J Supercomput (2015) 71:3037–3053
DOI 10.1007/s11227-015-1426-8

Multi-step-ahead host load prediction using autoencoder and echo state networks in cloud computing

Qiangpeng Yang[1] · Yu Zhou[1] · Yao Yu[1] · Jie Yuan[1] · Xianglei Xing[2] · Sidan Du[1]

Published online: 19 April 2015

Abstract Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction. There are many proposals for resource management approaches for cloud infrastructures, but effective resource management is still a major challenge for the leading cloud infrastructure operators (e.g., Amazon, Microsoft, Google), because the details of the underlying workloads and the real-world operational demands are too complex. Among those proposals, accurate host load prediction is one of the most effective measures to address this challenge. In this paper, we proposed a new method for host load prediction, which uses an autoencoder as the pre-recurrent feature layer of the echo state networks. The aim of our proposed method is to predict the host load in the future interval based on Google cluster usage dataset. Experiments performed on Google load traces show that our proposed method achieves higher accuracy than the state-of-the-art methods.

Keywords Host load prediction · Autoencoder · Echo state networks

✉ Qiangpeng Yang
yqp0424@gmail.com

Yu Zhou
nackzhou@nju.edu.cn

[1] School of Electronic Science and Engineering, Nanjing University, Nanjing, China

[2] Harbin Engineering University, Harbin, China

1 Introduction

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction [18]. A cloud is a type of parallel and distributed system, which means the ability to run a program or application on many connected computers at the same time. Although cloud computing has been widely adopted by industries, many existing issues have not been fully addressed, whereas new challenges [28] continue to emerge from industry applications such as dynamic resource provisioning [23], virtual machine migration [19], server consolidation and energy management [8].

There have been many proposals for resource management approaches for cloud infrastructures, but effective resource management is still a major challenge for the leading cloud infrastructure operators (e.g., Amazon, Microsoft, Google), because the details of the underlying workloads and the real-world operational demands are too complex. The large-scale distributed applications on a cloud require service-based software, which has the ability of monitoring the system status changes, analyzing the acquired information, and adapting its service configuration while considering tradeoffs among multiple QoS features simultaneously [14]. In distributed systems, we often use some host load monitor tools to record the host load data as a time series [26]. The aim of this work is to predict the host load in the future interval based on the information obtained from the past load traces. In distributed systems, predicting the host load in advance can benefit resource allocation, improve resource utilization and enable the system to take prompt actions.

In 2011, Google released a substantial cluster usage dataset which used Run-time Monitor (RTM) to monitor the characteristics of the hosts at run time. As cloud environment is likely to be constructed from a variety of machine classes and heterologous applications, the Google cluster trace lacks precise information about the purpose of jobs and configuration of machines. Thus, we cannot use these information to infer what is running. Nevertheless, we could predict the host load in the future interval using the information of the past load trace.

In this paper, the load trace provided by Google has been used as our experimental dataset. Most of the studied cluster load traces have fallen into one of three following categories: long-running services, DAG-of-task systems and high-performance (or throughput) computing [20], while the Google cluster trace captures a range of behaviors that includes all the above categories, among others. To cope with such a mixture of these and other categories, we proposed a new approach for host load prediction. In our approach, we choose echo state network (ESN) as our basic prediction model, which is a kind of recurrent neural network (RNN) belonging to a collection of techniques called Reservoir Computing [13]. To achieve a better representation of the inputs, we introduce an autoencoder to learn the higher level feature of the input data instead of using the data directly. The pre-recurrent feature layer can further capture the similarity between load traces.

The rest of the paper is organized as follows. The related work is presented in Sect. 2. Section 3 describes the unsupervised feature learning algorithm. And the comparison between RNN and ESN is discussed in Sect. 4. In Sect. 5, we give the details of our proposed approach. Afterwards, Sect. 6 presents the experimental results and comparison. Finally, we conclude our work in Sect. 7.

2 Related work

Host load prediction has received focus from researchers for a long time due to its potential profits. Many previous efforts have been made toward the host load in traditional Grids or HPC systems. Akioka et al. [1] combined the Markov model and seasonal analysis to predict the host load for one-step-ahead in a computational Grid. Wu et al. [25] used a hybrid model for multi-step-ahead host load prediction, which combined the autoregressive (AR) model and a Kalman filter. Duy et al. [6] applied an artificial neural network (ANN) to the task of host load prediction. Among these methods, ANN is the most widely used approach to analyze the stochastic nonlinear system and show good performance in traditional distributed system.

However, according to the comparison of workloads between Cloud and Grid [4], the average noise in a Cloud is approximately 20 times larger than the average noise in a Grid. Therefore, predicting the host load in a Cloud is more complicated than that in a Grid. Many traditional methods for time series mining have been evaluated in Cloud, such as Linear Regression [8] and Auto Regressive Integrated Moving Average [29]. But these methods achieve limited accuracy when they are applied to the cloud environment.

Di et al. [5] first used the Bayesian model to predict the host load in a Cloud. They proposed 9 novel features to characterize the recent load fluctuation in the evidence window and could predict the mean load over consecutive time intervals. However, their method has two limitations. The first is that the training period in evaluation type B should contain the test period, which is not suitable for the cloud environment. The other is that they used an exponentially segmented pattern, which means that the length of the segment increases exponentially. With the growth of the segment length, the mean load cannot fully reflect the fluctuation of the host.

In [27], the authors combine the Phase Space Reconstruction (PSR) and Group Method of Data Handling method based on Evolutionary Algorithm (EA-GMDH) for host load prediction. The PSR method is used to reconstruct the load trace from single-dimensional time series to a multi-dimensional phase space, and the time series after reconstruction are used as the input of the EA-GMDH network. The prediction performance of this method is closely related to the parameters they chose, as the evolutionary algorithm is a stochastic global search method which may fall into local optima.

ESN is a rather recent development in the field of RNN and it lead to a fast, simple and constructive algorithm for supervised training of RNN. The basic idea of ESN is to use a large *reservoir* RNN as a supplier of interesting dynamics from which the desired output is combined. This idea has been independently discovered and investigated under the name of *liquid state machines* by Wolfgang Maass and collaborators [17]. The philosophy adopted in Reservoir Computing is to consider the recurrent layer as a large reservoir of nonlinear transformations of the input data and decouple the learning of parameters inside and outside the reservoir.

Unsupervised feature learning refers to a class of machine learning techniques, developed rapidly since 2006, where many stages of nonlinear information processing in hierarchical architectures are exploited for pattern classification. Recently, unsupervised feature learning technologies have been successfully used in many research areas, such as handwritten digit images recognition [9], visual object classification [21] and nature language process [10]. After initializing the deep neural network with unsupervised feature learning algorithms [e.g., autoencoder, matrix factorization and restricted Boltzmann machines (RBM)], the weights are starting at a better location in parameter space than if they had been randomly initialized [22]. Because the deep neural network can also be considered to perform feature learning, since they learn a representation of their input at the hidden layers which is subsequently used for classification or regression at the output layer.

In this paper, we propose a new framework which adds a pre-recurrent feature layer to the conventional ESN. The intuition that led into the feature learning layer is that by capturing the similarity between load traces, similar traces will have similar trajectories in the reservoir state space. The feature learning algorithm we used is an autoencoder neural network, because it is simple to implement and achieves satisfactory performance. We give more details about the Autoencoder in Sect. 3, and discuss the update of the weights of feature matrix in Sect. 5.2.

3 Autoencoder

An autoencoder neural network is an unsupervised learning algorithm that sets the target values to be equal to the inputs, which means the autoencoder tries to learn a function $f(x) \approx x$. In other words, it is trying to learn an approximation to the identity function, so that the output $\hat{x}$ is similar to x. By applying constraints on the network, such as by limiting the number of hidden units or the average activation of the hidden units, we can discover high-level features of the input data. According to recent research [9,10,22], the features learned automatically can improve the accuracy of the classification and regression tasks comparing with the features designed manually.

The input of the autoencoder neural network is a set of n unlabeled examples $x_u^{(1)}, x_u^{(2)}, \ldots, x_u^{(n)}$, where each $x_u^{(i)} \in \mathbb{R}^n$ is an example of the host load in the history window in our approach. Subscript u here indicates that it is an unlabeled example. The unlabeled data are used to learn a slightly higher level, more succinct representation of the inputs.

The network architecture is shown in Fig. 1. We denote the input host load by x and the network reconstruction by $\hat{x} = f(\mathbf{W}^{(2)} f(\mathbf{W}^{(1)}x + b^{(1)}) + b^{(2)})$, where $\mathbf{W}^{(1)}$, $\mathbf{W}^{(2)}$ are the weights, $b^{(1)}$, $b^{(2)}$ are the bias terms, and $f(\cdot)$ is the activation function in the form of:

$$f(z) = \frac{1}{1 + \exp(-z)} \tag{1}$$

The middle layer of the network is called hidden layer h, whose output is a new representation of the inputs. To remove the constraint of the number of hidden units, autoencoder impose a sparsity constraint on the hidden units. As a result, the autoen-

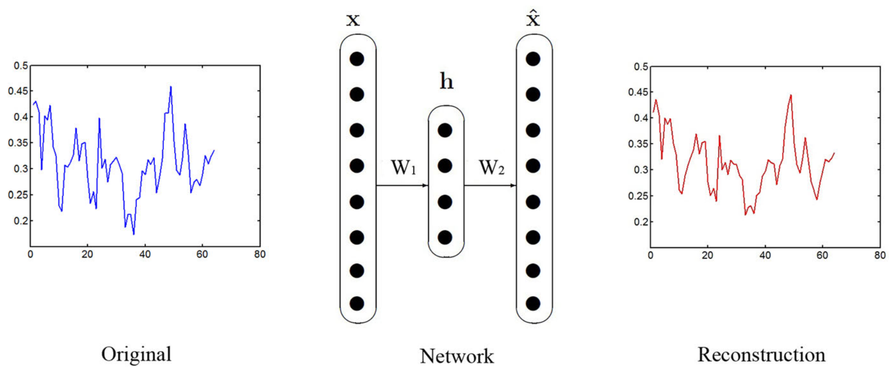


Fig. 1 Autoencoder network with input host load x (*left*) and reconstructed host load $\hat{x}$ (*right*)

coder still can discover high-level features in the data effectively, even if the number of hidden units is large.

Therefore, the optimization problem is to calculate the weights and bias item by minimizing the cost function $J(\mathbf{W}, b)$, whose form is as follows:

$$J(\mathbf{W}, b) = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{1}{2}\left\| f_{\mathbf{W},b}(x^{(i)}) - y^{(i)}\right\|^2\right) + \frac{\lambda}{2}\sum_{l=1}^{2}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(\mathbf{W}_{ji}^{(l)}\right)^2 + \beta\sum_{j=1}^{s_2}\mathrm{KL}(\rho\|\hat{\rho}_j) \tag{2}$$

The first term in cost function is to ensure each input $x_u^{(i)}$ to be reconstructed well by minimizing the average squares error, the second term is a regularization term that prevents overfitting, s_l is the number of the units in layer l, while the third term is to restrict that the activations to be sparse, which means most of the activations to be zero. In the third term,

$$\hat{\rho}_j = \frac{1}{m}\sum_{i=1}^{m} a_j^{(2)}(x^{(i)}) \tag{3}$$

which indicates the average activation of hidden unit j, where $a_j^{(2)}(x^{(i)})$ is the activation of the hidden unit j and ρ is the sparsity parameter. $\mathrm{KL}(\rho\|\hat{\rho}_j)$ is the Kullback–Leibler (KL) divergence [15], and can be derived by

$$\mathrm{KL}(\rho\|\hat{\rho}_j) = \rho \log\frac{\rho}{\hat{\rho}} + (1-\rho)\log\frac{1-\rho}{1-\hat{\rho}} \tag{4}$$

KL divergence is a standard function for measuring how different two distributions are, and the function has the property that $\mathrm{KL}(\rho\|\hat{\rho}_j) = 0$ if $\hat{\rho}_j = \rho$, and otherwise it increases monotonically as $\hat{\rho}_j$ diverges from ρ.

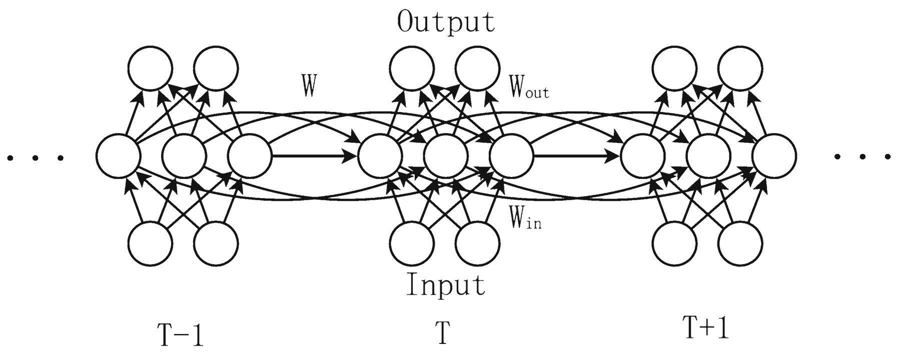


Fig. 2 The recurrent neural network

4 Echo state network

4.1 Recurrent neural network

RNN is an ANN with internal loops, which is an expressive model for sequence tasks. At each time step, the RNN receives an input, updates its hidden state, and makes a prediction, which is shown in Fig. 2. The RNN is powerful because it has a high-dimensional hidden state with nonlinear dynamics that enables it to remember and process past information. Even if the nonlinearity used by each unit is quite simple, iterating it over time leads to very rich dynamics.

The standard RNN is formalized as follows: given a sequence of input vectors $(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)$, the RNN computes the hidden states $(\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_n)$ and the outputs $(\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_n)$ using the following equations:

$$\mathbf{h}_t = f(\mathbf{W}_{\text{in}}\mathbf{x}_t + \mathbf{W}\mathbf{h}_{t-1}) \tag{5}$$

$$\mathbf{y}_t = \mathbf{W}_{\text{out}}\mathbf{h}_t \tag{6}$$

where $t = 1, 2, \ldots, n$, $\mathbf{W}_{\text{in}}$, $\mathbf{W}$, $\mathbf{W}_{\text{out}}$ are input-hidden, hidden-hidden, hidden-output connection' matrices.

The training algorithm of RNN is known as the back-propagation through time (BPTT) [24] method. The RNN is unfolded through time. Then back-propagation training is used to update the weights. However, Hochreiter [11] and Bengio [2] proved that the gradient decays exponentially as it is back-propagated through time, and argued that RNN cannot learn long-range temporal dependencies when gradient descent is used for training.

4.2 Echo state network

ESN [13] is a new RNN architecture, based on a rich reservoir of potentially interesting behavior. The reservoir of ESN is the recurrent layer formed of a large number of sparsely interconnected and randomly initialized recurrent layer composed of huge

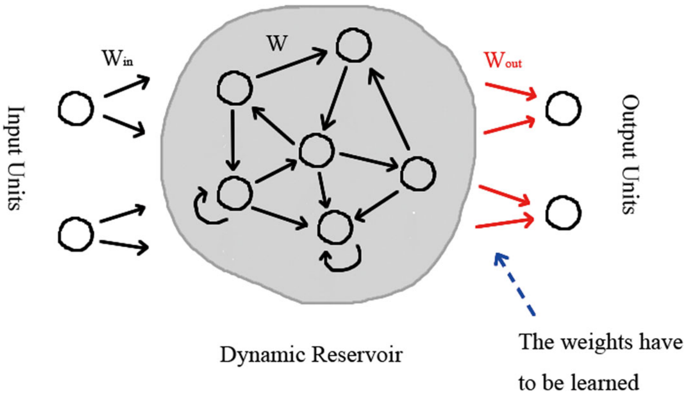


Fig. 3 The basic architecture of ESN

number of standard sigmoid units. Only output connections are modified during learning process. A significant advantage of this approach over standard RNN is that simple linear regression algorithms can be used for adjusting output weights. This is a much easier learning task and it works surprisingly well provided the recurrent connections are carefully initialized so that the intrinsic dynamics of the network exhibits a rich reservoir of temporal behaviors that can be selectively coupled to output.

The architecture of ESN used in this paper is shown in Fig. 3. A large RNN is used as a dynamic reservoir, which can be excited by suitably presented input and/or feedback output. The training algorithm of an ESN consists of the following steps:

(a) Generate a RNN randomly

The input-hidden matrix $\mathbf{W}_{\text{in}}$ and hidden-hidden matrix $\mathbf{W}$ are generated randomly. Once they have been generated, they will not change during the entire training process.

Only when a RNN has the "echo state property" [12] it can be used for dynamic system modeling. Echo state property means for each internal unit $\mathbf{h}_i$ there exits an echo function e_i such that the current state can be written as $\mathbf{h}_i = e_i(\mathbf{x}_t, \mathbf{x}_{t-1}, \ldots)$, the network state is an "echo" of the input history. The recent input presented to the network has more influence to the network state than an older input, the input influence gradually fades out. Echo states are crucial for successful operation of ESN, their existence is usually ensured by rescaling recurrent weight matrix $\mathbf{W}$ to specified spectral radius λ. This can be achieved by simply multiplying all elements of a randomly generated recurrent weight matrix with $\frac{\lambda}{\lambda_{\max}}$, where $\lambda_{\max}$ is the spectral radius of the original matrix.

(b) Feed the training data to the ESN

When the input units are fed to ESN, they will activate dynamics within the dynamic reservoir. At each time step, the internal dynamic reservoir states are computed according to Eq. (5). Then, we collect the input units $\mathbf{x}_i$ and $\mathbf{h}_i$ together as the i-th row of a matrix $\mathbf{M}$ at each time step. Meanwhile, the output data are collected into another matrix $\mathbf{T}$.

(c) Wash out the initial memory of the dynamic reservoir

Since the arbitrarily generated network states contain an initial memory which is not caused by the input, it is assumed that the effects of the arbitrary starting state have died out. So we only keep a part of matrix $\mathbf{M}$ and $\mathbf{T}$, the new matrices denoted as $\mathbf{M}^{\text{forget}}$ and $\mathbf{T}^{\text{forget}}$.

(d) Compute output weights

The output weights can be derived by the following equation:

$$\mathbf{W}_{\text{out}}^{\text{T}} = \text{pseudoinverse}(\mathbf{M}^{\text{forget}}) \cdot \mathbf{T}^{\text{forget}} \tag{7}$$

The reasons we chose ESN as our basic model are based on the following consideration:

- ESNs can be trained very fast because they just fit a linear model [16].
- ESNs work surprisingly well if we initialize weights carefully.
- ESNs can do impressive modeling of one-dimensional time series.

5 Our approach

In traditional ESNs, the reservoir constructs a representation of the input data by applying a complex nonlinear transformation to the input data. To achieve a better representation of the inputs, we introduce the autoencoder to learn the higher level feature of the input data instead of using the data directly. The intuition that led to the introduction of the autoencoder is that by capturing the similarity between load traces, similar traces will have similar trajectories in the reservoir state space. For example, if two load traces appear in similar context, most of the time they will have similar features and share the same function.

Another advantage of extracting features from the input data before feeding it to the reservoir is that the feature vectors of dimension is smaller than the input vector, which will reduce the computational complexity.

5.1 Model definition

The architecture of our approach is shown in Fig. 4. Suppose we have the input data $\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)$, the equations describing the network are:

$$\mathbf{h}_t = f(\mathbf{W}_{\text{in}} \cdot \mathbf{W}_f \cdot \mathbf{x}_t + \mathbf{W} \cdot \mathbf{h}_{t-1}) \tag{8}$$

$$\mathbf{y}_t = \mathbf{W}_{\text{out}} \cdot \mathbf{h}_t \tag{9}$$

where h_t is the output of the hidden units, f is the hidden units activation function, $\mathbf{W}_f$, $\mathbf{W}_{\text{in}}$, $\mathbf{W}$, and $\mathbf{W}_{\text{out}}$ are input-feature, feature-hidden, hidden-hidden, hidden-output connections' matrices, respectively.

Initialization strategies based on unsupervised pretraining of each layer have been shown to be important both for supervised and unsupervised training of deep architectures [3]. After initializing the deep neural network with unsupervised feature learning

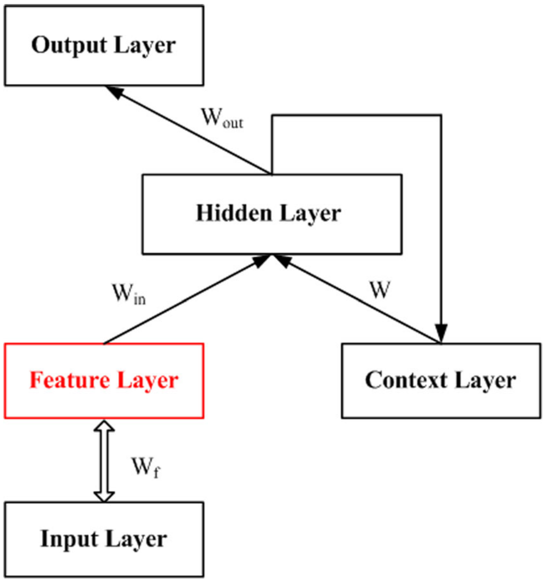


Fig. 4 Echo state network with an autoencoder

algorithms, the weights are starting at a better location in parameter space than if they had been randomly initialized. The ESN corresponds to a very deep architecture when unfolded in time, so we should ensure the quality of the learned weight matrices. It is found that initializing W_f with a trained autoencoder yields less noisy filters according to our experiments. The other matrices are initialized to small random values.

5.2 Details of gradient descent

The input-feature matrix $\mathbf{W}_f$ is learned by minimizing the cost function C of every load traces in the training set. At each time step, the prediction of the network $\mathbf{y}_t$ is compared to the real load $\mathbf{r}_t$,

$$C = \frac{1}{2}\sum_{t=1}^{n-1}(\mathbf{y}_t - \mathbf{r}_t)^{\mathrm{T}}(\mathbf{y}_t - \mathbf{r}_t) \tag{10}$$

To make the derivation of the learning procedure clearer, we briefly derive the matrix formulation of back-propagation through time. Considering the ESN with a load trace of length n ($\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)$), the equations describing the unrolled ESN are as follows,

$$\mathbf{f}_t = \mathbf{W}_f \mathbf{x}_t \tag{11}$$

$$\mathbf{a}_t = \mathbf{W}\mathbf{h}_{t-1} + \mathbf{W}_{\mathrm{in}}\mathbf{f}_t \tag{12}$$

$$\mathbf{h}_t = f(\mathbf{a}_t) \tag{13}$$

$$\mathbf{y}_t = \mathbf{W}_{\mathrm{out}}\mathbf{h}_t \tag{14}$$

According to gradient descent, each weight change in the network should be proportional to the negative gradient of the cost with respect to the specific weight we are interested in modifying.

$$\triangle W = -\eta \frac{\partial C}{\partial W} \tag{15}$$

The gradient of C with respect to the parameters of the autoencoder can be estimated as the following equations,

$$\frac{\partial C}{\partial \mathbf{W}_{f,t}} = \frac{\partial C}{\partial \mathbf{f}_t} \frac{\partial \mathbf{f}_t}{\partial \mathbf{W}_{f,t}} = \frac{\partial C}{\partial \mathbf{f}_t} \mathbf{x}_t \tag{16}$$

$$\frac{\partial C}{\partial \mathbf{f}_t} = \frac{\partial C}{\partial \mathbf{a}_t} \frac{\partial \mathbf{a}_t}{\partial \mathbf{f}_t} = \frac{\partial C}{\partial \mathbf{a}_t} \mathbf{W}_{\mathrm{in},t} \tag{17}$$

The gradient of C with respect to $\mathbf{f}_t$ can be rewritten in the matrix form:

$$\nabla_{\mathbf{f}_t} C = (\mathbf{W}_{\mathrm{in}})^{\mathrm{T}} \delta_t \tag{18}$$

where $\delta_t = \frac{\partial C}{\partial \mathbf{a}_t}$. The δ_t is usually called the error item at time t, and can be expressed using δ_{t+1}.

$$\begin{aligned}\delta_t &= \frac{\partial C}{\partial \mathbf{a}_t} \\ &= \frac{\partial C}{\partial \mathbf{a}_{t+1}} \frac{\partial \mathbf{a}_{t+1}}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{a}_t} \\ &= \frac{\partial C}{\partial \mathbf{a}_{t+1}} \mathbf{W} f^{'}(\mathbf{a}_t) \\ &= \delta_{t+1} \mathbf{W} f^{'}(\mathbf{a}_t) \end{aligned} \tag{19}$$

6 Performance evaluation

6.1 Dataset and parameters

To evaluate our approach, a dataset containing information of the host in Google data center [7] has been employed. The most studied cluster workloads have fallen into one of the three following categories [20]:

- long-running services: servers that require a certain amount of resources (usually CPU resource) to achieve acceptable performance and run indefinitely.
- DAG-of-task systems: MapReduce-like systems that run many independent short tasks that are assumed to CPU bound or I/O bound.
- high-performance (or throughput) computing: batch queuing systems that typically run CPU-bound programs can usually tolerate substantial wait times, and often require many machines simultaneously for a long period of time.

Each of these categories brings different challenges to resource management. Unlike the most previous clusters which have not been faced with the diverse workloads of multi-propose clouds, the Google cluster trace tends to capture a range of behaviors that includes all the above categories, among others. The Google cluster traced over 670,000 jobs and over 40 million task events across about 12,000 machines over 1-month period. The trace lacks precise information about the purpose of jobs and configuration of machines. The resource (RAM and CPU) information in the trace has already been normalized to [0, 1].

In our experiment, we only predict the CPU load. Because CPU load is a vivid aspect to exhibit the request and usage of resources. The fluctuation of CPU load can indicate the application execution process, while high CPU load will considerably slow down the host. And the prediction of CPU load is more challengeable comparing to the memory load, the fluctuation of CPU load is more drastic [5]. To apply our proposed method, we split Google's 29-day trace data into three sets. A training set which is used to update the feature weights and the output weights, a validating set which is used to prevent overfitting and a predicting set which is used to evaluate our proposed method.

To achieve best prediction performance, some important parameters should be chosen. The most important parameters concerned in feature extraction are the number of hidden units *hiddenSize* and the sparsity of the hidden units *sparsityParam*, which are estimated according to the reconstruction error. The optimal values of these parameters that computed using the tenfold cross-validation method. According to the experimental results in Fig. 5, the *hiddenSize* and *sparsityParam*, for the minimum reconstruction error, are 200 and 0.1, respectively.

The parameters of ESN are also selected by the cross-validation method. Input connection weights $\mathbf{W}_{\text{in}}$ are set to small random values sampled from a uniform distribution between $[-0.1, 0.1]$. The reservoir matrix $\mathbf{W}$ is rescaled to a spectral radius of 0.1, thus ensuring the echo state property. The size of the reservoir layer

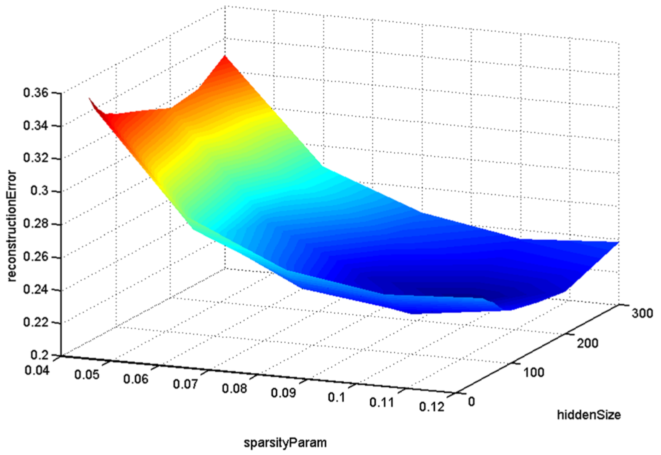


Fig. 5 The reconstruction error with different *hiddenSize* and *sparsityParam*. The axis *hiddenSize* represents the number of hidden units of autoencoder, while the axix *sparsityParm* represents the sparsity constraints of hidden units. The minimum reconstruct error is achieved when the number of hidden units is 200 and the sparsity constraints is 0.1

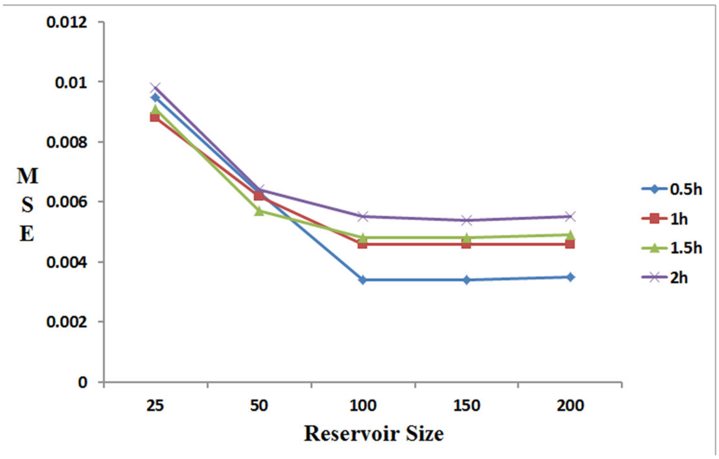


Fig. 6 The MSE as a function of the reservoir size. Different color of the *curves* indicates the different prediction length

is far important than in a traditional RNN at a similar level of performance. In our experiment, we chose the reservoir size by minimizing the MSE of the load traces in the validation set, where MSE is defined as Eq. (20). According to Fig. 6, we set the reservoir size to 100.

$$MSE = \frac{1}{T}\sum_{i=1}^{\mathrm{T}}(y_i - r_i)^2 \tag{20}$$

6.2 Performance evaluation

6.2.1 Methods for comparison

To show the effectiveness of our proposed method, we also rigorously and comprehensively implemented three other load prediction methods. The parameters of these baseline methods are chosen to achieve the best performance which are shown in Table 1. Some details are given as follows:

– Auto regressive method (AR): the classic AR method is performed according to Eq. (21), where $y(t)$, a_i, x_i and ε_t refer to the predicted value, the coefficient, the history value and the noise at time t, respectively.

$$y(t) = \sum_{i=1}^{n} a_i x_i + \varepsilon_t \tag{21}$$

In general, the AR method can only predict the load value for the next moment, while previous works extended it to long-term point prediction by applying the AR method recursively on the predicted values.

Table 1 Optimized parameters for the baseline methods

Methods	Key parameters	Values
AR	Order of AR	7
ANN	Input size	20
	Hidden size	10
	Learning rate	0.1
PSR+EA-GMDH	Maximum generation	60
	Population size	40
	Crossover rate	0.9
	Mutation rate	0.1
	Number of layers	4
	Number of neurons of each layer	9,6,3,1
	Number of inputs to be selected	2–4
	Polynomial type	1–3
Our method	Hidden size	200
	Sparsity constraint	0.1
	Input scaling	0.1
	Spectral radius	0.1
	Reservoir size	100
	Learning rate	0.01
	Momentum	0.9

- ANN [6] method: the ANN method uses the load in the history window as the input vector, and the load in the future interval as the output vector. After training the network, the model it learned will be used for host load prediction.
- Bayes [5] method: the aim of Bayes method is to predict the vector of the load values, denoted by $\mathbf{l} = (l_1, \ldots, l_n)^{\mathrm{T}}$, each of which represents the mean load value over a particular segment. According to Di [5], the Bayes method we applied is the best strategy using MMSE-BC with the single feature F_{ml} based on the evaluation type A.
- PSR+EA-GMDH method [27]: this method combines the PSR method and EA-GMDH method. The PSR is used to reconstruct the load trace from single-dimensional time series to a multi-dimensional phase space, and the time series after reconstruction are used as the input of the EA-GMDH network.

6.2.2 Experimental results

We evaluated the accuracy of the prediction result by the mean squared errors (MSE), defined as Eq. (20), where T is the length of the prediction step, y_i and r_i are the predicted and real values, respectively. To compare our proposed method with the Bayes method [5], we use our proposed method to predict the mean host load which is quantified using the mean segment squared error (MSSE) [5]. The MSSE is defined as follows:

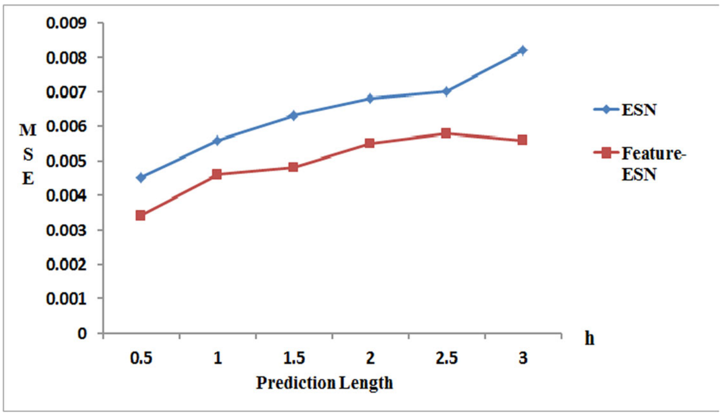


Fig. 7 Comparison of the conventional ESN and our proposed model

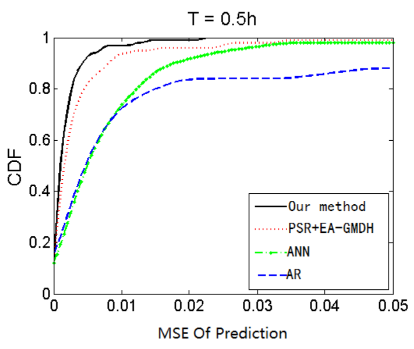
$$\mathrm{MSSE}(s) = \frac{1}{s}\sum_{i=1}^{n} s_i(l_i - L_i)^2 \tag{22}$$

where $s_1 = b$, $s_i = b \cdot 2^{i-2}$, $s = \sum_{i=1}^{n} s_i$, s_i is the separate segment; l_i and L_i are the predict value and real value, respectively; n is the total number of segments in the prediction interval. More details are presented in [5].

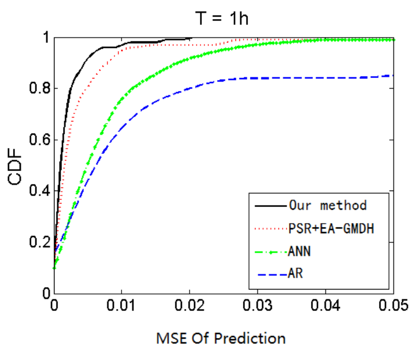
First, we compared our proposed method with the conventional ESN. According to Fig. 7, the addition of the feature layer improved the prediction accuracy. The introduction of the unsupervised feature learning layer before the recurrent layer leads to a novel organization of the reservoir activation pattern. The features can better capture the similarity between load traces, and therefore allow separating more sharply the activations corresponding to different load traces. In Fig. 7, it is also noted that although the prediction error increases with the prediction length increases, our proposed method can still get a satisfactory performance the prediction length is 3 h ahead (The interval of Google load trace is 5 min, so 3 h ahead is 36 steps ahead).

Figure 8 shows the comparison of our method and the methods we mentioned in Sect 6.2.1 in actual host load prediction. As the Bayes method can only predict the mean host load, we do not compare our method with Bayes method in this case. The x-axis in the Fig. 8 represents the value of MSE, while the y-axis represents cumulative distribution function (CDF) of MSE. The results show that our proposed method outperforms the others in terms of lower MSE. Specifically, it is obvious that our proposed method is better to predict different types of host traces, while the other methods, such as AR method, will have poor performance in some host traces.

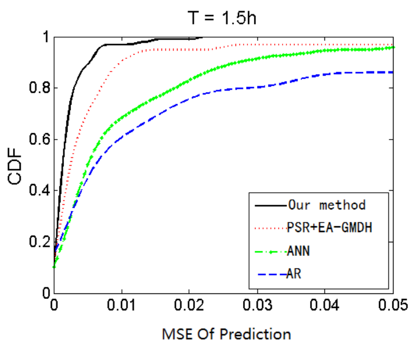
In Fig. 9, we compare our proposed method to other four methods in mean host load prediction. According to our statistics, our method improves the accuracy over the second best method (PSR+EA-GMDH) by 26.1 % in the 2.7 h ahead prediction, 22.4 % in 5.3 h, and about 32.3 % in 10.7 h, respectively.



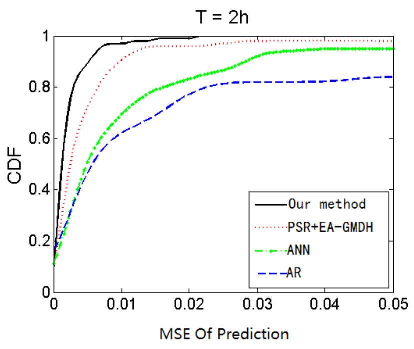
(a) CDF of MSE(T = 0.5h)



(b) CDF of MSE(T = 1h)



(c) CDF of MSE(T = 1.5h)



(d) CDF of MSE(T = 2h)

Fig. 8 MSE of host load prediction

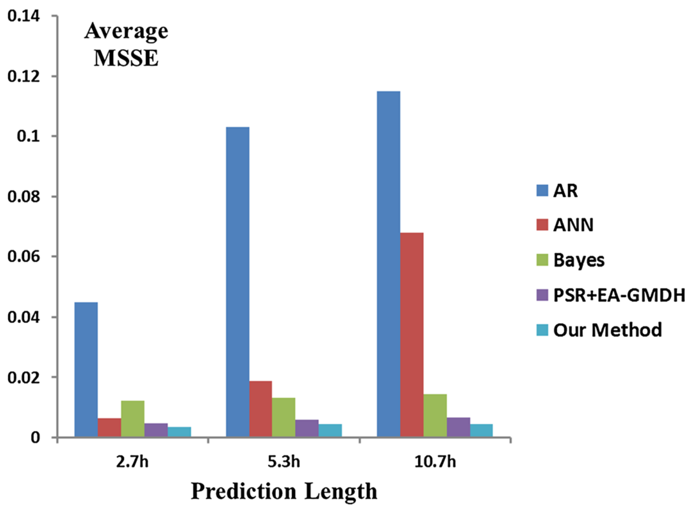


Fig. 9 The average of MSSE, where h represents hour, s_1 is 5 min

7 Conclusions and future work

In this paper, we proposed a novel method for CPU load prediction. In our proposed method, we use the ESN as our basic prediction model, considering that the ESN can be trained very fast and modeling one-dimensional time series impressively. To achieve a better representation of the inputs, we introduce a pre-recurrent feature layer which is an autoencoder neural network. The autoencoder can better capture the similarity between load traces. We evaluated our proposed method on one month Google load trace. Compared with some other state-of-arts methods, our proposed method could achieve higher accuracy. This implies that our work can be utilized to solve the schedule problem in the future work. However, the features learned by autoencoder is not easy to display when the input data is host load. We will try our best to visualize the features in the future work.

Acknowledgments This work was partially supported by Grant No. BE2011169, BK2011563 from the Natural Science Foundation of Jiangsu Province and Grant No. 61100111, 61300157, 61201425, 61271231 from the Natural Science Foundation of China.

References

1. Akioka S, Muraoka Y (2004) Extended forecast of cpu and network load on computational grid. In: IEEE international symposium on cluster computing and the grid, 2004. CCGrid 2004. IEEE, pp 765–772
2. Bengio Y, Simard P, Frasconi P (1994) Learning long-term dependencies with gradient descent is difficult. IEEE Trans Neural Netw 5(2):157–166
3. Bengio Y (2009) Learning deep architectures for ai. Found Trends® Mach Learn 2(1):1–127
4. Di S, Kondo D, Cirne W (2012) Characterization and comparison of cloud versus grid workloads. In: 2012 IEEE international conference on cluster computing (CLUSTER). IEEE, pp 230–238
5. Di S, Kondo D, Cirne W (2012) Host load prediction in a google compute cloud with a bayesian model. In: Proceedings of the international conference on high performance computing, networking, storage and analysis. IEEE Computer Society Press, p 21
6. Duy TVT, Sato Y, Inoguchi Y (2011) Improving accuracy of host load predictions on computational grids by artificial neural networks. Int J Parallel Emerg Distrib Syst 26(4):275–290
7. Google (2011) Google cluster data. Google reach blog. http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html. Accessed 14 Oct 2013
8. Guenter B, Jain N, Williams C (2011) Managing cost, performance, and reliability tradeoffs for energy-aware server provisioning. In: INFOCOM, 2011 Proceedings IEEE. IEEE, pp 1332–1340
9. Hinton GE, Osindero S, Teh YW (2006) A fast learning algorithm for deep belief nets. Neural Comput 18(7):1527–1554
10. Hinton G, Deng L, Yu D, Dahl GE, Mohamed Ar, Jaitl N, Senior A, Vanhoucke V, Nguyen P, Sainath TN et al (2012) Deep neural networks for acoustic modeling in speech recognition: the shared views of four research groups. IEEE Signal Process Mag 29(6):82–97
11. Hochreiter S (1991) Untersuchungen zu dynamischen neuronalen netzen. Master's thesis, Institut fur Informatik, Technische Universitat, Munchen
12. Jaeger H (2001) The echo state approach to analysing and training recurrent neural networks-with an erratum note. Bonn Ger Ger Natl Res Cent Inf Technol GMD Tech Rep 148:34
13. Jaeger H, Haas H (2004) Harnessing nonlinearity: predicting chaotic systems and saving energy in wireless communication. Science 304(5667):78–80
14. Kang M, Kang DI, Crago SP, Park GL, Lee J (2011) Design and development of a run-time monitor for multi-core architectures in cloud computing. Sensors 11(4):3595–3610
15. Kullback S, Leibler RA (1951) On information and sufficiency. Ann Math Stat 22:79–86
16. Lukoševičius M (2012) A practical guide to applying echo state networks. In: Neural networks: tricks of the trade. Springer, Berlin, pp 659–686

17. Maass W, Natschläger T, Markram H (2002) Real-time computing without stable states: a new framework for neural computation based on perturbations. Neural Comput 14(11):2531–2560
18. Mell P, Grance T (2009) The nist definition of cloud computing. Natl Inst Stand Technol 53(6):50
19. Osman S, Subhraveti D, Su G, Nieh J (2002) The design and implementation of zap: a system for migrating computing environments. ACM SIGOPS Opera Syst Rev 36(SI):361–376
20. Reiss C, Tumanov A, Ganger GR, Katz RH, Kozuch MA (2012) Heterogeneity and dynamicity of clouds at scale: google trace analysis. In: Proceedings of the third ACM symposium on cloud computing. ACM, p 7
21. Spratling MW (2006) Learning image components for object recognition. J Mach Learn Res 7:793–815
22. Sutskever I, Martens J, Dahl G, Hinton G (2013) On the importance of initialization and momentum in deep learning. In: Proceedings of the 30th international conference on machine learning (ICML-13), pp 1139–1147
23. Urgaonkar B, Shenoy P, Chandra A, Goyal P (2005) Dynamic provisioning of multi-tier internet applications. In: Second international conference on autonomic computing, 2005. ICAC 2005. Proceedings. IEEE, pp 217–228
24. Werbos PJ (1990) Backpropagation through time: what it does and how to do it. Proc IEEE 78(10):1550–1560
25. Wu Y, Yuan Y, Yang G, Zheng W (2007) Load prediction using hybrid model for computational grid. In: 2007 8th IEEE/ACM international conference on grid computing. IEEE, pp 235–242
26. Yang L, Foster I, Schopf JM (2003) Homeostatic and tendency-based cpu load predictions. In: International parallel and distributed processing symposium, 2003. Proceedings. IEEE, p 9
27. Yang Q, Peng C, Zhao H, Yu Y, Zhou Y, Wang Z, Du S (2014) A new method based on psr and ea-gmdh for host load prediction in cloud computing system. J Supercomput 68(3):1402–1417
28. Zhang Q, Cheng L, Boutaba R (2010) Cloud computing: state-of-the-art and research challenges. J Internet Serv Appl 1(1):7–18
29. Zhang Q, Zhani MF, Zhang S, Zhu Q, Boutaba R, Hellerstein JL (2012) Dynamic energy-aware capacity provisioning for cloud computing environments. In: Proceedings of the 9th international conference on autonomic computing. ACM, pp 145–154