difference among the above three methods. Notice that $s = (s_0, s_1)$ already includes the action input $\mathcal{A}$ (trip destination), which is defined in Section II.

V. Experiments

In this section, we will discuss the experiment settings and results. We use historical ExpressCar trip data obtained from the DiDi dispatching platform as our training data. The dataset is divided into training set $(2/3)$ and testing set $(1/3)$. Each

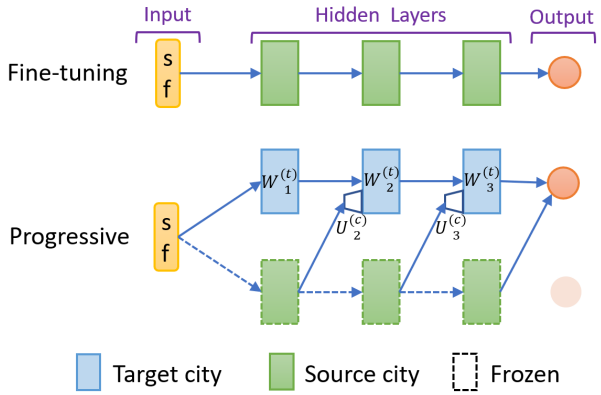


Fig. 3: Structures of finetuning and progressive network. We initialize the green blocks with trained weights from the source city. Frozen layers would keep the transferred weights during the target training.

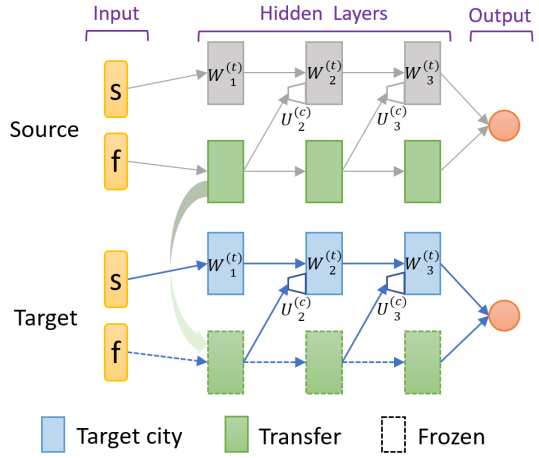
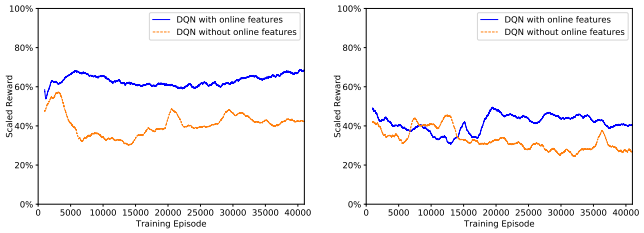


Fig. 4: CFPT network: we separate the input space and only transfer network weights shown as green blocks.

mini-batch can be viewed as a small set of sample points on the Q (state-action) value function. We used a discount factor $\gamma = 0.9$. For all DQN-based methods, we used a replay buffer with a size of 100000. We normalized all state vectors with their population mean and standard deviation. We found that this pre-processing is necessary for a stable training. For the training results, we use a sliding window of 1000 episodes to calculate the reward curve and the total training duration is 40000 episodes. For the testing results, we set five testing points during training: 0% , 25%, 50%, 75%, and 100%. At each checkpoint of training, we take a snapshot of the current network and evaluate it on the testing dataset for 5 trials of 100 episodes with random initial states.

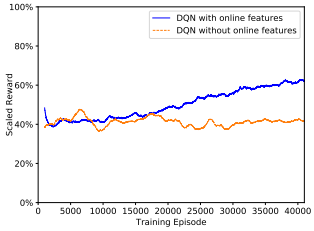
A. Single-agent evaluation environment

Since the transitions data is entirely historical and they do not necessarily form complete episodes, we build a single-driver dispatching environment from the past trip data (testing set) for a direct and explicit evaluation of the policy generated from the learned value function. Basically, we assume that after

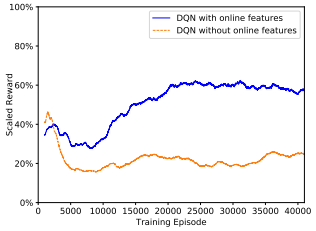


(a) City A

(b) City B



(c) City C



(d) City D

Fig. 5: Comparison between the two types of the inputs: original spatio-temporal state and state with expanded contextual features.

a driver drops off a passenger at the destination, he/she would be assigned a new trip request starting from the vicinity of the previous destination. The search space can be augmented as necessary following Section III-C to cover the cases where there are no historical trips around the last drop-off area. The next move is selected by the given policy from the action search output, which could be a combination of fee-generating trips or wait/reposition actions. The reward associated with the action is the actual trip fee if the action involves a historical trip; otherwise the reward is zero (for waiting or repositioning). We use the scaled reward percentage (with the same normalization constant for a given city) as the performance metric. We run the simulation for multiple episodes (days), and the cumulative reward gained is computed and averaged over the episodes. For plotting the training curves, we step the agent through this environment using a greedy policy with respect to the learned Q-value function at the same pace as the training steps. In addition to the original spatial-temporal input space, we find that the contextual features can benefit the training, which is shown in Figure 5. Hence in the following sections, we use the expanded state space as network input.

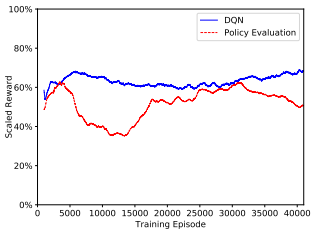
B. Baselines DQN training

We build the Q-network with three hidden dense layers and ReLU activations and trained dispatching policies for four cities in China, denoted by A, B, C, and D, respectively. They span different sizes of population and geographical regions. We summarize the characteristics of the four cities in Table I.

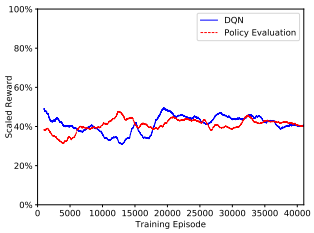
To show the optimization improvement of Algorithm III.2 as our baseline method, we first benchmark it with the policy evaluation mentioned in Section III-F. Figure 6 compares the training curves of DQN and policy evaluation, where the former is maximizing the accumulative rewards, and the latter is

TABLE I: Basic characteristics of the four Chinese cities in the experiments.

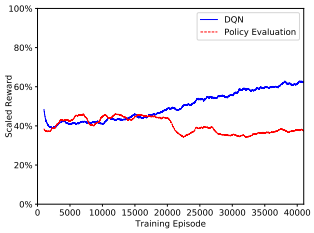
City	Size	Region
A	Large	Northern
B	Small	Southern
C	Medium	Southern
D	Large	Western



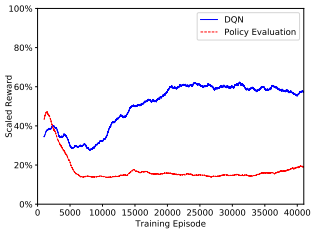
(a) City A



(b) City B



(c) City C

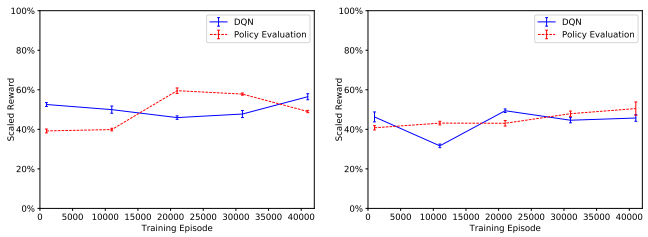


(d) City D

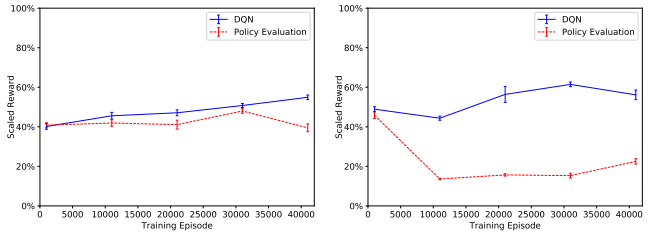
Fig. 6: Training curves of DQN with action search on the four cities.

just learning to evaluate the current policy that generates the historical data as described in Section V-A. The testing curves with standard deviation error bars are shown in Figure 7.

As we can see from the performance curves, our Algorithm III.2 is capable of learning to optimize the episode order revenue. Figure 8 shows the average Q values of training mini-batches, where we have observed convergence of the value functions in all cases. We compute the tabular-form state value function as described in Section III-H and plot the state values of 120 randomly sampled location cells by time id in Figure 9: each point indicates the potential future rewards discounted with $\gamma = 0.9$, so the state value is computed within a decaying future horizon. We see that the state value function correctly captures the decreasing monotonicity in the temporal space of the discounted cumulative rewards. However, the learning results, or improvements are not the same among different cities. In our experiments, we find that for those smaller cities with less trip data (due to fewer users in such areas), e.g. B in Figure 6b, the optimization improvement is not as significant as larger cities, such as City D. This is because in such cities with lighter order demand, the trip patterns are much simpler, and our current order dispatching system [8] could nearly achieve the optimal match between the drivers and the customers. It also indicates that there is not much potential gain for the policy improvement. However, in those cities with heavier dispatching load and lager trip data amount, the optimization improvement is much more obvious, such City D in Figure 6d.

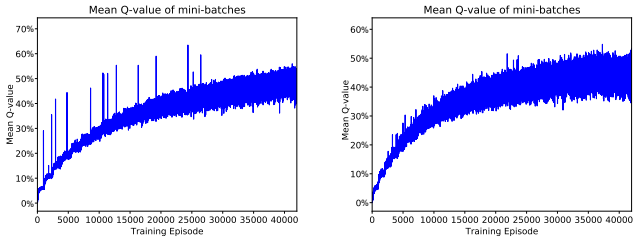


(a) City A (b) City B

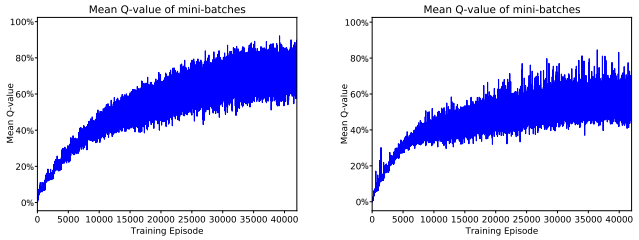


(c) City C (d) City D

Fig. 7: Testing evaluation of DQN with action search on the four cities, at 5 different checkpoints on the training phase.



(a) City A (b) City B



(c) City C (d) City D

Fig. 8: Average Q values of the training mini-batches.

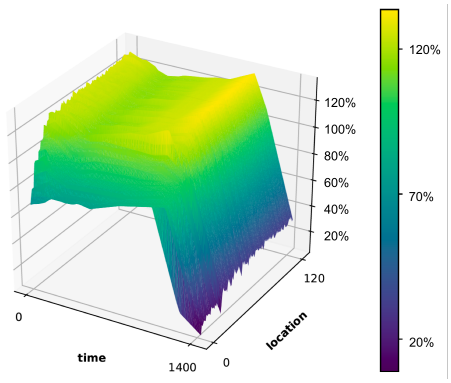


Fig. 9: V-values of 120 sampled location bins in City D.

C. Transfer improvements

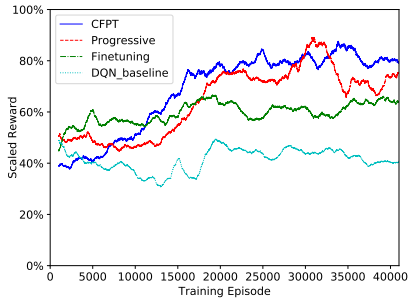
To achieve robust and effective network training on the target cities, we performed two types of transfer experiments, including spatial transfer and temporal transfer. For spatial transfer, among the four experiment cities mentioned in the previous section, City A is used as the source city, while the other three are used as the target cities. For temporal transfer, the city models trained on one month of data are used as the source, while the models of the same cities trained on a later month of data are used as the target. We use the trained network weights from the previous section V-B as the prior knowledge. Under each type of transfer experiments, we will compare and discuss the learning performance of the three transfer methods, including finetuning, progressive network, and correlated-feature progressive transfer (CFPT).

Training curves and testing checkpoints are shown in Figure 10. To highlight the results, we show the average of the accumulated episode reward sampled from the historical trip data in the real dispatching system. Comparing to the vanilla DQN training, we could see the following improvements: 1) Target cities will benefit from jumpstart, which is the increased initial performance at the start; 2) Learning is more efficient, which means the convergence would happen earlier during training; 3) The convergence performance is better. However, the effectiveness of the three methods are different. In particular, CFPT outperforms the other two transfer methods, indicating that the online features f can be more helpful for the learning adaption across different cities if used separately from the absolute spatio-temporal states (l, t). The weakness of the original progressive network is that it does not consider the correlation difference among all state/feature elements. Because of the catastrophic forgetting property of finetuning, the prior knowledge will easily be forgotten along with the update of weights, and we also see less significant improvement on the convergence performance. Temporal transfer results are shown in Figure 11. Although the transfer is performed within the same city, CFPT would significantly outperform the baseline finetuing method, which directly copies the network from a previous month and continue training.

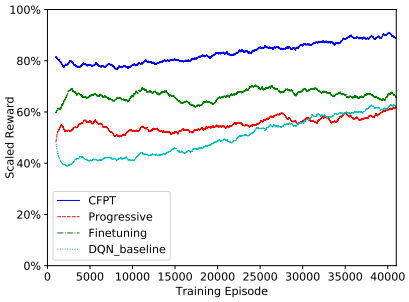
To get insight into how the knowledge transfer could improve the training in the target cities, we compare the average Q values in each batch during training. Taking City D as an example, we could find the distinct difference between the Q-value curves in Figure 12. For the original DQN training, there is still noticeable variance even though the mean Q value almost converges. In contrast, for CFPT, the batch's variance is much smaller. Such difference may indicate that through the lateral connection from the previous trained network, the direction of "gradient decent" in the target training is more explicit. In other words, transferring the prior knowledge may efficiently guide the learning direction.

VI. Conclusion

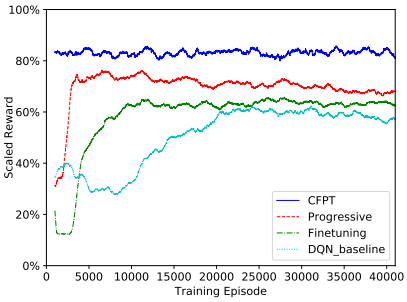
This paper has proposed an adapted DQN-based optimization method for order revenue on the DiDi ride-dispatching platform. Different from the vanilla DQN with enumerable output actions,



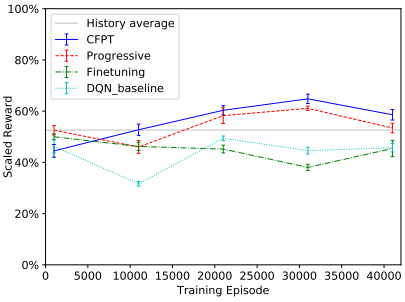
(a) City B



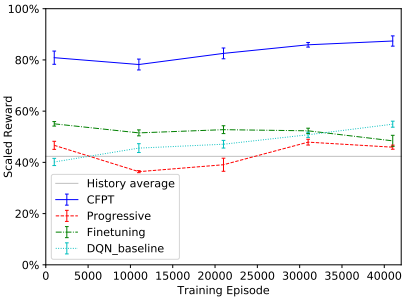
(b) City C



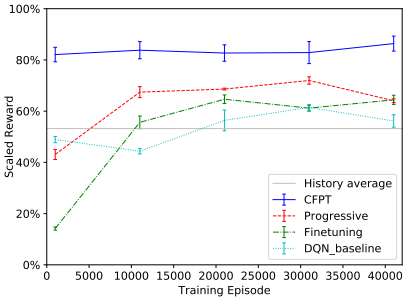
(c) City D



(d) City B

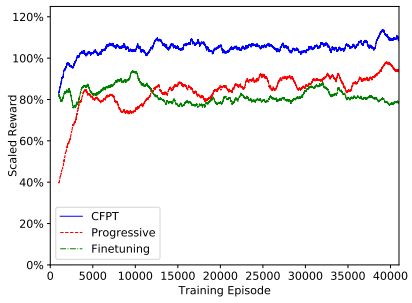


(e) City C

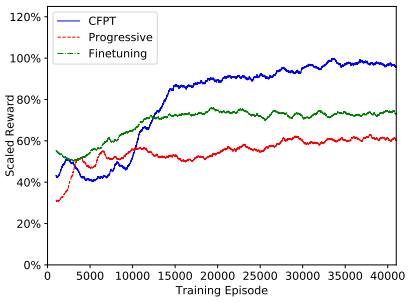


(f) City D

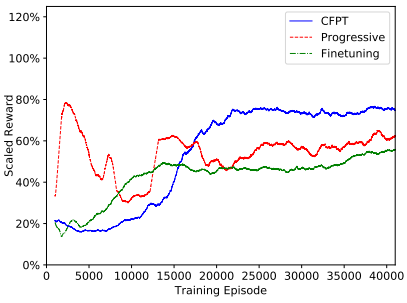
Fig. 10: Training and testing curves of the spatial transfer. (Top: training curves; bottom: testing rewards at different checkpoints during the training.)



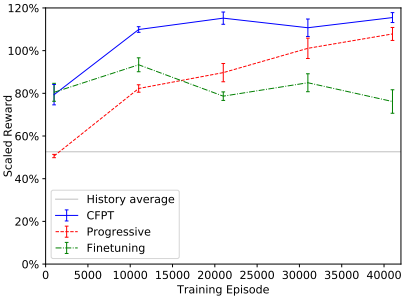
(a) City B



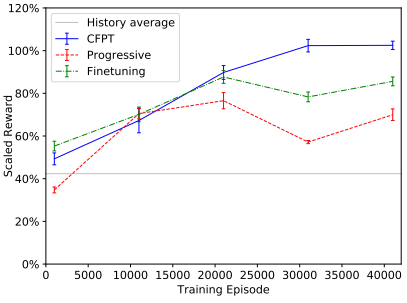
(b) City C



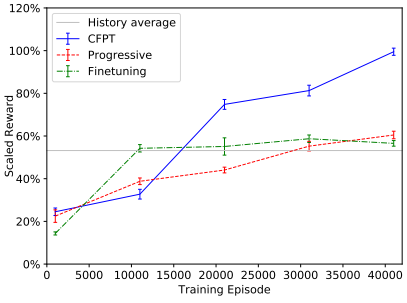
(c) City D



(d) City B

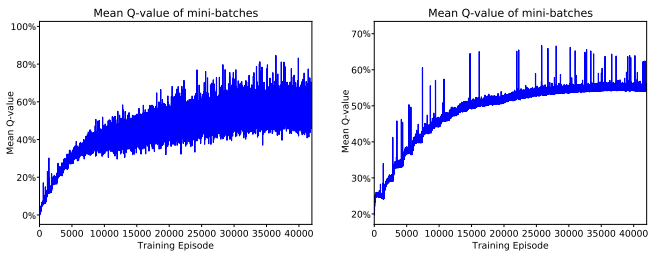


(e) City C



(f) City D

Fig. 11: Training and testing curves of the temporal transfer. (Top: training curves; bottom: testing rewards at different checkpoints during the training.)



(a) DQN training (b) CFPT training

Fig. 12: Comparison of the average mini-batch Q values for City D.

we encode the continuous action space (trip destination) as part of the input state space and provide a corresponding action search method. We show that our application is capable of optimizing the spatio-temporal problem from a single driver's point of view. By showing the diverse learning results due to the variety of cities' traffic patterns, we know that the learning process is not trivial across different cities. As the solution to combating such diversity, we have evaluated two existing transfer learning methods: finetuning and progressive network, and propose one online-feature based adaption method – CFPT. Results show that reusing trained models can speed up the learning and improve the robustness when dealing with new patterns. In particular, by focusing on the correlated features across different domain, CFPT can achieve the most effective transfer and outperform the other methods.

Our proposed optimization approach is from a single driver's standpoint with a local view. To overcome the stationary environment assumption, we can learn the dispatching policy using a multi-agent reinforcement learning method. We can also train a global value function to learn a centralized policy. We leave these ideas as future directions of research.

ACKNOWLEDGEMENTS

We would like to thank Satinder Singh and Zhe Xu for insightful discussions, and the anonymous reviewers for their constructive comments.

REFERENCES

[1] L. Moreira-Matias, J. Gama, M.-M. J. Ferreira, Michel, and L. Damas, "On predicting the taxi-passenger demand: A real-time approach," in *Portuguese Conference on Artificial Intelligence*. Springer, 2013, pp. 54–65.

[2] L. Moreira-Matias, J. Gama, M. Ferreira, J. Mendes-Moreira, and L. Damas, "Predicting taxi–passenger demand using streaming data," *IEEE Transactions on Intelligent Transportation Systems*, vol. 14, no. 3, pp. 1393–1402, 2013.

[3] Q. Li, Z. Zeng, B. Yang, and T. Zhang, "Hierarchical route planning based on taxi gps-trajectories," in *Geoinformatics, 2009 17th International Conference on*. IEEE, 2009, pp. 1–5.

[4] T. Xin-min, W. Yu-ting, and H. Song-chen, "Aircraft taxi route planning for a-smgcs based on discrete event dynamic system modeling," in *Computer Modeling and Simulation, 2010. ICCMS'10. Second International Conference on*, vol. 1. IEEE, 2010, pp. 224–228.

[5] J. Lee, G.-L. Park, H. Kim, Y.-K. Yang, P. Kim, and S.-W. Kim, "A telematics service system based on the linux cluster," in *International Conference on Computational Science*. Springer, 2007, pp. 660–667.

[6] A. Glaschenko, A. Ivaschenko, G. Rzevski, and P. Skobelev, "Multi-agent real time scheduling system for taxi companies," in *8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009), Budapest, Hungary*, 2009, pp. 29–36.

[7] D.-H. Lee, H. Wang, R. Cheu, and S. Teo, "Taxi dispatch system based on current demands and real-time traffic conditions," *Transportation Research Record: Journal of the Transportation Research Board*, no. 1882, pp. 193–200, 2004.

[8] L. Zhang, T. Hu, Y. Min, G. Wu, J. Zhang, P. Feng, P. Gong, and J. Ye, "A taxi order dispatch model based on combinatorial optimization," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2017, pp. 2151–2159.

[9] C. H. Papadimitriou and K. Steiglitz, *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998.

[10] T. Verma, P. Varakantham, S. Kraus, and H. C. Lau, "Augmenting decisions of taxi drivers through reinforcement learning for improving revenues," in *International Conference on Automated Planning and Scheduling*, 2017, pp. 409–417.

[11] M. E. Taylor and P. Stone, "Transfer learning for reinforcement learning domains: A survey," *Journal of Machine Learning Research*, vol. 10, no. Jul, pp. 1633–1685, 2009.

[12] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on knowledge and data engineering*, vol. 22, no. 10, pp. 1345–1359, 2010.

[13] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska *et al.*, "Overcoming catastrophic forgetting in neural networks," *Proceedings of the National Academy of Sciences*, vol. 114, no. 13, pp. 3521–3526, 2017.

[14] Y. Teh, V. Bapst, W. M. Czarnecki, J. Quan, J. Kirkpatrick, R. Hadsell, N. Heess, and R. Pascanu, "Distral: Robust multitask reinforcement learning," in *Advances in Neural Information Processing Systems*, 2017, pp. 4499–4509.

[15] A. A. Rusu, N. C. Rabinowitz, G. Desjardins, H. Soyer, J. Kirkpatrick, K. Kavukcuoglu, R. Pascanu, and R. Hadsell, "Progressive neural networks," *arXiv preprint arXiv:1606.04671*, 2016.

[16] E. Parisotto, J. L. Ba, and R. Salakhutdinov, "Actor-mimic: Deep multitask and transfer reinforcement learning," *arXiv preprint arXiv:1511.06342*, 2015.

[17] I. Higgins, A. Pal, A. A. Rusu, L. Matthey, C. P. Burgess, A. Pritzel, M. Botvinick, C. Blundell, and A. Lerchner, "Darla: Improving zero-shot transfer in reinforcement learning," *arXiv preprint arXiv:1707.08475*, 2017.

[18] A. Maurer, M. Pontil, and B. Romera-Paredes, "The benefit of multitask representation learning," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 2853–2884, 2016.

[19] Z. Luo, Y. Zou, J. Hoffman, and L. F. Fei-Fei, "Label efficient learning of transferable representations acrosss domains and tasks," in *Advances in Neural Information Processing Systems*, 2017, pp. 164–176.

[20] Z. Xu, Z. Li, Q. Guan, D. Zhang, W. Ke, Q. Li, J. Nan, C. Liu, W. Bian, and J. Ye, "Large-scale order dispatch in on-demand ride-sharing platforms: a learning and planning approach," in *Proceedings of the 24rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2018.

[21] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3-4, pp. 279–292, May 1992.

[22] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[23] Z. Wang and M. E. Taylor, "Improving Reinforcement Learning with Confidence-Based Demonstrations," in *Proceedings of the 26th International Conference on Artificial Intelligence (IJCAI)*, August 2017.

[24] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning." in *AAAI*, 2016, pp. 2094–2100.

[25] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press Cambridge, 1998, vol. 1, no. 1.

[26] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *science*, vol. 313, no. 5786, pp. 504–507, 2006.