Understanding & Developing Artificial Neural Network with Objective-C and Python

Deerfield Academy Advance Computer Science Research

Yongyang (Neil) Nie

Feb, 2016

Contents

1 Abstract:

Machine learning is a subset of artificial intelligence that provides computers with the ability to learn without being explicitly programmed. Machine learning focuses on the development of computer programs that can change when exposed to new data.

There are many methods and algorithms for learning algorithms, from Support

Vector Machine[1] to Artificial Neural Networks[2]. The purpose of them are similar, they can all be used to classify complex data, such as images and DNA samples.

2 Introduction:

There are two types of machine learning, one is supervised machine learning[3]. In this research, we will mainly focus on artificial neural networks and supervised machine learning. Supervised learning is the machine learning task of inferring a function from labeled training data. Each example is a pair consisting of an input object and a desired output value. By doing some calculation, the network can make rudimentary predictions and correct itself based on the training data to make more desired prediction.

You will discover that machine learning is beautiful and can be very simple. The paper will discuss the principles behind designing, building, and debugging an artificial neural network. All of the data are collect from a project I built that can recognize handwritten digits. The projects is written mainly in Objective-C and Python. You can find the resources on Github.

2.1 What's neural network

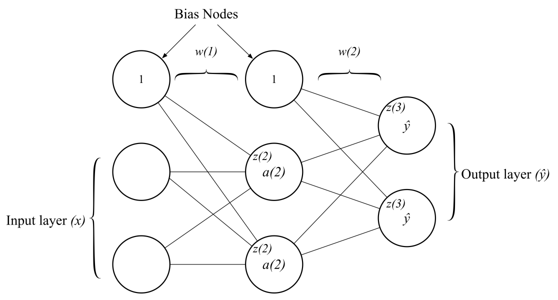

Figure 1

In the diagram above, you can find the most important components in a neural network. First of all, all neural networks have input and output layer(s). Many neural networks will have hidden layer(s), when there are plenty of hidden layers, we consider that as deep learning, which is out of our scope.

Secondly, there are synapses connecting every node to every node in the next layer. All synapses have weights, a floating point number that will govern how the network behaves. Since the input is often static, the most effective way to improve the prediction of the network is to modify the weights. Below, we will discuss extensively about how to adjust the weights and why.

Thirdly, there is one bias node connected to every layer in the network except the first layer. The bias will help the network to learn flexibly. Later on, we will discuss activation function, and bias will help us manipulate them, so the network can improve most efficiently.

2.2 Why neural networks

3 Forward Feed:

The neural network will begin be taking in some inputs and making some predictions based on the inputs and the weights between nodes. We can think of it like a one by two matrix. There will be a weight connecting every input layer node to every output layer node, therefore, there are six weights between the input layer and the second layer. The matrix calculation should yield us some result

$$\begin{bmatrix} a \end{bmatrix} \begin{bmatrix} w \end{bmatrix} = \begin{bmatrix} z \end{bmatrix}$$

If we give the matrices some names, call inputs x, weights $w_{(l)}$ and output $z_{(l)}$. In our case, l indicate the layer, for example, the first layer weights are $w_{(1)}$. $z_{(l)}$ represent the output matrix with layer l, in this case, the output z is the hidden layer output.

$$\mathrm{x}w_{(n)} = z_{(n)} \tag{1}$$

Afterwards, we have to apply an activation function[4]. The activation function that I used in this research and this paper will be the sigmoid function. This is a complete cycle, there is one more layer to go in order to yield a result. Note, in a multilayer neural network, this process will be repeated until we yield some output. In our case, we only have to do this twice. The activation function is shown as below.

$$\sigma(z) = \frac{1}{1+e^{-z}}$$
$$\mathrm{a}_{(2)} = f(z_{(2)}) \tag{2}$$

[1] https://en.wikipedia.org/wiki/Support_vector_machine
[2] https://en.wikipedia.org/wiki/Artificial_neural_network
[3] Mehryar Mohri, Afshin Rostamizadeh, Ameet Talwalkar (2012) *Foundations of Machine Learning*, The MIT Press ISBN 9780262018258.
[4] https://en.wikipedia.org/wiki/Activation_function

By applying the same process as before, this time, the inputs will be $a_{(2)}$ *then,* $z_{(3)}$ will be our final result. This process is known as forward feed. It's relatively straightforward.

$$z_{(3)} = \quad a_{(2)} w_{(2)}$$

After we yield some result, we can more on and look at how far off we are from the expected result. We need some methods to quantify this and so adjustments to the network to minimize error.

$$\sigma'(z) = \ \sigma(x)(1 - \sigma(z)) \tag{3}$$

4 Quantifying and Minimizing Cost:

Now the neural network can make calculation/predictions, however, the result is far from desired. In almost all learning algorithms, the input data cannot be altered, therefore the x term is constant in equation one. In order to change the output z the only option is to change the weights w.

First of all, we have to come up with ways to quantify the cost.

$$(3)\ \ C = \sum_j \frac{1}{2}(\hat{y} - y)^2 \tag{4}$$

C is the cost, which equals to the sum of all the differences between calculated result and actual result squared and times one half. We can take advantage of the equation derived above and substitute for some of the variable.

$$C = \ \sum_J \frac{1}{2}\left(y - f\left(f\left(xw_{(1)}\right) w_{(2)}\right)\right)^2 \tag{5}$$

Here we have it, a way to quantify the cost of the neural network. This function will be referred to as the cost function. Now, we have to solve the problem, how do we minimize C, will brute force work? It turns out, no, because in a three-node neural network we have to compute more than a million possible weights, which will be gruesome.

We can think of the equation above as a function of cost in terms of all possible weights. There will be one set of weights that will bring the cost to the lowest. Then, this becomes a minimization problem.

5 Gradient Descent:

The best way to minimization the cost is to use gradient descent, a very fast and classic way to solve problems like this. In fact, gradient descent is widely used in math, image process and machine learning.

Gradient descent is a first-order iterative optimization algorithm. To find a local minimum of a function using gradient descent, one takes steps proportional to the gradient (or of the approximate gradient) of the function at the current point.

Gradient descent is also known as steepest descent, or the method of steepest descent. The process can be seen as a ball rolling down a hill[5] and trying to find the lowest point. Note that actual physics doesn't apply here and we will define our own movement of the ball.

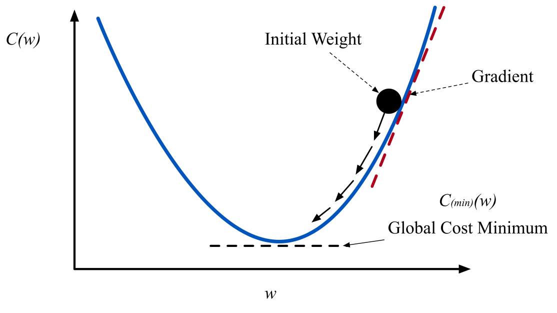

Figure 2

There are limitations to this method. First of all, what if we are stuck in a local minimum, our goal is to find the global minimum for the cost function. In another word, this method will not work properly for a non-convex function.

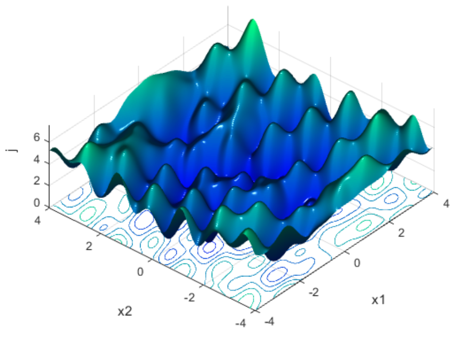

Figure 3

In fact, this problem is solved in equation (3), by squaring the difference in $(\hat{y} - y)^2$,

[5] https://iamtrask.github.io/2015/07/27/python-network-part2/ by Andrew

we are using the quadratic cost function, which is a convex function for any number of dimensions.

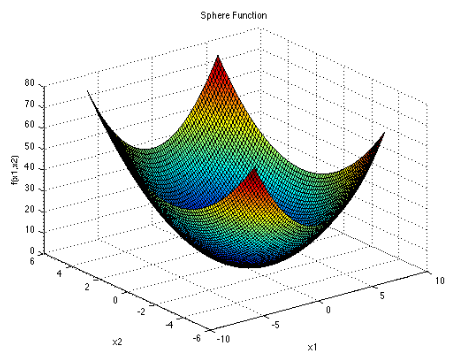

Figure 4

Thus, we can apply gradient descent without worrying about local minimums [6].

There are other types and variation of gradient descent as well. One of the most commonly used one is Stochastic gradient descent (SGD), also known as incremental gradient descent, is a stochastic approximation of the gradient descent optimization method for minimizing an objective function that is written as a sum of differentiable functions. In other words, SGD tries to find minima or maxima by iteration. [7]

6 Back propagation:

With gradient descent, we can create a set of routines that can help us to change the weights of the network to minimize the cost. This is call backprogation, which is the core of most of the sophisticated learning algorithms.

6.1 Back propagation overview

Phase 1: Propagation: Each propagation involves the following steps: [8]

1. Forward propagation of a training pattern's input through the neural network in order to generate the network's output value(s).

2. Backward propagation of the propagation's output activations through the neural network using the training pattern target in order to generate the deltas (the difference between the targeted and actual output values) of all output and hidden neurons.

Phase 2: Weight update: For each weight, the following steps must be followed:

1. The weight's output delta and input activation are multiplied to find the gradient of the weight.

2. A ratio (percentage) of the weight's gradient is subtracted from the weight.

6.2 Mathematics behind backpropagation

Backpropagation is based around four fundamental equations. Together, those equations give us a way of computing both the error $\delta^L$and the gradient cost of the function. In fact, the backpropagation equations are so rich that understanding them well requires considerable times and patience. [9]

6.2.1 An equation for the error in the output layer δ^L

$$\delta^L_J = \frac{\partial C}{\partial a^L_J}\sigma'(z^L_J) \qquad \text{(BP1)}$$

The first term on the right, $\frac{\partial J}{\partial a^L_J}$ measures how fast the cost is changing as a function of j^{th} output activation. Everything in (BP1) is easily calculated. We computed z^L_J while computing the behavior of the network. Depending on the cost function, in our case, the quadratic cost function is relatively easy to compute.

Equation (BP1) is a perfectly good expression, however, it's not matrix based, form that backpropagation desires. The fully matrix form becomes.

$$\delta^L = (a^l - y)(\sigma'(z^l) \qquad (6)$$

[6]Proof and definition of convex functions: http://mathworld.wolfram.com/ConvexFunction.html

[7]http://www.mit.edu/~dimitrib/Incremental_Survey_LIDS.pdf Dimitri P. Bertsekas Report LIDS - 2848

[8]The steps are referenced from Wikipedia who referenced: A Gentle Introduction to Backpropagation - An intuitive tutorial by Shashi Sathyanarayana The article contains pseudocode ("Training Wheels for Training Neural Networks") for implementing the algorithm.

[9]The equations were created by Michael A. Neilson "Neural Networks and Deep Learning", Determination Press, 2015

6.2.2 An equation for the error δ^l in terms of the error in the next layer, δ^{l+1} in particular:

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \bigodot \sigma'(z^l) \tag{BP2}$$

where $(w^{l+1})^T$ is the transpose of the weight matrix (w^{l+1}) for the *(l+1)th* layer. Suppose we know the error δ^{l+1} at the *(l+1)th* layer. When the transpose weight matrix is applied $(w^{l+1})^T$, we can think of this as moving the error backward through the network, giving us some sort of measure of the error at the output of the lth layer. Finally, we take the Hadamard product $\bigodot \sigma'(z^l)$.

By combining (BP1) with (BP2), the error δ^l for any layer in the network can by computed. We start by using (BP1) to compute δ^l, then apply Equation (BP2) to compute δ^{L-1}, then Equation (BP2) again to compute δ^{L-1}, and so on, all the way back through the network.

6.2.3 An equation for the rate of change of the cost with respect to any bias in the network

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \text{[10]} \tag{BP3}$$

This is the error δ_j^l is exactly equal to the rate of change $\frac{\partial C}{\partial b_j^l}$. This equation can be rewritten as

$$\frac{\partial C}{\partial b} = \delta$$

6.2.4 An equation for the rate of change of the cost with respect to any weights in the network

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l \text{[11]} \tag{BP4}$$

This equation can help us to compute the partial derivative $\frac{\partial C}{\partial w_{jk}^l}$ in terms of the quantity δ^l and a^{l-1}. It can be rewritten in a less index-heavy notation:

$$\frac{\partial C}{\partial w} = a_{in}\delta_{out} \tag{7}$$

6.3 Backpropagation with simple neural network example

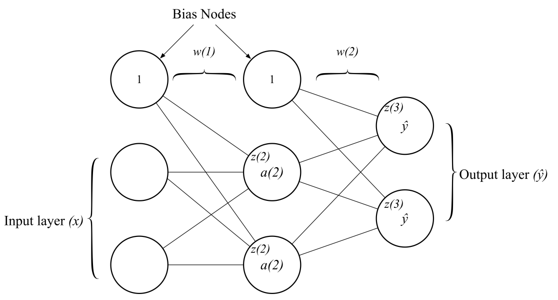

Figure 5

In a simple three-layer neural network, below are the four equations that we derive with the principle of gradient descend that will help us minimize the error.[12]

$$\delta_{(3)} = -(y - \hat{y})(\sigma'(z_{(3)}))$$

Calculate the δ for the third layer of the network.

$$\frac{\partial J}{\partial w_{(2)}} = (a^{(2)})^T \delta_{(3)}$$

Use the calculated result to help us modify the second layer of weights in the network.

$$\delta_{(2)} = \delta_{(3)} \left(w_{(2)}\right)^T \sigma'(z_{(2)})$$

Calculate the δ for the second layer of the network.

$$\frac{\partial J}{\partial w_{(1)}} = x^T \delta_{(2)}$$

Finally, we can modify the first layer of weights in the network. This process is often repeated until the accuracy of the network reaches a threshold. The backpropagation process above can be represented with this simple diagram of a neural network with one hidden layer.

[10]The equation was reference from Michael A. Neilson "Neural Networks and Deep Learning", Determination Press, 2015

[11]The equation was referenced from Michael A. Neilson "Neural Networks and Deep Learning", Determination Press, 2015

[12]The equation was referenced from Michael A. Neilson "Neural Networks and Deep Learning", Determination Press, 2015

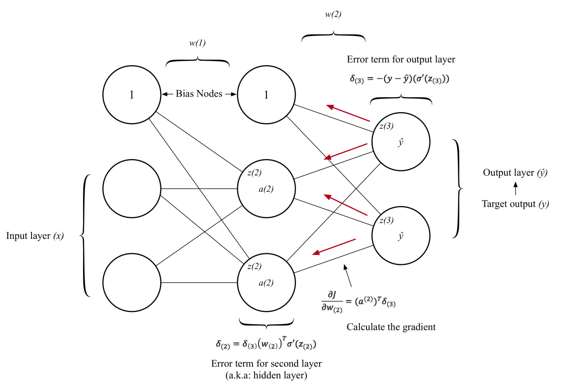
w(2)
w(1)
Error term for output layer
$\delta_{(3)} = -(y-\hat{y})(\sigma'(z_{(3)}))$
1
Bias Nodes
1
z(3)
$\hat{y}$
z(2)
a(2)
Output layer ($\hat{y}$)
Target output (y)
Input layer (x)
z(3)
$\hat{y}$
z(2)
a(2)
$\frac{\partial J}{\partial w_{(2)}} = (a^{(2)})^T\delta_{(3)}$
Calculate the gradient
$\delta_{(2)} = \delta_{(3)}(w_{(2)})^T\sigma'(z_{(2)})$
Error term for second layer
(a.k.a: hidden layer)

Figure 6

Here is a visualization of how individual output node is approaching the desired output.

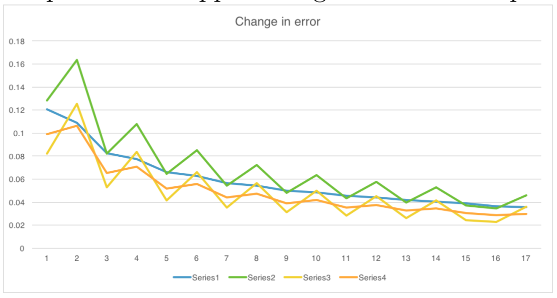

Figure 7

Line 0-3 are lowering as we train the network. Line 4, is approaching 1 as we keep train the network. This figure displayed that the error of the network C is being minimized.

Here is a graph of the correction of the neural network over times of training. Each time the network will train with 1000 out of 60,000 training data. After, The program will shuffle the training data and repeat the training process for 197 times, or until the network reaches 95% accuracy.

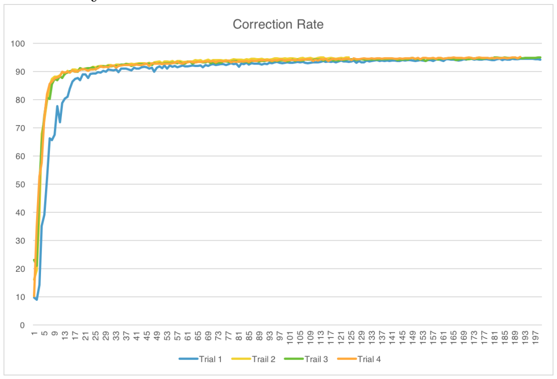

Figure 8

The neural network is trained four times with the same training and testing data. The results are similar and the network is steadily improving, thanks to gradient descent. There are many factors that contribute to a successful learning process. If these things are ignored or incorrect, the network will not be able to learn as well.

7 Improve neural network training result

Not all neural network behaves flawlessly. Generally, there are a few factors that will effect the training result of the network: hyperparameter, number of hidden nodes, human error, training data/testing data problem and more. Among them, hyperparameter, number of hidden nodes and be determined by estimation and trial and error. Huamn error can be eliminated by debugging. Training data and testing data problem is not easy to solve. That often contributes to the source of your data and the quality of your data, which are beyond the scope of this research. It's safe to assume that the training data and testing data that we are using are carefully chosen and considered.

7.1 Making good decision

In a pre-trained network, there are several critical hyperparameters and numbers that will help the network to improve. I mainly focused on learning rate and number of hidden nodes.

7.1.1 Hyperparameters

Learning rate is the hyperparameter that we will focus on. To make gradient descent work correctly, we need to choose the learning rate r to be small enough that Equation (9) is a good approximation. If we don't, we might end up with $\Delta C>0$, which will take us to the opposite of minimizing. Meanwhile, r can't be too small, since that will make the changes Δv tiny, and thus the gradient descent algorithm will work very slowly. In practical implementations, r is often varied so that Equation (9) remains a good approximation, but the algorithm isn't too slow. [13]

This is a graph of the training result with a different learning rate. The batch size is 1000 and the training is repeated 100 times.

[13] Michael A. Nielsen, "Neural Networks and Deep Learning", Determination Press, 2015. Chapter 1 Using neural nets to recognize handwritten digits

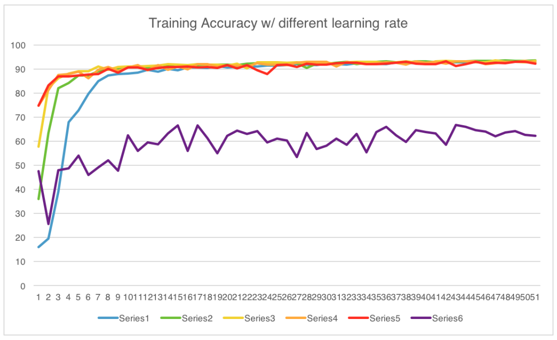

Figure 9

The range of the learning rate that I chose is from [0.1, 5]. When the learning rate is relatively low, the network began at a lower accuracy comparing to higher training rates. However, especially when the learning rate is 0.9, the training result became unstable, and the graph starts to oscillate.

On the other hand, when we set the training rate to 5.0, the network doesn't improve after 40%-50%. Then general rule of thumb for choosing the learning rate is somewhere between [0.1-1].

With this observation in mind, we might be able to alter the learning rate as the network progress. The training rate should we high in the beginning to quickly bring us to the desired place. Then, we can lower the rate so the network learning result will not have oscillation.

7.1.2 Hidden nodes

In a multilayer neural network, you will likely encounter the problem of how many hidden layers and hidden nodes should the network have. Seemingly simple, but complex question will help you improve the learning rate of the network. In this research, we will only focus on one hidden layer. Multiple hidden layer is known as deep learning, which is out of our scope.

It's difficult to form a good network topology just from the number of inputs and outputs. It depends critically on the number of training examples and the complexity of the classification you are trying to learn. There are problems with one input and one output that require millions of hidden units, and problems with a million inputs and a million outputs that require only one hidden unit, or none at all.

Below if a graph of a neural network with different number of hidden nodes. There is a healthy range of number of hidden nodes. If there is one hidden node, the accuracy will not be more than 25% and also defeat the purpose of the neural network. If there are too many nodes, calculation becomes gruesome and the accuracy is similar to lower number of nodes. Therefore, the number of hidden nodes highly depend your data, network, situation and needs.

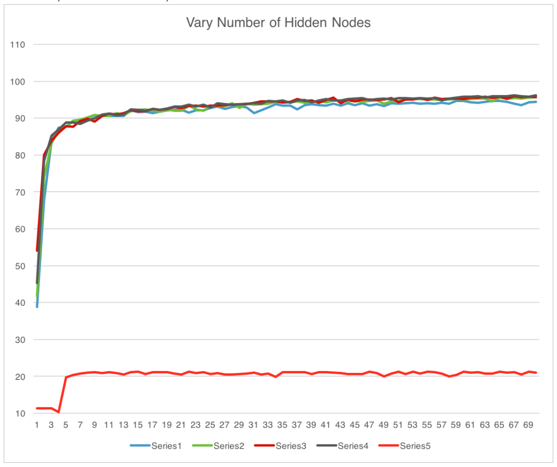

Figure 10

7.2 Methods to improve training result

There are many ways to improve the network's performance. In fact, gradient descent is a great optimization method to speed up the learning process. Here are some common ways to improve the accuracy.

- Cross-entropy
- Overfitting and regularization
- Altering the hyperparameters

In section 7.1.1, I stated that the learning rate will effect how the network learns in the beginning the fluctuation of the learning curve. This observation gives us the possibility to change the learning rate as the network learns. In the beginning, we will set the learning rate 0.9, then slowly lower the number as the network improves. This concept will seem intuitive in that you want the network slow down the learning rate when it's close to the desired result. Otherwise, a high learning rate will likely create some fluctuation like we have seen figure 2.

Below is a graph of the accuracy of a neural network with 0.8 learning rate. The network is trained with batches of 1000 data and shuffled before the next training. Training is stopped after the accuracy on test data reaches 95%. All of its hyparameters were static during training.

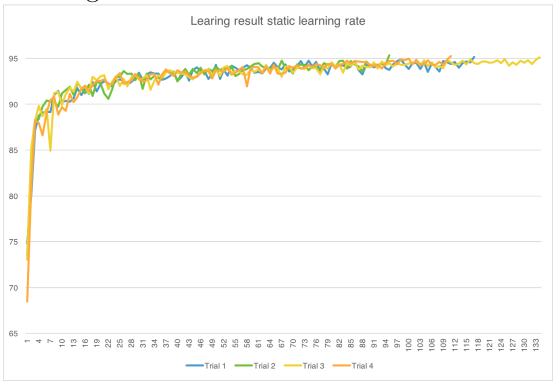

Figure 11

The training time is mediocre comparing to a well optimized network. The average training time is around 360 second and the number of epoch is around 110. In another word, the network took 6 minutes and 12,000 data to reach 95% accuracy.

This is a accuracy graph of a network with 0.8 initial learning rate. The rate is lowered when the accuracy reaches 93%. The new accuracy is calculated by

$$r_{(new)} = r_{(old)} * 0.9 \tag{8}$$

Everything else is the same as the training process above. The network is trained with batches of 1000 data and shuffled before the next training. Training is stopped after the accuracy on test data reaches 95%.

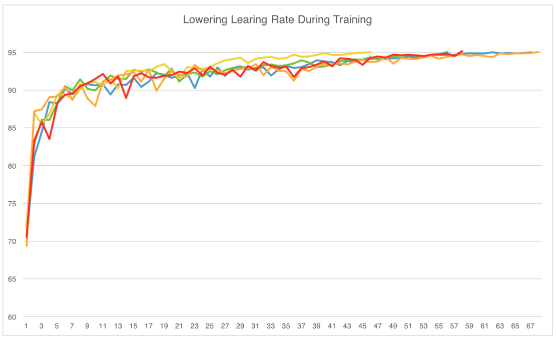

Figure 12

Clearly, the network took much less time and epoch to reach 95%. If we calculate a ln LBF (line of best fit) for figure (), the coefficient of the function will be greater. Therefore, the grow curve will larger.

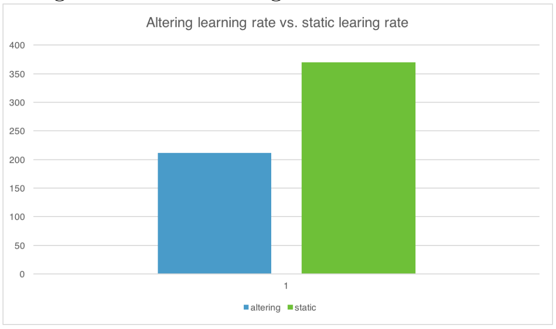

Figure 13

The training time difference between dynamic and static learning rate is more than 300 seconds.

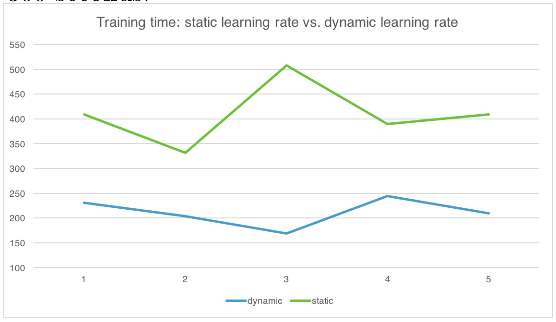

Figure 14