Neural Networks 101 (2018) 68–78

Contents lists available at ScienceDirect

Neural Networks

journal homepage: www.elsevier.com/locate/neunet

Effective neural network training with adaptive learning rate based on training loss

Tomoumi Takase *, Satoshi Oyama, Masahito Kurihara

Graduate School of Information Science and Technology, Hokkaido University, Kita 14 Nishi 9 Kita-ku, Sapporo, Japan

ARTICLE INFO

Article history:
Received 14 April 2017
Received in revised form 17 December 2017
Accepted 29 January 2018
Available online 13 February 2018

Keywords:
Multilayer perceptron
Deep learning
Neural network training
Stochastic gradient descent
Learning rate
Beam search

ABSTRACT

A method that uses an adaptive learning rate is presented for training neural networks. Unlike most conventional updating methods in which the learning rate gradually decreases during training, the proposed method increases or decreases the learning rate adaptively so that the training loss (the sum of cross-entropy losses for all training samples) decreases as much as possible. It thus provides a wider search range for solutions and thus a lower test error rate. The experiments with some well-known datasets to train a multilayer perceptron show that the proposed method is effective for obtaining a better test accuracy under certain conditions.

1. Introduction

Deep learning has recently demonstrated excellent performance for various image classification (Krizhevsky, Sutskever, & Hinton, 2012; Shi, Ye, & Wu, 2016; Wu & Gu, 2015) and speech recognition tasks (Fayek, Lech, & Cavedon, 2017; Hinton et al., 2012; Sainath et al., 2015). This success is mainly due to the development of improved parameter updating methods. AdaGrad (Duchi, Hazen, & Singer, 2011), RMSprop (Tieleman & Hinton, 2012), AdaDelta (Zeiler, 2012), and Adam (Kingma & Ba, 2015), which are updating methods based on stochastic gradient descent (SGD), are now widely used, and selection of a suitable updating method for each task can lead to a better performance.

Users of these updating methods need to define the initial parameters, including the initial learning rate. Defining this rate is especially important because an inappropriate learning rate can lead to poor local solutions where the value of the loss function is no better than other local solutions. Thus we should say that a major disadvantage of these methods is that they have sensitive hyper parameters which are difficult to tune appropriately.

One method without any sensitive hyper parameters is the step-size control method of Daniel, Taylor, and Nowozin (2016), in which the step-size for the learning rate is automatically controlled by reinforcement learning (Sutton & Barto, 1998) independent of its initial setting. Another such method is the LOG-BP algorithm of Kanada (2016), which exponentially reduces the learning rate by combining back propagation with the genetic algorithm.

A straightforward approach to adjusting the learning rate is to multiply it by a certain constant, such as 0.1, every fixed number of training epochs, such as 100 epochs. This approach is widely used to improve test accuracy. However, this approach is inflexible in the sense that the learning rate must be fixed to a single value until the next setting time.

In this paper, we present a more flexible method for automatically adjusting the learning rate during training by either increasing or decreasing its value adaptively based on a tree search for minimizing the training loss. Unlike the straightforward approach, our method performs the trainings independently in parallel with several learning rates during each epoch, choosing as the actual learning rate the one that has resulted in the smallest training loss (the sum of cross-entropy losses for all training samples). This is regarded as an optimization process. However, we find dynamic programming and reinforcement learning inappropriate for our task because (1) the training is one-way, (2) the training loss cannot be analytically calculated, and (3) the training for each epoch takes much time. To overcome this problem, we have developed an efficient search algorithm based on breadth-first beam search. In the sequel, our method will be referred as ALR technique (Adaptable Learning Rate Tree algorithm).

In Section 2, we analyze the behavior of the learning rate for various parameter updating methods. In Section 3, we describe ALR technique. In Section 4, we report the experimental results for both the proposed and conventional methods. In Sections 5 and 6, we discuss the effects of three main parameters. In Section 7, we summarize our work and discuss some future works.

* Corresponding author.
E-mail addresses: takase_t@complex.ist.hokudai.ac.jp (T. Takase), oyama@ist.hokudai.ac.jp (S. Oyama), kurihara@ist.hokudai.ac.jp (M. Kurihara).

https://doi.org/10.1016/j.neunet.2018.01.016
0893-6080/

2. Learning rate

Since SGD is the base of many updating methods, we first describe an updating method based on SGD. Many studies related to SGD have been conducted (Breuel, 2015; Hardt, Recht, & Singer, 2015; Mandt, Hoffman, & Blei, 2016; Neyshabur, Salakhutdinov, & Srebro, 2015).

In SGD, parameter updating is performed for each sample or for each mini batch:

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla_{\theta_t} E\left(\theta_t; x^{(i)}; y^{(i)}\right), \tag{1}$$

where θ are the weights (and biases), which are the neural network parameters, $x^{(i)}$ is input with training sample i, $y^{(i)}$ is the label, E is the loss function, and η is the learning rate.

Increasing η can widen a range of search, but too large η makes the convergence to a solution difficult. An effective way to avoid this problem is to reduce the learning rate during training. For example, AdaGrad replaces η in Eq. (1) with η_t defined as follows.

$$r_{t+1} = r_t + \nabla_{\theta_t} E \circ \nabla_{\theta_t} E \tag{2}$$

$$\eta_{t+1} = \frac{\eta_0}{\sqrt{r_{t+1}} + \varepsilon}, \tag{3}$$

where $\circ$ in Eq. (2) means the Hadamard (element-wise) product, and ε in Eq. (3) is a small constant used for stability. Since r_t is increasing, η_t is decreasing.

Most updating methods, such as Adam, are based on AdaGrad, and their updating equations were designed so that the learning rate decreases during training. The search range for a solution gradually narrows, and search for a better solution becomes difficult. In contrast, if the learning rate is increased during training so that the search range remains wide, poor local solutions can be easily avoided, but convergence becomes difficult. A combination of reducing and increasing the learning rate should thus be an effective way to improve training.

In ALR technique, the learning rate is increased or decreased so that the training loss is minimized, meaning that η in Eq. (1) is changed for each epoch on the basis of the loss function:

$$\theta_{t+1} = \theta_t - \eta_t \cdot \nabla_{\theta_t} E\left(\theta_t; x^{(i)}; y^{(i)}\right). \tag{4}$$

Unlike AdaGrad, ALR computes η_t common to all weights for efficiency reasons.

ALR modifies the learning rate on the basis of training loss, but generally, a decrease in training loss can lead to over-fitting to training data. However, the over-fitting can be restrained by using a technique such as weight upper limit (Srebro & Shraibman, 2005) or dropout (Srivastava, Hinton, Krizhevsky, Sutskever, & Salakhutdinov, 2014).

3. ALR technique

3.1. State transition

In this section, we describe ALR technique, which adjusts the learning rate on the basis of the loss function. As mentioned in Section 1, training is independently performed with several learning rates during each training epoch, and the rate that resulted in the smallest training loss is used as the actual learning rate at each epoch. Because the training loss is the value of the loss function, ALR technique indirectly uses the shape of the loss function. While a method for theoretically finding the minimum of a loss function without using its shape has been proposed (Song, Schwing, Zemel, & Urtasun, 2016), the range of application is limited because it depends on a specific loss function. Our method does not depend on a specific loss function.

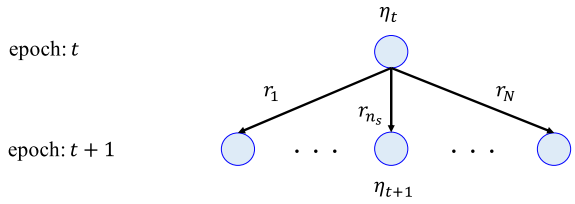

Fig. 1. State transition from epoch t to epoch $t+1$. N: number of branches; r_n: scale factor for each branch; η_t: learning rate at epoch t.

We use a tree structure to represent state transition during training. The state transition from epoch t to epoch $t+1$ is illustrated in Fig. 1. Each node represents the state (the learning rate) at an epoch in the training. The parent node at epoch t has N branches, and one of the different scale factors $r_1, \ldots, r_N$ (fixed a priori, common to every node) is assigned to each branch. The learning rate at epoch $t+1$ is obtained by multiplying the learning rate at epoch t by the scale factor r_n, if the nth child node was chosen at epoch t. By r_{n_t}, we denote the scale factor chosen in the transition from epoch t to $t+1$, where $n_t \in 1, 2, \ldots, N$. The relationship between η_t and η_{t+1} is given by

$$\eta_{t+1} = \eta_t \cdot r_{n_t}. \tag{5}$$

The repeated use of Eq. (5) leads to the learning rate at an arbitrary epoch s as follows:

$$\eta_s = \eta_0 \prod_{t=1}^{s-1} r_{n_t}, \tag{6}$$

where η_0 is the initial learning rate.

Actually, ALR combines multiple state transitions, as shown in Fig. 2. A search is performed using the tree structure. To perform it efficiently, ALR uses a breadth-first beam search, as described in Sections 3.2 and 3.3.

3.2. Parameters

ALR has three main parameters: the number of branches, the set of scale factors, and the beam size. The number of branches is N for each node, and the set of scale factors for each node is represented as $\{r_1, \ldots, r_N\}$. The beam size M represents the bounded breadth of the breadth-first search. If the number of nodes k at the same epoch exceeds M, the $(k - M)$ worst nodes (in terms of training loss) are eliminated.

The user must fix these parameters (common to all nodes and epochs) before training. Here, the number of branches and the scale factor are common to all nodes, and the beam size is common to each epoch. The effects of these main parameters and a minor parameter η_0 will be discussed in Sections 5 and 4.4, respectively.

3.3. Procedure

We explain the state transition procedure, using the example tree structure shown in Fig. 2, where the number of branches is $N = 3$, the scale factors are {2.0, 1.0, 0.5}, and the beam size is $M = 4$. The number inside each node represents the ranking of the training loss at each epoch (the smaller the training loss is, the smaller the number is). The state transitions at each epoch are as follows.

epoch 1 → 2: In node A, one-epoch trainings are independently performed for the three scale factors. The branch that produces the smallest training loss is chosen, and the state changes to node B. The two nodes other than node B are stored for the next step.
epoch 2 → 3: Generated as candidates for the next transition are 9 nodes, for each of which one-epoch training is independently