Effective Travel Time Estimation: When Historical Trajectories over Road Networks Matter

Haitao Yuan[†] Guoliang Li[†,*] Zhifeng Bao[‡] Ling Feng[†]
[†]Department of Computer Science, Tsinghua University, China [‡]RMIT University, Australia
[†]{yht16@mails.,liguoliang@,fengling@}tsinghua.edu.cn [‡]zhifeng.bao@rmit.edu.au

ABSTRACT

In this paper, we study the problem of origin-destination (OD) travel time estimation where the OD input consists of an OD pair and a departure time. We propose a novel neural network based prediction model that fully exploits an important fact neglected by the literature – for a past OD trip its travel time is usually affiliated with the trajectory it travels along, whereas it does not exist during prediction. At the training phase, our goal is to design novel representations for the OD input and its affiliated trajectory, such that they are close to each other in the latent space. First, we match the OD pairs and their affiliated (historical) trajectories to road networks, and utilize road segment embeddings to represent their spatial properties. Later, we match the timestamps associated with trajectories to time slots and utilize time slot embeddings to represent the temporal properties. Next, we build a temporal graph to capture the weekly and daily periodicity of time slot embeddings. Last, we design an effective encoding to represent the spatial and temporal properties of trajectories. To bind each OD input to its affiliated trajectory, we also encode the OD input into a hidden representation, and make the hidden representation close to the spatio-temporal representation of the trajectory. At the prediction phase, we only use the OD input, get the hidden representation of the OD input, and use it to generate the travel time. Extensive experiments on real datasets show that our method achieves high effectiveness and outperforms existing methods.

CCS CONCEPTS

• **Information systems** → *Spatial-temporal systems.*

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3389771

KEYWORDS

OD travel time estimation; trajectory; road networks

ACM Reference Format:
Haitao Yuan[†] Guoliang Li[†,*] Zhifeng Bao[‡] Ling Feng[†]. 2020. Effective Travel Time Estimation: When Historical Trajectories over Road Networks Matter. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3318464.3389771

1 INTRODUCTION

With the advent of ride-hailing services (e.g., 30 million and 18 million ride orders per day in Didi [2] and Uber [6] respectively), one of the most important operators is: given an OD input that consists of an origin point, a destination point and a departure time, how to estimate its travel time.

Hence, how to provide an effective OD travel time estimation has drawn extensive attentions. Essentially, methods used to solve this problem include non-learning methods [39] and learning-based methods [23, 27]. We review the following phylogeny of existing techniques on OD travel time prediction. At first, the experiments of [39] show that the non-learning method `TEMP` is better than the basic learning method (linear regression `LR`) and other non-learning methods. Later, the experiments in [23] show that the deep learning method `STNN` beats basic learning methods (`LR` and the gradient boosting decision tree based regression `GBM`) and the non-learning method `TEMP`. Most recently, the authors of [27] also propose a deep learning method `MURAT` and demonstrate that deep learning methods are better than other methods. Thus, we decide to employ deep learning to solve this problem and our experiments also confirm the superiority of deep learning methods.

We observe that for a given OD input of a past trip, its travel time is actually affiliated with the trajectory it travelled along, which is fairly useful in travel time estimation, as shown in Example 1. Unfortunately, to our best knowledge, trajectories of historical trip records have not been fully exploited by any existing deep learning work. However, it is not trivial to design a prediction model that can fully utilize

[*]Guoliang Li is the corresponding author.

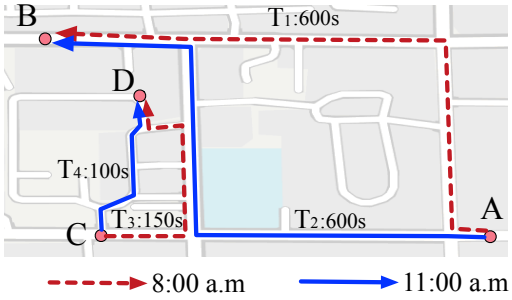


Figure 1: Different Trajectories for the Same OD Pair

such trajectories, due to two reasons. First, given a historical trip record, it remains a key challenge to design novel representations for the OD input and its affiliated trajectory respectively, such that they can be close to each other in the latent space. Second, such trajectories are only available in the model training phase (for past trips), while it does not exist in the prediction phase (for a future trip request).

Example 1. *As shown in Figure 1, for the same OD pair (A, B) at two departure timestamps (8:00 am and 11:00 am), there are two different historical trajectories T_1 and T_2, which have the same travel time. Existing methods [23, 27] would learn similar representations for these two trips, because they do not consider the difference of the trajectories and thus cannot distinguish these two trips. Therefore, for another OD pair (C, D), they would predict that the travel time is the same for both cases (T_3 and T_4), which is not true in real cases.*

To address the above challenges, we design a novel neural network based method DeepOD, aiming to harness the power of historical trajectories and road networks to achieve effective OD travel time estimation.

As for the representation of the OD input, its spatial information mainly includes GPS points (e.g., origin points, destination points). Since vehicles can only travel on road networks, we match the GPS points onto road segments and then use road segment embeddings to represent the points. Its temporal information consists of timestamps (e.g. the departure time), where different timestamps indicate different traffic conditions and thereby influence the travel time. Considering the neighboring smoothness and weekly periodicity of traffic conditions, we split one-week period into disjoint time slots. Then, we build a temporal graph where each node denotes a time slot and each edge denotes either a neighboring time slot or a neighboring day, and initialize the embeddings for time slots. For each timestamp, we first map it into a time slot and then use the embedding of the time slot to represent that timestamp. Later, we concatenate the representations of the spatial and temporal information, and convert them into a final representation of the OD input. In addition, it is worth mentioning that our model can incorporate more external features (if available) associated with the OD input, such as weather and traffic condition.

As for the representation of the trajectory associated with the OD input, we define a concept called *spatio-temporal path*, to jointly represent the spatial and temporal properties. The spatio-temporal path is a sequence, in which each element contains a road segment and a time interval. We use the road segment embeddings to represent the road segment, and design an encoding model to convert the time interval into a hidden representation based on the time slot embeddings. Finally, we concatenate the representations of the road segment and the time interval, and apply a sequence model to get the final representation.

As for applying the trained model, when given an OD input, we first generate its representation and then use it to generate the travel time. The process of generating this representation is analogous to generating a proper trajectory and thus the estimation would be more accurate.

In summary, we make the following contributions:
(1) We design a comprehensive neural network model, DeepOD, that can fully exploit historical trajectories, road networks and external data (e.g., weather and traffic condition) for travel time estimation (Section 3).
(2) We propose effective methods to generate hidden representations for spatial and temporal features in an OD input by exploiting the road network structure and the daily and weekly periodicity (Section 4.1-Section 4.3).
(3) We propose spatio-temporal paths to jointly represent the spatial and temporal features of a trajectory, and design an effective encoding model to generate the representation for the trajectory, which can be close to the representation of its corresponding OD input in the latent space (Section 4.4).
(4) We present an algorithm to illustrate the offline training process and the online estimation process (Section 5).
(5) We conduct a comprehensive evaluation on two real world datasets. The results show that our method outperforms existing approaches significantly (Section 6).

2 PROBLEM FORMULATION

Road Network. A road network is modeled as a directed, weighted graph $G = \langle V, E \rangle$, where V is a vertex set and E is an edge set. Each edge $e_k \in E$ represents a road segment while each vertex $v_i \in V$ denotes an end point of a road segment. Then, e_k can be represented as $\langle v_k^1 \rightarrow v_k^{-1}, w_k \rangle$, where $v_k^1 \in V$ is the first end point, $v_k^{-1} \in V$ is the last end point and w_k represents the weight (e.g. road length). For simplicity, we denote e_k as $\langle v_k^1, v_k^{-1} \rangle$.

Trajectory. A raw trajectory is a sequence of GPS points, and each point is denoted as $\langle [x_i, y_i], t_i \rangle$, where $g[i] = [x_i, y_i]$ denotes the spatial position and t_i denotes the timestamp. Since trajectory points should be on a road network, we align trajectories to road segments using existing map-matching

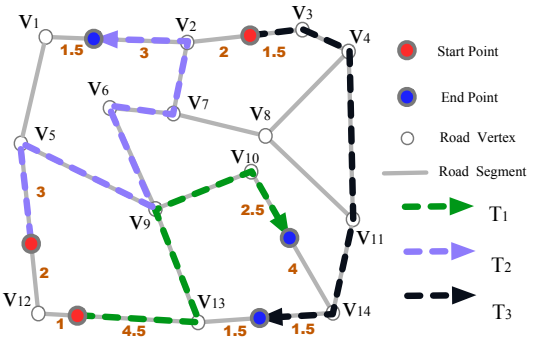


Figure 2: Example of Trajectories on a Road Network

algorithms [9, 43] that are widely adopted in trajectory analytics [40, 45]. Also, there is a time interval when a trajectory T passes through a road segment e_i. We denote the time interval as $[t_i[1], t_i[-1]]$, where $t_i[1]$ and $t_i[-1]$ represent the start and end timestamp, respectively. Specifically, we use the linear interpolation technique to calculate $t_i[1]$ and $t_i[-1]$. To this end, we use $\langle e_i, [t_i[1], t_i[-1]]\rangle$ to represent the road segment e_i of a trajectory. For a trajectory, since its origin point $g[1]$ and destination point $g[-1]$ may occur in the middle of edges, we use two ratios $r[1] = \frac{|v_1^1 \to g[1]|}{|v_1^1 \to v_1^{-1}|}$ and $r[-1] = \frac{|g[-1] \to v_{-1}^{-1}|}{|v_{-1}^1 \to v_{-1}^{-1}|}$ to capture the exact positions of $g[1]$ and $g[-1]$ in road segments $\langle v_1^1, v_1^{-1}\rangle$ and $\langle v_{-1}^1, v_{-1}^{-1}\rangle$ respectively. $|\cdot \to \cdot|$ denotes the distance between two points.

Definition 1 (Trajectory). *A trajectory on a road network contains two parts: a spatio-temporal path and two position ratios. The spatio-temporal path is a sequence of tuples. Each tuple is composed of a road segment and a time interval. We denote the spatio-temporal path by* $SP = \langle\langle e_1, [t_1[1], t_1[-1]]\rangle, \cdots, \langle e_n, [t_n[1], t_n[-1]]\rangle\rangle$, *and denote the two position ratios by* $PR = \langle r[1], r[-1]\rangle$. *Then, a trajectory is denoted as* $\langle SP, PR\rangle$.

Example 2. *Figure 2 contains three trajectories* $\{T_1, T_2, T_3\}$. T_1*'s spatio-temporal path is* $SP_{T_1} = \langle\langle\langle v_{12}, v_{13}\rangle, [t_1^1[1], t_1^1[-1]]\rangle, \cdots, \langle v_{10}, v_{14}\rangle, [t_4^1[1], t_4^1[-1]]\rangle\rangle$; *its spatial position ratio is* $PR_{T_1} = \langle \frac{1}{5.5}, \frac{2.5}{6.5}\rangle$. *For* T_2, $SP_{T_2} = \langle\langle\langle v_{12}, v_5\rangle, [t_1^2[1], t_1^2[-1]]\rangle, \cdots, \langle v_2, v_1\rangle, [t_6^2[1], t_6^2[-1]]\rangle\rangle$ *and* $PR_{T_2} = \langle \frac{2}{5}, \frac{3}{4.5}\rangle$.

Definition 2 (OD Input). *Essentially, an OD input consists of three parts: an origin point* ($g[1]$), *a destination point* ($g[-1]$) *and a departure time* (t). *Optionally, we use* f *to denote the external features (if available) that may influence the travel time, which is the fourth part.*

Definition 3 (OD Travel Time Estimation). *Given a road network* $G = \langle V, E\rangle$ *and historical trajectories* $\mathcal{T}$ *on the road network, the problem of estimating the travel time for a given OD input is called the OD travel time estimation problem.*

An intuitive solution is to fit a function to predict the travel time based on the given input. However, the actual trajectories associated with a certain OD input is only available

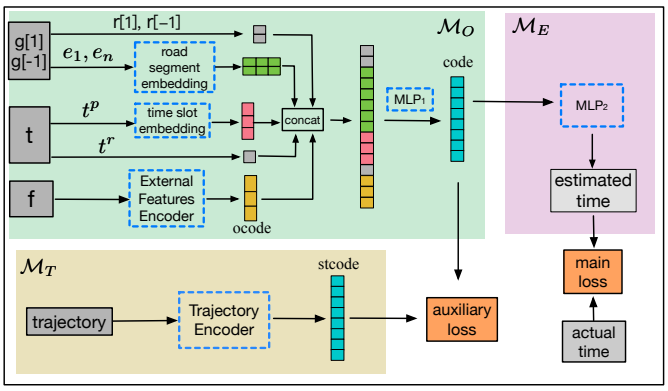


Figure 3: The Model Architecture of DeepOD

Table 1: Notations

Notation	Description
$\mathcal{M}_O, \mathcal{M}_T, \mathcal{M}_E$	three modules in DeepOD
$g[1], g[-1]$	spatial positions of GPS points
$r[1], r[-1]$	position ratios of GPS points on roads
$t, t_i, t_i[1], t_i[-1]$	timestamps
$[t_i[1], t_i[-1]]$	time intervals
$t^p, t^p[1], t^p[-1]$	time slots
Δt	the size of each time slot
$t^r, t^r[1], t^r[-1]$	time remainders
O_i^s, O_i^t	one-hot encodings of roads and time slots
D_i^s, D_i^t, D_i^{st}	dense encodings of roads and time slots
$\mathbf{W_s}, \mathbf{W_t}$	embedding matrices
$\mathbf{W_m^i}, b_m^i$	matrix and bias vector parameters of MLPs
$\mathbf{W_f}, \mathbf{W_i}, \mathbf{W_o}, \mathbf{W_c}$	matrix parameters of the LSTM model
h_j, c_j	state vectors of the LSTM model
b_f, b_i, b_o, b_c	bias vector parameters of the LSTM model
K^1, K^2, K^3	kernel parameters of the CNN model
$\mathbf{Z^i}$	hidden tensor, matrix or vector representations
d_s, d_t	the second dimensional size of $\mathbf{W_s}, \mathbf{W_t}$
d_m^i	the first dimensional size of $\mathbf{W_m^i}$ and b_m^i
d_h	the dimensional size of h_j and c_j

in the model training stage, while we have no actual routes in the estimation stage. To bridge this gap, we propose a novel neural network DeepOD that can be trained with trajectories and used to predict travel time without trajectories.

3 AN OVERVIEW OF OUR MODEL

Figure 3 presents the architecture of our proposed model DeepOD, which contains three modules:

(1) The first part, denoted as $\mathcal{M}_O$, represents the OD encoding model, aiming to extract the hidden representation *code* from the OD input. Here, an OD input consists of the origin point $g[1]$, the destination point $g[-1]$, the departure time t and external features f capturing the traffic condition. In particular, for $g[1]$ and $g[-1]$ that are two end points matched on road segments, we use the corresponding road segments (e_1, e_n) and position ratios $(r[1], r[-1])$ to represent them. Then we use the `road segment embedding` (in

Section 4.1) to convert e_1 and e_n into two fixed-length vectors. For the timestamp t, we represent it by a time slot t^p and a time remainder t^r to better extract temporal features, as defined in Section 4.2. Then we use the model `time slot embedding` (in Section 4.2) to encode the time slot into a fixed-length vector. Later, we design an encoding model `External Features Encoder` (in Section 4.5) to encode the external info f (if available) into a fixed-length vector. Lastly, we concatenate the above vectors and float values into a vector and then use a Multilayer Perceptron model ($\texttt{MLP}_1$) to encode the vector into a hidden representation *code*.

(2) The second part, denoted as $\mathcal{M}_T$, represents the trajectory encoding model, aiming to extract the spatio-temporal representation for a given trajectory. Specifically, we design an encoding model `Trajectory Encoder` (in Section 4.4) to learn the spatio-temporal representation *stcode*.

(3) The third part, denoted as $\mathcal{M}_E$, represents the travel time estimation model, aiming to generate the estimated travel time based on *code*. We utilize a Multilayer Perceptron model ($\texttt{MLP}_2$) to encode the vector into an estimated travel time. Afterwards, we use some loss functions to evaluate the difference between the estimated travel time and the actual travel time, and we denote the difference as *mainloss*.

Here, we would like to emphasize the following challenge: during the training stage, each training data (i.e., each historical trip record) consists of an OD input and an associated trajectory; however, in the test phase, only the OD input is available for prediction. To tackle this challenge, we design an auxiliary task, aiming to bimd each OD input to its corresponding trajectory when training the model. In particular, for each training input, we use $\mathcal{M}_O$ and $\mathcal{M}_T$ to encode the OD input and the trajectory into *code* and *stcode* respectively, and then bind *code* and *stcode* by minimizing their distance, which is denoted as *auxiliaryloss*. In contrast, we only use $\mathcal{M}_O$ to encode the OD input and then use $\mathcal{M}_E$ to estimate the travel time when applying the trained model. That is, we jointly train $\mathcal{M}_O$, $\mathcal{M}_T$ and $\mathcal{M}_E$ and only use $\mathcal{M}_O$ and $\mathcal{M}_E$ for estimating OD travel time. To this end, we summarize the notations in Table 1.

4 MODEL REPRESENTATION

Recall Section 3, `road segment embedding` aims to convert each road segment into a fixed-length vector, `time slot embedding` aims to convert each timestamp into a fixed-length vector, `External Features Encoder` aims to convert external features f into a fixed-length vector, and `Trajectory Encoder` aims to encode trajectory into a fixed-length vector. Here we will elaborate these four modules.

4.1 Road Segment Embedding

Embedding Matrix. Each road segment is identified by a unique id, while the input of any machine learning method

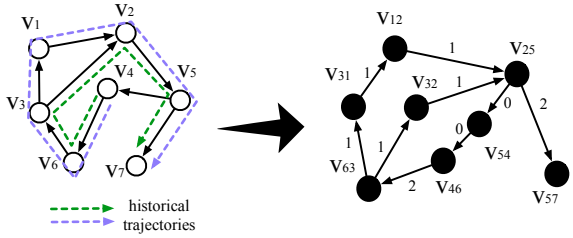


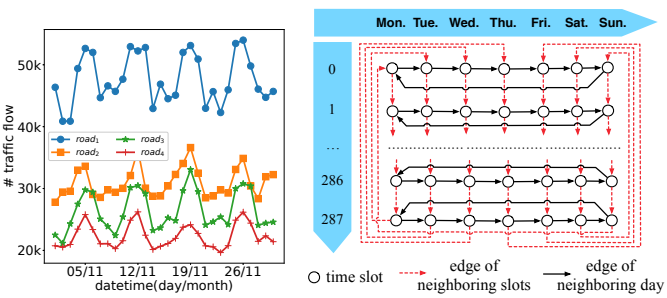
Figure 4: An example of converting a directed graph. In particular, if there are two edges $\langle v_i, v_k\rangle$ and $\langle v_k, v_j\rangle$ in the left graph, we would have an edge $\langle v_{ik}, v_{kj}\rangle$ in the right graph, where v_{ik} and v_{kj} respectively represent $\langle v_i, v_k\rangle$ and $\langle v_k, v_j\rangle$.

is usually a vector. One possible solution is to use a one-hot encoding to transform each road segment id into a $|E|$-dimensional vector, where the value of one particular dimension is 1 while the rest are 0. For example, if there are three road segments in E, their one-hot codes would be $[1, 0, 0]$, $[0, 1, 0]$ and $[0, 0, 1]$ respectively. However, one-hot representation is too sparse and the distance between any two one-hot codes is the same, so the distance between different road segments cannot be distinguished. To address this issue, we design a fully connected neural network to embed one-hot codes into dense vectors. Formally, the process is represented by the formula:

$$[D_1^s, D_2^s, \cdots, D_{|E|}^s]^{\mathsf{T}} = [O_1^s, O_2^s, \cdots, O_{|E|}^s]^{\mathsf{T}}\mathbf{W_s} \quad (1)$$

where $O_i^s \in \{0, 1\}^{|E|}$ represents the one-hot code of the i-th road segment, $D_i^s \in \mathbb{R}^{d_s}$ denotes the corresponding dense vector and $\mathbf{W_s}$ is the weight matrix of the fully connected neural network. In particular, the size of $\mathbf{W_s}$ is $|E| \times d_s$, where $d_s \ll |E|$. Therefore, we can learn the dense representation of each road network by learning $\mathbf{W_s}$ with training data.

Initialization of the Embedding. Considering that each road segment has influences on its linked road segments, adjacent road segments should have similar representations. However, the dense vector cannot capture the information of road network structure. Inspired by [27], we try to use some popular unsupervised graph embedding techniques (e.g., DeepWalk [30], Line [36], node2vec [16]) to generate the initial representation for each road segment. However, authors in [27] regard the road network as an undirected graph and they have not elaborated how to adopt these methods to embed road segments. In contrast, we find two issues are raised when using these graph embedding techniques. The first one is that these methods are designed to embed nodes while we need to embed edges. To address this issue, as illustrated in Figure 4, we convert the road network into a new graph, where each node represents a road segment. The second issue is how to measure link weights for the new graph, because link weights would influence the probability of randomly walking when using the above graph embedding methods. Intuitively, the probability is implicit in historical trajectories. Therefore, we count the co-occurrence frequency of two linked road segments on the same historical



(a) Weekly Periodicity (b) Temporal Graph

Figure 5: We select four roads from the city *Chengdu* of China. Figure (a) shows the weekly periodicity on the number of traffic flows on these four roads. Figure (b) shows the temporal graph for a week (five-minute period for each time slot). Each red directed line links two adjacent time slots while each black directed line links the same time slot in two adjacent days.

trajectories as link weights. Taking the new graph in Figure 4 as an example, the weight of $\langle v_{46}, v_{63} \rangle$ is set as 2 because both $\langle v_4, v_6 \rangle$ and $\langle v_6, v_3 \rangle$ are co-passed by two historical trajectories. Afterwards, we can use the aforementioned graph embedding methods to get the embeddings for all nodes in the new graph. Let W_i^0 be the embedding of the i-th road segment, we can thereby use the matrix $\mathbf{W_s}^0 = [W_1^0, \cdots, W_{|E|}^0]^\mathsf{T}$ to initialize the value of $\mathbf{W_s}$.

To summarize, we first use an unsupervised graph embedding method to initialize or pre-train the road segment embedding matrix and then fine-tune it by updating the weights based on supervised learning.

4.2 Time Slot Embedding

As aforementioned, the departure time t is a timestamp, and a temporal interval $[t_i[1], t_i[-1]]$ contains two end timestamps $t_i[1]$ and $t_i[-1]$. Therefore, we need to extract the temporal features from t. An intuitive approach is to regard each timestamp as a float value and then design a Multilayer Perceptron model to convert the value into a fixed-length vector. However, this approach has two drawbacks. First, each timestamp is usually a large integer number, so the direct use of timestamp would dominate other features. Second, there are weekly and daily periodicity between different timestamps, which cannot be captured by the timestamp alone. For example, the traffic conditions at the rush hour for different weekdays may be similar. To address this issue, we propose a new method to represent timestamps.

Time Slot. First, we normalize the timestamps by converting them into discrete time slots as defined below.

Definition 4 (Time Slot). *Given a base timestamp t_0 (to keep $t - t_0 \geq 0$, t_0 must be no larger than any timestamp in both training and test data) and a unit time Δt, we can get time intervals $[t_0, t_0 + \Delta t), [t_0 + \Delta t, t_0 + 2\Delta t), \cdots$. These intervals are called time slots and the size of each slot is Δt.*

For simplicity, each slot is denoted as its corresponding serial number. For example, $[t_0, t_0 + \Delta t)$ is denoted as 0. By Definition 4, a timestamp t can be projected into a particular time slot t^p , where $t \geq t_0$ and t^p is calculated as below.

$$t^p = \lfloor \frac{t - t_0}{\Delta t} \rfloor \tag{2}$$

To represent each timestamp in a more fine-grained way, we record the time remainder t^r for the uniqueness of t, where $0 \leq t^r < \Delta t$ and it is calculated as below.

$$t^r = t - t_0 - t^p \Delta t \tag{3}$$

In summary, each timestamp t can be represented as $\langle t^p, t^r \rangle$.

Building the Temporal Graph. Next, we study how to capture temporal features by embedding time slots. However, the number of time slots is unlimited and we cannot embed all time slots. Considering that there are weekly periodicity among traffic conditions (Figure 5(a) shows an example of weekly periodicity), we can only need to focus on all time slots of a week for simplicity. Inspired by [27], we try to build a temporal graph for time slots and then apply graph embedding methods to initialize the time slot embeddings. However, the authors in [27] build an undirected graph for time slots, which cannot capture the sequential relationship between time slots. In addition, they neglect the link between neighboring days, and thereby cannot capture daily periodicity. Therefore, we design a new temporal graph, denoted as $G' = \langle V', E' \rangle$. For the graph, each node $v' \in V'$ represents a time slot and edges in E' can be categorized into two groups: (1) edges for neighboring time slots, indicating that the representations of adjacent time slots should be smooth; (2) edges for neighboring days, meaning that the same time slot at adjacent weekdays should be similar. Take Figure 5(b) as an example, we first set Δt as 5 minutes and thus each day is split into 288 time slots. Later, we consider seven days for a week and build the directed temporal graph (the size is $288 \times 7 = 2016$). At last, t^p can be projected into one node $v' \in V'$, and the serial number of v' is calculated as $t^p \% 2016$, where % is the remainder operator.

Embedding of Time Slots in the Temporal Graph. Similar to road segment embedding in Section 4.1, we first leverage one-hot encoding to represent each time slot $O_i^t \in \mathbb{R}^{|V'|}$, where $|V'|$ is the number of all nodes in the temporal graph G'. Then we design a fully connected neural network (the weights matrix is $\mathbf{W}_t \in \mathbb{R}^{|V'| \times d_t}$) to convert each one-hot code O_i^t into a fixed-length dense vector $D_i^t = \mathbf{W_t}^\mathsf{T} O_i^t$. Later, we use graph embedding methods (e.g., node2vec) to embed all nodes of the temporal graph in Figure 5(b) and use the embedding codes as the initial value of $\mathbf{W_t}$.

4.3 Time Interval Encoder

According to Definition 1, the temporal feature of each road segment is composed of a time interval. Thus, we need to encode the time interval into a temporal representation. As shown in Figure 6, the encoding module can be split into two parts, which are called time slot embedding and merging.

Time Slot Embedding. Given a time interval $[t[1], t[-1]]$, we first normalize the two end timestamps $t[1]$ and $t[-1]$ into $\langle t^p[1], t^r[1]\rangle$ and $\langle t^p[-1], t^r[-1]\rangle$ according to Formula 2 and Formula 3. As a result, the time interval includes Δd time slots, where Δd is calculated as below.

$$\Delta d = t^p[-1] - t^p[1] + 1 \tag{4}$$

The Δd time slots are denoted as $t^p[1], t^p[1]{+}1, \cdots, t^p[-1]-1, t^p[-1]$. After matching them to different nodes in the temporal graph, their corresponding one-hot codes are $O^t_{t^p[1]}$, $O^t_{t^p[1]+1}, \cdots, O^t_{t^p[-1]}$. Then, according to the procedure of time slot embedding (Section 4.2), these one-hot codes are converted into fixed-length dense vectors $D^t_{t^p[1]} = \mathbf{W_t}^\top O^t_{t^p[1]}$, $D^t_{t^p[1]+1} = \mathbf{W_t}^\top O^t_{t^p[1]+1}, \cdots, D^t_{t^p[-1]} = \mathbf{W_t}^\top O^t_{t^p[-1]}$.

Merging. After getting these fixed-length dense vectors $D^t_{t^p[1]}, D^t_{t^p[1]+1}, \cdots, D^t_{t^p[-1]}$, we merge them into one matrix $\mathbf{D^t} = [D^t_{t^p[1]}, D^t_{t^p[1]+1}, \cdots, D^t_{t^p[-1]}]^\top \in \mathbb{R}^{\Delta d \times d_t}$. Then, we regard the matrix as a $1 \times \Delta d \times d_t$ tensor, whose channel has 1-dimension. We apply a deep Residual Network (ResNet [17]) block, which is efficient in many real-world applications [19, 35], to encode the tensor. In this model, the residual part of the ResNet block is implemented with the Convolutional Neural Network (CNN) model. The reason of using CNN is twofold. First, CNN is able to extract different local features. For example, if a time interval includes five time slots $[1, 2, 3, 4, 5]$, we can capture the local features of $[1, 2, 3]$, $[2, 3, 4]$, $[3, 4, 5]$ when we set the kernel size of one convolutional layer as 3. Second, CNN can guarantee translation invariance. For example, if we have two time intervals, where the respective time slots being included are $[1, 2, 3, 4]$ and $[2, 3, 4, 5]$; we can capture the features of $[2, 3, 4]$ for both time intervals when we set the kernel size as 3. In our settings, the CNN model contains three convolutional layers, two BatchNorm layers and two activation layers. In particular, the size of the output channels for these three convolutional layers are 4, 8 and 1 respectively, and the corresponding formulations are listed as below.

$$\mathbf{Z^1}_{i,j,k} = ReLU(BN(sum(K^1_i \otimes \mathbf{D^t}_{1,j-1:j+1,k}))) \tag{5}$$

$$\mathbf{Z^2}_{i,j,k} = ReLU(BN(sum(K^2_i \otimes \mathbf{Z^1}_{1:4,j-1:j+1,k}))) \tag{6}$$

$$\mathbf{Z^3}_{i,j,k} = sum(K^3_i \otimes \mathbf{Z^2}_{1:8,j,k}) \tag{7}$$

$$\mathbf{Z^4} = \mathbf{D^t} \oplus \mathbf{Z^3}_{1,:,:} \tag{8}$$

where $K^1_i \in \mathbb{R}^{1\times3\times1}, K^2_i \in \mathbb{R}^{4\times3\times1}$ and $K^3_i \in \mathbb{R}^{8\times1\times1}$ represent kernel tensors, $\mathbf{Z^1} \in \mathbb{R}^{4\times\Delta d\times d_t}$, $\mathbf{Z^2} \in \mathbb{R}^{8\times\Delta d\times d_t}$ and $\mathbf{Z^3} \in \mathbb{R}^{1\times\Delta d\times d_t}$ respectively denote the output of the three convolutional layers and $\mathbf{Z^4} \in \mathbb{R}^{\Delta d\times d_t}$ is the final output of the ResNet block. The operators $\otimes$ and $\oplus$ denote the element-wise product and addition respectively. The function $sum(\cdot)$ is used to compute the sum of all elements and the function $BN(\cdot)$ represents the BatchNorm layer, which is effective for converging the neural network [21]. In particular, we select ReLU (Rectified Linear Unit) as the activation function, which is calculated as below.

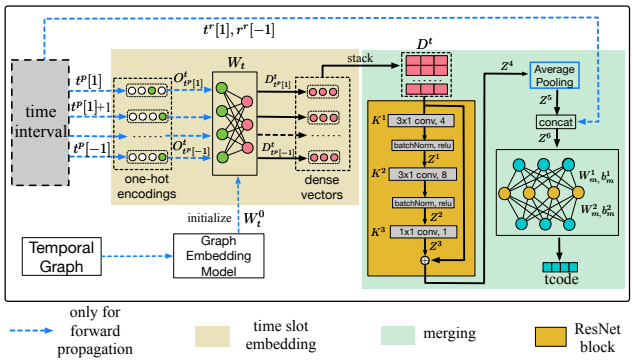


Figure 6: An Overview of Time Interval Encoder

$$ReLU(x) = max(0, x) \tag{9}$$

As reported in [15], ReLU can alleviate the vanishing gradient problem and thus get better gradient propagation. Considering that the size of $\mathbf{Z^4}$ is a variable, we utilize the pooling technique to compress $\mathbf{Z^4}$ into a fixed-length vector. Specifically, we regard $\mathbf{Z^4}$ as a matrix with the size of $\Delta d \times d_t$, and then we compute the average value for each column (Δd elements) of the matrix and thus get a d_t-dimensional vector. The formulation is listed as below.

$$\mathbf{Z^5}_i = avg(\mathbf{Z^4}_{1:\Delta d,i}) \tag{10}$$

where $\mathbf{Z^5} \in \mathbb{R}^{d_t}$ is the pooling vector and $avg(\cdot)$ is used to compute the average value of all elements.

Lastly, the final temporal representation $tcode$ is obtained in two steps. We first concatenate the vector $\mathbf{Z^5}$ and the two time remainders $t^r[1], t^r[-1]$. Then, we use a Multilayer Perceptron (MLP) model to encode the concatenated vector $\mathbf{Z^6} = concat(\mathbf{Z^5}, t^r[1], t^r[-1]) \in \mathbb{R}^{d_t+2}$ into the representation $tcode$. As shown in Figure 6, we implement the MLP model based on the Pytorch tutorial [5], which consists of three layers of nodes and two layers of edges. In particular, edges in each layer correspond to the parameters of a weight matrix and a bias vector. Hence we focus on the edges and denote the model as a two-layer MLP model for simplicity. Specifically, $tcode$ is computed as below.

$$tcode = \mathbf{W^2_m} ReLU(\mathbf{W^1_m}\mathbf{Z^6} + b^1_m) + b^2_m \tag{11}$$

where $\mathbf{W^1_m} \in \mathbb{R}^{d^1_m\times(d_t+2)}$ and $b^1_m \in \mathbb{R}^{d^1_m}$ respectively denote the weight matrix and bias vector parameters of the first layer, and $\mathbf{W^2_m} \in \mathbb{R}^{d^2_m\times d^1_m}$, $b^2_m \in \mathbb{R}^{d^2_m}$ indicate the parameters in the second layer.

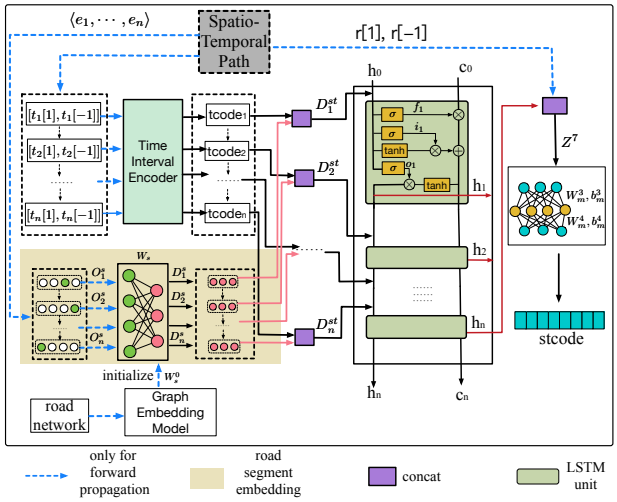


Figure 7: An Overview of Trajectory Encoder

4.4 Trajectory Encoder

In this section, we try to encode each trajectory $T = \langle SP, PR \rangle$ into a fixed-length vector in two steps, as shown in Figure 7. The first step is to encode the *spatio-temporal path* $SP = \langle \langle e_1, [t_1[1], t_1[-1]] \rangle, \cdots, \langle e_n, [t_n[1], t_n[-1]] \rangle \rangle$. Specifically, for each element $\langle e_i, [t_i[1], t_i[-1]] \rangle$, we use the `Time Interval Encoder` to encode the time interval $[t_i[1], t_i[-1]]$ into a fixed-length vector $tcode_i$, and then use `road segment embedding` to embed the road segment e_i. Next, we concatenate $tcode_i$ and the embedding of e_i into a dense vector, and thus get a sequence of concatenated representations. For the sequence, we use an RNN model (e.g., LSTM [18]) to embed it into a fixed-length vector, which is the representation of SP. The second step is to concatenate the representation of SP with the two position ratios $r[1]$ and $r[-1]$, and use a *Multilayer Perceptron* model to further encode the concatenated result into the final representation $stcode$. Details are illustrated as follows.

Time Interval Encoding. We first extract the sequence of time intervals from SP and then use `Time Interval Encoder` to encode each time interval $[t_i[1], t_i[-1]]$ into the temporal representation $tcode_i$. The detail of `Time Interval Encoder` has been described in Section 4.3.

Road Segment Embedding. Recall Section 4.1, we use a fully connected neural network to represent the embedding matrix and use the matrix to embed each road segment e_i in the spatial path into a dense vector D_i^s. In addition, we initialize the embedding matrix with the road network data.

Sequence Encoding. For each road segment e_i, we regard $tcode_i$ as the temporal representation and regard D_i^s as the spatial representation, and then we concatenate $tcode_i$ and D_i^s as its spatio-temporal representation, denoted by $D_i^{st} = concat(tcode_i, D_i^s) \in \mathbb{R}^{d_m^2+d_s}$. After that, we get a sequence of spatio-temporal representations $([D_1^{st}, \cdots, D_n^{st}])$. The next step is to encode this sequence into a fixed-length vector using the sequence model LSTM (Long Short-Term Memory). Given a sequence, LSTM would successively take each element in the sequence as an input vector of the LSTM unit, where different units share common weights. Specifically, the architecture of an LSTM unit is composed of a cell (the memory part of the LSTM unit) and three gates – an input gate, an output gate and a forget gate, as defined below:

$$f_j = \sigma(\mathbf{W}_f[D_j^{st}, h_{j-1}] + b_f) \tag{12}$$

$$i_j = \sigma(\mathbf{W}_i[D_j^{st}, h_{j-1}] + b_i) \tag{13}$$

$$o_j = \sigma(\mathbf{W}_o[D_j^{st}, h_{j-1}] + b_o) \tag{14}$$

$$c_j = f_j \otimes c_{j-1} + i_j \otimes tanh(\mathbf{W}_c[D_j^{st}, h_{j-1}] + b_c) \tag{15}$$

$$h_j = o_j \otimes tanh(c_j) \tag{16}$$

The initial values of c_j and h_j are $c_0 = \mathbf{0}$ and $h_0 = \mathbf{0}$. For each time step j, $D_j^{st} \in \mathbb{R}^{d_m^2+d_s}$ means the input vector, $f_j, i_j, o_j \in \mathbb{R}^{d_h}$ represent activation vectors of forget gate, input gate and output gate, respectively; $h_j \in \mathbb{R}^{d_h}$ is the hidden state vector, also known as the output vector of the LSTM unit; $c_j \in \mathbb{R}^{d_h}$ denotes the cell state vector. In addition, $\sigma(\cdot)$ and $tanh(\cdot)$ represent two kinds of activation functions, i.e., sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$ and hyperbolic tangent function $tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$. Importantly, we need to learn the weight matrix parameters $(\mathbf{W_f}, \mathbf{W_i}, \mathbf{W_o}, \mathbf{W_c} \in \mathbb{R}^{d_h \times (d_m^2+d_s+d_h)})$ and the bias vector parameters $(b_f, b_i, b_o, b_c \in \mathbb{R}^{d_h})$ in the training stage. As a result, we can get the final output vector $h_n \in \mathbb{R}^{d_h}$ for any given road segment sequence.

Final Representation. The final step is to merge the final vector h_n with the remaining information $r[1]$ and $r[-1]$, as shown in the following formula. First, we concatenate h_n with $r[1], r[-1]$ resulting in a $(d_h + 2)$-dimensional merged vector $\mathbf{Z}^7 = concat(h_n[:], r[1], r[-1])$. Then, we use a two-layer Multilayer Perceptron to encode the merged vector into the final representation $stcode$.

$$stcode = \mathbf{W}_\mathbf{m}^4 ReLU(\mathbf{W}_\mathbf{m}^3 \mathbf{Z}^7 + b_m^3) + b_m^4 \tag{17}$$

$\mathbf{W}_\mathbf{m}^3 \in \mathbb{R}^{d_m^3 \times (d_h+2)}$ and $b_m^3 \in \mathbb{R}^{d_m^3}$ respectively denote the weight matrix and bias vector parameters of the first layer network; $\mathbf{W}_\mathbf{m}^4 \in \mathbb{R}^{d_m^4 \times d_m^3}$ and $b_m^4 \in \mathbb{R}^{d_m^4}$ respectively represent the weight matrix and bias vector parameters of the second layer network. ReLU (Rectified Linear Unit) is selected as the activation function of the first neural network.

4.5 External Features Encoder

As shown in Figure 3, there can be external features f (if available) that may also influence the travel time. Hence, we design the `External Features Encoder` model to encode those external features. For this work's experimental study, f is composed of weather and current traffic condition.

The weather belongs to discrete feature, so we use the one-hot code to represent it. For example, we categorize the

weather into N_{wea} types (e.g., sunny, rainy), so each type can be denoted as an N_{wea}-dimensional one-hot code O_{wea}. Inspired by [26, 37], we use the average speed in a local area to represent the traffic condition of the local area. In particular, we split the whole area into different grids with the same size (e.g., $200m \times 200m$), and compute the average speed every Δt minutes for each grid. Thus, we get a speed matrix (average speeds of all grids) every Δt minutes. We select the nearest speed matrix $C \in \mathbb{R}^{\lceil L_{lat}/l \rceil \times \lceil L_{lon}/l \rceil}$ before the departure time as the current traffic condition feature, where L_{lat} and L_{long} denote the latitude length and the longitude length of the whole area, and l denotes the grid length. Similar to [26], we use a CNN model to convert the speed matrix into a fixed-length vector $D_{traf} \in \mathbb{R}^{d_{traf}}$. The CNN model comprises of three connected convolution blocks and an average pooling layer. Each convolution block consists of three layers: $Conv2d \rightarrow BatchNorm2d \rightarrow ReLU$.

To get the final representation of f, we first concatenate O_{wea} and D_{traf} into a fixed-length vector $\mathbf{Z^8} = concat(O_{wea}, D_{traf}) \in \mathbb{R}^{N_{wea}+d_{traf}}$, and then use a two-layer Multilayer Perceptron to encode the vector into the final representation $ocode$ as below.

$$ocode = \mathbf{W_m^6} ReLU(\mathbf{W_m^5 Z^8} + b_m^5) + b_m^6 \tag{18}$$

where $\mathbf{W_m^5} \in \mathbb{R}^{d_m^5 \times (N_{wea}+d_{traf})}$ and $b_m^5 \in \mathbb{R}^{d_m^5}$ are parameters of the first layer while $\mathbf{W_m^6} \in \mathbb{R}^{d_m^6 \times d_m^5}$ and $b_m^6 \in \mathbb{R}^{d_m^6}$ are parameters of the second layer.

4.6 Travel Time Estimation

As described in Section 3, we concatenate the vectors D_1^s, D_n^s, D^t, $ocode$ and the float values $r[1], r[-1], t^r$ into a fixed-length vector $\mathbf{Z^9} = concat(D_1^s, D_n^s, D^t, ocode, r[1], r[-1], t^r) \in \mathbb{R}^{(d_s \times 2 + d_t + d_m^6 + 3)}$, and then use the model $\texttt{MLP}_1$, which is a two-layer Multilayer Perceptron, to convert the vector into $code$. The whole process is formalized as below.

$$code = \mathbf{W_m^8} ReLU(\mathbf{W_m^7 Z^9} + b_m^7) + b_m^8 \tag{19}$$

where $\mathbf{W_m^7} \in \mathbb{R}^{d_m^7 \times (d_s \times 2 + d_t + d_m^6 + 3)}$ and $b_m^7 \in \mathbb{R}^{d_m^7}$ correspond to the first layer parameters of $\texttt{MLP}_1$ while $\mathbf{W_m^8} \in \mathbb{R}^{d_m^8 \times d_m^7}$ and $b_m^8 \in \mathbb{R}^{d_m^8}$ correspond to the second layer parameters of $\texttt{MLP}_1$. Considering that the dimensions of $code$ and $stcode$ should be equal, we set d_m^8 equal to d_m^4.

To generate the travel time, we use the model $\texttt{MLP}_2$, which is also a two-layer Multilayer Perceptron, to convert $code$ into a float value, which is represented as followings:

$$\hat{y} = \mathbf{W_m^{10}} ReLU(\mathbf{W_m^9} code + b_m^9) + b_m^{10} \tag{20}$$

where $\mathbf{W_m^9} \in \mathbb{R}^{d_m^9 \times d_m^8}$ and $b_m^9 \in \mathbb{R}^{d_m^9}$ denote the first layer parameters of $\texttt{MLP}_2$ while $\mathbf{W_m^{10}} \in \mathbb{R}^{1 \times d_m^9}$ and $b_m^{10} \in \mathbb{R}^1$ denote the second layer parameters of $\texttt{MLP}_2$.

Algorithm 1: Model Learning for DeepOD

Input: Road network graph $G = \langle V, E \rangle$, the base timestamp t_0, the time slot size Δt, training inputs X, training labels Y, test inputs X', trajectory encoding model $\mathcal{M}_T$, OD input encoding model $\mathcal{M}_O$, travel time estimation model $\mathcal{M}_E$, learning rate lr, training epochs I, batch size bs, $auxiliaryloss$ weight w.

Output: embedding matrices $\mathbf{W_s}, \mathbf{W_t}$, other parameters $\theta_T, \theta_O, \theta_E$ for the three models $\mathcal{M}_T$, $\mathcal{M}_O$ and $\mathcal{M}_E$, prediction results Y'

initialize road segment embeddings $\mathbf{W_s}^0 \leftarrow node2vec(G)$;
build temporal graph $G' = \langle V', E' \rangle$ with $t_0, \Delta t$;
initialize time slot embeddings $\mathbf{W_t}^0 \leftarrow node2vec(G')$;
initialize embedding matrices in $\mathbf{W_s}, \mathbf{W_t}$ with $\mathbf{W_s}^0, \mathbf{W_t}^0$;
initialize other parameters $\theta_T, \theta_O, \theta_E$ in $\mathcal{M}_T, \mathcal{M}_O, \mathcal{M}_E$ with normal distribution;
for $i \leftarrow 1 \cdots I$ **do**
  $ModelTrain(X, Y, \mathcal{M}_T, \mathcal{M}_O, \mathcal{M}_E, lr, bs, w)$;
$Y' \leftarrow Estimation(X', \mathcal{M}_O, \mathcal{M}_E)$;
return $\mathbf{W_s}, \mathbf{W_t}, \theta_T, \theta_O, \theta_E, Y'$;

Function ModelTrain-offline

Input: $X, Y, \mathcal{M}_T, \mathcal{M}_O, \mathcal{M}_E, lr, bs, w$

training iterations $I' = \lfloor \frac{|X|}{b_s} \rfloor$;
$shuffle(X, Y)$;
for $i \leftarrow 1 \cdots I'$ **do**
  collect $X_{(i-1)b_s+1:i \times b_s}, Y_{(i-1)b_s+1:i \times b_s}$;
  $[(g[1], g[-1], t, f, T)] \leftarrow X_{(i-1)b_s+1:i \times b_s}$;
  $[y] \leftarrow Y_{(i-1)b_s+1:i \times b_s}$;
  $[code] \leftarrow \mathcal{M}_O([(g[1], g[-1], t, f)])$;
  $[stcode] \leftarrow \mathcal{M}_T([T])$;
  $[\hat{y}] \leftarrow \mathcal{M}_E([code])$;
  $auxiliaryloss \leftarrow \sqrt{\sum_j (code[j] - stcode[j])^2}$;
  $mainloss \leftarrow MAE([y], [\hat{y}])$;
  $loss \leftarrow w \times auxiliaryloss + (1 - w) \times mainloss$;
  $\Delta\theta \leftarrow AdamOpt([\mathbf{W_s}, \mathbf{W_t}, \theta_T, \theta_O, \theta_E], loss, lr)$;
  update $\mathcal{M}_E, \mathcal{M}_W, \mathcal{M}_D$ with $\Delta\theta$;

Function Estimation-online

Input: $X', \mathcal{M}_O, \mathcal{M}_E$
Output: Y'

$[(g[1], g[-1], t, f)] \leftarrow X'$;
$[code'] \leftarrow \mathcal{M}_O([(g[1], g[-1], t, f)])$;
$Y' \leftarrow \mathcal{M}_E([code'])$;
return Y';

5 MODEL LEARNING

Algorithm 1 presents the whole learning process which consists of two steps. The first step is to offline train the whole model using the training data and the second step is to online estimate the travel time for test data using the trained model.

Offline Training. First of all, we initialize the road segment embedding matrix $\mathbf{W_s}$ based on the graph embedding method $node2vec$ [16]. Meanwhile, given the size of time slot Δt, we can build a temporal graph G' and initialize the

Table 2: Taxi Order Datasets

	Chengdu	Xi'an	Beijing
# of orders	5.8M	3.4M	56.7M
Avg # of points	180	205	23
Avg travel time(s)	500.65	757.07	1,180.87
Time interval	10/1-11/30,2016		3/1-3/31,2009
Avg # of road segments	17	25	48
Avg length(meter)	3,477.85	4,143.17	5,580.32

time slot embedding matrix $\mathbf{W_t}$ based on *node2vec*. Note that we tried three graph embedding methods (i.e., DeepWalk [30], Line [36], node2vec [16]) to get initial embeddings, and node2vec achieves the best result. As for other parameters of the whole model, we use normal distribution to initialize them (lines 1-5). Afterwards, we iteratively train the whole model with the given epochs I (lines 6-7). Specifically, the function `ModelTrain` explains the training process for each epoch. We first compute the training iterations I' based on given batch size b_s, then shuffle all training data X, Y (lines 1-2). In each iteration, we extract b_s training data from X, Y. Each element of training input X is composed of $g[1], g[-1], t, f$ and T. In particular, we use the input $g[1], g[-1], t, f$ and the model $\mathcal{M}_O$ to generate *code*. We use the input T and the model $\mathcal{M}_T$ to generate *stcode*. Afterwards, we use the hidden representation *code* and the model $\mathcal{M}_E$ to predict the travel time. Notably, we use the Euclidean metric $auxiliaryloss = \sqrt{\sum_j (code[j] - stcode[j])^2}$ to evaluate the distance between *code* and *stcode*. Meanwhile, we use MAE (Mean Absolute Error) to compute the loss of the estimated travel time, which is denoted as *mainloss*. Finally, we utilize Adam Optimizer [24] to optimize all parameters by minimizing the weighted sum of *auxiliaryloss* and *mainloss*. Formally, the loss is computed as $loss = w \times auxiliaryloss + (1 - w) \times mainloss$, where w is a tuning parameter that is fine-tuned by validation data as elaborated in Section 6.3.

Online Estimation. As mentioned before, each element of test inputs X' is composed of $g[1], g[-1], t$ and f, so we only need to consider the two parts $\mathcal{M}_O$ and $\mathcal{M}_E$. As listed in the function `Estimation`, we use $\mathcal{M}_O$ to generate the hidden representation *code'* (line 2). In the end, we generate the estimated travel time Y' with the model $\mathcal{M}_E$ and the representation *code'* (line 3).

6 EXPERIMENTS

6.1 Experimental Setup

Datasets. We used three real datasets.

(1) **Road Networks.** We used three road networks: *Chengdu Road Network (CRN)*, *Xi'an Road Network (XRN)*, and *Beijing Road Network (BRN)*. All of them were extracted from OpenStreetMap [4]. *CRN* includes 3, 191 vertices and 9, 468 edges, *XRN* contains 4, 576 vertices and 12, 668 edges, and *BRN* contains 82, 576 vertices and 241, 105 edges.

(2) **Taxi Orders.** We used taxi orders in *Chengdu*, *Xi'an* [3], and *Beijing* [45], and each order corresponds to a trip record, which consists of an OD input and a trajectory. Specifically, we aligned the GPS points in OD inputs and trajectories with road networks via a map-matching tool Valhalla [7]. Table 2 shows the statistics of these two datasets, where *Avg # of points* is the average number of GPS points per trajectory (the average time gap between two consecutive GPS points is 3 seconds for *Chengdu* and *Xi'an*, and 1 minute for *Beijing*), *Avg # of road segments* is the average number of road segments per trajectory after map-matching, and *Avg length* represents the average length of trajectories. Note that *Beijing* has similar size with the *BJS-Pickup* dataset used in [27], but *Beijing* has longer travel time than *BJS-Pickup*. In particular, the average travel time for *Beijing* is around 20 minutes while it's only 3 minutes for *BJS-Pickup*.

(3) **Data For External Features.** Two types of external data were included, the weather and the current traffic condition. The weather records were collected from the website [1]. Similar to the literature [47], the number of the weather type is $N_{wea} = 16$. As for the current traffic condition, we first split the area of each road network into disjoint grids with the same size $200m \times 200m$. The size of *CRN* is 8,166m× 8,330m, so *CRN* is split into 41×42 grids. Similarly, *XRN* is split into 41×40 grids as its size is 8,056m× 7,942m; while *BRN* is split into 354×311 grids as its size is 70,737m×62,180m. Afterwards, we calculated different speed matrices every 5 minutes for *CRN*, *XRN* and *BRN* respectively. Therefore, we denoted the speed matrix closest to the departure time as the traffic condition for each taxi order.

(4) **Training & Validation & Test Data.** The date of taxi orders for *Chengdu* and *Xi'an* are both from 10/01/2016 to 11/30/2016. We split each dataset into training data, validation data and test data with the ratio 42:7:12. That is, the taxi orders (with historical trajectories) during the time interval [10/01/2016, 11/11/2016] were used to train our proposed models, the taxi orders during [11/12/2016, 11/18/2016] were used to fine-tune parameters to get the best performance, and the taxi orders (without historical trajectories) during [11/19/2016, 11/30/2016] were used to test the models.

Baseline methods. We compared our models with five baseline methods for OD travel time estimation:

- Temporally weighted neighbors (`TEMP`) [39]: a nearest neighbor based approach which estimates the OD travel time by averaging the travel time of all historical trajectories falling in the same time slot with a similar origin and destination.
- Linear Regression (`LR`): a machine learning approach using a linear function to model travel time and computing the loss between the estimated travel time and the actual travel time with Euclidean distance.

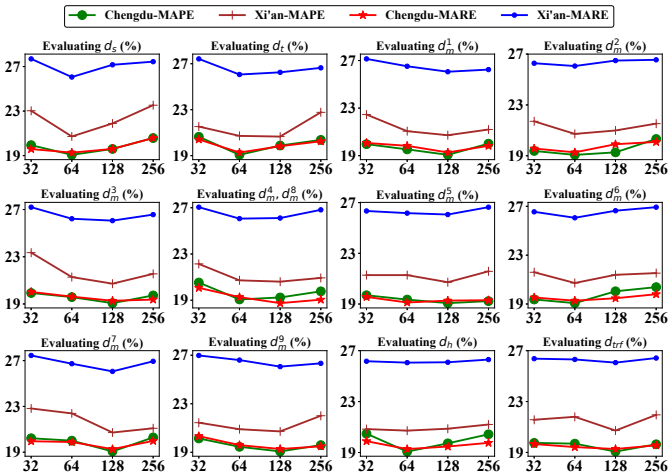
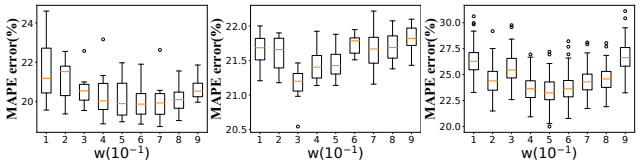


Figure 8: MAPE & MARE Loss vs. Hyper-parameters



(a) Chengdu (b) Xi'an (c) Beijing

Figure 9: MAPE Loss vs. Loss Weight w

- Gradient Boosted Machine (GBM): a gradient boosting decision tree based regression method which is implemented using XGBoost [10].
- Spatial Temporal deep Neural Network (STNN) [23]: a deep neural network based approach which first predicts the travel distance given an OD pair, and then combines this prediction with the departure time to estimate the travel time.
- Multi-task Representation Learning (MURAT) [27]: a deep neural network based approach which jointly predicts the travel distance and the travel time for taxi orders by learning representations of road segments and the origin-destination information. It is proposed from the largest ride-hailing company (Didi) in China. We implemented this model following the parameters suggested in [27].

Environment settings. All methods were implemented with PyTorch 1.0 and Python 3.6, and trained with a Tesla K40 GPU. The platform ran on Ubuntu 16.04 OS.

Evaluation metrics. We evaluated our proposed methods and baseline methods based on three popular metrics: MAE (Mean Absolute Error), MAPE (Mean Absolute Percent Error) and MARE (Mean Absolute Relative Error). Specifically, suppose that the ground truth is represented as $\mathbf{y} = \{y^i\}$ and the predicted result is denoted as $\hat{\mathbf{y}} = \{\hat{y}^i\}$, where $1 \leq i \leq N$, these metrics are computed as follows: $MAE(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{N}\sum_{i=1}^{N}|y^i - \hat{y}^i|$, $MAPE(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{N}\sum_{i=1}^{N}|\frac{y^i - \hat{y}^i}{y^i}|$, $MARE(\mathbf{y}, \hat{\mathbf{y}}) = \frac{\sum_{i=1}^{N}|y^i - \hat{y}^i|}{\sum_{i=1}^{N}|y^i|}$. According to Algorithm 1, we utilized the metric MAE as the loss function when training our DeepOD model. In particular, we used Adam [24] as the optimization method with the mini-batch size of 1,024. The initial learning rate was 0.01, and reduced by $\frac{1}{5}$ every 2 epochs.

6.2 Setting of DeepOD's Hyper-parameters

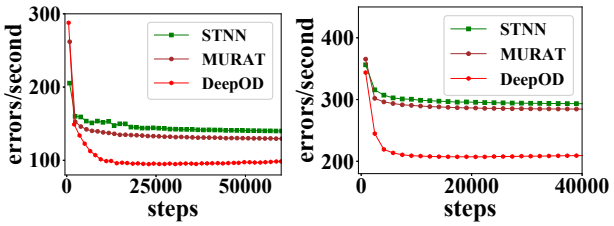
We mainly considered three hyper-parameters: (1) the embedding size (d_s) of road segments; (2) the embedding size (d_t) of time slots; (3) the sizes ($d_m^1, d_m^2, d_m^3, d_m^4, d_m^5, d_m^6, d_m^7, d_m^8, d_m^9, d_{traf}$) of different layer's neural networks. In particular, we respectively set each parameter's value as 32, 64, 128 and 256, and then evaluated DeepOD's performance on the validation data of *Chengdu* and *Xi'an*. In addition, as stated in Section 4.6, d_m^4 and d_m^8 should be set as the same value. As shown in Figure 8, we plotted the MAPE and MARE loss for different hyper-parameters. In summary, we set each hyper-parameter with the value corresponding to the optimal performance as follows: $d_s = 64$, $d_t = 64$, $d_m^1 = 128$, $d_m^2 = 64$, $d_h = 128$, $d_m^3 = 128$, $d_m^8 = d_m^4 = 64$, $d_m^5 = 128$, $d_m^6 = 64$, $d_m^7 = 128$, $d_m^9 = 128$, $d_{traf} = 128$.

6.3 Effectiveness of Loss Weight

To fine-tune the *auxiliaryloss* weight w, we varied the value of w to train the model DeepOD. Specifically, the value range was changed from 0.1 to 0.9 with a step of 0.1. Once the DeepOD model was trained, we computed the MAPE loss for each mini-batch of validation data, where the mini-batch size was 1,024. Then, we collected all mini-batches and drew the corresponding Box-plots for both datasets in Figure 9. From this figure, we can deduce two major findings: (1) In the beginning, the performance of DeepOD improves with the increasing of w. However, the performance is worsened when the value of w exceeded a certain threshold. For instance, the best value is 0.7, 0.3 and 0.5 for *Chengdu*, *Xi'an* and *Beijing* respectively. (2) The effectiveness of the weight w is different for different datasets. For example, the performance on *Chengdu* is more stable than *Xi'an* and *Beijing* when w is close to its threshold. In summary, we set the default value of w as 0.7, 0.3, 0.5 for *Chengdu*, *Xi'an* and *Beijing* respectively.

6.4 Comparison with Baselines

6.4.1 Training Comparison with Deep Learning Models. As aforementioned, STNN, MURAT and DeepOD are implemented based on deep learning technology. To visualize the process of training neural networks, we computed the MAE loss for validation data once a step of training was finished. Each step of training consists of one forward propagation and one backward propagation for a mini-batch of training data. Figure 10 shows the MAE loss for *Chengdu* and *Xi'an*. In particular, we considered 60,000 steps for *Chengdu* and 40,000 steps for *Xi'an*. From Figure 10, we can observe that:



(a) MAE Loss on Chengdu (b) MAE Loss on Xi'an

Figure 10: Validation Errors vs. Training Steps

Table 3: Convergence Steps and Convergence Time

Chengdu/Xi'an	STNN	MURAT	DeepOD
steps	32K/14.1K	24.2K/12.4K	25.7K/9.1K
time (hours)	1.01/0.67	3.17/2.17	3.01/1.58

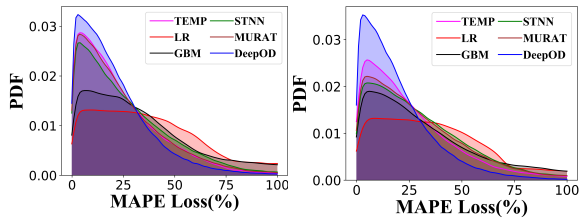
Table 4: Experimental Results on Test Errors

Datasets	Chengdu/Xi'an/Beijing		
Metric	MAE(second)	MAPE(%)	MARE(%)
TEMP	179.98/339.28/674.65	34.07/41.67/57.38	36.07/44.8/57.08
LR	210.60/369.22/531.86	50.77/55.63/47.00	42.20/48.76/46.58
GBM	168.25/287.51/387.37	44.14/43.61/33.90	33.72/37.97/32.81
STNN	136.84/289.52/395.99	29.03/35.13/29.94	27.41/38.00/33.41
MURAT	127.63/281.65/384.11	27.31/33.25/27.05	25.57/36.95/32.41
N-st	118.36/230.47/375.44	24.69/25.58/27.24	23.61/30.73/31.68
N-sp	110.31/217.02/368.21	23.61/23.44/25.78	22.01/28.30/31.06
N-tp	109.38/216.43/366.98	22.28/23.41/25.91	21.22/28.24/30.96
N-other	99.01/209.11/345.02	20.22/21.05/24.02	19.74/27.28/29.10
DeepOD	**94.67/205.37/335.78**	**19.07/20.72/23.48**	**19.27/26.06/28.32**

(1) Our DeepOD outperforms other methods. This is because: we considered actual trajectories when training our model, but existing methods cannot make full use of historical info; DeepOD contains the Trajectory Encoder module, by which we can more effectively represent the spatio-temporal features of trajectories.
(2) The performance of STNN is the worst. The reason is that STNN neglects the information of road networks, whose structure is important for travel time estimation.

In addition, we collected the number of steps and the time required to converge the models. As shown in Table 3, we can observe that:
(1) The convergence speed of all models on *Xi'an* is faster than *Chengdu*. Considering that the data size of *Chengdu* is greater than that of *Xi'an*, we needed more steps to traverse training data of *Chengdu*, and thus we needed more steps to get convergence values for *Chengdu*.
(2) STNN needs more convergence steps than other methods, but requires less convergence time than others. This is because the STNN model's simplicity causes it to require more steps to stabilize, but each step is quicker than other models.
(3) The convergence time of DeepOD is less than MURAT. DeepOD and MURAT spend a similar amount of time to converge for each step, but MURAT needs more steps to stabilize.

6.4.2 Effectiveness Comparison with All Baselines. Except for the comparison experiment of DeepOD with the five baseline methods, we replaced our model DeepOD by four variations, namely N-tp, N-sp, N-st and N-other, to evaluate the effectiveness of different parts of encodings in DeepOD (see Figure 3). In N-tp, we removed the temporal encoding of time intervals. In N-sp, we removed the spatial encoding of road segments. In N-st, we removed the trajectory encoding. In N-other, we removed the external feature encoding. Table 4 reports three metrics of all methods, from which we have the following observations:
(1) Compared with non-linear methods, the linear method LR is not a suitable solution. The reason is that the OD travel time and spatio-temporal features are not linearly related.
(2) The performance of all methods on *Chengdu* is better than that on *Xi'an* and *Beijing*. The causes are twofold. First, the size of *Chengdu* is larger than that of *Xi'an*, and more data often means better training for neural networks. Second, the average travel time of *Chengdu* is shorter than that of *Xi'an* and *Beijing*, and intuitively it is more difficult to accurately estimate longer travel time.
(3) Neural network based methods outperform other methods. It is well known that deep neural networks can approximately fit any function, so it is reasonable to get better performance using deep learning technology.
(4) TEMP is the second worst (only outperforms LR) in *Chengdu* and *Xi'an*, and is the worst in *Beijing*, probably because of the sparsity of trip records. First, there are not enough historical trips at some time slots due to the small size of time slots. Second, for some OD pairs, there may not be enough historical trip records whose origin and destination are similar to these pairs.
(5) According to results of N-st, N-sp, N-st, N-other and DeepOD, we find that the trajectory encoding is the most critical part of DeepOD, followed by the spatial encoding, the temporal encoding and the external features encoding. That is, the trajectory encoding is the main reason that DeepOD outperforms state-of-the-art methods.
(6) When comparing the MAPE loss with the MARE loss for each method, we can find that some methods have better performance on MAPE while others are better on MARE. According to the definitions of MAPE and MARE, the following inequality would be true if MAPE is greater than MARE: $\sum_{i=1}^{N} |y^i - \hat{y}^i|(\frac{1}{y^i} - \frac{N}{\sum_{j=1}^{N} y^j}) > 0$, which means that $|y^i - \hat{y}^i|$ is larger when $y^i < \sum_{j=1}^{N} y^j/N$. That is, the difference between the ground truth time and the prediction time is larger when the ground truth time is shorter.
(7) DeepOD performs the best on all metrics. For example, DeepOD outperforms existing best method MURAT by more than 12% on MAPE loss for the test data of *Xi'an*.



(a) Chengdu (b) Xi'an

Figure 11: MAPE Loss Distribution on Test Data

Table 5: Efficiency of Test Result

Datasets	Chengdu/Xi'an/Beijing		
	model size(Byte)	training time(hours)	estimation time(seconds/K)
TEMP	94.78M/31.17M/805.68M	-/-/-	13.17/6.97/50.08
LR	28K/28K/28K	0.20/0.12/0.53	0.20/0.25/0.28
GBM	0.42K/0.47K/0.59K	0.84/0.61/9.42	0.19/0.41/1.49
STNN	0.30M/0.30M/0.30M	1.01/0.67/2.72	0.31/0.33/0.41
MURAT	7.85M/8.62M/65M	3.17/2.17/10.32	1.56/2.46/4.93
DeepOD	6.24M/7.06M/63M	3.01/1.58/8.36	1.51/1.86/3.28

(8) The performance gap between DeepOD and others is smaller on *Beijing* than that on other datasets. The reason is that *Beijing* provides more training data and then other methods can also capture better representations. That also indicates that DeepOD would be more stable than other methods when training data is insufficient.

In addition, we collected the MAPE loss of test data for all methods and demonstrated their probability distribution curves, as shown in Figure 11. The horizontal axis represents the MAPE loss and the vertical axis represents the probability density function (PDF). We can find that the distribution of our method DeepOD has smaller mean value and smaller variance value than other methods.

6.4.3 Efficiency Comparison. To compare the efficiency of different methods, we respectively collected the model size, training time and estimation time for each method. First, the model size represents the size of required memory for applying the corresponding model, so we use it to evaluate the efficiency of memory usage, with bytes as the unit. Second, the training time can evaluate the offline learning efficiency of different methods. Third, the estimation time can evaluate the online execution efficiency of different methods. Specifically, we use different trained models to estimate the results for 1,000 OD pairs and recorded the latency respectively. As reported in Table 5, we find that:

(1) TEMP requires more memory than others. The reason is two-fold. On the one hand, TEMP needs to load historical trip info, whose size is proportional to the size of historical trajectories. On the other hand, other methods belong to machine learning methods and only need to load model parameters, whose size is constant.

(2) LR, as well as STNN, has the same model size for all datasets, but the remaining methods are different. The number of model parameters for LR and STNN is constant on different datasets. In contrast, GBM needs to fine-tune some hyper-parameters, such as the number and depth of decision trees, whose values vary from one dataset to another. Since the model size of TEMP is proportional to the size of historical trajectories, it also varies with the dataset. As for MURAT and DeepOD, their parameters include road segment embeddings whose size is different for different cities. In particular, *BRN* includes more road segments than *CRN* and *XRN*, so its model size on *Beijing* is bigger than that on *Chengdu* and *Xi'an* for MURAT and DeepOD.

(3) The training time of deep learning models (STNN, MURAT and DeepOD) is greater than other models (LR and GBM) since deep learning models contain more parameters than other models. In addition, TEMP is not a learning method, so we ignore its training time.

(4) All models incur more training time on *Beijing* than the other two. *Beijing* is larger than *Chengdu* and *Xi'an* and the associated models contain more parameters, so more steps are needed to traverse the training data of *Beijing* than that of *Chengdu* and *Xi'an*, which takes more time.

(5) The deep learning models incur more estimation time than other models. However, TEMP needs to look for similar trajectories, which increases the online estimation time, so its estimation time is far greater than other methods.

(6) LR and STNN take similar estimation time in both datasets, because they keep the same model size for different datasets. In contrast, the estimation time increases with the increasing of the model size for GBM, MURAT and DeepOD.

(7) Compared with the state-of-the-art method MURAT, DeepOD is more efficient since we need less offline training time, less memory usage and less online estimation time.

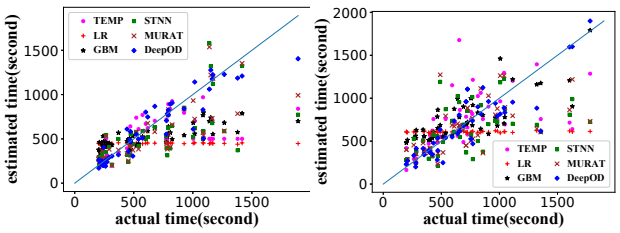
Table 6: Scalability of Test Result (Beijing)

	MAPE loss on Test Data(%)					
scale	TEMP	LR	GBM	STNN	MURAT	DeepOD
20%	88.68	112.92	49.27	39.29	36.56	28.15
40%	78.28	81.37	43.85	35.51	33.37	26.91
60%	72.10	64.45	39.39	32.41	30.27	25.15
80%	64.90	50.81	36.96	31.74	28.73	23.91
100%	57.38	47.00	33.9	29.94	27.05	23.48

6.4.4 Scalability Comparison. To compare the scalability, we trained different models by varying the training data size of *Beijing*. In particular, we sampled 20%, 40%, 60%, 80% and 100% from the training data of *Beijing* to train these models, and then collected the associated MAPE loss of online estimation on the test data of *Beijing*. From Table 6, we have the following observations.

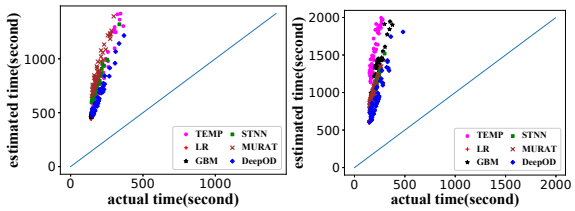
(1) All methods would perform better if we use more training data, because more data covers more situations and thus better models can be learned.

(2) Our method DeepOD is more stable and more effective than other methods. For example, the MAPE loss of DeepOD



(a) Chengdu (b) Xi'an

Figure 12: Estimated time vs. actual time



(a) Chengdu (b) Xi'an

Figure 13: Estimated time vs. actual time on MAPE

is increased by $\frac{28.15-23.48}{23.48} \times 100\% = 19.89\%$ when we only use 20% training data. In contrast, the linear regression model LR increases the loss by $\frac{112.92-47.00}{47.00} \times 100\% = 140.26\%$.

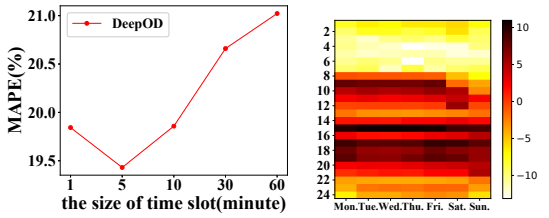
6.4.5 Case Study. We first randomly sampled 50 test data (travel time is less than 1 hour) from *Chengdu* and *Xi'an* respectively, and then used different methods to generate the estimated travel time. After that, we got 50 pairs of the actual time and the estimated time for each method. We illustrated all pairs with scatter points, as plotted in Figure 12. In each figure, we drew an auxiliary line $y = x$ as reference. We can find: (1) most of our DeepOD model's points are closer to the reference line than other methods; (2) the estimated time by LR almost forms a line, mainly because LR is a linear method; (3) with the increase of actual time duration, the errors of estimated time also increase for all methods, but DeepOD has the smallest degree of increase.

To study the performance of each method in the worst case, we selected 50 worst-performing cases for each method. Similarly, we drew them in Figure 13. Specifically, we compared them based on the MAPE loss. According to the definition of MAPE, shorter actual travel time and longer estimated travel time would cause a bigger MAPE loss. Therefore, almost all selected cases were located in the up-left corner of the corresponding figures. We can find that: (1) In most cases, our method DeepOD is closer to the reference line than other methods. (2) There are many extreme worst cases in TEMP, where the MAPE loss could be 200% – 300%. TEMP utilizes the average travel time of similar historical taxi orders to estimate the travel time, and it is difficult to define the similarity between different taxi orders. (3) The worst cases in *Chengdu* are better than those in *Xi'an* for all methods.

In summary, we can conclude that our method is more effective than others in most cases.

Table 7: MAPE Errors(%) of Embeddings in DeepOD

City	T-one	T-day	T-stamp	R-one
Chengdu	20.58(+7.9%)	20.33(+6.6%)	41.89(+119.7%)	21.16(+11.0%)
Xi'an	21.49(+3.7%)	21.59(+4.2%)	50.09(+141.7%)	21.92(+5.8%)
Beijing	**24.27**(+3.4%)	24.62(+4.9%)	34.19(+45.6%)	23.93(+1.9%)



(a) MAPE vs. time slot size (b) Heatmap of time slots

Figure 14: The Effectiveness of Time Slot on Chengdu

6.5 Effectiveness of Embeddings in DeepOD

We evaluated the respective effectiveness of road segment embedding and time slot embedding in DeepOD. As for the time slot embedding, we tried three variations, which were called as T-one, T-day and T-stamp, respectively. T-one used one-hot vectors to initialize the time slot embedding rather than the graph embedding, T-day considered the daily periodicity and built temporal graph using time slots in one day, and T-stamp directly used timestamps. As for the road segment embedding, we used one-hot vectors to initialize it and denoted the method as R-one. We used the MAPE loss to compare their performance. The result is shown in Table 7 (the percentage below the MAPE value shows the MAPE loss's increase percentage w.r.t. DeepOD.), from which the following observations are made.

(1) The effectiveness of DeepOD would deteriorate if we initialized the two embeddings with one-hot vectors, but such deterioration is notably small. This is because we only replaced the initial embeddings and would learn the final embeddings by the same training data anyway.

(2) For *Chengdu* and *Xi'an*, the road segment embedding plays a more important role than the time slot embedding in influencing the effectiveness of DeepOD, which can be deduced from the fact that T-one outperforms R-one. However, the time slot embedding is more important for *Beijing*.

(3) The decreasing degree of *Beijing* is greater than that of *Chengdu* and *Xi'an*. One possible reason is that the training data of *Beijing* is much larger, and the better representations can be generated over *Beijing* even without better initial embeddings.

(4) T-stamp has the worst performance. The main reason is that T-stamp ignores the weekly or daily periodicity of traveling conditions. In addition, the temporal feature of T-stamp has large values and dominates other features.

Furthermore, we utilized *Chengdu* to evaluate the effectiveness of the time slot size. First, we varied the size of time slot from 1 minute to 60 minutes, and then computed the MAPE loss for DeepOD. As shown in Figure 14(a), we got the

best performance when we set the size as 5 minutes. The reasons are twofold. On the one hand, the smaller the time slot size, the finer the embedding, which would improve the performance. One the other hand, the smaller the time slot size, the sparser the timestamps, which would deteriorate the performance. Second, we collected all time slot embeddings (the time slot size was 5 minutes) and converted each embedding into a 1-dimensional value using t-SNE [29]. After that, we computed the average value of every 12 neighboring time slots and drew the corresponding heat map in Figure 14(b). The heat map demonstrates the smoothness of neighboring time slots and the daily and weekly periodicity.

7 RELATED WORK

7.1 Travel Time Estimation

There are two broad categories of work – travel time estimation for paths and travel time estimation for OD inputs.

Travel time estimation for paths. The method of estimating travel time for paths can be divided into two groups depending on the availability of the data source: one is called loop-detector-data approach [8, 14, 22, 31], and the other is called floating-car-data approach [11, 20, 41, 49]. The first group infers the travel time for a road segment through collecting vehicles data from loop detector sensors, which are installed on both endpoints of the road segment. In contrast, the second group directly uses GPS trajectories collected from floating cars. However, neither of them considers the interaction between road segments and thus, accuracy is limited. To address this issue, some methods [28, 33, 42] use sub-paths instead of single segments when computing the travel time for the whole path. For example, Wang et al. [42] propose a dynamic programming method to find the optimal concatenation of sub-paths for estimating travel time.

However, the performance of these methods depend heavily on the historical data on road networks, which may not always be available in each road segment. Moreover, they may lead to inaccurate estimation because such approaches cannot accurately consider road intersections. To address these problems, Wang et al. [38] propose an end-to-end framework to predict the travel time with intermediate GPS points based on the deep learning technique. Zhang et al. [46] also use deep learning method to solve the problem, but it can make full use of temporal information of trajectory data.

Travel time estimation for OD inputs. The problem of travel time estimation for OD inputs [23, 27, 39] is motivated by the fact that many real-world applications cannot get the actual routes when estimating the arrival time for a given origin and destination points, and the only available data are the two points and the departure time. In particular, in [39], for a query origin and destination, they first utilize the nearest neighbor method to select trajectories with a relatively similar origin and destination, and then estimate travel time by averaging the travel time of the selected trajectories. The authors in [23] propose a multi-layer neural network called STNN. They first predict the travel distance based on a given origin and a given destination, and then they combine the predicted distance with the given temporal information to predict the travel time. However, they neglect the information about road network. To make full use of the road network, Li et al. [27] leverage road topological structure and spatio-temporal information of road network to predict travel time. However, they directly embed the longitude and latitude of the origin and the destination, which cannot accurately capture the spatial features on the road network. In addition, they ignore historical trajectories, which are useful for travel time estimation.

7.2 Deep Learning in Spatio-Temporal Data

With the development of AI, there is an increasing growth of deep learning applications in spatio-temporal data. First, Recurrent Neural Network (RNN) is recently applied to trajectory modeling. For example, Wu et al. [44] predicts next movement through modeling trajectory with RNN and outperforms existing shallow models. The authors in [13] represent and identify the semantics of user mobility patterns by embedding trajectories with RNN model. In addition, Dong et al. [12] design a stacked RNN model to characterize the driving style of different drivers. Second, many studies focus on other deep learning models (e.g. Convolutional Neural Network). Song et al. [34] propose an intelligent transportation system to simulate the human mobility and transportation mode. The authors in [25, 32, 48] regard the crowd density on road network as pictures and then propose a deep spatio-temporal residual network to predict the crowd flows.

8 CONCLUSIONS

In this paper, we studied the problem of OD travel time estimation, where the input consists of an origin-destination pair and a departure time. We proposed a comprehensive and novel neural network based approach that is able to fully exploit historical trajectories associated with the OD input. We mapped points into road segments to represent their spatial features. We built the temporal graph to initialize time slot embeddings to capture the weekly and daily periodicity. We design novel representations for the OD pair and its corresponding trajectory. We proposed an effective encoding model to encode the spatio-temporal properties of trajectories. Extensive experiments on real datasets verified the effectiveness of our proposed model.

Acknowledgement. This paper is supported by NSF of China (61925205, 61632016, 91646204), ARC DP200102611, DP180102050, Huawei, TAL, and a Google Faculty Award.

REFERENCES

[1] https://www.tianqi5.cn/lishitianqi/.
[2] Didi chuxing. https://www.didiglobal.com/.
[3] Gaya. https://outreach.didichuxing.com/research/opendata/.
[4] Openstreetmap. https://www.openstreetmap.org.
[5] Pytorch tutorial. https://pytorch.org/tutorials/beginner/examples_nn/two_layer_net_nn.html.
[6] Uber. https://www.uber.com/.
[7] Valhalla. https://github.com/valhalla/valhalla.
[8] M. Asghari, T. Emrich, U. Demiryurek, and C. Shahabi. Probabilistic estimation of link travel times in dynamic road networks. In *SIGSPATIAL*, pages 47:1–47:10, 2015.
[9] S. Brakatsoulas, D. Pfoser, R. Salas, and C. Wenk. On map-matching vehicle tracking data. In *VLDB*, pages 853–864, 2005.
[10] T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In *SIGKDD*, pages 785–794, 2016.
[11] Z. Ding, B. Yang, R. H. Güting, and Y. Li. Network-matched trajectory-based moving-object database: Models and applications. *TITS*, 16(4):1918–1928, 2015.
[12] W. Dong, J. Li, R. Yao, C. Li, T. Yuan, and L. Wang. Characterizing driving styles with deep learning. *CoRR*, 2016.
[13] Q. Gao, F. Zhou, K. Zhang, G. Trajcevski, X. Luo, and F. Zhang. Identifying human mobility via trajectory embeddings. In *IJCAI*, pages 1689–1695, 2017.
[14] Z. Gharibshah, X. Zhu, A. Hainline, and M. Conway. Deep learning for user interest and response prediction in online display advertising. *Data Science and Engineering*, 5(1):12–26, 2020.
[15] X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In *AISTATS*, pages 315–323, 2011.
[16] A. Grover and J. Leskovec. node2vec: Scalable feature learning for networks. In *SIGKDD*, pages 855–864, 2016.
[17] K. He, X. Zhang, S. Ren, and J. Sun. Identity mappings in deep residual networks. In *ECCV*, pages 630–645, 2016.
[18] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.
[19] F. Huang, J. T. Ash, J. Langford, and R. E. Schapire. Learning deep resnet blocks sequentially using boosting theory. In *ICML*, pages 2063–2072, 2018.
[20] T. Hunter, R. Herring, P. Abbeel, and A. Bayen. Path and travel time inference from gps probe vehicle data. *NIPS*, 12(1):2, 2009.
[21] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, pages 448–456, 2015.
[22] Z. Jia, C. Chen, B. Coifman, and P. Varaiya. The pems algorithms for accurate, real-time estimates of g-factors and speeds from single-loop detectors. In *ITSC*, pages 536–541, 2001.
[23] I. Jindal, X. Chen, M. Nokleby, J. Ye, et al. A unified neural network approach for estimating travel time and distance for a taxi trip. *CoRR*, 2017.
[24] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, 2014.
[25] M. Li, H. Wang, and J. Li. Mining conditional functional dependency rules on big data. *Big Data Mining and Analytics*, 03(01):68, 2020.
[26] X. Li, G. Cong, A. Sun, and Y. Cheng. Learning travel time distributions with deep generative model. In *WWW*, pages 1017–1027, 2019.
[27] Y. Li, K. Fu, Z. Wang, C. Shahabi, J. Ye, and Y. Liu. Multi-task representation learning for travel time estimation. In *SIGKDD*, pages 1695–1704, 2018.
[28] W. Luo, H. Tan, L. Chen, and L. M. Ni. Finding time period-based most frequent path in big trajectory data. In *SIGMOD*, pages 713–724, 2013.
[29] L. V. D. Maaten and H. Geoffrey. Visualizing data using t-sne. *JMLR*, 9:2579–2605, 2008.
[30] B. Perozzi, R. Al-Rfou, and S. Skiena. Deepwalk: online learning of social representations. In *SIGKDD*, pages 701–710, 2014.
[31] K. F. Petty, P. Bickel, and et.al. Accurate estimation of travel times from single-loop detectors. *Transportation Research Part A*, 32(1):1–17, 1998.
[32] X. Qin, Y. Luo, N. Tang, and G. Li. Deepeye: An automatic big data visualization framework. *Big Data Mining and Analytics*, 1(1):75, 2018.
[33] M. Rahmani, E. Jenelius, and H. N. Koutsopoulos. Route travel time estimation using low-frequency floating car data. In *ITSC*, pages 2292–2297, 2013.
[34] X. Song, H. Kanasugi, and R. Shibasaki. Deeptransport: Prediction and simulation of human mobility and transportation mode at a citywide level. In *IJCAI*, pages 2618–2624, 2016.
[35] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI*, pages 4278–4284, 2017.
[36] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei. LINE: large-scale information network embedding. In *WWW*, pages 1067–1077, 2015.
[37] Y. Tong, Y. Chen, Z. Zhou, L. Chen, J. Wang, Q. Yang, J. Ye, and W. Lv. The simpler the better: A unified approach to predicting original taxi demands based on large-scale online platforms. In *SIGKDD*, pages 1653–1662, 2017.
[38] D. Wang, J. Zhang, W. Cao, J. Li, and Y. Zheng. When will you arrive? estimating travel time based on deep neural networks. In *AAAI*, pages 2500–2507, 2018.
[39] H. Wang, Y. Kuo, D. Kifer, and Z. Li. A simple baseline for travel time estimation using large-scale trip data. In *SIGSPATIAL*, pages 61:1–61:4, 2016.
[40] S. Wang, Z. Bao, J. S. Culpepper, Z. Xie, Q. Liu, and X. Qin. Torch: A Search Engine for Trajectory Data. In *SIGIR*, pages 535–544, 2018.
[41] Y. Wang, Y. Yuan, Y. Ma, and G. Wang. Time-dependent graphs: Definitions, applications, and algorithms. *Data Science and Engineering*, 4(4):352–366, 2019.
[42] Y. Wang, Y. Zheng, and Y. Xue. Travel time estimation of a path using sparse trajectories. In *SIGKDD*, pages 25–34. ACM, 2014.
[43] C. Wenk, R. Salas, and D. Pfoser. Addressing the need for map-matching speed: Localizing globalb curve-matching algorithms. In *SSDBM*, pages 379–388, 2006.
[44] H. Wu, Z. Chen, W. Sun, B. Zheng, and W. Wang. Modeling trajectories with recurrent neural networks. In *IJCAI*, pages 3083–3090, 2017.
[45] H. Yuan and G. Li. Distributed in-memory trajectory similarity search and join on road network. In *ICDE*, pages 1262–1273, 2019.
[46] H. Zhang, H. Wu, W. Sun, and B. Zheng. Deeptravel: a neural network based travel time estimation model with auxiliary supervision. In *IJCAI*, pages 3655–3661, 2018.
[47] J. Zhang, Y. Zheng, and D. Qi. Deep spatio-temporal residual networks for citywide crowd flows prediction. In *AAAI*, pages 1655–1661, 2017.
[48] J. Zhang, Y. Zheng, D. Qi, R. Li, and X. Yi. Dnn-based prediction model for spatio-temporal data. In *SIGSPATIAL*, 2016.
[49] R. Zhong, G. Li, K. Tan, L. Zhou, and Z. Gong. G-tree: An efficient and scalable index for spatial search on road networks. *IEEE Trans. Knowl. Data Eng.*, 27(8):2175–2189, 2015.