

RESEARCH ARTICLE

A shared synapse architecture for efficient FPGA implementation of autoencoders

Akihiro Suzuki*, Takashi Morie, Hakaru Tamukoh

Graduate School of Life Science and Systems Engineering, Kyushu Institute of Technology, 2-4 Hibikino, Wakamatsu-ku, Kitakyushu 808-0196, Japan

* suzuki-akihiro@edu.brain.kyutech.ac.jp



Abstract

This paper proposes a shared synapse architecture for autoencoders (AEs), and implements an AE with the proposed architecture as a digital circuit on a field-programmable gate array (FPGA). In the proposed architecture, the values of the synapse weights are shared between the synapses of an input and a hidden layer, and between the synapses of a hidden and an output layer. This architecture utilizes less of the limited resources of an FPGA than an architecture which does not share the synapse weights, and reduces the amount of synapse modules used by half. For the proposed circuit to be implemented into various types of AEs, it utilizes three kinds of parameters; one to change the number of layers' units, one to change the bit width of an internal value, and a learning rate. By altering a network configuration using these parameters, the proposed architecture can be used to construct a stacked AE. The proposed circuits are logically synthesized, and the number of their resources is determined. Our experimental results show that single and stacked AE circuits utilizing the proposed shared synapse architecture operate as regular AEs and as regular stacked AEs. The scalability of the proposed circuit and the relationship between the bit widths and the learning results are also determined. The clock cycles of the proposed circuits are formulated, and this formula is used to estimate the theoretical performance of the circuit when the circuit is used to construct arbitrary networks.

OPEN ACCESS

Citation: Suzuki A, Morie T, Tamukoh H (2018) A shared synapse architecture for efficient FPGA implementation of autoencoders. PLoS ONE 13(3): e0194049. <https://doi.org/10.1371/journal.pone.0194049>

Editor: Jun Ma, Lanzhou University of Technology, CHINA

Received: November 6, 2017

Accepted: February 25, 2018

Published: March 15, 2018

Copyright: © 2018 Suzuki et al. This is an open access article distributed under the terms of the [Creative Commons Attribution License](https://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

Data Availability Statement: All relevant data are within the paper.

Funding: This work was supported by JSPS KAKENHI Grant Number 15H01706, URL <https://kaken.nii.ac.jp/en/grant/KAKENHI-PROJECT-15H01706/>; JSPS KAKENHI Grant Number 17H01798, URL <https://kaken.nii.ac.jp/en/grant/KAKENHI-PROJECT-17H01798/>; and JSPS KAKENHI Grant Number 17K20010, URL <https://kaken.nii.ac.jp/en/grant/KAKENHI-PROJECT-17K20010/>. The funders had no role in study

Introduction

Deep neural networks (DNNs) are known for their high levels of performance in machine learning applications [1]. DNNs are general models of deeply stacked neural networks and include multi-layer perceptrons (MLPs) and convolutional neural networks (CNNs), restricted Boltzmann machines (RBMs) and autoencoders (AEs).

MLPs are the most basic DNNs model and they are applied to classification tasks [2]. CNNs are modeled on the structure of the human visual cortex and mainly focus on image recognition tasks [3], while RBMs are generative models that can learn a probability distribution over an input dataset [1, 4]. AEs obtain a representation for an input dataset and have several versions [5–8]. AEs are used in a wide variety of different applications; low-light image enhancement [9] and dimensionality reduction for image processing [10] in image processing domain;

design, data collection and analysis, decision to publish, or preparation of the manuscript.

Competing interests: The authors have declared that no competing interests exist.

the hash function for data mining [11], tomographic reconstruction from dynamic positron emission tomography [12], and the extractor for latent representations of documents [13].

In most DNNs, processes are divided into pre-training and fine-tuning phases [14]. RBMs and AEs are stacked so as to be used in the pre-training phase [15, 16]. In this phase, the objectives are to train a part of an entire network and obtain effective initial parameters for the fine-tuning via unsupervised learning. In the fine-tuning phase, the pre-trained network is used as a feature extractor so that the entire network, including a classifier, can be tuned via supervised learning. In both phases, a great amount of data and computational resources are required for learning.

It is expected that in the future, DNNs will be loaded onto embedded systems such as mobile devices, robots, and automobiles [17–19]. Because embedded systems with DNNs are expected to be used in everyday items, the embedded systems should have the following features: real-time processing, low power consumption, and situational-responsive capabilities. With all of these considerations in mind, hardware is considered to be a better choice than software for these systems. The pieces of hardware often used for the implementation of DNNs are graphics processing units (GPUs), dedicated very large-scale integration (VLSI) chips, and field-programmable gate arrays (FPGAs).

Several libraries and frameworks have been developed for the implementation of DNNs via GPUs; these include Theano (which is a Python library) [20] and Caffe (a deep learning framework) [21], Tensor Flow and Chianer (Python-based deep learning frameworks) [22, 23]. Some DNNs, such as some MLPs [24, 25], RBMs [26, 27], and CNNs [28–31], have been developed as dedicated chips. One of the report uses RBMs for training and AEs for inference [32]. In addition to these examples of the hardware implementations of DNNs, FPGA implementations of CNNs and RBMs have also been reported in [33–38]. Unlike other hardware implementations, FPGAs are more suitable for implementing DNNs onto embedded systems, as FPGAs require less power than GPUs and, unlike dedicated chips, their internal circuits can be reconfigured by users.

A few reports have also been published about the implementation of AE into FPGAs [39, 40]. One of these proposed a digital circuit of AEs by high-level synthesis (HLS) [39]. In that particular study, the circuit generated by HLS was loaded onto an Altera Stratix V GS D5 FPGA. However, a comparison between regular GPUs, mobile GPUs, and FPGAs found that the HLS-designed circuit had several drawbacks. Because HLS automatically generates a circuit architecture from an algorithmic description, the generated circuit cannot effectively utilize FPGAs. The other relevant study to discuss here created a behavior model simulation for AEs [40]; in the simulation, a sparse AE was implemented and used for an image recognition task on a set of Kyoto pictures. Because the circuit used in the simulation required too many resources from the FPGA, it was impossible for the circuit to be implemented onto an actual FPGA. As stated above, the study of FPGA implementation for AEs has been still insufficient. The implementation of register transfer level (RTL) description is required to utilize the FPGA effectively, and small-but-adjustable architectures for AEs that can be implemented on actual FPGAs are also required.

This paper proposes a novel hardware architecture for AEs called a shared synapse architecture, and it demonstrates the results of implementing such AEs onto a digital circuit. This paper is an extended version of work published in [41]. The sharing synapse was originally reported in an earlier study [5], and another earlier study [42] called the method of sharing synapses as a tied weight technique; this technique restricts AEs so as to improve the performance of AEs' reconstruction. We use a tied weight technique to efficiently use the resources of the FPGAs in our study. By assuming that the reconstruction of an AE is divided into those

in the neuron and synapse modules, the shared synapse architecture is able to halve the number of the synapse modules.

In this paper, a digital circuit with the shared synapse architecture is implemented via a RTL description. The circuit is designed in an object-oriented manner and implemented as a basic model that could be scaled. In our experiments, we investigate the performance and scalability of the proposed digital circuit. This paper is organized as follows: Section 2 explains the basic algorithms of the AEs used along with the algorithm related to the proposed idea; in Section 3, we propose and describe in detail a novel hardware architecture for AEs; Section 4 looks at the implementation of the proposed AEs on a digital circuit via software experiments; Section 5 shows the results of the logic synthesis and verifies the circuit operation as AEs; in Section 6, the scalable functions of the proposed design circuit are investigated; Section 7 compares the present work with other related works; and in Section 8, we put forward our conclusions.

Autoencoders

Learning algorithm of autoencoders

As shown in Fig 1, AEs are composed of three layers: an input, a hidden, and an output layer. Each layer has neurons that are connected to each other via synapses between the layers. The data of the neurons are handed over to the next layer via synapses, and each synapse has a weight value representing the transmission efficiency. In the hidden and output layers, each neuron has a bias that controls the firing probability.

The difference between the input and output values is measured by the cross-entropy error. This error is minimized by the gradient descent method, and the values of the weights and the biases are updated so that this error is reduced. These processes are repeated until the output values become to reconstruct input values.

If an input vector is assumed to be \vec{x} , a hidden one \vec{y} , and an output one \vec{z} , then the reconstruction and update flow can be represented by the sequence of equations shown below. In

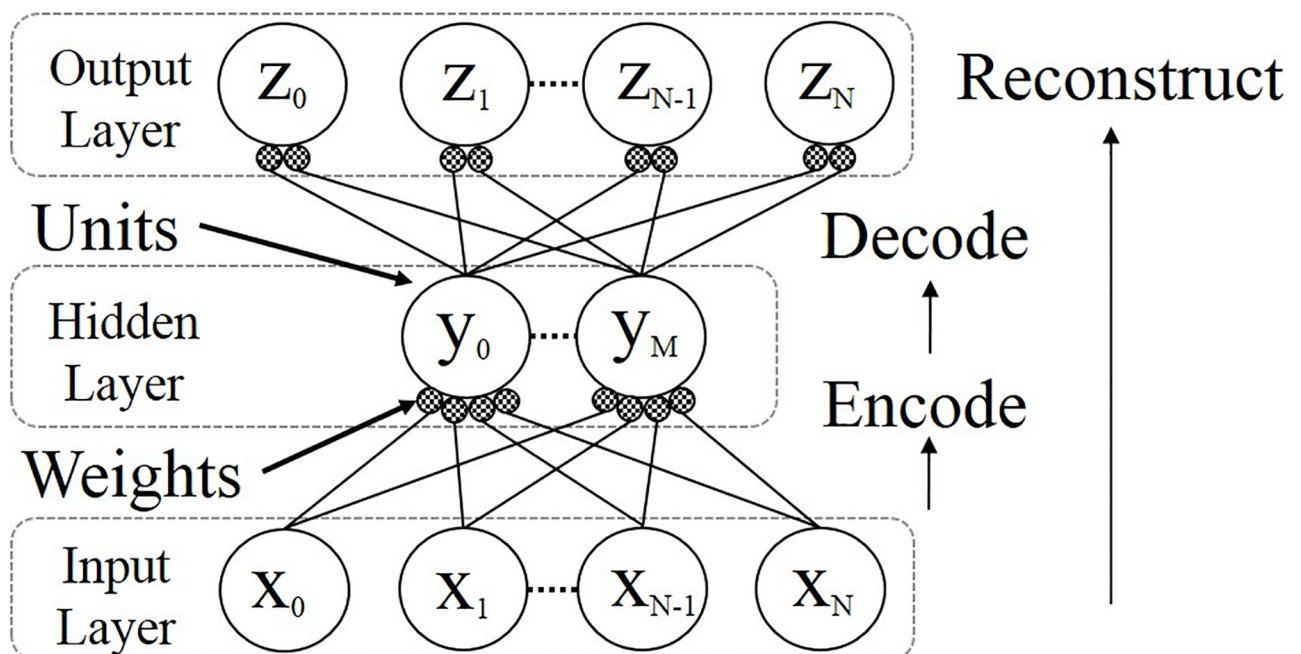


Fig 1. Autoencoder. Reprinted from [41] under a CC BY license, with permission from Springer Nature, original copyright 2016 (S1 File).

<https://doi.org/10.1371/journal.pone.0194049.g001>

this flow, the hidden weight and bias vectors are \vec{W} and \vec{b} , respectively, while the output ones are \vec{W}' and \vec{b}' , respectively. The function σ is an activation function such as a sigmoidal function, and η is the learning rate. It should be noted that $\mathbf{1}$ is a vector of all ones, \cdot is an inner product, and \circ is the Hadamard product. The reconstruction and update flow can be represented as follows:

1. \vec{x} is encoded and transformed in the hidden layer thereby becoming \vec{y} :

$$\vec{y}_m = \sigma\left(\sum_{n=0}^N \vec{W}_{mn} \cdot \vec{x}_n + \vec{b}_m\right) \quad m = 0, 1, \dots, M - 1, M \quad (1)$$

2. \vec{y} is decoded and transformed in the output layer, thereby becoming \vec{z} :

$$\vec{z}_n = \sigma\left(\sum_{m=0}^M \vec{W}'_{nm} \cdot \vec{y}_m + \vec{b}'_n\right) \quad n = 0, 1, \dots, N - 1, N \quad (2)$$

3. The cross entropy, C, is measured using \vec{x} and \vec{z} :

$$C(\vec{x}, \vec{z}) = -\vec{x} \log \vec{z} - (\mathbf{1} - \vec{x}) \log (\mathbf{1} - \vec{z}) \quad (3)$$

4. The update values for each parameters are obtained from the cross entropy error.

$$\Delta \vec{W} = \left(\vec{W} \cdot (\vec{x} - \vec{z}) \circ \vec{y} \circ (\mathbf{1} - \vec{y}) \cdot \vec{x}^T \right) + \left(((\vec{x} - \vec{z}) \cdot \vec{y}^T)^T \right) \quad (4)$$

$$\Delta \vec{b} = \left(\vec{W} \cdot (\vec{x} - \vec{z}) \circ \vec{y} \circ (\mathbf{1} - \vec{y}) \right) \quad (5)$$

$$\Delta \vec{b}' = (\vec{x} - \vec{z}) \quad (6)$$

5. Each update value is added to the old parameters, which allows the new parameters to be obtained.

$$\vec{W}^{new} = \vec{W}^{old} + \eta \Delta \vec{W} \quad (7)$$

$$\vec{b}^{new} = \vec{b}^{old} + \eta \Delta \vec{b} \quad (8)$$

$$\vec{b}'^{new} = \vec{b}'^{old} + \eta \Delta \vec{b}' \quad (9)$$

In this paper, these equations are computed on digital circuits; Eqs (1) and (2) are computed in a reconstruction module, Eqs (4), (5) and (6) at an update function module, and Eqs (7), (8) and (9) at an update execution module.

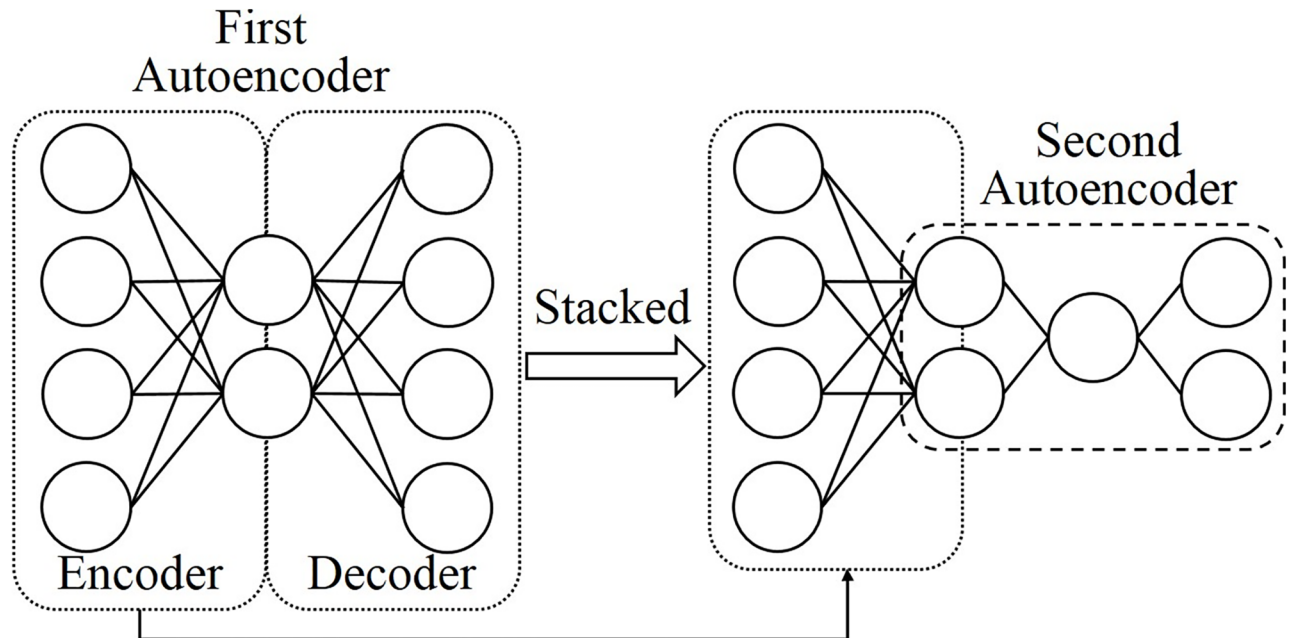


Fig 2. Stacked autoencoder.

<https://doi.org/10.1371/journal.pone.0194049.g002>

Stacked autoencoders

Fig 2 shows a stacked AE composed of two AEs. The learning process of the stacked AE is as follows:

1. The first AE (left-hand side of Fig 2) is trained by the learning algorithm described in Section 2.1 so that it can reconstruct the input dataset.
2. The second AE is stacked on the hidden layer of the first AE (right-hand side of Fig 2).
3. The second AE is trained by the same learning algorithm so that it can reconstruct the value encoded by the first AE.

After the learning process, the stacked AE is able to reduce the dimension of the input datasets and can be used as a feature extractor.

Tied weight

Tied Weight is a technique for restricting neural networks [42]; this is often used in AEs. The technique shares synapses between layers; namely, it shares synapses between the input and hidden layers, and the hidden and output layers. The weight matrix between the hidden and output layers is represented as a transposed matrix of the input and hidden layers; this is shown in Eq (10).

$$\vec{W}' = \vec{W}^T \tag{10}$$

By using the tied weight technique, AEs can be used to reduce the parameters that need to be updated and can improve the regularization ability.

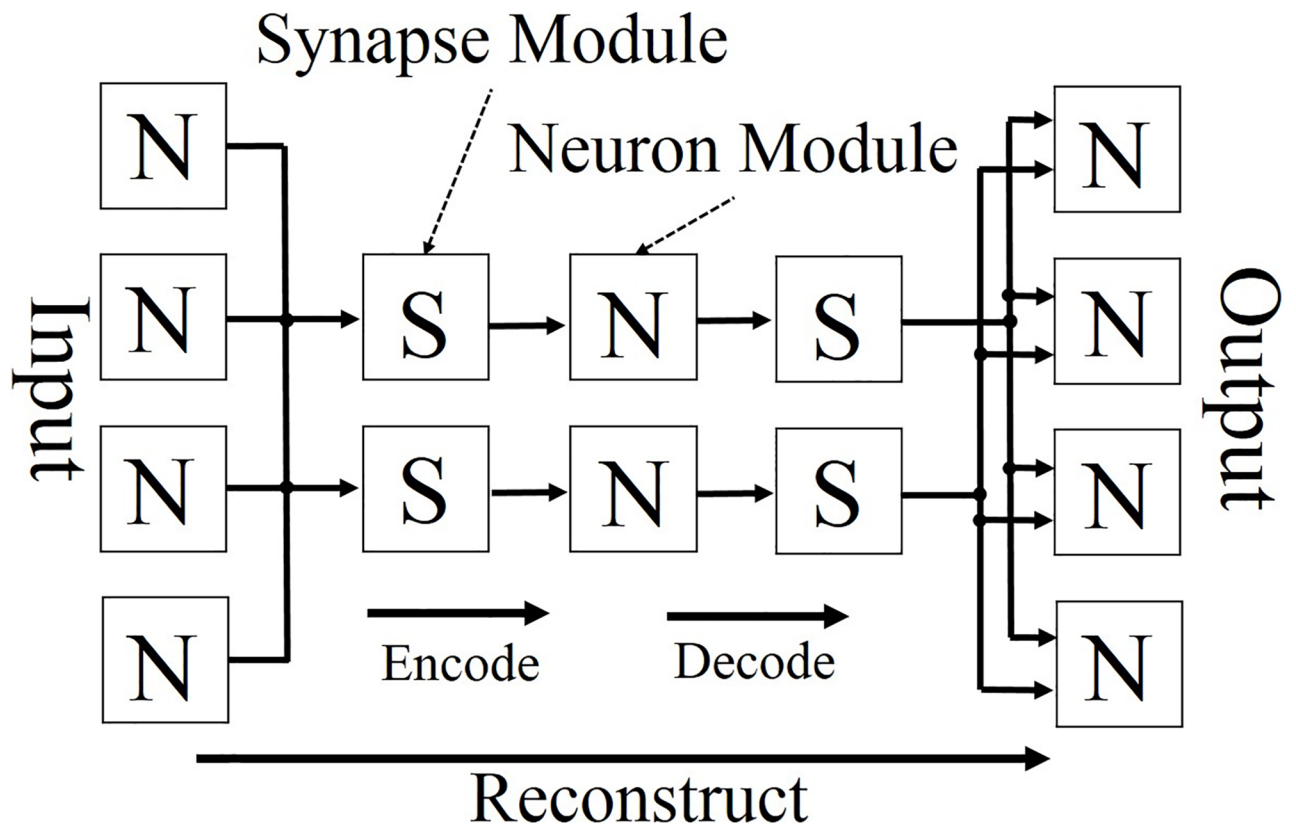


Fig 3. Reconstruction of AE. Reprinted from [41] under a CC BY license, with permission from Springer Nature, original copyright 2016 (S1 File).

<https://doi.org/10.1371/journal.pone.0194049.g003>

Shared synapse architecture

This paper proposes a shared synapse architecture based on the tied weight technique; the aim of this architecture is to decrease the utilizations of the recourses of the FPGAs. It is assumed that the reconstruction process for the AEs is described in the neuron and synapse modules; this process is shown in Fig 3. By sharing the synapses, the number of synapse modules used is halved; a representation of the architecture of an AE with shared synapses is shown in Fig 4. The data flow should be noted. Even though this architecture shares the synapse module, the data flow and timing of module driving do not change the operation order, which remains the same as that of a regular AE. The outputs of hidden neurons repass again the same synapse modules that were passed once; then they do not go forward to the hidden layer but to the output layer instead. Therefore, the synapse modules are executed twice according to every set of encode and decode processes.

The circuit proposed by this study consists of three modules, as shown in Fig 5: a reconstruction module, an update function module, and an update execution module. The reconstruction and update function modules contain sub-modules, and the internal structures of each of these two modules are shown individually in Figs 6 and 7, respectively.

The roles of these three modules and their respective sub-modules are as described in the following sentences in more detail. Eighteen-bit fixed point numbers are used in the modules shown in Figs 6 and 7; these are the memory for the weights, the multiplier, the memory for the bias, the accumulator, the adder, and the subtractor. The activation function module

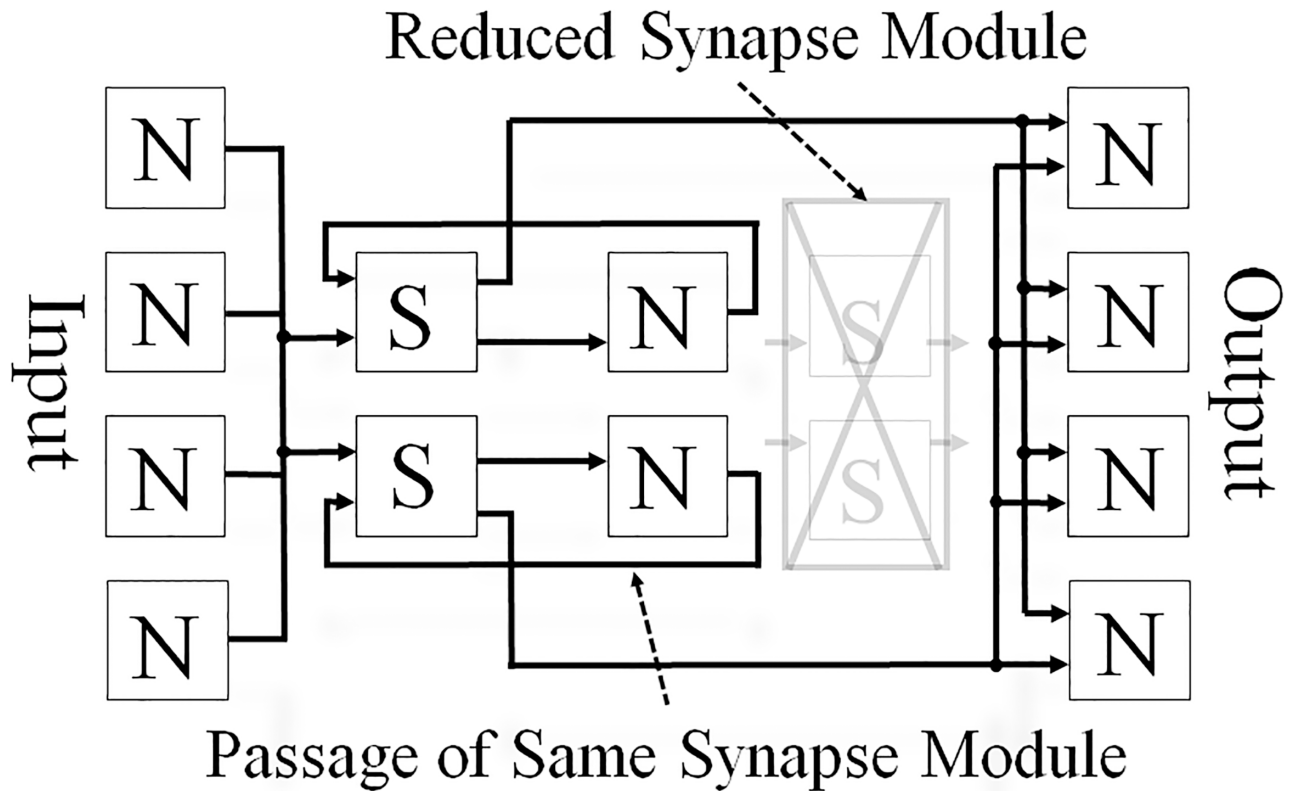


Fig 4. Reconstruction of AE with shared synapse architecture. Reprinted from [41] under a CC BY license, with permission from Springer Nature, original copyright 2016 (S1 File).

<https://doi.org/10.1371/journal.pone.0194049.g004>

shown in Fig 6 is designed using a look-up table (LUT), and it uses ten-bit numbers as inputs and eighteen-bit numbers as outputs. The values in the LUT can be rewritten so as to become other activation functions:

(a) Reconstruction Module

This module is composed of two sub-modules, as shown in Fig 6 and it has a reconstruction mode and an update mode.

- Reconstruction Mode

Via two processes (encode and decode), the input data are reconstructed on as output data.

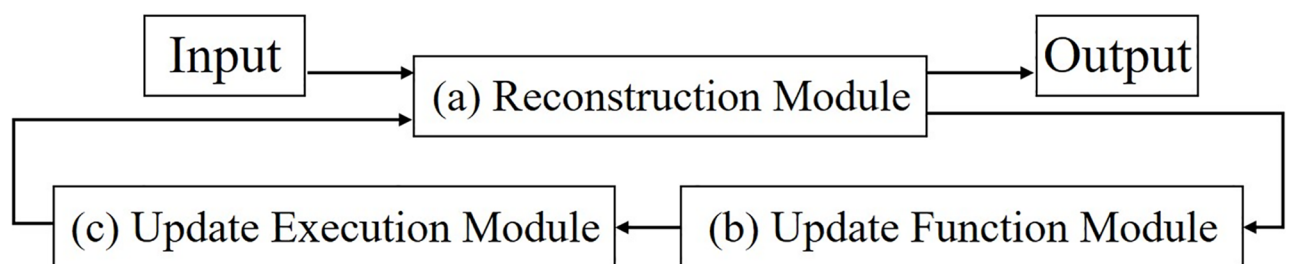


Fig 5. Entire circuit.

<https://doi.org/10.1371/journal.pone.0194049.g005>

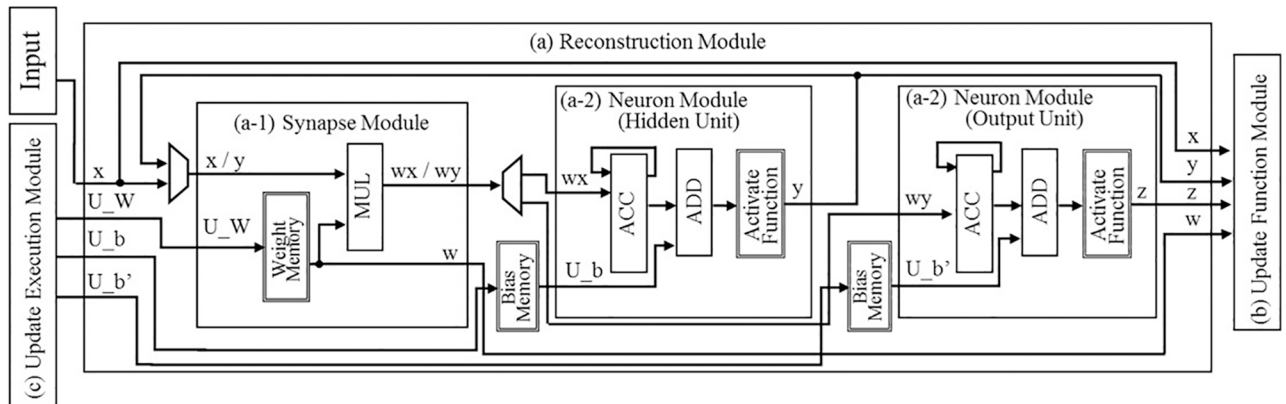


Fig 6. Reconstruction module.

<https://doi.org/10.1371/journal.pone.0194049.g006>

- a-1) Synapse Module
A weight value is read from the weight memory. The input data and corresponding weight values are multiplied.
- a-2) Neuron Module
The accumulation of all of the inputs and biases are determined. From the LUT, the corresponding value of the accumulation is given as a result output by the sigmoidal operation.
- Update Mode
This module receives the update values from the update execution module and updates all of the parameters in the weight memory and two bias memories.

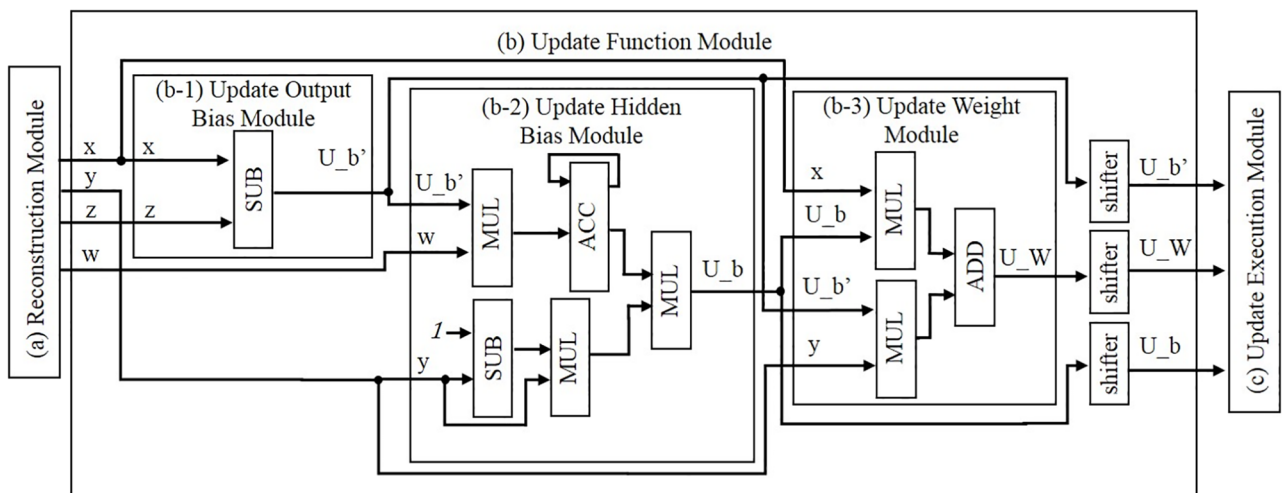


Fig 7. Update function module.

<https://doi.org/10.1371/journal.pone.0194049.g007>

(b) Update Function Module

This module is composed of three sub-modules, as shown in Fig 7. Utilizing a given input, output, and all of the parameters sent by the reconstruction module, the update values of the parameters are determined.

b-1) Update Output Bias Module

An input and an output value are sent from the reconstruction module. The update value of \vec{b} , is determined by Eq (6).

b-2) Update Hidden Bias Module

The values for the calculation are sent from the other modules; a weight and a hidden value from the reconstruction module, and the output of the update output bias module. The update value of \vec{b} is determined by Eq (5).

b-3) Update Weight Module

The values for the calculation are sent from the other modules; an input and an output value from the reconstruction module, the output of the output bias update module, and the output of the update hidden bias module. The update value of \vec{W} is determined by Eq (4).

(c) Update Execution Module

The update values of the parameters are given by the update function module. Each update value is added to a corresponding parameter by this module. Each updated value is then sent to the reconstruction module in the update process. Another function of this module is to switch the reconstruction module from the reconstruction mode to the update mode before the updated values are transferred to the reconstruction module. After the update, the update execution module switches from the update mode to the reconstruction mode.

Software implementation of the proposed AE

Because the number of processes involved in the hardware implementation is greater than the number for the software implementation, it is advisable to first verify the hardware implementation using software implementation. A quantized AE was implemented into a piece of software coded by C++. The quantized AE is able to handle eighteen-bit fixed-point numbers. In addition, an array with 1,024 elements was used as a quantized sigmoidal function generator in the piece of software.

The number of input and output neurons was set to four, and the number of hidden neurons was set to two; in total, sixteen sets of four tuples of binary data were prepared for the learning process. Both the quantized AE and a regular AE with 32-bit floating-point numbers were trained with one set from the dataset for each learning experiment. The number of epochs was set to 1,000, and the learning rate was set to 0.0078125. The initial value of each parameter of the AEs was set to zero.

Fig 8 shows the averages of the results of the sixteen learning processes using the quantized and regular AEs. In the figure, the straight line indicates the results of the quantized AE, while the dotted line indicates the results of the regular AE; the vertical and horizontal axes indicate the cross entropy and epochs, respectively. The results show that the quantized AE operated better than the regular AE. The cross-entropy errors of both of the AEs decreased as the learning processes proceeded. The cross-entropy errors of the quantized AE converged faster than those of the regular AE; this was because the update process of the quantized AE operated rougher than the regular AE.

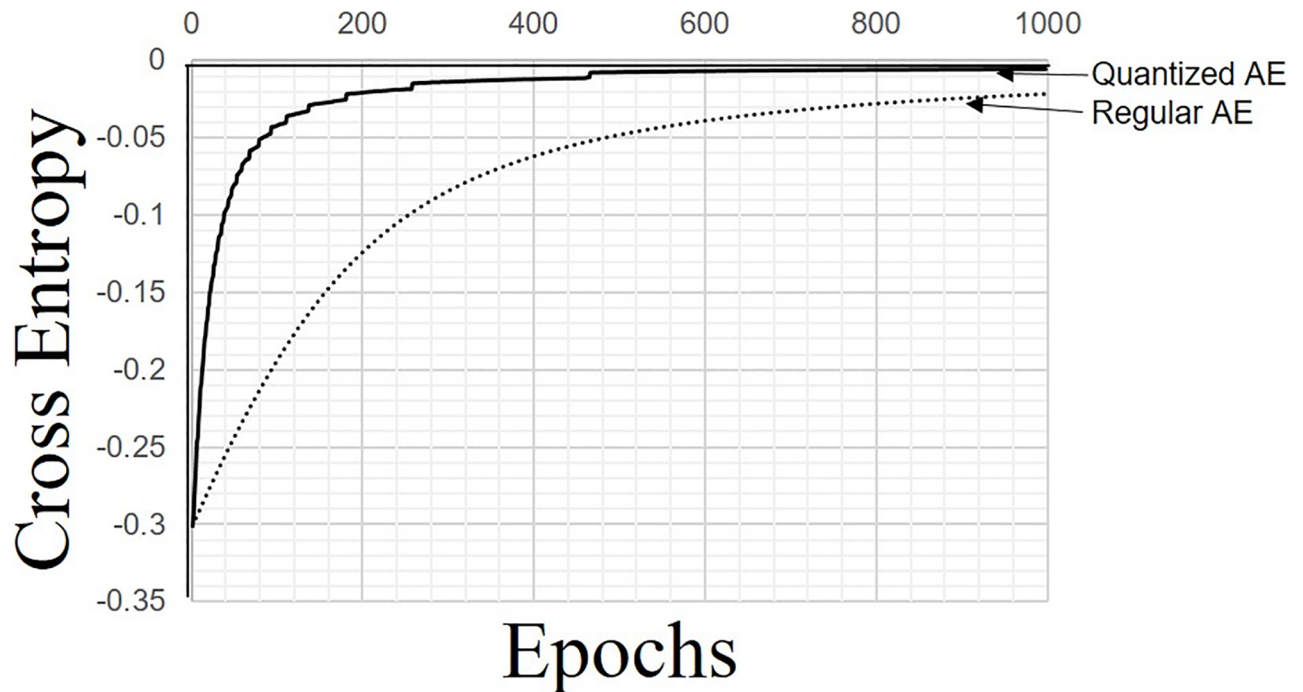


Fig 8. Comparison of learning results.

<https://doi.org/10.1371/journal.pone.0194049.g008>

Hardware implementation of the proposed AE

As the effectiveness of the proposed architecture was validated via the experiments conducted by the software implementation, the proposed AE was implemented in a digital circuit. As with the software implementation, the designed network had four input and output neurons and two hidden neurons. The digital circuit with the proposed design was described by the Verilog Hardware Description Language (HDL) with RTL.

Logic synthesis of the proposed circuit

The designed circuit was synthesized using Xilinx ISE 14.7, and the target device used was a Xilinx Virtex-6 xc6vlx240t. The results of the synthesis are shown in Table 1.

Table 1. Logic synthesis results for the shared synapse AEs.

Module	Registers	LUTs	DSPs*	Freq. (MHz)
Entire AE module	6,284 (2.08%)	6,198 (4.11%)	30 (3.91%)	230.654
Reconstruction	2,621 (0.87%)	2,442 (1.62%)	2 (0.26%)	242.777
Synapse	270 (0.09%)	262 (0.17%)	1 (0.13%)	242.424
Neuron	102 (0.03%)	153 (0.10%)	0 (0%)	363.769
Update function	6125 (2.03%)	6557 (4.35%)	28 (3.65%)	230.654
Update output bias	667 (0.22%)	454 (0.30%)	0 (0%)	419.639
Update hidden bias	1139 (0.38%)	1190 (0.79%)	6 (0.78%)	242.777
Update weight	399 (0.13%)	411 (0.27%)	2 (0.26%)	242.777
Update Execution	763 (0.25%)	7(0.004%)	0 (0%)	651.042

* Digital signal processors

<https://doi.org/10.1371/journal.pone.0194049.t001>

In addition to the modules shown in this table, several other sub-modules were used to control the weight memory; these were synthesized, and they consisted of a serial-parallel converter, a parallel-serial converter, and a signal selector. The proposed shared synapse architecture therefore reduced the number of the synapse modules as well as the number of the sub-modules belonging to the synapse module. Moreover, the larger the network that was constructed, the greater the reductions in the number of the module increase and the more effective the proposed architecture is for the implementation of AEs into FPGAs.

Verification of learning performance

To verify the performance of the designed AE circuit, the proposed circuit was logically simulated by a Veritak Verilog HDL simulator. As in the software experiments, the proposed designed circuit was trained using sixteen sets of four-bit binary data. The learning rate was set at 0.0078125, which corresponds to 2^{-7} , and therefore the multiplication of the update values with this learning rate were replaced by seven-bit shifting. The number of the epochs was set to 365.

The outputs produced after training are shown in Table 2. The cross entropy error between the corresponding input and output values was measured by Eq (3), and it can be seen from Table 2 that the output neurons reconstructed the corresponding input values. The averages of the four errors in each of the learning epochs are shown in Fig 9; in this figure, the vertical and horizontal axes express the cross entropy and epochs, respectively. As can be seen in Fig 9, as the number of epochs increased, the value of errors decreased. It can therefore be concluded that the AE circuit adjusted the parameters so that the input data could be reconstructed as the output data.

Evaluation of processing speed performance

To evaluate the processing speed performance of the proposed circuits, a processing speed performance index for hardware and the number of multipliers are computed.

“Operations per second (OPS)” is generally used as a performance index for hardware processing speed, and we have employed OPS. OPS represents the number of operations per second for updating the network according to Eqs (1)–(9) during the learning phase of AEs. The processing time for updating is decided by the number of clock cycles and the operation frequency of the circuit. Details about the number of clock cycles of the proposed circuits are given in Table 3, and the number of clock cycles is computed using Eq (11) when the AE structure is 4-2-4.

$$ClockCycles = 8N^i + 2N^h + 235 \tag{11}$$

where the number of the neurons in an input and in the hidden layers are represented by N^i and N^h , respectively.

Assuming that each addition, subtraction, and multiplication is counted as one operation, the number of operations is determined using Eqs (1)–(9), and is formulated as Eq (12). Moreover, the processing time is determined using Eq (13).

$$\# ofOperations = 6N^i + 11N^iN^h + 6N^h \tag{12}$$

$$ProcessingTime = \frac{ClockCycle}{Frequency} \tag{13}$$

where the clock cycles and frequency are computed using Eq (11) and Table 1, respectively.

Table 2. Output results of the proposed AE circuit after training.

Input: 0 0 0 0			
0.027267456	0.027267456	0.027267456	0.027267456
Input: 0 0 0 1			
0.034179688	0.034179688	0.034179688	0.96875
Input: 0 0 1 0			
0.034179688	0.034179688	0.96875	0.034179688
Input: 0 0 1 1			
0.048828125	0.048828125	0.953125	0.953125
Input: 0 1 0 0			
0.034179688	0.96875	0.034179688	0.034179688
Input: 0 1 0 1			
0.048828125	0.953125	0.048828125	0.953125
Input: 0 1 1 0			
0.025268555	0.96875	0.96875	0.025268555
Input: 0 1 1 1			
0.025756836	0.96875	0.96875	0.96875
Input: 1 0 0 0			
0.96875	0.034179688	0.034179688	0.034179688
Input: 1 0 0 1			
0.96875	0.025268555	0.025268555	0.96875
Input: 1 0 1 0			
0.953125	0.048828125	0.953125	0.048828125
Input: 1 0 1 1			
0.96875	0.025756836	0.96875	0.96875
Input: 1 1 0 0			
0.953125	0.953125	0.048828125	0.048828125
Input: 1 1 0 1			
0.96875	0.96875	0.025756836	0.96875
Input: 1 1 1 0			
0.96875	0.96875	0.96875	0.025756836
Input: 1 1 1 1			
0.96875	0.96875	0.96875	0.96875

<https://doi.org/10.1371/journal.pone.0194049.t002>

Therefore, the OPS is computed by Eq (14).

$$OPS = \frac{\# \text{ of Operations}}{\text{Processing Time}} = \frac{6N^i + 11N^iN^h + 6N^h}{8N^i + 2N^h + 235} * \text{Frequency} \quad (14)$$

A multiplier is synthesized as a digital signal processor (DSP), which is a specific customized primitive element of FPGAs. The number of DSPs is limited depending on FPGAs. The number of multipliers used in our circuits is determined using Eq (15) given below.

$$\# \text{ of Multipliers} = N^h * (2 * N^i + 9) \quad (15)$$

As expressed in Eq (15), the number of multipliers increases linearly along the equation until the limitation of DSPs.

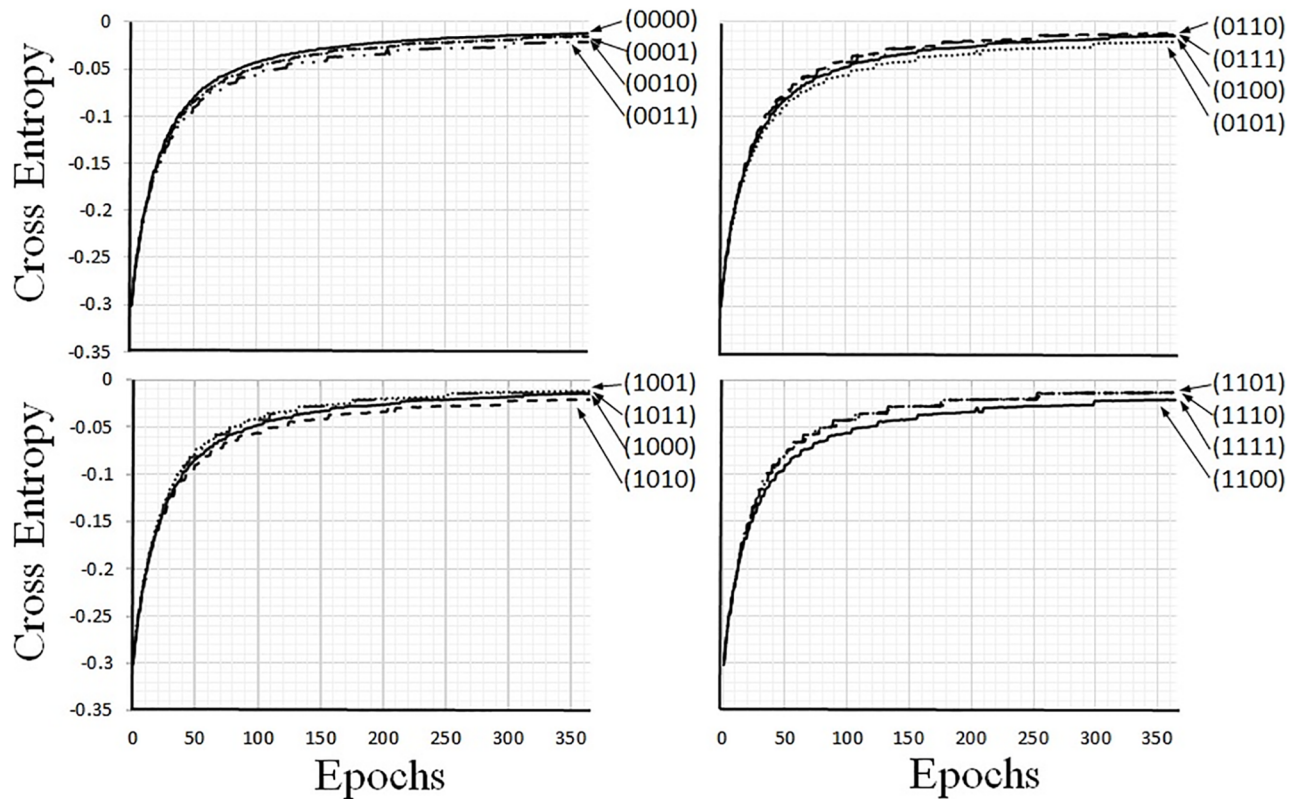


Fig 9. Training results represented with cross entropy errors.

<https://doi.org/10.1371/journal.pone.0194049.g009>

The OPS and the number of multipliers of various structure are given in Table 4, where the 4-2-4 structure is compared with four types of structures: 8-2-8 (double the number of in-out neurons) and 4-4-4 (double the number of hidden neurons), and 20-10-20 (five times the number of both of in-out and hidden neurons), 24-10-24 (maximum structure in this setup). As mentioned above, the target device of this paper is xc6vlx240t which has 768 DSPs. DSP utilization is not limited to a multiplier and the total number of DSP utilizations is cleared by a logic synthesis. Therefore, 24-10-24 is the maximum structure and this structure employs 750 DSPs.

According to Table 4, the OPS tends to be influenced by the number of synapses, which is determined by multiplexing the number of in/out and hidden neurons as expressed by Eq (15).

Table 3. Equations for the number of clock cycles.

Modules	Clock cycles
Neuron (hidden unit)	$N^i + 5$
Neuron (output unit)	$N^h + 5$
Update hidden bias	$N^i + 17$
Synapse	$(N^i + 8) * 2$
Weight memory controller	$N^i + N^h + 4$
Update execution	$N^i + 8$
Fixed clock cycle	164

<https://doi.org/10.1371/journal.pone.0194049.t003>

Table 4. Performance and resource comparison of various structure of AEs.

Network structure	MOPS*	Number of multipliers
4-2-4	116	30
8-2-8	201	46
4-4-4	206	60
20-10-20	1,529	470
24-10-24	1,713	550

* Mega operations per second. Bigger is better

<https://doi.org/10.1371/journal.pone.0194049.t004>

To construct larger AEs than the 24-10-24 structure, the proposed circuits require a memory controller and additional memory. To communicate with external modules such the memory controller and memory, the number of operations would increase, which acts as performance penalty and degrades the performance of the proposed circuits.

Verification of the scalability of the proposed circuit

All of the modules comprising the proposed circuits have a common interface and parameters that are controlled externally. The common interface is useful for combining the modules with one another, and the parameters can be used to define the network structures for the stacked AEs.

Common interface

In our study, the common interface used was based on the first-in, first-out (FIFO) principle; this was used in all of the modules, and it was designed to help users combine the modules easily. Fig 10 shows the input and output ports: iDS and oDD are for data; iES and iED are the enable signals; oFLL and oEMP mean that the module memory is full and empty, respectively; iSTART and iEND receive the flags to start and end a process, respectively; and oSTART and oEND send the corresponding flags for starting and ending a process, respectively. A module may have several input-output signal wires for data, as iDS and oDD, depending on its function. The input and output ports of all of the modules are shown in Table 5. For instance, in Fig 6, “x”, “U_W”, “U_b” or “U_b”, (left-hand side of Reconstruction module) are regarded as iDS, and “x”, “y”, “z”, or “w”. (right-hand side of Reconstruction module) are regarded as oDD, in Fig 10 respectively.

Parameters for the network structure

Construction of stacked AE. With the parameters for the network structure in the designed circuit, we constructed the first and second AEs, which consisted of 4-2-4 and 2-1-2 networks, respectively. These two AEs were then used to construct a 4-2-1-2-4 stacked AE that had the proposed architecture; this is shown in Fig 11. As is shown in Table 6, the stacked AE circuit required fewer registers and LUTs than the first and second AE circuits did, because several of the necessary parts were duplicated when the second AE was stacked on the first.

Learning operation of second AE. The learning operation for the stacked AE consisted of two steps; the first step was for the first AE, and the second step was for the second AE. Because the step for the first AE is described in Section 5, the step for the second AE is explained in this sub-section. The input data for the second AE is the value of the hidden neuron in the first AE, which had been trained with the sixteen kinds of binary data in Section 5.2. The learning rate

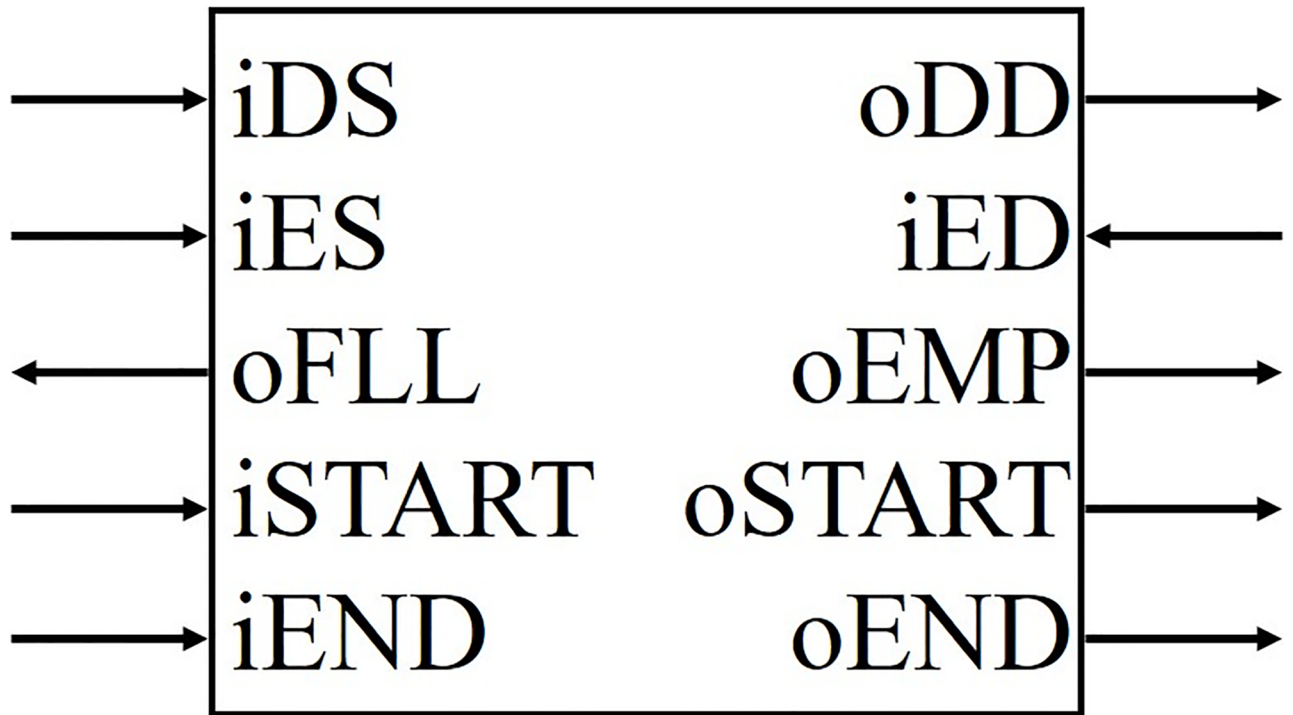


Fig 10. Common interface.

<https://doi.org/10.1371/journal.pone.0194049.g010>

and the number of epochs were set to the same values used in the training of the first AE. The results of the learning process are shown in Fig 12. To update the parameters of the AEs, we used Eqs (4)–(6); they were determined from the cross entropy. The root mean square error was used to measure the error in the second AE, because this error was a real value. Using the

Table 5. I/O ports.

Reconstruction circuit			Update function circuit		
Module	iDS	oDD	Module	iDS	oDD
(a)	x	x	(b)	x	U_b'
	U_W	y		y	U_W
	U_b	z		z	U_b
	U_b'	w		w	
(a-1)	x / y	wx / wy	(b-1)	x	U_b'
	U_W	w		z	
(a-2)	wx / wy	y / z	(b-2)	U_b'	U_b
	U_b / U_b'			w	
		y			
			(b-3)	x	U_W
				U_b	
				U_b'	
				y	

<https://doi.org/10.1371/journal.pone.0194049.t005>

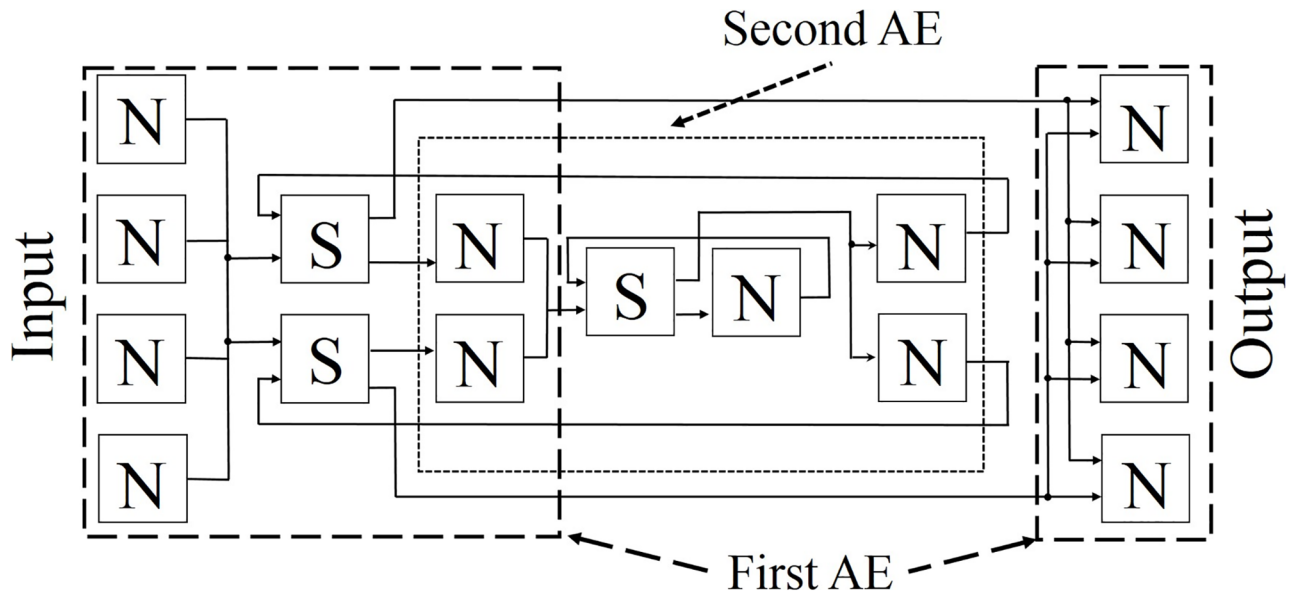


Fig 11. Stacked AE with shared synapse architecture.

<https://doi.org/10.1371/journal.pone.0194049.g011>

results of the experiments, we were able to confirm that all of the parameter values after training were close to the desired values.

Operation of the reconstruction of the stacked AE. The parameters obtained for the first and second AEs through training were input into the stacked AE. The sixteen kinds of binary datasets were used to verify whether the stacked AE was able to reconstruct the input data. The comparison between the outputs of the first AE and the reconstructed values of the stacked AE are shown in Table 7; although some of the results have an error of 20%, the output results were regarded as having successfully digitally reconstructed the input data. These errors were produced as a result of the learning process used for the second AE, i.e., the inputs were encoded into one neuron in the second AE, after they were encoded into two neurons in the first AE.

Parameters for the bit width

To evaluate the relationship between the bit widths of the proposed circuits and the learning performances, the bit widths were changed from eighteen to ten. The results of the experiments are shown in Fig 13, and it can be seen that the AE reaches the target values more closely as the number of bits increases. In the case of ten bits, the cross entropy error was found to barely decrease; this was because seven-bit shifting, which was used for the learning rate, leads to a result that is almost zero in the update process for ten-bit numbers.

Table 6. Logic synthesis results for the stacked and second AE reconstruction modules.

Module	Registers	LUTs	Freq. (MHz)
First AE (Reconstruction module)	2,477 (0.82%)	2,324 (1.54%)	242.424
Second AE	728 (0.24%)	798 (0.53%)	242.777
Stacked AE	2,746 (0.91%)	2,781 (1.85%)	242.777

<https://doi.org/10.1371/journal.pone.0194049.t006>

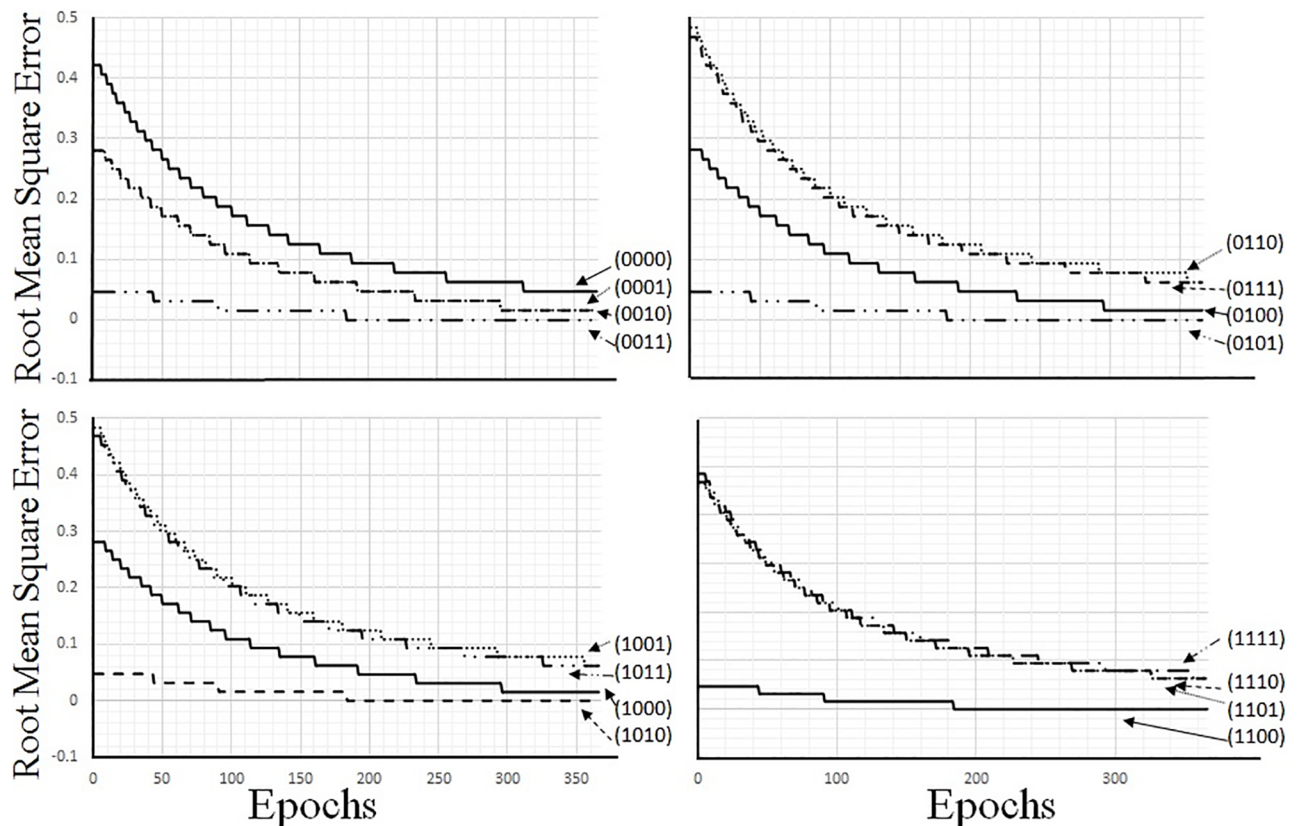


Fig 12. Training results represented with root mean square errors.

<https://doi.org/10.1371/journal.pone.0194049.g012>

Comparison with related works

The proposed circuit was compared with two related works about the implementations of AEs into FPGAs. The results of the comparisons are shown in Table 8.

In one of the studies, sparse AEs are described by a Verilog HDL as a behavior model and evaluated by pre-route simulations [40]; however, there is no description in this study about the detailed hardware architecture used and the logic synthesis results. Because the pre-route behavior model does not consider FPGA resources and timing issues, implementing it in an actual FPGA is difficult. According to Table 8, the performance of [40] is not available (N/A) and cannot be calculated, because both operation frequency and time period, key requirements for evaluating the processing speed of a digital circuit, are not given in [40].

In the other study, a three-layered stacked AE was designed using an OpenCL programming framework and implemented on an NVIDIA GTX Titan GPU, a Qualcomm Adreno 330 mobile GPU, and an Altera Stratix V D5 FPGA [39]. For the FPGA implementation, HLS was used to generate an RTL code automatically from the OpenCL description. The experimental results showed that the FPGA performed the worst in terms of processing speed. Generally, it is hard to find an optimal digital architecture from a high-level description by architecture exploration in HLS. The generated RTL code was therefore not optimized for the stacked AE, and its FPGA implementation resulted in the worst performance among the devices.

In contrast, the circuit proposed in the present study was manually designed by RTL and was based on the shared synapse architecture. The RTL design enabled us to formulate the

Table 7. Outputs of the proposed stacked AE after learning.

Input: 0 0 0 0			
0.083984375	0.083984375	0.083984375	0.083984375
Input: 0 0 0 1			
0.09765625	0.16015625	0.09765625	0.84375
Input: 0 0 1 0			
0.09765625	0.16015625	0.75	0.16015625
Input: 0 0 1 1			
0.107421875	0.203125	0.796875	0.890625
Input: 0 1 0 0			
0.16015625	0.75	0.16015625	0.09765625
Input: 0 1 0 1			
0.1484375	0.859375	0.1484375	0.859375
Input: 0 1 1 0			
0.22265625	0.796875	0.796875	0.22265625
Input: 0 1 1 1			
0.20703125	0.9375	0.8125	0.9375
Input: 1 0 0 0			
0.84375	0.09765625	0.16015625	0.09765625
Input: 1 0 0 1			
0.796875	0.22265625	0.22265625	0.796875
Input: 1 0 1 0			
0.859375	0.1484375	0.859375	0.1484375
Input: 1 0 1 1			
0.8125	0.484375	0.8125	0.9375
Input: 1 1 0 0			
0.890625	0.796875	0.203125	0.107421875
Input: 1 1 0 1			
0.9375	0.9375	0.484375	0.8125
Input: 1 1 1 0			
0.9375	0.8125	0.9375	0.20703125
Input: 1 1 1 1			
0.984375	0.984375	0.984375	0.984375

<https://doi.org/10.1371/journal.pone.0194049.t007>

number of clock cycles required for an AE used in the proposed circuit; this is shown in more detail in Table 3. A weight memory controller controls the read/write operations of the weight memory in the synapse module; the other modules are shown in Figs 6 and 7. By considering the equations in Table 3 and the operating frequency from the synthesized results shown in Tables 1 and 6, we can estimate the theoretical performance of a circuit by constructing arbitrary networks. From these estimates, we can see that the proposed circuit designed by RTL is much faster than a circuit designed by HLS. As shown in Table 8, unlike [40], the OPS can be calculated because both the processing time for one epoch and the number of operations are given in descriptions in [39]. Compared to [39], the performance of the proposed architecture is six times better than it.

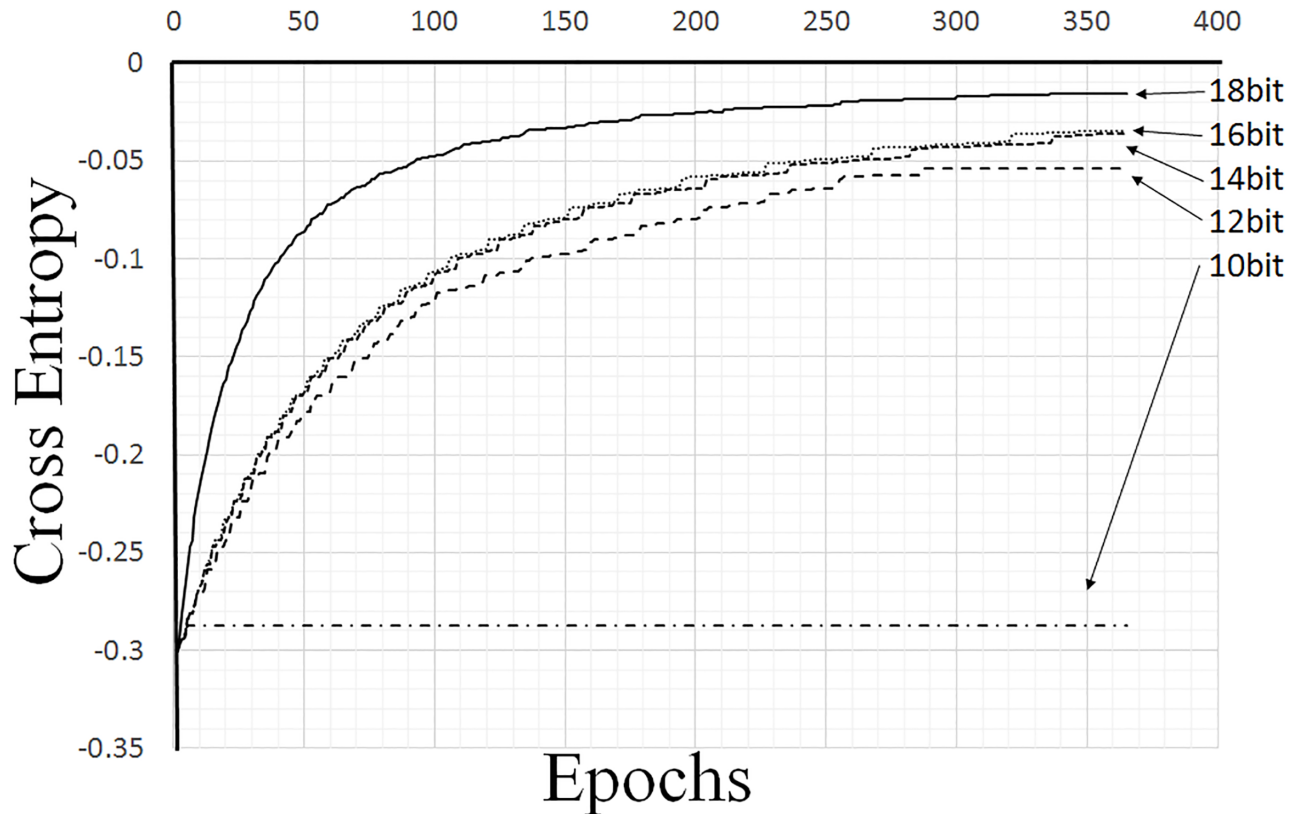


Fig 13. Relationship between accuracy and bit width.

<https://doi.org/10.1371/journal.pone.0194049.g013>

Table 8. Comparison of the implementations of the AEs.

	Algorithm	Method	Logic synthesis	MOPS*
Proposed	AE (Stacked AE)	RTL	Possible	1,713
[40]	Sparse AE	Behavior	Impossible	N/A
[39]	Stacked AE	HLS	Possible	357

* Mega operations per second. Bigger is better

<https://doi.org/10.1371/journal.pone.0194049.t008>

Conclusion

A novel hardware architecture for AEs, which we have called a shared synapse architecture, was proposed in this paper. This proposed architecture meant that the number of synapse modules in the AE circuit that were implemented in could be halved. To construct various types of AEs, the proposed architecture had three parameters; one to change a network structure, another to change the bit width of the internal values of the AE, and the other to change a learning rate. Additionally, the proposed circuits had a common interface that allows them to be combined easily. We designed two AEs that consisted of 4-2-4 and 2-1-2 networks, and we combined them to construct a 4-2-1-2-4 stacked AE. The designed circuits were logically synthesized and subsequently evaluated. Our experimental results show that the proposed AE and

the stacked AE circuits successfully reconstructed input data. By comparing our results with related works, we were able to discuss the effectiveness of the proposed RTL design.

Future work involves designing a memory controller that can divide AE processes along time series. A time series processes of large AE networks can be constructed using multiplexing the proposed architecture in time and implemented by the memory controller and a block RAM (high-capacity FPGA internal memory) or an external memory (SDRAM) that stores weights memories. With this future work, we intend to extend the proposed circuit to deep-stacked AEs and integrate it into embedded systems. We plan to apply an extended version of the proposed circuits to various applications such as binary value applications, image processing applications, and abnormal detection applications.

Supporting information

S1 File. RightsLink printable license. Permission from the original copyright holder of Figs 1, 3 and 4. (PDF)

Acknowledgments

This research was supported by JSPS KAKENHI Grant Number 15H01706, 17H01798, and 17K20010.

Author Contributions

Conceptualization: Akihiro Suzuki.

Formal analysis: Akihiro Suzuki.

Funding acquisition: Takashi Morie, Hakaru Tamukoh.

Investigation: Akihiro Suzuki.

Methodology: Akihiro Suzuki, Hakaru Tamukoh.

Project administration: Takashi Morie.

Software: Akihiro Suzuki.

Supervision: Takashi Morie, Hakaru Tamukoh.

Validation: Akihiro Suzuki, Takashi Morie, Hakaru Tamukoh.

Writing – original draft: Akihiro Suzuki.

Writing – review & editing: Akihiro Suzuki, Takashi Morie, Hakaru Tamukoh.

References

1. Hinton GE, Osindero S, Teh YW. A fast learning algorithm for deep belief nets. *Neural computation*. 2006; 18(7):1527–1554. <https://doi.org/10.1162/neco.2006.18.7.1527> PMID: 16764513
2. Ruck DW, Rogers SK, Kabrisky M, Oxley ME, Suter BW. The multilayer perceptron as an approximation to a Bayes optimal discriminant function. *IEEE Transactions on Neural Networks*. 1990; 1(4):296–298. <https://doi.org/10.1109/72.80266> PMID: 18282850
3. Krizhevsky A, Sutskever I, Hinton GE. Imagenet classification with deep convolutional neural networks. In: *Advances in neural information processing systems*; 2012. p. 1097–1105.
4. Fischer A, Igel C. Training restricted Boltzmann machines: An introduction. *Pattern Recognition*. 2014; 47(1):25–39. <https://doi.org/10.1016/j.patcog.2013.05.025>

5. Rumelhart DE, Hinton GE, Williams RJ. Learning internal representations by error propagation. DTIC Document; 1985.
6. Masci J, Meier U, Cireşan D, Schmidhuber J. Stacked convolutional auto-encoders for hierarchical feature extraction. In: International Conference on Artificial Neural Networks. Springer; 2011. p. 52–59.
7. Rifai S, Vincent P, Muller X, Glorot X, Bengio Y. Contractive auto-encoders: Explicit invariance during feature extraction. In: Proceedings of the 28th international conference on machine learning (ICML-11); 2011. p. 833–840.
8. Alain D, Olivier S. Gated autoencoders with tied input weights. In: Proceedings of the 30th International Conference on Machine Learning (ICML-13). vol. 28; 2013. p. 154–162.
9. Lore KG, Akintayo A, Sarkar S. LLNet: A Deep Autoencoder approach to Natural Low-light Image Enhancement. arXiv preprint arXiv:151103995. 2015;.
10. Wang Y, Yao H, Zhao S. Auto-encoder based dimensionality reduction. *Neurocomputing*. 2016; 184:232–242. <https://doi.org/10.1016/j.neucom.2015.08.104>
11. Carreira-Perpinán MA, Raziperchikolaei R. Hashing with binary autoencoders. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition; 2015. p. 557–566.
12. Jianan C, Xin L, Yile W, Huafeng L. Deep reconstruction model for dynamic PET images. *PLoS ONE*. 2017; 12(9):e0184667. <https://doi.org/10.1371/journal.pone.0184667>
13. Chao W, Senlin L, Xincheng M, Hao R, Ji Z, Limin P. Locally Embedding Autoencoders: A Semi-Supervised Manifold Learning Approach of Document Representation. *PLoS ONE*. 2016; 11(1):e0146672. <https://doi.org/10.1371/journal.pone.0146672>
14. Bengio Y. Learning deep architectures for AI. *Foundations and trends in Machine Learning*. 2009; 2(1):1–127. <https://doi.org/10.1561/22000000006>
15. Hinton GE, Salakhutdinov RR. Reducing the dimensionality of data with neural networks. *Science*. 2006; 313(5786):504–507. <https://doi.org/10.1126/science.1127647> PMID: 16873662
16. Bengio Y, Lamblin P, Popovici D, Larochelle H. Greedy Layer-Wise Training of Deep Networks. In: Bernhard S, John P, Thomas H, editors. *Advances in Neural Information Processing Systems 19*. MIT Press; 2007. p. 153–160.
17. Liu C, Cao Y, Luo Y, Chen G, Vokkarane V, Ma Y. DeepFood: Deep Learning-Based Food Image Recognition for Computer-Aided Dietary Assessment. In: International Conference on Smart Homes and Health Telematics. Springer; 2016. p. 37–48.
18. Casellato C, Antonietti A, Garrido JA, Carrillo RR, Luque NR, Ros E, et al. Adaptive robotic control driven by a versatile spiking cerebellar network. *PLoS one*. 2014; 9(11):e112265. <https://doi.org/10.1371/journal.pone.0112265> PMID: 25390365
19. Hu C, Bai X, Qi L, Chen P, Xue G, Mei L. Vehicle color recognition with spatial pyramid deep learning. *IEEE Transactions on Intelligent Transportation Systems*. 2015; 16(5):2925–2934. <https://doi.org/10.1109/TITS.2015.2430892>
20. Bastien F, Lamblin P, Pascanu R, Bergstra J, Goodfellow I, Bergeron A, et al. Theano: new features and speed improvements. arXiv preprint arXiv:12115590. 2012;.
21. Jia Y, Shelhamer E, Donahue J, Karayev S, Long J, Girshick R, et al. Caffe: Convolutional architecture for fast feature embedding. In: Proceedings of the 22nd ACM international conference on Multimedia. ACM; 2014. p. 675–678.
22. Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *CoRR*. 2016;.
23. Tokui S, Oono K, Hido S, Clayton J. Chainer: a next-generation open source framework for deep learning. In: Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS); 2015.
24. Whatmough PN, Lee SK, Lee H, Rama S, Brooks D, Wei GY. 14.3 A 28nm SoC with a 1.2GHz 568nJ/prediction sparse deep-neural-network engine with >0.1 timing error rate tolerance for IoT applications. In: 2017 IEEE International Solid-State Circuits Conference (ISSCC); 2017. p. 242–243.
25. Price M, Glass J, Chandrakasan AP. 14.4 A scalable speech recognizer with deep-neural-network acoustic models and voice-activated power gating. In: 2017 IEEE International Solid-State Circuits Conference (ISSCC); 2017. p. 244–245.
26. Park S, Bong K, Shin D, Lee J, Choi S, Yoo HJ. A 1.93 TOPS/W scalable deep learning/inference processor with tetra-parallel MIMD architecture for big-data applications. In: IEEE International Solid-State Circuits Conference (ISSCC); 2015. p. 80–82.
27. Du Z, Fasthuber R, Chen T, lenne P, Li L, Luo T, et al. ShiDianNao: shifting vision processing closer to the sensor. In: ACM SIGARCH Computer Architecture News. vol. 43. ACM; 2015. p. 92–104.

28. Chen Y, Luo T, Liu S, Zhang S, He L, Wang J, et al. DaDianNao: A Machine-Learning Supercomputer. In: 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture; 2014. p. 609–622.
29. Chen YH, Emer J, Sze V. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In: 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA); 2016. p. 367–379.
30. Desoli G, Chawla N, Boesch T, Singh S, Guidetti E, Ambroggi FD, et al. 14.1 A 2.9TOPS/W deep convolutional neural network SoC in FD-SOI 28nm for intelligent embedded systems. In: 2017 IEEE International Solid-State Circuits Conference (ISSCC); 2017. p. 238–239.
31. Shin D, Lee J, Lee J, Yoo HJ. 14.2 DNPU: An 8.1TOPS/W reconfigurable CNN-RNN processor for general-purpose deep neural networks. In: 2017 IEEE International Solid-State Circuits Conference (ISSCC); 2017. p. 240–241.
32. Cho H, Son H, Seong K, Kim B, Park HJ, Sim JY. An On-Chip Learning Neuromorphic Autoencoder With Current-Mode Transposable Memory Read and Virtual Lookup Table. *IEEE Transactions on Biomedical Circuits and Systems*. 2018; PP(99):1–10.
33. Yonekawa H, Nakahara H. On-Chip Memory Based Binarized Convolutional Deep Neural Network Applying Batch Normalization Free Technique on an FPGA. In: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW); 2017. p. 98–105.
34. Liang S, Yin S, Liu L, Luk W, Wei S. FP-BNN: Binarized neural network on FPGA. *Neurocomputing*. 2018; 275:1072–1086. <https://doi.org/10.1016/j.neucom.2017.09.046>
35. Umuroglu Y, Fraser NJ, Gambardella G, Blott M, Leong PHW, Jahre M, et al. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. *CoRR*. 2016;.
36. Li H, Fan X, Jiao L, Cao W, Zhou X, Wang L. A high performance FPGA-based accelerator for large-scale convolutional neural networks. In: *Proceedings of 26th International Conference on Field Programmable Logic and Applications*. IEEE; 2016. p. 1–9.
37. Kim SK, McAfee LC, McMahon PL, Olukotun K. A highly scalable restricted Boltzmann machine FPGA implementation. In: *Proceedings of the 19th International Conference on Field Programmable Logic and Applications*. IEEE; 2009. p. 367–372.
38. Ueyoshi K, Marukame T, Asai T, Motomura M, Schmid A. FPGA Implementation of a Scalable and Highly Parallel Architecture for Restricted Boltzmann Machines. *Circuits and Systems*. 2016; 7(9):2132–2141. <https://doi.org/10.4236/cs.2016.79185>
39. Joao M, Joao A, Gabriel F, Luis A A. Stacked Autoencoders Using Low-Power Accelerated Architectures for Object Recognition in Autonomous System. *Neural Processing Letters*. 2015; 43:1–14.
40. Jin Y, Kim D. Unsupervised Feature Learning by Pre-Route Simulation of Auto-Encoder Behavior Model. *International Journal of Computer, Electrical, Automation, Control and Information Engineering*. 2014; 8(5):668–672.
41. Suzuki A, Morie T, Tamukoh H. FPGA Implementation of Autoencoders Having Shared Synapse Architecture. In: *International Conference on Neural Information Processing*. Springer; 2016. p. 231–239.
42. Vincent P. A connection between score matching and denoising autoencoders. *Neural computation*. 2011; 23(7):1661–1674. https://doi.org/10.1162/NECO_a_00142 PMID: 21492012