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 $\theta$ 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 $\eta$ is the learning rate.

Increasing $\eta$ can widen a range of search, but too large $\eta$ 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 $\eta$ in Eq. (1) with $\eta_t$ 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 $\varepsilon$ in Eq. (3) is a small constant used for stability. Since $r_t$ is increasing, $\eta_t$ 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 $\eta$ 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 $\eta_t$ 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.

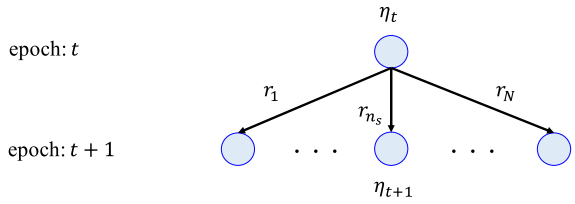



**Fig. 1.** State transition from epoch $t$ to epoch $t+1$. $N$: number of branches; $r_n$: scale factor for each branch; $\eta_t$: learning rate at epoch $t$.

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 $n$th 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 $\eta_t$ and $\eta_{t+1}$ 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 $\eta_0$ 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 $\eta_0$ 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

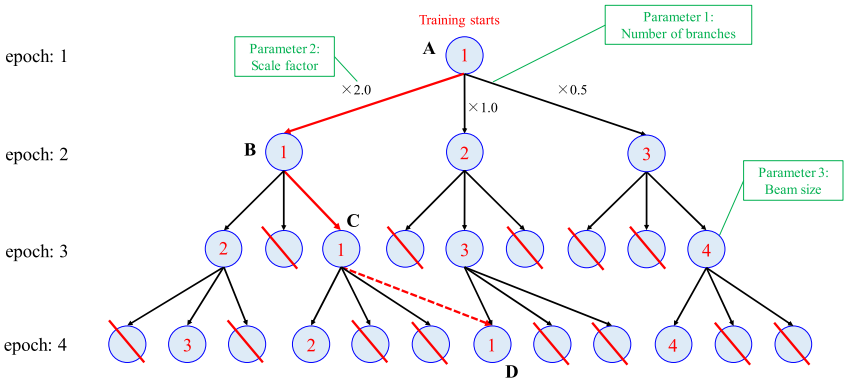



**Fig. 2.** Example state transition from epoch 1 to epoch 4. Parameter 1: the number of branches is 3. Parameter 2: the scale factors are 2.0, 1.0, or 0.5. Parameter 3: the beam size is 4. The numbers labeled on the nodes represent the ranking of training loss at each epoch. In the breadth-first beam search, the state moves to the child node labeled 1.

performed. As many as $M(=4)$ nodes are stored in the ascending order of training loss, and the other 5 nodes are eliminated. The state changes to node C, which produces the smallest training loss. **epoch 3 → 4:** Trainings are performed as in the previous epoch. The state changes to the best node D, although not directly connected to node C. This transition is possible because the nodes other than the best node have been stored.

A sketch of the algorithm used in ALR technique is given in Algorithm 1. A characteristic point is that several learning rates and weights are stored in ascending order of training loss. Training continues until it converges, but in practice, stopping at an earlier epoch might result in a better result.

**Algorithm 1** ALR technique

**Initialize**: beam size: $M$;
number of branches: $N$;
scale factors: $\{r_n\}_{n=1}^{N}$;
learning rate: $\{\eta_m\}_{m=1}^{M}$;
weights and biases: $\theta$, $\{\theta_m\}_{m=1}^{M}$;
number of batches for each epoch: $B$;
**Input**: training data: $\{x^{(b)}\}_{b=1}^{B}$, $\{y^{(b)}\}_{b=1}^{B}$;
$t \leftarrow 0$
**repeat**
  **for** $m = 1, 2, \cdots, M$ **do**
    **for** $n = 1, 2, \cdots, N$ **do**
      $\eta_{m,n} \leftarrow \eta_m \cdot r_n$
      $\theta_{m,n} \leftarrow \theta_m$
      **for** $b = 1, 2, \cdots, B$ **do**
        $E_{m,n} \leftarrow E(\theta_{m,n}; x^{(b)}; y^{(b)})$
        $\theta_{m,n} \leftarrow \theta_{m,n} - \eta_{m,n} \cdot \nabla_\theta E_{m,n}$
      **end for**
    **end for**
  **end for**
  $\theta \leftarrow \theta_{m_t,n_t}$ **where** $m_t, n_t \leftarrow \underset{m,n}{argmin}\{E\}$
  Sort $\{(\eta_{m,n}, \theta_{m,n}) \mid 1 \leqq m \leqq M, 1 \leqq n \leqq N\}$
  in ascending order of $\{E_{m,n}\}$ and
  Save the first $M$ pairs as $(\eta_1, \theta_1), \cdots, (\eta_M, \theta_M)$.
  $t \leftarrow t + 1$
**until** converged

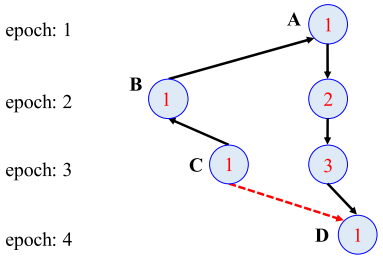



**Fig. 3.** A diagram to explain an interpretation for a transition in breadth-first beam search.

Compared to the conventional method, ALR is time-consuming when higher number of branches and beam size are used. For example, if the number of branches is 3 and the beam size is 20, this method will train ∼60 times more models than the conventional method. However, in most cases, the computation time of ALR is not a big problem for multilayer perceptron (MLP) because it has a relatively small structure.

Clearly, we can expect that parallel computation for trainings at each epoch will effectively reduce computation time as in Huang et al. (2013), where Huang et al. used parallel computation to reduce the computational cost of a recurrent neural network.

### 3.4. Breadth-first beam search

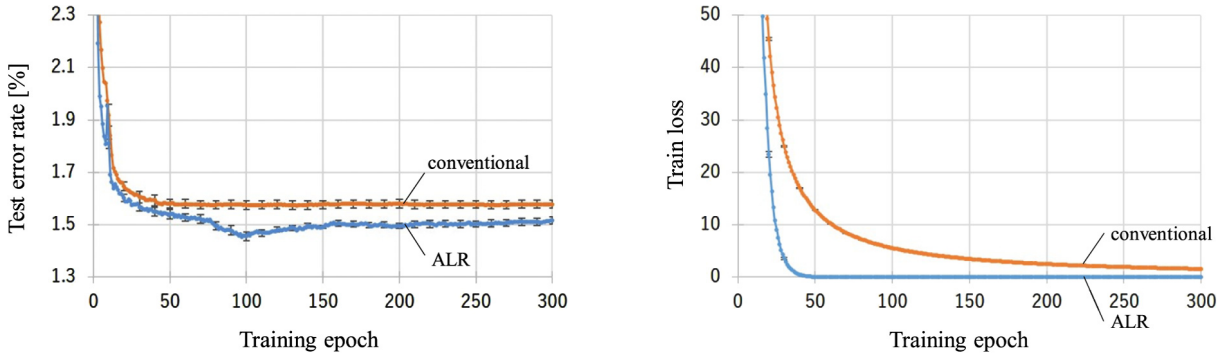
As discussed in the previous section, at the epoch 3 → 4, the breadth-first beam search enabled state transition to node D, which does have a direct connection to node C. A suitable diagram to explain this transition is shown in Fig. 3. As shown in the figure, we can interpret this transition as a transition via the route C → B → A → · · · → D. This is especially important in terms of searching for solutions.

Because the weights are updated so that the training loss tends to decrease, a poor local solution tends to be obtained, especially at the beginning of the training. The breadth-first beam search enables a poor local solution to be avoided and a better solution to be obtained. We will discuss the effect of beam size in Section 5.3.

## 4. Main experiments

### 4.1. Experimental conditions

We performed experiments using the MNIST dataset (LeCun, Bottou, Bengio, & Haffner, 1998) and compared the result by ALR technique and that by the standard SGD (the conventional method) as a baseline. The MNIST dataset consists of hand-written digital images and contains 60,000 training data and 10,000 test data. The




(a) Transition in test error rate.

(b) Transition in training loss.

**Fig. 4.** Results of experiments using MNIST dataset. Mean values for several trials with different initial weights are shown. Error bars represent standard error.

**Table 1**
Structure of MLP with one hidden layer. ReLU was used for the hidden layer and the softmax function was used for the output layer.

| Layer type | Units |
|---|---|
| Input | $28 \times 28$ |
| Hidden ReLU | 1000 |
| Output softmax | 10 |

**Table 2**
Parameters for the experiments in Section 4.1. The same initial learning rate, batch size, and number of epochs were used for both the proposed and conventional methods.

| Parameter | Value |
|---|---|
| Initial learning rate | 0.5 |
| Batch size | 100 |
| Number of epochs | 300 |
| Number of branches | 3 |
| Scale factors | {2.0, 1.0, 0.5} |
| Beam size | 20 |

image size is $28 \times 28$ pixels, and the input values are the intensity, from 0 to 255. Images are divided into 10 classes, from 0 to 9.

We trained a MLP in this experiment. Because MLP is the base of other neural networks, improving its performance would be an important achievement, as evidenced by the many MLP studies that have been conducted (Egmont-Petersen, Talmon, Hasman, & Ambergen, 1998; Park & Jo, 2016; Tang, Deng, & Huang, 2015). The structure of the MLP used is given in Table 1. An MLP with only one hidden layer was used as a model. A rectified linear unit (ReLU) (Glorot, Bordes, & Bengio, 2011) was used as the activation function of the hidden layer, and the softmax function was used as the activation function of the output layer. The parameters used are listed in Table 2.

When the initial learning rate is $\eta_0$ and the scale factor is $r$, the learning rate at the epoch $t+1$ is described by $\eta_0 \cdot r^t$. If each element of scale factors is described by the format of $a^k (a, k \in \mathbb{R})$, where the exponent $k$ can allow a bigger learning rate space than the base $a$, the possible learning rates are limited to specific values, $\eta_0 \cdot a^n (n \in \mathbb{R})$. This is convenient for a comparative experiment with the conventional method as performed in the next section. By setting $a$ to 2.0 and the number of branches to 3, we used {2.0, 1.0, 0.5} for the set of scale factors and 0.5 for the initial learning rate, when the possible learning rates are given by $0.5 \cdot 2^n$.

In actual use, the scale factor need not be limited to these values; the only requirement is that the set of values contains both a value larger than 1.0 and a value smaller than 1.0 so that the learning rate can increase and decrease. A recommendation for setting the scale factor is described in Section 5.2.

The beam size was set to 20. A Glorot's uniform distribution (Glorot & Bengio, 2010; LeCun, Bottou, Orr, & Müller, 1998) was used to initialize the weights in the neural network. Because the results are affected by the initialization, we performed several trials with the initialization done using different random seeds. The test error rate was calculated for each epoch.

We did not use validation data because there is not enough data in the MNIST dataset to provide validation data, so using validation data would degrade test accuracy. In actual use, validation data should be used to find better parameters.

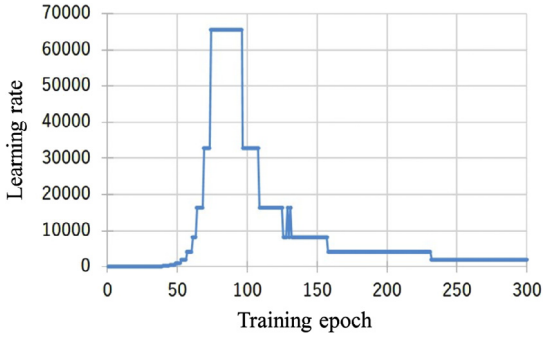



**Fig. 5.** Transition in learning rate during training for one trial with initial learning rate of 0.5. Highest learning rate achieved was 65 536.

### 4.2. Experimental results

As shown in Fig. 4(a) and (b), respectively, ALR had a lower test error rate and a much smaller training loss than the conventional method. The small training loss is the result of choosing an update with the smallest loss among several updates using different learning rates.

The transition in the learning rate during training (obtained for one trial) is plotted in Fig. 5. The rate increased as training proceeded. At epoch 73, it reached 65,536, which is a very high value. It began decreasing at epoch 96, when the test error rate was the lowest. To stop the training when the test error rate is the lowest, the training should be performed using the transition of the learning rate as well as using the validation error.

Note that the learning rate was automatically adjusted so that it achieved a very high value and that the lowest test error rate was obtained at the point where the rate reached the highest value. To investigate this finding in detail, we performed an experiment using the conventional method with a very high initial learning rate. The learning rate changed from 0.5 to 65 536 in ALR, so in this experiment we performed each training by setting all learning rates in the training with ALR to the initial learning rate in the conventional method.

**Table 3**
Comparison of lowest test error rates (Left: using ALR, Right: using conventional method).

| Learning rate | Lowest error rate (%) | Learning rate | Lowest error rate (%) | Learning rate | Lowest error rate (%) |
|---|---|---|---|---|---|
| Fig. 5 | 1.454 | 0.5 | 1.573 | 256 | 90.20 |
| | | 1.0 | 1.573 | 512 | 90.20 |
| | | 2.0 | 3.810 | 1 024 | 90.20 |
| | | 4.0 | 78.85 | 2 048 | 90.20 |
| | | 8.0 | 88.65 | 4 096 | 90.20 |
| | | 16 | 90.42 | 8 192 | 90.20 |
| | | 32 | 90.42 | 16 384 | 90.20 |
| | | 64 | 90.20 | 32 768 | 90.20 |
| | | 128 | 90.20 | 65 536 | 90.20 |

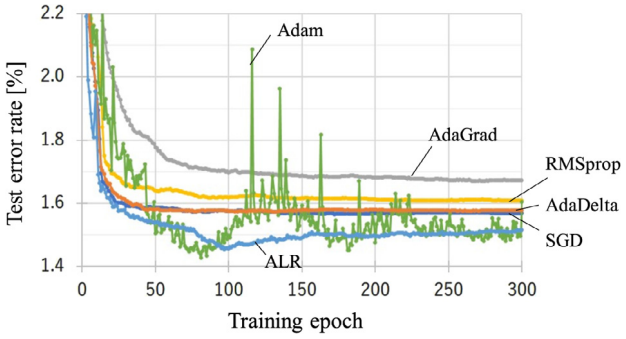



**Fig. 6.** Comparison with other updating methods. Mean values for several trials with different initial weights are shown.

As shown in Table 3, the error rates were the lowest when the learning rate was 0.5 or 1.0, but they were worse than with ALR. When the learning rate exceeded 4.0, the error rate was very large and saturated at around 90. Thus, training with a high initial learning rate is impossible.

We speculate that, with ALR, when the learning rate was the highest and the test error was the lowest, the shape of the loss function was almost flat. Therefore, a better solution was efficiently searched for by using an initially high learning rate and then reducing the rate as a solution was approached.

### 4.3. Comparison with other updating methods

We experimentally compared ALR using only SGD with the conventional updating method using SGD, AdaGrad, RMSprop, AdaDelta, and Adam updating methods. These methods have several parameters, respectively, but the main parameter corresponding to the initial learning rate has the greatest effect on the performance. In this experiment, the parameter was set to 0.01 in SGD, 0.01 in AdaGrad, 0.001 in RMSprop, 1.0 in AdaDelta, and 0.001 in Adam. Other experimental conditions were the same as described in Section 4.1.

As shown in Fig. 6, ALR had the lowest and most stable test error rate. While the lowest rate in Adam was almost the same as that in ALR, the transitions in Adam were unstable, so stopping the training when the error rate is the lowest is difficult, even if early-stopping is done using validation data. Thus, ALR had the best performance.

### 4.4. Effects of initial learning rate

The initial learning rate has a smaller effect in ALR than in the conventional methods because the learning rate is modified during training. In the conventional methods, parameter tuning takes much time because the program must be repeatedly run with various values to find an appropriate initial learning rate. In contrast, ALR does not require as much parameter tuning because it uses an adaptive learning rate.

To demonstrate this, we experimentally compared ALR with the conventional method, using SGD with the same initial learning rate. Other experimental conditions were the same as described in Section 4.1.

As shown in Table 4, the test error rate with ALR was lower than that with the conventional method for each initial learning rate. (The training failed for both methods when the rate was 10.0.) The effect of the initial learning rate for ALR was smaller than that for the conventional method because the range of the test error rates with the various initial learning rates was smaller. The difference between these methods was especially large when the initial learning rate 2.0. These results demonstrate that the initial learning rate in ALR can be more freely defined thanks to the adjustable learning rate. However, in practice, we should choose values commonly used in the default SGD, about 0.01–1.0. Among them, we recommend choosing an initial value between 0.1 and 1.0 because ALR needs to move the parameter largely for avoiding poor local solutions.

### 4.5. Experiments with convolutional neural network

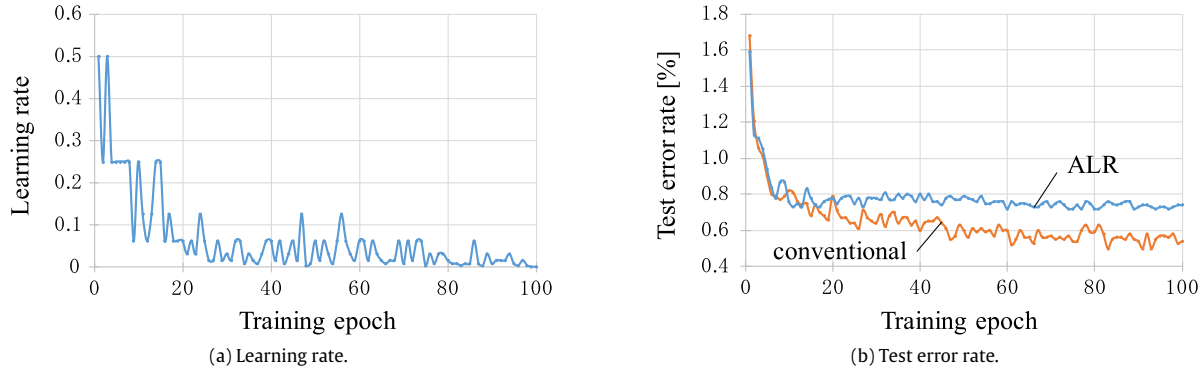
Here we present the experimental results for a convolutional neural network (CNN) with a structure as given in Table 5. The softmax function was used as the activation function of the output layer, and ReLU was used as the activation function of the other layers. Batch normalization (Ioffe & Szegedy, 2015), which changed the mean to 0 and the variance to 1, was applied to the output of each layer except the output layer. The dropout technique was used in the fully connected CNN layer to avoid overfitting, and the probability for selecting units was 0.5. Other experimental conditions were the same as described in Section 5.1.

Fig. 7(a) shows the transition in the learning rate, and Fig. 7(b) shows that in the test error rate. Unlike the case in which an MLP was used, the learning rate did not become large. It basically decreased, so a poor local solution was obtained, and the test error rate was worse than using the conventional method.

A cause for the downward trend in the learning rate is due to the instability of the loss function given by the dropout technique. When the loss function is cross-entropy, the loss function is given by

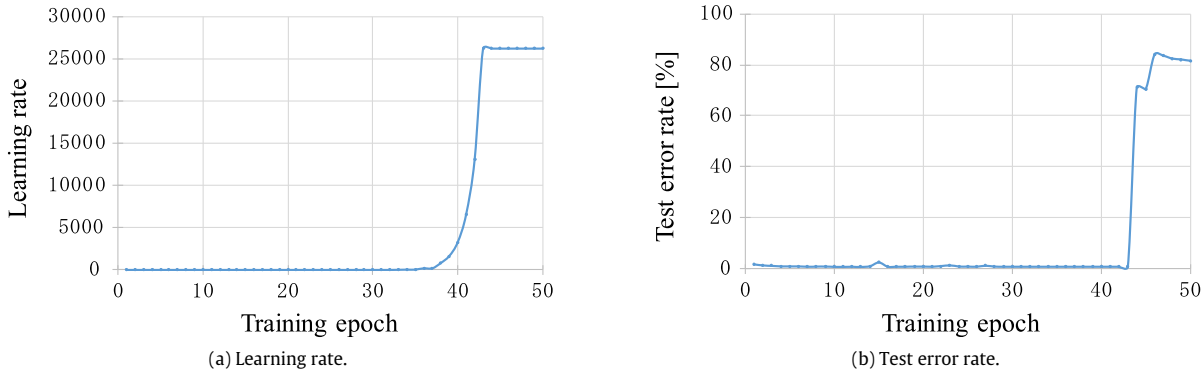
$$E(w) = -\sum_{c=1}^{C}\sum_{k=1}^{K} d_{ck} \log y_k(x_c; w), \tag{7}$$

where $C$ is mini-batch size, $K$ is the number of units in the output layer, $d_{ck}$ is the target output of unit $k$ for example $c$, $y_k$ is the actual output of unit $k$, $x_c$ is the input of example $c$, and $w$ is weights (and biases). Using the dropout technique improves the test accuracy by hiding units with a certain probability for each batch. As a result, $w$ changes greatly for each batch, so $E(w)$ also changes. With other loss functions, such as log-likelihood, they are similarly affected by the dropout technique. Because ALR assigns the same learning rate to all weights, in such complex and unstable loss function, a larger scale of factor tends to increase training loss and is not often selected.




(a) Learning rate. (b) Test error rate.

**Fig. 7.** Learning rate and test error rate when dropout technique was used.




(a) Learning rate. (b) Test error rate.

**Fig. 8.** Learning rate and test error rate when dropout technique was not used.

**Table 4**
Effects of initial learning rate. Mean values of best test error rates for several trials with different initial values of weights are shown. Initial values of weights were the same for both methods.

| Initial learning rate | Test error rate [%]/ALR | Test error rate [%]/Conv. method |
|---|---|---|
| 0.01 | 1.54 | 1.80 |
| 0.05 | 1.50 | 1.70 |
| 0.1 | 1.50 | 1.69 |
| 0.5 | 1.43 | 1.57 |
| 1.0 | 1.43 | 1.53 |
| 2.0 | 1.46 | 38.78 |
| 10.0 | 88.65 | 90.24 |

**Table 5**
CNN structure used for the experiments in Section 4.5. str. denotes the stride for pooling and p denotes the probability for selecting units in the dropout technique.

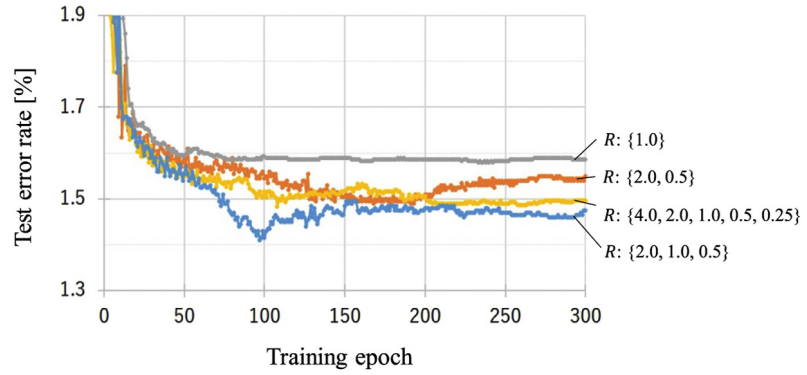
| Layer type | Channels/Units |
|---|---|
| $5 \times 5$ conv ReLU | 32 |
| $2 \times 2$ max pool, str. 2 | 32 |
| $5 \times 5$ conv ReLU | 64 |
| $2 \times 2$ max pool, str. 2 | 64 |
| Dropout with $p = 0.5$ | 64 |
| Fully connected | 1024 |
| Dropout with $p = 0.5$ | 1024 |
| Output softmax | 10 |

To eliminate the effect of an unstable loss function, we performed the same experiment without the dropout technique. The other conditions were the same as in the experiment with the dropout. As shown in Fig. 8(a) and (b), the learning rate increased, but the test error rate unexpectedly jumped to a much higher level.

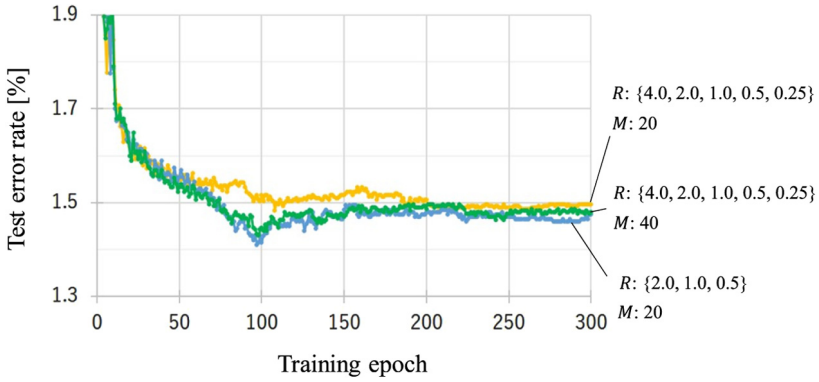
CNN had a larger number of weights (1,111,946) than MLP in the experiment, and we thought that this might have caused the difference in the results by the two networks. To compare MLP and CNN with similar numbers of weights, we increased the number of weights of MLP used in Section 4.1 to 1,192,510 by increasing the number of units in a hidden layer to 1500. The transition of the learning rate of the large MLP was similar to that of Fig. 5, that is, ALR worked well. Therefore, a large number of weight of CNN was not the cause of the failure of ALR.

Another possible reason is that CNN is composed of different types of layers such as convolutional layers and fully connected layers, which might make typical ranges of parameters more diverse than MLP. If so, the landscape of the loss function has a narrow valley. Since our method uses the same learning rate for all weights, the valley can be jumped over if the learning rate is too large for some of the weights, which results in increase of the training loss. Actually, in the experiment with CNN, the training loss sharply increased as the learning rate increased.

When the CIFAR-10 dataset (which is for the general object recognition task and requires a CNN structure for the model) was used, the training similarly failed. Thus, we investigate in the future work whether ALR works well on networks other than MLP and CNN.




(a) Beam size of 20.




(b) Beam size of 20 or 40.

**Fig. 9.** Test error rate for different numbers of branches for two beam sizes. $R$ denotes a set of scale factors, and $M$ denotes beam size. Number of branches corresponds with that of scale factors. Mean values for several trials with different initial weights are shown.

## 5. Effects of parameters

### 5.1. Number of branches

Increasing the number of branches (i.e., the number of scale factors) increases the number of routes that can be searched, resulting in a smaller training loss. However, as shown in Algorithm 1, increasing the number of branches $N$ leads to a proportional increase in computational cost and number of weight memories. Therefore, this parameter should be defined considering both factors.

The computational complexity is calculated on the basis of the number of step for each epoch. If the number of branches $N$ is not set and therefore an unlimited number of routes are searched, the computational complexity cannot be defined. However, by setting the number of branches, the computational complexity becomes $O\left(N^T\right)$, where $T$ is the number of epoch.

Here we experimentally investigated the effect of the number of branches. The number of branches should be large for a high precision but small for a short computation time. The numbers of branches were 1, 2, 3, and 5, and the set of scale factors was some combination of 4.0, 2.0, 1.0, 0.5, and 0.25, which are described by the format of $a^k$ and the beam size was 20. In those parameters, when the value is 1, ALR corresponds to the conventional method. The number of trials differed from the number given in Section 4.1. Other experimental conditions were the same as described in Section 4.1.

As shown in Fig. 9(a), the test error rate was the highest when the number of branches was 1; it was lower when the number was 2 or 3. However, the rate when the number of branches was 5 was higher than when it was 3. This is because a beam of size 20 is insufficient when the number of branches is 5.

We performed the same experiment for a beam size of 40 and 5 branches. As shown in Fig. 9(b), the ordering of the test error rate was the same as when the beam size was 20 and the number of branches was 3. Thus, we conclude that there is a lower limit on the test error rate and that it can be achieved by using a sufficiently large beam size relative to the number of branches.

Briefly speaking, we can identify the following four cases according to relative sizes of the number of branches $N$ and the beam size $M$ to guide how they should be optimally chosen. (1) large $N$, large $M$: then the best performance (in terms of the test error) is obtained, although the computational cost is the largest. (2) large $N$, small $M$: then nodes with relatively high training losses are not stored, so avoiding poor local solution becomes difficult, resulting in a bad performance. (3) small $N$, large $M$: then ALR can be effectively used, but a lower limit on the test error rate exists as described above, so very large $M$ is not preferable for the computational cost. (4) small $N$, small $M$: then it is close to SGD, which is the case of $N = M = 1$, and therefore reduces an advantage of ALR, but the computational cost is the smallest.

### 5.2. Scale factor

The scale factor should be defined so that the learning rate changes substantially. If it does not change substantially at each state transition, the effect of ALR is negligible; that is, avoiding poor local solutions is difficult. Fortunately, the scale factor does not affect the computational cost because it changes only the learning rate.

In setting this parameter, 1.0 should be included in the set of scale factor, which means including the conventional method. In other elements, the difference from 1.0 should not be too large or too small. If the difference is too large, each weight moves greatly

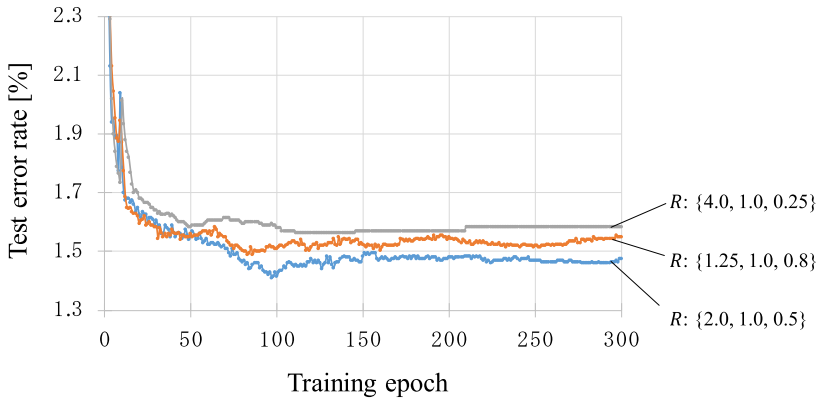



**Fig. 10.** Test error rate in experiments with several scale factors. *R* denotes a set of scale factors. Mean values for several trials with different initial weights are shown.

on training function. Some weights can greatly increase the loss, when the overall training loss with all weights will increase. Such scale of factor is not often selected. If the difference is too small, weights tend to be captured by local solution, which results in a bad generalization performance. The criterion of "large" and "small" is dependent on tasks, so parameter tuning is needed.

We experimentally investigated the effect of the scale factor, using several sets of the parameter described by the format of $a^k$. The number of branches was set to 3 and the beam size was set to 20. Other experimental conditions were the same as described in Section 4.1.

As shown in Fig. 10, the test error rate was the lowest when the set of scale factors was {2.0, 1.0, 0.5}. The result was better than that with scale factors {1.25, 1.0, 0.8} because larger differences among the scale factors in the set facilitate finding a better solution. The result with scale factors {4.0, 1.0, 0.25} was worse than that with scale factors {2.0, 1.0, 0.5} because scale factor 4.0 was rarely chosen and thus did not lead to a large improvement in the test error rate.

We slightly changed scale factors {2.0, 1.0, 0.5} and used scale factors {2.2, 1.0, 0.45}, where 2.2 is 10% larger than 2.0 and 0.45 is 10% smaller than 0.5. Because the product of 2.2 and 0.45 is 0.99, choosing both their values at successive epochs can decrease the learning rate by 1%. It enables a fine decreasing tuning for the learning rate during training, which is especially effective when training converges. When scale factors {2.2, 1.0, 0.45} were used for an experiment, the lowest test error rate was 1.435%, which is not greatly different from 1.410%, the result when scale factors {2.0, 1.0, 0.5} were used. On the other hand, when the difference among scale factors was large, the test error rate became large, as described in the previous section. Thus, the scale factor has some optimal range in which learning becomes successful.

The loss function is determined by the network structure and data, and therefore optimal scale factors depend on them. A general scale factor selection criterion we found through the experiments is that there should be sufficient chances to increase the learning rate at the beginning of training. Increasing the learning rate is important at the beginning of training so that a better solution can be explored without being captured by a poor local solution. If the learning rate hardly increases, ALR loses the advantage over the conventional techniques described in Section 1.

To demonstrate the appropriateness of the criterion with MNIST dataset, we investigated the relationship between the frequency of the increase of the learning rate during the beginning of training and the smallest test error rate during training. We defined epochs up to 50 and epochs up to 100 as the beginning of training. Changes in the learning rate from the previous epoch are categorized into *increase*, *decrease*, or *no change*, and the proportion of *increase* was calculated. We used several sets of scale factors,

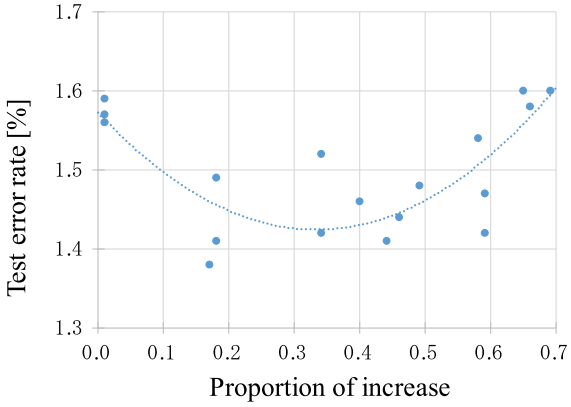


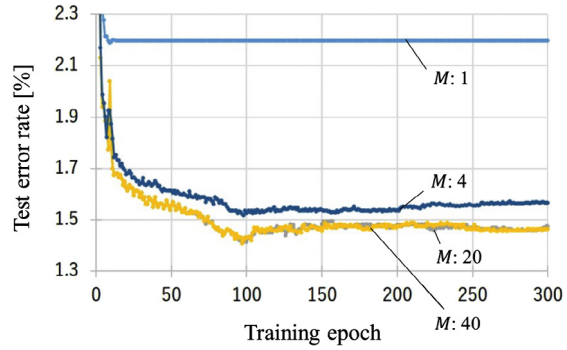

**Fig. 11.** Relationship between proportion of increase of learning rate and test error rate in MNIST dataset.

and trainings were performed for three times with different initial weights, respectively. Other experimental conditions were the same as described in Section 4.1.

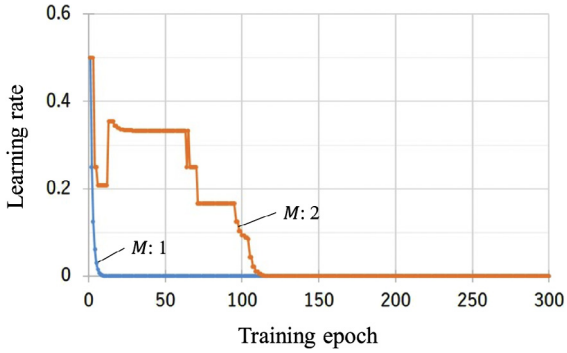
The experimental result given in Fig. 11 shows that when the proportion of *increase* in the learning rate was close to 0 or 1, a good solution could not be found and we consider the scale factors were not appropriate. We speculate that when the rate of *increase* was close to 0, which was for example obtained by scale factors {4.0, 1.0, 0.25}, the parameter was captured by a poor local solution, and when the rate of *increase* was close to 1, which was for example obtained by scale factors {1.001, 1.0, 0.999}, the parameter could not search a sufficiently wide region because the learning rate increased very slowly. When the rate of *increase* was between 0.1 and 0.6, a small test error rate was obtained. Scale factors {2.0, 1.0, 0.5} used in Section 4.1 met this criterion because the mean of the rates of *increase* up to epoch 100 was 0.18.

### 5.3. Beam size

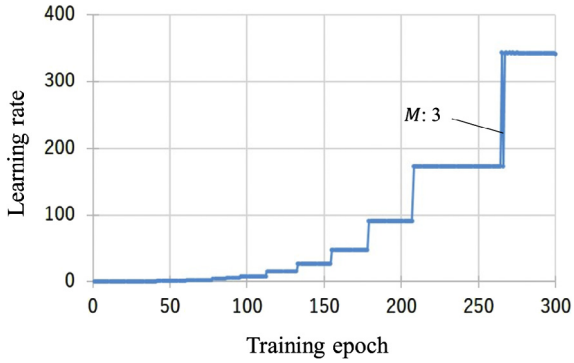
If the beam size *M* is set to 1, nodes with a training loss higher than the lowest loss are removed. However, even if a node has higher loss at an epoch, it may have the lowest training loss at a subsequent epoch. Therefore, it is important to set this parameter to a large value to avoid poor local solutions. Ideally, this parameter should not be limited in order to enable all states to be stored. However, as shown in Algorithm 1, increasing the beam size leads to a proportional increase in the computational cost and the number weight memories. Therefore, this parameter should be defined considering both effects.



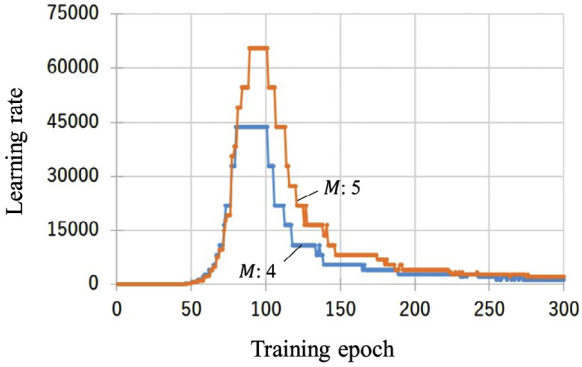

(a) Test error rate (Beam size: 1, 4, 20, 40).




(b) Learning rate (Beam size: 1, 2).




(c) Learning rate (Beam size: 3).




(d) Learning rate (Beam size: 4, 5).

**Fig. 12.** Test error rate and learning rate for different beam sizes. Trend in transition of learning rate depended on beam size. *M* denotes beam size.

As described in Section 5.1, the computational complexity is $O\left(N^T\right)$ when only the number of branches is used. When the beam size $M$ is used in addition to the number of branches, the computational complexity decreases to $O(NMT)$. Thus, we conclude that the use of both parameters greatly reduces computational cost, compared to the case that an unlimited number of routes are searched.

We experimentally investigated the effect of the beam size. Just like number of branches, the beam size should be also large for a high precision but small for a short computation time. We set the number of branches to 3 and the set of scale factors to {2.0, 1.0, 0.5}. Other experimental conditions were the same as described in Section 4.1.

As shown in Fig. 12(a), when the beam size was 1, the test error rate was the highest because the weights tended to be captured with a poor local solution. When the beam size was 20 or 40, the lowest error rate was obtained. The results for these two beam sizes were the same because a beam size of 20 was sufficient for obtaining the lowest error rate under these conditions. Thus, we conclude that by increasing the beam size, a wider range can be searched. This enables a poor local solution to be avoided, but there is an upper limit on this effect.

As shown in Fig. 12(b) to (d), the trend in the transition of the learning rate is affected by the beam size. When the beam size was small, 1 or 2, the size was insufficient under these conditions, so the weights tended to be captured with poor local solutions. When the beam size was large, 4 or 5, the transition was similar to that with a beam size of 20 (Fig. 5). That is, high learning rates were obtained during training.

## 6. Experiments with other datasets

To confirm the wide applicability of ALR, we performed experiments using other datasets. Car Evaluation, Wine, and Letter Recognition datasets, which are datasets for multi-classification included in UCI Machine Learning Repository (Lichman, 2013) and can be used with MLP.

Car Evaluation dataset contains 1728 samples and each sample is classified to one of 4 classes for overall evaluations. The number of features is 6, consisting of the values such as buying price and number of doors. Wine dataset contains 178 samples and each sample is classified to one of three kinds of wines. The number of features is 13, consisting of the chemical analysis such as alcohol and malic acid. With these datasets, we used 35% of samples as test data. Letter Recognition dataset contains 20,000 samples, and we used the first 16,000 samples as training data and remaining 4000 samples as test data, which is officially recommended. Each sample is classified to one of 26 classes from A to Z in the English alphabet. The number of features is 16, consisting of statistical moments and edge counts.

We set the number of branches to 3, the set of scale factors to {1.001, 1.000, 0.999}, and the beam size to 20. This set of scale factors is based on a general scale factor selection investigated later. By using the similar values for the scale factors, the initial learning rate affects the result more largely. For each ALR and conventional method, ten trials were performed using different initial learning rates (for each 0.1 from 0.1 to 1.0). Among those results, the best results were compared. Test loss with the loss function shown in Eq. (7) was used for the evaluation because the test error rate can be rough when small datasets are used, that is, the number of test samples is small. Moreover, Adam with the default parameter, which is the same as that used in the experiment in Section 4.3, was used for comparing.

The experimental results are given in Table 6. The numbers outside parentheses in the columns: ALR and Fixed, and the numbers in the column: Adam denote the lowest test loss in ten trials with different initial learning rates. The numbers inside parentheses in the columns: ALR and Fixed denote the initial learning rate when

**Table 6**
Experimental results using other datasets. The numbers outside parentheses in the columns: ALR and Fixed, and the numbers in the column: Adam denote the lowest test loss in ten trials with different initial learning rates. The numbers inside parentheses in the columns: ALR and Fixed denote the initial learning rate when the test loss was obtained.

| Dataset | Structure | ALR | Fixed | Adam |
|---|---|---|---|---|
| Car | **6-10-4** | **0.076 (lr:0.7)** | 0.079 (lr:0.6) | 0.103 |
| Wine | **13-10-3** | **0.062 (lr:0.8)** | 0.094 (lr:0.7) | 0.116 |
| Letter | **16-100-26** | **0.209 (lr:0.3)** | 0.220 (lr:0.4) | 0.258 |

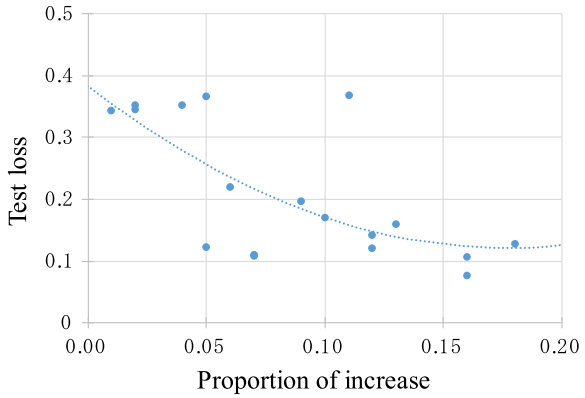



**Fig. 13.** Relationship between proportion of increase of learning rate and test loss in Car dataset.

the test loss was obtained. Moreover, the numbers of units for each layer of the neural networks used for these experiments are shown in the column: Structure. In all datasets, ALR resulted in the lowest test losses.

As is the case with the experiment at the last of Section 5.2, we investigated the relationship between the frequency of the increase of the learning rate during the beginning of training and the smallest test loss during training. Car Evaluation dataset was used on behalf of the datasets used in this section. The experimental result given in Fig. 13 shows that when the proportion of *increase* of the learning rate was close to 0, a good solution could not be found and we consider that the scale factors were not appropriate. Just like the result with MNIST dataset, increasing the learning rate at the beginning of training is important for obtaining a small test loss. Scale factors {1.001, 1.0, 0.999} used in this section met the scale factor selection criterion described at Section 5.2 because the mean of the rates of *increase* up to epoch 100 was 0.13.

Scale factors {2.0, 1.0, 0.5} worked well in the experiment with MNIST dataset, but in this dataset, did not meet the criterion because the mean of the rates of *increase* up to epoch 100 was 0.03. 0.5 in the scale factor was intensively chosen from beginning to end, and therefore we speculate that the parameter was captured by a poor local solution.

## 7. Conclusion

Based on a tree search with beam size, our method ALR technique has been designed as a machine learning algorithm adaptively increasing and/or decreasing the learning rate to reduce the training loss as much as possible, allowing us to effectively find good solutions, avoiding poor local solutions, unlike methods that mainly decrease the learning rate.

The experimental results have demonstrated that the method can lower the test error rate in various test datasets. Of particular interest is the observation that the learning rate was increased substantially and as a result a better solution was found outside the area leading to a poorer local solution. Furthermore, the method is more robust in the sense that it is less dependent on the initial learning rate.

We have investigated the effects of the three main parameters (i.e., the number of branches, the scale factor, and the beam size) and shown that the effects are moderate but the first and the last parameters should be large enough within appropriate upper limits, considering the computation time and the memory capacity.

We trained a convolutional neural network with the method but could not necessarily get a desired result. To clarify the reason for such a behavior and develop an improved method should be a part of our future work.

In ALR technique, all weights are updated with the same learning rate. Developing its efficient extension for individually controlling the learning rate for each weight as in AdaGrad is also a possible future work.

## Acknowledgment

This research was supported by Global Station for Big Data and CyberSecurity, a project of Global Institution for Collaborative Research and Education at Hokkaido University.

## References

Breuel, T. M. (2015). On the Convergence of SGD Training of Neural Networks. ArXiv Preprint arXiv:1508.02790.

Daniel, C., Taylor, J., & Nowozin, S. (2016). Learning step size controllers for robust neural network training. In *Proceedings of the thirtieth AAAI conference on artificial intelligence*.

Duchi, J., Hazen, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research (JMLR), 12*, 2121–2159.

Egmont-Petersen, M., Talmon, J. L., Hasman, A., & Ambergen, A. W. (1998). Assessing the importance of features for multi-layer perceptrons. *Neural Networks*, *11*(4), 623–635.

Fayek, H. M., Lech, M., & Cavedon, L. (2017). Evaluating deep learning architectures for speech emotion recognition. *Neural Networks*, *92*, 60–68.

Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training. deep feedforward neural networks. In *Proceedings of artificial intelligence and statistics conference , Vol. 9* (pp. 249–256).

Glorot, X., Bordes, A., & Bengio, Y. (2011). Deep sparse rectifier neural networks. In *Proceedings of artificial intelligence and statistics conference, Vol. 15* (pp. 315–323).

Hardt, M., Recht, B., & Singer, Y. (2015). Train faster. generalize better: Stability of stochastic gradient descent. ArXiv Preprint. arXiv:1509.01240.

Hinton, G., Deng, L., Yu, D., Dahl, G. E., Mohamed, A. R., Jaitly, N., et al. (2012). Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, *29*(6), 82–97.

Huang, Z., Zweig, G., Levit, M., Dumoulin, B., Oguz, B., & Chang, S. (2013). Accelerating recurrent neural network training via two stage classes and parallelization. In *2013 IEEE workshop on automatic speech recognition and understanding*. IEEE.

Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd international conference on machine learning*.

Kanada, Y. (2016). Optimizing neural-network learning rate by using a genetic algorithm with per-epoch mutations. In *Proceedings of international joint conference on neural networks*.

Kingma, D., & Ba, J. (2015). Adam: A Method for Stochastic Optimization. In *Proceedings of international conference on learning representations*.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. In *Advances in neural information processing systems*.

LeCun, Y., Bottou, L., Orr, G. B., & Müller, K. (1998). Efficient BackProp. In *Neural networks: Tricks of the trade* (pp. 9–48).

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278–2324.

Lichman, M. (2013). *UCI machine learning repository*. Irvine, CA: University of California, School of Information and Computer Science, [http://archive.ics.uci.edu/ml].

Mandt, S., Hoffman, M. D., & Blei, D. M. (2016). A Variational Analysis of Stochastic Gradient Algorithms. ArXiv Preprint arXiv:1602.02666.

Neyshabur, B., Salakhutdinov, R., & Srebro, N. (2015). Path-SGD:Path-normalized optimization in deep neural networks. In *Advances in neural information processing systems*.

Park, J. G., & Jo, S. (2016). Approximate Bayesian MLP regularization for regression in the presence of noise. *Neural Networks*, *83*, 75–85.

Sainath, T. N., Kingsbury, B., Saon, G., Soltau, H., Mohamed, A. R., Dahi, G., et al. (2015). Deep convolutional neural networks for large-scale speech tasks. *Neural Networks*, *64*, 39–48.

Shi, Z., Ye, Y., & Wu, Y. (2016). Rank-based pooling for deep convolutional neural networks. *Neural Networks*, *83*, 21–31.

Song, Y., Schwing, A. G., Zemel, R. S., & Urtasun, R. (2016.) Training deepneural networks via direct loss minimization. In *Proceedings of the 33rd international conference on machine learning*.

Srebro, N., & Shraibman, A. (2005). Rank, Trace-norm and max-norm. In *Proceedings of the 18th annual conference on learning theory* (pp. 545–560).

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research (JMLR)*, *15*(1), 1929–1958.

Sutton, R., & Barto, A. (1998). *Reinforcement learning: An introduction*. Cambridge: MIT Press.

Tang, J., Deng, C., & Huang, G.-B. (2015). Extreme learning machine for multilayer perceptron. *IEEE Transactions on Neural Networks and Learning Systems*, *27*(4), 809–821.

Tieleman, T., & Hinton, G. E. (2012). Lecture 6.5 - rmsprop, COURSERA: Neural networks for machine learning.

Wu, H., & Gu, X. (2015). Towards dropout training for convolutional neural networks. *Neural Networks*, *71*, 1–10.

Zeiler, M. D. (2012). ADADELTA: An Adaptive Learning Rate Method. ArXiv Preprint arXiv:1212.5701.