Developing a novel recurrent neural network architecture with fewer parameters and good learning performance

Kazunori D Yamada [1,2]*

[1]Graduate School of Information Sciences, Tohoku University, Sendai, Japan [2]Artificial Intelligence Research Center, National Institute of Advanced Industrial Science and Technology (AIST), Tokyo, Japan

ABSTRACT

Recurrent neural networks (RNNs) are among the most promising of the many artificial intelligence techniques now under development, showing great potential for memory, interaction, and linguistic understanding. Among the more sophisticated RNNs are long short-term memory (LSTM) and gated recurrent units (GRUs), which emulate animal brain behavior; these methods yield superior memory and learning speed because of the excellent core structure of their architectures. In this study, we attempted to make further improvements in core structure and develop a novel, compact architecture with a high learning speed. We stochastically generated 30000 RNN architectures, evaluated their performance, and selected the one most capable of memorizing long contexts with relatively few parameters. This RNN, YamRNN, had fewer parameters than LSTM and GRU by a factor of two-thirds or better and reduced the time required to achieve the same learning performance on a sequence classification task as LSTM and GRU by 80% at maximum. This novel RNN architecture is expected to be useful for addressing problems such as predictions and analyses on contextual data and also suggests that there is room for the development of better architectures.

INTRODUCTION

Studies on neural networks and deep learning are currently expanding worldwide. The boom stems from the fact that neural networks have succeeded in realizing various complex tasks like processing images, understanding conversations, and writing sentences. The ability to understand conversations and write sentences is derived from the ability to understand contextual information. This is required for artificial general intelligence, which is an important goal of artificial intelligence research. Thus, methods to realize these abilities will be continuously developed and improved in the future. In the context of neural networks, these abilities are realized by recurrent neural networks (RNNs). The term RNN originally denoted a neural network that has recurrent loops in its structure. Various RNNs have been developed so far, including Elman networks (ENs) [1], echo state networks (ESNs), Boltzmann machines, long short-term memories (LSTMs) [2, 3], and gated recurrent units (GRUs) [4]. The most basic RNN is the EN, which has only one simple recurrent loop in its architecture. The ESN has a unique and unusual feature: the values of its weight parameters do not change from the initial random values. Instead, an ESN changes connections between neurons in the learning process. LSTM and GRU are popular architectures. Like EN, both LSTM and GRU exhibit recurrent loops in their architectures and can understand and handle contextual information of the input sequence data.

According to previous studies, the performances of LSTM and GRU, including the ability to retain contextual information for a long time and the learning speed of contextual information, are superior to that of EN. LSTM and GRU exhibit better performance by introducing gate structures to the architecture, mimicking the behavior of animal brains. Theoretically, the memory of LSTM can last forever because of the complex core algorithm based on gate structures, although this may not be necessarily true when we apply it to real world problems. The essential difference between EN and sophisticated methods like LSTM and GRU is the sophisticated core algorithm of the architecture, which undoubtedly improves the performance of RNNs. However, we believe that LSTM and GRU are not the best architectures for RNNs. At least, we cannot deny the possibility that there are better architectures than the LSTM and GRU because all possible architectures have not yet been examined. Of course, the topological space of core algorithms is too large to examine all instances. Therefore, in the present study, we developed a novel RNN architecture, showing better performance than some more sophisticated architectures. While developing a novel RNN algorithm, we focused on the compactness of architectures. Currently, many important advances of artificial intelligence are actually derived from deep learning, in which various neural network layers are stacked, thus increasing the number of parameters to achieve higher expressive power. This does indeed improve the performance of neural networks, and, in principle, we can increase the number of parameters indefinitely in this way. However, this is impossible in practice, as the physical limit to improving computer performance has nearly been reached [5]. Soon it will no longer be possible to increase the number of transistors on a microprocessor chip by reducing the size of transistors. Thus, we cannot improve computer performance by increasing the number of parameters, because this will slow down the processing speed. From this viewpoint, we

*To whom correspondence should be addressed. Tel: +81 22 795 4264; Email: kyamada@ecei.tohoku.ac.jp

© 2020 The Author(s)
This is an Open Access article distributed under the terms of the Creative Commons Attribution Non-Commercial License (http://creativecommons.org/licenses/by-nc/2.0/uk/) which permits unrestricted non-commercial use, distribution, and reproduction in any medium, provided the original work is properly cited.

believe that a more compact architecture is better than a larger one. Moreover, a compact architecture has a lower computational cost [6]. In this study, based on the above ideas, we attempted to design a more compact architecture than GRU and LSTM. The minimum requirements of the architecture developed in this study are as follows: relative to existing RNNs, the architecture has at least the same performance in terms of holding contextual information and has a high speed of learning based on a compact architecture.

In this study, we attempted to develop an architecture with good performance by generating candidate architectures randomly, evaluating them on a training dataset, and choosing the best one. Several studies based on such a concept can be found in the literature. For example, one study mutated the core algorithm of LSTM and investigated the performance of the newly generated RNNs through various experiments [7]. However, the number of core algorithms generated in this study was limited. In another study, similar to the previous one, numerous core algorithms were generated by mutating existing core algorithms such as LSTM and GRU, and the performance of the newly generated architectures was evaluated [8]. Because the new architectures were essentially variations on old ones, their performance was not always superior to that of existing RNNs: the search space was not sufficiently large. In addition, the study did not set any restrictions on the new architectures: for example, it did not focus on the compactness of the core algorithm. On the other hand, several studies have indeed focused on this. For example, one study developed an RNN, simplified LSTM (S-LSTM), by reducing the number of gates from LSTM [9]. In another study, minimal gated units (MGUs) reduced a gate from GRU [10]. These compact RNNs were rationally designed based on modifying the existing RNNs. In the current study, by contrast, the topology of the core algorithm was randomly generated without any assumptions such as starting from the existing RNNs or modifying them. In our study, to generate compact architectures, the number of parameters in the core algorithms was restricted. The LSTM and GRU algorithms described in the next section ("Recurrent neural networks") use four equations from $\mathbf{v}_2$ to $\mathbf{v}_4$ and three equations from $\mathbf{v}_1$ to $\mathbf{v}_3$ in their structures. In this study, we restricted the number of equations to two: $\mathbf{v}_1$ and $\mathbf{v}_2$. Finally, the developed architecture, YamRNN (Yamada modified RNN), was evaluated on several test datasets and its properties, especially the performance of learning speed of contextual information, were investigated.

RECURRENT NEURAL NETWORKS

Among the RNNs that can incorporate contextual information of input data, the simplest architecture is EN [1]. EN is formulated by the following equation:

$$\mathbf{h}_t = \tau(\mathbf{W}_a\mathbf{u}_t + \mathbf{W}_b\mathbf{h}_{t-1} + \mathbf{b}), \quad \textbf{(1)}$$

where $\mathbf{u}_t$, $\mathbf{h}_t$, $+$, τ, $\mathbf{W}$, and $\mathbf{b}$ denote an input vector for the RNN layer at timestep t, an output vector of timestep t (= internal memory vector of timestep t), an operator for the summation of matrices, a hyperbolic tangent function, a weight matrix, and a bias vector to be optimized. Generally, RNNs that can memorize contextual input sequences possess an internal memory vector $\mathbf{h}_t$ at each timestep. The vector is an output from an RNN unit, and at the same time, is maintained as a memory of the previous context. Although EN can memorize contextual data of input sequences by this mechanism, its performance is limited, mainly because of learning inefficiencies caused by the gradient vanishing problem. LSTM was developed to overcome this. LSTM has become one of the most famous RNN architectures [2]. The LSTM mimics the behavior of animal brains to improve performance in holding contextual information and in learning efficiency. The architecture of LSTM used in this study is defined by the following equations [3]:

$$\mathbf{v}_1 = \tau(\mathbf{W}_{1a}\mathbf{u}_t + \mathbf{W}_{1b}\mathbf{h}_{t-1} + \mathbf{b}_1), \quad \textbf{(2)}$$

$$\mathbf{v}_2 = \sigma(\mathbf{W}_{2a}\mathbf{u}_t + \mathbf{W}_{2b}\mathbf{h}_{t-1} + \mathbf{b}_2), \quad \textbf{(3)}$$

$$\mathbf{v}_3 = \sigma(\mathbf{W}_{3a}\mathbf{u}_t + \mathbf{W}_{3b}\mathbf{h}_{t-1} + \mathbf{b}_3), \quad \textbf{(4)}$$

$$\mathbf{v}_4 = \sigma(\mathbf{W}_{4a}\mathbf{u}_t + \mathbf{W}_{4b}\mathbf{h}_{t-1} + \mathbf{b}_4), \quad \textbf{(5)}$$

$$\mathbf{s}_t = \mathbf{v}_1 \odot \mathbf{v}_2 + \mathbf{v}_3 \odot \mathbf{s}_{t-1}, \quad \textbf{(6)}$$

$$\mathbf{h}_t = \mathbf{v}_4 \odot \tau(\mathbf{s}_t), \quad \textbf{(7)}$$

where $\mathbf{v}$, $\mathbf{s}_t$, $\odot$, and σ denote a temporal vector of the calculations, a unit for constant error at timestep t, an Hadamard product operator, and a sigmoid function. Equations **3** to **5** represent calculations of an input gate, a forget gate, and an output gate, which mimic behavior of animal brains. Because the values of $\mathbf{v}_2$, $\mathbf{v}_3$, and $\mathbf{v}_4$ range from 0 to 1, these vectors can also be considered gates when used as operands of the Hadamard product in equations **6** and **7**. (For example, if $\mathbf{v}_2 = \mathbf{0}$, then $\mathbf{v}_1 \odot \mathbf{v}_2 = \mathbf{0}$, and the gate is closed; if $\mathbf{v}_2 = \mathbf{1}$, then $\mathbf{v}_1 \odot \mathbf{v}_2 = \mathbf{v}_1$ and the gate is open.) Equation **6** denotes the constant error carousel, which prevents the gradient from becoming too small or large. The final equation **7** denotes the memory vector, which is also used as an output vector from the LSTM cell. Using this memory vector, LSTM can memorize the previous input data and achieve the ability of reading and interpreting contextual data.

The other popular architecture, GRU, is newer than LSTM [4]. The architecture of GRU is defined by the following equations:

$$\mathbf{v}_1 = \sigma(\mathbf{W}_{1a}\mathbf{u}_t + \mathbf{W}_{1b}\mathbf{h}_{t-1} + \mathbf{b}_1), \quad \textbf{(8)}$$

$$\mathbf{v}_2 = \sigma(\mathbf{W}_{2a}\mathbf{u}_t + \mathbf{W}_{2b}\mathbf{h}_{t-1} + \mathbf{b}_2), \quad \textbf{(9)}$$

$$\mathbf{v}_3 = \tau(\mathbf{W}_{3a}\mathbf{u}_t + \mathbf{W}_{3b}(\mathbf{v}_2 \odot \mathbf{h}_{t-1}) + \mathbf{b}_3), \quad \textbf{(10)}$$

$$\mathbf{h}_t = (\mathbf{1} - \mathbf{v}_1) \odot \mathbf{h}_{t-1} + \mathbf{v}_1 \odot \mathbf{v}_3, \quad \textbf{(11)}$$

where $\mathbf{1}$ denotes the matrix (vector) of ones. There are a few differences between the core algorithms of LSTM and GRU. The first difference is the number of parameters used in the architectures. The core algorithm of LSTM has four intermediate vectors represented by $\mathbf{v}$ in equations **2** to **5**. In the equations, there are a total of eight weight parameter matrices and four bias vectors. In contrast, GRU has three intermediate vectors, and the equations of the vectors have a total of six weight parameter matrices and three bias vectors. Thus, the number of parameters of GRU is 75% of that of LSTM. The second difference between the architectures is the

topology of the variables in the core algorithm. For example, equations **3** to **5** define the gate structure in LSTM, and they are utilized to calculate only the memory vectors ($\mathbf{s}_t$ and $\mathbf{h}_t$). By contrast, the gate structures are used to generate both an intermediate vector and a memory vector in GRU. In addition, the equations or connections between variables to generate the output vector (equations **7** and **11** in LSTM and GRU are largely different from each other. Collectively, these differences result in different performances of RNNs.

In addition to these popular RNNs, there are a few RNNs of compact architecture. The architecture of S-LSTM is defined by the following equations:

$$\mathbf{v}_1 = \sigma(\mathbf{W}_{1a}\mathbf{u}_t + \mathbf{W}_{1b}\mathbf{h}_{t-1} + \mathbf{b}_1), \quad \textbf{(12)}$$

$$\mathbf{v}_2 = \tau(\mathbf{W}_{2a}\mathbf{u}_t + \mathbf{W}_{2b}\mathbf{h}_{t-1} + \mathbf{b}_2), \quad \textbf{(13)}$$

$$\mathbf{s}_t = (\mathbf{1} - \mathbf{v}_1) \odot \mathbf{v}_2 + \mathbf{v}_1 \odot \mathbf{s}_{t-1}, \quad \textbf{(14)}$$

$$\mathbf{h}_t = \tau(\mathbf{s}_t). \quad \textbf{(15)}$$

This architecture is quite similar to LSTM. It removes the output gates and replaces the input gate by the forget gate in the form of $\mathbf{1} - \mathbf{v}_1$. On the other hand, the architecture of MGU is defined by the following equations:

$$\mathbf{v}_1 = \sigma(\mathbf{W}_{1a}\mathbf{u}_t + \mathbf{W}_{1b}\mathbf{h}_{t-1} + \mathbf{b}_1), \quad \textbf{(16)}$$

$$\mathbf{v}_2 = \tau(\mathbf{W}_{2a}\mathbf{u}_t + \mathbf{W}_{2b}(\mathbf{v}_1 \odot \mathbf{h}_{t-1}) + \mathbf{b}_2), \quad \textbf{(17)}$$

$$\mathbf{h}_t = (\mathbf{1} - \mathbf{v}_1) \odot \mathbf{h}_{t-1} + \mathbf{v}_1 \odot \mathbf{v}_2. \quad \textbf{(18)}$$

This architecture is quite similar to GRU. The parameters of these architectures include four weight parameters and two bias parameters. Thus, the number of parameters of S-LSTM and MGU is 50% of that of LSTM.

METHODS

Generation of RNN architectures

Herein, we generated various core algorithms of RNNs at random. In the generation phase, all stochastic processing followed a uniform distribution. The core algorithms were generated in a completely random way, except that they were constrained to meet the minimum requirements of having at least two weight parameters and a recurrent loop. The RNNs were generated according to the following rules:

1. The architecture has to include intermediate vectors, $\mathbf{v}_1$ and $\mathbf{v}_2$.
2. $\mathbf{v}_2$ has to be calculated after $\mathbf{v}_1$ is calculated.
3. The memory vector of the next timestep, $\mathbf{h}_{t+1}$, has to be calculated after $\mathbf{v}_2$ is calculated.
4. The calculation of $\mathbf{v}_1$ and $\mathbf{v}_2$ must at least include a matrix product between the weight parameter matrix $\mathbf{W}$ and the input vector, $\mathbf{u}_t$ or the memory vector of the current timestep, $\mathbf{h}_t$. The bias parameter vector, $\mathbf{b}$, is stochastically included.
5. To calculate $\mathbf{v}_2$, it is decided stochastically whether $\mathbf{v}_1$ is used. If $\mathbf{v}_1$ is used, $\mathbf{v}_1$ is transformed to $\mathbf{u}_t$ or $\mathbf{h}_t$ using any operator. The operators include summation ($+$), subtraction ($-$), and Hadamard product ($\odot$) of vectors. The operator is stochastically chosen.

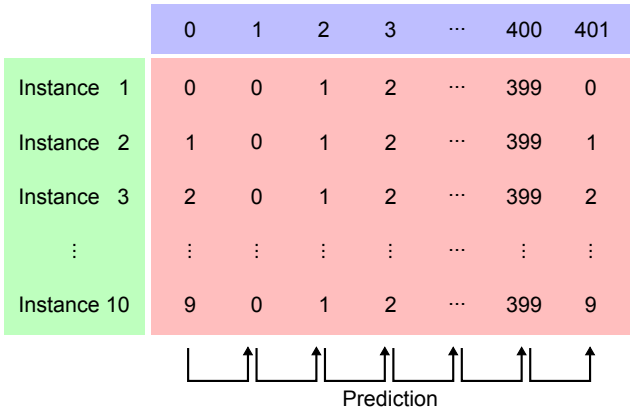

Figure 1. Illustration of the screening dataset. The numbers in the purple box denote the timestep t. In each instance, the initial and final elements are the same. The middle elements are common for all instances. The value of the next timestep is predicted using RNNs based on the value of the previous output as the input.

6. The equations to calculate $\mathbf{v}_1$ and $\mathbf{v}_2$ are required to include a sigmoid (σ) or hyperbolic tangent (τ) as an activation function.
7. $\mathbf{h}_{t+1}$ is calculated using $\mathbf{v}_1$, $\mathbf{v}_2$, $\mathbf{h}_t$, and operators. However, at this time, if $\mathbf{h}_t$ has already been used to calculate $\mathbf{v}_1$ or $\mathbf{v}_2$, it is not necessarily used again for the calculation of $\mathbf{h}_{t+1}$; whether or not to use it is decided stochastically. Similarly, if $\mathbf{v}_1$ has already been used for the calculation of $\mathbf{v}_2$, the decision whether or not to use it again for the calculation of $\mathbf{h}_{t+1}$ is made stochastically.
8. As operators to calculate $\mathbf{h}_{t+1}$, summation ($+$), subtraction ($-$), Hadamard product ($\odot$) of matrices, and matrix product ($\times$) are randomly chosen when variables $\mathbf{u}$, $\mathbf{v}_1$, $\mathbf{v}_2$, or $\mathbf{h}_t$ are stochastically chosen. In addition, each variable can be subtracted from a vector of ones ($\mathbf{1}-$) and can be squared in an elementwise manner. It is stochastically decided whether these operations are executed.
9. In the calculation of $\mathbf{h}_{t+1}$, the variables $\mathbf{u}$, $\mathbf{v}_1$, $\mathbf{v}_2$, or $\mathbf{h}_t$ can be repeatedly selected without any restrictions. The recruitment of the variables is stochastically terminated.

Based on these rules, we stochastically generated 30000 architectures. We confirmed that each architecture had a unique topology and was different from the other architectures.

Dataset

Screening and validation dataset As a screening dataset for the generated RNNs, we utilized the simplest and most general dataset we could in order to reduce the possibility of biased results due to specific datasets. Thus, we generated a dataset including 10 instances. Each instance had a length of 402 timesteps and consisted of a sequence of integers between 0 and 399, one integer per timestep. The integer corresponding to timestep 0 of instance n (where $n=0,1,\ldots,9$) was chosen to be simply n. The same value n was also assigned to timestep 401 of instance n, so the first element of each instance was the same as the last. The integers corresponding

to timesteps 1 to 400 did not depend on the instance but obeyed the simple rule: timestep t corresponds to the integer $t-1$, where $t=1,2,\ldots,400$. The difference between each instance in the dataset was therefore only the initial and final elements. Thus, if we face a problem in which the predictor of RNNs predicts a value of timestep $t+1$ by accepting a value of timestep t as an input vector, and if the predictor succeeds in predicting the final value, the predictor can at least hold 400 contextual elements. Therefore, this dataset is the most general benchmark dataset for the evaluation of RNNs that can memorize contextual information.

As a validation dataset for screened RNNs, we utilized the Modified National Institute of Standards and Technology (MNIST) [11] dataset. The MNIST is a famous benchmark dataset for classification of handwritten digits. Because instances in the dataset were image data with 28 rows and 28 columns, we regarded an instance as a sequential datum with a length of 28, in which each element consisted of a 28-dimensional vector. The number of instances in the dataset is 70000, and we utilized all of them for the validation process.

Test dataset As the test dataset for the developed RNN and other existing RNNs, we utilized S&P 500 stock data obtained from Kaggle (https://www.kaggle.com/camnugent/sandp500), which contained historical stock data for all S&P 500 companies for five years. From the original dataset, we removed nine instances including "BHF", "DHR", "ES", "FTV", "O", "REGN", "UA", "VRTX" and "WRX" because they contained missing values. The instances in the dataset are time series data. Each timestep in an instance corresponds to a five-dimensional vector in which each element can take the values "open", "high", "low", "close", and "volume". From the original dataset, in which an instance has a sequential vector structure with a length of 1259, we randomly extracted data of length 100, 300, and 500 for the test dataset by not allowing any overlap between sequences. We named these datasets test dataset 1, test dataset 2, and test dataset 3, respectively; each dataset included 1340, 851, and 631 instances, respectively. The values of each instance were standardized to z-values. For the dataset, RNNs predicted "up" or "down" of "close" values by reading the previous contextual information before the date of prediction. We chose this dataset because predicting stock price is known to be difficult, and thus we could easily recognize differences of learning performance between the architectures. Moreover, we could change the sequence length of instances freely (100, 300 and 500) to measure and compare the performance of different RNNs as a function of sequence length.

Screening and validation of architectures

The 30000 newly generated architectures were screened using the screening dataset. The size of the weight parameter was fixed as 24576, i.e., the product of the size of the bias parameter (512) and the length of the input vector (48), as explained below. In addition to the RNN layer, only one fully connected layer was added to produce an output vector of a suitable size for the given classification problem. The input value, which was an integer from 0 to 399, was encoded to a floating value vector of 48 dimensions. All weight and bias parameters were randomly initialized by the uniform distribution $U(-0.2,0.2)$. The size of the minibatch was fixed as the instance number, 10, and parameter update was conducted by the batch update method. As an optimizer of the gradient descent method, Adam was utilized [12]. In the process, for every candidate architecture, the value of the prediction accuracy was recorded for every epoch. If the accuracy was serially recorded as 1.0 for 10 epochs, the architecture was regarded as having "succeeded" and the epoch was recorded. This computation was repeated five times for each architecture. If the epoch exceeded 50000 times, the architecture was regarded as having "not succeeded" and the computation was stopped.

For the validation process, almost all settings were the same as those in the screening process. The differences were the encoding process of the input vector and the size of the minibatch. In the validation process, because the value of the input vector in the MNIST dataset was not an integer (categorical) value, the encoding process was not performed. The size of the minibatch was set to 1000. In addition, the threshold of the prediction accuracy was set to 0.999 instead of 1.0, and the computation was repeated 10 times because the computational burden was not so heavy for the problem.

Investigation and test conditions

The performance of YamRNN, the architecture we developed, was investigated from various aspects. It was examined on the screening dataset by changing the hyperparameter settings such as the size of weight and bias parameters, the minimum and maximum value of uniform distribution for parameter initialization, and the number of instances in the dataset. More details are provided in the next section ("Results and discussion") for each investigation.

The performance of YamRNN was compared with that of existing RNNs on the test datasets. The size of the weight parameter was changed from 256×5 to 1024×5 with an increment of 256×5, where "5" comes from the five values "open", "high", "low", "close", and "volume" of an element of an instance in the test dataset. The size of the bias parameter was changed from 256 to 1024 with an increment of 256. In addition to the RNN layer, only one fully connected layer was added to produce the output vector of a suitable size for the classification problem, i.e., two ("up" or "down"). All weight and bias parameters were randomly initialized by the uniform distribution $U(-0.2,0.2)$. The size of the minibatch was set to 300. As an optimizer of the gradient descent method, Adam was utilized. In the process, the value of the prediction accuracy was recorded for every epoch. If the accuracy was serially recorded 10 epochs as 0.7, the trial was regarded as having "succeeded", and the epoch was recorded. In this case, we set the threshold value to 0.7 because of the problem's high level of difficulty; it was difficult for any architecture to reach higher accuracy than approximately 0.7. The computation was repeated 10 times for each test. If the epochs exceeded 50000 times, the trial was regarded as having "not succeeded", and the computation was stopped. The number of successful trials was counted, and, in addition to the final epoch, the actual computation time (seconds) was recorded.

Computation environment

As a framework to implement the neural networks, we used Theano version 1.0.4 [13] with CUDA version 10.0 (NVIDIA) and cuDNN version 7.5 (NVIDIA). All calculations were performed on the NIG supercomputer at ROIS National Institute of Genetics with Tesla V100 (NVIDIA) as a GPU accelerator.

RESULTS AND DISCUSSION

Development of a novel architecture

In this study, we screened 30000 newly generated RNN architectures by benchmarking them on the screening dataset. Before the screening process, we evaluated the performances of existing method, such as LSTM and GRU, on the dataset. The results are shown in Figure 2. The vertical axis includes the number of epochs for each RNN predictor to reach the threshold accuracy (=1.0). Thus, a predictor with a small value on the vertical axis denoted that the predictor had a high learning speed for the input sequence. As seen from the data shown in Figure 2 the learning speed of GRU was found to be superior, which is consistent with the literature data [10]. Longer contextual information requires more time to learn. From these results, we could elucidate the approximate time required for learning with the RNNs. We tried to screen the architectures on as long a sequence dataset as possible; by considering the expected computational burden and the performance of our computer environment, we set the length of the context used for the screening process to 400. In addition, we set the maximum limit of screening time to 50000 epochs to save computational resources. The objective of the study was to develop an RNN that had a compact core algorithm (smaller number of parameters) and thus showed good learning speed with descent memorizing performance. First, we generated 30000 RNN architectures, based on the procedure described in the "Methods" section. The algorithms of each architecture were unique. Next, the performance of the architectures was examined on the screening dataset. Thus, 21148 out of 30000 architectures did not work as normal RNNs because they did not return valid values of cost and prediction accuracy. Only 579 of the remaining 8852 RNNs passed all the other screening criteria.

Next, we proceeded to a second screening process, which we call the validation process. As described in the "Methods" section, we utilized the MNIST dataset as contextual data consisting of 28 timesteps. In contrast to the screening dataset, which was a simple simulation dataset of longer contexts, the MNIST dataset had shorter contextual information, was derived from real world data, and included numerous instances. By using different validation datasets, we expected the RNNs remaining after two screenings to show more universal performance. As in the first screening, we first examined the performance of LSTM and GRU on the dataset before conducting the validation experiment. As shown in Table 1, the mean numbers of epochs for LSTM and GRU to reach the threshold accuracy (= 0.999) were 30.1 and 25.3, respectively. Again as in the previous experiment on the screening dataset, the learning speed of GRU was superior to that of LSTM. Subsequently, we conducted a validation experiment. Although none of the RNNs that went through the validation process output an invalid cost value or accuracy, only 304 architectures completed the computation in 50000 epochs. Because the number of epochs of GRU on the dataset was approximately 25, first we attempted to find architectures whose number of epochs was less than 25. At this time, we sorted the candidate RNNs in ascending order of the cost value. However, none of the surviving architectures recorded a number of epochs less than that of GRU. Therefore, we changed the threshold to 30 and further to 40; finally, we found an RNN architecture that recorded 36.0 as the number of epochs (Table 1). We gave name "YamRNN" to the one RNN architecture that succeeded in our study. The core algorithms of YamRNN were defined using the following equations:

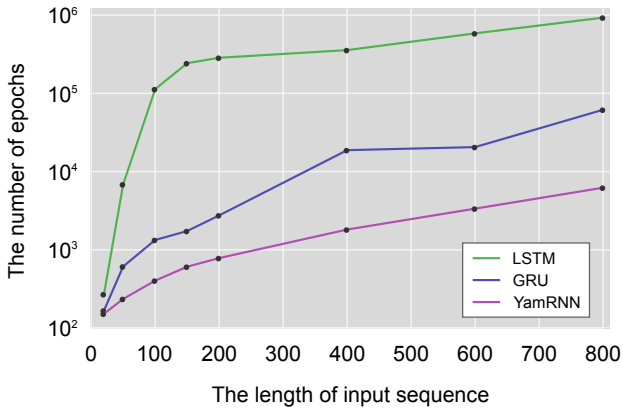

Figure 2. Comparison of learning speeds of RNNs on the screening dataset. The horizontal axis represents the size of units of weight and bias parameters for each RNN, and the vertical axis represents the number of epochs for the RNN predictors to serially reach the threshold accuracy (= 1.0) 10 times.

Table 1. Number of epochs for RNNs to reach the threshold accuracy on the validation dataset. The value was the mean of 10 trials. Three significant digits are displayed.

Architecture	The number of epochs
LSTM	30.1
GRU	25.3
YamRNN	36.0

$$\mathbf{v}_1 = \sigma(\mathbf{W}_{1a}\mathbf{u}_t + \mathbf{W}_{1b}\mathbf{h}_{t-1} + \mathbf{b}_1), \quad \textbf{(19)}$$

$$\mathbf{v}_2 = \tau(\mathbf{W}_2\mathbf{u}_t + \mathbf{b}_2), \quad \textbf{(20)}$$

$$\mathbf{h}_t = (1-\mathbf{v}_2)^2 - (\mathbf{h}_{t-1} + \mathbf{v}_2^2) \odot \mathbf{v}_1, \quad \textbf{(21)}$$

where $\mathbf{x}^2$ denotes $\mathbf{x} \odot \mathbf{x}$. The number of parameters of YamRNN is compared with the existing architectures in Table 2. As described, YamRNN has three weight and two bias parameters. The number of parameters of YamRNN is less than 50% that of LSTM and 67% that of GRU; it is also smaller than that of existing compact architectures such as S-LSTM and MGU. Equation **19** includes a sigmoid function, which can be considered to work as a gate, like those of existing RNNs. Next, we benchmarked YamRNN on the screening dataset by changing the length of the input

sequences. The performance of YamRNN on the dataset is shown by a purple line in Figure 2. As shown, the learning speed of YamRNN is superior to that of LSTM and GRU for any input length. This tendency is especially prominent for longer input sequences, and the learning speed of YamRNN was approximately 150 and 9.92 times that of LSTM and GRU, respectively, on the dataset in which the length of the input sequence was 800.

Table 2. Comparison of the number of weight and bias parameters in the equations of each RNN

Architecture	Weight parameters	Bias parameters
LSTM	8	4
GRU	6	3
S-LSTM	4	2
MGU	4	2
YamRNN	3	2

Investigating the performance of YamRNN

Next, we investigated the performance of YamRNN by changing the hyperparameter settings, including the number of units of RNN layer, the parameter of uniform distribution, and the number of instances in a dataset. In the experiments, we benchmarked the performance of YamRNN on the datasets, which were similar to the screening dataset but had different input-sequence lengths: 50, 100, 200, 400, or 800. If the accuracy was recorded as 1.0, 10 times serially, computation was stopped and the computation time (seconds) was recorded. The conditions other than the changed hyperparameter settings in each experiment were the same as those of the first screening. First, we benchmarked the performance of YamRNN by changing the number of units in the layer of the architecture. The benchmark results are shown in Figure 3a, where the five columns of different colors denote the computation time of learning by YamRNN on datasets of sequence lengths 50, 100, 200, 400 and 800 respectively. As the length of the instances increased, larger computation times were required. In contrast, the learning speed increased as the number of units in the layer increased. Generally, a model with more parameters has more expressive power. In this experiment, the learning process was completed quickly because of the high expressive power, so the calculation time of the model with larger number of parameters could be reduced. This tendency was observed in the subsequent test experiments described in the next section. Next, we conducted benchmarking by changing the parameter of uniform distribution used to initialize the weight and bias parameter of models. As shown in Figure 3b, the computation time increased depending on the length of the instances in each dataset. Regarding the change of parameters of uniform distribution, when we set the distribution as $U(-0.4,0.4)$, the computation time on almost all datasets was minimized. This means the learning speed was maximized on the problem when the parameter of uniform distribution was set to 0.4. However, we must note that the parameter of uniform distribution is only a hyperparameter, and the result is not a universal setting; the value of 0.4 was just used for reference. Finally, we performed benchmarking by changing

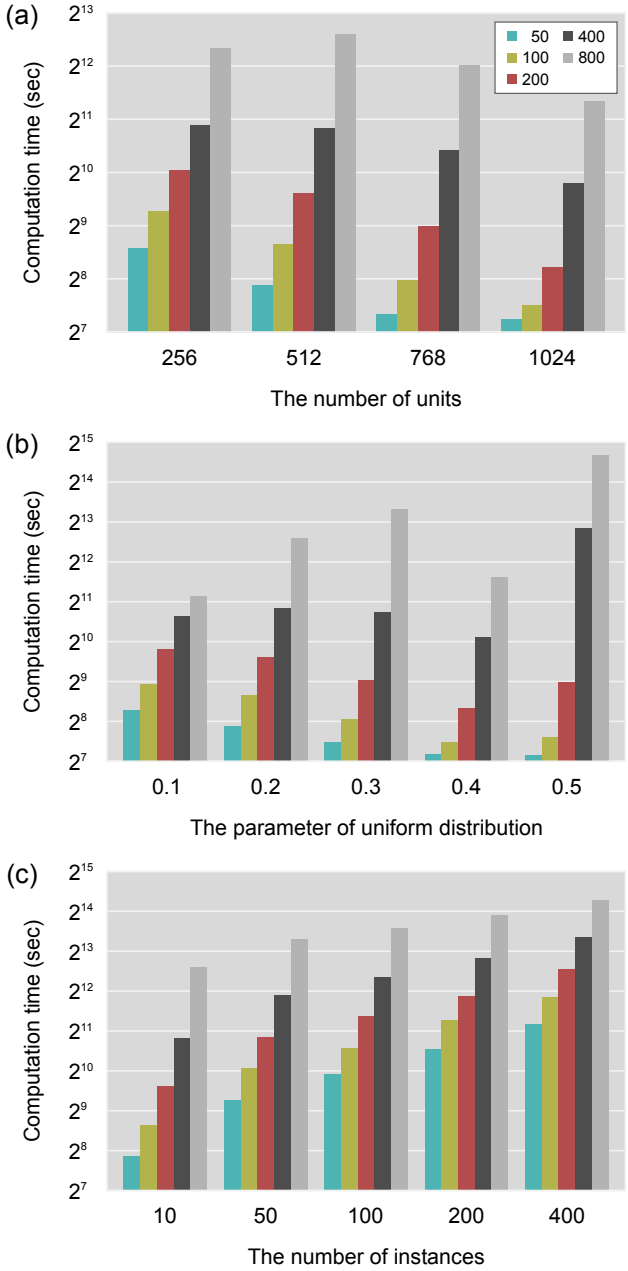

Figure 3. Comparison of learning speed of YamRNN under various conditions. The vertical axis represents the computation time for a model to reach threshold accuracy (= 1.0) 10 times serially on datasets, where instances in each dataset have different lengths from 50 to 800, represented by columns of different color. (a) The horizontal axis represents the number of units on weight and bias parameters. (b) The horizontal axis stands for the parameter of uniform distribution, by which the parameters of a model are initialized. (c) The horizontal axis represents the number of instances in the dataset.

the size of the dataset. As shown in Figure 3c, when the number of instances in the dataset increased, the computation time also increased. This result is unsurprising, because the computation load naturally increases with increase of the dataset size, and the problem becomes more difficult depending on the dimension of the output vector, which is the same as the number of instances.The dimensions of the output

vector on a dataset with 10 instances and 400 instances were 10 and 400, respectively. Collectively, by investigating the performance of YamRNN, we confirmed that the performance of the architecture changed proportionally depending on the change of parameters. Thus, the behavior of the architecture against parameter changes was considered to be normal.

Comparison to the existing methods

As the final part of the study, we compared the performance of YamRNN on the test datasets to that of existing architectures such as LSTM, GRU, S-LSTM and MGU. As in the previous experiments, the criterion to evaluate the goodness of architectures was the learning speed, as reflected in the number of epochs and the actual computation time (seconds) to reach the threshold accuracy. As described in the "Methods" section, in the test, we computed the learning speed on the five architectures by changing the number of units of the RNN from 256 to 1024 at an increment of 256. The number of times the computation was completed out of 10 trials was recorded. We judged the computation as completed when the prediction accuracy recorded the threshold accuracy (0.7) serially 10 times within the threshold number of epochs (50000). First, we conducted benchmarking on the test dataset 1, where the length of input sequences was 100. As shown in Table 3, almost all architectures completed the computation within the threshold number of epochs on each trial. When the number of units was 256, GRU and MGU showed the best learning speed. If the number of parameters of LSTM, GRU and MGU was 512, and that of S-LSTM was 768, the learning speed reached the best value; if the size was increased further, the performance deteriorated because the optimization phase of the increased parameters required more time. By contrast, the learning speed of YamRNN worsened with smaller sizes of parameters (256) but was excellent for larger sizes (768 and 1024). We could interpret the results as follows: YamRNN only achieved adequate expressive power at a certain parameter size. Actually, the number of parameters (both weight and bias) of YamRNN on 256 units was 4352, only about half that of GRU (8448). Collectively, the learning time required to achieve the same learning performance as that of LSTM and GRU, currently the most popular RNN, was shortened at maximum by 85% and 80% on the test dataset when the number of units was set to 1024. Next, we conducted the same benchmarking on the test dataset 2, where the length of the input sequences was 300. As shown in Table 4, the problem was more difficult than the previous test because the success count was smaller and the number of epochs and computation time were larger than the previous results. Note that LSTM and its derivative, S-LSTM, failed to complete the computation in most cases. In addition, GRU failed to complete the computation when the parameter size was 1024, possibly because the threshold time for the problem was not long enough for GRU to optimize parameters. As before, the learning speed of YamRNN was inferior to that of GRU and MGU when the number of parameters was small (256). By contrast, if the parameter size was large, the learning speed of YamRNN was superior to those of the other methods. Surprisingly, the learning speed of YamRNN did not diminish despite the increase in the number of parameters. This property was not seen in any of the other architectures and suggests YamRNN 's excellent potential. Finally, we conducted benchmarking on test dataset 3, where the length of the input sequences was 500. As shown in Table 5, the problem was more difficult than either of the other two tests. Neither LSTM nor S-LSTM succeeded in completing the computation on the tested parameter settings and even GRU and MGU could not finish the computation when the size of the parameter was large (768 and 1024). By contrast, YamRNN could complete the computation when the size of the parameter was large, although it failed to do so when the parameter size was small. This was once again because of the limited expressive power of YamRNN in the small parameter setting. When the parameter size was large, the learning speed of YamRNN was stably high. This tendency was one of the most outstanding features of YamRNN. As shown in the three test experiments, the learning speed of YamRNN was superior to those of the existing methods if the number of parameters was set to be large enough. The learning speed of the existing RNNs decreased depending on the increase in the number of parameters, resulting in unsuccessful completion of computation. The learning speed of YamRNN continued to increase depending on the increase of parameter size. This phenomenon could reflect the fundamental differences between YamRNN and the other architectures, caused by the difference in topology of the core architectures. At least, the high performance of YamRNN was not just because of the smaller number of parameters. In the test problems, the number of parameters of GRU on 512 units was 16896 and that of YamRNN on 1024 units was 17408. As shown in Table 4, the learning speed of GRU on 512 units (9520 seconds) was clearly

Table 3. Results on test dataset 1, where the length of input sequences was 100. The number of epochs and computation time is the mean value of the successful trials. The best values for each category are shown in bold. Three significant digits are displayed.

	The number of units			
	256	512	768	1024
Success count out of 10 trials				
LSTM	**10**	**10**	**10**	**10**
GRU	**10**	**10**	**10**	**10**
S-LSTM	9	9	**10**	**10**
MGU	**10**	**10**	**10**	**10**
YamRNN	**10**	**10**	**10**	**10**
Number of epochs				
LSTM	2900	1430	2970	4520
GRU	2160	1950	2140	4090
S-LSTM	4350	4940	3480	5750
MGU	**1940**	**1170**	4230	7830
YamRNN	4240	1850	**1690**	**1680**
Computation time (seconds)				
LSTM	1250	925	2120	3490
GRU	729	983	1220	2530
S-LSTM	1070	1040	1400	2500
MGU	**491**	**421**	1750	3480
YamRNN	795	461	**473**	**507**

Table 4. Results on test dataset 2, where the length of the input sequences is 300. The number of epochs and computation time is the mean value of the successful trials. The best values on each category are shown in bold. Three significant digits are displayed.

	The number of units			
	256	512	768	1024
Success count out of 10 trials				
LSTM	4	2	3	0
GRU	**10**	9	**10**	3
S-LSTM	2	5	7	5
MGU	**10**	7	2	0
YamRNN	7	**10**	**10**	**10**
Number of epochs				
LSTM	11500	26500	42200	–
GRU	7800	12800	31000	44700
S-LSTM	15900	24500	31900	44000
MGU	**7780**	25800	45400	–
YamRNN	19000	**5730**	**4930**	**3950**
Computation time (seconds)				
LSTM	7180	25200	44400	–
GRU	3910	9520	26000	41000
S-LSTM	5750	12800	18900	28600
MGU	**2890**	13800	27600	–
YamRNN	5270	**2120**	**2050**	**1770**

Table 5. Results on test dataset 3, where the length of the input sequences is 500. The number of epochs and computation time are the mean values of the successful trials. The best values for each category are shown in bold. Three significant digits are displayed.

	The number of units			
	256	512	768	1024
Success count out of 10 trials				
LSTM	0	0	0	0
GRU	**10**	8	1	0
S-LSTM	1	1	0	0
MGU	5	4	0	0
YamRNN	1	**10**	**10**	**10**
Number of epochs				
LSTM	–	–	–	–
GRU	**8900**	23300	48600	–
S-LSTM	22800	38100	–	–
MGU	11000	35800	–	–
YamRNN	26300	**7780**	**5070**	**4970**
Computation time (seconds)				
LSTM	–	–	–	–
GRU	7380	28800	67500	–
S-LSTM	13400	33500	–	–
MGU	**6730**	31600	–	–
YamRNN	11900	**4700**	**3410**	**3670**

different from that of YamRNN on 1024 units (1770 seconds), even though the number of parameters was almost the same in both the architectures. While S-LSTM is a derivative of LSTM and MGU is a derivative of GRU, YamRNN was developed in a completely different way. The biggest difference between YamRNN and the existing architectures lies in the topology implied by the last equation of each architecture for computing output and memory vectors, $\mathbf{h}_t$. It is one of the important results of the study that we can clarify that topology contributes to the difference in performance. However, in this study, we did not discuss the topology further because there were only few clues for consideration. This will be a topic for future study. One of the major advantages of YamRNN could be its scalability against parameter increase. At least, YamRNN could reach the answer with the large parameter settings for which LSTM and GRU failed during computation. From this perspective, YamRNN was useful and could be an option for learning context. However, we must note that we do not claim YamRNN is the best possible architecture, only that it will be a useful option for problems including contextual information. The most essential point related to machine learning applications is that there is no architecture which always the best performance for arbitrary problems. This was illustrated by our test results, where the best architecture changed depending on parameter settings such as data length and the number of units in the layer. When doing machine learning analysis, we must choose the optimal architecture suitable for problems by trial and error. Although in the study we could not analyze for what type of problem YamRNN was most suited, it was clear that YamRNN was compact and thus had an advantage in learning speed. Further advantages and weak points of YamRNN could be revealed by applying YamRNN to various real world problems. Finally, this study has the limitation that we could not investigate all possible architectures by our stochastic search method, and there may be architectures with higher performance than YamRNN. However, at the same time, this limitation could be considered as a chance for further improvement. To find better architectures, search optimization methods such as Bayesian optimization and evolutionary algorithm could be useful.

CONCLUSION

In this study, we developed a novel RNN architecture, YamRNN, with a compact structure. We examined the performance of the RNN and confirmed that it provides normal learning performance for sequence data and excellent learning speed. The number of parameters of YamRNN was about 50% and 67% that of LSTM and GRU, respectively. The learning times required to achieve the same learning performance LSTM and GRU were shortened at maximum by 85% and 80%, respectively, on a sequence classification task. One of the most outstanding advantages of YamRNN is its scalability against an increase in the number of parameters. This novel RNN architecture is expected to be useful for addressing various problems involving sequential data, such as speech recognition and generation, language translation, and image and text analysis. In addition, our success in developing this novel architecture would suggest that there is room for designing even better architectures, because we did not examine all feasible architectures in this study.

ACKNOWLEDGEMENT

Computations were partially performed on the NIG supercomputer at ROIS National Institute of Genetics. This work was supported in part by the Top Global University Project from the Ministry of Education, Culture, Sports, Science, and Technology of Japan (MEXT), KAKENHI from the Japan Society for the Promotion of Science (JSPS) under Grant Number 18K18143 and Platform Project for Supporting in Drug Discovery and Life Science Research (Basis for Supporting Innovative Drug Discovery and Life Science Research (BINDS)) from AMED.

AVAILABILITY

The code for the RNNs and the test datasets are available on git@github.com:yamada-kd/YamRNN.git.

REFERENCES

1. Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.
2. Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
3. Felix A. Gers, Jrgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. *Neural Computation*, 12(10):2451–2471, 2000.
4. Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.
5. M. Mitchell Waldrop. The chips are down for moore's law. *Nature*, 530:144–147, 2016.
6. Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model compression and acceleration for deep neural networks. *CoRR*, abs/1710.09282, 2017.
7. Justin Bayer, Daan Wierstra, Julian Togelius, and Jürgen Schmidhuber. Evolving memory cell structures for sequence learning. *International Conference on Artificial Neural Networks*, pages 755–764, 2009.
8. Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. An empirical exploration of recurrent network architectures. *International conference on machine learning*, pages 2342–2350, 2015.
9. Zhizheng Wu and Simon King. Investigating gated recurrent networks for speech synthesis. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5140–5144. IEEE, 2016.
10. Guo-Bing Zhou, Jianxin Wu, Chen-Lin Zhang, and Zhi-Hua Zhou. Minimal gated unit for recurrent neural networks. *International Journal of Automation and Computing*, 13(3):226–234, 2016.
11. Yann LeCun. The mnist database of handwritten digits. *http://yann.lecun.com/exdb/mnist/*, 1998.
12. Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
13. James Bergstra, O. Breuleux, Frederic Bastien, Pascal Lamblin, Razvan Pascanu, G. Desjardins, J. Turian, David Warde-Farley, and Y. Bengio. Theano: a cpu and gpu math expression compiler. *Proceedings of the Python for Scientific Computing Conference (SciPy)*, Vol. 4, 2010.