2019

Electric Load Forecasting Using Long Short-term Memory Algorithm

Tianshu Yang

Electric Load Forecasting Using Long Short-term Memory Algorithm

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Philosophy at Virginia Commonwealth University

By

Tianshu Yang

M.S in Electrical Engineering

Advisor: Zhifang Wang, Ph.D.

Associate Professor, Department of Electrical and Computer Engineering

Virginia Commonwealth University

Richmond, Virginia

July 2019

Acknowledgment

I want to recognize Professor Zhifang Wang for her support and help throughout the study, which enabled me to understand the principles and operations of the entire model more thoroughly and to implement this project. Her support and help are that I better use software to model and optimize theoretical principles to improve and achieve satisfactory results and data.

I would also like to thank Dr. Motai and Dr. Thang Dinh, who agreed to be a member of this committee.

Table of Contents

List of Figures

List of Tables

Abstract

Power system load forecasting refers to the study or uses a mathematical method to process past and future loads systematically, taking into account important system operating characteristics, capacity expansion decisions, natural conditions, and social impacts, to meet specific accuracy requirements. Dependence of this, determine the load value at a specific moment in the future. Improving the level of load forecasting technology is conducive to the planned power management, which is conducive to rationally arranging the grid operation mode and unit maintenance plan, and is conducive to formulating reasonable power supply construction plans and facilitating power improvement, and improve the economic and social benefits of the system.

At present, there are many methods for load forecasting. The newer algorithms mainly include the neural network method, time series method, regression analysis method, support vector machine method, and fuzzy prediction method. However, most of them do not apply to long-term time-series predictions, and as a result, the prediction accuracy for long-term power grids does not perform well.

This thesis describes the design of an algorithm that is used to predict the load in a long time-series. Predict the load is significant and necessary for a dynamic electrical network. Improved the forecasting algorithm can save a ton of the cost of the load. In this paper, we propose a load forecasting model using long short-term memory(LSTM). The proposed implementation of LSTM match with the time-series dataset very well, which can improve the accuracy of convergence of the training process. We experiment with the difference time-step to expedites the convergence of the training process. It is found that all cases achieve significant different forecasting accuracy while forecasting the difference time-steps.

Keywords—Load forecasting, long short-term memory, micro-grid

1 Introduction

Electric load forecasting has become one of the required fields in power engineering. The desirable forecast method can achieve satisfactory forecasting accuracy without high computation cost. Load forecasting can maintain the safety and stability of the grid operation, reduce the unnecessary rotating reserve capacity, and rationally arrange the unit maintenance plan. To ensure the average production and life of the society, effectively reduce the cost of power generation, and improve economic and social benefits

The results of load forecasting can also help determine the installation of new generator sets in the future, determine the size, location and time of installed capacity, determine the capacity expansion and reconstruction of the power grid, and determine the construction and development of the power grid.

Therefore, the level of power load forecasting work has become one of the significant indicators to measure whether an electrical network system management is modernizing. Power load forecasting has become an important and arduous task to solve the problem of electricity management.

In this paper, we pay attention to solving the problem to deal with the long-time data forecasting in the electrical network. We try to find a new Neural network algorithm that can give a higher forecasting accuracy in a long time sequence.

1.1 application of load forecasting

Load forecasting (electric load forecasting, power demand forecasting). Although "load" is a vague term, in load forecasting, "load" usually means demand (in kW) or energy (in kWh), and because the power and energy of the hourly data are the same, Therefore, it is usually not necessary to distinguish between demand and energy. Load forecasting involves accurate prediction of size and geographic location at different times during the planning period. The necessary amount of interest is usually the total system (or area) load per hour. However, load forecasting also involves hourly, daily, weekly, and monthly load and peak load forecasts.

Load forecasting is a technique used by electricity or energy supply companies to predict the power/energy needed to satisfy demand and supply equilibrium. The accuracy of the

forecast is essential to the operational and management of the load in a utility electrical network.

Load forecasting has long been considered the initial building block for all utility planning efforts, and it will decide the expected future energy sales and peak demand. The benefits of reasonable and accurate load forecasts are not only manifested in the economic field but also bring positive benefits to the development of the whole society. Therefore, the method of load forecasting is becoming more and more popular, and new methods are continually being put into use and continuously be improved.

1.2 Basic method of load forecasting

Due to various factors such as weather changes, social activities and festival types, the load appears as a non-stationary random process in time series, but most of the factors affecting system load have regularity, to achieve effective prediction. Foundation. In order to predict the load better, people use many methods to increase the accuracy of the short-term load. The newer algorithms mainly include the neural network method, time series method, regression analysis method, support vector machine method, and fuzzy prediction method[2]. The core problem of power load forecasting research is how to use existing historical data to establish a predictive model to predict the load value in future time or period. Therefore, the reliability of historical data information and prediction model are the central factors that affect load forecasting accuracy.

1.2.1 Neural network method

The neural network method is currently the most advanced load forecasting method. The application of neural network method in load forecasting is mainly divided into an artificial neural network (ANN) and recurrent neural network (also known as a recurrent neural network, referred to as RNN). Among them, RNN is an algorithm that works better than ANN. LSTM is an advanced neural network algorithm, and we will discuss its technical details in the following sections.[2]

1.2.2 Time series method

The historical data of the power load is an ordered set of samples and records at regular intervals is a time series. The time series method is based on historical data of load by using

a mathematical model describing the change of electrical load. Based on the model, establish the expression of the load forecast, and predict the future load.

The model has high requirements for the stationarity of the original time series, and is only suitable for short-term predictions with uniform load changes; If this model meets instability factors such as weather, holidays, the predicted result will have a huge error.

1.2.3 Regression analysis

The regression analysis prediction method is based on the change rule of historical data and the factors affecting the load change, find the correlation between the independent variable and the dependent variable; get its regression equation, determine the model parameters; and predict the load value at the later time.

The shortcomings are that the historical data is relatively high, and the linear method is used to describe the complicated problems. The structural form is too pure, and the precision is not excellent. The model cannot describe various factors affecting the load in detail. The model's initialization is complicated and needs abundant Experience and skills.

1.3 motivation of the LSTM

The emergence of RNNs is mainly because they can link the previous information to the present and solve the current problems. For example, using the previous screen can help us understand the content of the current screen. If RNNs can do this, then it is helpful for our mission.

Sometimes, when we are dealing with the current task, we only need to look at some of the more recent information. When need to predict a sentence word while it only has tens of letter, the RNN will be okay. However, while the number of the letter increasing to hundreds or thousands, the predicted error will increase fast. As the interval between prediction information and related information increases, it is difficult for RNNs to associate them.

In theory, RNNs can link such long-term dependencies ("long-term dependencies") and solve such problems by choosing the right parameters. However, unfortunately, in practice, RNNs can't solve this problem.

However, fortunately, LSTMs can help us solve this problem. The LSTM algorithm is a neural network algorithm used to predict long-term time series. It can significantly improve

the gradient explosion and gradient disappearance of neural network algorithms in the face of long time series data.[1]

1.4 challenges of the LSTM

A. hardware acceleration

A big problem with LSTM is that training them requires very high hardware. Also, it still requires many resources when we do not need to train these networks quickly. Running these models in the same cloud also requires a lot of resources. Training the nightmare of LSTM, LSTM training is difficult because they require storage bandwidth binding calculations, which is the worst nightmare for hardware designers and ultimately limits the applicability of neural network solutions. In short, LSTM requires four linear layers per unit to run in each sequence time step. Linear layers require a large amount of storage bandwidth to calculate. They cannot use many computing units, usually because the system does not have enough storage bandwidth to satisfy the computing unit. Moreover, it is easy to add more compute units, but it is hard to add more storage bandwidth (note that there are enough wires on the chip, from the processor to the long wires stored). Therefore, LSTM and its variants are not a good match for hardware acceleration.

B. Vanishing gradients

LSTM can reduce the problem of gradient disappearance (explosion) to a certain extent, but in essence, it's training still follows a sequential path from the past unit to the current unit. Moreover, this path is now more complicated, adding a lot of additional components. This path can now remember historical information for an extended time-series, but it is not infinite. As the number of time series increases, the degree of retention of historical information will decrease. Therefore LSTM is generally applied to sequences of 100 orders of magnitude instead of 1000 orders of magnitude, or longer sequences.

1.5 contributions

With the develop research of the LSTM algorithm, the application of LSTM has been continuously promoted, but the implementation of LSTM in long-term prediction of the power grid is rare. People still prefer to use the traditional RNN algorithm for prediction. Therefore, this paper focuses on the application of LSTM in the field of load forecasting

and compares the prediction efficiency and accuracy of the algorithm under different time series. In this process, we can find a more optimal prediction model.

This thesis describes the design of an algorithm that is used to predict the load in a long time-series. Predict the load is significant and necessary for a dynamic electrical network. Improved the forecasting algorithm can save a ton of the cost of the load. In this paper, we propose a load forecasting model using long short-term memory(LSTM). The proposed implementation of LSTM match with the time-series dataset very well, which can improve the accuracy of convergence of the training process. We experiment with the difference time-step to expedites the convergence of the training process. It is found that all cases achieve significant different forecasting accuracy while forecasting the difference time-steps.

2 Deep learning

There is a method called machine learning in the field of artificial intelligence. In the machine learning method, there is a class of algorithms called neural networks. The neural network is shown below[14]:

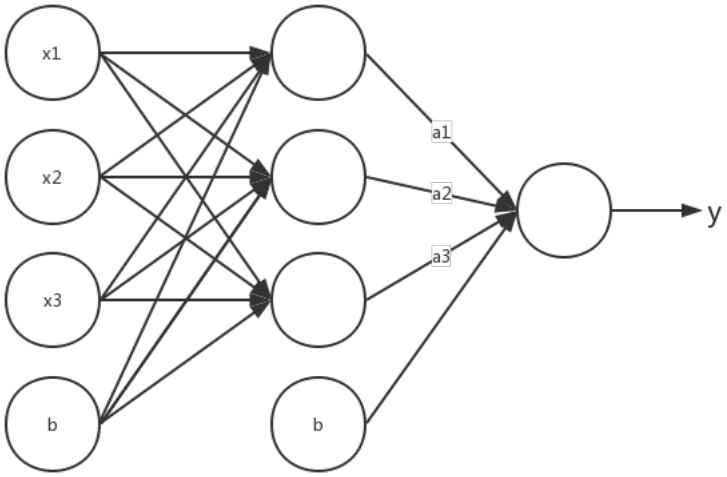

Figure1. structure of the neural network

Each circle in the above picture is a neuron, and each line represents the connection between neurons. We can see that the above neurons are divided into multiple layers, the neurons between the layers are connected, and the neurons in the layers are not connected. The leftmost layer is called the input layer. This layer is responsible for receiving input data. The rightmost layer is called the output layer. We can get the neural network output data from this layer. The layer between the input layer and the output layer is called a hidden layer.

A neural network with more hidden layers (greater than 2) is called a deep neural network. deep learning is one of the branch which use machine learning method in deep architectures(such as deep neural networks).

So what are the advantages of deep networks compared to shallow networks? Merely speaking, deep networks can express more power. An only one hidden layer neural network can fit any function, but it requires many neurons. Deep networks can fit the same function with much fewer neurons. That is to fit a function, either use a shallow and wide network or use a deep and narrow network, the latter tends to be more resource-efficient.

2.1 Perceptron

The figure below is a perceptron[3]:

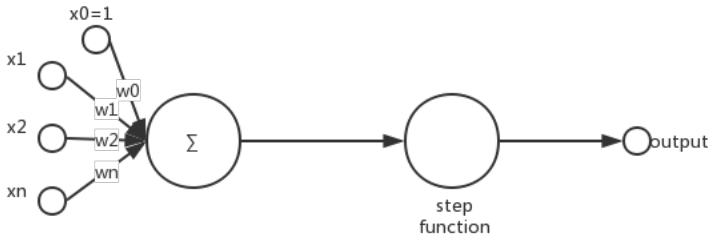

Figure 2. structure of perceptron

In the figure, a perceptron has the following components:

1.Input Weights: A perceptron can receive multiple inputs x_n, each with a weight ω_n, and an offset term b, as shown in the figure above ω_0.

2.Activation function

3.Output: the output of perceptron is calculated by the following formula

$$y = f(\omega \cdot x + b)$$

2.2 Linear unit

There is a problem with perceptrons. When the datasets that are facing are not linearly separable, the "perceptron rules" may not converge, which means that we can never complete a perceptron training. To solve this problem, we use a conductive linear function instead of the step function of the perceptron. This perceptron is called a linear unit. Linear units converge to an optimal approximation when faced with a linearly inseparable data set. The linear unit is shown below

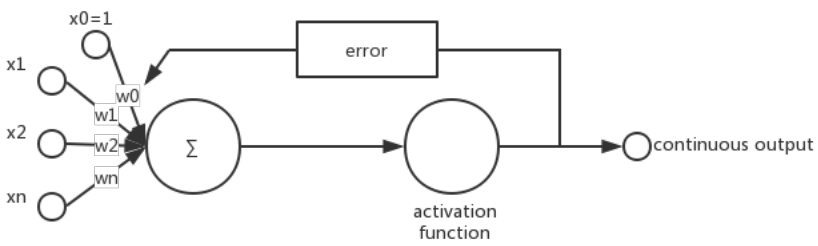

Figure 3. structure of the linear unit

After replacing the activation function, the linear unit will return a real value instead of a 0,1 classification. Therefore linear units are used to solve regression problems rather than classification problems.

2.3 Supervised learning and unsupervised learning

Machine learning has a learning method called supervised learning[4]. It means that to train a model, we need to provide a bunch of training samples: each training sample includes both the input feature x and the corresponding output y. In other words, we have to find a lot of people, we know their characteristics (working years, industry...) and know their income. We use this sample to train the model so that the model sees each of our questions (input feature x) and the answer to the corresponding question (true value y^*). When the model sees enough samples, it can summarize some of these rules. Then, you can predict the answer to the input that it has not seen.

Another type of learning method is called unsupervised learning[4]. The training sample of this method has only x and no y. The model can summarize some rules of the feature x, but can't know its corresponding answer y^*.

Many times, training samples with both x and y are rare, and most samples only have x. For example, in speech-to-text (STT) recognition tasks, x is speech and y^* is the text corresponding to this speech. It's easy to get a lot of voice recordings, but it's very laborious to split the voice into sections and mark the corresponding text. In this case, to compensate for the shortage of labeled samples, we can use the unsupervised learning method to do some clustering first, let the model summarize which syllables are similar, and then use a small number of labeled training samples to tell the model. Some syllables correspond to the text. In this way, the model can match similar syllables to the corresponding text and complete the training of the model.

2.4 The objective function of a linear unit

Under supervised learning, for a sample, we know its characteristic x and the true value y^*. At the same time, we can also calculate the output y based on the model h(x). Note that we use y^* to indicate the true value in the training sample, which is the actual value; the y indicates the predicted value calculated by the model. We certainly hope that the y and y^* calculated by the model are as close as possible.

There are many ways to express the closeness of y and y^* in mathematics. For example, we can use the square of the difference between y and y^* to indicate their closeness.

$$e = \frac{1}{2}(y^* - y)^2 \tag{1}$$

We call e the error of a single sample. It is convenient for later calculation multiplied by 1/2.

If there are N samples in the training data, We can use the sum of the errors of all the samples in the training data to represent the error E of the model, that is,

$$E = e^1 + e^2 + e^3 + \cdots + e^n \tag{2}$$

e^1 of the above formula represents the error of the first sample, and e^2 represents the error of the second sample…

For convenience, we can also write the above formula as a summation formula.

$$E = e^1 + e^2 + e^3 + \cdots + e^n \tag{3}$$

$$= \sum_{i=1}^{n} e^i \tag{4}$$

$$= \frac{1}{2}\sum_{i=1}^{n}\left(y^{*(i)} - y^{(i)}\right)^2 \tag{5}$$

Among them,

$$y^{(i)} = h\left(x^{(i)}\right) \tag{6}$$

$$= \mathrm{w}^T \mathrm{x}^{(\mathrm{i})} \tag{7}$$

$\mathrm{x}^{(\mathrm{i})}$ represents the feature of the i-th training sample, $y^{*(i)}$ represents the true value of the i-th sample, and we can also use the tuple ($\mathrm{x}^{(\mathrm{i})}$, $y^{*(i)}$) to represent the i-th training sample. $y^{(i)}$ is the predicted value of the model for the i-th sample.

For a training data set, the error needs to be as small as possible, which means the value in the formula (5) need to be small. For a particular training data set, the values of ($\mathrm{x}^{(\mathrm{i})}$, $y^{*(i)}$) are known, so formula (5) is a function of the parameters.

$$E(\mathrm{w}) = \frac{1}{2}\sum_{i=1}^{n}\left(y^{*(i)} - y^{(i)}\right)^2 \tag{8}$$

$$= \frac{1}{2}\sum_{i=1}^{n}\left(y^{*(i)} - \mathrm{w}^{T}\mathrm{x}^{(\mathrm{i})}\right)^{2} \tag{9}$$

The training of the model is actually to obtain the appropriate w, so that (Formula 5) obtains the minimum value which is called the optimization problem in mathematics, and $E(\mathrm{w})$ is the goal of our optimization, called the objective function.

2.5 Gradient descent optimization algorithm

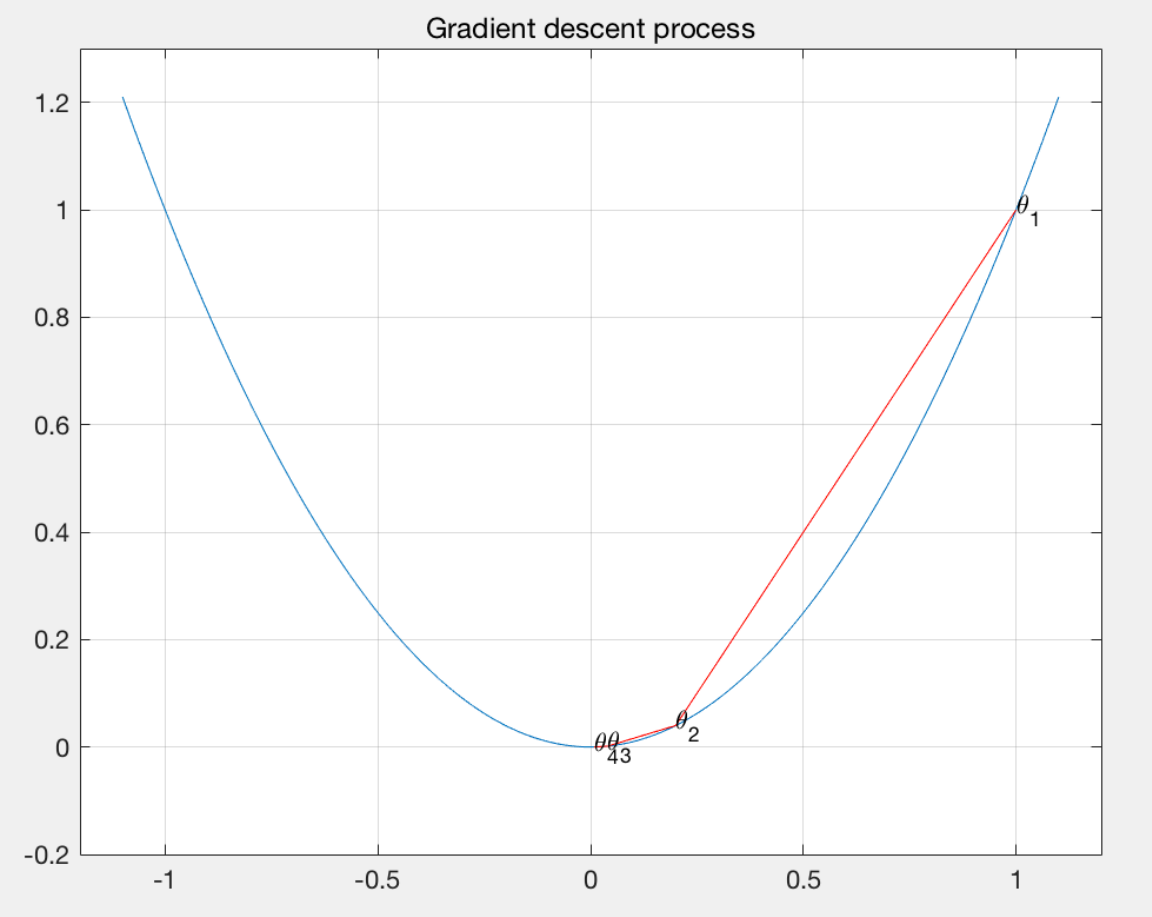

Figure 4. optimization of gradient descent

First of all, we choose a point to start, such as the point above[5]. Next, each iteration is modified to reach the function minimum point after several iterations finally. How can we move in the direction of the minimum value of the function, we need to modify the opposite direction of the gradient of the function y=f(x). The gradient is a vector that points to the fastest rising direction of the function value. The opposite direction of the gradient is, of course, the direction in which the function value drops the fastest. Each time we modify

the value of x along the opposite direction of the gradient, we can, of course, go near the minimum of the function(3).

According to the above discussion, we can write the formula of the gradient descent algorithm.

$$\mathrm{w}' = \mathrm{w} - \eta \nabla f(x) \tag{10}$$

From the formula (5)

$$E(\mathrm{w}) = \frac{1}{2}\sum_{i=1}^{n}\left(y^{*(i)} - y^{(i)}\right)^2 \tag{11}$$

if we need the maximum value of the objective function, we should use the gradient ascent algorithm, and its parameter modification rule is

$$\mathrm{w}' = \mathrm{w} + \eta \nabla f(x) \tag{12}$$

The gradient of the objective function is

$$\nabla E(\mathrm{w}) = -\sum_{i=1}^{n}\left(y^{*(i)} - y^{(i)}\right)\mathrm{x}^i \tag{13}$$

Therefore, the parameter modification rule of the linear unit is finally like this

$$\mathrm{w}' = \mathrm{w} + \eta\sum_{i=1}^{n}\left(y^{*(i)} - y^{(i)}\right)\mathrm{x}^i \tag{14}$$

2.5.1 Derivation of ∇E(w)

We know that the definition of the gradient of a function is it is partial derivative relative to each variable[5], so we write the following formula

$$\nabla E(\mathrm{w}) = \frac{\partial}{\partial \mathrm{w}} E(\mathrm{w}) \tag{15}$$

$$= \frac{\partial}{\partial \mathrm{w}}\frac{1}{2}\sum_{i=1}^{n}\left(y^{*(i)} - y^{(i)}\right)^2 \tag{16}$$

We know that the derivative of a sum is equal to the sum of the derivatives, so we can first find the derivatives in the summation symbol Σ, and then add them together

$$\frac{\partial}{\partial \mathrm{w}}\frac{1}{2}\sum_{i=1}^{n}\left(y^{*(i)} - y^{(i)}\right)^2 \tag{17}$$

$$= \frac{1}{2}\sum_{i=1}^{n}\frac{\partial}{\partial \mathrm{w}}\left(y^{*(i)} - y^{(i)}\right)^2 \tag{18}$$

Focus on the derivative inside

$$\frac{\partial}{\partial \mathrm{w}}\left(y^{*(i)}-y^{(i)}\right)^2 \tag{19}$$

$$=\frac{\partial}{\partial \mathrm{w}}\left(y^{*(i)2}-2y^{(i)}y^{*(i)}+y^{(i)2}\right)$$

We know that y^* is a constant that is independent of w, and $y=\mathrm{w}^T\mathrm{x}$. Let us follow the chain-based derivation rule

$$\frac{\partial E(\mathrm{w})}{\partial \mathrm{w}}=\frac{\partial E(y)}{\partial y}\frac{\partial y}{\partial \mathrm{w}} \tag{20}$$

Calculate the two partial derivatives to the right of the equal sign in the above equation

$$\frac{\partial E(\mathrm{w})}{\partial \mathrm{w}}=\frac{\partial}{\partial y}\left(y^{*(i)2}-2y^{(i)}y^{*(i)}+y^{(i)2}\right) \tag{21}$$

$$=-2y^{*(i)}+2y^{(i)} \tag{22}$$

$$\frac{\partial y}{\partial \mathrm{w}}=\frac{\partial}{\partial \mathrm{w}}\mathrm{w}^T\mathrm{x} \tag{23}$$

$$=\mathrm{x}$$

Substituting the result, we find that the partial derivative in Σ is

$$\frac{\partial}{\partial \mathrm{w}}\left(y^{*(i)}-y^{(i)}\right)^2 \tag{24}$$

$$=2\left(-y^{*(i)}+y^{(i)}\right)\mathrm{x} \tag{25}$$

Finally, calculate $\nabla E(\mathrm{w})$

$$\nabla E(\mathrm{w})=\frac{1}{2}\sum_{i=1}^{n}\frac{\partial}{\partial \mathrm{w}}\left(y^{(i)}-y^{*(i)}\right)^2 \tag{26}$$

$$=\frac{1}{2}\sum_{i=1}^{n}2\left(-y^{*(i)}+y^{(i)}\right)\mathrm{x} \tag{27}$$

$$=-\sum_{i=1}^{n}\left(y^{(i)}-y^{*(i)}\right)\mathrm{x} \tag{28}$$

2.6 Stochastic Gradient Descent, SGD

If we train the model according to formula 14, then we update the iterations of w every time, we have to traverse all the samples in the training data for calculation. We call this algorithm called Batch Gradient Descent. If our sample is extensive, such as millions to

hundreds of millions, then the amount of calculation is enormous. Therefore, a practical algorithm is the SGD algorithm[6]. In the SGD algorithm, only calculate one sample each time with the iteration of updated w. In this way, for training data with millions of samples, a traversal will update millions of times, and the efficiency will be significantly improved. Due to the noise and randomness of the sample, each update w does not necessarily follow the direction of decreasing E. However, although there is some randomness, a large number of updates generally progress in the direction of E reduction, and therefore can finally converge to near the minimum. The figure below shows the difference between SGD and BGD.

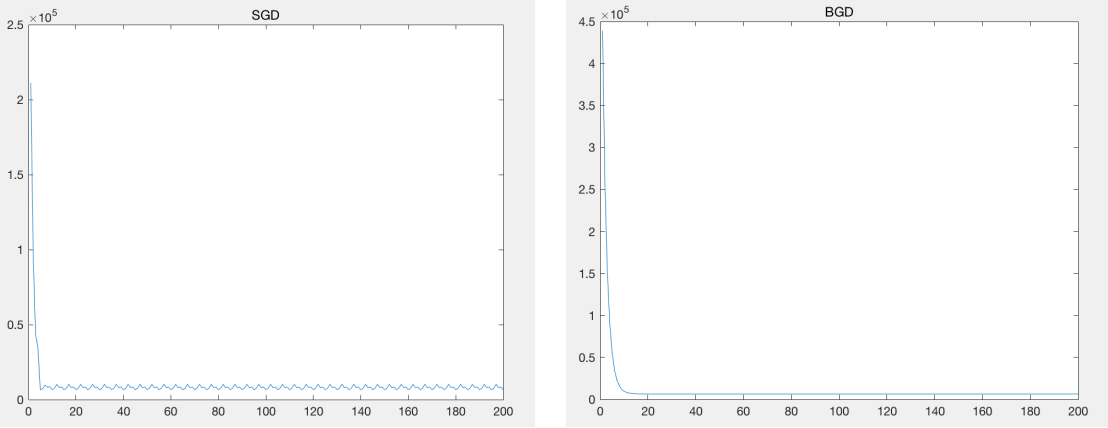

Figure 5. difference between SGD and BGD.

As shown above, ellipse represents the contour of the function value, and the center of the ellipse is the minimum point of the function. Red is the approximation curve of BGD, and purple is the approximation curve of SGD. We can see that BGD has been moving towards the lowest point, and SGD has obviously moved a lot, but overall, it is still approaching the lowest point.

We can see that SGD is not only efficient, but randomness is sometimes a good thing. The objective function in the figure is a "convex function" that finds a globally unique minimum along the opposite direction of the gradient. However, for non-convex functions, there are many local minima. Randomness helps us escape some very bad local minima to get a better model.

3 Artificial Neural Network

Artificial Neural Network (ANN), referred to as Neural Network (NN) or neural network in the field of machine learning and cognitive science, is a mimicking biological neural network. In particular, mathematical models or computational models of the structure and function of the brain used to estimate or approximate functions. The neural network is calculated by a large number of artificial neuronal connections. In most cases, artificial neural networks can change the internal structure based on external information. It is an adaptive system. In general, it has a learning function. New neural networks are a kind of non-linear statistical data modeling tools.

An artificial neural network (ANN) or a connected system is a computational system that is ambiguously inspired by the biological neural network that makes up the animal's brain. Neural networks are a combination of many different machine learning algorithms that handle advanced complex data inputs rather than a single algorithm. This kind of system goes through the sample to complete the task of learning, usually without any specific task rules. For example, in image recognition, they may learn to identify images containing cats as "cats" or "no cats" by analyzing manual sample images and use the advanced model to identify the target in the test images, they do this without any basic knowledge about the cat; for example, they have fur, tail, beard, and cat-like faces. Instead, they automatically generate recognition features from the learning materials they process. The structure of an original ANN is showed below.

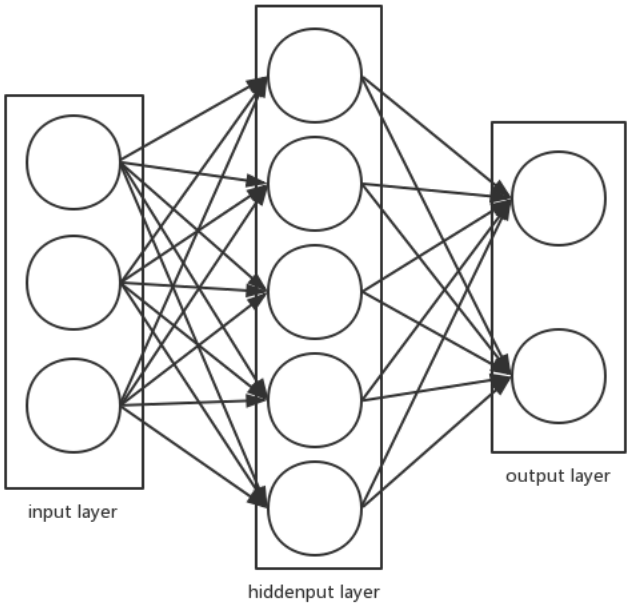

Figure 6. full connected Neural Networks

A neural network is a plurality of neurons connected according to specific rules. The above figure shows a fully connected (FC) neural network. By observing the above diagram, we can find that its rules include:

1. Neurons are always layered and generally consist of three layers in the neural network. The input layer is the first layer and is responsible for receiving input data. The last layer is called the output layer. We can get the neural network output data from this layer. The layers after the input layer and before output layer are called hidden layers because they are invisible to the outside.

2. Each neuron is independent of each other in the same layer。

3. Each neuron in the Nth layer is connected to all neurons in the N-1th layer (this is the meaning of full connected), and the output of the N-1th layer of neurons is the input to the Nth layer of neurons.

4. Each connection has a weight.

The above rules define the structure of a fully connected neural network. There are many other structural neural networks, such as Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN), which all have different connection rules.

3.1 Neurons

Neurons and perceptrons are mostly the same, except that when we say the perceptron, its activation function is a step function; when we say neurons, the activation function is often chosen to be a sigmoid function or a tanh function. As shown below:

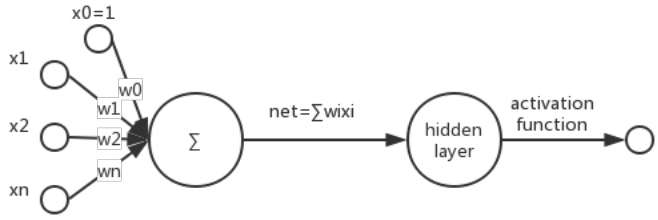

Figure 7. structure of a neuron

The method of calculating the output of a neuron is the same as calculating the output of a perceptron. Suppose the input of the neuron is vector $\bar{x}$, the weight vector is $\bar{w}$ (the bias term is $\bar{w}_0$), and the activation function is sigmoid function[8], then its output y:

$$y = sigmoid(\vec{\omega}^T \cdot \vec{x}) \tag{29}$$

The definition of the sigmoid function is below

$$sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{30}$$

Bring it into the previous formula and get

$$y = \frac{1}{1 + e^{-\vec{\omega}^T \cdot \vec{x}}} \tag{31}$$

The sigmoid function is a nonlinear function with a value range of (0,1). The function image is shown below

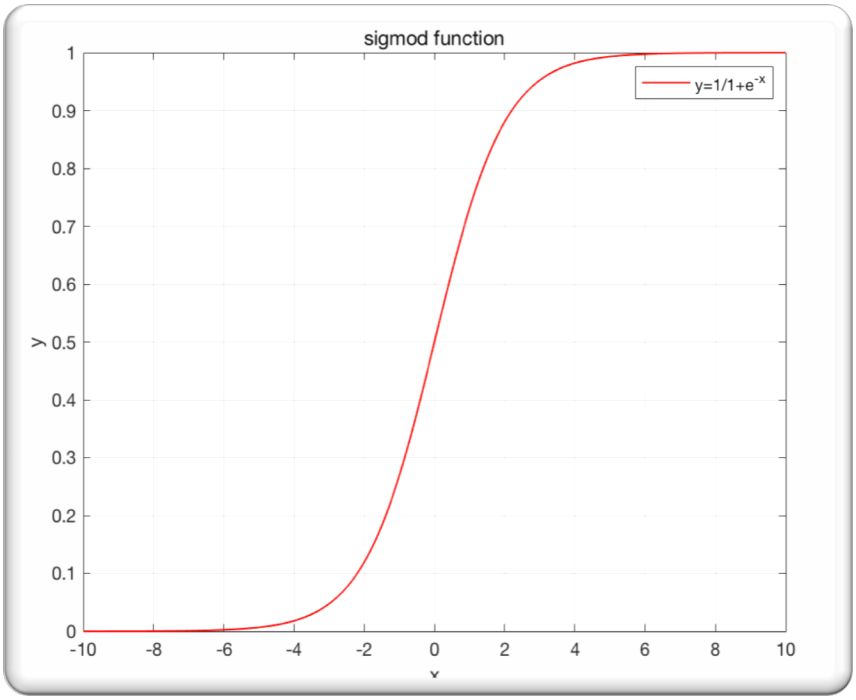

Figure 8. sigmoid function

The derivative of the sigmoid function

Let

$$y = sigmoid(x) \tag{32}$$

Then

$$y' = y(1 - y) = sigmoid(x) \tag{33}$$

As we can see, the derivative of the sigmoid function is special; the sigmoid function itself can represent it. Thus, once calculate the value of the sigmoid function, it is convenient to calculate the value of its derivative.

3.2 Calculate the output of the neural network

A neural network is a function of an input vector $\vec{x}$ to an output vector $\vec{y}$,

$$\vec{y} = f_{network}(\vec{x}) \tag{34}$$

To calculate the output of the neural network according to the input, it is necessary to first assign the value of each element $\vec{x}_i$ of the input vector $\vec{x}$ to the corresponding neuron of the input layer of the neural network, and then calculate each neuron of each layer in turn according to formula 1. The calculate process finished until the values of all neurons in the

last layer of the output layer are calculated. Finally, the output vector $\vec{y}$ is obtained by stringing together the values of each neuron in the output layer.

Let's take an example to illustrate this process. First number each cell of the neural network.

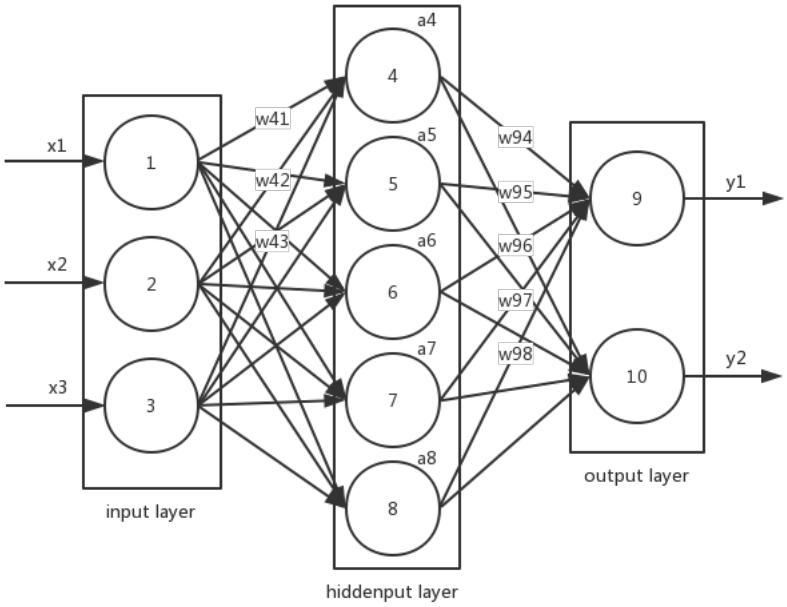

Figure 9 neural network

As shown in the figure above, the input layer has three nodes, which we numbered as 1, 2, and 3 in sequence; the four nodes in the hidden layer are numbered 4, 5, 6, and 7 in sequence; the last two nodes in the output layer are number 8 and 9. Because our neural network is fully connected, we can see that each node is connected to all nodes in the upper layer. For example, we can see node 4 of the hidden layer, which is connected to the three nodes 1, 2, and 3 of the input layer, and the weights on the connection are ω_{41} ω_{42}, ω_{43} In order to calculate the output value of node 4, we must first obtain the output values of all its upstream nodes (which are nodes 1, 2, 3). Nodes 1, 2, and 3 are nodes of the input layer, so their output value is the input vector $\vec{x}$ itself. According to the corresponding relationship in the above picture, the output values of nodes 1, 2, and 3 are x_1, x_2 and x_3 respectively. We require that the dimensions of the input vector be the same as the number of input layer neurons, and which input node of the input vector corresponds to which input node is freely determinable.

Once we have the output values of nodes 1, 2, and 3, we can calculate the output value a_4 of node 4 according to formula 29:

$$a_4 = sigmoid(\vec{\omega}^T \cdot \vec{x})$$
$$= sigmoid(\omega_{41} \cdot x_1 + \omega_{42} \cdot x_2 + \omega_{43} \cdot x_3 + \omega_{4b})$$

The above formula ω_{4b} is the offset term of node 4, which is not shown in the figure. ω_{41}, ω_{42}, and ω_{43} are the weights of the nodes 1, 2, and 3 to node 4, respectively. When number the weight ω_{ji}, we put the number j of the target node in front of the number i of the source node.

Similarly, we can continue to calculate the output values a_5, a_6, a_7, a_8 of nodes 5, 6, 7 and 8. Thus, the output values of the four nodes of the hidden layer are calculated, and we can then calculate the output value y_1 of node 9 of the output layer:

$$y_1 = sigmoid(\vec{\omega}^T \cdot \vec{a})$$
$$= sigmoid(\omega_{94} \cdot a_4 + \omega_{95} \cdot a_5 + \omega_{96} \cdot a_6 + \omega_{96} \cdot a_6 + \omega_{97} \cdot a_7 + \omega_{98} \cdot a_8 + \omega_{8b})$$

Similarly, we can also calculate the value of y_2. Calculate the output values of all nodes in the output layer, and we get the output vector $\vec{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$ of the neural network when the vector $\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$ is input. We can also see that the output vector has the same number of dimensions as the output layer neurons.

3.3 Back Propagation

We are using supervised learning as an example to explain the backpropagation algorithm[7,9]. We assume that each training sample is $(\vec{x}, \vec{t})$, where vector $\vec{x}$ is the characteristic of the training sample, and $\vec{t}$ is the target value of the sample.

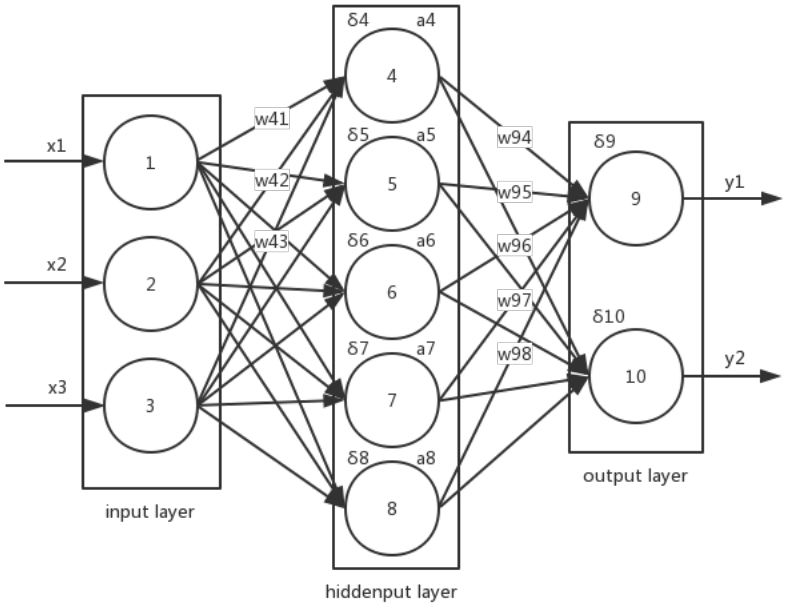

Figure 10. neural network

First, based on the algorithm introduced in the previous section, we use the feature $\vec{x}$ of the sample to calculate the output a_i of each hidden layer node in the neural network and the output y_i of each node in the output layer.[9]

Then, we calculate the error term δ_i of each node as below:

For output layer node i,

$$\delta_i = y_i(1 - y_i)(t_i - y_i) \tag{35}$$

Where δ_i is the error term of node i, y_i is the output value of node i, and t_i is the target value of the sample corresponding to node i. For example, according to the above figure, for the output layer node 9, its output value is y_1, and the target value of the sample is t_1, use the equation into the above gives the error term δ_9 of the node 9 should be:

$$\delta_9 = y_1(1 - y_1)(t_1 - y_1)$$

For hidden layer nodes[9],

$$\delta_i = a_i(1 - a_i) \sum_{k \in outputs} \omega_{ki}\, \delta_k \tag{36}$$

Where a_i is the output value of node i, ω_{ki} is the weight of the connection of the node i to its next layer node k, and δ_k is the error term of the node of the next layer of node k. For example, for hidden layer node 4, the calculation method is below:

$$\delta_4 = a_4(1 - a_4)(\omega_{94}\delta_9 + \omega_{10,4}\delta_{10})$$

Finally, update the weights on each connection:

$$\omega_{ji} \leftarrow \omega_{ji} + \eta\delta_j x_{ji} \tag{37}$$

Where ω_{ji} is the weight of node i to node j, η is a constant that becomes the learning rate, δ_j is the error term of node j, and x_{ji} is the input that node i passes to node j.

Obviously, to calculate the error term of a node, we need to calculate the error term of each node that connects to this node. This requires that the order of the error terms must be calculated from the output layer, and then calculate the error terms of each hidden layer reversely, until the hidden layer connected to the input layer. This is the reason why we called this algorithm of the backpropagation algorithm. When the error terms of all nodes are calculated, we can update all the weights according to Equation 37.

3.4 Derivation of the backpropagation algorithm

The backpropagation algorithm is the application of the chained derivation rule[9]. Next, we use the chain derivation rule to derive the backpropagation algorithm, which is the formula 35, 36, and 37 in the previous section.

According to the general routine of machine learning, we first determine the objective function of the neural network and then use the stochastic gradient descent optimization algorithm to find the parameter value at the minimum of the objective function.

We take the sum of the squared errors of all the output layer nodes of the network as the objective function:

$$E_d = \frac{1}{2}\sum_{i \in outputs}(t_i - y_i)^2 \tag{38}$$

Where E_d represents the error of sample d.

We use the stochastic gradient descent algorithm optimizes the objective function:

$$\omega_{ji} \leftarrow \omega_{ji} - \eta\frac{\partial E_d}{\partial \omega_{ji}} \tag{39}$$

The stochastic gradient descent algorithm also needs to find the partial derivative of the error E_d for each weight ω_{ji}

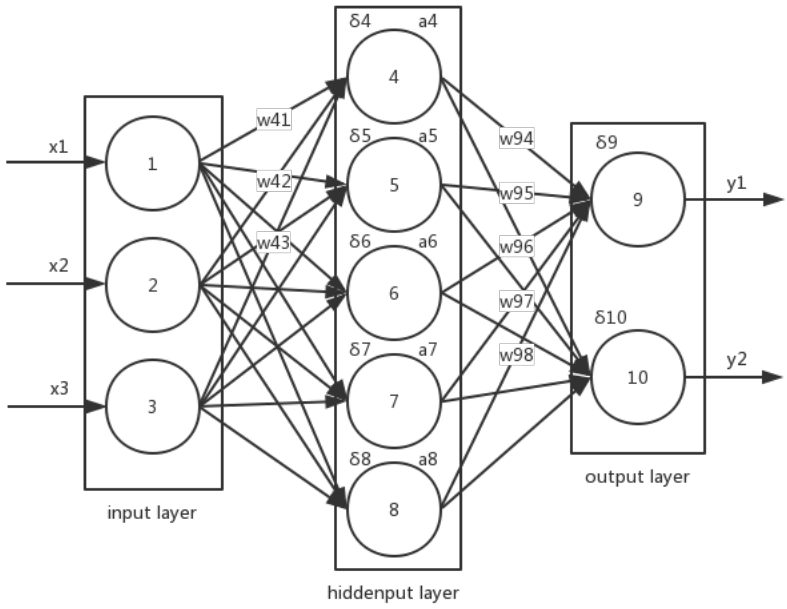

Figure 11. neural network

From the above figure, we find that the weight ω_{ji} can only affect other parts of the network by affecting the input value of the node j. Let net_j be the weighted input of the node j

$$net_j = \vec{\omega}_j \cdot \vec{x}_j \tag{40}$$

$$= \sum_i \omega_{ji} x_{ji} \tag{41}$$

E_d is a function of net_j, and net_j is a function of ω_{ji}. According to the chain derivation rule, we can get:

$$\frac{\partial E_d}{\partial \omega_{ji}} = \frac{\partial E_d}{\partial net_j} \frac{\partial E_d}{\partial net_j} \tag{42}$$

$$= \frac{\partial E_d}{\partial net_j} \frac{\partial \sum_i \omega_{ji} x_{ji}}{\partial net_j} \tag{43}$$

$$= \frac{\partial E_d}{\partial net_j} x_{ji} \tag{44}$$

In the above formula, x_{ji} is the input value that node i passes to node j, also is the output value of node i.

For the derivation of $\frac{\partial E_d}{\partial net_j}$, it is necessary to distinguish between the output layer and the hidden layer.

3.4.1 Output layer weight training

For the output layer, net_j can only affect the rest of the network by the output value y_j of node j, that is, E_d is a function of y_j, and y_j is a function of net_j, where $y_j = simoid(net_j)$. So we can use the chain derivation rule again:

$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial y_j}\frac{\partial y_j}{\partial net_j} \tag{45}$$

First part of the above formula:

$$\frac{\partial E_d}{\partial y_j} = \frac{\partial}{\partial y_i}\frac{1}{2}\sum_{i \in outputs}(t_i - y_i)^2 \tag{46}$$

$$= \frac{\partial}{\partial y_i}\frac{1}{2}(t_j - y_j)^2 \tag{47}$$

$$= -(t_j - y_j) \tag{48}$$

Second part of the above formula:

$$\frac{\partial y_j}{\partial net_j} = \frac{\partial sigmoid(net_j)}{\partial net_j} \tag{49}$$

$$= y_i(1 - y_j) \tag{50}$$

Bring the first and second part into the above formula:

$$\frac{\partial E_d}{\partial net_j} = -(t_j - y_j)y_i(1 - y_j) \tag{51}$$

If $\delta_j = -\frac{\partial E_d}{\partial net_j}$, that is the error term δ of a node is the inverse of the partial derivative of the network error input to this node. Bring into the above formula, get

$$\delta_j = (t_j - y_j)y_i(1 - y_j) \tag{52}$$

Which is the formula (35).

Bring the above derivation into the stochastic gradient descent formula to get:

$$\omega_{ji} \leftarrow \omega_{ji} - \eta \frac{\partial E_d}{\partial \omega_{ji}}$$

$$= \omega_{ji} + \eta(t_j - y_j) y_i (1 - y_j) x_{ji} \tag{53}$$

$$= \omega_{ji} + \eta \delta_j x_{ji} \tag{54}$$

Which is the formula (37)

3.4.2 Hidden layer weight training

Now we have to derive the $\frac{\partial E_d}{\partial net_j}$ of the hidden layer.

First, we need to define a set $Downstream(j)$ of all direct downstream nodes of the node j. For example, for node 4, its next downstream node is node 8, node 9. We can see that net_j can only affect E_d by affecting $Downstream(j)$. Let net_k be the input to the downstream node of node j, then E_d is a function of net_k, and net_k is a function of net_j. Since there are multiple net_ks, we apply the full derivative formula and can make the following derivation:

$$\frac{\partial E_d}{\partial net_j} = \sum_{k \in Downstream(j)} \frac{\partial E_d}{\partial net_k} \frac{\partial net_k}{\partial net_j} \tag{55}$$

$$= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial net_j} \tag{56}$$

$$= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial a_j} \frac{\partial a_j}{\partial net_j} \tag{57}$$

$$= \sum_{k \in Downstream(j)} -\delta_k \omega_{kj} \frac{\partial a_j}{\partial net_j} \tag{58}$$

$$= \sum_{k \in Downstream(j)} -\delta_k \omega_{kj} a_j (1 - a_j) \tag{59}$$

$$= -a_j (1 - a_j) \sum_{k \in Downstream(j)} \delta_k \omega_{kj} \tag{60}$$

Bring $\delta_j = -\frac{\partial E_d}{\partial net_j}$ to the above formula, get

$$\delta_j = (1 - a_j) \sum_{k \in Downstream(j)} \delta_k \omega_{kj} \tag{61}$$

Which is formula 36

4 Recurrent Neural Network

Some tasks need to be able to process the sequence information better, that is, the previous input is related to the subsequent input. For example, when we understand the meaning of a sentence, it is not enough to understand each word in isolation. We need to deal with the whole sequence of these words. When we process the video, we cannot just go alone. Analyze each frame and analyze the entire sequence of connections of these frames. At this time, we need another significant neural network in the field of deep learning: Recurrent Neural Network.

4.1 Basic Recurrent Neural Network

The following figure is a simple recurrent neural network, which consists of an input layer, a hidden layer, and an output layer[10]:

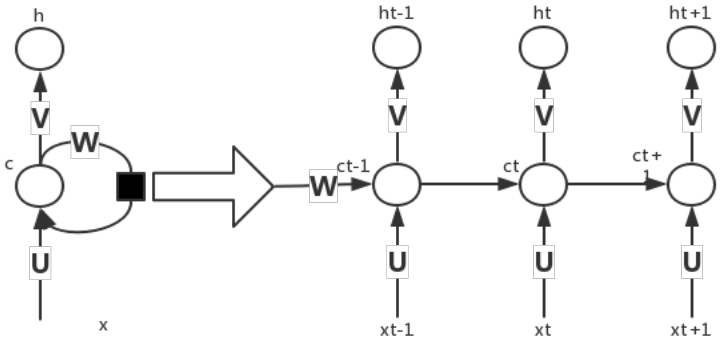

Figure 12. Recurrent neural network

After the network receives input x_t at time t, the value of the hidden layer is s_t and the output value is h_t. The critical point is that the value of st depends not only on x_t but also on c_{t-1}. We can use the following formula to represent the calculation method of the recurrent neural network[6]:

$$h_t = g(Vc_t) \tag{62}$$

$$c_t = f(Ux_t + Wc_{t-1}) \tag{63}$$

Equation 62 is the calculation formula of the output layer. The output layer is fully connected; That is each of its nodes connect to each node of the hidden layer. V is the weight matrix of the output layer, and g is the activation function. Equation 63 is a calculation formula for the hidden layer, which is a recurrent layer. U is the weight matrix

of the input x, W is the last value as the weight matrix for this input, and f is the activation function.

If we repeatedly bring Equation 63 into Equation 62, we will get[7]:

$$h_t = g(Vc_t) \tag{64}$$

$$= Vf(Ux_t + Wc_{t-1}) \tag{65}$$

$$= Vf\left(Ux_t + Wf(Ux_{t-1} + Wc_{t-2})\right) \tag{66}$$

$$= Vf\left(Ux_t + Wf\left(Ux_{t-1} + Wf(Ux_{t-2} + Wc_{t-3})\right)\right) \tag{67}$$

$$= Vf\left(Ux_t + Wf\left(Ux_{t-1} + Wf\left(Ux_{t-2} + Wf(Ux_{t-3} + \cdots)\right)\right)\right) \tag{68}$$

From the above, we know that the output value o_t of the recurrent neural network is affected by the previous input values x_t, x_{t-1}, x_{t-2}, x_{t-3}, ..., which is why the recurrent neural network can look forward with any number of input values

4.2 Training Algorithm for Recurrent Neural Networks: BPTT

The BPTT algorithm is a training algorithm for the recurrent layer. Its basic principle is the same as the BP algorithm. It also contains the same three steps:

1.Forward calculation of the output value of each neuron;

2.Calculating the error term δ_j value of each neuron in reverse, which is the partial derivative of the weighted input net_j of the error function E to the neuron j;

3.Calculate the gradient of each weight.

Finally, the weight is updated by the stochastic gradient descent algorithm.

The recurrent layer is shown below:

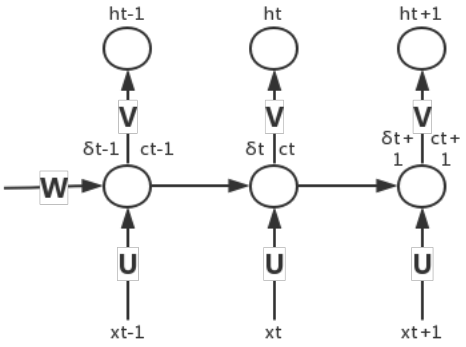

Figure 13. Recurrent layer

1.Forward calculation

Forward calculation of the recurrent layer using formula 63:

$$c_t = f(Ux_t + Wc_{t-1})$$

Note that the above c_t, x_t, and c_{t-1} are vectors, which are represented by bold letters; and U and V are matrices, which are represented by uppercase letters. The subscript of the vector represents the time, for example, c_t represents the value of the vector s at time t. We assume that the dimension of the input vector x is m, the dimension of the output vector c is n, then the dimension of the matrix U is $n \times m$, and the dimension of the matrix W is $n \times n$. Then we get the matrix from the above formula

$$\begin{bmatrix} c_1^t \\ c_2^t \\ \cdot \\ \cdot \\ c_n^t \end{bmatrix} = f\left(\begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1m} \\ u_{21} & u_{22} & \cdot & u_{2m} \\ & & \cdot & \\ u_{n1} & u_{n2} & \cdots & u_{nm} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ x_m \end{bmatrix} + \begin{bmatrix} \omega_{11} & \omega_{12} & \cdots & \omega_{1n} \\ \omega_{21} & \omega_{22} & \cdot & \omega_{2n} \\ & & \cdot & \\ \omega_{n1} & \omega_{n2} & \cdots & \omega_{nn} \end{bmatrix} \begin{bmatrix} c_1^{t-1} \\ c_2^{t-1} \\ \cdot \\ \cdot \\ c_n^{t-1} \end{bmatrix} \right) \tag{69}$$

Here we use handwritten letters to represent an element of a vector, its subscript indicates that it is the first element of the vector, and its superscript indicates the first few moments. For example, c_j^t represents the value of the jth element of the vector s at time t. u_{ji} represents the weight of the i-th neuron in the input layer to the j-th neuron in the recurrent layer. ω_{ji} represents the weight of the i-th neuron at the t-1th point of the recurrent layer to the j-th neuron at the tth moment of the recurrent layer.

2 Calculation of error terms

The BTPP algorithm propagates the error term δ_t^l value of the l layer time t in two directions. One direction is that transmitted to the upper layer network to obtain δ_t^{l-1}. This part is only related to the weight matrix U; The other is that the direction is along its timeline pass to the initial time t_1, to get yields δ_1^l, which is only related to the weight matrix W.

We use the vector net_t to represent the weighted input of the neuron at time t:

$$net_t = Ux_t + Wc_{t-1} \tag{70}$$

$$c_{t-1} = f(net_{t-1}) \tag{71}$$

So

$$\frac{\partial net_t}{\partial net_{t-1}} = \frac{\partial net_t}{\partial c_{t-1}} \frac{\partial c_{t-1}}{\partial net_{t-1}} \tag{72}$$

We use a for the column vector and a^T as the row vector. The first term of the above formula is the vector function to vector the derivative, and the result is the Jacobian matrix:

$$\frac{\partial net_t}{\partial s_{t-1}} = \begin{bmatrix} \frac{\partial net_1^t}{\partial c_1^{t-1}} & \frac{\partial net_1^t}{\partial c_2^{t-1}} & & \frac{\partial net_1^t}{\partial c_n^{t-1}} \\ \frac{\partial net_2^t}{\partial c_1^{t-1}} & \frac{\partial net_2^t}{\partial c_2^{t-1}} & \cdots & \frac{\partial net_2^t}{\partial c_n^{t-1}} \\ & \vdots & & \\ \frac{\partial net_n^t}{\partial c_1^{t-1}} & \frac{\partial net_n^t}{\partial c_2^{t-1}} & \cdots & \frac{\partial net_n^t}{\partial c_n^{t-1}} \end{bmatrix} \tag{73}$$

$$= \begin{bmatrix} \omega_{11} & \omega_{12} & & \omega_{1n} \\ \omega_{21} & \omega_{22} & \cdots & \omega_{1n} \\ & \vdots & \cdots & \\ \omega_{n1} & \omega_{n2} & \cdots & \omega_{nn} \end{bmatrix} \tag{74}$$

$$= W \tag{75}$$

Similarly, the second item of the above formula is also a Jacobian matrix.

$$\frac{\partial c_{t-1}}{\partial net_{t-1}} = \begin{bmatrix} \frac{\partial c_1^{t-1}}{\partial net_1^{t-1}} & \frac{\partial c_1^{t-1}}{\partial net_2^{t-1}} & & \frac{\partial c_n^{t-1}}{\partial net_n^{t-1}} \\ \frac{\partial c_2^{t-1}}{\partial net_1^{t-1}} & \frac{\partial c_2^{t-1}}{\partial net_2^{t-1}} & \cdots & \frac{\partial c_2^{t-1}}{\partial net_n^{t-1}} \\ & \vdots & & \\ \frac{\partial c_n^{t-1}}{\partial net_1^{t-1}} & \frac{\partial c_n^{t-1}}{\partial net_2^{t-1}} & \cdots & \frac{\partial c_n^{t-1}}{\partial net_n^{t-1}} \end{bmatrix} \tag{76}$$

$$= \begin{bmatrix} f'(net_1^{t-1}) & 0 & & 0 \\ 0 & f'(net_2^{t-1}) & \cdots & 0 \\ & \vdots & \cdots & \\ 0 & 0 & \ldots & f'(net_n^{t-1}) \end{bmatrix} \tag{77}$$

$$= diag[f'(net_{t-1})] \tag{78}$$

Where diag[a] represents the creation of a diagonal matrix based on the vector a:

$$diag(a) = \begin{bmatrix} a_1 & 0 & & 0 \\ 0 & a_2 & \cdots & 0 \\ & \cdot & \cdots & \\ & \cdot & & \\ 0 & 0 & \ldots & a_n \end{bmatrix} \tag{79}$$

Finally, combine the two items, we can get:

$$\frac{\partial net_t}{\partial net_{t-1}} = \frac{\partial net_t}{\partial s_{t-1}} \frac{\partial c_{t-1}}{\partial net_{t-1}} \tag{80}$$

$$= W diag[f'(net_{t-1})] \tag{81}$$

$$= \begin{bmatrix} \omega_{11} f'(net_1^{t-1}) & \omega_{11} f'(net_2^{t-1}) & & \omega_{1n} f'(net_n^{t-1}) \\ \omega_{21} f'(net_1^{t-1}) & \omega_{22} f'(net_2^{t-1}) & \cdots & \omega_{2n} f'(net_n^{t-1}) \\ & \cdot & \cdots & \\ & \cdot & & \\ \omega_{n1} f'(net_1^{t-1}) & \omega_{n2} f'(net_2^{t-1}) & \ldots & \omega_{nn} f'(net_2^{t-1}) \end{bmatrix} \tag{82}$$

The above formula describes the rule of passing δ a period forward along time. With this rule, we can find the error term δ_k at any time k:

$$\delta_k^T = \frac{\partial E}{\partial net_k} \tag{83}$$

$$= \frac{\partial E}{\partial net_t} \frac{\partial net_t}{\partial net_k} \tag{84}$$

$$= \frac{\partial E}{\partial net_t} \frac{\partial net_t}{\partial net_{t-1}} \frac{\partial net_{t-1}}{\partial net_{t-2}} \cdots \frac{\partial net_{k+1}}{\partial net_k} \tag{85}$$

$$= W diag[f'(net_{t-1})] W diag[f'(net_{t-2})] \cdots W diag[f'(net_k)] \delta_t^l \tag{86}$$

$$= \delta_t^T \prod_{i=k}^{t-1} W diag[f'(net_i)] \tag{87}$$

formula 87 is an algorithm that propagates the error term back in time. The recurrent layer passes the error term back to the upper layer network, which is the same as the standard full connection layer.

The weighted input net^l of the recurrent layer is related to the weighted input net^{l-1} of the previous layer as follows

$$net_t^l = U a_t^{l-1} + W c_{t-1} \tag{88}$$

$$a_t^{l-1} = f^{l-1}\left(net_t^{l-1}\right) \tag{89}$$

In the above formula, net_t^l is the weighted input of the l layer of neurons (assuming that the l layer is the circular layer); net_t^{l-1} is the weighted input of the $l-1$th layer of neurons; a_t^{l-1} is the output of the $l-1$th layer of neurons; f^{l-1} is The activation function of layer $l-1$.

$$\frac{\partial net_t^l}{\partial net_t^{l-1}} = \frac{\partial net_t^l}{\partial a_t^{l-1}} \frac{\partial a_t^{l-1}}{\partial net_t^{l-1}} \tag{90}$$

$$= Udiag\left[f'^{l-1}\left(net_t^{l-1}\right)\right] \tag{91}$$

So we get

$$(\delta_t^{l-1})^T = \frac{\partial E}{\partial net_t^{l-1}} \tag{92}$$

$$= \frac{\partial E}{\partial net_t^l} \frac{\partial net_t^l}{\partial net_t^{l-1}} \tag{93}$$

$$= (\delta_t^l)^T Udiag\left[f'^{l-1}\left(net_t^{l-1}\right)\right] \tag{94}$$

Formula 94 is the algorithm pass the error term to the previous

3 Weight gradient calculation

First, we calculate the gradient $\frac{\partial E}{\partial W}$ in weight matrix W with the error function E.

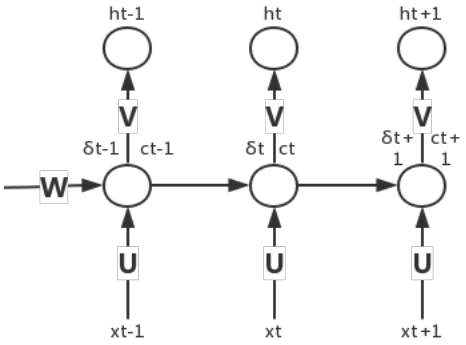

Figure 14 recurrent layer

The figure above shows the amount we have calculated so far in the first two steps, including the output value c_t of the recurrent layer at each time t, and the error term δ_t. According to the weight gradient calculation algorithm of the fully connected network: as long as the error term δ_t at any one time and the output value c_{t-1} of the recurrent layer at the previous time are known, the gradient $\nabla W_t E$ of the weight matrix at time t can be obtained according to the following formula :

$$\nabla_{W_t} E = \begin{bmatrix} \delta_1^t c_1^{t-1} & \delta_1^t c_2^{t-1} & \cdots & \delta_1^t c_n^{t-1} \\ \delta_2^t c_1^{t-1} & \delta_2^t c_2^{t-1} & \cdots & \delta_2^t c_n^{t-1} \\ \vdots & & & \\ \delta_n^t c_1^{t-1} & \delta_n^t c_2^{t-1} & \cdots & \delta_n^t c_n^{t-1} \end{bmatrix} \tag{95}$$

In formula 95, δ_i^t represents the i component of the error term vector at time t; c_i^{t-1}represents the output value of the i neuron in the cyclic layer at time $t-1$. Then we carry out the derivation of formula 95

.

$$net_t = Ux_t + Wc_{t-1} \tag{96}$$

$$\begin{bmatrix} net_1^t \\ net_2^t \\ \vdots \\ net_n^t \end{bmatrix} = Ux_t + \begin{bmatrix} \omega_{11} & \omega_{12} & \cdots & \omega_{1n} \\ \omega_{21} & \omega_{22} & \cdots & \omega_{2n} \\ & \vdots & & \\ \omega_{n1} & \omega_{n2} & \cdots & \omega_{nn} \end{bmatrix} \begin{bmatrix} c_1^{t-1} \\ c_2^{t-1} \\ \vdots \\ c_n^{t-1} \end{bmatrix} \tag{97}$$

$$= Ux_t + \begin{bmatrix} \omega_{11} c_1^{t-1} + \omega_{12} c_2^{t-1} \cdots \omega_{1n} c_n^{t-1} \\ \omega_{21} c_1^{t-1} + \omega_{22} c_2^{t-1} \cdots \omega_{2n} c_n^{t-1} \\ \vdots \\ \omega_{n1} c_1^{t-1} + \omega_{n2} c_2^{t-1} \cdots \omega_{nn} c_n^{t-1} \end{bmatrix} \tag{98}$$

Because the derivation of W has nothing to do with Ux_t, so we ignore it. Now we consider the weighting term ω_{ji}. By observing the above formula, we can see that ω_{ji} is only related to net_j^t, so:

$$\frac{\partial E}{\partial \omega_{ji}} = \frac{\partial E}{\partial net_j^t} \frac{\partial net_j^t}{\partial \omega_{ji}} \tag{99}$$

$$= \delta_j^t c_i^{t-1} \tag{100}$$

From the rule in formula 100, we can get formula 95

We have obtained the gradient $\nabla_{W_t} E$ of the weight matrix W at time t, and the total gradient $\nabla_W E$ is the sum of the gradients at each moment.

$$\nabla_W E = \sum_{i=1}^{t} \nabla_{W_i} E \tag{101}$$

$$= \begin{bmatrix} \delta_1^t c_1^{t-1} & \delta_1^t c_2^{t-1} & \cdots & \delta_1^t c_n^{t-1} \\ \delta_2^t c_1^{t-1} & \delta_2^t c_2^{t-1} & \cdots & \delta_2^t c_n^{t-1} \\ \vdots & & & \\ \delta_n^t c_1^{t-1} & \delta_n^t c_2^{t-1} & \ldots & \delta_n^t c_n^{t-1} \end{bmatrix} + \cdots + \begin{bmatrix} \delta_1^t c_1^0 & \delta_1^t c_2^0 & \cdots & \delta_1^t c_n^0 \\ \delta_2^t c_1^0 & \delta_2^t c_2^0 & \cdots & \delta_2^t c_n^0 \\ \vdots & & & \\ \delta_n^t c_1^0 & \delta_n^t c_2^0 & \ldots & \delta_n^t c_n^0 \end{bmatrix} \tag{102}$$

Formula 102 is a formula for calculating the gradient of the recurrent layer weight matrix W.

Similar to the weight matrix W, we can get the calculation method of the weight matrix U.

$$\nabla_{U_t} E = \begin{bmatrix} \delta_1^t c_1^t & \delta_1^t c_2^t & \cdots & \delta_1^t c_m^t \\ \delta_2^t c_1^t & \delta_2^t c_2^t & \cdots & \delta_2^t c_m^t \\ \vdots & & & \\ \delta_n^t c_1^t & \delta_n^t c_2^t & \ldots & \delta_n^t c_m^t \end{bmatrix} \tag{103}$$

Formula 103 is the gradient of the error function to the weight matrix U at time t. The same as the weight matrix W, the final gradient is also the sum of the gradients at each moment:

$$\nabla_U E = \sum_{i=1}^{t} \nabla_{U_i} E$$

Till now we finish the calculation of the BPTT.

5 Long Short-Term Memory Algorithm

Long Short-Term Memory Algorithm (LSTM), which successfully solves the defects of the original cyclic neural network, has become the most popular RNN and has been successfully applied in many fields such as speech recognition, picture description, and natural language processing.

Among them, it is an important reason to be able to deal with gradient explosions and gradient hours that occur in ordinary RNN. From the last section, we know that The formula for the error term to propagate back in time:

$$\delta_k^T = \delta_t^T \prod_{i=k}^{t-1} W diag[f'(net_i)] \tag{104}$$

We can get the upper bound of the modulus of δ_k^T according to the inequality below (the modulo is a measure of the size of each value in δ_k^T):

$$\|\delta_k^T\| \leq \| \delta_k^T \| \prod_{i=k}^{t-1} \| diag[f'(net_i)] \| \| W \| \tag{105}$$

$$\leq \| \delta_k^T \| \left(\beta_f \beta_W\right)^{t-k} \tag{106}$$

We can see that the error term δ passed from time t to time k, and the upper bound of its value is an exponential function of $\beta_f \beta_W$. $\beta_f \beta_W$ is the upper bound of the diagonal matrix$diag[f'(net_i)$and the matrix W absolute value, respectively. Obviously, unless the value of the product $\beta_f \beta_W$ is around 1, when the t-k is large (that is, when the error transmitted for many times), the value of the whole expression becomes extremely small (when the $\beta_f \beta_W$ product is less than 1) or (When the $\beta_f \beta_W$ product is greater than 1), the former is the gradient disappears, and the latter is the gradient explosion[11].

In general, gradient explosions are easier to handle. Our program will receive a NaN error when the gradient explodes. We can also set a gradient threshold that can be intercepted directly when the gradient exceeds this threshold.

The vanishing gradient is more challenging to detect and more difficult to handle. We have proved that the final gradient of the weighted array W is the sum of the gradients at each moment. When the vanishing gradient, starting from this time t and going forward, the resulting gradient (almost zero) will not contribute to the final gradient value, which is equivalent to what the network state h is before the time t, In training,

it will not affect the update of the weight array W, that is, the network has actually ignored the state before the time t.

That's why we try to use LSTM algorithm to make the load forecasting. The hidden layer of the original RNN has only one state, h, which is very sensitive to short-term inputs. Then, if we add another state, c, let it save the long-term state, the newly added state c, called the cell state, is expanded according to the time dimension, as shown in the following figure:

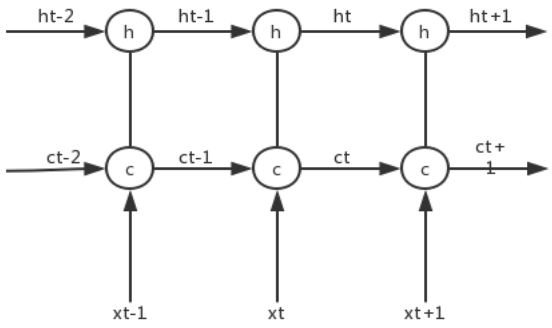

Figure 15. transmission in LSTM

The above figure is just a schematic diagram. We can see that at time t, there are three inputs to the LSTM: the input value x_t of the current time network, the output valueh_{t-1} of the previous time LSTM, and the unit state c_{t-1} of the previous moment; There are two outputs of LSTM: the current time LSTM output value h_t, and the current unit state c_t. Note that x, h, and c are all vectors.

The key to LSTM is how to control the long-term state c. Here, the idea of LSTM is to use three control gates. The first gate is responsible for controlling to continue saving the long-term state c; the second gate is responsible for controlling the input of the immediate state to the long-term state c; the third gate is responsible for controlling whether the long-term state c is the output of the current LSTM.

5.1 Forward calculation of LSTM

How is the switch described earlier implemented in the algorithm? This uses the concept of a gate. The gate is actually a layer of fully connected layers whose input is a vector and the output is a real vector between 0 and 1. Assuming W is the weight vector of the gate, which b is an offset term, the gate can be expressed as:

$$g(x) = \sigma(W_x + b) \tag{107}$$

The use of the gate is to multiply the element's output vector by the vector we want to control. Since the output of the gate is a real vector between 0 and 1, then when the gate output is 0, any vector multiplied will result in a 0 vector, which is equivalent to nothing can pass; when the output is 1, any vector There is no change in multiplication, which is equivalent to passing. Since the value range of x (the sigmoid function) is (0, 1), the state of the gate is half-open and half-closed.

The LSTM uses two gates to control the contents of the unit state c. One is the forget gate, which determines how much the unit state of c_{t-1} is retained to the current time c_t at the previous time; The other is the input gate, it determines how much the input x_t of the network is saved to the unit state c_t at the current moment. The LSTM uses an output gate to control how much of the unit state c_t is output to the current output value h_t of the LSTM.

5.1.1 Forgotten gate

First we calculate the forgotten gate[12]:

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right) \tag{108}$$

In the above formula, W_f is the weight matrix of the forgotten gate, $[h_{t-1}, x_t]$ represents the connection of two vectors into a longer vector, b_f is the bias term of the forgetting gate, and x is the sigmoid function. If the dimension entered is d, the dimension of the hidden layer is d_h, and the dimension of the cell state is d_c (usually d_c=d_h), then the weight matrix of the forgetting gate wf dimension is d_c*(d_h+d_x). In fact, the weight matrix W_f is a combination of two matrices: one is W_{fh}, which corresponds to the input ht-1, its dimension is d_c*d_h; one is W_{fx}, which corresponds to the input x_t, and its dimension is d_c*d_x. W_f can be written as:

$$\begin{aligned}\left[W_f\right]\begin{bmatrix}h_{t-1}\\ x_t\end{bmatrix} &= \begin{bmatrix}W_{fh} & W_{fx}\end{bmatrix}\begin{bmatrix}h_{t-1}\\ x_t\end{bmatrix}\\ &= W_{fh}h_{t-1} + W_{fx}x_t\end{aligned}$$

The calculation process of the Forgotten Gate is below:

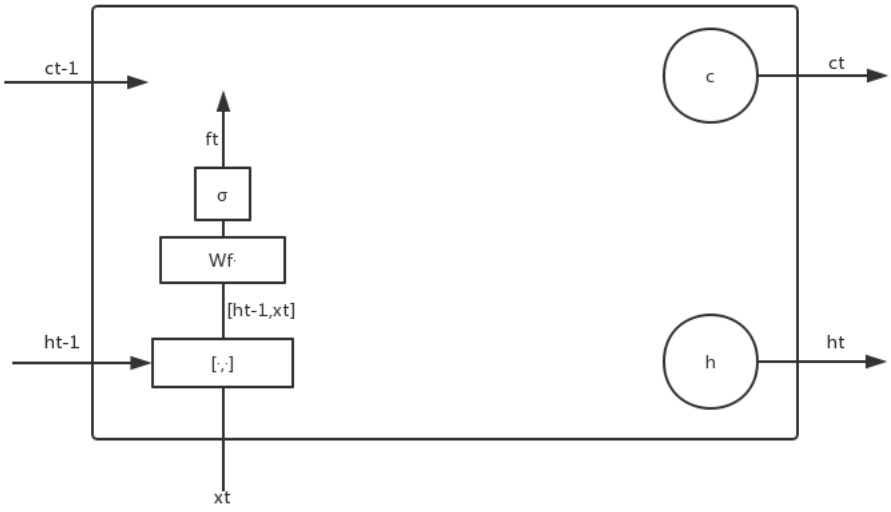

Figure 16. calculation of the Forgotten gate

5.1.2 Input gate

Then let us see the input gate[12]

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{109}$$

In the above formula, W_i is the weight matrix of the input gate, b_i is the offset term of the input gate. The calculation process of the Forgotten Gate input gate is below:

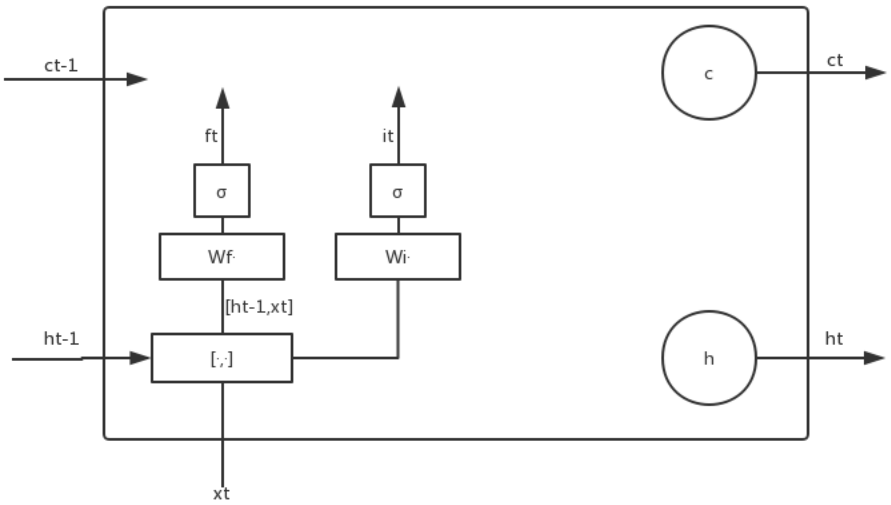

Figure 17. calculation of i_t in the Input gate

Next, we calculate the state of the cell used to describe the current input $\tilde{c}_t$ which is calculated based on the last output and this time input[12]:

$$\tilde{c}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \tag{110}$$

The figure below is the calculation of $\tilde{c}_t$:

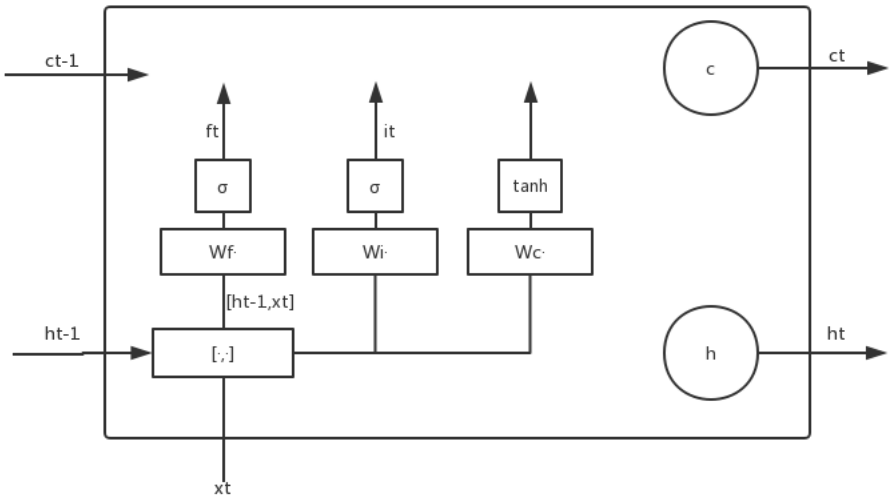

Figure 18. calculation of $\tilde{c}_t$ in the Input gate

Then we calculate the cell state c_t at the current time. It is multiplied by the last forgetting gate f_t by the element state c_{t-1}, and then multiplied by the input cell i_t, by the element state $\tilde{c}_t$ of the current input, and then generate the sum of the two products[12] :

$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \tag{111}$$

The symbol $\circ$ indicates multiplication by element. The figure below is the calculation of c_t

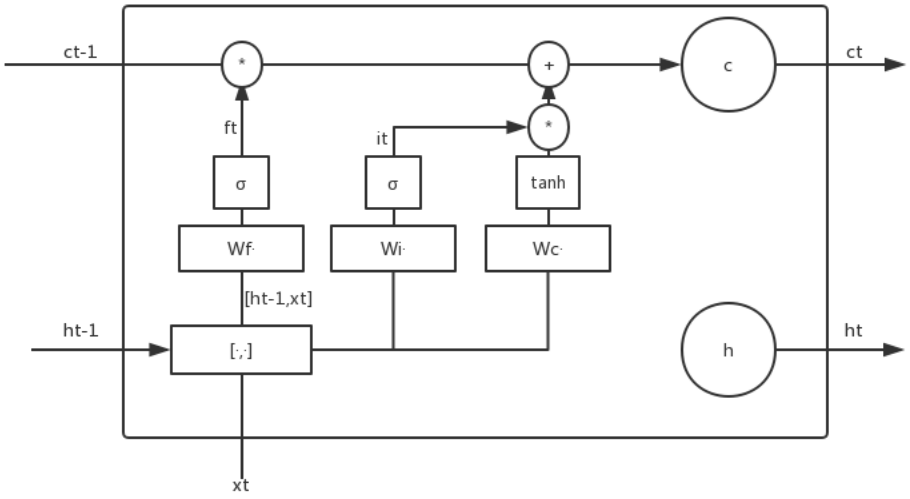

Figure 19. calculation of c_t in the Input gate

In this way, we combine the LSTM with the current memory $\tilde{c}_t$and the long-term memory c_{t-1} to form a new unit state c_t. Thanks to the control of the forgetting gate, it can save information long before a long time. Due to the control of the input gate, it can prevent the current irrelevant content from entering the memory.

5.1.3 Output gate

Next, let us look at the output gate, which controls the effect of long-term memory on the current output[12]:

$$o_t = \sigma(W_O \cdot [h_{t-1}, x_t] + b_0) \tag{112}$$

The figure below shows the calculation of the output gate:

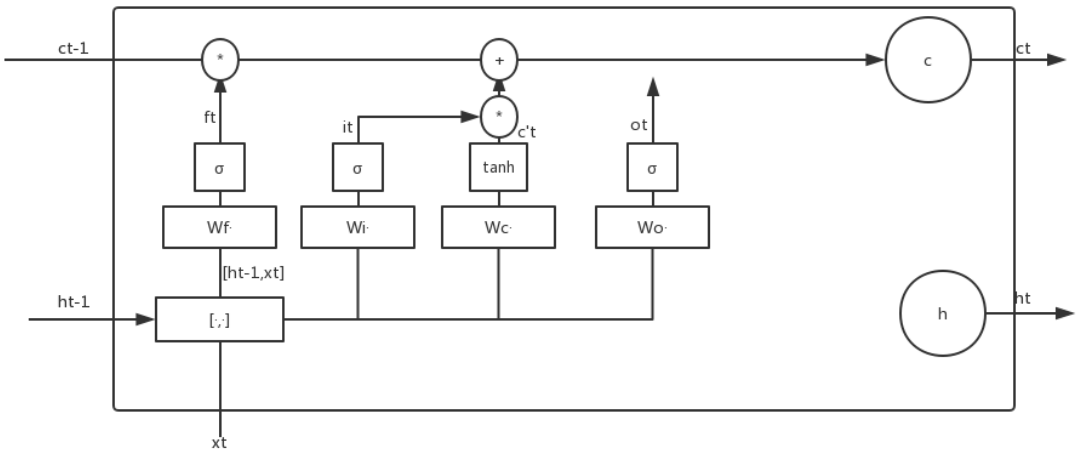

Figure 20. calculation of the output gate

The final output of the LSTM is determined by the output gate and the unit state:

$$h_t = o_t \circ \tanh(c_t) \tag{113}$$

5.1.4 Final output

The figure below shows the calculation of the final output of the LSTM:

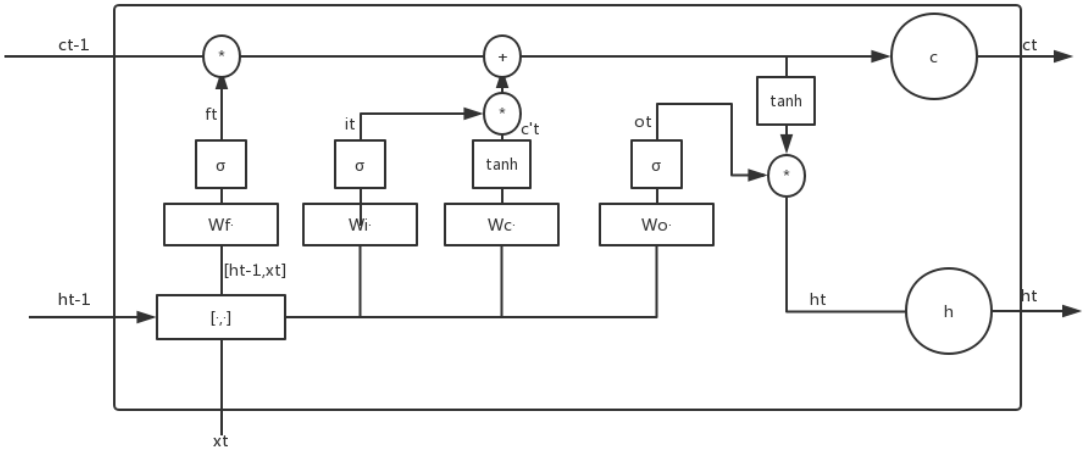

Figure 21. calculation of the final output

Till now, we have finished the LSTM forward calculation.

6 Training of the LSTM

Framework of LSTM training algorithm[13]:

The training algorithm of LSTM is still a backpropagation algorithm, which has the following three steps:

1.The output value of each neuron is Forward Propagation, for LSTM, that is, the five vectors of values f_t, i_t c_t, o_t and h_t. The calculation method has been described in the previous section.

2.Calculate the error term value δ for each neuron in reverse. Like the recurrent neural network, the backpropagation of the LSTM error term also includes two directions: one is the backpropagation along time: from the current t time, calculate the error term at each moment; The other is to move the error term up one layer.

3.Calculated the gradient of each weight based on the corresponding error term.

There are eight groups of parameters that LSTM needs to learn, which are: the weight matrix W_f and bias term b_f of the forgetting gate, the weight matrix W_i and bias term b_i of the input gate, the weight matrix W_o and bias term b_o of the output gate, and the weight matrix W_c of the state of the computing unit and offset item b_c.

6.1 Training process of LSTM in Matlab

When training networks for deep learning, it is often useful to monitor the training progress. By plotting various metrics during training, we can learn how the training is progressing. For example, We can see if the accuracy of the training network is rising and whether the network is overfit the training data by the training process.

When choosing to specify 'training-progress' as the 'Plots' value in training options and start network training, train network draw a chart to show the changes in parameters for each iteration of the training, each iteration is to update the gradients and weights in the model. In regression networks, Matlab plots the root mean square error (RMSE) but not the accuracy[13,15]. We choose the Stochastic Gradient Descent with Momentum(SGDM) to training my data.

Gradient descent is a first-order iterative optimization algorithm used to find the minimum value of a function. In order to ensure that the local minimum of the function can be found,

we need to calculate the step size opposite to the current function gradient. The options for the training process are listed below[16]:

6.1.1 Initial Weights and Biases

In the matlab, the default initial weights is a 0 mean and standard deviation of 0.01 of a gaussian distribution. The default for the initial bias value is 0. 1 choose the default value in the training process.

6.1.2 Initial learning rate

Initial learning rate used for training.this parameter has a heavyweight of effecting the training process. If we choose a value too low, then training takes a long time. If choose is too high, then training might reach a suboptimal result or diverge. Most of the learning rate is between 0.1-0.001. Choose the value between them and try to find the best value which can get the lowest loss function in the training process.

6.2 Training process results

By default, the training network uses this initial learning rate throughout the entire training process. I choose to modify the learning rate every specified number of epochs by multiplying the learning rate with a factor. Instead of using a low, fixed learning rate throughout the training process, I choose a more significant learning rate at the beginning of training and gradually reduce this value during optimization. Doing so can shorten the training time while enabling smaller steps towards the minimum of the loss as training progresses.

Set the other options in the training process as constant:

Minibatchsize:150

Maxexpoches:150

Gradient threshold:1

Learn rate drop period:100

Learn rate drop factor:0.2

Change the initial learning rate to find the best training parameter to get the lowest RMSE and Loss.

Show the figure which uses different initial learning rate with 0.1,0.01,0.001 below

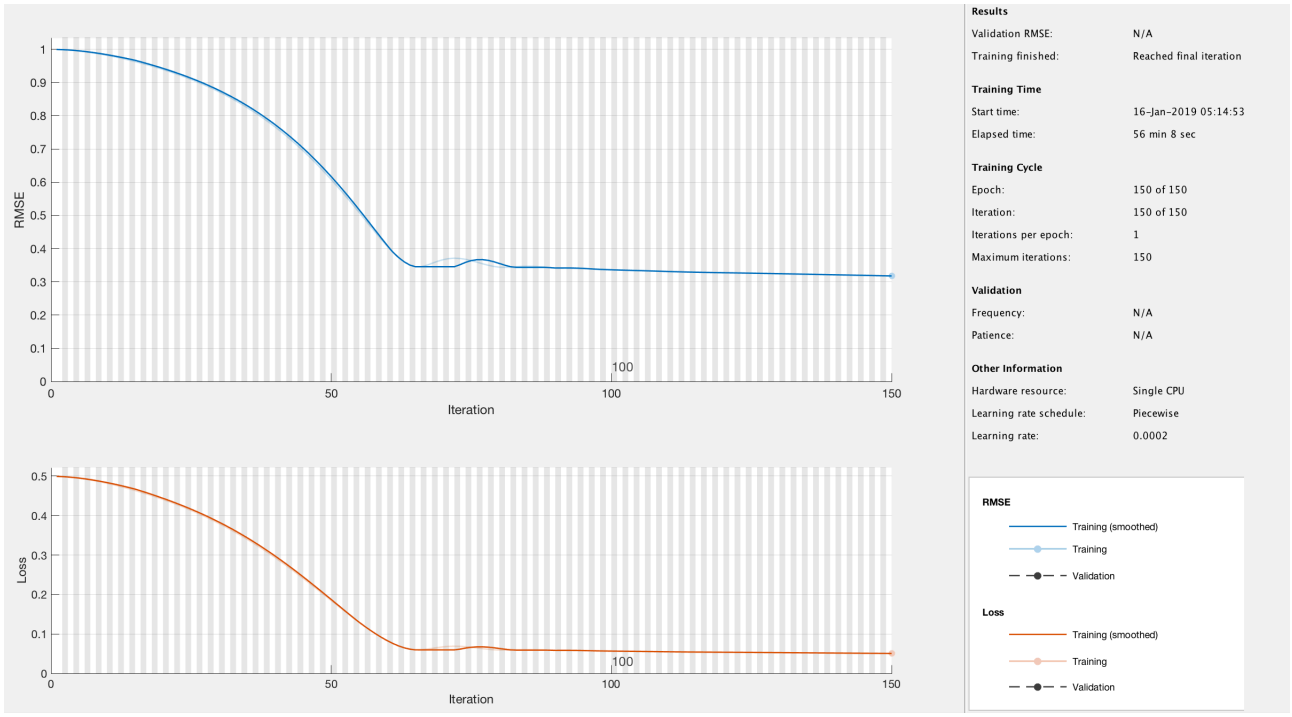

Figure 22. Training process with learning rate 0.001

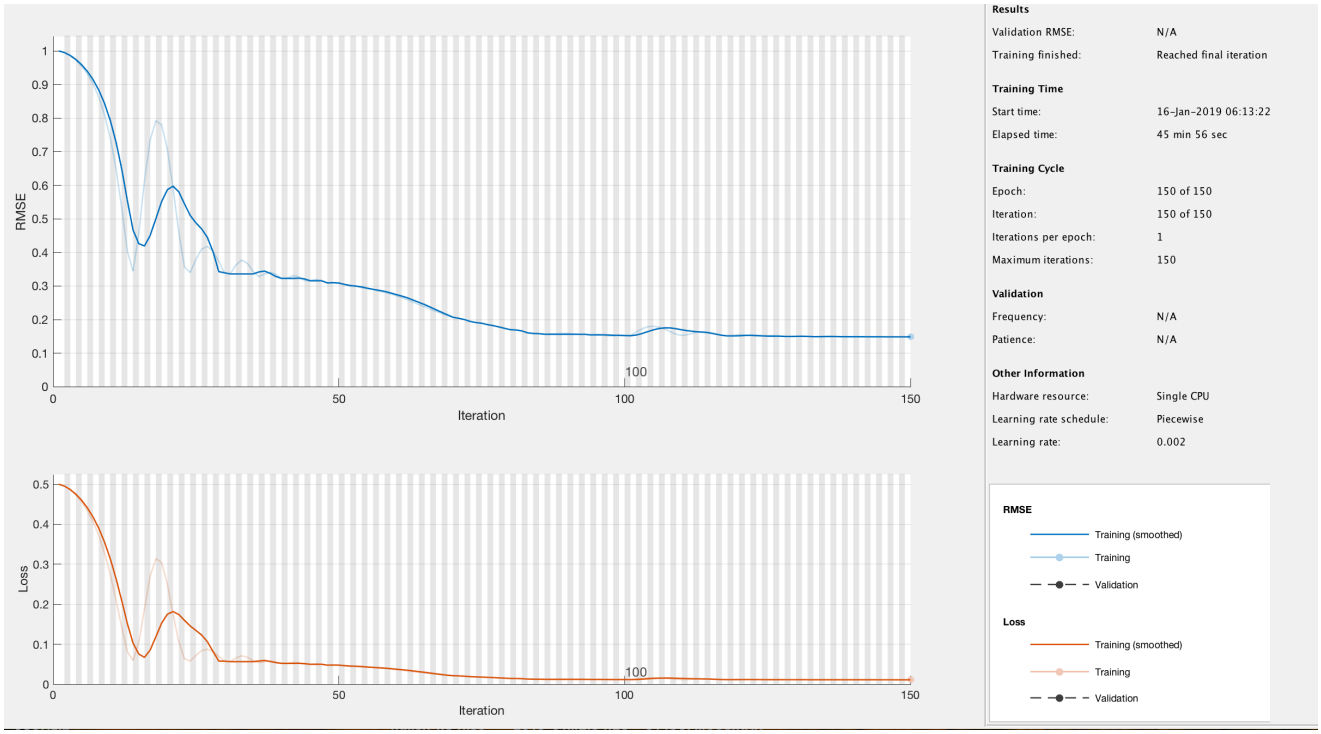

Figure 23. Training process with learning rate 0.01

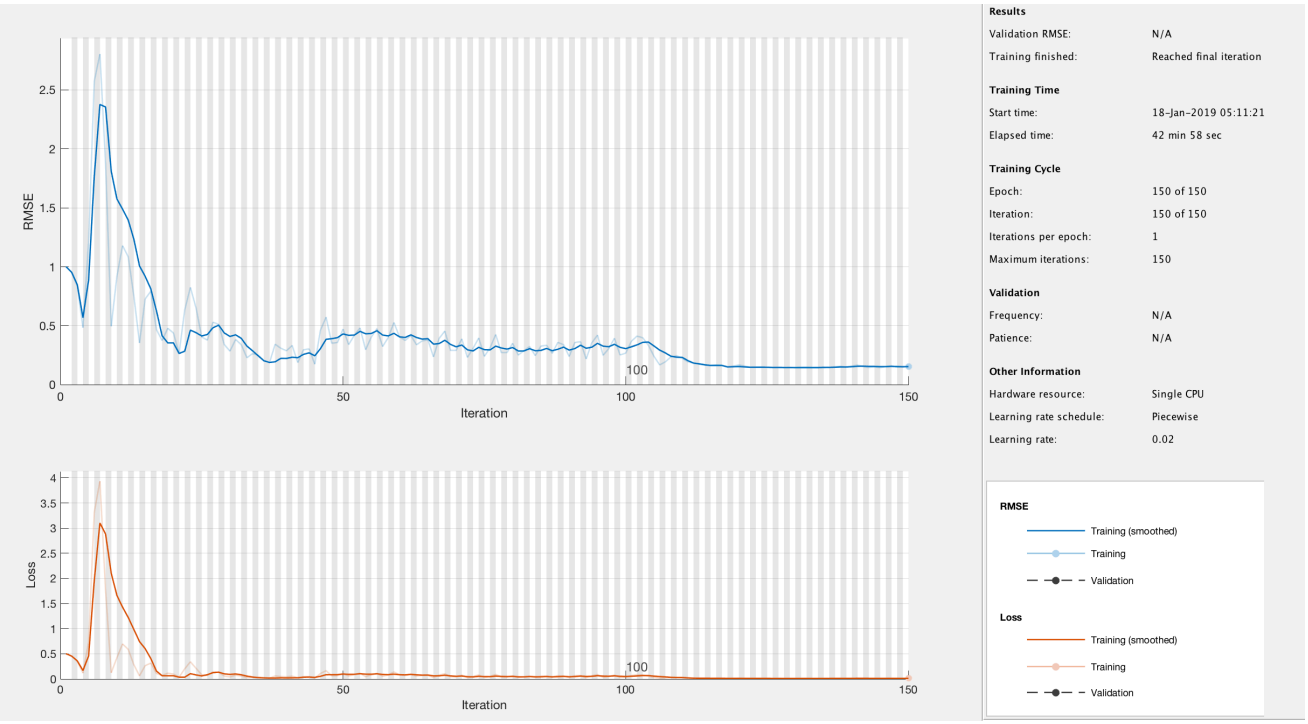

Figure 24. Training process with learning rate 0.1

From the figure, we can find that when the initial learning rate has a significant effect on the whole training process. When the learning rate in about 0.01, it needs more time to train, and it may not get the best point in the limit iterations. While we increased the rate, the training time decreased and may face a gradient explosion. If the learning rate is high, the training may not converge at all, or even diverge. The amount of change in weight can be huge, making the optimization cross the minimum, making the loss function worse. To improve the training process, we use a different learning rate. Moreover, we can find the 0.01 is the best value in the whole range of 0.001 to 0.1 of the learning rate. It gets the lowest RMSE and loss and takes the shortest time.

After we get the options of the training models, we can adjust the time steps of the training data. I chose the time steps with 150 data, so we want to compare the influence of the different time steps in the model. We set the time steps as 50, 300, and 600 to compare the figure of the training process below.

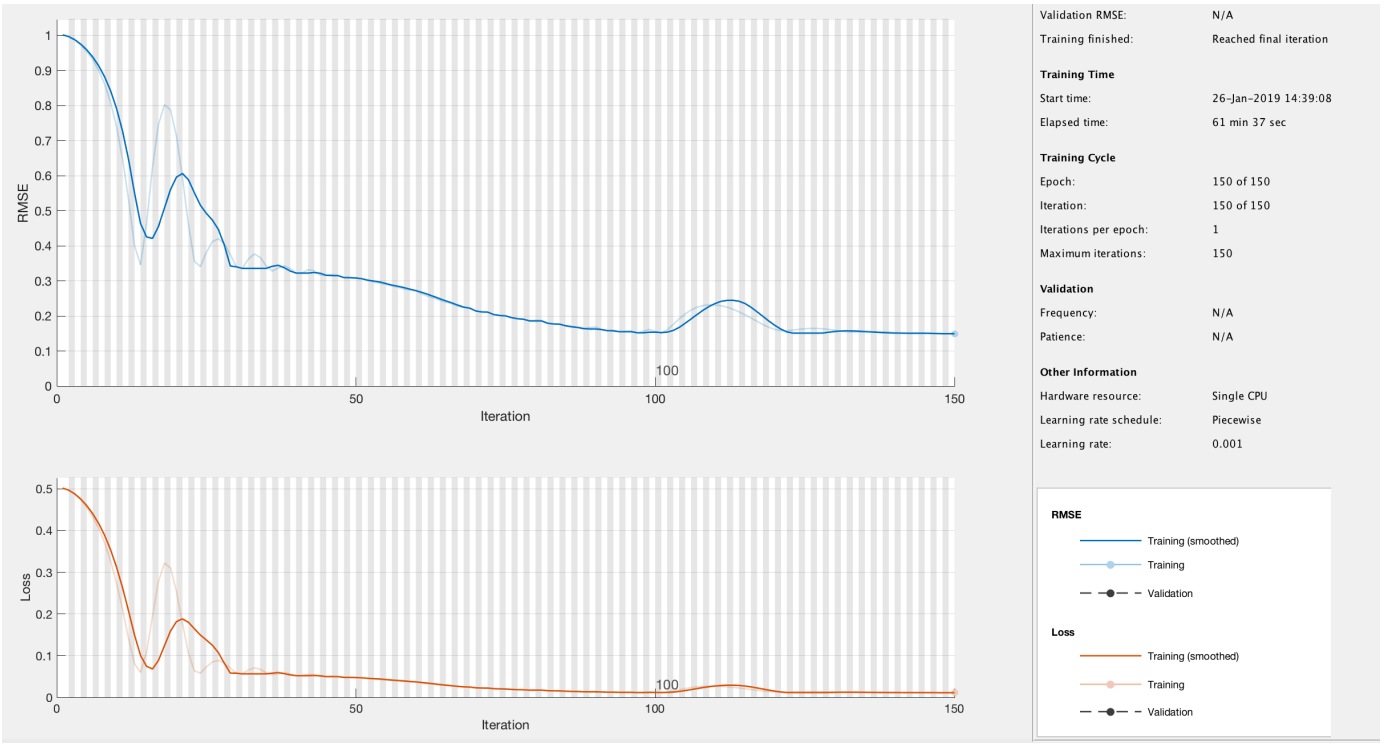

Figure 25. Training process with time steps size is 50

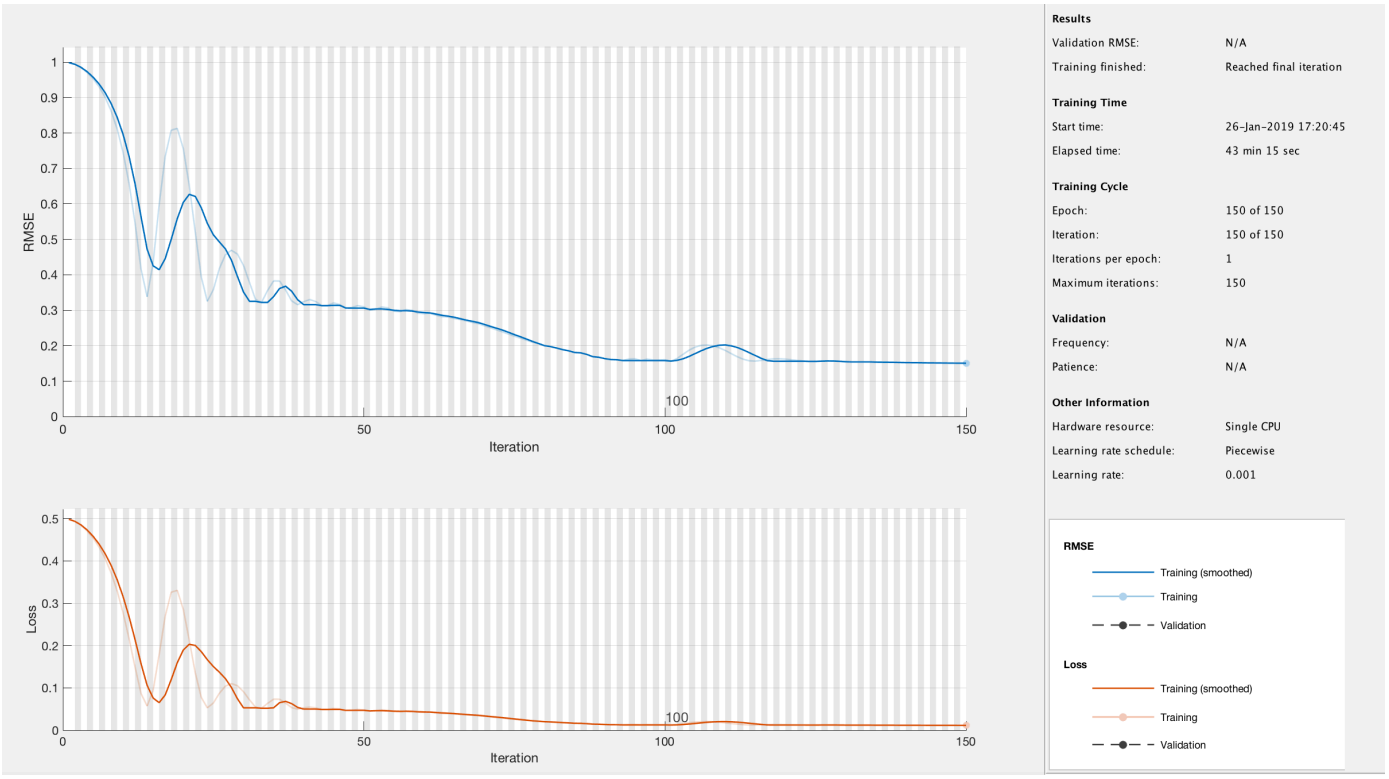

Figure 26. Training process with time steps size is 300

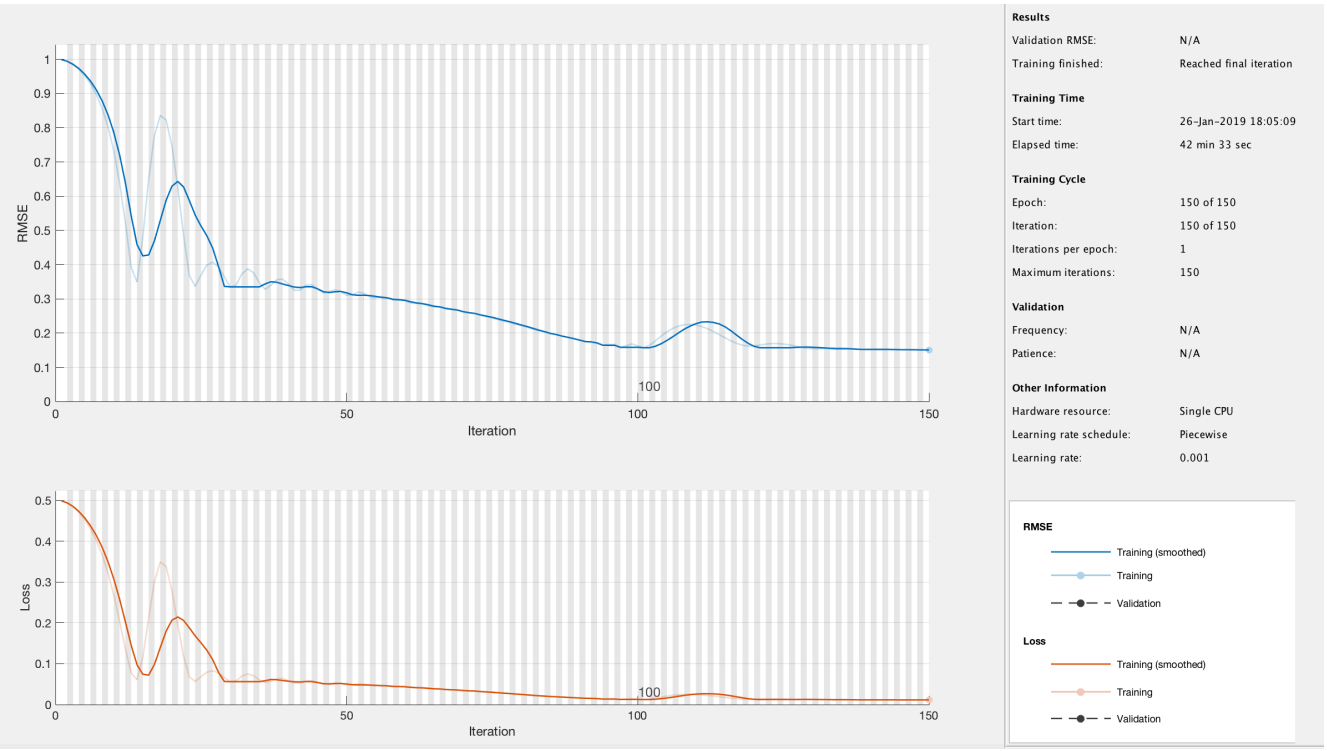

Figure 27. Training process with time steps size is 600

We can find that while we increase the time steps, our training result is also good enough for us and the loss of the function is in the range we can accept. This is also proved that LSTM has an excellent performance in a long time series data sequence. We will test its performance in the predicted result.

7 predict results

We have built the different sets of the training process model, but we need to test its performance in the test data to make sure this model is reliable. Thus we use the test data to predict the load and compare the accuracy of these models.

7.1 predict And Update State

When predicting multiple time steps in the future, we need to use the predicted values and the updated status values to make predictions on a time-by-step basis, and update the network status each time. Each prediction uses the previous predicted value as the input to the function.

Pre-processing the data to standardize it so that the convergence of the entire training cannot be expected to reach the expected level.

After we preprocess the test data, we have two choices to test our model. The first one is to update the state use the predict data from the model. Also, the other one has updated the state with the actual values of time steps between predictions.

7.1.1 predict by the training model

First, we test the model performance, which updates the state with the predict data.

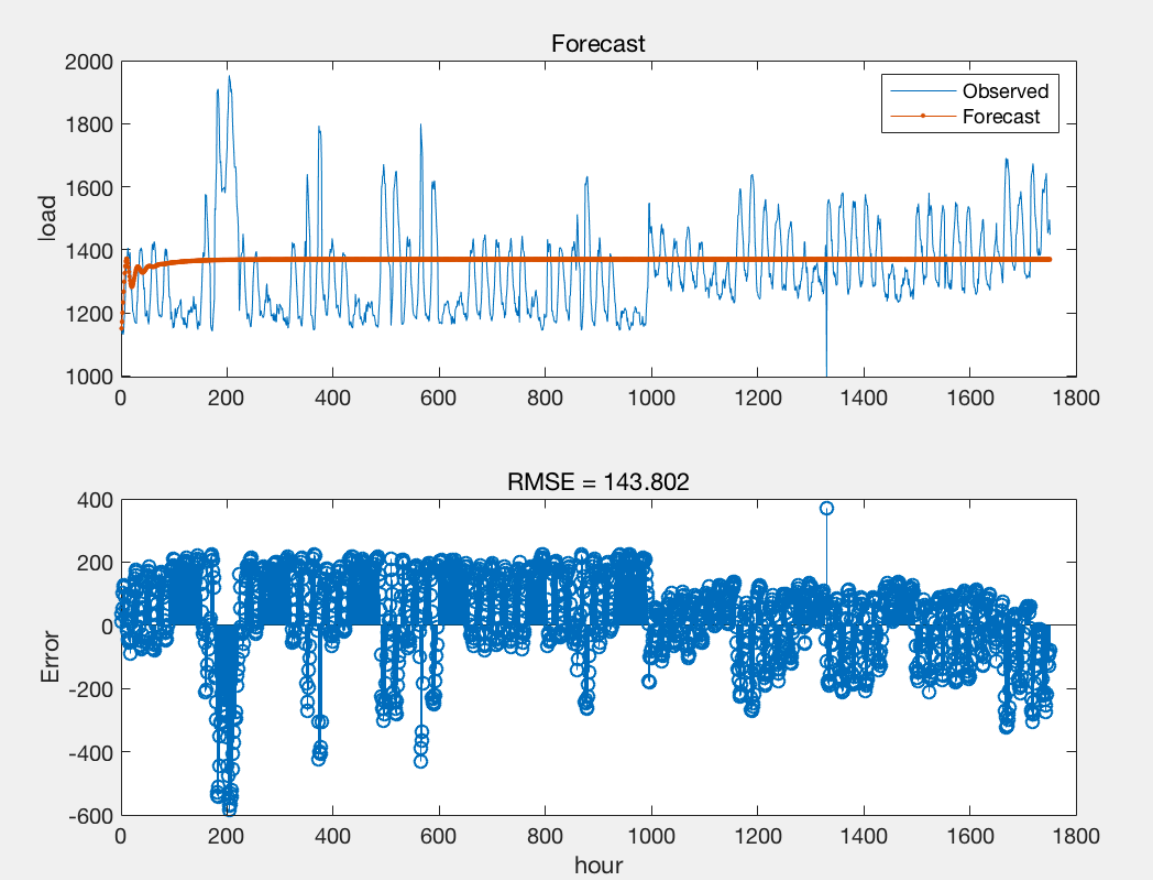

Figure 28. Predict result which update state with predict data

We can see that this result is not very good for the predict. The predicted result goes to the average value after the first time steps. The figure which we use this method is similar to different model parameters and time steps. So we choose to use the second one.

7.1.2 update network state with overserved values

We first compare the predict results with different initial learning rate between 0.001 to 0.01.

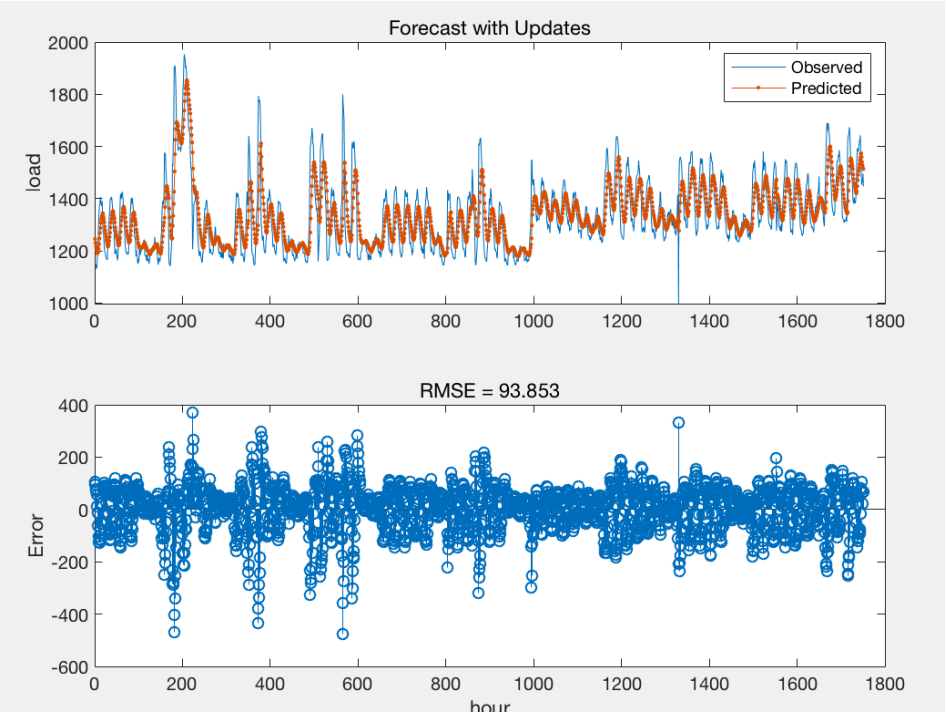

Figure 29. Predict result with learning rate 0.001

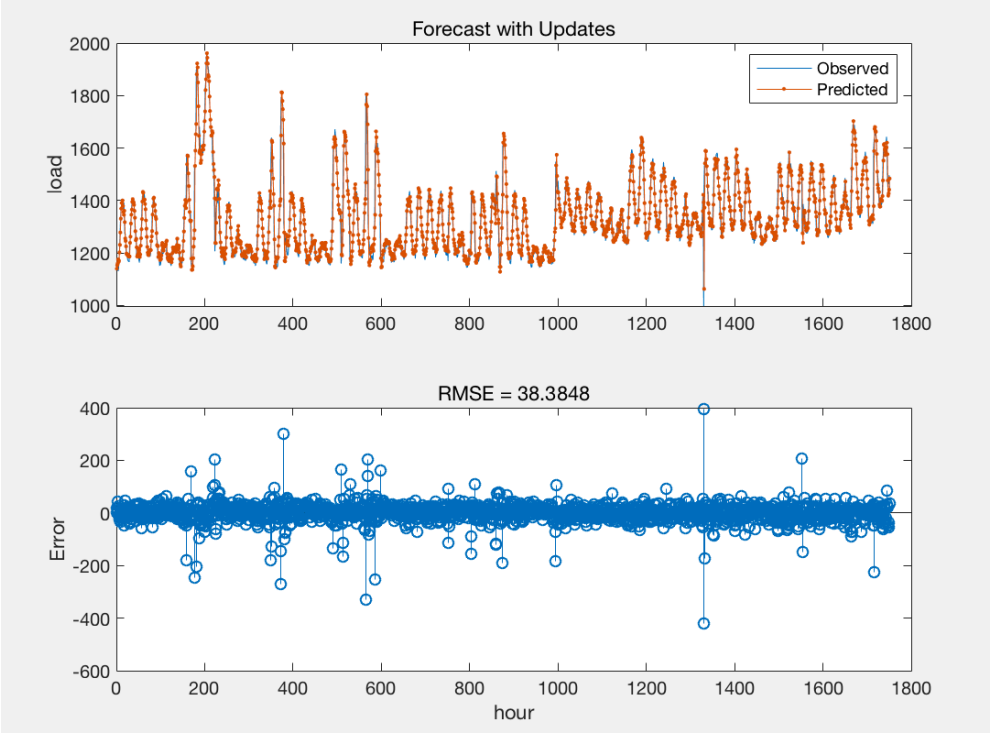

Figure 30. Predict result with learning rate 0.01

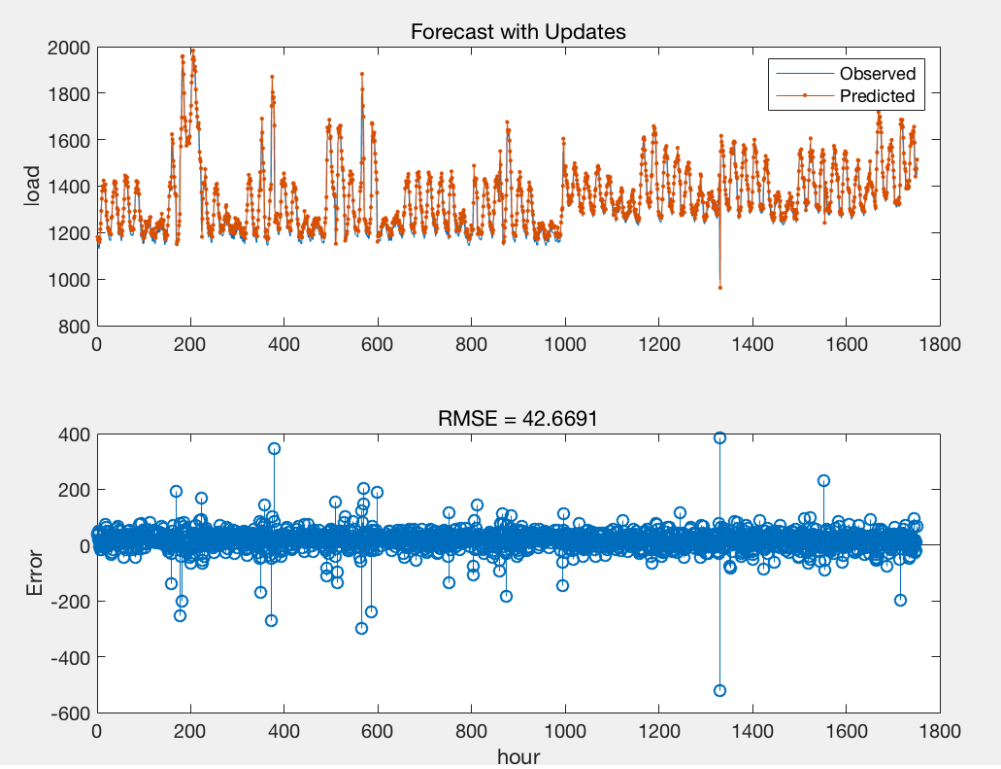

Figure 31. Predict result with learning rate 0.1

We can find that when the initial learning rate is 0.01, the predicted result is best. Thus we choose different time steps while setting the initial learning rate 0.01 as a default value.

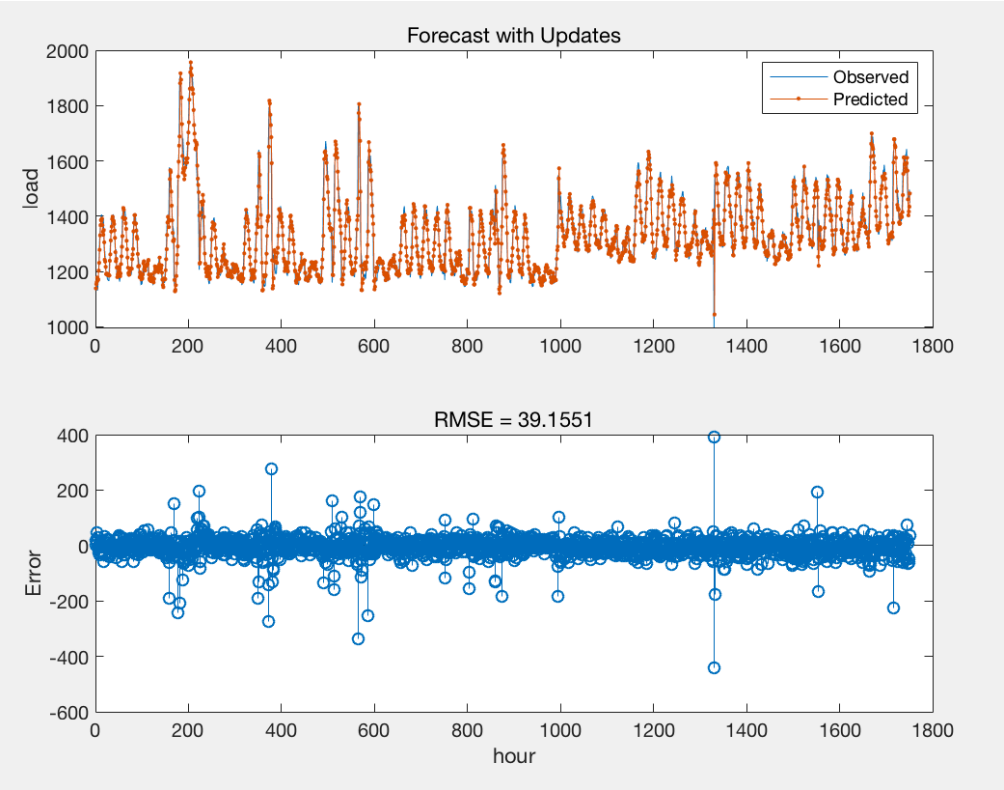

Figure 32. Predict result with time steps size is 50

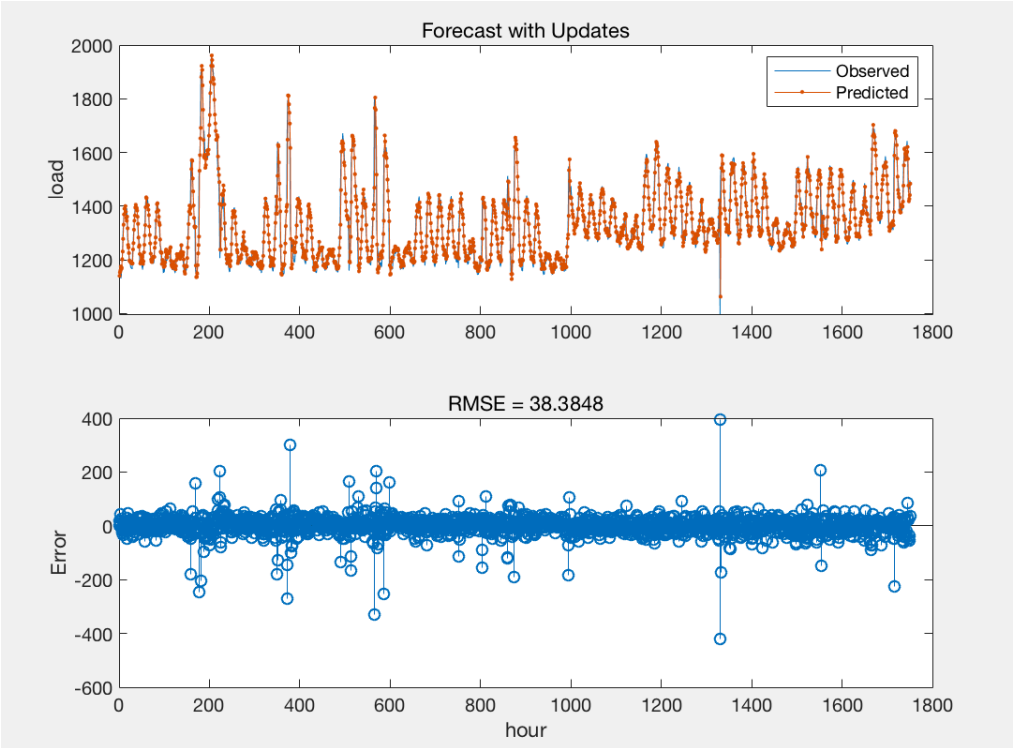

Figure 33. Predict result with time steps size is 150

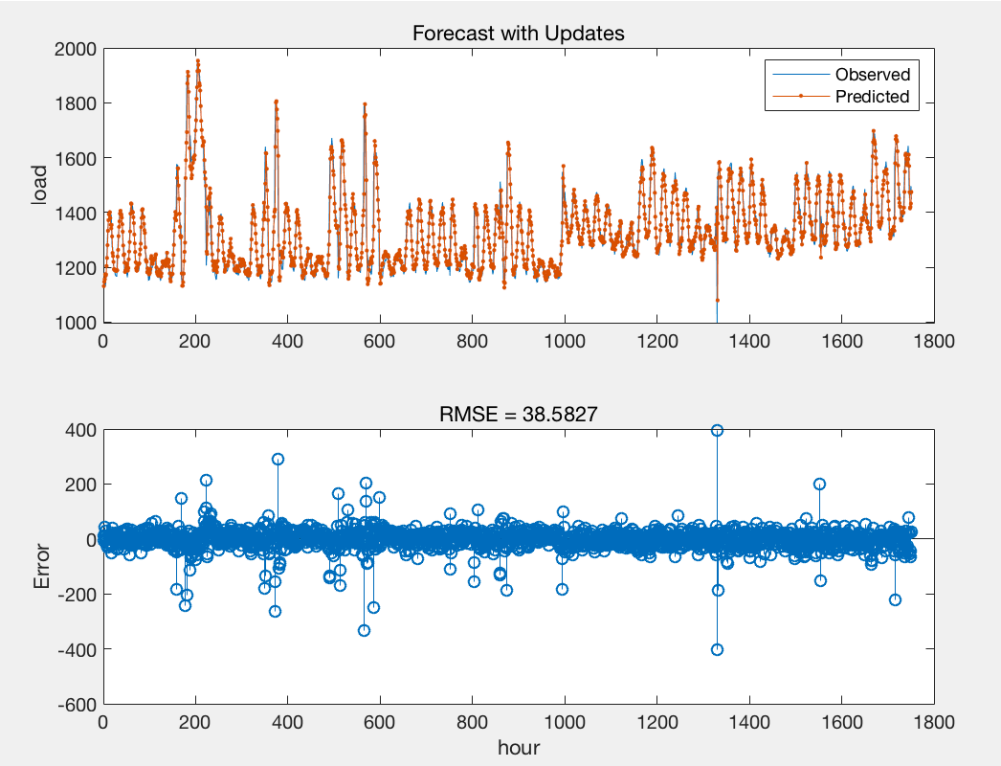

Figure 34. Predict result with time steps size is 300

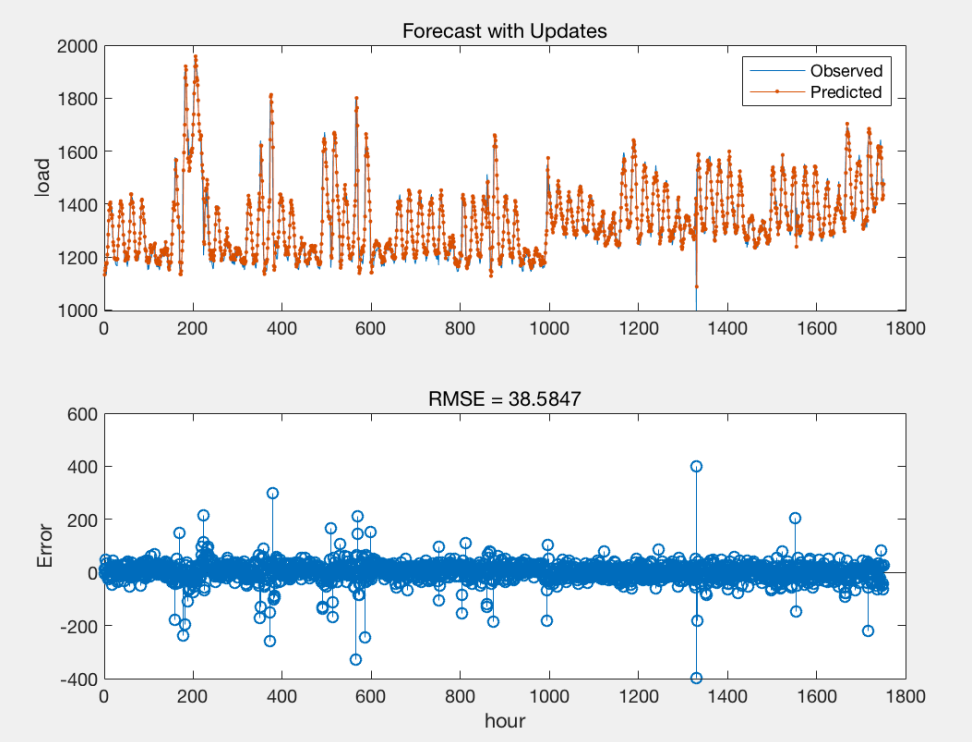

Figure 35 Predict result with time steps size is 600

7.2 Conclusion of the predict result

Initial learning rate	0.0002	0.002	0.02
RMSE	93.853	38.5827	38.5847
accuracy	5.8%	2.4%	2.4%

Table 1 summary of Predict result with initial learning rate

Number Hidden Units	150	300	600
RMSE	38.3848	38.5827	38.5647
accuracy	2.399%	2.411%	2.410%

Table 2 summary of Predict result with time steps size

From the figures of the different time-series above. We can find no matter how the time series of the data changes, the results are always stable, and there are no very sharp fluctuations. This means LSTM algorithm rare has gradient exploding or vanishing gradient. Moreover, the predicted results have been in a relatively good range, and there will be no cliff-like rise or fall changes.

8 Conclusion And Future Work

In this paper, we proposed a method to optimally tune the parameters of LSTM to forecast load community power demand. While most of the load data in the power system is always a long time-series data so it can be used in LSTM without too much preprocess. It has an outstanding performance to deal with the exploding and vanishing gradient problems in the long time-series data. We test the different parameters for training the models and try to find the best one to make load forecasting. After that, we test the model using different time steps to find the best performance to predict. We can find that the LSTM is a good algorithm while dealing with long time-series data. It has a very stable performance to make predictions in a long time steps. Compared with other neural network prediction methods, its superiority is unquestionable, especially in the case of hour-time in hundreds of units. That is to say; when we want to predict the load in the next few weeks, LSTM algorithm practicality is particularly significant. This results dramatically increases the applicability of LSTM.

From the paper, we know that we can get the result from the LSTM algorithm. However, we use the load data, which is a single character data. From that data, we can only build a one-dimensional model to training the data; this has a limited while we deal with the real data in life. We will try to put more features which can affect the results such as the country, the date, the season, and many other reasons. If we can put more features, the load forecasting should be more precise. We also need to make preprocessing data to clean the data. In this paper, we used all of the data we have. However, we should use preprocessing the data to select the right data, delete the redundancy data, and noise. These will make our model more robust.

REFERENCES

[1] Long short-term memory - Wikipedia. https://en.wikipedia.org/wiki/Long_short-term_memory

[2] Vossen, J., Feron, B., & Monti, A. (2018). Probabilistic Forecasting of Household Electrical Load Using Artificial Neural Networks. 2018 IEEE International Conference on Probabilistic Methods Applied to Power Systems (PMAPS). doi:10.1109/pmaps.2018.8440559

[3] Mitra, S., & Pal, S. K. (1995). Fuzzy multi-layer perceptron, inferencing and rule generation. *IEEE Transactions on Neural Networks*, *6*(1), 51-63.

[4] Tsang, W. (n.d.). Kernel methods in supervised and unsupervised learning. doi:10.14711/thesis-b805964

[5] Ruder, S. (2016). An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*.

[6] Johnson, R., & Zhang, T. (2013). Accelerating stochastic gradient descent using predictive variance reduction. In *Advances in neural information processing systems* (pp. 315-323).

[7] Neural Networks for Named Entity Recognition. (n.d.). Retrieved from http://nlp.stanford.edu/~socherr/pa4_ner.pdf

[8] Han, J., & Moraga, C. (1995). The influence of the sigmoid function parameters on the speed of backpropagation learning. Lecture Notes in Computer Science From Natural to Artificial Neural Computation,195-201. doi:10.1007/3-540-59497-3_175

[9] Werbos, P. J. (1990). Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, *78*(10), 1550-1560.

[10] Britz, D. (2015). Recurrent neural networks tutorial, part 1–introduction to rnns. 2017-12-23]. http://www. wildml. com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns.

[11] Pascanu, R., Mikolov, T., & Bengio, Y. (2013, February). On the difficulty of training recurrent neural networks. In *International conference on machine learning* (pp. 1310-1318).

[12] Yao, K., Peng, B., Zhang, Y., Yu, D., Zweig, G., & Shi, Y. (2014, December). Spoken language understanding using long short-term memory neural networks. In *2014 IEEE Spoken Language Technology Workshop (SLT)* (pp. 189-194). IEEE.

[13] BilstmLayer. (n.d.). Retrieved from https://www.mathworks.com/help/deeplearning/ug/long-short-term-memory-networks.html

[14] Qiu, X., Zhang, L., Ren, Y., Suganthan, P. N., & Amaratunga, G. (2014, December). Ensemble deep learning for regression and time series forecasting. In *2014 IEEE symposium on computational intelligence in ensemble learning (CIEL)* (pp. 1-6). IEEE.

[15] LstmLayer. (n.d.). Retrieved from https://www.mathworks.com/help/deeplearning/examples/time-series-forecasting-using-deep-learning.html

[16] SolverName. (n.d.). Retrieved from https://www.mathworks.com/help/deeplearning/ref/trainingoptions.html