A Randomized Hyperparameter Tuning of Adaptive Moment Estimation Optimizer of Binary Tree-Structured LSTM

Ruo Ando[1]
National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo
101-8430 Japan

Yoshiyasu Takefuji[2]
Musashino University
Musashino University Faculty of Data Science
3-3-3 Ariake, Koto-Ku, Tokyo 1358181, JAPAN

***Abstract*—Adam (Adaptive Moment Estimation) is one of the promising techniques for parameter optimization of deep learning. Because Adam is an adaptive learning rate method and easier to use than Gradient Descent. In this paper, we propose a novel randomized search method for Adam with randomizing parameters of beta1 and beta2. Random noise generated by normal distribution is added to the parameters of beta1 and beta2 every step of updating function is called. In the experiment, we have implemented binary tree-structured LSTM and adam optimizer function. It turned out that in the best case, randomized hyperparameter tuning with beta1 ranging from 0.88 to 0.92 and beta2 ranging from 0.9980 to 0.9999 is 3.81 times faster than the fixed parameter with beta1 = 0.999 and beta2 = 0.9. Our method is optimization algorithm independent and therefore performs well in using other algorithms such as NAG, AdaGrad, and RMSProp.**

***Keywords*—Adaptive moment estimation; gradient descent; tree-structured LSTM; hyperparameter tuning**

I. Introduction

Optimization is involved in many deep learning algorithms. Analytical optimization is the basis of design algorithms. Neural network training is the most challenging problem of all the many optimization fields.

Among the hyperparameters to tune, learning rate is one of the most difficult because it drastically affects model performance. In general, the momentum algorithm can handle and mitigate the problem of being highly sensitive to some directions in parameter space. Adopting a separate learning rate for each parameter and these learning rates make sense about the directions of sensitivity.

Adaptive optimization is the algorithm to adapt the learning rate of model parameters based on many incremental or mini-batch based methods. Naturally, the choice of which algorithm to use is an unavoidable question. However, the algorithm selection much depend on the user's familiarity. In this paper, we will follow the assumptions below.

Hypothesis 1. There is no theorem or algorithm to determine which adaptive optimization is the best.

Then, as long as the optimal algorithm cannot be determined, we need to invent a method that can be commonly used to refine any algorithm. Parameter randomization is the promising approach to develop a speeding up procedure which is the adaptive optimization independent.

Hypothesis 2. According to Hypothesis 1, no optimization algorithms will be able to prove their own superiority over another algorithm.

Concerning the solution for Hypothesis 2, we propose the novel method concerning randomized hyperparameter tuning of adaptive moment estimation optimizer. The thrust of this paper is that the proposed method is adaptive optimization independent.

This paper is organized as follows. In Section II, three basic and popular algorithms are introduced: AdaGrad, RMSProp, and Adam. In Section III of related work, we introduce the related works of recurrent neural networks, recursive neural networks, LSTM, and adaptive optimization algorithms. In Section IV, we discuss the proposed method based on the binary tree of LSTM, linear activation unit, and constrained breadth-first search. In Section V, we discuss the research methodology for improving backpropagation, recurrent neural networks, and adaptive optimization algorithm. Experimental results are shown in Section VI. Our randomized hyperparameter tuning method is applied for Adam. In Section VII, we provide insights into the current situation of the research efforts of hyperparameter optimization. Then we conclude our paper in Section VIII.

II. Theoretical Background

A. AdaGrad

AdaGrad [18] algorithm adopting the learning rates with gradually changing them in proportion to the square root of the sum of all the historical squared values of the gradient.

$$s \leftarrow s + \bigtriangledown_\theta J(\theta) \tag{1}$$
$$\theta \leftarrow \theta - \eta \bigtriangledown_\theta J(\theta) \oslash \sqrt{s + \epsilon} \tag{2}$$

In equation (1), the square of gradients is accumulated into the vector s. Each S_i calculates the squares of the partial derivative of the cost function corresponding to the point to the parameter θ_i. In the case that the cost function is steep along the i^th dimension, s_1 will get larger and larger at each iteration.

Parameter s1 will get larger and larger at each iteration as long as the cost function is steep along the i^th dimension,

Equation (2) is almost identical to one of Gradient Descent. However, there is one big difference. That is, the gradient vector is scaled down by a factor of $\sqrt{s+\epsilon}$.

AdaGrad is called an adaptive learning rate because AdaGrad decays the learning rate so that the learning rate for steep dimensions is faster than gentler slopes. The parameters of AdaGrad decrease rapidly in their learning rate corresponding to the largest partial derivative of the loss. On the other hand, the parameters decrease in their learning rate with the small partial derivatives. As a result, if the learning process has the more moderately directions of parameter space, the effect whole becomes greater.

B. RMSProp

RMSProp [19] algorithm improves AdaGrad to perform better in the nonconvex setting. AdaGrad is designed to converge with changing the gradient accumulation into an exponentially weighted moving average. Generally, a nonconvex function is used to train a neural network. The learning trajectory eventually reaches a region that is a locally convex bowl after the trajectory goes through many different structures. RMSProp has advantages compared with AdaGrad in the point that AdaGrad slows down rapidly and consequently finished never converging the global optimum.

For doing this, the RMSProp accumulates only the gradients from the most recent iterations.

$$s \leftarrow \beta s + (1-\beta)\bigtriangledown_\theta J(\theta) \otimes \bigtriangledown_\theta J(\theta) \quad (3)$$
$$\theta \leftarrow \theta - \eta \bigtriangledown_\theta J(\theta) \oslash \sqrt{s+\epsilon} \quad (4)$$

In equation (3), exponential decay is used. The decay rate β is typically set to 0.9. Except for very simple problems, this optimizer almost always performs much better than AdaGrad. In fact, it was the preferred optimization algorithm of many researchers until Adam algorithm came around.

C. Adam

Adam [20] is an adaptive learning rate optimization algorithm. The name Adam derives from the phase Adaptive moments. Adam can be described as a variant on the hybrid of momentum and RMSProp with some important distinctions. Adam incorporates momentum directly as an estimation value of the first-order moment. The first-order moment is called as exponential weighting. Adam adopts momentum and bias corrections. Momentum is used in combination with rescaling, which have a clear theoretical motivation. Bias corrections are used to estimate both first-order moments and second-order moments to account for their initialization at the origin.

III. Related Work

Recurrent neural networks [1], or RNNs are feedforward neural networks for processing sequential data by extending with incorporating edges that span adjacent time steps. In general, RNNs suffer the difficulty of training by gradient-based optimization procedures. Local numerical optimization includes stochastic gradient descent or second-order methods, which causes the exploding and the vanishing gradient problems [13][14][15]. Werbos et al. [11] propose the backpropagation through time (BPTT), which is a training algorithm for RNN. BPTT is derived from the popular backpropagation training algorithm used in MLPN training [12]. Derivatives of errors are computed with backpropagation over structures [6].

Recursive neural networks are yet another representation of the generalization of recurrent networks by using a different form of computation graph. The computation graph adopted in recursive neural networks is a deep tree instead of the chain-like structure of RNNs. Pollack [2] proposes recursive neural networks. Bottou [3] discuss the potential use of the recursive neural network in learning to reason. In [4] and [5], recursive neural networks are more effective in performing on different problems such as semantic analysis in natural language processing and image segmentation.

There is a long line of research efforts on extending the standard LSTM [7] in order to adopt more sophisticated structures. Tai et al. [8] and Zhu et al. [9] tree-structured LSTMs extended from chain-like structured LSTMs by adopting branching factors. They demonstrated that such extensions outperform competitive LSTM baselines on several tasks such as semantic relatedness prediction and sentiment classification. Furthermore, Li et al. [10] show the effectiveness of tree-structured LSTM on various tasks and situations in which tree-like structure is effective.

Boris Polyak proposes Momentum optimization with terminal velocity [21]. In 1983, Yurii Nesterov proposes Nesterov Momentum Optimization (NAG) [22]. NAG adopts the gradient of the cost function which is not measured in the local position but slightly ahead in the direction of the momentum. RMSProp [19] is an improved version of AdaGrad. RMSProp extends AdaGrad by accumulating the gradients from the most recent iterations. Adam [20] is based on the idea of both Momentum optimization and RMSProp. Adam keeps track of an exponentially decaying average of past gradients and an exponentially decaying average of past squared gradients.

Santa (Stochastic Annealing Thermostats with Adaptive Momentum) [23] is an adaptation method of Adam and RMSprop by leveraging MCMC (Markov Chain Monte Carlo) methods. In GD by GD (Gradient Descent by Gradient Descent) [24] is based on the idea that the optimization algorithm is a learning problem, and the optimization structure is determined by learning. They also propose LSTM optimizer.

In Adam, RMSProp, exponential decay by exponential moving average was adopted. However, it has been reported that when the gradient in that mini-batch disappears immediately due to exponential decay, consequently, Adam and RMSProp does not converge to the optimal solution. Therefore, AMSGrad [25] is an improved version of Adam that prevents important gradient information from disappearing immediately.

In Adam, the adaptive learning rate is efficient for fast learning, but even after learning has progressed, the validation error is not well converged due to high volatility in the learning rate. On the other hand, in SGD, which uses a fixed learning rate, the final validation error can be reduced, but it takes too much time to get to that point. Concerning these drawbacks, AdaBound and AMSBound were proposed as optimization

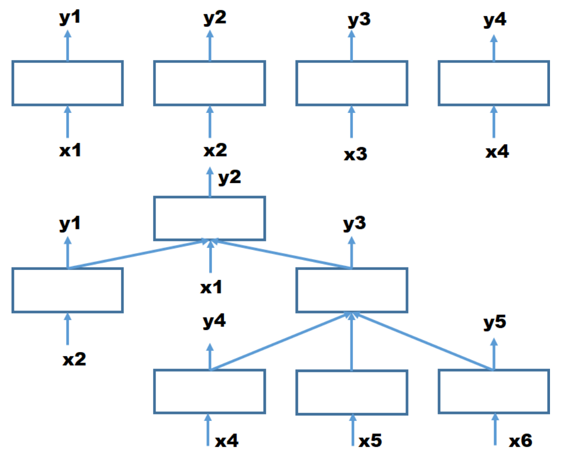

Fig. 1. True-structured LSTM with Arbitrary Branching Factor.

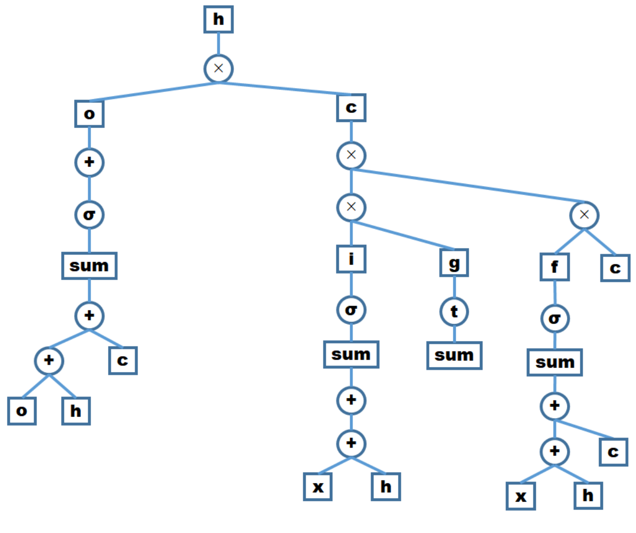

Fig. 2. Binary-tree-LSTM with Unary Operator.

methods as the combination Adam in the beginning and like SGD, in the end, [26].

As we have introduced in this section, adaptive optimization algorithms are constantly evolving, but there is still no theorem or algorithm to judge which algorithm is the best. Even one of the latest algorithm of [26] has not proven to be superior to everything else.

IV. Proposal Model and Method

A. Binary Tree of LSTM

The Tree-LSTM is a generalization of long short-term memory (LSTM) networks to tree-structured network topologies, introduced in [9]. Here, the core design concept introduces syntactic information for language tasks by extending the chain-structured LSTM to a tree-structured LSTM.

Fig. 1 shows the comparison of two kinds of LSTM network structures. The upper side of Fig. 1 shows a chain-structured LSTM network. The lower side of Fig. 1 depicts a tree-structured LSTM network with an arbitrary branching factor. It is shown that Tree-structured LSTM has a good performance in the case that the networks cope with the combination of words and phrases in natural language processing [8].

Recursion is a fundamental process in any different modalities. It is associated with many phases. A recursive procedure and hierarchical structure is formed commonly indifferent modalities. Also, recursion is a core technique for traversing the binary tree. Fig. 2 depicts the representation of binary-tree-LSTM with the unary operator. A binary tree is a tree whose elements have at most two children.

If each element in a binary tree includes only two children, these two children are typically called as the left and right child. In the case of the forward computation of a S-LSTM memory block, it is represented in the following equations.

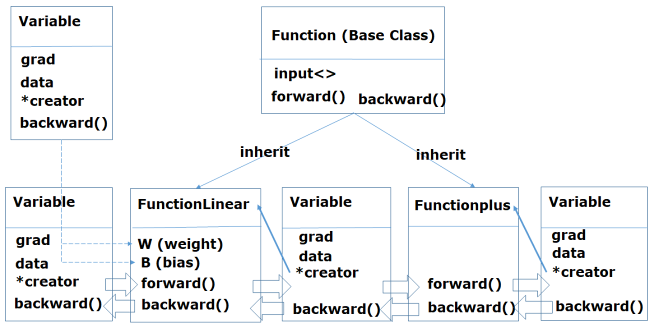

Fig. 3. Implementation of Linear Activation unit by Object-oriented Programming Language. The base Class of Function has the Inheritance of FunctionLinear.

$$i_t = \sigma(W^L_{hi}h^L_{t-1} + W^R_{hi}h^R_{t-1} + W^L_{ci}c^L_{t-1} + W^R_{ci}c_{t-1}R + b_i) \quad (5)$$

$$f^L_t = \sigma(W^L_{fl}h^L_{t-1} + W^R_{hfl}h^R_{t-1} + W^L_{cfl}c^L_{t-1} + W^R_{cfl}c_{t-1}R + b_{fl}) \quad (6)$$

$$f^R_t = \sigma(W^L_{ft}h^L_{t-1} + W^R_{hfr}h^R_{t-1} + W^L_{cfr}c^L_{t-1} + W^R_{cfr}c_{t-1}R + b_{fr}) \quad (7)$$

$$x_t = W^L_{hx}h^L_{t-1} + W^R_{hx}h^R_{t-1} + b_x) \quad (8)$$

$$c_t = f^L_t * c^L_{t-1} + f^R_t * i_t * tanh(x_t) \quad (9)$$

$$o_t = \sigma(W^L_{ho}h^L_{t-1} + W^R_{ho}h^R_{t-1} + W_{co}c_t + b_o) \quad (10)$$

$$h_t = o_t * tanh(c_t) \quad (11)$$

Here, σ is the element-wise logistic function. σ is adopted to restricts the gating signals to be in the range of [0, 1]. f_L and f_R denotes the left (L) and right (R) forget gate. b is biased, and W is the weighting matrices of the network. Finally, the sign * is a Hadamard product which is also called element-wise product.

More importantly, equation (14)-(20) consists of a binary operator. Therefore, this equation can be represented as a binary tree. A binary tree is a fundamental data structure in different modalities. In binary tree, the elements have at most two children. We typically name them the left and right children because each element in a binary tree can have only two children, In computation, a binary tree consists of nodes, where each node contains a L("left") reference, a R("right") reference, and a data element. The topmost (or bottommost) node in the tree is called the root node.

Algorithm 1 recursive function

```
1: $variable \rightarrow generator \rightarrow backward(grad)$
2: while $i \leq variable \rightarrow generator.inputs\_size()$ do
3:     $nv = variable \rightarrow generator.inputs()$
4:     if $nv = isGetGrad$ then
5:         $this \rightarrow backward(nv.get())$
6:     end if
7: end while
```

B. Linear Activation Unit

Fig. 3 depicts our implementation of a linear activation unit for the reverse-mode auto diff of linear activation. In artificial neural networks, a node's activation function defines the output of that node given an input or set of inputs. The input-output model is defined as follows:

$$f(x) = \psi * (\sum_{i=0}^{n} w_i * x_i + b)$$

Here, ψ is an activation function such as Tanh and RELU. Class FunctionLinear implements the function of $\sum_{i=0}^{n} w_i * x_i + b$. The notation of *creator is the pointer to the function which generates its variable. For example, FunctionLinear outputs r which is equal to $\sum_{i=0}^{n} w_i * x_i + b$ and is passed to FunctionTanh. The creator of variable r is FunctionLinear. Fig. 3 also illustrates the detailed implementation of the inheritance of functions and variables of tree-structured LSTM. Inheritance in the middle of Fig. 3 enables us to define classes for modeling relationships among types by sharing what is common and specializing only on that which is inherently different. Its derived classes inherit members defined by the base class. We can use derived class without change. Deriving class do not depend on the specifics of the derived type. Those operations redefine those member functions depending on its type, specializing the function to take into account the peculiarities of the derived type. Consequently, a derived class may define additional members beyond those it inherits from its base class.

C. Constrained Breadth-first Search

As we discussed in Section I-B, a tree-structured LSTM graph is generated for each mini-batch. Fig. 4 depicts the model of a few tree-structured LSTM graphs for mini-batches. As usual, a breadth-first search (BFS) is applied for the recursive search of the tree structure. However, other procedures on our model, such as loss, MSE (Mean-Square Error), and Tanh should be skipped before the program reaches the LSTM tree, as shown in the lower-left side of Fig. 4.

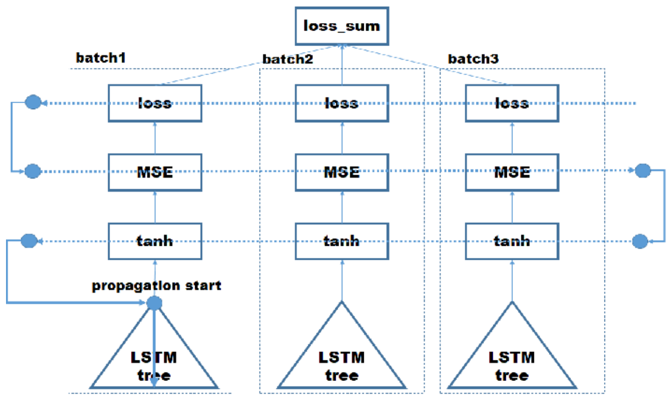

Fig. 4. Constrained Breadth First Search.

We modify the recursion algorithm as shown in Algorithm 1. Broadly, the breadth-first search is an algorithm for the traversal of tree (or graph) data structures. BFS begins at the tree root. It then searches all of the neighbor nodes at the present depth before it proceed to the nodes at the next depth.

Algorithm 2 Constrained model traversal

```
1: if $v \rightarrow last_{opt} \neq NULL \wedge \rightarrow opt = *v \rightarrow last_{opt}$ then
2:     $*v \rightarrow is\_last\_backward = true$
3: end if
4: if $if(v \rightarrow is\_last\_backward \neq NULL \wedge *v \rightarrow is\_last\_backward = false$ then
5:     $return$
6: end if
7: $back\_propagation()$
```

V. Research Methodology

A. Truncated Backpropagation through Time

Backpropagation through Time, or BPTT, is a specific application of backpropagation in neural networks for coping with sequence to sequence data like a time series. A recurrent neural network has one input each time step and predicts one output. Conceptually, BPTT performs by unrolling all input time steps, as shown in Fig. 5. In each time step, BPTT has one input time step and one copy of the network s_t , and one output o_t. Errors are then calculated and accumulated for each time slot with w.

Fig. 5 has outputs at each time step. The network is rolled back up, and the weights are updated. BPTT would be impractical in an online manner because its memory footprint grows linearly with time.

Truncated Backpropagation Through Time (TBPTT), which is an online version of BPTT, is proposed in [16]. TBPTT works analogously to BPTT. But, the sequence is calculated one-time step at a time periodically. The BPTT update is performed back for a fixed number of time steps. In [16], the accumulation stops after a fixed number of time steps. Truncated BPTT performs well if the truncated chains are effective in learning the recursive target functions.

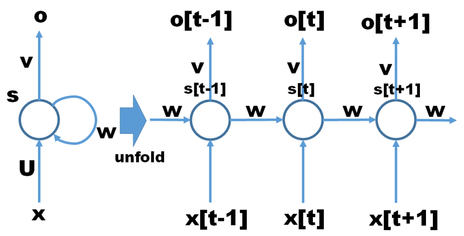

Fig. 5. Back Propagation through Time. It Works by Unrolling All Input Time Steps.

B. LSTM

Long short-term memory (LSTM) [7] is a family of recurrent neural networks. Like other recurrent neural networks, LSTM has feedback connections. Concerning the memory cell itself, it is controlled with a forget gate, which can reset the memory. unit with a sigmoid function. In detail, given a sequence data $x_1, ..., x_T$we have the gate definition as follows:

$$f_t = \sigma(W_{fh}h_{t-1} + W_{fx}x_t + P_f * c_{t-1} * b_f) \tag{12}$$
$$i_t = \sigma(W_i x_t + U_i h_{t-1} + P_i * c_{t-1} * b_i) \tag{13}$$
$$g_t = tanh(W_g x_t + U_g h_{t-1} + b_g) \tag{14}$$
$$c_t = i_t \Theta g_t + f_t \Theta c_{t-1} \tag{15}$$
$$o_t = \sigma(W_o x_t + U_o h_{t-1} + P_o * c_t + b_o) \tag{16}$$
$$h_t = o_t \Theta tanh(c_t) \tag{17}$$

where f_t is forget gate, i_t input gate, ot output gate and $g_t$input modulation gate. Particularly $P_f, P_i P_o$ indicates the peephole weights for the forget gate. The peephole connections introduced in [17] enable the LSTM cell to inspect its current internal states. Then, the backpropagation of the LSTM at the current time step t is as follows:

$$\delta o_t = tanh(c_t)\delta h_t \tag{18}$$
$$\delta c_t = (1 - tanh(c_t)^2) o_t \delta h_t \tag{19}$$
$$\delta f_t = c_{t-1}\delta c_t \tag{20}$$
$$\delta c_{t-1} = f_t \theta \delta c_t \tag{21}$$
$$\delta i_t = g_t \delta c_t \tag{22}$$
$$\delta g_t = i_t \delta c_t \tag{23}$$

C. Adam

Adam [20] stands for adaptive moment estimation. Adam optimization is the hybrid based on the ideas of momentum optimization and RMSProp. In Adam, momentum optimization keeps track of an exponentially decaying average of past gradients. On the other hand, RMSProp keeps track of an exponentially decaying average of past squared gradients.

$$m \leftarrow \beta_1 m - (1 - \beta_1) \bigtriangledown_\theta J(\theta) \tag{24}$$
$$s \leftarrow \beta_2 * s + (1 - \beta_2) \bigtriangledown_\theta J(\theta) \otimes J(\theta) \tag{25}$$
$$m \leftarrow \frac{m}{1 - \beta_1^t} \tag{26}$$
$$s \leftarrow \frac{m}{1 - \beta_2^t} \tag{27}$$
$$\theta \leftarrow \theta + \eta m \oslash \sqrt{s + \epsilon} \tag{28}$$

As far as steps 1, 2 and 5, Adam is closely similar to both momentum optimization and RMSProp.

The only difference is that instead of an exponentially decaying sum in momentum optimization and RMSProp, step 1 of Adam computes an exponentially decaying average rather than an exponentially decaying sum. Actually, these decaying sums are equivalent except for a content factor.

Steps 3 and 4 are technically specific detail. In steps 3 and 4, m and s are initialized at 0 at default. Then, m and s will be biased towards 0 at the starting phase of training. Consequently, steps 3 and 4 will help boost m and s in the early phase of training.

VI. Experiment

In this section, we describe the experimental results of the training and generating a sine wave. In the experiment, we use a workstation with Intel(R) Xeon(R) CPU E5-2620 v4 (2.10GHz) and 252G RAM.

Adam uses the moving average of gradient m_t as well as v_t, which is the squared moving average adopted by RMSProp and AdaDelta.

$$m_t = \beta_1 * m_t - 1 + (1 - \beta_1) g_t \tag{29}$$
$$v_t = \beta_2 * v_t - 1 + (1 - \beta_2) g_t^2 \tag{30}$$

Optimization problem requires the search for good hyperparameters. The hyperparameters are variable to decide. The cost to be optimized is the validation set error. For evaluating our method, we generate a sin wave with random noise by the normal distribution. Then, we apply curve fitting to the generated sin wave.

For hyperparameter tuning, we use three test scenarios. In first case, we set the parameter $\beta 1$ and $\beta 2$ fixed to 0.9 and 0.999. In second case, we set the parameter $\beta 1$ ranging from 0.89 to 0.91 and $\beta 2$ ranging from 0.9985 to 0.9995. Finally, we set the parameter $\beta 1$ ranging from 0.88 to 0.92 and $\beta 2$ ranging from 0.9980 to 0.9999.

Fig. 6, 7, and 8 are the results of the curve fitting of three test cases. Fig. 9 shows the comparison of validation loss of three test cases. It turned out that test case 3 with the parameter $\beta 1$ ranging from 0.88 to 0.92 and $\beta 2$ ranging from 0.9980 to 0.9999 has the best performance. The plot of test case 3 decreases rapidly comaware to the other two cases.

The results in Fig. 9 suggest that early stopping may be applicable. Early stopping is another approach to regularize iterative learning algorithms, including Adam and Gradient

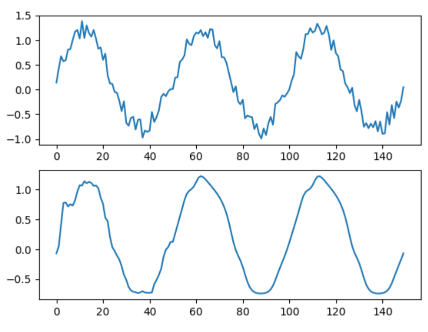

Fig. 6. $\beta 1 = 0.9/\beta 2 = 0.999$.

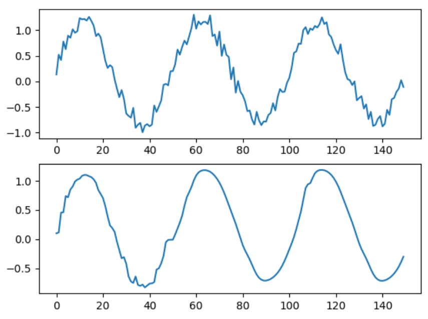

Fig. 7. $\beta 1 = 0.89 - 0.91/\beta 2 = 0.9985 - 0.9995$.

Descent, to stop training immediately after the validation loss reaches out a minimum value. In other words, leveraging early stopping, we control and terminate training as soon as the validation loss falls to a minimum. Early stopping is a simple and powerful regularization technique.

Table I shows the validation loss of three test cases. At step 5 with epoch size 750, elapsed time of the third case with $\beta 1$ ranging from 0.88 to 0.92 and $\beta 2$ ranging from 0.9980 to 0.9999 is 0.0493266. The elapsed time of 0.0493266 is 1.81 times faster than the second case and 3.81 times faster than the first case.

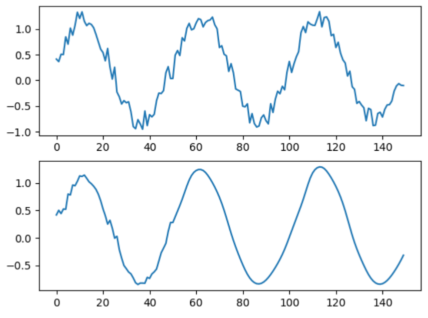

Fig. 8. $\beta 1 = 0.88 - 0.92/\beta 2 = 0.9980 - 0.9999$.

TABLE I. Comparison of Three Test Cases

	Elaplse time in epoch=750 (sec)		
	0.9/0.999	0.89-0.91/0.9985-0.9995	0.88-0.92/0.9980-0.9999
1	0.392805	0.155601	0.140308
2	0.310886	0.130887	0.0976786
3	0.259277	0.113318	0.0733389
4	0.219546	0.100228	0.059955
5	0.187943	0.0893164	0.0493266

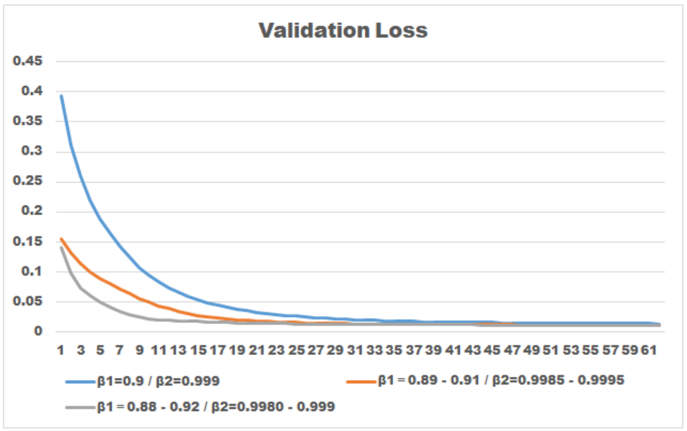

Fig. 9. Validation Loss with Parameter $\beta 1$ and $\beta 2$ Changed.

VII. Discussion

In the process of discussing a series of optimization algorithms, a question now arises - which algorithm should one choose?

Schaul et al. [27] presents a comparative study of a large number of optimization algorithms across a wide range of learning tasks. According to this, although the series of optimization algorithms with adaptive learning rates such as RMSProp and AdaDelta works fairly robustly, no single best algorithm has emerged.

On the other hand, the drawback common to most hyperparameter optimization algorithms is the need for a training experiment to run before they can retrieve any information from the experiment. A more sophisticated (automated) random search is usually much less efficient than a manual search by a human practitioner. Partly because the set of hyperparameters is often completely pathological. Broadly, in this context, the choice of which algorithm to use largely depends on the user's familiarity with the algorithm.

Generally, adaptive optimization algorithms are recommended. However, Ashia C. Wilson et al. [28] pointed out that AdaGrad, RMSProp, and Adam generalize poorly on some datasets. According to this, it follows that we may stick to other alternatives such as Momentum optimization or Nesterov Accelerated Gradient as long as researchers have a better understanding of this issue. In this situation, we can conclude that our method is helpful and practical because our method is optimization algorithm independent.

VIII. Conclusion

Adam (Adaptive Moment Estimation) is one of the promising techniques for parameter optimization of deep learning.

In this paper, we propose a novel random search method for Adam with randomizing parameters of $\beta1$ and $\beta2$. Random noise generated by normal distribution is added to the parameters of $\beta1$ and $\beta2$ every step of updating function is called. In the experiment, we have implemented binary tree-structured LSTM and adam optimizer function.

There have been lots of research efforts on algorithms which each seek to address the challenge of optimizing deep models by adapting the learning rate for each model parameter. However, there is currently no consensus on which algorithm is best to choose. Our method of randomized hyperparameter tuning is an optimization method independent. Therefore, our method can be applied for various kinds of algorithms such as NAG, AdaGrad and RMSProp, and so on. Updating function is called. In the experiment, we have implemented binary tree-structured LSTM and adam optimizer function. It turned out that in best case, randomized hyperparameter tuning with $\beta1$ ranging from 0.88 to 0.92 and $\beta2$ ranging from 0.9980 to 0.9999 is 3.81 times faster than the fixed parameter with $\beta1$ = 0.999 and $\beta2$ = 0.9. We can conclude that adding random noise to the fixed-parameter of $\beta1$ and $\beta2$ is effective and reasonable compared with a naive manual search.

REFERENCES

[1] Rumelhart, David E.; Hinton, Geoffrey E., Williams, Ronald J., Learning representations by back-propagating errors. Nature 323 (6088): 533-536, 1986/10

[2] Jordan B. Pollack: Recursive Distributed Representations. Artif. Intell. 46(1-2): 77-105 (1990)

[3] Leon Bottou: From Machine Learning to Machine Reasoning. CoRR abs/1102.1808 (2011)

[4] Socher, Richard, Lin, Cliff C., Ng, Andrew Y., and Manning, Christopher D. Parsing Natural Scenes and Natural Language with Recursive Neural Networks. In Proceedings of the 26th International Conference on Machine Learning (ICML), 2011.

[5] Socher, Richard, Perelygin, Alex, Wu, Jean Y., Chuang, Jason, Manning, Christopher D., Ng, Andrew Y., and Potts, Christopher. Recursive deep models for semantic compositionality over a sentiment treebank. In Proceedings of the Conference on Empirical Methods in Natural Language Processing, EMNLP ?f13, Seattle, USA, 2013. Association for Computational Linguistics.

[6] Goller, Christoph, and Kohler, Andreas. Learning task-dependent distributed representations by backpropagation through structure. In In Proc. of the ICNN-96, pp. 347-352, Bochum, Germany, 1996. IEEE.

[7] Sepp Hochreiter and J.Nurgen Schmidhuber. 1997. Long short-term memory. Neural Computation 9(8):1735-1780.

[8] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. In Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing. ACL, pages 1556-1566.

[9] Xiaodan Zhu, Parinaz Sobhani, and Hongyu Guo. 2015. Long short-term memory over recursive structures. In Proceedings of the 32nd International Conference on Machine Learning. ICML, pages 1604-1612.

[10] Jiwei Li, Minh-Thang Luong, Dan Jurafsky, and Eduard Holy. 2015. When are tree structures necessary for deep learning of representations. In Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing. EMNLP, pages 2304-2314.

[11] P. J. Werbos, Backpropagation through time: What does it does and how to do it. In Proc. IEEE, vol. 78, no. 10, pp. 1550-1560, Oct. 1990.

[12] P. J.Werbos, The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting, 1st ed. Hoboken, NJ:Wiley, 1994.

[13] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. IEEE transactions on neural networks, 5(2):157-166, 1994.

[14] John F. Kolen and Stefan C. Kremer. Gradient Flow in Recurrent Nets: The Difficulty of Learning LongTerm Dependencies, pages 464-479. Wiley-IEEE Press, 2001.

[15] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28, ICML?f13, pages III310-III318. JMLR.org, 2013.

[16] Ronald J. Williams and Jing Peng. An efficient gradient-based algorithm for online training of recurrent network trajectories. In Neural Computation, 1990.

[17] Gers, F. A., and Schmidhuber, J. (2001). LSTM recurrent networks learn simple context-free and context-sensitive languages. IEEE Trans. Neural. Netw., 12(6), 1333-1340.

[18] J. Duchi, E. Hazan, and Y. Singer., Adaptive subgradient methods for online learning and stochastic optimization, Journal of Machine Learning Research 12 (Jul): 2121–2159 (2011)

[19] Tijmen Tieleman, Geoffrey Hinton, Rmsprop: Divide the gradient by a running average of its recent magnitude. Coursera: Neural networks for machine learning, COURSERA Neural Networks Mach. Learn, 2012

[20] Diederik P. Kingma, Jimmy Ba, Adam: A Method for Stochastic Optimization, ICLR (Poster) 2015

[21] BT Polyak, Some methods of speeding up the convergence of iteration methods, USSR Computational Mathematics and Mathematical Physics 4 (5), 1-17,1964

[22] NESTEROV Y., A method for unconstrained convex minimization problem with the rate of convergence o(1/k$\hat{2}$), Doklady AN USSR 269, 543-547, 1983

[23] Changyou Chen, David E. Carlson, Zhe Gan, Chunyuan Li, Lawrence Carin: Bridging the Gap between Stochastic Gradient MCMC and Stochastic Optimization. ASSISTANTS 2016: 1051-1060

[24] Marcin Andrychowicz, Misha Denil, Sergio Gomez Colmenarejo, Matthew W. Hoffman, David Pfau, Tom Schaul, Nando de Freitas: Learning to learn by gradient descent by gradient descent. NIPS 2016: 3981-3989

[25] Alexandre Défossez, Léon Bottou, Francis R. Bach, Nicolas Usunier: On the Convergence of Adam and Adagrad. CoRR abs/2003.02395 (2020)

[26] Liangchen Luo, Yuanhao Xiong, Yan Liu, Xu Sun: Adaptive Gradient Methods with Dynamic Bound of Learning Rate. CoRR abs/1902.09843 (2019)

[27] Tom Schaul, Ioannis Antonoglou, David Silver: Unit Tests for Stochastic Optimization. ICLR 2014

[28] Ashia C. Wilson, Rebecca Roelofs, Mitchell Stern, Nati Srebro, Benjamin Recht: The Marginal Value of Adaptive Gradient Methods in Machine Learning. NIPS 2017: 4148-4158