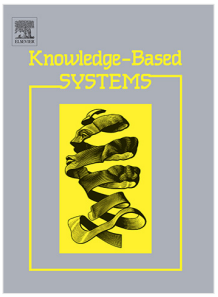
Accepted Manuscript



Parallel computing method of deep belief networks and its application to traffic flow prediction

Lu Zhao, Yonghua Zhou, Huapu Lu, Hamido Fujita

PII: S0950-7051(18)30511-2
DOI: https://doi.org/10.1016/j.knosys.2018.10.025
Reference: KNOSYS 4544

To appear in: *Knowledge-Based Systems*

Received date : 29 July 2018
Revised date : 12 October 2018
Accepted date : 14 October 2018

Please cite this article as: L. Zhao, et al., Parallel computing method of deep belief networks and its application to traffic flow prediction, *Knowledge-Based Systems* (2018), https://doi.org/10.1016/j.knosys.2018.10.025

Parallel computing method of deep belief networks and its application to traffic flow prediction

Lu Zhao [a], Yonghua Zhou [a,*], Huapu Lu [b,*], Hamido Fujita [c,*]

[a] *School of Electronic and Information Engineering, Beijing Jiaotong University, China*
[b] *Institute of Transportation Engineering, Tsinghua University, China*
[c] *Faculty of Software and Information Science, Iwate Prefectural University, Japan*

* Corresponding author.
E-mail address: yhzhou@bjtu.edu.cn (Y. Zhou), lulp@mail.tsinghua.edu.cn (H. Lu), hfujita-799@acm.org (H. Fujita)

Parallel computing method of deep belief networks and its application to traffic flow prediction

ABSTRACT

Deep belief networks (DBNs) with outstanding advantages of learning input data features have attained particular attention and are applied widely in image processing, speech recognition, natural language interpretation, disease diagnosis, among others. However, owing to large data, the training processes of DBNs are time-consuming and may not satisfy the requirements of real-time application systems. In this study, a single dataset is decomposed into multiple subdatasets that are distributed to multiple computing nodes. Multiple computing nodes learn the features of their own subdatasets. On the precondition of the remaining features where one computing node learns from the total dataset, the single dataset learning models and algorithms are extended to the cases where multiple computing nodes learn multiple subdatasets in a parallel manner. Learning models and algorithms are proposed for the parallel computing of DBN learning processes. A master–slave parallel computing structure is designed, where the slave computing nodes learn the features of their respective subdatasets and transmit them to the master computing node. The master computing node is critical in synthesizing the learned features from the respective slave computing nodes. The broadcast, synchronization, and synthesis are repeated until all features of subdatasets have been learned. The proposed parallel computing method is applied to traffic flow prediction using practical traffic flow data. Our experimental results verify the effectiveness of the parallel computing method of DBN learning processes in terms of decreasing pre-training and fine-tuning times and maintaining the prominent feature learning abilities.

Keywords: deep learning, deep belief network, parallel computing, traffic flow prediction

1. Introduction

A fast learning algorithm for the deep belief network (DBN) presented by Hinton *et al*. [1] triggered a research culmination on deep learning. Deep neural networks have been gradually recognized as possessing excellent capabilities of feature learning that cannot be rivaled by shallow neural networks. Using deep neural networks, favorable performances can be achieved in the fields of intelligent information processing such as face recognition, human mood discernment, medical image processing, process behavior prediction, classification and regression, and social relationship extraction [2-8]. In most tasks, deep learning algorithms can accomplish intelligence levels similar to or even better than those of certain human brain functions [9]. Although deep learning can fulfil superior traits in various fields, certain deficiencies still exist. Particularly, models and algorithms await further development when handling a large dataset in a parallel manner to realize prompt learning, for which an attempt is made to provide a solution in this study.

Deep learning models that are used extensively include the convolution neural network (CNN), long short-term memory (LSTM), autoencoder (AE), and deep belief network (DBN). The CNN exhibits powerful feature extraction through layered and cascaded learning and has been employed

to handle image processing tasks such as disease diagnosis [5] and face detection [10]. Meanwhile, LSTM includes memory blocks and is advantageous in handling sequence processing such as speech recognition [11], and symptom diagnosis such as disease judgment [12] and emergency event forecasting [13]. The AE can reconstruct input signals well; therefore, it can learn bilingual word embeddings accurately in natural language handling [14], and extract input features to predict recombination hotspots in the biological field [15]. Owing to its deep architecture that can automatically extract the most representative features, the DBN can perform well in fields of facial expression interpretation [2], time-series prediction [16], natural language handling, and word dependency learning [17]. Among all models and algorithms of deep learning, the DBN is developed earlier and used widely [18]; it can achieve favorable performances in terms of accuracy, stability, and anti-interference. However, the training processes are time-consuming when the DBN models encounter a large dataset. In practical applications, the speed performance is always a concern; thus, accelerated learning methods should be investigated.

Hitherto, models and algorithms in traffic flow prediction exist, such as the backpropagation neural network (BP-NN) and support vector machine (SVM). The BP-NN is of a simple structure, contains a concise training algorithm, and can be implemented for both single-step and multiple-step ahead traffic predictions [19]. Further, the SVM can dispose a small dataset and has been employed to manage many prediction tasks such as inter-urban traffic flow prediction [20] and freeway traffic flow prediction [21]. However, the structurally shallow models cannot reveal the intrinsic features of the input data well, depend excessively on the features of the input data, and require the manual extraction of the input data features, thereby consume a large amount of manpower and cannot be applied adaptively in practice. In this case, deep learning models and algorithms are introduced into traffic flow prediction, including DBN, AE, CNN, and LSTM, which are categorized into two types. The first type includes the DBN and AE that extract the features of traffic flow data by reconstructing the input data layer by layer. The DBN rebuilds the input traffic data with restricted Boltzmann machines (RBMs) [22], and the AE accomplishes the process by encoding and decoding the input data [23]. The other type involves the CNN and LSTM that extract the features of traffic data with a forward process. The CNN extracts the spatial and temporal features of traffic flow data with convolution and pooling layers [24], and LSTM fulfills the process using input, output, and forget gates [25].

Among all deep learning models, the DBN reconstructs training samples involving Gibbs sampling processes, and may be time-consuming when handling a large dataset. This study attempts to explore the parallelization models and algorithms of the DBN and applies them to the short-term real-time traffic flow prediction to verify the parallelization effectiveness. Generally, parallel computing can be divided into structure and data parallelization. The structure of parallel computing involves a complicated algorithm design, while the data parallel computing decomposes the total dataset into some subdatasets, distributes the subdatasets to some computing nodes, and the computing nodes execute the respective algorithms to handle the corresponding distributed subdatasets. Data parallel computing is a type of "coarse-grained" parallelization that can balance computing loads easily, decrease communication overheads, and reduce algorithm complexities. The primary problem in data parallel computing is that each computing node captures only the partial features of the total dataset. In this study, we develop a parallel computing method to theoretically guarantee that the gradually combined parameters can capture all the features of the total dataset. Moreover, the performance superiorities of the proposed

parallel computing method are validated using practical data.

The remainder of this paper is organized as follows. Section 2 introduces the DBN structure, and the serial training models and algorithms where the data are serially processed. Section 3 extends the serial models and algorithms to the cases of parallel computing where the subdatasets are handled synchronously. Section 4 describes the design of a traffic volume predictor based on the DBN. In Section 5, setting traffic volume prediction as a case, the effectiveness of the parallel computing method for the DBNs with the disposal of massive data is validated through real traffic flow data. The conclusions and prospects are presented in Section 6.

2. DBN basic principle

2.1. DBN structure

A DBN is a type of non-convolutional network that can be regarded as a stacked combination of several RBMs. Figure 1 shows the basic structure of a DBN composed of an input layer and r hidden layers. Every two adjacent layers construct an RBM, denoted sequentially as RBM1 to RBMr.

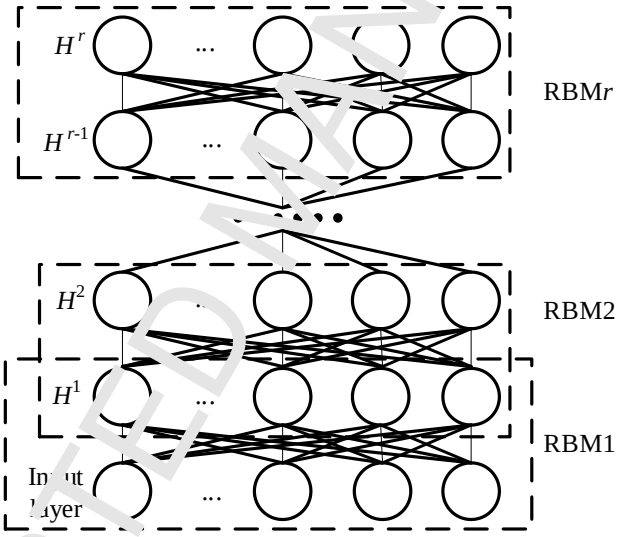


Fig. 1. Basic structure of a DBN.

An RBM is a two-layer graph, including a visible layer at the bottom and a hidden layer at the top, as shown in Fig. 2. The RBM is slightly different from the classical Boltzmann machine. In the RBM, only connections between nodes in the adjacent layers exist, and no connections exist between nodes in the same layer, thus simplifying the training processes and increasing the training efficiencies.

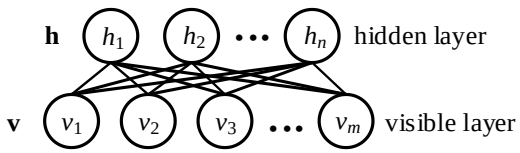


Fig. 2. Basic structure of an RBM.

We denote the node units in the visible and hidden layers as **v** and **h**, respectively. Each unit in the visible layer is represented as v_i (i = 1, 2, …, m), and each unit in the hidden layer as h_j (j = 1,

2, …, *n*), where *m* and *n* are the number of units in the visible and hidden layers, respectively. An RBM presumes that each unit should satisfy a binary distribution, denoted as $v_i \in \{0, 1\}$ and $h_j \in \{0, 1\}$. Each unit has an activation function $\sigma(x)$, typically selected as a sigmoid function $\sigma(x) = 1/(1+e^{-x})$. When a unit is situated at state 1, it is activated, and the weighted inputs plus the bias of the unit can be output. The outputs of the units in the lower layer are used as the inputs of the units in the upper layer.

The energy function of an RBM is formulated as

$$E(\mathbf{v},\mathbf{h}) = -\sum_{i=1}^{m} a_i v_i - \sum_{j=1}^{n} b_j h_j - \sum_{i=1}^{m}\sum_{j=1}^{n} v_i w_{ij} h_j \tag{1}$$

where w_{ij} is the weight between unit *i* in the visible layer and unit *j* in the hidden layer; a_i is the bias of unit *i* in the visible layer; b_j is the bias of unit *j* in the hidden layer. w_{ij}, a_i, and b_j are the parameters to be learned, presented collectively as **θ**.

According to the principle of statistical thermodynamics, a unit state (**v**, **h**) possesses the following joint probability distribution function:

$$P(\mathbf{v},\mathbf{h}) = e^{-E(\mathbf{v},\mathbf{h})} / \sum_{\mathbf{v},\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})} \tag{2}$$

where $\sum_{\mathbf{v},\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})}$ is a normalization factor. Similarly, other joint and conditional probability distribution functions are further defined as

$$P(\mathbf{v}) = \sum_{\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})} / \sum_{\mathbf{v},\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})} \tag{3}$$

$$P(\mathbf{h}|\mathbf{v}) = e^{-E(\mathbf{v},\mathbf{h})} / \sum_{\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})} \tag{4}$$

$$P(\mathbf{v}|\mathbf{h}) = e^{-E(\mathbf{v},\mathbf{h})} / \sum_{\mathbf{v}} e^{-E(\mathbf{v},\mathbf{h})} . \tag{5}$$

2.2. Training algorithm

2.2.1. Global framework

The process of training a DBN model consists of pre-training and fine-tuning two phases. The pre-training phase is to endow a DBN with the appropriate weights and biases that can capture the inherent features of the input training samples. The fine-tuning phase is to further adjust the weights and biases accurately through the error backpropagation (BP) algorithm based on the weights and biases obtained from the pre-training phase. The global framework of training a DBN model is demonstrated in Algorithm 1.

Algorithm 1: Training rules for a DBN model

Input: Training samples

Output: Updated weights and biases of a DBN model

// Pre-training phase

1: Initialize the structure and the weights and biases of a DBN.

2: **for** each RBM **do**

3: Obtain the inputs from the outputs of the lower RBM (training samples are the inputs for RBM1).
4: Train the RBM.
5: **end for**
// Fine-tuning phase
6: Initialize the structure and the weights and biases of a DBN model according to those acquired in the pre-training phase.
7: **for** each training epoch **do**
8: Update the weights and biases of the DBN model.
9: **end for**

2.2.2. Pre-training algorithm

The objective of pre-training an RBM is to reconstruct the training samples through tuning the parameters $\boldsymbol{\theta}$ such that the likelihood estimation is maximized. The log-likelihood function is described as

$$\ln L(\boldsymbol{\theta}) = \ln \prod_{s=1}^{S} P(\mathbf{v}_s) = \sum_{s=1}^{S} \ln P(\mathbf{v}_s) \tag{6}$$

where S is the number of training samples, and $\mathbf{v}_s$ is the states of the units in the visible layer related to the s-th training sample, representing one type of possible joint state.

The gradient of the log-likelihood function is

$$\frac{\partial \ln L(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = \sum_{s=1}^{S} \frac{\partial \ln P(\mathbf{v}_s)}{\partial \boldsymbol{\theta}}. \tag{7}$$

In Eq. (7), the derivative of $\ln P(\mathbf{v}_s)$ with regard to $\boldsymbol{\theta}$ is deduced as

$$\begin{aligned} \frac{\partial \ln P(\mathbf{v}_s)}{\partial \boldsymbol{\theta}} &= \frac{\partial}{\partial \boldsymbol{\theta}} \left(\ln \sum_{\mathbf{h}} e^{-E(\mathbf{v}_s, \mathbf{h})} - \ln \sum_{\mathbf{v}, \mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})} \right) \\ &= -\sum_{\mathbf{h}} P(\mathbf{h} \mid \mathbf{v}_s) \frac{\partial E(\mathbf{v}_s, \mathbf{h})}{\partial \boldsymbol{\theta}} + \sum_{\mathbf{v}, \mathbf{h}} P(\mathbf{v}, \mathbf{h}) \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial \boldsymbol{\theta}} \end{aligned} \tag{8}$$

Subsequently, the gradient of the log-likelihood function is rewritten as follows:

$$\frac{\partial \ln L(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = \sum_{s=1}^{S} \left(\sum_{\mathbf{h}} P(\mathbf{h} \mid \mathbf{v}_s) \frac{\partial E(\mathbf{v}_s, \mathbf{h})}{\partial \boldsymbol{\theta}} + \sum_{\mathbf{v}, \mathbf{h}} P(\mathbf{v}, \mathbf{h}) \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial \boldsymbol{\theta}} \right). \tag{9}$$

According to the gradient descent approach, the weight update function at step t is described as

$$\boldsymbol{\theta}(t) = \boldsymbol{\theta}(t-1) + \lambda_p \frac{\partial \ln L(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \tag{10}$$

where λ_p is the learning rate in the pre-training phase.

To avoid the local minimum, a momentum factor is introduced to the training processes. Further, Eq. (10) is amended to

$$\boldsymbol{\theta}(t) = m_p \boldsymbol{\theta}(t-1) + \lambda_p \frac{\partial \ln L(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \tag{11}$$

where m_p is the momentum factor in the pre-training phase.

The gradient in Eq. (9) is difficult to be calculated directly; however, it can be solved using the

k-step contrastive divergence (CD-*k*) algorithm proposed by Hinton, *et al.* [1]. Next, assign a sample $\mathbf{X}_s$ in the training data set to the visible layer, denoted as $\mathbf{v}_s^{(0)}$. The basic principle of the CD-*k* algorithm can be represented using the following formulae:

$$P(h_{sj}^{(0)}=1|\,\mathbf{v}_s^{(0)})=\sigma(\sum_{i=1}^{m}w_{ij}v_{si}^{(0)}+b_j) \tag{12}$$

$$h_{sj}^{(0)}\sim P(h_{sj}^{(0)}|\,\mathbf{v}_s^{(0)}) \tag{13}$$

$$P(v_{si}^{(1)}=1|\,\mathbf{h}_s^{(0)})=\sigma(\sum_{j=1}^{n}w_{ij}h_{sj}^{(0)}+a_i) \tag{14}$$

$$v_{si}^{(1)}\sim P(v_{si}^{(1)}|\,\mathbf{h}_s^{(0)}) \tag{15}$$

$$P(h_{sj}^{(1)}=1|\,\mathbf{v}_s^{(1)})=\sigma(\sum_{i=1}^{m}w_{ij}v_{si}^{(1)}+b_j) \tag{16}$$

$$h_{sj}^{(1)}\sim P(h_{sj}^{(1)}|\,\mathbf{v}_s^{(1)}) \tag{17}$$

$$\vdots$$

$$P(h_{sj}^{(k)}=1|\,\mathbf{v}_s^{(k)})=\sigma(\sum_{i=1}^{m}w_{ij}v_{si}^{(k)}+b_j) \tag{18}$$

$$h_{sj}^{(k)}\sim P(h_{sj}^{(k)}|\,\mathbf{v}_s^{(k)}) \tag{19}$$

where Eqs. (12), (16), and (18) denote the activated probabilities of the units in the corresponding hidden layers; meanwhile, Eq. (14) indicates the activated probabilities of the units in the visible layer. The activated state of each unit in the hidden or visible layer is independent of each other. Equations (13), (15), (17), and (19) express the Gibbs sampling processes that determine the states of the units in the visible or hidden layer. For example, the state of $h_{sj}^{(0)}$ is determined by the following two steps: First, produce a random number *rand* satisfying the uniform distribution $\mathcal{U}(0, 1)$; If $P(h_{sj}^{(0)}|\,\mathbf{v}_s^{(0)})>rand$, $h_{sj}^{(0)}=1$, otherwise $h_{sj}^{(0)}=0$. If k = 1, sufficiently good results can be attained [1].

Successively, the weights and biases in the visible and hidden layers are updated as

$$w_{ij}(t)=m_p w_{ij}(t-1)+\lambda_p\frac{1}{S}\sum_{s=1}^{S}\left(P(h_{sj}^{(0)}=1|\,\mathbf{v}_s^{(0)})v_{si}^{(0)}-P(h_{sj}^{(1)}=1|\,\mathbf{v}_s^{(1)})v_{si}^{(1)}\right) \tag{20}$$

$$a_i(t)=m_p a_i(t-1)+\lambda_p\frac{1}{S}\sum_{s=1}^{S}(v_{si}^{(0)}-v_{si}^{(1)}) \tag{21}$$

$$b_j(t)=m_p b_j(t-1)+\lambda_p\frac{1}{S}\sum_{s=1}^{S}\left(P(h_{sj}^{(0)}=1|\,\mathbf{v}_s^{(0)})-P(h_{sj}^{(1)}=1|\,\mathbf{v}_s^{(1)})\right). \tag{22}$$

The weight and bias update rules of an RBM in the pre-training phase using the CD-1 algorithm is described in Algorithm 2.

Algorithm 2: Weight and bias update rules in the pre-training phase

Input: Training samples $\mathbf{X}_s$, (s=1, 2, …, S)

Output: Updated weights and biases of RBM*n* (*n*=1, 2, …, *r*)

1: Specify the number of epochs to train RBM*n* (*n*=1, 2, …, *r*).

//Gibbs sampling

2: **for** each training sample **do**

3: Assign a training sample $\mathbf{X}_s$ as $\mathbf{v}_s^{(0)}$.

4: **for** each unit in the hidden layer **do**

5: Calculate $P(h_{sj}^{(0)}=1|\mathbf{v}_s^{(0)})$ with Eq. (12).

6: Sample $h_{sj}^{(0)}$ with Eq. (13).

7: **end for**

8: **for** each unit in the visible layer **do**

9: Calculate $P(v_{si}^{(1)}=1|\mathbf{h}_s^{(0)})$ with Eq. (14).

10: Sample $v_{si}^{(1)}$ with Eq. (15).

11: **end for**

12: **for** each unit in the hidden layer **do**

13: Calculate $P(h_{sj}^{(1)}=1|\mathbf{v}_s^{(1)})$ with Eq. (16).

14: Sample $h_{sj}^{(1)}$ with Eq. (17).

15: **end for**

16: **end for**

//**Weight and bias update**

17: **for** each unit in the visible layer **do**

18: **for** each unit in the hidden layer **do**

19: Update the weights and biases of RBM*n* with Eqs. (20) – (22).

20: **end for**

21: **end for**

22: Repeat steps 2 to 21 until the specified number of training epochs is reached.

2.2.3. Fine-tuning algorithm

The second phase of training a DBN model is the fine-tuning phase; it involves further adjusting the weights and biases through the BP algorithm. In this phase, an output layer will be used to output the predicted values. The structure of a DBN with an output layer at the top, called a DBN model, as displayed in Fig. 3, consists of *r*+2 layers: DBN layers L^1~L^{r+1} and an output layer.

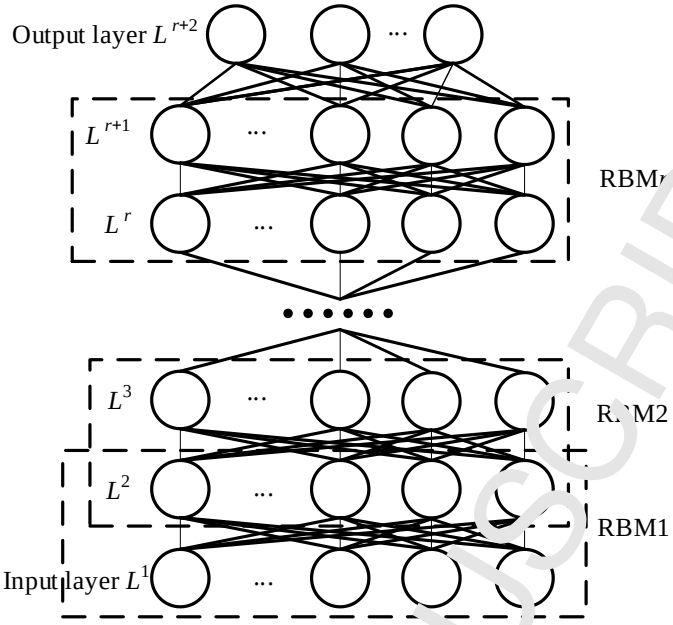


Fig. 3. Structure of a DBN model.

The objective of fine-tuning is to further acquire better parameters of a DBN model through minimizing the error function:

$$E = \frac{1}{2S}\sum_{s=1}^{S}\sum_{k=1}^{K^{r+2}}(y_{sk}\text{-}y_{sk})^2 \tag{23}$$

where K^{r+2} is the dimension of the outputs at layer L^{r+2}. y_{sk} and y_{sk} are the *k*-th real values and the DBN model output corresponding to the *s*-th training sample, respectively.

According to the gradient descent approach, the weights and biases are updated according to

$$\begin{aligned} w_{ji}^{u}(t) &= m_f w_{ji}^{u}(t-1) - \lambda_f \frac{\partial E}{\partial w_{ji}^{u}} \\ &= m_f w_{ji}^{u}(t-1) + \frac{\lambda_f}{S}\sum_{s=1}^{S}\delta_{sj}^{u} o_{si}^{u-1} \end{aligned} \tag{24}$$

$$\begin{aligned} c_{j}^{u}(t) &= m_f c_{j}^{u}(t-1) - \lambda_f \frac{\partial E}{\partial c_{j}^{u}} \\ &= m_f c_{j}^{u}(t-1) + \frac{\lambda_f}{S}\sum_{s=1}^{S}\delta_{sj}^{u} \end{aligned} \tag{25}$$

where $w_{ji}^{u}(t)$ is the weight at step *t* between unit *j* in the layer L^u, and unit *i* in the layer L^{u-1} ($2 \le u \le r+2$); $c_j(t)$ is the bias at step *t* of unit *j* in the layer L^u. m_f is the momentum factor, and λ_f is the learning rate in the fine-tuning phase. δ_{sj}^{u} is the generalized error of unit *j* propagated back from the layer L^u to L^{u-1}, and o_{si}^{u-1} is the output of unit *i* in the layer L^{u-1} regarding the data sample $\boldsymbol{X}_s$. δ_{sj}^{u} is updated as follows:

$$\delta_{sj}^{u}=\begin{cases}(y_{sj}-y_{sj})\dfrac{\partial\sigma_{j}^{u}}{\partial Net_{sj}^{u}} & u=r+2\\ \left(\sum\limits_{k=1}^{K^{u+1}}\delta_{sk}^{u+1}w_{kj}^{u+1}\right)\dfrac{\partial\sigma_{j}^{u}}{\partial Net_{sj}^{u}} & 2\le u\le r+1\end{cases} \tag{26}$$

where Net_{sj}^{u} is the net input of unit j in the layer L^u corresponding to the data sample $\boldsymbol{X}_s$, i.e., $Net_{sj}^{u}=\sum_{k=1}^{K^{u-1}}w_{jk}^{u}o_{sk}^{u-1}+c_{j}^{u}$. K^{u-1} is the number of units in the layer L^{u-1}. $\partial\sigma_{j}^{u}/\partial Net_{sj}^{u}=o_{sj}^{u}(1-o_{sj}^{u})$, if the activation function of unit j in the layer L^u is a sigmoid function. The weight and bias update rules in the fine-tuning phase are illustrated in Algorithm 3.

Algorithm 3: Weight and bias update rules in the fine-tuning phase

Input: Training samples $\boldsymbol{X}_s$, (s=1, 2, …, S)

Output: Updated weights and biases of a DBN model

```
for each layer do
    for each unit do
        Calculate the net input Net_sj^u and output o_sj^u of a unit.
        Calculate the generalized error δ_sj^u with Eq. (26).
    end for
end for
for each layer do
    for each unit do
        Update the weights and biases of the DBN model with Eqs. (24) and (25).
    end for
end for
```

3. Parallel computing method

Models and algorithms in machine learning require considering the real-time performance when processing massive data. Learning algorithms that are time-consuming will reduce the application efficiencies. With parallel computing, the learning algorithms are executed by multiple processors to improve the processing speeds. The method of parallel computing in this study attempts to accelerate the pre-training and fine-tuning phases and decrease the training time of a DBN model.

3.1. Parallel pre-training

The whole set of training samples is defined as X, and the number of training samples in X is S. Divide X into Q portions and distribute them to the corresponding Q computing nodes. The dataset at each node is defined as X_q (q=1, 2, …, Q). Subsequently, X is represented as the union set of X_q

(q=1, 2, …, Q):

$$X = \bigcup_{q=1}^{Q} X_q \ . \tag{27}$$

Consequently, Eq. (6) can be changed to

$$\begin{aligned} \ln L(\boldsymbol{\theta}) &= \ln\left(\prod_{s=1}^{S_1} P(\mathbf{v}_s) \prod_{s=1}^{S_2} P(\mathbf{v}_s) \cdots \prod_{s=1}^{S_Q} P(\mathbf{v}_s) \right) \\ &= \sum_{s=1}^{S_1} \ln P(\mathbf{v}_s) + \sum_{s=1}^{S_2} \ln P(\mathbf{v}_s) + \cdots + \sum_{s=1}^{S_Q} \ln P(\mathbf{v}_s) \end{aligned} \tag{28}$$

where S_q is the number of training samples in the subdataset X_q (q=1, 2, …, Q). Equation (28) implies that the log-likelihood function with regard to the total dataset X is the summation of the respective log-likelihood functions related to the subdatasets X_q (q=1, 2, …, Q).

Similarly, Eq. (7) can be converted to

$$\begin{aligned} \frac{\partial \ln L(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} &= \sum_{s=1}^{S_1} \frac{\partial \ln P(\mathbf{v}_s)}{\partial \boldsymbol{\theta}} + \sum_{s=1}^{S_2} \frac{\partial \ln P(\mathbf{v}_s)}{\partial \boldsymbol{\theta}} + \cdots + \sum_{s=1}^{S_Q} \frac{\partial \ln P(\mathbf{v}_s)}{\partial \boldsymbol{\theta}} \\ &= \sum_{q=1}^{Q} \sum_{s=1}^{S_q} \frac{\partial \ln P(\mathbf{v}_s)}{\partial \boldsymbol{\theta}} \end{aligned} \ . \tag{29}$$

Equation (29) expounds that the gradient of the log-likelihood function with respect to the learned parameters $\boldsymbol{\theta}$ can be calculated first, corresponding to different subdatasets X_q (q=1, 2, …, Q); subsequently, the results are cumulated into the gradient related to the total dataset X.

Subsequently, Eq. (11) is revised as follows:

$$\begin{aligned} \boldsymbol{\theta}(t) &= m_p \boldsymbol{\theta}(t-1) + \lambda_p \frac{\partial \ln L(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \\ &= m_p \boldsymbol{\theta}(t-1) + \lambda_p \sum_{q=1}^{Q} \sum_{s=1}^{S_q} \frac{\partial \ln P(\mathbf{v}_s)}{\partial \boldsymbol{\theta}} \end{aligned} \tag{30}$$

Equation (30) indicates that the renewal part of the learned parameters can be calculated according to the respective subdatasets X_q (q=1, 2, …, Q) in a parallel manner, but the inherited part is fetched from the learned parameters at the last time, t -1.

Ultimately, the weight and bias update of Eqs. (20)–(22) are transformed into the following:

$$w_{ij}(t) = m_p w_{ij}(t-1) + \lambda_p \frac{1}{S} \sum_{q=1}^{Q} \sum_{s=1}^{S_q} \left(P(h_{sj}^{(0)} = 1 \mid \mathbf{v}_s^{(0)}) v_{si}^{(0)} - P(h_{sj}^{(1)} = 1 \mid \mathbf{v}_s^{(1)}) v_{si}^{(1)} \right) \tag{31}$$

$$a_i(t) = m_p a_i(t-1) + \lambda_p \frac{1}{S} \sum_{q=1}^{Q} \sum_{s=1}^{S_q} (v_{si}^{(0)} - v_{si}^{(1)}) \tag{32}$$

$$b_j(t) = m_p b_j(t-1) + \lambda_p \frac{1}{S} \sum_{q=1}^{Q} \sum_{s=1}^{S_q} \left(P(h_{sj}^{(0)} = 1 \mid \mathbf{v}_s^{(0)}) - P(h_{sj}^{(1)} = 1 \mid \mathbf{v}_s^{(1)}) \right). \tag{33}$$

As shown in Eqs. (31)–(33), the total weight and bias variations corresponding to the original dataset X is the summation of the respective weight and bias variations corresponding to the subdatasets X_q (q=1, 2, …, Q) at all computer nodes. Therefore, parallel computing in the pre-training phase can be fulfilled through a master–slave structure. A master computing node is configured to be in charge of collecting the weight and bias variations from the slave computing

nodes, summing the weight and bias variations, updating the weights and biases, and releasing them to the slave computing nodes. Meanwhile, the slave computing nodes calculate the weight and bias variations corresponding to the respective subdatasets and transmit them to the master computing node. The process of parallel computing in the pre-training phase is revealed in Algorithm 4.

Algorithm 4: Pre-training rules with a parallel computing method

Input: Training dataset X

Output: Updated weights and biases of a DBN

1: Distribute X into Q nodes with Eq. (27).
2: Master node initializes the structure and initial parameters of a DBN.
3: **for** each training epoch **do**
4: Master node broadcasts the structure, weights, and biases of a DBN to slave nodes.
5: Each salve node calculates the weight and bias variations corresponding to X_q (q=1, 2, …, Q) with Eqs. (12) – (22).
6: Master node receives the weight and bias variations from each salve node.
7: Master node updates the weights and biases correspondingly to X with Eqs. (31) – (33).
8: **end for**

3.2. Parallel fine-tuning

After the model is pre-trained, the weights and biases will be fine-tuned to further adjust the weights and biases using the BP algorithm. Similar to the pre-training phase, the original training dataset X is divided into Q portions. The error function of Eq. (23) is revised as

$$\begin{aligned} E &= \frac{1}{2S}\sum_{q=1}^{Q}\sum_{s=1}^{S_q}\sum_{k=1}^{K^{r+2}}(y_{sk}-y_{sk})^2 \\ &= \frac{1}{S}\sum_{q=1}^{Q}E_q \end{aligned} . \tag{34}$$

Equation (34) demonstrates that the error function with regard to the total dataset X is the average of the error functions related to the respective subdatasets X_q (q=1, 2, …, Q).

Equations (24) and (25) are correspondingly altered to

$$\begin{aligned} w_{ji}^{u}(t) &= m_f w_{ji}(t-1) - \lambda_f \sum_{q=1}^{Q}\frac{\partial E_q}{\partial w_{ji}^{u}} \\ &= m_f w_{ji}^{u}(t-1) + \frac{\lambda_f}{S}\sum_{q=1}^{Q}\sum_{s=1}^{S_q}\delta_{sj}^{u}o_{si}^{u-1} \end{aligned} \tag{35}$$

$$\begin{aligned} c_j^{u}(t) &= m_f c_j^{u}(t-1) - \lambda_f \sum_{q=1}^{Q}\frac{\partial E_q}{\partial c_j^{u}} \\ &= m_f c_j^{u}(t-1) + \frac{\lambda_f}{S}\sum_{q=1}^{Q}\sum_{s=1}^{S_q}\delta_{sj}^{u} \end{aligned} . \tag{36}$$

As shown, the gradients of E to the weights and biases for dataset X is the summation of the

weight and bias variations related to all subdatasets X_q (q=1, 2, …, Q). Therefore, parallel computing in the fine-tuning phase can adopt the master–slave structure similar to that of the pre-training phase. This concrete procedure of parallel computing is depicted in Algorithm 5.

Algorithm 5: Fine-tuning rules with a parallel computing method

Input: Training dataset X

Output: Updated weights and biases of a DBN model

1: Distribute X to Q nodes with Eq. (27).
2: Master node initializes the structure and initial parameters of the DBN model.
3: **for** each training epoch **do**
4: Master node broadcasts the structure, weights, and biases to slave nodes.
5: Each slave node calculates the weight and bias variations $\partial E_q / \partial w_{ji}^u$ and $\partial E_q / \partial c_j^u$ regarding local subdatasets X_q (q=1, 2, …, Q) according to the rightmost summation items in Eqs. (24) and (25).
6: Master node receives the weight and bias variations from each slave node.
7: Master node updates the weights and biases corresponding to X with Eqs. (35) and (36).
8: **end for**

3.3. Parallel architecture

Figure 4 demonstrates the architecture of the parallel computing method, and is compared with that of the serial computing method. The serial method employs one computing node and a total dataset X; however, the parallel method makes avail of multiple computing nodes and multiple datasets X_q (q=1, 2, …, Q). The parallel method adopts the similar objective functions to those of the serial method in the pre-training and fine-tuning phases. Therefore, the optimization results of the parallel method may be theoretically in alignment with those of the serial method. Both the pre-training and fine-tuning phases exploit the master–slave computing structure. As shown in Fig. 4, in practice, Q computing nodes can be configured. The Q-th node, a master node, also acts as the slave node, and is responsible for the same computing tasks as those of the Q-1 slave nodes, in addition to the tasks of broadcasting, synchronization, and synthesis.

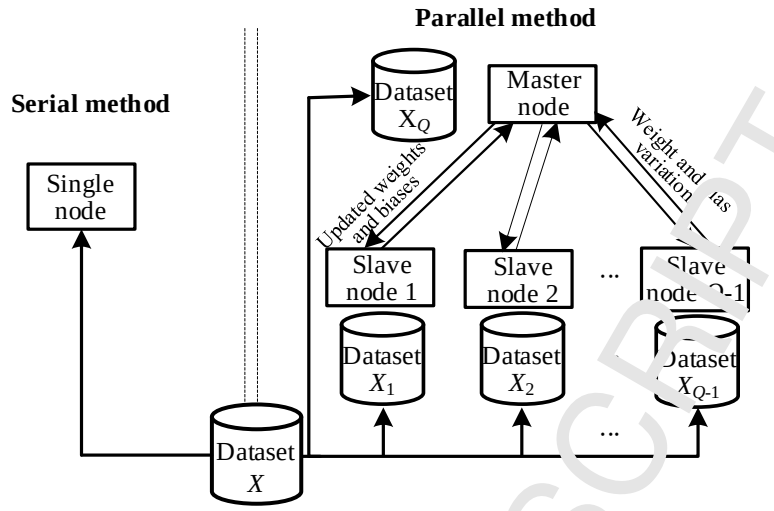


Fig. 4. Architecture of parallel computing and its comparison with that of serial computing.

Figure 5 further represents the similar flow charts for the pre-training and fine-tuning phases. The master node broadcasts the structure and parameters of the network and distributes the training datasets to all computing nodes. Each computing node reads its own local data, calculates the weight and bias variations, and transmits the calculation results to the master node. The master node synthetically processes the results transmitted from the respective computing nodes, and subsequently broadcasts the structure and updated parameters to each computing node. Such procedure is repeated until the end condition is satisfied.

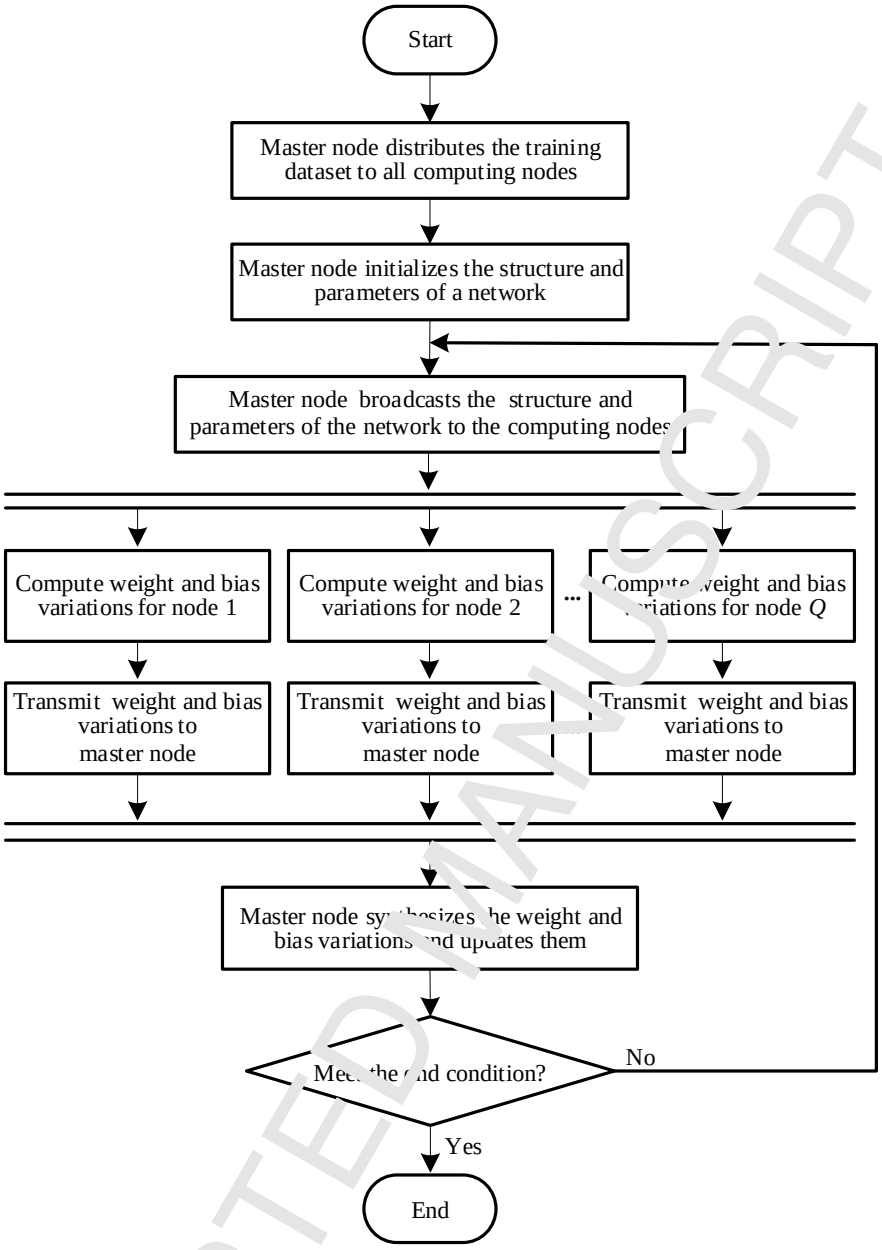


Fig. 5. Flow chart of parallel computing for the pre-training and fine-tuning phases.

4. Traffic flow prediction model based on DBN

Although stochastic fluctuations exist for urban traffic flow, a relatively stable law can be revealed for the traffic flow of urban roads. Therefore, according to traffic flow historical data, traffic flow information can be predicted. Traffic flow prediction benefits daily traveling and transportation management. The traffic flow of road segments is, in general, predicted in several time intervals in the future. Traffic flow datasets have become extremely large recently; consequently, traditional prediction methods cannot be applied effectively. Deep learning provides an effective solution to capture the features of massive traffic flow datasets [26]. DBNs have the advantages of learning the stable characteristic of input samples with randomness; this has enabled favorable application effectiveness to be achieved in fields such as handwriting, and voice and face recognitions [18]. Hence, the DBN is employed to capture the inherent characteristic of

stochastic traffic flow.

The output of a nonlinear system with single input and single output at instant t can be described as a nonlinear function related to the past outputs and inputs:

$$y(t) = g(x(t-1), x(t-2), \cdots, x(t-n_x), y(t-1), y(t-2), \cdots, y(t-n_y)) \tag{37}$$

where $x(t)$ and $y(t)$ are the system input and output at instant t, respectively. n_x and n_y are the adjustable parameters, and g is a nonlinear function to be identified. In Eq. (37), the sampling period T is omitted for the discrete time system.

Traffic volume is a major parameter of traffic flow that is often predicted for urban road segments. It is measured by the number of vehicles passing through an intersection within a time unit. The traffic volume of a road segment can be detected through sensors such as inductive loops and the global positioning system. Therefore, traffic volumes can act as inputs and outputs of a nonlinear traffic flow system that is described by Eq. (37). Suppose that $x(t-l_x)$ ($l_x = 1, 2, \cdots, n_x$) and $y(t-l_y)$ ($l_y = 1, 2, \cdots, n_y$) in Eq. (37) are the sampled traffic volumes at past instants. They are the inputs of the traffic flow prediction model described by a DBN model. The adjustable parameters n_x and n_y are regarded jointly as the number of input nodes of the DBN model that is to be tuned in the pre-training and fine-tuning phases. According to Eq. (37), if the traffic volume is predicted, an output layer, also called the regression layer, can be configured with one node to predict traffic volume. Consequently, the structure of a traffic flow prediction model based on the DBN model is formed as demonstrated in Fig. 6.

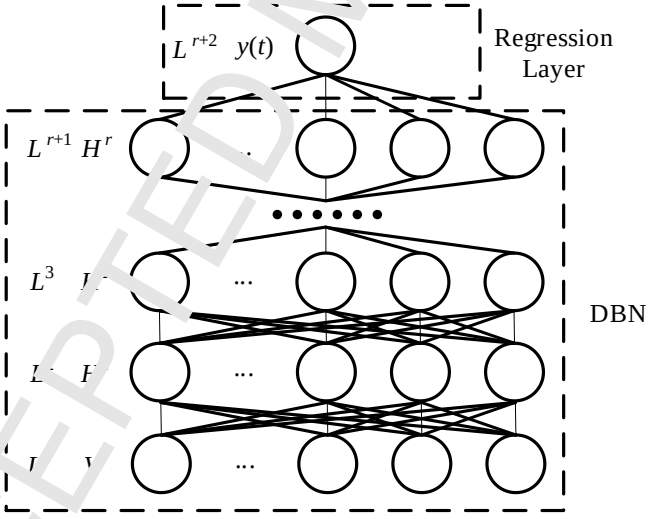


Fig. 6. Structure of traffic flow prediction model based on DBN model.

We denote the number of input nodes of traffic flow prediction model as K^1. The sampled dataset of the traffic volume is represented as $\{d(1), d(2), \cdots, d(D)\}$, and are sampled at instants T, $2T$, $\cdots$, DT respectively, where D is the number of sampled data. For short-term traffic flow prediction, the sampling period T, in general, ranges from 5 min to 15 min; in practice, one typical configuration is 5 min [27]. Consequently, the input dataset X and the output dataset Y for the DBN model are respectively constructed as follows:

$$X = \left\{ \mathbf{x}(t-1) \middle| \mathbf{x}(t-1) = [d(t-K^1), d(t-K^1+1), \cdots, d(t-1)] \right\} \tag{38}$$

$$Y = \left\{ y(t) \middle| y(t) = d(t) \right\} \tag{39}$$

where t moves forward and is indicated sequentially as $K^1+1,\ K^1+2,\ \cdots,\ K^1+S$.

Corresponding to the pre-training and fine-tuning phases, the traffic flow prediction model based on the DBN model can be categorized into the feature extraction model and the supervised learning model, respectively described as

$$F = g_{fe}(X) \tag{40}$$

$$\hat{Y} = g_{sl}(F) \tag{41}$$

where F is the learned features, g_{fe} is the feature extraction model from the input dataset X, $\hat{Y}$ is the output set, and g_{sl} is the supervised learning model. Through feature learning, the appropriate traffic flow features F are extracted from the original dataset X, and subsequently used as the inputs to the supervised learning model in Eq. (41). The supervised learning is to minimize the error functions as Eqs. (23) and (34), such that for any input $\mathbf{x}(t\text{-}1)$ at any instant t-1, an appropriate output $y(t)$ can be predicted.

5. Experimental results

We test the efficiency of the proposed parallel computing method of DBNs utilizing the traffic flow prediction model described in Section 4. First, we utilize the practical data to establish a DBN-based traffic flow prediction model by the serial training method. Subsequently, we compare the prediction results of the BP-NN (backpropagation neural network) and DBN-based traffic flow prediction models. Finally, we compare the pre-training and fine-tuning processes of the serial and parallel training processes, analyze the alignment of the training results, and illustrate the superiority of the parallel training speed.

5.1. Datasets

The datasets used in this study are sampled from the California Freeway Performance Measurement System that is utilized widely in transportation research [28]. The chosen road segment is Ashby Avenue in Alameda, California. The datasets are measured by inductive loop sensors and the time interval is 5 min. The actual traffic flow data of 38 months from January 2015 to February 2018 are divided into a training dataset and a testing dataset according to the ratio of 4:1. In other words, the training dataset is sampled from January 1, 2015 to March 30, 2017, and the testing dataset from March 31, 2017 to February 28, 2018. The training dataset is used to establish the prediction model, and the testing dataset is employed to test the prediction results. Figure 7 demonstrates a sample of the actual traffic flow data of Ashby Avenue during one week from September 11, 2017 to September 17, 2017.

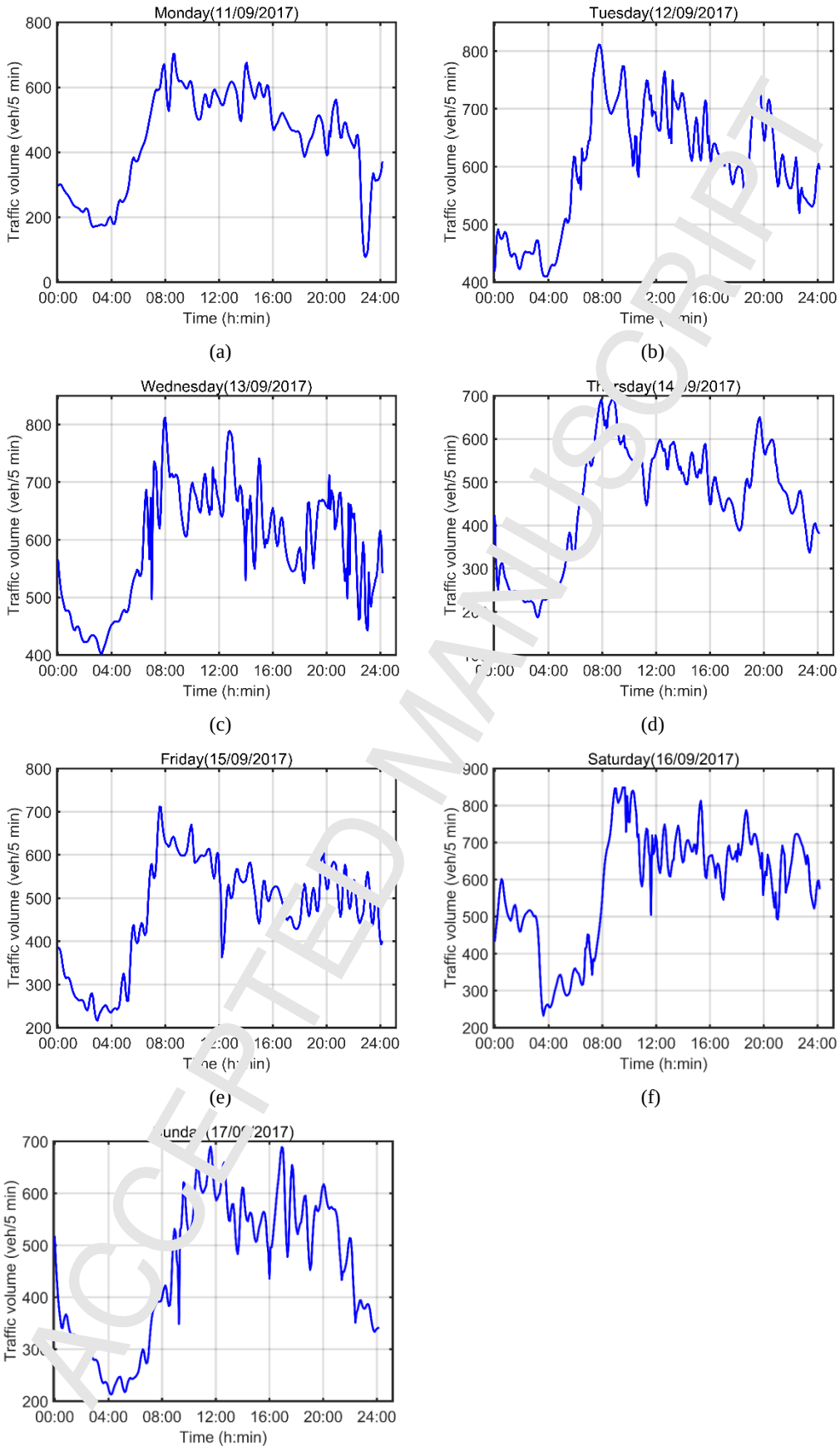


Fig. 7. Samples of traffic flow data during one week.

5.2. Error evaluation indices

To evaluate the prediction results of the traffic flow prediction model, three performance indices are defined, i.e., root mean square error (*RMSE*), mean absolute error (*MAE*), and mean absolute percentage error (*MAPE*). They are formulated as follows:

$$RMSE = \sqrt{\frac{1}{S}\sum_{s=1}^{S}(y_s - y_s)^2} \tag{42}$$

$$MAE = \frac{1}{S}\sum_{s=1}^{S}| y_s - y_s | \tag{43}$$

$$MAPE = \frac{1}{S}\sum_{s=1}^{S}| \frac{y_s - y_s}{y_s} | \tag{44}$$

where S is the number of datasets; y_s and y_s are the real and predicted value of the s-th sampling data, respectively.

5.3. DBN structure and parameter determination

The primary parameters of a DBN that to be determined include the number of nodes in the input and output layers, number of hidden layers, number of nodes in each hidden layer, number of epochs, learning rate, momentum factor, and the activation function. Currently, the accurate theoretical guidance for the determination of these parameters is insufficient. The method used to determine the parameters in this study is a combination of the cut-and-trial and empirical methods. The activation function is selected empirically as a sigmoid function.

The inputs of the DBN prediction model are the traffic volumes within last k intervals on the road segment, and the output is the prediction volume at the next interval. The effects of the number of input nodes on the output errors are shown in Fig. 8. The number of input nodes is finally determined to be three. Two hidden layers are first predetermined. The method of determining the number of hidden layer nodes is similar to that of the input layer. Ultimately, the DBN structure is confirmed as 3-100-100-1; that is, 3 input nodes, 2 hidden layers with 100 nodes existing in each hidden layer, and one output layer can achieve favorable results. The learning rates include the learning rate λ_p in the pre-training phase, and the learning rate λ_f in the fine-tuning phase. λ_p is adjusted to the range of [0.2, 1], and λ_f to the range of [0.2, 3]. Their effects on the errors are displayed in Fig. 9. Ultimately, λ_p is chosen as 0.6, and λ_f as 2.5. Similarly, the momentum factors m_p and n_f in the pre-training and fine-tuning phases are selected as 0.5 and 0.4, respectively. The influences of the number of epochs on the errors are shown in Fig. 10 in the fine-tuning phase. The number of epochs is finally determined as 1000 for the fine-tuning phase. By the cut-and-trial method, the number of epochs is confirmed as 200 for the pre-training phase.

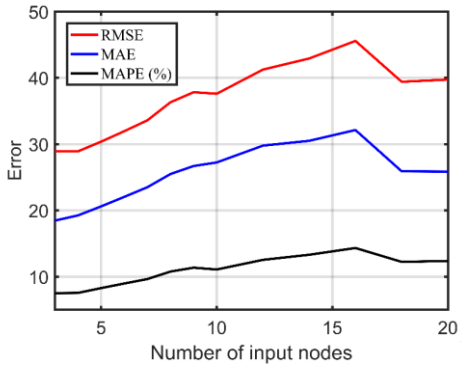
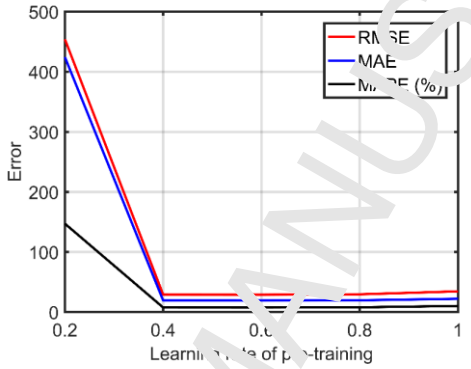
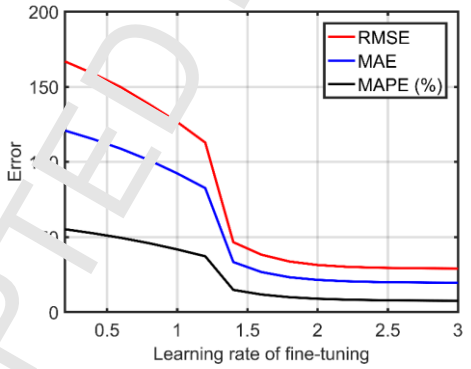


Fig. 8. Influences of the number of input nodes on the errors.



(a) Learning rate λ_p in the pre-training phase.



(b) Learning rate λ_f in the fine-tuning phase.

Fig. 9. Influences of the learning rates on the errors.

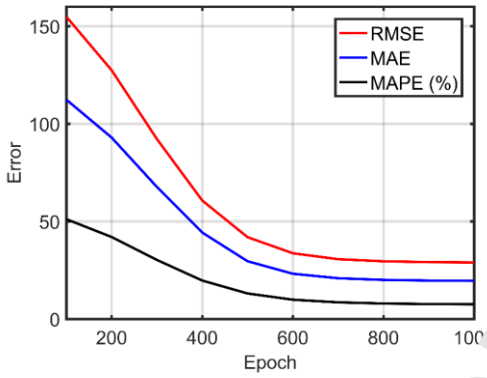


Fig. 10. Influences of the number of epochs on the errors in the fine-tuning phase.

5.4. Serial computing

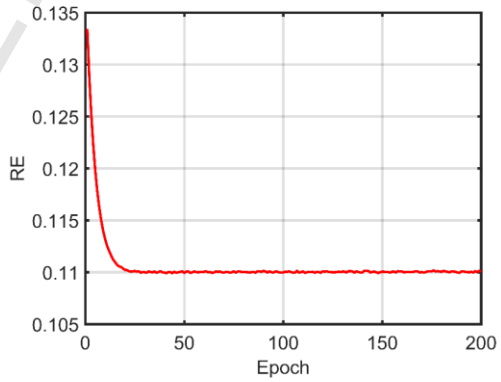
5.4.1. DBN training processes

Training an RBM can be regarded as a process of reconstructing and fitting the input data in the visible layer of an RBM. The effect of training an RBM can be evaluated by the reconstruction error (RE), which is described as follows:

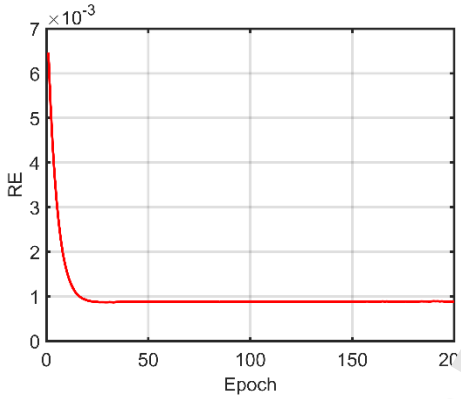
$$RE = \frac{1}{S}\sum_{s=1}^{S}\sum_{k=1}^{K}(v_{sk}^{\mathrm{In}} - v_{sk}^{\mathrm{Re}})^2 \tag{45}$$

where s represents a sample of input data, S the number of input data, k the input node, and K the dimension of the visible layer. v_{sk}^{In} denotes the input data sample s at node k in the visible layer, and v_{sk}^{Re} indicates the reconstructed data corresponding to the data sample s at node k in the visible layer.

The pre-training processes are to train the RBMs layer by layer. The learning processes of the DBN in the pre-training phase are illustrated in Fig. 11, where RBM1 represents RBM layer 1 and RBM2 layer 2. As shown, the RE of RBM1 declines regularly as the number of epochs increases, and RBM2 renders smaller decreasing *RE*s than RBM1 with the increase in epoch number. This fully demonstrates the convergence of the pre-training algorithm.



(a) RBM1.



(b) RBM2.

Fig. 11. Learning processes in the pre-training phase.

The learning process of the DBN model in the fine-tuning phase is illustrated in Fig. 12. As shown, the model becomes convergent throughout 1000 epochs. The training error corresponds to the training dataset, while the testing error corresponds to the testing dataset.

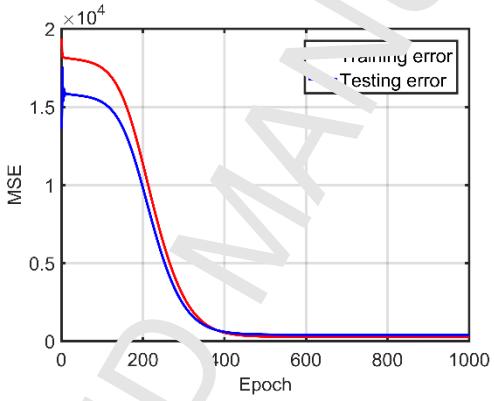
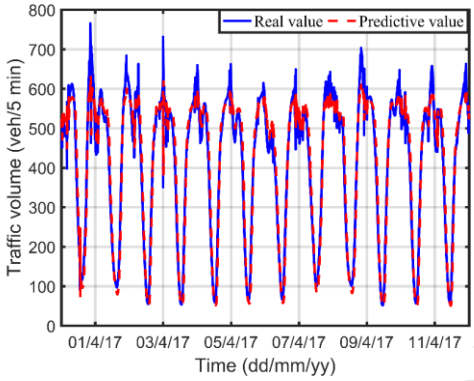


Fig. 12. Learning process in the fine-tuning phase.

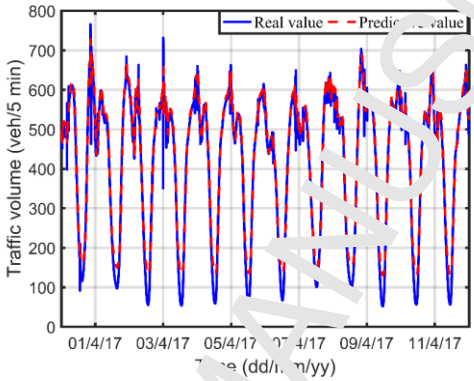
5.4.2. Comparison with BP-NN

To verify the effectiveness of the DBN model, the prediction results of the DBN model are compared with those of the BP-NN model, as shown in Fig. 13. The structure of the BP-NN model is optimally adjusted to 6-12-1, i.e., six input nodes, one hidden layer with 12 nodes, and one output node. The learning rate for the BP-NN model is set as 0.01, and the number of epochs is 500.

The testing dataset from March 31, 2017 to February 28, 2018 is utilized for the prediction of the BP-NN and DBN models. Figure 13 demonstrates the prediction results from March 31, 2017 to April 12, 2017. Although the testing data are not employed to train the BP-NN and DBN models, both models can reflect the fluctuation tendencies of traffic flow on Ashby Avenue (see Fig. 13), thus demonstrating the powerful description abilities by the nonlinear dynamics of these two models. However, the DBN model can predict more accurately for the peak hours of a day than the BP-NN model. The DBN model exhibits better fitting performances during the peak hours compared to the slack hours.



(a) BP-NN.



(b) DBN.

Fig. 13. Prediction results.

To illustrate the superiority of the DBN model over the BP-NN model quantitatively, the prediction errors of each model are compared using the total testing dataset, as shown in Table 1 and Fig. 14. As shown, the DBN model outperforms the BP-NN model. The prediction accuracies of the DBN model are approximately 14.99%, 10.70%, and 3.41% higher than those of the BP-NN model in terms of the three error evaluation indices, respectively. This exhibits the advantages of the DBN deep learning model used in traffic flow prediction.

Table 1

Comparison between the BP-NN and DBN prediction models.

Model	RMSE	MAE	MAPE (%)
BP-NN	43.6975	29.8890	10.8863
DBN	28.8045	19.1914	7.4761

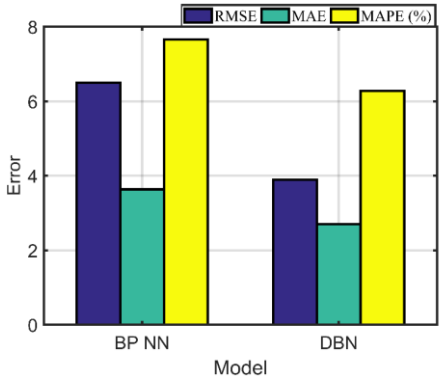


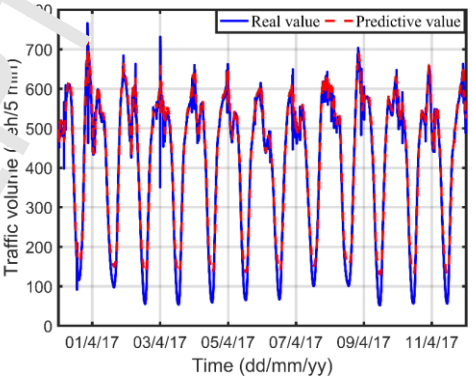
Fig. 14. Errors of the BP-NN and DBN prediction models.

5.5. Parallel computing

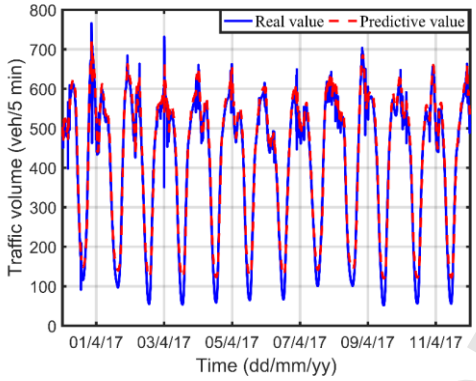
5.5.1. Prediction results

Parallel computing can enhance the processing ability of large data and improve the real-time performance. The proposed parallel computing method is implemented on the campus' high-performance computing platform. The hardware configuration is the Huawei Tecal CH121 servers, and the programming environment is Matlab2014b. Further, 16 nodes can be used in the parallel computing platform, and every four nodes are used as a computing group. One of these nodes is selected as a master node.

In the same dataset, the DBN model after training in the parallel computing platform should yield the similar prediction results as those obtained from the training algorithm of serial computing, for which only one computing node is utilized. Figure 15 demonstrates the prediction results of the DBN model using the training methods of serial computing and parallel computing with 16 nodes. As shown, the prediction results from the serial and parallel computing methods are almost similar, thereby justifying the alignment of the proposed parallel DBN training algorithms.



(a) Serial computing method.



(b) Parallel computing method.

Fig. 15. Prediction results of the DBN model using the serial and parallel methods.

5.5.2. Parallel computing evaluation indices

The effects of parallel computing are generally evaluated by evaluation indices such as acceleration ratio and efficiency, which are defined as follows:

$$S = \frac{T_S}{T_P} \tag{46}$$

$$P = \frac{S}{M_P} \tag{47}$$

where S is the acceleration ratio, and P is the efficiency. T_S is the runtime of a serial program, T_P is the runtime of a parallel program, and M_P is the number of computing nodes used in the parallel program. The processes of parallel DBN training are divided into the fine-tuning and pre-training phases. The following compares the runtimes, acceleration ratios, and efficiencies of these two phases and those of the whole training process.

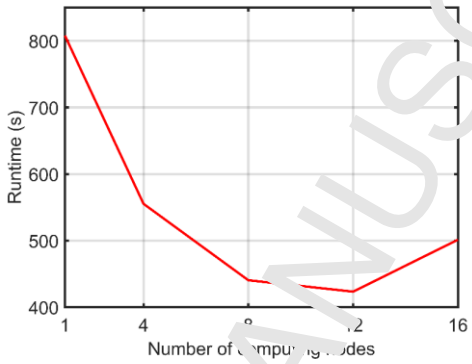
5.5.3. Parallel computing in the pre-training phase

Parallel computing in the pre-training phase refers to the parallelization of RBM training. Table 2 and Fig. 16 display the performance comparison in the pre-training phase between the serial and parallel training methods. The runtime of the serial program in the pre-training phase is 807.9532 s; however, the runtime of the parallel program with 12 computing nodes has reduced to 423.2987 s. When the number of computing nodes continues increasing to 16, owing to the communication overheads between computing nodes, the runtime increases to 501.2908 s instead. Figure 16 (a) represents such concave function of the runtime against the number of computing nodes. The acceleration ratio increases when the number of computing nodes is less than and equal to 12; on the contrary, it decreases when the number of computing nodes increases to 16. Figure 16 (b) depicts the relationship between the acceleration ratio and the number of computing nodes, as a convex function. Because the computing speeds do not proportionally decrease with the increasing node number, and the communication overheads will increase correspondingly, the efficiencies present a slowly declining tendency with the increasing number of computing nodes.

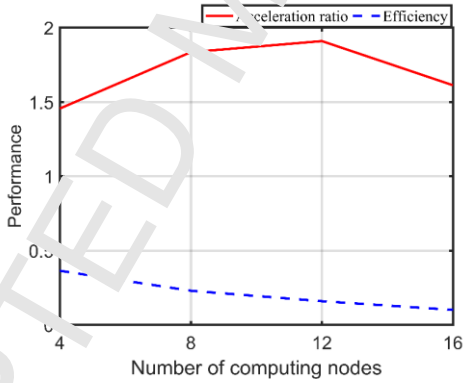
Table 2

Comparison between the serial and parallel training methods in the pre-training phase.

Method	Runtime (s)	Acceleration ratio	Efficiency
Serial	807.9532	-	-
Parallel with 4 nodes	555.2277	1.4552	0.36[illegible]8
Parallel with 8 nodes	440.4647	1.8343	0.2293
Parallel with 12 nodes	423.2987	1.9087	0.1591
Parallel with 16 nodes	501.2908	1.6117	0.[illegible]



(a) Runtime.



(b) Acceleration ratio and efficiency.

Fig. 16. Performance variations of parallel computing in the pre-training phase.

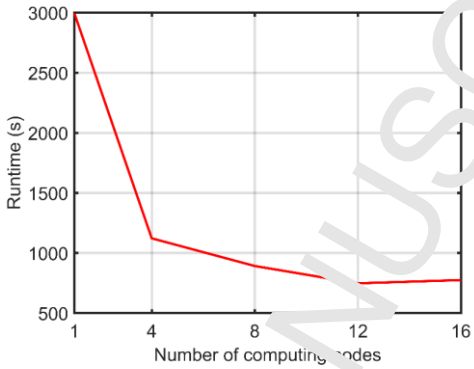
5.5.4. Parallel computing in the fine-tuning phase

Table 3 and Fig. 17 demonstrate the performance comparison in the fine-tuning phase between the serial and parallel training methods. In the fine-tuning phase, the runtime of serial computing is 2993.1 s. However, with the increasing number of computing nodes, the runtimes decrease until the node number reaches 12; when the node number is 16, the runtime begins to increase, as shown in Fig. 17 (a). The acceleration ratios manifest the opposite change tendency to that of the runtimes with increasing node number, and the efficiencies decrease monotonously, as depicted in Fig. 17 (b). The variation tendencies of parallel computing in the fine-tuning phase are similar to those in the pre-training phase.

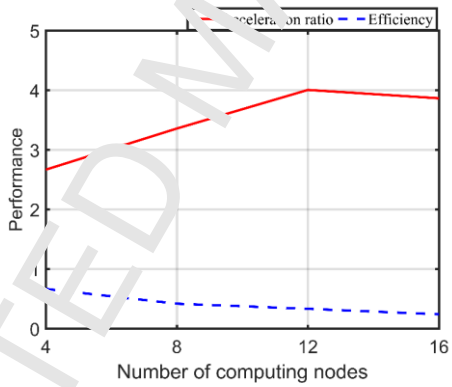
Table 3

Comparison between the serial and parallel methods in the fine-tuning phase.

Method	Runtime (s)	Acceleration ratio	Efficiency
Serial	2993.1000	-	-
Parallel with 4 nodes	1121.8000	2.6681	0.667[illegible]
Parallel with 8 nodes	891.3680	3.3579	0.4197
Parallel with 12 nodes	747.3352	4.0050	0.3338
Parallel with 16 nodes	774.4680	3.8647	0.2[illegible]15



(a) Runtime.



(b) Acceleration ratio and efficiency.

Fig. 17. Performance variations of parallel computing in the fine-tuning phase.

5.5.5. Parallel computing in the whole process

The overall comparisons of the parallel and serial methods are revealed in Table 4 for the whole training process. As shown in Table 4, the errors of the parallel method are in alignment with that of the serial method. Figure 18 shows the errors of the serial and parallel methods graphically. As shown, the parallel method, which divides a dataset into several subdatasets and employs these subdatasets to train the DBN models, can achieve almost similar training errors under the control of the master node, compared with the serial method that trains the DBN model using the whole training dataset. However, regarding the computing speed, the runtime of 12 nodes reduces by 68.1679%, compared with that of the serial method; this is a distinct speed improvement of the parallel method. The time reductions of the parallel method are shown in Fig. 19, which are compared with the runtime of the serial method. The parallel method with 12 nodes

demonstrates an acceleration ratio of 3.1415 compared with the serial method. The performances of the parallel computing method are shown in Fig. 20. When the number of computing nodes is less than 12, the training runtime decreases and the acceleration ratio increases continuously as the number of computing nodes increases. However, the efficiency slightly falls with the increase of node number, which is the average acceleration rate of every computing node.

Table 4

Comparison between serial and parallel methods in the whole process.

Model	RMSE	MAE	MAPE (%)	Runtime (s)	Time reduction (%)	Acceleration ratio	Efficiency
Serial	29.5103	19.5350	7.8691	3812.5000	-	-	-
Parallel with 4 nodes	28.8944	19.2163	7.5320	1708.3000	55.1921	2.2318	0.5580
Parallel with 8 nodes	29.0653	19.6304	7.6057	1364.9000	64.[illegible]993	2.7932	0.3492
Parallel with 12 nodes	29.2378	19.7748	7.7284	1213.6000	68.1679	3.1415	0.2618
Parallel with 16 nodes	28.9664	19.5470	7.5308	1313.4000	[illegible].5502	2.9028	0.1814

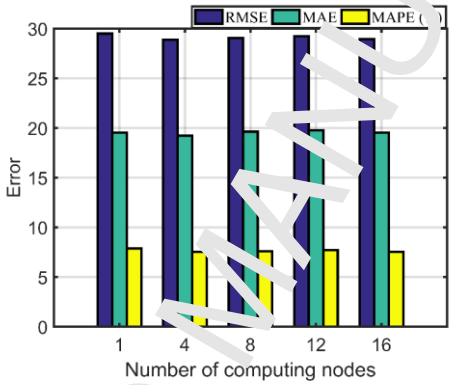


Fig. 18. Errors of the serial and parallel methods.

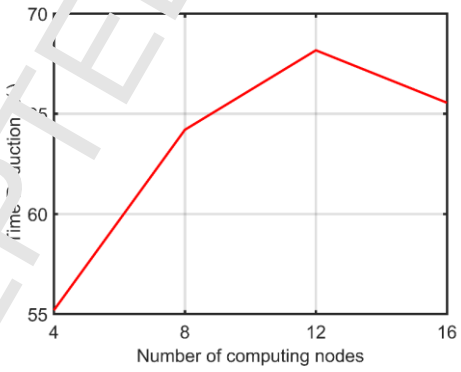
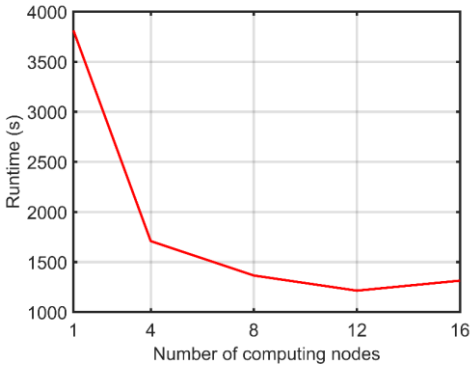
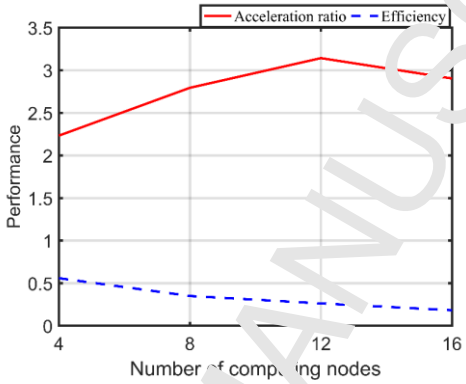


Fig. 19. Time reductions of the parallel method.



(a) Runtime.



(b) Acceleration ratio and efficiency.

Fig. 20. Performance variation of parallel computing in the whole process.

6. Conclusions

Based on the principle of deep learning, neural networks, and parallel computing, a parallel computing method was proposed for the DBN learning processes. From the learning objectives in the DBN pre-training and fine-tuning phases with a single dataset, the parallel computing method with multiple learning subdatasets was directly deduced. It is a type of data parallel processing method that decomposes a large dataset into some subdatasets, and distribute them to the respective computing nodes to be processed in an independent and parallel manner. A master–slave parallel computing structure was developed to involve the complete features of the total dataset. The slave computing nodes perform local learning according to the respective subdatasets. After all subdatasets have been learned, the learned weight and bias variations will be transmitted to the master node for synthesis. The synthesized weight and bias variations are issued by the master computing node to the slave nodes. Such procedure is repeated such that the master computing node captures all the features of all subdatasets. The parallel computing method was applied to traffic flow prediction. Some parts of the practical data were utilized for traffic volume feature learning, and other parts were applied to the prediction. The experimental results verified that the parallel computing method of the DBN learning processes with multiple subdatasets could achieve prediction accuracies similar to those of the DBN learning processes with a massive single dataset; however, the learning times were reduced. The acceleration performances were observed in the pre-training and fine-tuning phases. However, an optimum value appeared in terms of the number of computing nodes for a specific dataset because the learning time did not decrease

proportionally with the increasing number of computing nodes, which occurred with the increasing communication overheads between computing nodes.

Although DBNs have been successfully applied to many fields, many problems are still pending. The feature extraction was to treat the learning processes as a "black box" but the fault adjustment and gradual learning mechanism such as human brains were not incorporated explicitly into the learning processes. In this study, the DBN parallel computing method was applied to traffic flow prediction. The parallel computing methods of other deep learning models and the performance comparisons for traffic flow prediction will be studied in the future.

Acknowledgements

This work was financially supported by the National Natural Science Foundation of China (Grant No. 61673049 and No. 61074138) and the Fundamental Research Funds for the Central Universities of China (Grant No. 2017JBM302).

References

[1] G.E. Hinton, S. Osindero, Y.W. Teh, A fast learning algorithm for deep belief nets, Neural Computation 18 (7) (2006) 1527-1554.

[2] M.Z. Uddin, M.M. Hassan, A. Almogren, A. Alamri, M. Alrubaian, G. Fortino, Facial expression recognition utilizing local direction-based robust features and deep belief network, IEEE Access 5 (2017) 4525-4536.

[3] A. Polyak, L. Wolf, Channel-level acceleration of deep face representation, IEEE Access 3 (2015) 2163-2175.

[4] D. Liu, Y. Jiang, M. Pei, S. Liu, Emotional image color transfer via deep learning, Pattern Recognition Letters 110 (2018) 16-22.

[5] U.R. Acharya, H. Fujita, O.S. Lih, M. Adam, J.H. Tan, C.K. Chua, Automated detection of coronary artery disease using different durations of ECG segments with convolutional neural network, Knowledge-Based Systems 132 (2017) 62-71.

[6] J. Evermann, J.R. Rehse, P. Fettke, Predicting process behavior using deep learning, Decision Support Systems 100 (2017) 129-140.

[7] Z. Qi, B. Wang, Y. Tian, P. Zhang, When ensemble learning meets deep learning: a new deep support vector machine for classification, Knowledge-Based Systems 107 (2016) 54-60.

[8] J. Leng, P. Jiang, A deep learning approach for relationship extraction from interaction context in social manufacturing paradigm, Knowledge-Based Systems 100 (2016) 188-199.

[9] M. Stettler, G. Francis, Using a model of human visual perception to improve deep learning, Neural Networks 104 (2018) 40-49.

[10] Q. Tao, S. Zhan, X. Li, T. Kurihara, Robust face detection using local CNN and SVM based on kernel combination, Neurocomputing 211 (2016) 98-105.

[11] M. Cai, J. Liu, Maxout neurons for deep convolutional and LSTM neural networks in speech recognition, Speech Communication 77 (2016) 53-64.

[12] A. Zhao, L. Qi, J. Dong, H. Yu, Dual channel LSTM based multi-feature extraction in gait for diagnosis of Neurodegenerative diseases, Knowledge-Based Systems 145 (2018) 91-97.

[13] B. Cortez, B. Carrera, Y.J. Kim, J.Y. Jung, An architecture for emergency event prediction using LSTM recurrent neural networks, Expert Systems With Applications 97 (2018) 315-324.

[14] J. Su, S. Wu, B. Zhang, A neural generative autoencoder for bilingual word embeddings, Information Sciences 424 (2018) 287-300.

[15] A. Nath, S. Karthikeyan, Enhanced prediction of recombination hotspots using input features extracted by class specific autoencoders, Journal of Theoretical Biology 44[illegible] (2018) 73-82.

[16] X. Sun, T. Li, Q. Li, Y. Huang, Y. Li, Deep belief echo-state network and its application to time series prediction, Knowledge-Based Systems 130 (2017) 17-29.

[17] I. Chaturvedi, Y.S. Ong, I.W. Tsang, R.E. Welsch, E. Cambria, Learning word dependencies in text by means of a deep recurrent belief network, Knowledge-Based Systems 108 (2016) 144-154.

[18] L. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT Press, Cambridge, 2017.

[19] B.G. Cetiner, M. Sari, Q. Borat, A neural network based traffic flow prediction model, Mathematical & Computational Applications 15 (2) (2010) 269-278.

[20] W. Hong, Y. Dong, F. Zheng, C. Lai, Forecasting urban traffic flow by SVR with continuous ACO, Applied Mathematical Modelling 35 (3) (2011) 1282-1291.

[21] X. Wang, K. An, L. Tan, X. Chen, Short term prediction of freeway exiting volume based on SVM and KNN, International Journal of Transportation Science and Technology 4 (3) (2015) 337-352.

[22] W. Huang, G. Song, H. Hong, K. Xie, Deep architecture for traffic flow prediction: Deep belief networks with multitask learning, IEEE Transactions on Intelligent Transportation Systems 15 (5) (2014) 2191-2201.

[23] T. Zhou, G. Han, X. Xu, Z. Lin, C. Han, Y. Huang, J. Qin, δ-agree AdaBoost stacked autoencoder for short-term traffic flow forecasting, Neurocomputing 247 (2017) 31-38.

[24] Y. Wu, H. Tan, L. Qin, B. Ran, Z. Jiang, A hybrid deep learning based traffic flow prediction method and its understanding, Transportation Research Part C 90 (2018) 166-180.

[25] Y. Tian, K. Zhang, J. Li, X. Lin, B. Yang, LSTM-based traffic flow prediction with missing data, Neurocomputing 318 (2018) 297-305.

[26] N.G. Polson, V.O. Sokolov, Deep learning for short-term traffic flow prediction, Transportation Research Part C: Emerging Technologies 79 (2017) 1-17.

[27] D. Yu, C. Liu, Y. Wu, S. Liao, T. Anwar, W. Li, C. Zhou, Forecasting short-term traffic speed based on multiple attributes of adjacent roads, Knowledge-Based Systems (2018), https://doi.org/10.1016/j.knosys.2018.09.003.

[28] A.A. Kurzhanskiy, P. Varaiya, Guaranteed prediction and estimation of the state of a road network, Transportation Research Part C: Emerging Technologies 21 (1) (2012) 163-180.

A parallel computing method for the DBN pre-training and fine-tuning phases is proposed.
The parallel computing method is based on the master-slave and data parallel processing structure.
The parallel computing method is applied to traffic flow prediction.
Experimental results testify the effectiveness of the parallel computing method.
The performances are compared between the serial and parallel computing method.