International Journal of Smart Computing and Artificial Intelligence
International Institute of Applied Informatics
2023, Vol. 7, No. 1, IJSCAI669

A Constrained Recursion Algorithm for Tree-structured LSTM with Mini-batch SGD

Ruo Ando [*], Yoshiyasu Takefuji [†]

Abstract

Tree-structured LSTM (Long Short-Term Memory) is a promising concept to consider long-distance interaction over hierarchies with syntactic information. Besides, compared with chain-structured one, tree-structured LSTM has better modularity of learning process. However, there still remains the challenge concerning hyperparameter tuning in tree-structured LSTM. Mainly, hyperparameter of mini-batch SGD (Stochastic Gradient Descent) is one of the most important factors which decides the quality of the prediction of LSTM. For more sophisticated hyperparameter tuning of mini-batch SGD, we propose a constrained recursion algorithm of tree-structured LSTM. Our algorithm enables the program to generate an LSTM tree for each batch. By doing this, we can evaluate the tuning of hyperparameter of mini-batch size more correctly compared with chain-structured one. Besides, our constrained recursion algorithm can traverse the LSTM and update the weights over several LSTM tree with a breadth-first search. In the experiment, we have measured the validation loss and elapsed time in changing the size of mini-batch. We have succeeded in measuring the learning process's stability with small batch size and the instability of overfitting with large batch size more precisely than chain-structured LSTM.

Keywords: Constrained recursion, hyperparameter tuning, tree-structured LSTM, mini-batch SGD.

1 Introduction

There two main problems still remains unsolved despite recent advances in training RNN (Recurrent Neural Network) - vanishing gradient and scalability. First, RNN suffers the difficulty of training by gradient-based optimization procedures. Second, capturing long-term dependencies is still a fundamental challenge for RNN. Unfortunately, Many proposals leveraging backpropagation are difficult to scale to long-term dependencies.

As the solution to the difficulties of coping with the long sequences, LSTM (Long Short-Term Memory) has been proposed for providing the resilience to gradient problems. Although there have been many successes by adopting chain-structured LSTM, many other

[*] National Institute of Informatics, Tokyo, Japan
[†] Musashino University, Tokyo, Japan

essential domains are inherently associated with input structures, which are more complicated than the input sequence itself. For example, it is pointed out that sentences in natural languages are believed to be carried by not merely a linear series of words; instead, semantics and their meaning are thought to be nonlinear structures. Zhu et al. [9] propose a new method for adopting memory blocks in recursive structures. It is called S-LSTM of which model utilizes the structures and performs better than chain-structured LSTM, ignoring such priori structures.

SGD (Stochastic gradient descent) is a method for interactively optimizing an objective function. SDG takes advantage of achieving smoothness properties for both differentiable and subdifferentiable. Mini-batch SGD is also regarded as a stochastic approximation of gradient descent optimization. Because mini-batch SDG replaces the actual gradient figured out from the entire data set by some parts of the whole estimation, which is calculated from a randomly picked up subset of data. Mini-batch SGD reduces the computational burden with faster iterations instead of a lower convergence rate, especially in high-dimensional optimization problems. Figure 1 depicts mini-batch SDG. Mini-batch SDG figures out the gradients on a few random sets of instances, which is described as mini-batches. As shown on the right side of Figure 1, the noise (nonlinearity) is added to the path of gradient descent. In some cases, mini-batches can provide the regularizing effect.

At each step, mini-batch SGD calculates the gradients on small random sets of instances while SGD computes the based on the full training set.

When we compute two gradients for the two data instances with each mini-batch, we need to divide them by two to obtain the gradient average over the mini-batches.

$$\theta_j = \theta_j - \varepsilon \frac{1}{n} \sum_{i=n*k}^{(k+1)*n} \bigtriangledown * \theta_j * DIFF(\hat{y}_i - y_i) \tag{1}$$

In equation (1), n is the batch size, and k is the number of batches. Mini-batch SGD has some advantages as follows:

1. Computational Efficiency: In terms of computational efficiency, this technique lies between the two previously introduced techniques.

2. Stable Convergence: Another advantage is the more stable converge towards the global minimum since we calculate an average gradient over n samples that results in less noise.

3. Faster Learning: As we perform weight updates more often than with stochastic gradient descent, in this case, we achieve a much faster learning process.

The algorithm's progress in parameter space is less erratic than with SGD, especially with fairly large mini-batches. As a result, mini-batch GD will end up walking around a bit to the minimum than SGD. But, on the other hand, it may be harder for it to escape from local minima.

Our hypothesis in this paper is as follows:

Hypothesis 1 (H2)**.** *Our tree-structured LSTM makes it possible to evaluate the tuning of hyperparameter of mini-batch size more correctly compared with conventional chain-structured one.*

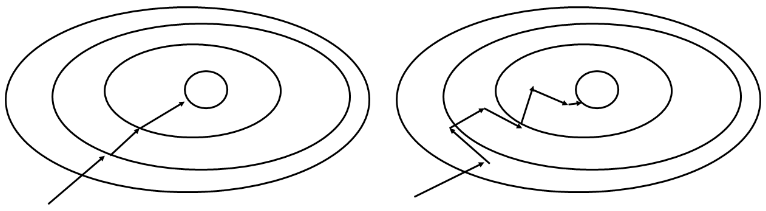

Figure 1: Mini-batch SGD. It figures out the gradients on a few random sets of instances, which is described as mini-batches.

Hypothesis 2 (H2)**.** *Our tree-structured LSTM takes advantage in measuring learning process's stability with small batch size the instability of overfitting with large batch size more precisely than chain-structured LSTM.*

Besides, one of the most important mathematical challenge of recurrent networks is long term dependencies.

Definition 1. *The basic problem of long-term dependencies is that gradients propagated across many layers tend to either vanish or explode.*

The thrust of recursive neural network over recurrent network is that for every sequence with the same length τ, the depth, which is the number of compositions of nonlinear operations can be radically reduced from reduced from τ to $O(log\tau)$. In this case, a recursive neural network might help deal with long-term dependencies.

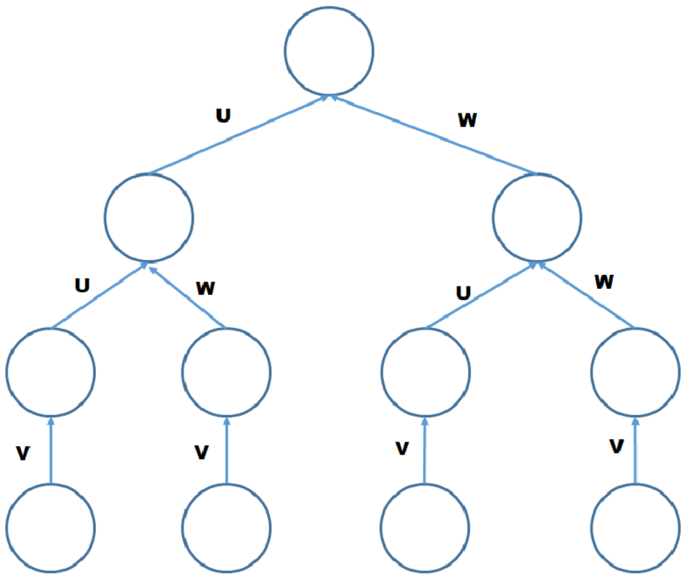

Figure 2: A recursive neural network as a generalization of the recurrent network from a chain to tree.

Figure 2 depicts a recursive neural network.

This is a computational graph that generalizes that of the recurrent network from a chain to a tree. In recurrent network, a variable-size sequence $x^{(1)}, x^{(2)}, ..., x^{(t)}$ is mapped to a fixed-size representation (the output o), with a fixed set of parameters (the weight matrices U, V, W). The figure illustrates a supervised learning case in which some target y is provided that is associated with the whole sequence.

2 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 recurrent networks' generalization with a different kind 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 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 complex structures. Tai et al [8] and Zhu et al. [9] extended chain-like structured LSTMs to tree-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 a l. [10] show the effectiveness of tree-structured LSTM on various tasks and situations in which tree-like structure is effective.

Truncated BPTT [17] is one of the most popular variants of BPTT. In [17], the accumulation stops after a fixed number of time s teps. Truncated BPTT performs well if the truncated chains are effective to learn the target recursive functions. Saon et al. [18] improved the original truncated BPTT with batch decoding. In [18], the number of context frames in batch decoding equals the number of unrolled steps before truncation.

Practical recurrent networks are combined of BPTT, batch decoding, and consecutive prediction [19] [20] for speeding up training. Besides, the hidden vectors are sometimes cached in [18]. These techniques sometimes cause a situation of mismatching between training and testing. So far, this mismatch has not been addressed. However, in [21], the distinction between online and batch decoding under running BPTT is explored.

3 Methogology

3.1 Truncated Backpropagation Through Time

Backpropagation Through Time, or BPTT, is a specific application of backpropagation in neural networks applied to sequence data like a time series. A recurrent neural network is shown one input each time step and predicts one output. Conceptually, BPTT works by unrolling all input time steps, as shown in Figure 2. Each time step has one input time step, one copy of the network s_t , and one output o_t. Errors are then calculated and accumulated for each time step with w. Figure 3 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.

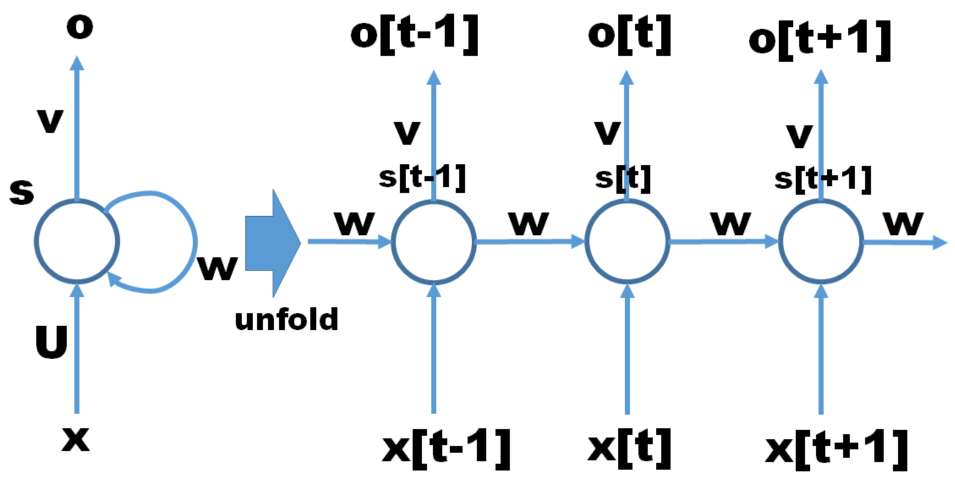

Figure 3: Back Propagation Through Time. It works by unrolling all input time steps.

Truncated Backpropagation Through Time (TBPTT), which is an online version of BPTT is proposed in [17]. TBPTT works analogously to BPTT, but the sequence is processed one time step at a time and periodically. The BPTT update is performed back for a fixed number of time steps. In [17], 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.

3.2 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:

$$f_t = \sigma(W_{fh}h_{t-1} + W_{fx}x_t + P_f * c_{t-1} * b_f) \tag{2}$$
$$i_t = \sigma(W_i x_t + U_i h_{t-1} + P_i * c_{t-1} * b_i) \tag{3}$$
$$g_t = tanh(W_g x_t + U_g h_{t-1} + b_g) \tag{4}$$
$$c_t = i_t \Theta g_t + f_t \Theta c_{t-1} \tag{5}$$
$$o_t = \sigma(W_o x_t + U_o h_{t-1} + P_o * c_t + b_o) \tag{6}$$
$$h_t = o_t \Theta tanh(c_t) \tag{7}$$

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 [30] enables the LSTM cell to inspect its current internal states. Then, the backpropagation of the LSTM at the current time step t is as follows:

$$\delta o_t = tanh(c_t)\delta h_t \tag{8}$$

$$\delta c_t = (1 - tanh(c_t)^2)o_t\delta h_t \tag{9}$$

$$\delta f_t = c_{t-1}\delta c_t \tag{10}$$

$$\delta c_{t-1} = f_t\theta\delta c_t \tag{11}$$

$$\delta i_t = g_t\delta c_t \tag{12}$$

$$\delta g_t = i_t\delta c_t \tag{13}$$

3.3 Implementation of Mini-batch SGD

In mini-batch SGD, we apply a batch of a fixed number of training data set which is smaller than the actual dataset. After generating the mini-batches of fixed size, we proceed to the following step in each epoch. Implementation of mini-batch SGD is as follows:

1. Pick a mini-batch
2. Feed it to Neural Network
3. Calculate the mean gradient of the mini-batch
4. Use the mean gradient we calculated in step 3 to update the weights
5. Repeat steps 1-4 for the mini-batches we created

The average cost over the epochs in mini-batch SGD changes because we are averaging the number of examples at once.

In general, mini-batch SGD is faster than SGD. Besides, mini-batch SGD has a smaller risk when measured in terms of CPU elapsed time.

4 Proposal Method

4.1 Tree-structured LSTM

The Tree-structured 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.

Figure 4 shows the comparison of two kinds of LSTM network structures. The upper side of Figure 4 shows a chain-structured LSTM network. The lower side of Figure 4 depicts a tree-structured LSTM network with an arbitrary branching factor. Tree-structured LSTM performs well when the networks need to combine words and phrases in natural language processing [8].

Definition 2 . *Tree-structured LSTM. Tree-structured LSTM can be represented as a computation graph that generalized that of the network from chain to a tree. In a recursive network, variable-size sequences x(1), x(2), ... x(t) can be mapped to a fixed-size representation, with a fixed set of parameters of weight.*

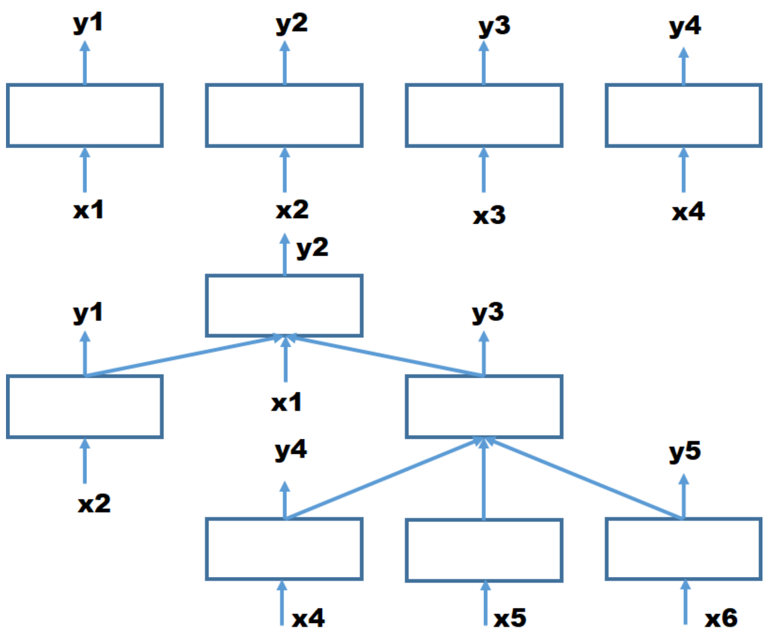

Figure 4: Tree-structured LSTM with arbitrary branching factor.

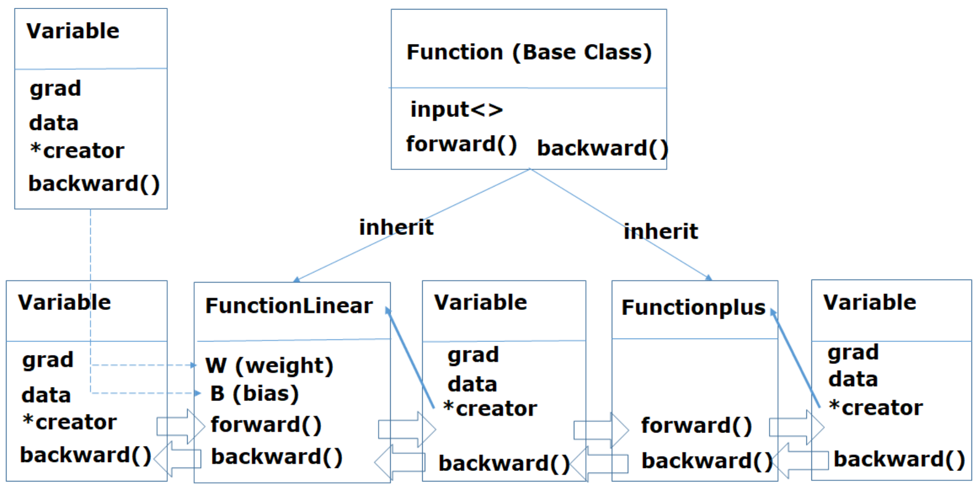

Figure 5: Implementation of Linear activation unit by object-oriented programming language. The base class of Function has the inheritance of FunctionLinear.

4.2 Reverse-mode Diff

Autodiff (Automatic differentiation) is the fundamental technology upon which most deep learning frameworks are based. Autodiff is one of the gradient-based techniques that deep learning models use, and autodiff enables it to calculate gradients, even from enormous and complicated models, easily. Reverse-mode autodiff which is implemented in Tensorflow [28] is a compelling and accurate technique, especially in the case that there are many inputs and few outputs. At the first phase of the reverse-mode diff, the program goes through the graph in the forward direction from the input to the output to calculate each mode's value. A second phase program goes through the reverse direction in turn to compute all the partial derivatives. Besides, reverse-mode diff can deal with functions defined by arbitrary code.

Figure 5 depicts our implementation of linear activation unit for the reverse-mode autodiff 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. 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.

Figure 5 also illustrates the detailed implementation of the inheritance of functions and variables of tree-structured LSTM. Inheritance lets us define classes that model relationships among types, sharing what is common and specializing only that which is inherently different. its derived classes inherit members defined by the base c lass. The derived class can use, without change, those operations that do not depend on the specifics of the derived type. It can redefine those member functions which do depend on its type, specializing the function to take into account the peculiarities of the derived type. Finally, a derived class may define additional members beyond those it inherits from its base class.

4.3 Constrained Recursion of Tree-structured LSTM

As we discussed in section I-B, a tree-structured LSTM graph is generated for each mini-batch. Figure 6 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 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 Figure 6.

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, which is also referred to as a search key. It then explores all of the neighbor nodes at the present depth before it goes on to the nodes at the next depth.

4.4 Deduplication of Multiple Graph Connection

At the first phase of LSTM, the current input layer x(t) and the previous short-term state h(t-1) are fed to four differently connected layers. Four connected layers are g (main gate), i (input gate), f (forget gate), and o (output gate). Here, let us call the multiple connections

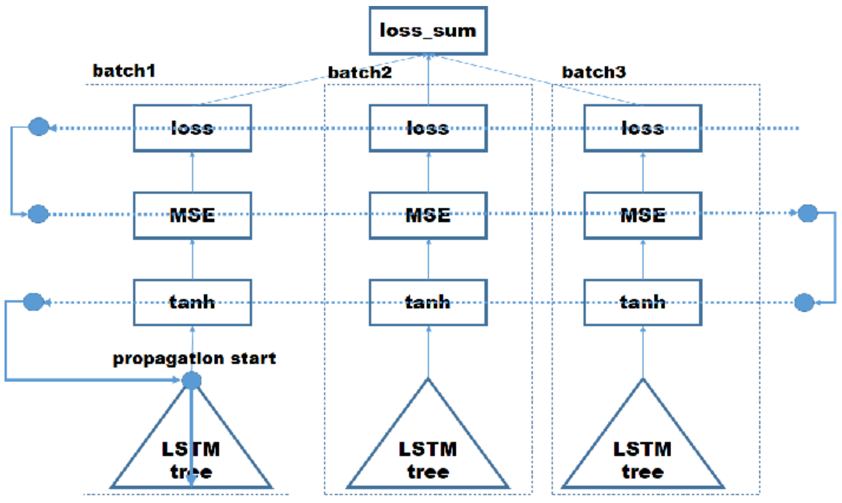

Figure 6: Constrained recursion of tree-structured LSTM.

Algorithm 1 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: $back\_propagation()$
```

as a branch, as shown in Figure 6. Without any constrain, the propagation between x(t) and these four layers are duplicated. We have a solution for the problem of this duplication with Algorithm 2.

In Figure 7, variable X has four branches reaching to g, i, f and o. In the forward propagation phase, variable x has been allocated four times as x(1), x(2), x(3) and x(4). The backpropagation from x will be caused four times to fx, gx, ix and ox without any constraint. To stop this duplicated propagation, we set the condition at line 4 of Algorithm 2.

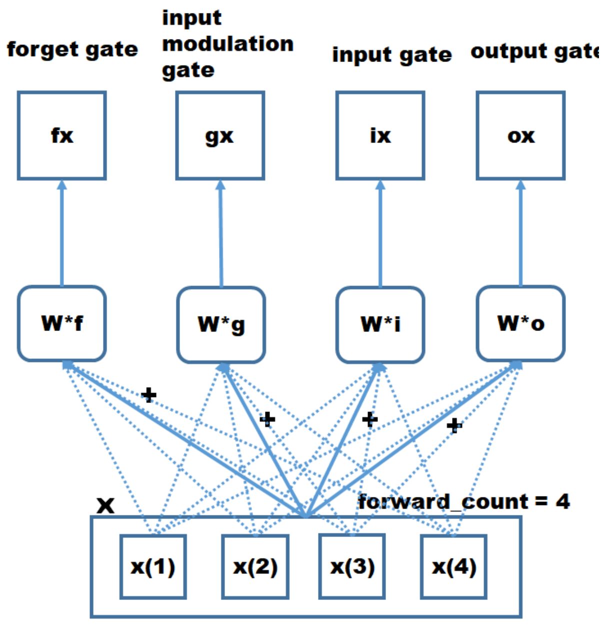

Figure 7: Deduplication of multiple graph connection. Without any constrain, the propagation between x(t) and these four layers are duplicated.

Algorithm 2 Deduplication of multiple graph connection

```
1: if $viv \rightarrow forward\_count > 0$ then
2:     $v \rightarrow forward\_count --$
3: end if
4: if $v \rightarrow forward_count \neq 0$ then
5:     $reutrn$
6: end if
7: $back\_propagation()$
```

5 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. Open source code for this experiment is available at [38].

We generated a sine wave with a length of 140 to 200 and a period of 4.

We generated the sine wave for training/testing data by C++ program as follows:

```
int steps_per_cycle = 50;
int number_of_cycles = 100;

void createSinData(float data[],
int steps_per_cycle,
int number_of_cycles){
for (int j=0; j<number_of_cycles; j++){
for (int i=0; i<steps_per_cycle; i++){
float v = std::sin(i * 2 *
std::atan(1) * 4
/ steps_per_cycle);
data[steps_per_cycle * j + i] = v;
}
}
}
```

Figure 8 shows 12 plots ranging from the interval model reset and batch size. X-axis is the length of the sine wave. Y represents the value of sin(X) ranging from -1 to 1. We divided 12 plots into three settings.

I Plot 1-4 has the parameter of interval=40 and batch size=3.

II Plot 5-8 has the parameter of interval=40 and batch size=6.

III Plot 9-12 has the parameter of interval=40 and batch size=10.

In each set I, II, and III, we changed the epoch size from 250 to 1000. It turned out that as the number of batch size is increasing, the prediction plots are likely to be collapsed. Specifically, in plot 12, the prediction plot becomes disturbed compared with the previous plot of 11.

Figures 9, 10, and 11 show the validation loss with epoch=1000 and interval = 40 and batch size ranging from 3, 6 to 10. In Figure 11 with batch size 3, the validation loss converges faster than batch size 6 and 10. However, the elapsed time of the processing with batch size 3 takes a longer time than Figure 9 and 10. As shown in Table I, it takes about four times (194min/59min) between batch sizes 3 and 10.

Table 1: Elapsed time with epoch 1000

bach size	elapsed time
10	3541.25sec(59.0208min)
6	5688.8sec(94.8133min)
3	11686.8sec(194.779min)

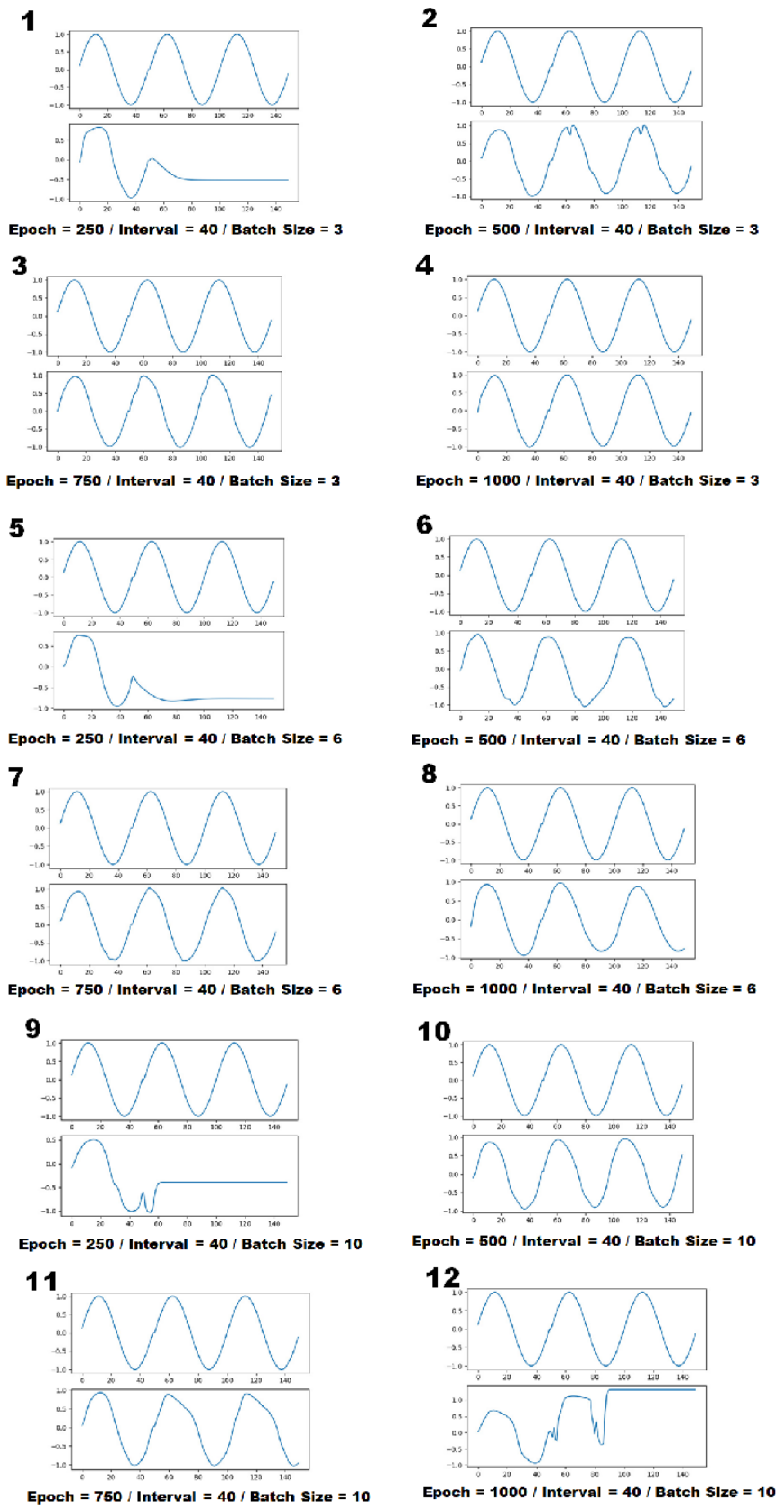

Figure 8: Experimental result of batch sizes 3, 6, and 10. For each batch size, epoch size is set to 250, 500, 750, and 1,000.

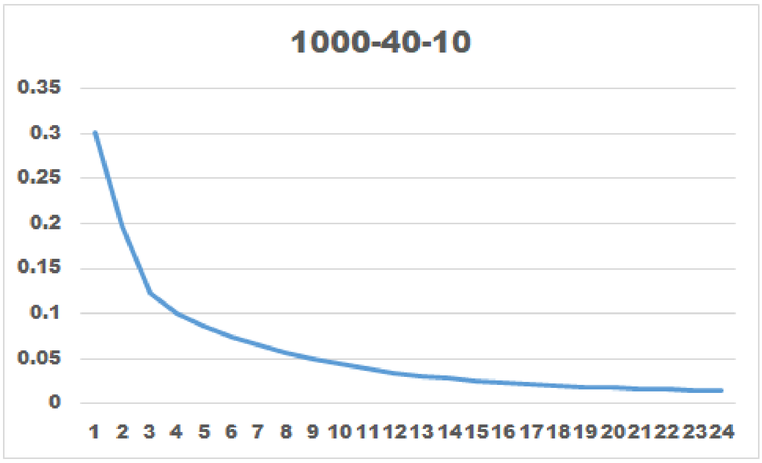

Figure 9: Validation loss with epoch = 1000, interval = 40 and batch size = 6.

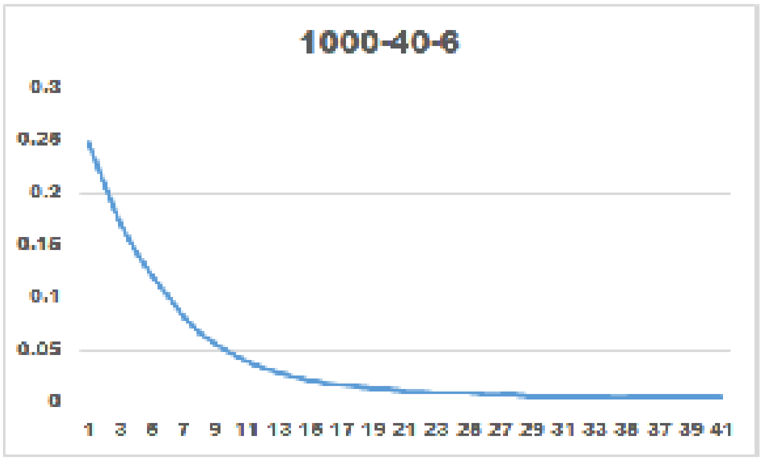

Figure 10: Validation loss with epoch = 1000, interval = 40 and batch size = 10.

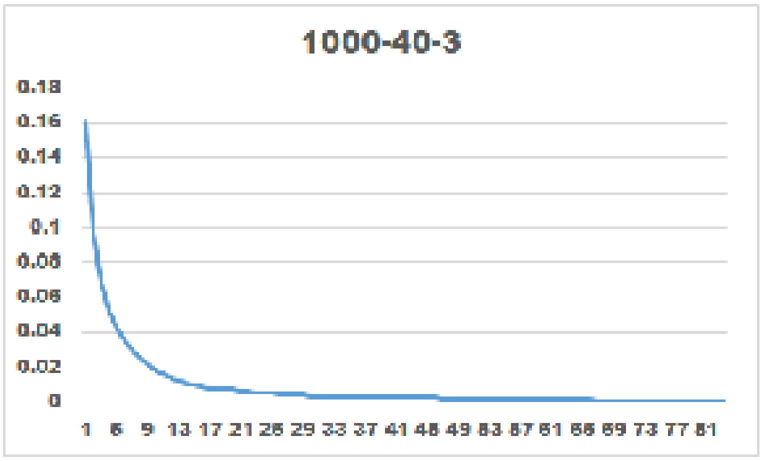

Figure 11: Validation loss with epoch = 1000, interval = 40 and batch size = 3.

6 Discussion

6.1 Result of Experiments

As we already know, when we apply mini-batch SGD, an alternative to batch gradient descent, we will cope with a new hyperparameter - batch size. To make our discussion more straightforward, we build up a hypothesis concerning the batch size.

Hypothesis 3. *Typically, the algorithm speeds up as batch size increases. On the other hand, an increase in batch size reduces the likelihood of successful convergence, especially as epochs increases in the long-term learning process.*

To test this hypothesis, we discuss the result in section 5 from short-term convergence (6.1.1) and long-term stability (6.1.2).

6.1.1 Convergence and over-learning (epoch 250, 500, 750).

Figure 8 has 12 subplots. Let us divide 12 subplots into three groups in Figure 8. Our main findings and insights in these three groups are as follows:

1. batch size = 3. (subplot 1-4): The learning process is slower (subplot 2 of epoch 250) but stable. (subplot of 4)
2. batch size = 6. (subplot 5-8): Result seems best in subplot 7 (epoch 750).
3. batch size = 10. (subplot 9-10): The learning process is faster (subplot 10 of epoch 250) but unstable (subplots 10 and 11).

Based on the observation of subplots 2, 6, and 10, the plot with batch size 10 seems best. However, at the end of subplot 10, the plot rises drastically. As possibly one of the guesses, it seems that the over-learning is caused, which is the typical case for long-size batches. Hence, we can conclude that the process with a large batch size of 10 tends to be more than batches 3 and 6. In addition, when we compare subplots 10 and 11 with batch size 10, the learning program gets out of local minima, and the learning process fails (from batch size 750 to 1000).

6.1.2 Stability in the long-term learning process (more than 1000 epochs)

There are three categories of GD (gradient descent): batch-GD, SGD (stochastic gradient descent), and mini-batch SDG. Three algorithms will converge near the minimum. But batch-GD's path ends up at the minimum, while both stochastic GD and mini-batch SGD continue to walk around.

Until it reaches the minima, the cost function of mini-batch SGD bounces up and down while one of batch GD gradually decreases until it researches the minima. Mini-batch SGD decreases on only average and continues to bounce around forever while batch-GD settles down to the minimum. In the long run, the probability of dropping out from the minima increases (batch size more than 1000).

Subplots 10, 11, and 12 in Figure 8 shows this tendency. In subplot 10, the program seems well on its way to the local minimum. However, the program went out of minima and collapsed through subplots 11 and 12. From this result of subplots 10, 11, and 12, we can conclude that a bigger batch size of more than ten often brings instability in the learning process compared to batch sizes 3 and 6.

Other interesting results are shown in Figures 9, 10, and 11. Figure 9, 10, and 11 compares the validation loss in the learning process concerning the case of batch size 3, 6, and 10.

1. batch sie = 10. (Figure 9): validation loss is 0.05 after nine epochs.
2. batch size = 6. (Figure 10): validation loss is 0.05 after nine epochs.
3. batch size = 3. (Figure 11): validation loss is 0.02 after nine epochs.

Curiously, in the long-term learning with epoch 1000, batch size three seems best in two aspects both convergence speed and stability. One possible reason is that small batches can offer a regularizing effect [31]. The algorithm's progress in parameter space with batch size 3 walks around closer to minima than the case of 6 and 10.

6.2 Advantage

As we noted in the previous section, recursive neural networks use a different kind of computational graph for adopting yet another generalization of recurrent networks. Originally, Pollack [2] introduces recursive neural networks. Bottou [3] shows the potential use for learning to reason. So far, what deep learning achieved is the ability to map set X to set Y using continuous geometric transformation with given large amounts of human-annotated data. This straightforward geometric morphing from space X to space Y of deep-learning models do is called as a local generalization. In the future, a necessary transformational development in the field of machine learning is to move away from local generalization with purely pattern recognition towards a model capable of extreme generalization with abstraction and reasoning. As we already know, a likely appropriate substrate in various situations is a computer program. Takefuji [32] points out that the importance of modularity and abstraction for the progress of open-source software such as Keras, chainer, and PyTorch. With software engineering progress, modularity and abstraction in software development will become a fundamental sense for achieving higher reusability with being more resilient against input/output/process interactions.

Our network with batch normalization can be described as a shallow learning network and similar to a functional link network. Pao and Takefuji [33] propose the functional link model, which eliminates all layers between input and output by using the single step of processing, is one way to avoid nonlinear learning. One benefit of a functional link neural network is flexibility when the learning time is based on the numerous processing elements necessary for computing.

Since the proposed method in this paper applies recurrent networks, it should be able to benefit from the above ability to reduce the depth, measured as the number of components of the nonlinear operation.

7 Conclusion

In this paper, we have proposed the constrained recursion algorithm for tree-structured LSTM. Our recursion algorithm's thrust is the ability of tree traversal over mini-batch SDB based tree structure LSTM. Besides, compared with chain-structured LSTM, tree-structured

LSTM has suitable for the validation of hyper-parameter tuning. Tree-structured LSTM has better modularity of the learning process.

Mainly, the hyperparameter of mini-batch SGD (Stochastic Gradient Descent) is one of the most important factors which decides the quality of the prediction of LSTM. In the experiment, we have measured 12 parameter tunings of sine curve fitting. It turned out that among three parameters of (1) epochs, (2) model reset interval, and (3) batch size, batch size is the most critical parameter for the stability of the learning process.

It turned out that the small batch size (=3) is the best parameter for after 1000 epochs. On the other hand, the big batch size (=10) seems faster to converge. However, after 750 epochs, the validation loss is increasing drastically. We can conclude that small batches can yield a regularizing effect. In other words, the noise which batch size 3 adds a regularizing product to the learning process. Our tree-based LSTM algorithm makes it possible such a hyperparameter tuning of batch size and provides a new perspective of parameter tuning. We have succeeded to measure the stability of the learning process with small batch size and the instability of overfitting with large batch size more precisely compared with chain-structured LSTM.

Our tree-structured LSTM is based on recursive network. Recursive networks have a variety of applications: A transformed data into a tree structure, mapping input and correct answers to individual nodes of that tree. The method proposed in this paper should be applicable to a variety of applications of recursive networks.

References

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

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

[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, EMNLP13, Seattle, USA, 2013.

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

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

[8] Kai Sheng Tai, Richard Socher, and Christopher D. Manning, “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, 2015, pp.1556-1566.

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

[10] Jiwei Li, Minh-Thang Luong, Dan Jurafsky, and Eduard Holy, "When are tree structures necessary for deep learning of representations", In Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing. EMNLP, 2015, pp.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, Oct. 1990, pp. 1550-1560,

[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), 1994, pp.157-166.

[14] John F. Kolen and Stefan C. Kremer, "Gradient Flow in Recurrent Nets: The Difficulty of Learning LongTerm Dependencies", Wiley-IEEE Press, 2001, pp464-479.

[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, ICML13,JMLR.org, 2013 pp. III310-III318.

[16] Yoshua Bengio, Nicolas Boulanger-Lewandowski, and Razvan Pascanu, "Advances in optimizing recurrent networks", In 2013 IEEE International Conference on Acoustics, Speech and Signal Processing, 2013, pp.8624-8628.

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

[18] George Saon, Hagen Soltau, Ahmad Enami, and Michael Picheny, "Unfolded recurrent neural networks for speech recognition", In Interspeech, 2014.

[19] Hasim Sak, Andrew Senior, and Francoise Beaufays, "Long short-term memory recurrent neural network architectures for large scale acoustic modeling", In Interspeech, 2014.

[20] Kai Chen, Zhi-Jie Yan, and Qiang Huo, "Training, deep bidirectional LSTM acoustic model for LVCSR by a context-sensitive-chunk BPTT approach", In Interspeech, 2015.

[21] Naoyuki Kanda, Mitsuyoshi Tachimori, Xugang Lu, and Hisashi Kawai, "Training data pseudo-shuffling and direct decoding framework for recurrent neural network based acoustic modeling", In Proc of IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU), 2015.

[22] Alan Graves, Abdel-rahman Mohamed, and Geoffrey Hinton, "Speech recognition with deep recurrent neural networks", in Proc of Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on. IEEE, 2013, pp. 6645-6649.

[23] Ilya Sutskever, Oriol Vinyals, and Quoc Le, "Sequence to sequence learning with neural networks", In Advances in Neural Information Processing Systems, 2014, pp. 3104-3112.

[24] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio, "Neural machine translation by jointly learning to align and translate", arXiv preprint arXiv:1409.0473, 2014.

[25] Tomas Mikolov, "Statistical language models based on neural networks", In Presentation at Google, Mountain View, 2nd April, 2012.

[26] Bergstra, J.S., Barnet, R., Bengio, Y., Kgl, B., 2011. Algorithms for hyper-parameter optimization, in: Shawe-Taylor, J., Zemel, R.S.,

[27] Jozefowicz, R., Zaremba, W., Sutskever, I., "An empirical exploration of recurrent network architectures", in: Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37, JMLR.org. 2015, pp.2342-2350

[28] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, Xiaoqiang Zheng: "TensorFlow: A System for Large-Scale Machine Learning". OSDI 2016: pp265-283

[29] Sergey Ioffe, Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". CoRR abs/1502.03167 (2015)

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

[31] D. Randall Wilson, Tony R. Martinez: The general inefficiency of batch training for gradient descent learning. Neural Networks 16(10): 1429-1451 (2003)

[32] Yoshiyasu Takefuji, "Open source software is indeed based on modularity and abstraction", Science, eLetter, July 24, 2017

[33] Yoh-Han Pao, Yoshiyasu Takefuji, "Functional-Link Net Computing: Theory, System Architecture, and Functionalities", Computer 25(5), 1992, pp76-79

[34] Keras: Deep Learning for humans, https://github.com/keras-team/keras

[35] Chainer: A deep learning framework, https://github.com/chainer/chainer

[36] Tensors and Dynamic neural networks in Python with strong GPU acceleration, https://github.com/pytorch/pytorch

[37] Paolo Frasconi, Marco Gori, Alessandro Sperduti, “On the Efficient Classification of Data Structures by Neural Networks”, IJCAI 1997, pp.1066-1071

[38] LSTM implementation for IJSCAI https://github.com/RuoAndo/LSTM-IJSCAI