Sensitivity
– Local Index to Control Chaoticity or Gradient Globally – [*]

Katsunari Shibata[†]   **Takuya Ejima**   **Yuki Tokumaru**[‡]   **Toshitaka Matsuki**

katsunarishibata@gmail.com, matsuki@oita-u.ac.jp

Oita University, Japan

July 20, 2021

ABSTRACT

Here, we introduce a fully local index named "sensitivity" for each neuron to control chaoticity or gradient globally in a neural network (NN). We also propose a learning method to adjust it named "sensitivity adjustment learning (SAL)". The index is the gradient magnitude of its output with respect to its inputs. By adjusting its time average to 1.0 in each neuron, information transmission in the neuron changes to be moderate without shrinking or expanding for both forward and backward computations. That results in moderate information transmission through a layer of neurons when the weights and inputs are random. Therefore, SAL can control the chaoticity of the network dynamics in a recurrent NN (RNN). It can also solve the vanishing gradient problem in error backpropagation (BP) learning in a deep feedforward NN or an RNN. We demonstrate that when applying SAL to an RNN with small and random initial weights, log-sensitivity, which is the logarithm of RMS (root mean square) sensitivity over all the neurons, is equivalent to the maximum Lyapunov exponent until it reaches 0.0. We also show that SAL works with BP or BPTT (BP through time) to avoid the vanishing gradient problem in a 300-layer NN or an RNN that learns a problem with a lag of 300 steps between the first input and the output. Compared with manually fine-tuning the spectral radius of the weight matrix before learning, SAL's continuous nonlinear learning nature prevents loss of sensitivities during learning, resulting in a significant improvement in learning performance.

⟨ Highlights ⟩

- "Sensitivity" is a local version of the maximum Lyapunov exponent for each neuron.

- Log sensitivity is equivalent to the maximum Lyapunov exponent until the dynamics reach the "edge of chaos".

- Sensitivity Adjustment Learning (SAL) adjusts the sensitivity in each neuron and realizes "edge of chaos" in the network as a result.

- SAL also prevents "vanishing gradient" in gradient-based learning such as BP or BPTT, which greatly improves learning performance.

- Compared with the adjustment of weight matrix, SAL can consider non-linearity and prevent the loss of sensitivity caused by another learning.

Keywords Sensitivity, Sensitivity adjustment learning (SAL), Edge of chaos, Recurrent neural network (RNN), Deep feedforward neural network (DFNN), Vanishing gradient problem

1 Introduction

Deep learning using a deep feedforward neural network (DFNN) or a recurrent neural network (RNN) has attracted great attention due to its drastic performance improvement especially in recognition of patterns including the case of time-series signals [Jiao et al. (2019); Nassif et al. (2019); Fawaz et al. (2018); Otter et al. (2018)]. They have shown us the overwhelming power of massively parallel processing systems acquired through learning. That suggests the way towards human-like processing, which human designers have struggled to design for a long time, but has not been achieved by conventional artificial intelligence.

Since higher functions such as thinking and communication cannot be discussed without processing in the time axis, there is no doubt that the significance of temporal processing or dynamics will increase more and more, and longer processing will be required from now on. In temporal processing, we need to think of the internal state of a network not as points in space but as lines or flow formed around a point moving along time.

When solving a task with long-term dependency is required, the effective preservation of information in a neural network through time becomes critical. That is deeply related to the chaoticity of the network. If the maximum Lyapunov exponent of the network is negative, much of the information shrinks and disappears over time. If positive, the network expands even trivial pieces of information one after another, but in contrast, it cannot magnify rather large pieces of information so much due to the nonlinearity by the limited value range in each neuron. Therefore, even though they were large, it is challenging to retrieve the old information from the current internal state after a long time lag. Accordingly, for effective information preservation through time, dynamics around the "edge of chaos" would be the right choice.

As for the learning of long-term dependency in an RNN, vanishing/exploding gradient in BPTT (error Back Propagation Through Time) as a gradient-based learning method has been discussed since far before the boom of deep learning [Bengio et al. (1994); Hochreiter (1998); Pascanu et al. (2013)]. This problem is caused by the same logic as the information preservation for a long time lag mentioned above. The gradient here is the gradient of some evaluation (can be cost or error) function of final outputs with respect to the weight vector or neuron state vector. By decomposing the gradient by chain rule, we can understand the problem comes from the repetition of the information scaling through each time-step. A similar discussion can be made for the same problem in deep feedforward NNs (DFNNs). Detailed and related works are described in the next section.

Reservoir computing is often used to learn tasks that need temporal processing and shows excellent results even though it usually does not learn the connection weights among the reservoir neurons. Here, an important parameter is the scale or spectral radius of the weight matrix in the reserver. Echo state property is considered so that the effect of initial conditions vanish as time passes [Yildiz et al. (2012)]. For learning long-term dependency, the spectral radius should be close to 1.0. In FORCE learning, the learning performance is good when the network dynamics are around the edge of chaos [Sussillo (2009)]. We have also shown in reward-modulated Hebbian learning with chaotic exploration using a reservoir network, dynamics around the edge of chaos brings out good learning performance [Matsuki and Shibata (2016, 2020)]. However, the reservoir does not learn its dynamics directly. That is similar to the perceptron that fixes its randomly-decide hidden weights and does not learn its hidden representation. Furthermore, we cannot find how such random weights with an appropriate size are realized in each neuron autonomously without any centralized system. We believe that learning dynamics must be crucial when developing higher functions like "thinking".

To develop a "thinking machine" as an ultimate artificial intelligence system, autonomous and rational state transitions even without the help of external stimuli must be essential in its RNN. Similar to the other functions such as "recognition" and "memory", we expect "thinking" emerges within the framework of end-to-end reinforcement learning using an RNN [Shibata (2017)]. Acquiring appropriate memories that do not need transitions but need convergence is relatively easy for an agent [Shibata and Sugisaka (2004); Utsunomiya and Shibata (2009); Shibata and Utsunomiya (2011); Shibata and Goto (2013)]. However, it is not easy to acquire rational transitions through reinforcement learning using a regular RNN even though the transitions are externally driven [Sawatsubashi et al. (2012)].

Then, we focused on chaotic dynamics generated in an RNN and proposed a new reinforcement learning (RL) paradigm [Shibata and Sakashita (2015); Shibata and Goto (2017)]. Chaotic dynamics expand tiny variations in its network state through time. That destabilizes the dynamics and enables autonomous state transition in the RNN, although the transition is irregular. In the new proposed RL, exploration does not use stochastic action selection by random noises but uses the autonomous state transition based on the internal chaotic dynamics. We expect that the irregular dynamics would become rational by forming attractors on the dynamics through learning. According to the above discussion, we set up a hypothesis that "exploration" grows into "thinking" through learning. In our new RL, the chaoticity of the RNN also should not be either too strong or too weak [Goto and Shibata (2017); Sato et al. (2019)]. Adjustment of chaoticity is critical in this learning paradigm as well.

Processing in each neuron produces the dynamics of the network. The propagated error signals in gradient-based BP (error backpropagation) or BPTT learning represent the influence of tiny variation in each neuron's state on the final error function. Therefore, we expect that by adjusting the influence in each neuron, we can control both chaoticity and error backpropagation of the network simultaneously. In this paper, we define the gradient magnitude of its output with respect to its input vector as "sensitivity" in each neuron and use it as a local index to control the chaoticity and error signal propagation globally in the network. We also propose a learning method to adjust the sensitivity based on hill-climbing and call it "sensitivity adjustment learning (SAL)". Then we show that an RNN comes to generate chaotic dynamics through SAL, and observe the relationship between the sensitivities as a local index and the maximum Lyapunov exponent as a global index during learning. We also show some supervised learning results of a simple problem with a lag of 300 steps between the first input and final output timings when using both BPTT and SAL in an RNN, and the learning process is analyzed. Finally, we apply SAL to DFNNs with BP and show a 300-layer DFNN can learn a simple problem with noise addition and observe the processing.

2   Related Works

Several techniques have been proposed already to avoid the vanishing/exploding gradient problem in error backpropagation (BP) or backpropagation through time (BPTT). They can be roughly divided into two categories.

In the first category, which mainly targets on RNNs whose feedback weight matrix is square, the spectral radius of the Jacobian matrix between the inputs and outputs of a layer or group of neurons is directly set to be around 1.0. The simplest way is to set the Jacobian matrix close to the identity matrix. In this case, the output vector is close to the input vector. Therefore, a slight variation in each neuron state does not change so much through the forward processing. The error signal in each neuron also does not change so much through the backward processing.

LSTM [Hochreiter and Schmidhuber (1997)], a quite popular RNN, employs special units named LSTM cells. If the forget gate is fully open, the cell state does not change, and each diagonal element of the Jacobian matrix is close to 1.0 when only the signal flow inside the cell is focused on. GRU [Chung et al. (2014)] has a simpler but similar structure to LSTM. We have adopted a far simpler method using a regular RNN with setting all the self-feedback connection weights to the reciprocal of the maximum derivative of the activation function. Concretely, the weight value is 4.0 for the sigmoid function and 1.0 for the hyperbolic tangent function. All the other feedback connection weights are set to 0.0 or small random values. In this case, the Jacobian matrix is close to the identity matrix when the activations are small enough in the activation function's linear region. It works well in learning memory-required tasks [Shibata and Sugisaka (2004); Utsunomiya and Shibata (2009); Shibata and Utsunomiya (2011); Shibata and Goto (2013)], which need to form fixed-point attractors or static associative memory. In DFNNs, shortcut connections through one or more layers as in a ResNet [He et al. (2015)], which is widely used mainly in convolutional NNs (CNNs), also make its Jacobian matrix close to the identity matrix if the other connection weights are small.

However, the transformation represented by the identity matrix is equivalent to applying no processing. Therefore, it is suitable to keep some information without any change, but it is not appropriate to learn a complicated conversion of input signals or internal dynamic state transition like "thinking". Actually, in our work [Sawatsubashi et al. (2012)], although an agent could learn simple state transitions in this approach, learning was so difficult that careful design of task shaping was necessary.

In the second category, the mean and variance of neuron activations are normalized, usually to zero mean and unit variance. Batch normalization [Ioffe and Szegedy (2015)], layer normalization [Ba et al. (2016)], weight normalization [Salimans and Kingma (2016)], self-normalizing neural networks [Klambauer et al. (2017)] can be categorized here in a broad meaning. In the self-normalizing neural network, by setting the weight matrix and activation function appropriately, the activations close to zero mean and unit variance converge towards zero mean and unit variance. Furthermore, the variance of neuron activations is bounded, and the network does not suffer from a vanishing gradient. In the weight normalization process, the weight vectors are initialized depending on the given data so that the mean and variance of neuron activations are normalized.

Different from the approaches mentioned above, moderatism aims to acquire necessary processing by keeping the variation of both inputs and output in each neuron moderate [Okabe et al. (1998)]. Here, we focus on the sensitivity that represents magnification or contraction of a small variation in each neuron's processing. By controlling it in each neuron, the network's global dynamics can be controlled. We show that adjusting each neuron's sensitivity enables learning for DFNNs or RNNs solving a long time-lag problem.

On the other hand, from the viewpoint of chaos control, previous studies have mainly focused on stabilizing a system with chaotic dynamics [Ott et al. (1990); Ighneiwaa et al. (2017)] and have not positively utilized the chaos or

edge of chaos dynamics. From the information storage or processing perspective, the importance of the "edge of chaos" has been highlighted by several studies [Langton (1990); Legenstein et al. (2010); Boedecker et al. (2012)]. Moreover, from the biological viewpoint, the relation between known input stimuli and attractors in brain dynamics was investigated, and the role of the chaotic dynamics for new input stimuli was addressed [Skarda and Freeman (1987); Freeman (1991)]. Based on the study, an associative memory model using a chaotic neural network was also proposed [Osana and Hagiwara (1999)]. However, it is not easy to find a method to generate and control chaotic dynamics by learning in an RNN. Consequently, dynamics around the edge of chaos produced in an RNN with fixed weights have been utilized widely as a reservoir as mentioned above. In this paper, aiming to positively utilize chaos or edge of chaos dynamics, we propose to generate and control chaotic dynamics in RNNs by adjusting local sensitivity in each neuron.

3   Sensitivity and Sensitivity Adjustment Learning (SAL)

3.1   Definition of Sensitivity

Fig. 1 shows a general static-type neuron model with m inputs. Its internal state u is derived as the inner product of the input vector $\mathbf{x} = (x_1, \ldots, x_m)^{\mathrm{T}}$ and connection weight vector $\mathbf{w} = (w_1, \ldots, w_m)^{\mathrm{T}}$ as

$$u = \mathbf{w} \cdot \mathbf{x}, \tag{1}$$

and the output o is derived as

$$o = f(U) = f(u + \theta), \tag{2}$$

where $U = u + \theta$, θ is the bias, and $f(\cdot)$ is an activation function that can be hyperbolic tangent or sigmoid function. The sensitivity introduced here is a local index for each neuron to show how the neuron is sensitive to a small change in its inputs. It is defined as the Euclidean norm of the output gradient with respect to the input vector $\mathbf{x}$ as

$$s(U; \mathbf{w}) = \|\nabla_{\mathbf{x}} o\| = f'(U)\|\mathbf{w}\| \tag{3}$$

if the activation function $f(\cdot)$ is a monotonically increasing function. If the vector elements are used, it is rewritten as

$$s(U; \mathbf{w}) = \sqrt{\sum_i^m \left(\frac{\partial o}{\partial x_i}\right)^2} = f'(U)\sqrt{\sum_i^m w_i^2} \,. \tag{4}$$

3.2   Sensitivity in the forward computation

In the forward computation in the neuron, as shown in Fig. 1, an infinitesimal change in the inputs $d\mathbf{x}$ produces the infinitesimal change in the output do as

$$do = \nabla_{\mathbf{x}} o \cdot d\mathbf{x} = f'(U)\mathbf{w} \cdot d\mathbf{x}, \tag{5}$$

and can be rewritten as

$$do = \|\nabla_{\mathbf{x}} o\| \, \|d\mathbf{x}\| \cos \phi = s(U; \mathbf{w}) \, \|d\mathbf{x}\| \cos \phi \tag{6}$$

using the direction cosine $\cos \phi$ between the input change vector $d\mathbf{x}$ and the weight vector $\mathbf{w}$ whose direction is the same as the gradient vector $\nabla_{\mathbf{x}} o$.

Here, $d\mathbf{x}$ is assumed to be an m-dimensional standard normal random vector multiplied by an infinitesimal constant ϵ as $d\mathbf{x} \sim \mathcal{N}(0, \epsilon^2 I_m)$ where I_m is the m-dimensional unit matrix. The distribution of the vector direction is uniform,

Figure 1: A neuron model and the influence of a small deviation in its inputs $\mathbf{x}$ on its output o.

Figure 2: The relation between an infinitesimal input change vector $d\mathbf{x}$ and the output change do in the case of the number of inputs $m = 3$. When the sensitivity s is 1.0, do is the projection of $d\mathbf{x}$ onto the weight vector $\mathbf{w}$ whose direction is the same as $\nabla_{\mathbf{x}} o$. Therefore, when $d\mathbf{x}$ is distributed uniformly on a super sphere surface, the variance of do is equivalent to that of one of the m input signals.

and that is the same as the uniform distribution on the m-dimensional super sphere surface as presented in Fig. 2. In an m-dimensional Euclidean space, the square of the vector $d\mathbf{x}$ is identical to the sum of its m squared elements. In other words, the sum of the m squared direction cosine to individual standard bases is 1.0. Therefore, from the symmetry among m dimensions, the square of the projection of $d\mathbf{x}$ on any direction is expected to be $1/m$ of the square of $d\mathbf{x}$ itself. By applying it to the relation between the vector $d\mathbf{x}$ and its elements dx_i, the square of the vector $d\mathbf{x}$ is expected to be the m-times of the variance of each element dx_i as

$$E\left[\|d\mathbf{x}\|^2\right] = \sum_i^m V\left[dx_i\right] = m\,V\left[dx_i\right] = m\epsilon^2. \tag{7}$$

By applying it to the projection of the vector $d\mathbf{x}$ onto the weight vector $\mathbf{w}$, the variance of the direction cosine $\cos\phi$ is derived as

$$V\left[\cos\phi\right] = E\left[\cos^2\phi\right] = \frac{1}{m} \tag{8}$$

where the expected direction cosine is 0.0 from the symmetry of the $d\mathbf{x}$ distribution. The direction of $d\mathbf{x}$ is uniformly distributed and independent with the size of $\|d\mathbf{x}\|$. Then, in Eq. (6), $\|d\mathbf{x}\|$ and $\cos\phi$ are independent. Therefore, from Eqs. (7) and (8), the variance of the output change do can be written for a given sensitivity s using the variance of one input change dx_i as

$$V\left[do\right] = \{s(U;\mathbf{w})\}^2 E\left[\|d\mathbf{x}\|^2\right] E\left[cos^2\phi\right] = s^2\,V\left[dx_i\right] = s^2\epsilon^2. \tag{9}$$

If the sensitivity s is 1.0, then the output change in Eq. (6) can be written as

$$do = \|d\mathbf{x}\|\cos\phi, \tag{10}$$

and that is the projection of the m-dimensional infinitesimal vector $d\mathbf{x}$ onto the direction of the weight vector $\mathbf{w}$ as shown in Fig. 2. Therefore, its distribution becomes $do \sim \mathcal{N}(0, \epsilon^2)$ not depending on the number of inputs m, and Eq. (9) is changed as follows:

$$V\left[do\right] = V\left[dx_i\right] = \epsilon^2. \tag{11}$$

This is significant because the neuron maintains the variance of just one input signal as its own output variance without being shrunk or expanded not depending on the number of connections m.

Next, the case of a group or layer of n neurons as can be seen in Fig. 3 is considered, and its output vector is indicated as $\mathbf{o} = (o_1, \ldots, o_n)^{\mathrm{T}}$. The infinitesimal output change $d\mathbf{o}$ can be represented using the infinitesimal input change $d\mathbf{x}$ as

$$d\mathbf{o} = \mathbf{J}(\mathbf{U};\mathbf{W})d\mathbf{x} = f'(\mathbf{U}) \circ (\mathbf{W}d\mathbf{x}) \tag{12}$$

where $f'(\mathbf{U}) = (f'(U_1), \ldots, f'(U_n))^{\mathrm{T}}$ and '$\circ$' indicates element-wise multiplication. $\mathbf{W} = (\mathbf{w}_1, \ldots, \mathbf{w}_n)^{\mathrm{T}}$ is an $n \times m$ weight matrix where $\mathbf{w}_j(j = 1, \ldots, n)$ is the weight vector of the jth neuron. $\mathbf{J}(\mathbf{U};\mathbf{W})$ is the Jacobian matrix as

$$\mathbf{J}(\mathbf{U};\mathbf{W}) = \begin{pmatrix} \frac{\partial o_1}{\partial x_1} & \cdots & \frac{\partial o_1}{\partial x_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial o_n}{\partial x_1} & \cdots & \frac{\partial o_n}{\partial x_m} \end{pmatrix} = \begin{pmatrix} (\nabla_{\mathbf{x}} o_1)^{\mathrm{T}} \\ \vdots \\ (\nabla_{\mathbf{x}} o_n)^{\mathrm{T}} \end{pmatrix} = \begin{pmatrix} f'(U_1)\mathbf{w}_1^{\mathrm{T}} \\ \vdots \\ f'(U_n)\mathbf{w}_n^{\mathrm{T}} \end{pmatrix}. \tag{13}$$

5

Figure 3: Forward (output) computation through a layer of neurons.

Eq. (12) can be rewritten using sensitivities of the neurons as

$$d\mathbf{o} = \begin{pmatrix} \nabla_{\mathbf{x}} o_1 \cdot d\mathbf{x} \\ \vdots \\ \nabla_{\mathbf{x}} o_n \cdot d\mathbf{x} \end{pmatrix} = \begin{pmatrix} s_1 \, \|d\mathbf{x}\| \cos \phi_1 \\ \vdots \\ s_n \, \|d\mathbf{x}\| \cos \phi_n \end{pmatrix}. \tag{14}$$

Here, in addition to the assumption of $d\mathbf{x}$ mentioned above, it is further assumed that the weight matrix $\mathbf{W}$ has been randomly chosen with i.i.d. elements with mean 0, and also $f'(\mathbf{U})$ is a random vector having elements with an identical distribution. If the number of output n is not greater than the number of input m, the elements of the infinitesimal change of the output vector do_j are i.i.d. with mean 0. The mean square of the magnitude of the vector $d\mathbf{o}$ is n times the variance of one output change do_j as

$$E\left[\|d\mathbf{o}\|^2\right] = \sum_j^n \mathrm{V}[do_j] = n\mathrm{V}[do_j]. \tag{15}$$

From Eqs. (8) and (9),

$$\frac{1}{n} E\left[\|d\mathbf{o}\|^2\right] = \frac{\langle s^2 \rangle}{m} E\left[\|d\mathbf{x}\|^2\right] = \langle s^2 \rangle \epsilon^2 \tag{16}$$

on the assumption that n is large enough and

$$\langle s^2 \rangle = E\left[\{s(U_j; \mathbf{w}_j)\}^2\right] = E\left[\{f'(U_j)\mathbf{w}_j\}^2\right] \quad (j = 1, \dots, n) \tag{17}$$

where $\langle s^2 \rangle$ is the mean of s^2 over the n neurons. If $\langle s^2 \rangle = 1.0$ or the sensitivity of each neuron is 1.0, then

$$\frac{\mathrm{RMS}\left[\|d\mathbf{o}\|\right]}{\sqrt{n}} = \frac{\mathrm{RMS}\left[\|d\mathbf{x}\|\right]}{\sqrt{m}} = \epsilon \tag{18}$$

where RMS means root mean square. If the numbers of inputs m and outputs n are the same, then

$$\mathrm{RMS}\left[\|d\mathbf{o}\|\right] = \mathrm{RMS}\left[\|d\mathbf{x}\|\right] = \sqrt{m}\epsilon \tag{19}$$

When the number of neurons n is less than the number of inputs m, it is expected that the amount of information is decreased through the layer.

From the above discussion, if each neuron's sensitivity is 1.0, the distribution of the small change in each neuron is maintained not depending on the number of neurons or number of inputs. Accordingly, a small change in an input signal of the network reaches the final network output even though (1) the number of layers is large, (2) the number of neurons in each layer is varied and/or (3) the connection between layers is sparse. The same discussion can be applied to an RNN. The distribution of small output change is maintained through time regardless of the structure of the RNN, such as sparse or full connections, flat (non-layered) or layered. Therefore, if the sensitivities for all the neurons are around 1.0, the maximum Lyapunov exponent as the logarithm of time development of small change is expected to be around 0.0.

3.3   Sensitivity in the backward computation

As mentioned in the Introduction, when a small change in network inputs reaches the network outputs, the error signals for learning reach the input layer from the output layer in the backward computation for gradient-based learning such

Figure 4: Backward (error signal) computation in a neuron.

as BP or BPTT. A neuron receives an error signal for its output $\hat{\delta} = -\partial E/\partial o$ where E is a given cost or error function. Error signal δ for its internal state u is derived by multiplying the derivative of its activation function to $\hat{\delta}$ as

$$\delta = -\frac{\partial E}{\partial u} = -\frac{\partial E}{\partial o}\frac{do}{du} = \hat{\delta}f'(U). \tag{20}$$

As depicted in Fig. 4, the error signal is propagated after being weighted by the weight vector $\mathbf{w}$ before propagating to the one-level lower-layer neurons. Then the propagated error signal vector $\delta\mathbf{w}$ can be written as

$$\delta\mathbf{w} = \hat{\delta}f'(U)\mathbf{w} = \hat{\delta}\nabla_{\mathbf{x}}o. \tag{21}$$

Therefore, its magnitude can be expressed as

$$\|\delta\mathbf{w}\| = \left\|\hat{\delta}\nabla_{\mathbf{x}}o\right\| = \left|\hat{\delta}\right| \|\nabla_{\mathbf{x}}o\| = s(U;\mathbf{w})\left|\hat{\delta}\right|. \tag{22}$$

If the sensitivity $s(U;\mathbf{w}) = \|\nabla_{\mathbf{x}}o\|$ is 1.0, then

$$\|\delta\mathbf{w}\| = \left|\hat{\delta}\right|, \tag{23}$$

and the error signal is propagated efficiently without being shrunk or expanded through the neuron.

From the view of group or layer of neurons as can be seen in Fig. 5, the relation between the error signal before and after the layer $\hat{\boldsymbol{\delta}}_{upper}$, $\hat{\boldsymbol{\delta}}_{lower}$ can be written as

$$\hat{\boldsymbol{\delta}}_{lower} = \mathbf{W}^{\mathrm{T}}\left(f'(\mathbf{U}) \circ \hat{\boldsymbol{\delta}}_{upper}\right). \tag{24}$$

The Jacobian matrix as presented in Eq. (13) can be modified as

$$\begin{aligned}\mathbf{J}(\mathbf{U};\mathbf{W}) &= \begin{pmatrix} f'(U_1)w_{11} & \cdots & f'(U_1)w_{1m} \\ \vdots & \ddots & \vdots \\ f'(U_n)w_{n1} & \cdots & f'(U_n)w_{nm} \end{pmatrix} \\ &= \left(f'(U_1)\mathbf{w}_1, \ldots, f'(U_n)\mathbf{w}_n\right)^{\mathrm{T}}.\end{aligned} \tag{25}$$

Then, Eq. (24) is changed to

$$\hat{\boldsymbol{\delta}}_{lower} = \mathbf{J}^{\mathrm{T}}(\mathbf{U};\mathbf{W})\hat{\boldsymbol{\delta}}_{upper}. \tag{26}$$

Accordingly, though the Jacobian is transposed, the similar discussion as the relation between $\|d\mathbf{o}\|$ and $\|d\mathbf{x}\|$ in the forward computation can be made. The same assumptions are made for $\mathbf{W}$ and $f'(U)$ as before, and $\hat{\boldsymbol{\delta}}_{upper}$ is assumed to be a random vector whose elements are i.i.d. with mean 0. The relation between the mean squared error signal vectors before and after the layer can be found as

$$\frac{1}{m}E\left[\left\|\hat{\boldsymbol{\delta}}_{lower}\right\|^2\right] = \frac{\langle s^2 \rangle}{n}E\left[\left\|\hat{\boldsymbol{\delta}}_{upper}\right\|^2\right], \tag{27}$$

in the same way as for Eq. (16). If the sensitivity of each neuron is 1.0,

$$\frac{\mathrm{RMS}\left[\left\|\hat{\boldsymbol{\delta}}_{lower}\right\|\right]}{\sqrt{m}} = \frac{\mathrm{RMS}\left[\left\|\hat{\boldsymbol{\delta}}_{upper}\right\|\right]}{\sqrt{n}}, \tag{28}$$

and the error signals in backward computation do not either disappear or explode. If the numbers of inputs and outputs are the same, RMS remains unchanged before and after the layer as stated below:

$$\mathrm{RMS}\left[\left\|\hat{\boldsymbol{\delta}}_{lower}\right\|\right] = \mathrm{RMS}\left[\left\|\hat{\boldsymbol{\delta}}_{upper}\right\|\right]. \tag{29}$$

Therefore, if the sensitivities of all network neurons can be controlled around 1.0, it is expected that the error signals propagate backward without being shrunk or expanded.

Figure 5: Backward (error signal) computation through a layer of neurons.

3.4 Learning Method (Sensitivity Adjustment Learning (SAL))

In sensitivity adjustment learning (SAL), which we also propose in this paper, each neuron has a small weight vector initially and updates them to increase its sensitivity by hill-climbing that is the steepest ascent according to

$$\Delta \mathbf{w} = \eta_{SAL} \nabla_{\mathbf{w}} s(U; \mathbf{w}) = \eta_{SAL} \nabla_{\mathbf{w}} \|\nabla_{\mathbf{x}} o\| \tag{30}$$

where η_{SAL} is the learning rate for SAL. From Eq. (3),

$$\begin{aligned}
\nabla_{\mathbf{w}} \|\nabla_{\mathbf{x}} o\| &= \nabla_{\mathbf{w}} \{ f'(U) \|\mathbf{w}\| \} \\
&= f'(U) \nabla_{\mathbf{w}} \|\mathbf{w}\| + \|\mathbf{w}\| \nabla_{\mathbf{w}} f'(U) \\
&= f'(U) \frac{\mathbf{w}}{\|\mathbf{w}\|} + \|\mathbf{w}\| \nabla_{\mathbf{w}} f'(U)
\end{aligned} \tag{31}$$

Then the update rule can be written as

$$\Delta \mathbf{w} = \eta_{SAL} \left(f'(U) \frac{\mathbf{w}}{\|\mathbf{w}\|} + \|\mathbf{w}\| \nabla_{\mathbf{w}} f'(U) \right). \tag{32}$$

In the following simulations, we use $\tanh$ as activation function $f(\cdot)$ for all the neurons. In this case,

$$f'(U) = \frac{1}{\cosh^2(U)} = 1 - o^2 \tag{33}$$

and so

$$\begin{aligned}
\nabla_{\mathbf{w}} f'(U) &= \nabla_{\mathbf{w}} (1 - o^2) \\
&= -2o f'(U) \nabla_{\mathbf{w}} U \\
&= -2o(1 - o^2) \mathbf{x}.
\end{aligned} \tag{34}$$

Then the update rule is as

$$\Delta \mathbf{w} = \eta_{SAL}(1 - o^2) \left(\frac{\mathbf{w}}{\|\mathbf{w}\|} - 2o \|\mathbf{w}\| \mathbf{x} \right). \tag{35}$$

In this equation, the term $\eta_{SAL}(1 - o^2)\mathbf{w}/\|\mathbf{w}\|$ is originated from the first term of the right-hand side of Eq. (31), and is called 'linear term'. $\mathbf{w}/\|\mathbf{w}\|$ is the unit vector whose direction is the same as $\mathbf{w}$. Therefore, the sensitivity is increased directly by making the size of the weight vector greater while keeping its direction. Accordingly, if the neuron output o is around the linear region, in which the output is around 0.0, the weight vector $\mathbf{w}$ becomes greater at a constant rate decided by η_{SAL}. The second term $-2\eta_{SAL}(1 - o^2)o \|\mathbf{w}\| \mathbf{x}$ is originated from the second term of the right-hand side of Eq. (31), and is called 'non-linear term'. This means the sensitivity is increased indirectly by making the magnitude of the internal state's smaller for larger $f'(U)$. This term works only after the neuron goes out of the linear region of the activation function.

Different from the case of weight, bias θ cannot increase the sensitivity directly but can increase it indirectly by updating the bias so that the value U becomes closer to 0.0. Accordingly, the update rule for the bias is indicated as

$$\Delta \theta = \eta_{SAL} \|\mathbf{w}\| \frac{\partial f'(U)}{\partial \theta}. \tag{36}$$

8

Furthermore, assuming the activation function is tanh, it can be rewritten as

$$\Delta\theta = -2\eta_{SAL}o(1 - o^2)\,\|\mathbf{w}\|\,.\tag{37}$$

In this paper, to control the sensitivity around 1.0, SAL is applied only when the sensitivity is not greater than 1.0 in each neuron. However, in practice, since the sensitivity is a function of U (internal state after bias is added) or output o, it fluctuates largely as U or o fluctuates. Therefore, in this paper, the moving average of sensitivity $\bar{s}$ in each neuron is computed as

$$\bar{s}_n \leftarrow \beta\bar{s}_{n-1} + (1 - \beta)s_n\tag{38}$$

where β is a decay rate. n indicates the number of forward computations in the whole learning for each neuron, and the initial value is set as $\bar{s}_0 = 0$ here. In the case of an RNN, one neuron executes its forward computation repeatedly, and so n indicates the time step, but is not reset for a new pattern or epoch. It is used as the criterion to decide the application of SAL in each neuron.

When supervised learning is performed together with SAL learning, error backpropagation (BP) learning based on stochastic gradient descent (SGD) including BPTT is used such as

$$\Delta\mathbf{w} = -\eta_{BP}\nabla_{\mathbf{w}}E\tag{39}$$

where E is the squared error when a training signal vector $\mathbf{d}$ is given and is expressed as

$$E = \frac{1}{2}(\mathbf{d} - \mathbf{o}^{out})^2\tag{40}$$

where $\mathbf{o}^{out}$ is the output vector in the output layer. In this paper, assuming simple on-line learning, no batch learning is done, and the network is trained for each presented pattern one by one. However, we expect no hurdles for the extension to batch learning because of the SAL's local learning nature.

Since the sensitivity fluctuates largely as mentioned, the satisfaction of $\bar{s} = 1$ in each neuron does not fully guarantee that the error signals will not explode in the backward computation. Then, we introduce an additional technique to avoid the exploding gradient problem here. The function tanh is equivalent to the identity function when the input is around 0.0, but the output is saturated when the absolute value of the input is large. Then, in the backward computation of the error signals, the function tanh is applied after multiplicting the derivative $f'(U)$. Instead of applying Eq. (20), the error signal is computed as

$$\delta = \tanh(\hat{\delta}f'(U)),\tag{41}$$

and the weight vector is updated as

$$\Delta\mathbf{w} = \eta_{BP}\delta\mathbf{x}.\tag{42}$$

In this paper, when we apply SAL with BP or BPTT, the weights are updated by SAL just after the forward computation at each timing in each neuron when the moving average of its sensitivity $\bar{s}$ is not greater than 1.0. Once the network finishes all the forward computations and SAL learning for the presented pattern, BP or BPTT is applied in the backward computation. Although the error signal is not propagated backward using the original weights but using those after the SAL learning, we think the way is more effective than applying either SAL or (BP or BPTT) for one forward computation. The flowchart is shown in Fig. 6 for the case of learning of an RNN with BPTT as an example.

Figure 6: Flow chart for the parallel learning of SAL and BPTT for one pattern presentation. SAL is applied only when the moving average of the sensitivity $\bar{s}$ is not greater than 1.0. T_{max} is the timing of output. This conditional branch and SAL itself are performed individually in each neuron.

4   Simulations

The following simulations are roughly divided into two parts. In the first part, as described in Section 4.1, only sensitivity adjustment learning (SAL) is applied to recurrent neural networks (RNNs). Then, generation and control of chaotic dynamics are examined, and the relation between sensitivities and maximum Lyapunov exponent is focused on. In the second part described in Section 4.2, SAL is applied with supervised learning using BP or BPTT. Then the performance is observed, and the learning process is analyzed. In all the simulations, hyperbolic tangent is used as the activation function of each neuron.

4.1   Generation and Control of Chaos

At first, we show the generation of chaos dynamics in RNNs and adjustment of its chaoticity by sensitivity adjustment learning (SAL) for various network architectures.

Before entering the simulation result, the way to estimate the maximum Lyapunov exponent is explained [Sprott (2010)]. At every 100 steps ($t = 0, 100, 200....$), two internal states, $\mathbf{u}_0^{(1)} = \mathbf{u}_t$ and $\mathbf{u}_0^{(2)} = \mathbf{u}_t + \mathbf{rnd}$ where $\mathbf{rnd}$ is a small random vector whose Euclidian norm is 10^{-3}, are prepared. The two states are updated separately without updating the weights. At each step τ, the distance ratio between before and after one step update is calculated as

$$l_\tau = ln \left(\frac{\mathbf{u}_\tau^{(2)} - \mathbf{u}_\tau^{(1)}}{10^{-3}} \right). \tag{43}$$

After that, the distance is normalized to 10^{-3} as

$$\mathbf{u}_\tau^{(2)} \leftarrow \frac{10^{-3}}{\left\| \mathbf{u}_\tau^{(2)} - \mathbf{u}_\tau^{(1)} \right\|} \left(\mathbf{u}_\tau^{(2)} - \mathbf{u}_\tau^{(1)} \right) + \mathbf{u}_\tau^{(1)}. \tag{44}$$

Finally, the maximum Lyapunov exponent is computed using the 1,000 steps of the forward computation after 100 steps to eliminate the influence of the initial perturbation as

$$\lambda = \frac{1}{1,000} \sum_{\tau=101}^{1,100} l_\tau. \tag{45}$$

When the network has two layers, that is computed at the 1st layer.

4.1.1   Case of Flat RNN

Firstly, investigation using a flat RNN with no layer structure is introduced. Table 1 presents the parameters. The initial connection weights are small and uniform random numbers, and no bias is used here. The learning rate is small to see the dynamics for each stage during learning. A small random perturbation is added to the internal state vector $\mathbf{u}$ at every 1000 steps from the 1st step as a trigger of activations.

In the first simulation, using an RNN with fully connected 100 neurons, the learning process is shown in detail. Fig. 7 shows the output change of four sample neurons at each of the four stages from (1) to (4) during learning. In early phase of learning, as shown in Fig. 7 (1), although a small perturbation is added at every 1,000 steps, the output is decayed soon. Around the 50,000th step, the outputs change almost periodically as shown in Fig. 7 (2), and around the 53,000th step, the outputs change irregularly as shown in Fig. 7 (3). Around the 100,000th step, as shown in Fig. 7 (4), the outputs still change irregularly, but the value range is greater than in (3).

Table 1: Parameters for chaos generation by SAL using a flat RNN

Number of neurons	100 or 30
Connection rate (%)	100 or 30
Initial connection weights	Uniformly random $[-0.01, 0.01]$
Learning rate η_{SAL} in Eq. (35)	0.00002
Decay rate β in Eq. (38)	0.99
Perturbation (interval, size)	$(1000, 0.001)$

Figure 7: Change of four sample time-series of four outputs as learning progresses from (1) to (4) when applying SAL to a flat RNN.

Fig. 8 shows how the learning progressed from various aspects. Fig. 8(a) provides the maximum and mean absolute value of the output over all the 100 neurons. Before around the 45,000th step indicated by the thick vertical broken line, all the outputs were almost 0.0 though small perturbations were added. Around the 45,000th step, the outputs increased suddenly. After that, they increased gradually with some fluctuations, and finally they varied in most of the value range as can be seen in Fig. 7(4). Fig. 8(b) shows how some sample weights changed during learning. Until around the 45,000th step when the output of all the neurons is in the linear region, the absolute value of each weight grows constantly, which is consistent with Eq. (35). After that, the absolute weight value still tends to increase, but sometimes it decreases, and sometimes the magnitude relation between two weights is switched by the influence of non-linear term in Eq. (35).

Fig. 8(c) presents the change of RMS of the sensitivities over all the neurons during learning. It increases almost linearly until it reaches 1.0 right after the neurons take non-zero values. After that, the increase rate becomes smaller, and the value is fluctuating. To compare it with the maximum Lyapunov exponent, which shows the chaoticity of the network dynamics, the RMS sensitivity is plotted on the log-scale in Fig. 8(d). We call the value 'log-sensitivity' here. The maximum Lyapunov exponent λ is also plotted on the same graph. They are almost the same until around the 45,000th step. Afterward, the slope for λ becomes less than that for log-sensitivity, but λ is still increasing. Then Fig. 9(A) shows the relationship between the maximum Lyapunov exponent λ and log-sensitivity $ln(RMS[s])$ by plotting them on the x- and y-axis respectively. Five lines indicate five cases with different initial connection weights decided randomly. The relations are similar in all the five cases, and when the log-sensitivity reaches 0.0, the maximum Lyapunov exponent reaches almost 0.0.

Then the number of neurons or connection rate is varied and it is examined whether the same relationship can be seen between the maximum Lyapunov exponent and the log-sensitivity. Fig. 9(B) shows the results when decreasing the

11

(a) absolute value of output
((1) - (4) show the timing of the outputs shown in Fig. 7)

(b) connection weights
(sample 10 weights in one neuron)

(c) sensitivity (RMS over all neurons)

(d) maximum Lyapunov exponent & log sensitivity

Figure 8: Learning process when applying SAL to a flat RNN. (a) The maximum and average absolute value over all the outputs. (b) Sample connection weights. (c) RMS of sensitivities over all the neurons. (d) Maximum Lyapunov exponent and log sensitivity (logarithm of the value in (c)). All the data are plotted at every 100 steps. The thick vertical broken line shows the boundary of whether the outputs decay to 0.0 or not.

(A) number of neurons: 100
connection rate: 100%

(B) number of neurons: 30
connection rate: 100%

(C) number of neurons: 100
connection rate: 30%

Figure 9: Relation between the maximum Lyapunov exponent and log-sensitivity during SAL for 5 cases, varying (B) number of neurons or (C) connection rate.

number of neurons to 30, and Fig. 9(C) shows the results when decreasing the connection rate to 30%. In both figures, the maximum Lyapunov exponent is almost identical to the log-sensitivity until 0.0 as in the fully connected RNN with 100 neurons. Only the difference is that the initial value is almost -3.5 in (B) and (C), but originally it is around -2.8 in (A). Those are almost the same as the expected value $ln\left(\mathrm{RMS}\left[\|\mathbf{w}\|\right]\right)$ derived from Eq. (3). Since the initial weight values are decided as a uniform random number from -0.01 to 0.01, $\mathrm{Var}\left[w_i\right] = 10^{-4}/3$. Expected number of inputs is 100 in (A), 30 in (B) and (C), and $f'(U) \approx 1.0$.

4.1.2   Case of Two-Layer RNN

The relationship between the maximum Lyapunov exponent and log-sensitivity is examined in a multi-layer recurrent neural network (RNN). Here, a two-layer RNN as shown in Fig. 10 is employed. The first layer has 1000 neurons, and the second layer has 100 neurons. The connection rate from the first layer to the second layer is 100%. In contrast, the feedback connection rate from the second layer to the first layer is 10%. The expected number of connections is 10 for the first layer neurons, while 1000 for the second layer neurons. No bias is used. Table 2 summarizes the parameters.

Figure 10: Two-layer RNN used in this paper. SAL was applied to all the neurons in the network.

Table 2: Parameters for the chaos generation by SAL using a 2-layer RNN

Number of neurons	1000 (1st) 100 (2nd)
Connection rate (%)	100 (2nd ← 1st) 10 (1st ← 2nd)
Initial connection weights	Uniformly random $[-0.03, 0.03]$
Learning rate η_{SAL} in Eq. (35)	0.00002
Decay rate β in Eq. (38)	0.99
Perturbation (interval, size)	$(1000, 0.001)$

13

(A) SAL was always applied

(B) SAL was applied when average sensitivity is less than 0.0

Figure 11: Relation between the maximum Lyapunov exponent and log-sensitivity during SAL. (1000–100 neurons, 10%–100% connection rate) (B) shows the relation when the target sensitivity was set to 1.0.

Fig. 11(A) shows the relationship between each layer's log-sensitivity and the maximum Lyapunov exponent. Here 'total log-sensitivity' is introduced that is the sum of the log-sensitivities of both layers. The sensitivity value is different between the two layers because the number of connections is different largely. As learning progresses, the log-sensitivities increased, and in the second layer, it became greater than 0.0 though the maximum Lyapunov exponent was still negative. However, it can be seen that the total log-sensitivity is almost the same as the maximum Lyapunov exponent, and when the total log-sensitivity reached 0.0, the maximum Lyapunov exponent reached around 0.0.

Then to adjust the network dynamics to be around the edge of chaos, SAL was stopped in each neuron when the moving average of its sensitivity in Eq. (38) reached 1.0. Fig. 11 (B) displays the results. It shows that the second layer's log-sensitivity did not become greater than 0.0 by stopping SAL in each of the 2nd layer neurons. Finally the network dynamics reached around the edge of chaos, which is the dynamical state with the maximum Lyapunov Exponent $\lambda = 0.0$, and maintained it by stopping SAL also in each of the 1st layer neurons.

4.2   Solving the Vanishing Gradient Problem

Next, let us focus on how sensitivity adjustment learning (SAL) works to avoid vanishing gradient problems in the backward computation for gradient-based learning. Here, SAL is applied in each neuron only when the sensitivity is not greater than 1.0, following the flowchart as presented in Fig. 6.

In the following, two cases of supervised learning are shown. The first one is a recurrent neural network (RNN) solving a problem with long-term dependency, and the network is trained by BPTT (error Back Propagation Through Learning). The second one is a deep feedforward network (DFNN) trained by BP (error Back Propagation). In each case, in order to see how SAL works in gradient-based learning, a simple learning problem and stochastic gradient descent (SGD) is used. Therefore, learning is not applied to a batch or mini-batch but applied for each pattern presentation.

4.2.1   Case of RNN Solving a Long Time-Lag Problem

Here, to see how SAL affects gradient-based supervised learning, a simple 3-layer Elman-type RNN is used whose hidden outputs are fed back to themselves at the next time step. As a simple learning problem, a sequential 3-bit parity problem with a lag of 300 steps as shown in Fig. 12 is given. In the problem, three inputs are given sequentially at every 100 steps, and the training signal is given 300 steps after the timing of the first input. Eight patterns are presented in one epoch.

Table 3 presents the parameters used here. Here, a bias was used in each neuron. Its initial value and learning rate were decided in the same way as the connection weights in the same layer. For the hidden neurons, they were decided in the same way as the feedback connection weights. The propagated error signals and sensitivities during learning are observed as well as the learning performance. One hundred simulations are performed with different random initial connection weights. Here, only the feedback weight vector $\mathbf{w}_{FB}$ is used to compute the sensitivities according to

14

Figure 12: Sequential 3-bit parity problem with a lag of 300 steps. Three inputs were given in turn with intervals of 100 steps.

Table 3: Parameters for supervised learning using SAL and BPTT in an RNN.

Number of neurons (input, hidden, output)		(3, 20, 1)
Initial connection weights (uniformly random)	input $\rightarrow$ hidden	0.0
	hidden $\rightarrow$ output	$[-0.3, 0.3]$
	hidden $\rightarrow$ hidden	$[-0.1, 0.1]$
Learning rate η_{SAL} in Eq. (35)		0.0002
Learning rate η_{BP} in Eq. (39)	input $\rightarrow$ hidden	0.4
	hidden $\rightarrow$ output	0.1
	hidden $\rightarrow$ hidden	0.00004
Decay rate β in Eq. (38)		0.999

Eq. (3), and they are computed at every timing except for $t = 0$ in each hidden neuron. The learning is considered successful if the absolute value of the error becomes less than 0.01 for each of the eight patterns within 1000 epochs.

Here, to see the effect of SAL and the reason why the combination of SAL and BPTT works well, the performance is compared among the various conditions from the case (A) to (G) as shown in Table 4. Each of the conditions will be explained below. Fig. 13 shows the success ratio for each case.

First of all, let us see the case (A) where SAL and BPTT are applied normally, and this case is called 'original' for the following comparisons. There is only one failure in total 100 runs, but even in the failure run, the network could learn it in 3000 epochs. Fig. 14 (A) shows the learning process of a standard sample run whose learning is the 46th fastest in the 100 runs. (These initial weights will be used later as a sample of failures in other cases) In Fig. 14, subfigures (a) and (b) depict the learning curve and the change in the network output for each of the eight patterns during learning. Note that in the 0th epoch that includes eight pattern presentations, no learning was applied to show the output and error signals before learning. As shown in Fig. 14(A)-(a), after the 20th epoch, the error decreased gradually except when it temporarily increased around the 30th epoch.

Subfigure (c) in Fig. 14 shows how the RMS of the error signal δ changed in BPTT in the hidden layer during learning. Before learning, the error signal at the 300th step, which is the output timing, was around 10^{-1}, but for the other input timings (200th, 100th, 0th step), it was far less than 10^{-30}. At the 0th step, it was less than 10^{-160} actually. That indicates the gradient vanished through the backward propagation. However, by applying SAL at every step, the error signals increased rapidly, and in the 2nd epoch, they reached the same order as that at the output timing. Then they stopped to increase. After around 30 epochs of learning, all the four error signals gradually decreased as the final error in (a) decreased.

Subfigure (d) in Fig. 14 shows the average and standard deviation of sensitivities over all the 20 hidden neurons and all the 300 steps during learning. The lower graph in the subfigure (d) shows the number of neurons to which SAL was applied even once during the 300 steps. As shown in Fig. 14(A)-(d), soon after the learning began, the sensitivity increased and its average reached 1.0. Then each neuron stopped SAL, and the value did not change so much, keeping the average a bit greater than 1.0. However, SAL was applied in some neurons afterward when their average sensitivity decreased below 1.0. It is also noticed that the sudden change in the output in Fig. 14(A)-(b) mentioned above was caused by the SAL application around the 30th epoch.

On the other hand, in the case (F) when only performing BPTT without SAL, no successful run can be seen as in Fig. 13(F). In RNNs such as a reservoir, the spectral radius of the feedback weight matrix is usually tuned manually to control the network dynamics before learning. Then the spectral radius of $\mathbf{W}_{FB}$ was increased by 0.01 from 1.0 in

Table 4: List of simulations performed for comparison in the supervised learning of sequential 3bit-parity problem using SAL+BPTT in an RNN.

Case	SAL	Criterion	Others
(A)	Nonlinear	Nonlinear	'original'
(A')	Nonlinear	Nonlinear	No $\tanh$ in BPTT
(B)	Nonlinear	Nonlinear	Applied only initially
(C)	Nonlinear	Linear	
(D)	Linear	Nonlinear	
(E)	Linear	Linear	
(F)	Not applied	–	
(G)	Not applied	–	Tuned $init\_W$

Figure 13: Comparison of success ratio in 100 simulation runs among various conditions in Table 4 in the supervised learning using SAL and BPTT in an RNN.

the case (G). The maximum success ratio was 43 when the spectral radius is 1.38 as in Fig. 13(G), but the ratio was still far below the case when SAL was applied.

To analyze how SAL affects the learning positively, five other cases from (A') to (E) in Table 4 were simulated. Before entering the analysis of SAL itself, let us see the effect of applying $\tanh$ function in the backward error computation as expressed in Eq. (41). Fig. 13(A') shows the success ratio when SAL is applied without using $\tanh$ function in error backpropagation in BPTT. In the failure cases, although each neuron stopped to apply SAL when its average sensitivity was greater than 1.0, the gradient or propagated error signals exploded. The reason could be fluctuation of the sensitivity due to the term $f'(U)$ in Eq. (3) and/or the delay due to the average computation in Eq. (38). Therefore, to avoid such a gradient explosion, it is a good idea to use $\tanh$ also in the backward error computation. In the following, $\tanh$ is always used in the backward computation.

At first, the effect of continuous learning is shown. In the case of (B), SAL was applied only until the sensitivity reached 1.0 for the first time. As shown in Fig. 13(B), the ratio was worse than the 'original' case (A). Fig. 14 (B) shows the learning process when the initial weights were the same as the 'original' case (A), but the learning failed as can be seen in Fig. 14 (B)-(a,b). As shown in Fig. 14(B)-(c), the propagated error signals reached 10^{-1} order at the second epoch as well as the case of (A). However, soon after that, the error signal at the earlier steps decreased more even though the error signal at the output timing (300th step) was not decreased. As shown in Fig. 14(B)-(d), the mean sensitivities became more than 1.0 once, but they decreased below 1.0 after that. By comparing with the 'original' case of (A), it is suggested that SAL is useful not only at the beginning of learning but also works to prevent the loss in sensitivities caused by BPTT during learning.

Secondly, the effect of nonlinear part $f'(U)$ of the sensitivity expressed in Eq. (3) is shown. The nonlinear property influences mainly two parts in SAL. One of them is the second term in the second parenthesis in Eq. (35) for weight update in SAL itself. The other part is that the sensitivity including $f'(U)$ is used as the criteria to decide whether SAL is applied or not. Here, the results of four combinations of the two conditions, each of which is linear or nonlinear, are shown. The case when the non-linear term in Eq. (35) is deleted in SAL is called 'linear SAL'. The case when $|\mathbf{w}_{FB}|$

(a) learning curve

(b) output change

(c) propagated error signal size

(d) Sensitivity and ratio of SAL-applied neurons

(A) SAL was always applied when the sensitivity was below 1.0. A sample of (A) in Fig. 13.

(B) SAL was applied until the sensitivity reached 1.0 for the first time. A sample of (B) in Fig. 13.

Figure 14: Sample learning process when SAL was applied in an RNN together with BPTT and its comparison between two cases (A) and (B) in Table 4. In (B), SAL was not applied again in each neuron once its sensitivity reached 1.0. The initial connection weights are the same between (A) and (B). From the top, (a) change of RMS error over eight patterns, (b) change in the network output for each of eight patterns, (c) RMS of the magnitude of propagated error signal over all the neurons at several timings in 300 steps of backward computation for BPTT. (d) Upper: distribution of sensitivity over all the neurons and steps. Three lines show the mean (center) and standard deviation. Lower: number of neurons in which SAL was applied at least once. (c) and (d) are plotted for each pattern presentation, and so eight points are plotted in order for each epoch. In the 0th epoch, no learning is applied.

Figure 15: Comparison of the early stage of the learning process (sensitivity s, absolute output $|o|$, weight size $|\mathbf{w}|$ of the first three hidden neurons.) depending on whether the nonlinear term is included (A) or not (D) in SAL when SAL was applied to an RNN together with BPTT. Each of them is an example in the case of Fig. 13 (A) or (D), and the initial weights are the same as in Fig. 14.

is used as the criterion to apply SAL instead of using sensitivity $f'(U)|\mathbf{w}_{FB}|$, is called 'linear criterion'. As can be seen in Table 4, both are nonlinear in the 'original' case (A). In the case of (C), SAL itself is nonlinear, but criterion is linear. Then SAL was stopped when $|\mathbf{w}_{FB}|$ was greater than 1.0. The success ratio for the case (C) is less than the 'original' case (A) as shown in Fig. 13. Since $f'(U)$ is usually less than 1.0, the adjustment target value should be greater than 1.0 to keep the sensitivity to be around 1.0, but it is not easy to find the optimal target value.

The result for the combination of 'linear SAL' and 'nonlinear criterion' is shown in Fig. 13(D), but learning succeeded only in one simulation run. To investigate the failures, a sample learning process for the 50 epochs from the beginning is shown in Fig. 15. The initial weights were the same as the case of Fig. 14. In the 'original' case (A), as shown in Fig. 15(A)-(a) and (c), once the sensitivity was greater than 1.0, the weight vector did not increase anymore. The mean absolute value of the output was almost less than 0.5 as in Fig. 15(A)-(b). On the other hand, in the case of linear SAL (D), as in Fig. 15(D)-(a), the sensitivities dropped from around 1.0 in the 2nd epoch in two of the three hidden neurons (the first (blue) and third (green) lines) even though their weight size continued to increase as in Fig. 15(D)-(c). At that time, the mean absolute outputs of the corresponding neurons in (D)-(b) also increased even with a large fluctuation due to the presented pattern. That means that the decrease in $f'(U)$ due to the output increase towards the saturation region caused the sensitivities to decrease. Therefore, it is suggested that the nonlinear term in SAL worked to keep the sensitivities high by avoiding the outputs from entering the saturation area and made the difference in performance between the cases (A) and (D).

18

In the case (E) when SAL and criterion were both linear, learning was successful in half of the runs as shown in Fig. 13(E). Since the linear criterion does not include $f'(\cdot)$, the linear SAL stopped before the weights became very large. The success ratio is only slightly greater than the best ratio when the scale of the weight matrix is manually explored in Fig. 13(G). That suggests that the nonlinear property is the origin of the positive influence of SAL on BPTT learning.

4.2.2 Case of Deep Feedforward Neural Network (DFNN)

Finally, sensitivity adjustment learning (SAL) is applied to a DFNN. To see the effect of the number of layers clearly, an 8-bit parity problem with noise addition is employed as a simple nonlinear problem. There are eight inputs, each of them takes a value of -1 or 1. There is only one output, and if the number of 1 values in the inputs is odd, the ideal value is 0.8, and -0.8 otherwise. The number of patterns is $2^8 = 256$. Number of layers is varied, and each layer has 20 neurons.

Table 5 presents the used parameters. Except for the bottom hidden layer that receives inputs, the learning rate η_{BP} for SGD is the same for all the layers but different depending on the number of layers as shown in Table 6. Initial connection weights are decided randomly in the range of $[-init\_W, init\_W]$. Here, $init\_W$ is called 'initial weight scale' that is also the same for all the hidden layers. The initial values and learning rate for the biases were decided in the same way as the weights in the same layer.

Table 5: Parameters for supervised learning of 8-bit parity problem with noise addition using SAL and BP in a DFNN.

Number of neurons (input, ..., output)		$(8, 20, \ldots, 20, 1)$
Initial connection weights (uniformly random)	input → hidden hidden → output	$[-0.1, 0.1]$
	hidden → hidden	varied or $[-0.1, 0.1]$
Learning rate η_{BP} in Eq. (39)	input → hidden	0.02
Learning rate η_{SAL} in Eq. (35)		0.001
Decay rate β in Eq. (38)		0.99
Noise added to each input (uniformly random)		$[-0.2, 0.2]$
Learning epochs		5000

Table 6: Learning rate η_{BP} in Eq. (39) for the weights other than those from the inputs to the hidden neurons. It depends on the number of layers of the used DFNN.

Number of layers	3	5	10	30	100	200	300	1000
Learning rate	0.01	0.003	0.001	0.0007	0.0005	0.0004	0.0003	0.0001

At first, learning performance is observed by changing the number of layers from 3 to 1000. Fig. 16 shows the mean and standard deviation of log-scaled RMS error over 256 patterns after learning in 20 simulation runs with a different random sequence for initial weights and noises. The NN can learn the problem even with 300 hidden layers without employing special architectures. It can be seen that the performance is improved as the number of layers increases until 300 layers, and the errors lie almost on the linear approximation in the log-scale. In the case of 1000 layers, learning succeeded only once. However, the error was the smallest of all the 20×8 runs and lay on the linear approximation for the range from 3 to 300 layers.

Next, learning performances with various initial weight scales are observed. Fig. 17(A) shows the results when SAL was not applied, and Fig. 17(B) shows the results when SAL was applied. We call them 'without SAL' and 'with SAL', respectively. The errors are plotted for four cases varying the number of layers as 30, 100, 200, 300, and each plot is the average error over 20 simulation runs with different random sequences.

In Fig. 17(A) for 'without SAL', when the initial weight scale was less than 0.4, learning failed even in the case of 30 layers. The learning performance was the best around the initial weight scale from 0.55 to 0.6, where the expected spectral radius (maximum absolute eigenvalue) of the weight matrix is around 1.6. The error around there is lower as the number of layers is smaller. When the weight scale becomes greater than the optimal one, the error increases again. Those can be due to the influence of the vanishing/exploding gradient. However, when the number of layers is

Figure 16: The relation between the number of layers and RMS error over 256 patterns in log-scale after learning when SAL was applied with BP in a DFNN with an initial weight scale of 0.1. The average and standard deviation over 20 simulation runs are shown for each plot. The '×' mark at the 1000 layers shows the error for only one successful run. The line indicates the linear approximation for the range from 3 layers to 300 layers.

(A) without SAL

(B) with SAL

Figure 17: Comparison of the error after supervised learning for various initial weight scales in a DFNN between the two cases of (A) only BP ('without SAL') and (B) SAL+BP ('with SAL'). The number of layers was varied in 30, 100, 200, 300. Each plot shows the RMS error for 256 patterns averaged over 20 simulation runs. In (A), the case without $\tanh$ in BP is also shown only for the case of 30 layers. In (B), the case of applying SAL only until the sensitivity reaches 1.0 for the first time is also shown for the case of 300 layers. The vertical line and the arrow around 0.36 on the horizontal axis indicate that the expected spectral radius of the weight matrix is 1.0.

300, even though the step size of varying the initial weight scale is as small as 0.01, there is no scale with which the error becomes less than around 0.8.

Fig. 17(B) shows the result when SAL is applied with BP. When the initial weight scale was large, learning failed. However, it can be seen that when the initial weight scale was set to a small value like 0.1, learning succeeded in all the 20 simulation runs regardless of the number of layers. The boundary scale between learning success and failure is smaller as the number of layers is greater. From Fig. 16 where the vertical axis is log-scaled, the error is smaller as the number of layers is larger when the weight scale is 0.1. That is the opposite trend of the case of 'without SAL'.

Fig. 18 shows the size of the propagated error signal δ at the bottom hidden layer before learning varying the initial weight scale for the cases of 30, 100, 300 layers. That at the top hidden layer is also plotted for comparison and is around 10^{-1} not depending so much on the number of layers or the initial weight scale. At the arrow around the weight scale of 0.36, the spectral radius of each weight matrix between hidden layers is expected to be 1.0. However, when the scale is around 0.58, the error signal vector has almost the same size between the top and bottom hidden layers. The weight scale is almost the same as when the performance is the best in Fig. 17(A). That suggests the vanishing/exploding gradient made the learning difficult when SAL was not applied.

Figure 18: RMS of the propagated error signal δ at the bottom hidden layer before learning for the cases of 30, 100, 300 layers. For comparison, the error signal at the top hidden layer is also shown by three dotted lines, but because of the overlap of them, it is difficult to find two hidden lines.

On the other hand, when the number of layers was 300, the error never went down for any weight scale as in Fig. 17(A) even though the error signal reached the bottom layer for the weight scale around 0.6. In the case of 200 layers, the error becomes slightly small around initial weight scale 0.58, but the error was considerably smaller when SAL was applied with a small initial weight scale as in Fig. 17(B). That implies that simply adjusting the initial weight scale or spectral radius is insufficient to learn appropriately.

We consider two reasons for the excellent performance when SAL was applied. The first one is the effect of continuous application of SAL. The error when applying SAL only at the beginning of learning, is additionally plotted in Fig. 17(B) for the case of 300 layers. The learning sometimes failed as well as the case of RNN in Fig. 13 (B). That means that the sensitivity needs to be around its moderate level not only at the beginning of learning but also during learning. However, in this case, although it occasionally failed to learn, in many other runs, the error was equivalent to the case when SAL is always applied. The second reason could be the limitation of adjusting the spectral radius of the weight matrix. Even though the spectral radius is appropriate, the sensitivities may not be appropriate for all the neurons, especially when the number of inputs is small. In contrast, SAL makes them moderate in each neuron through learning. Actually, it was confirmed that when the Euclidean norm of the weight vector $|\mathbf{w}|$ was adjusted around 1.2 individually in each neuron before learning, the average RMS error went down around 0.75 even without applying SAL in a 300 layer DFNN.

Finally, processing through layers is investigated by observing the output when continuous random inputs were given to the 20 NNs trained with different initial weights. 1000 sets of 8 continuous uniform random numbers ranging from -1.0 to 1.0 were used as input, and the output was observed for the total of $1000 \times 20 = 20000$ cases. Fig. 19 shows the histogram of the output when the number of layers was varied in 3, 10, 30, 100, 300. As the number of layers increases, the frequency becomes larger for the output being around -0.8 or 0.8. Attractor-like processing through layers can be seen when the number of layers is large. We think that brought out the high learning performance for the noisy inputs as presented in Fig. 16.

21

Figure 19: Histogram of the network output when 1000 continuous random input vectors whose element was decided randomly from -1.0 to 1.0 was given to 20 networks after learning of 8 bit parity problem with noise addition. The number of layers is varied from 3 to 300.

Then the transition of the hidden representation is observed when the number of layers is 300. Fig. 20 shows the hidden outputs for the random 1000 sets of continuous inputs after PCA (principal component analysis) in each of the 1st (bottom), 100th, 199th, 298th (top) hidden layers. Among 20 simulation runs, one sample that makes it easy to see the transition in representation is picked up. It can be seen that the representation is extended and folded gradually like the baker's transformation, and the internal states are divided into two lumps finally. □∘

(a) 1st (lowest) hidden layer

(b) 100th hidden layer

(c) 199th hidden layer

(d) 298th (highest) hidden layer

Figure 20: Change of internal representation after PCA through layers for 1000 continuous random input patterns after learning. (One sample of the 20 networks in Fig. 19) The plot color and shape indicate the final network output as '■'(red): $0.75 < o < 0.85$, '●'(green): $-0.85 < o < -0.75$, '×'(gold): others. Each of the three arrows shows how a specific point or group of points moves through layers.

5   Discussion & Conclusion

We believe that in the future, with the increasing demand for higher functions such as conversation and thinking, the control of dynamics in recurrent neural networks (RNNs) will be more and more critical. Towards such a future, we have shown the possibility that only by controlling the sensitivity in each neuron, the global dynamics of RNNs can be controlled. When applying the sensitivity adjustment learning (SAL) to an RNN, the log sensitivity (for layered RNNs, sum of each layer's log sensitivity) is almost identical to the maximum Lyapunov exponent until the network dynamics get into chaos not depending on the number of neurons, connection ratio or number of layers. In particular, when the sensitivity of each neuron is adjusted to 1.0, the maximum Lyapunov exponent of the entire network becomes 0.0. That means the dynamical state of 'edge of chaos', which is very important from an information-processing perspective, is achieved autonomously in an RNN through SAL.

In gradient-based learning, maintaining a small deviation through layers in the forward computation and maintaining the gradient or error signals in the backward computation are both represented by using the Jacobian of the layer's computation, and are equivalent to each other. Therefore, SAL with 1.0 target sensitivity leads to avoiding 'vanishing gradient' in gradient-based learning in RNNs, which is also valid in deep feedforward neural networks (DFNNs). The combination of SAL and BP or BPTT enables to solve the problems without introducing special architecture or computation except for the general and local learning from small initial weights. The performance is better than when the initial weight scale is finely tuned without applying SAL. When the values of the weights are small, SAL simply increases the magnitude of each weight vector without changing the vector direction, and so it seems to be equivalent to just increasing the weights gradually at a glance. However, the advantage of locality, continuity, and nonlinearity in SAL can be seen from the above learning results as follows.

- The computation is entirely local, and also the decision to stop the SAL can be made autonomously by each neuron itself. However, each neuron cannot compute the spectral radius or eigenvalues of the weight matrix between layers locally.
- Sensitivity is not adjusted for each layer but more finely adjusted for each neuron.
- The sensitivity considers not only linear transformation by the weight vector but also the nonlinear transformation by the activation function. Therefore, different from the adjustment of the spectral radius of the weight matrix, SAL can control the actual dynamics directly considering non-linear processing. By setting the target value to be 1.0, 'edge of chaos' can be realized easily.
- Nonlinear term in SAL prevents the output from going into the saturation area of the activation function. That enables the network to maintain good information transmission in both forward (output) and backward (error) computation.
- Applying SAL with another learning together prevents the loss of sensitivity caused by the other learning.

They give us the hope that SAL will make us free from the fine-tuning of the initial weight and bring us better learning performance than the manual tuning. In reservoirs, which have yielded outstanding results in time-series data processing and temporal pattern generation, an appropriate scale of random weight values is essential. The proposed method can also be an answer to the question of how each neuron gets such moderate weights autonomously.

The following are the immediate issues we are currently facing.

(1) Application to more complicated problems and comparison with other methods
This paper has focused on the analysis of SAL behavior, and we employed simple learning problems when examining the combination of SAL and BP or BPTT. Therefore, it is strongly expected to apply SAL to more practical and large-scale complicated problems and compare the performance with other methods.

(2) SAL for different kinds of input
Another essential unsolved issue is to verify whether SAL works more generally, especially, when a neuron receives both external signals and those from other neurons. The case also should be examined in which the RNN's architecture is not simple, but more complicated with multiple loops. In reinforcement learning problems, the feedback loop through the outer world from actions to perceptions for an agent also influences the network dynamics. In this paper, SAL could control the global dynamics of a two-layer network even though the neurons in the other layer were considered as an external world during learning.

(3) Setting initial connection weights
SAL autonomously adjusts connection weights, but small and random initial weights are still assumed. For being more plausible, a way to determine the initial weights without using random numbers should be investigated, for example, by considering the physical distance to downstream neurons. The influence of non-random initial weights should be also investigated.

(4) Adjustment of learning rate

While we hope SAL reduces the load of tuning the initial weights, naive learning rate adjustment is still necessary.

(5) Conflict to other learning

SAL sometimes increases the error for the other learning like the case in Fig. 14(b). The influence should be investigated in more detail.

(6) SAL on dynamical neurons

It also should be examined whether SAL works in dynamical neurons (continuous-time model), enabling various time constant or chaotic neurons with refractoriness.

SAL is also significant in terms of autonomous, distributed, and asynchronous processing that enables to control the global dynamics of an RNN by local learning in each neuron. In the current neural network computation, we often utilize the parallel processing in GPUs, but they are still under a centralized system. In the future, if the network becomes larger and more flexible like our brain and is trained and utilized in real-time, parallelization at the level of each neuron and autonomous decentralized processing will be essential [Okabe et al. (1998)]. The proposed method in this paper lies in this direction, and we also expect to accelerate researches in this direction.

We believe that SAL shows its potential by using it together with another learning for its original purposes, such as reinforcement learning and supervised learning. Here, we try to generalize the idea about this framework. In an RNN, the other learning forms attractors to increase reproducibility for better states or actions and turns the state transitions from irregular to rational. However, the formation of attractors causes a decrease in sensitivity (chaoticity). As a result, it is possible that the network is stuck in an attractor and remains inactive for a long time. That would mean "death" for the learning agent and should never happen.

If the time average of the sensitivity in each neuron is 1.0, the network dynamics is Edge of Chaos that lies on the boundary between chaos and non-chaos. However, this does not mean that the network always keeps the size of a tiny variation of a signal constant without diverging or converging. When an attractor is formed by the other learning under the constraint of average sensitivity to be 1.0, the attractor is not complete but becomes a pseudo-attractor, and both convergence and divergence appear alternately while being balanced in the course of time. That enables the agent can balance exploration and learning and maintain autonomous state transitions, ensuring the agent does not reach the "death". In the high-dimensional space formed by a large number of neurons, pseudo-attractors would created through learning and chaotic itinerancy among them emerges, which leads to realizing "thinking" including "inspiration" and "discovery", we expect.

We are living not only in space but also in time. However, most of the existing learning methods have focused on the state or output at a timing. We can express it as a point in space, and learning has moved it to a better place. Even though we use an RNN and its state or output changes over time, learning has not taken into account its flow or lines in space. The sensitivity is an index for the flow in the processing of one neuron. We hope the sensitivity is the key to developing learning methods to control the flow or dynamics. What we want to do most is to bring this idea of "learning of dynamics" into reinforcement learning and establish a new paradigm. Learning does not aim to improve the output or state at a specific time but improve the dynamics from the viewpoint of "exploration" or "reproducibility" in high-dimensional systems. We believe that is a fundamental idea towards the ultimate goal: "emergence of thinking".

Acknowledgment

References

J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization, 2016. arXiv:1607.06450.

Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994. doi:10.1109/72.279181.

J. Boedecker, O. Obst, J. T. Lizier, N. M. Mayer, and M. Asada. Information processing in echo state networks at the edge of chaos. *Theory in Biosciences*, 131(3):205–213, 2012. ISSN 16117530. doi:10.1007/s12064-011-0146-8.

J. Chung, Ç. Gülçehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. 2014. arXiv:1412.3555.

H. I. Fawaz, G. Forestier, J. Weber, L. Idoumghar, and P. Muller. Deep learning for time series classification: a review. 2018. arXiv:1809.04356.

W. J. Freeman. The physiology of perception. *Scientific American*, 264(2):78–85, February 1991. doi:10.1038/scientificamerican0291-78.

Y. Goto and K. Shibata. Influence of the chaotic property on reinforcement learning using a chaotic neural network. In D. Liu and et al., editors, *Neural Information Processing. ICONIP 2017. Lecture Notes in Computer Science*, volume 10634, pages 759–767. Springer International Publishing, 2017. doi:10.1007/978-3-319-70087-8_78.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. 2015. arXiv:1512.03385.

S. Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *Int. J. Uncertain. Fuzziness Knowledge-Based System*, 6(2):107–116, 1998. ISSN 0218-4885. doi:10.1142/S0218488598000094.

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, Nov. 1997. ISSN 0899-7667. doi:10.1162/neco.1997.9.8.1735.

I. Ighneiwaa, S. Hamidatoua, and F. B. Ismaela. Using artificial neural networks (ANN) to control chaos. 2017. arXiv:1701.00754.

S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proc. of the 32nd Int'l Conf. on Machine Learning*, volume 37 of *ICML'15*, pages 448–456. JMLR.org, 2015. https://proceedings.mlr.press/v37/ioffe15.pdf.

L. Jiao, F. Zhang, F. Liu, S. Yang, L. Li, Z. Feng, and R. Qu. A survey of deep learning-based object detection. 2019. arXiv:1907.09408.

G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter. Self-normalizing neural networks. In *Advances in Neural Information Processing Systems 30 (NIPS)*, 2017. https://dl.acm.org/doi/epdf/10.5555/3294771.3294864.

C. G. Langton. Computation at the edge of chaos: phase transitions and emergent computation. *Physical D: Nonlinear Phenomena*, 42:12–37, 1990. doi:10.1016/0167-2789(90)90064-V.

R. Legenstein, S. M. Chase, A. B. Schwartz, and W. Maass. A reward-modulated hebbian learning rule can explain experimentally observed network reorganization in a brain control task. *J. of Neuroscience*, 30:8400–8410, 2010. doi:10.1523/JNEUROSCI.4284-09.2010.

T. Matsuki and K. Shibata. Reward-based learning of a memory-required task based on the internal dynamics of a chaotic neural network. In A. Hirose and et al., editors, *Neural Information Processing. ICONIP 2016. Lecture Notes in Computer Science*, volume 9947, pages 376–383. Springer International Publishing, 2016. doi:10.1007/978-3-319-46687-3_42.

T. Matsuki and K. Shibata. Adaptive balancing of exploration and exploitation around the edge of chaos in internal-chaos-based learning. *Neural Networks*, 132:19–29, 2020. doi:10.1016/j.neunet.2020.08.002.

A. Nassif, I. Shahin, I. Attili, M. Azzeh, and K. Shaalan. Speech recognition using deep neural networks: A systematic review. *IEEE Access*, 7:19143–19165, 2019. doi:10.1109/ACCESS.2019.2896880.

Y. Okabe, T. Kouhara, H. Hayashi, A. Narusawa, M. Kitagawa, and K. Miyao. Moderatism: New concept for self-organization of neural networks. In *Proc. of 2nd Int'l Conf. on Knowledge-based Intelligent Electronic Systems*, pages 246–249, 1998. doi:10.1109/KES.1998.725979.

Y. Osana and M. Hagiwara. Successive learning in hetero-associative memory using chaotic neural networks. *Int'l Journal of Neural Systems*, 9:285–299, 1999. doi:10.1142/S0129065799000290.

E. Ott, C. Grebogi, and J. A. Yorke. Controlling chaos. *Physical Review Letter*, 64:1196–1199, 1990. doi:10.1103/PhysRevLett.64.1196.

D. W. Otter, J. R. Medina, and J. K. Kalita. A survey of the usages of deep learning in natural language processing. 2018. arXiv:1807.10854.

R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks. In *Proceedings of the 30th Int'l Conf. on Machine Learning - Volume 28*, ICML'13, pages III–1310–1318. JMLR.org, 2013. URL `https://dblp.org/rec/conf/icml/PascanuMB13`.

T. Salimans and D. P. Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In D. D. Lee and et al., editors, *Advances in Neural Information Processing Systems 29*, pages 901–909. Curran Associates, Inc., 2016. `https://proceedings.neurips.cc/paper/2016/file/ed265bc903a5a097f61d3ec064d96d2e-Paper.pdf`.

K. Sato, Y. Goto, and K. Shibata. Chaos-based reinforcement learning when introducing refractoriness in each neuron. In K. JH. and et al., editors, *Robot Intelligence Technology and Applications. RiTA 2018. Communications in Computer and Information Science*, volume 1015, pages 76–84, 2019. doi:10.1007/978-981-13-7780-8_7.

Y. Sawatsubashi, M. F. bin Samsudin, and K. Shibata. Emergence of discrete and abstract state representation through reinforcement learning in a continuous input task. In *Advances in Intelligent Systems and Computing, Robot Intelligence Technology and Applications 2012*, pages 13–22, 2012. doi:10.1007/978-3-642-37374-9_2.

K. Shibata. Functions that emerge through end-to-end reinforcement learning – the direction for artificial general intelligence –. In *The 3rd Multidiscipliary Conf. on Reinforcement Learning and Decision Making (RLDM)17*, 2017. arXiv:1703.02239.

K. Shibata and K. Goto. Emergence of flexible prediction-based discrete decision making and continuous motion generation through actor-q-learning. In *Proc. of ICDL-Epirob 2013*, page ID 15, 2013. doi:10.1109/DevLrn.2013.6652559.

K. Shibata and Y. Goto. New reinforcement learning using a chaotic neural network for emergence of "thinking" - "exploration" grows into "thinking" through learning -. 2017. arXiv:1705.05551.

K. Shibata and Y. Sakashita. Reinforcement learning with internal-dynamics-based exploration using a chaotic neural network. In *Proc. of Int'l Joint Conf. on Neural Networks (IJCNN) 2015*, page #15231, 2015. doi:10.1109/IJCNN.2015.7280430.

K. Shibata and M. Sugisaka. Dynamics of a recurrent neural network acquired through learning a context-based attention task. *Aritificial Life Robotics*, 7:145–150, 2004. doi:10.1007/BF02471196.

K. Shibata and H. Utsunomiya. Discovery of pattern meaning from delayed rewards by reinforcement learning with a recurrent neural network. In *Proc. of IJCNN 2011*, pages 1445–1452, 2011. doi:10.1109/IJCNN.2011.6033394.

C. A. Skarda and W. J. Freeman. How brains make chaos in order to make sense of the world. *Behavioral and brain sciences*, 10:161–173, 1987. doi:10.1017/S0140525X00047336.

J. Sprott. *Chaos and time-series analysis*, chapter 5.6. Oxford University Press, Oxford, 2010. doi:10.1007/s12064-011-0146-8.

D. C. Sussillo. *Learning in Chaotic Recurrent Neural Networks*. PhD thesis, USA, 2009. URL `https://www.researchgate.net/publication/36712167_Learning_in_chaotic_recurrent_neural_networks`. AAI3346497.

H. Utsunomiya and K. Shibata. Contextual behavior and internal representations acquired by reinforcement learning with a recurrent neural network in a continuous state and action space task. In *Advances in Neuro-Information Processing, Lecture Note in Computer Science*, volume 9, pages 970–978, 2009. doi:10.1007/978-3-642-03040-6_118.

I. B. Yildiz, H. Jaeger, and S. J. Kiebel. Re-visiting the echo state property. *Neural networks*, 35:1–9, 2012. doi:10.1016/j.neunet.2012.07.005.